

Reverse Engineering для начинающих



Денис Юричев

Reverse Engineering для начинающих

Денис Юричев

<dennis(a)yurichev.com>



©2013-2016, Денис Юричев.

Это произведение доступно по лицензии Creative Commons «Attribution-ShareAlike 4.0 International» (CC BY-SA 4.0).

Чтобы увидеть копию этой лицензии, посетите <https://creativecommons.org/licenses/by-sa/4.0/>.

Версия этого текста (11 августа 2016 г.).

Самая новая версия текста (а также англоязычная версия) доступна на сайте beginners.re. Версия для электронных читалок так же доступна на сайте.

Обложка нарисована Андреем Нечаевским: [facebook](https://www.facebook.com/).

Нужны переводчики!

Возможно, вы захотите мне помочь с переводом этой работы на другие языки, кроме английского и русского. Просто пришлите мне любой фрагмент переведенного текста (не важно, насколько короткий), и я добавлю его в исходный код на LaTeX.

Скорость не важна, потому что это опен-сорсный проект все-таки. Ваше имя будет указано в числе участников проекта. Корейский, китайский и персидский языки зарезервированы издателями. Английскую и русскую версии я делаю сам, но английский у меня все еще ужасный, так что я буду очень признателен за коррективы, итд. Даже мой русский несовершенный, так что я благодарен за коррективы и русского текста!

Не стесняйтесь писать мне: `dennis(a)yurichev.com`.

Краткое оглавление

I	Образцы кода	1
II	Важные фундаментальные вещи	444
III	Более сложные примеры	453
IV	Java	603
V	Поиск в коде того что нужно	641
VI	Специфичное для ОС	671
VII	Инструменты	727
VIII	Примеры из практики	733
IX	Примеры разбора закрытых (proprietary) форматов файлов	850
X	Прочее	883
XI	Что стоит почитать	901
	Послесловие	906
	Приложение	908
	Список принятых сокращений	938

Оглавление

I	Образцы кода	1
1	Метод	3
2	Некоторые базовые понятия	4
2.1	Краткое введение в CPU	4
2.1.1	Несколько слов о разнице между ISA ¹	4
3	Простейшая функция	6
3.1	x86	6
3.2	ARM	6
3.3	MIPS	7
3.3.1	Еще кое-что об именах инструкций и регистров в MIPS	7
4	Hello, world!	8
4.1	x86	8
4.1.1	MSVC	8
4.1.2	GCC	9
4.1.3	GCC: Синтаксис AT&T	10
4.2	x86-64	12
4.2.1	MSVC — x86-64	12
4.2.2	GCC — x86-64	12
4.3	GCC — ещё кое-что	13
4.4	ARM	14
4.4.1	Неоптимизирующий Keil 6/2013 (Режим ARM)	14
4.4.2	Неоптимизирующий Keil 6/2013 (Режим Thumb)	15
4.4.3	Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)	16
4.4.4	Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)	17
4.4.5	ARM64	18
4.5	MIPS	20
4.5.1	О «глобальном указателе» («global pointer»)	20
4.5.2	Оптимизирующий GCC	20
4.5.3	Неоптимизирующий GCC	22
4.5.4	Роль стекового фрейма в этом примере	23
4.5.5	Оптимизирующий GCC: загрузим в GDB	23
4.6	Вывод	24
4.7	Упражнения	24
5	Пролог и эпилог функций	25
5.1	Рекурсия	25
6	Стек	26
6.1	Почему стек растет в обратную сторону?	26
6.2	Для чего используется стек?	27
6.2.1	Сохранение адреса возврата управления	27
6.2.2	Передача параметров функции	28
6.2.3	Хранение локальных переменных	29
6.2.4	x86: Функция <code>alloca()</code>	29
6.2.5	(Windows) SEH	31
6.2.6	Защита от переполнений буфера	31
6.2.7	Автоматическое освобождение данных в стеке	32
6.3	Разметка типичного стека	32

¹Instruction Set Architecture (Архитектура набора команд)

6.4	Мусор в стеке	32
6.4.1	MSVC 2013	35
6.5	Упражнения	36
7	printf() с несколькими аргументами	37
7.1	x86	37
7.1.1	x86: 3 аргумента	37
7.1.2	x64: 8 аргументов	45
7.2	ARM	48
7.2.1	ARM: 3 аргумента	48
7.2.2	ARM: 8 аргументов	50
7.3	MIPS	53
7.3.1	3 аргумента	53
7.3.2	8 аргументов	56
7.4	Вывод	59
7.5	Кстати	60
8	scanf()	62
8.1	Простой пример	62
8.1.1	Об указателях	62
8.1.2	x86	63
8.1.3	MSVC + OllyDbg	65
8.1.4	x64	68
8.1.5	ARM	69
8.1.6	MIPS	70
8.2	Глобальные переменные	71
8.2.1	MSVC: x86	72
8.2.2	MSVC: x86 + OllyDbg	74
8.2.3	GCC: x86	75
8.2.4	MSVC: x64	75
8.2.5	ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)	76
8.2.6	ARM64	77
8.2.7	MIPS	78
8.3	Проверка результата scanf()	81
8.3.1	MSVC: x86	81
8.3.2	MSVC: x86: IDA	82
8.3.3	MSVC: x86 + OllyDbg	86
8.3.4	MSVC: x86 + Hiew	88
8.3.5	MSVC: x64	89
8.3.6	ARM	90
8.3.7	MIPS	91
8.3.8	Упражнение	92
8.4	Упражнение	92
9	Доступ к переданным аргументам	93
9.1	x86	93
9.1.1	MSVC	93
9.1.2	MSVC + OllyDbg	94
9.1.3	GCC	94
9.2	x64	95
9.2.1	MSVC	95
9.2.2	GCC	97
9.2.3	GCC: uint64_t вместо int	98
9.3	ARM	98
9.3.1	Неоптимизирующий Keil 6/2013 (Режим ARM)	98
9.3.2	Оптимизирующий Keil 6/2013 (Режим ARM)	99
9.3.3	Оптимизирующий Keil 6/2013 (Режим Thumb)	99
9.3.4	ARM64	100
9.4	MIPS	101
10	Ещё о возвращаемых результатах	103
10.1	Попытка использовать результат функции возвращающей <i>void</i>	103
10.2	Что если не использовать результат функции?	104
10.3	Возврат структуры	104

11 Указатели	106
11.1 Пример с глобальными переменными	106
11.2 Пример с локальными переменными	112
11.3 Вывод	115
12 Оператор GOTO	116
12.1 Мертвый код	118
12.2 Упражнение	119
13 Условные переходы	120
13.1 Простой пример	120
13.1.1 x86	120
13.1.2 ARM	131
13.1.3 MIPS	134
13.2 Вычисление абсолютной величины	137
13.2.1 Оптимизирующий MSVC	137
13.2.2 Оптимизирующий Keil 6/2013: Режим Thumb	137
13.2.3 Оптимизирующий Keil 6/2013: Режим ARM	138
13.2.4 Неоптимизирующий GCC 4.9 (ARM64)	138
13.2.5 MIPS	138
13.2.6 Версия без переходов?	139
13.3 Тернарный условный оператор	139
13.3.1 x86	139
13.3.2 ARM	140
13.3.3 ARM64	141
13.3.4 MIPS	141
13.3.5 Перепишем, используя обычный <code>if/else</code>	142
13.3.6 Вывод	142
13.4 Поиск минимального и максимального значения	142
13.4.1 32-bit	142
13.4.2 64-bit	145
13.4.3 MIPS	146
13.5 Вывод	147
13.5.1 x86	147
13.5.2 ARM	147
13.5.3 MIPS	147
13.5.4 Без инструкций перехода	148
13.6 Упражнение	148
14 switch()/case/default	149
14.1 Если вариантов мало	149
14.1.1 x86	149
14.1.2 ARM: Оптимизирующий Keil 6/2013 (Режим ARM)	159
14.1.3 ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)	160
14.1.4 ARM64: Неоптимизирующий GCC (Linaro) 4.9	160
14.1.5 ARM64: Оптимизирующий GCC (Linaro) 4.9	161
14.1.6 MIPS	161
14.1.7 Вывод	162
14.2 И если много	162
14.2.1 x86	163
14.2.2 ARM: Оптимизирующий Keil 6/2013 (Режим ARM)	169
14.2.3 ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)	171
14.2.4 MIPS	172
14.2.5 Вывод	173
14.3 Когда много <code>case</code> в одном блоке	174
14.3.1 MSVC	175
14.3.2 GCC	176
14.3.3 ARM64: Оптимизирующий GCC 4.9.1	176
14.4 Fall-through	178
14.4.1 MSVC x86	178
14.4.2 ARM64	179
14.5 Упражнения	180
14.5.1 Упражнение #1	180

15 Циклы	181
15.1 Простой пример	181
15.1.1 x86	181
15.1.2 x86: OllyDbg	185
15.1.3 x86: tracer	185
15.1.4 ARM	187
15.1.5 MIPS	190
15.1.6 Ещё кое-что	191
15.2 Функция копирования блоков памяти	191
15.2.1 Простейшая реализация	191
15.2.2 ARM в режиме ARM	192
15.2.3 MIPS	193
15.2.4 Векторизация	194
15.3 Вывод	194
15.4 Упражнения	195
16 Простая работа с Си-строками	196
16.1 strlen()	196
16.1.1 x86	196
16.1.2 ARM	203
16.1.3 MIPS	206
17 Замена одних арифметических инструкций на другие	207
17.1 Умножение	207
17.1.1 Умножение при помощи сложения	207
17.1.2 Умножение при помощи сдвигов	207
17.1.3 Умножение при помощи сдвигов, сложений и вычитаний	208
17.2 Деление	212
17.2.1 Деление используя сдвиги	212
17.3 Упражнение	213
18 Работа с FPU	214
18.1 IEEE 754	214
18.2 x86	214
18.3 ARM, MIPS, x86/x64 SIMD	214
18.4 Си/Си++	215
18.5 Простой пример	215
18.5.1 x86	215
18.5.2 ARM: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)	222
18.5.3 ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)	223
18.5.4 ARM64: Оптимизирующий GCC (Linaro) 4.9	224
18.5.5 ARM64: Неоптимизирующий GCC (Linaro) 4.9	224
18.5.6 MIPS	225
18.6 Передача чисел с плавающей запятой в аргументах	226
18.6.1 x86	226
18.6.2 ARM + Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)	227
18.6.3 ARM + Неоптимизирующий Keil 6/2013 (Режим ARM)	227
18.6.4 ARM64 + Оптимизирующий GCC (Linaro) 4.9	227
18.6.5 MIPS	228
18.7 Пример с сравнением	229
18.7.1 x86	229
18.7.2 ARM	258
18.7.3 ARM64	261
18.7.4 MIPS	262
18.8 Стек, калькуляторы и обратная польская запись	263
18.9 x64	263
18.10 Упражнения	263
19 Массивы	264
19.1 Простой пример	264
19.1.1 x86	264
19.1.2 ARM	267
19.1.3 MIPS	270
19.2 Переполнение буфера	271
19.2.1 Чтение за пределами массива	271

19.2.2	Запись за пределы массива	274
19.3	Защита от переполнения буфера	279
19.3.1	Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)	280
19.4	Еще немного о массивах	282
19.5	Массив указателей на строки	282
19.5.1	x64	283
19.5.2	32-битный ARM	284
19.5.3	ARM64	285
19.5.4	MIPS	286
19.5.5	Переполнение массива	287
19.6	Многомерные массивы	289
19.6.1	Пример с двумерным массивом	289
19.6.2	Работа с двумерным массивом как с одномерным	291
19.6.3	Пример с трехмерным массивом	292
19.6.4	Ещё примеры	295
19.7	Набор строк как двумерный массив	296
19.7.1	32-bit ARM	297
19.7.2	ARM64	298
19.7.3	MIPS	298
19.7.4	Вывод	299
19.8	Вывод	299
19.9	Упражнения	299
20	Работа с отдельными битами	300
20.1	Проверка какого-либо бита	300
20.1.1	x86	300
20.1.2	ARM	302
20.2	Установка и сброс отдельного бита	304
20.2.1	x86	304
20.2.2	ARM + Оптимизирующий Keil 6/2013 (Режим ARM)	310
20.2.3	ARM + Оптимизирующий Keil 6/2013 (Режим Thumb)	310
20.2.4	ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)	310
20.2.5	ARM: ещё об инструкции BIC	310
20.2.6	ARM64: Оптимизирующий GCC (Linaro) 4.9	311
20.2.7	ARM64: Неоптимизирующий GCC (Linaro) 4.9	311
20.2.8	MIPS	311
20.3	Сдвиги	312
20.4	Установка и сброс отдельного бита: пример с FPU ²	312
20.4.1	x86	312
20.4.2	MIPS	314
20.4.3	ARM	314
20.5	Подсчет выставленных бит	316
20.5.1	x86	318
20.5.2	x64	325
20.5.3	ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)	327
20.5.4	ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)	328
20.5.5	ARM64 + Оптимизирующий GCC 4.9	328
20.5.6	ARM64 + Неоптимизирующий GCC 4.9	329
20.5.7	MIPS	329
20.6	Вывод	331
20.6.1	Проверка определенного бита (известного на стадии компиляции)	331
20.6.2	Проверка определенного бита (заданного во время исполнения)	331
20.6.3	Установка определенного бита (известного во время компиляции)	332
20.6.4	Установка определенного бита (заданного во время исполнения)	332
20.6.5	Сброс определенного бита (известного во время компиляции)	332
20.6.6	Сброс определенного бита (заданного во время исполнения)	333
20.7	Упражнения	333
21	Линейный конгруэнтный генератор	334
21.1	x86	334
21.2	x64	335
21.3	32-bit ARM	336
21.4	MIPS	337

²Floating-point unit

21.4.1	Перемещения в MIPS («relocs»)	338
21.5	Версия этого примера для многопоточной среды	339
22	Структуры	340
22.1	MSVC: Пример SYSTEMTIME	340
22.1.1	OllyDbg	342
22.1.2	Замена структуры массивом	343
22.2	Выделяем место для структуры через malloc()	344
22.3	UNIX: struct tm	345
22.3.1	Linux	345
22.3.2	ARM	348
22.3.3	MIPS	349
22.3.4	Структура как набор переменных	351
22.3.5	Структура как массив 32-битных слов	352
22.3.6	Структура как массив байт	354
22.4	Упаковка полей в структуре	355
22.4.1	x86	356
22.4.2	ARM	360
22.4.3	MIPS	361
22.4.4	Еще кое-что	362
22.5	Вложенные структуры	362
22.5.1	OllyDbg	364
22.6	Работа с битовыми полями в структуре	364
22.6.1	Пример CPUID	364
22.6.2	Работа с типом float как со структурой	368
22.7	Упражнения	371
23	Объединения (union)	372
23.1	Пример генератора случайных чисел	372
23.1.1	x86	373
23.1.2	MIPS	374
23.1.3	ARM (Режим ARM)	375
23.2	Вычисление машинного эпсилона	376
23.2.1	x86	377
23.2.2	ARM64	377
23.2.3	MIPS	378
23.2.4	Вывод	378
23.3	Быстрое вычисление квадратного корня	378
24	Указатели на функции	380
24.1	MSVC	381
24.1.1	MSVC + OllyDbg	383
24.1.2	MSVC + tracer	385
24.1.3	MSVC + tracer (code coverage)	387
24.2	GCC	387
24.2.1	GCC + GDB (с исходными кодами)	388
24.2.2	GCC + GDB (без исходных кодов)	389
25	64-битные значения в 32-битной среде	392
25.1	Возврат 64-битного значения	392
25.1.1	x86	392
25.1.2	ARM	392
25.1.3	MIPS	393
25.2	Передача аргументов, сложение, вычитание	393
25.2.1	x86	393
25.2.2	ARM	394
25.2.3	MIPS	395
25.3	Умножение, деление	396
25.3.1	x86	396
25.3.2	ARM	398
25.3.3	MIPS	399
25.4	Сдвиг вправо	400
25.4.1	x86	400
25.4.2	ARM	400
25.4.3	MIPS	401

25.5	Конвертирование 32-битного значения в 64-битное	401
25.5.1	x86	401
25.5.2	ARM	401
25.5.3	MIPS	402
26	SIMD	403
26.1	Векторизация	403
26.1.1	Пример сложения	404
26.1.2	Пример копирования блоков	409
26.2	Реализация <code>strlen()</code> при помощи SIMD	413
27	64 бита	417
27.1	x86-64	417
27.2	ARM	424
27.3	Числа с плавающей запятой	424
28	Работа с числами с плавающей запятой используя SIMD	425
28.1	Простой пример	425
28.1.1	x64	425
28.1.2	x86	426
28.2	Передача чисел с плавающей запятой в аргументах	433
28.3	Пример с сравнением	434
28.3.1	x64	434
28.3.2	x86	436
28.4	Вычисление машинного эпсилона: x64 и SIMD	436
28.5	И снова пример генератора случайных чисел	437
28.6	Итог	437
29	Кое-что специфичное для ARM	439
29.1	Знак номера (#) перед числом	439
29.2	Режимы адресации	439
29.3	Загрузка констант в регистр	440
29.3.1	32-битный ARM	440
29.3.2	ARM64	440
29.4	Релоки в ARM64	441
30	Кое-что специфичное для MIPS	443
30.1	Загрузка констант в регистр	443
30.2	Книги и прочие материалы о MIPS	443
II	Важные фундаментальные вещи	444
31	Представление знака в числах	446
32	Endianness (порядок байт)	448
32.1	Big-endian (от старшего к младшему)	448
32.2	Little-endian (от младшего к старшему)	448
32.3	Пример	448
32.4	Bi-endian (переключаемый порядок)	449
32.5	Конвертирование	449
33	Память	450
34	CPU	451
34.1	Предсказатели переходов	451
34.2	Зависимости между данными	451
35	Хеш-функции	452
35.1	Как работает односторонняя функция?	452
III	Более сложные примеры	453
36	Конвертирование температуры	454
36.1	Целочисленные значения	454

36.1.1	Оптимизирующий MSVC 2012 x86	454
36.1.2	Оптимизирующий MSVC 2012 x64	456
36.2	Числа с плавающей запятой	456
37	Числа Фибоначчи	459
37.1	Пример #1	459
37.2	Пример #2	462
37.3	Итог	465
38	Пример вычисления CRC32	466
39	Пример вычисления адреса сети	469
39.1	calc_network_address()	470
39.2	form_IP()	471
39.3	print_as_IP()	472
39.4	form_netmask() и set_bit()	473
39.5	Итог	474
40	Циклы: несколько итераторов	475
40.1	Три итератора	475
40.2	Два итератора	476
40.3	Случай Intel C++ 2011	477
41	Duff's device	480
42	Деление на 9	483
42.1	x86	483
42.2	ARM	484
42.2.1	Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)	484
42.2.2	Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)	485
42.2.3	Неоптимизирующий Xcode 4.6.3 (LLVM) и Keil 6/2013	485
42.3	MIPS	485
42.4	Как это работает	486
42.4.1	Больше теории	487
42.5	Определение делителя	487
42.5.1	Вариант #1	487
42.5.2	Вариант #2	488
42.6	Упражнение	488
43	Конверсия строки в число (atoi())	489
43.1	Простой пример	489
43.1.1	Оптимизирующий MSVC 2013 x64	489
43.1.2	Оптимизирующий GCC 4.9.1 x64	490
43.1.3	Оптимизирующий Keil 6/2013 (Режим ARM)	490
43.1.4	Оптимизирующий Keil 6/2013 (Режим Thumb)	491
43.1.5	Оптимизирующий GCC 4.9.1 ARM64	491
43.2	Немного расширенный пример	492
43.2.1	Оптимизирующий GCC 4.9.1 x64	493
43.2.2	Оптимизирующий Keil 6/2013 (Режим ARM)	494
43.3	Упражнение	495
44	Inline-функции	496
44.1	Функции работы со строками и памятью	497
44.1.1	strcmp()	497
44.1.2	strlen()	499
44.1.3	strcpy()	499
44.1.4	memset()	500
44.1.5	memcpy()	501
44.1.6	memcmp()	503
44.1.7	strcat()	504
44.1.8	Скрипт для IDA	504
45	C99 restrict	505
46	Функция abs() без переходов	508
46.1	Оптимизирующий GCC 4.9.1 x64	508

46.2 Оптимизирующий GCC 4.9 ARM64	509
47 Функции с переменным количеством аргументов (<i>variadic</i>)	510
47.1 Вычисление среднего арифметического	510
47.1.1 Соглашение о вызовах <i>cdecl</i>	511
47.1.2 Соглашения о вызовах на основе регистров	511
47.2 Случай с функцией <i>vprintf()</i>	513
48 Обрезка строк	515
48.1 x64: Оптимизирующий MSVC 2013	516
48.2 x64: Неоптимизирующий GCC 4.9.1	517
48.3 x64: Оптимизирующий GCC 4.9.1	518
48.4 ARM64: Неоптимизирующий GCC (Linaro) 4.9	519
48.5 ARM64: Оптимизирующий GCC (Linaro) 4.9	520
48.6 ARM: Оптимизирующий Keil 6/2013 (Режим ARM)	521
48.7 ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)	521
48.8 MIPS	522
49 Функция <i>toupper()</i>	524
49.1 x64	524
49.1.1 Две операции сравнения	524
49.1.2 Одна операция сравнения	525
49.2 ARM	526
49.2.1 GCC для ARM64	526
49.3 Итог	527
50 Неверно дизассемблированный код	528
50.1 Дизассемблирование началось в неверном месте (x86)	528
50.2 Как выглядят случайные данные в дизассемблированном виде?	529
51 Обфускация	534
51.1 Текстовые строки	534
51.2 Исполняемый код	535
51.2.1 Вставка мусора	535
51.2.2 Замена инструкций на раздутые эквиваленты	535
51.2.3 Всегда исполняющийся/никогда не исполняющийся код	535
51.2.4 Сделать побольше путаницы	536
51.2.5 Использование косвенных указателей	536
51.3 Виртуальная машина / псевдо-код	536
51.4 Еще кое-что	536
51.5 Упражнение	537
52 Си++	538
52.1 Классы	538
52.1.1 Простой пример	538
52.1.2 Наследование классов	544
52.1.3 Инкапсуляция	547
52.1.4 Множественное наследование	549
52.1.5 Виртуальные методы	552
52.2 <i>ostream</i>	554
52.3 <i>References</i>	555
52.4 <i>STL</i>	556
52.4.1 <i>std::string</i>	556
52.4.2 <i>std::list</i>	562
52.4.3 <i>std::vector</i>	571
52.4.4 <i>std::map</i> и <i>std::set</i>	579
53 Отрицательные индексы массивов	589
54 Windows 16-bit	592
54.1 Пример #1	592
54.2 Пример #2	592
54.3 Пример #3	593
54.4 Пример #4	594
54.5 Пример #5	596

54.6 Пример #6	600
54.6.1 Глобальные переменные	601
IV Java	603
55 Java	604
55.1 Введение	604
55.2 Возврат значения	604
55.3 Простая вычисляющая функция	608
55.4 Модель памяти в JVM ³	611
55.5 Простой вызов функций	611
55.6 Вызов <code>beer()</code>	613
55.7 Линейный конгруэнтный ГПСЧ ⁴	613
55.8 Условные переходы	614
55.9 Передача аргументов	616
55.10 Битовые поля	617
55.11 Циклы	619
55.12 <code>switch()</code>	620
55.13 Массивы	621
55.13.1 Простой пример	621
55.13.2 Суммирование элементов массива	622
55.13.3 Единственный аргумент <code>main()</code> это также массив	623
55.13.4 Заранее инициализированный массив строк	624
55.13.5 Функции с переменным кол-вом аргументов (<i>variadic</i>)	625
55.13.6 Двухмерные массивы	627
55.13.7 Трехмерные массивы	628
55.13.8 Итоги	629
55.14 Строки	629
55.14.1 Первый пример	629
55.14.2 Второй пример	630
55.15 Исключения	631
55.16 Классы	634
55.17 Простейшая модификация	636
55.17.1 Первый пример	636
55.17.2 Второй пример	637
55.18 Итоги	640
V Поиск в коде того что нужно	641
56 Идентификация исполняемых файлов	643
56.1 Microsoft Visual C++	643
56.1.1 Name mangling	643
56.2 GCC	643
56.2.1 Name mangling	643
56.2.2 Cygwin	643
56.2.3 MinGW	643
56.3 Intel FORTRAN	644
56.4 Watcom, OpenWatcom	644
56.4.1 Name mangling	644
56.5 Borland	644
56.5.1 Delphi	644
56.6 Другие известные DLL	645
57 Связь с внешним миром (win32)	646
57.1 Часто используемые функции Windows API	646
57.2 <code>tracer</code> : Перехват всех функций в отдельном модуле	647
58 Строки	648
58.1 Текстовые строки	648
58.1.1 Си/Си++	648
58.1.2 Borland Delphi	648

³Java virtual machine⁴Генератор псевдослучайных чисел

58.1.3 Unicode	649
58.1.4 Base64	652
58.2 Сообщения об ошибках и отладочные сообщения	652
58.3 Подозрительные магические строки	652
59 Вызовы assert()	653
60 Константы	654
60.1 Магические числа	654
60.1.1 Даты	655
60.1.2 DHCP	655
60.2 Поиск констант	656
61 Поиск нужных инструкций	657
62 Подозрительные паттерны кода	659
62.1 Инструкции XOR	659
62.2 Вручную написанный код на ассемблере	659
63 Использование magic numbers для трассировки	661
64 Прочее	662
64.1 Общая идея	662
64.2 Си++	662
64.3 Некоторые паттерны в бинарных файлах	662
64.3.1 Массивы	663
64.3.2 Разреженные файлы	665
64.3.3 Сжатый файл	666
64.3.4 CDFS ⁵	667
64.3.5 32-битный x86 исполняемый код	668
64.3.6 Графические BMP-файлы	669
64.4 Сравнение «снимков» памяти	669
64.4.1 Реестр Windows	670
64.4.2 Блинк-компаратор	670
VI Специфичное для ОС	671
65 Способы передачи аргументов при вызове функций	672
65.1 cdecl	672
65.2 stdcall	672
65.2.1 Функции с переменным количеством аргументов	673
65.3 fastcall	673
65.3.1 GCC regparm	674
65.3.2 Watcom/OpenWatcom	674
65.4 thiscall	674
65.5 x86-64	675
65.5.1 Windows x64	675
65.5.2 Linux x64	677
65.6 Возвращение переменных типа <i>float</i> , <i>double</i>	677
65.7 Модификация аргументов	677
65.8 Указатель на аргумент функции	678
66 Thread Local Storage	680
66.1 Вернемся к линейному конгруэнтному генератору	680
66.1.1 Win32	680
66.1.2 Linux	684
67 Системные вызовы (syscall-ы)	686
67.1 Linux	686
67.2 Windows	687
68 Linux	688
68.1 Адресно-независимый код	688

⁵Compact Disc File System

68.1.1 Windows	690
68.2 Трюк с <i>LD_PRELOAD</i> в Linux	690
69 Windows NT	693
69.1 CRT (win32)	693
69.2 Win32 PE	696
69.2.1 Терминология	696
69.2.2 Базовый адрес	697
69.2.3 Subsystem	697
69.2.4 Версия ОС	698
69.2.5 Секции	698
69.2.6 Релоки	699
69.2.7 Экспорты и импорты	699
69.2.8 Ресурсы	702
69.2.9 .NET	702
69.2.10 TLS	702
69.2.11 Инструменты	702
69.2.12 Further reading	702
69.3 Windows SEH	703
69.3.1 Забудем на время о MSVC	703
69.3.2 Теперь вспомним MSVC	707
69.3.3 Windows x64	721
69.3.4 Больше о SEH	725
69.4 Windows NT: Критические секции	725
VII Инструменты	727
70 Дизассемблер	728
70.1 IDA	728
71 Отладчик	729
71.1 OllyDbg	729
71.2 GDB	729
71.3 tracer	729
72 Трассировка системных вызовов	730
72.0.1 strace / dtruss	730
73 Декомпиляторы	731
74 Прочие инструменты	732
VIII Примеры из практики	733
75 Шутка с task manager (Windows Vista)	735
75.1 Использование LEA для загрузки значений	737
76 Шутка с игрой Color Lines	739
77 Сапёр (Windows XP)	743
77.1 Упражнения	747
78 Донглы	748
78.1 Пример #1: MacOS Classic и PowerPC	748
78.2 Пример #2: SCO OpenServer	755
78.2.1 Дешифровка сообщений об ошибке	762
78.3 Пример #3: MS-DOS	764
79 «QR9»: Любительская криптосистема, вдохновленная кубиком Рубика	770
80 SAP	798
80.1 Касательно сжимания сетевого трафика в клиенте SAP	798
80.2 Функции проверки пароля в SAP 6.0	808

81 Oracle RDBMS	813
81.1 Таблица V\$VERSION в Oracle RDBMS	813
81.2 Таблица X\$KSMLRU в Oracle RDBMS	820
81.3 Таблица V\$TIMER в Oracle RDBMS	822
82 Вручную написанный на ассемблере код	826
82.1 Тестовый файл EICAR	826
83 Демо	828
83.1 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10	828
83.1.1 Версия 42-х байт от Trixter	828
83.1.2 Моя попытка укоротить версию Trixter: 27 байт	829
83.1.3 Использование случайного мусора в памяти как источника случайных чисел	829
83.1.4 Вывод	830
83.2 Множество Мандельброта	831
83.2.1 Теория	832
83.2.2 Вернемся к демо	837
83.2.3 Моя «исправленная» версия	839
84 "Прикуп" в игре "Марьяж"	841
85 Другие примеры	849
IX Примеры разбора закрытых (proprietary) форматов файлов	850
86 Прimitives XOR-шифрование	851
86.1 Norton Guide: простейшее однобайтное XOR-шифрование	852
86.1.1 Энтропия	853
86.2 Простейшее четырехбайтное XOR-шифрование	855
86.2.1 Упражнение	858
87 Файл сохранения состояния в игре Millenium	859
88 Oracle RDBMS: .SYM-файлы	866
89 Oracle RDBMS: .MSB-файлы	876
89.1 Вывод	881
X Прочее	883
90 nrad	884
91 Модификация исполняемых файлов	886
91.1 Текстовые строки	886
91.2 x86-код	886
92 Compiler intrinsic	887
93 Аномалии компиляторов	888
94 OpenMP	889
94.1 MSVC	891
94.2 GCC	893
95 Itanium	895
96 Модель памяти в 8086	898
97 Перестановка basic block-ов	899
97.1 Profile-guided optimization	899
XI Что стоит почитать	901
98 Книги и прочие материалы	902

98.1 Reverse Engineering	902
98.2 Windows	902
98.3 Си/Си++	902
98.4 Си/Си++	902
98.5 x86 / x86-64	903
98.6 ARM	903
98.7 Java	903
98.8 UNIX	903
98.9 Криптография	903
98.10Cryptography	903
99 Прочее	904
Послесловие	906
100Вопросы?	906
Приложение	908
A x86	908
A.1 Терминология	908
A.2 Регистры общего пользования	908
A.2.1 RAX/EAX/AX/AL	908
A.2.2 RBX/EBX/BX/BL	909
A.2.3 RCX/ECX/CX/CL	909
A.2.4 RDX/EDX/DX/DI	909
A.2.5 RSI/ESI/SI/SIL	909
A.2.6 RDI/EDI/DI/DIL	909
A.2.7 R8/R8D/R8W/R8L	909
A.2.8 R9/R9D/R9W/R9L	910
A.2.9 R10/R10D/R10W/R10L	910
A.2.10 R11/R11D/R11W/R11L	910
A.2.11 R12/R12D/R12W/R12L	910
A.2.12 R13/R13D/R13W/R13L	910
A.2.13 R14/R14D/R14W/R14L	910
A.2.14 R15/R15D/R15W/R15L	911
A.2.15 RSP/ESP/SP/SPL	911
A.2.16 RBP/EBP/BP/BPL	911
A.2.17 RIP/EIP/IP	911
A.2.18 CS/DS/ES/SS/FS/GS	911
A.2.19 Регистр флагов	912
A.3 FPU регистры	912
A.3.1 Регистр управления	912
A.3.2 Регистр статуса	913
A.3.3 Tag Word	913
A.4 SIMD регистры	914
A.4.1 MMX регистры	914
A.4.2 SSE и AVX регистры	914
A.5 Отладочные регистры	914
A.5.1 DR6	914
A.5.2 DR7	914
A.6 Инструкции	915
A.6.1 Префиксы	915
A.6.2 Наиболее часто используемые инструкции	916
A.6.3 Реже используемые инструкции	920
A.6.4 Инструкции FPU	924
A.6.5 Инструкции с печатаемым ASCII-опкодом	925
B ARM	927
B.1 Терминология	927
B.2 Версии	927
B.3 32-битный ARM (AArch32)	927
B.3.1 Регистры общего пользования	927

B.3.2	Current Program Status Register (CPSR)	928
B.3.3	Регистры VFP (для чисел с плавающей точкой) и NEON	928
B.4	64-битный ARM (AArch64)	928
B.4.1	Регистры общего пользования	928
B.5	Инструкции	929
B.5.1	Таблица условных кодов	929
C	MIPS	930
C.1	Регистры	930
C.1.1	Регистры общего пользования GPR ⁶	930
C.1.2	Регистры для работы с числами с плавающей точкой	930
C.2	Инструкции	930
C.2.1	Инструкции перехода	931
D	Некоторые библиотечные функции GCC	932
E	Некоторые библиотечные функции MSVC	933
F	Cheatsheets	934
F.1	IDA	934
F.2	OllyDbg	934
F.3	MSVC	935
F.4	GCC	935
F.5	GDB	935
	Список принятых сокращений	938
	Глоссарий	942
	Предметный указатель	944

⁶General Purpose Registers (регистры общего пользования)

Предисловие

У термина «[reverse engineering](#)» несколько популярных значений: 1) исследование скомпилированных программ; 2) сканирование трехмерной модели для последующего копирования; 3) восстановление структуры СУБД. Настоящий сборник заметок связан с первым значением.

Рассмотренные темы

x86/x64, ARM/ARM64, MIPS, Java/JVM.

Затронутые темы

Oracle RDBMS ([81](#) (стр. [813](#))), Itanium ([95](#) (стр. [895](#))), донглы для защиты от копирования ([78](#) (стр. [748](#))), LD_PRELOAD ([68.2](#) (стр. [690](#))), переполнение стека, ELF⁷, формат файла PE в win32 ([69.2](#) (стр. [696](#))), x86-64 ([27.1](#) (стр. [417](#))), критические секции ([69.4](#) (стр. [725](#))), системные вызовы ([67](#) (стр. [686](#))), TLS⁸, адресно-независимый код (PIC⁹) ([68.1](#) (стр. [688](#))), profile-guided optimization ([97.1](#) (стр. [899](#))), C++ STL ([52.4](#) (стр. [556](#))), OpenMP ([94](#) (стр. [889](#))), SEH ([69.3](#) (стр. [703](#))).

Упражнения и задачи

...все перемещены на отдельный сайт: <http://challenges.re>.

Об авторе



Денис Юричев — опытный reverse engineer и программист. С ним можно контактировать по емейлу: [dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com).

Отзывы о книге *Reverse Engineering для начинающих*

- «It's very well done .. and for free .. amazing.»¹⁰ Daniel Bilar, Siege Technologies, LLC.
- «... excellent and free»¹¹ Pete Finnigan, гуру по безопасности Oracle RDBMS.
- «... book is interesting, great job!» Michael Sikorski, автор книги *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.
- «... my compliments for the very nice tutorial!» Herbert Bos, профессор университета Vrije Universiteit Amsterdam, соавтор *Modern Operating Systems (4th Edition)*.
- «... It is amazing and unbelievable.» Luis Rocha, CISSP / ISSAP, Technical Manager, Network & Information Security at Verizon Business.
- «Thanks for the great work and your book.» Joris van de Vis, специалист по SAP Netweaver & Security .

⁷Формат исполняемых файлов, использующийся в Linux и некоторых других *NIX

⁸Thread Local Storage

⁹Position Independent Code: [68.1](#) (стр. [688](#))

¹⁰twitter.com/daniel_bilar/status/436578617221742593

¹¹twitter.com/petefinnigan/status/400551705797869568

- «... reasonable intro to some of the techniques.»¹² Mike Stay, преподаватель в Federal Law Enforcement Training Center, Georgia, US.
- «I love this book! I have several students reading it at the moment, plan to use it in graduate course.»¹³ Сергей Братусь, Research Assistant Professor в отделе Computer Science в Dartmouth College
- «Dennis @Yurichev has published an impressive (and free!) book on reverse engineering»¹⁴ Tanel Poder, эксперт по настройке производительности Oracle RDBMS .
- «This book is some kind of Wikipedia to beginners...» Archer, Chinese Translator, IT Security Researcher.
- «Прочел Вашу книгу — отличная работа, рекомендую на своих курсах студентам в качестве учебного пособия». Николай Ильин, преподаватель в ФТИ НТУУ «КПИ» и DefCon-UA

Благодарности

Тем, кто много помогал мне отвечая на массу вопросов: Андрей «herm1t» Баранович, Слава «Avid» Казаков.

Тем, кто присылал замечания об ошибках и неточностях: Станислав «Beaver» Бобрицкий, Александр Лысенко, Shell Rocket, Zhu Ruijin, Changmin Heo, Александр «Solar Designer» Песляк, Vitor Vidal, Федерико Рамондино, Марк Уилсон, Stijn Crevits..

Просто помогли разными способами: Андрей Зубинский, Arnaud Patard (rtp на #debian-arm IRC), noshadow на #gcc IRC, Александр Автаев, Mohsen Mostafa Jokar.

Переводчикам на китайский язык: Antiy Labs (antiy.cn), Archer.

Переводчику на корейский язык: Byungho Min.

Переводчику на голландский язык: Cedric Sambre (AKA Midas).

Переводчикам на испанский язык: Diego Boy, Luis Alberto Espinosa Calvo.

Переводчикам на португальский язык: Thales Stevan de A. Gois.

Переводчику на итальянский язык: Федерико Рамондино.

Корректорам: Александр «Lstar» Черненький, Владимир Ботов, Андрей Бражук, Марк «Logxen» Купер, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Васил Колев сделал очень много исправлений и указал на многие ошибки.

За иллюстрации и обложку: Андрей Нечаевский.

И ещё всем тем на github.com кто присылал замечания и исправления.

Было использовано множество пакетов $\text{\LaTeX}$. Их авторов я также хотел бы поблагодарить.

Жертвователи

Те, кто поддерживал меня во время написании этой книги:

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar (\$50), James Truscott (\$4.5), Luis Rocha (\$63), Joris van de Vis (\$127), Richard S Shultz (\$20), Jang Minchang (\$20), Shade Atlas (5 AUD), Yao Xiao (\$10), Pawel Szczur (40 CHF), Justin Simms (\$20), Shawn the R0ck (\$27), Ki Chan Ahn (\$50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao (\$50), Philippe Teuwen (\$4), Martin Haeberli (\$10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai (\$15), Bayna AlZaabi (\$75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop (\$50), Nicolas Werner (€12), Jeremy Brown (\$100), Alexandre Borges (\$25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent (\$30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe (\$14), Maxim Dyakonov (\$3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov (\$100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei (\$40), Z0vsky (€10), Yu Dai (\$10).

Огромное спасибо каждому!

¹² [reddit](https://www.reddit.com)

¹³ twitter.com/sergeybratus/status/505590326560833536

¹⁴ twitter.com/TanelPoder/status/524668104065159169

Q: Зачем в наше время нужно изучать язык ассемблера?

A: Если вы не разработчик [ОС](#)¹⁵, вам наверное не нужно писать на ассемблере: современные компиляторы оптимизируют код намного лучше человека ¹⁶.

К тому же, современные [CPU](#)¹⁷ это крайне сложные устройства и знание ассемблера вряд ли поможет узнать их внутренности.

Но все-таки остается по крайней мере две области, где знание ассемблера может хорошо помочь: 1) исследование malware (*зловредов*) с целью анализа; 2) лучшее понимание вашего скомпилированного кода в процессе отладки. Таким образом, эта книга предназначена для тех, кто хочет скорее понимать ассемблер, нежели писать на нем, и вот почему здесь масса примеров, связанных с результатами работы компиляторов.

Q: Я кликнул на ссылку внутри PDF-документа, как теперь вернуться назад?

A: В Adobe Acrobat Reader нажмите сочетание Alt+LeftArrow.

Q: Могу ли я распечатать эту книгу? Использовать её для обучения?

A: Конечно, поэтому книга и лицензирована под лицензией Creative Commons (CC BY-SA 4.0).

Q: Почему эта книга бесплатная? Вы проделали большую работу. Это подозрительно, как и многие другие бесплатные вещи.

A: По моему опыту, авторы технической литературы делают это, в основном ради само-рекламы. Такой работой зарабатывать приличные деньги невозможно.

Q: Как можно найти работу reverse engineer-а?

A: На reddit, посвященному RE¹⁸, время от времени бывают hiring thread ([2013 Q3](#), [2014](#)). Посмотрите там.

В смежном субреддите «netsec» имеется похожий тред: [2014 Q2](#).

Q: Куда пойти учиться в Украине?

A: [НТУУ «КПИ»: «Аналіз програмного коду та бінарних вразливостей»](#); факультативы.

Q: У меня есть вопрос...

A: Напишите мне его емейлом ([dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com)).

О переводе на корейский язык

В январе 2015, издательство Ascom в Южной Корее сделало много работы в переводе и издании моей книги (по состоянию на август 2014) на корейский язык. Она теперь доступна на [их сайте](#).

Переводил Byungho Min ([twitter/tais9](#)). Обложку нарисовал мой хороший знакомый художник Андрей Нечаевский [facebook/andy](#). Они также имеют права на издании книги на корейском языке. Так что если вы хотите иметь *настоящую* книгу на полке на корейском языке и хотите поддержать мою работу, вы можете купить её.

¹⁵Операционная Система

¹⁶Очень хороший текст на эту тему: [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

¹⁷Central processing unit

¹⁸reddit.com/r/ReverseEngineering/

Часть I

Образцы кода

Глава 1

Метод

Когда автор этой книги учил Си, а затем Си++, он просто писал небольшие фрагменты кода, компилировал и смотрел, что получилось на ассемблере. Так было намного проще понять¹. Он делал это такое количество раз, что связь между кодом на Си/Си++ и тем, что генерирует компилятор, вбилась в его подсознание достаточно глубоко. После этого не трудно, глядя на код на ассемблере, сразу в общих чертах понимать, что там было написано на Си. Возможно это поможет кому-то ещё.

Иногда здесь используются достаточно древние компиляторы, чтобы получить самый короткий (или простой) фрагмент кода.

Упражнения

Когда автор этой книги учил ассемблер, он также часто компилировал короткие функции на Си и затем постепенно переписывал их на ассемблер, с целью получить как можно более короткий код. Наверное, этим не стоит заниматься в наше время на практике (потому что конкурировать с современными компиляторами в плане эффективности очень трудно), но это очень хороший способ разобраться в ассемблере лучше. Так что вы можете взять любой фрагмент кода на ассемблере в этой книге и постараться сделать его короче. Но не забывайте о тестировании своих результатов.

Уровни оптимизации и отладочная информация

Исходный код можно компилировать различными компиляторами с различными уровнями оптимизации. В типичном компиляторе этих уровней около трёх, где нулевой уровень — отключить оптимизацию. Различают также направления оптимизации кода по размеру и по скорости. Неоптимизирующий компилятор работает быстрее, генерирует более понятный (хотя и более объемный) код. Оптимизирующий компилятор работает медленнее и старается сгенерировать более быстрый (хотя и не обязательно краткий) код. Наряду с уровнями и направлениями оптимизации компилятор может включать в конечный файл отладочную информацию, производя таким образом код, который легче отлаживать. Одна очень важная черта отладочного кода в том, что он может содержать связи между каждой строкой в исходном коде и адресом в машинном коде. Оптимизирующие компиляторы обычно генерируют код, где целые строки из исходного кода могут быть оптимизированы и не присутствовать в итоговом машинном коде. Практикующий reverse engineer обычно сталкивается с обоими версиями, потому что некоторые разработчики включают оптимизацию, некоторые другие — нет. Вот почему мы постараемся поработать с примерами для обеих версий.

¹Честно говоря, он и до сих пор так делает, когда не понимает, как работает некий код.

Глава 2

Некоторые базовые понятия

2.1. Краткое введение в CPU

CPU это устройство исполняющее все программы.

Немного терминологии:

Инструкция : примитивная команда **CPU**. Простейшие примеры: перемещение между регистрами, работа с памятью, примитивные арифметические операции. Как правило, каждый **CPU** имеет свой набор инструкций (**ISA**).

Машинный код : код понимаемый **CPU**. Каждая инструкция обычно кодируется несколькими байтами.

Язык ассемблера : машинный код плюс некоторые расширения, призванные облегчить труд программиста: макросы, имена, и т.д.

Регистр CPU : Каждый **CPU** имеет некоторый фиксированный набор регистров общего назначения (**GPR**). ≈ 8 в x86, ≈ 16 в x86-64, ≈ 16 в ARM. Проще всего понимать регистр как временную переменную без типа. Можно представить, что вы пишете на **ЯП**¹ высокого уровня и у вас только 8 переменных шириной 32 (или 64) бита. Можно сделать очень много используя только их!

Откуда взялась разница между машинным кодом и **ЯП** высокого уровня? Ответ в том, что люди и **CPU**-ы отличаются друг от друга — человеку проще писать на **ЯП** высокого уровня вроде Си/Си++, Java, Python, а **CPU** проще работать с абстракциями куда более низкого уровня. Возможно, можно было бы придумать **CPU** исполняющий код **ЯП** высокого уровня, но он был бы значительно сложнее, чем те, что мы имеем сегодня. И наоборот, человеку очень неудобно писать на ассемблере из-за его низкоуровневости, к тому же, крайне трудно обойтись без мелких ошибок. Программа, переводящая код из **ЯП** высокого уровня в ассемблер называется *компилятором*².

2.1.1. Несколько слов о разнице между **ISA**

x86 всегда был архитектурой с опкодами переменной длины, так что когда пришла 64-битная эра, расширения x64 не очень сильно повлияли на **ISA**. ARM это **RISC**³-процессор разработанный с учетом опкодов одинаковой длины, что было некоторым преимуществом в прошлом. Так что в самом начале все инструкции ARM кодировались 4-мя байтами⁴. Это то, что сейчас называется «режим ARM». Потом они подумали, что это не очень экономично. На самом деле, самые используемые инструкции⁵ процессора на практике могут быть закодированы с использованием меньшего количества информации. Так что они добавили другую **ISA** с названием Thumb, где каждая инструкция кодируется всего лишь 2-мя байтами. Теперь это называется «режим Thumb». Но не все инструкции ARM могут быть закодированы в двух байтах, так что набор инструкций Thumb ограниченный. Код, скомпилированный для режима ARM и Thumb может сосуществовать в одной программе. Затем создатели ARM решили, что Thumb можно расширить: так появился Thumb-2 (в ARMv7). Thumb-2 это всё ещё двухбайтные инструкции, но некоторые новые инструкции имеют длину 4 байта. Распространено заблуждение, что Thumb-2 — это смесь ARM и Thumb. Это не верно. Режим Thumb-2 был дополнен до более полной поддержки возможностей процессора и теперь может легко конкурировать с режимом ARM. Основное количество приложений для iPod/iPhone/iPad скомпилировано для набора инструкций Thumb-2, потому что Xcode делает так по умолчанию. Потом появился 64-битный ARM. Это **ISA** снова с 4-байтными опкодами, без дополнительного режима Thumb. Но 64-битные требования повлияли на **ISA**, так что теперь у нас 3 набора инструкций ARM: режим ARM,

¹Язык Программирования

²В более старой русскоязычной литературе также часто встречается термин «транслятор».

³Reduced instruction set computing

⁴Кстати, инструкции фиксированного размера удобны тем, что всегда можно легко узнать адрес следующей (или предыдущей) инструкции. Эта особенность будет рассмотрена в секции об операторе switch() (14.2.2 (стр. 169)).

⁵А это MOV/PUSH/CALL/Jcc

2.1. КРАТКОЕ ВВЕДЕНИЕ В CPU

режим Thumb (включая Thumb-2) и ARM64. Эти наборы инструкций частично пересекаются, но можно сказать, это скорее разные наборы, нежели вариации одного. Следовательно, в этой книге постараемся добавлять фрагменты кода на всех трех ARM ISA. Существует ещё много RISC ISA с опкодами фиксированной 32-битной длины — это как минимум MIPS, PowerPC и Alpha AXP.

Глава 3

Простейшая функция

Наверное, простейшая из возможных функций это та что возвращает некоторую константу:

Вот, например:

Листинг 3.1: Код на Си/Си++

```
int f()
{
    return 123;
};
```

Скомпилируем её!

3.1. x86

И вот что делает оптимизирующий GCC:

Листинг 3.2: Оптимизирующий GCC/MSVC (вывод на ассемблере)

```
f:
    mov     eax, 123
    ret
```

Здесь только две инструкции. Первая помещает значение 123 в регистр **EAX**, который используется для передачи возвращаемых значений. Вторая это **RET**, которая возвращает управление в вызывающую функцию.

Вызывающая функция возьмет результат из регистра **EAX**.

3.2. ARM

А что насчет ARM?

Листинг 3.3: Оптимизирующий Keil 6/2013 (Режим ARM) ASM Output

```
f PROC
    MOV     r0,#0x7b ; 123
    BX      lr
    ENDP
```

ARM использует регистр **R0** для возврата значений, так что здесь 123 помещается в **R0**.

Адрес возврата (**RA**¹) в ARM не сохраняется в локальном стеке, а в регистре **LR**². Так что инструкция **BX LR** делает переход по этому адресу, и это то же самое что и вернуть управление в вызывающую ф-цию.

Нужно отметить, что название инструкции **MOV** в x86 и ARM сбивает с толку.

На самом деле, данные не *перемещаются*, а скорее *копируются*.

¹Адрес возврата

²Link Register

3.3. MIPS

Есть два способа называть регистры в мире MIPS. По номеру (от \$0 до \$31) или по псевдоимени (\$V0, \$A0, и т.д.).

Вывод на ассемблере в GCC показывает регистры по номерам:

Листинг 3.4: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
j      $31
li      $2, 123          # 0x7b
```

...а [IDA³](#) — по псевдоименам:

Листинг 3.5: Оптимизирующий GCC 4.4.5 (IDA)

```
jr      $ra
li      $v0, 0x7B
```

Так что регистр \$2 (или \$V0) используется для возврата значений. **LI** это “Load Immediate”, и это эквивалент **MOV** в MIPS.

Другая инструкция это инструкция перехода (J или JR), которая возвращает управление в [вызывающую ф-цию](#), переходя по адресу в регистре \$31 (или \$RA).

Это аналог регистра [LR](#) в ARM.

Но почему инструкция загрузки (LI) и инструкция перехода (J или JR) поменены местами? Это артефакт [RISC](#) и называется он “branch delay slot”.

На самом деле, нам не нужно вникать в эти детали. Нужно просто запомнить: в MIPS инструкция после инструкции перехода выполняется *перед* инструкцией перехода.

Таким образом, инструкция перехода всегда поменена местами с той, которая должна быть исполнена перед ней.

3.3.1. Еще кое-что об именах инструкций и регистров в MIPS

Имена регистров и инструкций в мире MIPS традиционно пишутся в нижнем регистре. Но мы будем использовать верхний регистр, потому что имена инструкций и регистров других [ISA](#) в этой книге так же в верхнем регистре.

³ Интерактивный дизассемблер и отладчик, разработан [Hex-Rays](#)

Глава 4

Hello, world!

Продолжим, используя знаменитый пример из книги [Брайан Керниган, Деннис Ритчи, *Язык программирования Си*, (1988)]:

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

4.1. x86

4.1.1. MSVC

Компилируем в MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(Ключ `/Fa` означает сгенерировать листинг на ассемблере)

Листинг 4.1: MSVC 2010

```
CONST    SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST    ENDS
PUBLIC   _main
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG3830
    call    _printf
    add     esp, 4
    xor     eax, eax
    pop     ebp
    ret     0
_main    ENDP
_TEXT    ENDS
```

MSVC выдает листинги в синтаксисе Intel. Разница между синтаксисом Intel и AT&T будет рассмотрена немного позже:

Компилятор сгенерировал файл `1.obj`, который впоследствии будет слинкован линкером в `1.exe`. В нашем случае этот файл состоит из двух сегментов: `CONST` (для данных-констант) и `_TEXT` (для кода).

Строка `hello, world` в Си/Си++ имеет тип `const char[]` [Bjarne Stroustrup, *The C++ Programming Language*, 4th Edition, (2013)p176, 7.3.2], однако не имеет имени. Но компилятору нужно как-то с ней работать, поэтому он дает ей внутреннее имя `$SG3830`.

4.1. X86

Поэтому пример можно было бы переписать вот так:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Вернемся к листингу на ассемблере. Как видно, строка заканчивается нулевым байтом — это требования стандарта Си/Си++ для строк. Больше о строках в Си/Си++: [58.1.1](#) (стр. 648).

В сегменте кода `_TEXT` находится пока только одна функция: `main()`. Функция `main()`, как и практически все функции, начинается с пролога и заканчивается эпилогом¹.

Далее следует вызов функции `printf()`: `CALL _printf`. Перед этим вызовом адрес строки (или указатель на неё) с нашим приветствием ("Hello, world!") при помощи инструкции `PUSH` помещается в стек.

После того, как функция `printf()` возвращает управление в функцию `main()`, адрес строки (или указатель на неё) всё ещё лежит в стеке. Так как он больше не нужен, то [указатель стека](#) (регистр `ESP`) корректируется.

`ADD ESP, 4` означает прибавить 4 к значению в регистре `ESP`.

Почему 4? Так как это 32-битный код, для передачи адреса нужно 4 байта. В x64-коде это 8 байт. `ADD ESP, 4` эквивалентно `POP регистр`, но без использования какого-либо регистра².

Некоторые компиляторы, например, Intel C++ Compiler, в этой же ситуации могут вместо `ADD` сгенерировать `POP ECX` (подобное можно встретить, например, в коде Oracle RDBMS, им скомпилированном), что почти то же самое, только портится значение в регистре `ECX`. Возможно, компилятор применяет `POP ECX`, потому что эта инструкция короче (1 байт у `POP` против 3 у `ADD`).

Вот пример использования `POP` вместо `ADD` из Oracle RDBMS:

Листинг 4.2: Oracle RDBMS 10.2 Linux (файл app.o)

.text:0800029A	push	ebx
.text:0800029B	call	qksfroChild
.text:080002A0	pop	ecx

После вызова `printf()` в оригинальном коде на Си/Си++ указано `return 0` — вернуть 0 в качестве результата функции `main()`.

В сгенерированном коде это обеспечивается инструкцией `XOR EAX, EAX`.

`XOR`, как легко догадаться — «исключающее ИЛИ»³, но компиляторы часто используют его вместо простого `MOV EAX, 0` — снова потому, что опкод короче (2 байта у `XOR` против 5 у `MOV`).

Некоторые компиляторы генерируют `SUB EAX, EAX`, что значит *отнять значение в `EAX` от значения в `EAX`*, что в любом случае даст 0 в результате.

Самая последняя инструкция `RET` возвращает управление в вызывающую функцию. Обычно это код Си/Си++ [CRT](#)⁴, который, в свою очередь, вернёт управление в [ОС](#).

4.1.2. GCC

Теперь скомпилируем то же самое компилятором GCC 4.4.1 в Linux: `gcc 1.c -o 1`. Затем при помощи [IDA](#) посмотрим как скомпилировалась функция `main()`. [IDA](#), как и MSVC, показывает код в синтаксисе Intel⁵.

¹Об этом смотрите подробнее в разделе о прологе и эпилоге функции ([5](#) (стр. 25)).

²Флаги процессора, впрочем, модифицируются

³[wikipedia](#)

⁴C runtime library

⁵Мы также можем заставить GCC генерировать листинги в этом формате при помощи ключей `-S -masm=intel`.

```

main      proc near
var_10    = dword ptr -10h

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     eax, offset aHelloWorld ; "hello, world\n"
        mov     [esp+10h+var_10], eax
        call    _printf
        mov     eax, 0
        leave
        retn
main      endp

```

Почти то же самое. Адрес строки `hello, world`, лежащей в сегменте данных, вначале сохраняется в `EAX`, затем записывается в стек. А ещё в прологе функции мы видим `AND ESP, 0FFFFFFF0h` — эта инструкция выравнивает значение в `ESP` по 16-байтной границе, делая все значения в стеке также выровненными по этой границе (процессор более эффективно работает с переменными, расположенными в памяти по адресам кратным 4 или 16)⁶.

`SUB ESP, 10h` выделяет в стеке 16 байт. Хотя, как будет видно далее, здесь достаточно только 4.

Это происходит потому, что количество выделяемого места в локальном стеке тоже выровнено по 16-байтной границе. Адрес строки (или указатель на строку) затем записывается прямо в стек без помощи инструкции `PUSH`. `var_10` одновременно и локальная переменная и аргумент для `printf()`. Подробнее об этом будет ниже.

Затем вызывается `printf()`.

В отличие от MSVC, GCC в компиляции без включенной оптимизации генерирует `MOV EAX, 0` вместо более короткого опкода.

Последняя инструкция `LEAVE` — это аналог команд `MOV ESP, EBP` и `POP EBP` — то есть возврат [указателя стека](#) и регистра `EBP` в первоначальное состояние. Это необходимо, т.к. в начале функции мы модифицировали регистры `ESP` и `EBP` (при помощи `MOV EBP, ESP` / `AND ESP, ...`).

4.1.3. GCC: Синтаксис AT&T

Попробуем посмотреть, как выглядит то же самое в синтаксисе AT&T языка ассемблера. Этот синтаксис больше распространен в UNIX-мире.

Листинг 4.4: компилируем в GCC 4.7.3

```
gcc -S 1_1.c
```

Получим такой файл:

Листинг 4.5: GCC 4.7.3

```

.file    "1_1.c"
.section .rodata
.LC0:
.string  "hello, world\n"
.text
.globl  main
.type   main, @function
main:
.LFB0:
.cfi_startproc
pushl   %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl    %esp, %ebp
.cfi_def_cfa_register 5

```

⁶Wikipedia: Выравнивание данных

4.1. X86

```
    andl    $-16, %esp
    subl    $16, %esp
    movl    $.LC0, (%esp)
    call    printf
    movl    $0, %eax
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE0:
    .size    main, .-main
    .ident   "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
    .section .note.GNU-stack,"",@progbits
```

Здесь много макросов (начинающихся с точки). Они нам пока не интересны.

Пока что, ради упрощения, мы можем их игнорировать (кроме макроса *.string*, при помощи которого кодируется последовательность символов, оканчивающихся нулем — такие же строки как в Си). И тогда получится следующее ⁷:

Листинг 4.6: GCC 4.7.3

```
.LC0:
    .string "hello, world\n"
main:
    pushl    %ebp
    movl     %esp, %ebp
    andl     $-16, %esp
    subl     $16, %esp
    movl     $.LC0, (%esp)
    call     printf
    movl     $0, %eax
    leave
    ret
```

Основные отличия синтаксиса Intel и AT&T следующие:

- Операнды записываются наоборот.

В Intel-синтаксисе: <инструкция> <операнд назначения> <операнд-источник>.

В AT&T-синтаксисе: <инструкция> <операнд-источник> <операнд назначения>.

Чтобы легче понимать разницу, можно запомнить следующее: когда вы работаете с синтаксисом Intel — можете в уме ставить знак равенства (=) между операндами, а когда с синтаксисом AT&T — мысленно ставьте стрелку направо (→) ⁸.

- AT&T: Перед именами регистров ставится символ процента (%), а перед числами символ доллара (\$). Вместо квадратных скобок используются круглые.
- AT&T: К каждой инструкции добавляется специальный символ, определяющий тип данных:
 - q — quad (64 бита)
 - l — long (32 бита)
 - w — word (16 бит)
 - b — byte (8 бит)

Возвращаясь к результату компиляции: он идентичен тому, который мы посмотрели в IDA. Одна мелочь: `0FFFFFFF0h` записывается как `$-16`. Это то же самое: `16` в десятичной системе это `0x10` в шестнадцатеричной. `-0x10` будет как раз `0xFFFFFFFF0` (в рамках 32-битных чисел).

Возвращаемый результат устанавливается в 0 обычной инструкцией `MOV`, а не `XOR`. `MOV` просто загружает значение в регистр. Её название не очень удачное (данные не перемещаются, а копируются). В других архитектурах подобная инструкция обычно носит название «LOAD» или «STORE» или что-то в этом роде.

⁷Кстати, для уменьшения генерации «лишних» макросов, можно использовать такой ключ GCC: `-fno-asynchronous-unwind-tables`

⁸Кстати, в некоторых стандартных функциях библиотеки Си (например, `memcpy()`, `strcpy()`) также применяется расстановка аргументов как в синтаксисе Intel: вначале указатель в памяти на блок назначения, затем указатель на блок-источник.

4.2. x86-64

4.2.1. MSVC – x86-64

Попробуем также 64-битный MSVC:

Листинг 4.7: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main PROC
    sub     rsp, 40
    lea     rcx, OFFSET FLAT:$SG2989
    call    printf
    xor     eax, eax
    add     rsp, 40
    ret     0
main ENDP
```

В x86-64 все регистры были расширены до 64-х бит и теперь имеют префикс **R-**. Чтобы поменьше задействовать стек (иными словами, поменьше обращаться кэш и внешней памяти), уже давно имелся довольно популярный метод передачи аргументов функции через регистры (*fastcall*) 65.3 (стр. 673). Т.е. часть аргументов функции передается через регистры и часть — через стек. В Win64 первые 4 аргумента функции передаются через регистры **RCX**, **RDX**, **R8**, **R9**. Это мы здесь и видим: указатель на строку в `printf()` теперь передается не через стек, а через регистр **RCX**. Указатели теперь 64-битные, так что они передаются через 64-битные части регистров (имеющие префикс **R-**). Но для обратной совместимости можно обращаться и к нижним 32 битам регистров используя префикс **E-**. Вот как выглядит регистр **RAX / EAX / AX / AL** в x86-64:

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RAX ^{x64}							
				EAX			
						AX	
						AH	AL

Функция `main()` возвращает значение типа *int*, который в Си/Си++, вероятно для лучшей совместимости и переносимости, оставили 32-битным. Вот почему в конце функции `main()` обнуляется не **RAX**, а **EAX**, т.е. 32-битная часть регистра. Также видно, что 40 байт выделяются в локальном стеке. Это «shadow space» которое мы будем рассматривать позже: 9.2.1 (стр. 96).

4.2.2. GCC – x86-64

Попробуем GCC в 64-битном Linux:

Листинг 4.8: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
    sub     rsp, 8
    mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
    xor     eax, eax ; количество переданных векторных регистров
    call    printf
    xor     eax, eax
    add     rsp, 8
    ret
```

В Linux, *BSD и Mac OS X для x86-64 также принят способ передачи аргументов функции через регистры [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]⁹.

6 первых аргументов передаются через регистры **RDI**, **RSI**, **RDX**, **RCX**, **R8**, **R9**, а остальные — через стек.

Так что указатель на строку передается через **EDI** (32-битную часть регистра). Но почему не через 64-битную часть, **RDI**?

⁹Также доступно здесь: <http://x86-64.org/documentation/abi.pdf>

4.3. GCC – ЕЩЁ КОЕ-ЧТО

Важно запомнить что в 64-битном режиме все инструкции `MOV`, записывающие что-либо в младшую 32-битную часть регистра, обнуляют старшие 32-бита (это можно найти в документации от Intel: [98.5](#) (стр. 903)). То есть, инструкция `MOV EAX, 011223344h` корректно запишет это значение в `RAX`, старшие биты сбросятся в ноль.

Если посмотреть в [IDA](#) скомпилированный объектный файл (.o), увидим также опкоды всех инструкций ¹⁰:

Листинг 4.9: GCC 4.4.6 x64

```
.text:00000000004004D0      main  proc near
.text:00000000004004D0  48 83 EC 08      sub     rsp, 8
.text:00000000004004D4  BF E8 05 40 00   mov     edi, offset format ; "hello, world\n"
.text:00000000004004D9  31 C0           xor     eax, eax
.text:00000000004004DB  E8 D8 FE FF FF   call    _printf
.text:00000000004004E0  31 C0           xor     eax, eax
.text:00000000004004E2  48 83 C4 08      add     rsp, 8
.text:00000000004004E6  C3             retn
.text:00000000004004E6      main  endp
```

Как видно, инструкция, записывающая в `EDI` по адресу `0x4004D4`, занимает 5 байт. Та же инструкция, записывающая 64-битное значение в `RDI`, занимает 7 байт. Возможно, GCC решил немного сэкономить. К тому же, вероятно, он уверен, что сегмент данных, где хранится строка, никогда не будет расположен в адресах выше `4GiB`.

Здесь мы также видим обнуление регистра `EAX` перед вызовом `printf()`. Это делается потому что по упомянутому выше стандарту передачи аргументов в *NIX для x86-64 в `EAX` передается количество задействованных векторных регистров.

4.3. GCC – ещё кое-что

Тот факт, что *анонимная* Си-строка имеет тип `const` ([4.1.1](#) (стр. 8)), и тот факт, что выделенные в сегменте констант Си-строки гарантировано неизменяемые (immutable), ведет к интересному следствию: компилятор может использовать определенную часть строки.

Вот простой пример:

```
#include <stdio.h>

int f1()
{
    printf ("world\n");
}

int f2()
{
    printf ("hello world\n");
}

int main()
{
    f1();
    f2();
}
```

Среднестатистический компилятор с Си/Си++ (включая MSVC) выделит место для двух строк, но вот что делает GCC 4.8.1:

Листинг 4.10: GCC 4.8.1 + листинг в IDA

```
f1      proc near
s       = dword ptr -1Ch

        sub     esp, 1Ch
        mov     [esp+1Ch+s], offset s ; "world\n"
        call    _puts
        add     esp, 1Ch
        retn
f1      endp
```

¹⁰Это нужно задать в **Options** → **Disassembly** → **Number of opcode bytes**

4.4. ARM

```
f2      proc near
s      = dword ptr -1Ch

        sub     esp, 1Ch
        mov     [esp+1Ch+s], offset aHello ; "hello "
        call    _puts
        add     esp, 1Ch
        retn
f2      endp

aHello  db 'hello '
s       db 'world',0xa,0
```

Действительно, когда мы выводим строку «hello world», эти два слова расположены в памяти вплотную друг к другу и `puts()`, вызываясь из функции `f2()`, вообще не знает, что эти строки разделены. Они и не разделены на самом деле, они разделены только «виртуально», в нашем листинге.

Когда `puts()` вызывается из `f1()`, он использует строку «world» плюс нулевой байт. `puts()` не знает, что там ещё есть какая-то строка перед этой!

Этот трюк часто используется (по крайней мере в GCC) и может сэкономить немного памяти.

4.4. ARM

Для экспериментов с процессором ARM несколько компиляторов было использовано:

- Популярный в embedded-среде Keil Release 6/2013.
- Apple Xcode 4.6.3 с компилятором LLVM-GCC 4.2 ¹¹.
- GCC 4.9 (Linaro) (для ARM64), доступный в виде исполняемого файла для win32 на <http://go.yurichev.com/17325>.

Везде в этой книге, если не указано иное, идет речь о 32-битном ARM (включая режимы Thumb и Thumb-2). Когда речь идет о 64-битном ARM, он называется здесь ARM64.

4.4.1. Неоптимизирующий Keil 6/2013 (Режим ARM)

Для начала скомпилируем наш пример в Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

Компилятор `armcc` генерирует листинг на ассемблере в формате Intel. Этот листинг содержит некоторые высокоуровневые макросы, связанные с ARM ¹², а нам важнее увидеть инструкции «как есть», так что посмотрим скомпилированный результат в IDA.

Листинг 4.11: Неоптимизирующий Keil 6/2013 (Режим ARM) IDA

```
.text:00000000      main
.text:00000000 10 40 2D E9      STMFD    SP!, {R4,LR}
.text:00000004 1E 0E 8F E2      ADR      R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB      BL       __2printf
.text:0000000C 00 00 A0 E3      MOV      R0, #0
.text:00000010 10 80 BD E8      LDMFD    SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld DCB "hello, world",0 ; DATA XREF: main+4
```

В вышеприведённом примере можно легко увидеть, что каждая инструкция имеет размер 4 байта. Действительно, ведь мы же компилировали наш код для режима ARM, а не Thumb.

Самая первая инструкция, `STMFD SP!, {R4,LR}` ¹³, работает как инструкция `PUSH` в x86, записывая значения двух регистров (`R4` и `LR`) в стек. Действительно, в выдаваемом листинге на ассемблере компилятор `armcc` для упрощения

¹¹Это действительно так: Apple Xcode 4.6.3 использует open-сорсный GCC как компилятор переднего плана и кодогенератор LLVM

¹²например, он показывает инструкции `PUSH` / `POP`, отсутствующие в режиме ARM

¹³`STMFD`¹⁴

4.4. ARM

указывает здесь инструкцию `PUSH {r4,lr}`. Но это не совсем точно, инструкция `PUSH` доступна только в режиме Thumb, поэтому, во избежание путаницы, я предложил работать в `IDA`.

Итак, эта инструкция уменьшает `SP`¹⁵, чтобы он указывал на место в стеке, свободное для записи новых значений, затем записывает значения регистров `R4` и `LR` по адресу в памяти, на который указывает измененный регистр `SP`.

Эта инструкция, как и инструкция `PUSH` в режиме Thumb, может сохранить в стеке одновременно несколько значений регистров, что может быть очень удобно. Кстати, такого в x86 нет. Также следует заметить, что `STMFD` — генерализация инструкции `PUSH` (то есть расширяет её возможности), потому что может работать с любым регистром, а не только с `SP`. Другими словами, `STMFD` можно использовать для записи набора регистров в указанном месте памяти.

Инструкция `ADR R0, aHelloWorld` прибавляет или отнимает значение регистра `PC`¹⁶ к смещению, где хранится строка `hello, world`. Причем здесь `PC`, можно спросить? Притом, что это так называемый «адресно-независимый код»¹⁷. Он предназначен для исполнения будучи не привязанным к каким-либо адресам в памяти. Другими словами, это относительная от `PC` адресация. В опкоде инструкции `ADR` указывается разница между адресом этой инструкции и местом, где хранится строка. Эта разница всегда будет постоянной, вне зависимости от того, куда был загружен `OC` наш код. Поэтому всё, что нужно — это прибавить адрес текущей инструкции (из `PC`), чтобы получить текущий абсолютный адрес нашей Си-строки.

Инструкция `BL __2printf`¹⁸ вызывает функцию `printf()`. Работа этой инструкции состоит из двух фаз:

- записать адрес после инструкции `BL (0xC)` в регистр `LR`;
- передать управление в `printf()`, записав адрес этой функции в регистр `PC`.

Ведь когда функция `printf()` закончит работу, нужно знать, куда вернуть управление, поэтому закончив работу, всякая функция передает управление по адресу, записанному в регистре `LR`.

В этом разница между «чистыми» `RISC`-процессорами вроде ARM и `CISC`¹⁹-процессорами как x86, где адрес возврата обычно записывается в стек (6 (стр. 26)).

Кстати, 32-битный абсолютный адрес (либо смещение) невозможно закодировать в 32-битной инструкции `BL`, в ней есть место только для 24-х бит. Поскольку все инструкции в режиме ARM имеют длину 4 байта (32 бита) и инструкции могут находиться только по адресам кратным 4, то последние 2 бита (всегда нулевых) можно не кодировать. В итоге имеем 26 бит, при помощи которых можно закодировать $current\_PC \pm \approx 32M$.

Следующая инструкция `MOV R0, #0`²⁰ просто записывает 0 в регистр `R0`. Ведь наша Си-функция возвращает 0, а возвращаемое значение всякая функция оставляет в `R0`.

Последняя инструкция `LDMFD SP!, R4,PC`²¹. Она загружает из стека (или любого другого места в памяти) значения для сохранения их в `R4` и `PC`, увеличивая *указатель стека* `SP`. Здесь она работает как аналог `POP`.

Н.В. Самая первая инструкция `STMFD` сохранила в стеке `R4` и `LR`, а *восстанавливаются* во время исполнения `LDMFD` регистры `R4` и `PC`.

Как мы уже знаем, в регистре `LR` обычно сохраняется адрес места, куда нужно всякой функции вернуть управление. Самая первая инструкция сохраняет это значение в стеке, потому что наша функция `main()` позже будет сама пользоваться этим регистром в момент вызова `printf()`. А затем, в конце функции, это значение можно сразу записать прямо в `PC`, таким образом, передав управление туда, откуда была вызвана наша функция.

Так как функция `main()` обычно самая главная в Си/Си++, управление будет возвращено в загрузчик `OC`, либо куда-то в `CRT` или что-то в этом роде.

Всё это позволяет избавиться от инструкции `BX LR` в самом конце функции.

`DCB` — директива ассемблера, описывающая массивы байт или ASCII-строк, аналог директивы `DB` в x86-ассемблере.

4.4.2. Неоптимизирующий Keil 6/2013 (Режим Thumb)

Скомпилируем тот же пример в Keil для режима Thumb:

¹⁵stack pointer. SP/ESP/RSP в x86/x64. SP в ARM.

¹⁶Program Counter. IP/EIP/RIP в x86/64. PC в ARM.

¹⁷Читайте больше об этом в соответствующем разделе (68.1 (стр. 688))

¹⁸Branch with Link

¹⁹Complex instruction set computing

²⁰Означает MOVE

²¹`LDMFD`²² — это инструкция, обратная `STMFD`

```
armcc.exe --thumb --c90 -O0 1.c
```

Получим (в IDA):

Листинг 4.12: Неоптимизирующий Keil 6/2013 (Режим Thumb) + IDA

```
.text:00000000      main
.text:00000000 10 B5      PUSH    {R4,LR}
.text:00000002 C0 A0      ADR     R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9 BL     __2printf
.text:00000008 00 20      MOVS    R0, #0
.text:0000000A 10 BD      POP     {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld DCB "hello, world",0 ; DATA XREF: main+2
```

Сразу бросаются в глаза двухбайтные (16-битные) опкоды — это, как уже было отмечено, Thumb.

Кроме инструкции `BL`. Но на самом деле она состоит из двух 16-битных инструкций. Это потому что в одном 16-битном опкоде слишком мало места для задания смещения, по которому находится функция `printf()`. Так что первая 16-битная инструкция загружает старшие 10 бит смещения, а вторая — младшие 11 бит смещения.

Как уже было упомянуто, все инструкции в Thumb-режиме имеют длину 2 байта (или 16 бит). Поэтому невозможна такая ситуация, когда Thumb-инструкция начинается по нечетному адресу.

Учитывая сказанное, последний бит адреса можно не кодировать. Таким образом, в Thumb-инструкции `BL` можно закодировать адрес $current\_PC \pm \approx 2M$.

Остальные инструкции в функции (`PUSH` и `POP`) здесь работают почти так же, как и описанные `STMFD` / `LDMFD`, только регистр `SP` здесь не указывается явно. `ADR` работает так же, как и в предыдущем примере. `MOVS` записывает 0 в регистр `R0` для возврата нуля.

4.4.3. Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Xcode 4.6.3 без включенной оптимизации выдает слишком много лишнего кода, поэтому включим оптимизацию компилятора (ключ `-O3`), потому что там меньше инструкций.

Листинг 4.13: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```
__text:000028C4      _hello_world
__text:000028C4 80 40 2D E9      STMFD    SP!, {R7,LR}
__text:000028C8 86 06 01 E3      MOV     R0, #0x1686
__text:000028CC 0D 70 A0 E1      MOV     R7, SP
__text:000028D0 00 00 40 E3      MOVT    R0, #0
__text:000028D4 00 00 8F E0      ADD     R0, PC, R0
__text:000028D8 C3 05 00 EB      BL      _puts
__text:000028DC 00 00 A0 E3      MOV     R0, #0
__text:000028E0 80 80 BD E8      LDMFD    SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0 DCB "Hello world!",0
```

Инструкции `STMFD` и `LDMFD` нам уже знакомы.

Инструкция `MOV` просто записывает число `0x1686` в регистр `R0` — это смещение, указывающее на строку «Hello world!».

Регистр `R7` (по стандарту, принятому в [iOS ABI Function Call Guide, (2010)]²³) это frame pointer, о нем будет рассказано позже.

Инструкция `MOVT R0, #0` (MOVE Top) записывает 0 в старшие 16 бит регистра. Дело в том, что обычная инструкция `MOV` в режиме ARM может записывать какое-либо значение только в младшие 16 бит регистра, ведь в ней нельзя закодировать больше. Помните, что в режиме ARM опкоды всех инструкций ограничены длиной в 32 бита. Конечно, это ограничение не касается перемещений данных между регистрами.

Поэтому для записи в старшие биты (с 16-го по 31-й включительно) существует дополнительная команда `MOVT`. Впрочем, здесь её использование избыточно, потому что инструкция `MOV R0, #0x1686` выше и так обнулила старшую часть регистра. Возможно, это недочет компилятора.

²³Также доступно здесь: <http://go.yurichev.com/17276>

4.4. ARM

Инструкция `ADD R0, PC, R0` прибавляет `PC` к `R0` для вычисления действительного адреса строки «Hello world!». Как нам уже известно, это «адресно-независимый код», поэтому такая корректура необходима.

Инструкция `BL` вызывает `puts()` вместо `printf()`.

Компилятор заменил вызов `printf()` на `puts()`. Действительно, `printf()` с одним аргументом это почти аналог `puts()`.

Почти, если принять условие, что в строке не будет управляющих символов `printf()`, начинающихся со знака процента. Тогда эффект от работы этих двух функций будет разным ²⁴.

Зачем компилятор заменил один вызов на другой? Наверное потому что `puts()` работает быстрее ²⁵. Видимо потому что `puts()` проталкивает символы в `stdout` не сравнивая каждый со знаком процента.

Далее уже знакомая инструкция `MOV R0, #0`, служащая для установки в 0 возвращаемого значения функции.

4.4.4. Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

По умолчанию Xcode 4.6.3 генерирует код для режима Thumb-2 примерно в такой манере:

Листинг 4.14: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

```
__text:00002B6C          _hello_world
__text:00002B6C 80 B5          PUSH          {R7,LR}
__text:00002B6E 41 F2 D8 30      MOVW           R0, #0x13D8
__text:00002B72 6F 46          MOV            R7, SP
__text:00002B74 C0 F2 00 00      MOVT.W        R0, #0
__text:00002B78 78 44          ADD           R0, PC
__text:00002B7A 01 F0 38 EA      BLX           _puts
__text:00002B7E 00 20          MOVS          R0, #0
__text:00002B80 80 BD          POP          {R7,PC}

...

__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld DCB "Hello world!",0xA,0
```

Инструкции `BL` и `BLX` в Thumb, как мы помним, кодируются как пара 16-битных инструкций, а в Thumb-2 эти *суперопкоды* расширены так, что новые инструкции кодируются здесь как 32-битные инструкции. Это можно заметить по тому что опкоды Thumb-2 инструкций всегда начинаются с `0xFx` либо с `0Ex`. Но в листинге IDA байты опкода переставлены местами. Это из-за того, что в процессоре ARM инструкции кодируются так: в начале последний байт, потом первый (для Thumb и Thumb-2 режима), либо, (для инструкций в режиме ARM) в начале четвертый байт, затем третий, второй и первый (т.е. другой *endianness*).

Вот так байты следуют в листингах IDA:

- для режимов ARM и ARM64: 4-3-2-1;
- для режима Thumb: 2-1;
- для пары 16-битных инструкций в режиме Thumb-2: 2-1-4-3.

Так что мы видим здесь что инструкции `MOVW`, `MOVT.W` и `BLX` начинаются с `0xFx`.

Одна из Thumb-2 инструкций это `MOVW R0, #0x13D8` — она записывает 16-битное число в младшую часть регистра `R0`, очищая старшие биты.

Ещё `MOVT.W R0, #0` — эта инструкция работает так же, как и `MOVT` из предыдущего примера, но она работает в Thumb-2.

Помимо прочих отличий, здесь используется инструкция `BLX` вместо `BL`. Отличие в том, что помимо сохранения адреса возврата в регистре `LR` и передаче управления в функцию `puts()`, происходит смена режима процессора с Thumb/Thumb-2 на режим ARM (либо назад). Здесь это нужно потому, что инструкция, куда ведет переход, выглядит так (она закодирована в режиме ARM):

```
__symbolstub1:00003FEC _puts          ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5      LDR PC, =__imp_puts
```

²⁴Также нужно заметить, что `puts()` не требует символа перевода строки `\n` в конце строки, поэтому его здесь нет.

²⁵ciselant.de/projects/gcc_printf/gcc_printf.html

4.4. ARM

Это просто переход на место, где записан адрес `puts()` в секции импортов. Итак, внимательный читатель может задать справедливый вопрос: почему бы не вызывать `puts()` сразу в том же месте кода, где он нужен? Но это не очень выгодно из-за экономии места и вот почему.

Практически любая программа использует внешние динамические библиотеки (будь то DLL в Windows, .so в *NIX либо .dylib в Mac OS X). В динамических библиотеках находятся часто используемые библиотечные функции, в том числе стандартная функция Си `puts()`.

В исполняемом бинарном файле (Windows PE .exe, ELF либо Mach-O) имеется секция импортов, список символов (функций либо глобальных переменных) импортируемых из внешних модулей, а также названия самих модулей. Загрузчик ОС загружает необходимые модули и, перебирая импортируемые символы в основном модуле, проставляет правильные адреса каждого символа. В нашем случае, `__imp_puts` это 32-битная переменная, куда загрузчик ОС запишет правильный адрес этой же функции во внешней библиотеке. Так что инструкция `LDR` просто берет 32-битное значение из этой переменной, и, записывая его в регистр `PC`, просто передает туда управление. Чтобы уменьшить время работы загрузчика ОС, нужно чтобы ему пришлось записать адрес каждого символа только один раз, в соответствующее, выделенное для них, место.

К тому же, как мы уже убедились, нельзя одной инструкцией загрузить в регистр 32-битное число без обращений к памяти. Так что наиболее оптимально выделить отдельную функцию, работающую в режиме ARM, чья единственная цель — передавать управление дальше, в динамическую библиотеку. И затем ссылаться на эту короткую функцию из одной инструкции (так называемую [thunk-функцию](#)) из Thumb-кода.

Кстати, в предыдущем примере (скомпилированном для режима ARM), переход при помощи инструкции `BL` ведет на такую же [thunk-функцию](#), однако режим процессора не переключается (отсюда отсутствие «X» в мнемонике инструкции).

Еще о thunk-функциях

Thunk-функции трудновато понять, вероятно, из-за путаницы в терминах. Проще всего представлять их как адаптеры-переходники из одного типа разъемов в другой. Например, адаптер позволяющее вставить в американскую розетку британскую вилку, или наоборот. Thunk-функции также иногда называются *wrapper-ами*. *Wrap* в английском языке это *обертывать, завертывать*. Вот еще несколько описаний этих функций:

“A piece of coding which provides an address:”, according to P. Z. Ingerman, who invented thunks in 1961 as a way of binding actual parameters to their formal definitions in Algol-60 procedure calls. If a procedure is called with an expression in the place of a formal parameter, the compiler generates a thunk which computes the expression and leaves the address of the result in some standard location.

...
Microsoft and IBM have both defined, in their Intel-based systems, a “16-bit environment” (with bletcherous segment registers and 64K address limits) and a “32-bit environment” (with flat addressing and semi-real memory management). The two environments can both be running on the same computer and OS (thanks to what is called, in the Microsoft world, WOW which stands for Windows On Windows). MS and IBM have both decided that the process of getting from 16- to 32-bit and vice versa is called a “thunk”; for Windows 95, there is even a tool, THUNK.EXE, called a “thunk compiler”.

([The Jargon File](#))

4.4.5. ARM64

GCC

Компилируем пример в GCC 4.8.1 для ARM64:

Листинг 4.15: Неоптимизирующий GCC 4.8.1 + objdump

```
1 0000000000400590 <main>:
2   400590:    a9bf7bfd      stp     x29, x30, [sp,#-16]!
3   400594:    910003fd      mov     x29, sp
4   400598:    90000000      adrp    x0, 400000 <_init-0x3b8>
5   40059c:    91192000      add     x0, x0, #0x648
6   4005a0:    97ffffa0      bl      400420 <puts@plt>
7   4005a4:    52800000      mov     w0, #0x0                                // #0
8   4005a8:    a8c17bfd      ldp     x29, x30, [sp],#16
9   4005ac:    d65f03c0      ret
10 ...
11 ...
```


4.4. ARM

```
12 Contents of section .rodata:
13 400640 01000200 00000000 48656c6c 6f210a00 .....Hello!..
14
```

В ARM64 нет режима Thumb и Thumb-2, только ARM, так что тут только 32-битные инструкции.

Регистров тут в 2 раза больше: B.4.1 (стр. 928). 64-битные регистры теперь имеют префикс **X-**, а их 32-битные части — **W-**.

Инструкция **STP** (*Store Pair*) сохраняет в стеке сразу два регистра: **X29** и **X30**. Конечно, эта инструкция может сохранять эту пару где угодно в памяти, но здесь указан регистр **SP**, так что пара сохраняется именно в стеке.

Регистры в ARM64 64-битные, каждый имеет длину в 8 байт, так что для хранения двух регистров нужно именно 16 байт.

Восклицательный знак ("!") после операнда означает, что сначала от **SP** будет отнято 16 и только затем значения из пары регистров будут записаны в стек.

Это называется *pre-index*. Больше о разнице между *post-index* и *pre-index* описано здесь: 29.2 (стр. 439).

Таким образом, в терминах более знакомого всем процессора x86, первая инструкция — это просто аналог пары инструкций **PUSH X29** и **PUSH X30**. **X29** в ARM64 используется как **FP**²⁶, а **X30** как **LR**, поэтому они сохраняются в прологе функции и восстанавливаются в эпилоге.

Вторая инструкция копирует **SP** в **X29** (или **FP**). Это нужно для установки стекового фрейма функции.

Инструкции **ADRP** и **ADD** нужны для формирования адреса строки «Hello!» в регистре **X0**, ведь первый аргумент функции передается через этот регистр. Но в ARM нет инструкций, при помощи которых можно записать в регистр длинное число (потому что сама длина инструкции ограничена 4-я байтами. Больше об этом здесь: 29.3.1 (стр. 440)). Так что нужно использовать несколько инструкций. Первая инструкция (**ADRP**) записывает в **X0** адрес 4-килобайтной страницы где находится строка, а вторая (**ADD**) просто прибавляет к этому адресу остаток. Читайте больше об этом: 29.4 (стр. 441).

$0x400000 + 0x648 = 0x400648$, и мы видим, что в секции данных **.rodata** по этому адресу как раз находится наша Си-строка «Hello!».

Затем при помощи инструкции **BL** вызывается **puts()**. Это уже рассматривалось ранее: 4.4.3 (стр. 17).

Инструкция **MOV** записывает 0 в **W0**. **W0** это младшие 32 бита 64-битного регистра **X0**:

Старшие 32 бита	младшие 32 бита
X0	
	W0

А результат функции возвращается через **X0**, и **main()** возвращает 0, так что вот так готовится возвращаемый результат.

Почему именно 32-битная часть? Потому в ARM64, как и в x86-64, тип *int* оставили 32-битным, для лучшей совместимости.

Следовательно, раз уж функция возвращает 32-битный *int*, то нужно заполнить только 32 младших бита регистра **X0**.

Для того, чтобы удостовериться в этом, немного отредактируем этот пример и перекомпилируем его.

Теперь **main()** возвращает 64-битное значение:

Листинг 4.16: **main()** возвращающая значение типа **uint64_t**

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
    printf ("Hello!\n");
    return 0;
}
```

Результат точно такой же, только **MOV** в той строке теперь выглядит так:

Листинг 4.17: Неоптимизирующий GCC 4.8.1 + objdump

```
4005a4:      d2800000      mov     x0, #0x0          // #0
```

²⁶Frame Pointer

4.5. MIPS

Далее при помощи инструкции `LDP` (*Load Pair*) восстанавливаются регистры `X29` и `X30`.

Восклицательного знака после инструкции нет. Это означает, что сначала значения достаются из стека, и только потом `SP` увеличивается на 16.

Это называется *post-index*.

В ARM64 есть новая инструкция: `RET`. Она работает так же как и `BX LR`, но там добавлен специальный бит, подсказывающий процессору, что это именно выход из функции, а не просто переход, чтобы процессор мог более оптимально исполнять эту инструкцию.

Из-за простоты этой функции оптимизирующий GCC генерирует точно такой же код.

4.5. MIPS

4.5.1. О «глобальном указателе» («global pointer»)

«Глобальный указатель» («global pointer») — это важная концепция в MIPS. Как мы уже возможно знаем, каждая инструкция в MIPS имеет размер 32 бита, поэтому невозможно закодировать 32-битный адрес внутри одной инструкции. Вместо этого нужно использовать пару инструкций (как это сделал GCC для загрузки адреса текстовой строки в нашем примере). С другой стороны, используя только одну инструкцию, возможно загружать данные по адресам в пределах $register - 32768 \dots register + 32767$, потому что 16 бит знакового смещения можно закодировать в одной инструкции). Так мы можем выделить какой-то регистр для этих целей и ещё выделить буфер в 64KiB для самых частоиспользуемых данных. Выделенный регистр называется «глобальный указатель» («global pointer») и он указывает на середину области 64KiB. Эта область обычно содержит глобальные переменные и адреса импортированных функций вроде `printf()`, потому что разработчики GCC решили, что получение адреса функции должно быть как можно более быстрой операцией, исполняющейся за одну инструкцию вместо двух. В ELF-файле эта 64KiB-область находится частично в секции `.sbss` («small BSS²⁷») для неинициализированных данных и в секции `.sdata` («small data») для инициализированных данных. Это значит что программист может выбирать, к чему нужен как можно более быстрый доступ, и затем расположить это в секциях `.sdata/.sbss`. Некоторые программисты «старой школы» могут вспомнить модель памяти в MS-DOS 96 (стр. 898) или в менеджерах памяти вроде XMS/EMS, где вся память делилась на блоки по 64KiB.

Эта концепция применяется не только в MIPS. По крайней мере PowerPC также использует эту технику.

4.5.2. Оптимизирующий GCC

Рассмотрим следующий пример, иллюстрирующий концепцию «глобального указателя».

Листинг 4.18: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
1 $LC0:
2 ; \000 это ноль в восьмиричной системе:
3     .ascii "Hello, world!\012\000"
4 main:
5 ; пролог функции
6 ; установить GP:
7     lui    $28,%hi(__gnu_local_gp)
8     addiu  $sp,$sp,-32
9     addiu  $28,$28,%lo(__gnu_local_gp)
10 ; сохранить RA в локальном стеке:
11     sw     $31,28($sp)
12 ; загрузить адрес функции puts() из GP в $25:
13     lw     $25,%call16(puts)($28)
14 ; загрузить адрес текстовой строки в $4 ($a0):
15     lui    $4,%hi($LC0)
16 ; перейти на puts(), сохранив адрес возврата в link-регистре:
17     jalr   $25
18     addiu  $4,$4,%lo($LC0) ; branch delay slot
19 ; восстановить RA:
20     lw     $31,28($sp)
21 ; скопировать 0 из $zero в $v0:
22     move   $2,$0
23 ; вернуть управление сделав переход по адресу в RA:
24     j      $31
25 ; эпилог функции:
```

²⁷Block Started by Symbol

Как видно, регистр \$GP в прологе функции выставляется в середину этой области. Регистр RA сохраняется в локальном стеке. Здесь также используется puts() вместо printf(). Адрес функции puts() загружается в \$25 инструкцией LW («Load Word»). Затем адрес текстовой строки загружается в \$4 парой инструкций LUI («Load Upper Immediate») и ADDIU («Add Immediate Unsigned Word»). LUI устанавливает старшие 16 бит регистра (поэтому в имени инструкции присутствует «upper») и ADDIU прибавляет младшие 16 бит к адресу. ADDIU следует за JALR (помните о branch delay slots?). Регистр \$4 также называется \$A0, который используется для передачи первого аргумента функции²⁸. JALR («Jump and Link Register») делает переход по адресу в регистре \$25 (там адрес puts()) при этом сохраняя адрес следующей инструкции (LW) в RA. Это так же как и в ARM. И ещё одна важная вещь: адрес сохраняемый в RA это адрес не следующей инструкции (потому что это delay slot и выполняется перед инструкцией перехода), а инструкции после неё (после delay slot). Таким образом во время исполнения JALR в RA записывается PC + 8. В нашем случае это адрес инструкции LW следующей после ADDIU.

LW («Load Word») в строке 20 восстанавливает RA из локального стека (эта инструкция скорее часть эпилога функции).

MOVE в строке 22 копирует значение из регистра \$0 (\$ZERO) в \$2 (\$V0).

В MIPS есть константный регистр, всегда содержащий ноль. Должно быть, разработчики MIPS решили что 0 это самая востребованная константа в программировании, так что пусть будет использоваться регистр \$0, всякий раз, когда будет нужен 0. Другой интересный факт: в MIPS нет инструкции, копирующей значения из регистра в регистр. На самом деле, MOVE DST, SRC это ADD DST, SRC, \$ZERO (DST = SRC + 0), которая делает тоже самое. Очевидно, разработчики MIPS хотели сделать как можно более компактную таблицу опкодов. Это не значит, что сложение происходит во время каждой инструкции MOVE. Скорее всего, эти псевдоинструкции оптимизируются в CPU и АЛУ²⁹ никогда не используется.

J в строке 24 делает переход по адресу в RA, и это работает как выход из функции. ADDIU после J на самом деле выполняется перед J (помните о branch delay slots?) и это часть эпилога функции.

Вот листинг сгенерированный IDA. Каждый регистр имеет свой псевдоним:

Листинг 4.19: Оптимизирующий GCC 4.4.5 (IDA)

```

1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10      = -0x10
4 .text:00000000 var_4      = -4
5 .text:00000000
6 ; пролог функции
7 ; установить GP:
8 .text:00000000          lui     $gp, (__gnu_local_gp >> 16)
9 .text:00000004          addiu   $sp, -0x20
10 .text:00000008          la      $gp, (__gnu_local_gp & 0xFFFF)
11 ; сохранить RA в локальном стеке:
12 .text:0000000C          sw      $ra, 0x20+var_4($sp)
13 ; сохранить GP в локальном стеке:
14 ; по какой-то причине, этой инструкции не было в ассемблерном выводе в GCC:
15 .text:00000010          sw      $gp, 0x20+var_10($sp)
16 ; загрузить адрес функции puts() из GP в $t9:
17 .text:00000014          lw      $t9, (puts & 0xFFFF)($gp)
18 ; сформировать адрес текстовой строки в $a0:
19 .text:00000018          lui     $a0, ($LC0 >> 16) # "Hello, world!"
20 ; перейти на puts(), сохранив адрес возврата в link-регистре:
21 .text:0000001C          jalr    $t9
22 .text:00000020          la      $a0, ($LC0 & 0xFFFF) # "Hello, world!"
23 ; восстановить RA:
24 .text:00000024          lw      $ra, 0x20+var_4($sp)
25 ; скопировать 0 из $zero в $v0:
26 .text:00000028          move    $v0, $zero
27 ; вернуть управление сделав переход по адресу в RA:
28 .text:0000002C          jr      $ra
29 ; эпилог функции:
30 .text:00000030          addiu   $sp, 0x20

```

Инструкция в строке 15 сохраняет GP в локальном стеке. Эта инструкция мистическим образом отсутствует в листинге от GCC, может быть из-за ошибки в самом GCC³⁰. Значение GP должно быть сохранено, потому что всякая функция

²⁸Таблица регистров в MIPS доступна в приложении C.1 (стр. 930)

²⁹Арифметико-логическое устройство

³⁰Очевидно, функция вывода листингов не так критична для пользователей GCC, поэтому там вполне могут быть неисправленные ошибки.

4.5. MIPS

может работать со своим собственным окном данных размером 64KiB. Регистр, содержащий адрес функции `puts()` называется `$T9`, потому что регистры с префиксом T- называются «temporaries» и их содержимое можно не сохранять.

4.5.3. Неоптимизирующий GCC

Неоптимизирующий GCC более многословный.

Листинг 4.20: Неоптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
1 $LC0:
2     .ascii "Hello, world!\012\000"
3 main:
4 ; пролог функции
5 ; сохранить RA ($31) и FP в стеке:
6     addiu $sp,$sp,-32
7     sw    $31,28($sp)
8     sw    $fp,24($sp)
9 ; установить FP (указатель стекового фрейма):
10    move  $fp,$sp
11 ; установить GP:
12    lui   $28,%hi(__gnu_local_gp)
13    addiu $28,$28,%lo(__gnu_local_gp)
14 ; загрузить адрес текстовой строки:
15    lui   $2,%hi($LC0)
16    addiu $4,$2,%lo($LC0)
17 ; загрузить адрес функции puts() используя GP:
18    lw    $2,%call16(puts)($28)
19    nop
20 ; вызвать puts():
21    move  $25,$2
22    jalr  $25
23    nop ; branch delay slot
24
25 ; восстановить GP из локального стека:
26    lw    $28,16($fp)
27 ; установить регистр $2 ($V0) в ноль:
28    move  $2,$0
29 ; эпилог функции.
30 ; восстановить SP:
31    move  $sp,$fp
32 ; восстановить RA:
33    lw    $31,28($sp)
34 ; восстановить FP:
35    lw    $fp,24($sp)
36    addiu $sp,$sp,32
37 ; переход на RA:
38    j     $31
39    nop ; branch delay slot
```

Мы видим, что регистр FP используется как указатель на фрейм стека. Мы также видим 3 `NOP`³¹-а. Второй и третий следуют за инструкциями перехода. Вероятно, компилятор GCC всегда добавляет `NOP`-ы (из-за *branch delay slots*) после инструкций переходов и затем, если включена оптимизация, от них может избавляться. Так что они остались здесь.

Вот также листинг от [IDA](#):

Листинг 4.21: Неоптимизирующий GCC 4.4.5 ([IDA](#))

```
1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10          = -0x10
4 .text:00000000 var_8          = -8
5 .text:00000000 var_4          = -4
6 .text:00000000
7 ; пролог функции
8 ; сохранить RA и FP в стеке:
9 .text:00000000                addiu    $sp, -0x20
10 .text:00000004                sw       $ra, 0x20+var_4($sp)
11 .text:00000008                sw       $fp, 0x20+var_8($sp)
```

³¹No OPeration

4.5. MIPS

```
12 ; установить FP (указатель стекового фрейма):
13 .text:0000000C          move    $fp, $sp
14 ; установить GP:
15 .text:00000010          la      $gp, __gnu_local_gp
16 .text:00000018          sw      $gp, 0x20+var_10($sp)
17 ; загрузить адрес текстовой строки:
18 .text:0000001C          lui     $v0, (aHelloWorld >> 16) # "Hello, world!"
19 .text:00000020          addiu   $a0, $v0, (aHelloWorld & 0xFFFF) # "Hello, world!"
20 ; загрузить адрес функции puts() используя GP:
21 .text:00000024          lw      $v0, (puts & 0xFFFF)($gp)
22 .text:00000028          or      $at, $zero ; NOP
23 ; вызвать puts():
24 .text:0000002C          move    $t9, $v0
25 .text:00000030          jalr    $t9
26 .text:00000034          or      $at, $zero ; NOP
27 ; восстановить GP из локального стека:
28 .text:00000038          lw      $gp, 0x20+var_10($fp)
29 ; установить регистр $2 ($V0) в ноль:
30 .text:0000003C          move    $v0, $zero
31 ; эпилог функции.
32 ; восстановить SP:
33 .text:00000040          move    $sp, $fp
34 ; восстановить RA:
35 .text:00000044          lw      $ra, 0x20+var_4($sp)
36 ; восстановить FP:
37 .text:00000048          lw      $fp, 0x20+var_8($sp)
38 .text:0000004C          addiu   $sp, 0x20
39 ; переход на RA:
40 .text:00000050          jr      $ra
41 .text:00000054          or      $at, $zero ; NOP
```

Интересно что **IDA** распознала пару инструкций **LUI** / **ADDIU** и собрала их в одну псевдоинструкцию **LA** («Load Address») в строке 15. Мы также видим, что размер этой псевдоинструкции 8 байт! Это псевдоинструкция (или *макрос*), потому что это не настоящая инструкция MIPS, а скорее просто удобное имя для пары инструкций.

Ещё кое что: **IDA** не распознала **NOP**-инструкции в строках 22, 26 и 41.

Это **OR \$AT, \$ZERO**. По своей сути это инструкция, применяющая операцию *ИЛИ* к содержимому регистра **\$AT** с нулем, что, конечно же, холостая операция. MIPS, как и многие другие **ISA**, не имеет отдельной **NOP**-инструкции.

4.5.4. Роль стекового фрейма в этом примере

Адрес текстовой строки передается в регистре. Так зачем устанавливать локальный стек? Причина в том, что значения регистров **RA** и **GP** должны быть сохранены где-то (потому что вызывается **printf()**) и для этого используется локальный стек.

Если бы это была **leaf function**, тогда можно было бы избавиться от пролога и эпилога функции. Например: 3.3 (стр. 7).

4.5.5. Оптимизирующий GCC: загрузим в GDB

Листинг 4.22: пример сессии в GDB

```
root@debian-mips:~# gcc hw.c -O3 -o hw
root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "mips-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
```

4.6. Вывод

```
Starting program: /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:    lui      gp,0x42
0x00400644 <main+4>:    addiu    sp,sp,-32
0x00400648 <main+8>:    addiu    gp,gp,-30624
0x0040064c <main+12>:   sw        ra,28(sp)
0x00400650 <main+16>:   sw        gp,16(sp)
0x00400654 <main+20>:   lw        t9,-32716(gp)
0x00400658 <main+24>:   lui      a0,0x40
0x0040065c <main+28>:   jalr     t9
0x00400660 <main+32>:   addiu    a0,a0,2080
0x00400664 <main+36>:   lw        ra,28(sp)
0x00400668 <main+40>:   move     v0,zero
0x0040066c <main+44>:   jr       ra
0x00400670 <main+48>:   addiu    sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:      "hello, world"
(gdb)
```

4.6. Вывод

Основная разница между кодом x86/ARM и x64/ARM64 в том, что указатель на строку теперь 64-битный. Действительно, ведь для того современные CPU и стали 64-битными, потому что подешевела память, её теперь можно поставить в компьютер намного больше, и чтобы её адресовать, 32-х бит уже недостаточно. Поэтому все указатели теперь 64-битные.

4.7. Упражнения

- <http://challenges.re/48>
- <http://challenges.re/49>

Глава 5

Пролог и эпилог функций

Пролог функции это инструкции в самом начале функции. Как правило это что-то вроде такого фрагмента кода:

```
push    ebp
mov     ebp, esp
sub     esp, X
```

Эти инструкции делают следующее: сохраняют значение регистра **EBP** на будущее, выставляют **EBP** равным **ESP**, затем подготавливают место в стеке для хранения локальных переменных.

EBP сохраняет свое значение на протяжении всей функции, он будет использоваться здесь для доступа к локальным переменным и аргументам. Можно было бы использовать и **ESP**, но он постоянно меняется и это не очень удобно.

Эпилог функции аннулирует выделенное место в стеке, восстанавливает значение **EBP** на старое и возвращает управление в вызывающую функцию:

```
mov     esp, ebp
pop     ebp
ret     0
```

Пролог и эпилог функции обычно находятся в дизассемблерах для отделения функций друг от друга.

5.1. Рекурсия

Наличие эпилога и пролога может несколько ухудшить эффективность рекурсии.

Больше о рекурсии в этой книге: [37.3](#) (стр. [465](#)).

Глава 6

Стек

Стек в информатике — это одна из наиболее фундаментальных структур данных ¹. АКА² LIFO³.

Технически это просто блок памяти в памяти процесса + регистр **ESP** в x86 или **RSP** в x64, либо **SP** в ARM, который указывает где-то в пределах этого блока.

Часто используемые инструкции для работы со стеком — это **PUSH** и **POP** (в x86 и Thumb-режиме ARM). **PUSH** уменьшает **ESP / RSP / SP** на 4 в 32-битном режиме (или на 8 в 64-битном), затем записывает по адресу, на который указывает **ESP / RSP / SP**, содержимое своего единственного операнда.

POP это обратная операция — сначала достает из **указателя стека** значение и помещает его в операнд (который очень часто является регистром) и затем увеличивает указатель стека на 4 (или 8).

В самом начале **регистр-указатель** указывает на конец стека. Конец стека находится в начале блока памяти, выделенного под стек. Это странно, но это так. **PUSH** уменьшает **регистр-указатель**, а **POP** — увеличивает.

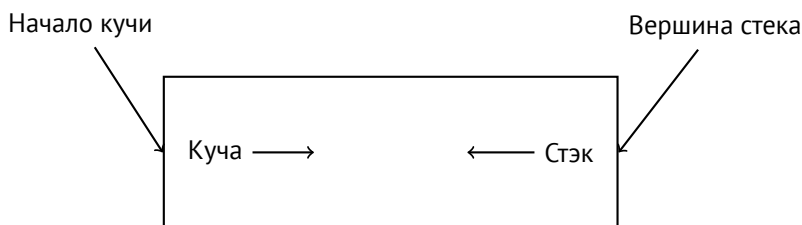
В процессоре ARM, тем не менее, есть поддержка стеков, растущих как в сторону уменьшения, так и в сторону увеличения.

Например, инструкции **STMTD/LDMFD**, **STMED⁴/LDMED⁵** предназначены для descending-стека (растет назад, начиная с высоких адресов в сторону низких). Инструкции **STMFA⁶/LDMFA⁷**, **STMEA⁸/LDMEA⁹** предназначены для ascending-стека (растет вперед, начиная с низких адресов в сторону высоких).

6.1. Почему стек растет в обратную сторону?

Интуитивно мы можем подумать, что, как и любая другая структура данных, стек мог бы расти вперед, т.е. в сторону увеличения адресов.

Причина, почему стек растет назад, вероятно, историческая. Когда компьютеры были большие и занимали целую комнату, было очень легко разделить сегмент на две части: для **кучи** и для стека. Заранее было неизвестно, насколько большой может быть **куча** или стек, так что это решение было самым простым.



В [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)]¹⁰ можно прочитать:

¹wikipedia.org/wiki/Call_stack

²Also Known As - (Также известный как)

³Last In First Out (последним вошел, первым вышел)

⁴Store Multiple Empty Descending (инструкция ARM)

⁵Load Multiple Empty Descending (инструкция ARM)

⁶Store Multiple Full Ascending (инструкция ARM)

⁷Load Multiple Full Ascending (инструкция ARM)

⁸Store Multiple Empty Ascending (инструкция ARM)

⁹Load Multiple Empty Ascending (инструкция ARM)

¹⁰Также доступно здесь: <http://go.yurichev.com/17270>

6.2. ДЛЯ ЧЕГО ИСПОЛЬЗУЕТСЯ СТЕК?

The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

Это немного напоминает как некоторые студенты пишут два конспекта в одной тетрадке: первый конспект начинается обычным образом, второй пишется с конца, перевернув тетрадку. Конспекты могут встретиться где-то посередине, в случае недостатка свободного места.

6.2. Для чего используется стек?

6.2.1. Сохранение адреса возврата управления

x86

При вызове другой функции через `CALL` сначала в стек записывается адрес, указывающий на место после инструкции `CALL`, затем делается безусловный переход (почти как `JMP`) на адрес, указанный в операнде.

`CALL` — это аналог пары инструкций `PUSH address_after_call / JMP`.

`RET` вытаскивает из стека значение и передает управление по этому адресу — это аналог пары инструкций `POP tmp / JMP tmp`.

Крайне легко устроить переполнение стека, запустив бесконечную рекурсию:

```
void f()
{
    f();
};
```

MSVC 2008 предупреждает о проблеме:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, function will cause runtime
↳ stack overflow
```

...но, тем не менее, создает нужный код:

```
?f@@YAXXZ PROC                                ; f
; File c:\tmp6\ss.cpp
; Line 2
    push    ebp
    mov     ebp, esp
; Line 3
    call    ?f@@YAXXZ                          ; f
; Line 4
    pop     ebp
    ret     0
?f@@YAXXZ ENDP                                ; f
```

...причем, если включить оптимизацию (`/Ox`), то будет даже интереснее, без переполнения стека, но работать будет корректно¹¹:

```
?f@@YAXXZ PROC                                ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
```

¹¹здесь ирония

6.2. ДЛЯ ЧЕГО ИСПОЛЬЗУЕТСЯ СТЕК?

```
; Line 3
      jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                ; f
```

GCC 4.4.1 генерирует точно такой же код в обоих случаях, хотя и не предупреждает о проблеме.

ARM

Программы для ARM также используют стек для сохранения [RA](#), куда нужно вернуться, но несколько иначе. Как уже упоминалось в секции «Hello, world!» ([4.4](#) (стр. 14)), [RA](#) записывается в регистр [LR](#) (link register). Но если есть необходимость вызывать какую-то другую функцию и использовать регистр [LR](#) ещё раз, его значение желательно сохранить.

Обычно это происходит в прологе функции, часто мы видим там инструкцию вроде `PUSH {R4-R7, LR}`, а в эпилоге `POP {R4-R7, PC}` — так сохраняются регистры, которые будут использоваться в текущей функции, в том числе [LR](#).

Тем не менее, если некая функция не вызывает никаких более функций, в терминологии [RISC](#) она называется *leaf function*¹². Как следствие, «leaf»-функция не сохраняет регистр [LR](#) (потому что не изменяет его). А если эта функция небольшая, использует мало регистров, она может не использовать стек вообще. Таким образом, в ARM возможен вызов небольших leaf-функций не используя стек. Это может быть быстрее чем в старых x86, ведь внешняя память для стека не используется¹³. Либо это может быть полезным для тех ситуаций, когда память для стека ещё не выделена, либо недоступна,

Некоторые примеры таких функций: [9.3.2](#) (стр. 99), [9.3.3](#) (стр. 99), [20.17](#) (стр. 310), [20.33](#) (стр. 327), [20.5.4](#) (стр. 328), [16.4](#) (стр. 205), [16.2](#) (стр. 203), [18.3](#) (стр. 222).

6.2.2. Передача параметров функции

Самый распространенный способ передачи параметров в x86 называется «cdecl»:

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

Вызываемая функция получает свои параметры также через указатель стека.

Следовательно, так расположены значения в стеке перед исполнением самой первой инструкции функции `f()`:

ESP	адрес возврата
ESP+4	аргумент#1, маркируется в IDA как <code>arg_0</code>
ESP+8	аргумент#2, маркируется в IDA как <code>arg_4</code>
ESP+0xC	аргумент#3, маркируется в IDA как <code>arg_8</code>
...	...

См. также в соответствующем разделе о других способах передачи аргументов через стек ([65](#) (стр. 672)).

Важно отметить, что, в общем, никто не заставляет программистов передавать параметры именно через стек, это не является требованием к исполняемому коду. Вы можете делать это совершенно иначе, не используя стек вообще.

В каком-то смысле, популярный метод среди начинающих использовать язык ассемблера, это передавать аргументы в глобальных переменных, например:

Листинг 6.1: Код на ассемблере

```
...

mov     X, 123
mov     Y, 123
call    do_something

...
```

¹²infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

¹³Когда-то, очень давно, на PDP-11 и VAX на инструкцию CALL (вызов других функций) могло тратиться вплоть до 50% времени (возможно из-за работы с памятью), поэтому считалось, что много небольших функций это [анти-паттерн](#) [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

6.2. ДЛЯ ЧЕГО ИСПОЛЬЗУЕТСЯ СТЕК?

```
X      dd      ?
Y      dd      ?

do_something proc near
    ; take X
    ; take Y
    ; do something
    retn
do_something endp
```

Но у этого метода есть очевидный недостаток: ф-ция `do_something()` не сможет вызвать саму себя рекурсивно (либо, через какую-то стороннюю ф-цию), потому что тогда придется затереть свои собственные аргументы. Та же история с локальными переменными: если хранить их в глобальных переменных, ф-ция не сможет вызывать сама себя. К тому же, этот метод не безопасный для мультитредовой среды¹⁴. Способ хранения подобной информации в стеке заметно всё упрощает — он может хранить столько аргументов ф-ций и/или значений вообще, сколько в нем есть места.

В [Donald E. Knuth, *The Art of Computer Programming*, Volume 1, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997), 189] можно прочитать про еще более странные схемы передачи аргументов, которые были очень удобны на IBM System/360.

Так что, традиционно всё сложилось, что в x86 и ARM передача аргументов происходит именно через стек.

Кстати, вызываемая функция не имеет информации о количестве переданных ей аргументов. Функции Си с переменным количеством аргументов (как `printf()`) определяют их количество по спецификаторам строки формата (начинающиеся со знака %).

Если написать что-то вроде:

```
printf("%d %d %d", 1234);
```

`printf()` выведет 1234, затем ещё два случайных числа, которые волею случая оказались в стеке рядом.

Вот почему не так уж и важно, как объявлять функцию `main()`: как `main()`, `main(int argc, char *argv[])` либо `main(int argc, char *argv[], char *envp[])`.

В реальности, CRT-код вызывает `main()` примерно так:

```
push envp
push argv
push argc
call main
...
```

Если вы объявляете `main()` без аргументов, они, тем не менее, присутствуют в стеке, но не используются. Если вы объявите `main()` как `main(int argc, char *argv[])`, вы можете использовать два первых аргумента, а третий останется для вашей функции «невидимым». Более того, можно даже объявить `main(int argc)`, и это будет работать.

6.2.3. Хранение локальных переменных

Функция может выделить для себя некоторое место в стеке для локальных переменных, просто отодвинув [указатель стека](#) глубже к концу стека.

Это очень быстро вне зависимости от количества локальных переменных. Хранить локальные переменные в стеке не является необходимым требованием. Вы можете хранить локальные переменные где угодно. Но по традиции всё сложилось так.

6.2.4. x86: Функция `alloca()`

Интересен случай с функцией `alloca()`¹⁵. Эта функция работает как `malloc()`, но выделяет память прямо в стеке. Память освобождать через `free()` не нужно, так как эпилог функции (5 (стр. 25)) вернет `ESP` в изначальное состояние и выделенная память просто *выкидывается*. Интересна реализация функции `alloca()`. Эта функция, если упрощенно,

¹⁴При корректной реализации, каждый тред будет иметь свой собственный стек со своими аргументами/переменными.

¹⁵В MSVC, реализацию функции можно посмотреть в файлах `alloca16.asm` и `chkstk.asm` в `C:\Program Files (x86)\Microsoft Visual Studio 1`

6.2. ДЛЯ ЧЕГО ИСПОЛЬЗУЕТСЯ СТЕК?

просто сдвигает `ESP` вглубь стека на столько байт, сколько вам нужно и возвращает `ESP` в качестве указателя на выделенный блок.

Попробуем:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

Функция `_snprintf()` работает так же, как и `printf()`, только вместо выдачи результата в `stdout` (т.е. на терминал или в консоль), записывает его в буфер `buf`. Функция `puts()` выдает содержимое буфера `buf` в `stdout`. Конечно, можно было бы заменить оба этих вызова на один `printf()`, но здесь нужно проиллюстрировать использование небольшого буфера.

MSVC

Компилируем (MSVC 2010):

Листинг 6.2: MSVC 2010

```
...

mov     eax, 600           ; 00000258H
call    __alloca_probe_16
mov     esi, esp

push    3
push    2
push    1
push    OFFSET $SG2672
push    600               ; 00000258H
push    esi
call    __snprintf

push    esi
call    _puts
add     esp, 28           ; 0000001CH

...
```

Единственный параметр в `alloca()` передается через `EAX`, а не как обычно через стек ¹⁶.

GCC + Синтаксис Intel

А GCC 4.4.1 обходится без вызова других функций:

¹⁶Это потому, что `alloca()` — это не столько функция, сколько т.н. *compiler intrinsic* (92 (стр. 887)) Одна из причин, почему здесь нужна именно функция, а не несколько инструкций прямо в коде в том, что в реализации функции `alloca()` от MSVC¹⁷ есть также код, читающий из только что выделенной памяти, чтобы ОС подключила физическую память к этому региону VM¹⁸. После вызова `alloca()` `ESP` указывает на блок в 600 байт, который мы можем использовать под `buf`.

```
.LC0:
.string "hi! %d, %d, %d\n"
f:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 660
    lea     ebx, [esp+39]
    and     ebx, -16                ; выровнять указатель по 16-байтной границе
    mov     DWORD PTR [esp], ebx    ; s
    mov     DWORD PTR [esp+20], 3
    mov     DWORD PTR [esp+16], 2
    mov     DWORD PTR [esp+12], 1
    mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
    mov     DWORD PTR [esp+4], 600  ; maxlen
    call    _snprintf
    mov     DWORD PTR [esp], ebx    ; s
    call    puts
    mov     ebx, DWORD PTR [ebp-4]
    leave
    ret
```

GCC + Синтаксис AT&T

Посмотрим на тот же код, только в синтаксисе AT&T:

Листинг 6.4: GCC 4.7.3

```
.LC0:
.string "hi! %d, %d, %d\n"
f:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $660, %esp
    leal    39(%esp), %ebx
    andl    $-16, %ebx
    movl    %ebx, (%esp)
    movl    $3, 20(%esp)
    movl    $2, 16(%esp)
    movl    $1, 12(%esp)
    movl    $.LC0, 8(%esp)
    movl    $600, 4(%esp)
    call    _snprintf
    movl    %ebx, (%esp)
    call    puts
    movl    -4(%ebp), %ebx
    leave
    ret
```

Всё то же самое, что и в прошлом листинге.

Кстати, `movl $3, 20(%esp)` — это аналог `mov DWORD PTR [esp+20], 3` в синтаксисе Intel. Адресация памяти в виде *регистр+смещение* записывается в синтаксисе AT&T как `смещение(%регистр)`.

6.2.5. (Windows) SEH

В стеке хранятся записи [SEH¹⁹](#) для функции (если они присутствуют). Читайте больше о нем здесь: ([69.3](#) (стр. [703](#))).

6.2.6. Защита от переполнений буфера

Здесь больше об этом ([19.2](#) (стр. [271](#))).

¹⁹Structured Exception Handling

6.2.7. Автоматическое освобождение данных в стеке

Возможно, причина хранения локальных переменных и SEH-записей в стеке в том, что после выхода из функции, всё эти данные освобождаются автоматически, используя только одну инструкцию корректирования указателя стека (часто это `ADD`). Аргументы функций, можно сказать, тоже освобождаются автоматически в конце функции. А всё что хранится в куче (*heap*) нужно освобождать явно.

6.3. Разметка типичного стека

Разметка типичного стека в 32-битной среде перед исполнением самой первой инструкции функции выглядит так:

...	...
ESP-0xC	локальная переменная#2, маркируется в IDA как <code>var_8</code>
ESP-8	локальная переменная#1, маркируется в IDA как <code>var_4</code>
ESP-4	сохраненное значение EBP
ESP	Адрес возврата
ESP+4	аргумент#1, маркируется в IDA как <code>arg_0</code>
ESP+8	аргумент#2, маркируется в IDA как <code>arg_4</code>
ESP+0xC	аргумент#3, маркируется в IDA как <code>arg_8</code>
...	...

6.4. Мусор в стеке

Часто в этой книге говорится о «шуме» или «мусоре» в стеке или памяти. Откуда он берется? Это то, что осталось там после исполнения предыдущих функций.

Короткий пример:

```
#include <stdio.h>

void f1()
{
    int a=1, b=2, c=3;
};

void f2()
{
    int a, b, c;
    printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
    f1();
    f2();
};
```

Компилируем...

Листинг 6.5: Неоптимизирующий MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H

_c$ = -12      ; size = 4
_b$ = -8       ; size = 4
_a$ = -4       ; size = 4
_f1  PROC
    push      ebp
    mov       ebp, esp
    sub       esp, 12
    mov       DWORD PTR _a$[ebp], 1
    mov       DWORD PTR _b$[ebp], 2
    mov       DWORD PTR _c$[ebp], 3
    mov       esp, ebp
```

6.4. MYCOP B CTEKE

```
        pop     ebp
        ret     0
_f1     ENDP

_c$ = -12      ; size = 4
_b$ = -8       ; size = 4
_a$ = -4       ; size = 4
_f2     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        push    OFFSET $SG2752 ; '%d, %d, %d'
        call    DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2     ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        call    _f1
        call    _f2
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Компилятор поворчит немного...

```
c:\Polygon>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:st.exe
st.obj
```

Но когда мы запускаем...

```
c:\Polygon>st
1, 2, 3
```

Ох. Вот это странно. Мы ведь не устанавливали значения никаких переменных в `f2()`. Эти значения — это «привидения», которые всё ещё в стеке.

6.4. МУСОР В СТЕКЕ

Загрузим пример в OllyDbg:

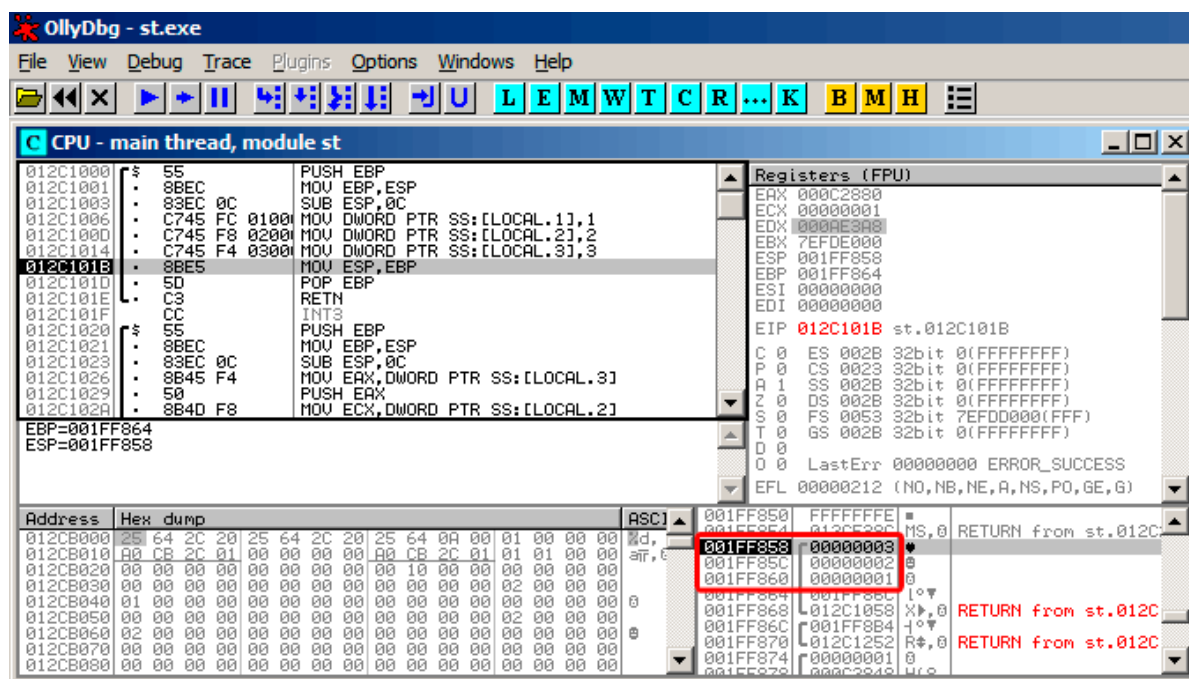


Рис. 6.1: OllyDbg: f1()

Когда f1() заполняет переменные a, b и c они сохраняются по адресу 0x1FF860 и т.д.

6.4. МУСОР В СТЕКЕ

А когда выполняется `f2()`:

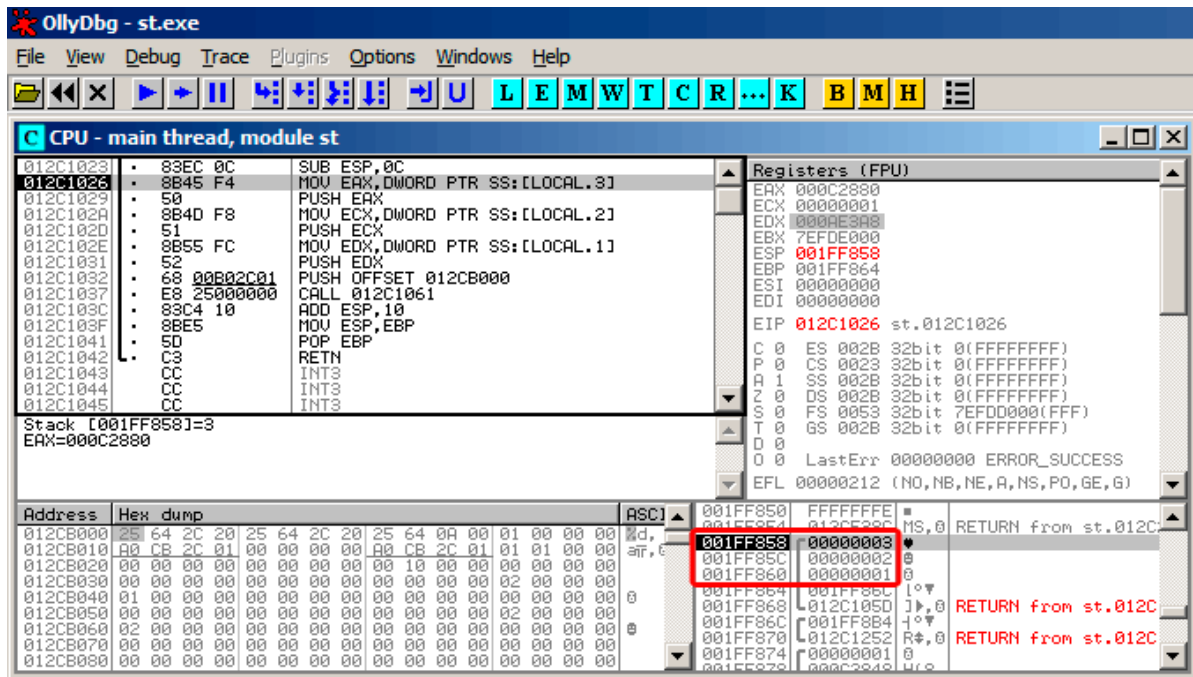


Рис. 6.2: OllyDbg: `f2()`

... `a`, `b` и `c` в функции `f2()` находятся по тем же адресам! Пока никто не перезаписал их, так что они здесь в нетронут-ном виде. Для создания такой странной ситуации несколько функций должны выполняться друг за другом и `SP` должен быть одинаковым при входе в функции, т.е. у функций должно быть равное количество аргументов). Тогда локальные переменные будут расположены в том же месте стека. Подводя итоги, все значения в стеке (да и памяти вообще) это значения оставшиеся от исполнения предыдущих функций. Строго говоря, они не случайны, они скорее непредсказуемы. А как иначе? Можно было бы очищать части стека перед исполнением каждой функции, но это слишком много лишней (и ненужной) работы.

6.4.1. MSVC 2013

Этот пример был скомпилирован в MSVC 2010. Но один читатель этой книги сделал попытку скомпилировать пример в MSVC 2013, запустил и увидел 3 числа в обратном порядке:

```
c:\Polygon\c>st
3, 2, 1
```

Почему? Я также попробовал скомпилировать этот пример в MSVC 2013 и увидел это:

Листинг 6.6: MSVC 2013

```
_a$ = -12 ; size = 4
_b$ = -8 ; size = 4
_c$ = -4 ; size = 4
_f2 PROC
...
_f2 ENDP

_c$ = -12 ; size = 4
_b$ = -8 ; size = 4
_a$ = -4 ; size = 4
_f1 PROC
...
_f1 ENDP
```

6.5. УПРАЖНЕНИЯ

В отличие от MSVC 2010, MSVC 2013 разместил переменные a/b/c в функции `f2()` в обратном порядке. И это полностью корректно, потому что в стандартах Си/Си++ нет правила, в каком порядке локальные переменные должны быть размещены в локальном стеке, если вообще. Разница есть из-за того что MSVC 2010 делает это одним способом, а в MSVC 2013, вероятно, что-то немного изменили во внутренностях компилятора, так что он ведет себя слегка иначе.

6.5. Упражнения

- <http://challenges.re/51>
- <http://challenges.re/52>

Глава 7

printf() с несколькими аргументами

Попробуем теперь немного расширить пример *Hello, world!* (4 (стр. 8)), написав в теле функции `main()` :

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
    return 0;
};
```

7.1. x86

7.1.1. x86: 3 аргумента

MSVC

Компилируем при помощи MSVC 2010 Express, и в итоге получим:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H
...
        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                ; 00000010H
```

Всё почти то же, за исключением того, что теперь видно, что аргументы для `printf()` заталкиваются в стек в обратном порядке: самый первый аргумент заталкивается последним.

Кстати, вспомним, что переменные типа *int* в 32-битной системе, как известно, имеет ширину 32 бита, это 4 байта.

Итак, у нас всего 4 аргумента. $4 * 4 = 16$ — именно 16 байт занимают в стеке указатель на строку плюс ещё 3 числа типа *int*.

Когда при помощи инструкции `ADD ESP, X` корректируется [указатель стека](#) `ESP` после вызова какой-либо функции, зачастую можно сделать вывод о том, сколько аргументов у вызываемой функции было, разделив X на 4.

Конечно, это относится только к `cdecl`-методу передачи аргументов через стек, и только для 32-битной среды.

См. также в соответствующем разделе о способах передачи аргументов через стек (65 (стр. 672)).

Иногда бывает так, что подряд идут несколько вызовов разных функций, но стек корректируется только один раз, после последнего вызова:

```
push a1
push a2
call ...
```

7.1. X86

```
...  
push a1  
call ...  
...  
push a1  
push a2  
push a3  
call ...  
add esp, 24
```

Вот пример из реальной жизни:

Листинг 7.1: x86

.text:100113E7	push	3	
.text:100113E9	call	sub_100018B0	; берет один аргумент (3)
.text:100113EE	call	sub_100019D0	; не имеет аргументов вообще
.text:100113F3	call	sub_10006A90	; не имеет аргументов вообще
.text:100113F8	push	1	
.text:100113FA	call	sub_100018B0	; берет один аргумент (1)
.text:100113FF	add	esp, 8	; выбрасывает из стека два аргумента

MSVC и OllyDbg

Попробуем этот же пример в OllyDbg. Это один из наиболее популярных win32-отладчиков пользовательского режима. Мы можем компилировать наш пример в MSVC 2012 с опцией `/MD` что означает линковать с библиотекой `MSVCR*.DLL`, чтобы импортируемые функции были хорошо видны в отладчике.

Затем загружаем исполняемый файл в OllyDbg. Самая первая точка останова в `ntdll.dll`, нажмите F9 (запустить). Вторая точка останова в `CRT`-коде. Теперь мы должны найти функцию `main()`.

Найдите этот код, прокрутив окно кода до самого верха (MSVC располагает функцию `main()` в самом начале секции кода):

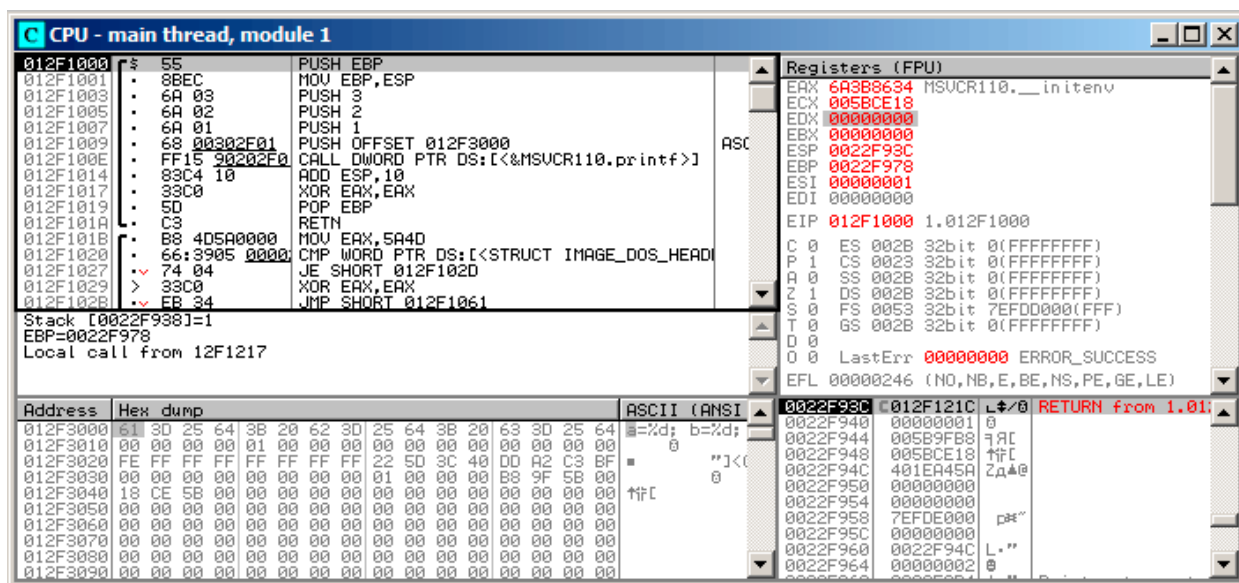


Рис. 7.1: OllyDbg: самое начало функции `main()`

Кликните на инструкции `PUSH EBP`, нажмите F2 (установка точки останова) и нажмите F9 (запустить). Нам нужно про-извести все эти манипуляции, чтобы пропустить `CRT`-код, потому что нам он пока не интересен.

Нажмите F8 (сделать шаг, не входя в функцию) 6 раз, т.е. пропустить 6 инструкций:

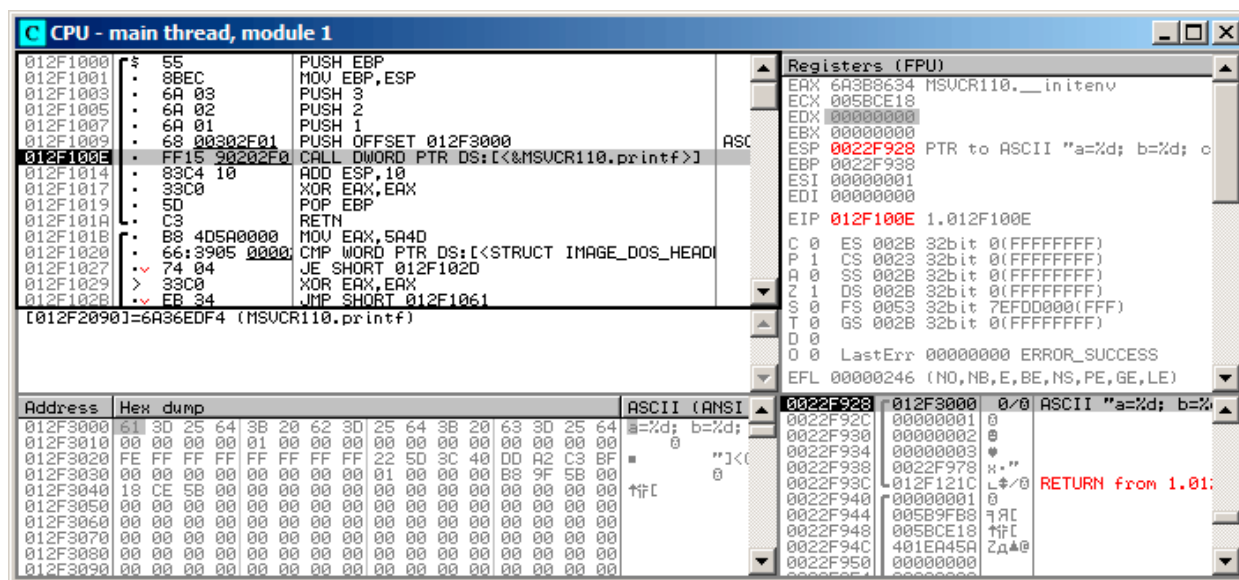


Рис. 7.2: OllyDbg: перед исполнением `printf()`

Теперь **PC** указывает на инструкцию `CALL printf`. OllyDbg, как и другие отладчики, подсвечивает регистры со значениями, которые изменились. Поэтому каждый раз когда мы нажимаем F8, **EIP** изменяется и его значение подсвечивается красным. **ESP** также меняется, потому что значения заталкиваются в стек.

Где находятся эти значения в стеке? Посмотрите на правое нижнее окно в отладчике:

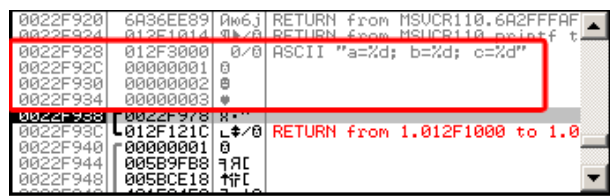


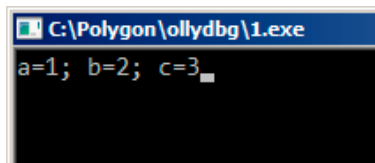
Рис. 7.3: OllyDbg: стек с сохраненными значениями (красная рамка добавлена в графическом редакторе)

Здесь видно 3 столбца: адрес в стеке, значение в стеке и ещё дополнительный комментарий от OllyDbg. OllyDbg понимает `printf()`-строки, так что он показывает здесь и строку и 3 значения *привязанных* к ней.

Можно кликнуть правой кнопкой мыши на строке формата, кликнуть на «Follow in dump» и строка формата появится в окне слева внизу, где всегда виден какой-либо участок памяти. Эти значения в памяти можно редактировать. Можно изменить саму строку формата, и тогда результат работы нашего примера будет другой. В данном случае пользы от этого немного, но для упражнения это полезно, чтобы начать чувствовать как тут всё работает.

Нажмите F8 (сделать шаг, не входя в функцию).

В консоли мы видим вывод:



```
C:\Polygon\ollydbg\1.exe
a=1; b=2; c=3
```

Рис. 7.4: Функция `printf()` исполнилась

Посмотрим как изменились регистры и состояние стека:

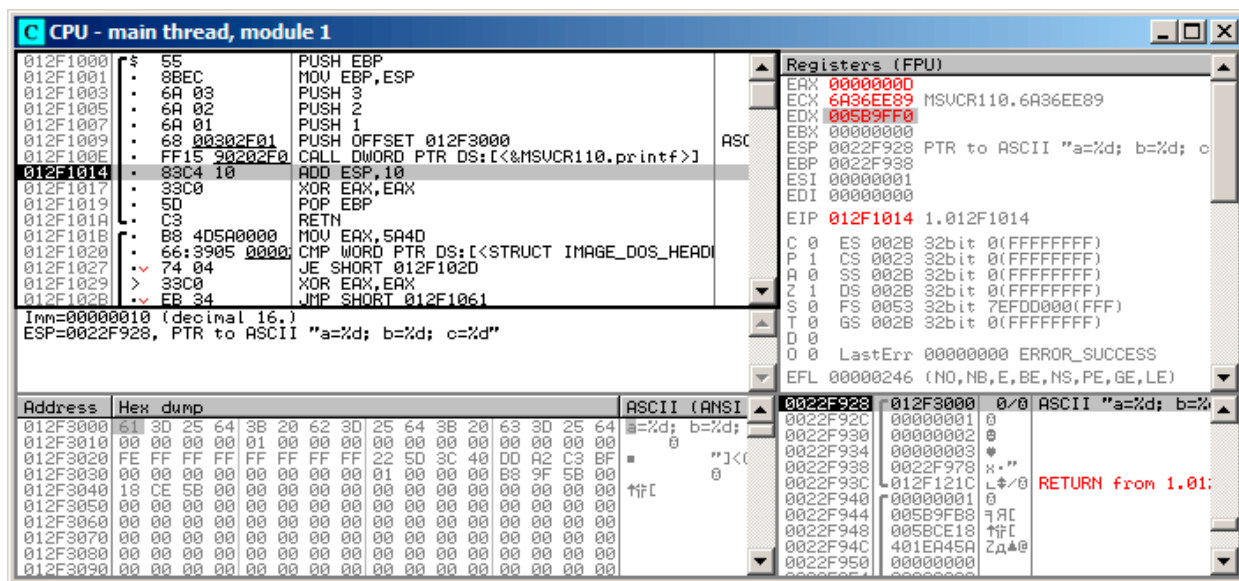


Рис. 7.5: OllyDbg после исполнения `printf()`

Регистр `EAX` теперь содержит `0xD` (13). Всё верно: `printf()` возвращает количество выведенных символов. Значение `EIP` изменилось. Действительно, теперь здесь адрес инструкции после `CALL printf`. Значения регистров `ECX` и `EDX` также изменились. Очевидно, внутренности функции `printf()` используют их для каких-то своих нужд.

Очень важно то, что значение `ESP` не изменилось. И аргументы-значения в стеке также! Мы ясно видим здесь и строку формата и соответствующие ей 3 значения, они всё ещё здесь. Действительно, по соглашению вызовов *cdecl*, вызываемая функция не возвращает `ESP` назад. Это должна делать вызывающая функция (caller).

Нажмите F8 снова, чтобы исполнилась инструкция `ADD ESP, 10`:

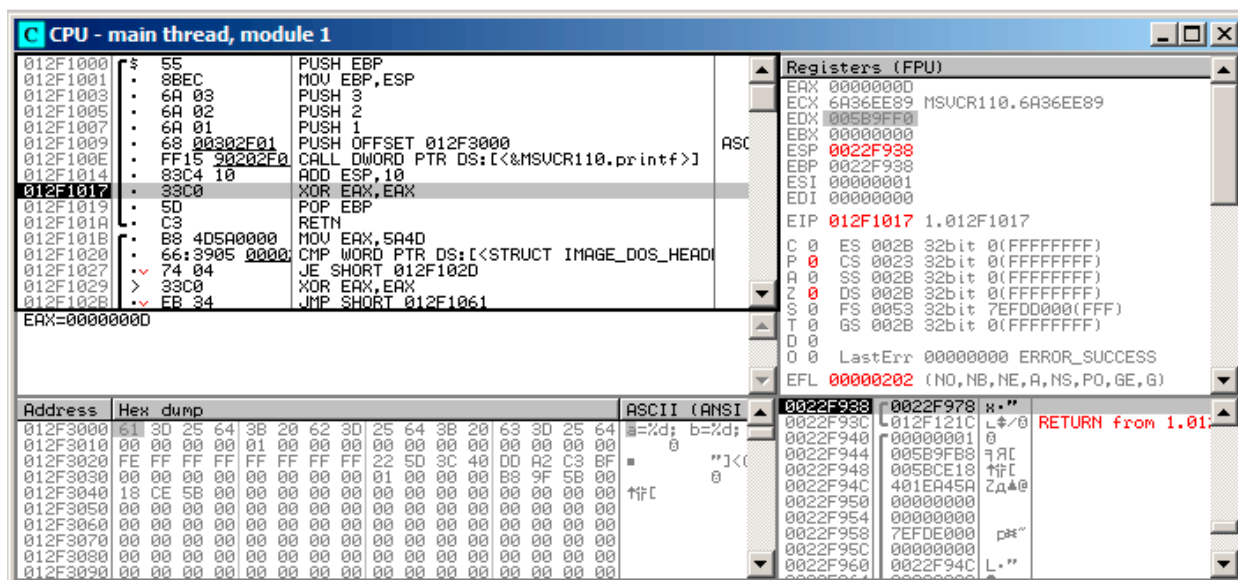


Рис. 7.6: OllyDbg: после исполнения инструкции `ADD ESP, 10`

`ESP` изменился, но значения всё ещё в стеке! Конечно, никому не нужно заполнять эти значения нулями или что-то в этом роде. Всё что выше указателя стека (`SP`) это *шум* или *мусор* и не имеет особой ценности. Было бы очень затратно по времени очищать ненужные элементы стека, к тому же, никому это и не нужно.

GCC

Скомпилируем то же самое в Linux при помощи GCC 4.4.1 и посмотрим на результат в `IDA`:

```
main      proc near
var_10    = dword ptr -10h
var_C     = dword ptr -0Ch
var_8     = dword ptr -8
var_4     = dword ptr -4

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     eax, offset aADBD CD ; "a=%d; b=%d; c=%d"
        mov     [esp+10h+var_4], 3
        mov     [esp+10h+var_8], 2
        mov     [esp+10h+var_C], 1
        mov     [esp+10h+var_10], eax
        call    _printf
        mov     eax, 0
        leave
        retn
main      endp
```

Можно сказать что этот короткий код, созданный GCC, отличается от кода MSVC только способом помещения значений в стек. Здесь GCC снова работает со стеком напрямую без `PUSH` / `POP`.

GCC и GDB

Попробуем также этот пример и в `GDB`¹ в Linux.

`-g` означает генерировать отладочную информацию в выходном исполняемом файле.

¹GNU debugger


```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/1...done.
```

Листинг 7.2: установим точку останова на `printf()`

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Запускаем. У нас нет исходного кода функции, поэтому `GDB` не может его показать.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

Выдать 10 элементов стека. Левый столбец — это адрес в стеке.

```
(gdb) x/10w $esp
0xbffff11c:    0x0804844a    0x080484f0    0x00000001    0x00000002
0xbffff12c:    0x00000003    0x08048460    0x00000000    0x00000000
0xbffff13c:    0xb7e29905    0x00000001
```

Самый первый элемент это `RA` (`0x0804844a`). Мы можем удостовериться в этом, дизассемблируя память по этому адресу:

```
(gdb) x/5i 0x0804844a
0x0804844a <main+45>: mov     $0x0,%eax
0x0804844f <main+50>: leave
0x08048450 <main+51>: ret
0x08048451:    xchg    %ax,%ax
0x08048453:    xchg    %ax,%ax
```

Две инструкции `XCHG` это холостые инструкции, аналогичные `NOP`.

Второй элемент (`0x080484f0`) это адрес строки формата:

```
(gdb) x/s 0x080484f0
0x080484f0:    "a=%d; b=%d; c=%d"
```

Остальные 3 элемента (1, 2, 3) это аргументы функции `printf()`. Остальные элементы это может быть и мусор в стеке, но могут быть и значения от других функций, их локальные переменные, и т.д. Пока что мы можем игнорировать их.

Исполняем «finish». Это значит исполнять все инструкции до самого конца функции. В данном случае это означает исполнять до завершения `printf()`.

```
(gdb) finish
Run till exit from #0 __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
main () at 1.c:6
6      return 0;
Value returned is $2 = 13
```

`GDB` показывает, что вернула `printf()` в `EAX` (13). Это, так же как и в примере с `OllyDbg`, количество напечатанных символов.

А ещё мы видим «return 0;» и что это выражение находится в файле `1.c` в строке 6. Действительно, файл `1.c` лежит в текущем директории и `GDB` находит там эту строку. Как `GDB` знает, какая строка Си-кода сейчас исполняется? Компилятор,

7.1. X86

генерируя отладочную информацию, также сохраняет информацию о соответствии строк в исходном коде и адресов инструкций. GDB это всё-таки отладчик уровня исходных текстов.

Посмотрим регистры. 13 в **EAX** :

```
(gdb) info registers
eax          0xd      13
ecx          0x0      0
edx          0x0      0
ebx          0xb7fc0000 -1208221696
esp          0xbffff120 0xbffff120
ebp          0xbffff138 0xbffff138
esi          0x0      0
edi          0x0      0
eip          0x804844a   0x804844a <main+45>
...
```

Попробуем дизассемблировать текущие инструкции. Стрелка указывает на инструкцию, которая будет исполнена следующей.

```
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:   push    %ebp
   0x0804841e <+1>:   mov     %esp,%ebp
   0x08048420 <+3>:   and     $0xffffffff,%esp
   0x08048423 <+6>:   sub     $0x10,%esp
   0x08048426 <+9>:   movl    $0x3,0xc(%esp)
   0x0804842e <+17>:  movl    $0x2,0x8(%esp)
   0x08048436 <+25>:  movl    $0x1,0x4(%esp)
   0x0804843e <+33>:  movl    $0x80484f0,(%esp)
   0x08048445 <+40>:  call    0x80482f0 <printf@plt>
=> 0x0804844a <+45>:  mov     $0x0,%eax
   0x0804844f <+50>:  leave   0(%eax)
   0x08048450 <+51>:  ret
End of assembler dump.
```

По умолчанию **GDB** показывает дизассемблированный листинг в формате AT&T. Но можно также переключиться в формат Intel:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:   push    ebp
   0x0804841e <+1>:   mov     ebp,esp
   0x08048420 <+3>:   and     esp,0xffffffff
   0x08048423 <+6>:   sub     esp,0x10
   0x08048426 <+9>:   mov     DWORD PTR [esp+0xc],0x3
   0x0804842e <+17>:  mov     DWORD PTR [esp+0x8],0x2
   0x08048436 <+25>:  mov     DWORD PTR [esp+0x4],0x1
   0x0804843e <+33>:  mov     DWORD PTR [esp],0x80484f0
   0x08048445 <+40>:  call    0x80482f0 <printf@plt>
=> 0x0804844a <+45>:  mov     eax,0x0
   0x0804844f <+50>:  leave   0(%eax)
   0x08048450 <+51>:  ret
End of assembler dump.
```

Исполняем следующую инструкцию. **GDB** покажет закрывающуюся скобку, означая, что это конец блока в функции.

```
(gdb) step
7      };
```

Посмотрим регистры после исполнения инструкции **MOV EAX, 0**. **EAX** здесь уже действительно ноль.

```
(gdb) info registers
eax          0x0      0
ecx          0x0      0
edx          0x0      0
ebx          0xb7fc0000 -1208221696
esp          0xbffff120 0xbffff120
ebp          0xbffff138 0xbffff138
```

7.1. X86

esi	0x0	0
edi	0x0	0
eip	0x804844f	0x804844f <main+50>
...		

7.1.2. x64: 8 аргументов

Для того чтобы посмотреть, как остальные аргументы будут передаваться через стек, изменим пример ещё раз, увеличив количество передаваемых аргументов до 9 (строка формата `printf()` и 8 переменных типа `int`):

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
};
```

MSVC

Как уже было сказано ранее, первые 4 аргумента в Win64 передаются в регистрах `RCX`, `RDX`, `R8`, `R9`, а остальные — через стек. Здесь мы это и видим. Впрочем, инструкция `PUSH` не используется, вместо неё при помощи `MOV` значения сразу записываются в стек.

Листинг 7.3: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main      PROC
          sub     rsp, 88

          mov     DWORD PTR [rsp+64], 8
          mov     DWORD PTR [rsp+56], 7
          mov     DWORD PTR [rsp+48], 6
          mov     DWORD PTR [rsp+40], 5
          mov     DWORD PTR [rsp+32], 4
          mov     r9d, 3
          mov     r8d, 2
          mov     edx, 1
          lea     rcx, OFFSET FLAT:$SG2923
          call    printf

          ; возврат 0
          xor     eax, eax

          add     rsp, 88
          ret     0
main      ENDP
_TEXT     ENDS
END
```

Наблюдательный читатель может спросить, почему для значений типа `int` отводится 8 байт, ведь нужно только 4? Да, это нужно запомнить: для значений всех типов более коротких чем 64-бита, отводится 8 байт. Это сделано для удобства: так всегда легко рассчитать адрес того или иного аргумента. К тому же, все они расположены по выровненным адресам в памяти. В 32-битных средах точно также: для всех типов резервируется 4 байта в стеке.

GCC

В *NIX-системах для x86-64 ситуация похожая, вот только первые 6 аргументов передаются через `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`. Остальные — через стек. GCC генерирует код, записывающий указатель на строку в `EDI` вместо `RDI` — это мы уже рассмотрели чуть раньше: [4.2.2](#) (стр. 13).

Почему перед вызовом `printf()` очищается регистр `EAX` мы уже рассмотрели ранее [4.2.2](#) (стр. 13).

```
.LC0:
    .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
    sub     rsp, 40

    mov     r9d, 5
    mov     r8d, 4
    mov     ecx, 3
    mov     edx, 2
    mov     esi, 1
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax ; количество переданных векторных регистров
    mov     DWORD PTR [rsp+16], 8
    mov     DWORD PTR [rsp+8], 7
    mov     DWORD PTR [rsp], 6
    call    printf

    ; возврат 0

    xor     eax, eax
    add     rsp, 40
    ret
```

GCC + GDB

Попробуем этот пример в [GDB](#).

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/2...done.
```

Листинг 7.5: ставим точку останова на `printf()`, запускаем

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n") at printf.c:29
↳ c:29
29 printf.c: No such file or directory.
```

В регистрах `RSI` / `RDX` / `RCX` / `R8` / `R9` всё предсказуемо. А `RIP` содержит адрес самой первой инструкции функции `printf()`.

```
(gdb) info registers
rax            0x0            0
rbx            0x0            0
rcx            0x3            3
rdx            0x2            2
rsi            0x1            1
rdi            0x400628 4195880
rbp            0x7fffffffdf60 0x7fffffffdf60
rsp            0x7fffffffdf38 0x7fffffffdf38
r8             0x4            4
```

7.1. X86

r9	0x5	5	
r10	0x7fffffffdfce0	140737488346336	
r11	0x7ffff7a65f60	140737348263776	
r12	0x400440	4195392	
r13	0x7fffffffef040	140737488347200	
r14	0x0	0	
r15	0x0	0	
rip	0x7ffff7a65f60	0x7ffff7a65f60	<__printf>
...			

Листинг 7.6: смотрим на строку формата

```
(gdb) x/s $rdi
0x400628:      "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
```

Дампим стек на этот раз с командой `x/g` — `g` означает *giant words*, т.е. 64-битные слова.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576      0x0000000000000006
0x7fffffffdf48: 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000      0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78: 0x00007ffffffe048      0x0000000100000000
```

Самый первый элемент стека, как и в прошлый раз, это **RA**. Через стек также передаются 3 значения: 6, 7, 8. Видно, что 8 передается с неочищенной старшей 32-битной частью: `0x00007fff00000008`. Это нормально, ведь передаются числа типа *int*, а они 32-битные. Так что в старшей части регистра или памяти стека остался «случайный мусор».

GDB показывает всю функцию `main()`, если попытаться посмотреть, куда вернется управление после исполнения `printf()`.

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main:
   0x000000000040052d <+0>:      push    rbp
   0x000000000040052e <+1>:      mov     rbp, rsp
   0x0000000000400531 <+4>:      sub     rsp, 0x20
   0x0000000000400535 <+8>:      mov     DWORD PTR [rsp+0x10], 0x8
   0x000000000040053d <+16>:     mov     DWORD PTR [rsp+0x8], 0x7
   0x0000000000400545 <+24>:     mov     DWORD PTR [rsp], 0x6
   0x000000000040054c <+31>:     mov     r9d, 0x5
   0x0000000000400552 <+37>:     mov     r8d, 0x4
   0x0000000000400558 <+43>:     mov     ecx, 0x3
   0x000000000040055d <+48>:     mov     edx, 0x2
   0x0000000000400562 <+53>:     mov     esi, 0x1
   0x0000000000400567 <+58>:     mov     edi, 0x400628
   0x000000000040056c <+63>:     mov     eax, 0x0
   0x0000000000400571 <+68>:     call   0x400410 <printf@plt>
   0x0000000000400576 <+73>:     mov     eax, 0x0
   0x000000000040057b <+78>:     leave
   0x000000000040057c <+79>:     ret
End of assembler dump.
```

Заканчиваем исполнение `printf()`, исполняем инструкцию обнуляющую **EAX**, удостоверимся что в регистре **EAX** именно ноль. **RIP** указывает сейчас на инструкцию **LEAVE**, т.е. предпоследнюю в функции `main()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n")
↳ at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6      return 0;
Value returned is $1 = 39
(gdb) next
7      };
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x26     38
```

7.2. ARM

rdx	0x7ffff7dd59f0	140737351866864
rsi	0x7fffffd9	2147483609
rdi	0x0	0
rbp	0x7ffffffffffdf60	0x7ffffffffffdf60
rsp	0x7ffffffffffdf40	0x7ffffffffffdf40
r8	0x7ffff7dd26a0	140737351853728
r9	0x7ffff7a60134	140737348239668
r10	0x7ffffffffffd5b0	140737488344496
r11	0x7ffff7a95900	140737348458752
r12	0x400440	4195392
r13	0x7ffffffffffe040	140737488347200
r14	0x0	0
r15	0x0	0
rip	0x40057b	0x40057b <main+78>
...		

7.2. ARM

7.2.1. ARM: 3 аргумента

В ARM традиционно принята такая схема передачи аргументов в функцию: 4 первых аргумента через регистры **R0** - **R3**; а остальные — через стек. Это немного похоже на то, как аргументы передаются в `fastcall` (65.3 (стр. 673)) или `win64` (65.5.1 (стр. 675)).

32-битный ARM

Неоптимизирующий Keil 6/2013 (Режим ARM)

Листинг 7.7: Неоптимизирующий Keil 6/2013 (Режим ARM)

```
.text:00000000 main
.text:00000000 10 40 2D E9   STMFD    SP!, {R4,LR}
.text:00000004 03 30 A0 E3   MOV     R3, #3
.text:00000008 02 20 A0 E3   MOV     R2, #2
.text:0000000C 01 10 A0 E3   MOV     R1, #1
.text:00000010 08 00 8F E2   ADR     R0, aADBDCD      ; "a=%d; b=%d; c=%d"
.text:00000014 06 00 00 EB   BL     __2printf
.text:00000018 00 00 A0 E3   MOV     R0, #0           ; return 0
.text:0000001C 10 80 BD E8   LDMFD   SP!, {R4,PC}
```

Итак, первые 4 аргумента передаются через регистры **R0** - **R3**, по порядку: указатель на формат-строку для `printf()` в **R0**, затем 1 в **R1**, 2 в **R2** и 3 в **R3**.

Инструкция на **0x18** записывает 0 в **R0** — это выражение в Си `return 0`. Пока что здесь нет ничего необычного. Оптимизирующий Keil 6/2013 генерирует точно такой же код.

Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 7.8: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:00000000 main
.text:00000000 10 B5          PUSH    {R4,LR}
.text:00000002 03 23          MOVS    R3, #3
.text:00000004 02 22          MOVS    R2, #2
.text:00000006 01 21          MOVS    R1, #1
.text:00000008 02 A0          ADR     R0, aADBDCD      ; "a=%d; b=%d; c=%d"
.text:0000000A 00 F0 0D F8   BL     __2printf
.text:0000000E 00 20          MOVS    R0, #0
.text:00000010 10 BD          POP     {R4,PC}
```

Здесь нет особых отличий от неоптимизированного варианта для режима ARM.

Немного переделаем пример, убрав *return 0*:

```
#include <stdio.h>

void main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
};
```

Результат получится необычным:

Листинг 7.9: Оптимизирующий Keil 6/2013 (Режим ARM)

```
.text:00000014 main
.text:00000014 03 30 A0 E3    MOV     R3, #3
.text:00000018 02 20 A0 E3    MOV     R2, #2
.text:0000001C 01 10 A0 E3    MOV     R1, #1
.text:00000020 1E 0E 8F E2    ADR     R0, aADBDCD      ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA    B       __2printf
```

Это оптимизированная версия (-O3) для режима ARM, и здесь мы видим последнюю инструкцию **B** вместо привычной нам **BL**. Отличия между этой оптимизированной версией и предыдущей, скомпилированной без оптимизации, ещё и в том, что здесь нет пролога и эпилога функции (инструкций, сохраняющих состояние регистров **R0** и **LR**). Инstrukция **B** просто переходит на другой адрес, без манипуляций с регистром **LR**, то есть это аналог **JMP** в x86. Почему это работает нормально? Потому что этот код эквивалентен предыдущему.

Основных причин две: 1) стек не модифицируется, как и **указатель стека SP**; 2) вызов функции **printf()** последний, после него ничего не происходит. Функция **printf()**, отработав, просто возвращает управление по адресу, записанному в **LR**.

Но в **LR** находится адрес места, откуда была вызвана наша функция! А следовательно, управление из **printf()** вернется сразу туда.

Значит нет нужды сохранять **LR**, потому что нет нужды модифицировать **LR**. А нет нужды модифицировать **LR**, потому что нет иных вызовов функций, кроме **printf()**, к тому же, после этого вызова не нужно ничего здесь больше делать! Поэтому такая оптимизация возможна.

Эта оптимизация часто используется в функциях, где последнее выражение — это вызов другой функции.

Ещё один похожий пример описан здесь: [14.1.1](#) (стр. 151).

ARM64

Неоптимизирующий GCC (Linaro) 4.9

Листинг 7.10: Неоптимизирующий GCC (Linaro) 4.9

```
.LC1:
.string "a=%d; b=%d; c=%d"
f2:
; сохранить FP и LR в стековом фрейме:
stp     x29, x30, [sp, -16]!
; установить стековый фрейм (FP=SP):
add     x29, sp, 0
adrp    x0, .LC1
add     x0, x0, :lo12:.LC1
mov     w1, 1
mov     w2, 2
mov     w3, 3
bl      printf
mov     w0, 0
; восстановить FP и LR
ldp     x29, x30, [sp], 16
ret
```

7.2. ARM

Итак, первая инструкция `STP` (*Store Pair*) сохраняет `FP` (X29) и `LR` (X30) в стеке. Вторая инструкция `ADD X29, SP, 0` формирует стековый фрейм. Это просто запись значения `SP` в X29.

Далее уже знакомая пара инструкций `ADRP / ADD` формирует указатель на строку.

`lo12` означает младшие 12 бит, т.е., линкер запишет младшие 12 бит адреса метки `LC1` в опкод инструкции `ADD . %d` в формате `printf()` это 32-битный `int`, так что 1, 2 и 3 заносятся в 32-битные части регистров. Оптимизирующий GCC (Linaro) 4.9 генерирует почти такой же код.

7.2.2. ARM: 8 аргументов

Снова воспользуемся примером с 9-ю аргументами из предыдущей секции: 7.1.2 (стр. 45).

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
};
```

Оптимизирующий Keil 6/2013: Режим ARM

```
.text:00000028      main
.text:00000028
.text:00000028      var_18 = -0x18
.text:00000028      var_14 = -0x14
.text:00000028      var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5 STR    LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2 SUB     SP, SP, #0x14
.text:00000030 08 30 A0 E3 MOV     R3, #8
.text:00000034 07 20 A0 E3 MOV     R2, #7
.text:00000038 06 10 A0 E3 MOV     R1, #6
.text:0000003C 05 00 A0 E3 MOV     R0, #5
.text:00000040 04 C0 8D E2 ADD     R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8 STMIA   R12, {R0-R3}
.text:00000048 04 00 A0 E3 MOV     R0, #4
.text:0000004C 00 00 8D E5 STR     R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3 MOV     R3, #3
.text:00000054 02 20 A0 E3 MOV     R2, #2
.text:00000058 01 10 A0 E3 MOV     R1, #1
.text:0000005C 6E 0F 8F E2 ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
.text:00000060 BC 18 00 EB BL      __2printf
.text:00000064 14 D0 8D E2 ADD     SP, SP, #0x14
.text:00000068 04 F0 9D E4 LDR     PC, [SP+4+var_4],#4
```

Этот код можно условно разделить на несколько частей:

- Пролог функции:

Самая первая инструкция `STR LR, [SP,#var_4]!` сохраняет в стеке `LR`, ведь нам придется использовать этот регистр для вызова `printf()`. Восклицательный знак в конце означает *pre-index*. Это значит, что в начале `SP` должно быть уменьшено на 4, затем по адресу в `SP` должно быть записано значение `LR`.

Это аналог знакомой в x86 инструкции `PUSH`. Читайте больше об этом: 29.2 (стр. 439).

Вторая инструкция `SUB SP, SP, #0x14` уменьшает *указатель стека* `SP`, но, на самом деле, эта процедура нужна для выделения в локальном стеке места размером `0x14` (20) байт. Действительно, нам нужно передать 5 32-битных значений через стек в `printf()`. Каждое значение занимает 4 байта, все вместе — $5 * 4 = 20$. Остальные 4 32-битных значения будут переданы через регистры.

- Передача 5, 6, 7 и 8 через стек: они записываются в регистры `R0`, `R1`, `R2` и `R3` соответственно. Затем инструкция `ADD R12, SP, #0x18+var_14` записывает в регистр `R12` адрес места в стеке, куда будут помещены эти 4 значения. `var_14` — это макрос ассемблера, равный `-0x14`. Такие макросы создает `IDA`, чтобы удобнее было показывать, как код обращается к стеку.

Макросы `var_?`, создаваемые `IDA`, отражают локальные переменные в стеке. Так что в `R12` будет записано `SP+4`.

Следующая инструкция `STMIA R12, R0-R3` записывает содержимое регистров `R0 - R3` по адресу в памяти, на который указывает `R12`.

Инструкция `STMIA` означает *Store Multiple Increment After*. «Increment After» означает, что `R12` будет увеличиваться на 4 после записи каждого значения регистра.

- Передача 4 через стек: 4 записывается в `R0`, затем инструкция `STR R0, [SP, #0x18+var_18]` записывает его в стек. `var_18` равен `-0x18`, смещение будет 0, так что значение из регистра `R0` (4) запишется туда, куда указывает `SP`.
- Передача 1, 2 и 3 через регистры:

Значения для первых трех чисел (a, b, c) (1, 2, 3 соответственно) передаются в регистрах `R1`, `R2` и `R3` перед самым вызовом `printf()`, а остальные 5 значений передаются через стек, и вот как:

- Вызов `printf()`.

- Эпилог функции:

Инструкция `ADD SP, SP, #0x14` возвращает `SP` на прежнее место, аннулируя таким образом всё, что было записано в стек. Конечно, то что было записано в стек, там пока и останется, но всё это будет многократно перезаписано во время исполнения последующих функций.

Инструкция `LDR PC, [SP+4+var_4], #4` загружает в `PC` сохраненное значение `LR` из стека, обеспечивая таким образом выход из функции.

Здесь нет восклицательного знака — действительно, сначала `PC` загружается из места, куда указывает `SP` ($4 + var\_4 = 4 + (-4) = 0$), так что эта инструкция аналогична `LDR PC, [SP], #4`, затем `SP` увеличивается на 4. Это называется *post-index*². Почему `IDA` показывает инструкцию именно так? Потому что она хочет показать разметку стека и тот факт, что `var_4` выделена в локальном стеке именно для сохраненного значения `LR`. Эта инструкция в каком-то смысле аналогична `POP PC` в x86³.

Оптимизирующий Keil 6/2013: Режим Thumb

```
.text:0000001C      printf_main2
.text:0000001C
.text:0000001C      var_18 = -0x18
.text:0000001C      var_14 = -0x14
.text:0000001C      var_8  = -8
.text:0000001C
.text:0000001C 00 B5      PUSH    {LR}
.text:0000001E 08 23      MOVS     R3, #8
.text:00000020 85 B0      SUB      SP, SP, #0x14
.text:00000022 04 93      STR      R3, [SP, #0x18+var_8]
.text:00000024 07 22      MOVS     R2, #7
.text:00000026 06 21      MOVS     R1, #6
.text:00000028 05 20      MOVS     R0, #5
.text:0000002A 01 AB      ADD      R3, SP, #0x18+var_14
.text:0000002C 07 C3      STMIA    R3!, {R0-R2}
.text:0000002E 04 20      MOVS     R0, #4
.text:00000030 00 90      STR      R0, [SP, #0x18+var_18]
.text:00000032 03 23      MOVS     R3, #3
.text:00000034 02 22      MOVS     R2, #2
.text:00000036 01 21      MOVS     R1, #1
.text:00000038 A0 A0      ADR      R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
.text:0000003A 06 F0 D9 F8  BL      __2printf
.text:0000003E
.text:0000003E      loc_3E   ; CODE XREF: example13_f+16
.text:0000003E 05 B0      ADD      SP, SP, #0x14
.text:00000040 00 BD      POP      {PC}
```

Это почти то же самое что и в предыдущем примере, только код для Thumb и значения помещаются в стек немного иначе: сначала 8 за первый раз, затем 5, 6, 7 за второй раз и 4 за третий раз.

²Читайте больше об этом: [29.2](#) (стр. 439).

³В x86 невозможно установить значение `IP/EIP/RIP` используя `POP`, но будем надеяться, вы поняли аналогию.

Оптимизирующий Xcode 4.6.3 (LLVM): Режим ARM

```

__text:0000290C      _printf_main2
__text:0000290C
__text:0000290C      var_1C = -0x1C
__text:0000290C      var_C  = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9  STMFD  SP!, {R7,LR}
__text:00002910 0D 70 A0 E1  MOV    R7, SP
__text:00002914 14 D0 4D E2  SUB    SP, SP, #0x14
__text:00002918 70 05 01 E3  MOV    R0, #0x1570
__text:0000291C 07 C0 A0 E3  MOV    R12, #7
__text:00002920 00 00 40 E3  MOVT   R0, #0
__text:00002924 04 20 A0 E3  MOV    R2, #4
__text:00002928 00 00 8F E0  ADD    R0, PC, R0
__text:0000292C 06 30 A0 E3  MOV    R3, #6
__text:00002930 05 10 A0 E3  MOV    R1, #5
__text:00002934 00 20 8D E5  STR    R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9  STMFA  SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3  MOV    R9, #8
__text:00002940 01 10 A0 E3  MOV    R1, #1
__text:00002944 02 20 A0 E3  MOV    R2, #2
__text:00002948 03 30 A0 E3  MOV    R3, #3
__text:0000294C 10 90 8D E5  STR    R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB  BL     _printf
__text:00002954 07 D0 A0 E1  MOV    SP, R7
__text:00002958 80 80 BD E8  LDMFD  SP!, {R7,PC}

```

Почти то же самое, что мы уже видели, за исключением того, что **STMFA** (Store Multiple Full Ascending) – это синоним инструкции **STMIB** (Store Multiple Increment Before). Эта инструкция увеличивает **SP** и только затем записывает в память значение очередного регистра, но не наоборот.

Далее бросается в глаза то, что инструкции как будто бы расположены случайно. Например, значение в регистре **R0** подготавливается в трех местах, по адресам **0x2918**, **0x2920** и **0x2928**, когда это можно было бы сделать в одном месте. Однако, у оптимизирующего компилятора могут быть свои доводы о том, как лучше составлять инструкции друг с другом для лучшей эффективности исполнения. Процессор обычно пытается исполнять одновременно идущие друг за другом инструкции. К примеру, инструкции **MOVT R0, #0** и **ADD R0, PC, R0** не могут быть исполнены одновременно, потому что обе инструкции модифицируют регистр **R0**. А вот инструкции **MOVT R0, #0** и **MOV R2, #4** легко можно исполнить одновременно, потому что эффекты от их исполнения никак не конфликтуют друг с другом. Вероятно, компилятор старается генерировать код именно таким образом там, где это возможно.

Оптимизирующий Xcode 4.6.3 (LLVM): Режим Thumb-2

```

__text:00002BA0      _printf_main2
__text:00002BA0
__text:00002BA0      var_1C = -0x1C
__text:00002BA0      var_18 = -0x18
__text:00002BA0      var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5      PUSH    {R7,LR}
__text:00002BA2 6F 46      MOV     R7, SP
__text:00002BA4 85 B0      SUB     SP, SP, #0x14
__text:00002BA6 41 F2 D8 20  MOVW    R0, #0x12D8
__text:00002BAA 4F F0 07 0C  MOV.W   R12, #7
__text:00002BAE C0 F2 00 00  MOVT.W  R0, #0
__text:00002BB2 04 22      MOVS    R2, #4
__text:00002BB4 78 44      ADD     R0, PC ; char *
__text:00002BB6 06 23      MOVS    R3, #6
__text:00002BB8 05 21      MOVS    R1, #5
__text:00002BBA 0D F1 04 0E  ADD.W   LR, SP, #0x1C+var_18
__text:00002BBE 00 92      STR     R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09  MOV.W   R9, #8
__text:00002BC4 8E E8 0A 10  STMIA.W LR, {R1,R3,R12}
__text:00002BC8 01 21      MOVS    R1, #1
__text:00002BCA 02 22      MOVS    R2, #2
__text:00002BCC 03 23      MOVS    R3, #3
__text:00002BCE CD F8 10 90  STR.W   R9, [SP,#0x1C+var_C]

```

7.3. MIPS

```
__text:00002BD2 01 F0 0A EA    BLX    __printf
__text:00002BD6 05 B0          ADD    SP, SP, #0x14
__text:00002BD8 80 BD          POP    {R7,PC}
```

Почти то же самое, что и в предыдущем примере, лишь за тем исключением, что здесь используются Thumb-инструкции.

ARM64

Неоптимизирующий GCC (Linaro) 4.9

Листинг 7.11: Неоптимизирующий GCC (Linaro) 4.9

```
.LC2:
.string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; взять больше места в стеке:
sub    sp, sp, #32
; сохранить FP и LR в стековом фрейме:
stp    x29, x30, [sp,16]
; установить стековый фрейм (FP=SP):
add    x29, sp, 16
adrp   x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
add    x0, x0, :lo12:.LC2
mov    w1, 8      ; 9-й аргумент
str    w1, [sp]   ; сохранить 9-й аргумент в стеке
mov    w1, 1
mov    w2, 2
mov    w3, 3
mov    w4, 4
mov    w5, 5
mov    w6, 6
mov    w7, 7
bl     printf
sub    sp, x29, #16
; восстановить FP и LR
ldp    x29, x30, [sp,16]
add    sp, sp, 32
ret
```

Первые 8 аргументов передаются в X- или W-регистрах: [Procedure Call Standard for the ARM 64-bit Architecture (AArch64), (2013)]⁴. Указатель на строку требует 64-битного регистра, так что он передается в **X0**. Все остальные значения имеют 32-битный тип *int*, так что они записываются в 32-битные части регистров (W-). Девятый аргумент (8) передается через стек. Действительно, невозможно передать большое количество аргументов в регистрах, потому что количество регистров ограничено.

Оптимизирующий GCC (Linaro) 4.9 генерирует почти такой же код.

7.3. MIPS

7.3.1. 3 аргумента

Оптимизирующий GCC 4.4.5

Главное отличие от примера «Hello, world!» в том, что здесь на самом деле вызывается **printf()** вместо **puts()** и ещё три аргумента передаются в регистрах \$5...\$7 (или \$A0...\$A2). Вот почему эти регистры имеют префикс A-. Это значит, что они используются для передачи аргументов.

Листинг 7.12: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
$LC0:
.ascii "a=%d; b=%d; c=%d\000"
main:
```

⁴Также доступно здесь: <http://go.yurichev.com/17287>

7.3. MIPS

```
; пролог функции:
    lui    $28,%hi(__gnu_local_gp)
    addiu  $sp,$sp,-32
    addiu  $28,$28,%lo(__gnu_local_gp)
    sw     $31,28($sp)
; загрузить адрес printf():
    lw     $25,%call16(printf)($28)
; загрузить адрес текстовой строки и установить первый аргумент printf():
    lui    $4,%hi($LC0)
    addiu  $4,$4,%lo($LC0)
; установить второй аргумент printf():
    li     $5,1                # 0x1
; установить третий аргумент printf():
    li     $6,2                # 0x2
; вызов printf():
    jalr   $25
; установить четвертый аргумент printf() (branch delay slot):
    li     $7,3                # 0x3

; эпилог функции:
    lw     $31,28($sp)
; установить возвращаемое значение в 0:
    move   $2,$0
; возврат
    j      $31
    addiu  $sp,$sp,32 ; branch delay slot
```

Листинг 7.13: Оптимизирующий GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4          = -4
.text:00000000
; пролог функции:
.text:00000000                lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                addiu  $sp, -0x20
.text:00000008                la     $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                sw     $ra, 0x20+var_4($sp)
.text:00000010                sw     $gp, 0x20+var_10($sp)
; загрузить адрес printf():
.text:00000014                lw     $t9, (printf & 0xFFFF)($gp)
; загрузить адрес текстовой строки и установить первый аргумент printf():
.text:00000018                la     $a0, $LC0          # "a=%d; b=%d; c=%d"
; установить второй аргумент printf():
.text:00000020                li     $a1, 1
; установить третий аргумент printf():
.text:00000024                li     $a2, 2
; вызов printf():
.text:00000028                jalr   $t9
; установить четвертый аргумент printf() (branch delay slot):
.text:0000002C                li     $a3, 3
; эпилог функции:
.text:00000030                lw     $ra, 0x20+var_4($sp)
; установить возвращаемое значение в 0:
.text:00000034                move   $v0, $zero
; возврат
.text:00000038                jr     $ra
.text:0000003C                addiu  $sp, 0x20 ; branch delay slot
```

IDA объединила пару инструкций `LUI` и `ADDIU` в одну псевдоинструкцию `LA`. Вот почему здесь нет инструкции по адресу 0x1C: потому что `LA` занимает 8 байт.

Неоптимизирующий GCC 4.4.5

Неоптимизирующий GCC более многословен:

```

$LC0:
    .ascii  "a=%d; b=%d; c=%d\000"
main:
; пролог функции:
    addiu   $sp,$sp,-32
    sw      $31,28($sp)
    sw      $fp,24($sp)
    move    $fp,$sp
    lui     $28,%hi(__gnu_local_gp)
    addiu   $28,$28,%lo(__gnu_local_gp)
; загрузить адрес текстовой строки:
    lui     $2,%hi($LC0)
    addiu   $2,$2,%lo($LC0)
; установить первый аргумент printf():
    move    $4,$2
; установить второй аргумент printf():
    li      $5,1                # 0x1
; установить третий аргумент printf():
    li      $6,2                # 0x2
; установить четвертый аргумент printf():
    li      $7,3                # 0x3
; получить адрес printf():
    lw      $2,%call16(printf)($28)
    nop
; вызов printf():
    move    $25,$2
    jalr    $25
    nop

; эпилог функции:
    lw      $28,16($fp)
; установить возвращаемое значение в 0:
    move    $2,$0
    move    $sp,$fp
    lw      $31,28($sp)
    lw      $fp,24($sp)
    addiu   $sp,$sp,32
; возврат
    j       $31
    nop

```

Листинг 7.15: Неоптимизирующий GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8          = -8
.text:00000000 var_4          = -4
.text:00000000
; пролог функции:
.text:00000000                addiu   $sp, -0x20
.text:00000004                sw      $ra, 0x20+var_4($sp)
.text:00000008                sw      $fp, 0x20+var_8($sp)
.text:0000000C                move    $fp, $sp
.text:00000010                la      $gp, __gnu_local_gp
.text:00000018                sw      $gp, 0x20+var_10($sp)
; загрузить адрес текстовой строки:
.text:0000001C                la      $v0, aADBDCD      # "a=%d; b=%d; c=%d"
; установить первый аргумент printf():
.text:00000024                move    $a0, $v0
; установить второй аргумент printf():
.text:00000028                li      $a1, 1
; установить третий аргумент printf():
.text:0000002C                li      $a2, 2
; установить четвертый аргумент printf():
.text:00000030                li      $a3, 3
; получить адрес printf():
.text:00000034                lw      $v0, (printf & 0xFFFF)($gp)
.text:00000038                or      $at, $zero

```

7.3. MIPS

```
; вызов printf():
.text:0000003C          move    $t9, $v0
.text:00000040          jalr     $t9
.text:00000044          or       $at, $zero ; NOP
; эпилог функции:
.text:00000048          lw       $gp, 0x20+var_10($fp)
; установить возвращаемое значение в 0:
.text:0000004C          move    $v0, $zero
.text:00000050          move    $sp, $fp
.text:00000054          lw       $ra, 0x20+var_4($sp)
.text:00000058          lw       $fp, 0x20+var_8($sp)
.text:0000005C          addiu   $sp, 0x20
; возврат
.text:00000060          jr       $ra
.text:00000064          or       $at, $zero ; NOP
```

7.3.2. 8 аргументов

Снова воспользуемся примером с 9-ю аргументами из предыдущей секции: [7.1.2](#) (стр. 45).

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
};
```

Оптимизирующий GCC 4.4.5

Только 4 первых аргумента передаются в регистрах \$A0 ...\$A3, так что остальные передаются через стек.

Это соглашение о вызовах O32 (самое популярное в мире MIPS). Другие соглашения о вызовах (например N32) могут наделять регистры другими функциями.

SW означает «Store Word» (записать слово из регистра в память). В MIPS нет инструкции для записи значения в память, так что для этого используется пара инструкций (**LI** / **SW**).

Листинг 7.16: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
$LC0:
.ascii "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; пролог функции:
    lui     $28,%hi(__gnu_local_gp)
    addiu   $sp,$sp,-56
    addiu   $28,$28,%lo(__gnu_local_gp)
    sw      $31,52($sp)
; передать 5-й аргумент в стеке:
    li      $2,4                # 0x4
    sw      $2,16($sp)
; передать 6-й аргумент в стеке:
    li      $2,5                # 0x5
    sw      $2,20($sp)
; передать 7-й аргумент в стеке:
    li      $2,6                # 0x6
    sw      $2,24($sp)
; передать 8-й аргумент в стеке:
    li      $2,7                # 0x7
    lw      $25,%call16(printf)($28)
    sw      $2,28($sp)
; передать 1-й аргумент в $a0:
    lui     $4,%hi($LC0)
; передать 9-й аргумент в стеке:
    li      $2,8                # 0x8
    sw      $2,32($sp)
    addiu   $4,$4,%lo($LC0)
```

7.3. MIPS

```
; передать 2-й аргумент в $a1:
    li    $5,1                # 0x1
; передать 3-й аргумент в $a2:
    li    $6,2                # 0x2
; вызов printf():
    jalr   $25
; передать 4-й аргумент в $a3 (branch delay slot):
    li    $7,3                # 0x3

; эпилог функции:
    lw     $31,52($sp)
; установить возвращаемое значение в 0:
    move   $2,$0
; возврат
    j      $31
    addiu  $sp,$sp,56 ; branch delay slot
```

Листинг 7.17: Оптимизирующий GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; пролог функции:
.text:00000000                lui    $gp, (__gnu_local_gp >> 16)
.text:00000004                addiu  $sp, -0x38
.text:00000008                la     $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                sw     $ra, 0x38+var_4($sp)
.text:00000010                sw     $gp, 0x38+var_10($sp)
; передать 5-й аргумент в стеке:
.text:00000014                li     $v0, 4
.text:00000018                sw     $v0, 0x38+var_28($sp)
; передать 6-й аргумент в стеке:
.text:0000001C                li     $v0, 5
.text:00000020                sw     $v0, 0x38+var_24($sp)
; передать 7-й аргумент в стеке:
.text:00000024                li     $v0, 6
.text:00000028                sw     $v0, 0x38+var_20($sp)
; передать 8-й аргумент в стеке:
.text:0000002C                li     $v0, 7
.text:00000030                lw     $t9, (printf & 0xFFFF)($gp)
.text:00000034                sw     $v0, 0x38+var_1C($sp)
; готовить 1-й аргумент в $a0:
.text:00000038                lui    $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g%
    ↪ =%"...
; передать 9-й аргумент в стеке:
.text:0000003C                li     $v0, 8
.text:00000040                sw     $v0, 0x38+var_18($sp)
; передать 1-й аргумент в $a1:
.text:00000044                la     $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g%
    ↪ =%"...
; передать 2-й аргумент в $a1:
.text:00000048                li     $a1, 1
; передать 3-й аргумент в $a2:
.text:0000004C                li     $a2, 2
; вызов printf():
.text:00000050                jalr   $t9
; передать 4-й аргумент в $a3 (branch delay slot):
.text:00000054                li     $a3, 3
; эпилог функции:
.text:00000058                lw     $ra, 0x38+var_4($sp)
; установить возвращаемое значение в 0:
.text:0000005C                move   $v0, $zero
; возврат
```

7.3. MIPS

```
.text:00000060      jr      $ra
.text:00000064      addiu   $sp, 0x38 ; branch delay slot
```

Неоптимизирующий GCC 4.4.5

Неоптимизирующий GCC более многословен:

Листинг 7.18: Неоптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
$LC0:
    .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; пролог функции:
    addiu   $sp,$sp,-56
    sw      $31,52($sp)
    sw      $fp,48($sp)
    move    $fp,$sp
    lui     $28,%hi(__gnu_local_gp)
    addiu   $28,$28,%lo(__gnu_local_gp)
    lui     $2,%hi($LC0)
    addiu   $2,$2,%lo($LC0)
; передать 5-й аргумент в стеке:
    li      $3,4                # 0x4
    sw      $3,16($sp)
; передать 6-й аргумент в стеке:
    li      $3,5                # 0x5
    sw      $3,20($sp)
; передать 7-й аргумент в стеке:
    li      $3,6                # 0x6
    sw      $3,24($sp)
; передать 8-й аргумент в стеке:
    li      $3,7                # 0x7
    sw      $3,28($sp)
; передать 9-й аргумент в стеке:
    li      $3,8                # 0x8
    sw      $3,32($sp)
; передать 1-й аргумент в $a0:
    move    $4,$2
; передать 2-й аргумент в $a1:
    li      $5,1                # 0x1
; передать 3-й аргумент в $a2:
    li      $6,2                # 0x2
; передать 4-й аргумент в $a3:
    li      $7,3                # 0x3
; вызов printf():
    lw      $2,%call16(printf)($28)
    nop
    move    $25,$2
    jalr    $25
    nop
; эпилог функции:
    lw      $28,40($fp)
; установить возвращаемое значение в 0:
    move    $2,$0
    move    $sp,$fp
    lw      $31,52($sp)
    lw      $fp,48($sp)
    addiu   $sp,$sp,56
; возврат
    j       $31
    nop
```

Листинг 7.19: Неоптимизирующий GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
```


7.4. Вывод

```
.text:00000000 var_1C      = -0x1C
.text:00000000 var_18      = -0x18
.text:00000000 var_10      = -0x10
.text:00000000 var_8       = -8
.text:00000000 var_4       = -4
.text:00000000
; пролог функции:
.text:00000000      addiu   $sp, -0x38
.text:00000004      sw      $ra, 0x38+var_4($sp)
.text:00000008      sw      $fp, 0x38+var_8($sp)
.text:0000000C      move    $fp, $sp
.text:00000010      la      $gp, __gnu_local_gp
.text:00000018      sw      $gp, 0x38+var_10($sp)
.text:0000001C      la      $v0, aADBDCDDDEDFDGD # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g↵
↵ =%"...
; передать 5-й аргумент в стеке:
.text:00000024      li      $v1, 4
.text:00000028      sw      $v1, 0x38+var_28($sp)
; передать 6-й аргумент в стеке:
.text:0000002C      li      $v1, 5
.text:00000030      sw      $v1, 0x38+var_24($sp)
; передать 7-й аргумент в стеке:
.text:00000034      li      $v1, 6
.text:00000038      sw      $v1, 0x38+var_20($sp)
; передать 8-й аргумент в стеке:
.text:0000003C      li      $v1, 7
.text:00000040      sw      $v1, 0x38+var_1C($sp)
; передать 9-й аргумент в стеке:
.text:00000044      li      $v1, 8
.text:00000048      sw      $v1, 0x38+var_18($sp)
; передать 1-й аргумент в $a0:
.text:0000004C      move    $a0, $v0
; передать 2-й аргумент в $a1:
.text:00000050      li      $a1, 1
; передать 3-й аргумент в $a2:
.text:00000054      li      $a2, 2
; передать 4-й аргумент в $a3:
.text:00000058      li      $a3, 3
; вызов printf():
.text:0000005C      lw      $v0, (printf & 0xFFFF)($gp)
.text:00000060      or      $at, $zero
.text:00000064      move    $t9, $v0
.text:00000068      jalr    $t9
.text:0000006C      or      $at, $zero ; NOP
; эпилог функции:
.text:00000070      lw      $gp, 0x38+var_10($fp)
; установить возвращаемое значение в 0:
.text:00000074      move    $v0, $zero
.text:00000078      move    $sp, $fp
.text:0000007C      lw      $ra, 0x38+var_4($sp)
.text:00000080      lw      $fp, 0x38+var_8($sp)
.text:00000084      addiu   $sp, 0x38
; возврат
.text:00000088      jr      $ra
.text:0000008C      or      $at, $zero ; NOP
```

7.4. Вывод

Вот примерный скелет вызова функции:

Листинг 7.20: x86

```
...
PUSH третий аргумент
PUSH второй аргумент
PUSH первый аргумент
CALL функция
; модифицировать указатель стека если( нужно)
```

Листинг 7.21: x64 (MSVC)

```

MOV RCX, первый аргумент
MOV RDX, второй аргумент
MOV R8, третий аргумент
MOV R9, й4- аргумент
...
PUSH й5-, й6- аргумент, итд~.. если( нужно)
CALL функция
; модифицировать указатель стека если( нужно)

```

Листинг 7.22: x64 (GCC)

```

MOV RDI, первый аргумент
MOV RSI, второй аргумент
MOV RDX, третий аргумент
MOV RCX, й4- аргумент
MOV R8, й5- аргумент
MOV R9, й6- аргумент
...
PUSH й7-, й8- аргумент, итд~.. если( нужно)
CALL функция
; модифицировать указатель стека если( нужно)

```

Листинг 7.23: ARM

```

MOV R0, первый аргумент
MOV R1, второй аргумент
MOV R2, третий аргумент
MOV R3, й4- аргумент
; передать й5-, й6- аргумент, итд~..., в стеке если( нужно)
BL функция
; модифицировать указатель стека если( нужно)

```

Листинг 7.24: ARM64

```

MOV X0, первый аргумент
MOV X1, второй аргумент
MOV X2, третий аргумент
MOV X3, й4- аргумент
MOV X4, й5- аргумент
MOV X5, й6- аргумент
MOV X6, й7- аргумент
MOV X7, й8- аргумент
; передать й9-, й10- аргумент, итд~..., в стеке если( нужно)
BL CALL функция
; модифицировать указатель стека если( нужно)

```

Листинг 7.25: MIPS (соглашение о вызовах O32)

```

LI $4, первый аргумент ; AKA $A0
LI $5, второй аргумент ; AKA $A1
LI $6, третий аргумент ; AKA $A2
LI $7, й4- аргумент ; AKA $A3
; передать й5-, й6- аргумент, итд~..., в стеке если( нужно)
LW temp_reg, адрес функции
JALR temp_reg

```

7.5. Кстати

Кстати, разница между способом передачи параметров принятая в x86, x64, fastcall, ARM и MIPS неплохо иллюстрирует тот важный момент, что процессору, в общем, всё равно, как будут передаваться параметры функций. Можно создать гипотетический компилятор, который будет передавать их при помощи указателя на структуру с параметрами, не пользуясь стеком вообще.

7.5. КСТАТИ

Регистры \$A0...\$A3 в MIPS так названы только для удобства (это соглашение о вызовах O32). Программисты могут использовать любые другие регистры (может быть, только кроме \$ZERO) для передачи данных или любое другое соглашение о вызовах.

CPU не знает о соглашениях о вызовах вообще.

Можно также вспомнить, что начинающие программисты на ассемблере передают параметры в другие функции обычно через регистры, без всякого явного порядка, или даже через глобальные переменные. И всё это нормально работает.

Глава 8

scanf()

Теперь попробуем использовать `scanf()`.

8.1. Простой пример

```
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
};
```

Использовать `scanf()` в наши времена для того, чтобы спросить у пользователя что-то — не самая хорошая идея. Но так мы проиллюстрируем передачу указателя на переменную типа `int`.

8.1.1. Об указателях

Это одна из фундаментальных вещей в информатике. Часто большой массив, структуру или объект передавать в другую функцию путем копирования данных невыгодно, а передать адрес массива, структуры или объекта куда проще. К тому же, если вызываемая функция (*callee*) должна изменить что-то в этом большом массиве или структуре, то возвращать её полностью так же абсурдно. Так что самое простое, что можно сделать, это передать в функцию-*callee* адрес массива или структуры, и пусть *callee* что-то там изменит.

Указатель в Си/Си++ — это просто адрес какого-либо места в памяти.

В x86 адрес представляется в виде 32-битного числа (т.е. занимает 4 байта), а в x86-64 как 64-битное число (занимает 8 байт). Кстати, отсюда негодование некоторых людей, связанное с переходом на x86-64 — на этой архитектуре все указатели занимают в 2 раза больше места, в том числе и в “дорогой” кэш-памяти.

При некотором упорстве можно работать только с безтиповыми указателями (`void*`), например, стандартная функция Си `memcpy()`, копирующая блок из одного места памяти в другое принимает на вход 2 указателя типа `void*`, потому что нельзя заранее предугадать, какого типа блок вы собираетесь копировать. Для копирования тип данных не важен, важен только размер блока.

Также указатели широко используются, когда функции нужно вернуть более одного значения (мы ещё вернемся к этому в будущем (11 (стр. 106))).

Функция `scanf()` — это как раз такой случай.

Помимо того, что этой функции нужно показать, сколько значений было прочитано успешно, ей ещё и нужно вернуть сами значения.

8.1. ПРОСТОЙ ПРИМЕР

Тип указателя в Си/Си++ нужен только для проверки типов на стадии компиляции.

Внутри, в скомпилированном коде, никакой информации о типах указателей нет вообще.

8.1.2. x86

MSVC

Что получаем на ассемблере, компилируя в MSVC 2010:

```
CONST    SEGMENT
$SG3831  DB    'Enter X:', 0aH, 00H
$SG3832  DB    '%d', 00H
$SG3833  DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC   _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_x$ = -4                                ; size = 4
_main    PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    push    OFFSET $SG3831 ; 'Enter X:'
    call    _printf
    add     esp, 4
    lea     eax, DWORD PTR _x$[ebp]
    push    eax
    push    OFFSET $SG3832 ; '%d'
    call    _scanf
    add     esp, 8
    mov     ecx, DWORD PTR _x$[ebp]
    push    ecx
    push    OFFSET $SG3833 ; 'You entered %d...'
    call    _printf
    add     esp, 8

    ; возврат 0
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP
_TEXT     ENDS
```

Переменная `x` является локальной.

По стандарту Си/Си++ она доступна только из этой же функции и нигде более. Так получилось, что локальные переменные располагаются в стеке. Может быть, можно было бы использовать и другие варианты, но в x86 это традиционно так.

Следующая после пролога инструкция `PUSH ECX` не ставит своей целью сохранить значение регистра `ECX`. (Заметьте отсутствие соответствующей инструкции `POP ECX` в конце функции).

Она на самом деле выделяет в стеке 4 байта для хранения `x` в будущем.

Доступ к `x` будет осуществляться при помощи объявленного макроса `_x$` (он равен -4) и регистра `EBP` указывающего на текущий фрейм.

Во всё время исполнения функции `EBP` указывает на текущий фрейм и через `EBP+смещение` можно получить доступ как к локальным переменным функции, так и аргументам функции.

Можно было бы использовать `ESP`, но он во время исполнения функции часто меняется, а это не удобно. Так что можно сказать, что `EBP` это *замороженное состояние* `ESP` на момент начала исполнения функции.

Разметка типичного стекового фрейма в 32-битной среде:

8.1. ПРОСТОЙ ПРИМЕР

...	...
EBP-8	локальная переменная #2, маркируется в IDA как var_8
EBP-4	локальная переменная #1, маркируется в IDA как var_4
EBP	сохраненное значение EBP
EBP+4	адрес возврата
EBP+8	аргумент#1, маркируется в IDA как arg_0
EBP+0xC	аргумент#2, маркируется в IDA как arg_4
EBP+0x10	аргумент#3, маркируется в IDA как arg_8
...	...

У функции `scanf()` в нашем примере два аргумента.

Первый – указатель на строку, содержащую `%d` и второй – адрес переменной `x`.

Вначале адрес `x` помещается в регистр `EAX` при помощи инструкции `lea eax, DWORD PTR _x$[ebp]`.

Инструкция `LEA` означает *load effective address*, и часто используется для формирования адреса чего-либо (A.6.2 (стр. 917)).

Можно сказать, что в данном случае `LEA` просто помещает в `EAX` результат суммы значения в регистре `EBP` и макроса `_x$`.

Это тоже что и `lea eax, [ebp-4]`.

Итак, от значения `EBP` отнимается 4 и помещается в `EAX`. Далее значение `EAX` заталкивается в стек и вызывается `scanf()`.

После этого вызывается `printf()`. Первый аргумент вызова строка: `You entered %d...\n`.

Второй аргумент: `mov ecx, [ebp-4]`. Эта инструкция помещает в `ECX` не адрес переменной `x`, а её значение.

Далее значение `ECX` заталкивается в стек и вызывается `printf()`.

8.1.3. MSVC + OllyDbg

Попробуем этот же пример в OllyDbg. Загружаем, нажимаем F8 (сделать шаг, не входя в функцию) до тех пор, пока не окажемся в своем исполняемом файле, а не в `ntdll.dll`. Прокручиваем вверх до тех пор, пока не найдем `main()`. Щелкаем на первой инструкции (`PUSH EBP`), нажимаем F2 (*set a breakpoint*), затем F9 (*Run*) и точка останова срабатывает на начале `main()`.

Трассируем до того места, где готовится адрес переменной `x`:

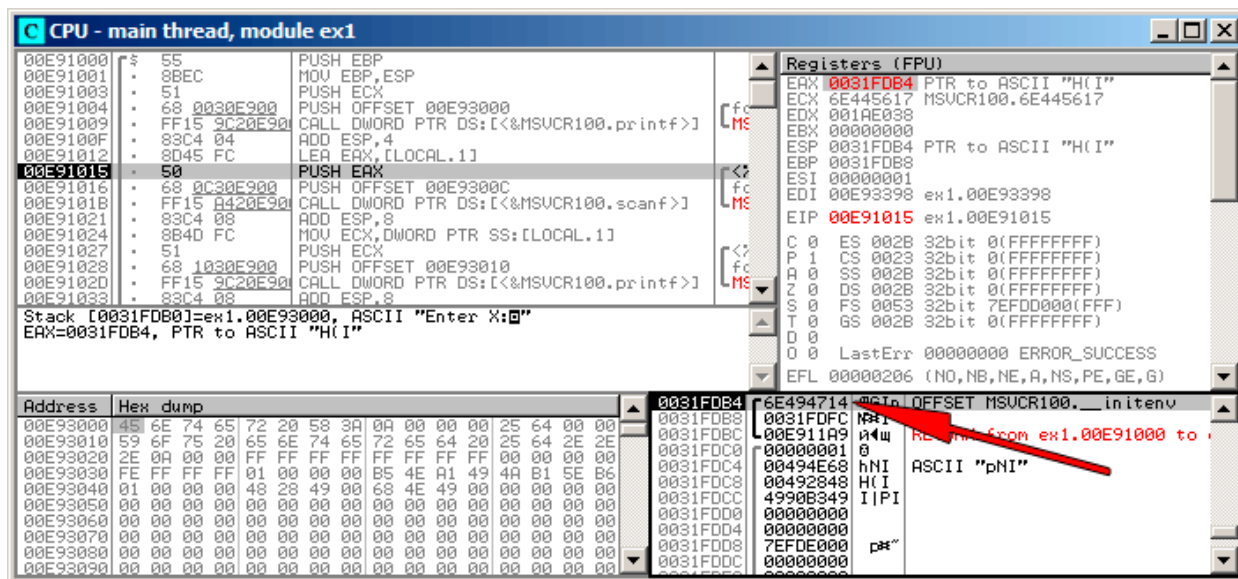


Рис. 8.1: OllyDbg: вычисляется адрес локальной переменной

На `EAX` в окне регистров можно нажать правой кнопкой и далее выбрать «Follow in stack». Этот адрес покажется в окне стека.

Смотрите, это переменная в локальном стеке. Там дорисована красная стрелка. И там сейчас какой-то мусор (`0x6E494714`). Адрес этого элемента стека сейчас, при помощи `PUSH` запишется в этот же стек рядом. Трассируем при помощи F8 вплоть до конца исполнения `scanf()`. А пока `scanf()` выполняется, в консольном окне, вводим, например, 123:

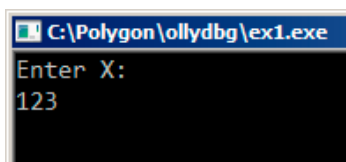


Рис. 8.2: Ввод пользователя в консольном окне

8.1. ПРОСТОЙ ПРИМЕР

Вот тут `scanf()` отработал:

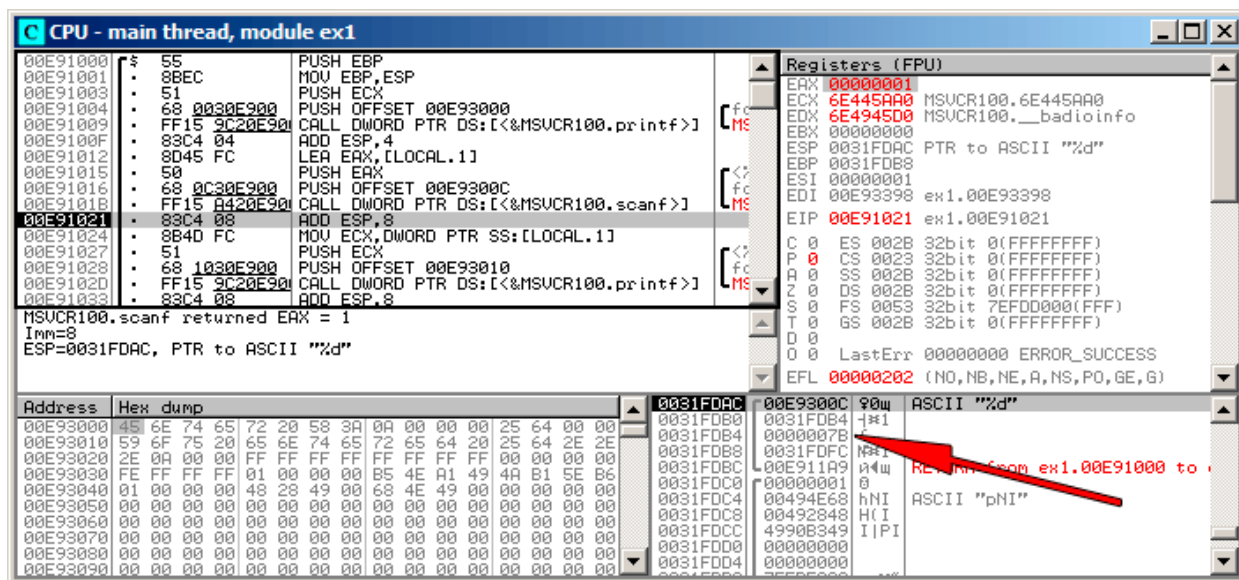


Рис. 8.3: OllyDbg: `scanf()` исполнилась

`scanf()` вернул 1 в `EAX`, что означает, что он успешно прочитал одно значение. В наблюдаемом нами элементе стека теперь `0x7B` (123).

8.1. ПРОСТОЙ ПРИМЕР

Чуть позже это значение копируется из стека в регистр `ECX` и передается в `printf()` :

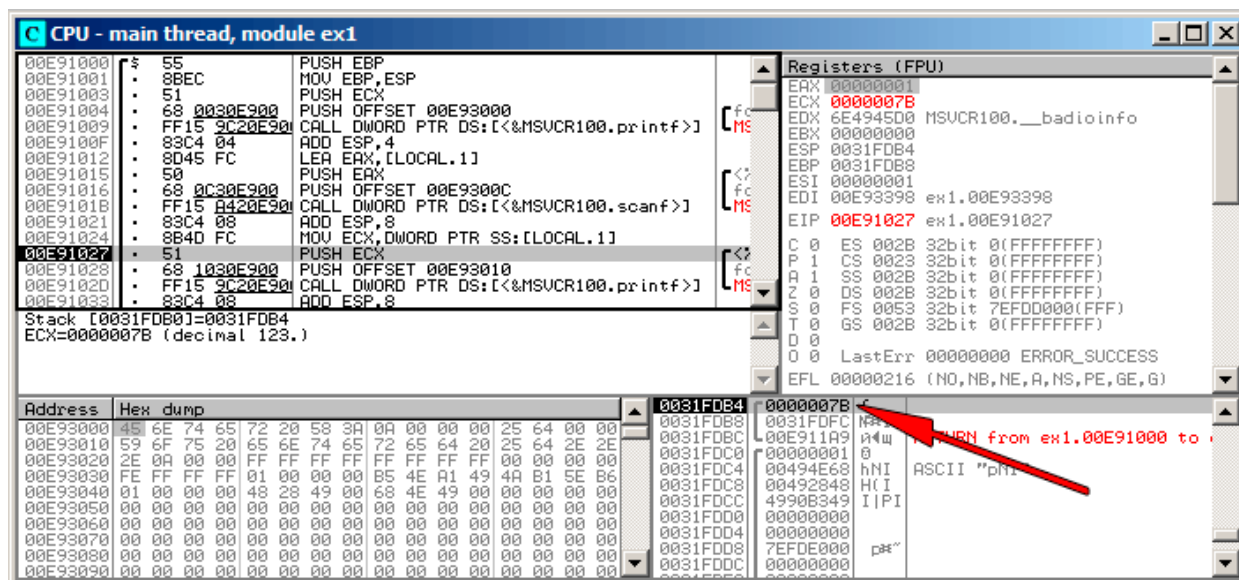


Рис. 8.4: OllyDbg: готовим значение для передачи в `printf()`

GCC

Попробуем тоже самое скомпилировать в Linux при помощи GCC 4.4.1:

```
main      proc near
var_20    = dword ptr -20h
var_1C    = dword ptr -1Ch
var_4     = dword ptr -4

    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFF0h
    sub     esp, 20h
    mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
    call    _puts
    mov     eax, offset aD ; "%d"
    lea     edx, [esp+20h+var_4]
    mov     [esp+20h+var_1C], edx
    mov     [esp+20h+var_20], eax
    call    ___isoc99_scanf
    mov     edx, [esp+20h+var_4]
    mov     eax, offset aYouEnteredD ; "You entered %d...\n"
    mov     [esp+20h+var_1C], edx
    mov     [esp+20h+var_20], eax
    call    _printf
    mov     eax, 0
    leave
    retn
main      endp
```

GCC заменил первый вызов `printf()` на `puts()`. Почему это было сделано, уже было описано ранее (4.4.3 (стр. 17)).

Далее всё как и прежде — параметры заталкиваются через стек при помощи `MOV`.

Кстати

Кстати, этот простой пример иллюстрирует то обстоятельство, что компилятор преобразует список выражений в Си/Си++ блоке просто в последовательный набор инструкций. Между выражениями в Си/Си++ ничего нет, и в итоговом машинном коде между ними тоже ничего нет, управление переходит от одной инструкции к следующей за ней.

8.1.4. x64

Всё то же самое, только используются регистры вместо стека для передачи аргументов функций.

MSVC

Листинг 8.1: MSVC 2012 x64

```

_DATA SEGMENT
$SG1289 DB 'Enter X:', 0aH, 00H
$SG1291 DB '%d', 00H
$SG1292 DB 'You entered %d...', 0aH, 00H
_DATA ENDS

_TEXT SEGMENT
x$ = 32
main PROC
$LN3:
    sub     rsp, 56
    lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
    call    printf
    lea     rdx, QWORD PTR x$[rsp]
    lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
    call    scanf
    mov     edx, DWORD PTR x$[rsp]
    lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
    call    printf

    ; возврат 0
    xor     eax, eax
    add     rsp, 56
    ret     0
main ENDP
_TEXT ENDS

```

GCC

Листинг 8.2: Оптимизирующий GCC 4.4.6 x64

```

.LC0:
.string "Enter X:"
.LC1:
.string "%d"
.LC2:
.string "You entered %d...\n"

main:
    sub     rsp, 24
    mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
    call    puts
    lea     rsi, [rsp+12]
    mov     edi, OFFSET FLAT:.LC1 ; "%d"
    xor     eax, eax
    call    __isoc99_scanf
    mov     esi, DWORD PTR [rsp+12]
    mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
    xor     eax, eax
    call    printf

    ; возврат 0
    xor     eax, eax
    add     rsp, 24
    ret

```

8.1.5. ARM

Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:00000042      scanf_main
.text:00000042
.text:00000042      var_8          = -8
.text:00000042
.text:00000042 08 B5          PUSH    {R3,LR}
.text:00000044 A9 A0          ADR     R0, aEnterX      ; "Enter X:\n"
.text:00000046 06 F0 D3 F8      BL     __2printf
.text:0000004A 69 46          MOV     R1, SP
.text:0000004C AA A0          ADR     R0, aD        ; "%d"
.text:0000004E 06 F0 CD F8      BL     __0scanf
.text:00000052 00 99          LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0          ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000056 06 F0 CB F8      BL     __2printf
.text:0000005A 00 20          MOVS    R0, #0
.text:0000005C 08 BD          POP     {R3,PC}
```

Чтобы `scanf()` мог вернуть значение, ему нужно передать указатель на переменную типа *int*. *int* – 32-битное значение, для его хранения нужно только 4 байта, и оно помещается в 32-битный регистр.

Место для локальной переменной `x` выделяется в стеке, *IDA* наименовала её `var_8`. Впрочем, место для неё выделять не обязательно, т.к. [указатель стека SP](#) уже указывает на место, свободное для использования. Так что значение указателя `SP` копируется в регистр `R1`, и вместе с `format`-строкой, передается в `scanf()`.

Позже, при помощи инструкции `LDR`, это значение перемещается из стека в регистр `R1`, чтобы быть переданным в `printf()`.

ARM64

Листинг 8.3: Неоптимизирующий GCC 4.9.1 ARM64

```
1  .LC0:
2      .string "Enter X:"
3  .LC1:
4      .string "%d"
5  .LC2:
6      .string "You entered %d...\n"
7  scanf_main:
8      ; вычесть 32 из SP, затем сохранить FP и LR в стековом фрейме:
9      stp    x29, x30, [sp, -32]!
10     ; установить стековый фрейм (FP=SP)
11     add     x29, sp, 0
12     ; загрузить указатель на строку "Enter X:"
13     adrp    x0, .LC0
14     add     x0, x0, :lo12:.LC0
15     ; X0=указатель на строку "Enter X:"
16     ; вывести её:
17     bl      puts
18     ; загрузить указатель на строку "%d":
19     adrp    x0, .LC1
20     add     x0, x0, :lo12:.LC1
21     ; найти место в стековом фрейме для переменной "x" (X1=FP+28):
22     add     x1, x29, 28
23     ; X1=адрес переменной "x"
24     ; передать адрес в scanf() и вызвать её:
25     bl      __isoc99_scanf
26     ; загрузить 32-битное значение из переменной в стековом фрейме:
27     ldr     w1, [x29,28]
28     ; W1=x
29     ; загрузить указатель на строку "You entered %d...\n"
30     ; printf() возьмет текстовую строку из X0 и переменную "x" из X1 (или W1)
31     adrp    x0, .LC2
32     add     x0, x0, :lo12:.LC2
33     bl      printf
```

8.1. ПРОСТОЙ ПРИМЕР

```
34 ; возврат 0
35     mov     w0, 0
36 ; восстановить FP и LR, затем прибавить 32 к SP:
37     ldp     x29, x30, [sp], 32
38     ret
```

Под стековый фрейм выделяется 32 байта, что больше чем нужно. Вероятно, это связано с выравниваем по границе памяти? Самая интересная часть — это поиск места под переменную x в стековом фрейме (строка 22). Почему 28? Почему-то, компилятор решил расположить эту переменную в конце стекового фрейма, а не в начале. Адрес потом передается в `scanf()`, которая просто сохраняет значение, введенное пользователем, в памяти по этому адресу. Это 32-битное значение типа `int`. Значение загружается в строке 27 и затем передается в `printf()`.

8.1.6. MIPS

Для переменной x выделено место в стеке, и к нему будут производиться обращения как $\$sp + 24$.

Её адрес передается в `scanf()`, а значение прочитанное от пользователя загружается используя инструкцию `LW` («Load Word» — загрузить слово) и затем оно передается в `printf()`.

Листинг 8.4: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
$LC0:
    .ascii  "Enter X:\000"
$LC1:
    .ascii  "%d\000"
$LC2:
    .ascii  "You entered %d...\012\000"
main:
; пролог функции:
    lui     $28,%hi(__gnu_local_gp)
    addiu   $sp,$sp,-40
    addiu   $28,$28,%lo(__gnu_local_gp)
    sw      $31,36($sp)
; вызов puts():
    lw      $25,%call16(puts)($28)
    lui     $4,%hi($LC0)
    jalr    $25
    addiu   $4,$4,%lo($LC0) ; branch delay slot
; вызов scanf():
    lw      $28,16($sp)
    lui     $4,%hi($LC1)
    lw      $25,%call16(__isoc99_scanf)($28)
; установить второй аргумент для scanf(), $a1=$sp+24:
    addiu   $5,$sp,24
    jalr    $25
    addiu   $4,$4,%lo($LC1) ; branch delay slot

; вызов printf():
    lw      $28,16($sp)
; установить второй аргумент для printf(),
; загрузить слово по адресу $sp+24:
    lw      $5,24($sp)
    lw      $25,%call16(printf)($28)
    lui     $4,%hi($LC2)
    jalr    $25
    addiu   $4,$4,%lo($LC2) ; branch delay slot

; эпилог функции:
    lw      $31,36($sp)
; установить возвращаемое значение в 0:
    move    $2,$0
; возврат:
    j       $31
    addiu   $sp,$sp,40 ; branch delay slot
```

IDA показывает разметку стека следующим образом:

```
.text:00000000 main:
.text:00000000
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; пролог функции:
.text:00000000                lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                addiu   $sp, -0x28
.text:00000008                la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                sw      $ra, 0x28+var_4($sp)
.text:00000010                sw      $gp, 0x28+var_18($sp)
; вызов puts():
.text:00000014                lw      $t9, (puts & 0xFFFF)($gp)
.text:00000018                lui     $a0, ($LC0 >> 16) # "Enter X:"
.text:0000001C                jalr    $t9
.text:00000020                la      $a0, ($LC0 & 0xFFFF) # "Enter X:" ; branch delay slot
; вызов scanf():
.text:00000024                lw      $gp, 0x28+var_18($sp)
.text:00000028                lui     $a0, ($LC1 >> 16) # "%d"
.text:0000002C                lw      $t9, (__isoc99_scanf & 0xFFFF)($gp)
; установить второй аргумент для scanf(), $a1=$sp+24:
.text:00000030                addiu   $a1, $sp, 0x28+var_10
.text:00000034                jalr    $t9 ; branch delay slot
.text:00000038                la      $a0, ($LC1 & 0xFFFF) # "%d"
; вызов printf():
.text:0000003C                lw      $gp, 0x28+var_18($sp)
; установить второй аргумент для printf(),
; загрузить слово по адресу $sp+24:
.text:00000040                lw      $a1, 0x28+var_10($sp)
.text:00000044                lw      $t9, (printf & 0xFFFF)($gp)
.text:00000048                lui     $a0, ($LC2 >> 16) # "You entered %d...\n"
.text:0000004C                jalr    $t9
.text:00000050                la      $a0, ($LC2 & 0xFFFF) # "You entered %d...\n" ; branch delay slot
; ↙ slot
; эпилог функции:
.text:00000054                lw      $ra, 0x28+var_4($sp)
; установить возвращаемое значение в 0:
.text:00000058                move    $v0, $zero
; возврат:
.text:0000005C                jr      $ra
.text:00000060                addiu   $sp, 0x28 ; branch delay slot
```

8.2. Глобальные переменные

А что если переменная `x` из предыдущего примера будет глобальной переменной, а не локальной? Тогда к ней смогут обращаться из любого другого места, а не только из тела функции. Глобальные переменные считаются [анти-паттерном](#), но ради примера мы можем себе это позволить.

```
#include <stdio.h>

// теперь x это глобальная переменная
int x;

int main()
{
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
};
```

8.2.1. MSVC: x86

```

_DATA    SEGMENT
COMM     _x:DWORD
$SG2456  DB      'Enter X:', 0aH, 00H
$SG2457  DB      '%d', 00H
$SG2458  DB      'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC   _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push  ebp
    mov   ebp, esp
    push  OFFSET $SG2456
    call  _printf
    add   esp, 4
    push  OFFSET _x
    push  OFFSET $SG2457
    call  _scanf
    add   esp, 8
    mov   eax, DWORD PTR _x
    push  eax
    push  OFFSET $SG2458
    call  _printf
    add   esp, 8
    xor   eax, eax
    pop   ebp
    ret   0
_main    ENDP
_TEXT    ENDS

```

В целом ничего особенного. Теперь `x` объявлена в сегменте `_DATA`. Память для неё в стеке более не выделяется. Все обращения к ней происходит не через стек, а уже напрямую. Неинициализированные глобальные переменные не занимают места в исполняемом файле (и действительно, зачем в исполняемом файле нужно выделять место под изначально нулевые переменные?), но тогда, когда к этому месту в памяти кто-то обратится, [ОС](#) подставит туда блок, состоящий из нулей¹.

Попробуем изменить объявление этой переменной:

```
int x=10; // значение по умолчанию
```

Выйдет в итоге:

```

_DATA    SEGMENT
_x       DD      0aH
...

```

Здесь уже по месту этой переменной записано `0xA` с типом DD (dword = 32 бита).

Если вы откроете скомпилированный .exe-файл в [IDA](#), то увидите, что `x` находится в начале сегмента `_DATA`, после этой переменной будут текстовые строки.

А вот если вы откроете в [IDA](#).exe скомпилированный в прошлом примере, где значение `x` не определено, то вы увидите:

```

.data:0040FA80 _x                dd ?                ; DATA XREF: _main+10
.data:0040FA80                ; _main+22
.data:0040FA84 dword_40FA84      dd ?                ; DATA XREF: _memset+1E
.data:0040FA84                ; unknown_libname_1+28
.data:0040FA88 dword_40FA88      dd ?                ; DATA XREF: __sbh_find_block+5
.data:0040FA88                ; __sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem            dd ?                ; DATA XREF: __sbh_find_block+B
.data:0040FA8C                ; __sbh_free_block+2CA
.data:0040FA90 dword_40FA90      dd ?                ; DATA XREF: _V6_HeapAlloc+13

```

¹Так работает [VM](#)

8.2. ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

.data:0040FA90				; __calloc_impl+72
.data:0040FA94	dword_40FA94	dd	?	; DATA XREF: ____sbh_free_block+2FE

`__x` обозначен как `?`, наряду с другими переменными не требующими инициализации. Это означает, что при загрузке .exe в память, место под всё это выделено будет и будет заполнено нулевыми байтами [ISO/IEC 9899:TC3 (C99 standard), (2007)6.7.8p10]. Но в самом .exe ничего этого нет. Неинициализированные переменные не занимают места в исполняемых файлах. Это удобно для больших массивов, например.

8.2.2. MSVC: x86 + OllyDbg

Тут даже проще:

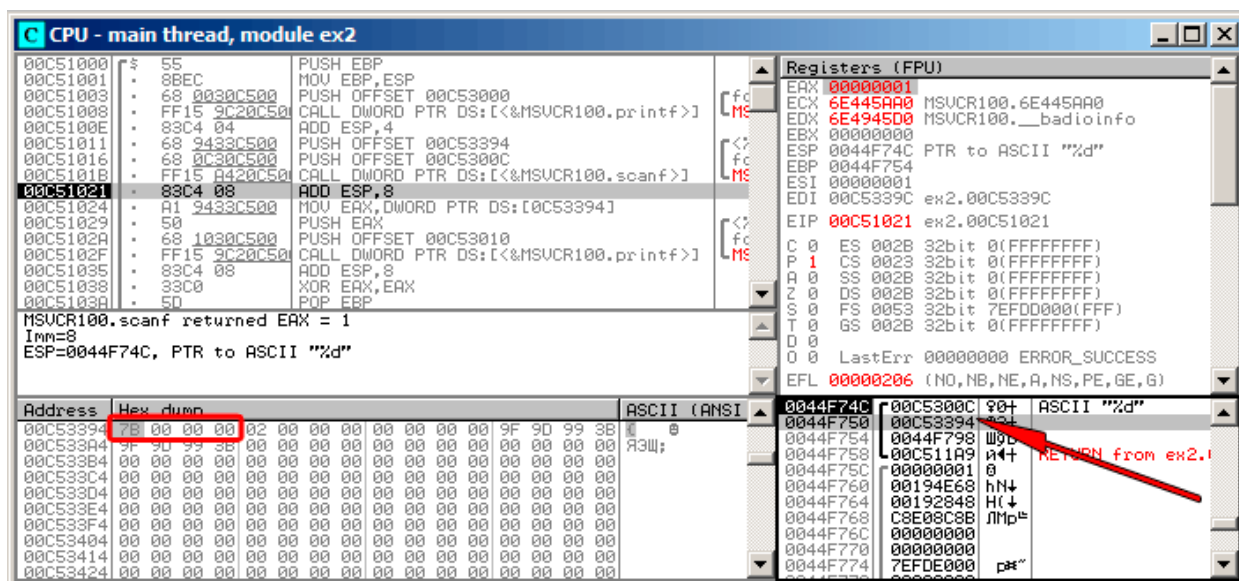


Рис. 8.5: OllyDbg: после исполнения `scanf()`

Переменная хранится в сегменте данных. Кстати, после исполнения инструкции `PUSH` (заталкивающей адрес x) адрес появится в стеке, и на этом элементе можно нажать правой кнопкой, выбрать «Follow in dump». И в окне памяти слева появится эта переменная.

После того как в консоли введем 123, здесь появится `0x7B`.

Почему самый первый байт это `7B`? По логике вещей, здесь должно было бы быть `00 00 00 7B`. Это называется [endianness](#), и в x86 принят формат *little-endian*. Это означает, что в начале записывается самый младший байт, а заканчивается самым старшим байтом. Больше об этом: [32](#) (стр. 448).

Позже из этого места в памяти 32-битное значение загружается в `EAX` и передается в `printf()`.

Адрес переменной x в памяти `0x00C53394`.

8.2. ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

В OllyDbg мы можем посмотреть карту памяти процесса (Alt-M) и увидим, что этот адрес внутри PE-сегмента `.data` нашей программы:

Memory map									
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as	
00070000	00067000			Heap	Map	R	R	C:\Windows\System32\locale.nls	
00190000	00005000				Priv	RW	RW		
00200000	00007000				Priv	RW	Gua	RW	Gua
0044C000	00001000				Priv	RW	Gua	RW	Gua
0044D000	00003000			Stack of main thread	Priv	RW	RW		
00590000	00007000				Priv	RW	RW		
00750000	0000C000			Default heap	Priv	RW	RW		
00C50000	00001000	ex2		PE header	Img	R	RWE	Cop	
00C51000	00001000	ex2	.text	Code	Img	R E	RWE	Cop	
00C52000	00001000	ex2	.rdata	Imports	Img	R	RWE	Cop	
00C53000	00001000	ex2	.data	Data	Img	RW	RWE	Cop	
00C54000	00001000	ex2	.reloc	Relocations	Img	R	RWE	Cop	
6E3E0000	00001000	MSUCR100		PE header	Img	R	RWE	Cop	
6E3E1000	0000B2000	MSUCR100	.text	Code, imports, exports	Img	R E	RWE	Cop	
6E493000	00006000	MSUCR100	.data	Data	Img	RW	Cop	RWE	Cop
6E499000	00001000	MSUCR100	.rsrc	Resources	Img	R	RWE	Cop	
6E49A000	00005000	MSUCR100	.reloc	Relocations	Img	R	RWE	Cop	
755D0000	00001000	Mod_755D		PE header	Img	R	RWE	Cop	
755D1000	00003000				Img	R E	RWE	Cop	
755D4000	00001000				Img	RW	RWE	Cop	
755D5000	00003000				Img	R	RWE	Cop	
755E0000	00001000	Mod_755E		PE header	Img	R	RWE	Cop	
755E1000	0004D000				Img	R E	RWE	Cop	
7562E000	00005000				Img	RW	Cop	RWE	Cop
75633000	00009000				Img	R	RWE	Cop	
75640000	00001000	Mod_7564		PE header	Img	R	RWE	Cop	
75641000	00038000				Img	R E	RWE	Cop	
75679000	00002000				Img	RW	RWE	Cop	
7567B000	00004000				Img	R	RWE	Cop	
76F50000	00010000	kernel32		PE header	Img	R	RWE	Cop	
76F60000	000D0000	kernel32	.text	Code, imports, exports	Img	R E	RWE	Cop	
77030000	00010000	kernel32	.data	Data	Img	RW	Cop	RWE	Cop
77040000	00010000	kernel32	.rsrc	Resources	Img	R	RWE	Cop	
77050000	0000B000	kernel32	.reloc	Relocations	Img	R	RWE	Cop	
77810000	00001000	KERNELBASE		PE header	Img	R	RWE	Cop	
77811000	00040000	KERNELBASE	.text	Code, imports, exports	Img	R E	RWE	Cop	
77851000	00002000	KERNELBASE	.data	Data	Img	RW	RWE	Cop	
77853000	00001000	KERNELBASE	.rsrc	Resources	Img	R	RWE	Cop	
77854000	00003000	KERNELBASE	.reloc	Relocations	Img	R	RWE	Cop	
77B20000	00001000	Mod_77B2		PE header	Img	R	RWE	Cop	
77B21000	00102000				Img	R E	RWE	Cop	
77C23000	0002F000				Img	R	RWE	Cop	
77C52000	0000C000				Img	RW	Cop	RWE	Cop
77C5E000	0006B000				Img	R	RWE	Cop	
77D00000	00001000	ntdll		PE header	Img	R	RWE	Cop	
77D10000	000D6000	ntdll	.text	Code, exports	Img	R E	RWE	Cop	
77DF0000	00001000	ntdll	RT	Code	Img	R E	RWE	Cop	
77E00000	00009000	ntdll	.data	Data	Img	RW	Cop	RWE	Cop

Рис. 8.6: OllyDbg: карта памяти процесса

8.2.3. GCC: x86

В Linux всё почти также. За исключением того, что если значение `x` не определено, то эта переменная будет находится в сегменте `_bss`. В ELF этот сегмент имеет такие атрибуты:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

Ну а если сделать статическое присвоение этой переменной какого-либо значения, например, 10, то она будет находится в сегменте `_data`, это сегмент с такими атрибутами:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

8.2.4. MSVC: x64

Листинг 8.6: MSVC 2012 x64

```
_DATA SEGMENT
COMM x:DWORD
$SG2924 DB 'Enter X:', 0aH, 00H
$SG2925 DB '%d', 00H
$SG2926 DB 'You entered %d...', 0aH, 00H
_DATA ENDS

_TEXT SEGMENT
main PROC
$LN3:
    sub    rsp, 40

    lea    rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
```

8.2. ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
call    printf
lea     rdx, OFFSET FLAT:x
lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
call    scanf
mov     edx, DWORD PTR x
lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
call    printf

; возврат 0
xor     eax, eax

add     rsp, 40
ret     0
main    ENDP
_TEXT   ENDS
```

Почти такой же код как и в x86. Обратите внимание что для `scanf()` адрес переменной `x` передается при помощи инструкции `LEA`, а во второй `printf()` передается само значение переменной при помощи `MOV`. `DWORD PTR` — это часть языка ассемблера (не имеющая отношения к машинным кодам) показывающая, что тип переменной в памяти именно 32-битный, и инструкция `MOV` должна быть здесь закодирована соответственно.

8.2.5. ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:00000000 ; Segment type: Pure code
.text:00000000 AREA .text, CODE
...
.text:00000000 main
.text:00000000 PUSH {R4,LR}
.text:00000002 ADR R0, aEnterX ; "Enter X:\n"
.text:00000004 BL __2printf
.text:00000008 LDR R1, =x
.text:0000000A ADR R0, aD ; "%d"
.text:0000000C BL __0scanf
.text:00000010 LDR R0, =x
.text:00000012 LDR R1, [R0]
.text:00000014 ADR R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000016 BL __2printf
.text:0000001A MOVS R0, #0
.text:0000001C POP {R4,PC}
...
.text:00000020 aEnterX DCB "Enter X:",0xA,0 ; DATA XREF: main+2
.text:0000002A DCB 0
.text:0000002B DCB 0
.text:0000002C off_2C DCD x ; DATA XREF: main+8
.text:0000002C ; main+10
.text:00000030 aD DCB "%d",0 ; DATA XREF: main+A
.text:00000033 DCB 0
.text:00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; DATA XREF: main+14
.text:00000047 DCB 0
.text:00000047 ; .text ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048 AREA .data, DATA
.data:00000048 ; ORG 0x48
.data:00000048 EXPORT x
.data:00000048 x DCD 0xA ; DATA XREF: main+8
.data:00000048 ; main+10
.data:00000048 ; .data ends
```

Итак, переменная `x` теперь глобальная, и она расположена, почему-то, в другом сегменте, а именно сегменте данных (`.data`). Можно спросить, почему текстовые строки расположены в сегменте кода (`.text`), а `x` нельзя было разместить тут же?

Потому что эта переменная, и как следует из определения, она может меняться. И может быть, меняться часто.

Ну а текстовые строки имеют тип констант, они не будут меняться, поэтому они располагаются в сегменте `.text`.

8.2. ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

Сегмент кода иногда может быть расположен в ПЗУ микроконтроллера (не забывайте, мы сейчас имеем дело с embedded-микроэлектроникой, где дефицит памяти — обычное дело), а изменяемые переменные — в ОЗУ.

Хранить в ОЗУ неизменяемые данные, когда в наличии есть ПЗУ, не экономно.

К тому же, сегмент данных в ОЗУ с константами нужно инициализировать перед работой, ведь, после включения ОЗУ, очевидно, она содержит в себе случайную информацию.

Далее мы видим в сегменте кода хранится указатель на переменную `x` (`off_2C`) и все операции с переменной происходят через этот указатель.

Это связано с тем, что переменная `x` может быть расположена где-то довольно далеко от данного участка кода, так что её адрес нужно сохранить в непосредственной близости к этому коду.

Инструкция `LDR` в Thumb-режиме может адресовать только переменные в пределах вплоть до 1020 байт от своего местоположения.

Эта же инструкция в ARM-режиме — переменные в пределах ± 4095 байт.

Таким образом, адрес глобальной переменной `x` нужно расположить в непосредственной близости, ведь нет никакой гарантии, что компоновщик² сможет разместить саму переменную где-то рядом, она может быть даже в другом чипе памяти!

Ещё одна вещь: если переменную объявить как *const*, то компилятор Keil разместит её в сегменте `.constdata`.

Должно быть, впоследствии компоновщик и этот сегмент сможет разместить в ПЗУ вместе с сегментом кода.

8.2.6. ARM64

Листинг 8.7: Неоптимизирующий GCC 4.9.1 ARM64

```
1      .comm    x,4,4
2  .LC0:
3      .string "Enter X:"
4  .LC1:
5      .string "%d"
6  .LC2:
7      .string "You entered %d...\n"
8  f5:
9  ; сохранить FP и LR в стековом фрейме:
10     stp      x29, x30, [sp, -16]!
11 ; установить стековый фрейм (FP=SP)
12     add      x29, sp, 0
13 ; загрузить указатель на строку "Enter X:":
14     adrp     x0, .LC0
15     add      x0, x0, :lo12:.LC0
16     bl       puts
17 ; загрузить указатель на строку "%d":
18     adrp     x0, .LC1
19     add      x0, x0, :lo12:.LC1
20 ; сформировать адрес глобальной переменной x:
21     adrp     x1, x
22     add      x1, x1, :lo12:x
23     bl       __isoc99_scanf
24 ; снова сформировать адрес глобальной переменной x:
25     adrp     x0, x
26     add      x0, x0, :lo12:x
27 ; загрузить значение из памяти по этому адресу:
28     ldr      w1, [x0]
29 ; загрузить указатель на строку "You entered %d...\n":
30     adrp     x0, .LC2
31     add      x0, x0, :lo12:.LC2
32     bl       printf
33 ; возврат 0
34     mov      w0, 0
35 ; восстановить FP и LR:
36     ldp      x29, x30, [sp], 16
37     ret
```

²linker в англоязычной литературе

8.2. ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

Теперь *x* это глобальная переменная, и её адрес вычисляется при помощи пары инструкций `ADRP / ADD` (строки 21 и 25).

8.2.7. MIPS

Неинициализированная глобальная переменная

Так что теперь переменная *x* глобальная. Сделаем исполняемый файл вместо объектного и загрузим его в [IDA](#). IDA показывает присутствие переменной *x* в ELF-секции `.sbss` (помните о «Global Pointer»? [4.5.1](#) (стр. 20)), так как переменная не инициализируется в самом начале.

Листинг 8.8: Оптимизирующий GCC 4.4.5 (IDA)

```
.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10      = -0x10
.text:004006C0 var_4      = -4
.text:004006C0
; пролог функции:
.text:004006C0          lui      $gp, 0x42
.text:004006C4          addiu   $sp, -0x20
.text:004006C8          li      $gp, 0x418940
.text:004006CC          sw      $ra, 0x20+var_4($sp)
.text:004006D0          sw      $gp, 0x20+var_10($sp)
; вызов puts():
.text:004006D4          la      $t9, puts
.text:004006D8          lui      $a0, 0x40
.text:004006DC          jalr    $t9 ; puts
.text:004006E0          la      $a0, aEnterX      # "Enter X:" ; branch delay slot
; вызов scanf():
.text:004006E4          lw      $gp, 0x20+var_10($sp)
.text:004006E8          lui      $a0, 0x40
.text:004006EC          la      $t9, __isoc99_scanf
; подготовить адрес x:
.text:004006F0          la      $a1, x
.text:004006F4          jalr    $t9 ; __isoc99_scanf
.text:004006F8          la      $a0, aD          # "%d"          ; branch delay slot
; вызов printf():
.text:004006FC          lw      $gp, 0x20+var_10($sp)
.text:00400700          lui      $a0, 0x40
; взять адрес x:
.text:00400704          la      $v0, x
.text:00400708          la      $t9, printf
; загрузить значение из переменной "x" и передать его в printf() в $a1:
.text:0040070C          lw      $a1, (x - 0x41099C)($v0)
.text:00400710          jalr    $t9 ; printf
.text:00400714          la      $a0, aYouEnteredD___ # "You entered %d...\n" ; branch delay slot
        ↵ slot
; эпилог функции:
.text:00400718          lw      $ra, 0x20+var_4($sp)
.text:0040071C          move    $v0, $zero
.text:00400720          jr      $ra
.text:00400724          addiu   $sp, 0x20 ; branch delay slot

...

.sbss:0041099C # Segment type: Uninitialized
.sbss:0041099C .sbss
.sbss:0041099C .globl x
.sbss:0041099C x: .space 4
.sbss:0041099C
```

IDA уменьшает количество информации, так что сделаем также листинг используя `objdump` и добавим туда свои комментарии:

Листинг 8.9: Оптимизирующий GCC 4.4.5 (objdump)

```
1 004006c0 <main>:
2 ; пролог функции:
```

8.2. ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
3  4006c0:      3c1c0042      lui      gp,0x42
4  4006c4:      27bdf0e0      addiu    sp,sp,-32
5  4006c8:      279c8940      addiu    gp,gp,-30400
6  4006cc:      afbf001c      sw       ra,28(sp)
7  4006d0:      afbc0010      sw       gp,16(sp)
8  ; вызов puts():
9  4006d4:      8f998034      lw       t9,-32716(gp)
10 4006d8:      3c040040      lui      a0,0x40
11 4006dc:      0320f809      jalr     t9
12 4006e0:      248408f0      addiu    a0,a0,2288 ; branch delay slot
13 ; вызов scanf():
14 4006e4:      8fbc0010      lw       gp,16(sp)
15 4006e8:      3c040040      lui      a0,0x40
16 4006ec:      8f998038      lw       t9,-32712(gp)
17 ; подготовить адрес x:
18 4006f0:      8f858044      lw       a1,-32700(gp)
19 4006f4:      0320f809      jalr     t9
20 4006f8:      248408fc      addiu    a0,a0,2300 ; branch delay slot
21 ; вызов printf():
22 4006fc:      8fbc0010      lw       gp,16(sp)
23 400700:      3c040040      lui      a0,0x40
24 ; взять адрес x:
25 400704:      8f828044      lw       v0,-32700(gp)
26 400708:      8f99803c      lw       t9,-32708(gp)
27 ; загрузить значение из переменной "x" и передать его в printf() в $a1:
28 40070c:      8c450000      lw       a1,0(v0)
29 400710:      0320f809      jalr     t9
30 400714:      24840900      addiu    a0,a0,2304 ; branch delay slot
31 ; эпилог функции:
32 400718:      8fbf001c      lw       ra,28(sp)
33 40071c:      00001021      move     v0,zero
34 400720:      03e00008      jr       ra
35 400724:      27bd0020      addiu    sp,sp,32 ; branch delay slot
36 ; набор NOP-ов для выравнивания начала следующей ф-ции по 16-байтной границе:
37 400728:      00200825      move     at,at
38 40072c:      00200825      move     at,at
```

Теперь мы видим, как адрес переменной *x* берется из буфера 64KiB, используя GP и прибавление к нему отрицательного смещения (строка 18).

И даже более того: адреса трех внешних функций, используемых в нашем примере (`puts()`, `scanf()`, `printf()`) также берутся из буфера 64KiB используя GP (строки 9, 16 и 26).

GP указывает на середину буфера, так что такие смещения могут нам подсказать, что адреса всех трех функций, а также адрес переменной *x* расположены где-то в самом начале буфера. Действительно, ведь наш пример крохотный.

Ещё нужно отметить что функция заканчивается двумя **NOP**-ами (`MOVE $AT,$AT` — это холостая инструкция), чтобы выравнивать начало следующей функции по 16-байтной границе.

Инициализированная глобальная переменная

Немного изменим наш пример и сделаем, чтобы у *x* было значение по умолчанию:

```
int x=10; // значение по умолчанию
```

Теперь IDA показывает что переменная *x* располагается в секции `.data`:

Листинг 8.10: Оптимизирующий GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_10      = -0x10
.text:004006A0 var_8       = -8
.text:004006A0 var_4       = -4
.text:004006A0
.text:004006A0      lui      $gp, 0x42
.text:004006A4      addiu    $sp, -0x20
.text:004006A8      li       $gp, 0x418930
.text:004006AC      sw       $ra, 0x20+var_4($sp)
```

8.2. ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
.text:004006B0      sw      $s0, 0x20+var_8($sp)
.text:004006B4      sw      $gp, 0x20+var_10($sp)
.text:004006B8      la      $t9, puts
.text:004006BC      lui      $a0, 0x40
.text:004006C0      jalr     $t9 ; puts
.text:004006C4      la      $a0, aEnterX      # "Enter X:"
.text:004006C8      lw      $gp, 0x20+var_10($sp)
; подготовить старшую часть адреса x:
.text:004006CC      lui      $s0, 0x41
.text:004006D0      la      $t9, __isoc99_scanf
.text:004006D4      lui      $a0, 0x40
; прибавить младшую часть адреса x:
.text:004006D8      addiu    $a1, $s0, (x - 0x410000)
; теперь адрес x в $a1.
.text:004006DC      jalr     $t9 ; __isoc99_scanf
.text:004006E0      la      $a0, aD          # "%d"
.text:004006E4      lw      $gp, 0x20+var_10($sp)
; загрузить слово из памяти:
.text:004006E8      lw      $a1, x
; значение x теперь в $a1.
.text:004006EC      la      $t9, printf
.text:004006F0      lui      $a0, 0x40
.text:004006F4      jalr     $t9 ; printf
.text:004006F8      la      $a0, aYouEnteredD__ # "You entered %d...\n"
.text:004006FC      lw      $ra, 0x20+var_4($sp)
.text:00400700      move     $v0, $zero
.text:00400704      lw      $s0, 0x20+var_8($sp)
.text:00400708      jr      $ra
.text:0040070C      addiu    $sp, 0x20

...

.data:00410920      .globl x
.data:00410920 x:      .word 0xA
```

Почему не `.sdata`? Может быть, нужно было указать какую-то опцию в GCC? Тем не менее, `x` теперь в `.data`, а это уже общая память и мы можем посмотреть как происходит работа с переменными там.

Адрес переменной должен быть сформирован парой инструкций. В нашем случае это **LUI** («Load Upper Immediate» — загрузить старшие 16 бит) и **ADDIU** («Add Immediate Unsigned Word» — прибавить значение). Вот так же листинг сгенерированный `objdump`-ом для лучшего рассмотрения:

Листинг 8.11: Оптимизирующий GCC 4.4.5 (`objdump`)

```
004006a0 <main>:
 4006a0:      3c1c0042      lui      gp,0x42
 4006a4:      27bdf0e0      addiu    sp,sp,-32
 4006a8:      279c8930      addiu    gp,gp,-30416
 4006ac:      afbf001c      sw      ra,28(sp)
 4006b0:      afb00018      sw      s0,24(sp)
 4006b4:      afbc0010      sw      gp,16(sp)
 4006b8:      8f998034      lw      t9,-32716(gp)
 4006bc:      3c040040      lui      a0,0x40
 4006c0:      0320f809      jalr     t9
 4006c4:      248408d0      addiu    a0,a0,2256
 4006c8:      8fbc0010      lw      gp,16(sp)
; подготовить старшую часть адреса x:
 4006cc:      3c100041      lui      s0,0x41
 4006d0:      8f998038      lw      t9,-32712(gp)
 4006d4:      3c040040      lui      a0,0x40
; прибавить младшую часть адреса x:
 4006d8:      26050920      addiu    a1,s0,2336
; теперь адрес x в $a1.
 4006dc:      0320f809      jalr     t9
 4006e0:      248408dc      addiu    a0,a0,2268
 4006e4:      8fbc0010      lw      gp,16(sp)
; старшая часть адреса x всё еще в $s0.
; прибавить младшую часть к ней и загрузить слово из памяти:
 4006e8:      8e050920      lw      a1,2336(s0)
; значение x теперь в $a1.
```

8.3. ПРОВЕРКА РЕЗУЛЬТАТА SCANF()

4006ec:	8f99803c	lw	t9, -32708(gp)
4006f0:	3c040040	lui	a0, 0x40
4006f4:	0320f809	jalr	t9
4006f8:	248408e0	addiu	a0, a0, 2272
4006fc:	8fbf001c	lw	ra, 28(sp)
400700:	00001021	move	v0, zero
400704:	8fb00018	lw	s0, 24(sp)
400708:	03e00008	jr	ra
40070c:	27bd0020	addiu	sp, sp, 32

Адрес формируется используя `LUI` и `ADDIU`, но старшая часть адреса всё ещё в регистре `$S0`, и можно закодировать смещение в инструкции `LW` («Load Word»), так что одной `LW` достаточно для загрузки значения из переменной и передачи его в `printf()`. Регистры хранящие временные данные имеют префикс `T`, но здесь есть также регистры с префиксом `S`, содержимое которых должно быть сохранено в других функциях (т.е. «saved»).

Вот почему `$S0` был установлен по адресу `0x4006cc` и затем был использован по адресу `0x4006e8` после вызова `scanf()`.

Функция `scanf()` не изменяет это значение.

8.3. Проверка результата scanf()

Как уже было упомянуто, использовать `scanf()` в наше время слегка старомодно. Но если уж пришлось, нужно хотя бы проверять, сработал ли `scanf()` правильно или пользователь ввел вместо числа что-то другое, что `scanf()` не смог трактовать как число.

```
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    if (scanf ("%d", &x)==1)
        printf ("You entered %d...\n", x);
    else
        printf ("What you entered? Huh?\n");

    return 0;
};
```

По стандарту, `scanf()`³ возвращает количество успешно полученных значений.

В нашем случае, если всё успешно и пользователь ввел таки некое число, `scanf()` вернет 1. А если нет, то 0 (или `EOF`⁴).

Добавим код, проверяющий результат `scanf()` и в случае ошибки он сообщает пользователю что-то другое.

Это работает предсказуемо:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

8.3.1. MSVC: x86

Вот что выходит на ассемблере (MSVC 2010):

³scanf, wscanf: [MSDN](#)

⁴End of file (конец файла)

```

    lea     eax, DWORD PTR _x$[ebp]
    push    eax
    push    OFFSET $SG3833 ; '%d', 00H
    call    _scanf
    add     esp, 8
    cmp     eax, 1
    jne     SHORT $LN2@main
    mov     ecx, DWORD PTR _x$[ebp]
    push    ecx
    push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
    call    _printf
    add     esp, 8
    jmp     SHORT $LN1@main
$LN2@main:
    push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
    call    _printf
    add     esp, 4
$LN1@main:
    xor     eax, eax

```

Для того чтобы вызывающая функция имела доступ к результату вызываемой функции, вызываемая функция (в нашем случае `scanf()`) оставляет это значение в регистре `EAX`.

Мы проверяем его инструкцией `CMP EAX, 1` (*CoMPare*), то есть сравниваем значение в `EAX` с 1.

Следующий за инструкцией `CMP`: условный переход `JNE`. Это означает *Jump if Not Equal*, то есть условный переход *если не равно*.

Итак, если `EAX` не равен 1, то `JNE` заставит `CPU` перейти по адресу указанном в операнде `JNE`, у нас это `$LN2@main`. Передав управление по этому адресу, `CPU` начнет исполнять вызов `printf()` с аргументом `What you entered? Huh?`. Но если всё нормально, перехода не случится и исполнится другой `printf()` с двумя аргументами: `'You entered %d...'` и значением переменной `x`.

Для того чтобы после этого вызова не исполнился сразу второй вызов `printf()`, после него есть инструкция `JMP`, безусловный переход, который отправит процессор на место после второго `printf()` и перед инструкцией `XOR EAX, EAX`, которая реализует `return 0`.

Итак, можно сказать что в подавляющих случаях сравнение какой-либо переменной с чем-то другим происходит при помощи пары инструкций `CMP` и `Jcc`, где *cc* это *condition code*. `CMP` сравнивает два значения и выставляет флаги процессора⁵. `Jcc` проверяет нужные ему флаги и выполняет переход по указанному адресу (или не выполняет).

Но на самом деле, как это не парадоксально поначалу звучит, `CMP` это почти то же самое что и инструкция `SUB`, которая отнимает числа одно от другого. Все арифметические инструкции также выставляют флаги в соответствии с результатом, не только `CMP`. Если мы сравним 1 и 1, от единицы отнимется единица, получится 0, и выставится флаг `ZF` (*zero flag*), означающий, что последний полученный результат был 0. Ни при каких других значениях `EAX`, флаг `ZF` не может быть выставлен, кроме тех, когда операнды равны друг другу. Инструкция `JNE` проверяет только флаг `ZF`, и совершает переход только если флаг не поднят. Фактически, `JNE` это синоним инструкции `JNZ` (*Jump if Not Zero*). Ассемблер транслирует обе инструкции в один и тот же опкод. Таким образом, можно `CMP` заменить на `SUB` и всё будет работать также, но разница в том, что `SUB` всё-таки испортит значение в первом операнде. `CMP` это *SUB без сохранения результата, но изменяющая флаги*.

8.3.2. MSVC: x86: IDA

Наверное, уже пора делать первые попытки анализа кода в `IDA`. Кстати, начинающим полезно компилировать в `MSVC` с ключом `/MD`, что означает что все эти стандартные функции не будут скомпонованы с исполняемым файлом, а будут импортироваться из файла `MSVCR*.DLL`. Так будет легче увидеть, где какая стандартная функция используется.

Анализируя код в `IDA`, очень полезно делать пометки для себя (и других). Например, разбирая этот пример, мы сразу видим, что `JNZ` срабатывает в случае ошибки. Можно навести курсор на эту метку, нажать «п» и переименовать метку в «error». Ещё одну метку — в «exit». Вот как у меня получилось в итоге:

```

.text:00401000 _main proc near
.text:00401000

```

⁵См. также о флагах x86-процессора: [wikipedia](https://ru.wikipedia.org/wiki/Список_флагов_процессора_x86).

8.3. ПРОВЕРКА РЕЗУЛЬТАТА SCANF()

```
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc  = dword ptr  8
.text:00401000 argv  = dword ptr  0Ch
.text:00401000 envp  = dword ptr  10h
.text:00401000
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      push    ecx
.text:00401004      push    offset Format ; "Enter X:\n"
.text:00401009      call    ds:printf
.text:0040100F      add     esp, 4
.text:00401012      lea     eax, [ebp+var_4]
.text:00401015      push    eax
.text:00401016      push    offset aD ; "%d"
.text:0040101B      call    ds:scanf
.text:00401021      add     esp, 8
.text:00401024      cmp     eax, 1
.text:00401027      jnz     short error
.text:00401029      mov     ecx, [ebp+var_4]
.text:0040102C      push    ecx
.text:0040102D      push    offset aYou ; "You entered %d...\n"
.text:00401032      call    ds:printf
.text:00401038      add     esp, 8
.text:0040103B      jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D      push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042      call    ds:printf
.text:00401048      add     esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B      xor     eax, eax
.text:0040104D      mov     esp, ebp
.text:0040104F      pop     ebp
.text:00401050      retn
.text:00401050 _main endp
```

Так понимать код становится чуть легче. Впрочем, меру нужно знать во всем и комментировать каждую инструкцию не стоит.

В IDA также можно скрывать части функций: нужно выделить скрываемую часть, нажать «-» на цифровой клавиатуре и ввести текст.

Скроем две части и придумаем им названия:

```
.text:00401000 _text segment para public 'CODE' use32
.text:00401000      assume cs:_text
.text:00401000      ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024      cmp     eax, 1
.text:00401027      jnz     short error
.text:00401029 ; print result
.text:0040103B      jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D      push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042      call    ds:printf
.text:00401048      add     esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B      xor     eax, eax
.text:0040104D      mov     esp, ebp
.text:0040104F      pop     ebp
.text:00401050      retn
.text:00401050 _main endp
```

Раскрывать скрытые части функций можно при помощи «+» на цифровой клавиатуре.

Нажав «пробел», мы увидим, как IDA может представить функцию в виде графа:

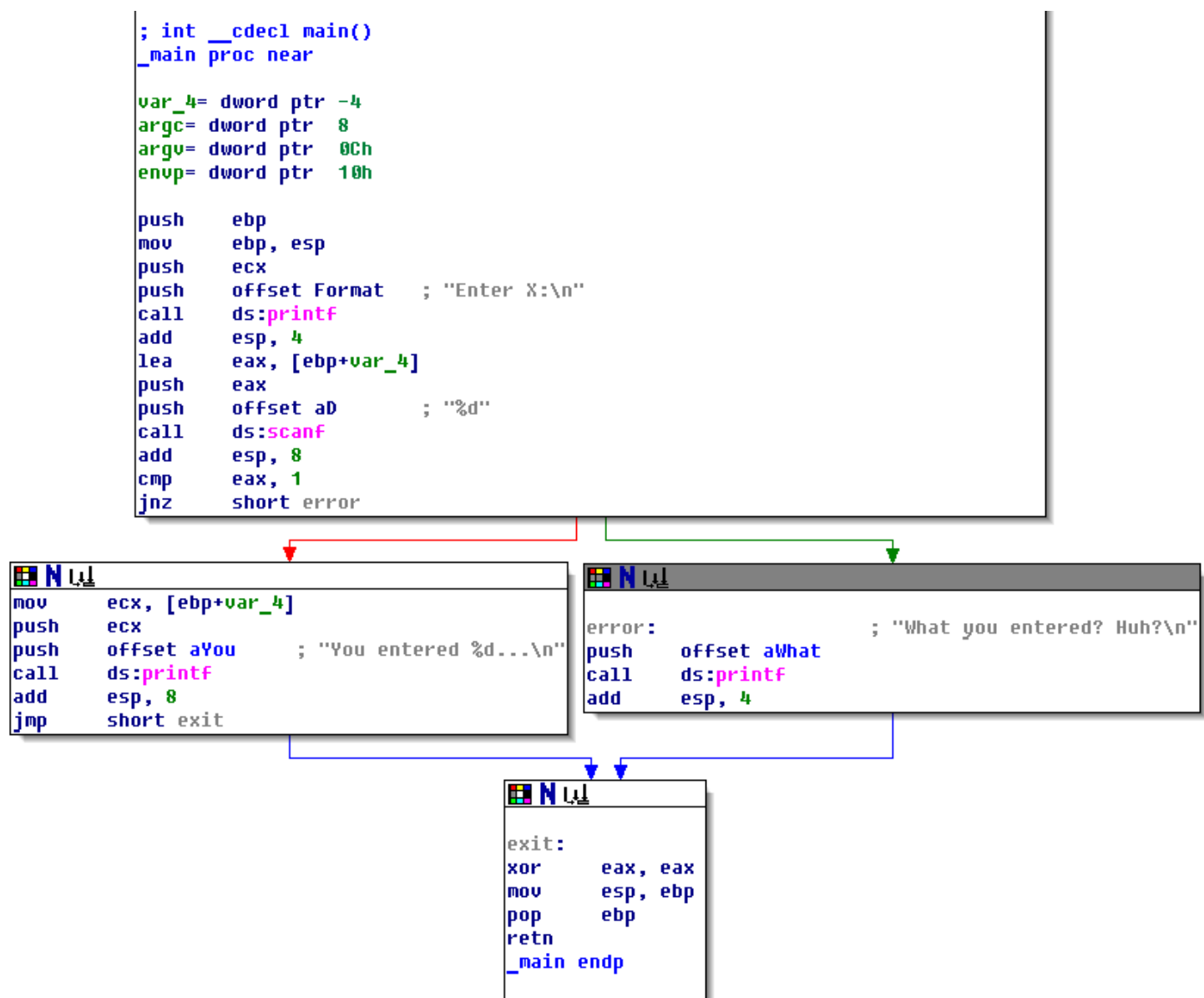


Рис. 8.7: Отображение функции в IDA в виде графа

После каждого условного перехода видны две стрелки: зеленая и красная. Зеленая ведет к тому блоку, который исполнится если переход сработает, а красная — если не сработает.

8.3. ПРОВЕРКА РЕЗУЛЬТАТА SCANF()

В этом режиме также можно сворачивать узлы и давать им названия («group nodes»). Сделаем это для трех блоков:

```
; int __cdecl main()
_main proc near

var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h
```

```
push    ebp
mov     ebp, esp
push    ecx
push    offset Format    ; "Enter X:\n"
call    ds:printf
add     esp, 4
lea     eax, [ebp+var_4]
push    eax
push    offset aD        ; "%d"
call    ds:scanf
add     esp, 8
cmp     eax, 1
jnz     short error
```

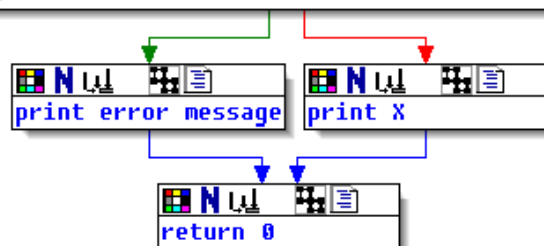
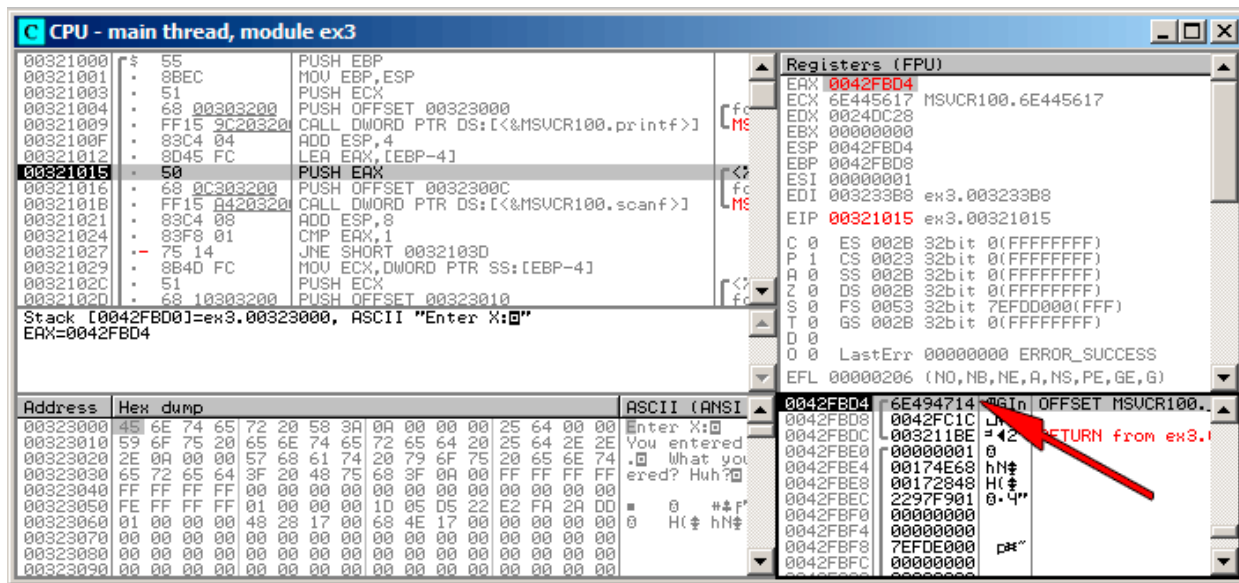


Рис. 8.8: Отображение в IDA в виде графа с тремя свернутыми блоками

Всё это очень полезно делать. Вообще, очень важная часть работы реверсера (да и любого исследователя) состоит в том, чтобы уменьшать количество имеющейся информации.

8.3.3. MSVC: x86 + OllyDbg

Попробуем в OllyDbg немного хакнуть программу и сделать вид, что `scanf()` срабатывает всегда без ошибок. Когда в `scanf()` передается адрес локальной переменной, изначально в этой переменной находится некий мусор. В данном случае это `0x6E494714`:

Рис. 8.9: OllyDbg: передача адреса переменной в `scanf()`

8.3. ПРОВЕРКА РЕЗУЛЬТАТА SCANF()

Когда `scanf()` запускается, вводим в консоли что-то непохожее на число, например «asdasd». `scanf()` заканчивается с 0 в `EAX`, что означает, что произошла ошибка:

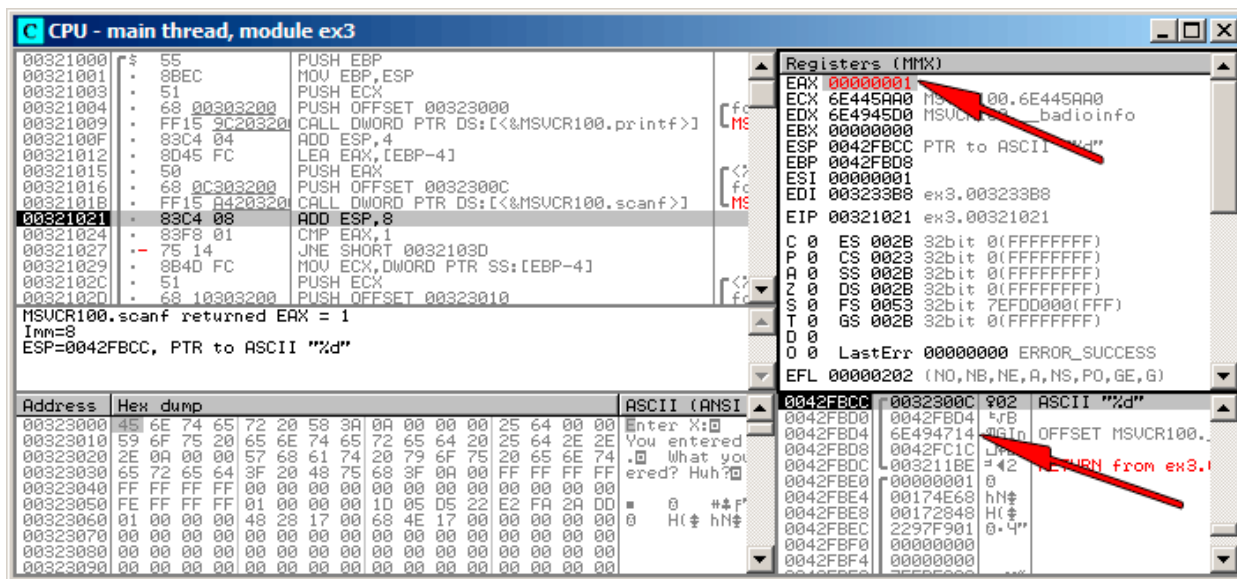


Рис. 8.10: OllyDbg: `scanf()` закончился с ошибкой

Вместе с этим мы можем посмотреть на локальную переменную в стеке — она не изменилась. Действительно, ведь что туда записала бы функция `scanf()`? Она не делала ничего кроме возвращения нуля. Попробуем ещё немного «хакнуть» нашу программу. Щелкнем правой кнопкой на `EAX`, там, в числе опций, будет также «Set to 1». Это нам и нужно.

В `EAX` теперь 1, последующая проверка пройдет как надо, и `printf()` выведет значение переменной из стека.

Запускаем (F9) и видим в консоли следующее:

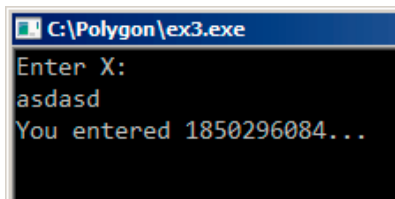


Рис. 8.11: консоль

Действительно, 1850296084 это десятичное представление числа в стеке (0x6E494714)!

Исполняемый файл скомпилирован с импортированием функций из `MSVCR*.DLL` (т.е. с опцией `/MD`)⁶, поэтому мы можем отыскать функцию `main()` в самом начале секции `.text`. Откроем исполняемый файл в Hiew, найдем самое начало секции `.text` (Enter, F8, F6, Enter, Enter).

Мы увидим следующее:

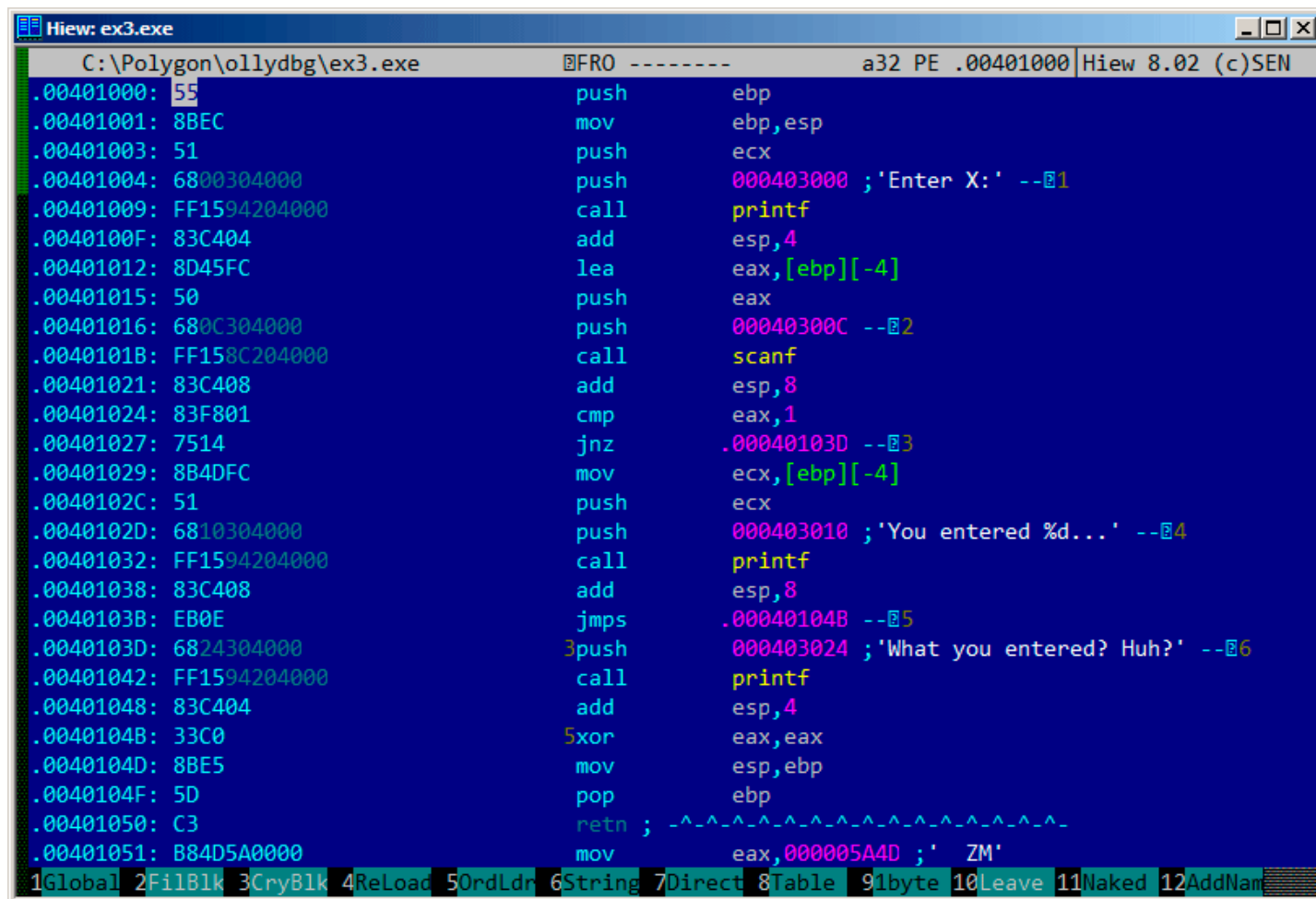


Рис. 8.12: Hiew: функция `main()`

Нiew находит [ASCIIZ⁷](#)-строки и показывает их, также как и имена импортируемых функций.

⁶то, что ещё называют «dynamic linking»

⁷ ASCII Zero (ASCII-строка заканчивающаяся нулем)

8.3. ПРОВЕРКА РЕЗУЛЬТАТА SCANF()

Переведите курсор на адрес `.00401027` (с инструкцией `JNZ`, которую мы хотим заблокировать), нажмите F3, затем наберите «9090» (что означает два `NOP`-а):

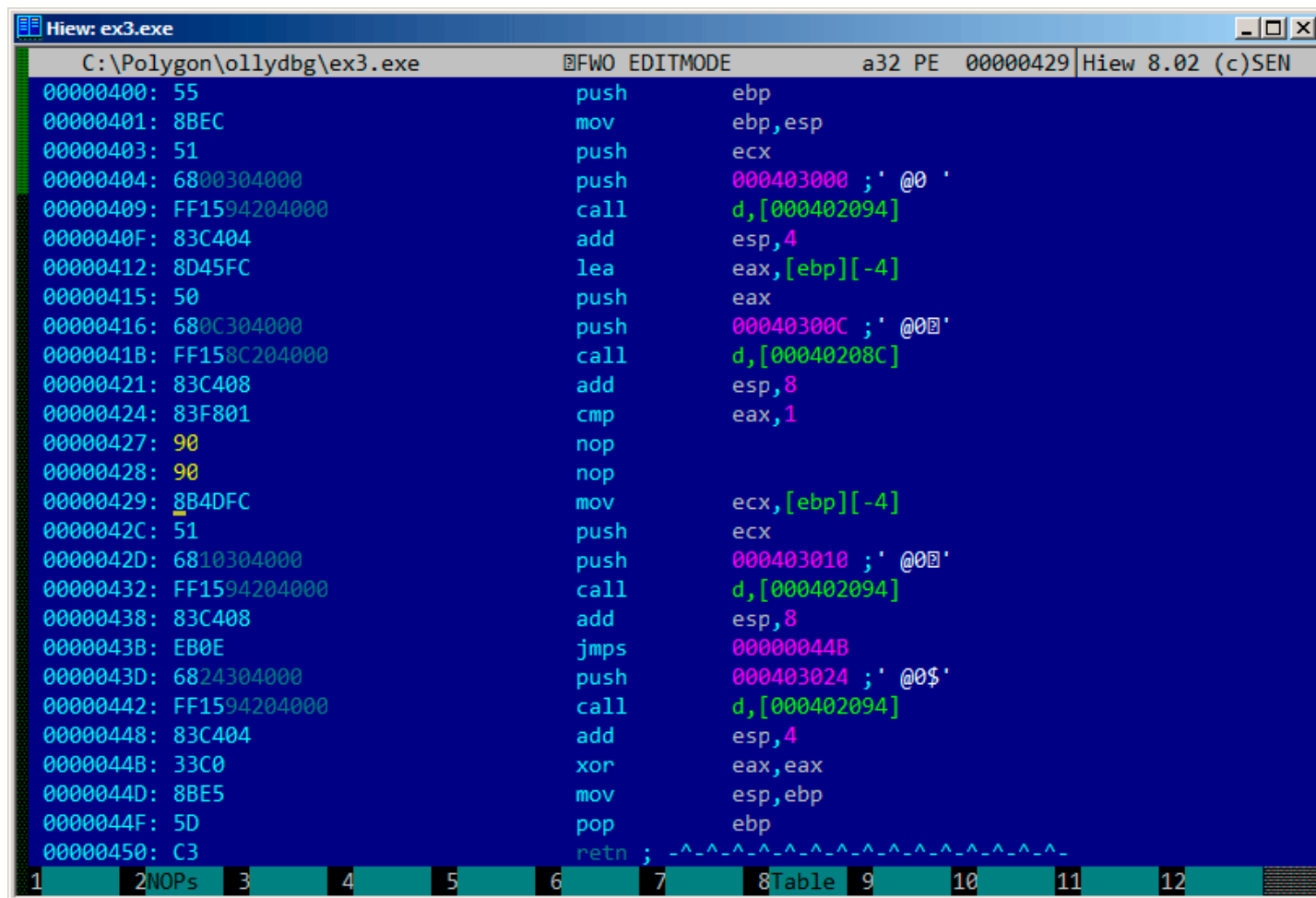


Рис. 8.13: Hiew: замена `JNZ` на два `NOP`-а

Затем F9 (update). Теперь исполняемый файл записан на диск. Он будет вести себя так, как нам надо.

Два `NOP`-а, возможно, не так эстетично, как могло бы быть. Другой способ изменить инструкцию это записать 0 во второй байт опкода (смещение перехода), так что `JNZ` всегда будет переходить на следующую инструкцию.

Можно изменить и наоборот: первый байт заменить на `EB`, второй байт (смещение перехода) не трогать. Получится всегда срабатывающий безусловный переход. Теперь сообщение об ошибке будет выдаваться всегда, даже если мы ввели число.

8.3.5. MSVC: x64

Так как здесь мы работаем с переменными типа `int`, а они в x86-64 остались 32-битными, то мы здесь видим, как продолжают использоваться регистры с префиксом `E-`. Но для работы с указателями, конечно, используются 64-битные части регистров с префиксом `R-`.

Листинг 8.12: MSVC 2012 x64

```
_DATA SEGMENT
$SG2924 DB 'Enter X:', 0aH, 00H
$SG2926 DB '%d', 00H
$SG2927 DB 'You entered %d...', 0aH, 00H
$SG2929 DB 'What you entered? Huh?', 0aH, 00H
_DATA ENDS

_TEXT SEGMENT
x$ = 32
main PROC
```

8.3. ПРОВЕРКА РЕЗУЛЬТАТА SCANF()

```
$LN5:
    sub     rsp, 56
    lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
    call    printf
    lea     rdx, QWORD PTR x$[rsp]
    lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
    call    scanf
    cmp     eax, 1
    jne     SHORT $LN2@main
    mov     edx, DWORD PTR x$[rsp]
    lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
    call    printf
    jmp     SHORT $LN1@main
$LN2@main:
    lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
    call    printf
$LN1@main:
    ; возврат 0
    xor     eax, eax
    add     rsp, 56
    ret     0
main      ENDP
_TEXT     ENDS
END
```

8.3.6. ARM

ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 8.13: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
var_8      = -8

    PUSH    {R3,LR}
    ADR     R0, aEnterX      ; "Enter X:\n"
    BL      __2printf
    MOV     R1, SP
    ADR     R0, aD           ; "%d"
    BL      __0scanf
    CMP     R0, #1
    BEQ     loc_1E
    ADR     R0, aWhatYouEntered ; "What you entered? Huh?\n"
    BL      __2printf

loc_1A      ; CODE XREF: main+26
    MOVS    R0, #0
    POP     {R3,PC}

loc_1E      ; CODE XREF: main+12
    LDR     R1, [SP,#8+var_8]
    ADR     R0, aYouEnteredD__ ; "You entered %d...\n"
    BL      __2printf
    B       loc_1A
```

Здесь для нас есть новые инструкции: **CMP** и **BEQ**⁸.

CMP аналогична той что в x86: она отнимает один аргумент от второго и сохраняет флаги.

BEQ совершает переход по другому адресу, если операнды при сравнении были равны, либо если результат последнего вычисления был 0, либо если флаг Z равен 1. То же что и **JZ** в x86.

Всё остальное просто: исполнение разветвляется на две ветки, затем они сходятся там, где в **R0** записывается 0 как возвращаемое из функции значение и происходит выход из функции.

⁸(PowerPC, ARM) Branch if Equal

Листинг 8.14: Неоптимизирующий GCC 4.9.1 ARM64

```

1  .LC0:
2      .string "Enter X:"
3  .LC1:
4      .string "%d"
5  .LC2:
6      .string "You entered %d...\n"
7  .LC3:
8      .string "What you entered? Huh?"
9  f6:
10 ; сохранить FP и LR в стековом фрейме:
11     str    x29, x30, [sp, -32]!
12 ; установить стековый фрейм (FP=SP)
13     add    x29, sp, 0
14 ; загрузить указатель на строку "Enter X:"
15     adrp   x0, .LC0
16     add    x0, x0, :lo12:.LC0
17     bl     puts
18 ; загрузить указатель на строку "%d":
19     adrp   x0, .LC1
20     add    x0, x0, :lo12:.LC1
21 ; вычислить адрес переменной x в локальном стеке
22     add    x1, x29, 28
23     bl     __isoc99_scanf
24 ; scanf() возвращает результат в W0.
25 ; проверяем его:
26     cmp    w0, 1
27 ; BNE это Branch if Not Equal (переход, если не равно)
28 ; так что если W0<>0, произойдет переход на L2
29     bne    .L2
30 ; в этот момент W0=1, означая, что ошибки не было
31 ; загрузить значение x из локального стека
32     ldr     w1, [x29,28]
33 ; загрузить указатель на строку "You entered %d...\n":
34     adrp   x0, .LC2
35     add    x0, x0, :lo12:.LC2
36     bl     printf
37 ; пропустить код, печатающий строку "What you entered? Huh?" :
38     b      .L3
39 .L2:
40 ; загрузить указатель на строку "What you entered? Huh?" :
41     adrp   x0, .LC3
42     add    x0, x0, :lo12:.LC3
43     bl     puts
44 .L3:
45 ; возврат 0
46     mov    w0, 0
47 ; восстановить FP и LR:
48     ldp    x29, x30, [sp], 32
49     ret

```

Исполнение здесь разветвляется, используя пару инструкций **CMP / BNE** (Branch if Not Equal: переход если не равно).

8.3.7. MIPS

Листинг 8.15: Оптимизирующий GCC 4.4.5 (IDA)

```

.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18          = -0x18
.text:004006A0 var_10          = -0x10
.text:004006A0 var_4           = -4
.text:004006A0
.text:004006A0                lui     $gp, 0x42
.text:004006A4                addiu  $sp, -0x28

```

8.4. УПРАЖНЕНИЕ

```
.text:004006A8      li      $gp, 0x418960
.text:004006AC      sw      $ra, 0x28+var_4($sp)
.text:004006B0      sw      $gp, 0x28+var_18($sp)
.text:004006B4      la      $t9, puts
.text:004006B8      lui     $a0, 0x40
.text:004006BC      jalr    $t9 ; puts
.text:004006C0      la      $a0, aEnterX      # "Enter X:"
.text:004006C4      lw      $gp, 0x28+var_18($sp)
.text:004006C8      lui     $a0, 0x40
.text:004006CC      la      $t9, __isoc99_scanf
.text:004006D0      la      $a0, aD           # "%d"
.text:004006D4      jalr    $t9 ; __isoc99_scanf
.text:004006D8      addiu   $a1, $sp, 0x28+var_10 # branch delay slot
.text:004006DC      li      $v1, 1
.text:004006E0      lw      $gp, 0x28+var_18($sp)
.text:004006E4      beq     $v0, $v1, loc_40070C
.text:004006E8      or      $at, $zero      # branch delay slot, NOP
.text:004006EC      la      $t9, puts
.text:004006F0      lui     $a0, 0x40
.text:004006F4      jalr    $t9 ; puts
.text:004006F8      la      $a0, aWhatYouEntered # "What you entered? Huh?"
.text:004006FC      lw      $ra, 0x28+var_4($sp)
.text:00400700      move    $v0, $zero
.text:00400704      jr      $ra
.text:00400708      addiu   $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C      la      $t9, printf
.text:00400710      lw      $a1, 0x28+var_10($sp)
.text:00400714      lui     $a0, 0x40
.text:00400718      jalr    $t9 ; printf
.text:0040071C      la      $a0, aYouEnteredD__ # "You entered %d...\n"
.text:00400720      lw      $ra, 0x28+var_4($sp)
.text:00400724      move    $v0, $zero
.text:00400728      jr      $ra
.text:0040072C      addiu   $sp, 0x28
```

`scanf()` возвращает результат своей работы в регистре `$V0` и он проверяется по адресу `0x004006E4` сравнивая значения в `$V0` и `$V1` (1 записан в `$V1` ранее, на `0x004006DC`). `BEQ` означает «Branch Equal» (переход если равно). Если значения равны (т.е. в случае успеха), произойдет переход по адресу `0x0040070C`.

8.3.8. Упражнение

Как мы можем увидеть, инструкцию `JNE / JNZ` можно вполне заменить на `JE / JZ` или наоборот (или `BNE` на `BEQ` и наоборот). Но при этом ещё нужно переставить базовые блоки местами. Попробуйте сделать это в каком-нибудь примере.

8.4. Упражнение

- <http://challenges.re/53>

Глава 9

Доступ к переданным аргументам

Как мы уже успели заметить, вызывающая функция передает аргументы для вызываемой через стек. А как вызываемая функция получает к ним доступ?

Листинг 9.1: простой пример

```
#include <stdio.h>

int f (int a, int b, int c)
{
    return a*b+c;
};

int main()
{
    printf ("%d\n", f(1, 2, 3));
    return 0;
};
```

9.1. x86

9.1.1. MSVC

Рассмотрим пример, скомпилированный в (MSVC 2010 Express):

Листинг 9.2: MSVC 2010 Express

```
_TEXT    SEGMENT
_a$ = 8                                     ; size = 4
_b$ = 12                                    ; size = 4
_c$ = 16                                    ; size = 4
_f      PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    imul    eax, DWORD PTR _b$[ebp]
    add     eax, DWORD PTR _c$[ebp]
    pop     ebp
    ret     0
_f      ENDP

_main    PROC
    push    ebp
    mov     ebp, esp
    push    3 ; третий аргумент
    push    2 ; второй аргумент
    push    1 ; первый аргумент
    call    _f
    add     esp, 12
    push    eax
    push    OFFSET $SG2463 ; '%d', 0aH, 00H
```

9.1. X86

```
call    _printf
add     esp, 8
; возврат 0
xor     eax, eax
pop     ebp
ret     0
_main   ENDP
```

Итак, здесь видно: в функции `main()` заталкиваются три числа в стек и вызывается функция `f(int,int,int)`.

Внутри `f()` доступ к аргументам, также как и к локальным переменным, происходит через макросы: `_a$ = 8`, но разница в том, что эти смещения со знаком *плюс*, таким образом если прибавить макрос `_a$` к указателю на `EBP`, то адресуется *внешняя* часть *фрейма* стека относительно `EBP`.

Далее всё более-менее просто: значение *a* помещается в `EAX`. Далее `EAX` умножается при помощи инструкции `IMUL` на то, что лежит в `_b`, и в `EAX` остается *произведение* этих двух значений.

Далее к регистру `EAX` прибавляется то, что лежит в `_c`.

Значение из `EAX` никуда не нужно перекладывать, оно уже лежит где надо. Возвращаем управление вызываемой функции — она возьмет значение из `EAX` и отправит его в `printf()`.

9.1.2. MSVC + OllyDbg

Проиллюстрируем всё это в OllyDbg. Когда мы протрассируем до первой инструкции в `f()`, которая использует какой-то из аргументов (первый), мы увидим, что `EBP` указывает на *фрейм стека*. Он выделен красным прямоугольником.

Самый первый элемент *фрейма стека* — это сохраненное значение `EBP`, затем `RA`. Третий элемент это первый аргумент функции, затем второй аргумент и третий.

Для доступа к первому аргументу функции нужно прибавить к `EBP` 8 (2 32-битных слова).

OllyDbg в курсе этого, так что он добавил комментарии к элементам стека вроде «RETURN from» и «Arg1 = ...», и т.д.

Н.В.: аргументы функции являются членами фрейма стека вызывающей функции, а не текущей. Поэтому OllyDbg отметил элементы «Arg» как члены другого фрейма стека.

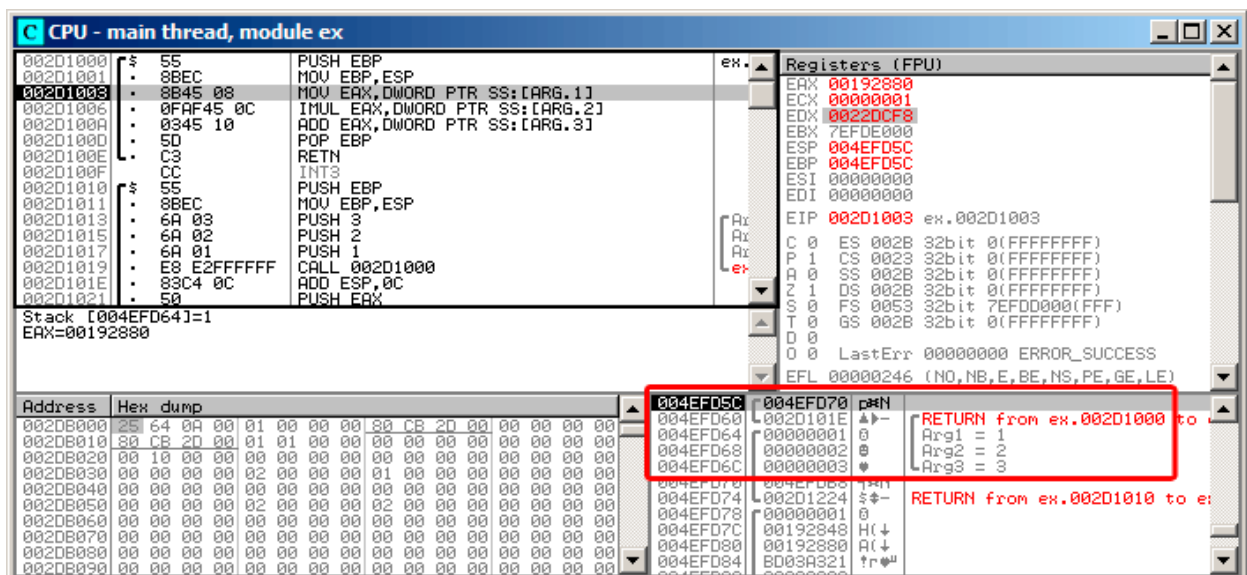


Рис. 9.1: OllyDbg: внутри функции `f()`

9.1.3. GCC

Скомпилируем то же в GCC 4.4.1 и посмотрим результат в `IDA`:

```

f                public f
                proc near

arg_0            = dword ptr 8
arg_4            = dword ptr 0Ch
arg_8            = dword ptr 10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0] ; первый аргумент
                imul    eax, [ebp+arg_4] ; второй аргумент
                add     eax, [ebp+arg_8] ; третий аргумент
                pop     ebp
                retn

f                endp

                public main
                proc near

main            = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_8], 3 ; третий аргумент
                mov     [esp+10h+var_C], 2 ; второй аргумент
                mov     [esp+10h+var_10], 1 ; первый аргумент
                call    f
                mov     edx, offset aD ; "%d\n"
                mov     [esp+10h+var_C], eax
                mov     [esp+10h+var_10], edx
                call    _printf
                mov     eax, 0
                leave
                retn

main            endp

```

Практически то же самое, если не считать мелких отличий описанных ранее.

После вызова обеих функций [указатель стека](#) не возвращается назад, потому что предпоследняя инструкция `LEAVE` (A.6.2 (стр. 917)) делает это за один раз, в конце исполнения.

9.2. x64

В x86-64 всё немного иначе, здесь аргументы функции (4 или 6) передаются через регистры, а [callee](#) из читает их из регистров, а не из стека.

9.2.1. MSVC

Оптимизирующий MSVC:

Листинг 9.4: Оптимизирующий MSVC 2012 x64

```

$SG2997 DB      '%d', 0aH, 00H

main PROC
    sub     rsp, 40
    mov     edx, 2
    lea     r8d, QWORD PTR [rdx+1] ; R8D=3
    lea     ecx, QWORD PTR [rdx-1] ; ECX=1
    call    f
    lea     rcx, OFFSET FLAT:$SG2997 ; '%d'

```

9.2. X64

```
    mov     edx, eax
    call    printf
    xor     eax, eax
    add     rsp, 40
    ret     0
main      ENDP

f         PROC
; ECX – первый аргумент
; EDX – второй аргумент
; R8D – третий аргумент
    imul    ecx, edx
    lea     eax, DWORD PTR [r8+rcx]
    ret     0
f         ENDP
```

Как видно, очень компактная функция `f()` берет аргументы прямо из регистров.

Инструкция `LEA` используется здесь для сложения чисел. Должно быть компилятор посчитал, что это будет эффективнее использования `ADD`.

В самой `main()` `LEA` также используется для подготовки первого и третьего аргумента: должно быть, компилятор решил, что `LEA` будет работать здесь быстрее, чем загрузка значения в регистр при помощи `MOV`.

Попробуем посмотреть вывод неоптимизирующего MSVC:

Листинг 9.5: MSVC 2012 x64

```
f         proc near

; shadow space:
arg_0     = dword ptr  8
arg_8     = dword ptr 10h
arg_10    = dword ptr 18h

; ECX – первый аргумент
; EDX – второй аргумент
; R8D – третий аргумент
    mov     [rsp+arg_10], r8d
    mov     [rsp+arg_8], edx
    mov     [rsp+arg_0], ecx
    mov     eax, [rsp+arg_0]
    imul    eax, [rsp+arg_8]
    add     eax, [rsp+arg_10]
    retn

f         endp

main      proc near
    sub     rsp, 28h
    mov     r8d, 3 ; третий аргумент
    mov     edx, 2 ; второй аргумент
    mov     ecx, 1 ; первый аргумент
    call    f
    mov     edx, eax
    lea     rcx, $SG2931 ; "%d\n"
    call    printf

; возврат 0
    xor     eax, eax
    add     rsp, 28h
    retn

main      endp
```

Немного путаннее: все 3 аргумента из регистров зачем-то сохраняются в стеке.

Это называется «shadow space»¹: каждая функция в Win64 может (хотя и не обязана) сохранять значения 4-х регистров там.

¹[MSDN](#)

9.2. X64

Это делается по крайней мере из-за двух причин: 1) в большой функции отвести целый регистр (а тем более 4 регистра) для входного аргумента слишком расточительно, так что к нему будет обращение через стек;

2) отладчик всегда знает, где найти аргументы функции в момент останова ².

Так что, какие-то большие функции могут сохранять входные аргументы в «shadow space» для использования в будущем, а небольшие функции, как наша, могут этого и не делать.

Место в стеке для «shadow space» выделяет именно [caller](#).

9.2.2. GCC

Оптимизирующий GCC также делает понятный код:

Листинг 9.6: Оптимизирующий GCC 4.4.6 x64

```
f:
    ; EDI – первый аргумент
    ; ESI – второй аргумент
    ; EDX – третий аргумент
    imul    esi, edi
    lea     eax, [rdx+rsi]
    ret

main:
    sub     rsp, 8
    mov     edx, 3
    mov     esi, 2
    mov     edi, 1
    call    f
    mov     edi, OFFSET FLAT:.LC0 ; "%d\n"
    mov     esi, eax
    xor     eax, eax ; количество переданных векторных регистров
    call    printf
    xor     eax, eax
    add     rsp, 8
    ret
```

Неоптимизирующий GCC:

Листинг 9.7: GCC 4.4.6 x64

```
f:
    ; EDI – первый аргумент
    ; ESI – второй аргумент
    ; EDX – третий аргумент
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     DWORD PTR [rbp-8], esi
    mov     DWORD PTR [rbp-12], edx
    mov     eax, DWORD PTR [rbp-4]
    imul    eax, DWORD PTR [rbp-8]
    add     eax, DWORD PTR [rbp-12]
    leave
    ret

main:
    push    rbp
    mov     rbp, rsp
    mov     edx, 3
    mov     esi, 2
    mov     edi, 1
    call    f
    mov     edx, eax
    mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
    mov     esi, edx
    mov     rdi, rax
    mov     eax, 0 ; количество переданных векторных регистров
    call    printf
```

²MSDN

9.3. ARM

```
mov    eax, 0
leave
ret
```

В соглашении о вызовах System V *NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]³) нет «shadow space», но *callee* тоже иногда должен сохранять где-то аргументы, потому что, опять же, регистров может и не хватить на все действия. Что мы здесь и видим.

9.2.3. GCC: uint64_t вместо int

Наш пример работал с 32-битным *int*, поэтому использовались 32-битные части регистров с префиксом **E-**.

Его можно немного переделать, чтобы он заработал с 64-битными значениями:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b*c;
};

int main()
{
    printf ("%lld\n", f(0x1122334455667788,
                        0x1111111122222222,
                        0x3333333344444444));
    return 0;
};
```

Листинг 9.8: Оптимизирующий GCC 4.4.6 x64

```
f      proc near
      imul    rsi, rdi
      lea     rax, [rdx+rsi]
      retn
f      endp

main   proc near
      sub     rsp, 8
      mov     rdx, 3333333344444444h ; третий аргумент
      mov     rsi, 1111111122222222h ; второй аргумент
      mov     rdi, 1122334455667788h ; первый аргумент
      call    f
      mov     edi, offset format ; "%lld\n"
      mov     rsi, rax
      xor     eax, eax ; количество переданных векторных регистров
      call    _printf
      xor     eax, eax
      add     rsp, 8
      retn
main   endp
```

Собственно, всё то же самое, только используются регистры *целиком*, с префиксом **R-**.

9.3. ARM

9.3.1. Неоптимизирующий Keil 6/2013 (Режим ARM)

```
.text:000000A4 00 30 A0 E1      MOV     R3, R0
.text:000000A8 93 21 20 E0      MLA     R0, R3, R1, R2
.text:000000AC 1E FF 2F E1      BX      LR
...
```

³Также доступно здесь: <http://x86-64.org/documentation/abi.pdf>

9.3. ARM

```
.text:000000B0          main
.text:000000B0 10 40 2D E9      STMFD    SP!, {R4,LR}
.text:000000B4 03 20 A0 E3      MOV     R2, #3
.text:000000B8 02 10 A0 E3      MOV     R1, #2
.text:000000BC 01 00 A0 E3      MOV     R0, #1
.text:000000C0 F7 FF FF EB      BL      f
.text:000000C4 00 40 A0 E1      MOV     R4, R0
.text:000000C8 04 10 A0 E1      MOV     R1, R4
.text:000000CC 5A 0F 8F E2      ADR     R0, aD_0          ; "%d\n"
.text:000000D0 E3 18 00 EB      BL      __2printf
.text:000000D4 00 00 A0 E3      MOV     R0, #0
.text:000000D8 10 80 BD E8      LDMFD    SP!, {R4,PC}
```

В функции `main()` просто вызываются две функции, в первую (`f()`) передается три значения. Как уже было упомянуто, первые 4 значения в ARM обычно передаются в первых 4-х регистрах (`R0 - R3`). Функция `f()`, как видно, использует три первых регистра (`R0 - R2`) как аргументы.

Инструкция `MLA` (*Multiply Accumulate*) перемножает два первых операнда (`R3` и `R1`), прибавляет к произведению третий операнд (`R2`) и помещает результат в нулевой регистр (`R0`), через который, по стандарту, возвращаются значения функций.

Умножение и сложение одновременно⁴ (*Fused multiply-add*) это часто применяемая операция. Кстати, аналогичной инструкции в x86 не было до появления FMA-инструкций в SIMD⁵.

Самая первая инструкция `MOV R3, R0`, по-видимому, избыточна (можно было бы обойтись только одной инструкцией `MLA`). Компилятор не оптимизировал её, ведь, это компиляция без оптимизации.

Инструкция `BX` возвращает управление по адресу, записанному в `LR` и, если нужно, переключает режимы процессора с Thumb на ARM или наоборот. Это может быть необходимым потому, что, как мы видим, функции `f()` неизвестно, из какого кода она будет вызываться, из ARM или Thumb. Поэтому, если она будет вызываться из кода Thumb, `BX` не только возвращает управление в вызывающую функцию, но также переключает процессор в режим Thumb. Либо не переключит, если функция вызывалась из кода для режима ARM: [ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition, (2012)A2.3.2].

9.3.2. Оптимизирующий Keil 6/2013 (Режим ARM)

```
.text:00000098          f
.text:00000098 91 20 20 E0      MLA     R0, R1, R0, R2
.text:0000009C 1E FF 2F E1      BX      LR
```

А вот и функция `f()`, скомпилированная компилятором Keil в режиме полной оптимизации (`-O3`). Инструкция `MOV` была оптимизирована: теперь `MLA` использует все входящие регистры и помещает результат в `R0`, где вызываемая функция будет его читать и использовать.

9.3.3. Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:0000005E 48 43      MULS    R0, R1
.text:00000060 80 18      ADDS    R0, R0, R2
.text:00000062 70 47      BX      LR
```

В режиме Thumb инструкция `MLA` недоступна, так что компилятору пришлось сгенерировать код, делающий обе операции по отдельности.

Первая инструкция `MULS` умножает `R0` на `R1`, оставляя результат в `R0`. Вторая (`ADDS`) складывает результат и `R2`, оставляя результат в `R0`.

⁴Wikipedia: Умножение-сложение

⁵wikipedia

9.3.4. ARM64

Оптимизирующий GCC (Linaro) 4.9

Тут всё просто. `MADD` это просто инструкция, производящая умножение и сложение одновременно (как `MLA`, которую мы уже видели). Все 3 аргумента передаются в 32-битных частях X-регистров. Действительно, типы аргументов это 32-битные `int`'ы. Результат возвращается в `W0`.

Листинг 9.9: Оптимизирующий GCC (Linaro) 4.9

```
f:
    madd    w0, w0, w1, w2
    ret

main:
; сохранить FP и LR в стековом фрейме:
    stp     x29, x30, [sp, -16]!
    mov     w2, 3
    mov     w1, 2
    add     x29, sp, 0
    mov     w0, 1
    bl      f
    mov     w1, w0
    adrp    x0, .LC7
    add     x0, x0, :lo12:.LC7
    bl      printf
; возврат 0
    mov     w0, 0
; восстановить FP и LR
    ldp     x29, x30, [sp], 16
    ret

.LC7:
    .string "%d\n"
```

Также расширим все типы данных до 64-битных `uint64_t` и попробуем:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
};

int main()
{
    printf ("%lld\n", f(0x1122334455667788,
                        0x1111111122222222,
                        0x3333333344444444));
    return 0;
};
```

```
f:
    madd     x0, x0, x1, x2
    ret

main:
    mov     x1, 13396
    adrp    x0, .LC8
    stp     x29, x30, [sp, -16]!
    movk    x1, 0x27d0, lsl 16
    add     x0, x0, :lo12:.LC8
    movk    x1, 0x122, lsl 32
    add     x29, sp, 0
    movk    x1, 0x58be, lsl 48
    bl      printf
    mov     w0, 0
    ldp     x29, x30, [sp], 16
    ret
```

9.4. MIPS

```
.LC8:
    .string "%lld\n"
```

Функция `f()` точно такая же, только теперь используются полные части 64-битных X-регистров. Длинные 64-битные значения загружаются в регистры по частям, это описано здесь: [29.3.1](#) (стр. 440).

Неоптимизирующий GCC (Linaro) 4.9

Неоптимизирующий компилятор выдает немного лишнего кода:

```
f:
    sub    sp, sp, #16
    str    w0, [sp,12]
    str    w1, [sp,8]
    str    w2, [sp,4]
    ldr    w1, [sp,12]
    ldr    w0, [sp,8]
    mul    w1, w1, w0
    ldr    w0, [sp,4]
    add    w0, w1, w0
    add    sp, sp, 16
    ret
```

Код сохраняет входные аргументы в локальном стеке на случай если кому-то (или чему-то) в этой функции понадобится использовать регистры `W0...W2`, перезаписывая оригинальные аргументы функции, которые могут понадобиться в будущем. Это называется *Register Save Area*. ([*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)]⁶) Вызываемая функция не обязана сохранять их. Это то же что и «Shadow Space»: [9.2.1](#) (стр. 96).

Почему оптимизирующий GCC 4.9 убрал этот, сохраняющий аргументы, код?

Потому что он провел дополнительную работу по оптимизации и сделал вывод, что аргументы функции не понадобятся в будущем и регистры `W0...W2` также не будут использоваться.

Также мы видим пару инструкций `MUL / ADD` вместо одной `MADD`.

9.4. MIPS

Листинг 9.10: Оптимизирующий GCC 4.4.5

```
.text:00000000 f:
; $a0=a
; $a1=b
; $a2=c
.text:00000000          mult    $a1, $a0
.text:00000004          mflo    $v0
.text:00000008          jr      $ra
.text:0000000C          addu    $v0, $a2, $v0    ; branch delay slot
; результат в $v0 во время выхода

.text:00000010 main:
.text:00000010
.text:00000010 var_10    = -0x10
.text:00000010 var_4     = -4
.text:00000010
.text:00000010          lui     $gp, (__gnu_local_gp >> 16)
.text:00000014          addiu   $sp, -0x20
.text:00000018          la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000001C          sw      $ra, 0x20+var_4($sp)
.text:00000020          sw      $gp, 0x20+var_10($sp)
; установить c:
.text:00000024          li      $a2, 3
; установить a:
.text:00000028          li      $a0, 1
.text:0000002C          jal     f
```

⁶Также доступно здесь: <http://go.yurichev.com/17287>

9.4. MIPS

```
; установить b:
.text:00000030      li      $a1, 2          ; branch delay slot
; результат сейчас в $v0
.text:00000034      lw      $gp, 0x20+var_10($sp)
.text:00000038      lui     $a0, ($LC0 >> 16)
.text:0000003C      lw      $t9, (printf & 0xFFFF)($gp)
.text:00000040      la      $a0, ($LC0 & 0xFFFF)
.text:00000044      jalr    $t9
; взять результат ф-ции f() и передать его как второй аргумент в printf():
.text:00000048      move    $a1, $v0          ; branch delay slot
.text:0000004C      lw      $ra, 0x20+var_4($sp)
.text:00000050      move    $v0, $zero
.text:00000054      jr      $ra
.text:00000058      addiu   $sp, 0x20          ; branch delay slot
```

Первые 4 аргумента функции передаются в четырех регистрах с префиксами A-.

В MIPS есть два специальных регистра: HI и LO, которые выставляются в 64-битный результат умножения во время исполнения инструкции `MULT`.

К регистрам можно обращаться только используя инструкции `MFLO` и `MFHI`. Здесь `MFLO` берет младшую часть результата умножения и записывает в `$V0`. Так что старшая 32-битная часть результата игнорируется (содержимое регистра HI не используется). Действительно, мы ведь работаем с 32-битным типом `int`.

И наконец, `ADDU` («Add Unsigned» — добавить беззнаковое) прибавляет значение третьего аргумента к результату.

В MIPS есть две разные инструкции сложения: `ADD` и `ADDU`. На самом деле, дело не в знаковых числах, а в исключениях: `ADD` может вызвать исключение во время переполнения. Это иногда полезно⁷ и поддерживается, например, в ЯП Ada.

`ADDU` не вызывает исключения во время переполнения. А так как Си/Си++ не поддерживает всё это, мы видим здесь `ADDU` вместо `ADD`.

32-битный результат оставляется в `$V0`.

В `main()` есть новая для нас инструкция: `JAL` («Jump and Link»). Разница между `JAL` и `JALR` в том, что относительное смещение кодируется в первой инструкции, а `JALR` переходит по абсолютному адресу, записанному в регистр («Jump and Link Register»).

Обе функции `f()` и `main()` расположены в одном объектном файле, так что относительный адрес `f()` известен и фиксирован.

⁷<http://go.yurichev.com/17326>

Глава 10

Ещё о возвращаемых результатах

Результат выполнения функции в x86 обычно возвращается ¹ через регистр `EAX`, а если результат имеет тип байт или символ (*char*), то в самой младшей части `EAX` — `AL`. Если функция возвращает число с плавающей запятой, то будет использован регистр FPU `ST(0)`. В ARM обычно результат возвращается в регистре `R0`.

10.1. Попытка использовать результат функции возвращающей *void*

Кстати, что будет, если возвращаемое значение в функции `main()` объявлять не как *int*, а как *void*? Т.н. startup-код вызывает `main()` примерно так:

```
push envp
push argv
push argc
call main
push eax
call exit
```

Иными словами:

```
exit(main(argc,argv,envp));
```

Если вы объявите `main()` как *void*, и ничего не будете возвращать явно (при помощи выражения *return*), то в единственный аргумент `exit()` попадет то, что лежало в регистре `EAX` на момент выхода из `main()`. Там, скорее всего, будет какие-то случайное число, оставшееся от работы вашей функции. Так что код завершения программы будет псевдослучайным.

Мы можем это проиллюстрировать. Заметьте, что у функции `main()` тип возвращаемого значения именно *void*:

```
#include <stdio.h>

void main()
{
    printf ("Hello, world!\n");
};
```

Скомпилируем в Linux.

GCC 4.8.1 заменила `printf()` на `puts()` (мы видели это прежде: 4.4.3 (стр. 17)), но это нормально, потому что `puts()` возвращает количество выведенных символов, так же как и `printf()`. Обратите внимание на то, что `EAX` не обнуляется перед выходом из `main()`. Это значит что `EAX` перед выходом из `main()` содержит то, что `puts()` оставляет там.

Листинг 10.1: GCC 4.8.1

```
.LC0:
    .string "Hello, world!"
main:
```

¹См. также: MSDN: Return Values (C++): [MSDN](#)

10.2. ЧТО ЕСЛИ НЕ ИСПОЛЬЗОВАТЬ РЕЗУЛЬТАТ ФУНКЦИИ?

```
push    ebp
mov     ebp, esp
and     esp, -16
sub     esp, 16
mov     DWORD PTR [esp], OFFSET FLAT:.LC0
call    puts
leave
ret
```

Напишем небольшой скрипт на `bash`, показывающий статус возврата («exit status» или «exit code»):

Листинг 10.2: `tst.sh`

```
#!/bin/sh
./hello_world
echo $?
```

И запустим:

```
$ tst.sh
Hello, world!
14
```

14 это как раз количество выведенных символов.

10.2. Что если не использовать результат функции?

`printf()` возвращает количество успешно выведенных символов, но результат работы этой функции редко используется на практике.

Можно даже явно вызывать функции, чей смысл именно в возвращаемых значениях, но явно не использовать их:

```
int f()
{
    // пропускаем первые 3 случайных значения:
    rand();
    rand();
    rand();
    // и используем e4-:
    return rand();
};
```

Результат работы `rand()` остается в `EAX` во всех четырех случаях. Но в первых трех случаях значение, лежащее в `EAX`, просто не используется.

10.3. Возврат структуры

Вернемся к тому факту, что возвращаемое значение остается в регистре `EAX`. Вот почему старые компиляторы Си не способны создавать функции, возвращающие нечто большее, нежели помещается в один регистр (обычно тип `int`), а когда нужно, приходится возвращать через указатели, указываемые в аргументах. Так что как правило, если функция должна вернуть несколько значений, она возвращает только одно, а остальные — через указатели. Хотя позже и стало возможным, вернуть, скажем, целую структуру, но этот метод до сих пор не очень популярен. Если функция должна вернуть структуру, вызывающая функция должна сама, скрыто и прозрачно для программиста, выделить место и передать указатель на него в качестве первого аргумента. Это почти то же самое что и сделать это вручную, но компилятор прячет это.

Небольшой пример:

```
struct s
{
    int a;
    int b;
    int c;
};
```

10.3. ВОЗВРАТ СТРУКТУРЫ

```
struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};
```

...получим (MSVC 2010 /Ox):

```
$T3853 = 8 ; size = 4
_a$ = 12 ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC ; get_some_values
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     eax, DWORD PTR $T3853[esp-4]
    lea     edx, DWORD PTR [ecx+1]
    mov     DWORD PTR [eax], edx
    lea     edx, DWORD PTR [ecx+2]
    add     ecx, 3
    mov     DWORD PTR [eax+4], edx
    mov     DWORD PTR [eax+8], ecx
    ret     0
?get_some_values@@YA?AUs@@H@Z ENDP ; get_some_values
```

\$T3853 это имя внутреннего макроса для передачи указателя на структуру.

Этот пример можно даже переписать, используя расширения C99:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

Листинг 10.3: GCC 4.8.1

```
_get_some_values proc near
ptr_to_struct = dword ptr 4
a = dword ptr 8

    mov     edx, [esp+a]
    mov     eax, [esp+ptr_to_struct]
    lea     ecx, [edx+1]
    mov     [eax], ecx
    lea     ecx, [edx+2]
    add     edx, 3
    mov     [eax+4], ecx
    mov     [eax+8], edx
    retn
_get_some_values endp
```

Как видно, функция просто заполняет поля в структуре, выделенной вызывающей функцией. Как если бы передавался просто указатель на структуру. Так что никаких проблем с эффективностью нет.

Глава 11

Указатели

Указатели также часто используются для возврата значений из функции (вспомните случай со `scanf()` (8 (стр. 62))).

Например, когда функции нужно вернуть сразу два значения.

11.1. Пример с глобальными переменными

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
    *sum=x+y;
    *product=x*y;
};

int sum, product;

void main()
{
    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
};
```

Это компилируется в:

Листинг 11.1: Оптимизирующий MSVC 2010 (/Ob0)

```
COMM    _product:DWORD
COMM    _sum:DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H

_x$ = 8                                ; size = 4
_y$ = 12                               ; size = 4
_sum$ = 16                             ; size = 4
_product$ = 20                         ; size = 4
_f1     PROC
    mov     ecx, DWORD PTR _y$[esp-4]
    mov     eax, DWORD PTR _x$[esp-4]
    lea     edx, DWORD PTR [eax+ecx]
    imul    eax, ecx
    mov     ecx, DWORD PTR _product$[esp-4]
    push    esi
    mov     esi, DWORD PTR _sum$[esp]
    mov     DWORD PTR [esi], edx
    mov     DWORD PTR [ecx], eax
    pop     esi
    ret     0
_f1     ENDP

_main   PROC
    push    OFFSET _product
    push    OFFSET _sum
```


11.1. ПРИМЕР С ГЛОБАЛЬНЫМИ ПЕРЕМЕННЫМИ

```
    push    456                      ; 000001c8H
    push    123                      ; 0000007bH
    call    _f1
    mov     eax, DWORD PTR _product
    mov     ecx, DWORD PTR _sum
    push    eax
    push    ecx
    push    OFFSET $SG2803
    call    DWORD PTR __imp__printf
    add     esp, 28                   ; 0000001cH
    xor     eax, eax
    ret     0
_main     ENDP
```

11.1. ПРИМЕР С ГЛОБАЛЬНЫМИ ПЕРЕМЕННЫМИ

Посмотрим это в OllyDbg:

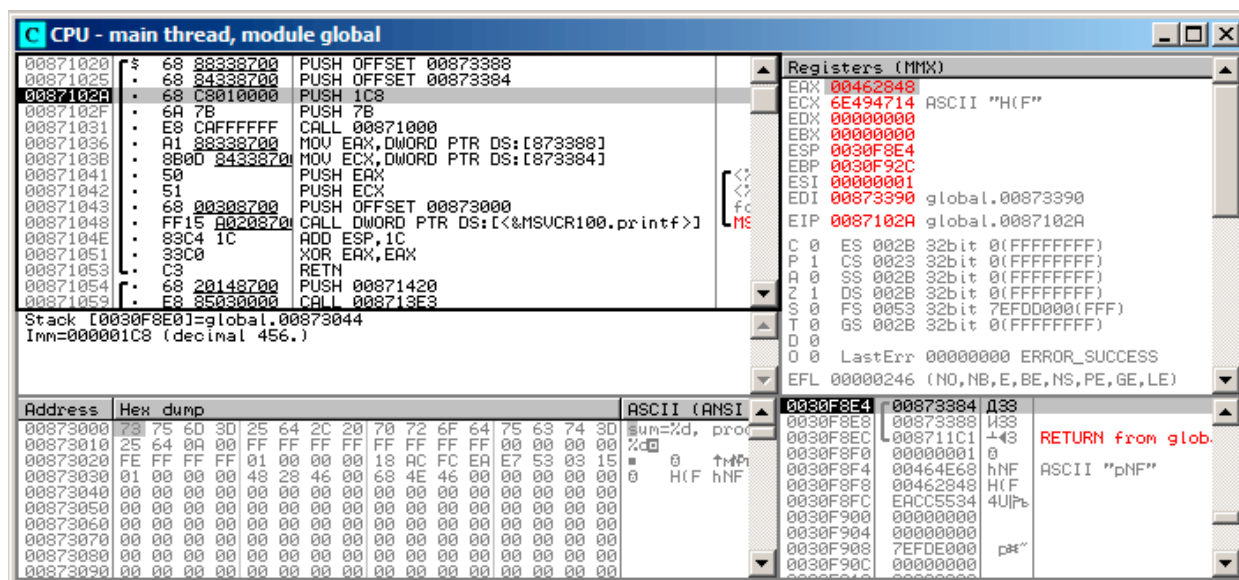


Рис. 11.1: OllyDbg: передаются адреса двух глобальных переменных в f1()

В начале адреса обеих глобальных переменных передаются в f1(). Можно нажать «Follow in dump» на элементе стека и в окне слева увидим место в сегменте данных, выделенное для двух переменных.

Эти переменные обнулены, потому что по стандарту неинициализированные данные (BSS) обнуляются перед началом исполнения: [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.8p10].

11.1. ПРИМЕР С ГЛОБАЛЬНЫМИ ПЕРЕМЕННЫМИ

И они находятся в сегменте данных, о чем можно удостовериться, нажав Alt-M и увидев карту памяти:

Memory map									
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as	
00050000	00004000				Map	R	R		
00060000	00001000				Priv	RW	RW		
00070000	00007000				Map	R	R		
00159000	00007000				Priv	RW	Gua: RW	Gua: RW	C:\Windows\System32\loc
0030D000	00001000				Priv	RW	Gua: RW	Gua: RW	
0030E000	00002000			Stack of main thread	Priv	RW	RW		
00460000	00005000			Heap	Priv	RW	RW		
004A0000	00007000				Priv	RW	RW		
006B0000	0000C000			Default heap	Priv	RW	RW		
00870000	00001000	global		PE header	Img	R	RWE Cop		
00871000	00001000	global	.text	Code	Img	R E	RWE Cop		
00872000	00001000	global	.rdata	Imports	Img	R	RWE Cop		
00873000	00001000	global	.data	Data	Img	RW	RWE Cop		
00874000	00001000	global	.reloc	Relocations	Img	R	RWE Cop		
6E3E0000	00001000	MSUCR100		PE header	Img	R	RWE Cop		
6E3E1000	00002000	MSUCR100	.text	Code, imports, exports	Img	R E	RWE Cop		
6E493000	00006000	MSUCR100	.data	Data	Img	RW	Cop: RWE Cop		
6E499000	00001000	MSUCR100	.rsrc	Resources	Img	R	RWE Cop		
6E49A000	00005000	MSUCR100	.reloc	Relocations	Img	R	RWE Cop		
755D0000	00001000	Mod_755D		PE header	Img	R	RWE Cop		
755D1000	00003000				Img	R E	RWE Cop		
755D4000	00001000				Img	RW	RWE Cop		
755D5000	00003000				Img	R	RWE Cop		
755E0000	00001000	Mod_755E		PE header	Img	R	RWE Cop		
755E1000	00004000				Img	R E	RWE Cop		
7562E000	00005000				Img	RW	Cop: RWE Cop		
75633000	00009000				Img	R	RWE Cop		

Рис. 11.2: OllyDbg: карта памяти

11.1. ПРИМЕР С ГЛОБАЛЬНЫМИ ПЕРЕМЕННЫМИ

Трассируем (F7) до начала исполнения `f1()` :

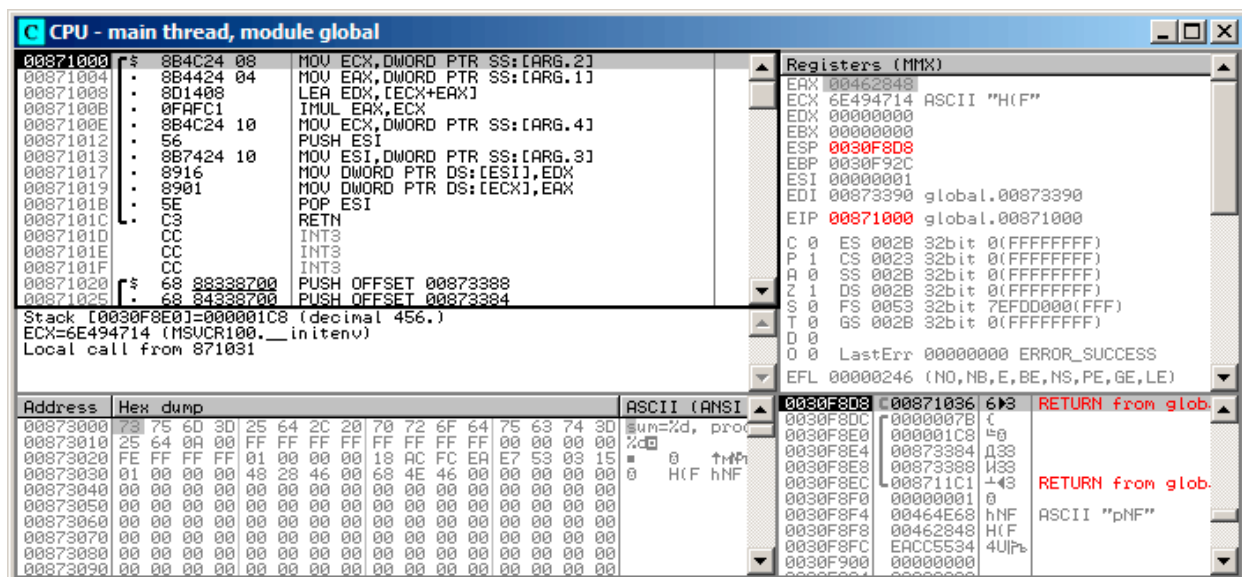


Рис. 11.3: OllyDbg: начало работы `f1()`

В стеке видны значения 456 (`0x1C8`) и 123 (`0x7B`), а также адреса двух глобальных переменных.

11.1. ПРИМЕР С ГЛОБАЛЬНЫМИ ПЕРЕМЕННЫМИ

Трассируем до конца `f1()`. Мы видим в окне слева, как результаты вычисления появились в глобальных переменных:

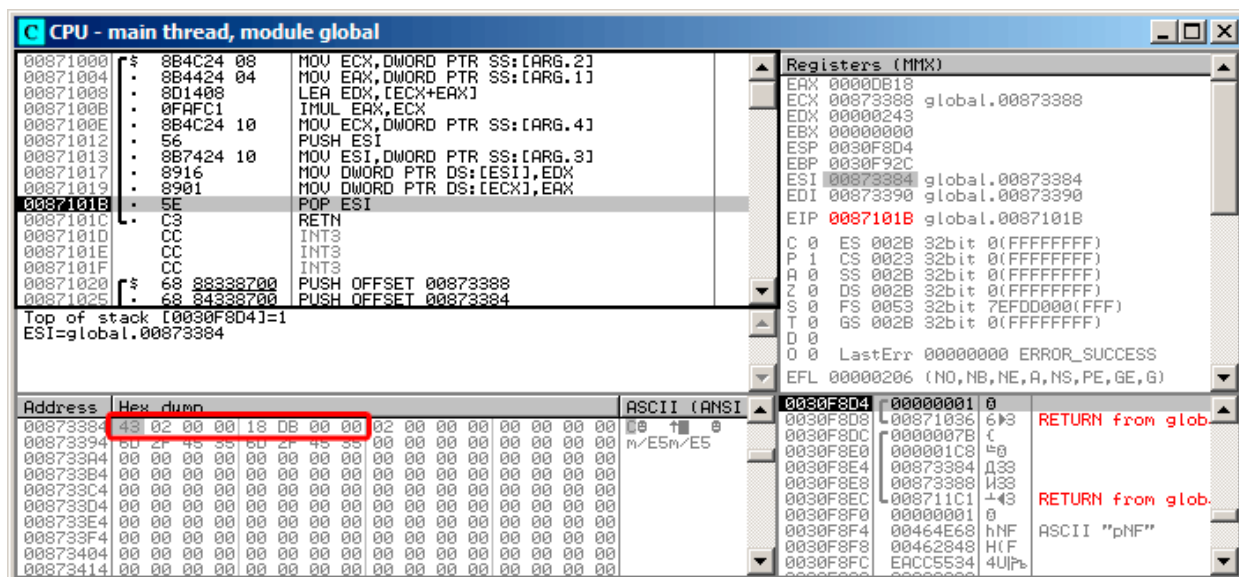


Рис. 11.4: OllyDbg: `f1()` заканчивает работу

11.2. ПРИМЕР С ЛОКАЛЬНЫМИ ПЕРЕМЕННЫМИ

Теперь из глобальных переменных значения загружаются в регистры для передачи в `printf()` :

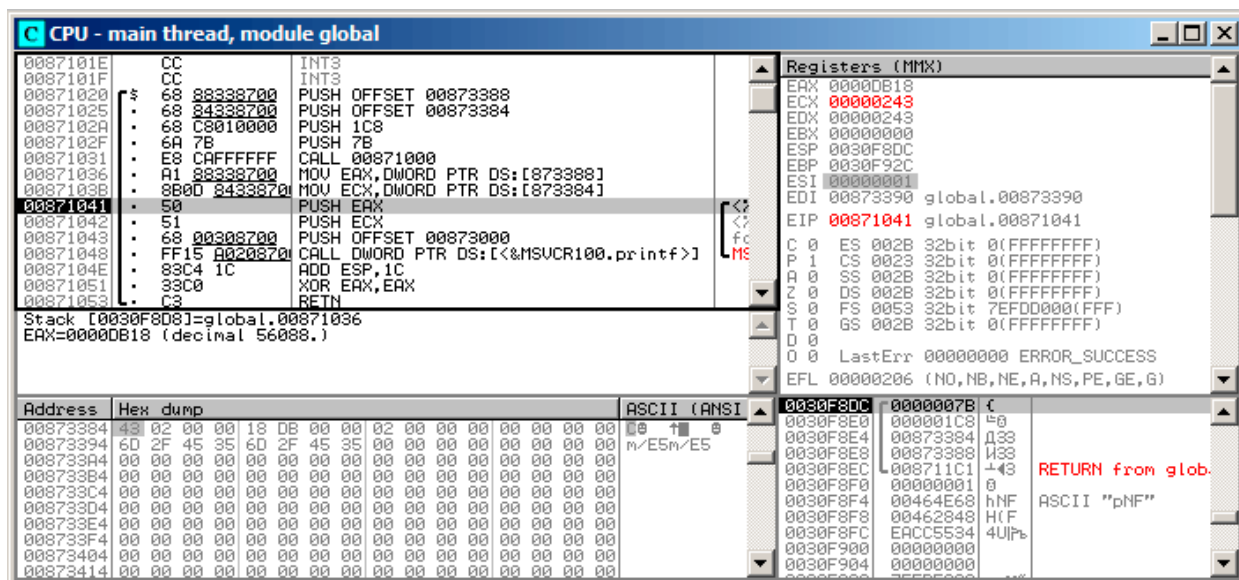


Рис. 11.5: OllyDbg: адреса глобальных переменных передаются в `printf()`

11.2. Пример с локальными переменными

Немного переделаем пример:

Листинг 11.2: теперь переменные локальные

```
void main()
{
    int sum, product; // теперь переменные локальные в этой ф-ции

    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
};
```

Код функции `f1()` не изменится. Изменится только `main()` :

Листинг 11.3: Оптимизирующий MSVC 2010 (/Ob0)

```
_product$ = -8 ; size = 4
_sum$ = -4 ; size = 4
_main PROC
; Line 10
sub esp, 8
; Line 13
lea eax, DWORD PTR _product$[esp+8]
push eax
lea ecx, DWORD PTR _sum$[esp+12]
push ecx
push 456 ; 000001c8H
push 123 ; 0000007bH
call _f1
; Line 14
mov edx, DWORD PTR _product$[esp+24]
mov eax, DWORD PTR _sum$[esp+24]
push edx
push eax
push OFFSET $SG2803
call DWORD PTR __imp__printf
; Line 15
xor eax, eax
add esp, 36 ; 00000024H
ret 0
```

11.2. ПРИМЕР С ЛОКАЛЬНЫМИ ПЕРЕМЕННЫМИ

Снова посмотрим в OllyDbg. Адреса локальных переменных в стеке это 0x2EF854 и 0x2EF858. Видно, как они заталкиваются в стек:

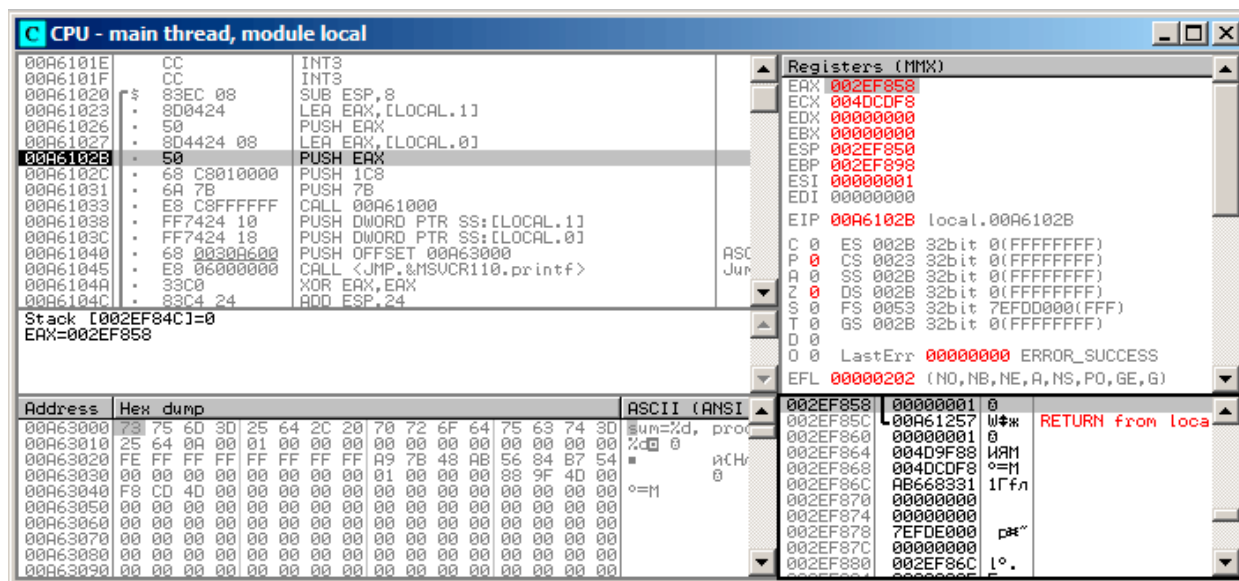


Рис. 11.6: OllyDbg: адреса локальных переменных заталкиваются в стек

11.2. ПРИМЕР С ЛОКАЛЬНЫМИ ПЕРЕМЕННЫМИ

Начало работы `f1()`. В стеке по адресам `0x2EF854` и `0x2EF858` пока находится случайный мусор:

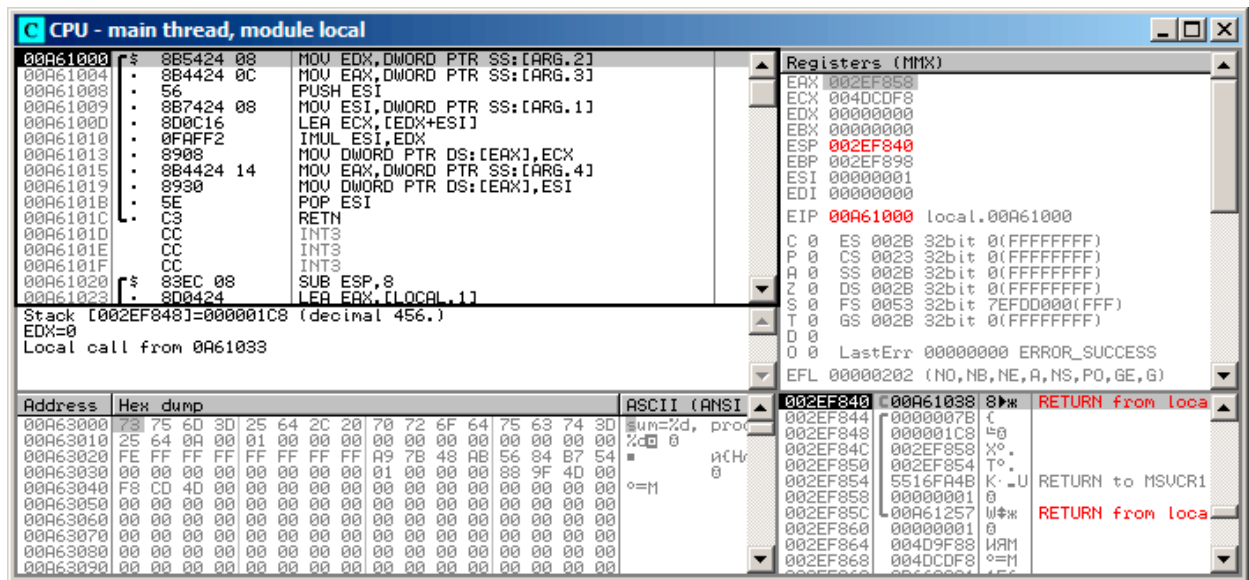


Рис. 11.7: OllyDbg: `f1()` начинает работу

11.3. ВЫВОД

Конец работы `f1()` :

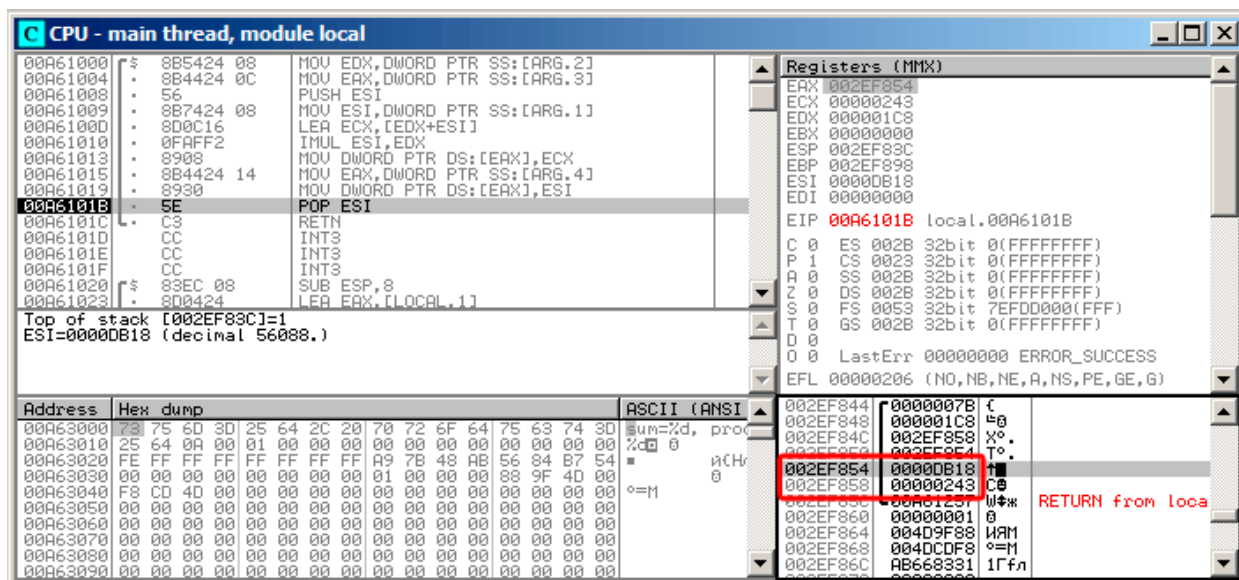


Рис. 11.8: OllyDbg: `f1()` заканчивает работу

В стеке по адресам `0x2EF854` и `0x2EF858` теперь находятся значения `0xDB18` и `0x243`, это результаты работы `f1()`.

11.3. Вывод

`f1()` может одинаково хорошо возвращать результаты работы в любые места памяти. В этом суть и удобство указателей. Кстати, *references* в Си++ работают точно так же. Читайте больше об этом: (52.3 (стр. 555)).

Глава 12

Оператор GOTO

Оператор GOTO считается анти-паттерном, см: [Edgar Dijkstra, *Go To Statement Considered Harmful* (1968)¹]. Но тем не менее, его можно использовать в разумных пределах, см: [Donald E. Knuth, *Structured Programming with go to Statements* (1974)²]³.

Вот простейший пример:

```
#include <stdio.h>

int main()
{
    printf ("begin\n");
    goto exit;
    printf ("skip me!\n");
exit:
    printf ("end\n");
};
```

Вот что мы получаем в MSVC 2012:

Листинг 12.1: MSVC 2012

```
$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main  PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2934 ; 'begin'
        call    _printf
        add     esp, 4
        jmp     SHORT $exit$3
        push    OFFSET $SG2936 ; 'skip me!'
        call    _printf
        add     esp, 4
$exit$3:
        push    OFFSET $SG2937 ; 'end'
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main  ENDP
```

Выражение *goto* заменяется инструкцией **JMP**, которая работает точно также: безусловный переход в другое место. Вызов второго **printf()** может исполниться только при помощи человеческого вмешательства, используя отладчик или модифицирование кода.

¹<http://yurichev.com/mirrors/Dijkstra68.pdf>

²<http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf>

³В [Денис Юричев, *Заметки о языке программирования Си/Си++*] также есть примеры.

Это также может быть простым упражнением на модификацию кода.

Откроем исполняемый файл в Niew:

```

C:\Polygon\goto.exe      FRO -----      a32 PE .00401000
.00401000: 55      push      ebp
.00401001: 8BEC     mov       ebp,esp
.00401003: 6800304000  push     000403000 ;'begin' --1
.00401008: FF1590204000 call     printf
.0040100E: 83C404    add      esp,4
.00401011: EB0E     jmps     .00401021 --2
.00401013: 6808304000  push     000403008 ;'skip me!' --3
.00401018: FF1590204000 call     printf
.0040101E: 83C404    add      esp,4
.00401021: 6814304000 2push     000403014 --4
.00401026: FF1590204000 call     printf
.0040102C: 83C404    add      esp,4
.0040102F: 33C0     xor       eax,eax
.00401031: 5D       pop       ebp
.00401032: C3      retn ; _^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_ ^_

```

Рис. 12.1: Hiew

12.1. МЕРТВЫЙ КОД

Поместите курсор по адресу `JMP (0x410)`, нажмите F3 (редактирование), нажмите два нуля, так что опкод становится `EB 00`:

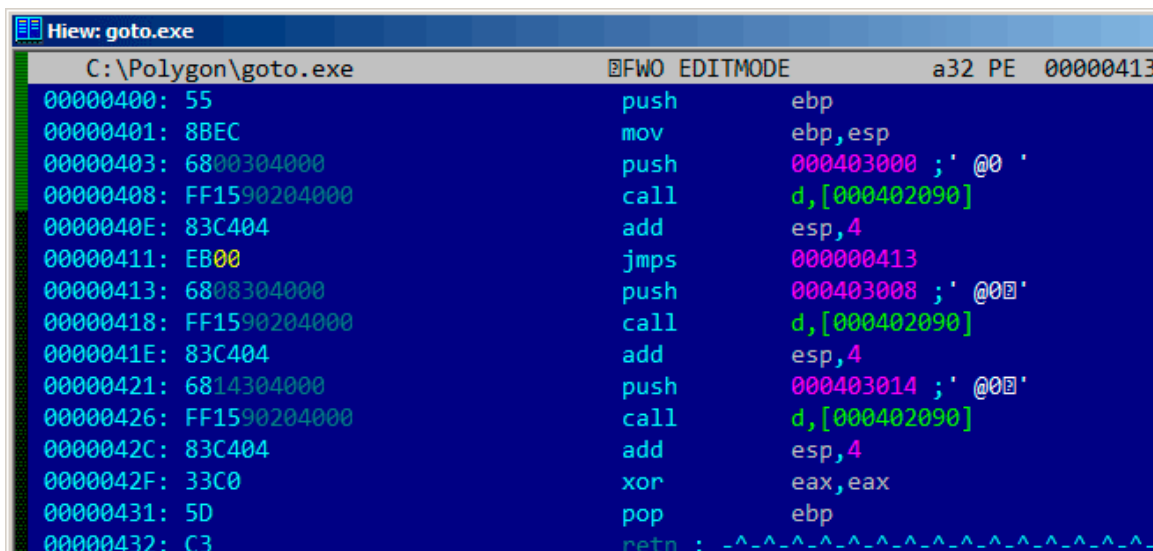


Рис. 12.2: Hiew

Второй байт опкода `JMP` это относительное смещение от перехода. 0 означает место прямо после текущей инструкции. Теперь `JMP` не будет пропускать следующий вызов `printf()`. Нажмите F9 (запись) и выйдите. Теперь мы запускаем исполняемый файл и видим это:

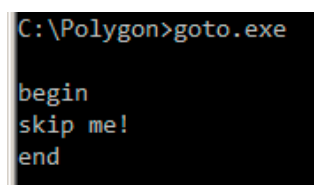


Рис. 12.3: Результат

Подобного же эффекта можно достичь, если заменить инструкцию `JMP` на две инструкции `NOP`. `NOP` имеет опкод `0x90` и длину в 1 байт, так что нужно 2 инструкции для замены.

12.1. Мертвый код

Вызов второго `printf()` также называется «мертвым кодом» («dead code») в терминах компиляторов. Это значит, что он никогда не будет исполнен. Так что если вы компилируете этот пример с оптимизацией, компилятор удаляет «мертвый код» не оставляя следа:

Листинг 12.2: Оптимизирующий MSVC 2012

```
$SG2981 DB      'begin', 0aH, 00H
$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main PROC
    push        OFFSET $SG2981 ; 'begin'
    call        _printf
    push        OFFSET $SG2984 ; 'end'
$exit$4:
    call        _printf
    add         esp, 8
    xor         eax, eax
    ret         0
_main ENDP
```

Впрочем, строку «skip me!» компилятор убрать забыл.

12.2. Упражнение

Попробуйте добиться того же самого в вашем любимом компиляторе и отладчике.

Глава 13

Условные переходы

13.1. Простой пример

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

13.1.1. x86

x86 + MSVC

Имеем в итоге функцию `f_signed()` :

Листинг 13.1: Неоптимизирующий MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    cmp     eax, DWORD PTR _b$[ebp]
    jle     SHORT $LN3@f_signed
    push    OFFSET $SG737      ; 'a>b'
    call    _printf
    ;
```

13.1. ПРОСТОЙ ПРИМЕР

```
add    esp, 4
$LN3@f_signed:
mov     ecx, DWORD PTR _a$[ebp]
cmp     ecx, DWORD PTR _b$[ebp]
jne     SHORT $LN2@f_signed
push    OFFSET $SG739          ; 'a==b'
call    _printf
add     esp, 4
$LN2@f_signed:
mov     edx, DWORD PTR _a$[ebp]
cmp     edx, DWORD PTR _b$[ebp]
jge     SHORT $LN4@f_signed
push    OFFSET $SG741          ; 'a<b'
call    _printf
add     esp, 4
$LN4@f_signed:
pop     ebp
ret     0
_f_signed ENDP
```

Первая инструкция **JLE** значит *Jump if Less or Equal*. Если второй операнд больше первого или равен ему, произойдет переход туда, где будет следующая проверка.

А если это условие не срабатывает (то есть второй операнд меньше первого), то перехода не будет, и сработает первый **printf()**.

Вторая проверка это **JNE** : *Jump if Not Equal*. Переход не произойдет, если операнды равны.

Третья проверка **JGE** : *Jump if Greater or Equal* — переход если первый операнд больше второго или равен ему. Кстати, если все три условных перехода сработают, ни один **printf()** не вызовется. Но без внешнего вмешательства это невозможно.

Функция **f_unsigned()** точно такая же, за тем исключением, что используются инструкции **JBE** и **JAЕ** вместо **JLE** и **JGE** :

Листинг 13.2: GCC

```
_a$ = 8    ; size = 4
_b$ = 12   ; size = 4
_f_unsigned PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    cmp     eax, DWORD PTR _b$[ebp]
    jbe     SHORT $LN3@f_unsigned
    push    OFFSET $SG2761      ; 'a>b'
    call    _printf
    add     esp, 4
$LN3@f_unsigned:
    mov     ecx, DWORD PTR _a$[ebp]
    cmp     ecx, DWORD PTR _b$[ebp]
    jne     SHORT $LN2@f_unsigned
    push    OFFSET $SG2763      ; 'a==b'
    call    _printf
    add     esp, 4
$LN2@f_unsigned:
    mov     edx, DWORD PTR _a$[ebp]
    cmp     edx, DWORD PTR _b$[ebp]
    jae     SHORT $LN4@f_unsigned
    push    OFFSET $SG2765      ; 'a<b'
    call    _printf
    add     esp, 4
$LN4@f_unsigned:
    pop     ebp
    ret     0
_f_unsigned ENDP
```

Здесь всё то же самое, только инструкции условных переходов немного другие:

13.1. ПРОСТОЙ ПРИМЕР

`JBE` — *Jump if Below or Equal* и `JAE` — *Jump if Above or Equal*. Эти инструкции (`JA` / `JAE` / `JB` / `JBE`) отличаются от `JG` / `JGE` / `JL` / `JLE` тем, что работают с беззнаковыми переменными.

Отступление: смотрите также секцию о представлении знака в числах ([31](#) (стр. 446)). Таким образом, увидев где используется `JG` / `JL` вместо `JA` / `JB` и наоборот, можно сказать почти уверенно насчет того, является ли тип переменной знаковым (signed) или беззнаковым (unsigned).

Далее функция `main()`, где ничего нового для нас нет:

Листинг 13.3: `main()`

```
_main PROC
    push    ebp
    mov     ebp, esp
    push    2
    push    1
    call    _f_signed
    add     esp, 8
    push    2
    push    1
    call    _f_unsigned
    add     esp, 8
    xor     eax, eax
    pop     ebp
    ret     0
_main ENDP
```


Если попробовать этот пример в OllyDbg, можно увидеть, как выставляются флаги. Начнем с функции `f_unsigned()`, которая работает с беззнаковыми числами.

В целом в каждой функции `CMP` выполняется три раза, но для одних и тех же аргументов, так что флаги все три раза будут одинаковы.

Результат первого сравнения:

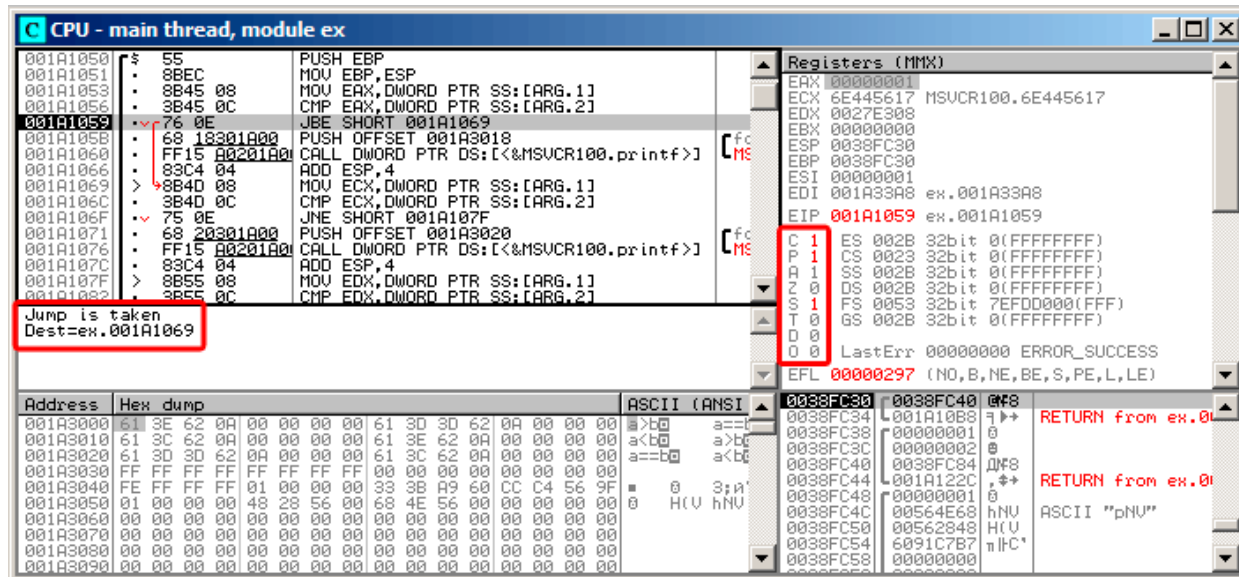
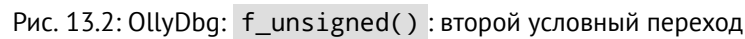


Рис. 13.1: OllyDbg: `f_unsigned()` : первый условный переход

Итак, флаги: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Для краткости, в OllyDbg флаги называются только одной буквой.

OllyDbg подсказывает, что первый переход (`JBE`) сейчас сработает. Действительно, если заглянуть в документацию от Intel, (98.5 (стр. 903)) прочитаем там, что `JBE` срабатывает в случаях если CF=1 или ZF=1. Условие здесь выполняется, так что переход срабатывает.

Следующий переход:



124

13.1. ПРОСТОЙ ПРИМЕР

Третий переход, JNB :

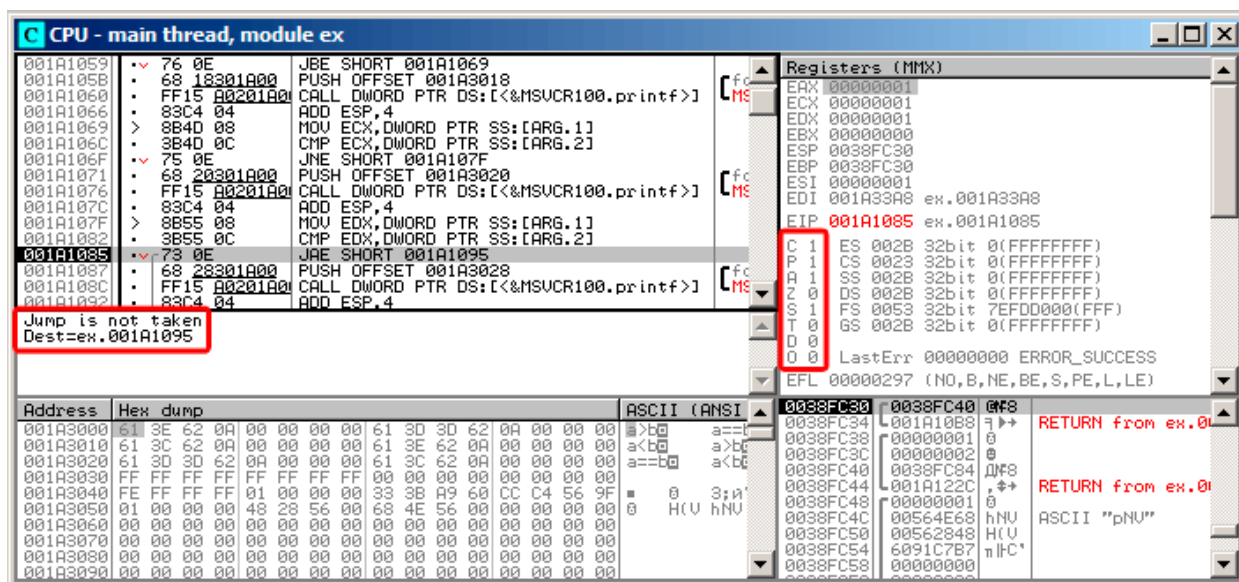


Рис. 13.3: OllyDbg: f_unsigned() : третий условный переход

В документации от Intel (98.5 (стр. 903)) мы можем найти, что JNB срабатывает если CF=0 (carry flag). В нашем случае это не так, переход не срабатывает, и выполняется третий по счету printf() .

13.1. ПРОСТОЙ ПРИМЕР

Теперь можно попробовать в OllyDbg функцию `f_signed()`, работающую с знаковыми величинами. Флаги выставлены точно так же: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Первый переход `JLE` работает:

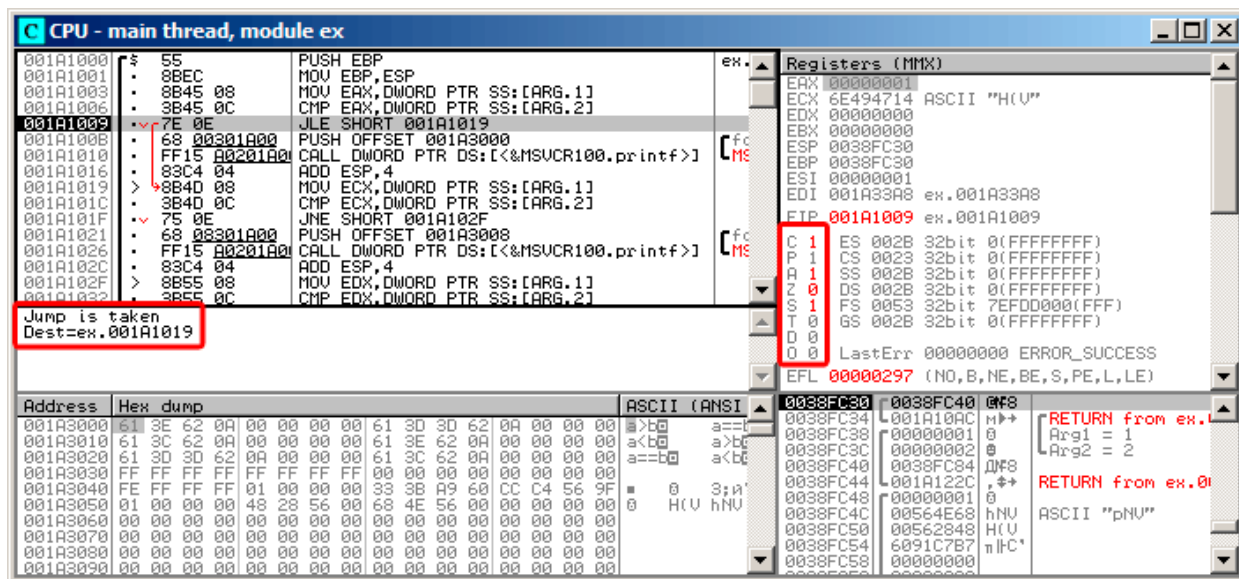


Рис. 13.4: OllyDbg: `f_signed()` : первый условный переход

В документации от Intel (98.5 (стр. 903)) мы можем прочитать, что эта инструкция срабатывает если $ZF=1$ или $SF \neq OF$. В нашем случае $SF \neq OF$, так что переход срабатывает.

13.1. ПРОСТОЙ ПРИМЕР

Второй переход `JNZ` работает: он срабатывает если `ZF=0` (zero flag):

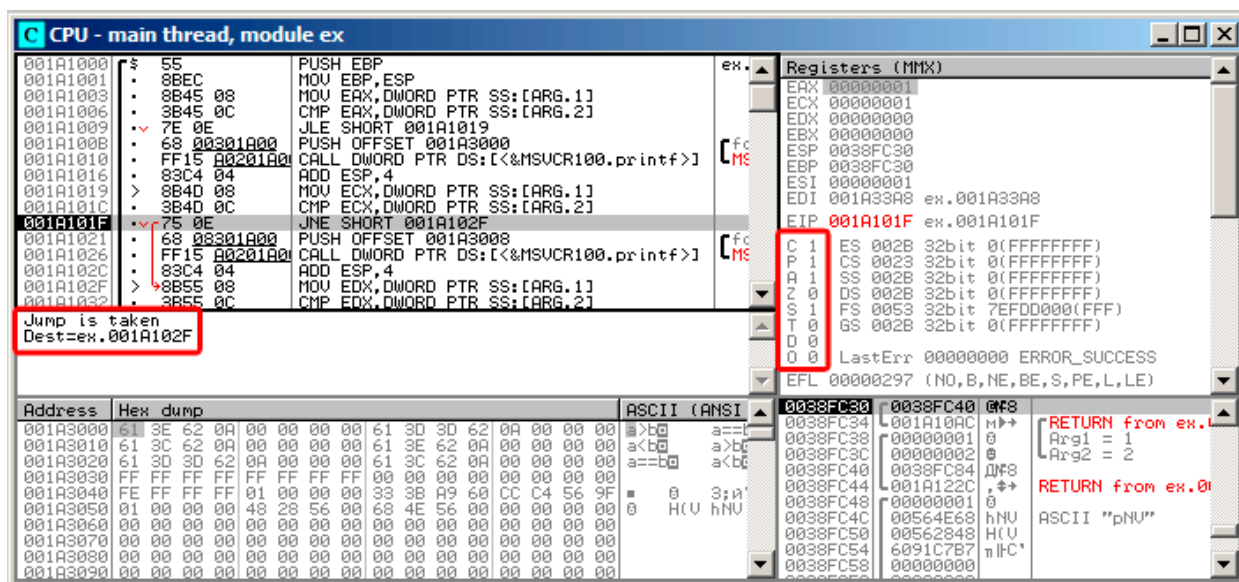


Рис. 13.5: OllyDbg: `f_signed()` : второй условный переход

13.1. ПРОСТОЙ ПРИМЕР

Третий переход JGE не сработает, потому что он срабатывает, только если SF=OF, что в нашем случае не так:

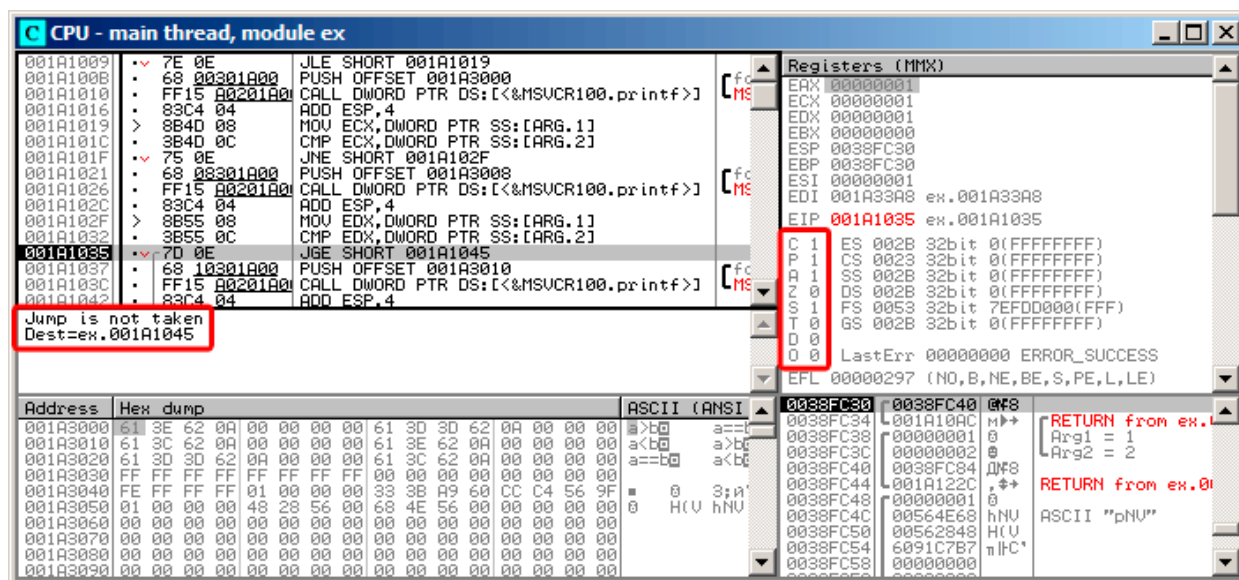


Рис. 13.6: OllyDbg: f_signed() : третий условный переход

Можем попробовать модифицировать исполняемый файл так, чтобы функция `f_unsigned()` всегда показывала «a==b», при любых входящих значениях. Вот как она выглядит в Hiew:

```

.00401000: 55          push     ebp
.00401001: 8BEC       mov     ebp,esp
.00401003: 8B4508     mov     eax,[ebp+8]
.00401006: 3B450C     cmp     eax,[ebp+0C]
.00401009: 7E0D      jle     .00401018 --B1
.0040100B: 680B0400  push     00040B00 --B2
.00401010: E8AA0000  call    .004010BF --B3
.00401015: 83C404     add     esp,4
.00401018: 8B4D08     1mov    ecx,[ebp+8]
.0040101B: 3B4D0C     cmp     ecx,[ebp+0C]
.0040101E: 750D      jnz     .0040102D --B4
.00401020: 680B0400  push     00040B00 ; 'a==b' --B5
.00401025: E8950000  call    .004010BF --B3
.0040102A: 83C404     add     esp,4
.0040102D: 8B5508     4mov    edx,[ebp+8]
.00401030: 3B550C     cmp     edx,[ebp+0C]
.00401033: 7D0D      jge     .00401042 --B6
.00401035: 6810B040  push     00040B10 --B7
.0040103A: E8800000  call    .004010BF --B3
.0040103F: 83C404     add     esp,4
.00401042: 5D        6pop    ebp
.00401043: C3        retn    ; ^^^^
.00401044: CC        int     3
.00401045: CC        int     3
.00401046: CC        int     3
.00401047: CC        int     3
.00401048: CC        int     3
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 9byte 10Leave 11Naked 12AddNam

```

Рис. 13.7: Hiew: функция `f_unsigned()`

Собственно, задач три:

- заставить первый переход срабатывать всегда;
- заставить второй переход не срабатывать никогда;
- заставить третий переход срабатывать всегда.

Так мы направим путь исполнения кода (code flow) во второй `printf()`, и он всегда будет срабатывать и выводить на консоль «a==b».

Для этого нужно изменить три инструкции (или байта):

- Первый переход теперь будет `JMP`, но смещение перехода (jump offset) останется прежним.
- Второй переход может быть и будет срабатывать иногда, но в любом случае он будет совершать переход только на следующую инструкцию, потому что мы выставим смещение перехода (jump offset) в 0.

В этих инструкциях смещение перехода просто прибавляется к адресу следующей инструкции.

Когда смещение 0, переход будет на следующую инструкцию.

- Третий переход конвертируем в `JMP` точно так же, как и первый, он будет срабатывать всегда.

Что и делаем:

Рис. 13.8: Hiew: модифицируем функцию `f_unsigned()`

Если забыть про какой-то из переходов, то тогда будет срабатывать несколько вызовов `printf()`, а нам ведь нужно чтобы исполнялся только один.

Неоптимизирующий GCC

Неоптимизирующий GCC 4.4.1 производит почти такой же код, за исключением `puts()` (4.4.3 (стр. 17)) вместо `printf()`.

Оптимизирующий GCC

Наблюдательный читатель может спросить, зачем исполнять `CMP` так много раз, если флаги всегда одни и те же? По видимому, оптимизирующий MSVC не может этого делать, но GCC 4.8.1 делает больше оптимизаций:

Листинг 13.4: GCC 4.8.1 `f_signed()`

```
f_signed:
    mov     eax, DWORD PTR [esp+8]
    cmp     DWORD PTR [esp+4], eax
    jg      .L6
    je      .L7
    jge     .L1
    mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
    jmp     puts
.L6:
    mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
    jmp     puts
.L1:
    rep ret
```


13.1. ПРОСТОЙ ПРИМЕР

```
.L7:
    mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
    jmp     puts
```

Мы здесь также видим `JMP puts` вместо `CALL puts / RETN`. Этот прием описан немного позже: [14.1.1](#) (стр. 150).

Нужно сказать, что x86-код такого типа редок. MSVC 2012, как видно, не может генерировать подобное. С другой стороны, программисты на ассемблере прекрасно осведомлены о том, что инструкции `Jcc` можно располагать последовательно.

Так что если вы видите это где-то, имеется немалая вероятность, что этот фрагмент кода был написан вручную.

Функция `f_unsigned()` получилась не настолько эстетически короткой:

Листинг 13.5: GCC 4.8.1 `f_unsigned()`

```
f_unsigned:
    push    esi
    push    ebx
    sub     esp, 20
    mov     esi, DWORD PTR [esp+32]
    mov     ebx, DWORD PTR [esp+36]
    cmp     esi, ebx
    ja      .L13
    cmp     esi, ebx      ; эту инструкцию можно было бы убрать
    je      .L14
.L10:
    jb      .L15
    add     esp, 20
    pop     ebx
    pop     esi
    ret
.L15:
    mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
    add     esp, 20
    pop     ebx
    pop     esi
    jmp     puts
.L13:
    mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
    call    puts
    cmp     esi, ebx
    jne     .L10
.L14:
    mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
    add     esp, 20
    pop     ebx
    pop     esi
    jmp     puts
```

Тем не менее, здесь 2 инструкции `CMP` вместо трех.

Так что, алгоритмы оптимизации GCC 4.8.1, наверное, ещё пока не идеальны.

13.1.2. ARM

32-битный ARM

Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 13.6: Оптимизирующий Keil 6/2013 (Режим ARM)

```
.text:000000B8          EXPORT f_signed
.text:000000B8          f_signed      ; CODE XREF: main+C
.text:000000B8 70 40 2D E9      STMFD    SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1      MOV     R4, R1
.text:000000C0 04 00 50 E1      CMP     R0, R4
.text:000000C4 00 50 A0 E1      MOV     R5, R0
.text:000000C8 1A 0E 8F C2      ADRGT   R0, aAB      ; "a>b\n"
```

13.1. ПРОСТОЙ ПРИМЕР

```
.text:000000CC A1 18 00 CB      BLGT    __2printf
.text:000000D0 04 00 55 E1      CMP     R5, R4
.text:000000D4 67 0F 8F 02      ADREQ   R0, aAB_0      ; "a==b\n"
.text:000000D8 9E 18 00 0B      BLEQ    __2printf
.text:000000DC 04 00 55 E1      CMP     R5, R4
.text:000000E0 70 80 BD A8      LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8      LDMFD   SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2      ADR     R0, aAB_1      ; "a<b\n"
.text:000000EC 99 18 00 EA      B       __2printf
.text:000000EC          ; End of function f_signed
```

Многие инструкции в режиме ARM могут быть исполнены только при некоторых выставленных флагах.

Это нередко используется для сравнения чисел.

К примеру, инструкция **ADD** на самом деле называется **ADDAL** внутри, **AL** означает *Always*, то есть, исполнять всегда. Предикаты кодируются в 4-х старших битах инструкции 32-битных ARM-инструкций (*condition field*). Инструкция безусловного перехода **B** на самом деле условная и кодируется так же, как и прочие инструкции условных переходов, но имеет **AL** в *condition field*, то есть исполняется всегда (*execute Always*), игнорируя флаги.

Инструкция **ADRGT** работает так же, как и **ADR**, но исполняется только в случае, если предыдущая инструкция **CMP**, сравнивая два числа, обнаруживает, что одно из них больше второго (*Greater Than*).

Следующая инструкция **BLGT** ведет себя так же, как и **BL** и сработает, только если результат сравнения был такой же (*Greater Than*). **ADRGT** записывает в **R0** указатель на строку **a>b\n**, а **BLGT** вызывает **printf()**. Следовательно, эти инструкции с суффиксом **-GT** исполнятся только в том случае, если значение в **R0** (там *a*) было больше, чем значение в **R4** (там *b*).

Далее мы увидим инструкции **ADREQ** и **BLEQ**. Они работают так же, как и **ADR** и **BL**, но исполняются только если значения при последнем сравнении были равны. Перед ними расположен ещё один **CMP**, потому что вызов **printf()** мог испортить состояние флагов.

Далее мы увидим **LDMGEFD**. Эта инструкция работает так же, как и **LDMFD**¹, но сработает только если в результате сравнения одно из значений было больше или равно второму (*Greater or Equal*). Смысл инструкции **LDMGEFD SP!, {R4-R6,PC}** в том, что это как бы эпилог функции, но он сработает только если $a \geq b$, только тогда работа функции закончится.

Но если это не так, то есть $a < b$, то исполнение дойдет до следующей инструкции **LDMFD SP!, {R4-R6,LR}**. Это ещё один эпилог функции. Эта инструкция восстанавливает состояние регистров **R4-R6**, но и **LR** вместо **PC**, таким образом, пока что, не делая возврата из функции.

Последние две инструкции вызывают **printf()** со строкой «a<b\n» в качестве единственного аргумента. Безусловный переход на **printf()** вместо возврата из функции мы уже рассматривали в секции «printf() с несколькими аргументами» (7.2.1 (стр. 49)).

Функция **f_unsigned** точно такая же, но там используются инструкции **ADRH**, **BLH**, и **LDMCSFD**. Эти предикаты (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) аналогичны рассмотренным, но служат для работы с беззнаковыми значениями.

В функции **main()** ничего нового для нас нет:

Листинг 13.7: **main()**

```
.text:00000128          EXPORT main
.text:00000128          main
.text:00000128 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:0000012C 02 10 A0 E3      MOV     R1, #2
.text:00000130 01 00 A0 E3      MOV     R0, #1
.text:00000134 DF FF FF EB      BL      f_signed
.text:00000138 02 10 A0 E3      MOV     R1, #2
.text:0000013C 01 00 A0 E3      MOV     R0, #1
.text:00000140 EA FF FF EB      BL      f_unsigned
.text:00000144 00 00 A0 E3      MOV     R0, #0
.text:00000148 10 80 BD E8      LDMFD   SP!, {R4,PC}
.text:00000148          ; End of function main
```

Так, в режиме ARM можно обойтись без условных переходов.

¹**LDMFD**

13.1. ПРОСТОЙ ПРИМЕР

Почему это хорошо? Читайте здесь: [34.1](#) (стр. 451).

В x86 нет аналогичной возможности, если не считать инструкцию `CMOVcc`, это то же что и `MOV`, но она срабатывает только при определенных выставленных флагах, обычно выставленных при помощи `CMP` во время сравнения.

Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 13.8: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:00000072      f_signed ; CODE XREF: main+6
.text:00000072 70 B5      PUSH    {R4-R6,LR}
.text:00000074 0C 00      MOVS    R4, R1
.text:00000076 05 00      MOVS    R5, R0
.text:00000078 A0 42      CMP     R0, R4
.text:0000007A 02 DD      BLE     loc_82
.text:0000007C A4 A0      ADR     R0, aAB          ; "a>b\n"
.text:0000007E 06 F0 B7 F8    BL      __2printf
.text:00000082
.text:00000082      loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42      CMP     R5, R4
.text:00000084 02 D1      BNE     loc_8C
.text:00000086 A4 A0      ADR     R0, aAB_0        ; "a==b\n"
.text:00000088 06 F0 B2 F8    BL      __2printf
.text:0000008C
.text:0000008C      loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42      CMP     R5, R4
.text:0000008E 02 DA      BGE     locret_96
.text:00000090 A3 A0      ADR     R0, aAB_1        ; "a<b\n"
.text:00000092 06 F0 AD F8    BL      __2printf
.text:00000096
.text:00000096      locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD      POP     {R4-R6,PC}
.text:00000096      ; End of function f_signed
```

В режиме Thumb только инструкции `B` могут быть дополнены условием исполнения (*condition code*), так что код для режима Thumb выглядит привычнее.

`BLE` это обычный переход с условием *Less than or Equal*, `BNE` — *Not Equal*, `BGE` — *Greater than or Equal*.

Функция `f_unsigned` точно такая же, но для работы с беззнаковыми величинами там используются инструкции `BLS` (*Unsigned lower or same*) и `BCS` (*Carry Set (Greater than or equal)*).

ARM64: Оптимизирующий GCC (Linaro) 4.9

Листинг 13.9: `f_signed()`

```
f_signed:
; w0=a, w1=b
    cmp     w0, w1
    bgt     .L19      ; Branch if Greater Than (переход, если больше чем)(a>b)
    beq     .L20      ; Branch if Equal (переход, если равно)(a==b)
    bge     .L15      ; Branch if Greater than or Equal (переход, если больше или равно)(a>=b) (здесь
это невозможно)
; a<b
    adrp    x0, .LC11      ; "a<b"
    add     x0, x0, :lo12:LC11
    b       puts
.L19:
    adrp    x0, .LC9       ; "a>b"
    add     x0, x0, :lo12:LC9
    b       puts
.L15:
; попасть сюда невозможно
    ret
.L20:
    adrp    x0, .LC10      ; "a==b"
    add     x0, x0, :lo12:LC10
    b       puts
```

```

f_unsigned:
    stp        x29, x30, [sp, -48]!
; w0=a, w1=b
    cmp        w0, w1
    add        x29, sp, 0
    str        x19, [sp,16]
    mov        w19, w0
    bhi        .L25      ; Branch if Higher (переход, если выше)(a>b)
    cmp        w19, w1
    beq        .L26      ; Branch if Equal (переход, если равно)(a==b)
.L23:
    bcc        .L27      ; Branch if Carry Clear (если нет переноса)(если меньше, чем) (a<b)
; эпилог функции, сюда попасть невозможно
    ldr        x19, [sp,16]
    ldp        x29, x30, [sp], 48
    ret
.L27:
    ldr        x19, [sp,16]
    adrp        x0, .LC11      ; "a<b"
    ldp        x29, x30, [sp], 48
    add        x0, x0, :lo12:LC11
    b          puts
.L25:
    adrp        x0, .LC9       ; "a>b"
    str        x1, [x29,40]
    add        x0, x0, :lo12:LC9
    bl         puts
    ldr        x1, [x29,40]
    cmp        w19, w1
    bne        .L23      ; Branch if Not Equal (переход, если не равно)
.L26:
    ldr        x19, [sp,16]
    adrp        x0, .LC10      ; "a==b"
    ldp        x29, x30, [sp], 48
    add        x0, x0, :lo12:LC10
    b          puts

```

Комментарии добавлены автором этой книги. В глаза бросается то, что компилятор не в курсе, что некоторые ситуации невозможны, поэтому кое-где в функциях остается код, который никогда не исполнится.

Упражнение

Попробуйте вручную оптимизировать функции по размеру, убрав избыточные инструкции и не добавляя новых.

13.1.3. MIPS

Одна отличительная особенность MIPS это отсутствие регистра флагов. Очевидно, так было сделано для упрощения анализа зависимости данных (data dependency).

Так что здесь есть инструкция, похожая на **SETcc** в x86: **SLT** («Set on Less Than» — установить если меньше чем, знаковая версия) и **SLTU** (беззнаковая версия). Эта инструкция устанавливает регистр-получатель в 1 если условие верно или в 0 в противном случае.

Затем регистр-получатель проверяется, используя инструкцию **BEQ** («Branch on Equal» — переход если равно) или **BNE** («Branch on Not Equal» — переход если не равно) и может произойти переход. Так что эта пара инструкций должна использоваться в MIPS для сравнения и перехода. Начнем с знаковой версии нашей функции:

Листинг 13.11: Неоптимизирующий GCC 4.4.5 (IDA)

```

.text:00000000 f_signed:                                # CODE XREF: main+18
.text:00000000
.text:00000000 var_10                                  = -0x10
.text:00000000 var_8                                   = -8
.text:00000000 var_4                                   = -4
.text:00000000 arg_0                                   = 0

```

13.1. ПРОСТОЙ ПРИМЕР

```
.text:00000000 arg_4          = 4
.text:00000000
.text:00000000          addiu   $sp, -0x20
.text:00000004          sw      $ra, 0x20+var_4($sp)
.text:00000008          sw      $fp, 0x20+var_8($sp)
.text:0000000C          move    $fp, $sp
.text:00000010          la      $gp, __gnu_local_gp
.text:00000018          sw      $gp, 0x20+var_10($sp)
; сохранить входные значения в локальном стеке:
.text:0000001C          sw      $a0, 0x20+arg_0($fp)
.text:00000020          sw      $a1, 0x20+arg_4($fp)
; reload them.
.text:00000024          lw      $v1, 0x20+arg_0($fp)
.text:00000028          lw      $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text:0000002C          or      $at, $zero ; NOP
; это псевдоинструкция. на самом деле, там "slt $v0,$v0,$v1" .
; так что $v0 будет установлен в 1, если $v0<$v1 (b<a) или в 0 в противном случае:
.text:00000030          slt     $v0, $v1
; перейти на loc_5c, если условие не верно.
; это псевдоинструкция. на самом деле, там "beq $v0,$zero,loc_5c" :
.text:00000034          beqz    $v0, loc_5c
; вывести "a>b" и выйти
.text:00000038          or      $at, $zero ; branch delay slot, NOP
.text:0000003C          lui     $v0, (unk_230 >> 16) # "a>b"
.text:00000040          addiu   $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text:00000044          lw      $v0, (puts & 0xFFFF)($gp)
.text:00000048          or      $at, $zero ; NOP
.text:0000004C          move    $t9, $v0
.text:00000050          jalr    $t9
.text:00000054          or      $at, $zero ; branch delay slot, NOP
.text:00000058          lw      $gp, 0x20+var_10($fp)
.text:0000005C
.text:0000005C loc_5c:          # CODE XREF: f_signed+34
.text:0000005C          lw      $v1, 0x20+arg_0($fp)
.text:00000060          lw      $v0, 0x20+arg_4($fp)
.text:00000064          or      $at, $zero ; NOP
; проверить a==b, перейти на loc_90, если это не так:
.text:00000068          bne     $v1, $v0, loc_90
.text:0000006C          or      $at, $zero ; branch delay slot, NOP
; условие верно, вывести "a==b" и закончить:
.text:00000070          lui     $v0, (aAB >> 16) # "a==b"
.text:00000074          addiu   $a0, $v0, (aAB & 0xFFFF) # "a==b"
.text:00000078          lw      $v0, (puts & 0xFFFF)($gp)
.text:0000007C          or      $at, $zero ; NOP
.text:00000080          move    $t9, $v0
.text:00000084          jalr    $t9
.text:00000088          or      $at, $zero ; branch delay slot, NOP
.text:0000008C          lw      $gp, 0x20+var_10($fp)
.text:00000090
.text:00000090 loc_90:          # CODE XREF: f_signed+68
.text:00000090          lw      $v1, 0x20+arg_0($fp)
.text:00000094          lw      $v0, 0x20+arg_4($fp)
.text:00000098          or      $at, $zero ; NOP
; проверить условие $v1<$v0 (a<b), установить $v0 в 1, если условие верно:
.text:0000009C          slt     $v0, $v1, $v0
; если условие не верно (т.е. $v0==0), перейти на loc_c8:
.text:000000A0          beqz    $v0, loc_c8
.text:000000A4          or      $at, $zero ; branch delay slot, NOP
; условие верно, вывести "a<b" и закончить
.text:000000A8          lui     $v0, (aAB_0 >> 16) # "a<b"
.text:000000AC          addiu   $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"
.text:000000B0          lw      $v0, (puts & 0xFFFF)($gp)
.text:000000B4          or      $at, $zero ; NOP
.text:000000B8          move    $t9, $v0
.text:000000BC          jalr    $t9
.text:000000C0          or      $at, $zero ; branch delay slot, NOP
.text:000000C4          lw      $gp, 0x20+var_10($fp)
.text:000000C8
```

13.1. ПРОСТОЙ ПРИМЕР

```
; все 3 условия были неверны, так что просто заканчиваем:
.text:000000C8 loc_C8:                                # CODE XREF: f_signed+A0
.text:000000C8                move    $sp, $fp
.text:000000CC                lw      $ra, 0x20+var_4($sp)
.text:000000D0                lw      $fp, 0x20+var_8($sp)
.text:000000D4                addiu   $sp, 0x20
.text:000000D8                jr      $ra
.text:000000DC                or      $at, $zero ; branch delay slot, NOP
.text:000000DC                # End of function f_signed
```

SLT REG0, REG0, REG1 сокращается в IDA до более короткой формы SLT REG0, REG1. Мы также видим здесь псевдоинструкцию BEQZ («Branch if Equal to Zero» – переход если равно нулю), которая, на самом деле, BEQ REG, \$ZERO, LABEL.

Беззнаковая версия точно такая же, только здесь используется SLTU (беззнаковая версия, отсюда «U» в названии) вместо SLT:

Листинг 13.12: Неоптимизирующий GCC 4.4.5 (IDA)

```
.text:000000E0 f_unsigned:                            # CODE XREF: main+28
.text:000000E0
.text:000000E0 var_10                = -0x10
.text:000000E0 var_8                 = -8
.text:000000E0 var_4                 = -4
.text:000000E0 arg_0                 = 0
.text:000000E0 arg_4                 = 4
.text:000000E0
.text:000000E0                addiu   $sp, -0x20
.text:000000E4                sw      $ra, 0x20+var_4($sp)
.text:000000E8                sw      $fp, 0x20+var_8($sp)
.text:000000EC                move    $fp, $sp
.text:000000F0                la      $gp, __gnu_local_gp
.text:000000F8                sw      $gp, 0x20+var_10($sp)
.text:000000FC                sw      $a0, 0x20+arg_0($fp)
.text:00000100                sw      $a1, 0x20+arg_4($fp)
.text:00000104                lw      $v1, 0x20+arg_0($fp)
.text:00000108                lw      $v0, 0x20+arg_4($fp)
.text:0000010C                or      $at, $zero
.text:00000110                sltu    $v0, $v1
.text:00000114                beqz    $v0, loc_13C
.text:00000118                or      $at, $zero
.text:0000011C                lui     $v0, (unk_230 >> 16)
.text:00000120                addiu   $a0, $v0, (unk_230 & 0xFFFF)
.text:00000124                lw      $v0, (puts & 0xFFFF)($gp)
.text:00000128                or      $at, $zero
.text:0000012C                move    $t9, $v0
.text:00000130                jalr    $t9
.text:00000134                or      $at, $zero
.text:00000138                lw      $gp, 0x20+var_10($fp)
.text:0000013C
.text:0000013C loc_13C:                            # CODE XREF: f_unsigned+34
.text:0000013C                lw      $v1, 0x20+arg_0($fp)
.text:00000140                lw      $v0, 0x20+arg_4($fp)
.text:00000144                or      $at, $zero
.text:00000148                bne     $v1, $v0, loc_170
.text:0000014C                or      $at, $zero
.text:00000150                lui     $v0, (aAB >> 16) # "a==b"
.text:00000154                addiu   $a0, $v0, (aAB & 0xFFFF) # "a==b"
.text:00000158                lw      $v0, (puts & 0xFFFF)($gp)
.text:0000015C                or      $at, $zero
.text:00000160                move    $t9, $v0
.text:00000164                jalr    $t9
.text:00000168                or      $at, $zero
.text:0000016C                lw      $gp, 0x20+var_10($fp)
.text:00000170
.text:00000170 loc_170:                            # CODE XREF: f_unsigned+68
.text:00000170                lw      $v1, 0x20+arg_0($fp)
.text:00000174                lw      $v0, 0x20+arg_4($fp)
.text:00000178                or      $at, $zero
.text:0000017C                sltu    $v0, $v1, $v0
.text:00000180                beqz    $v0, loc_1A8
```

13.2. ВЫЧИСЛЕНИЕ АБСОЛЮТНОЙ ВЕЛИЧИНЫ

```
.text:00000184          or      $at, $zero
.text:00000188          lui     $v0, (aAB_0 >> 16) # "a<b"
.text:0000018C          addiu   $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"
.text:00000190          lw      $v0, (puts & 0xFFFF)($gp)
.text:00000194          or      $at, $zero
.text:00000198          move    $t9, $v0
.text:0000019C          jalr    $t9
.text:000001A0          or      $at, $zero
.text:000001A4          lw      $gp, 0x20+var_10($fp)
.text:000001A8          loc_1A8:          # CODE XREF: f_unsigned+A0
.text:000001A8          move    $sp, $fp
.text:000001AC          lw      $ra, 0x20+var_4($sp)
.text:000001B0          lw      $fp, 0x20+var_8($sp)
.text:000001B4          addiu   $sp, 0x20
.text:000001B8          jr      $ra
.text:000001BC          or      $at, $zero
.text:000001BC          # End of function f_unsigned
```

13.2. Вычисление абсолютной величины

Это простая функция:

```
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
};
```

13.2.1. Оптимизирующий MSVC

Обычный способ генерации кода:

Листинг 13.13: Оптимизирующий MSVC 2012 x64

```
i$ = 8
my_abs PROC
; ECX = input
    test    ecx, ecx
; проверить знак входного значения
; пропустить инструкцию NEG если знак положительный
    jns     SHORT $LN2@my_abs
; поменять знак
    neg     ecx
$LN2@my_abs:
; подготовить результат в EAX:
    mov     eax, ecx
    ret     0
my_abs ENDP
```

GCC 4.9 делает почти то же самое.

13.2.2. Оптимизирующий Keil 6/2013: Режим Thumb

Листинг 13.14: Оптимизирующий Keil 6/2013: Режим Thumb

```
my_abs PROC
    CMP     r0, #0
; входное значение равно нулю или больше нуля?
; в таком случае, пропустить инструкцию RSBS
    BGE     |L0.6|
; отнять входное значение от 0 0:
```

13.2. ВЫЧИСЛЕНИЕ АБСОЛЮТНОЙ ВЕЛИЧИНЫ

L0.6	RSBS	r0,r0,#0
	BX	lr
	ENDP	

В ARM нет инструкции для изменения знака, так что компилятор Keil использует инструкцию «Reverse Subtract», которая просто вычитает, но с операндами, переставленными наоборот.

13.2.3. Оптимизирующий Keil 6/2013: Режим ARM

В режиме ARM можно добавлять коды условий к некоторым инструкций, что компилятор Keil и сделал:

Листинг 13.15: Оптимизирующий Keil 6/2013: Режим ARM

```
my_abs PROC
    CMP        r0,#0
; исполнить инструкцию "Reverse Subtract" только в случае, если входное значение меньше 0:
    RSBLT     r0,r0,#0
    BX        lr
    ENDP
```

Теперь здесь нет условных переходов и это хорошо:

[34.1](#) (стр. 451).

13.2.4. Неоптимизирующий GCC 4.9 (ARM64)

В ARM64 есть инструкция **NEG** для смены знака:

Листинг 13.16: Оптимизирующий GCC 4.9 (ARM64)

```
my_abs:
    sub     sp, sp, #16
    str     w0, [sp,12]
    ldr     w0, [sp,12]
; сравнить входное значение с содержимым регистра WZR
; (который всегда содержит ноль)
    cmp     w0, wzr
    bge     .L2
    ldr     w0, [sp,12]
    neg     w0, w0
    b       .L3
.L2:
    ldr     w0, [sp,12]
.L3:
    add     sp, sp, 16
    ret
```

13.2.5. MIPS

Листинг 13.17: Оптимизирующий GCC 4.4.5 (IDA)

```
my_abs:
; перейти если $a0<0:
    bltz    $a0, locret_10
; просто вернуть входное значение ($a0) в $v0:
    move    $v0, $a0
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP
locret_10:
; поменять у значения знак и сохранить его в $v0:
    jr      $ra
; это псевдоинструкция. на самом деле, это "subu $v0,$zero,$a0" ($v0=0-$a0)
    negu    $v0, $a0
```


13.3. ТЕРНАРНЫЙ УСЛОВНЫЙ ОПЕРАТОР

Видим здесь новую инструкцию: **BLTZ** («Branch if Less Than Zero»). Тут есть также псевдоинструкция **NEGU**, которая на самом деле вычитает из нуля. Суффикс «U» в обеих инструкциях **SUBU** и **NEGU** означает, что при целочисленном переполнении исключение не сработает.

13.2.6. Версия без переходов?

Возможна также версия и без переходов, мы рассмотрим её позже: [46](#) (стр. [508](#)).

13.3. Тернарный условный оператор

Тернарный условный оператор (ternary conditional operator) в Си/Си++ это:

```
expression ? expression : expression
```

И вот пример:

```
const char* f (int a)
{
    return a==10 ? "it is ten" : "it is not ten";
};
```

13.3.1. x86

Старые и неоптимизирующие компиляторы генерируют код так, как если бы выражение **if/else** было использовано вместо него:

Листинг 13.18: Неоптимизирующий MSVC 2008

```
$SG746 DB      'it is ten', 00H
$SG747 DB      'it is not ten', 00H

tv65 = -4 ; будет использовано как временная переменная
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
; сравнить входное значение с 10
        cmp     DWORD PTR _a$[ebp], 10
; переход на $LN3@f если не равно
        jne     SHORT $LN3@f
; сохранить указатель на строку во временной переменной:
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; перейти на выход
        jmp     SHORT $LN4@f
$LN3@f:
; сохранить указатель на строку во временной переменной:
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; это выход. скопировать указатель на строку из временной переменной в EAX.
        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Листинг 13.19: Оптимизирующий MSVC 2008

```
$SG792 DB      'it is ten', 00H
$SG793 DB      'it is not ten', 00H

_a$ = 8 ; size = 4
_f      PROC
; сравнить входное значение с 10
```

13.3. ТЕРНАРНЫЙ УСЛОВНЫЙ ОПЕРАТОР

```
    cmp     DWORD PTR _a$[esp-4], 10
    mov     eax, OFFSET $SG792 ; 'it is ten'
; переход на $LN4@f если равно
    je      SHORT $LN4@f
    mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
    ret     0
_f       ENDP
```

Новые компиляторы могут быть более краткими:

Листинг 13.20: Оптимизирующий MSVC 2012 x64

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
f       PROC
; загрузить указатели на обе строки
    lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
    lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; сравнить входное значение с 10
    cmp     ecx, 10
; если равно, скопировать значение из RDX ("it is ten")
; если нет, ничего не делаем. указатель на строку "it is not ten" всё еще в RAX.
    cmov     rax, rdx
    ret     0
f       ENDP
```

Оптимизирующий GCC 4.8 для x86 также использует инструкцию **CMOVcc**, тогда как неоптимизирующий GCC 4.8 использует условные переходы.

13.3.2. ARM

Оптимизирующий Keil для режима ARM также использует инструкцию **ADRcc**, срабатывающую при некотором условии:

Листинг 13.21: Оптимизирующий Keil 6/2013 (Режим ARM)

```
f PROC
; сравнить входное значение с 10
    CMP     r0,#0xa
; если результат сравнения Equal (равно), скопировать указатель на строку "it is ten" в R0
    ADREQ   r0,|L0.16| ; "it is ten"
; если результат сравнения Not Equal (не равно), скопировать указатель на строку "it is not ten" в R0
    ADRNE   r0,|L0.28| ; "it is not ten"
    BX      lr
    ENDP

|L0.16|
    DCB     "it is ten",0
|L0.28|
    DCB     "it is not ten",0
```

Без внешнего вмешательства инструкции **ADREQ** и **ADRNE** никогда не исполнятся одновременно. Оптимизирующий Keil для режима Thumb вынужден использовать инструкции условного перехода, потому что тут нет инструкции загрузки значения, поддерживающей флаги условия:

Листинг 13.22: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
f PROC
; сравнить входное значение с 10
    CMP     r0,#0xa
; переход на |L0.8| если Equal (равно)
    BEQ     |L0.8|
    ADR     r0,|L0.12| ; "it is not ten"
    BX      lr
|L0.8|
    ADR     r0,|L0.28| ; "it is ten"
    BX      lr
```

13.3. ТЕРНАРНЫЙ УСЛОВНЫЙ ОПЕРАТОР

```
ENDP

|L0.12|
DCB    "it is not ten",0
|L0.28|
DCB    "it is ten",0
```

13.3.3. ARM64

Оптимизирующий GCC (Linaro) 4.9 для ARM64 также использует условные переходы:

Листинг 13.23: Оптимизирующий GCC (Linaro) 4.9

```
f:
    cmp    x0, 10
    beq    .L3          ; branch if equal (переход, если равно)
    adrp   x0, .LC1      ; "it is ten"
    add    x0, x0, :lo12:.LC1
    ret

.L3:
    adrp   x0, .LC0      ; "it is not ten"
    add    x0, x0, :lo12:.LC0
    ret

.LC0:
    .string "it is ten"

.LC1:
    .string "it is not ten"
```

Это потому что в ARM64 нет простой инструкции загрузки с флагами условия, как `ADRCc` в 32-битном режиме ARM или `CMOVcc` в x86.

Но с другой стороны, там есть инструкция `CSEL` («Conditional SElect») [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)р390, C5.5], но GCC 4.9 наверное, пока не так хорош, чтобы генерировать её в таком фрагменте кода

13.3.4. MIPS

GCC 4.4.5 для MIPS тоже не так хорош, к сожалению:

Листинг 13.24: Оптимизирующий GCC 4.4.5 (вывод на ассемблере)

```
$LC0:
    .ascii "it is not ten\000"
$LC1:
    .ascii "it is ten\000"
f:
    li     $2,10          # 0xa
; сравнить $a0 и 10, переход, если равно:
    beq    $4,$2,$L2
    nop    ; branch delay slot

; оставить адрес строки "it is not ten" в $v0 и выйти:
    lui    $2,%hi($LC0)
    j      $31
    addiu  $2,$2,%lo($LC0)

$L2:
; оставить адрес строки "it is ten" в $v0 и выйти:
    lui    $2,%hi($LC1)
    j      $31
    addiu  $2,$2,%lo($LC1)
```

13.3.5. Перепишем, используя обычный if/else

```
const char* f (int a)
{
    if (a==10)
        return "it is ten";
    else
        return "it is not ten";
};
```

Интересно, оптимизирующий GCC 4.8 для x86 также может генерировать `CMOVcc` в этом случае:

Листинг 13.25: Оптимизирующий GCC 4.8

```
.LC0:
    .string "it is ten"
.LC1:
    .string "it is not ten"
f:
.LFB0:
; сравнить входное значение с 10
    cmp     DWORD PTR [esp+4], 10
    mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
    mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; если результат сравнение Not Equal (не равно), скопировать значение из EDX в EAX
; а если нет, то ничего не делать
    cmovne  eax, edx
    ret
```

Оптимизирующий Keil в режиме ARM генерирует код идентичный этому: листинг.13.21.

Но оптимизирующий MSVC 2012 пока не так хорош.

13.3.6. Вывод

Почему оптимизирующие компиляторы стараются избавиться от условных переходов? Читайте больше об этом здесь: [34.1](#) (стр. 451).

13.4. Поиск минимального и максимального значения**13.4.1. 32-bit**

```
int my_max(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
};

int my_min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
};
```

Листинг 13.26: Неоптимизирующий MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
    push    ebp
    mov     ebp, esp
```

13.4. ПОИСК МИНИМАЛЬНОГО И МАКСИМАЛЬНОГО ЗНАЧЕНИЯ

```
        mov     eax, DWORD PTR _a$[ebp]
; сравнить A и B:
        cmp     eax, DWORD PTR _b$[ebp]
; переход, если A больше или равно B:
        jge     SHORT $LN2@my_min
; перезагрузить A в EAX в противном случае и перейти на выход
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_min
        jmp     SHORT $LN3@my_min ; это избыточная JMP
$LN2@my_min:
; возврат B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min:
        pop     ebp
        ret     0
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; сравнить A и B:
        cmp     eax, DWORD PTR _b$[ebp]
; переход, если A меньше или равно B:
        jle     SHORT $LN2@my_max
; перезагрузить A в EAX в противном случае и перейти на выход
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_max
        jmp     SHORT $LN3@my_max ; это избыточная JMP
$LN2@my_max:
; возврат B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
        pop     ebp
        ret     0
_my_max ENDP
```

Эти две функции отличаются друг от друга только инструкцией условного перехода: **JGE** («Jump if Greater or Equal» — переход если больше или равно) используется в первой и **JLE** («Jump if Less or Equal» — переход если меньше или равно) во второй.

Здесь есть ненужная инструкция **JMP** в каждой функции, которую MSVC, наверное, оставил по ошибке.

Без переходов

ARM в режиме Thumb напоминает нам x86-код:

Листинг 13.27: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
my_max PROC
; R0=A
; R1=B
; сравнить A и B:
        CMP     r0,r1
; переход, если A больше B:
        BGT     |L0.6|
; в противном случае (A<=B) возврат R1 (B):
        MOVS    r0,r1
|L0.6|
; возврат
        BX      lr
        ENDP

my_min PROC
; R0=A
; R1=B
; сравнить A и B:
```

13.4. ПОИСК МИНИМАЛЬНОГО И МАКСИМАЛЬНОГО ЗНАЧЕНИЯ

```
        CMP        r0,r1
; переход, если A меньше B:
        BLT        |L0.14|
; в противном случае (A>=B) возврат R1 (B):
        MOVS       r0,r1
|L0.14|
; возврат
        BX         lr
        ENDP
```

Функции отличаются только инструкцией перехода: **BGT** и **BLT**. А в режиме ARM можно использовать условные суффиксы, так что код более плотный. **MOVcc** будет исполнена только если условие верно:

Листинг 13.28: Оптимизирующий Keil 6/2013 (Режим ARM)

```
my_max PROC
; R0=A
; R1=B
; сравнить A и B:
        CMP        r0,r1
; вернуть B вместо A копируя B в R0
; эта инструкция сработает только если A<=B (т.е. LE - Less or Equal, меньше или равно)
; если инструкция не сработает (в случае A>B), A всё еще в регистре R0
        MOVLE      r0,r1
        BX         lr
        ENDP

my_min PROC
; R0=A
; R1=B
; сравнить A и B:
        CMP        r0,r1
; вернуть B вместо A копируя B в R0
; эта инструкция сработает только если A>=B (т.е. GE - Greater or Equal, больше или равно)
; если инструкция не сработает (в случае A<B), A всё еще в регистре R0
        MOVGE      r0,r1
        BX         lr
        ENDP
```

Оптимизирующий GCC 4.8.1 и оптимизирующий MSVC 2013 могут использовать инструкцию **CMOVcc**, которая аналогична **MOVcc** в ARM:

Листинг 13.29: Оптимизирующий MSVC 2013

```
my_max:
        mov        edx, DWORD PTR [esp+4]
        mov        eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; сравнить A и B:
        cmp        edx, eax
; если A>=B, загрузить значение A в EAX
; в противном случае, эта инструкция ничего не делает (если A<B)
        cmovge    eax, edx
        ret

my_min:
        mov        edx, DWORD PTR [esp+4]
        mov        eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; сравнить A и B:
        cmp        edx, eax
; если A<=B, загрузить значение A в EAX
; в противном случае, эта инструкция ничего не делает (если A>B)
        cmovle    eax, edx
        ret
```

13.4.2. 64-bit

```
#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
    if (a>b)
        return a;
    else
        return b;
};

int64_t my_min(int64_t a, int64_t b)
{
    if (a<b)
        return a;
    else
        return b;
};
```

Тут есть ненужные перетасовки значений, но код в целом понятен:

Листинг 13.30: Неоптимизирующий GCC 4.9.1 ARM64

```
my_max:
    sub    sp, sp, #16
    str    x0, [sp,8]
    str    x1, [sp]
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    cmp    x1, x0
    ble    .L2
    ldr    x0, [sp,8]
    b      .L3
.L2:
    ldr    x0, [sp]
.L3:
    add    sp, sp, 16
    ret

my_min:
    sub    sp, sp, #16
    str    x0, [sp,8]
    str    x1, [sp]
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    cmp    x1, x0
    bge    .L5
    ldr    x0, [sp,8]
    b      .L6
.L5:
    ldr    x0, [sp]
.L6:
    add    sp, sp, 16
    ret
```

Без переходов

Нет нужды загружать аргументы функции из стека, они уже в регистрах:

Листинг 13.31: Оптимизирующий GCC 4.9.1 x64

```
my_max:
; RDI=A
; RSI=B
; сравнить A и B:
    cmp    rdi, rsi
; подготовить B в RAX для возврата:
    mov    rax, rsi
```

13.4. ПОИСК МИНИМАЛЬНОГО И МАКСИМАЛЬНОГО ЗНАЧЕНИЯ

```
; если A>=B, оставить A (RDI) в RAX для возврата.  
; в противном случае, инструкция ничего не делает (если A<B)  
    cmovge    rax, rdi  
    ret  
  
my_min:  
; RDI=A  
; RSI=B  
; сравнить A и B:  
    cmp      rdi, rsi  
; подготовить B в RAX для возврата:  
    mov      rax, rsi  
; если A<=B, оставить A (RDI) в RAX для возврата.  
; в противном случае, инструкция ничего не делает (если A>B)  
    cmovle    rax, rdi  
    ret
```

MSVC 2013 делает то же самое.

В ARM64 есть инструкция **CSEL**, которая работает точно также, как и **MOVcc** в ARM и **CMOVcc** в x86, но название другое: «Conditional SElect».

Листинг 13.32: Оптимизирующий GCC 4.9.1 ARM64

```
my_max:  
; X0=A  
; X1=B  
; сравнить A и B:  
    cmp      x0, x1  
; выбрать X0 (A) в X0 если X0>=X1 или A>=B (Greater or Equal: больше или равно)  
; выбрать X1 (B) в X0 если A<B  
    csel     x0, x0, x1, ge  
    ret  
  
my_min:  
; X0=A  
; X1=B  
; сравнить A и B:  
    cmp      x0, x1  
; выбрать X0 (A) в X0 если X0<=X1 (Less or Equal: меньше или равно)  
; выбрать X1 (B) в X0 если A>B  
    csel     x0, x0, x1, le  
    ret
```

13.4.3. MIPS

А GCC 4.4.5 для MIPS не так хорош, к сожалению:

Листинг 13.33: Оптимизирующий GCC 4.4.5 (IDA)

```
my_max:  
; установить $v1 to 1, ifb 1, если $a1<$a0:  
    slt      $v1, $a1, $a0  
; переход, если $a1<$a0:  
    beqz     $v1, locret_10  
; это branch delay slot  
; подготовить $a1 в $v0 на случай, если переход сработает:  
    move     $v0, $a1  
; переход не сработал, подготовить $a0 в $v0:  
    move     $v0, $a0  
  
locret_10:  
    jr      $ra  
    or      $at, $zero ; branch delay slot, NOP  
  
; функция min() точно такая же, но входные операнды в инструкции SLT поменены местами:  
my_min:  
    slt      $v1, $a0, $a1  
    beqz     $v1, locret_28
```


13.5. Вывод

```
        move    $v0, $a1
        move    $v0, $a0

locret_28:
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP
```

Не забывайте о *branch delay slots*: первая **MOVE** выполняется *перед* **BEQZ**, вторая **MOVE** выполняется только если переход не произошел.

13.5. Вывод

13.5.1. x86

Примерный скелет условных переходов:

Листинг 13.34: x86

```
CMP register, register/value
Jcc true ; cскод= условия
false:
... код, исполняющийся, если сравнение ложно ...
JMP exit
true:
... код, исполняющийся, если сравнение истинно ...
exit:
```

13.5.2. ARM

Листинг 13.35: ARM

```
CMP register, register/value
Bcc true ; cскод= условия
false:
... код, исполняющийся, если сравнение ложно ...
JMP exit
true:
... код, исполняющийся, если сравнение истинно ...
exit:
```

13.5.3. MIPS

Листинг 13.36: Проверка на ноль

```
BEQZ REG, label
...
```

Листинг 13.37: Меньше ли нуля?

```
BLTZ REG, label
...
```

Листинг 13.38: Проверка на равенство

```
BEQ REG1, REG2, label
...
```

Листинг 13.39: Проверка на неравенство

```
BNE REG1, REG2, label
...
```

Листинг 13.40: Проверка на меньше, больше (знаковое)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

Листинг 13.41: Проверка на меньше, больше (беззнаковое)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

13.5.4. Без инструкций перехода

Если тело условного выражения очень короткое, может быть использована инструкция условного копирования: **MOVcc** в ARM (в режиме ARM), **CSEL** в ARM64, **CMOVcc** в x86.

ARM

В режиме ARM можно использовать условные суффиксы для некоторых инструкций:

Листинг 13.42: ARM (Режим ARM)

```
CMP register, register/value
instr1_cc ; инструкция, которая будет исполнена, если условие истинно
instr2_cc ; еще инструкция, которая будет исполнена, если условие истинно
... итд~.....
```

Нет никаких ограничений на количество инструкций с условными суффиксами до тех пор, пока флаги CPU не были модифицированы одной из таких инструкций.

В режиме Thumb есть инструкция **IT**, позволяющая дополнить следующие 4 инструкции суффиксами, задающими условие.

Читайте больше об этом: [18.7.2](#) (стр. 259).

Листинг 13.43: ARM (Режим Thumb)

```
CMP register, register/value
ITEEE EQ ; выставить такие суффиксы: if-then-else-else-else
instr1 ; инструкция будет исполнена, если истинно
instr2 ; инструкция будет исполнена, если ложно
instr3 ; инструкция будет исполнена, если ложно
instr4 ; инструкция будет исполнена, если ложно
```

13.6. Упражнение

(ARM64) Попробуйте переписать код в листинг [13.23](#) убрав все инструкции условного перехода, и используйте инструкцию **CSEL**.

Глава 14

switch()/case/default

14.1. Если вариантов мало

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
        default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

14.1.1. x86

Неоптимизирующий MSVC

Это дает в итоге (MSVC 2010):

Листинг 14.1: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 0
    je      SHORT $LN4@f
    cmp     DWORD PTR tv64[ebp], 1
    je      SHORT $LN3@f
    cmp     DWORD PTR tv64[ebp], 2
    je      SHORT $LN2@f
    jmp     SHORT $LN1@f
$LN4@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN3@f:
```

14.1. ЕСЛИ ВАРИАНТОВ МАЛО

```
push    OFFSET $SG741 ; 'one', 0aH, 00H
call    _printf
add     esp, 4
jmp     SHORT $LN7@f
$LN2@f:
push    OFFSET $SG743 ; 'two', 0aH, 00H
call    _printf
add     esp, 4
jmp     SHORT $LN7@f
$LN1@f:
push    OFFSET $SG745 ; 'something unknown', 0aH, 00H
call    _printf
add     esp, 4
$LN7@f:
mov     esp, ebp
pop     ebp
ret     0
_f      ENDP
```

Наша функция со оператором `switch()`, с небольшим количеством вариантов, это практически аналог подобной конструкции:

```
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

Когда вариантов немного и мы видим подобный код, невозможно сказать с уверенностью, был ли в оригинальном исходном коде `switch()`, либо просто набор операторов `if()`.

То есть, `switch()` это синтаксический сахар для большого количества вложенных проверок при помощи `if()`.

В самом выходном коде ничего особо нового, за исключением того, что компилятор зачем-то перекладывает входящую переменную (*a*) во временную в локальном стеке `v64`¹.

Если скомпилировать это при помощи GCC 4.4.1, то будет почти то же самое, даже с максимальной оптимизацией (ключ `-O3`).

Оптимизирующий MSVC

Попробуем включить оптимизацию кодегенератора MSVC (`/Ox`): `cl 1.c /Fa1.asm /Ox`

Листинг 14.2: MSVC

```
_a$ = 8 ; size = 4
_f    PROC
mov    eax, DWORD PTR _a$[esp-4]
sub    eax, 0
je     SHORT $LN4@f
sub    eax, 1
je     SHORT $LN3@f
sub    eax, 1
je     SHORT $LN2@f
mov    DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH, 00H
jmp    _printf
$LN2@f:
mov    DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
jmp    _printf
$LN3@f:
mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
jmp    _printf
```

¹Локальные переменные в стеке с префиксом `tv` — так MSVC называет внутренние переменные для своих нужд

14.1. ЕСЛИ ВАРИАНТОВ МАЛО

```
$LN4@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
    jmp     _printf
_f      ENDP
```

Вот здесь уже всё немного по-другому, причем не без грязных трюков.

Первое: `a` помещается в `EAX` и от него отнимается 0. Звучит абсурдно, но нужно это для того, чтобы проверить, 0 ли в `EAX` был до этого? Если да, то выставится флаг `ZF` (что означает, что результат вычитания 0 от числа стал 0) и первый условный переход `JE` (*Jump if Equal* или его синоним `JZ` — *Jump if Zero*) сработает на метку `$LN4@f`, где выводится сообщение `'zero'`. Если первый переход не сработал, от значения отнимается по единице, и если на какой-то стадии в результате образуется 0, то сработает соответствующий переход.

И в конце концов, если ни один из условных переходов не сработал, управление передается `printf()` со строковым аргументом `'something unknown'`.

Второе: мы видим две, мягко говоря, необычные вещи: указатель на сообщение помещается в переменную `a`, и затем `printf()` вызывается не через `CALL`, а через `JMP`. Объяснение этому простое. Вызывающая функция заталкивает в стек некоторое значение и через `CALL` вызывает нашу функцию. `CALL` в свою очередь заталкивает в стек адрес возврата (`RA`) и делает безусловный переход на адрес нашей функции. Наша функция в самом начале (да и в любом её месте, потому что в теле функции нет ни одной инструкции, которая меняет что-то в стеке или в `ESP`) имеет следующую разметку стека:

- `ESP` — хранится `RA`
- `ESP+4` — хранится значение `a`

С другой стороны, чтобы вызвать `printf()`, нам нужна почти такая же разметка стека, только в первом аргументе нужен указатель на строку. Что, собственно, этот код и делает.

Он заменяет свой первый аргумент на адрес строки, и затем передает управление `printf()`, как если бы вызвали не нашу функцию `f()`, а сразу `printf()`. `printf()` выводит некую строку на `stdout`, затем исполняет инструкцию `RET`, которая из стека достает `RA` и управление передается в ту функцию, которая вызвала `f()`, минуя при этом конец функции `f()`.

Всё это возможно, потому что `printf()` вызывается в `f()` в самом конце. Всё это чем-то даже похоже на `longjmp()`². И всё это, разумеется, сделано для экономии времени исполнения.

Похожая ситуация с компилятором для ARM описана в секции «`printf()` с несколькими аргументами» (7.2.1 (стр. 49)).

²[wikipedia](#)

OllyDbg

Так как этот пример немного запутанный, попробуем оттрассировать его в OllyDbg.

OllyDbg может распознавать подобные switch()-конструкции, так что он добавляет полезные комментарии. EAX в начале равен 2, это входное значение функции:

CPU - main thread, module few

Address	Hex dump	ASCII (ANSI - Cy)
00FF1000	8B 44 24 04	MOV EAX, DWORD PTR SS:[ARG.1]
00FF1004	83 E8 00	SUB EAX, 8
00FF1007	74 30	JZ SHORT 00FF1039
00FF1009	48	DEC EAX
00FF100A	74 1F	JZ SHORT 00FF102B
00FF100C	48	DEC EAX
00FF100D	74 0E	JZ SHORT 00FF101D
00FF100F	C7 44 24 04 18	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3018
00FF1017	FF 25 A0 20 FF 00	JMP DWORD PTR DS:[&MSUCR100.printf]
00FF101D	C7 44 24 04 10	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3010
00FF1025	FF 25 A0 20 FF 00	JMP DWORD PTR DS:[&MSUCR100.printf]
00FF102B	C7 44 24 04 08	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3008
00FF1033	FF 25 A0 20 FF 00	JMP DWORD PTR DS:[&MSUCR100.printf]
00FF1039	C7 44 24 04 00	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3000
00FF1041	FF 25 A0 20 FF 00	JMP DWORD PTR DS:[&MSUCR100.printf]
00FF1047	CC	INT3
00FF1048	CC	INT3
00FF1049	CC	INT3
00FF104A	CC	INT3
00FF104B	CC	INT3
00FF104C	CC	INT3

Registers (MMX)

Register	Value
EAX	00000002
ECX	6E494719
EDX	00000000
EBX	00000000
ESP	001EF84C
EBP	001EF894
ESI	00000001
EDI	00FF33A8
EIP	00FF1004

Imm=0
EAX=2

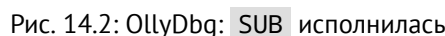
Switch (cases 0..2, 4 ex)

00FF1004: 8B 44 24 04 MOV EAX, DWORD PTR SS:[ARG.1]
00FF1007: 83 E8 00 SUB EAX, 8
00FF1009: 74 30 JZ SHORT 00FF1039
00FF100A: 48 DEC EAX
00FF100C: 74 1F JZ SHORT 00FF102B
00FF100D: 48 DEC EAX
00FF100F: 74 0E JZ SHORT 00FF101D
00FF1010: C7 44 24 04 18 MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3018
00FF1013: FF 25 A0 20 FF 00 JMP DWORD PTR DS:[&MSUCR100.printf]
00FF1016: C7 44 24 04 10 MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3010
00FF1019: FF 25 A0 20 FF 00 JMP DWORD PTR DS:[&MSUCR100.printf]
00FF101C: C7 44 24 04 08 MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3008
00FF101F: FF 25 A0 20 FF 00 JMP DWORD PTR DS:[&MSUCR100.printf]
00FF1022: C7 44 24 04 00 MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3000
00FF1025: FF 25 A0 20 FF 00 JMP DWORD PTR DS:[&MSUCR100.printf]
00FF1028: CC INT3
00FF1029: CC INT3
00FF102A: CC INT3
00FF102B: CC INT3
00FF102C: CC INT3
00FF102D: CC INT3

001EF84C: 00FF1057 RETURN from f
001EF850: 00000002
001EF854: 00FF11CA RETURN from f
001EF858: 00000001
001EF85C: 002A4E68 hN# ASCII "pN#"
001EF860: 002A2848 H(*
001EF864: 466BAC00 ankF
001EF868: 00000000
001EF86C: 00000000
001EF870: 7EFDE000 p#
001EF874: 00000000
001EF878: 00000000
001EF87C: 001EF864 d%
001EF880: 03389310 p%
001EF884: 001EF800 Pointer to ne

Рис. 14.1: OllyDbg: EAX содержит первый (и единственный) аргумент функции

0 отнимается от 2 в `EAX`. Конечно же, `EAX` всё ещё содержит 2. Но флаг `ZF` теперь 0, что означает, что последнее вычисленное значение не было нулевым:



14.1. ЕСЛИ ВАРИАНТОВ МАЛО

DEC исполнилась и EAX теперь содержит 1. Но 1 не ноль, так что флаг ZF всё ещё 0:

CPU - main thread, module few

Address	Hex dump	ASCII (ANSI - Cy)
00FF1000	8B 42 04 00	MOV EAX, DWORD PTR SS:[ARG.1]
00FF1004	83 E8 00 00	SUB EAX, 0
00FF1007	74 30 00 00	JZ SHORT 00FF1039
00FF1009	48 00 00 00	DEC EAX
00FF100A	74 1F 00 00	JZ SHORT 00FF102B
00FF100C	48 00 00 00	DEC EAX
00FF100D	74 0E 00 00	JZ SHORT 00FF101D
00FF100F	C7 44 24 04 18 00 00 00 00	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3018
00FF1017	FF 25 00 20 FF 00 00 00 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF101D	C7 44 24 04 10 00 00 00 00	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3010
00FF1025	FF 25 00 20 FF 00 00 00 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF102B	C7 44 24 04 08 00 00 00 00	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3008
00FF1033	FF 25 00 20 FF 00 00 00 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF1039	C7 44 24 04 00 00 00 00 00	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3000
00FF1041	FF 25 00 20 FF 00 00 00 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF1047	CC 00 00 00	INT3
00FF1048	CC 00 00 00	INT3
00FF1049	CC 00 00 00	INT3
00FF104A	CC 00 00 00	INT3
00FF104B	CC 00 00 00	INT3
00FF104C	CC 00 00 00	INT3

Jump is not taken
Dest=few.00FF102B

Registers (MMX)

Register	Value
EAX	00000001
ECX	6E494714 ASCII "H(*)"
EDX	00000000
EBX	00000000
ESP	001EF84C
EBP	001EF894
ESI	00000001
EDI	00FF33A8 few.00FF33A8
EIP	00FF100A few.00FF100A
EAX	00000001
ECX	00000000
EDX	00000000
EBX	00000000
ESP	00000000
EBP	00000000
ESI	00000000
EDI	00000000
EIP	00000000

MMX

Register	Value
MM0	0000 0000 0000 0000
MM1	0000 0000 0000 0000
MM2	0000 0000 0000 0000
MM3	0000 0000 0000 0000
MM4	0000 0000 0000 0000

Address **Hex dump** **ASCII (ANSI - Cy)**

Address	Hex dump	ASCII (ANSI - Cy)
00FF3000	7A 65 72 6F 0A 00 00 00 00 6F 6E 65 0A 00 00 00 00	zero one
00FF3010	74 77 6F 0A 00 00 00 00 00 73 6F 6D 65 74 68 69 6E	two something
00FF3020	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
00FF3030	FF FF FF FF 01 00 00 00 00 00 00 00 00 00 00 00	
00FF3040	FE FF FF FF 01 00 00 00 34 54 75 46 CB AB 8A B9	
00FF3050	01 00 00 00 48 28 2A 00 68 4E 2A 00 00 00 00 00	
00FF3060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

001EF84C **00FF1057** **RETURN from**

001EF850 **00000002** **RETURN from**

001EF854 **00FF11CA** **RETURN from**

001EF858 **00000001** **hN*** ASCII "pN*"

001EF85C **002A4E68** **hN*** ASCII "pN*"

001EF860 **002A2848** **H(*)** ASCII "pN*"

001EF864 **466BAC00** **ankF**

001EF868 **00000000** **pN***

001EF86C **00000000** **pN***

001EF870 **7EFDE000** **pN***

001EF874 **00000000** **pN***

001EF878 **00000000** **pN***

001EF87C **001EF864** **d°**

001EF880 **D3883810** **y8**

001EF884 **001EF880** **Pointer to**

Рис. 14.3: OllyDbg: первая DEC исполнилась

Следующая **DEC** исполнилась. **EAX** наконец 0 и флаг **ZF** выставлен, потому что результат — ноль:



155

14.1. ЕСЛИ ВАРИАНТОВ МАЛО

Указатель на строку «two» сейчас будет записан в стек:

CPU - main thread, module few

Address	Hex dump	ASCII (ANSI - Cy)
00FF1000	8B 44 24 04	MOV EAX, DWORD PTR SS:[ARG.1]
00FF1004	83 E8 00	SUB EAX, 0
00FF1007	74 30	JZ SHORT 00FF1039
00FF1009	48	DEC EAX
00FF100A	74 1F	JZ SHORT 00FF102B
00FF100C	48	DEC EAX
00FF100D	74 0E	JZ SHORT 00FF101D
00FF100F	C7 44 24 04 18	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3018: ASCII "something unknown"
00FF1010	FF 25 A0 20 FF 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF101D	C7 44 24 04 10	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3010: ASCII "two", case 2 of
00FF1025	FF 25 A0 20 FF 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF102B	C7 44 24 04 08	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3008: ASCII "one", case 1 of
00FF1033	FF 25 A0 20 FF 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF1039	C7 44 24 04 00	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3000: ASCII "zero", case 0 of
00FF1041	FF 25 A0 20 FF 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF1047	CC	INT3
00FF1048	CC	INT3
00FF1049	CC	INT3
00FF104A	CC	INT3
00FF104B	CC	INT3
00FF104C	CC	INT3

Imm=few.00FF3010, ASCII "two"
Stack [001EF850]=2
Jump from 00FF100D

Registers (MMX)

Register	Value
EAX	00000000
ECX	6E494714 ASCII "H(*)"
EDX	00000000
EBX	00000000
ESP	001EF84C
EBP	001EF894
ESI	00000001
EDI	00FF33A8 few.00FF33A8
EIP	00FF101D few.00FF101D

Stack

Address	Value
MM0	0000 0000 0000 0000
MM1	0000 0000 0000 0000
MM2	0000 0000 0000 0000
MM3	0000 0000 0000 0000
MM4	0000 0000 0000 0000

Return from

Address	Value
001EF84C	00FF1057
001EF850	00000000
001EF854	00FF11C4
001EF858	00000001
001EF85C	002A4E63
001EF860	002A2843
001EF864	466BAC90
001EF868	00000000
001EF86C	00000000
001EF870	7EFD0000
001EF874	00000000
001EF878	00000000
001EF87C	001EF864
001EF880	03389310
001EF884	001EF800

Рис. 14.5: OllyDbg: указатель на строку сейчас запишется на место первого аргумента

Обратите внимание: текущий аргумент функции это 2 и 2 прямо сейчас в стеке по адресу 0x001EF850 .

14.1. ЕСЛИ ВАРИАНТОВ МАЛО

MOV записывает указатель на строку по адресу 0x001EF850 (см. окно стека). Переход сработал. Это самая первая инструкция функции printf() в MSVCR100.DLL (этот пример был скомпилирован с опцией /MD):

CPU - main thread, module MSVCR100

Address	Hex dump	ASCII (ANSI - Cy)
00FF3000	7A 65 72 6F 0A 00 00 00 6F 6E 65 0A 00 00 00	one
00FF3010	74 77 6F 0A 00 00 00 73 6F 6D 65 74 68 69 6E	two something unknown
00FF3020	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF	
00FF3030	FF FF FF FF 01 00 00 00 34 54 75 46 CB AB 8A B7	
00FF3040	FE FF FF FF 01 00 00 00 63 4E 2A 00 00 00 00	
00FF3050	01 00 00 00 43 23 2A 00 63 4E 2A 00 00 00 00	
00FF3060	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3070	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3080	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3090	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Registers (MMX)

Register	Value
EAX	00000000
ECX	6E494714 ASCII "H(*"
EDX	00000000
EBX	00000000
ESP	001EF84C
EBP	001EF894
ESI	00000001
EDI	00FF33A8 few.00FF33A8
EIP	6E445584 MSVCR100.printf

**Stack [001EF84C]=few.00FF3064
Imm=0000000C (decimal 12.)**

MSVCR100.printf

Address	Hex dump	ASCII (ANSI - Cy)
001EF84C	00FF1057	RETURN from few
001EF850	00FF3010	ASCII "two"
001EF854	00000001	RETURN from few
001EF858	00000001	
001EF85C	002A4E68	hN* ASCII "pN"
001EF860	002A2848	H(*
001EF864	4668ACA0	ankF
001EF868	00000000	
001EF86C	00000000	
001EF870	7EFD0000	p*
001EF874	00000000	
001EF878	00000000	
001EF87C	001EF864	d*
001EF880	03389310	y8
001EF884	001EF800	Printer to next

Рис. 14.6: OllyDbg: первая инструкция в printf() в MSVCR100.DLL

Теперь printf() считает строку на 0x00FF3010 как свой единственный аргумент и выводит строку.

14.1. ЕСЛИ ВАРИАНТОВ МАЛО

Это самая последняя инструкция функции `printf()` :

CPU - main thread, module MSVCRT100

Address	Disassembly	Comment
6E4455DD	PUSH EAX	
6E4455DE	PUSH ESI	
6E4455DF	PUSH DWORD PTR SS:[EBP+8]	
6E4455E2	CALL __iob_func	
6E4455E7	ADD EAX,EBX	
6E4455E9	PUSH EAX	
6E4455EA	CALL 6E45C71D	
6E4455ED	MOV DWORD PTR SS:[EBP-1C],EAX	
6E4455F2	CALL __iob_func	
6E4455F7	ADD EAX,EBX	
6E4455F9	PUSH EAX	
6E4455FA	PUSH EDI	
6E4455FB	CALL 6E4006AC	Arg2 Arg1 MSVCRT100.6E4006AC
6E445600	ADD ESP,18	
6E445603	MOV DWORD PTR SS:[EBP-4],-2	
6E44560A	CALL 6E445618	
6E44560F	MOV EAX,DWORD PTR SS:[EBP-1C]	
6E445612	CALL 6E3F0995	
6E445617	RETN	
6E445618	CALL __iob_func	
6E44561D	ADD EAX,20	

Top of stack [001EF84C]=few.00FF1057

MSVCRT100.printf+93

Address	Hex dump	ASCII (ANSI - Cy)
00FF3000	7A 65 72 6F 0A 00 00 00 6F 6E 65 0A 00 00 00 00	zero one
00FF3010	74 77 6F 0A 00 00 00 00 73 6F 6D 65 74 68 69 6E	two something unknown
00FF3020	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF FF	
00FF3030	FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3040	FE FF FF FF 01 00 00 00 34 54 75 46 CB AB 8A B9	0 4TuFTrnKl
00FF3050	01 00 00 00 48 28 2A 00 63 4E 2A 00 00 00 00 00	0 H(* hN*
00FF3060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Register	Value
EAX	00000004
ECX	6E445617 MSVCRT100.6E445617
EDX	0009DC88
EBX	00000000
ESP	001EF84C
EBP	001EF894
ESI	00000001
EDI	00FF33A8 few.00FF33A8
EIP	6E445617 MSVCRT100.6E445617
C 0	ES 002B 32bit 0(FFFFFFFF)
P 1	CS 0023 32bit 0(FFFFFFFF)
A 0	SS 002B 32bit 0(FFFFFFFF)
Z 1	DS 002B 32bit 0(FFFFFFFF)
S 0	FS 0053 32bit 7EFD0000(FFF)
T 0	GS 002B 32bit 0(FFFFFFFF)
D 0	
O 0	LastErr 00000000 ERROR_SUCCESS
EFL	00000246 (NO,NB,E,BE,NS,PE,GE,LE)
MM0	0000 0000 0000 0000
MM1	0000 0000 0000 0000
MM2	0000 0000 0000 0000
MM3	0000 0000 0000 0000
MM4	0000 0000 0000 0000

Address	Hex dump	ASCII (ANSI - Cy)
001EF84C	00FF1057	RETURN from f
001EF850	00FF3010	ASCII "two"
001EF854	00FF11CA	RETURN from f
001EF858	00000001	
001EF85C	002A4E68	hN* ASCII "pN"
001EF860	002A2848	H(*
001EF864	466BACA0	ankF
001EF868	00000000	
001EF86C	00000000	
001EF870	7EFDE000	p#"
001EF874	00000000	
001EF878	00000000	
001EF87C	001EF864	d°
001EF880	D3389310	y8u
001EF884	001EF800	Printer to ne

Рис. 14.7: OllyDbg: последняя инструкция в `printf()` в MSVCRT100.DLL

Строка «two» была только что выведена в консоли.

14.1. ЕСЛИ ВАРИАНТОВ МАЛО

Нажмем F7 или F8 (сделать шаг, не входя в функцию) и вернемся...нет, не в функцию `f()` но в `main()` :

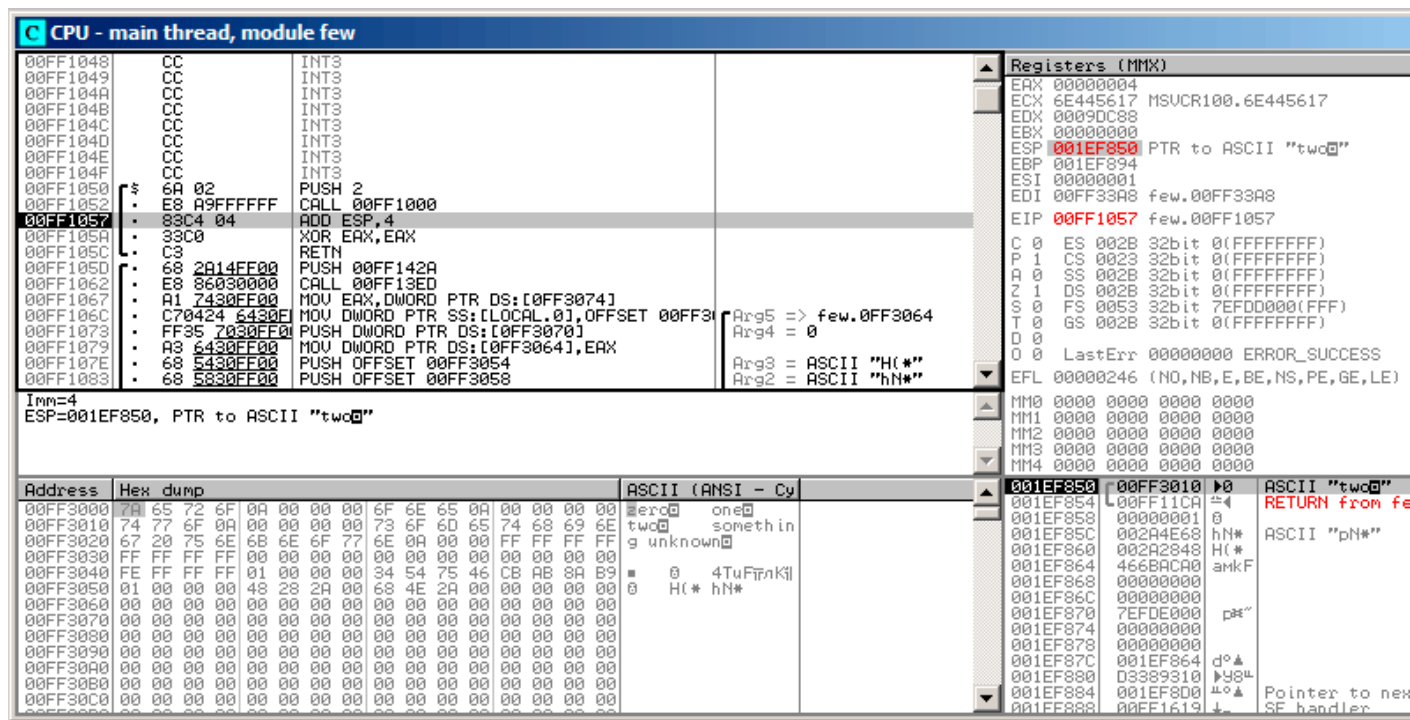


Рис. 14.8: OllyDbg: возврат в `main()`

Да, это прямой переход из внутренностей `printf()` в `main()` . Потому как `RA` в стеке указывает не на какое-то место в функции `f()` а в `main()` . И `CALL 0x00FF1000` это инструкция вызывающая функцию `f()` .

14.1.2. ARM: Оптимизирующий Keil 6/2013 (Режим ARM)

```
.text:0000014C          f1:
.text:0000014C 00 00 50 E3      CMP     R0, #0
.text:00000150 13 0E 8F 02      ADREQ   R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A      BEQ     loc_170
.text:00000158 01 00 50 E3      CMP     R0, #1
.text:0000015C 4B 0F 8F 02      ADREQ   R0, aOne ; "one\n"
.text:00000160 02 00 00 0A      BEQ     loc_170
.text:00000164 02 00 50 E3      CMP     R0, #2
.text:00000168 4A 0F 8F 12      ADNE    R0, aSomethingUnkno ; "something unknown\n"
.text:0000016C 4E 0F 8F 02      ADREQ   R0, aTwo ; "two\n"
.text:00000170
.text:00000170          loc_170: ; CODE XREF: f1+8
.text:00000170          ; f1+14
.text:00000170 78 18 00 EA      B       __2printf
```

Мы снова не сможем сказать, глядя на этот код, был ли в оригинальном исходном коде `switch()` либо же несколько операторов `if()`.

Так или иначе, мы снова видим здесь инструкции с предикатами, например, `ADREQ` (*Equal*), которая будет исполняться только если `R0 = 0`, и тогда в `R0` будет загружен адрес строки `«zero\n»`.

Следующая инструкция `BEQ` перенаправит исполнение на `loc_170`, если `R0 = 0`.

Кстати, наблюдательный читатель может спросить, сработает ли `BEQ` нормально, ведь `ADREQ` перед ним уже заполнила регистр `R0` чем-то другим?

Сработает, потому что `BEQ` проверяет флаги, установленные инструкцией `CMP`, а `ADREQ` флаги никак не модифицирует.

Далее всё просто и знакомо. Вызов `printf()` один, и в самом конце, мы уже рассматривали подобный трюк (7.2.1 (стр. 49)). К вызову функции `printf()` в конце ведут три пути.

14.1. ЕСЛИ ВАРИАНТОВ МАЛО

Последняя инструкция `CMP R0, #2` здесь нужна, чтобы узнать $a = 2$ или нет.

Если это не так, то при помощи `ADRNE (Not Equal)` в `R0` будет загружен указатель на строку «*something unknown* \n», ведь a уже было проверено на 0 и 1 до этого, и здесь a точно не попадает под эти константы.

Ну а если $R0 = 2$, в `R0` будет загружен указатель на строку «*two* \n» при помощи инструкции `ADREQ`.

14.1.3. ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:000000D4          f1:
.text:000000D4 10 B5      PUSH    {R4,LR}
.text:000000D6 00 28      CMP     R0, #0
.text:000000D8 05 D0      BEQ     zero_case
.text:000000DA 01 28      CMP     R0, #1
.text:000000DC 05 D0      BEQ     one_case
.text:000000DE 02 28      CMP     R0, #2
.text:000000E0 05 D0      BEQ     two_case
.text:000000E2 91 A0      ADR     R0, aSomethingUnkno ; "something unknown\n"
.text:000000E4 04 E0      B       default_case

.text:000000E6          zero_case: ; CODE XREF: f1+4
.text:000000E6 95 A0      ADR     R0, aZero ; "zero\n"
.text:000000E8 02 E0      B       default_case

.text:000000EA          one_case: ; CODE XREF: f1+8
.text:000000EA 96 A0      ADR     R0, aOne ; "one\n"
.text:000000EC 00 E0      B       default_case

.text:000000EE          two_case: ; CODE XREF: f1+C
.text:000000EE 97 A0      ADR     R0, aTwo ; "two\n"
.text:000000F0          default_case ; CODE XREF: f1+10
.text:000000F0                               ; f1+14
.text:000000F0 06 F0 7E F8  BL      __2printf
.text:000000F4 10 BD      POP     {R4,PC}
```

Как уже было отмечено, в Thumb-режиме нет возможности добавлять условные предикаты к большинству инструкций, так что Thumb-код вышел похожим на код x86 в стиле [CISC](#), вполне понятный.

14.1.4. ARM64: Неоптимизирующий GCC (Linaro) 4.9

```
.LC12:
.string "zero"
.LC13:
.string "one"
.LC14:
.string "two"
.LC15:
.string "something unknown"
f12:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0
    str    w0, [x29,28]
    ldr    w0, [x29,28]
    cmp    w0, 1
    beq    .L34
    cmp    w0, 2
    beq    .L35
    cmp    w0, wzr
    bne    .L38          ; переход на метку по умолчанию
    adrp   x0, .LC12      ; "zero"
    add    x0, x0, :lo12:.LC12
    bl     puts
    b      .L32
.L34:
    adrp   x0, .LC13      ; "one"
    add    x0, x0, :lo12:.LC13
    bl     puts
```

14.1. ЕСЛИ ВАРИАНТОВ МАЛО

```
.L35:      b      .L32
          adrp   x0, .LC14      ; "two"
          add    x0, x0, :lo12:.LC14
          bl     puts
          b      .L32
.L38:      adrp   x0, .LC15      ; "something unknown"
          add    x0, x0, :lo12:.LC15
          bl     puts
          nop
.L32:      ldp    x29, x30, [sp], 32
          ret
```

Входное значение имеет тип *int*, поэтому для него используется регистр **W0**, а не целая часть регистра **X0**.

Указатели на строки передаются в **puts()** при помощи пары инструкций **ADRP/ADD**, как было показано в примере «Hello, world!»: 4.4.5 (стр. 19).

14.1.5. ARM64: Оптимизирующий GCC (Linaro) 4.9

```
f12:
    cmp     w0, 1
    beq     .L31
    cmp     w0, 2
    beq     .L32
    cbz     w0, .L35
; метка по умолчанию
    adrp    x0, .LC15      ; "something unknown"
    add     x0, x0, :lo12:.LC15
    b       puts
.L35:      adrp    x0, .LC12      ; "zero"
    add     x0, x0, :lo12:.LC12
    b       puts
.L32:      adrp    x0, .LC14      ; "two"
    add     x0, x0, :lo12:.LC14
    b       puts
.L31:      adrp    x0, .LC13      ; "one"
    add     x0, x0, :lo12:.LC13
    b       puts
```

Фрагмент кода более оптимизированный. Инструкция **CBZ** (*Compare and Branch on Zero* – сравнить и перейти если ноль) совершает переход если **W0** ноль. Здесь также прямой переход на **puts()** вместо вызова, как уже было описано: 14.1.1 (стр. 150).

14.1.6. MIPS

Листинг 14.3: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
    lui     $gp, (__gnu_local_gp >> 16)
; это 1?
    li      $v0, 1
    beq     $a0, $v0, loc_60
    la      $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; это 2?
    li      $v0, 2
    beq     $a0, $v0, loc_4C
    or      $at, $zero ; branch delay slot, NOP
; перейти, если не равно 0:
    bnez    $a0, loc_38
```


14.2. И ЕСЛИ МНОГО

```
or      $at, $zero ; branch delay slot, NOP
; случай нуля:
lui     $a0, ($LC0 >> 16) # "zero"
lw      $t9, (puts & 0xFFFF)($gp)
or      $at, $zero ; load delay slot, NOP
jr      $t9 ; branch delay slot, NOP
la      $a0, ($LC0 & 0xFFFF) # "zero" ; branch delay slot
# -----
loc_38:                                # CODE XREF: f+1C
lui     $a0, ($LC3 >> 16) # "something unknown"
lw      $t9, (puts & 0xFFFF)($gp)
or      $at, $zero ; load delay slot, NOP
jr      $t9
la      $a0, ($LC3 & 0xFFFF) # "something unknown" ; branch delay slot
# -----
loc_4C:                                # CODE XREF: f+14
lui     $a0, ($LC2 >> 16) # "two"
lw      $t9, (puts & 0xFFFF)($gp)
or      $at, $zero ; load delay slot, NOP
jr      $t9
la      $a0, ($LC2 & 0xFFFF) # "two" ; branch delay slot
# -----
loc_60:                                # CODE XREF: f+8
lui     $a0, ($LC1 >> 16) # "one"
lw      $t9, (puts & 0xFFFF)($gp)
or      $at, $zero ; load delay slot, NOP
jr      $t9
la      $a0, ($LC1 & 0xFFFF) # "one" ; branch delay slot
```

Функция всегда заканчивается вызовом `puts()`, так что здесь мы видим переход на `puts()` (`JR`: «Jump Register») вместо перехода с сохранением `RA` («jump and link»).

Мы говорили об этом ранее: [14.1.1](#) (стр. 150).

Мы также часто видим `NOP`-инструкции после `LW`. Это «load delay slot»: ещё один *delay slot* в MIPS. Инструкция после `LW` может исполняться в тот момент, когда `LW` загружает значение из памяти.

Впрочем, следующая инструкция не должна использовать результат `LW`.

Современные MIPS-процессоры ждут, если следующая инструкция использует результат `LW`, так что всё это уже устарело, но GCC всё ещё добавляет `NOP`-ы для более старых процессоров.

Вообще, это можно игнорировать.

14.1.7. Вывод

Оператор `switch()` с малым количеством вариантов трудно отличим от применения конструкции `if/else`: листинг [14.1.1](#).

14.2. И если много

Если ветвлений слишком много, то генерировать слишком длинный код с многочисленными `JE` / `JNE` уже не так удобно.

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
        case 3: printf ("three\n"); break;
        case 4: printf ("four\n"); break;
        default: printf ("something unknown\n"); break;
    }
}
```



```

};
};

int main()
{
    f (2); // test
};

```

14.2.1. x86

Неоптимизирующий MSVC

Рассмотрим пример, скомпилированный в (MSVC 2010):

Листинг 14.4: MSVC 2010

```

tv64 = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
    push ebp
    mov ebp, esp
    push ecx
    mov eax, DWORD PTR _a$[ebp]
    mov DWORD PTR tv64[ebp], eax
    cmp DWORD PTR tv64[ebp], 4
    ja SHORT $LN1@f
    mov ecx, DWORD PTR tv64[ebp]
    jmp DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push OFFSET $SG739 ; 'zero', 0aH, 00H
    call _printf
    add esp, 4
    jmp SHORT $LN9@f
$LN5@f:
    push OFFSET $SG741 ; 'one', 0aH, 00H
    call _printf
    add esp, 4
    jmp SHORT $LN9@f
$LN4@f:
    push OFFSET $SG743 ; 'two', 0aH, 00H
    call _printf
    add esp, 4
    jmp SHORT $LN9@f
$LN3@f:
    push OFFSET $SG745 ; 'three', 0aH, 00H
    call _printf
    add esp, 4
    jmp SHORT $LN9@f
$LN2@f:
    push OFFSET $SG747 ; 'four', 0aH, 00H
    call _printf
    add esp, 4
    jmp SHORT $LN9@f
$LN1@f:
    push OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call _printf
    add esp, 4
$LN9@f:
    mov esp, ebp
    pop ebp
    ret 0
prad 2 ; выровнять следующую метку
$LN11@f:
    DD $LN6@f ; 0
    DD $LN5@f ; 1
    DD $LN4@f ; 2
    DD $LN3@f ; 3
    DD $LN2@f ; 4

```

_f	ENDP

Здесь происходит следующее: в теле функции есть набор вызовов `printf()` с разными аргументами. Все они имеют, конечно же, адреса, а также внутренние символические метки, которые присвоил им компилятор. Также все эти метки указываются во внутренней таблице `$LN11@f`.

В начале функции, если a больше 4, то сразу происходит переход на метку `$LN1@f`, где вызывается `printf()` с аргументом `'something unknown'`.

А если a меньше или равно 4, то это значение умножается на 4 и прибавляется адрес таблицы с переходами (`$LN11@f`). Таким образом, получается адрес внутри таблицы, где лежит нужный адрес внутри тела функции. Например, возьмем a равным 2. $2 * 4 = 8$ (ведь все элементы таблицы — это адреса внутри 32-битного процесса, таким образом, каждый элемент занимает 4 байта). 8 прибавить к `$LN11@f` — это будет элемент таблицы, где лежит `$LN4@f`. `JMP` вытаскивает из таблицы адрес `$LN4@f` и делает безусловный переход туда.

Эта таблица иногда называется *jump table* или *branch table*³.

А там вызывается `printf()` с аргументом `'two'`. Дословно, инструкция `jmp DWORD PTR $LN11@f[ecx*4]` означает *перейти по DWORD, который лежит по адресу $\$LN11@f + ecx * 4$* .

`prad` (90 (стр. 884)) это макрос ассемблера, выравнивающий начало таблицы, чтобы она располагалась по адресу кратному 4 (или 16). Это нужно для того, чтобы процессор мог эффективнее загружать 32-битные значения из памяти через шину с памятью, кэш-память, итд.

³Сам метод раньше назывался *computed GOTO* В ранних версиях FORTRAN: [wikipedia](https://en.wikipedia.org/wiki/Computed_goto). Не очень-то и полезно в наше время, но каков термин!

Попробуем этот пример в OllyDbg. Входное значение функции (2) загружается в EAX:

CPU - main thread, module lot

Address	Hex dump	ASCII (ANSI - Cy)
010B1000	7A 65 72 6F 0A 00 00 00 6F 6E 65 0A 00 00 00 00	zero one
010B1001	74 77 6F 0A 00 00 00 00 74 68 72 65 65 0A 00 00	two three
010B1002	66 6F 75 72 0A 00 00 00 73 6F 6D 65 74 68 69 6E	four somethin
010B1003	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
010B1004	FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00	
010B1005	FE FF FF FF 01 00 00 00 9A E2 68 1D 65 1D 97 E2	0 0 bth#e#4t
010B1006	01 00 00 00 48 28 03 00 68 4E 03 00 00 00 00 00	0 H(hN
010B1007	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B1008	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B1009	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B100A	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B100B	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B100C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B100D	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Registers (MMX)

Register	Value
EAX	00000002
ECX	6E494714 MSUCR100.__initenv
EDX	00000000
EBX	00000000
ESP	003CFDA8
EBP	003CFDA8
ESI	00000001
EDI	010B33B8 lot.010B33B8
EIP	010B1007 lot.010B1007
C 0	ES 002B 32bit 0(FFFFFFFF)
P 1	CS 0023 32bit 0(FFFFFFFF)
A 0	SS 002B 32bit 0(FFFFFFFF)
Z 1	DS 002B 32bit 0(FFFFFFFF)
S 0	FS 0053 32bit 7EFD0000(FFF)
T 0	GS 002B 32bit 0(FFFFFFFF)
D 0	
O 0	LastErr 00000000 ERROR_SUCCESS
EFL	00000246 (NO,NB,E,BE,NS,PE,GE,LE)
MM0	0000 0000 0000 0000
MM1	0000 0000 0000 0000
MM2	0000 0000 0000 0000
MM3	0000 0000 0000 0000
MM4	0000 0000 0000 0000

Stack [003CFDA8]=6E494714 (MSUCR100.__initenv)

Address	Hex dump	ASCII (ANSI - Cy)	Offset	MSUCR100
003CFDA8	6E494714	7A 65 72 6F	0	RETURN from lot
003CFDA9	003CFDA8	74 77 6F 0A	4	RETURN from lot
003CFDAA	003CFDA8	66 6F 75 72	8	RETURN from lot
003CFDAB	003CFDA8	67 20 75 6E	12	RETURN from lot
003CFDAC	003CFDA8	68 6E 6F 77	16	RETURN from lot
003CFDAD	003CFDA8	6E 0A 00 00	20	RETURN from lot
003CFDAE	003CFDA8	FF FF FF FF	24	RETURN from lot
003CFDAF	003CFDA8	FE FF FF FF	28	RETURN from lot
003CFDB0	003CFDA8	01 00 00 00	32	RETURN from lot
003CFDB1	003CFDA8	9A E2 68 1D	36	RETURN from lot
003CFDB2	003CFDA8	65 1D 97 E2	40	RETURN from lot
003CFDB3	003CFDA8	00 00 00 00	44	RETURN from lot
003CFDB4	003CFDA8	00 00 00 00	48	RETURN from lot
003CFDB5	003CFDA8	00 00 00 00	52	RETURN from lot
003CFDB6	003CFDA8	00 00 00 00	56	RETURN from lot
003CFDB7	003CFDA8	00 00 00 00	60	RETURN from lot
003CFDB8	003CFDA8	00 00 00 00	64	RETURN from lot
003CFDB9	003CFDA8	00 00 00 00	68	RETURN from lot
003CFDBA	003CFDA8	00 00 00 00	72	RETURN from lot
003CFDBB	003CFDA8	00 00 00 00	76	RETURN from lot
003CFDBC	003CFDA8	00 00 00 00	80	RETURN from lot
003CFDBD	003CFDA8	00 00 00 00	84	RETURN from lot
003CFDBE	003CFDA8	00 00 00 00	88	RETURN from lot
003CFDBF	003CFDA8	00 00 00 00	92	RETURN from lot
003CFDC0	003CFDA8	00 00 00 00	96	RETURN from lot
003CFDC1	003CFDA8	00 00 00 00	100	RETURN from lot
003CFDC2	003CFDA8	00 00 00 00	104	RETURN from lot
003CFDC3	003CFDA8	00 00 00 00	108	RETURN from lot
003CFDC4	003CFDA8	00 00 00 00	112	RETURN from lot
003CFDC5	003CFDA8	00 00 00 00	116	RETURN from lot
003CFDC6	003CFDA8	00 00 00 00	120	RETURN from lot
003CFDC7	003CFDA8	00 00 00 00	124	RETURN from lot
003CFDC8	003CFDA8	00 00 00 00	128	RETURN from lot
003CFDC9	003CFDA8	00 00 00 00	132	RETURN from lot
003CFDCA	003CFDA8	00 00 00 00	136	RETURN from lot
003CFDCB	003CFDA8	00 00 00 00	140	RETURN from lot
003CFDCC	003CFDA8	00 00 00 00	144	RETURN from lot
003CFDCD	003CFDA8	00 00 00 00	148	RETURN from lot
003CFDCE	003CFDA8	00 00 00 00	152	RETURN from lot
003CFDCF	003CFDA8	00 00 00 00	156	RETURN from lot
003CFDD0	003CFDA8	00 00 00 00	160	RETURN from lot
003CFDD1	003CFDA8	00 00 00 00	164	RETURN from lot
003CFDD2	003CFDA8	00 00 00 00	168	RETURN from lot
003CFDD3	003CFDA8	00 00 00 00	172	RETURN from lot
003CFDD4	003CFDA8	00 00 00 00	176	RETURN from lot
003CFDD5	003CFDA8	00 00 00 00	180	RETURN from lot
003CFDD6	003CFDA8	00 00 00 00	184	RETURN from lot
003CFDD7	003CFDA8	00 00 00 00	188	RETURN from lot
003CFDD8	003CFDA8	00 00 00 00	192	RETURN from lot
003CFDD9	003CFDA8	00 00 00 00	196	RETURN from lot
003CFDDA	003CFDA8	00 00 00 00	200	RETURN from lot
003CFDDB	003CFDA8	00 00 00 00	204	RETURN from lot
003CFDDC	003CFDA8	00 00 00 00	208	RETURN from lot
003CFDDD	003CFDA8	00 00 00 00	212	RETURN from lot
003CFDDE	003CFDA8	00 00 00 00	216	RETURN from lot
003CFDDF	003CFDA8	00 00 00 00	220	RETURN from lot
003CFDE0	003CFDA8	00 00 00 00	224	RETURN from lot

Рис. 14.9: OllyDbg: входное значение функции загружено в EAX

14.2. И ЕСЛИ МНОГО

Входное значение проверяется, не больше ли оно чем 4? Нет, переход по умолчанию («default») не будет выполнен:

CPU - main thread, module lot

Address	Hex dump	ASCII (ANSI - Cy)
010B1000	55	PUSH EBP
010B1001	8BEC	MOV EBP,ESP
010B1003	51	PUSH ECX
010B1004	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]
010B1007	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
010B1009	837D FC 04	CMP DWORD PTR SS:[EBP-4],4
010B100E	77 5A	JA SHORT 010B106A
010B1010	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
010B1013	FF248D 7C100	JMP DWORD PTR DS:[ECX*4+010B107C]
010B101A	68 00300001	PUSH OFFSET 010B3000
010B101F	FF15 00200000	CALL DWORD PTR DS:[&MSUCR100.printf]
010B1025	83C4 04	ADD ESP,4
010B1028	EB 4E	JMP SHORT 010B1078
010B102A	68 00300001	PUSH OFFSET 010B3000
010B102F	FF15 00200000	CALL DWORD PTR DS:[&MSUCR100.printf]
010B1035	83C4 04	ADD ESP,4
010B1038	EB 3E	JMP SHORT 010B1078
010B103A	68 00300001	PUSH OFFSET 010B3000
010B103F	FF15 00200000	CALL DWORD PTR DS:[&MSUCR100.printf]
010B1045	83C4 04	ADD ESP,4
010B1048	EB 2E	JMP SHORT 010B1078

Jump is not taken
Dest=010B106A

Registers (MMX)

Register	Value
EAX	00000002
ECX	6E494714 MSUCR100.__initenv
EDX	00000000
EBX	00000000
ESP	003CFDA8
EBP	003CFDA8
ESI	00000001
EDI	010B33B8 lot.010B33B8
EIP	010B100E lot.010B100E

MMX

Register	Value
MM0	0000 0000 0000 0000
MM1	0000 0000 0000 0000
MM2	0000 0000 0000 0000
MM3	0000 0000 0000 0000
MM4	0000 0000 0000 0000

003CFDA8 00000002

Address	Hex dump	ASCII (ANSI - Cy)
010B3000	7A 65 72 6F 0A 00 00 00 6F 6E 65 0A 00 00 00 00	zero one
010B3010	74 77 6F 0A 00 00 00 00 74 68 72 65 65 0A 00 00	two three
010B3020	66 6F 75 72 0A 00 00 00 73 6F 6D 65 74 68 69 6E	four something
010B3030	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
010B3040	FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00	
010B3050	FE FF FF FF 01 00 00 00 9A E2 68 1D 65 1D 97 E2	
010B3060	01 00 00 00 48 28 03 00 68 4E 03 00 00 00 00 00	
010B3070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B3080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B3090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B30A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B30B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
010B30C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

003CFDA8 003CFDA8

Address	Hex dump	ASCII (ANSI - Cy)
003CFDA8	7A 65 72 6F 0A 00 00 00 6F 6E 65 0A 00 00 00 00	zero one
003CFDA9	74 77 6F 0A 00 00 00 00 74 68 72 65 65 0A 00 00	two three
003CFDAA	66 6F 75 72 0A 00 00 00 73 6F 6D 65 74 68 69 6E	four something
003CFDAB	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
003CFDAC	FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDAD	FE FF FF FF 01 00 00 00 9A E2 68 1D 65 1D 97 E2	
003CFDAE	01 00 00 00 48 28 03 00 68 4E 03 00 00 00 00 00	
003CFDAF	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB1	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB2	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB3	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB5	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB6	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB7	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDB9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDBA	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDBB	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDBC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDBD	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDBE	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDBF	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC1	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC2	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC3	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC5	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC6	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC7	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDC9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDCA	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDCB	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDCC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDCD	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDCE	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDCF	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD1	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD2	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD3	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD5	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD6	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD7	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDD9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDDA	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDDB	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDDC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDDD	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDDE	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDDF	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
003CFDE0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Рис. 14.10: OllyDbg: 2 не больше чем 4: переход не работает

14.2. И ЕСЛИ МНОГО

Здесь мы видим `jumptable`:

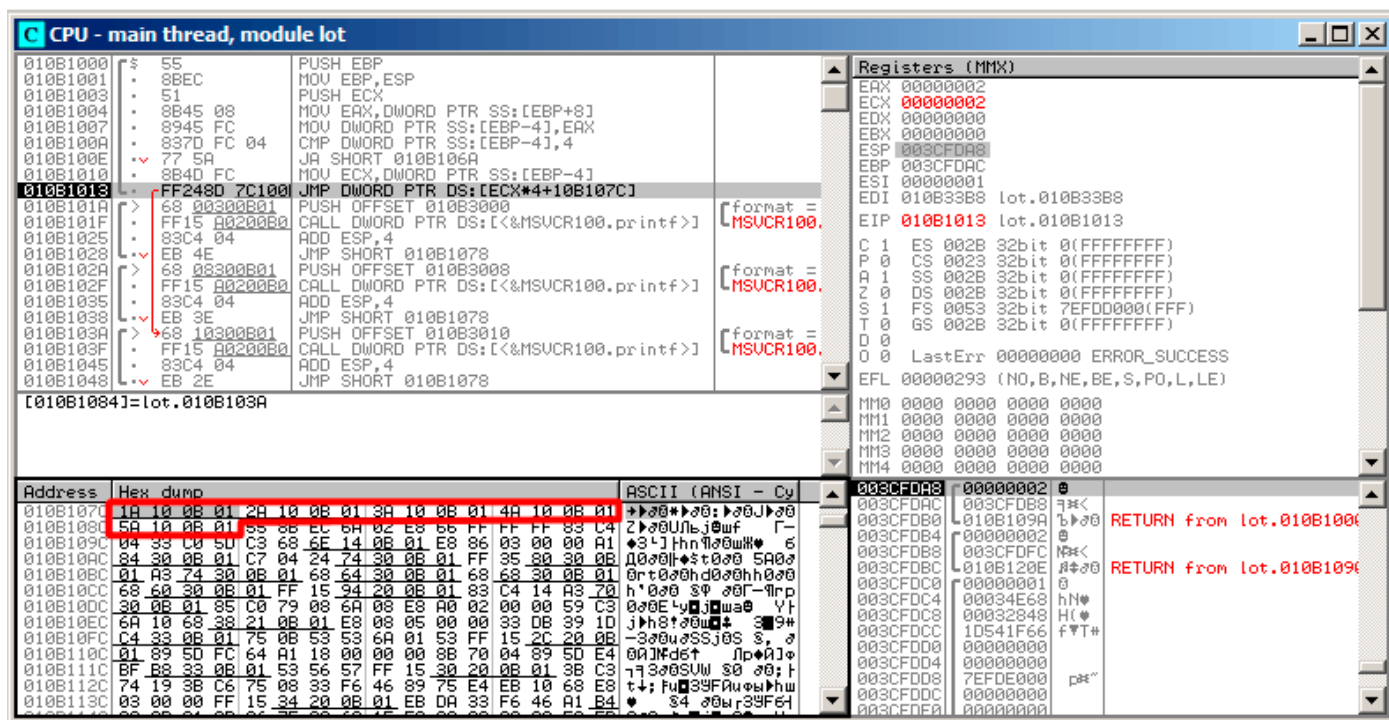


Рис. 14.11: OllyDbg: вычисляем адрес для перехода используя `jumptable`

Кстати, щелчком по «Follow in Dump» → «Address constant», так что теперь `jumptable` видна в окне данных.

Это 5 32-битных значений⁴. `ECX` сейчас содержит 2, так что второй элемент (считая с нулевого) таблицы будет использован. Кстати, можно также щелкнуть «Follow in Dump» → «Memory address» и OllyDbg покажет элемент, который сейчас адресуется в инструкции `JMP`. Это `0x010B103A`.

⁴Они подчеркнуты в OllyDbg, потому что это также и FIXUP-ы: 69.2.6 (стр. 699), мы вернемся к ним позже

14.2. И ЕСЛИ МНОГО

Переход сработал и мы теперь на `0x010B103A` : сейчас будет исполнен код, выводящий строку «two»:

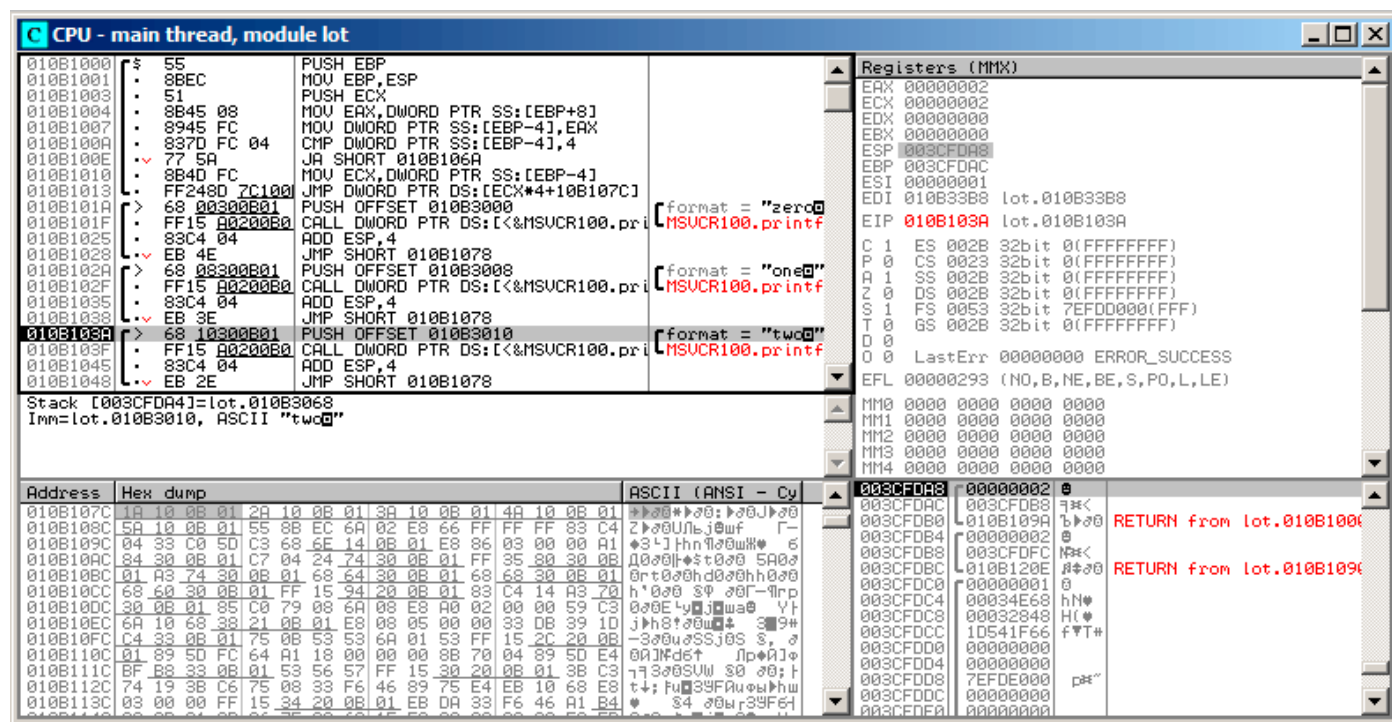


Рис. 14.12: OllyDbg: теперь мы на соответствующей метке `case`:

Неоптимизирующий GCC

Посмотрим, что сгенерирует GCC 4.4.1:

Листинг 14.5: GCC 4.4.1

```
public f
f
proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0 = dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 18h
cmp     [ebp+arg_0], 4
ja      short loc_8048444
mov     eax, [ebp+arg_0]
shl     eax, 2
mov     eax, ds:off_804855C[eax]
jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
mov     [esp+18h+var_18], offset aZero ; "zero"
call    _puts
jmp     short locret_8048450

loc_804840C: ; DATA XREF: .rodata:08048560
mov     [esp+18h+var_18], offset aOne ; "one"
call    _puts
jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
mov     [esp+18h+var_18], offset aTwo ; "two"
call    _puts
jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
```

14.2. И ЕСЛИ МНОГО

```
mov    [esp+18h+var_18], offset aThree ; "three"
call   _puts
jmp     short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
mov     [esp+18h+var_18], offset aFour ; "four"
call    _puts
jmp     short locret_8048450

loc_8048444: ; CODE XREF: f+A
mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
call    _puts

locret_8048450: ; CODE XREF: f+26
           ; f+34...
leave
retn
f        endp

off_804855C dd offset loc_80483FE ; DATA XREF: f+12
           dd offset loc_804840C
           dd offset loc_804841A
           dd offset loc_8048428
           dd offset loc_8048436
```

Практически то же самое, за исключением мелкого нюанса: аргумент из `arg_0` умножается на 4 при помощи сдвига влево на 2 бита (это почти то же самое что и умножение на 4) (17.2.1 (стр. 212)). Затем адрес метки внутри функции берется из массива `off_804855C` и адресуется при помощи вычисленного индекса.

14.2.2. ARM: Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 14.6: Оптимизирующий Keil 6/2013 (Режим ARM)

```
00000174          f2
00000174 05 00 50 E3    CMP     R0, #5          ; switch 5 cases
00000178 00 F1 8F 30    ADDCC   PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA    B       default_case ; jumtable 00000178 default case

00000180
00000180          loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA    B       zero_case ; jumtable 00000178 case 0

00000184
00000184          loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA    B       one_case ; jumtable 00000178 case 1

00000188
00000188          loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA    B       two_case ; jumtable 00000178 case 2

0000018C
0000018C          loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA    B       three_case ; jumtable 00000178 case 3

00000190
00000190          loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA    B       four_case ; jumtable 00000178 case 4

00000194
00000194          zero_case ; CODE XREF: f2+4
00000194          ; f2:loc_180
00000194 EC 00 8F E2    ADR     R0, aZero ; jumtable 00000178 case 0
00000198 06 00 00 EA    B       loc_1B8

0000019C
0000019C          one_case ; CODE XREF: f2+4
0000019C          ; f2:loc_184
0000019C EC 00 8F E2    ADR     R0, aOne ; jumtable 00000178 case 1
```

14.2. И ЕСЛИ МНОГО

```

000001A0 04 00 00 EA      B      loc_1B8

000001A4
000001A4          two_case ; CODE XREF: f2+4
000001A4          ; f2:loc_188
000001A4 01 0C 8F E2      ADR      R0, aTwo          ; jumtable 00000178 case 2
000001A8 02 00 00 EA      B      loc_1B8

000001AC
000001AC          three_case ; CODE XREF: f2+4
000001AC          ; f2:loc_18C
000001AC 01 0C 8F E2      ADR      R0, aThree        ; jumtable 00000178 case 3
000001B0 00 00 00 EA      B      loc_1B8

000001B4
000001B4          four_case ; CODE XREF: f2+4
000001B4          ; f2:loc_190
000001B4 01 0C 8F E2      ADR      R0, aFour          ; jumtable 00000178 case 4
000001B8
000001B8          loc_1B8 ; CODE XREF: f2+24
000001B8          ; f2+2C
000001B8 66 18 00 EA      B      __2printf

000001BC
000001BC          default_case ; CODE XREF: f2+4
000001BC          ; f2+8
000001BC D4 00 8F E2      ADR      R0, aSomethingUnkno ; jumtable 00000178 default case
000001C0 FC FF FF EA      B      loc_1B8

```

В этом коде используется та особенность режима ARM, что все инструкции в этом режиме имеют фиксированную длину 4 байта.

Итак, не будем забывать, что максимальное значение для *a* это 4: всё что выше, должно вызвать вывод строки «*something unknown*».

Самая первая инструкция `CMP R0, #5` сравнивает входное значение в *a* с 5.

⁵ Следующая инструкция `ADDCC PC, PC, R0, LSL#2` сработает только в случае если $R0 < 5$ (*CC=Carry clear / Less than*). Следовательно, если `ADDCC` не сработает (это случай с $R0 \geq 5$), выполнится переход на метку `default_case`.

Но если $R0 < 5$ и `ADDCC` сработает, то произойдет следующее.

Значение в `R0` умножается на 4. Фактически, `LSL#2` в суффиксе инструкции означает «сдвиг влево на 2 бита».

Но как будет видно позже (17.2.1 (стр. 212)) в секции «Сдвиги», сдвиг влево на 2 бита эквивалентен его умножению на 4.

Затем полученное $R0 * 4$ прибавляется к текущему значению `PC`, совершая, таким образом, переход на одну из расположенных ниже инструкций `B` (*Branch*).

На момент исполнения `ADDCC`, содержимое `PC` на 8 байт больше (`0x180`), чем адрес по которому расположена сама инструкция `ADDCC` (`0x178`), либо, говоря иным языком, на 2 инструкции больше.

Это связано с работой конвейера процессора ARM: пока исполняется инструкция `ADDCC`, процессор уже начинает обрабатывать инструкцию после следующей, поэтому `PC` указывает туда. Этот факт нужно запомнить.

Если $a = 0$, тогда к `PC` ничего не будет прибавлено и в `PC` запишется актуальный на тот момент `PC` (который больше на 8) и произойдет переход на метку `loc_180`. Это на 8 байт дальше места, где находится инструкция `ADDCC`.

Если $a = 1$, тогда в `PC` запишется $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 12 = 0x184$. Это адрес метки `loc_184`.

При каждой добавленной к *a* единице итоговый `PC` увеличивается на 4.

4 это длина инструкции в режиме ARM и одновременно с этим, длина каждой инструкции `B`, их здесь следует 5 в ряд.

Каждая из этих пяти инструкций `B` передает управление дальше, где собственно и происходит то, что запрограммировано в операторе `switch()`. Там происходит загрузка указателя на свою строку, итд.

⁵ADD—складывание чисел

14.2.3. ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 14.7: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

000000F6          EXPORT f2
000000F6          f2
000000F6 10 B5      PUSH    {R4,LR}
000000F8 03 00      MOVS     R3, R0
000000FA 06 F0 69 F8  BL      __ARM_common_switch8_thumb ; switch 6 cases

000000FE 05          DCB     5
000000FF 04 06 08 0A 0C 10  DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for switch statement
00000105 00          ALIGN 2
00000106          zero_case ; CODE XREF: f2+4
00000106 8D A0      ADR      R0, aZero ; jumtable 000000FA case 0
00000108 06 E0      B        loc_118

0000010A          one_case ; CODE XREF: f2+4
0000010A 8E A0      ADR      R0, aOne ; jumtable 000000FA case 1
0000010C 04 E0      B        loc_118

0000010E          two_case ; CODE XREF: f2+4
0000010E 8F A0      ADR      R0, aTwo ; jumtable 000000FA case 2
00000110 02 E0      B        loc_118

00000112          three_case ; CODE XREF: f2+4
00000112 90 A0      ADR      R0, aThree ; jumtable 000000FA case 3
00000114 00 E0      B        loc_118

00000116          four_case ; CODE XREF: f2+4
00000116 91 A0      ADR      R0, aFour ; jumtable 000000FA case 4
00000118          loc_118 ; CODE XREF: f2+12
00000118          ; f2+16
00000118 06 F0 6A F8      BL      __2printf
0000011C 10 BD      POP      {R4,PC}

0000011E          default_case ; CODE XREF: f2+4
0000011E 82 A0      ADR      R0, aSomethingUnkno ; jumtable 000000FA default case
00000120 FA E7      B        loc_118

000061D0          EXPORT __ARM_common_switch8_thumb
000061D0          __ARM_common_switch8_thumb ; CODE XREF: example6_f2+4
000061D0 78 47      BX        PC

000061D2 00 00      ALIGN 4
000061D2          ; End of function __ARM_common_switch8_thumb
000061D2
000061D4          __32__ARM_common_switch8_thumb ; CODE XREF: __ARM_common_switch8_thumb
000061D4 01 C0 5E E5      LDRB     R12, [LR,#-1]
000061D8 0C 00 53 E1      CMP      R3, R12
000061DC 0C 30 DE 27      LDRCSB   R3, [LR,R12]
000061E0 03 30 DE 37      LDRCCB   R3, [LR,R3]
000061E4 83 C0 8E E0      ADD      R12, LR, R3,LSL#1
000061E8 1C FF 2F E1      BX      R12
000061E8          ; End of function __32__ARM_common_switch8_thumb

```

В режимах Thumb и Thumb-2 уже нельзя надеяться на то, что все инструкции имеют одну длину.

Можно даже сказать, что в этих режимах инструкции переменной длины, как в x86.

Так что здесь добавляется специальная таблица, содержащая информацию о том, как много вариантов здесь, не включая варианта по умолчанию, и смещения, для каждого варианта. Каждое смещение кодирует метку, куда нужно передать

14.2. И ЕСЛИ МНОГО

управление в соответствующем случае.

Для того чтобы работать с таблицей и совершить переход, вызывается служебная функция

`__ARM_common_switch8_thumb`. Она начинается с инструкции `BX PC`, чья функция – переключить процессор в ARM-режим.

Далее функция, работающая с таблицей. Она слишком сложная для рассмотрения в данном месте, так что пропустим это.

Но можно отметить, что эта функция использует регистр `LR` как указатель на таблицу.

Действительно, после вызова этой функции, в `LR` был записан адрес после инструкции

`BL __ARM_common_switch8_thumb`, а там как раз и начинается таблица.

Ещё можно отметить, что код для этого выделен в отдельную функцию для того, чтобы не нужно было каждый раз генерировать точно такой же фрагмент кода для каждого выражения `switch()`.

IDA распознала эту служебную функцию и таблицу автоматически дописала комментарии к меткам вроде `jumptable 000000FA case 0`.

14.2.4. MIPS

Листинг 14.8: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
    lui    $gp, (__gnu_local_gp >> 16)
; перейти на loc_24, если входное значение меньше 5:
    sltiu  $v0, $a0, 5
    bnez   $v0, loc_24
    la     $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; входное значение больше или равно 5
; вывести "something unknown" и закончить:
    lui    $a0, ($LC5 >> 16) # "something unknown"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC5 & 0xFFFF) # "something unknown" ; branch delay slot

loc_24:
                                # CODE XREF: f+8
; загрузить адрес таблицы переходов
; LA это псевдоинструкция, на самом деле здесь пара LUI и ADDIU:
    la     $v0, off_120
; умножить входное значение на 4:
    sll    $a0, 2
; прибавить умноженное значение к адресу таблицы:
    addu   $a0, $v0, $a0
; загрузить элемент из таблицы переходов:
    lw     $v0, 0($a0)
    or     $at, $zero ; NOP
; перейти по адресу, полученному из таблицы:
    jr     $v0
    or     $at, $zero ; branch delay slot, NOP

sub_44:
                                # DATA XREF: .rodata:0000012C
; вывести "three" и закончить
    lui    $a0, ($LC3 >> 16) # "three"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC3 & 0xFFFF) # "three" ; branch delay slot

sub_58:
                                # DATA XREF: .rodata:00000130
; вывести "four" и закончить
    lui    $a0, ($LC4 >> 16) # "four"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC4 & 0xFFFF) # "four" ; branch delay slot
```

14.2. И ЕСЛИ МНОГО

```
sub_6C:                                # DATA XREF: .rodata:off_120
; вывести "zero" и закончить
    lui    $a0, ($LC0 >> 16) # "zero"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC0 & 0xFFFF) # "zero" ; branch delay slot

sub_80:                                # DATA XREF: .rodata:00000124
; вывести "one" и закончить
    lui    $a0, ($LC1 >> 16) # "one"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC1 & 0xFFFF) # "one" ; branch delay slot

sub_94:                                # DATA XREF: .rodata:00000128
; вывести "two" и закончить
    lui    $a0, ($LC2 >> 16) # "two"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC2 & 0xFFFF) # "two" ; branch delay slot

; может быть размещено в секции .rodata:
off_120:    .word sub_6C
            .word sub_80
            .word sub_94
            .word sub_44
            .word sub_58
```

Новая для нас инструкция здесь это **SLTIU** («Set on Less Than Immediate Unsigned» – установить, если меньше чем значение, беззнаковое сравнение).

На самом деле, это то же что и **SLTU** («Set on Less Than Unsigned»), но «I» означает «immediate», т.е. число может быть задано в самой инструкции.

BNEZ это «Branch if Not Equal to Zero» (переход если не равно нулю).

Код очень похож на код для других **ISA**. **SLL** («Shift Word Left Logical» – логический сдвиг влево) совершает умножение на 4. MIPS всё-таки это 32-битный процессор, так что все адреса в таблице переходов (*jump table*) 32-битные.

14.2.5. Вывод

Примерный скелет оператора *switch()*:

Листинг 14.9: x86

```
MOV REG, input
CMP REG, 4 ; максимальное количество меток
JA default
SHL REG, 2 ; найти элемент в таблице. сдвинуть на 3 бита в x64
MOV REG, jump_table[REG]
JMP REG

case1:
    ; делать что-то
    JMP exit
case2:
    ; делать что-то
    JMP exit
case3:
    ; делать что-то
    JMP exit
case4:
    ; делать что-то
    JMP exit
case5:
    ; делать что-то
```

14.3. КОГДА МНОГО CASE В ОДНОМ БЛОКЕ

```
JMP exit

default:

    ...

exit:

    ....

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5
```

Переход по адресу из таблицы переходов может быть также реализован такой инструкцией: `JMP jump_table[REG*4]`. Или `JMP jump_table[REG*8]` в x64.

Таблица переходов (*jump table*) это просто массив указателей, как это будет вскоре описано: [19.5](#) (стр. [282](#)).

14.3. Когда много *case* в одном блоке

Вот очень часто используемая конструкция: несколько *case* может быть использовано в одном блоке:

```
#include <stdio.h>

void f(int a)
{
    switch (a)
    {
        case 1:
        case 2:
        case 7:
        case 10:
            printf ("1, 2, 7, 10\n");
            break;

        case 3:
        case 4:
        case 5:
        case 6:
            printf ("3, 4, 5\n");
            break;

        case 8:
        case 9:
        case 20:
        case 21:
            printf ("8, 9, 21\n");
            break;

        case 22:
            printf ("22\n");
            break;

        default:
            printf ("default\n");
            break;
    };
};

int main()
{
    f(4);
};
```

Слишком расточительно генерировать каждый блок для каждого случая, поэтому обычно генерируется каждый блок плюс некий диспетчер.

14.3.1. MSVC

Листинг 14.10: Оптимизирующий MSVC 2010

```

1  $SG2798 DB      '1, 2, 7, 10', 0aH, 00H
2  $SG2800 DB      '3, 4, 5', 0aH, 00H
3  $SG2802 DB      '8, 9, 21', 0aH, 00H
4  $SG2804 DB      '22', 0aH, 00H
5  $SG2806 DB      'default', 0aH, 00H
6
7  _a$ = 8
8  _f      PROC
9          mov     eax, DWORD PTR _a$[esp-4]
10         dec     eax
11         cmp     eax, 21
12         ja      SHORT $LN1@f
13         movzx   eax, BYTE PTR $LN10@f[eax]
14         jmp     DWORD PTR $LN11@f[eax*4]
15 $LN5@f:
16         mov     DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
17         jmp     DWORD PTR __imp__printf
18 $LN4@f:
19         mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
20         jmp     DWORD PTR __imp__printf
21 $LN3@f:
22         mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
23         jmp     DWORD PTR __imp__printf
24 $LN2@f:
25         mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
26         jmp     DWORD PTR __imp__printf
27 $LN1@f:
28         mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
29         jmp     DWORD PTR __imp__printf
30         nrad    2 ; выровнять таблицу $LN11@f по 16-байтной границе
31 $LN11@f:
32         DD      $LN5@f ; вывести '1, 2, 7, 10'
33         DD      $LN4@f ; вывести '3, 4, 5'
34         DD      $LN3@f ; вывести '8, 9, 21'
35         DD      $LN2@f ; вывести '22'
36         DD      $LN1@f ; вывести 'default'
37 $LN10@f:
38         DB      0 ; a=1
39         DB      0 ; a=2
40         DB      1 ; a=3
41         DB      1 ; a=4
42         DB      1 ; a=5
43         DB      1 ; a=6
44         DB      0 ; a=7
45         DB      2 ; a=8
46         DB      2 ; a=9
47         DB      0 ; a=10
48         DB      4 ; a=11
49         DB      4 ; a=12
50         DB      4 ; a=13
51         DB      4 ; a=14
52         DB      4 ; a=15
53         DB      4 ; a=16
54         DB      4 ; a=17
55         DB      4 ; a=18
56         DB      4 ; a=19
57         DB      2 ; a=20
58         DB      2 ; a=21
59         DB      3 ; a=22
60 _f      ENDP

```

Здесь видим две таблицы: первая таблица (`$LN10@f`) это таблица индексов, а вторая таблица (`$LN11@f`) это массив указателей на блоки.

В начале, входное значение используется как индекс в таблице индексов (строка 13).

14.3. КОГДА МНОГО CASE В ОДНОМ БЛОКЕ

Вот краткое описание значений в таблице: 0 это первый блок *case* (для значений 1, 2, 7, 10), 1 это второй (для значений 3, 4, 5), 2 это третий (для значений 8, 9, 21), 3 это четвертый (для значений 22), 4 это для блока по умолчанию.

Мы получаем индекс для второй таблицы указателей на блоки и переходим туда (строка 14).

Ещё нужно отметить то, что здесь нет случая для нулевого входного значения.

Поэтому мы видим инструкцию `DEC` на строке 10 и таблица начинается с $a = 1$. Потому что незачем выделять в таблице элемент для $a = 0$.

Это очень часто используемый шаблон.

В чем же экономия? Почему нельзя сделать так, как уже обсуждалось (14.2.1 (стр. 168)), используя только одну таблицу, содержащую указатели на блоки? Причина в том, что элементы в таблице индексов занимают только по 8-битному байту, поэтому всё это более компактно.

14.3.2. GCC

GCC делает так, как уже обсуждалось (14.2.1 (стр. 168)), используя просто таблицу указателей.

14.3.3. ARM64: Оптимизирующий GCC 4.9.1

Во-первых, здесь нет кода, срабатывающего в случае если входное значение — 0, так что GCC пытается сделать таблицу переходов более компактной и начинает с случая, когда входное значение — 1.

GCC 4.9.1 для ARM64 использует даже более интересный трюк. Он может закодировать все смещения как 8-битные байты. Вспомним, что все инструкции в ARM64 имеют размер в 4 байта.

GCC также использует тот факт, что все смещения в моем крохотном примере находятся достаточно близко друг от друга.

Так что таблица переходов состоит из байт.

Листинг 14.11: Оптимизирующий GCC 4.9.1 ARM64

```
f14:
; входное значение в W0
    sub    w0, w0, #1
    cmp    w0, 21
; переход если меньше или равно (беззнаковое):
    bls    .L9
.L2:
; вывести "default":
    adrp   x0, .LC4
    add    x0, x0, :lo12:.LC4
    b      puts
.L9:
; загрузить адрес таблицы переходов в X1:
    adrp   x1, .L4
    add    x1, x1, :lo12:.L4
; W0=input_value-1
; загрузить байт из таблицы:
    ldrb   w0, [x1,w0,uxtw]
; загрузить адрес метки Lrtx:
    adr    x1, .Lrtx4
; умножить элемент из таблицы на 4 (сдвинув на 2 бита влево) и прибавить (или вычесть) к адресу Lrtx:
    add    x0, x1, w0, sxtb #2
; перейти на вычисленный адрес:
    br     x0
; эта метка указывает на сегмент кода (text):
.Lrtx4:
    .section      .rodata
; всё после выражения ".section" выделяется в сегменте только для чтения (rodata):
.L4:
    .byte      (.L3 - .Lrtx4) / 4      ; case 1
    .byte      (.L3 - .Lrtx4) / 4      ; case 2
    .byte      (.L5 - .Lrtx4) / 4      ; case 3
    .byte      (.L5 - .Lrtx4) / 4      ; case 4
    .byte      (.L5 - .Lrtx4) / 4      ; case 5
    .byte      (.L5 - .Lrtx4) / 4      ; case 6
    .byte      (.L3 - .Lrtx4) / 4      ; case 7
```

14.3. КОГДА МНОГО CASE В ОДНОМ БЛОКЕ

```

.byte (.L6 - .Lrtx4) / 4 ; case 8
.byte (.L6 - .Lrtx4) / 4 ; case 9
.byte (.L3 - .Lrtx4) / 4 ; case 10
.byte (.L2 - .Lrtx4) / 4 ; case 11
.byte (.L2 - .Lrtx4) / 4 ; case 12
.byte (.L2 - .Lrtx4) / 4 ; case 13
.byte (.L2 - .Lrtx4) / 4 ; case 14
.byte (.L2 - .Lrtx4) / 4 ; case 15
.byte (.L2 - .Lrtx4) / 4 ; case 16
.byte (.L2 - .Lrtx4) / 4 ; case 17
.byte (.L2 - .Lrtx4) / 4 ; case 18
.byte (.L2 - .Lrtx4) / 4 ; case 19
.byte (.L6 - .Lrtx4) / 4 ; case 20
.byte (.L6 - .Lrtx4) / 4 ; case 21
.byte (.L7 - .Lrtx4) / 4 ; case 22
.text
; всё после выражения ".text" выделяется в сегменте кода (text):
.L7:
; вывести "22"
    adrp    x0, .LC3
    add     x0, x0, :lo12:.LC3
    b       puts
.L6:
; вывести "8, 9, 21"
    adrp    x0, .LC2
    add     x0, x0, :lo12:.LC2
    b       puts
.L5:
; вывести "3, 4, 5"
    adrp    x0, .LC1
    add     x0, x0, :lo12:.LC1
    b       puts
.L3:
; вывести "1, 2, 7, 10"
    adrp    x0, .LC0
    add     x0, x0, :lo12:.LC0
    b       puts
.LC0:
    .string "1, 2, 7, 10"
.LC1:
    .string "3, 4, 5"
.LC2:
    .string "8, 9, 21"
.LC3:
    .string "22"
.LC4:
    .string "default"

```

Скомпилируем этот пример как объектный файл и откроем его в [IDA](#). Вот таблица переходов:

Листинг 14.12: jumptable in IDA

.rodata:0000000000000064	AREA .rodata, DATA, READONLY
.rodata:0000000000000064	; ORG 0x64
.rodata:0000000000000064 \$d	DCB 9 ; case 1
.rodata:0000000000000065	DCB 9 ; case 2
.rodata:0000000000000066	DCB 6 ; case 3
.rodata:0000000000000067	DCB 6 ; case 4
.rodata:0000000000000068	DCB 6 ; case 5
.rodata:0000000000000069	DCB 6 ; case 6
.rodata:000000000000006A	DCB 9 ; case 7
.rodata:000000000000006B	DCB 3 ; case 8
.rodata:000000000000006C	DCB 3 ; case 9
.rodata:000000000000006D	DCB 9 ; case 10
.rodata:000000000000006E	DCB 0xF7 ; case 11
.rodata:000000000000006F	DCB 0xF7 ; case 12
.rodata:0000000000000070	DCB 0xF7 ; case 13
.rodata:0000000000000071	DCB 0xF7 ; case 14
.rodata:0000000000000072	DCB 0xF7 ; case 15
.rodata:0000000000000073	DCB 0xF7 ; case 16
.rodata:0000000000000074	DCB 0xF7 ; case 17

14.4. FALL-THROUGH

```
.rodata:0000000000000075      DCB 0xF7      ; case 18
.rodata:0000000000000076      DCB 0xF7      ; case 19
.rodata:0000000000000077      DCB      3      ; case 20
.rodata:0000000000000078      DCB      3      ; case 21
.rodata:0000000000000079      DCB      0      ; case 22
.rodata:000000000000007B ; .rodata      ends
```

В случае 1, 9 будет умножено на 9 и прибавлено к адресу метки `Lrtx4`.

В случае 22, 0 будет умножено на 4, в результате это 0.

Место сразу за меткой `Lrtx4` это метка `L7`, где находится код, выводющий «22».

В сегменте кода нет таблицы переходов, место для нее выделено в отдельной секции `.rodata` (нет особой нужды располагать её в сегменте кода).

Там есть также отрицательные байты (0xF7). Они используются для перехода назад, на код, выводющий строку «default» (на `.L2`).

14.4. Fall-through

Ещё одно популярное использование оператора `switch()` это т.н. «fallthrough» («провал»). Вот простой пример:

```
1 #define R 1
2 #define W 2
3 #define RW 3
4
5 void f(int type)
6 {
7     int read=0, write=0;
8
9     switch (type)
10    {
11        case RW:
12            read=1;
13        case W:
14            write=1;
15            break;
16        case R:
17            read=1;
18            break;
19        default:
20            break;
21    };
22    printf ("read=%d, write=%d\n", read, write);
23 };
```

Если `type = 1 (R)`, `read` будет выставлен в 1, если `type = 2 (W)`, `write` будет выставлен в 1. В случае `type = 3 (RW)`, обе `read` и `write` будут выставлены в 1.

Фрагмент кода на строке 14 будет исполнен в двух случаях: если `type = RW` или если `type = W`. Там нет «break» для «case RW», и это нормально.

14.4.1. MSVC x86

Листинг 14.13: MSVC 2012

```
$SG1305 DB      'read=%d, write=%d', 0aH, 00H

_write$ = -12    ; size = 4
_read$ = -8      ; size = 4
tv64 = -4        ; size = 4
_type$ = 8       ; size = 4
_f        PROC
    push     ebp
    mov      ebp, esp
    sub      esp, 12
```


14.4. FALL-THROUGH

```
    mov     DWORD PTR _read$[ebp], 0
    mov     DWORD PTR _write$[ebp], 0
    mov     eax, DWORD PTR _type$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 1 ; R
    je      SHORT $LN2@f
    cmp     DWORD PTR tv64[ebp], 2 ; W
    je      SHORT $LN3@f
    cmp     DWORD PTR tv64[ebp], 3 ; RW
    je      SHORT $LN4@f
    jmp     SHORT $LN5@f
$LN4@f: ; case RW:
    mov     DWORD PTR _read$[ebp], 1
$LN3@f: ; case W:
    mov     DWORD PTR _write$[ebp], 1
    jmp     SHORT $LN5@f
$LN2@f: ; case R:
    mov     DWORD PTR _read$[ebp], 1
$LN5@f: ; default
    mov     ecx, DWORD PTR _write$[ebp]
    push    ecx
    mov     edx, DWORD PTR _read$[ebp]
    push    edx
    push    OFFSET $SG1305 ; 'read=%d, write=%d'
    call    _printf
    add     esp, 12
    mov     esp, ebp
    pop     ebp
    ret     0
_f
ENDP
```

Код почти полностью повторяет то, что в исходнике.

Там нет переходов между метками `$LN4@f` и `$LN3@f`: так что когда управление (code flow) находится на `$LN4@f`, `read` в начале выставляется в 1, затем `write`.

Наверное, поэтому всё это и называется «проваливаться»: управление проваливается через один фрагмент кода (выставляющий `read`) в другой (выставляющий `write`). Если `type = W`, мы оказываемся на `$LN3@f`, так что код выставляющий `read` в 1 не исполнится.

14.4.2. ARM64

Листинг 14.14: GCC (Linaro) 4.9

```
.LC0:
.string "read=%d, write=%d\n"
f:
    stp     x29, x30, [sp, -48]!
    add     x29, sp, 0
    str     w0, [x29, 28]
    str     wzr, [x29, 44] ; установить локальные переменные "read" и "write" в ноль
    str     wzr, [x29, 40]
    ldr     w0, [x29, 28] ; загрузить аргумент "type"
    cmp     w0, 2          ; type=W?
    beq     .L3
    cmp     w0, 3          ; type=RW?
    beq     .L4
    cmp     w0, 1          ; type=R?
    beq     .L5
    b       .L6            ; в противном случае...
.L4: ; case RW
    mov     w0, 1
    str     w0, [x29, 44] ; read=1
.L3: ; case W
    mov     w0, 1
    str     w0, [x29, 40] ; write=1
    b       .L6
.L5: ; case R
    mov     w0, 1
```

14.5. УПРАЖНЕНИЯ

```
    str    w0, [x29,44] ; read=1
    nop
.L6: ; default
    adrp   x0, .LC0 ; "read=%d, write=%d\n"
    add    x0, x0, :lo12:.LC0
    ldr    w1, [x29,44] ; загрузить "read"
    ldr    w2, [x29,40] ; загрузить "write"
    bl     printf
    ldp    x29, x30, [sp], 48
    ret
```

Почти то же самое. Здесь нет переходов между метками `.L4` и `.L3`.

14.5. Упражнения

14.5.1. Упражнение #1

Вполне возможно переделать пример на Си в листинге [14.2](#) (стр. [162](#)) так, чтобы при компиляции получалось даже ещё меньше кода, но работать всё будет точно так же. Попробуйте этого добиться.

Глава 15

Циклы

15.1. Простой пример

15.1.1. x86

Для организации циклов в архитектуре x86 есть старая инструкция `LOOP`. Она проверяет значение регистра `ECX` и если оно не 0, делает **декремент** `ECX` и переход по метке, указанной в операнде. Возможно, эта инструкция не слишком удобная, потому что уже почти не бывает современных компиляторов, которые использовали бы её. Так что если вы видите где-то `LOOP`, то с большой вероятностью это вручную написанный код на ассемблере.

Обычно, циклы на Си/Си++ создаются при помощи `for()`, `while()`, `do/while()`. Начнем с `for()`. Это выражение описывает инициализацию, условие, операцию после каждой итерации (**инкремент/декремент**) и тело цикла.

```
for (инициализация; условие; после каждой итерации)
{
    тело_цикла;
}
```

Примерно так же, генерируемый код и будет состоять из этих четырех частей. Возьмем пример:

```
#include <stdio.h>

void printing_function(int i)
{
    printf ("f(%d)\n", i);
};

int main()
{
    int i;

    for (i=2; i<10; i++)
        printing_function(i);

    return 0;
};
```

Имеем в итоге (MSVC 2010):

Листинг 15.1: MSVC 2010

```
_i$ = -4
_main PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     DWORD PTR _i$[ebp], 2    ; инициализация цикла
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp] ; то что мы делаем после каждой итерации:
    add     eax, 1                    ; добавляем 1 к i
    mov     DWORD PTR _i$[ebp], eax
```

15.1. ПРОСТОЙ ПРИМЕР

```
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 10 ; это условие проверяется *перед* каждой итерацией
    jge     SHORT $LN1@main        ; если i больше или равно 10, заканчиваем цикл
    mov     ecx, DWORD PTR _i$[ebp] ; тело цикла: вызов функции printing_function(i)
    push    ecx
    call    _printing_function
    add     esp, 4
    jmp     SHORT $LN2@main         ; переход на начало цикла
$LN1@main:                          ; конец цикла
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP
```

В принципе, ничего необычного.

GCC 4.4.1 выдает примерно такой же код, с небольшой разницей:

Листинг 15.2: GCC 4.4.1

```
main      proc near
var_20     = dword ptr -20h
var_4      = dword ptr -4

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 20h
        mov     [esp+20h+var_4], 2 ; инициализация i
        jmp     short loc_8048476

loc_8048465:
        mov     eax, [esp+20h+var_4]
        mov     [esp+20h+var_20], eax
        call    printing_function
        add     [esp+20h+var_4], 1 ; инкремент i

loc_8048476:
        cmp     [esp+20h+var_4], 9
        jle     short loc_8048465 ; если i<=9, продолжаем цикл
        mov     eax, 0
        leave
        retn
main      endp
```

Интересно становится, если скомпилируем этот же код при помощи MSVC 2010 с включенной оптимизацией (/Ox):

Листинг 15.3: Оптимизирующий MSVC

```
_main     PROC
    push    esi
    mov     esi, 2
$LL3@main:
    push    esi
    call    _printing_function
    inc     esi
    add     esp, 4
    cmp     esi, 10 ; 0000000aH
    jl      SHORT $LL3@main
    xor     eax, eax
    pop     esi
    ret     0
_main     ENDP
```

Здесь происходит следующее: переменную *i* компилятор не выделяет в локальном стеке, а выделяет целый регистр под нее: **ESI** . Это возможно для маленьких функций, где мало локальных переменных.

15.1. ПРОСТОЙ ПРИМЕР

В принципе, всё то же самое, только теперь одна важная особенность: `f()` не должна менять значение `ESI`. Наш компилятор уверен в этом, а если бы и была необходимость использовать регистр `ESI` в функции `f()`, то её значение сохранялось бы в стеке. Примерно так же как и в нашем листинге: обратите внимание на `PUSH ESI/POP ESI` в начале и конце функции.

Попробуем GCC 4.4.1 с максимальной оптимизацией (`-O3`):

Листинг 15.4: Оптимизирующий GCC 4.4.1

```
main      proc near
var_10    = dword ptr -10h

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFF0h
        sub     esp, 10h
        mov     [esp+10h+var_10], 2
        call    printing_function
        mov     [esp+10h+var_10], 3
        call    printing_function
        mov     [esp+10h+var_10], 4
        call    printing_function
        mov     [esp+10h+var_10], 5
        call    printing_function
        mov     [esp+10h+var_10], 6
        call    printing_function
        mov     [esp+10h+var_10], 7
        call    printing_function
        mov     [esp+10h+var_10], 8
        call    printing_function
        mov     [esp+10h+var_10], 9
        call    printing_function
        xor     eax, eax
        leave
        retn
main      endp
```

Однако GCC просто *развернул* цикл¹.

Делается это в тех случаях, когда итераций не слишком много (как в нашем примере) и можно немного сэкономить время, убрав все инструкции, обеспечивающие цикл. В качестве обратной стороны медали, размер кода увеличился.

Использовать большие развернутые циклы в наше время не рекомендуется, потому что большие функции требуют больше кэш-памяти².

Увеличим максимальное значение *i* в цикле до 100 и попробуем снова. GCC выдает:

Листинг 15.5: GCC

```
main      public main
          proc near
var_20    = dword ptr -20h

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFF0h
        push    ebx
        mov     ebx, 2      ; i=2
        sub     esp, 1Ch

; выравнивание метки loc_80484D0 (начало тела цикла) по 16-байтной границе:
        nop

loc_80484D0:
; передать i как первый аргумент для printing_function():
        mov     [esp+20h+var_20], ebx
```

¹loop unwinding в англоязычной литературе

²Очень хорошая статья об этом: [Ulrich Drepper, *What Every Programmer Should Know About Memory*, (2007)]³. А также о рекомендациях о развернутых циклах от Intel можно прочитать здесь: [[Intel® 64 and IA-32 Architectures Optimization Reference Manual, (2014)]3.4.1.7].

15.1. ПРОСТОЙ ПРИМЕР

```
add    ebx, 1    ; i++
call   printing_function
cmp     ebx, 64h  ; i==100?
jnz     short loc_80484D0 ; если нет, продолжать
add     esp, 1Ch
xor     eax, eax  ; возврат 0
pop     ebx
mov     esp, ebp
pop     ebp
retn
main    endp
```

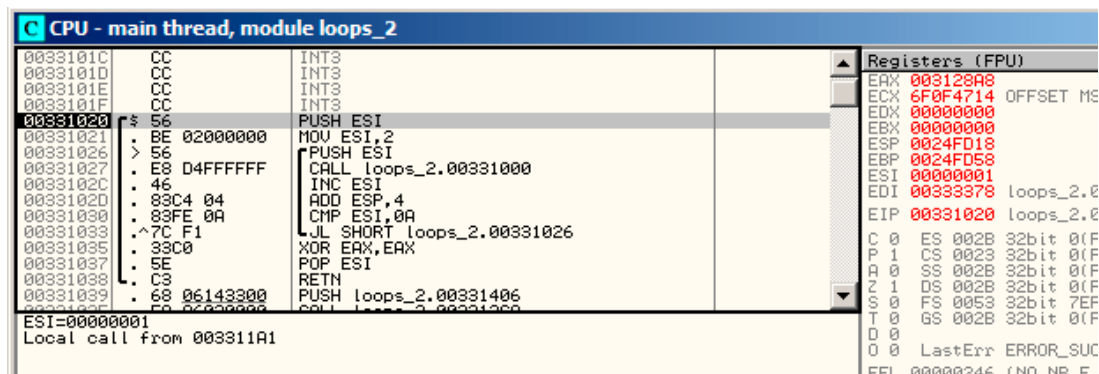
Это уже похоже на то, что сделал MSVC 2010 в режиме оптимизации (/Ox). За исключением того, что под переменную *i* будет выделен регистр **EBX**.

GCC уверен, что этот регистр не будет модифицироваться внутри **f()**, а если вдруг это и придётся там сделать, то его значение будет сохранено в начале функции, прямо как в **main()**.

15.1.2. x86: OllyDbg

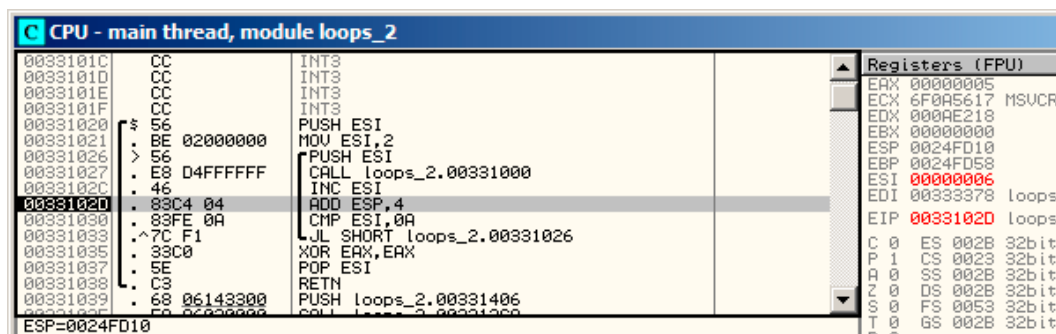
Скомпилируем наш пример в MSVC 2010 с `/Ox` и `/Ob0` и загрузим в OllyDbg.

Оказывается, OllyDbg может обнаруживать простые циклы и показывать их в квадратных скобках, для удобства:

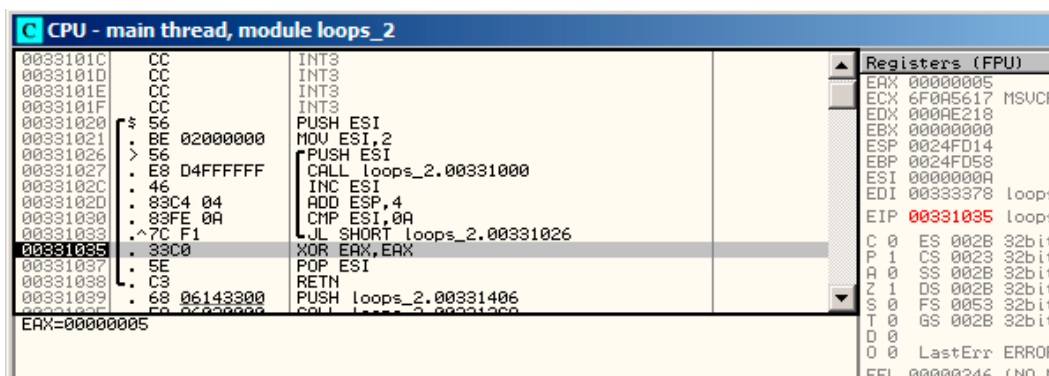
Рис. 15.1: OllyDbg: начало `main()`

Трассируя (F8 — сделать шаг, не входя в функцию) мы видим, как `ESI` увеличивается на 1.

Например, здесь $ESI = i = 6$:

Рис. 15.2: OllyDbg: тело цикла только что отработало с $i = 6$

9 это последнее значение цикла. Поэтому `JL` после [инкремента](#) не срабатывает и функция заканчивается:

Рис. 15.3: OllyDbg: $ESI = 10$, конец цикла

15.1.3. x86: tracer

Как видно, трассировать вручную цикл в отладчике — это не очень удобно. Поэтому попробуем [tracer](#). Открываем скомпилированный пример в [IDA](#), находим там адрес инструкции `PUSH ESI` (передающей единственный аргумент в `f()`), а это `0x401026` в нашем случае и запускаем [tracer](#):

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

15.1. ПРОСТОЙ ПРИМЕР

Опция `BPX` просто ставит точку останова по адресу и затем `tracer` будет выдавать состояние регистров. В `tracer.log` после запуска я вижу следующее:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

Видно, как значение `ESI` последовательно изменяется от 2 до 9. И даже более того, в `tracer` можно собирать значения регистров по всем адресам внутри функции.

Там это называется *trace*. Каждая инструкция трассируется, значения самых интересных регистров запоминаются. Затем генерируется `.idc`-скрипт для `IDA`, который добавляет комментарии. Итак, в `IDA` я узнал что адрес `main()` это `0x00401020` и запускаю:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

`BPF` означает установить точку останова на функции.

Получаю в итоге скрипты `loops_2.exe.idc` и `loops_2.exe_clear.idc`.

15.1. ПРОСТОЙ ПРИМЕР

Загружаю `loops_2.exe.idc` в IDA и увижу следующее:

```
.text:00401020
.text:00401020 ; ===== S U B R O U T I N E =====
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main      proc near      ; CODE XREF: __tmainCRTStartup+11D↓p
.text:00401020
.text:00401020     argc      = dword ptr  4
.text:00401020     argv      = dword ptr  8
.text:00401020     envp      = dword ptr 0Ch
.text:00401020
.text:00401020     push     esi          ; ESI=1
.text:00401021     mov      esi, 2
.text:00401026
.text:00401026 loc_401026:      ; CODE XREF: _main+13↓j
.text:00401026     push     esi          ; ESI=2..9
.text:00401027     call     sub_401000    ; tracing nested maximum level (1) reached,
.text:00401027     inc      esi          ; ESI=2..9
.text:0040102C     add      esp, 4        ; ESP=0x38fcbc
.text:0040102D     cmp      esi, 0Ah      ; ESI=3..0xa
.text:00401030     jnl     short loc_401026 ; SF=false,true OF=false
.text:00401033     xor      eax, eax
.text:00401035     pop      esi
.text:00401037     retn
.text:00401038     _main      endp      ; EAX=0
```

Рис. 15.4: IDA с загруженным .idc-скриптом

Видно, что `ESI` меняется от 2 до 9 в начале тела цикла, но после [инкремента](#) он в пределах `[3..0xA]`. Видно также, что функция `main()` заканчивается с 0 в `EAX`.

`tracer` также генерирует `loops_2.exe.txt`, содержащий адреса инструкций, сколько раз была исполнена каждая и значения регистров:

Листинг 15.6: loops_2.exe.txt

```
0x401020 (.text+0x20), e=      1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=      1 [MOV ESI, 2]
0x401026 (.text+0x26), e=      8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=      8 [CALL 8D1000h] tracing nested maximum level (1) reached, skipping ↗
    ↳ this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=      8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=      8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=      8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=      8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=      1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=      1 [POP ESI]
0x401038 (.text+0x38), e=      1 [RETN] EAX=0
```

Так можно использовать `grep`.

15.1.4. ARM

Неоптимизирующий Keil 6/2013 (Режим ARM)

```
main
    STMFD    SP!, {R4,LR}
    MOV      R4, #2
    B        loc_368
loc_35C:    ; CODE XREF: main+1C
    MOV      R0, R4
    BL       printing_function
    ADD      R4, R4, #1
```

15.1. ПРОСТОЙ ПРИМЕР

```
loc_368 ; CODE XREF: main+8
        CMP     R4, #0xA
        BLT     loc_35C
        MOV     R0, #0
        LDMFD   SP!, {R4,PC}
```

Счетчик итераций i будет храниться в регистре `R4`. Инструкция `MOV R4, #2` просто инициализирует i . Инструкции `MOV R0, R4` и `BL printing_function` составляют тело цикла. Первая инструкция готовит аргумент для функции, `f()` а вторая вызывает её. Инструкция `ADD R4, R4, #1` прибавляет единицу к i при каждой итерации. `CMP R4, #0xA` сравнивает i с `0xA` (10). Следующая за ней инструкция `BLT` (*Branch Less Than*) совершит переход, если i меньше чем 10. В противном случае в `R0` запишется 0 (потому что наша функция возвращает 0) и произойдет выход из функции.

Оптимизирующий Keil 6/2013 (Режим Thumb)

```
_main
        PUSH    {R4,LR}
        MOVS    R4, #2

loc_132 ; CODE XREF: _main+E
        MOVS    R0, R4
        BL      printing_function
        ADDS    R4, R4, #1
        CMP     R4, #0xA
        BLT     loc_132
        MOVS    R0, #0
        POP     {R4,PC}
```

Практически всё то же самое.

Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

```
_main
        PUSH    {R4,R7,LR}
        MOVW    R4, #0x1124 ; "%d\n"
        MOVS    R1, #2
        MOVT.W  R4, #0
        ADD     R7, SP, #4
        ADD     R4, PC
        MOV     R0, R4
        BLX     _printf
        MOV     R0, R4
        MOVS    R1, #3
        BLX     _printf
        MOV     R0, R4
        MOVS    R1, #4
        BLX     _printf
        MOV     R0, R4
        MOVS    R1, #5
        BLX     _printf
        MOV     R0, R4
        MOVS    R1, #6
        BLX     _printf
        MOV     R0, R4
        MOVS    R1, #7
        BLX     _printf
        MOV     R0, R4
        MOVS    R1, #8
        BLX     _printf
        MOV     R0, R4
        MOVS    R1, #9
        BLX     _printf
        MOVS    R0, #0
        POP     {R4,R7,PC}
```

15.1. ПРОСТОЙ ПРИМЕР

На самом деле, в моей функции `f()` было такое:

```
void printing_function(int i)
{
    printf ("%d\n", i);
};
```

Так что LLVM не только *развернул* цикл, но также и представил мою очень простую функцию `f()` как *inline-функцию*, и вставил её тело вместо цикла 8 раз. Это возможно, когда функция очень простая (как та что у меня) и когда она вызывается не очень много раз, как здесь.

ARM64: Оптимизирующий GCC 4.9.1

Листинг 15.7: Оптимизирующий GCC 4.9.1

```
printing_function:
; подготовить второй аргумент printf():
    mov     w1, w0
; загрузить адрес строки "f(%d)\n"
    adrp    x0, .LC0
    add     x0, x0, :lo12:LC0
; здесь просто переход вместо перехода с сохранением адреса и инструкции возврата:
    b       printf
main:
; сохранить FP и LR в локальном стеке:
    stp     x29, x30, [sp, -32]!
; установить стековый фрейм:
    add     x29, sp, 0
; сохранить содержимое регистра X19 в локальном стеке:
    str     x19, [sp, 16]
; будем использовать регистр W19 как счетчик.
; установить начальное значение в 2:
    mov     w19, 2
.L3:
; подготовить первый аргумент printing_function():
    mov     w0, w19
; инкремент регистра счетчика.
    add     w19, w19, 1
; W0 все еще содержит значение счетчика перед инкрементом.
    bl     printing_function
; конец?
    cmp     w19, 10
; нет, перейти на начало тела цикла:
    bne     .L3
; возврат 0
    mov     w0, 0
; восстановить содержимое регистра X19:
    ldr     x19, [sp, 16]
; восстановить значения FP и LR:
    ldp     x29, x30, [sp], 32
    ret
.LC0:
    .string "f(%d)\n"
```

ARM64: Неоптимизирующий GCC 4.9.1

Листинг 15.8: Неоптимизирующий GCC 4.9.1 -fno-inline

```
printing_function:
; подготовить второй аргумент printf():
    mov     w1, w0
; загрузить адрес строки "f(%d)\n"
    adrp    x0, .LC0
    add     x0, x0, :lo12:LC0
; здесь просто переход вместо перехода с сохранением адреса и инструкции возврата:
    b       printf
main:
```

15.1. ПРОСТОЙ ПРИМЕР

```
; сохранить FP и LR в локальном стеке:
    stp    x29, x30, [sp, -32]!
; установить стековый фрейм:
    add    x29, sp, 0
; сохранить содержимое регистра X19 в локальном стеке:
    str    x19, [sp, 16]
; будем использовать регистр W19 как счетчик.
; установить начальное значение в 2:
    mov    w19, 2
.L3:
; подготовить первый аргумент printing_function():
    mov    w0, w19
; инкремент регистра счетчика.
    add    w19, w19, 1
; W0 все еще содержит значение счетчика перед инкрементом.
    bl     printing_function
; конец?
    cmp    w19, 10
; нет, перейти на начало тела цикла:
    bne    .L3
; возврат 0
    mov    w0, 0
; восстановить содержимое регистра X19:
    ldr    x19, [sp, 16]
; восстановить значения FP и LR:
    ldp    x29, x30, [sp], 32
    ret
.LC0:
    .string "f(%d)\n"
```

15.1.5. MIPS

Листинг 15.9: Неоптимизирующий GCC 4.4.5 (IDA)

```
main:
; IDA не знает названия переменных в локальном стеке
; Это мы назвали их вручную:
i      = -0x10
saved_FP = -8
saved_RA = -4

; пролог функции:
    addiu   $sp, -0x28
    sw      $ra, 0x28+saved_RA($sp)
    sw      $fp, 0x28+saved_FP($sp)
    move    $fp, $sp
; инициализировать счетчик значением 2 и сохранить это значение в локальном стеке
    li      $v0, 2
    sw      $v0, 0x28+i($fp)
; псевдоинструкция. здесь на самом деле "BEQ $ZERO, $ZERO, loc_9C":
    b       loc_9C
    or      $at, $zero ; branch delay slot, NOP
# -----

loc_80:                                # CODE XREF: main+48
; загрузить значение счетчика из локального стека и вызвать printing_function():
    lw      $a0, 0x28+i($fp)
    jal     printing_function
    or      $at, $zero ; branch delay slot, NOP
; загрузить счетчик, инкрементировать его и записать назад:
    lw      $v0, 0x28+i($fp)
    or      $at, $zero ; NOP
    addiu   $v0, 1
    sw      $v0, 0x28+i($fp)

loc_9C:                                # CODE XREF: main+18
; проверить счетчик, он больше 10?
```

15.2. ФУНКЦИЯ КОПИРОВАНИЯ БЛОКОВ ПАМЯТИ

```
        lw      $v0, 0x28+i($fp)
        or      $at, $zero ; NOP
        slti    $v0, 0xA
; если он меньше 10, перейти на loc_80 (начало тела цикла):
        bnez    $v0, loc_80
        or      $at, $zero ; branch delay slot, NOP
; заканчиваем, возвращаем 0:
        move    $v0, $zero
; эпилог функции:
        move    $sp, $fp
        lw      $ra, 0x28+saved_RA($sp)
        lw      $fp, 0x28+saved_FP($sp)
        addiu   $sp, 0x28
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP
```

Новая для нас инструкция это **B**. Вернее, это псевдоинструкция (**BEQ**).

15.1.6. Ещё кое-что

По генерируемому коду мы видим следующее: после инициализации *i* выполняется сразу проверка условия *i*, а лишь затем выполняется тело цикла. И это правильно. Потому что если условие в самом начале не выполняется, тело цикла исполнять нельзя.

Так может быть, например, в таком случае:

```
for (i=0; i<total_entries_to_process; i++)
    тело_цикла;
```

Если *total_entries_to_process* равно 0, тело цикла не должно исполниться ни разу. Поэтому проверка условия происходит перед тем как исполнить само тело.

Впрочем, оптимизирующий компилятор может переставить проверку условия и тело цикла местами, если он уверен, что описанная здесь ситуация невозможна, как в случае с нашим простейшим примером и компиляторами Keil, Xcode (LLVM), MSVC и GCC в режиме оптимизации.

15.2. Функция копирования блоков памяти

Настоящие функции копирования памяти могут копировать по 4 или 8 байт на каждой итерации, использовать [SIMD⁴](#), векторизацию, и т.д.

Но ради простоты, этот пример настолько прост, насколько это возможно.

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
};
```

15.2.1. Простейшая реализация

Листинг 15.10: GCC 4.9 x64 оптимизация по размеру (-Os)

```
my_memcpy:
; RDI = целевой адрес
; RSI = исходный адрес
; RDX = размер блока

; инициализировать счетчик (i) в 0
```

⁴Single instruction, multiple data

15.2. ФУНКЦИЯ КОПИРОВАНИЯ БЛОКОВ ПАМЯТИ

```
        xor     eax, eax
.L2:
; все байты скопированы? тогда заканчиваем:
        cmp     rax, rdx
        je      .L5
; загружаем байт по адресу RSI+i:
        mov     cl, BYTE PTR [rsi+rax]
; записываем байт по адресу RDI+i:
        mov     BYTE PTR [rdi+rax], cl
        inc     rax ; i++
        jmp     .L2
.L5:
        ret
```

Листинг 15.11: GCC 4.9 ARM64 оптимизация по размеру (-Os)

```
my_memcpy:
; X0 = целевой адрес
; X1 = исходный адрес
; X2 = размер блока

; инициализировать счетчик (i) в 0
        mov     x3, 0
.L2:
; все байты скопированы? тогда заканчиваем:
        cmp     x3, x2
        beq     .L5
; загружаем байт по адресу X1+i:
        ldrb     w4, [x1,x3]
; записываем байт по адресу X1+i:
        strb     w4, [x0,x3]
        add     x3, x3, 1 ; i++
        b       .L2
.L5:
        ret
```

Листинг 15.12: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
my_memcpy PROC
; R0 = целевой адрес
; R1 = исходный адрес
; R2 = размер блока

        PUSH    {r4,lr}
; инициализировать счетчик (i) в 0
        MOVS    r3,#0
; условие проверяется в конце ф-ции, так что переходим туда:
        B       |L0.12|
|L0.6|
; загружаем байт по адресу R1+i:
        LDRB     r4,[r1,r3]
; записываем байт по адресу R1+i:
        STRB     r4,[r0,r3]
; i++
        ADDS     r3,r3,#1
|L0.12|
; i<size?
        CMP      r3,r2
; перейти на начало цикла, если это так
        BCC      |L0.6|
        POP      {r4,pc}
        ENDP
```

15.2.2. ARM в режиме ARM

Keil в режиме ARM пользуется условными суффиксами:

```

my_memcpy PROC
; R0 = целевой адрес
; R1 = исходный адрес
; R2 = размер блока

; инициализировать счетчик (i) в 0
    MOV     r3,#0
|L0.4|
; все байты скопированы?
    CMP     r3,r2
; следующий блок исполнится только в случае условия "меньше чем",
; т.е. если R2<R3 или i<size.
; загружаем байт по адресу R1+i:
    LDRBCC  r12,[r1,r3]
; записываем байт по адресу R1+i:
    STRBCC  r12,[r0,r3]
; i++
    ADDCC   r3,r3,#1
; последняя инструкция "условного блока".
; перейти на начало цикла, если i<size
; в противном случае, ничего не делать (т.е. если i>=size)
    BCC     |L0.4|
; возврат
    BX      lr
    ENDP

```

Вот почему здесь только одна инструкция перехода вместо двух.

15.2.3. MIPS

Листинг 15.14: GCC 4.4.5 оптимизация по размеру (-Os) (IDA)

```

my_memcpy:
; перейти на ту часть цикла, где проверяется условие:
    b       loc_14
; инициализировать счетчик (i) в 0
; он будет всегда находится в регистре $v0:
    move    $v0, $zero ; branch delay slot

loc_8:                                     # CODE XREF: my_memcpy+1C
; загрузить байт как беззнаковый по адресу $t0 в $v1:
    lbu     $v1, 0($t0)
; инкремент счетчика (i):
    addiu   $v0, 1
; записываем байт по адресу $a3
    sb      $v1, 0($a3)

loc_14:                                    # CODE XREF: my_memcpy
; проверить, до сих пор ли счетчик (i) в $v0 меньше чем третий аргумент ("cnt" в $a2)
    sltu    $v1, $v0, $a2
; сформировать адрес байта исходного блока:
    addu    $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; перейти на тело цикла, если счетчик всё еще меньше чем "cnt":
    bnez    $v1, loc_8
; сформировать адрес байта в целевом блоке ($a3 = $a0+$v0 = dst+i):
    addu    $a3, $a0, $v0 ; branch delay slot
; закончить, если BNEZ не сработала
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP

```

Здесь две новых для нас инструкции: **LBU** («Load Byte Unsigned») и **SB** («Store Byte»). Так же как и в ARM, все регистры в MIPS имеют длину в 32 бита. Здесь нет частей регистров равных байту, как в x86.

Так что когда нужно работать с байтами, приходится выделять целый 32-битный регистр для этого.

LBU загружает байт и сбрасывает все остальные биты («Unsigned»).

15.3. Вывод

И напротив, инструкция `LB` («Load Byte») расширяет байт до 32-битного значения учитывая знак.

`SB` просто записывает байт из младших 8 бит регистра в память.

15.2.4. Векторизация

Оптимизирующий GCC может из этого примера сделать намного больше: [26.1.2](#) (стр. 409).

15.3. Вывод

Примерный скелет цикла от 2 до 9 включительно:

Листинг 15.15: x86

```
mov [counter], 2 ; инициализация
jmp check
body:
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в локальном стеке
add [counter], 1 ; инкремент
check:
cmp [counter], 9
jle body
```

Операция инкремента может быть представлена как 3 инструкции в неоптимизированном коде:

Листинг 15.16: x86

```
MOV [counter], 2 ; инициализация
JMP check
body:
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в локальном стеке
MOV REG, [counter] ; инкремент
INC REG
MOV [counter], REG
check:
CMP [counter], 9
JLE body
```

Если тело цикла короткое, под переменную счетчика можно выделить целый регистр:

Листинг 15.17: x86

```
MOV EBX, 2 ; инициализация
JMP check
body:
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в EBX, но не изменяем её!
INC EBX ; инкремент
check:
CMP EBX, 9
JLE body
```

Некоторые части цикла могут быть сгенерированы компилятором в другом порядке:

Листинг 15.18: x86

```
MOV [counter], 2 ; инициализация
JMP label_check
label_increment:
ADD [counter], 1 ; инкремент
label_check:
CMP [counter], 10
JGE exit
```


15.4. УПРАЖНЕНИЯ

```
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в локальном стеке
JMP label_increment
exit:
```

Обычно условие проверяется *перед* телом цикла, но компилятор может перестроить цикл так, что условие проверяется *после* тела цикла.

Это происходит тогда, когда компилятор уверен, что условие всегда будет *истинно* на первой итерации, так что тело цикла исполнится как минимум один раз:

Листинг 15.19: x86

```
MOV REG, 2 ; инициализация
body:
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в REG, но не изменяем её!
INC REG ; инкремент
CMP REG, 10
JL body
```

Используя инструкцию `LOOP`. Это редкость, компиляторы не используют её. Так что если вы её видите, это верный знак, что этот фрагмент кода написан вручную:

Листинг 15.20: x86

```
; считать от 10 до 1
MOV ECX, 10
body:
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в ECX, но не изменяем её!
LOOP body
```

ARM. В этом примере регистр `R4` выделен для переменной счетчика:

Листинг 15.21: ARM

```
MOV R4, 2 ; инициализация
B check
body:
; тело цикла
; делаем тут что-нибудь
; используем переменную счетчика в R4, но не изменяем её!
ADD R4, R4, #1 ; инкремент
check:
CMP R4, #10
BLT body
```

15.4. Упражнения

- <http://challenges.re/54>
- <http://challenges.re/55>
- <http://challenges.re/56>
- <http://challenges.re/57>

Глава 16

Простая работа с Си-строками

16.1. strlen()

Ещё немного о циклах. Часто функция `strlen()`¹ реализуется при помощи `while()`. Например, вот как это сделано в стандартных библиотеках MSVC:

```
int my_strlen (const char * str)
{
    const char *eos = str;

    while( *eos++ );

    return( eos - str - 1 );
}

int main()
{
    // test
    return my_strlen("hello!");
};
```

16.1.1. x86

Неоптимизирующий MSVC

Итак, компилируем:

```
_eos$ = -4                ; size = 4
_str$ = 8                 ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp] ; взять указатель на символ из "str"
    mov     DWORD PTR _eos$[ebp], eax ; и переложить его в нашу локальную переменную "eos"
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp] ; ECX=eos

    ; взять байт, на который указывает ECX и положить его в EDX расширяя до 32-х бит, учитывая знак

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp] ; EAX=eos
    add     eax, 1                    ; инкремент EAX
    mov     DWORD PTR _eos$[ebp], eax ; положить eax назад в "eos"
    test    edx, edx                  ; EDX ноль?
    je      SHORT $LN1@strlen_        ; да, то что лежит в EDX это ноль, выйти из цикла
    jmp     SHORT $LN2@strlen_        ; продолжаем цикл
$LN1@strlen_:
```

¹ подсчет длины строки в Си

16.1. STRLEN()

```
    ; здесь мы вычисляем разницу двух указателей

    mov     eax, DWORD PTR _eos$[ebp]
    sub     eax, DWORD PTR _str$[ebp]
    sub     eax, 1                ; отнимаем от разницы еще единицу и возвращаем результат
    mov     esp, ebp
    pop     ebp
    ret     0
_strlen_   ENDP
```

Здесь две новых инструкции: **MOVSX** и **TEST**.

О первой. **MOVSX** предназначена для того, чтобы взять байт из какого-либо места в памяти и положить его, в нашем случае, в регистр **EDX**. Но регистр **EDX** — 32-битный. **MOVSX** означает *MOV with Sign-Extend*. Оставшиеся биты с 8-го по 31-й **MOVSX** сделает единицей, если исходный байт в памяти имеет знак *минус*, или заполнит нулями, если знак *плюс*.

И вот зачем всё это.

По умолчанию в MSVC и GCC тип *char* — знаковый. Если у нас есть две переменные, одна *char*, а другая *int* (*int* тоже знаковый), и если в первой переменной лежит -2 (что кодируется как **0xFE**) и мы просто переложим это в *int*, то там будет **0x000000FE**, а это, с точки зрения *int*, даже знакового, будет 254, но никак не -2. -2 в переменной *int* кодируется как **0xFFFFFFFF**. Для того чтобы значение **0xFE** из переменной типа *char* переложить в знаковый *int* с сохранением всего, нужно узнать его знак и затем заполнить остальные биты. Это делает **MOVSX**.

См. также об этом раздел «Представление знака в числах» ([31](#) (стр. [446](#))).

Хотя конкретно здесь компилятору вряд ли была особая надобность хранить значение *char* в регистре **EDX**, а не его восьмибитной части, скажем **DL**. Но получилось, как получилось. Должно быть *register allocator* компилятора сработал именно так.

Позже выполняется **TEST EDX, EDX**. Об инструкции **TEST** читайте в разделе о битовых полях ([20](#) (стр. [300](#))). Конкретно здесь эта инструкция просто проверяет состояние регистра **EDX** на 0.

Неоптимизирующий GCC

Попробуем GCC 4.4.1:

```
strlen      public strlen
proc near
eos         = dword ptr -4
arg_0       = dword ptr 8

    push    ebp
    mov     ebp, esp
    sub     esp, 10h
    mov     eax, [ebp+arg_0]
    mov     [ebp+eos], eax

loc_80483F0:
    mov     eax, [ebp+eos]
    movzx   eax, byte ptr [eax]
    test    al, al
    setnz   al
    add     [ebp+eos], 1
    test    al, al
    jnz     short loc_80483F0
    mov     edx, [ebp+eos]
    mov     eax, [ebp+arg_0]
    mov     ecx, edx
    sub     ecx, eax
    mov     eax, ecx
    sub     eax, 1
    leave
    retn
strlen     endp
```

16.1. STRLEN()

Результат очень похож на MSVC, только здесь используется `MOVZX`, а не `MOVSX`. `MOVZX` означает *MOV with Zero-Extend*. Эта инструкция перекладывает какое-либо значение в регистр и остальные биты выставляет в 0. Фактически, преимущество этой инструкции только в том, что она позволяет заменить две инструкции сразу: `xor eax, eax / mov al, [...]`.

С другой стороны, нам очевидно, что здесь можно было бы написать вот так: `mov al, byte ptr [eax] / test al, al` — это тоже самое, хотя старшие биты `EAX` будут «замусорены». Но будем считать, что это погрешность компилятора — он не смог сделать код более экономным или более понятным. Строго говоря, компилятор вообще не нацелен на то, чтобы генерировать понятный (для человека) код.

Следующая новая инструкция для нас — `SETNZ`. В данном случае, если в `AL` был не ноль, то `test al, al` выставит флаг `ZF` в 0, а `SETNZ`, если `ZF==0` (`NZ` значит *not zero*) выставит 1 в `AL`. Смысл этой процедуры в том, что *если AL не ноль, выполнить переход на loc_80483F0*. Компилятор выдал немного избыточный код, но не будем забывать, что оптимизация выключена.

Оптимизирующий MSVC

Теперь скомпилируем всё то же самое в MSVC 2012, но с включенной оптимизацией (`/Ox`):

Листинг 16.1: Оптимизирующий MSVC 2012 /Ob0

```
_str$ = 8 ; size = 4
_strlen PROC
    mov     edx, DWORD PTR _str$[esp-4] ; EDX -> указатель на строку
    mov     eax, edx ; переложить в EAX
$LL2@strlen:
    mov     cl, BYTE PTR [eax] ; CL = *EAX
    inc     eax ; EAX++
    test    cl, cl ; CL==0?
    jne     SHORT $LL2@strlen ; нет, продолжаем цикл
    sub     eax, edx ; вычисляем разницу указателей
    dec     eax ; декремент EAX
    ret     0
_strlen ENDP
```

Здесь всё попроще стало. Но следует отметить, что компилятор обычно может так хорошо использовать регистры только на небольших функциях с небольшим количеством локальных переменных.

`INC` / `DEC` — это инструкции **инкремента-декремента**. Попросту говоря — увеличить на единицу или уменьшить.

Можем попробовать этот (соптимизированный) пример в OllyDbg. Вот самая первая итерация:

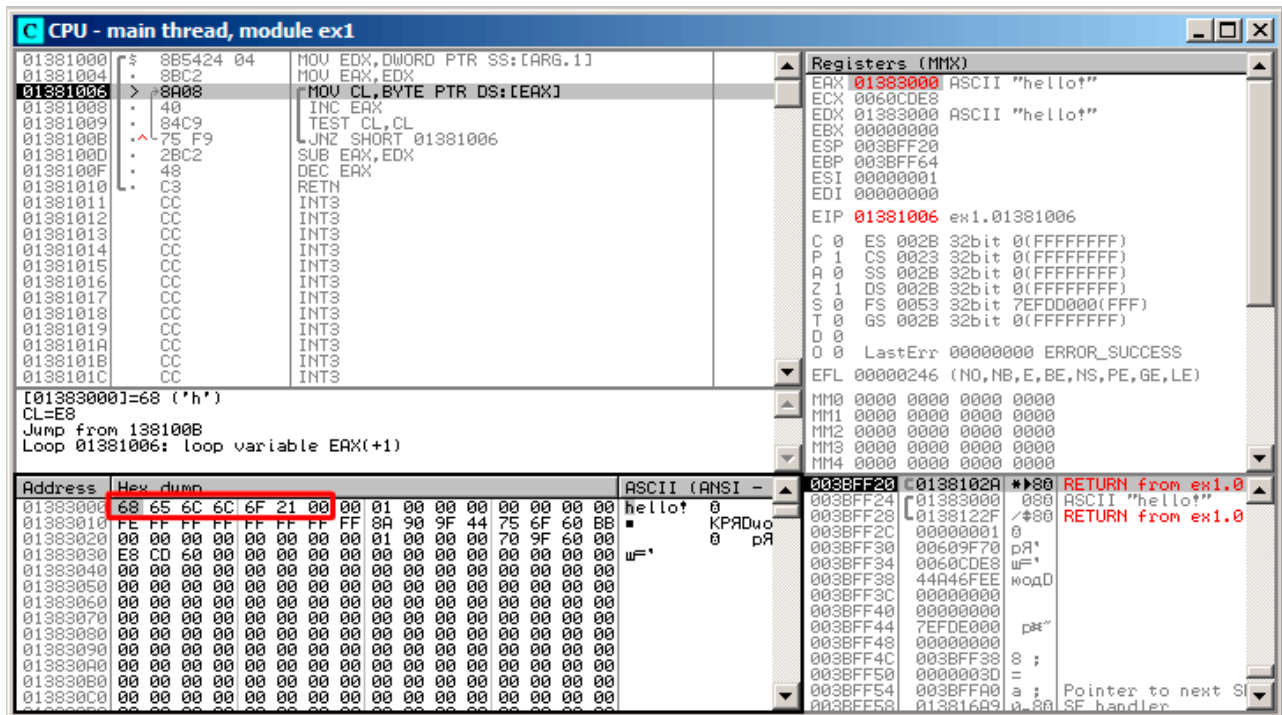


Рис. 16.1: OllyDbg: начало первой итерации

Видно, что OllyDbg обнаружил цикл и, для удобства, свернул инструкции тела цикла в скобке.

Нажав правой кнопкой на **EAX**, можно выбрать «Follow in Dump» и позиция в окне памяти будет как раз там, где надо.

Здесь мы видим в памяти строку «hello!». После неё имеется как минимум 1 нулевой байт, затем случайный мусор. Если OllyDbg видит, что в регистре содержится адрес какой-то строки, он показывает эту строку.

16.1. STRLEN()

Нажмем F8 (сделать шаг, не входя в функцию) столько раз, чтобы текущий адрес снова был в начале тела цикла:

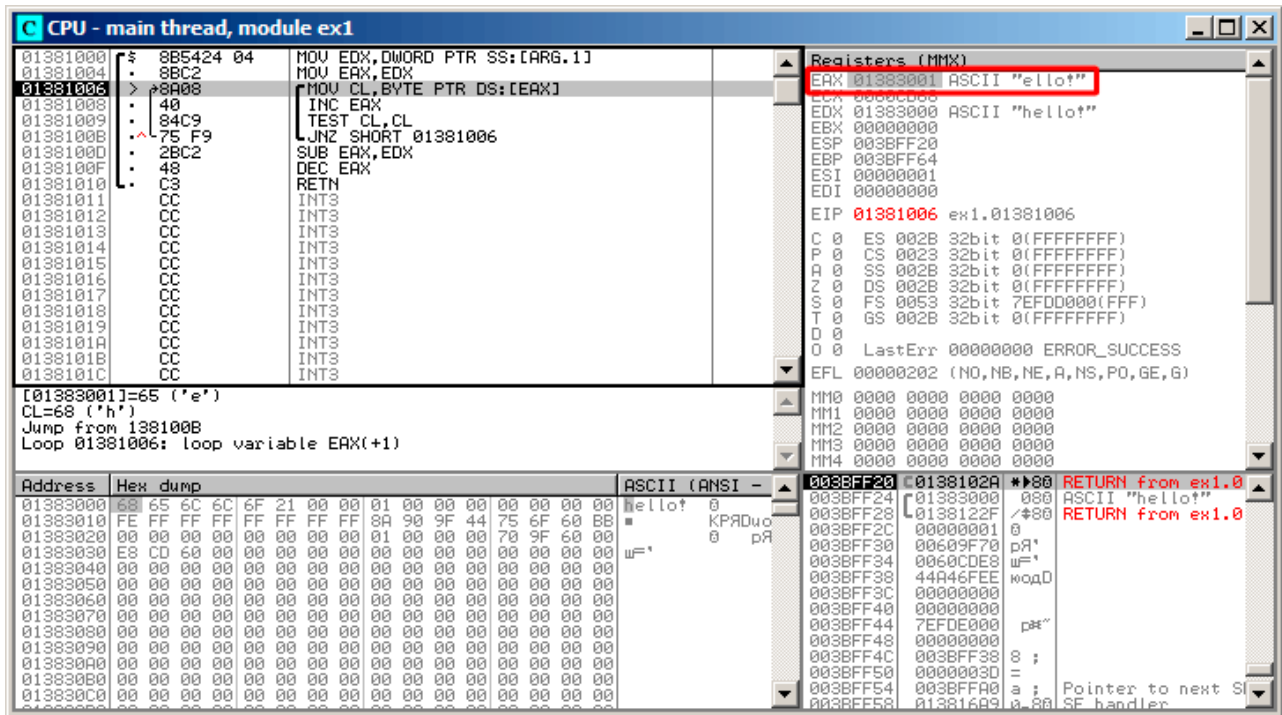


Рис. 16.2: OllyDbg: начало второй итерации

Видно, что **EAX** уже содержит адрес второго символа в строке.

16.1. STRLEN()

Будем нажимать F8 достаточное количество раз, чтобы выйти из цикла:

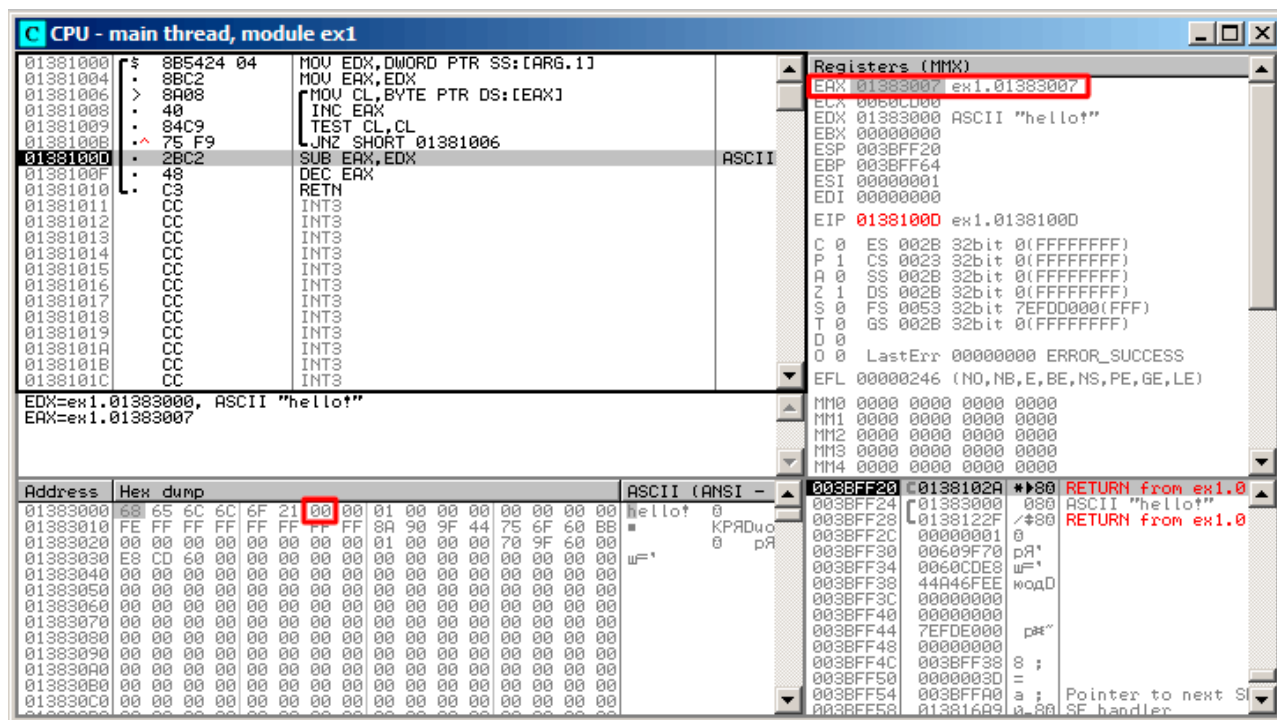


Рис. 16.3: OllyDbg: сейчас будет вычисление разницы указателей

Увидим, что **EAX** теперь содержит адрес нулевого байта, следующего сразу за строкой.

А **EDX** так и не менялся — он всё ещё указывает на начало строки. Здесь сейчас будет вычисляться разница между этими двумя адресами.

16.1. STRLEN()

Инструкция `SUB` исполнилась:

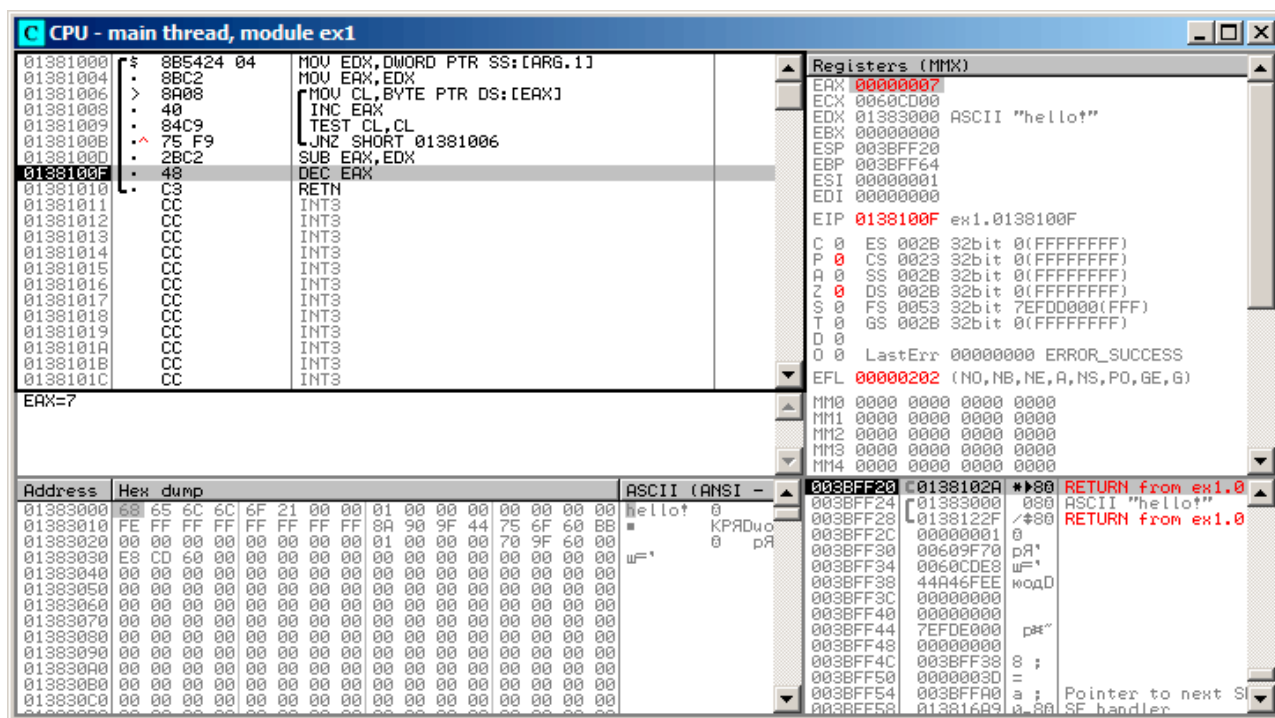


Рис. 16.4: OllyDbg: сейчас будет декремент `EAX`

Разница указателей сейчас в регистре `EAX` — 7.

Действительно, длина строки «hello!» — 6, но вместе с нулевым байтом — 7. Но `strlen()` должна возвращать количество ненулевых символов в строке. Так что сейчас будет исполняться декремент и выход из функции.

Оптимизирующий GCC

Попробуем GCC 4.4.1 с включенной оптимизацией (ключ `-O3`):

```
strlen      public strlen
proc near

arg_0       = dword ptr 8

push       ebp
mov        ebp, esp
mov        ecx, [ebp+arg_0]
mov        eax, ecx

loc_8048418:
movzx      edx, byte ptr [eax]
add        eax, 1
test       dl, dl
jnz        short loc_8048418
not        ecx
add        eax, ecx
pop        ebp
retn

strlen      endp
```

Здесь GCC не очень отстаёт от MSVC за исключением наличия `MOVZX`.

Впрочем, `MOVZX` здесь явно можно заменить на `mov dl, byte ptr [eax]`.

Но возможно, компилятору GCC просто проще помнить, что у него под переменную типа `char` отведен целый 32-битный регистр `EDX` и быть уверенным в том, что старшие биты регистра не будут замусорены.

16.1. STRLEN()

Далее мы видим новую для нас инструкцию `NOT`. Эта инструкция инвертирует все биты в операнде. Можно сказать, что здесь это синонимично инструкции `XOR ECX, 0xffffffffh`. `NOT` и следующая за ней инструкция `ADD` вычисляют разницу указателей и отнимают от результата единицу. Только происходит это слегка по-другому. Сначала `ECX`, где хранится указатель на `str`, инвертируется и от него отнимается единица. См. также раздел: «Представление знака в числах» (31 (стр. 446)).

Иными словами, в конце функции, после цикла, происходит примерно следующее:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

... что эквивалентно:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Но почему GCC решил, что так будет лучше? Трудно угадать. Но наверное, оба эти варианта работают примерно одинаково в плане эффективности и скорости.

16.1.2. ARM

32-битный ARM

Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Листинг 16.2: Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```
_strlen

eos  = -8
str  = -4

    SUB    SP, SP, #8 ; выделить 8 байт для локальных переменных
    STR    R0, [SP,#8+str]
    LDR    R0, [SP,#8+str]
    STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
    LDR    R0, [SP,#8+eos]
    ADD    R1, R0, #1
    STR    R1, [SP,#8+eos]
    LDRSB  R0, [R0]
    CMP    R0, #0
    BEQ    loc_2CD4
    B      loc_2CB8
loc_2CD4 ; CODE XREF: _strlen+24
    LDR    R0, [SP,#8+eos]
    LDR    R1, [SP,#8+str]
    SUB    R0, R0, R1 ; R0=eos-str
    SUB    R0, R0, #1 ; R0=R0-1
    ADD    SP, SP, #8 ; освободить выделенные 8 байт
    BX     LR
```

Неоптимизирующий LLVM генерирует слишком много кода. Зато на этом примере можно посмотреть, как функции работают с локальными переменными в стеке.

В нашей функции только локальных переменных две — это два указателя: `eos` и `str`. В этом листинге сгенерированном при помощи [IDA](#) мы переименовали `var_8` и `var_4` в `eos` и `str` вручную.

Итак, первые несколько инструкций просто сохраняют входное значение в обоих переменных `str` и `eos`.

С метки `loc_2CB8` начинается тело цикла.

16.1. STRLEN()

Первые три инструкции в теле цикла (LDR , ADD , STR) загружают значение *eos* в R0 . Затем происходит инкремент значения и оно сохраняется в локальной переменной *eos* расположенной в стеке.

Следующая инструкция LDRSB R0, [R0] («Load Register Signed Byte») загружает байт из памяти по адресу R0 , расширяет его до 32-бит считая его знаковым (signed) и сохраняет в R0 ². Это немного похоже на инструкцию MOVSB в x86. Компилятор считает этот байт знаковым (signed), потому что тип *char* по стандарту Си — знаковый.

Об этом уже было немного написано (16.1.1 (стр. 197)) в этой же секции, но посвященной x86.

Следует также заметить, что в ARM нет возможности использовать 8-битную или 16-битную часть регистра, как это возможно в x86.

Вероятно, это связано с тем, что за x86 тянется длинный шлейф совместимости со своими предками, вплоть до 16-битного 8086 и даже 8-битного 8080, а ARM разрабатывался с чистого листа как 32-битный RISC-процессор.

Следовательно, чтобы работать с отдельными байтами на ARM, так или иначе придется использовать 32-битные регистры.

Итак, LDRSB загружает символы из строки в R0 , по одному.

Следующие инструкции CMP и BEQ проверяют, является ли этот символ 0.

Если не 0, то происходит переход на начало тела цикла. А если 0, выходим из цикла.

В конце функции вычисляется разница между *eos* и *str*, вычитается единица, и вычисленное значение возвращается через R0 .

Н.В. В этой функции не сохранялись регистры. По стандарту регистры R0 - R3 называются также «scratch registers». Они предназначены для передачи аргументов и их значения не нужно восстанавливать при выходе из функции, потому что они больше не нужны в вызывающей функции. Таким образом, их можно использовать как захочется.

А так как никакие больше регистры не используются, то и сохранять нечего.

Поэтому управление можно вернуть вызывающей функции простым переходом (BX) по адресу в регистре LR.

Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

Листинг 16.3: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

```
_strlen
    MOV        R1, R0

loc_2DF6
    LDRB.W     R2, [R1], #1
    CMP        R2, #0
    BNE        loc_2DF6
    MVNS       R0, R0
    ADD        R0, R1
    BX         LR
```

Оптимизирующий LLVM решил, что под переменные *eos* и *str* выделять место в стеке не обязательно, и эти переменные можно хранить прямо в регистрах.

Перед началом тела цикла *str* будет находиться в R0 , а *eos* — в R1 .

Инструкция LDRB.W R2, [R1], #1 загружает в R2 байт из памяти по адресу R1 , расширяя его как знаковый (signed), до 32-битного значения, но не только это.

#1 в конце инструкции означает «Post-indexed addressing», т.е. после загрузки байта к R1 добавится единица.

Читайте больше об этом: 29.2 (стр. 439).

Далее в теле цикла можно увидеть CMP и BNE³. Они продолжают работу цикла до тех пор, пока не будет встречен 0.

После конца цикла MVNS⁴ (инвертирование всех бит, NOT в x86) и ADD вычисляют $eos - str - 1$. На самом деле, эти две инструкции вычисляют $R0 = str + eos$, что эквивалентно тому, что было в исходном коде. Почему это так, уже было описано чуть раньше, здесь (16.1.1 (стр. 202)).

Вероятно, LLVM, как и GCC, посчитал, что такой код может быть короче (или быстрее).

²Компилятор Keil считает тип *char* знаковым, как и MSVC и GCC.

³(PowerPC, ARM) Branch if Not Equal

⁴MoVe Not

Листинг 16.4: Оптимизирующий Keil 6/2013 (Режим ARM)

```

_strlen
    MOV     R1, R0

loc_2C8
    LDRB    R2, [R1], #1
    CMP     R2, #0
    SUBEQ   R0, R1, R0
    SUBEQ   R0, R0, #1
    BNE     loc_2C8
    BX      LR

```

Практически то же самое, что мы уже видели, за тем исключением, что выражение $str - eos - 1$ может быть вычислено не в самом конце функции, а прямо в теле цикла.

Суффикс **-EQ** означает, что инструкция будет выполнена только если операнды в исполненной перед этим инструкции **CMP** были равны.

Таким образом, если в **R0** будет 0, обе инструкции **SUBEQ** исполнятся и результат останется в **R0**.

ARM64

Оптимизирующий GCC (Linaro) 4.9

```

my_strlen:
    mov     x1, x0
    ; X1 теперь временный регистр (eos), работающий, как курсор
.L58:
    ; загрузить байт из X1 в W2, инкремент X1 (пост-индекс)
    ldrb    w2, [x1], 1
    ; Compare and Branch if NonZero: сравнить W0 с нулем, перейти на .L58 если не ноль
    cbnz    w2, .L58
    ; вычислить разницу между изначальным указателем в X0 и текущим адресом в X1
    sub     x0, x1, x0
    ; декремент младших 32-х бит результата
    sub     w0, w0, #1
    ret

```

Алгоритм такой же как и в [16.1.1](#) (стр. 198): найти нулевой байт, затем вычислить разницу между указателями, затем отнять 1 от результата. Комментарии добавлены автором книги.

Стоит добавить, что наш пример имеет ошибку: **my_strlen()** возвращает 32-битный *int*, тогда как должна возвращать **size_t** или иной 64-битный тип.

Причина в том, что теоретически, **strlen()** можно вызывать для огромных блоков в памяти, превышающих 4GB, так что она должна иметь возможность вернуть 64-битное значение на 64-битной платформе.

Так что из-за моей ошибки, последняя инструкция **SUB** работает над 32-битной частью регистра, тогда как предпоследняя **SUB** работает с полными 64-битными частями (она вычисляет разницу между указателями).

Это моя ошибка, но лучше оставить это как есть, как пример кода, который возможен в таком случае.

Неоптимизирующий GCC (Linaro) 4.9

```

my_strlen:
; пролог функции
    sub     sp, sp, #32
; первый аргумент (str) будет записан в [sp,8]
    str     x0, [sp,8]
    ldr     x0, [sp,8]
; скопировать переменную "str" в "eos"
    str     x0, [sp,24]

```

16.1. STRLEN()

```
        nop
.L62:
; eos++
        ldr     x0, [sp,24] ; загрузить "eos" в X0
        add     x1, x0, 1   ; инкремент X0
        str     x1, [sp,24] ; сохранить X0 в "eos"
; загрузить байт из памяти по адресу в X0 в W0
        ldrb    w0, [x0]
; это ноль? (WZR это 32-битный регистр всегда содержащий ноль)
        cmp     w0, wzr
; переход если не ноль (Branch Not Equal)
        bne     .L62
; найден нулевой байт. вычисляем разницу.
; загрузить "eos" в X1
        ldr     x1, [sp,24]
; загрузить "str" в X0
        ldr     x0, [sp,8]
; вычислить разницу
        sub     x0, x1, x0
; декремент результата
        sub     w0, w0, #1
; эпилог функции
        add     sp, sp, 32
        ret
```

Более многословно. Переменные часто сохраняются в память и загружаются назад (локальный стек). Здесь та же ошибка: операция декремента происходит над 32-битной частью регистра.

16.1.3. MIPS

Листинг 16.5: Оптимизирующий GCC 4.4.5 (IDA)

```
my_strlen:
; переменная "eos" всегда будет находиться в $v1:
        move    $v1, $a0

loc_4:
; загрузить байт по адресу в "eos" в $a1:
        lb      $a1, 0($v1)
        or      $at, $zero ; load delay slot, NOP
; если загруженный байт не ноль, перейти на loc_4:
        bnez    $a1, loc_4
; в любом случае, инкрементируем "eos":
        addiu   $v1, 1 ; branch delay slot
; цикл закончен. инвертируем переменную "str":
        nor     $v0, $zero, $a0
; $v0=-str-1
        jr      $ra
; возвращаемое значение = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
        addu    $v0, $v1, $v0 ; branch delay slot
```

В MIPS нет инструкции **NOT**, но есть **NOR** — операция **OR + NOT**.

Эта операция широко применяется в цифровой электронике⁵. Например, космический компьютер Apollo Guidance Computer использовавшийся в программе «Аполлон» был построен исключительно на 5600 элементах **NOR**: [Jens Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, (2011)]. Но элемент NOR не очень популярен в программировании.

Так что операция **NOT** реализована здесь как **NOR DST, \$ZERO, SRC**.

Из фундаментальных знаний [31](#) (стр. 446), мы можем знать, что побитовое инвертирование знакового числа это то же что и смена его знака с вычитанием 1 из результата.

Так что **NOT** берет значение *str* и трансформирует его в $-str - 1$.

Следующая операция сложения готовит результат.

⁵ **NOR** называют «универсальным элементом»

Глава 17

Замена одних арифметических инструкций на другие

В целях оптимизации одна инструкция может быть заменена другой, или даже группой инструкций. Например, `ADD` и `SUB` могут заменять друг друга: строка 18 в листинг.53.1.

Более того, не всегда замена тривиальна. Инструкция `LEA`, несмотря на оригинальное назначение, нередко применяется для простых арифметических действий: A.6.2 (стр. 917).

17.1. Умножение

17.1.1. Умножение при помощи сложения

Вот простой пример:

Листинг 17.1: Оптимизирующий MSVC 2010

```
unsigned int f(unsigned int a)
{
    return a*8;
};
```

Умножение на 8 заменяется на три инструкции сложения, делающих то же самое. Должно быть, оптимизатор в MSVC решил, что этот код может быть быстрее.

```
_TEXT SEGMENT
_a$ = 8 ; size = 4
_f PROC
; File c:\polygon\c\2.c
    mov     eax, DWORD PTR _a$[esp-4]
    add     eax, eax
    add     eax, eax
    add     eax, eax
    ret     0
_f ENDP
_TEXT ENDS
END
```

17.1.2. Умножение при помощи сдвигов

Ещё очень часто умножения и деления на числа вида 2^n заменяются на инструкции сдвигов.

```
unsigned int f(unsigned int a)
{
    return a*4;
};
```

Листинг 17.2: Неоптимизирующий MSVC 2010

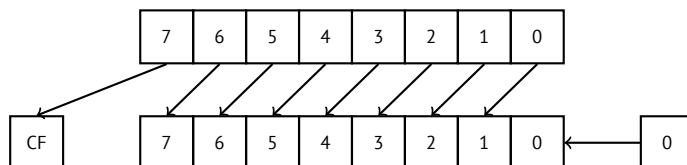
```

_a$ = 8      ; size = 4
_f PROC
    push     ebp
    mov      ebp, esp
    mov      eax, DWORD PTR _a$[ebp]
    shl      eax, 2
    pop      ebp
    ret      0
_f ENDP

```

Умножить на 4 это просто сдвинуть число на 2 бита влево, вставив 2 нулевых бита справа (как два самых младших бита). Это как умножить 3 на 100 — нужно просто дописать два нуля справа.

Вот как работает инструкция сдвига влево:



Добавленные биты справа — всегда нули.

Умножение на 4 в ARM:

Листинг 17.3: Неоптимизирующий Keil 6/2013 (Режим ARM)

```

f PROC
    LSL      r0, r0, #2
    BX       lr
    ENDP

```

Умножение на 4 в MIPS:

Листинг 17.4: Оптимизирующий GCC 4.4.5 (IDA)

```

jr      $ra
sll      $v0, $a0, 2 ; branch delay slot

```

SLL это «Shift Left Logical».

17.1.3. Умножение при помощи сдвигов, сложений и вычитаний

Можно избавиться от операции умножения, если вы умножаете на числа вроде 7 или 17, и использовать сдвиги.

Здесь используется относительно простая математика.

32-бита

```

#include <stdint.h>

int f1(int a)
{
    return a*7;
};

int f2(int a)
{
    return a*28;
};

int f3(int a)
{
    return a*17;
};

```

Листинг 17.5: Оптимизирующий MSVC 2012

```

; a*7
_a$ = 8
_f1 PROC
    mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
    lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
    sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    ret     0
_f1 ENDP

; a*28
_a$ = 8
_f2 PROC
    mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
    lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
    sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    shl     eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
    ret     0
_f2 ENDP

; a*17
_a$ = 8
_f3 PROC
    mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
    shl     eax, 4
; EAX=EAX<<4=EAX*16=a*16
    add     eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
    ret     0
_f3 ENDP

```

ARM

Keil, генерируя код для режима ARM, использует модификаторы инструкции, в которых можно задавать сдвиг для второго операнда:

Листинг 17.6: Оптимизирующий Keil 6/2013 (Режим ARM)

```

; a*7
||f1|| PROC
    RSB     r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
    BX     lr
    ENDP

; a*28
||f2|| PROC
    RSB     r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
    LSL     r0,r0,#2
; R0=R0<<2=R0*4=a*7*4=a*28
    BX     lr
    ENDP

; a*17
||f3|| PROC
    ADD     r0,r0,r0,LSL #4

```

17.1. УМНОЖЕНИЕ

```
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
    BX    lr
    ENDP
```

Но таких модификаторов в режиме Thumb нет.

И он также не смог оптимизировать функцию `f2()` :

Листинг 17.7: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
; a*7
||f1|| PROC
    LSL    r1,r0,#3
; R1=R0<<3=a<<3=a*8
    SUB    r0,r1,r0
; R0=R1-R0=a*8-a=a*7
    BX    lr
    ENDP

; a*28
||f2|| PROC
    MOVS    r1,#0x1c ; 28
; R1=28
    MUL    r0,r1,r0
; R0=R1*R0=28*a
    BX    lr
    ENDP

; a*17
||f3|| PROC
    LSL    r1,r0,#4
; R1=R0<<4=R0*16=a*16
    ADDS    r0,r0,r1
; R0=R0+R1=a+a*16=a*17
    BX    lr
    ENDP
```

MIPS

Листинг 17.8: Оптимизирующий GCC 4.4.5 (IDA)

```
_f1:
    sll     $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
    jr      $ra
    subu    $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2:
    sll     $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
    sll     $a0, 2
; $a0 = $a0<<2 = $a0*4
    jr      $ra
    subu    $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

_f3:
    sll     $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
    jr      $ra
    addu    $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17
```

64-бита


```
#include <stdint.h>

int64_t f1(int64_t a)
{
    return a*7;
};

int64_t f2(int64_t a)
{
    return a*28;
};

int64_t f3(int64_t a)
{
    return a*17;
};
```

x64

Листинг 17.9: Оптимизирующий MSVC 2012

```
; a*7
f1:
    lea    rax, [0+rdi*8]
; RAX=RDI*8=a*8
    sub    rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
    ret

; a*28
f2:
    lea    rax, [0+rdi*4]
; RAX=RDI*4=a*4
    sal    rdi, 5
; RDI=RDI<<5=RDI*32=a*32
    sub    rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
    mov    rax, rdi
    ret

; a*17
f3:
    mov    rax, rdi
    sal    rax, 4
; RAX=RAX<<4=a*16
    add    rax, rdi
; RAX=a*16+a=a*17
    ret
```

ARM64

GCC 4.9 для ARM64 также очень лаконичен благодаря модификаторам сдвига:

Листинг 17.10: Оптимизирующий GCC (Linaro) 4.9 ARM64

```
; a*7
f1:
    lsl    x1, x0, 3
; X1=X0<<3=X0*8=a*8
    sub    x0, x1, x0
; X0=X1-X0=a*8-a=a*7
    ret

; a*28
f2:
```

17.2. ДЕЛЕНИЕ

```
    lsl    x1, x0, 5
; X1=X0<<5=a*32
    sub    x0, x1, x0, lsl 2
; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
    ret

; a*17
f3:
    add    x0, x0, x0, lsl 4
; X0=X0+X0<<4=a+a*16=a*17
    ret
```

17.2. Деление

17.2.1. Деление используя сдвиги

Например, возьмем деление на 4:

```
unsigned int f(unsigned int a)
{
    return a/4;
};
```

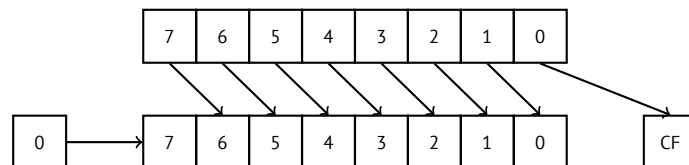
Имеем в итоге (MSVC 2010):

Листинг 17.11: MSVC 2010

```
_a$ = 8 ; size = 4
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    shr     eax, 2
    ret     0
_f ENDP
```

Инструкция **SHR** (*Shift Right*) в данном примере сдвигает число на 2 бита вправо. При этом освободившиеся два бита слева (т.е. самые старшие разряды) выставляются в нули. А самые младшие 2 бита выкидываются. Фактически, эти два выкинутых бита — остаток от деления.

Инструкция **SHR** работает так же как и **SHL**, только в другую сторону.



Для того, чтобы это проще понять, представьте себе десятичную систему счисления и число 23. 23 можно разделить на 10 просто откинув последний разряд (3 — это остаток от деления). После этой операции останется 2 как **частное**.

Так что остаток выбрасывается, но это нормально, мы все-таки работаем с целочисленными значениями, а не с **вещественными**!

Деление на 4 в ARM:

Листинг 17.12: Неоптимизирующий Keil 6/2013 (Режим ARM)

```
f PROC
    LSR     r0, r0, #2
    BX      lr
    ENDP
```

Деление на 4 в MIPS:

Листинг 17.13: Оптимизирующий GCC 4.4.5 (IDA)

```
jr    $ra
srli   $v0, $a0, 2 ; branch delay slot
```

Инструкция SRL это «Shift Right Logical».

17.3. Упражнение

- <http://challenges.re/59>

Глава 18

Работа с FPU

FPU — блок в процессоре работающий с числами с плавающей запятой.

Раньше он назывался «сопроцессором» и он стоит немного в стороне от **CPU**.

18.1. IEEE 754

Число с плавающей точкой в формате IEEE 754 состоит из *знака, мантиссы*¹ и *экспоненты*.

18.2. x86

Перед изучением **FPU** в x86 полезно ознакомиться с тем как работают стековые машины² или ознакомиться с основами языка Forth³.

Интересен факт, что в свое время (до 80486) сопроцессор был отдельным чипом на материнской плате, и вследствие его высокой цены, он не всегда присутствовал. Его можно было докупить и установить отдельно⁴. Начиная с 80486 DX в состав процессора всегда входит FPU.

Этот факт может напоминать такой рудимент как наличие инструкции **FWAIT**, которая заставляет **CPU** ожидать, пока **FPU** закончит работу. Другой рудимент это тот факт, что опкоды **FPU**-инструкций начинаются с т.н. «escape»-опкодов (**D8** . **DF**) как опкоды, передающиеся в отдельный сопроцессор.

FPU имеет стек из восьми 80-битных регистров: **ST(0)** .. **ST(7)** . Для краткости, IDA и OllyDbg отображают **ST(0)** как **ST**, что в некоторых учебниках и документах означает «Stack Top» («вершина стека»). Каждый регистр может содержать число в формате IEEE 754⁵.

18.3. ARM, MIPS, x86/x64 SIMD

В ARM и MIPS FPU это не стек, а просто набор регистров.

Такая же идеология применяется в расширениях SIMD в процессорах x86/x64.

¹*significand* или *fraction* в англоязычной литературе

²wikipedia.org/wiki/Stack_machine

³[wikipedia.org/wiki/Forth_\(programming_language\)](https://wikipedia.org/wiki/Forth_(programming_language))

⁴Например, Джон Кармак использовал в своей игре Doom числа с фиксированной запятой (ru.wikipedia.org/wiki/Число_с_фиксированной_запятой), хранящиеся в обычных 32-битных **GPR** (16 бит на целую часть и 16 на дробную), чтобы Doom работал на 32-битных компьютерах без FPU, т.е. 80386 и 80486 SX.

⁵wikipedia.org/wiki/IEEE_floating_point

18.4. Си/Си++

В стандартных Си/Си++ имеются два типа для работы с числами с плавающей запятой: *float* (число одинарной точности⁶, 32 бита)⁷ и *double* (число двойной точности⁸, 64 бита).

GCC также поддерживает тип *long double* (extended precision⁹, 80 бит), но MSVC — нет.

Несмотря на то, что *float* занимает столько же места, сколько и *int* на 32-битной архитектуре, представление чисел, разумеется, совершенно другое.

18.5. Простой пример

Рассмотрим простой пример:

```
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
};
```

18.5.1. x86

MSVC

Компилируем в MSVC 2010:

Листинг 18.1: MSVC 2010: f()

```
CONST    SEGMENT
__real@4010666666666666 DQ 0401066666666666r    ; 4.1
CONST    ENDS
CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_f PROC
    push    ebp
    mov     ebp, esp
    fld     QWORD PTR _a$[ebp]

; текущее состояние стека: ST(0) = _a

    fdiv    QWORD PTR __real@40091eb851eb851f

; текущее состояние стека: ST(0) = результат деления _a на 3.14

    fld     QWORD PTR _b$[ebp]

; текущее состояние стека: ST(0) = _b; ST(1) = результат деления _a на 3.14

    fmul    QWORD PTR __real@4010666666666666
```

⁶wikipedia.org/wiki/Single-precision_floating-point_format

⁷Формат представления чисел с плавающей точкой одинарной точности затрагивается в разделе *Работа с типом float как со структурой* (22.6.2 (стр. 368)).

⁸wikipedia.org/wiki/Double-precision_floating-point_format

⁹wikipedia.org/wiki/Extended_precision

18.5. ПРОСТОЙ ПРИМЕР

```
; текущее состояние стека:  
; ST(0) = результат умножения _b на 4.1;  
; ST(1) = результат деления _a на 3.14  
  
faddp ST(1), ST(0)  
  
; текущее состояние стека: ST(0) = результат сложения  
  
pop    ebp  
ret    0  
_f    ENDP
```

FLD берет 8 байт из стека и загружает их в регистр **ST(0)**, автоматически конвертируя во внутренний 80-битный формат (*extended precision*).

FDIV делит содержимое регистра **ST(0)** на число, лежащее по адресу `__real@40091eb851eb851f` — там закодировано значение 3,14. Синтаксис ассемблера не поддерживает подобные числа, поэтому мы там видим шестнадцатеричное представление числа 3,14 в формате IEEE 754.

После выполнения **FDIV** в **ST(0)** остается *частное*.

Кстати, есть ещё инструкция **FDIVP**, которая делит **ST(1)** на **ST(0)**, выталкивает эти числа из стека и заталкивает результат. Если вы знаете язык Forth¹⁰, то это как раз оно и есть — стековая машина¹¹.

Следующая **FLD** заталкивает в стек значение *b*.

После этого в **ST(1)** перемещается результат деления, а в **ST(0)** теперь *b*.

Следующий **FMUL** умножает *b* из **ST(0)** на значение `__real@4010666666666666` — там лежит число 4,1 — и оставляет результат в **ST(0)**.

Самая последняя инструкция **FADDP** складывает два значения из вершины стека в **ST(1)** и затем выталкивает значение, лежащее в **ST(0)**. Таким образом результат сложения остается на вершине стека в **ST(0)**.

Функция должна вернуть результат в **ST(0)**, так что больше ничего здесь не производится, кроме эпилога функции.

¹⁰[wikipedia.org/wiki/Forth_\(programming_language\)](https://wikipedia.org/wiki/Forth_(programming_language))

¹¹wikipedia.org/wiki/Stack_machine

2 пары 32-битных слов обведены в стеке красным. Каждая пара – это числа двойной точности в формате IEEE 754, переданные из `main()`.

Видно, как первая `FLD` загружает значение 1,2 из стека и помещает в регистр `ST(0)` :

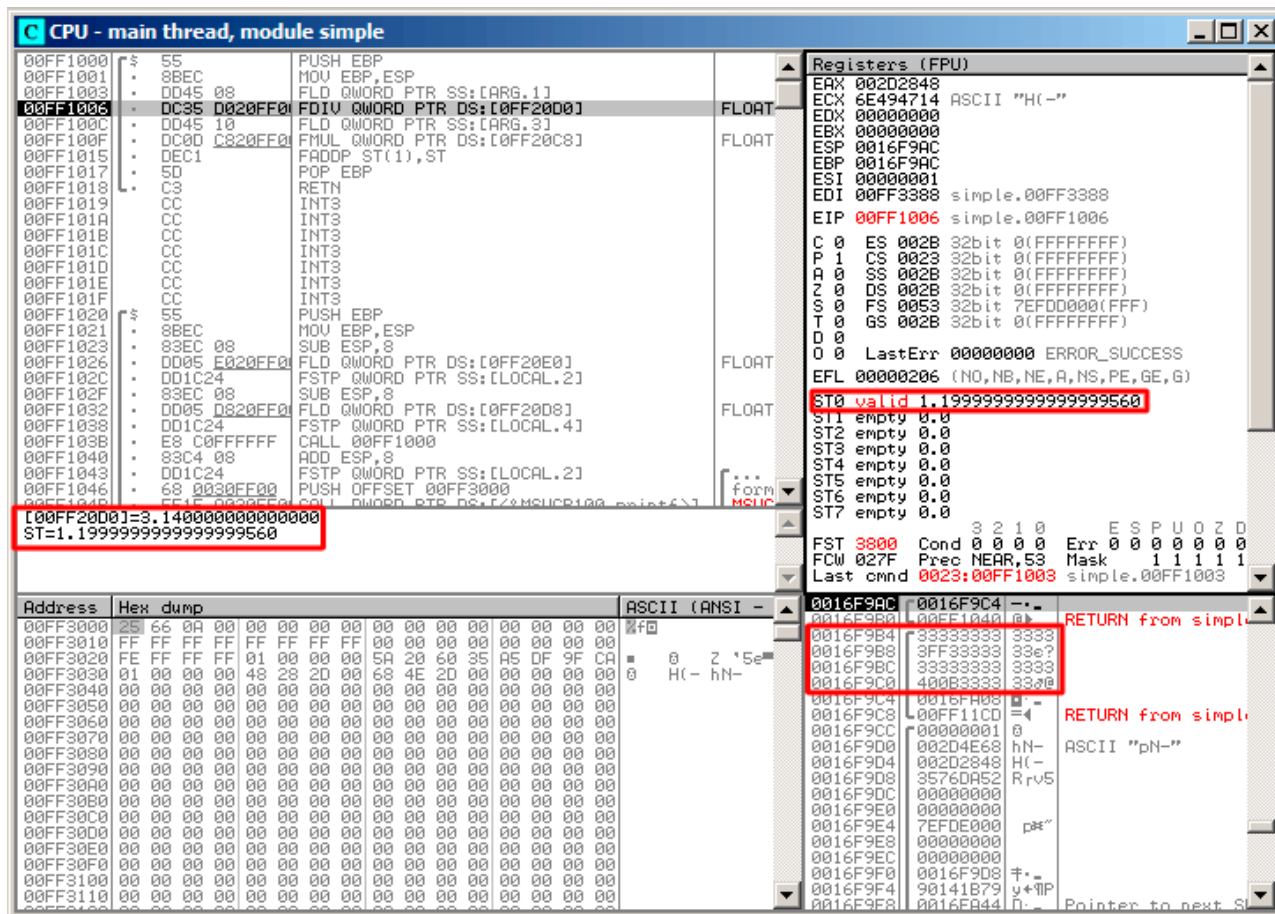


Рис. 18.1: OllyDbg: первая `FLD` исполнилась

Из-за неизбежных ошибок конвертирования числа из 64-битного IEEE 754 в 80-битное (внутреннее в FPU), мы видим здесь 1,1999..., что очень близко к 1,2.

Прямо сейчас `EIP` указывает на следующую инструкцию (`FDIV`), загружающую константу двойной точности из памяти. Для удобства, OllyDbg показывает её значение: 3,14.

18.5. ПРОСТОЙ ПРИМЕР

Трассируем дальше. `FDIV` исполнилась, теперь `ST(0)` содержит 0,382...(quotient):

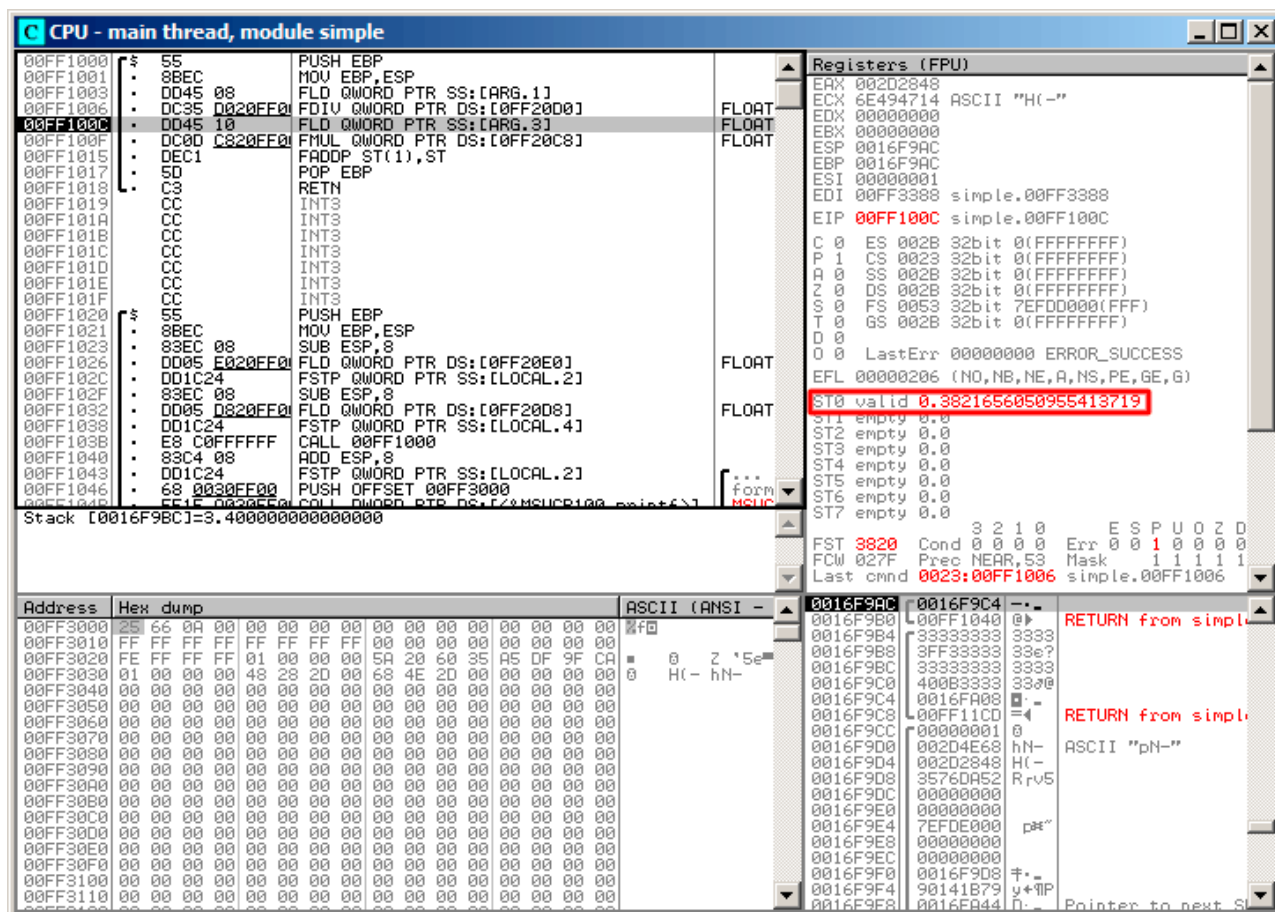


Рис. 18.2: OllyDbg: `FDIV` исполнилась

18.5. ПРОСТОЙ ПРИМЕР

Третий шаг: вторая **FLD** исполнилась, загрузив в **ST(0)** 3,4 (мы видим приближенное число 3,39999...):

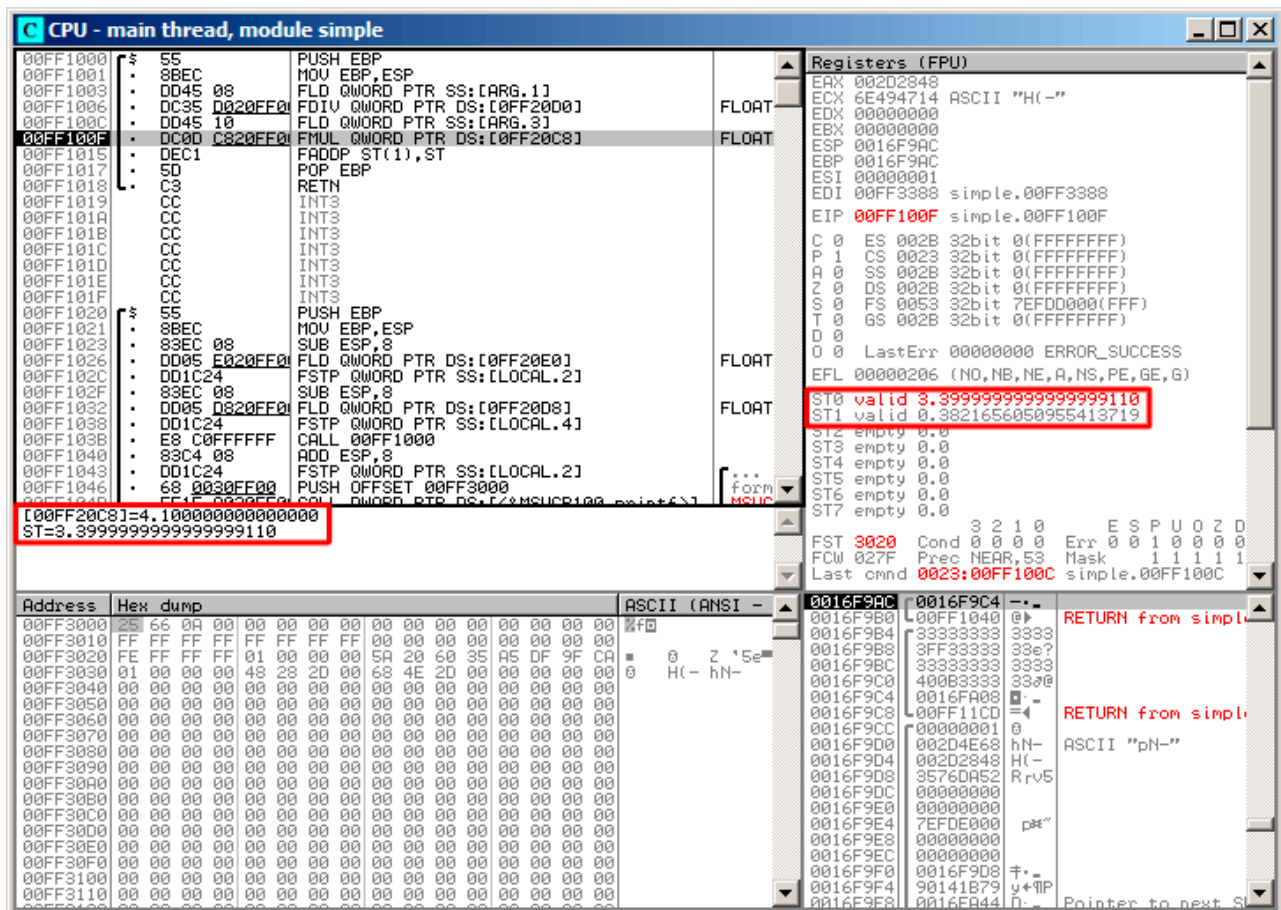


Рис. 18.3: OllyDbg: вторая **FLD** исполнилась

В это время **quotient** провалилось в **ST(1)**. **EIP** указывает на следующую инструкцию: **FMUL**. Она загружает константу 4,1 из памяти, так что OllyDbg тоже показывает её здесь.

18.5. ПРОСТОЙ ПРИМЕР

Затем: FMUL исполнилась, теперь в ST(0) произведение:

The screenshot shows the OllyDbg interface with the CPU window displaying assembly code for the 'simple' module. The instruction at address 00FF1015 is FMUL QWORD PTR DS:[0FF20C8], which has been executed. The Registers (FPU) window shows the state of the FPU registers, with ST(0) and ST(1) highlighted in red. The value in ST(0) is 13.93999999999997730, and the value in ST(1) is 0.3821656050955413719. The CPU window also shows the state of the registers, with EAX, ECX, EDI, and ESI highlighted in red.

Address	Hex dump	ASCII (ANSI -)
00FF1000	55 8B EC 08	PUSH EBP
00FF1003	DD 45 08	MOV EBP, ESP
00FF1006	DC 35 00 20 FF 00	FLOP QWORD PTR DS:[0FF20D0]
00FF1009	DD 45 10	FLOP QWORD PTR SS:[ARG.3]
00FF100F	DC 0D C8 20 FF 00	FMUL QWORD PTR DS:[0FF20C8]
00FF1015	DEC 1	ADD SP, 4
00FF1017	5D	POP EBP
00FF1018	C3	RET
00FF1019	CC	INT3
00FF101A	CC	INT3
00FF101B	CC	INT3
00FF101C	CC	INT3
00FF101D	CC	INT3
00FF101E	CC	INT3
00FF101F	CC	INT3
00FF1020	55 8B EC 08	PUSH EBP
00FF1023	DD 45 08	MOV EBP, ESP
00FF1026	DD 45 08	SUB ESP, 8
00FF102C	DD 1C 24	FSTP QWORD PTR SS:[LOCAL.2]
00FF102F	8B EC 08	SUB ESP, 8
00FF1032	DD 05 08 20 FF 00	FLOP QWORD PTR DS:[0FF20D8]
00FF1038	DD 1C 24	FSTP QWORD PTR SS:[LOCAL.4]
00FF103B	E8 C0 FF FF	CALL 00FF1000
00FF1040	8B EC 08	ADD ESP, 8
00FF1043	DD 1C 24	FSTP QWORD PTR SS:[LOCAL.2]
00FF1046	68 00 30 FF 00	PUSH OFFSET 00FF3000
00FF104D	FF 15 00 30 FF 00	CALL QWORD PTR DS:[0FF3000]

Register	Value
EAX	002D2848
ECX	6E494714 ASCII "H(-"
EDX	00000000
EBX	00000000
ESP	0016F9AC
EBP	0016F9AC
ESI	00000001
EDI	00FF3388 simple.00FF3388
EIP	00FF1015 simple.00FF1015
C 0	ES 002B 32bit 0(FFFFFFFF)
P 1	CS 0023 32bit 0(FFFFFFFF)
A 0	SS 002B 32bit 0(FFFFFFFF)
Z 0	DS 002B 32bit 0(FFFFFFFF)
S 0	FS 0053 32bit 7EFD0000(FFF)
T 0	GS 002B 32bit 0(FFFFFFFF)
D 0	
O 0	LastErr 00000000 ERROR_SUCCESS
EFL	00000206 (NO,NB,NE,A,NS,PE,GE,G)
ST0	valid 13.93999999999997730
ST1	valid 0.3821656050955413719
ST2	empty 0.0
ST3	empty 0.0
ST4	empty 0.0
ST5	empty 0.0
ST6	empty 0.0
ST7	empty 0.0

Рис. 18.4: OllyDbg: FMUL исполнилась

18.5. ПРОСТОЙ ПРИМЕР

Затем: `FADDP` исполнилась, теперь в `ST(0)` сумма, а `ST(1)` очистился:

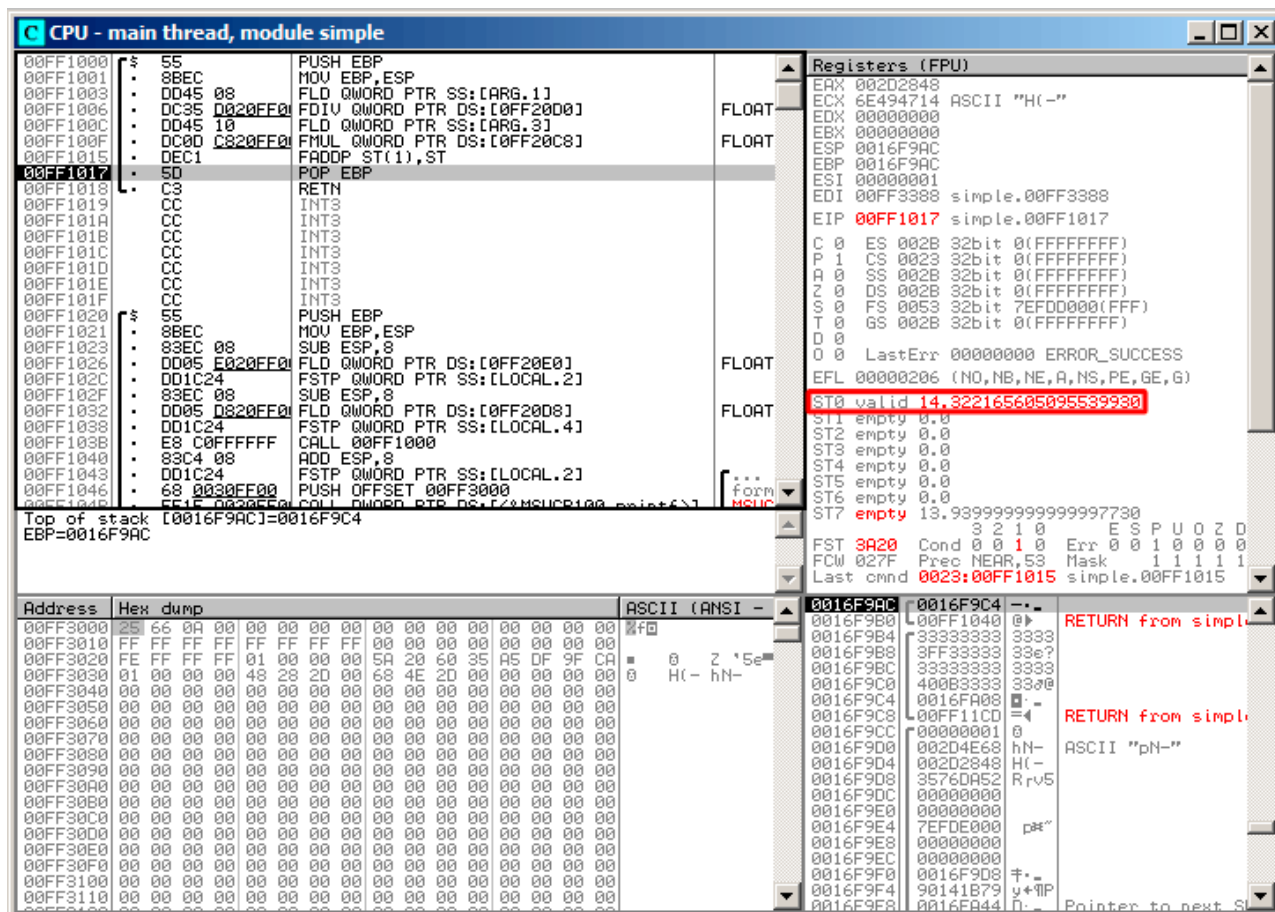


Рис. 18.5: OllyDbg: `FADDP` исполнилась

Сумма остается в `ST(0)` потому что функция возвращает результат своей работы через `ST(0)`.

Позже `main()` возьмет это значение оттуда.

Мы также видим кое-что необычное: значение 13,93...теперь находится в `ST(7)`.

Почему?

Мы читали в этой книге, что регистры в `FPU` представляют собой стек: 18.2 (стр. 214). Но это упрощение. Представьте, если бы в железе было бы так, как описано. Тогда при каждом заталкивании (или выталкивании) в стек, все остальные 7 значений нужно было бы передвигать (или копировать) в соседние регистры, а это слишком затратно.

Так что в реальности у `FPU` есть просто 8 регистров и указатель (называемый `TOP`), содержащий номер регистра, который в текущий момент является «вершиной стека».

При заталкивании значения в стек регистр `TOP` меняется, и указывает на свободный регистр. Затем значение записывается в этот регистр.

При выталкивании значения из стека процедура обратная. Однако освобожденный регистр не обнуляется (наверное, можно было бы сделать, чтобы обнулялся, но это лишняя работа и работало бы медленнее). Так что это мы здесь и видим. Можно сказать, что `FADDP` сохранила сумму, а затем вытолкнула один элемент.

Но в реальности, эта инструкция сохранила сумму и затем передвинула регистр `TOP`.

Было бы ещё точнее сказать, что регистры `FPU` представляют собой кольцевой буфер.

GCC

GCC 4.4.1 (с опцией `-O3`) генерирует похожий код, хотя и с некоторой разницей:

Листинг 18.2: Оптимизирующий GCC 4.4.1

```
public f
f proc near
```

18.5. ПРОСТОЙ ПРИМЕР

```
arg_0      = qword ptr 8
arg_8      = qword ptr 10h

        push    ebp
        fld     ds:dbl_8048608 ; 3.14

; состояние стека сейчас: ST(0) = 3.14

        mov     ebp, esp
        fdivr   [ebp+arg_0]

; состояние стека сейчас: ST(0) = результат деления

        fld     ds:dbl_8048610 ; 4.1

; состояние стека сейчас: ST(0) = 4.1, ST(1) = результат деления

        fmul    [ebp+arg_8]

; состояние стека сейчас: ST(0) = результат умножения, ST(1) = результат деления

        pop     ebp
        faddp   st(1), st

; состояние стека сейчас: ST(0) = результат сложения

        retn
f        endp
```

Разница в том, что в стек сначала заталкивается 3,14 (в `ST(0)`), а затем значение из `arg_0` делится на то, что лежит в регистре `ST(0)`.

`FDIVR` означает *Reverse Divide* — делить, поменяв делитель и делимое местами. Точно такой же инструкции для умножения нет, потому что она была бы бессмысленна (ведь умножение операция коммутативная), так что остается только `FMUL` без соответствующей ей `-R` инструкции.

`FADDP` не только складывает два значения, но также и выталкивает из стека одно значение. После этого в `ST(0)` остается только результат сложения.

18.5.2. ARM: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Пока в ARM не было стандартного набора инструкций для работы с числами с плавающей точкой, разные производители процессоров могли добавлять свои расширения для работы с ними. Позже был принят стандарт VFP (*Vector Floating Point*).

Важное отличие от x86 в том, что там вы работаете с FPU-стеком, а здесь стека нет, вы работаете просто с регистрами.

Листинг 18.3: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```
f
        VLDR     D16, =3.14
        VMOV     D17, R0, R1 ; загрузить "a"
        VMOV     D18, R2, R3 ; загрузить "b"
        VDIV.F64 D16, D17, D16 ; a/3.14
        VLDR     D17, =4.1
        VMUL.F64 D17, D18, D17 ; b*4.1
        VADD.F64 D16, D17, D16 ; +
        VMOV     R0, R1, D16
        BX       LR

dbl_2C98    DCFD 3.14          ; DATA XREF: f
dbl_2CA0    DCFD 4.1          ; DATA XREF: f+10
```

Итак, здесь мы видим использование новых регистров с префиксом D.

Это 64-битные регистры. Их 32 и их можно использовать для чисел с плавающей точкой двойной точности (double) и для SIMD (в ARM это называется NEON).

18.5. ПРОСТОЙ ПРИМЕР

Имеются также 32 32-битных S-регистра. Они применяются для работы с числами с плавающей точкой одинарной точности (float).

Запомнить легко: D-регистры предназначены для чисел double-точности, а S-регистры — для чисел single-точности.

Больше об этом: [B.3.3](#) (стр. 928).

Обе константы (3,14 и 4,1) хранятся в памяти в формате IEEE 754.

Инструкции `VLDR` и `VMOV`, как можно догадаться, это аналоги обычных `LDR` и `MOV`, но они работают с D-регистрами.

Важно отметить, что эти инструкции, как и D-регистры, предназначены не только для работы с числами с плавающей точкой, но пригодны также и для работы с SIMD (NEON), и позже это также будет видно.

Аргументы передаются в функцию обычным путем через R-регистры, однако каждое число, имеющее двойную точность, занимает 64 бита, так что для передачи каждого нужны два R-регистра.

`VMOV D17, R0, R1` в самом начале составляет два 32-битных значения из `R0` и `R1` в одно 64-битное и сохраняет в `D17`.

`VMOV R0, R1, D16` в конце это обратная процедура: то что было в `D16` остается в двух регистрах `R0` и `R1`, потому что число с двойной точностью, занимающее 64 бита, возвращается в паре регистров `R0` и `R1`.

`VDIV`, `VMUL` и `VADD`, это инструкции для работы с числами с плавающей точкой, вычисляющие, соответственно, [частное](#), [произведение](#) и сумму.

Код для Thumb-2 такой же.

18.5.3. ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
f
    PUSH    {R3-R7,LR}
    MOVS    R7, R2
    MOVS    R4, R3
    MOVS    R5, R0
    MOVS    R6, R1
    LDR     R2, =0x66666666 ; 4.1
    LDR     R3, =0x40106666
    MOVS    R0, R7
    MOVS    R1, R4
    BL      __aeabi_dmul
    MOVS    R7, R0
    MOVS    R4, R1
    LDR     R2, =0x51EB851F ; 3.14
    LDR     R3, =0x40091EB8
    MOVS    R0, R5
    MOVS    R1, R6
    BL      __aeabi_ddiv
    MOVS    R2, R7
    MOVS    R3, R4
    BL      __aeabi_dadd
    POP     {R3-R7,PC}

; 4.1 в формате IEEE 754:
dword_364    DCD 0x66666666      ; DATA XREF: f+A
dword_368    DCD 0x40106666      ; DATA XREF: f+C
; 3.14 в формате IEEE 754:
dword_36C    DCD 0x51EB851F      ; DATA XREF: f+1A
dword_370    DCD 0x40091EB8      ; DATA XREF: f+1C
```

Keil компилировал для процессора, в котором может и не быть поддержки FPU или NEON. Так что числа с двойной точностью передаются в парах обычных R-регистров, а вместо FPU-инструкций вызываются сервисные библиотечные функции `__aeabi_dmul`, `__aeabi_ddiv`, `__aeabi_dadd`, эмулирующие умножение, деление и сложение чисел с плавающей точкой.

Конечно, это медленнее чем FPU-сопроцессор, но лучше, чем ничего.

Кстати, похожие библиотеки для эмуляции сопроцессорных инструкций были очень распространены в x86 когда сопроцессор был редким и дорогим и присутствовал далеко не во всех компьютерах.

18.5. ПРОСТОЙ ПРИМЕР

Эмуляция FPU-сопроцессора в ARM называется *soft float* или *armel (emulation)*, а использование FPU-инструкций сопроцессора – *hard float* или *armhf*.

18.5.4. ARM64: Оптимизирующий GCC (Linaro) 4.9

Очень компактный код:

Листинг 18.4: Оптимизирующий GCC (Linaro) 4.9

```
f:
; D0 = a, D1 = b
    ldr    d2, .LC25        ; 3.14
; D2 = 3.14
    fdiv   d0, d0, d2
; D0 = D0/D2 = a/3.14
    ldr    d2, .LC26        ; 4.1
; D2 = 4.1
    fmadd  d0, d1, d2, d0
; D0 = D1*D2+D0 = b*4.1+a/3.14
    ret

; константы в формате IEEE 754:
.LC25:
    .word  1374389535      ; 3.14
    .word  1074339512
.LC26:
    .word  1717986918      ; 4.1
    .word  1074816614
```

18.5.5. ARM64: Неоптимизирующий GCC (Linaro) 4.9

Листинг 18.5: Неоптимизирующий GCC (Linaro) 4.9

```
f:
    sub    sp, sp, #16
    str    d0, [sp,8]      ; сохранить "a" в Register Save Area
    str    d1, [sp]        ; сохранить "b" в Register Save Area
    ldr    x1, [sp,8]
; X1 = a
    ldr    x0, .LC25
; X0 = 3.14
    fmov   d0, x1
    fmov   d1, x0
; D0 = a, D1 = 3.14
    fdiv   d0, d0, d1
; D0 = D0/D1 = a/3.14

    fmov   x1, d0
; X1 = a/3.14
    ldr    x2, [sp]
; X2 = b
    ldr    x0, .LC26
; X0 = 4.1
    fmov   d0, x2
; D0 = b
    fmov   d1, x0
; D1 = 4.1
    fmul   d0, d0, d1
; D0 = D0*D1 = b*4.1

    fmov   x0, d0
; X0 = D0 = b*4.1
    fmov   d0, x1
; D0 = a/3.14
    fmov   d1, x0
; D1 = X0 = b*4.1
    fadd   d0, d0, d1
```

18.5. ПРОСТОЙ ПРИМЕР

```
; D0 = D0+D1 = a/3.14 + b*4.1

    fmov    x0, d0 ; \ избыточный код
    fmov    d0, x0 ; /
    add     sp, sp, 16
    ret

.LC25:
    .word   1374389535      ; 3.14
    .word   1074339512

.LC26:
    .word   1717986918      ; 4.1
    .word   1074816614
```

Неоптимизирующий GCC более многословный. Здесь много ненужных перетасовок значений, включая явно избыточный код (последние две инструкции `GMOV`). Должно быть, GCC 4.9 пока ещё не очень хорош для генерации кода под ARM64. Интересно заметить что у ARM64 64-битные регистры и D-регистры так же 64-битные. Так что компилятор может сохранять значения типа *double* в `GPR` вместо локального стека. Это было невозможно на 32-битных CPU. И снова, как упреждение, вы можете попробовать оптимизировать эту функцию вручную, без добавления новых инструкций вроде `FMADD`.

18.5.6. MIPS

MIPS может поддерживать несколько сопроцессоров (вплоть до 4), нулевой из которых это специальный управляющий сопроцессор, а первый — это FPU.

Как и в ARM, сопроцессор в MIPS это не стековая машина. Он имеет 32 32-битных регистра (\$F0-\$F31):

[C.1.2](#) (стр. 930). Когда нужно работать с 64-битными значениями типа *double*, используется пара 32-битных F-регистров.

Листинг 18.6: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
; $f12-$f13=A
; $f14-$f15=B
    lui     $v0, (dword_C4 >> 16) ; ?
; загрузить младшую 32-битную часть константы 3.14 в $f0:
    lwc1    $f0, dword_BC
    or      $at, $zero           ; load delay slot, NOP
; загрузить старшую 32-битную часть константы 3.14 в $f1:
    lwc1    $f1, $LC0
    lui     $v0, ($LC1 >> 16)    ; ?
; A в $f12-$f13, константа 3.14 в $f0-$f1, произвести деление:
    div.d   $f0, $f12, $f0
; $f0-$f1=A/3.14
; загрузить младшую 32-битную часть константы 4.1 в $f2:
    lwc1    $f2, dword_C4
    or      $at, $zero           ; load delay slot, NOP
; загрузить старшую 32-битную часть константы 4.1 в $f3:
    lwc1    $f3, $LC1
    or      $at, $zero           ; load delay slot, NOP
; B в $f14-$f15, константа 4.1 в $f2-$f3, произвести умножение:
    mul.d   $f2, $f14, $f2
; $f2-$f3=B*4.1
    jr      $ra
; суммировать 64-битные части и оставить результат в $f0-$f1:
    add.d   $f0, $f2             ; branch delay slot, NOP

.rodata.cst8:000000B8 $LC0:      .word 0x40091EB8      # DATA XREF: f+C
.rodata.cst8:000000BC dword_BC: .word 0x51EB851F      # DATA XREF: f+4
.rodata.cst8:000000C0 $LC1:      .word 0x40106666      # DATA XREF: f+10
.rodata.cst8:000000C4 dword_C4: .word 0x66666666      # DATA XREF: f
```

Новые инструкции:

- `LWC1` загружает 32-битное слово в регистр первого сопроцессора (отсюда «1» в названии инструкции).

Пара инструкций `LWC1` может быть объединена в одну псевдоинструкцию `L.D`.

18.6. ПЕРЕДАЧА ЧИСЕЛ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ В АРГУМЕНТАХ

- `DIV.D`, `MUL.D`, `ADD.D` производят деление, умножение и сложение соответственно («.D» в суффиксе означает двойную точность, «.S» — одинарную точность)

Здесь также имеется странная аномалия компилятора: инструкция `LUI` помеченная нами вопросительным знаком.

Мне трудно понять, зачем загружать часть 64-битной константы типа *double* в регистр `$V0`.

От этих инструкций нет толка. Если кто-то об этом что-то знает, пожалуйста, напишите автору емейл [12](#).

18.6. Передача чисел с плавающей запятой в аргументах

```
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

    return 0;
}
```

18.6.1. x86

Посмотрим, что у нас вышло (MSVC 2010):

Листинг 18.7: MSVC 2010

```
CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54
CONST    ENDS

_main    PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 8 ; выделить место для первой переменной
    fld     QWORD PTR __real@3ff8a3d70a3d70a4
    fstp    QWORD PTR [esp]
    sub     esp, 8 ; выделить место для второй переменной
    fld     QWORD PTR __real@40400147ae147ae1
    fstp    QWORD PTR [esp]
    call    _pow
    add     esp, 8 ; "вернуть" место от одной переменной.

; в локальном стеке сейчас все еще зарезервировано 8 байт для нас.
; результат сейчас в ST(0)

    fstp    QWORD PTR [esp] ; перегрузить результат из ST(0) в локальный стек для printf()
    push    OFFSET $SG2651
    call    _printf
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0
_main    ENDP
```

`FLD` и `FSTP` перемещают переменные из сегмента данных в FPU-стек или обратно. `pow()` [13](#) достает оба значения из FPU-стека и возвращает результат в `ST(0)`. `printf()` берет 8 байт из стека и трактует их как переменную типа *double*.

Кстати, с тем же успехом можно было бы перекладывать эти два числа из памяти в стек при помощи пары `MOV`: ведь в памяти числа в формате IEEE 754, `pow()` также принимает их в том же формате, и никакая конверсия не требуется. Собственно, так и происходит в следующем примере с ARM: [18.6.2](#) (стр. [227](#)).

¹²dennis(a)yurichev.com

¹³стандартная функция Си, возводящая число в степень

18.6.2. ARM + Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

```

_main
var_C          = -0xC

        PUSH        {R7,LR}
        MOV         R7, SP
        SUB         SP, SP, #4
        VLDR        D16, =32.01
        VMOV        R0, R1, D16
        VLDR        D16, =1.54
        VMOV        R2, R3, D16
        BLX         _pow
        VMOV        D16, R0, R1
        MOV         R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
        ADD         R0, PC
        VMOV        R1, R2, D16
        BLX         _printf
        MOVS        R1, 0
        STR         R0, [SP,#0xC+var_C]
        MOV         R0, R1
        ADD         SP, SP, #4
        POP         {R7,PC}

dbl_2F90      DCFD 32.01          ; DATA XREF: _main+6
dbl_2F98      DCFD 1.54          ; DATA XREF: _main+E

```

Как уже было указано, 64-битные числа с плавающей точкой передаются в парах R-регистров.

Этот код слегка избыточен (наверное, потому что не включена оптимизация), ведь можно было бы загружать значения напрямую в R-регистры минуя загрузку в D-регистры.

Итак, видно, что функция `_pow` получает первый аргумент в `R0` и `R1`, а второй в `R2` и `R3`. Функция оставляет результат в `R0` и `R1`. Результат работы `_pow` перекадывается в `D16`, затем в пару `R1` и `R2`, откуда `printf()` берет это число-результат.

18.6.3. ARM + Неоптимизирующий Keil 6/2013 (Режим ARM)

```

_main
        STMFD       SP!, {R4-R6,LR}
        LDR         R2, =0xA3D70A4 ; y
        LDR         R3, =0x3FF8A3D7
        LDR         R0, =0xAE147AE1 ; x
        LDR         R1, =0x40400147
        BL          pow
        MOV         R4, R0
        MOV         R2, R4
        MOV         R3, R1
        ADR         R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"
        BL          __2printf
        MOV         R0, #0
        LDMFD       SP!, {R4-R6,PC}

y        DCD 0xA3D70A4          ; DATA XREF: _main+4
dword_520 DCD 0x3FF8A3D7          ; DATA XREF: _main+8
x        DCD 0xAE147AE1          ; DATA XREF: _main+C
dword_528 DCD 0x40400147          ; DATA XREF: _main+10
a32_011_54Lf DCB "32.01 ^ 1.54 = %lf",0xA,0
                                         ; DATA XREF: _main+24

```

Здесь не используются D-регистры, используются только пары R-регистров.

18.6.4. ARM64 + Оптимизирующий GCC (Linaro) 4.9

```
f:
    stp    x29, x30, [sp, -16]!
    add    x29, sp, 0
    ldr    d1, .LC1 ; загрузить 1.54 в D1
    ldr    d0, .LC0 ; загрузить 32.01 в D0
    bl     pow
; результат pow() в D0
    adrp   x0, .LC2
    add    x0, x0, :lo12:.LC2
    bl     printf
    mov    w0, 0
    ldp    x29, x30, [sp], 16
    ret

.LC0:
; 32.01 в формате IEEE 754
    .word  -1374389535
    .word  1077936455

.LC1:
; 1.54 в формате IEEE 754
    .word  171798692
    .word  1073259479

.LC2:
    .string "32.01 ^ 1.54 = %lf\n"
```

Константы загружаются в **D0** и **D1**: функция **pow()** берет их оттуда. Результат в **D0** после исполнения **pow()**. Он пропускается в **printf()** без всякой модификации и перемещений, потому что **printf()** берет аргументы **интегральных типов** и указатели из X-регистров, а аргументы типа плавающей точки из D-регистров.

18.6.5. MIPS

```
main:

var_10      = -0x10
var_4       = -4

; пролог функции:
    lui    $gp, (dword_9C >> 16)
    addiu  $sp, -0x20
    la     $gp, (__gnu_local_gp & 0xFFFF)
    sw     $ra, 0x20+var_4($sp)
    sw     $gp, 0x20+var_10($sp)
    lui    $v0, (dword_A4 >> 16) ; ?
; загрузить младшую 32-битную часть числа 32.01:
    lwc1   $f12, dword_9C
; загрузить адрес функции pow():
    lw     $t9, (pow & 0xFFFF)($gp)
; загрузить страшную 32-битную часть числа 32.01:
    lwc1   $f13, $LC0
    lui    $v0, ($LC1 >> 16) ; ?
; загрузить младшую 32-битную часть числа 1.54:
    lwc1   $f14, dword_A4
    or     $at, $zero ; load delay slot, NOP
; загрузить страшную 32-битную часть числа 1.54:
    lwc1   $f15, $LC1
; вызвать pow():
    jalr   $t9
    or     $at, $zero ; branch delay slot, NOP
    lw     $gp, 0x20+var_10($sp)
; скопировать результат из $f0 и $f1 в $a3 и $a2:
    mfc1   $a3, $f0
    lw     $t9, (printf & 0xFFFF)($gp)
    mfc1   $a2, $f1
; вызвать printf():
    lui    $a0, ($LC2 >> 16) # "32.01 ^ 1.54 = %lf\n"
```

18.7. ПРИМЕР С СРАВНЕНИЕМ

```
jalr    $t9
la      $a0, ($LC2 & 0xFFFF) # "32.01 ^ 1.54 = %lf\n"
; эпилог функции:
lw      $ra, 0x20+var_4($sp)
; возврат 0:
move    $v0, $zero
jr      $ra
addiu   $sp, 0x20

.rodata.str1.4:00000084 $LC2:          .ascii "32.01 ^ 1.54 = %lf\n"<0>

; 32.01:
.rodata.cst8:00000098 $LC0:          .word 0x40400147      # DATA XREF: main+20
.rodata.cst8:0000009C dword_9C:      .word 0xAE147AE1      # DATA XREF: main
.rodata.cst8:0000009C                                # main+18
; 1.54:
.rodata.cst8:000000A0 $LC1:          .word 0x3FF8A3D7      # DATA XREF: main+24
.rodata.cst8:000000A0                                # main+30
.rodata.cst8:000000A4 dword_A4:      .word 0xA3D70A4       # DATA XREF: main+14
```

И снова мы здесь видим, как `LUI` загружает 32-битную часть числа типа *double* в `$V0`. И снова трудно понять почему.

Новая для нас инструкция это `MFC1` («Move From Coprocessor 1») (копировать из первого сопроцессора). FPU это сопроцессор под номером 1, вот откуда «1» в имени инструкции. Эта инструкция переносит значения из регистров сопроцессора в регистры основного CPU (`GPR`). Так что результат исполнения `pow()` в итоге копируется в регистры `$A3` и `$A2` и из этой пары регистров `printf()` берет его как 64-битное значение типа *double*.

18.7. Пример с сравнением

Попробуем теперь вот это:

```
#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;

    return b;
};

int main()
{
    printf ("%f\n", d_max (1.2, 3.4));
    printf ("%f\n", d_max (5.6, -4));
};
```

Несмотря на кажущуюся простоту этой функции, понять, как она работает, будет чуть сложнее.

18.7.1. x86

Неоптимизирующий MSVC

Вот что выдал MSVC 2010:

Листинг 18.10: Неоптимизирующий MSVC 2010

```
PUBLIC      _d_max
_TEXT SEGMENT
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_d_max PROC
    push    ebp
    mov     ebp, esp
    fld     QWORD PTR _b$[ebp]
```

```

; состояние стека сейчас: ST(0) = _b
; сравниваем _b (в ST(0)) и _a, затем выталкиваем значение из стека

    fcomp  QWORD PTR _a$[ebp]

; стек теперь пустой

    fnstsw ax
    test   ah, 5
    jp     SHORT $LN1@d_max

; мы здесь только если a>b

    fld     QWORD PTR _a$[ebp]
    jmp     SHORT $LN2@d_max
$LN1@d_max:
    fld     QWORD PTR _b$[ebp]
$LN2@d_max:
    pop     ebp
    ret     0
_d_max     ENDP

```

Итак, **FLD** загружает **_b** в регистр **ST(0)**.

FCOMP сравнивает содержимое **ST(0)** с тем, что лежит в **_a** и выставляет биты **C3 / C2 / C0** в регистре статуса FPU. Это 16-битный регистр отражающий текущее состояние FPU.

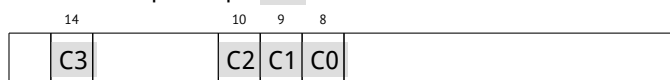
После этого инструкция **FCOMP** также выдергивает одно значение из стека. Это отличает её от **FCOM**, которая просто сравнивает значения, оставляя стек в таком же состоянии.

К сожалению, у процессоров до Intel P6 ¹⁴ нет инструкций условного перехода, проверяющих биты **C3 / C2 / C0**. Возможно, так сложилось исторически (вспомните о том, что FPU когда-то был вообще отдельным чипом). А у Intel P6 появились инструкции **FCOMI / FCOMIP / FUCOMI / FUCOMIP**, делающие то же самое, только напрямую модифицирующие флаги **ZF / PF / CF**.

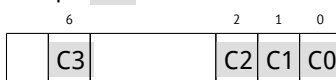
Так что **FNSTSW** копирует содержимое регистра статуса в **AX**. Биты **C3 / C2 / C0** занимают позиции, соответственно, 14, 10, 8. В этих позициях они и остаются в регистре **AX**, и все они расположены в старшей части регистра — **АН**.

- Если $b > a$ в нашем случае, то биты **C3 / C2 / C0** должны быть выставлены так: 0, 0, 0.
- Если $a > b$, то биты будут выставлены: 0, 0, 1.
- Если $a = b$, то биты будут выставлены так: 1, 0, 0.
- Если результат не определен (в случае ошибки), то биты будут выставлены так: 1, 1, 1.

Вот как биты **C3 / C2 / C0** расположены в регистре **AX**:



Вот как биты **C3 / C2 / C0** расположены в регистре **АН**:



После исполнения **test ah, 5** ¹⁵ будут учтены только биты **C0** и **C2** (на позициях 0 и 2), остальные просто проигнорированы.

Теперь немного о *parity flag* ¹⁶. Ещё один замечательный рудимент эпохи.

Этот флаг выставляется в 1 если количество единиц в последнем результате чётно. И в 0 если нечётно.

Заглянем в Wikipedia ¹⁷:

¹⁴Intel P6 это Pentium Pro, Pentium II, и последующие модели

¹⁵5=101b

¹⁶флаг четности

¹⁷wikipedia.org/wiki/Parity_flag

One common reason to test the parity flag actually has nothing to do with parity. The FPU has four condition flags (C0 to C3), but they can not be tested directly, and must instead be first copied to the flags register. When this happens, C0 is placed in the carry flag, C2 in the parity flag and C3 in the zero flag. The C2 flag is set when e.g. incomparable floating point values (NaN or unsupported format) are compared with the FUCOM instructions.

Как упоминается в Wikipedia, флаг четности иногда используется в FPU-коде и сейчас мы увидим как.

Флаг `PF` будет выставлен в 1, если `C0` и `C2` оба 1 или оба 0. И тогда сработает последующий `JP` (*jump if PF==1*). Если мы вернемся чуть назад и посмотрим значения `C3 / C2 / C0` для разных вариантов, то увидим, что условный переход `JP` сработает в двух случаях: если $b > a$ или если $a = b$ (ведь бит `C3` перестал учитываться после исполнения `test ah, 5`).

Дальше всё просто. Если условный переход сработал, то `FLD` загрузит значение `_b` в `ST(0)`, а если не сработал, то загрузится `_a` и произойдет выход из функции.

А как же проверка флага `C2` ?

Флаг `C2` включается в случае ошибки (NaN, и т.д.), но наш код его не проверяет.

Если программисту нужно знать, не произошла ли FPU-ошибка, он должен позаботиться об этом дополнительно, добавив соответствующие проверки.

Первый пример с OllyDbg: $a=1,2$ и $b=3,4$

Загружаем пример в OllyDbg:

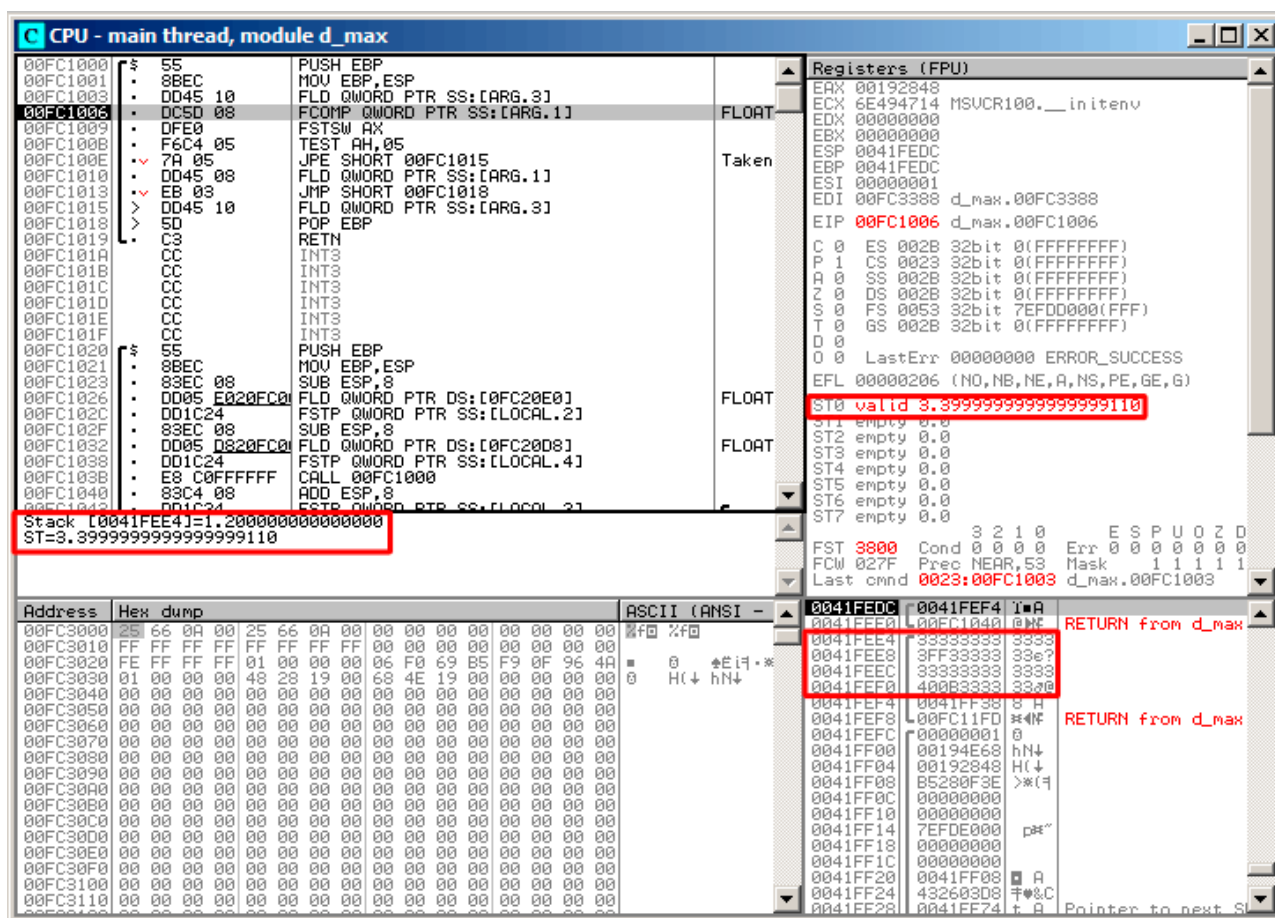


Рис. 18.6: OllyDbg: первая `FLD` исполнилась

Текущие параметры функции: $a = 1, 2$ и $b = 3, 4$ (их видно в стеке: 2 пары 32-битных значений). b (3,4) уже загружено в `ST(0)`. Сейчас будет исполняться `FCOMP`. OllyDbg показывает второй аргумент для `FCOMP`, который сейчас находится в стеке.

FCOMP отработал:

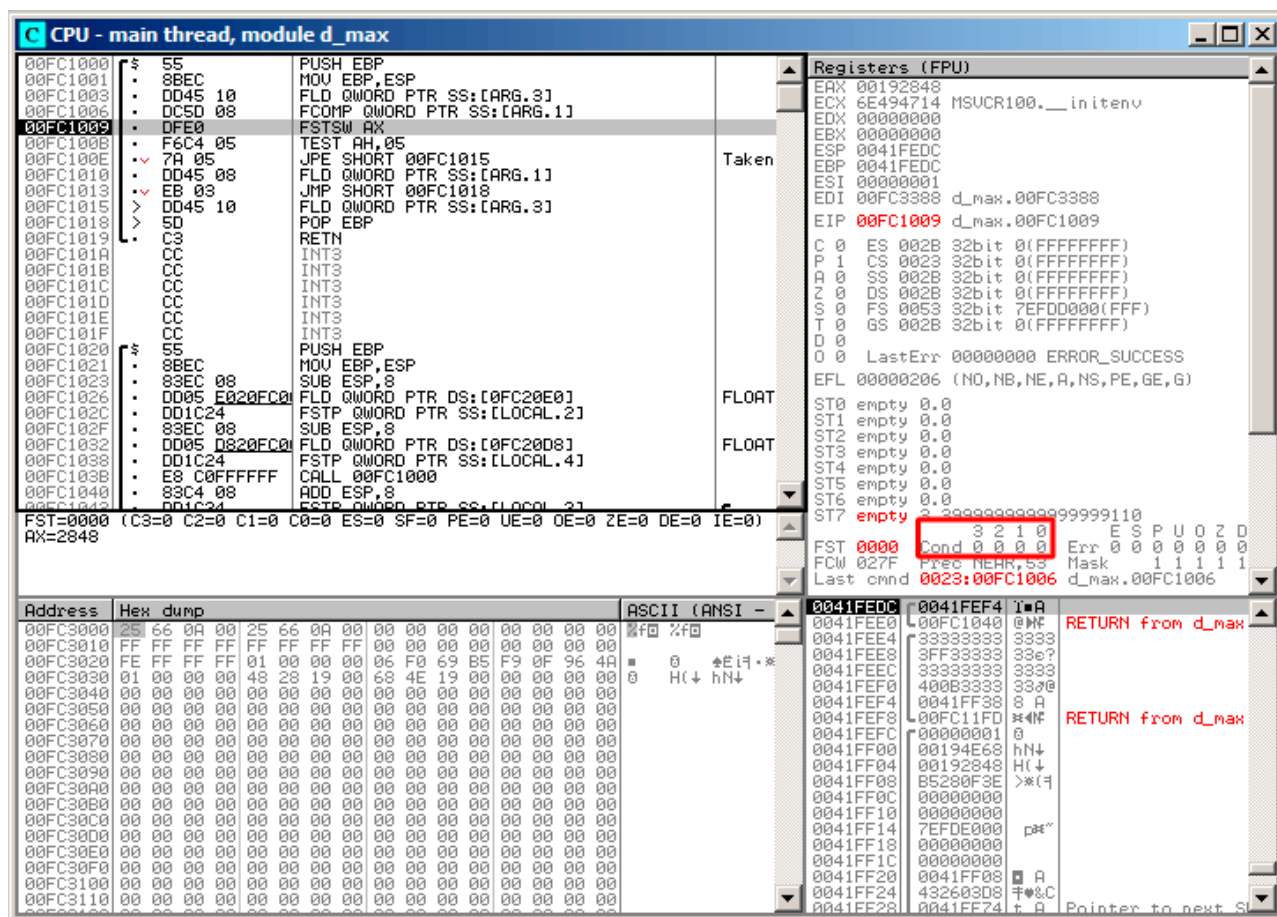


Рис. 18.7: OllyDbg: FCOMP исполнилась

Мы видим состояния condition-флагов FPU: все нули. Вытолкнутое значение отображается как ST(7). Почему это так, объяснялось ранее: 18.5.1 (стр. 221).

FNSTSW сработал:

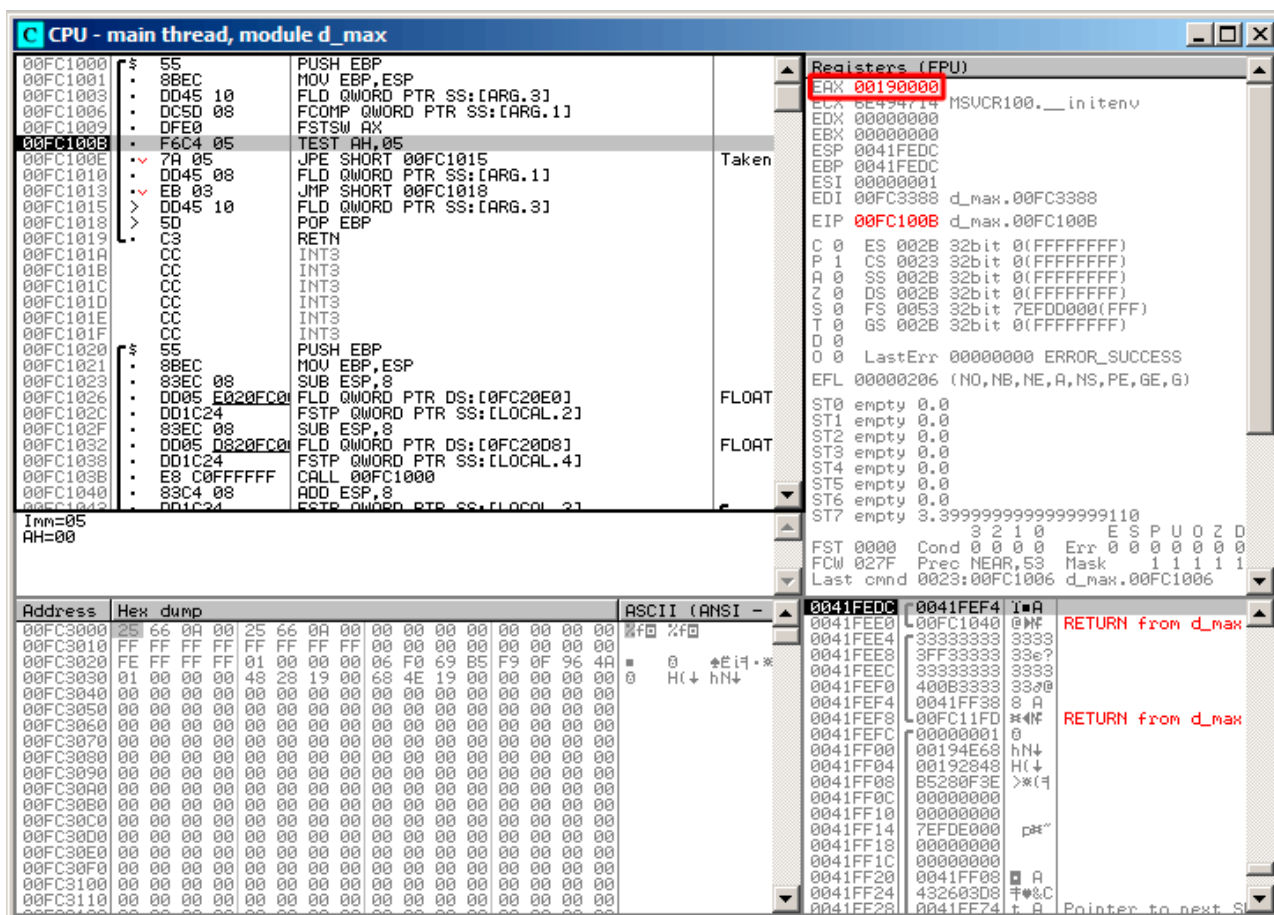


Рис. 18.8: OllyDbg: FNSTSW исполнилась

Видно, что регистр `AX` содержит нули. Действительно, ведь все condition-флаги тоже содержали нули.

(OllyDbg дизассемблирует команду `FNSTSW` как `FSTSW` — это синоним).

TEST сработал:

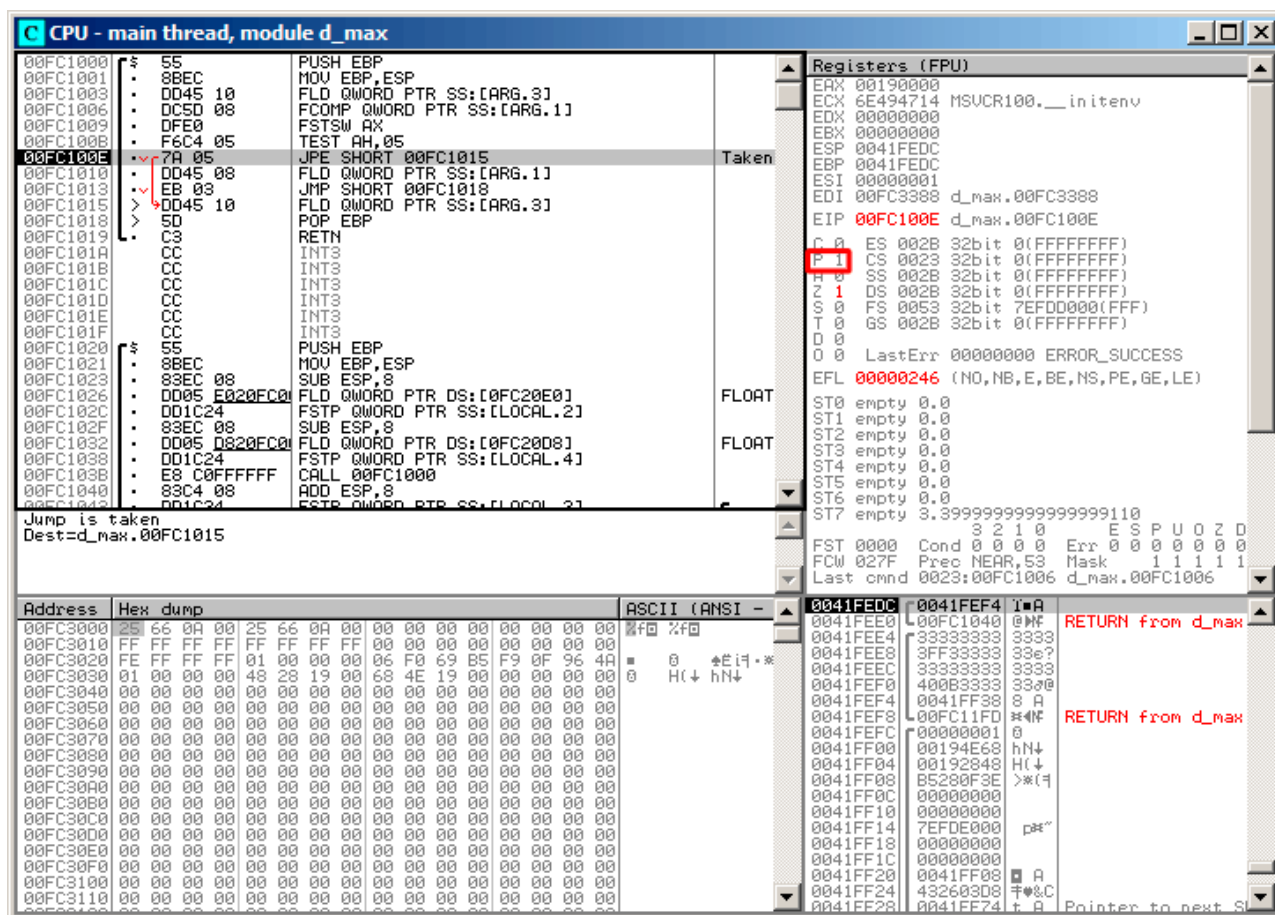


Рис. 18.9: OllyDbg: TEST исполнилась

Флаг PF равен единице. Всё верно: количество выставленных бит в 0 — это 0, а 0 — это четное число.

OllyDbg дизассемблирует JP как JPE¹⁸ — это синонимы. И она сейчас работает.

¹⁸Jump Parity Even (инструкция x86)

18.7. ПРИМЕР С СРАВНЕНИЕМ

JPE сработала, FLD загрузила в ST(0) значение b(3,4):

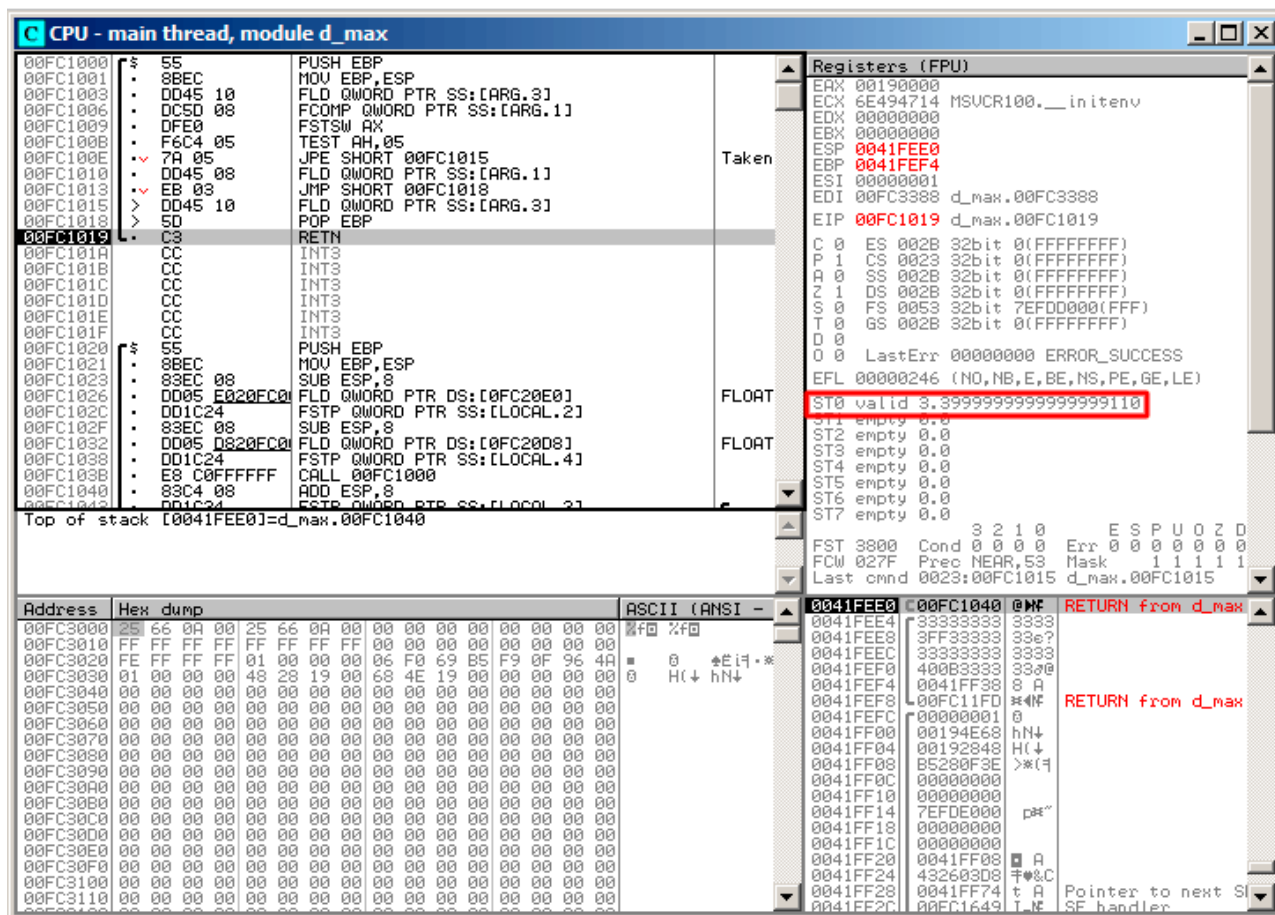


Рис. 18.10: OllyDbg: вторая FLD исполнилась

Функция заканчивает свою работу.

Второй пример с OllyDbg: $a=5,6$ и $b=-1$

Загружаем пример в OllyDbg:

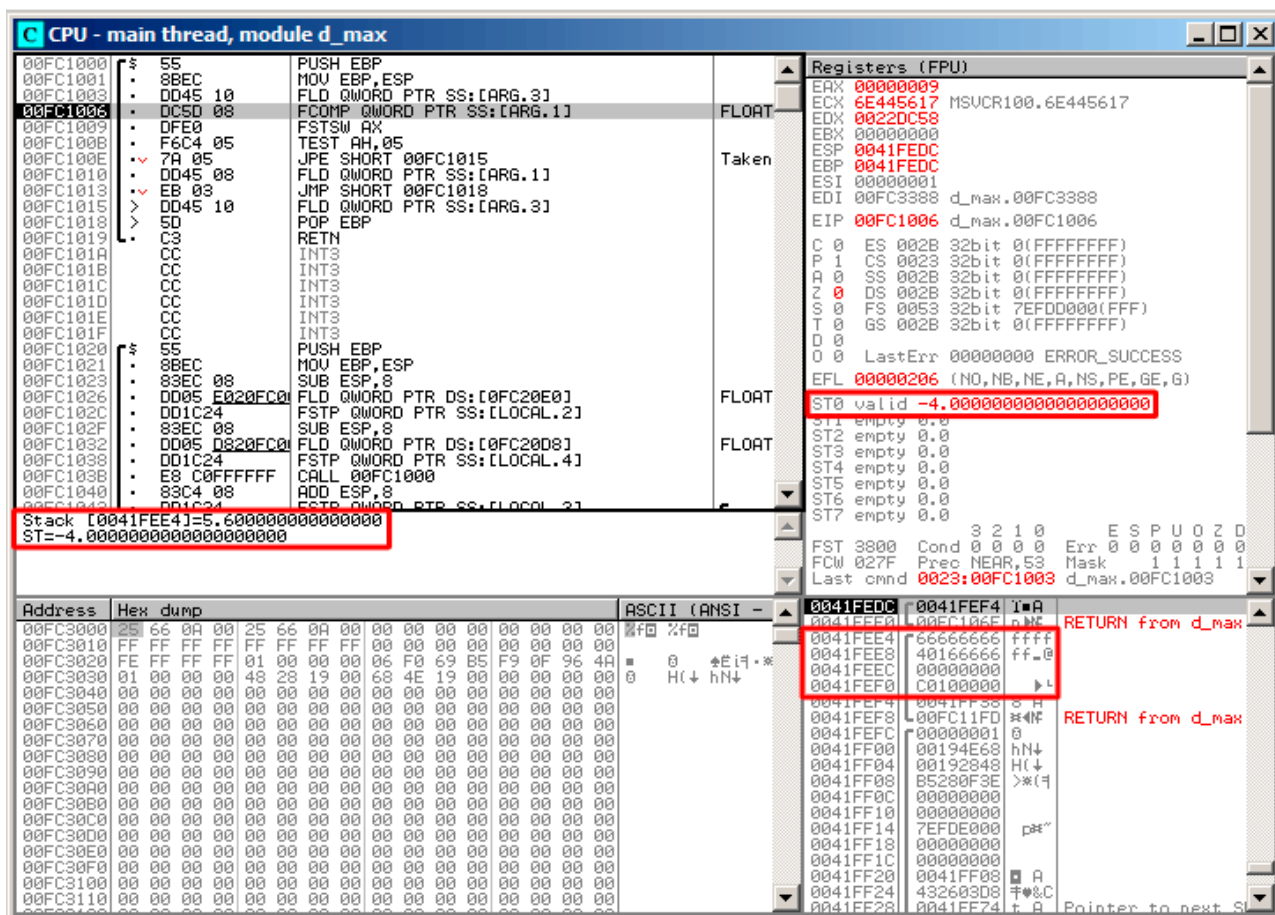


Рис. 18.11: OllyDbg: первая `FLD` исполнилась

Текущие параметры функции: $a = 5,6$ и $b = -4$. $b (-4)$ уже загружено в `ST(0)`. Сейчас будет исполняться `FCOMP`. OllyDbg показывает второй аргумент `FCOMP`, который сейчас находится в стеке.

FCOMP отработал:

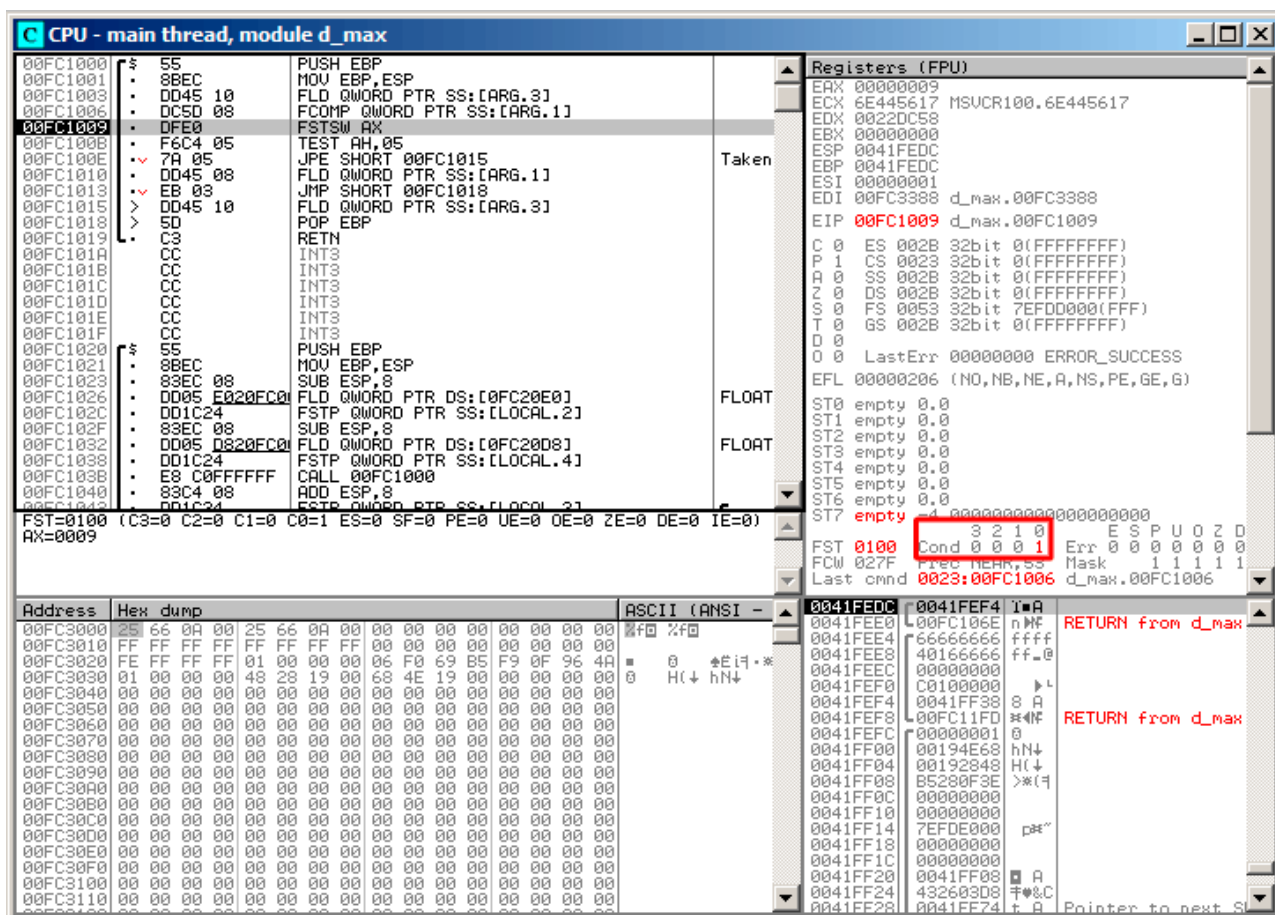


Рис. 18.12: OllyDbg: FCOMP исполнилась

Мы видим значения condition-флагов FPU: все нули, кроме C0.

FNSTSW сработал:

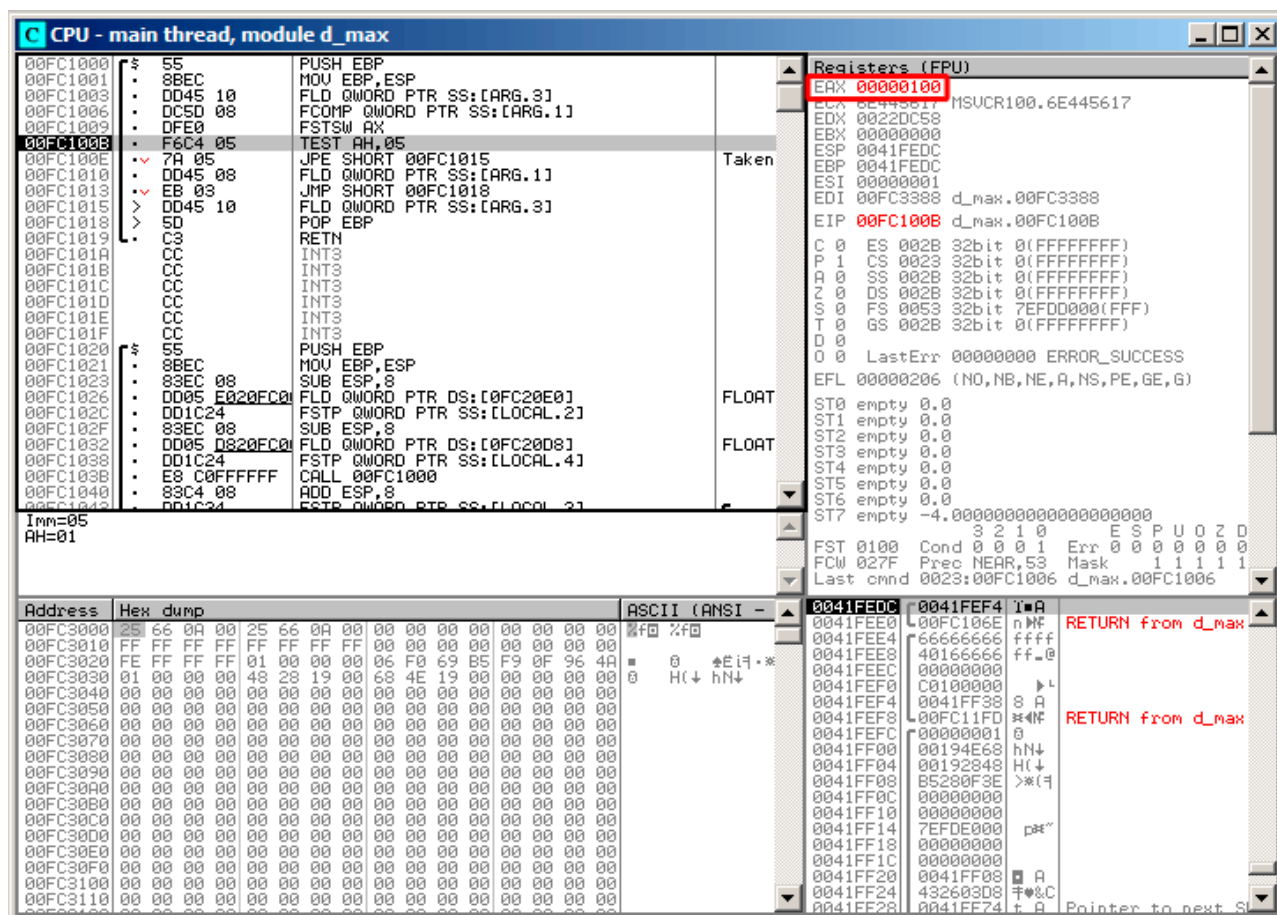


Рис. 18.13: OllyDbg: FNSTSW исполнилась

Видно, что регистр `AX` содержит `0x100`: флаг `C0` стал на место 16-го бита.

TEST сработал:

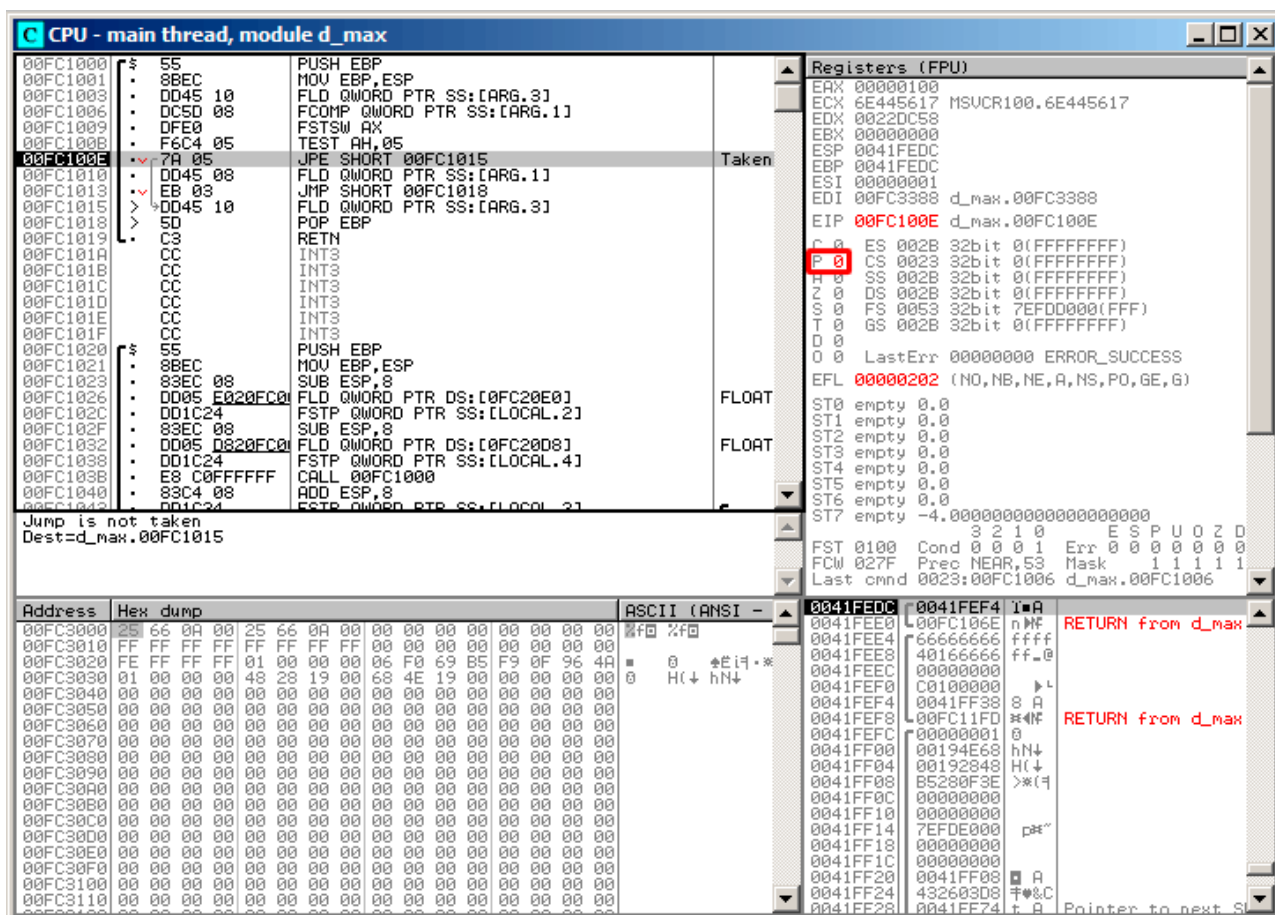


Рис. 18.14: OllyDbg: TEST исполнилась

Флаг PF равен нулю. Всё верно: количество единичных бит в 0x100 — 1, а 1 — нечетное число.

JPE сейчас не работает.

18.7. ПРИМЕР С СРАВНЕНИЕМ

JPE не сработала, FLD загрузила в ST(0) значение a (5,6):

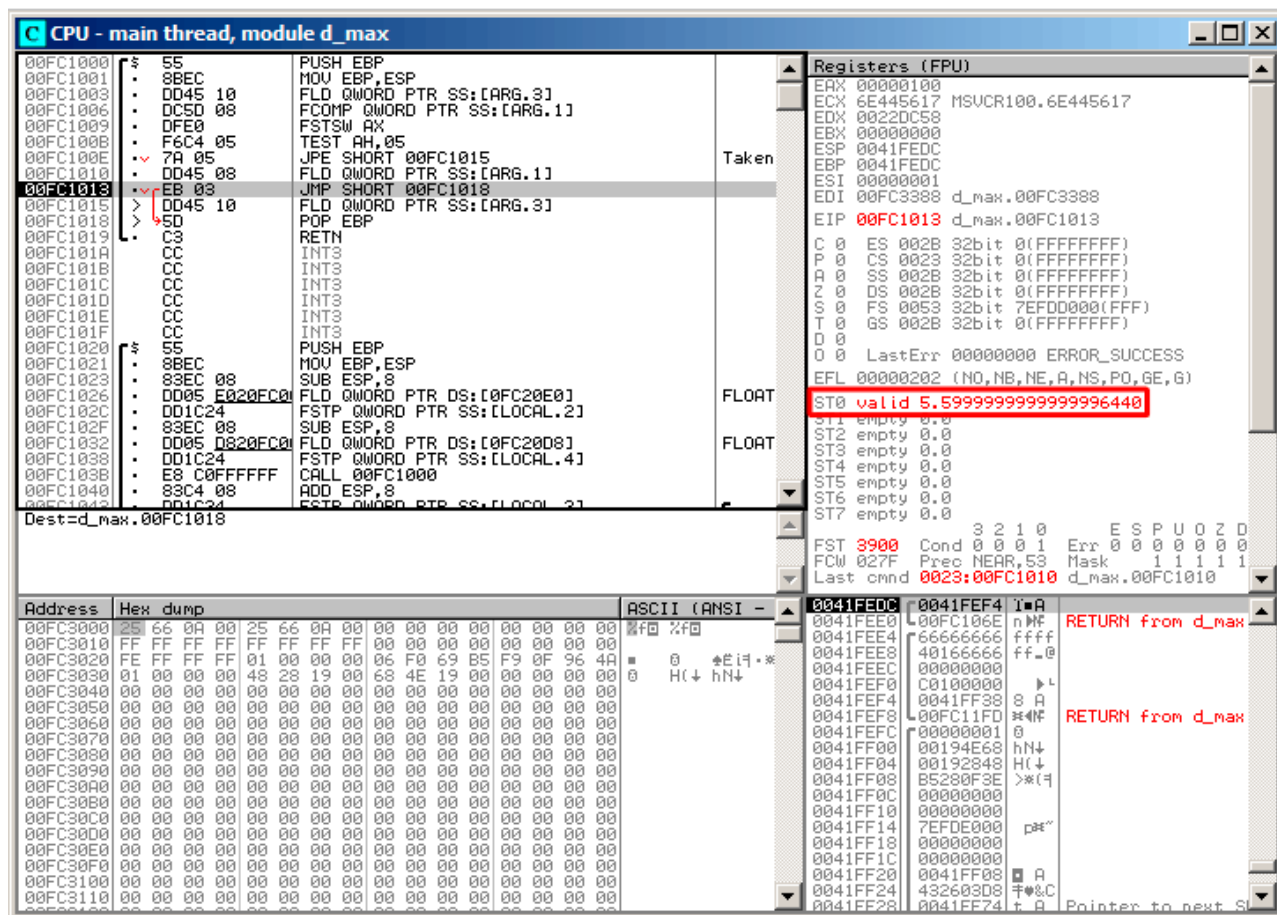


Рис. 18.15: OllyDbg: вторая FLD исполнилась

Функция заканчивает свою работу.

Оптимизирующий MSVC 2010

Листинг 18.11: Оптимизирующий MSVC 2010

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_d_max PROC
    fld QWORD PTR _b$[esp-4]
    fld QWORD PTR _a$[esp-4]

; состояние стека сейчас: ST(0) = _a, ST(1) = _b

    fcom ST(1) ; сравнить _a и ST(1) = (_b)
    fnstsw ax
    test ah, 65 ; 00000041H
    jne SHORT $LN5@d_max
; копировать содержимое ST(0) в ST(1) и вытолкнуть значение из стека,
; оставив _a на вершине
    fstp ST(1)

; состояние стека сейчас: ST(0) = _a

    ret 0
$LN5@d_max:
; копировать содержимое ST(0) в ST(0) и вытолкнуть значение из стека,
; оставив _b на вершине
    fstp ST(0)

; состояние стека сейчас: ST(0) = _b
```

```

    ret     0
_d_max    ENDP

```

FCOM отличается от **FCOMP** тем, что просто сравнивает значения и оставляет стек в том же состоянии. В отличие от предыдущего примера, операнды здесь в обратном порядке. Поэтому и результат сравнения в **C3 / C2 / C0** будет отличаться:

- Если $a > b$, то биты **C3 / C2 / C0** должны быть выставлены так: 0, 0, 0.
- Если $b > a$, то биты будут выставлены так: 0, 0, 1.
- Если $a = b$, то биты будут выставлены так: 1, 0, 0.

Инструкция **test ah, 65** как бы оставляет только два бита — **C3** и **C0**. Они оба будут нулями, если $a > b$; в таком случае переход **JNE** не сработает. Далее имеется инструкция **FSTP ST(1)** — эта инструкция копирует значение **ST(0)** в указанный операнд и выдергивает одно значение из стека. В данном случае, она копирует **ST(0)** (где сейчас лежит **_a**) в **ST(1)**. После этого на вершине стека два раза лежит **_a**. Затем одно значение выдергивается. После этого в **ST(0)** остается **_a** и функция завершается.

Условный переход **JNE** сработает в двух других случаях: если $b > a$ или $a = b$. **ST(0)** скопируется в **ST(0)** (как бы холостая операция). Затем одно значение из стека вылетит и на вершине стека останется то, что до этого лежало в **ST(1)** (то есть **_b**). И функция завершится. Эта инструкция используется здесь видимо потому что в FPU нет другой инструкции, которая просто выдергивает значение из стека и выбрасывает его.

Обе FLD отработали:

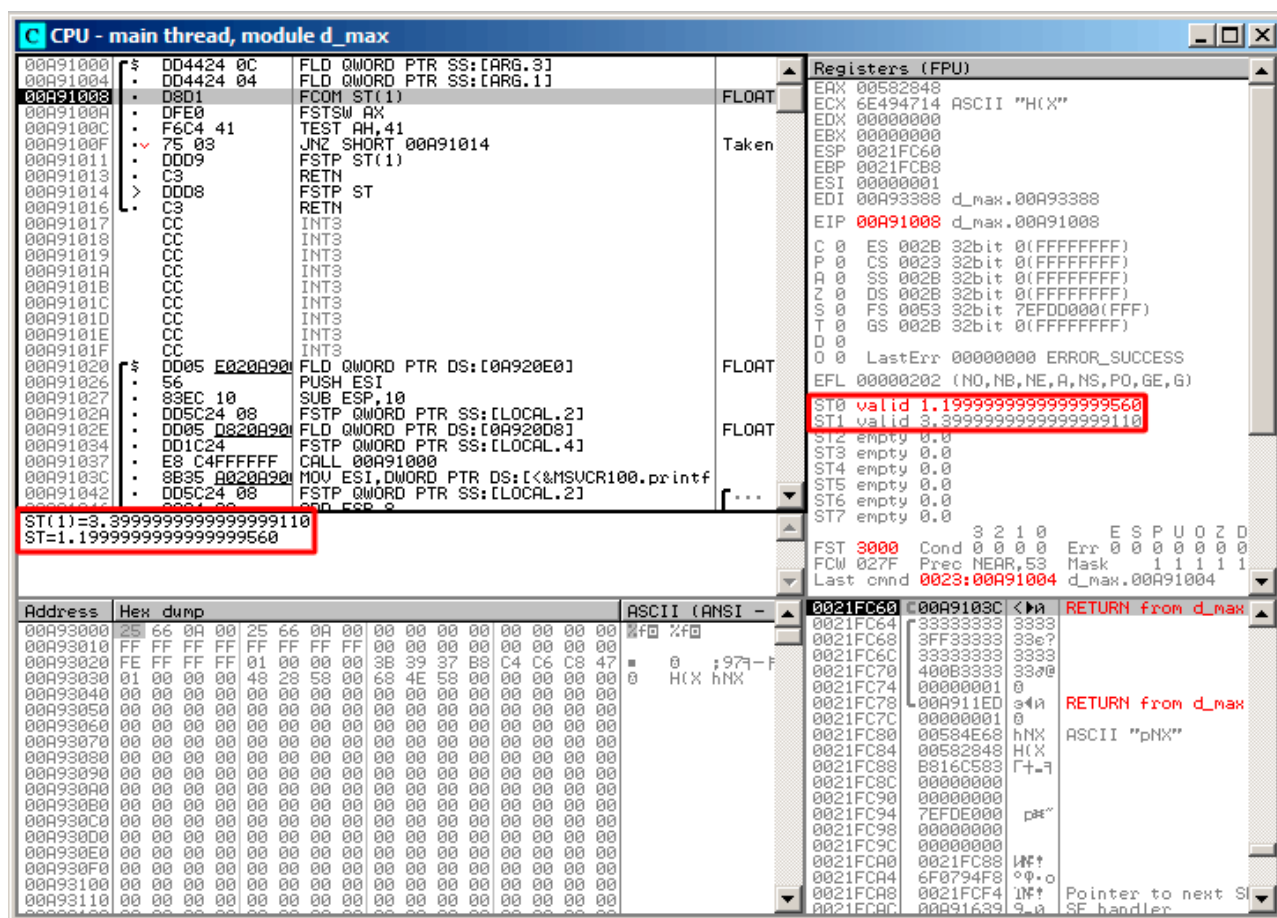


Рис. 18.16: OllyDbg: обе FLD исполнились

Сейчас будет исполняться FCOM : OllyDbg показывает содержимое ST(0) и ST(1) для удобства.

18.7. ПРИМЕР С СРАВНЕНИЕМ

FCOM сработала:

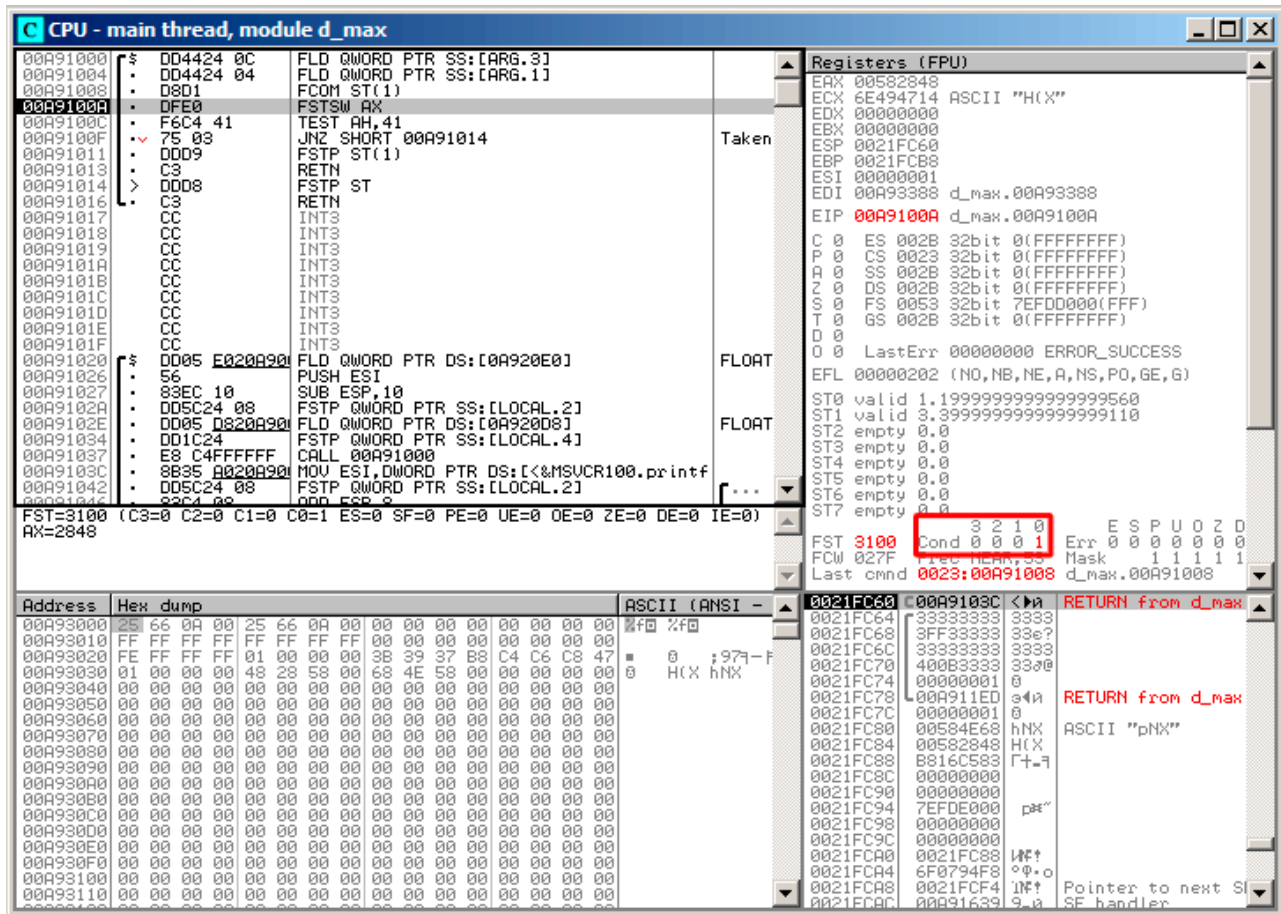


Рис. 18.17: OllyDbg: FCOM исполнилась

C0 установлен, остальные флаги сброшены.

18.7. ПРИМЕР С СРАВНЕНИЕМ

FNSTSW сработала, AX =0x3100:

The screenshot shows the OllyDbg interface with the following details:

- CPU - main thread, module d_max**: The assembly window shows instructions at addresses 00A91000 to 00A9104F. The instruction at 00A91000 is `F6C4 41 TEST AH,41`, which is highlighted. The instruction at 00A91001 is `75 03 JNZ SHORT 00A91014`, which is marked as "Taken".
- Registers (FPU)**: The EAX register is highlighted and contains the value `00583100`, which is the ASCII string "Administrator".
- Hex dump**: The memory dump shows the contents of the memory starting at address 00A91000. The ASCII column shows the string "Administrator" starting at offset 00A91000.

Рис. 18.18: OllyDbg: FNSTSW исполнилась

18.7. ПРИМЕР С СРАВНЕНИЕМ

TEST сработала:

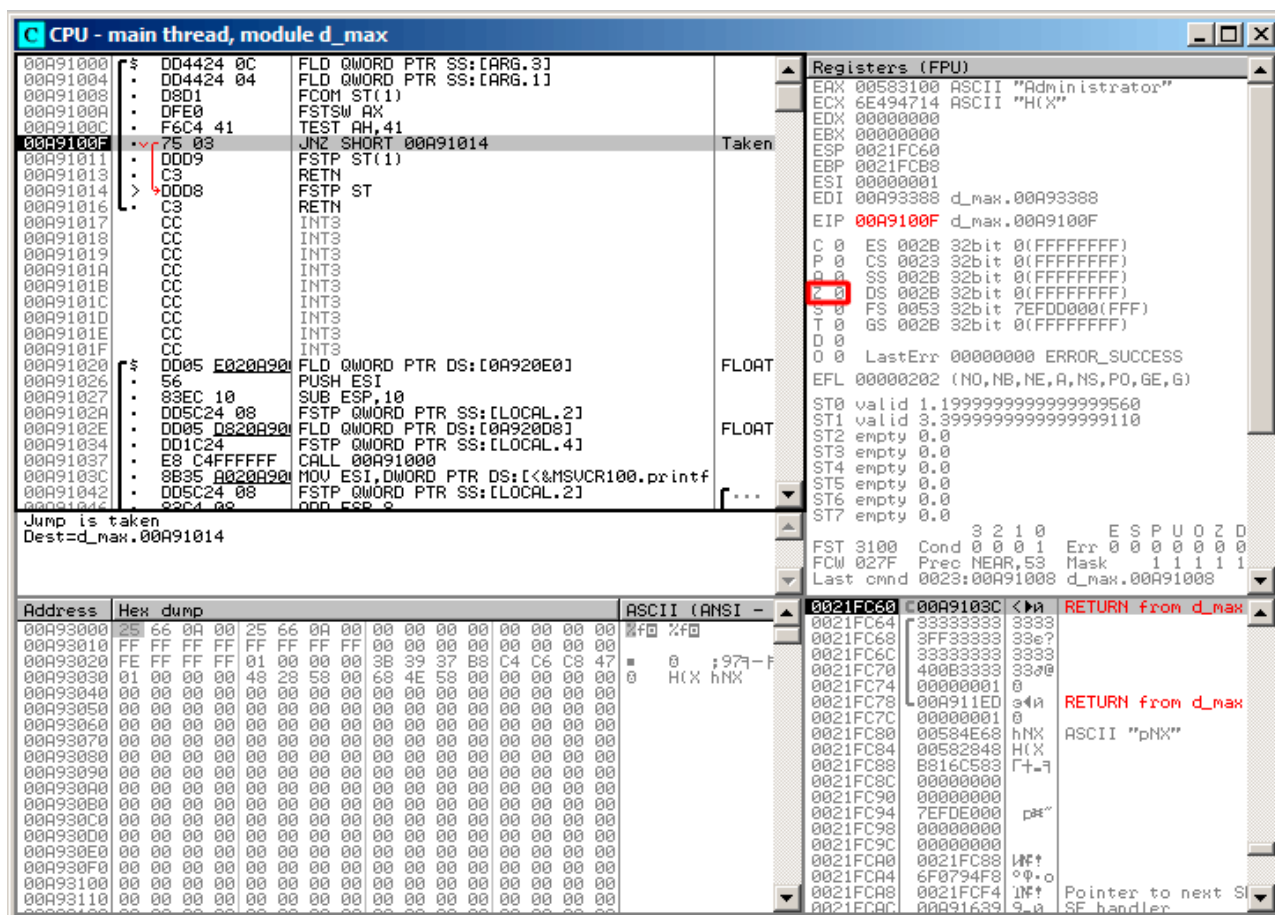


Рис. 18.19: OllyDbg: TEST исполнилась

ZF=0, переход сейчас произойдет.

18.7. ПРИМЕР С СРАВНЕНИЕМ

FSTP ST (или FSTP ST(0)) сработала – 1,2 было вытолкнуто из стека, и на вершине осталось 3,4:

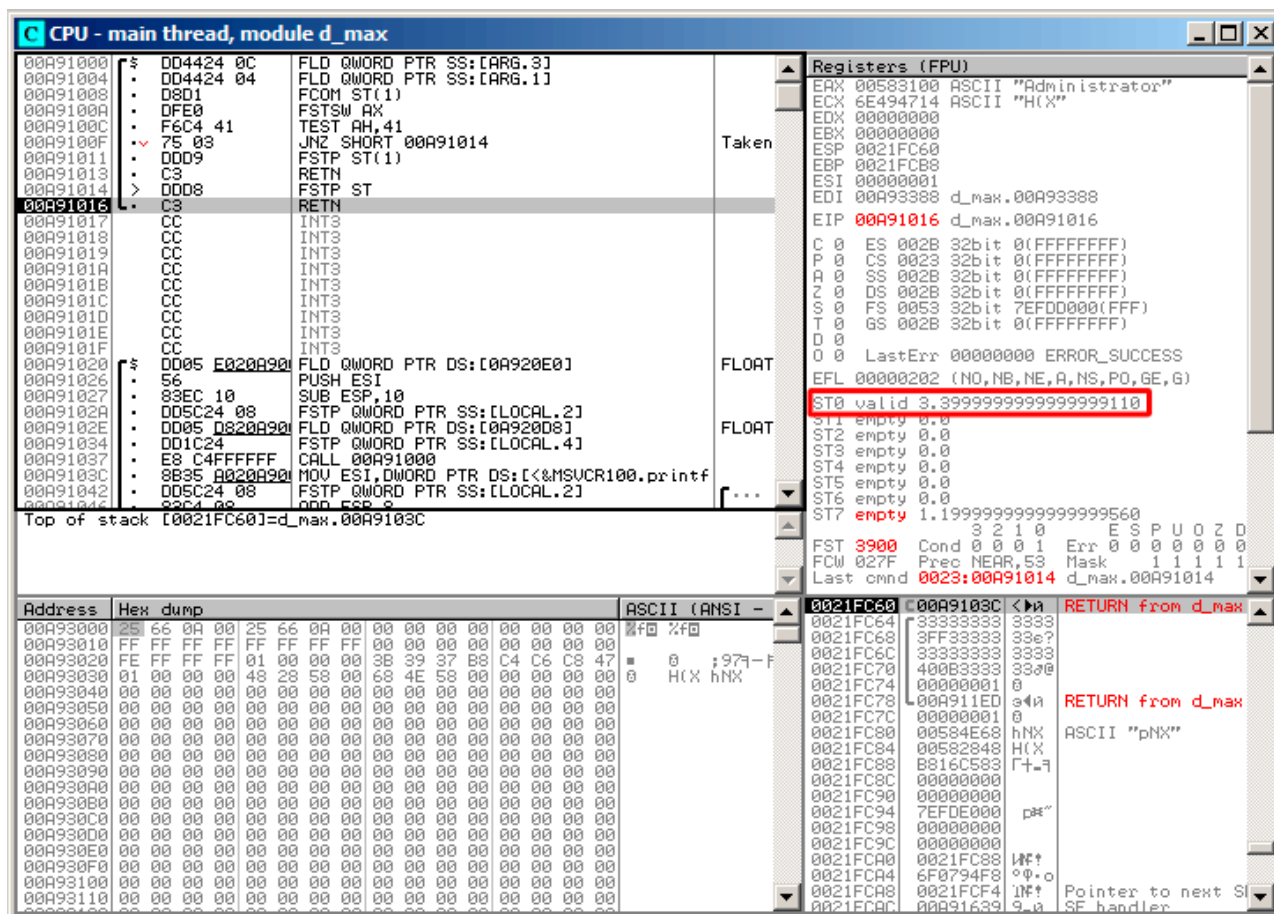


Рис. 18.20: OllyDbg: FSTP исполнилась

Видно, что инструкция FSTP ST работает просто как выталкивание одного значения из FPU-стека.

Второй пример с OllyDbg: a=5,6 и b=-4

Обе FLD отработали:

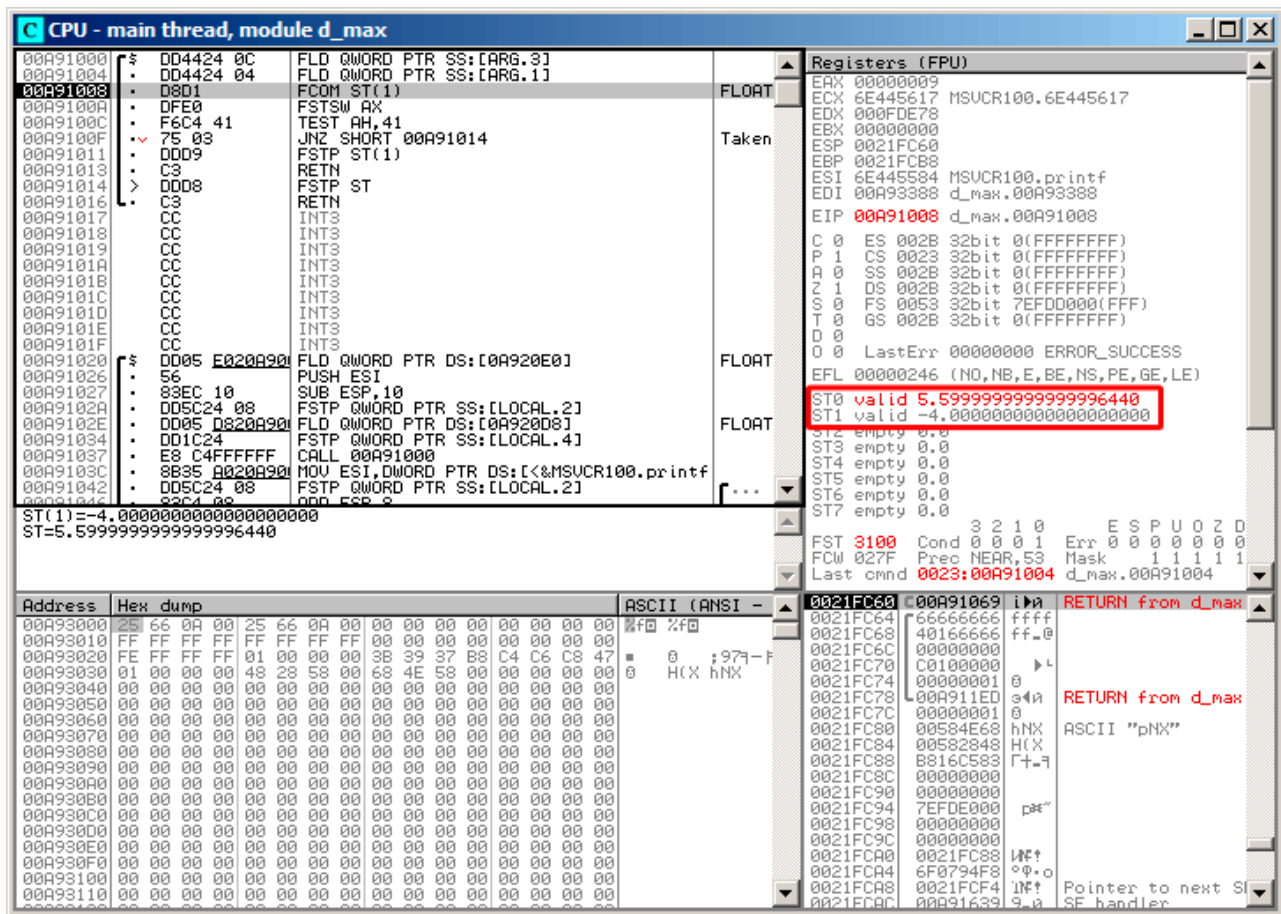


Рис. 18.21: OllyDbg: обе FLD исполнились

Сейчас будет исполняться FCOM.

18.7. ПРИМЕР С СРАВНЕНИЕМ

FCOM сработала:

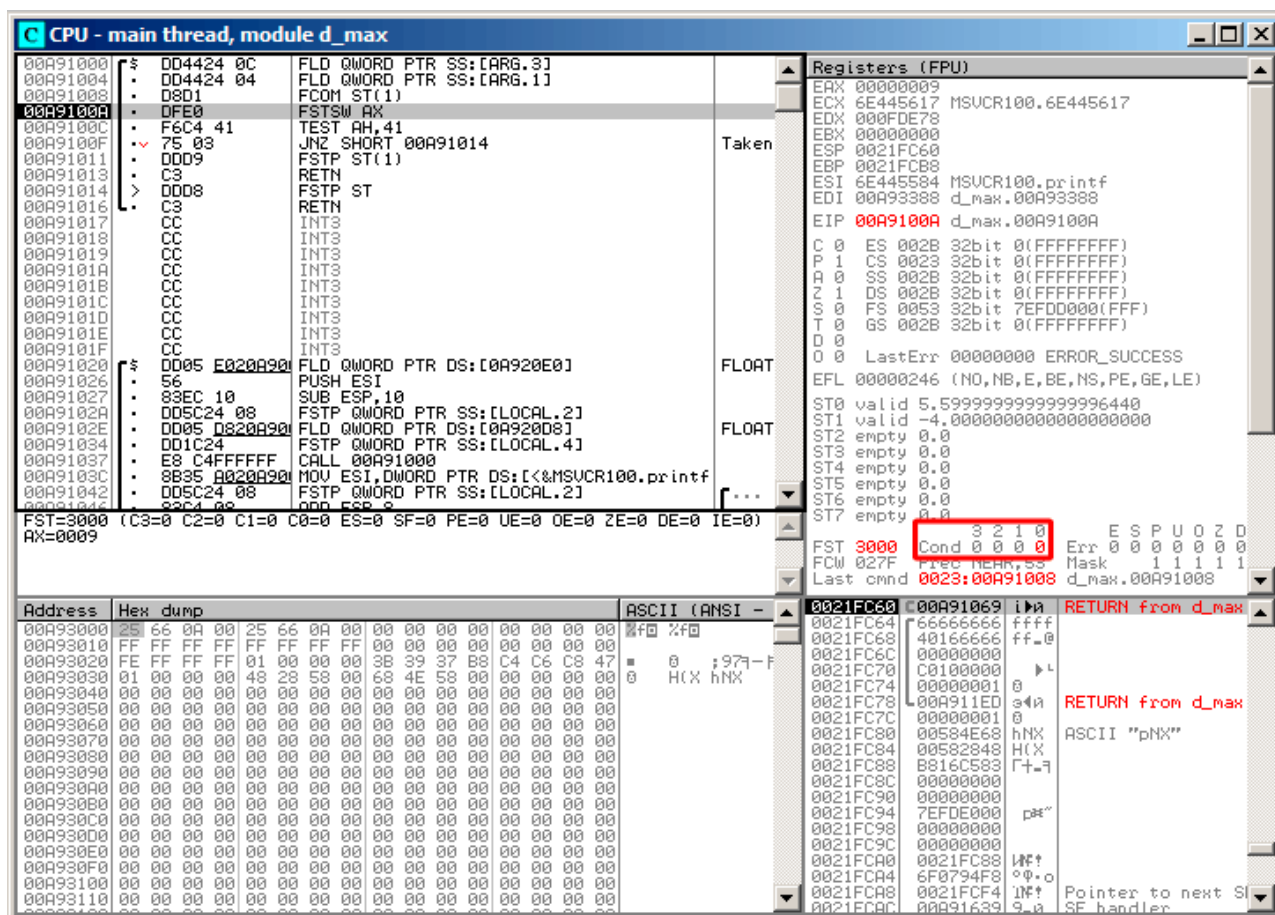


Рис. 18.22: OllyDbg: FCOM исполнилась

Все condition-флаги сброшены.

FNSTSW сработала, AX =0x3000:



18.7. ПРИМЕР С СРАВНЕНИЕМ

TEST сработала:

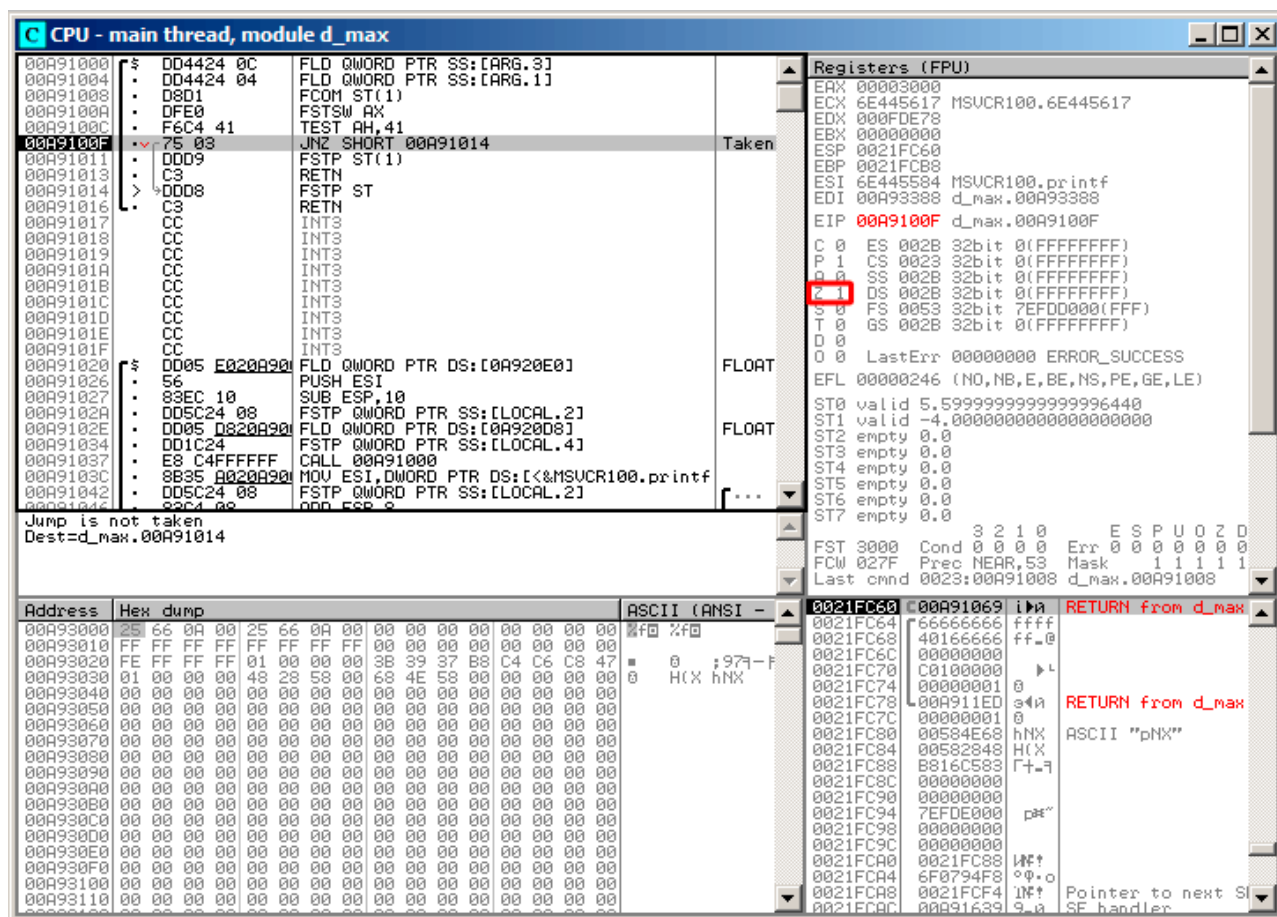


Рис. 18.24: OllyDbg: TEST исполнилась

ZF=1, переход сейчас не произойдет.

FSTP ST(1) сработала: на вершине FPU-стека осталось значение 5,6.

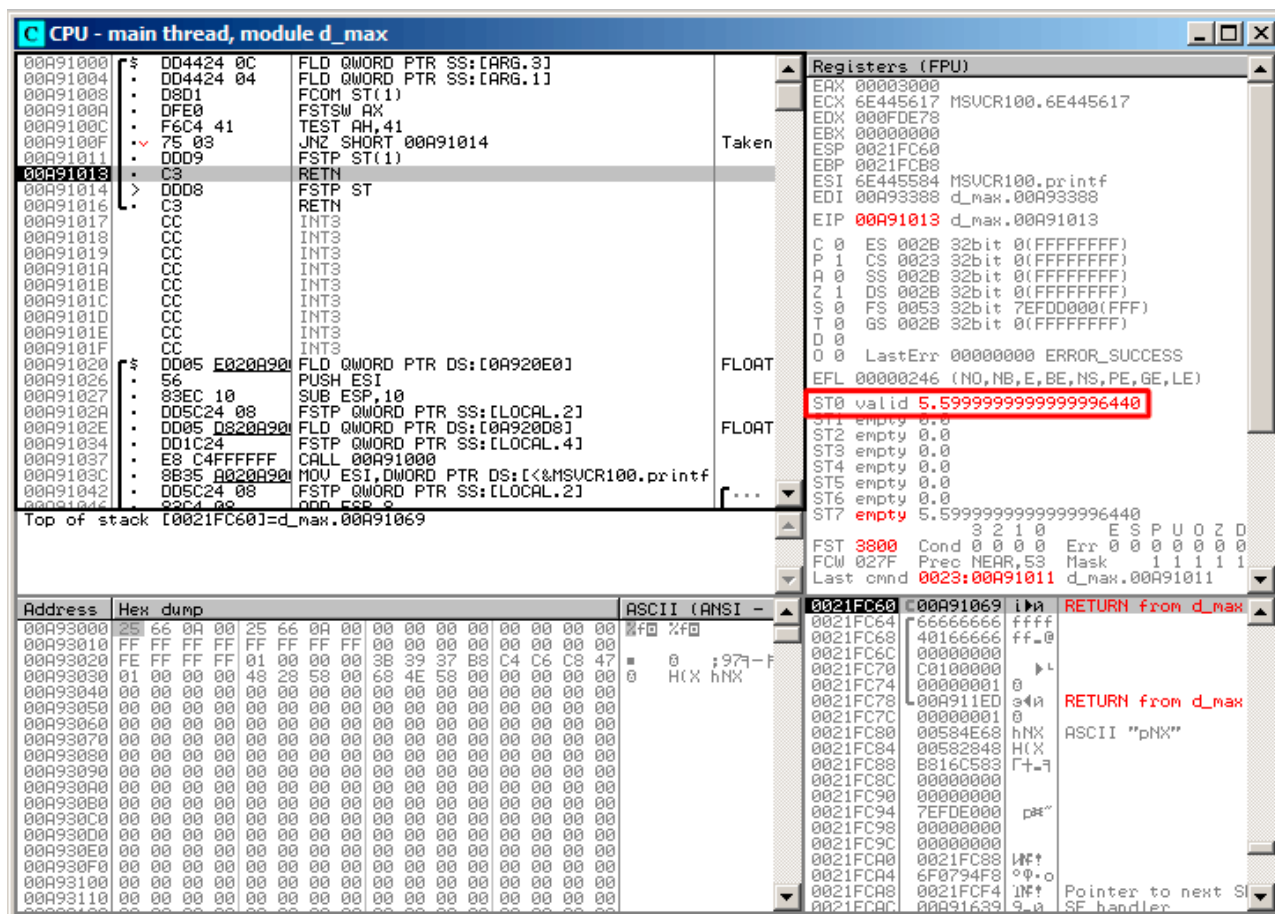


Рис. 18.25: OllyDbg: FSTP исполнилась

Видно, что инструкция FSTP ST(1) работает так: оставляет значение на вершине стека, но обнуляет регистр ST(1).

GCC 4.4.1

Листинг 18.12: GCC 4.4.1

```
d_max proc near
b      = qword ptr -10h
a      = qword ptr -8
a_first_half = dword ptr 8
a_second_half = dword ptr 0Ch
b_first_half = dword ptr 10h
b_second_half = dword ptr 14h

push    ebp
mov     ebp, esp
sub     esp, 10h

; переложим а и b в локальный стек:

mov     eax, [ebp+a_first_half]
mov     dword ptr [ebp+a], eax
mov     eax, [ebp+a_second_half]
mov     dword ptr [ebp+a+4], eax
mov     eax, [ebp+b_first_half]
mov     dword ptr [ebp+b], eax
mov     eax, [ebp+b_second_half]
mov     dword ptr [ebp+b+4], eax

; загружаем а и b в стек FPU:
```

```

fld    [ebp+a]
fld    [ebp+b]

; текущее состояние стека: ST(0) – b; ST(1) – a

fych   st(1) ; эта инструкция меняет ST(1) и ST(0) местами

; текущее состояние стека: ST(0) – a; ST(1) – b

fucmprr ; сравнить a и b и выдернуть из стека два значения, т.е. a и b
fnstsw ax ; записать статус FPU в AX
sahf     ; загрузить состояние флагов SF, ZF, AF, PF, и CF из AH
setnbe al ; записать 1 в AL, если CF=0 и ZF=0
test    al, al ; AL==0 ?
jz      short loc_8048453 ; да
fld     [ebp+a]
jmp     short locret_8048456

loc_8048453:
fld     [ebp+b]

locret_8048456:
leave
retn
d_max endp

```

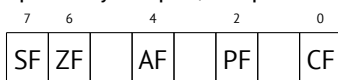
FUCOMPP — это почти то же что и **FCOM**, только выкидывает из стека оба значения после сравнения, а также несколько иначе реагирует на «не-числа».

Немного о *не-числах*.

FPU умеет работать со специальными переменными, которые числами не являются и называются «не числа» или NaN¹⁹. Это бесконечность, результат деления на ноль, и так далее. Нечисла бывают «тихие» и «сигнализирующие». С первыми можно продолжать работать и далее, а вот если вы попытаетесь совершить какую-то операцию с сигнализирующим нечислом, то сработает исключение.

Так вот, **FCOM** вызовет исключение если любой из операндов какое-либо нечисло. **FUCOM** же вызовет исключение только если один из операндов именно «сигнализирующее нечисло».

Далее мы видим **SAHF** (*Store AH into Flags*) — это довольно редкая инструкция в коде, не использующим FPU. 8 бит из **AH** перекладываются в младшие 8 бит регистра статуса процессора в таком порядке:



Вспомним, что **FNSTSW** перегружает интересующие нас биты **C3 / C2 / C0** в **AH**, и соответственно они будут в позициях 6, 2, 0 в регистре **AH**:



Иными словами, пара инструкций **fnstsw ax / sahf** перекладывает биты **C3 / C2 / C0** в флаги **ZF**, **PF**, **CF**.

Теперь снова вспомним, какие значения бит **C3 / C2 / C0** будут при каких результатах сравнения:

- Если a больше b в нашем случае, то биты **C3 / C2 / C0** должны быть выставлены так: 0, 0, 0.
- Если a меньше b , то биты будут выставлены так: 0, 0, 1.
- Если $a = b$, то так: 1, 0, 0.

Иными словами, после трех инструкций **FUCOMPP / FNSTSW / SAHF** возможны такие состояния флагов:

- Если $a > b$ в нашем случае, то флаги будут выставлены так: **ZF=0**, **PF=0**, **CF=0**.
- Если $a < b$, то флаги будут выставлены так: **ZF=0**, **PF=0**, **CF=1**.
- Если $a = b$, то так: **ZF=1**, **PF=0**, **CF=0**.

¹⁹ru.wikipedia.org/wiki/NaN

18.7. ПРИМЕР С СРАВНЕНИЕМ

Инструкция `SETNBE` выставит в `AL` единицу или ноль в зависимости от флагов и условий. Это почти аналог `JNBE`, за тем лишь исключением, что `SETcc`²⁰ выставляет 1 или 0 в `AL`, а `Jcc` делает переход или нет. `SETNBE` запишет 1 только если `CF=0` и `ZF=0`. Если это не так, то запишет 0 в `AL`.

`CF` будет 0 и `ZF` будет 0 одновременно только в одном случае: если $a > b$.

Тогда в `AL` будет записана 1, последующий условный переход `JZ` выполнен не будет и функция вернет `_a`. В остальных случаях, функция вернет `_b`.

Оптимизирующий GCC 4.4.1

Листинг 18.13: Оптимизирующий GCC 4.4.1

```
d_max      public d_max
proc near

arg_0      = qword ptr 8
arg_8      = qword ptr 10h

        push    ebp
        mov     ebp, esp
        fld     [ebp+arg_0] ; _a
        fld     [ebp+arg_8] ; _b

; состояние стека сейчас: ST(0) = _b, ST(1) = _a
        fxch    st(1)

; состояние стека сейчас: ST(0) = _a, ST(1) = _b
        fucom   st(1) ; сравнить _a и _b
        fnstsw  ax
        sahf
        ja      short loc_8048448

; записать ST(0) в ST(0) (холостая операция), выкинуть значение лежащее на вершине стека,
; оставить _b на вершине стека
        fstp    st
        jmp     short loc_804844A

loc_8048448:
; записать _a в ST(0), выкинуть значение лежащее на вершине стека, оставить _a на вершине стека
        fstp    st(1)

loc_804844A:
        pop     ebp
        retn
d_max      endp
```

Почти всё что здесь есть, уже описано мною, кроме одного: использование `JA` после `SAHF`. Действительно, инструкции условных переходов «больше», «меньше» и «равно» для сравнения беззнаковых чисел (а это `JA`, `JAE`, `JB`, `JBE`, `JE` / `JZ`, `JNA`, `JNAE`, `JNB`, `JNBE`, `JNE` / `JNZ`) проверяют только флаги `CF` и `ZF`.

Вспомним, как биты `C3` / `C2` / `C0` располагаются в регистре `AH` после исполнения `FSTSW` / `FNSTSW`:



Вспомним также, как располагаются биты из `AH` во флагах CPU после исполнения `SAHF`:



Биты `C3` и `C0` после сравнения переключаются в флаги `ZF` и `CF` так, что перечисленные инструкции переходов могут работать. `JA` сработает, если `CF` и `ZF` обнулены.

Таким образом, перечисленные инструкции условного перехода можно использовать после инструкций `FNSTSW` / `SAHF`.

²⁰cc это *condition code*

18.7. ПРИМЕР С СРАВНЕНИЕМ

Может быть, биты статуса FPU C3 / C2 / C0 преднамеренно были размещены таким образом, чтобы переноситься на базовые флаги процессора без перестановок?

GCC 4.8.1 с оптимизацией -O3

В линейке процессоров P6 от Intel появились новые FPU-инструкции²¹. Это `FUCOMI` (сравнить операнды и выставить флаги основного CPU) и `FCMOVcc` (работает как `CMOVcc`, но на регистрах FPU). Очевидно, разработчики GCC решили отказаться от поддержки процессоров до линейки P6 (ранние Pentium, 80486, etc.).

И кстати, FPU уже давно не отдельная часть процессора в линейке P6, так что флаги основного CPU можно модифицировать из FPU.

Вот что имеем:

Листинг 18.14: Оптимизирующий GCC 4.8.1

```
fld    QWORD PTR [esp+4]    ; загрузить "a"
fld    QWORD PTR [esp+12]   ; загрузить "b"
; ST0=b, ST1=a
fxch   st(1)
; ST0=a, ST1=b
; сравнить "a" и "b"
fucomi st, st(1)
; скопировать ST1 (там "b") в ST0 если a<=b
; в противном случае, оставить "a" в ST0
fcmovbe st, st(1)
; выбросить значение из ST1
fstp   st(1)
ret
```

Не совсем понимаю, зачем здесь `FXCH` (поменять местами операнды).

От нее легко избавиться поменяв местами инструкции `FLD` либо заменив `FCMOVBE` (*below or equal* — меньше или равно) на `FCMOVA` (*above* — больше).

Должно быть, неаккуратность компилятора.

Так что `FUCOMI` сравнивает `ST(0)` (*a*) и `ST(1)` (*b*) и затем устанавливает флаги основного CPU. `FCMOVBE` проверяет флаги и копирует `ST(1)` (в тот момент там находится *b*) в `ST(0)` (там *a*) если $ST0(a) \leq ST1(b)$. В противном случае ($a > b$), она оставляет *a* в `ST(0)`.

Последняя `FSTP` оставляет содержимое `ST(0)` на вершине стека, выбрасывая содержимое `ST(1)`.

Попробуем оттрассировать функцию в GDB:

Листинг 18.15: Оптимизирующий GCC 4.8.1 and GDB

```
1 dennis@ubuntuvm:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
2 dennis@ubuntuvm:~/polygon$ gdb d_max
3 GNU gdb (GDB) 7.6.1-ubuntu
4 Copyright (C) 2013 Free Software Foundation, Inc.
5 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
6 This is free software: you are free to change and redistribute it.
7 There is NO WARRANTY, to the extent permitted by law. Type "show copying"
8 and "show warranty" for details.
9 This GDB was configured as "i686-linux-gnu".
10 For bug reporting instructions, please see:
11 <http://www.gnu.org/software/gdb/bugs/>...
12 Reading symbols from /home/dennis/polygon/d_max...(no debugging symbols found)...done.
13 (gdb) b d_max
14 Breakpoint 1 at 0x80484a0
15 (gdb) run
16 Starting program: /home/dennis/polygon/d_max
17
18 Breakpoint 1, 0x080484a0 in d_max ()
19 (gdb) ni
20 0x080484a4 in d_max ()
21 (gdb) disas $eip
```

²¹Начиная с Pentium Pro, Pentium-II, и т.д.

18.7. ПРИМЕР С СРАВНЕНИЕМ

```

22 Dump of assembler code for function d_max:
23     0x080484a0 <+0>:    fldl    0x4(%esp)
24 => 0x080484a4 <+4>:    fldl    0xc(%esp)
25     0x080484a8 <+8>:    fxch    %st(1)
26     0x080484aa <+10>:   fucomi %st(1),%st
27     0x080484ac <+12>:   fcmovbe %st(1),%st
28     0x080484ae <+14>:   fstp    %st(1)
29     0x080484b0 <+16>:   ret
30 End of assembler dump.
31 (gdb) ni
32 0x080484a8 in d_max ()
33 (gdb) info float
34   R7: Valid    0x3fff99999999999800 +1.19999999999999956
35 =>R6: Valid    0x4000d99999999999800 +3.39999999999999911
36   R5: Empty    0x00000000000000000000
37   R4: Empty    0x00000000000000000000
38   R3: Empty    0x00000000000000000000
39   R2: Empty    0x00000000000000000000
40   R1: Empty    0x00000000000000000000
41   R0: Empty    0x00000000000000000000
42
43 Status Word:    0x3000
44                TOP: 6
45 Control Word:   0x037f    IM DM ZM OM UM PM
46                PC: Extended Precision (64-bits)
47                RC: Round to nearest
48 Tag Word:      0x0fff
49 Instruction Pointer: 0x73:0x080484a4
50 Operand Pointer:  0x7b:0xbffff118
51 Opcode:        0x0000
52 (gdb) ni
53 0x080484aa in d_max ()
54 (gdb) info float
55   R7: Valid    0x4000d99999999999800 +3.39999999999999911
56 =>R6: Valid    0x3fff999999999999800 +1.19999999999999956
57   R5: Empty    0x00000000000000000000
58   R4: Empty    0x00000000000000000000
59   R3: Empty    0x00000000000000000000
60   R2: Empty    0x00000000000000000000
61   R1: Empty    0x00000000000000000000
62   R0: Empty    0x00000000000000000000
63
64 Status Word:    0x3000
65                TOP: 6
66 Control Word:   0x037f    IM DM ZM OM UM PM
67                PC: Extended Precision (64-bits)
68                RC: Round to nearest
69 Tag Word:      0x0fff
70 Instruction Pointer: 0x73:0x080484a8
71 Operand Pointer:  0x7b:0xbffff118
72 Opcode:        0x0000
73 (gdb) disas $eip
74 Dump of assembler code for function d_max:
75     0x080484a0 <+0>:    fldl    0x4(%esp)
76     0x080484a4 <+4>:    fldl    0xc(%esp)
77     0x080484a8 <+8>:    fxch    %st(1)
78 => 0x080484aa <+10>:   fucomi %st(1),%st
79     0x080484ac <+12>:   fcmovbe %st(1),%st
80     0x080484ae <+14>:   fstp    %st(1)
81     0x080484b0 <+16>:   ret
82 End of assembler dump.
83 (gdb) ni
84 0x080484ac in d_max ()
85 (gdb) info registers
86 eax                0x1                1
87 ecx                0xbffff1c4         -1073745468
88 edx                0x8048340          134513472
89 ebx                0xb7fbf000         -1208225792
90 esp                0xbffff10c         0xbffff10c
91 ebp                0xbffff128         0xbffff128

```

18.7. ПРИМЕР С СРАВНЕНИЕМ

```

92 esi          0x0      0
93 edi          0x0      0
94 eip          0x80484ac    0x80484ac <d_max+12>
95 eflags       0x203     [ CF IF ]
96 cs           0x73      115
97 ss           0x7b      123
98 ds           0x7b      123
99 es           0x7b      123
100 fs          0x0        0
101 gs          0x33       51
102 (gdb) ni
103 0x080484ae in d_max ()
104 (gdb) info float
105   R7: Valid   0x4000d99999999999800 +3.39999999999999911
106 =>R6: Valid   0x4000d99999999999800 +3.39999999999999911
107   R5: Empty   0x000000000000000000000
108   R4: Empty   0x000000000000000000000
109   R3: Empty   0x000000000000000000000
110   R2: Empty   0x000000000000000000000
111   R1: Empty   0x000000000000000000000
112   R0: Empty   0x000000000000000000000
113
114 Status Word:      0x3000
115                  TOP: 6
116 Control Word:     0x037f   IM DM ZM OM UM PM
117                  PC: Extended Precision (64-bits)
118                  RC: Round to nearest
119 Tag Word:         0x0fff
120 Instruction Pointer: 0x73:0x080484ac
121 Operand Pointer:   0x7b:0xbffff118
122 Opcode:           0x0000
123 (gdb) disas $eip
124 Dump of assembler code for function d_max:
125   0x080484a0 <+0>:    fldl    0x4(%esp)
126   0x080484a4 <+4>:    fldl    0xc(%esp)
127   0x080484a8 <+8>:    fxch    %st(1)
128   0x080484aa <+10>:   fucomi  %st(1),%st
129   0x080484ac <+12>:   fcmovbe %st(1),%st
130 => 0x080484ae <+14>:   fstp    %st(1)
131   0x080484b0 <+16>:   ret
132 End of assembler dump.
133 (gdb) ni
134 0x080484b0 in d_max ()
135 (gdb) info float
136 =>R7: Valid   0x4000d99999999999800 +3.39999999999999911
137   R6: Empty   0x4000d99999999999800
138   R5: Empty   0x000000000000000000000
139   R4: Empty   0x000000000000000000000
140   R3: Empty   0x000000000000000000000
141   R2: Empty   0x000000000000000000000
142   R1: Empty   0x000000000000000000000
143   R0: Empty   0x000000000000000000000
144
145 Status Word:      0x3800
146                  TOP: 7
147 Control Word:     0x037f   IM DM ZM OM UM PM
148                  PC: Extended Precision (64-bits)
149                  RC: Round to nearest
150 Tag Word:         0x3fff
151 Instruction Pointer: 0x73:0x080484ae
152 Operand Pointer:   0x7b:0xbffff118
153 Opcode:           0x0000
154 (gdb) quit
155 A debugging session is active.
156
157     Inferior 1 [process 30194] will be killed.
158
159 Quit anyway? (y or n) y
160 dennis@ubuntuv:~/polygon$

```

18.7. ПРИМЕР С СРАВНЕНИЕМ

Используя «*pi*», дадим первым двум инструкциям `FLD` исполниться.

Посмотрим регистры FPU (строка 33).

Как уже было указано ранее, регистры FPU это скорее кольцевой буфер, нежели стек (18.5.1 (стр. 221)). И GDB показывает не регистры `STx`, а внутренние регистры FPU (`Rx`). Стрелка (на строке 35) указывает на текущую вершину стека.

Вы можете также увидеть содержимое регистра `TOP` в «Status Word» (строка 44). Там сейчас 6, так что вершина стека сейчас указывает на внутренний регистр 6.

Значения *a* и *b* меняются местами после исполнения `FXCH` (строка 54).

`FUCOMI` исполнилась (строка 83). Посмотрим флаги: `CF` выставлен (строка 95).

`FCMOVBE` действительно скопировал значение *b* (см. строку 104).

`FSTP` оставляет одно значение на вершине стека (строка 136). Значение `TOP` теперь 7, так что вершина FPU-стека указывает на внутренний регистр 7.

18.7.2. ARM

Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Листинг 18.16: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

<code>VMOV</code>	<code>D16, R2, R3 ; b</code>
<code>VMOV</code>	<code>D17, R0, R1 ; a</code>
<code>VCMPE.F64</code>	<code>D17, D16</code>
<code>VMRS</code>	<code>APSR_nzcv, FPSCR</code>
<code>VMOVGT.F64</code>	<code>D16, D17 ; скопировать "b" в D16</code>
<code>VMOV</code>	<code>R0, R1, D16</code>
<code>BX</code>	<code>LR</code>

Очень простой случай. Входные величины помещаются в `D17` и `D16` и сравниваются при помощи инструкции `VCMPE`. Как и в сопроцессорах x86, сопроцессор в ARM имеет свой собственный регистр статуса и флагов (`FPSCR`²²), потому что есть необходимость хранить специфичные для его работы флаги.

И так же, как и в x86, в ARM нет инструкций условного перехода, проверяющих биты в регистре статуса сопроцессора. Поэтому имеется инструкция `VMRS`, копирующая 4 бита (N, Z, C, V) из статуса сопроцессора в биты *общего* статуса (регистр `APSR`²³).

`VMOVGT` это аналог `MOVGT`, инструкция для D-регистров, срабатывающая, если при сравнении один операнд был больше чем второй (*GT* – *Greater Than*).

Если она сработает, в `D16` запишется значение *b*, лежащее в тот момент в `D17`. В обратном случае в `D16` остается значение *a*.

Предпоследняя инструкция `VMOV` готовит то, что было в `D16`, для возврата через пару регистров `R0` и `R1`.

Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

Листинг 18.17: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

<code>VMOV</code>	<code>D16, R2, R3 ; b</code>
<code>VMOV</code>	<code>D17, R0, R1 ; a</code>
<code>VCMPE.F64</code>	<code>D17, D16</code>
<code>VMRS</code>	<code>APSR_nzcv, FPSCR</code>
<code>IT GT</code>	
<code>VMOVGT.F64</code>	<code>D16, D17</code>
<code>VMOV</code>	<code>R0, R1, D16</code>
<code>BX</code>	<code>LR</code>

Почти то же самое, что и в предыдущем примере, за парой отличий. Как мы уже знаем, многие инструкции в режиме ARM можно дополнять условием. Но в режиме Thumb такого нет. В 16-битных инструкций просто нет места для лишних 4 битов, при помощи которых можно было бы закодировать условие выполнения.

²²(ARM) Floating-Point Status and Control Register

²³(ARM) Application Program Status Register

18.7. ПРИМЕР С СРАВНЕНИЕМ

Поэтому в Thumb-2 добавили возможность дополнять Thumb-инструкции условиями. В листинге, сгенерированном при помощи IDA, мы видим инструкцию `VMOVGT`, такую же как и в предыдущем примере.

В реальности там закодирована обычная инструкция `VMOV`, просто IDA добавила суффикс `-GT` к ней, потому что перед этой инструкцией стоит `IT GT`.

Инструкция `IT` определяет так называемый *if-then block*. После этой инструкции можно указывать до четырех инструкций, к каждой из которых будет добавлен суффикс условия.

В нашем примере `IT GT` означает, что следующая за ней инструкция будет исполнена, если условие *GT* (*Greater Than*) справедливо.

Теперь более сложный пример. Кстати, из Angry Birds (для iOS):

Листинг 18.18: Angry Birds Classic

```
...
ITE NE
VMOVNE      R2, R3, D16
VMOVEQ      R2, R3, D17
BLX         _objc_msgSend ; без суффикса
...
```

`ITE` означает *if-then-else* и кодирует суффиксы для двух следующих за ней инструкций.

Первая из них исполнится, если условие, закодированное в `ITE` (*NE, not equal*) будет в тот момент справедливо, а вторая — если это условие не сработает. (Обратное условие от `NE` это `EQ` (*equal*)).

Инструкция следующая за второй `VMOV` (или `VMOVEQ`) нормальная, без суффикса (`BLX`).

Ещё чуть сложнее, и снова этот фрагмент из Angry Birds:

Листинг 18.19: Angry Birds Classic

```
...
ITTTT EQ
MOVEQ       R0, R4
ADDEQ       SP, SP, #0x20
POPEQ.W     {R8,R10}
POPEQ       {R4-R7,PC}
BLX         __stack_chk_fail ; без суффикса
...
```

Четыре символа «Т» в инструкции означают, что четыре последующие инструкции будут исполнены если условие соблюдается. Поэтому IDA добавила ко всем четырем инструкциям суффикс `-EQ`. А если бы здесь было, например, `ITEEE EQ` (*if-then-else-else-else*), тогда суффиксы для следующих четырех инструкций были бы расставлены так:

```
-EQ
-NE
-NE
-NE
```

Ещё фрагмент из Angry Birds:

Листинг 18.20: Angry Birds Classic

```
...
CMP.W       R0, #0xFFFFFFFF
ITTE LE
SUBLE.W     R10, R0, #1
NEGLE       R0, R0
MOVGT       R10, R0
MOVS        R6, #0 ; без суффикса
CBZ         R0, loc_1E7E32 ; без суффикса
...
```

`ITTE` (*if-then-then-else*) означает, что первая и вторая инструкции исполняются, если условие `LE` (*Less or Equal*) справедливо, а третья — если справедливо обратное условие (`GT` — *Greater Than*).

Компиляторы способны генерировать далеко не все варианты.

Например, в вышеупомянутой игре Angry Birds (версия *classic* для iOS)

18.7. ПРИМЕР С СРАВНЕНИЕМ

встречаются только такие варианты инструкции `IT`: `IT`, `ITE`, `ITT`, `ITTE`, `ITTT`, `ITTTT`. Как это узнать? В [IDA](#) можно сгенерировать листинг (что и было сделано), только в опциях был установлен показ 4 байтов для каждого опкода.

Затем, зная что старшая часть 16-битного опкода (`IT` это `0xBF`), сделаем при помощи `grep` это:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

Кстати, если писать на ассемблере для режима Thumb-2 вручную, и дополнять инструкции суффиксами условия, то ассемблер автоматически будет добавлять инструкцию `IT` с соответствующими флагами там, где надо.

Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Листинг 18.21: Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```
b          = -0x20
a          = -0x18
val_to_return = -0x10
saved_R7   = -4

        STR        R7, [SP,#saved_R7]!
        MOV        R7, SP
        SUB        SP, SP, #0x1C
        BIC        SP, SP, #7
        VMOV       D16, R2, R3
        VMOV       D17, R0, R1
        VSTR       D17, [SP,#0x20+a]
        VSTR       D16, [SP,#0x20+b]
        VLDR       D16, [SP,#0x20+a]
        VLDR       D17, [SP,#0x20+b]
        VCMPE.F64  D16, D17
        VMRS       APSR_nzcv, FPSCR
        BLE        loc_2E08
        VLDR       D16, [SP,#0x20+a]
        VSTR       D16, [SP,#0x20+val_to_return]
        B          loc_2E10

loc_2E08
        VLDR       D16, [SP,#0x20+b]
        VSTR       D16, [SP,#0x20+val_to_return]

loc_2E10
        VLDR       D16, [SP,#0x20+val_to_return]
        VMOV       R0, R1, D16
        MOV        SP, R7
        LDR        R7, [SP+0x20+b],#4
        BX        LR
```

Почти то же самое, что мы уже видели, но много избыточного кода из-за хранения *a* и *b*, а также выходного значения, в локальном стеке.

Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 18.22: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
PUSH    {R3-R7,LR}
MOVS    R4, R2
MOVS    R5, R3
MOVS    R6, R0
MOVS    R7, R1
BL      __aeabi_cdrcmple
BCS     loc_1C0
MOVS    R0, R6
MOVS    R1, R7
POP     {R3-R7,PC}

loc_1C0
MOVS    R0, R4
MOVS    R1, R5
```

Keil не генерирует FPU-инструкции, потому что не рассчитывает на то, что они будут поддерживаться, а простым сравнением побитово здесь не обойтись.

Для сравнения вызывается библиотечная функция `__aeabi_cdrcmple`.

N.B. Результат сравнения эта функция оставляет в флагах, чтобы следующая за вызовом инструкция `BCS` (*Carry set – Greater than or equal*) могла работать без дополнительного кода.

18.7.3. ARM64

Оптимизирующий GCC (Linaro) 4.9

```
d_max:
; D0 – a, D1 – b
    fcmpe    d0, d1
    fcsel    d0, d0, d1, gt
; теперь результат в D0
    ret
```

В ARM64 ISA теперь есть FPU-инструкции, устанавливающие флаги CPU `APSR` вместо `FPSCR` для удобства. FPU больше не отдельное устройство (по крайней мере логически). Это `FCMPE`. Она сравнивает два значения, переданных в `D0` и `D1` (а это первый и второй аргументы функции) и выставляет флаги в `APSR` (N, Z, C, V).

`FCSEL` (*Floating Conditional Select*) копирует значение `D0` или `D1` в `D0` в зависимости от условия (`GT` – Greater Than – больше чем), и снова, она использует флаги в регистре `APSR` вместо `FPSCR`. Это куда удобнее, если сравнивать с тем набором инструкций, что был в процессорах раньше.

Если условие верно (`GT`), тогда значение из `D0` копируется в `D0` (т.е. ничего не происходит). Если условие не верно, то значение `D1` копируется в `D0`.

Неоптимизирующий GCC (Linaro) 4.9

```
d_max:
; сохранить входные аргументы в "Register Save Area"
    sub     sp, sp, #16
    str     d0, [sp, 8]
    str     d1, [sp]
; перезагрузить значения
    ldr     x1, [sp, 8]
    ldr     x0, [sp]
    fmov    d0, x1
    fmov    d1, x0
; D0 – a, D1 – b
    fcmpe   d0, d1
    ble     .L76
; a>b; загрузить D0 (a) в X0
    ldr     x0, [sp, 8]
    b       .L74
.L76:
; a<=b; загрузить D1 (b) в X0
    ldr     x0, [sp]
.L74:
; результат в X0
    fmov    d0, x0
; результат в D0
    add     sp, sp, 16
    ret
```

Неоптимизирующий GCC более многословен. В начале функция сохраняет значения входных аргументов в локальном стеке (*Register Save Area*). Затем код перезагружает значения в регистры `X0 / X1` и наконец копирует их в `D0 / D1` для сравнения инструкцией `FCMPE`. Много избыточного кода, но так работают неоптимизирующие компиляторы. `FCMPE` сравнивает значения и устанавливает флаги в `APSR`. В этот момент компилятор ещё не думает о более удобной инструкции `FCSEL`, так что он работает старым методом: использует инструкцию `BLE` (*Branch if Less than or Equal* (переход

18.7. ПРИМЕР С СРАВНЕНИЕМ

если меньше или равно)). В одном случае ($a > b$) значение a перезагружается в **X0**. В другом случае ($a \leq b$) значение b загружается в **X0**. Наконец, значение из **X0** копируется в **D0**, потому что возвращаемое значение оставляется в этом регистре.

Упражнение

Для упражнения вы можете попробовать оптимизировать этот фрагмент кода вручную, удалив избыточные инструкции, но не добавляя новых (включая **FCSEL**).

Оптимизирующий GCC (Linaro) 4.9 – float

Перепишем пример. Теперь здесь *float* вместо *double*.

```
float f_max (float a, float b)
{
    if (a>b)
        return a;

    return b;
};
```

```
f_max:
; S0 – a, S1 – b
    fcmpe    s0, s1
    fcsele   s0, s0, s1, gt
; теперь результат в S0
    ret
```

Всё то же самое, только используются S-регистры вместо D-. Так что числа типа *float* передаются в 32-битных S-регистрах (а это младшие части 64-битных D-регистров).

18.7.4. MIPS

В сопроцессоре MIPS есть бит результата, который устанавливается в FPU и проверяется в CPU. Ранние MIPS имели только один бит (с названием FCC0), а у поздних их 8 (с названием FCC7-FCC0). Этот бит (или биты) находятся в регистре с названием FCCR.

Листинг 18.23: Оптимизирующий GCC 4.4.5 (IDA)

```
d_max:
; установить бит условия FPU в 1, если $f14<$f12 (b<a):
    c.lt.d   $f14, $f12
    or      $at, $zero ; NOP
; перейти на locret_14 если бит условия выставлен
    bc1t     locret_14
; эта инструкция всегда выполняется (установить значение для возврата в "a"):
    mov.d    $f0, $f12 ; branch delay slot
; эта инструкция выполняется только если переход не произошел (т.е. если b>=a)
; установить значение для возврата в "b":
    mov.d    $f0, $f14

locret_14:
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP
```

C.LT.D сравнивает два значения. **LT** это условие «Less Than» (меньше чем). **D** означает переменные типа *double*. В зависимости от результата сравнения, бит FCC0 устанавливается или очищается.

BC1T проверяет бит FCC0 и делает переход, если бит выставлен. **T** означает что переход произойдет если бит выставлен («True»). Имеется также инструкция «BC1F» которая срабатывает, если бит сброшен («False»).

В зависимости от перехода один из аргументов функции помещается в регистр \$F0.

18.8. Стек, калькуляторы и обратная польская запись

Теперь понятно, почему некоторые старые калькуляторы использовали обратную польскую запись ²⁴.

Например для сложения 12 и 34 нужно было набрать 12, потом 34, потом нажать знак «плюс».

Это потому что старые калькуляторы просто реализовали стековую машину и это было куда проще, чем обрабатывать сложные выражения со скобками.

18.9. x64

О том, как происходит работа с числами с плавающей запятой в x86-64, читайте здесь: [28](#) (стр. [425](#)).

18.10. Упражнения

- <http://challenges.re/60>
- <http://challenges.re/61>

²⁴ru.wikipedia.org/wiki/Обратная_польская_запись

Глава 19

Массивы

Массив это просто набор переменных в памяти, обязательно лежащих рядом и обязательно одного типа¹.

19.1. Простой пример

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

19.1.1. x86

MSVC

Компилируем:

Листинг 19.1: MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84                                ; size = 4
_a$ = -80                                ; size = 80
_main     PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84                      ; 00000054H
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN6@main
$LN5@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp     DWORD PTR _i$[ebp], 20      ; 00000014H
    jge     SHORT $LN4@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
```

¹ АКА «гомогенный контейнер»

19.1. ПРОСТОЙ ПРИМЕР

```
mov     edx, DWORD PTR _i$[ebp]
mov     DWORD PTR _a$[ebp+edx*4], ecx
jmp     SHORT $LN5@main
$LN4@main:
mov     DWORD PTR _i$[ebp], 0
jmp     SHORT $LN3@main
$LN2@main:
mov     eax, DWORD PTR _i$[ebp]
add     eax, 1
mov     DWORD PTR _i$[ebp], eax
$LN3@main:
cmp     DWORD PTR _i$[ebp], 20      ; 00000014H
jge     SHORT $LN1@main
mov     ecx, DWORD PTR _i$[ebp]
mov     edx, DWORD PTR _a$[ebp+ecx*4]
push    edx
mov     eax, DWORD PTR _i$[ebp]
push    eax
push    OFFSET $SG2463
call    _printf
add     esp, 12                    ; 0000000cH
jmp     SHORT $LN2@main
$LN1@main:
xor     eax, eax
mov     esp, ebp
pop     ebp
ret     0
_main   ENDP
```

Ничего особенного, просто два цикла. Один изменяет массив, второй печатает его содержимое. Команда `shl ecx, 1` используется для умножения `ECX` на 2, об этом ниже ([17.2.1](#) (стр. 212)).

Под массив выделено в стеке 80 байт, это 20 элементов по 4 байта.

19.1. ПРОСТОЙ ПРИМЕР

Попробуем этот пример в OllyDbg.

Видно, как заполнился массив: каждый элемент это 32-битное слово типа *int*, с шагом 2:

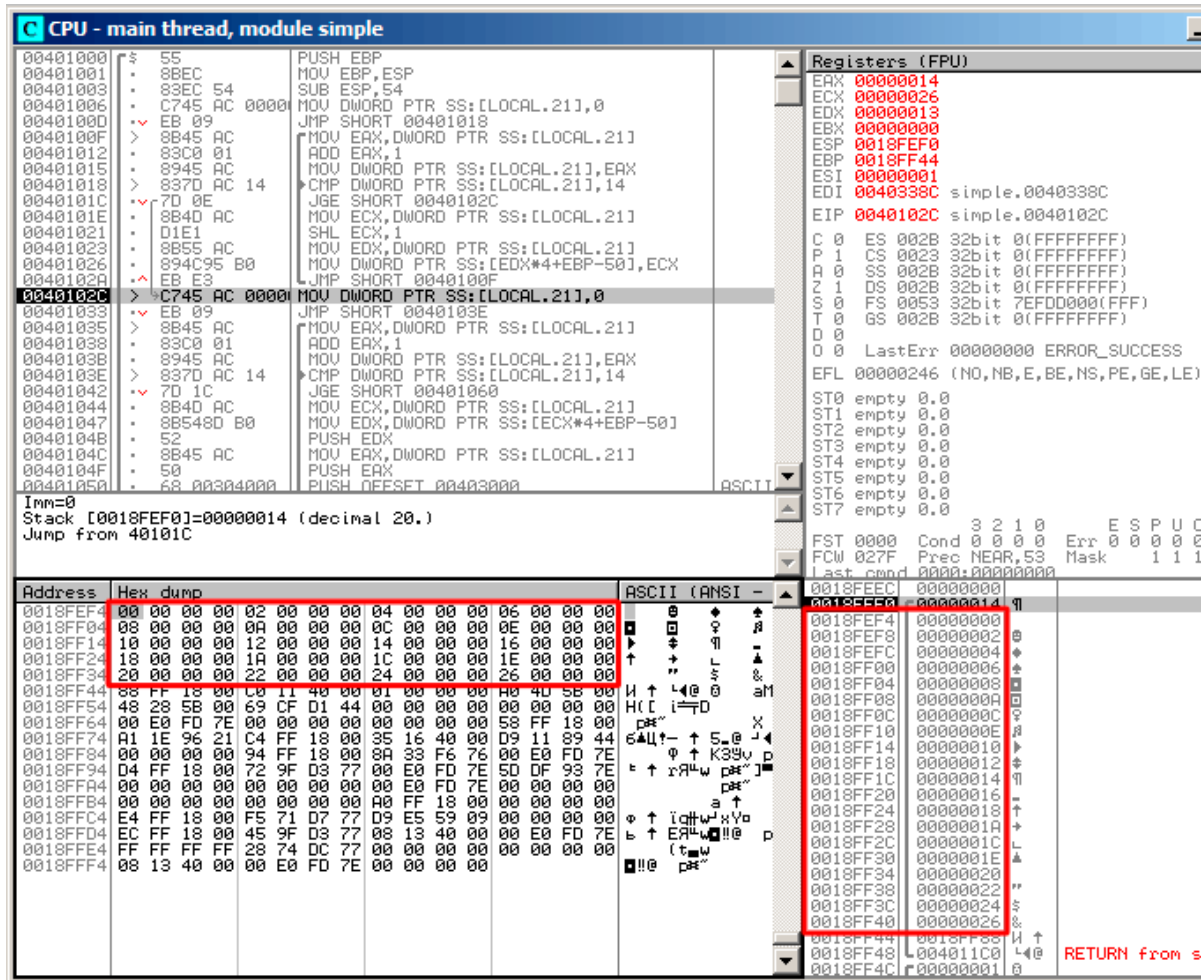


Рис. 19.1: OllyDbg: после заполнения массива

А так как этот массив находится в стеке, то мы видим все его 20 элементов внутри стека.

GCC

Рассмотрим результат работы GCC 4.4.1:

Листинг 19.2: GCC 4.4.1

main	public main	
	proc near	; DATA XREF: _start+17
var_70	= dword ptr -70h	
var_6C	= dword ptr -6Ch	
var_68	= dword ptr -68h	
i_2	= dword ptr -54h	
i	= dword ptr -4	
	push ebp	
	mov ebp, esp	
	and esp, 0FFFFFFF0h	
	sub esp, 70h	
	mov [esp+70h+i], 0	; i=0
	jmp short loc_804840A	
loc_80483F7:	mov eax, [esp+70h+i]	
	mov edx, [esp+70h+i]	
	add edx, edx	; edx=i*2

19.1. ПРОСТОЙ ПРИМЕР

```
        mov     [esp+eax*4+70h+i_2], edx
        add     [esp+70h+i], 1          ; i++

loc_804840A:
        cmp     [esp+70h+i], 13h
        jle     short loc_80483F7
        mov     [esp+70h+i], 0
        jmp     short loc_8048441

loc_804841B:
        mov     eax, [esp+70h+i]
        mov     edx, [esp+eax*4+70h+i_2]
        mov     eax, offset aADD ; "a[%d]=%d\n"
        mov     [esp+70h+var_68], edx
        mov     edx, [esp+70h+i]
        mov     [esp+70h+var_6C], edx
        mov     [esp+70h+var_70], eax
        call    _printf
        add     [esp+70h+i], 1

loc_8048441:
        cmp     [esp+70h+i], 13h
        jle     short loc_804841B
        mov     eax, 0
        leave
        retn

main
        endp
```

Переменная *a* в нашем примере имеет тип *int** (указатель на *int*). Вы можете попробовать передать в другую функцию указатель на массив, но точнее было бы сказать, что передается указатель на первый элемент массива (а адреса остальных элементов массива можно вычислить очевидным образом).

Если индексировать этот указатель как *a[idx]*, *idx* просто прибавляется к указателю и возвращается элемент, расположенный там, куда ссылается вычисленный указатель.

Вот любопытный пример. Строка символов вроде «*string*» это массив из символов. Она имеет тип *const char[]*. К этому указателю также можно применять индекс.

Поэтому можно написать даже так: «*string*»[*i*] — это совершенно легальное выражение в Си/Си++!

19.1.2. ARM

Неоптимизирующий Keil 6/2013 (Режим ARM)

```
EXPORT _main
_main
        STMFD   SP!, {R4,LR}
        SUB     SP, SP, #0x50          ; выделить место для 20-и переменных типа int

; первый цикл

        MOV     R4, #0                ; i
        B       loc_4A0

loc_494
        MOV     R0, R4,LSL#1          ; R0=R4*2
        STR     R0, [SP,R4,LSL#2]     ; сохранить R0 в SP+R4<<2 (то же что и SP+R4*4)
        ADD     R4, R4, #1            ; i=i+1

loc_4A0
        CMP     R4, #20               ; i<20?
        BLT     loc_494               ; да, запустить тело цикла снова

; второй цикл

        MOV     R4, #0                ; i
        B       loc_4C4

loc_4B0
        LDR     R2, [SP,R4,LSL#2]     ; (второй аргумент printf) R2=*(SP+R4<<4) (то же что и *(SP+R4*4))
```

19.1. ПРОСТОЙ ПРИМЕР

```
MOV    R1, R4          ; (первый аргумент printf) R1=i
ADR    R0, aADD         ; "a[%d]=%d\n"
BL     __2printf
ADD    R4, R4, #1       ; i=i+1

loc_4C4
CMP     R4, #20          ; i<20?
BLT     loc_4B0          ; да, запустить тело цикла снова
MOV     R0, #0           ; значение для возврата
ADD     SP, SP, #0x50     ; освободить блок в стеке, выделенное для 20 переменных
LDMFD  SP!, {R4,PC}
```

Тип *int* требует 32 бита для хранения (или 4 байта),

так что для хранения 20 переменных типа *int*, нужно 80 (0x50) байт.

Поэтому инструкция `SUB SP, SP, #0x50` в прологе функции выделяет в локальном стеке под массив именно столько места.

И в первом и во втором цикле итератор цикла *i* будет постоянно находится в регистре `R4`.

Число, которое нужно записать в массив, вычисляется так: $i * 2$, и это эквивалентно сдвигу на 1 бит влево, так что инструкция `MOV R0, R4, LSL#1` делает это.

`STR R0, [SP, R4, LSL#2]` записывает содержимое `R0` в массив. Указатель на элемент массива вычисляется так: `SP` указывает на начало массива, `R4` это *i*.

Так что сдвигаем *i* на 2 бита влево, что эквивалентно умножению на 4 (ведь каждый элемент массива занимает 4 байта) и прибавляем это к адресу начала массива.

Во втором цикле используется обратная инструкция `LDR R2, [SP, R4, LSL#2]`. Она загружает из массива нужное значение и указатель на него вычисляется точно так же.

Оптимизирующий Keil 6/2013 (Режим Thumb)

```
_main
    PUSH    {R4,R5,LR}
; выделить место для 20 переменных типа int + еще одной переменной
    SUB     SP, SP, #0x54

; первый цикл

    MOVS    R0, #0        ; i
    MOV     R5, SP        ; указатель на первый элемент массива

loc_1CE
    LSLS    R1, R0, #1     ; R1=i<<1 (то же что и i*2)
    LSLS    R2, R0, #2     ; R2=i<<2 (то же что и i*4)
    ADDS    R0, R0, #1     ; i=i+1
    CMP     R0, #20        ; i<20?
    STR     R1, [R5,R2]    ; сохранить R1 в *(R5+R2) (то же что и R5+i*4)
    BLT     loc_1CE        ; да, i<20, запустить тело цикла снова

; второй цикл

    MOVS    R4, #0        ; i=0

loc_1DC
    LSLS    R0, R4, #2     ; R0=i<<2 (то же что и i*4)
    LDR     R2, [R5,R0]    ; загрузить из *(R5+R0) (то же что и R5+i*4)
    MOVS    R1, R4
    ADR     R0, aADD       ; "a[%d]=%d\n"
    BL     __2printf
    ADDS    R4, R4, #1     ; i=i+1
    CMP     R4, #20        ; i<20?
    BLT     loc_1DC        ; да, i<20, запустить тело цикла снова
    MOVS    R0, #0        ; значение для возврата
; освободить блок в стеке, выделенное для 20-и переменных типа int и еще одной переменной
    ADD     SP, SP, #0x54
    POP     {R4,R5,PC}
```

19.1. ПРОСТОЙ ПРИМЕР

Код для Thumb очень похожий. В Thumb имеются отдельные инструкции для битовых сдвигов (как `LSLS`), вычисляющие и число для записи в массив и адрес каждого элемента массива.

Компилятор почему-то выделил в локальном стеке немного больше места, однако последние 4 байта не используются.

Неоптимизирующий GCC 4.9.1 (ARM64)

Листинг 19.3: Неоптимизирующий GCC 4.9.1 (ARM64)

```
.LC0:
    .string "a[%d]=%d\n"
main:
; сохранить FP и LR в стековом фрейме:
    stp    x29, x30, [sp, -112]!
; установить стековый фрейм (FP=SP)
    add    x29, sp, 0
; установить начальное значение счетчика в 0 (WZR это регистр всегда содержащий ноль):
    str    wzr, [x29,108]
; перейти на код проверки условия цикла:
    b      .L2
.L3:
; загрузить значение переменной "i":
    ldr    w0, [x29,108]
; умножить на 2:
    lsl    w2, w0, 1
; найти место массива в локальном стеке:
    add    x0, x29, 24
; загрузить 32-битное число из локального стека и расширить его до 64-битного учитывая знак:
    ldrsw  x1, [x29,108]
; вычислить адрес элемента (X0+X1<<2=array address+i*4) и записать W2 (i*2) там:
    str    w2, [x0,x1,lsl 2]
; инкремент счетчика (i):
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L2:
; проверить, закончили ли:
    ldr    w0, [x29,108]
    cmp    w0, 19
; перейти на L3 (начало тела цикла), если нет:
    ble    .L3
; здесь начинается вторая часть функции.
; устанавливаем первичного значение счетчика в 0.
; кстати, это же место в локальном стеке было уже использовано под счетчик
; потому что та же локальная переменная (i) используется как счетчик.
    str    wzr, [x29,108]
    b      .L4
.L5:
; вычислить адрес массива:
    add    x0, x29, 24
; загрузить переменную "i":
    ldrsw  x1, [x29,108]
; загрузить значение из массива по адресу (X0+X1<<2 = адрес массива + i*4)
    ldr    w2, [x0,x1,lsl 2]
; загрузить адрес строки "a[%d]=%d\n" :
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
; загрузить переменную "i" в W1 и передать её в printf() как второй аргумент:
    ldr    w1, [x29,108]
; W2 всё еще содержит загруженный элемент из массива.
; вызов printf():
    bl     printf
; инкремент переменной "i":
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L4:
; закончили?
    ldr    w0, [x29,108]
```

19.1. ПРОСТОЙ ПРИМЕР

```
        cmp      w0, 19
; перейти на начало тела цикла, если нет:
        ble      .L5
; возврат 0
        mov      w0, 0
; восстановить FP и LR:
        ldp      x29, x30, [sp], 112
        ret
```

19.1.3. MIPS

Функция использует много S-регистров, которые должны быть сохранены. Вот почему их значения сохраняются в прологе функции и восстанавливаются в эпилоге.

Листинг 19.4: Оптимизирующий GCC 4.4.5 (IDA)

```
main:
var_70      = -0x70
var_68      = -0x68
var_14      = -0x14
var_10      = -0x10
var_C       = -0xC
var_8       = -8
var_4       = -4
; пролог функции:
        lui      $gp, (__gnu_local_gp >> 16)
        addiu    $sp, -0x80
        la       $gp, (__gnu_local_gp & 0xFFFF)
        sw       $ra, 0x80+var_4($sp)
        sw       $s3, 0x80+var_8($sp)
        sw       $s2, 0x80+var_C($sp)
        sw       $s1, 0x80+var_10($sp)
        sw       $s0, 0x80+var_14($sp)
        sw       $gp, 0x80+var_70($sp)
        addiu    $s1, $sp, 0x80+var_68
        move     $v1, $s1
        move     $v0, $zero
; это значение используется как терминатор цикла.
; оно было вычислено компилятором GCC на стадии компиляции:
        li       $a0, 0x28 # '('

loc_34:                                # CODE XREF: main+3C
; сохранить значение в памяти:
        sw       $v0, 0($v1)
; увеличивать значение (которое будет записано) на 2 на каждой итерации:
        addiu    $v0, 2
; дошли до терминатора цикла?
        bne      $v0, $a0, loc_34
; в любом случае, добавляем 4 к адресу:
        addiu    $v1, 4
; цикл заполнения массива закончился
; начало второго цикла
        la       $s3, $LC0           # "a[%d]=%d\n"
; переменная "i" будет находится в $s0:
        move     $s0, $zero
        li       $s2, 0x14

loc_54:                                # CODE XREF: main+70
; вызов printf():
        lw       $t9, (printf & 0xFFFF)($gp)
        lw       $a2, 0($s1)
        move     $a1, $s0
        move     $a0, $s3
        jalr     $t9
; инкремент "i":
        addiu    $s0, 1
        lw       $gp, 0x80+var_70($sp)
```

19.2. ПЕРЕПОЛНЕНИЕ БУФЕРА

```
; перейти на начало тела цикла, если конец еще не достигнут:
    bne     $s0, $s2, loc_54
; передвинуть указатель на следующее 32-битное слово:
    addiu   $s1, 4
; эпилог функции
    lw      $ra, 0x80+var_4($sp)
    move    $v0, $zero
    lw      $s3, 0x80+var_8($sp)
    lw      $s2, 0x80+var_C($sp)
    lw      $s1, 0x80+var_10($sp)
    lw      $s0, 0x80+var_14($sp)
    jr      $ra
    addiu   $sp, 0x80

$LCO:      .ascii "a[%d]=%d\n"<0> # DATA XREF: main+44
```

Интересная вещь: здесь два цикла и в первом не нужна переменная i , а нужна только переменная $i * 2$ (скачущая через 2 на каждой итерации) и ещё адрес в памяти (скачущий через 4 на каждой итерации).

Так что мы видим здесь две переменных: одна (в $\$V0$) увеличивается на 2 каждый раз, и вторая (в $\$V1$) — на 4.

Второй цикл содержит вызов `printf()`. Он должен показывать значение i пользователю, поэтому здесь есть переменная, увеличивающаяся на 1 каждый раз (в $\$S0$), а также адрес в памяти (в $\$S1$) увеличивающийся на 4 каждый раз.

Это напоминает нам оптимизацию циклов, которую мы рассматривали ранее: 40 (стр. 475). Цель оптимизации в том, чтобы избавиться от операций умножения.

19.2. Переполнение буфера

19.2.1. Чтение за пределами массива

Итак, индексация массива — это просто *массив[индекс]*. Если вы присмотритесь к коду, в цикле печати значений массива через `printf()` вы не увидите проверок индекса, *меньше ли он двадцати?* А что будет если он будет 20 или больше? Эта одна из особенностей Си/Си++, за которую их, собственно, и ругают.

Вот код, который и компилируется и работает:

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    printf ("a[20]=%d\n", a[20]);

    return 0;
};
```

Вот результат компиляции в (MSVC 2008):

Листинг 19.5: Неоптимизирующий MSVC 2008

```
$SG2474 DB      'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN2@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
```

19.2. ПЕРЕПОЛНЕНИЕ БУФЕРА

```
add     eax, 1
mov     DWORD PTR _i$[ebp], eax
$LN3@main:
cmp     DWORD PTR _i$[ebp], 20
jge     SHORT $LN1@main
mov     ecx, DWORD PTR _i$[ebp]
shl     ecx, 1
mov     edx, DWORD PTR _i$[ebp]
mov     DWORD PTR _a$[ebp+edx*4], ecx
jmp     SHORT $LN2@main
$LN1@main:
mov     eax, DWORD PTR _a$[ebp+80]
push    eax
push    OFFSET $SG2474 ; 'a[20]=%'
call    DWORD PTR __imp__printf
add     esp, 8
xor     eax, eax
mov     esp, ebp
pop     ebp
ret     0
_main   ENDP
_TEXT   ENDS
END
```

Данный код при запуске выдал вот такой результат:

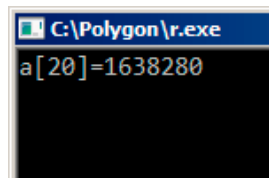


Рис. 19.2: OllyDbg: вывод в консоль

Это просто *что-то*, что волею случая лежало в стеке рядом с массивом, через 80 байт от его первого элемента.

19.2. ПЕРЕПОЛНЕНИЕ БУФЕРА

Попробуем узнать в OllyDbg, что это за значение. Загружаем и находим это значение, находящееся точно после последнего элемента массива:

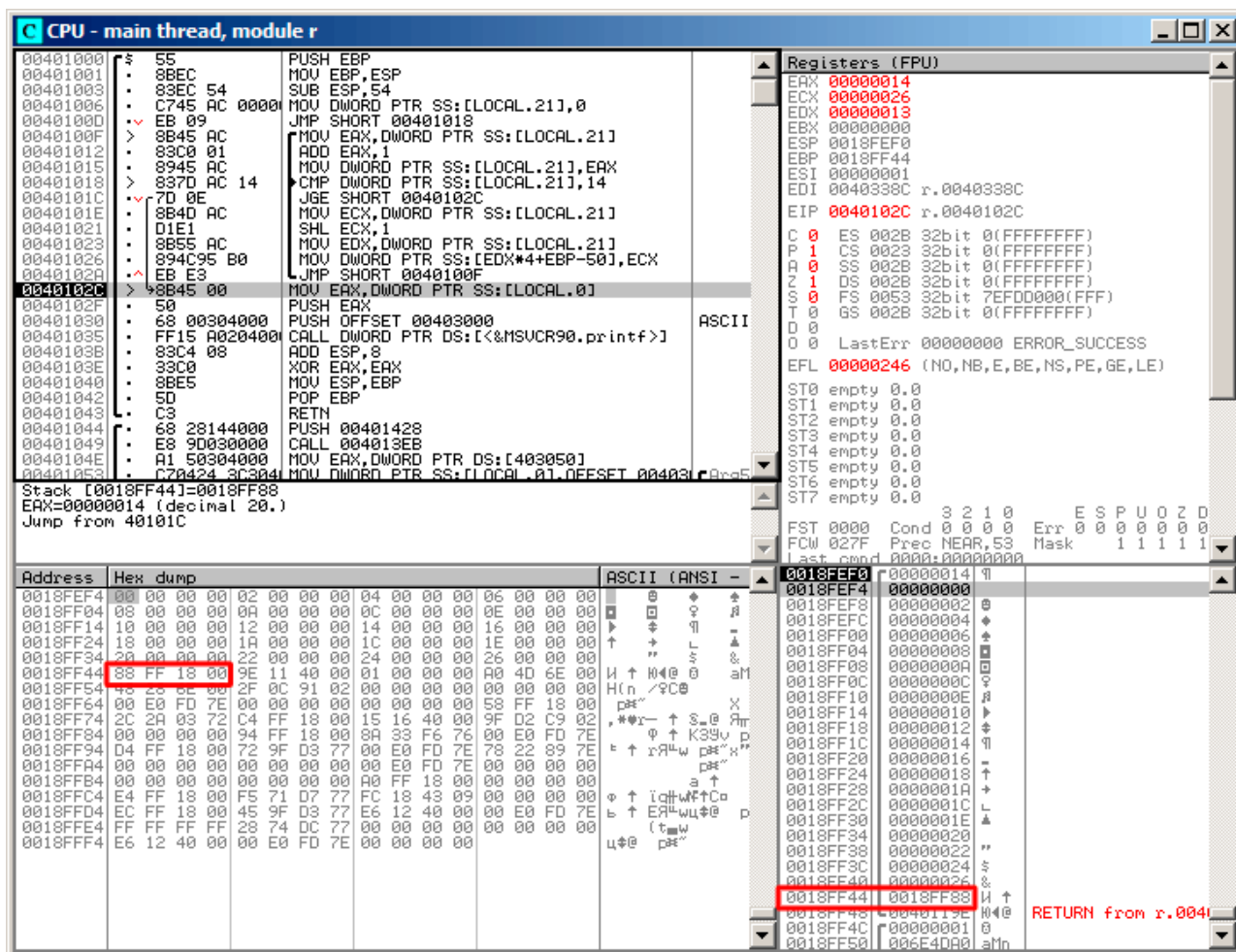


Рис. 19.3: OllyDbg: чтение 20-го элемента и вызов `printf()`

Что это за значение? Судя по разметке стека, это сохраненное значение регистра EBP.

19.2. ПЕРЕПОЛНЕНИЕ БУФЕРА

Трассируем далее, и видим, как оно восстанавливается:

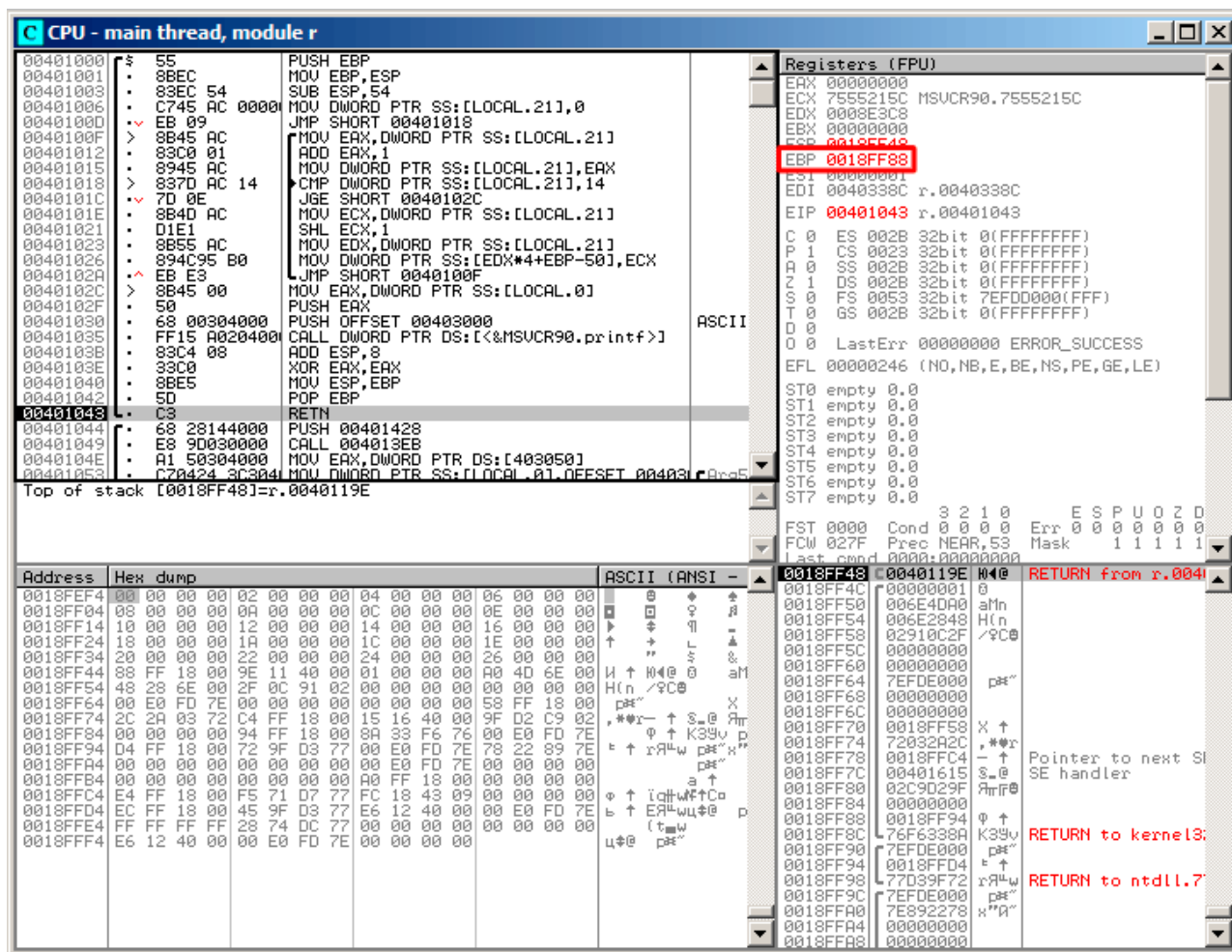


Рис. 19.4: OllyDbg: восстановление EBP

Действительно, а как могло бы быть иначе? Компилятор мог бы встроить какой-то код, каждый раз проверяющий индекс на соответствие пределам массива, как в языках программирования более высокого уровня², что делало бы запускаемый код медленнее.

19.2.2. Запись за пределы массива

Итак, мы прочитали какое-то число из стека явно *нелегально*, а что если мы запишем?

Вот что мы пишем:

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<30; i++)
        a[i]=i;

    return 0;
};
```

MSVC

И вот что имеем на ассемблере:

²Java, Python, и т.д.


```
_TEXT    SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main    PROC
push     ebp
mov      ebp, esp
sub      esp, 84
mov      DWORD PTR _i$[ebp], 0
jmp      SHORT $LN3@main
$LN2@main:
mov      eax, DWORD PTR _i$[ebp]
add      eax, 1
mov      DWORD PTR _i$[ebp], eax
$LN3@main:
cmp      DWORD PTR _i$[ebp], 30 ; 0000001eH
jge      SHORT $LN1@main
mov      ecx, DWORD PTR _i$[ebp]
mov      edx, DWORD PTR _i$[ebp]      ; явный промах компилятора. эта инструкция лишняя.
mov      DWORD PTR _a$[ebp+ecx*4], edx ; а здесь в качестве второго операнда подошел бы ECX.
jmp      SHORT $LN2@main
$LN1@main:
xor      eax, eax
mov      esp, ebp
pop      ebp
ret      0
_main    ENDP
```

Запускаете скомпилированную программу, и она падает. Немудрено. Но давайте теперь узнаем, где именно.

19.2. ПЕРЕПОЛНЕНИЕ БУФЕРА

Загружаем в OllyDbg, трассируем пока запишутся все 30 элементов:

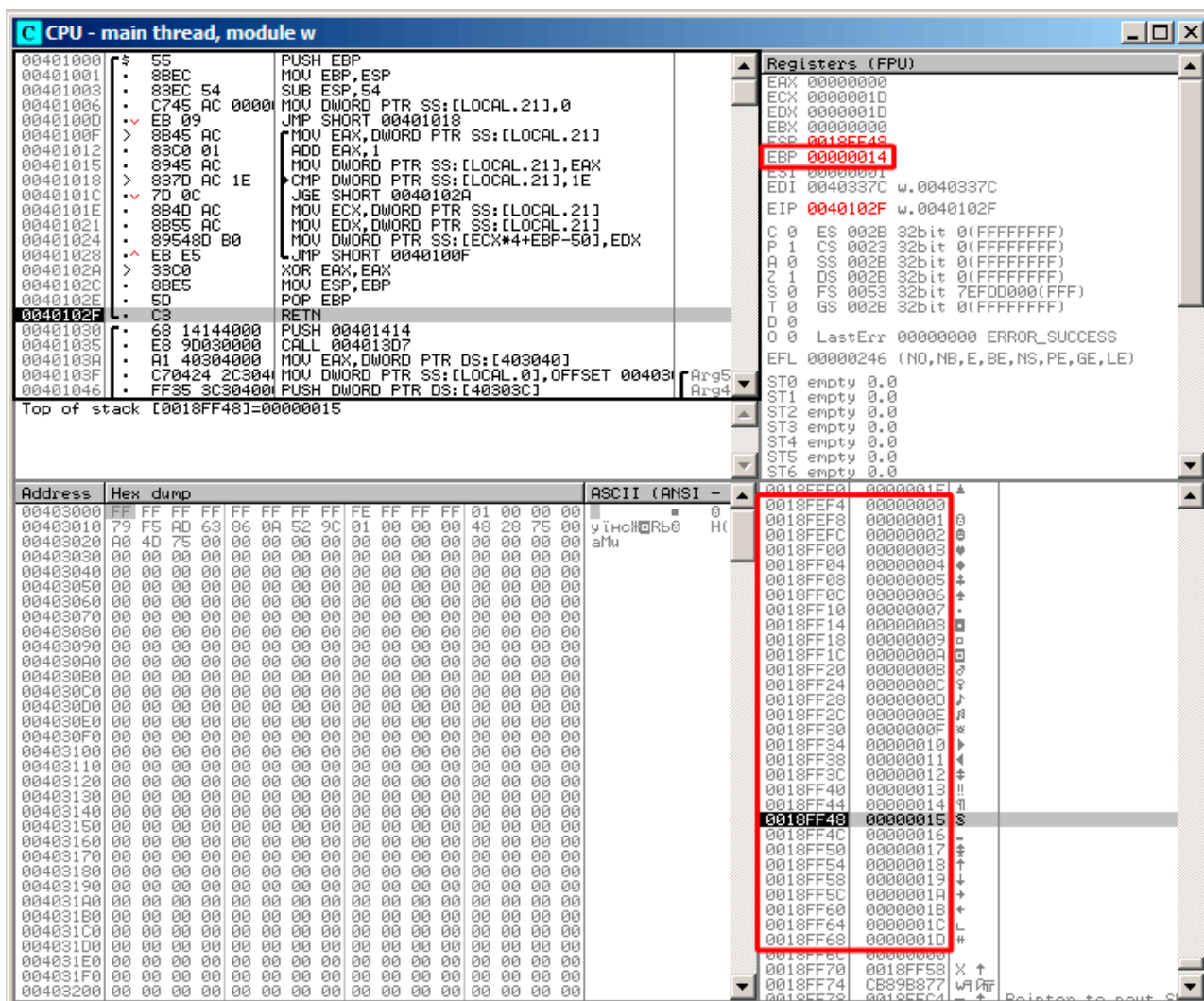


Рис. 19.5: OllyDbg: после восстановления EBP

Доходим до конца функции:

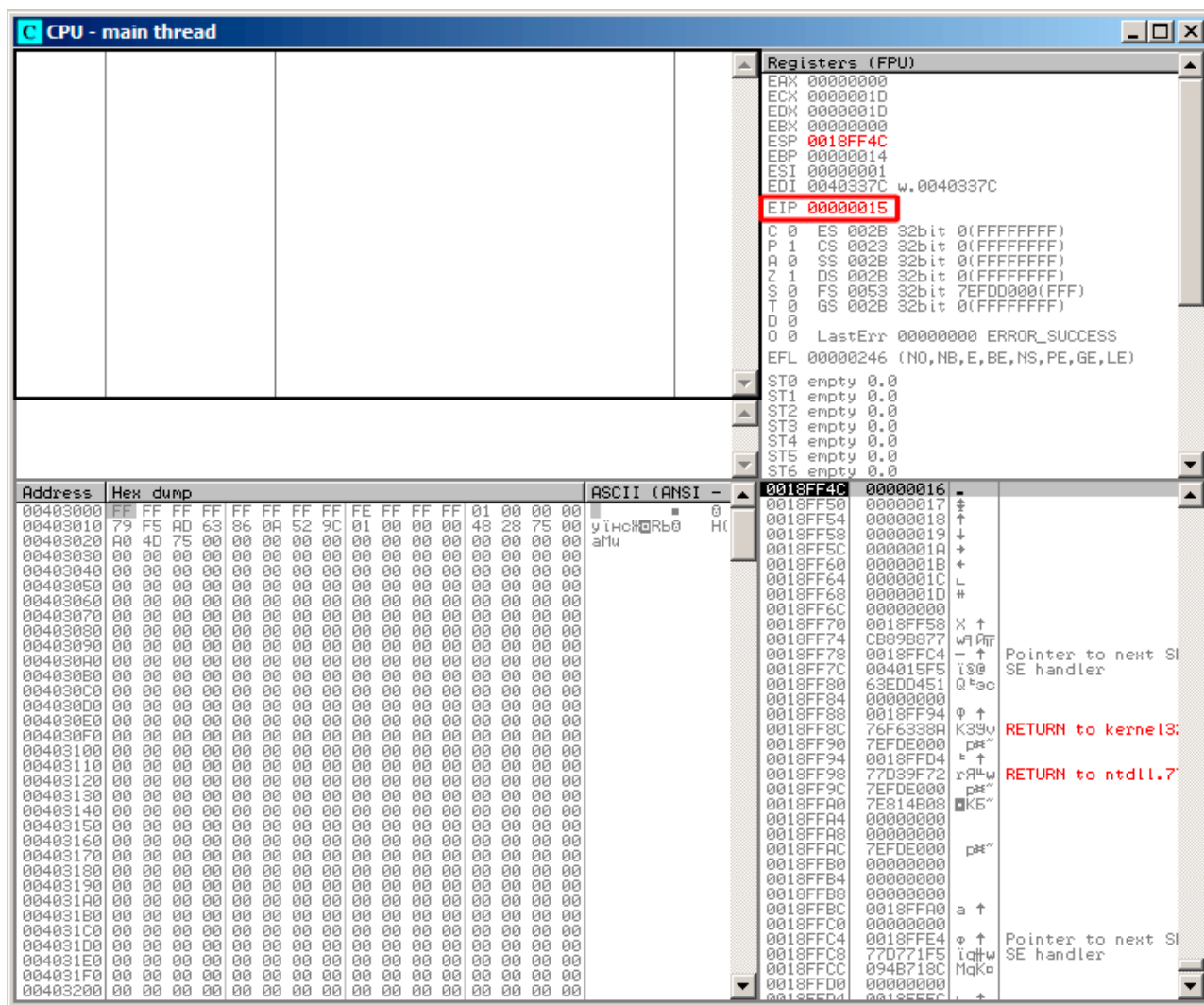


Рис. 19.6: OllyDbg: EIP восстановлен, но OllyDbg не может дизассемблировать по адресу 0x15

Итак, следите внимательно за регистрами.

EIP теперь 0x15. Это явно нелегальный адрес для кода — по крайней мере, win32-кода! Мы там как-то очутились, причем, сами того не хотели. Интересен также тот факт, что в EBP хранится 0x14, а в ECX и EDX хранится 0x1D.

Ещё немного изучим разметку стека.

После того как управление передалось в main(), в стек было сохранено значение EBP. Затем для массива и переменной *i* было выделено 84 байта. Это (20+1)*sizeof(int). ESP сейчас указывает на переменную *_i* в локальном стеке и при исполнении следующего PUSH что-либо, что-либо появится рядом с *_i*.

Вот так выглядит разметка стека пока управление находится внутри main():

ESP	4 байта выделенных для переменной <i>i</i>
ESP+4	80 байт выделенных для массива <i>a[20]</i>
ESP+84	сохраненное значение EBP
ESP+88	адрес возврата

Выражение *a[19]=что_нибудь* записывает последний *int* в пределах массива (пока что в пределах!)

Выражение *a[20]=что_нибудь* записывает *что_нибудь* на место где сохранено значение EBP.

Обратите внимание на состояние регистров на момент падения процесса. В нашем случае в 20-й элемент записалось значение 20. И вот всё дело в том, что заканчиваясь, эпилог функции восстанавливал значение EBP (20 в десятичной системе это как раз 0x14 в шестнадцатеричной). Далее выполнялась инструкция RET, которая на самом деле эквивалентна POP EIP.

19.2. ПЕРЕПОЛНЕНИЕ БУФЕРА

Инструкция `RET` вытащила из стека адрес возврата (это адрес где-то внутри `CRT`), которая вызвала `main()`, а там было записано 21 в десятичной системе, то есть 0x15 в шестнадцатеричной. И вот процессор оказался по адресу 0x15, но исполняемого кода там нет, так что случилось исключение.

Добро пожаловать! Это называется *buffer overflow*³.

Замените массив `int` на строку (массив `char`), нарочно создайте слишком длинную строку, передайте её в ту программу, в ту функцию, которая не проверяя длину строки скопирует её в слишком короткий буфер, и вы сможете указать программе, по какому именно адресу перейти. Не всё так просто в реальности, конечно, но началось всё с этого. Классическая статья об этом: [Aleph One, *Smashing The Stack For Fun And Profit*, (1996)]⁴.

GCC

Попробуем то же самое в GCC 4.4.1. У нас выходит такое:

```
main      public main
          proc near
a          = dword ptr -54h
i          = dword ptr -4

          push    ebp
          mov     ebp, esp
          sub     esp, 60h ; 96
          mov     [ebp+i], 0
          jmp     short loc_80483D1

loc_80483C3:
          mov     eax, [ebp+i]
          mov     edx, [ebp+i]
          mov     [ebp+eax*4+a], edx
          add     [ebp+i], 1

loc_80483D1:
          cmp     [ebp+i], 1Dh
          jle     short loc_80483C3
          mov     eax, 0
          leave
          retn
main      endp
```

Запуск этого в Linux выдаст: `Segmentation fault`.

Если запустить полученное в отладчике GDB, получим:

```
(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax                0x0          0
ecx                0xd2f96388   -755407992
edx                0x1d         29
ebx                0x26eff4     2551796
esp                0xbffff4b0   0xbffff4b0
ebp                0x15         0x15
esi                0x0          0
edi                0x0          0
eip                0x16         0x16
eflags             0x10202     [ IF RF ]
cs                 0x73         115
ss                 0x7b         123
ds                 0x7b         123
es                 0x7b         123
fs                 0x0          0
gs                 0x33         51
(gdb)
```

³[wikipedia](#)

⁴Также доступно здесь: <http://go.yurichev.com/17266>

Значения регистров немного другие, чем в примере win32, потому что разметка стека чуть другая.

19.3. Защита от переполнения буфера

В наше время пытаются бороться с переполнением буфера невзирая на халатность программистов на Си/Си++. В MSVC есть опции вроде⁵:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

Одним из методов является вставка в прологе функции некоего случайного значения в область локальных переменных и проверка этого значения в эпилоге функции перед выходом. Если проверка не прошла, то не выполнять инструкцию RET, а остановиться (или зависнуть). Процесс зависнет, но это лучше, чем удаленная атака на ваш компьютер.

Это случайное значение иногда называют «канарейкой»⁶, по аналогии с шахтной канарейкой⁷. Раньше использовали шахтеры, чтобы определять, есть ли в шахте опасный газ. Канарейки очень к нему чувствительны и либо проявляли сильное беспокойство, либо гибли от газа.

Если скомпилировать наш простейший пример работы с массивом (19.1 (стр. 264)) в MSVC с опцией RTC1 или RTCs, в конце нашей функции будет вызов функции `@_RTC_CheckStackVars@8`, проверяющей корректность «канарейки».

Посмотрим, как дела обстоят в GCC. Возьмем пример из секции про `alloca()` (6.2.4 (стр. 29)):

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

По умолчанию, без дополнительных ключей, GCC 4.7.3 вставит в код проверку «канарейки»:

Листинг 19.7: GCC 4.7.3

```
.LC0:
.string "hi! %d, %d, %d\n"

f:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 676
    lea     ebx, [esp+39]
    and     ebx, -16
    mov     DWORD PTR [esp+20], 3
    mov     DWORD PTR [esp+16], 2
    mov     DWORD PTR [esp+12], 1
    mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
    mov     DWORD PTR [esp+4], 600
    mov     DWORD PTR [esp], ebx
    mov     eax, DWORD PTR gs:20 ; канарейка
    mov     DWORD PTR [ebp-12], eax
    xor     eax, eax
    call    _snprintf
```

⁵описания защит, которые компилятор может вставлять в код: wikipedia.org/wiki/Buffer_overflow_protection

⁶«canary» в англоязычной литературе

⁷miningwiki.ru/wiki/Канарейка_в_шахте

19.3. ЗАЩИТА ОТ ПЕРЕПОЛНЕНИЯ БУФЕРА

```
mov    DWORD PTR [esp], ebx
call   puts
mov    eax, DWORD PTR [ebp-12]
xor    eax, DWORD PTR gs:20                ; проверка канарейки
jne    .L5
mov    ebx, DWORD PTR [ebp-4]
leave
ret
.L5:
call   __stack_chk_fail
```

Случайное значение находится в `gs:20`. Оно записывается в стек, затем, в конце функции, значение в стеке сравнивается с корректной «канарейкой» в `gs:20`. Если значения не равны, будет вызвана функция `__stack_chk_fail` и в консоли мы увидим что-то вроде такого (Ubuntu 13.04 x86):

```
*** buffer overflow detected ***: ./2_1 terminated
===== Backtrace: =====
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
===== Memory map: =====
08048000-08049000 r-xp 00000000 08:01 2097586 /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586 /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586 /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0 [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781 /lib/i386-linux-gnu/libc-2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781 /lib/i386-linux-gnu/libc-2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781 /lib/i386-linux-gnu/libc-2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0 [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794 /lib/i386-linux-gnu/ld-2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794 /lib/i386-linux-gnu/ld-2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794 /lib/i386-linux-gnu/ld-2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0 [stack]
Aborted (core dumped)
```

`gs` это так называемый сегментный регистр. Эти регистры широко использовались во времена MS-DOS и DOS-экстендеров. Сейчас их функция немного изменилась. Если говорить кратко, в Linux `gs` всегда указывает на [TLS \(66 \(стр. 680\)\)](#) — там находится различная информация, специфичная для выполняющегося потока. Кстати, в win32 эту же роль играет сегментный регистр `fs`, он всегда указывает на [TIB^{8 9}](#).

Больше информации можно почерпнуть из исходных кодов Linux (по крайней мере, в версии 3.11): в файле `arch/x86/include/asm/s` в комментариях описывается эта переменная.

19.3.1. Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

Возвращаясь к нашему простому примеру (19.1 (стр. 264)), можно посмотреть, как LLVM добавит проверку «канарейки»:

```
_main
var_64      = -0x64
var_60      = -0x60
var_5C      = -0x5C
```

⁸Thread Information Block

⁹wikipedia.org/wiki/Win32_Thread_Information_Block

19.3. ЗАЩИТА ОТ ПЕРЕПОЛНЕНИЯ БУФЕРА

```

var_58      = -0x58
var_54      = -0x54
var_50      = -0x50
var_4C      = -0x4C
var_48      = -0x48
var_44      = -0x44
var_40      = -0x40
var_3C      = -0x3C
var_38      = -0x38
var_34      = -0x34
var_30      = -0x30
var_2C      = -0x2C
var_28      = -0x28
var_24      = -0x24
var_20      = -0x20
var_1C      = -0x1C
var_18      = -0x18
canary      = -0x14
var_10      = -0x10

PUSH        {R4-R7,LR}
ADD         R7, SP, #0xC
STR.W       R8, [SP,#0xC+var_10]!
SUB         SP, SP, #0x54
MOVW        R0, #a0bjc_methtype ; "objc_methtype"
MOVS        R2, #0
MOVT.W      R0, #0
MOVS        R5, #0
ADD         R0, PC
LDR.W       R8, [R0]
LDR.W       R0, [R8]
STR         R0, [SP,#0x64+canary]
MOVS        R0, #2
STR         R2, [SP,#0x64+var_64]
STR         R0, [SP,#0x64+var_60]
MOVS        R0, #4
STR         R0, [SP,#0x64+var_5C]
MOVS        R0, #6
STR         R0, [SP,#0x64+var_58]
MOVS        R0, #8
STR         R0, [SP,#0x64+var_54]
MOVS        R0, #0xA
STR         R0, [SP,#0x64+var_50]
MOVS        R0, #0xC
STR         R0, [SP,#0x64+var_4C]
MOVS        R0, #0xE
STR         R0, [SP,#0x64+var_48]
MOVS        R0, #0x10
STR         R0, [SP,#0x64+var_44]
MOVS        R0, #0x12
STR         R0, [SP,#0x64+var_40]
MOVS        R0, #0x14
STR         R0, [SP,#0x64+var_3C]
MOVS        R0, #0x16
STR         R0, [SP,#0x64+var_38]
MOVS        R0, #0x18
STR         R0, [SP,#0x64+var_34]
MOVS        R0, #0x1A
STR         R0, [SP,#0x64+var_30]
MOVS        R0, #0x1C
STR         R0, [SP,#0x64+var_2C]
MOVS        R0, #0x1E
STR         R0, [SP,#0x64+var_28]
MOVS        R0, #0x20
STR         R0, [SP,#0x64+var_24]
MOVS        R0, #0x22
STR         R0, [SP,#0x64+var_20]
MOVS        R0, #0x24
STR         R0, [SP,#0x64+var_1C]
MOVS        R0, #0x26

```

19.4. ЕЩЕ НЕМНОГО О МАССИВАХ

```
STR      R0, [SP,#0x64+var_18]
MOV      R4, 0xFDA ; "a[%d]=%d\n"
MOV      R0, SP
ADDS     R6, R0, #4
ADD      R4, PC
B        loc_2F1C

; начало второго цикла

loc_2F14
ADDS     R0, R5, #1
LDR.W    R2, [R6,R5,LSL#2]
MOV      R5, R0

loc_2F1C
MOV      R0, R4
MOV      R1, R5
BLX      _printf
CMP      R5, #0x13
BNE      loc_2F14
LDR.W    R0, [R8]
LDR      R1, [SP,#0x64+canary]
CMP      R0, R1
ITTTT EQ ; канарейка все еще верна?
MOVEQ    R0, #0
ADDEQ    SP, SP, #0x54
LDREQ.W  R8, [SP+0x64+var_64],#4
POPEQ    {R4-R7,PC}
BLX      ___stack_chk_fail
```

Во-первых, LLVM «развернул» цикл и все значения записываются в массив по одному, уже вычисленные, потому что LLVM посчитал что так будет быстрее. Кстати, инструкции режима ARM позволяют сделать это ещё быстрее и это может быть вашим домашним заданием.

В конце функции мы видим сравнение «канареек» — той что лежит в локальном стеке и корректной, на которую ссылается регистр `R8`. Если они равны, срабатывает блок из четырех инструкций при помощи `ITTTT EQ`. Это запись 0 в `R0`, эпилог функции и выход из нее. Если «канарейки» не равны, блок не срабатывает и происходит переход на функцию `___stack_chk_fail`, которая, вероятно, остановит работу программы.

19.4. Еще немного о массивах

Теперь понятно, почему нельзя написать в исходном коде на Си/Си++ что-то вроде:

```
void f(int size)
{
    int a[size];
    ...
};
```

Чтобы выделить место под массив в локальном стеке, компилятору нужно знать размер массива, чего он на стадии компиляции, разумеется, знать не может.

Если вам нужен массив произвольной длины, то выделите столько, сколько нужно, через `malloc()`, а затем обращайтесь к выделенному блоку байт как к массиву того типа, который вам нужен.

Либо используйте возможность стандарта C99 [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.5/2], и внутри это очень похоже на `alloca()` (6.2.4 (стр. 29)).

Для работы с памятью, можно также воспользоваться библиотекой сборщика мусора в Си. А для языка Си++ есть библиотеки с поддержкой умных указателей.

19.5. Массив указателей на строки

Вот пример массива указателей.


```
#include <stdio.h>

const char* month1[]=
{
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
};

// в пределах 0..11
const char* get_month1 (int month)
{
    return month1[month];
};
```

19.5.1. x64

Листинг 19.9: Оптимизирующий MSVC 2013 x64

```
_DATA SEGMENT
month1 DQ FLAT:$SG3122
        DQ FLAT:$SG3123
        DQ FLAT:$SG3124
        DQ FLAT:$SG3125
        DQ FLAT:$SG3126
        DQ FLAT:$SG3127
        DQ FLAT:$SG3128
        DQ FLAT:$SG3129
        DQ FLAT:$SG3130
        DQ FLAT:$SG3131
        DQ FLAT:$SG3132
        DQ FLAT:$SG3133
$SG3122 DB 'January', 00H
$SG3123 DB 'February', 00H
$SG3124 DB 'March', 00H
$SG3125 DB 'April', 00H
$SG3126 DB 'May', 00H
$SG3127 DB 'June', 00H
$SG3128 DB 'July', 00H
$SG3129 DB 'August', 00H
$SG3130 DB 'September', 00H
$SG3156 DB '%s', 0aH, 00H
$SG3131 DB 'October', 00H
$SG3132 DB 'November', 00H
$SG3133 DB 'December', 00H
_DATA ENDS

month$ = 8
get_month1 PROC
    movsxd rax, ecx
    lea rcx, OFFSET FLAT:month1
    mov rax, QWORD PTR [rcx+rax*8]
    ret 0
get_month1 ENDP
```

Код очень простой:

19.5. МАССИВ УКАЗАТЕЛЕЙ НА СТРОКИ

- Первая инструкция `MOVSXD` копирует 32-битное значение из `ECX` (где передается аргумент *month*) в `RAX` с знаковым расширением (потому что аргумент *month* имеет тип *int*). Причина расширения в том, что это значение будет использоваться в вычислениях наряду с другими 64-битными значениями. Таким образом, оно должно быть расширено до 64-битного¹⁰.
- Затем адрес таблицы указателей загружается в `RCX`.
- В конце концов, входное значение (*month*) умножается на 8 и прибавляется к адресу. Действительно: мы в 64-битной среде и все адреса (или указатели) требуют для хранения именно 64 бита (или 8 байт). Следовательно, каждый элемент таблицы имеет ширину в 8 байт. Вот почему для выбора элемента под нужным номером нужно пропустить *month* * 8 байт от начала. Это то, что делает `MOV`. Эта инструкция также загружает элемент по этому адресу. Для 1, элемент будет указателем на строку, содержащую «February», и т.д.

Оптимизирующий GCC 4.9 может это сделать даже лучше¹¹:

Листинг 19.10: Оптимизирующий GCC 4.9 x64

```
movsx    rdi, edi
mov      rax, QWORD PTR month1[0+rdi*8]
ret
```

32-bit MSVC

Скомпилируем также в 32-битном компиляторе MSVC:

Листинг 19.11: Оптимизирующий MSVC 2013 x86

```
_month$ = 8
_get_month1 PROC
    mov     eax, DWORD PTR _month$[esp-4]
    mov     eax, DWORD PTR _month1[eax*4]
    ret     0
_get_month1 ENDP
```

Входное значение не нужно расширять до 64-битного значения, так что оно используется как есть. И оно умножается на 4, потому что элементы таблицы имеют ширину 32 бита или 4 байта.

19.5.2. 32-БИТНЫЙ ARM

ARM в режиме ARM

Листинг 19.12: Оптимизирующий Keil 6/2013 (Режим ARM)

```
get_month1 PROC
    LDR     r1, |L0.100|
    LDR     r0, [r1, r0, LSL #2]
    BX     lr
ENDP

|L0.100|
DCD      ||.data||

DCB      "January", 0
DCB      "February", 0
DCB      "March", 0
DCB      "April", 0
DCB      "May", 0
DCB      "June", 0
DCB      "July", 0
DCB      "August", 0
DCB      "September", 0
DCB      "October", 0
DCB      "November", 0
```

¹⁰Это немного странная вещь, но отрицательный индекс массива может быть передан как *month* (отрицательные индексы массивов будут рассмотрены позже: 53 (стр. 589)). И если так будет, отрицательное значение типа *int* будет расширено со знаком корректно и соответствующий элемент перед таблицей будет выбран. Всё это не будет корректно работать без знакового расширения.

¹¹В листинге осталось «0+», потому что вывод ассемблера GCC не так скрупулёзен, чтобы убрать это. Это *displacement* и он здесь нулевой.

19.5. МАССИВ УКАЗАТЕЛЕЙ НА СТРОКИ

```
DCB      "December",0

AREA    ||.data||, DATA, ALIGN=2
month1
DCD      ||.conststring||
DCD      ||.conststring||+0x8
DCD      ||.conststring||+0x11
DCD      ||.conststring||+0x17
DCD      ||.conststring||+0x1d
DCD      ||.conststring||+0x21
DCD      ||.conststring||+0x26
DCD      ||.conststring||+0x2b
DCD      ||.conststring||+0x32
DCD      ||.conststring||+0x3c
DCD      ||.conststring||+0x44
DCD      ||.conststring||+0x4d
```

Адрес таблицы загружается в R1. Всё остальное делается, используя только одну инструкцию **LDR**. Входное значение *month* сдвигается влево на 2 (что тоже самое что и умножение на 4), это значение прибавляется к R1 (где находится адрес таблицы) и затем элемент таблицы загружается по этому адресу. 32-битный элемент таблицы загружается в R0 из таблицы.

ARM в режиме Thumb

Код почти такой же, только менее плотный, потому что здесь, в инструкции **LDR**, нельзя задать суффикс **LSL**:

```
get_month1 PROC
    LSLS    r0,r0,#2
    LDR     r1,|L0.64|
    LDR     r0,[r1,r0]
    BX      lr
ENDP
```

19.5.3. ARM64

Листинг 19.13: Оптимизирующий GCC 4.9 ARM64

```
get_month1:
    adrp    x1, .LANCHOR0
    add     x1, x1, :lo12: .LANCHOR0
    ldr     x0, [x1,w0,sxtw 3]
    ret

.LANCHOR0 = . + 0
.type     month1, %object
.size     month1, 96
month1:
    .xword  .LC2
    .xword  .LC3
    .xword  .LC4
    .xword  .LC5
    .xword  .LC6
    .xword  .LC7
    .xword  .LC8
    .xword  .LC9
    .xword  .LC10
    .xword  .LC11
    .xword  .LC12
    .xword  .LC13

.LC2:
    .string "January"
.LC3:
    .string "February"
.LC4:
    .string "March"
.LC5:
```

19.5. МАССИВ УКАЗАТЕЛЕЙ НА СТРОКИ

```
.LC6:      .string "April"
.LC7:      .string "May"
.LC8:      .string "June"
.LC9:      .string "July"
.LC10:     .string "August"
.LC11:     .string "September"
.LC12:     .string "October"
.LC13:     .string "November"
           .string "December"
```

Адрес таблицы загружается в X1 используя пару **ADRP / ADD**. Соответствующий элемент выбирается используя одну инструкцию **LDR**, которая берет W0 (регистр, где находится значение входного аргумента *month*), сдвигает его на 3 бита влево (что то же самое что и умножение на 8), расширяет его, учитывая знак (это то, что означает суффикс «sxtw») и прибавляет к X0. Затем 64-битное значение загружается из таблицы в X0.

19.5.4. MIPS

Листинг 19.14: Оптимизирующий GCC 4.4.5 (IDA)

```
get_month1:
; загрузить адрес таблицы в $v0:
        la      $v0, month1
; взять входное значение и умножить его на 4:
        sll     $a0, 2
; сложить адрес таблицы и умноженное значение:
        addu    $a0, $v0
; загрузить элемент таблицы по этому адресу в $v0:
        lw      $v0, 0($a0)
; возврат
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP

        .data # .data.rel.local
month1:  .globl month1
        .word aJanuary      # "January"
        .word aFebruary     # "February"
        .word aMarch        # "March"
        .word aApril        # "April"
        .word aMay          # "May"
        .word aJune         # "June"
        .word aJuly         # "July"
        .word aAugust       # "August"
        .word aSeptember    # "September"
        .word aOctober      # "October"
        .word aNovember     # "November"
        .word aDecember     # "December"

        .data # .rodata.str1.4
aJanuary: .ascii "January"<0>
aFebruary: .ascii "February"<0>
aMarch: .ascii "March"<0>
aApril: .ascii "April"<0>
aMay: .ascii "May"<0>
aJune: .ascii "June"<0>
aJuly: .ascii "July"<0>
aAugust: .ascii "August"<0>
aSeptember: .ascii "September"<0>
aOctober: .ascii "October"<0>
aNovember: .ascii "November"<0>
aDecember: .ascii "December"<0>
```

19.5.5. Переполнение массива

Наша функция принимает значения в пределах 0..11, но что будет, если будет передано 12? В таблице в этом месте нет элемента. Так что функция загрузит какое-то значение, которое волею случая находится там, и вернет его. Позже, какая-то другая функция попытается прочитать текстовую строку по этому адресу и, возможно, упадет.

Скомпилируем этот пример в MSVC для win64 и откроем его в IDA чтобы посмотреть, что линкер расположил после таблицы:

Листинг 19.15: Исполняемый файл в IDA

```

off_140011000    dq offset aJanuary_1    ; DATA XREF: .text:0000000140001003
                ; "January"
                dq offset aFebruary_1    ; "February"
                dq offset aMarch_1       ; "March"
                dq offset aApril_1       ; "April"
                dq offset aMay_1         ; "May"
                dq offset aJune_1        ; "June"
                dq offset aJuly_1        ; "July"
                dq offset aAugust_1      ; "August"
                dq offset aSeptember_1   ; "September"
                dq offset aOctober_1     ; "October"
                dq offset aNovember_1    ; "November"
                dq offset aDecember_1    ; "December"
aJanuary_1      db 'January',0          ; DATA XREF: sub_140001020+4
                ; .data:off_140011000
aFebruary_1     db 'February',0         ; DATA XREF: .data:0000000140011008
                align 4
aMarch_1        db 'March',0           ; DATA XREF: .data:0000000140011010
                align 4
aApril_1        db 'April',0           ; DATA XREF: .data:0000000140011018

```

Имена месяцев идут сразу после. Наша программа все-таки крошечная, так что здесь не так уж много данных (всего лишь названия месяцев) для расположения их в сегменте данных. Но нужно заметить, что там может быть действительно что удобно, что линкер решит там расположить, случайным образом.

Так что будет если 12 будет передано в функцию? Вернется 13-й элемент таблицы. Посмотрим, как CPU обходится с байтами как с 64-битным значением:

Листинг 19.16: Исполняемый файл в IDA

```

off_140011000    dq offset qword_140011060
                ; DATA XREF: .text:0000000140001003
                dq offset aFebruary_1    ; "February"
                dq offset aMarch_1       ; "March"
                dq offset aApril_1       ; "April"
                dq offset aMay_1         ; "May"
                dq offset aJune_1        ; "June"
                dq offset aJuly_1        ; "July"
                dq offset aAugust_1      ; "August"
                dq offset aSeptember_1   ; "September"
                dq offset aOctober_1     ; "October"
                dq offset aNovember_1    ; "November"
                dq offset aDecember_1    ; "December"
qword_140011060 dq 797261756E614Ah      ; DATA XREF: sub_140001020+4
                ; .data:off_140011000
aFebruary_1     db 'February',0         ; DATA XREF: .data:0000000140011008
                align 4
aMarch_1        db 'March',0           ; DATA XREF: .data:0000000140011010

```

И это 0x797261756E614A. После этого, какая-то другая функция (вероятно, работающая со строками) попытается загрузить байты по этому адресу, ожидая найти там Си-строку. И скорее всего упадет, потому что это значение не выглядит как действительный адрес.

Защита от переполнения массива

Если какая-нибудь неприятность может случиться,
она случается

Закон Мерфи

19.5. МАССИВ УКАЗАТЕЛЕЙ НА СТРОКИ

Немного наивно ожидать что всякий программист, кто будет использовать вашу функцию или библиотеку, никогда не передаст аргумент больше 11. Существует также хорошая философия «fail early and fail loudly» или «fail-fast», которая учит сообщать об ошибках как можно раньше и останавливаться. Один из таких методов в Си/Си++ это макрос `assert()`. Мы можем немного изменить нашу программу, чтобы она падала при передаче неверного значения:

Листинг 19.17: `assert()` добавлен

```
const char* get_month1_checked (int month)
{
    assert (month<12);
    return month1[month];
};
```

Макрос будет проверять на верные значения во время каждого старта функции и падать если выражение возвращает `false`.

Листинг 19.18: Оптимизирующий MSVC 2013 x64

```
$SG3143 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
        DB      'c', 00H, 00H, 00H
$SG3144 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H
        DB      '1', 00H, '2', 00H, 00H, 00H

month$ = 48
get_month1_checked PROC
$LN5:
    push     rbx
    sub     rsp, 32
    movsxd  rbx, ecx
    cmp     ebx, 12
    jl      SHORT $LN3@get_month1
    lea     rdx, OFFSET FLAT:$SG3143
    lea     rcx, OFFSET FLAT:$SG3144
    mov     r8d, 29
    call    _wassert
$LN3@get_month1:
    lea     rcx, OFFSET FLAT:month1
    mov     rax, QWORD PTR [rcx+rbx*8]
    add     rsp, 32
    pop     rbx
    ret     0
get_month1_checked ENDP
```

На самом деле, `assert()` это не функция, а макрос. Он проверяет условие и передает также номер строки и название файла в другую функцию, которая покажет эту информацию пользователю. Мы видим, что здесь и имя файла и выражение закодировано в UTF-16. Номер строки также передается (это 29).

Этот механизм, пожалуй, одинаковый во всех компиляторах. Вот что делает GCC:

Листинг 19.19: Оптимизирующий GCC 4.9 x64

```
.LC1:
    .string "month.c"
.LC2:
    .string "month<12"

get_month1_checked:
    cmp     edi, 11
    jg      .L6
    movsx   rdi, edi
    mov     rax, QWORD PTR month1[0+rdi*8]
    ret

.L6:
    push    rax
    mov     ecx, OFFSET FLAT:__PRETTY_FUNCTION__.2423
    mov     edx, 29
    mov     esi, OFFSET FLAT:.LC1
    mov     edi, OFFSET FLAT:.LC2
    call    __assert_fail

__PRETTY_FUNCTION__.2423:
    .string "get_month1_checked"
```

Так что макрос в GCC также передает и имя функции, для удобства.

Ничего не бывает бесплатным и проверки на корректность тоже. Это может замедлить работу вашей программы, особенно если макрос `assert()` используется в маленькой критичной ко времени функции. Так что, например, MSVC оставляет проверки в отладочных сборках, но в окончательных сборках они исчезают.

Ядра Microsoft **Windows NT** также идут в виде сборок «checked» и «free»¹². В первых есть проверки на корректность аргументов (отсюда «checked»), а во вторых — нет (отсюда «free», т.е. «свободные» от проверок).

Разумеется, «checked»-ядро работает медленнее из-за всех этих проверок, поэтому его обычно используют только на время отладки драйверов, либо самого ядра.

19.6. Многомерные массивы

Внутри многомерный массив выглядит так же как и линейный. Ведь память компьютера линейная, это одномерный массив. Но для удобства этот одномерный массив легко представить как многомерный.

К примеру, вот как элементы массива 3x4 расположены в одномерном массиве из 12 ячеек:

Смещение в памяти	элемент массива
0	[0][0]
1	[0][1]
2	[0][2]
3	[0][3]
4	[1][0]
5	[1][1]
6	[1][2]
7	[1][3]
8	[2][0]
9	[2][1]
10	[2][2]
11	[2][3]

Таблица 19.1: Двухмерный массив представляется в памяти как одномерный

Вот по каким адресам в памяти располагается каждая ячейка двумерного массива 3*4:

0	1	2	3
4	5	6	7
8	9	10	11

Таблица 19.2: Адреса в памяти каждой ячейки двумерного массива

Чтобы вычислить адрес нужного элемента, сначала умножаем первый индекс (строку) на 4 (ширину массива), затем прибавляем второй индекс (столбец). Это называется *row-major order*, и такой способ представления массивов и матриц используется по крайней мере в Си/Си++ и Python. Термин *row-major order* означает по-русски примерно следующее: «сначала записываем элементы первой строки, затем второй, ... и элементы последней строки в самом конце».

Другой способ представления называется *column-major order* (индексы массива используются в обратном порядке) и это используется по крайней мере в FORTRAN, MATLAB и R. Термин *column-major order* означает по-русски следующее: «сначала записываем элементы первого столбца, затем второго, ... и элементы последнего столбца в самом конце».

Какой из способов лучше? В терминах производительности и кэш-памяти, лучший метод организации данных это тот, при котором к данным обращаются последовательно. Так что если ваша функция обращается к данным построчно, то *row-major order* лучше, и наоборот.

19.6.1. Пример с двумерным массивом

Мы будем работать с массивом типа *char*. Это значит, что каждый элемент требует только одного байта в памяти.

¹²[msdn.microsoft.com/en-us/library/windows/hardware/ff543450\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff543450(v=vs.85).aspx)

Пример с заполнением строки

Заполняем вторую строку значениями 0..3:

Листинг 19.20: Пример с заполнением строки

```
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // очистить массив
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // заполнить вторую строку значениями 0..3:
    for (y=0; y<4; y++)
        a[1][y]=y;
};
```

Все три строки обведены красным. Видно, что во второй теперь имеются байты 0, 1, 2 и 3:

Address	Hex dump															
00C33370	00	00	00	00	00	01	02	03	00	00	00	00	00	00	00	00
00C33380	02	00	00	00	C3	66	47	4E	C3	66	47	4E	00	00	00	00
00C33390	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00C333A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00C333B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Рис. 19.7: OllyDbg: массив заполнен

Пример с заполнением столбца

Заполняем третий столбец значениями 0..2:

Листинг 19.21: Пример с заполнением столбца

```
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // очистить массив
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // заполнить третий столбец значениями 0..2:
    for (x=0; x<3; x++)
        a[x][2]=x;
};
```

Здесь также обведены красным три строки. Видно, что в каждой строке, на третьей позиции, теперь записаны 0, 1 и 2.

Address	Hex dump
01033380	00 00 00 00 00 00 01 00 00 00 02 00 02 00 00 00
01033390	00 00 00 00 00 1E 44 EF 31 1E 44 EF 31 00 00 00 00
010333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Рис. 19.8: OllyDbg: массив заполнен

19.6.2. Работа с двухмерным массивом как с одномерным

Мы можем легко убедиться, что можно работать с двухмерным массивом как с одномерным, используя по крайней мере два метода:

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
    return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
    // обращаться с входным массивом как с одномерным
    // 4 здесь это ширина массива
    return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
    // обращаться с входным массивом как с указателем,
    // вычислить адрес, получить значение оттуда
    // 4 здесь это ширина массива
    return *(array+a*4+b);
};

int main()
{
    a[2][3]=123;
    printf ("%d\n", get_by_coordinates1(a, 2, 3));
    printf ("%d\n", get_by_coordinates2(a, 2, 3));
    printf ("%d\n", get_by_coordinates3(a, 2, 3));
};
```

Компилируете и запускаете: мы увидим корректные значения.

Очарователен результат работы MSVC 2013 – все три процедуры одинаковые!

Листинг 19.22: Оптимизирующий MSVC 2013 x64

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=адрес массива
; RDX=a
; R8=b
    movsxd    rax, r8d
; EAX=b
    movsxd    r9, edx
; R9=a
    add       rax, rcx
; RAX=b+адрес массива
    movzx     eax, BYTE PTR [rax+r9*4]
; AL=загрузить байт по адресу RAX+R9*4=b+адрес массива+a*4=адрес массива+a*4+b
    ret       0
get_by_coordinates3 ENDP
```

```

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
    movsxd    rax, r8d
    movsxd    r9, edx
    add       rax, rcx
    movzx     eax, BYTE PTR [rax+r9*4]
    ret       0
get_by_coordinates2 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
    movsxd    rax, r8d
    movsxd    r9, edx
    add       rax, rcx
    movzx     eax, BYTE PTR [rax+r9*4]
    ret       0
get_by_coordinates1 ENDP

```

GCC сгенерировал практически одинаковые процедуры:

Листинг 19.23: Оптимизирующий GCC 4.9 x64

```

; RDI=адрес массива
; RSI=a
; RDX=b

get_by_coordinates1:
; расширить входные 32-битные значения "a" и "b" до 64-битных
    movsx    rsi, esi
    movsx    rdx, edx
    lea      rax, [rdi+rsi*4]
; RAX=RDI+RSI*4=адрес массива+a*4
    movzx    eax, BYTE PTR [rax+rdx]
; AL=загрузить байт по адресу RAX+RDX=адрес массива+a*4+b
    ret

get_by_coordinates2:
    lea      eax, [rdx+rsi*4]
; RAX=RDX+RSI*4=b+a*4
    cdqe
    movzx    eax, BYTE PTR [rdi+rax]
; AL=загрузить байт по адресу RDI+RAX=адрес массива+b+a*4
    ret

get_by_coordinates3:
    sal      esi, 2
; ESI=a<<2=a*4
; расширить входные 32-битные значения "a*4" и "b" до 64-битных
    movsx    rdx, edx
    movsx    rsi, esi
    add      rdi, rsi
; RDI=RDI+RSI=адрес массива+a*4
    movzx    eax, BYTE PTR [rdi+rdx]
; AL=загрузить байт по адресу RDI+RDX=адрес массива+a*4+b
    ret

```

19.6.3. Пример с трехмерным массивом

То же самое и для многомерных массивов. На этот раз будем работать с массивом типа *int*: каждый элемент требует 4 байта в памяти.

Попробуем:

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
    a[x][y][z]=value;
};
```

x86

В итоге (MSVC 2010):

Листинг 19.25: MSVC 2010

```
_DATA    SEGMENT
COMM     _a:DWORD:01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8           ; size = 4
_y$ = 12          ; size = 4
_z$ = 16          ; size = 4
_value$ = 20      ; size = 4
_insert   PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _x$[ebp]
    imul    eax, 2400           ; eax=600*4*x
    mov     ecx, DWORD PTR _y$[ebp]
    imul    ecx, 120           ; ecx=30*4*y
    lea     edx, DWORD PTR _a[ecx+eax] ; edx=a + 600*4*x + 30*4*y
    mov     eax, DWORD PTR _z$[ebp]
    mov     ecx, DWORD PTR _value$[ebp]
    mov     DWORD PTR [edx+eax*4], ecx ; *(edx+z*4)=значение
    pop     ebp
    ret     0
_insert   ENDP
_TEXT    ENDS
```

В принципе, ничего удивительного. В `insert()` для вычисления адреса нужного элемента массива три входных аргумента перемножаются по формуле $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$, чтобы представить массив трехмерным. Не забывайте также, что тип `int` 32-битный (4 байта), поэтому все коэффициенты нужно умножить на 4.

Листинг 19.26: GCC 4.4.1

```
insert      public insert
            proc near

x            = dword ptr 8
y            = dword ptr 0Ch
z            = dword ptr 10h
value       = dword ptr 14h

            push    ebp
            mov     ebp, esp
            push    ebx
            mov     ebx, [ebp+x]
            mov     eax, [ebp+y]
            mov     ecx, [ebp+z]
            lea     edx, [eax+eax]           ; edx=y*2
            mov     eax, edx                ; eax=y*2
            shl     eax, 4                   ; eax=(y*2)<<4 = y*2*16 = y*32
            sub     eax, edx                ; eax=y*32 - y*2=y*30
            imul    edx, ebx, 600           ; edx=x*600
            add     eax, edx                ; eax=eax+edx=y*30 + x*600
            lea     edx, [eax+ecx]          ; edx=y*30 + x*600 + z
```

19.6. МНОГОМЕРНЫЕ МАССИВЫ

```
mov     eax, [ebp+value]
mov     dword ptr ds:a[edx*4], eax ; *(a+edx*4)=значение
pop     ebx
pop     ebp
retn
insert  endp
```

Компилятор GCC решил всё сделать немного иначе. Для вычисления одной из операций ($30y$), GCC создал код, где нет самой операции умножения. Происходит это так: $(y + y) \ll 4 - (y + y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$. Таким образом, для вычисления $30y$ используется только операция сложения, операция битового сдвига и операция вычитания. Это работает быстрее.

ARM + Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

Листинг 19.27: Неоптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

```
_insert
value  = -0x10
z      = -0xC
y      = -8
x      = -4

; выделить место в локальном стеке для 4 переменных типа int
SUB     SP, SP, #0x10
MOV     R9, 0xFC2 ; a
ADD     R9, PC
LDR.W   R9, [R9]
STR     R0, [SP,#0x10+x]
STR     R1, [SP,#0x10+y]
STR     R2, [SP,#0x10+z]
STR     R3, [SP,#0x10+value]
LDR     R0, [SP,#0x10+value]
LDR     R1, [SP,#0x10+z]
LDR     R2, [SP,#0x10+y]
LDR     R3, [SP,#0x10+x]
MOV     R12, 2400
MUL.W   R3, R3, R12
ADD     R3, R9
MOV     R9, 120
MUL.W   R2, R2, R9
ADD     R2, R3
LSLS    R1, R1, #2 ; R1=R1<<2
ADD     R1, R2
STR     R0, [R1] ; R1 – адрес элемента массива
; освободить блок в локальном стеке, выделенное для 4 переменных
ADD     SP, SP, #0x10
BX      LR
```

Неоптимизирующий LLVM сохраняет все переменные в локальном стеке, хотя это и избыточно. Адрес элемента массива вычисляется по уже рассмотренной формуле.

ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

Листинг 19.28: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb)

```
_insert
MOVW    R9, #0x10FC
MOV.W   R12, #2400
MOVT.W  R9, #0
RSB.W   R1, R1, R1, LSL#4 ; R1 = y. R1=y<<4 - y = y*16 - y = y*15
ADD     R9, PC ; R9 = указатель на массив
LDR.W   R9, [R9]
MLA.W   R0, R0, R12, R9 ; R0 = x, R12 = 2400, R9 = указатель на а. R0=x*2400 + указатель на а
ADD.W   R0, R0, R1, LSL#3 ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + указатель на а + y*15*8 =
; указатель на а + y*30*4 + x*600*4
STR.W   R3, [R0,R2,LSL#2] ; R2 = z, R3 = значение. адрес=R0+z*4 =
```

BX	LR	; указатель на $a + y*30*4 + x*600*4 + z*4$

Тут используются уже описанные трюки для замены умножения на операции сдвига, сложения и вычитания.

Также мы видим новую для себя инструкцию **RSB** (*Reverse Subtract*). Она работает так же, как и **SUB**, только меняет операнды местами. Зачем? **SUB** и **RSB** это те инструкции, ко второму операнду которых можно применить коэффициент сдвига, как мы видим и здесь: (**LSL#4**). Но этот коэффициент можно применить только ко второму операнду. Для коммутативных операций, таких как сложение или умножение, операнды можно менять местами и это не влияет на результат. Но вычитание — операция некоммутативная, так что для этих случаев существует инструкция **RSB**.

Инструкция **LDR.W R9, [R9]** работает как **LEA** (A.6.2 (стр. 917)) в x86, и здесь она ничего не делает, она избыточна. Вероятно, компилятор не оптимизировал её.

MIPS

Мой пример такой крошечный, что компилятор GCC решил разместить массив *a* в 64KiB-области, адресуемой при помощи Global Pointer.

Листинг 19.29: Оптимизирующий GCC 4.4.5 (IDA)

```
insert:
; $a0=x
; $a1=y
; $a2=z
; $a3=значение
        sll      $v0, $a0, 5
; $v0 = $a0<<5 = x*32
        sll      $a0, 3
; $a0 = $a0<<3 = x*8
        addu     $a0, $v0
; $a0 = $a0+$v0 = x*8+x*32 = x*40
        sll      $v1, $a1, 5
; $v1 = $a1<<5 = y*32
        sll      $v0, $a0, 4
; $v0 = $a0<<4 = x*40*16 = x*640
        sll      $a1, 1
; $a1 = $a1<<1 = y*2
        subu     $a1, $v1, $a1
; $a1 = $v1-$a1 = y*32-y*2 = y*30
        subu     $a0, $v0, $a0
; $a0 = $v0-$a0 = x*640-x*40 = x*600
        la       $gp, __gnu_local_gp
        addu     $a0, $a1, $a0
; $a0 = $a1+$a0 = y*30+x*600
        addu     $a0, $a2
; $a0 = $a0+$a2 = y*30+x*600+z
; загрузить адрес таблицы:
        lw       $v0, (a & 0xFFFF)($gp)
; умножить индекс на 4 для поиска элемента таблицы:
        sll      $a0, 2
; сложить умноженный индекс и адрес таблицы:
        addu     $a0, $v0, $a0
; записать значение в таблицу и вернуть управление:
        jr       $ra
        sw       $a3, 0($a0)

        .comm a:0x1770
```

19.6.4. Ещё примеры

Компьютерный экран представляет собой двумерный массив, но видеобuffer это линейный одномерный массив. Мы рассматриваем это здесь: 83.2 (стр. 831).

Еще один пример в этой книге это игра “Сапер”: её поле это тоже двухмерный массив: 77.

19.7. Набор строк как двумерный массив

Снова вернемся к примеру, который возвращает название месяца: листинг.19.8. Как видно, нужна как минимум одна операция загрузки из памяти для подготовки указателя на строку, состоящую из имени месяца. Возможно ли избавиться от операции загрузки из памяти? Да, если представить список строк как двумерный массив:

```
#include <stdio.h>
#include <assert.h>

const char month2[12][10]=
{
    { 'J','a','n','u','a','r','y', 0, 0, 0 },
    { 'F','e','b','r','u','a','r','y', 0, 0 },
    { 'M','a','r','c','h', 0, 0, 0, 0, 0 },
    { 'A','p','r','i','l', 0, 0, 0, 0, 0 },
    { 'M','a','y', 0, 0, 0, 0, 0, 0 },
    { 'J','u','n','e', 0, 0, 0, 0, 0, 0 },
    { 'J','u','l','y', 0, 0, 0, 0, 0, 0 },
    { 'A','u','g','u','s','t', 0, 0, 0, 0, 0 },
    { 'S','e','p','t','e','m','b','e','r', 0 },
    { 'O','c','t','o','b','e','r', 0, 0, 0 },
    { 'N','o','v','e','m','b','e','r', 0, 0 },
    { 'D','e','c','e','m','b','e','r', 0, 0 }
};

// в пределах 0..11
const char* get_month2 (int month)
{
    return &month2[month][0];
};
```

Вот что получаем:

Листинг 19.30: Оптимизирующий MSVC 2013 x64

```
month2  DB      04aH
        DB      061H
        DB      06eH
        DB      075H
        DB      061H
        DB      072H
        DB      079H
        DB      00H
        DB      00H
        DB      00H
        ...

get_month2 PROC
; расширить входное значение до 64-битного, учитывая знак
        movsxd  rax, ecx
        lea     rcx, QWORD PTR [rax+rax*4]
; RCX=месяц*4=месяц*5
        lea     rax, OFFSET FLAT:month2
; RAX=указатель на таблицу
        lea     rax, QWORD PTR [rax+rcx*2]
; RAX=указатель на таблицу + RCX*2=указатель на таблицу + месяц*5*2=указатель на таблицу + месяц*10
        ret     0
get_month2 ENDP
```

Здесь нет обращений к памяти вообще. Эта функция только вычисляет место, где находится первый символ названия месяца: *pointer_to_the_table + month * 10*. Там также две инструкции **LEA**, которые работают как несколько инструкций **MUL** и **MOV**.

Ширина массива — 10 байт. Действительно, самая длинная строка это «September» (9 байт) плюс оканчивающий ноль, получается 10 байт. Остальные названия месяцев дополнены нулевыми байтами, чтобы они занимали столько же места (10 байт). Таким образом, наша функция и работает быстрее, потому что все строки начинаются с тех адресов, которые легко вычислить.

Оптимизирующий GCC 4.9 может ещё короче:

```

movsx    rdi, edi
lea      rax, [rdi+rdi*4]
lea      rax, month2[rax+rax]
ret

```

LEA здесь также используется для умножения на 10.

Неоптимизирующие компиляторы делают умножение по-разному.

```

get_month2:
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    movsx   rdx, eax
; RDX = входное значение, расширенное учитывая знак
    mov     rax, rdx
; RAX = месяц
    sal     rax, 2
; RAX = месяц<<2 = месяц*4
    add     rax, rdx
; RAX = RAX+RDX = месяц*4+месяц = месяц*5
    add     rax, rax
; RAX = RAX*2 = месяц*5*2 = месяц*10
    add     rax, OFFSET FLAT:month2
; RAX = месяц*10 + указатель на таблицу
    pop     rbp
    ret

```

Неоптимизирующий MSVC просто использует инструкцию **IMUL** :

```

month$ = 8
get_month2 PROC
    mov     DWORD PTR [rsp+8], ecx
    movsxd  rax, DWORD PTR month$[rsp]
; RAX = расширенное до 64-битного входное значение, учитывая знак
    imul    rax, rax, 10
; RAX = RAX*10
    lea     rcx, OFFSET FLAT:month2
; RCX = указатель на таблицу
    add     rcx, rax
; RCX = RCX+RAX = указатель на таблицу+month*10
    mov     rax, rcx
; RAX = указатель на таблицу+месяц*10
    mov     ecx, 1
; RCX = 1
    imul    rcx, rcx, 0
; RCX = 1*0 = 0
    add     rax, rcx
; RAX = указатель на таблицу+месяц*10 + 0 = указатель на таблицу+месяц*10
    ret     0
get_month2 ENDP

```

Но вот что странно: зачем добавлять умножение на ноль и добавлять ноль к конечному результату? Это выглядит как странность кодегенератора компилятора, который не был покрыт тестами компилятора. Но так или иначе, итоговый код работает корректно. Мы сознательно рассматриваем такие фрагменты кода, чтобы читатель понимал, что иногда не нужно ломать себе голову над подобными артефактами компиляторов.

19.7.1. 32-bit ARM

Оптимизирующий Keil для режима Thumb использует инструкцию умножения **MULS** :

Листинг 19.34: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

; R0 = месяц
    MOVS    r1,#0ха
; R1 = 10
    MULS    r0,r1,r0
; R0 = R1*R0 = 10*месяц
    LDR     r1,|L0.68|
; R1 = указатель на таблицу
    ADDS    r0,r0,r1
; R0 = R0+R1 = 10*месяц + указатель на таблицу
    BX     lr

```

Оптимизирующий Keil для режима ARM использует операции сложения и сдвига:

Листинг 19.35: Оптимизирующий Keil 6/2013 (Режим ARM)

```

; R0 = месяц
    LDR     r1,|L0.104|
; R1 = указатель на таблицу
    ADD     r0,r0,r0,LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = месяц*5
    ADD     r0,r1,r0,LSL #1
; R0 = R1+R0<<2 = указатель на таблицу + месяц*5*2 = указатель на таблицу + месяц*10
    BX     lr

```

19.7.2. ARM64

Листинг 19.36: Оптимизирующий GCC 4.9 ARM64

```

; W0 = месяц
    sxtw    x0, w0
; X0 = расширить входное значение учитывая знак
    adrp    x1, .ANCHOR1
    add     x1, x1, :lo12:.ANCHOR1
; X1 = указатель на таблицу
    add     x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
    add     x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = указатель на таблицу + X0*10
    ret

```

SXTW используется для знакового расширения и расширения входного 32-битного значения в 64-битное и сохранения его в X0. Пара **ADRP / ADD** используется для загрузки адреса таблицы. У инструкции **ADD** также есть суффикс **LSL**, что помогает с умножением.

19.7.3. MIPS

Листинг 19.37: Оптимизирующий GCC 4.4.5 (IDA)

```

        .globl get_month2
get_month2:
; $a0=месяц
    sll     $v0, $a0, 3
; $v0 = $a0<<3 = месяц*8
    sll     $a0, 1
; $a0 = $a0<<1 = месяц*2
    addu    $a0, $v0
; $a0 = месяц*2+месяц*8 = месяц*10
; загрузить адрес таблицы:
    la      $v0, month2
; сложить адрес таблицы и вычисленный индекс и вернуть управление:
    jr      $ra
    addu    $v0, $a0

month2:
    .ascii "January"<0>
    .byte 0, 0

```


19.8. Вывод

```
aFebruary:  .ascii "February"<0>
              .byte   0
aMarch:      .ascii "March"<0>
              .byte 0, 0, 0, 0
aApril:      .ascii "April"<0>
              .byte 0, 0, 0, 0
aMay:        .ascii "May"<0>
              .byte 0, 0, 0, 0, 0, 0
aJune:       .ascii "June"<0>
              .byte 0, 0, 0, 0, 0
aJuly:       .ascii "July"<0>
              .byte 0, 0, 0, 0, 0
aAugust:     .ascii "August"<0>
              .byte 0, 0, 0
aSeptember:  .ascii "September"<0>
aOctober:    .ascii "October"<0>
              .byte 0, 0
aNovember:   .ascii "November"<0>
              .byte   0
aDecember:   .ascii "December"<0>
              .byte 0, 0, 0, 0, 0, 0, 0, 0
```

19.7.4. Вывод

Это немного старинная техника для хранения текстовых строк. Много такого можно найти в Oracle RDBMS, например. Трудно сказать, стоит ли оно того на современных компьютерах. Так или иначе, это был хороший пример массивов, поэтому он был добавлен в эту книгу.

19.8. Вывод

Массив это просто набор значений в памяти, расположенных рядом друг с другом. Это справедливо для любых типов элементов, включая структуры. Доступ к определенному элементу массива это просто вычисление его адреса.

19.9. Упражнения

- <http://challenges.re/62>
- <http://challenges.re/63>
- <http://challenges.re/64>
- <http://challenges.re/65>
- <http://challenges.re/66>

Глава 20

Работа с отдельными битами

Немало функций задают различные флаги в аргументах при помощи битовых полей¹.

Наверное, вместо этого можно было бы использовать набор переменных типа *bool*, но это было бы не очень экономно.

20.1. Проверка какого-либо бита

20.1.1. x86

Например в Win32 API:

```
HANDLE fh;

fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_ALWAYS, ↵
FILE_ATTRIBUTE_NORMAL, NULL);
```

Получаем (MSVC 2010):

Листинг 20.1: MSVC 2010

```
push    0
push    128                                ; 00000080H
push    4
push    0
push    1
push    -1073741824                       ; c0000000H
push    OFFSET $SG78813
call    DWORD PTR __imp__CreateFileA@28
mov     DWORD PTR _fh$[ebp], eax
```

Заглянем в файл WinNT.h:

Листинг 20.2: WinNT.h

```
#define GENERIC_READ      (0x80000000L)
#define GENERIC_WRITE     (0x40000000L)
#define GENERIC_EXECUTE   (0x20000000L)
#define GENERIC_ALL       (0x10000000L)
```

Всё ясно, `GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000`, и это значение используется как второй аргумент для функции `CreateFile()`².

Как `CreateFile()` будет проверять флаги? Заглянем в KERNEL32.DLL от Windows XP SP3 x86 и найдем в функции `CreateFileW()` в том числе и такой фрагмент кода:

Листинг 20.3: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429      test     byte ptr [ebp+dwDesiredAccess+3], 40h
```

¹bit fields в англоязычной литературе

²[msdn.microsoft.com/en-us/library/aa363858\(VS.85\).aspx](https://msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx)

20.1. ПРОВЕРКА КАКОГО-ЛИБО БИТА

```
.text:7C83D42D      mov     [ebp+var_8], 1
.text:7C83D434      jz      short loc_7C83D417
.text:7C83D436      jmp     loc_7C810817
```

Здесь мы видим инструкцию `TEST`. Впрочем, она берет не весь второй аргумент функции, а только его самый старший байт (`ebp+dwDesiredAccess+3`) и проверяет его на флаг `0x40` (имеется ввиду флаг `GENERIC_WRITE`).

`TEST` это то же что и `AND`, только без сохранения результата (вспомните что `CMP` это то же что и `SUB`, только без сохранения результатов (8.3.1 (стр. 82))).

Логика данного фрагмента кода примерно такая:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

Если после операции `AND` останется этот бит, то флаг `ZF` не будет поднят и условный переход `JZ` не сработает. Переход возможен, только если в переменной `dwDesiredAccess` отсутствует бит `0x40000000` – тогда результат `AND` будет 0, флаг `ZF` будет поднят и переход сработает.

Попробуем GCC 4.4.1 и Linux:

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
    int handle;

    handle=open ("file", O_RDWR | O_CREAT);
};
```

Получим:

Листинг 20.4: GCC 4.4.1

```
main      public main
          proc near

var_20    = dword ptr -20h
var_1C    = dword ptr -1Ch
var_4     = dword ptr -4

          push    ebp
          mov     ebp, esp
          and     esp, 0FFFFFFF0h
          sub     esp, 20h
          mov     [esp+20h+var_1C], 42h
          mov     [esp+20h+var_20], offset aFile ; "file"
          call    _open
          mov     [esp+20h+var_4], eax
          leave
          retn
main      endp
```

Заглянем в реализацию функции `open()` в библиотеке `libc.so.6`, но обнаружим что там только системный вызов:

Листинг 20.5: `open()` (libc.so.6)

```
.text:000BE69B      mov     edx, [esp+4+mode] ; mode
.text:000BE69F      mov     ecx, [esp+4+flags] ; flags
.text:000BE6A3      mov     ebx, [esp+4+filename] ; filename
.text:000BE6A7      mov     eax, 5
.text:000BE6AC      int     80h              ; LINUX – sys_open
```

Значит, битовые поля флагов `open()` проверяются где-то в ядре Linux.

Разумеется, и стандартные библиотеки Linux и ядро Linux можно получить в виде исходников, но нам интересно попробовать разобраться без них.

При системном вызове `sys_open` управление в конечном итоге передается в `do_sys_open` в ядре Linux 2.6. Оттуда – в `do_filp_open()` (эта функция находится в исходниках ядра в файле `fs/namei.c`).

20.1. ПРОВЕРКА КАКОГО-ЛИБО БИТА

N.B. Помимо передачи параметров функции через стек, существует также возможность передавать некоторые из них через регистры. Такое соглашение о вызове называется *fastcall* (65.3 (стр. 673)). Оно работает немного быстрее, так как для чтения аргументов процессору не нужно обращаться к стеку, лежащему в памяти. В GCC есть опция *regparm*³, и с её помощью можно задать, сколько аргументов можно передать через регистры.

Ядро Linux 2.6 собирается с опцией `-mregparm=3`^{4 5}.

Для нас это означает, что первые три аргумента функции будут передаваться через регистры `EAX`, `EDX` и `ECX`, а остальные через стек. Разумеется, если аргументов у функции меньше трех, то будет задействована только часть этих регистров.

Итак, качаем ядро 2.6.31, собираем его в Ubuntu, открываем в IDA, находим функцию `do_filp_open()`. В начале мы увидим что-то такое (комментарии мои):

Листинг 20.6: `do_filp_open()` (linux kernel 2.6.31)

```
do_filp_open    proc near
...
    push    ebp
    mov     ebp, esp
    push    edi
    push    esi
    push    ebx
    mov     ebx, ecx
    add     ebx, 1
    sub     esp, 98h
    mov     esi, [ebp+arg_4] ; acc_mode (пятый аргумент)
    test    bl, 3
    mov     [ebp+var_80], eax ; dfd (первый аргумент)
    mov     [ebp+var_7C], edx ; pathname (второй аргумент)
    mov     [ebp+var_78], ecx ; open_flag (третий аргумент)
    jnz     short loc_C01EF684
    mov     ebx, ecx          ; ebx <- open_flag
```

GCC сохраняет значения первых трех аргументов в локальном стеке. Иначе, если эти три регистра не трогать вообще, то функции компилятора, распределяющей переменные по регистрам (так называемый *register allocator*), будет очень тесно.

Далее находим примерно такой фрагмент кода:

Листинг 20.7: `do_filp_open()` (linux kernel 2.6.31)

```
loc_C01EF6B4:    ; CODE XREF: do_filp_open+4F
                ; O_CREAT
    test     bl, 40h
    jnz     loc_C01EF810
    mov     edi, ebx
    shr     edi, 11h
    xor     edi, 1
    and     edi, 1
    test     ebx, 10000h
    jz      short loc_C01EF6D3
    or      edi, 2
```

`0x40` — это значение макроса `O_CREAT`. `open_flag` проверяется на наличие бита `0x40` и если бит равен 1, то выполняется следующие за `JNZ` инструкции.

20.1.2. ARM

В ядре Linux 3.8.0 бит `O_CREAT` проверяется немного иначе.

Листинг 20.8: linux kernel 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                          const struct open_flags *op)
{
```

³ohse.de/uwe/articles/gcc-attributes.html#func-regparm

⁴kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f

⁵См. также файл `arch/x86/include/asm/calling.h` в исходниках ядра

20.1. ПРОВЕРКА КАКОГО-ЛИБО БИТА

```
...
    filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                                struct nameidata *nd, const struct open_flags *op, int flags)
{
    ...
    error = do_last(nd, &path, file, op, &opened, pathname);
    ...
}

static int do_last(struct nameidata *nd, struct path *path,
                  struct file *file, const struct open_flags *op,
                  int *opened, struct filename *name)
{
    ...
    if (!(open_flag & O_CREAT)) {
        ...
        error = lookup_fast(nd, path, &inode);
        ...
    } else {
        ...
        error = complete_walk(nd);
    }
    ...
}
```

Вот как это выглядит в IDA, ядро скомпилированное для режима ARM:

Листинг 20.9: do_last() из vmlinux (IDA)

```
...
.text:C0169EA8      MOV            R9, R3 ; R3 - (4th argument) open_flag
...
.text:C0169ED4      LDR            R6, [R9] ; R6 - open_flag
...
.text:C0169F68      TST            R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C      BNE            loc_C016A128
.text:C0169F70      LDR            R2, [R4,#0x10]
.text:C0169F74      ADD            R12, R4, #8
.text:C0169F78      LDR            R3, [R4,#0xC]
.text:C0169F7C      MOV            R0, R4
.text:C0169F80      STR            R12, [R11,#var_50]
.text:C0169F84      LDRB           R3, [R2,R3]
.text:C0169F88      MOV            R2, R8
.text:C0169F8C      CMP            R3, #0
.text:C0169F90      ORRNE          R1, R1, #3
.text:C0169F94      STRNE          R1, [R4,#0x24]
.text:C0169F98      ANDS           R3, R6, #0x200000
.text:C0169F9C      MOV            R1, R12
.text:C0169FA0      LDRNE          R3, [R4,#0x24]
.text:C0169FA4      ANDNE          R3, R3, #1
.text:C0169FA8      EORNE          R3, R3, #1
.text:C0169FAC      STR            R3, [R11,#var_54]
.text:C0169FB0      SUB            R3, R11, #-var_38
.text:C0169FB4      BL             lookup_fast
...
.text:C016A128      loc_C016A128                                ; CODE XREF: do_last.isra.14+DC
.text:C016A128      MOV            R0, R4
.text:C016A12C      BL             complete_walk
...
```

TST это аналог инструкции **TEST** в x86. Мы можем «узнать» визуально этот фрагмент кода по тому что в одном случае исполнится функция `lookup_fast()`, а в другом `complete_walk()`. Это соответствует исходному коду функции `do_last()`. Макрос `O_CREAT` здесь так же равен `0x40`.

20.2. Установка и сброс отдельного бита

Например:

```
#include <stdio.h>

#define IS_SET(flag, bit)      ((flag) & (bit))
#define SET_BIT(var, bit)     ((var) |= (bit))
#define REMOVE_BIT(var, bit)  ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};

int main()
{
    f(0x12340678);
};
```

20.2.1. x86

Неоптимизирующий MSVC

Имеем в итоге (MSVC 2010):

Листинг 20.10: MSVC 2010

```
_rt$ = -4          ; size = 4
_a$ = 8           ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR _rt$[ebp], eax
    mov     ecx, DWORD PTR _rt$[ebp]
    or      ecx, 16384          ; 00004000H
    mov     DWORD PTR _rt$[ebp], ecx
    mov     edx, DWORD PTR _rt$[ebp]
    and     edx, -513          ; ffffffffH
    mov     DWORD PTR _rt$[ebp], edx
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f ENDP
```

Инструкция **OR** здесь устанавливает в переменной ещё один бит, игнорируя остальные.

А **AND** сбрасывает некий бит. Можно также сказать, что **AND** здесь копирует все биты, кроме одного. Действительно, во втором операнде **AND** выставлены в единицу те биты, которые нужно сохранить, кроме одного, копировать который мы не хотим (и который 0 в битовой маске). Так проще понять и запомнить.

0x200 (00000000000000000000**1**000000000) (т.е. 10-й бит (считая с первого)).

0x4000 (00000000000000001000000000000000) (т.е. 15-й бит).

Входное значение это: 0x12340678 (10010001101000000011001111000). Видим, как оно загрузилось:

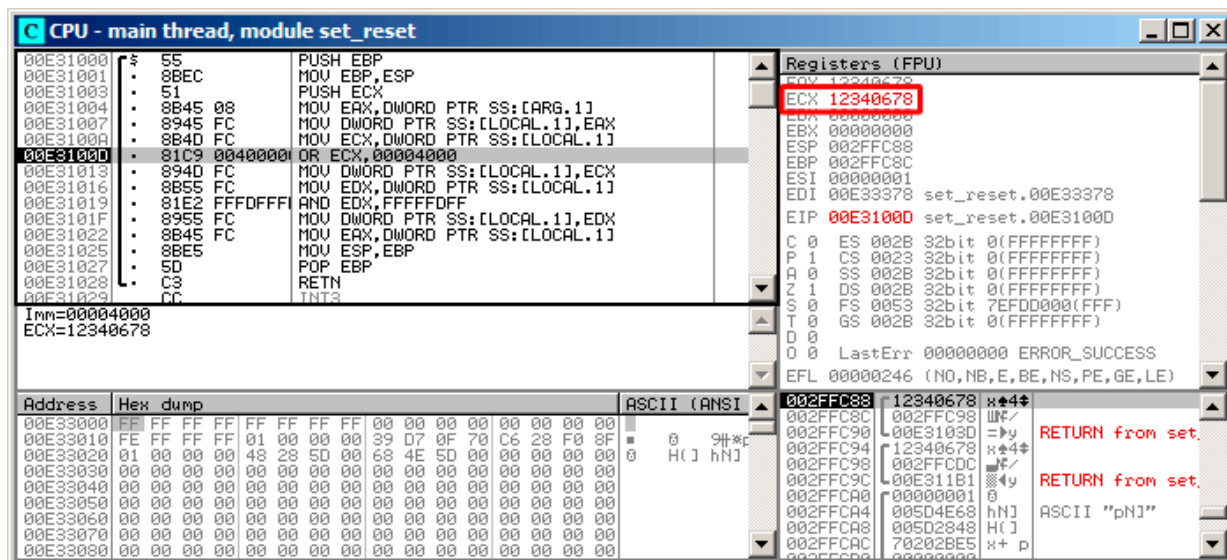


Рис. 20.1: OllyDbg: значение загружено в ECX

OR исполнилась:

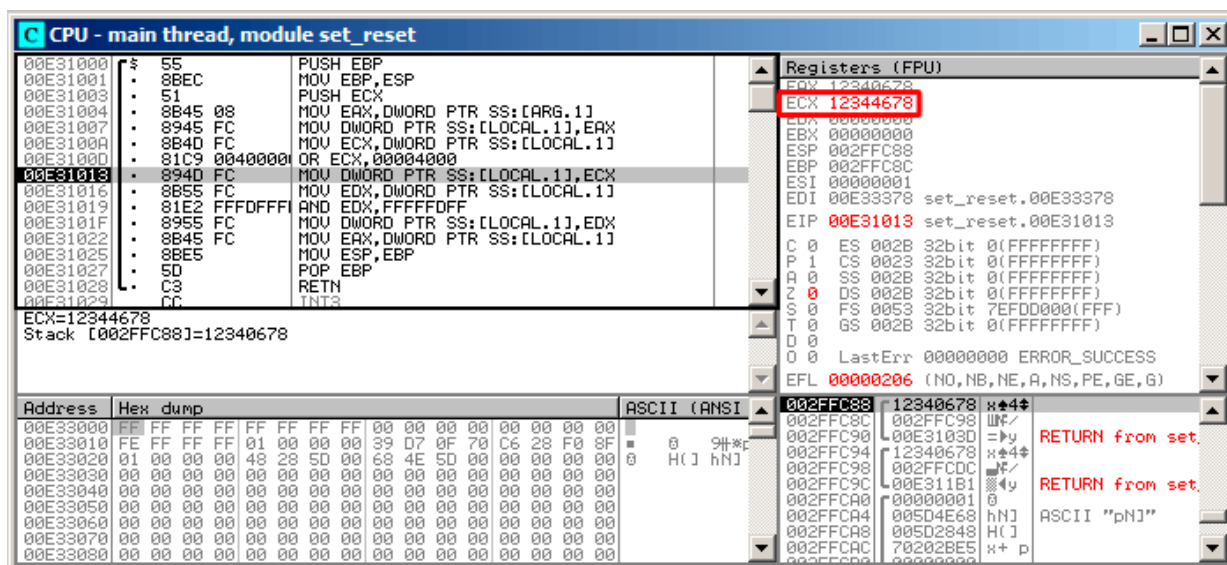


Рис. 20.2: OllyDbg: OR сработал

15-й бит выставлен: 0x1234**4**678 (10010001101000**1**00011001111000).

20.2. УСТАНОВКА И СБРОС ОТДЕЛЬНОГО БИТА

Значение перезагружается снова (потому что использовался режим компилятора без оптимизации):

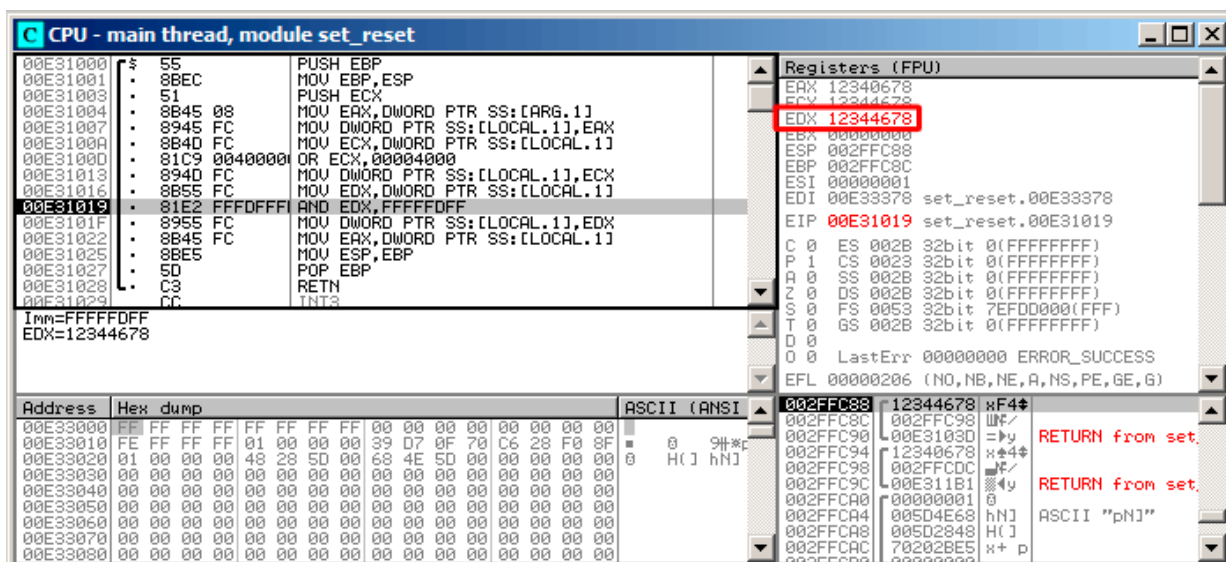


Рис. 20.3: OllyDbg: значение перезагрузилось в EDX

AND исполнилась:

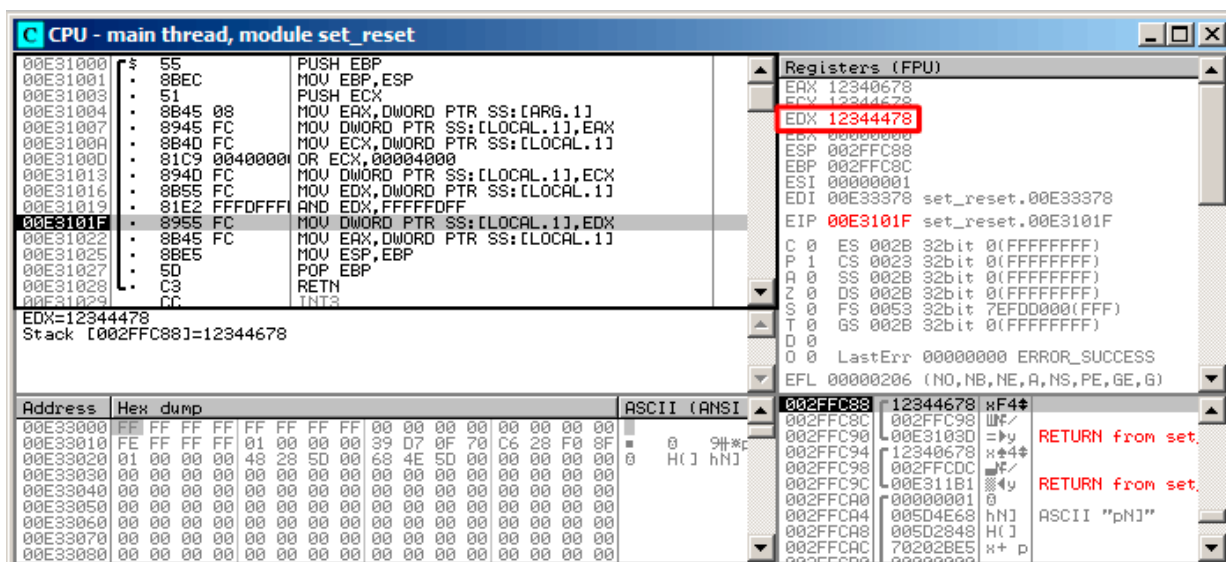


Рис. 20.4: OllyDbg: AND сработал

10-й бит очищен (или, иным языком, оставлены все биты кроме 10-го) и итоговое значение это 0x12344478 (100100011010001000100011111000).

Оптимизирующий MSVC

Если скомпилировать в MSVC с оптимизацией (/Ox), то код еще короче:

Листинг 20.11: Оптимизирующий MSVC

```
_a$ = 8 ; size = 4
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    and     eax, -513 ; ffffffffh
    or      eax, 16384 ; 00004000h
    ret     0
_f ENDP
```

Неоптимизирующий GCC

Попробуем GCC 4.4.1 без оптимизации:

Листинг 20.12: Неоптимизирующий GCC

```
f
public f
proc near

var_4      = dword ptr -4
arg_0      = dword ptr  8

    push    ebp
    mov     ebp, esp
    sub     esp, 10h
    mov     eax, [ebp+arg_0]
    mov     [ebp+var_4], eax
    or      [ebp+var_4], 4000h
    and     [ebp+var_4], 0FFFFFFFh
    mov     eax, [ebp+var_4]
    leave
    retn

f
endp
```

Также избыточный код, хотя короче, чем у MSVC без оптимизации.

Попробуем теперь GCC с оптимизацией `-O3`:

Оптимизирующий GCC

Листинг 20.13: Оптимизирующий GCC

```

f      public f
      proc near
arg_0  = dword ptr 8

      push    ebp
      mov     ebp, esp
      mov     eax, [ebp+arg_0]
      pop     ebp
      or      ah, 40h
      and     ah, 0FDh
      retn
f      endp

```

Уже короче. Важно отметить, что через регистр `AH` компилятор работает с частью регистра `EAX`. Это его часть от 8-го до 15-го бита включительно.

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RAX ^{x64}							
				EAX			
						AX	
						AH	AL

N.B. В 16-битном процессоре 8086 аккумулятор имел название `AX` и состоял из двух 8-битных половин — `AL` (младшая часть) и `AH` (старшая). В 80386 регистры были расширены до 32-бит, аккумулятор стал называться `EAX`, но в целях совместимости, к его *более старым* частям всё ещё можно обращаться как к `AX / AH / AL`.

Из-за того, что все x86 процессоры — наследники 16-битного 8086, эти *старые* 16-битные опкоды короче нежели более новые 32-битные. Поэтому инструкция `or ah, 40h` занимает только 3 байта. Было бы логичнее сгенерировать здесь `or eax, 04000h`, но это уже 5 байт, или даже 6 (если регистр в первом операнде не `EAX`).

Оптимизирующий GCC и `regparm`

Если мы скомпилируем этот же пример не только с включенной оптимизацией `-O3`, но ещё и с опцией `regparm=3`, о которой я писал немного выше, то получится ещё короче:

Листинг 20.14: Оптимизирующий GCC

```

f      public f
      proc near
      push    ebp
      or      ah, 40h
      mov     ebp, esp
      and     ah, 0FDh
      pop     ebp
      retn
f      endp

```

Действительно — первый аргумент уже загружен в `EAX`, и прямо здесь можно начинать с ним работать. Интересно, что и пролог функции (`push ebp / mov ebp, esp`) и эпилог (`pop ebp`) функции можно смело выкинуть за ненужностью, но возможно GCC ещё не так хорош для подобных оптимизаций по размеру кода. Впрочем, в реальной жизни подобные короткие функции лучше всего автоматически делать в виде *inline-функций* (44 (стр. 496)).

20.2.2. ARM + Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 20.15: Оптимизирующий Keil 6/2013 (Режим ARM)

02 0C C0 E3	BIC	R0, R0, #0x200
01 09 80 E3	ORR	R0, R0, #0x4000
1E FF 2F E1	BX	LR

BIC (*Bitwise bit Clear*) это инструкция сбрасывающая заданные биты. Это как аналог **AND**, но только с инвертированным операндом.

Т.е. это аналог инструкций **NOT** + **AND**.

ORR это «логическое или», аналог **OR** в x86.

Пока всё понятно.

20.2.3. ARM + Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 20.16: Оптимизирующий Keil 6/2013 (Режим Thumb)

01 21 89 03	MOVS	R1, 0x4000
08 43	ORRS	R0, R1
49 11	ASRS	R1, R1, #5 ; generate 0x200 and place to R1
88 43	BICS	R0, R1
70 47	BX	LR

Вероятно, Keil решил, что код в режиме Thumb,

получающий **0x200** из **0x4000**, более компактный, нежели код, записывающий **0x200** в какой-нибудь регистр.

Поэтому при помощи инструкции **ASRS** (арифметический сдвиг вправо), это значение вычисляется как **0x4000** >> 5.

20.2.4. ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Листинг 20.17: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

42 0C C0 E3	BIC	R0, R0, #0x4200
01 09 80 E3	ORR	R0, R0, #0x4000
1E FF 2F E1	BX	LR

Код, который был сгенерирован LLVM, в исходном коде, на самом деле, выглядел бы так:

```
REMOVE_BIT (rt, 0x4200);
SET_BIT (rt, 0x4000);
```

И он делает в точности что нам нужно. Но почему **0x4200**? Возможно, это артефакт оптимизатора LLVM⁶. Возможно, ошибка оптимизатора компилятора, но создаваемый код всё же работает верно.

Об аномалиях компиляторов, подробнее читайте здесь (93 (стр. 888)).

Оптимизирующий Xcode 4.6.3 (LLVM) для режима Thumb генерирует точно такой же код.

20.2.5. ARM: ещё об инструкции BIC

Если немного переделать пример:

```
int f(int a)
{
    int rt=a;

    REMOVE_BIT (rt, 0x1234);

    return rt;
};
```

⁶Это был LLVM build 2410.2.00 входящий в состав Apple Xcode 4.6.3

20.2. УСТАНОВКА И СБРОС ОТДЕЛЬНОГО БИТА

То оптимизирующий Keil 5.03 в режиме ARM сделает такое:

```
f PROC
    BIC    r0,r0,#0x1000
    BIC    r0,r0,#0x234
    BX     lr
ENDP
```

Здесь две инструкции **BIC**, т.е. биты **0x1234** сбрасываются в два прохода.

Это потому что в инструкции **BIC** нельзя закодировать значение **0x1234**, но можно закодировать **0x1000** либо **0x234**.

20.2.6. ARM64: Оптимизирующий GCC (Linaro) 4.9

Оптимизирующий GCC, компилирующий для ARM64, может использовать **AND** вместо **BIC**:

Листинг 20.18: Оптимизирующий GCC (Linaro) 4.9

```
f:
    and    w0, w0, -513      ; 0xFFFFFFFFFFFFDFF
    orr    w0, w0, 16384     ; 0x4000
    ret
```

20.2.7. ARM64: Неоптимизирующий GCC (Linaro) 4.9

Неоптимизирующий GCC генерирует больше избыточного кода, но он работает также:

Листинг 20.19: Неоптимизирующий GCC (Linaro) 4.9

```
f:
    sub    sp, sp, #32
    str    w0, [sp,12]
    ldr    w0, [sp,12]
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    orr    w0, w0, 16384     ; 0x4000
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    and    w0, w0, -513      ; 0xFFFFFFFFFFFFDFF
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    add    sp, sp, 32
    ret
```

20.2.8. MIPS

Листинг 20.20: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
; $a0=a
ori    $a0, 0x4000
; $a0=a|0x4000
li     $v0, 0xFFFFDFF
jr     $ra
and    $v0, $a0, $v0
; на выходе: $v0 = $a0&$v0 = a|0x4000 & 0xFFFFDFF
```

ORI это, конечно, операция «ИЛИ», «l» в имени инструкции означает что значение встроено в машинный код.

И напротив, есть **AND**. Здесь нет возможности использовать **ANDI**, потому что невозможно встроить число **0xFFFFDFF** в одну инструкцию, так что компилятору приходится в начале загружать значение **0xFFFFDFF** в регистр **\$V0**, а затем генерировать **AND**, которая возьмет все значения из регистров.

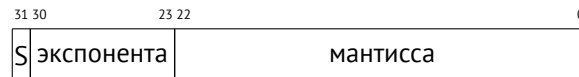
20.3. Сдвиги

Битовые сдвиги в Си/Си++ реализованы при помощи операторов $\ll$ и $\gg$. В x86 есть инструкции SHL (SHift Left) и SHR (SHift Right) для этого. Инструкции сдвига также активно применяются при делении или умножении на числа-степени двойки: 2^n (т.е. 1, 2, 4, 8, и т.д.): [17.1.2](#) (стр. 207), [17.2.1](#) (стр. 212).

Операции сдвига ещё потому так важны, потому что они часто используются для изолирования определенного бита или для конструирования значения из нескольких разрозненных бит.

20.4. Установка и сброс отдельного бита: пример с FPU

Как мы уже можем знать, вот как биты расположены в значении типа *float* в формате IEEE 754:



(S – знак)

Знак числа — это [MSB](#)⁷. Возможно ли работать со знаком числа с плавающей точкой, не используя FPU-инструкций?

```
#include <stdio.h>

float my_abs (float i)
{
    unsigned int tmp=*(unsigned int*)&i & 0x7FFFFFFF;
    return *(float*)&tmp;
};

float set_sign (float i)
{
    unsigned int tmp=*(unsigned int*)&i | 0x80000000;
    return *(float*)&tmp;
};

float negate (float i)
{
    unsigned int tmp=*(unsigned int*)&i ^ 0x80000000;
    return *(float*)&tmp;
};

int main()
{
    printf ("my_abs():\n");
    printf ("%f\n", my_abs (123.456));
    printf ("%f\n", my_abs (-456.123));
    printf ("set_sign():\n");
    printf ("%f\n", set_sign (123.456));
    printf ("%f\n", set_sign (-456.123));
    printf ("negate():\n");
    printf ("%f\n", negate (123.456));
    printf ("%f\n", negate (-456.123));
};
```

Придется использовать эти трюки в Си/Си++ с типами данных чтобы копировать из значения типа *float* и обратно без конверсии. Так что здесь три функции: `my_abs()` сбрасывает [MSB](#); `set_sign()` устанавливает [MSB](#) и `negate()` меняет его на противоположный.

[XOR](#) может использоваться для смены бита: ?? (стр. ??).

20.4.1. x86

Код прямолинеен:

⁷Most significant bit/byte (самый старший бит/байт)

```

_tmp$ = 8
_i$ = 8
_my_abs PROC
    and     DWORD PTR _i$[esp-4], 2147483647      ; 7fffffffH
    fld     DWORD PTR _tmp$[esp-4]
    ret     0
_my_abs ENDP

_tmp$ = 8
_i$ = 8
_set_sign PROC
    or      DWORD PTR _i$[esp-4], -2147483648     ; 80000000H
    fld     DWORD PTR _tmp$[esp-4]
    ret     0
_set_sign ENDP

_tmp$ = 8
_i$ = 8
_negate PROC
    xor     DWORD PTR _i$[esp-4], -2147483648     ; 80000000H
    fld     DWORD PTR _tmp$[esp-4]
    ret     0
_negate ENDP

```

Входное значение типа *float* берется из стека, но мы обходимся с ним как с целочисленным значением.

AND и **OR** сбрасывают и устанавливают нужный бит. **XOR** переворачивает его.

В конце измененное значение загружается в **ST0**, потому что числа с плавающей точкой возвращаются в этом регистре.

Попробуем оптимизирующий MSVC 2012 для x64:

Листинг 20.22: Оптимизирующий MSVC 2012 x64

```

tmp$ = 8
i$ = 8
my_abs PROC
    movss   DWORD PTR [rsp+8], xmm0
    mov     eax, DWORD PTR i$[rsp]
    btr     eax, 31
    mov     DWORD PTR tmp$[rsp], eax
    movss   xmm0, DWORD PTR tmp$[rsp]
    ret     0
my_abs ENDP
_TEXT ENDS

tmp$ = 8
i$ = 8
set_sign PROC
    movss   DWORD PTR [rsp+8], xmm0
    mov     eax, DWORD PTR i$[rsp]
    bts     eax, 31
    mov     DWORD PTR tmp$[rsp], eax
    movss   xmm0, DWORD PTR tmp$[rsp]
    ret     0
set_sign ENDP

tmp$ = 8
i$ = 8
negate PROC
    movss   DWORD PTR [rsp+8], xmm0
    mov     eax, DWORD PTR i$[rsp]
    btc     eax, 31
    mov     DWORD PTR tmp$[rsp], eax
    movss   xmm0, DWORD PTR tmp$[rsp]
    ret     0
negate ENDP

```

Во-первых, входное значение передается в **XMM0**, затем оно копируется в локальный стек и затем мы видим новые для нас инструкции: **BTR**, **BTS**, **BTC**. Эти инструкции используются для сброса определенного бита (**BTR**: «reset»),

20.4. УСТАНОВКА И СБРОС ОТДЕЛЬНОГО БИТА: ПРИМЕР С FPU

установки (BTS : «set») и инвертирования (BTC : «complement»). 31-й бит это MSB, если считать начиная с нуля. И наконец, результат копируется в регистр XMM0, потому что значения с плавающей точкой возвращаются в регистре XMM0 в среде Win64.

20.4.2. MIPS

GCC 4.4.5 для MIPS делает почти то же самое:

Листинг 20.23: Оптимизирующий GCC 4.4.5 (IDA)

```
my_abs:
; скопировать из сопроцессора 1:
    mfc1    $v1, $f12
    li      $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; применить И:
    and     $v0, $v1
; скопировать в сопроцессор 1:
    mtc1    $v0, $f0
; возврат
    jr      $ra
    or      $at, $zero ; branch delay slot

set_sign:
; скопировать из сопроцессора 1:
    mfc1    $v0, $f12
    lui     $v1, 0x8000
; $v1=0x80000000
; применить ИЛИ:
    or      $v0, $v1, $v0
; скопировать в сопроцессор 1:
    mtc1    $v0, $f0
; возврат
    jr      $ra
    or      $at, $zero ; branch delay slot

negate:
; скопировать из сопроцессора 1:
    mfc1    $v0, $f12
    lui     $v1, 0x8000
; $v1=0x80000000
; применить исключающее ИЛИ:
    xor     $v0, $v1, $v0
; скопировать в сопроцессор 1:
    mtc1    $v0, $f0
; возврат
    jr      $ra
    or      $at, $zero ; branch delay slot
```

Для загрузки константы 0x80000000 в регистр используется только одна инструкция LUI, потому что LUI сбрасывает младшие 16 бит и это нули в константе, так что одной LUI без ORI достаточно.

20.4.3. ARM

Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 20.24: Оптимизирующий Keil 6/2013 (Режим ARM)

```
my_abs PROC
; очистить бит:
    BIC     r0,r0,#0x80000000
    BX      lr
    ENDP

set_sign PROC
; применить ИЛИ:
```


20.4. УСТАНОВКА И СБРОС ОТДЕЛЬНОГО БИТА: ПРИМЕР С FPU

```
    ORR    r0,r0,#0x80000000
    BX     lr
    ENDP

negate PROC
; применить исключающее ИЛИ:
    EOR    r0,r0,#0x80000000
    BX     lr
    ENDP
```

Пока всё понятно. В ARM есть инструкция **BIC** для сброса заданных бит.

EOR это инструкция в ARM которая делает то же что и **XOR** («Exclusive OR»).

Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 20.25: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
my_abs PROC
    LSLS    r0,r0,#1
; r0=i<<1
    LSRS    r0,r0,#1
; r0=(i<<1)>>1
    BX     lr
    ENDP

set_sign PROC
    MOVS    r1,#1
; r1=1
    LSLS    r1,r1,#31
; r1=1<<31=0x80000000
    ORRS    r0,r0,r1
; r0=r0 | 0x80000000
    BX     lr
    ENDP

negate PROC
    MOVS    r1,#1
; r1=1
    LSLS    r1,r1,#31
; r1=1<<31=0x80000000
    EORS    r0,r0,r1
; r0=r0 ^ 0x80000000
    BX     lr
    ENDP
```

В режиме Thumb 16-битные инструкции, в которых нельзя задать много данных, поэтому здесь применяется пара инструкций **MOVS / LSLS** для формирования константы 0x80000000.

Это работает как выражение: $1 \ll 31 = 0x80000000$.

Код `my_abs` выглядит странно и работает как выражение: $(i \ll 1) \gg 1$. Это выражение выглядит бессмысленным. Но тем не менее, когда выполняется `input << 1`, **MSB** (бит знака) просто выбрасывается. Когда выполняется следующее выражение `result >> 1`, все биты становятся на свои места, а **MSB** ноль, потому что все «новые» биты появляющиеся во время операций сдвига это всегда нули. Таким образом, пара инструкций **LSLS / LSRS** сбрасывают **MSB**.

Оптимизирующий GCC 4.6.3 (Raspberry Pi, Режим ARM)

Листинг 20.26: Оптимизирующий GCC 4.6.3 для Raspberry Pi (Режим ARM)

```
my_abs
; скопировать из S0 в R2:
    FMRS    R2, S0
; очистить бит:
    BIC     R3, R2, #0x80000000
; скопировать из R3 в S0:
    FMSR    S0, R3
```

20.5. ПОДСЧЕТ ВЫСТАВЛЕННЫХ БИТ

```

BX      LR

set_sign
; скопировать из S0 в R2:
      FMRS    R2, S0
; применить ИЛИ:
      ORR     R3, R2, #0x80000000
; скопировать из R3 в S0:
      FMSR    S0, R3
      BX      LR

negate
; скопировать из S0 в R2:
      FMRS    R2, S0
; применить операцию сложения:
      ADD     R3, R2, #0x80000000
; скопировать из R3 в S0:
      FMSR    S0, R3
      BX      LR
```

Запустим Raspberry Pi Linux в QEMU и он эмулирует FPU в ARM, так что здесь используются S-регистры для передачи значений с плавающей точкой вместо R-регистров.

Инструкция `FMRS` копирует данные из GPR в FPU и назад. `my_abs()` и `set_sign()` выглядят предсказуемо, но `negate()`? Почему там `ADD` вместо `XOR`?

Трудно поверить, но инструкция `ADD register, 0x80000000` работает так же как и `XOR register, 0x80000000`. Прежде всего, какая наша цель? Цель в том, чтобы поменять MSB на противоположный, и давайте забудем пока об операции `XOR`.

Из школьной математики мы можем помнить, что прибавление числа вроде 1000 к другому никогда не затрагивает последние 3 цифры.

Например: $1234567 + 10000 = 1244567$ (последние 4 цифры никогда не меняются). Но мы работаем с двоичной системой исчисления, и `0x80000000` это `0b10000000000000000000000000000000` в двоичной системе, т.е. только старший бит установлен.

Прибавление `0x80000000` к любому значению никогда не затронет младших 31 бит, а только MSB.

Прибавление 1 к 0 в итоге даст 1. Прибавление 1 к 1 даст `0b10` в двоичном виде, но 32-й бит (считая с нуля) выброшен, потому что наши регистры имеют ширину в 32 бита. Так что результат — 0.

Вот почему `XOR` здесь можно заменить на `ADD`. Трудно сказать, почему GCC решил сделать так, но это работает корректно.

20.5. Подсчет выставленных бит

Вот этот несложный пример иллюстрирует функцию, считающую количество бит-единиц во входном значении.

Эта операция также называется «population count»⁸.

```
#include <stdio.h>

#define IS_SET(flag, bit)      ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
};
```

⁸современные x86-процессоры (поддерживающие SSE4) даже имеют инструкцию `POPCNT` для этого

20.5. ПОДСЧЕТ ВЫСТАВЛЕННЫХ БИТ

```
int main()
{
    f(0x12345678); // test
};
```

В этом цикле счетчик итераций i считает от 0 до 31, а $1 \ll i$ будет от 1 до 0x80000000. Описывая это словами, можно сказать *сдвинуть единицу на n бит влево*. Т.е. в некотором смысле, выражение $1 \ll i$ последовательно выдает все возможные позиции бит в 32-битном числе. Освободившийся бит справа всегда обнуляется.

Вот таблица всех возможных значений $1 \ll i$ для $i = 0 \dots 31$:

Выражение в Си/Си++	Степень двойки	Десятичная форма	Шестнадцатеричная форма
$1 \ll 0$	1	1	1
$1 \ll 1$	2^1	2	2
$1 \ll 2$	2^2	4	4
$1 \ll 3$	2^3	8	8
$1 \ll 4$	2^4	16	0x10
$1 \ll 5$	2^5	32	0x20
$1 \ll 6$	2^6	64	0x40
$1 \ll 7$	2^7	128	0x80
$1 \ll 8$	2^8	256	0x100
$1 \ll 9$	2^9	512	0x200
$1 \ll 10$	2^{10}	1024	0x400
$1 \ll 11$	2^{11}	2048	0x800
$1 \ll 12$	2^{12}	4096	0x1000
$1 \ll 13$	2^{13}	8192	0x2000
$1 \ll 14$	2^{14}	16384	0x4000
$1 \ll 15$	2^{15}	32768	0x8000
$1 \ll 16$	2^{16}	65536	0x10000
$1 \ll 17$	2^{17}	131072	0x20000
$1 \ll 18$	2^{18}	262144	0x40000
$1 \ll 19$	2^{19}	524288	0x80000
$1 \ll 20$	2^{20}	1048576	0x100000
$1 \ll 21$	2^{21}	2097152	0x200000
$1 \ll 22$	2^{22}	4194304	0x400000
$1 \ll 23$	2^{23}	8388608	0x800000
$1 \ll 24$	2^{24}	16777216	0x1000000
$1 \ll 25$	2^{25}	33554432	0x2000000
$1 \ll 26$	2^{26}	67108864	0x4000000
$1 \ll 27$	2^{27}	134217728	0x8000000
$1 \ll 28$	2^{28}	268435456	0x10000000
$1 \ll 29$	2^{29}	536870912	0x20000000
$1 \ll 30$	2^{30}	1073741824	0x40000000
$1 \ll 31$	2^{31}	2147483648	0x80000000

Это числа-константы (битовые маски), которые крайне часто попадают в практике reverse engineer-а, и их нужно уметь распознавать.

Числа в десятичном виде заучивать, пожалуй, незачем, а числа в шестнадцатеричном виде их легко запомнить.

Эти константы очень часто используются для определения отдельных бит как флагов.

Например, это из файла `ssl_private.h` из исходников Apache 2.4.6:

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE (0)
#define SSL_OPT_RELSET (1<<0)
#define SSL_OPT_STDENVVARS (1<<1)
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH (1<<4)
#define SSL_OPT_STRICTREQUIRE (1<<5)
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

Вернемся назад к нашему примеру.

20.5. ПОДСЧЕТ ВЫСТАВЛЕННЫХ БИТ

Макрос `IS_SET` проверяет наличие этого бита в *a*.

Макрос `IS_SET` на самом деле это операция логического И (*AND*) и она возвращает 0 если бита там нет, либо эту же битовую маску, если бит там есть. В Си/Си++, конструкция `if()` срабатывает, если выражение внутри её не ноль, пусть хоть 123456, поэтому все будет работать.

20.5.1. x86

MSVC

Компилируем (MSVC 2010):

Листинг 20.27: MSVC 2010

```
_rt$ = -8          ; size = 4
_i$ = -4          ; size = 4
_a$ = 8           ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 8
    mov     DWORD PTR _rt$[ebp], 0
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN4@f
$LN3@f:
    mov     eax, DWORD PTR _i$[ebp] ; инкремент i
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp     DWORD PTR _i$[ebp], 32  ; 00000020H
    jge     SHORT $LN2@f           ; цикл закончился?
    mov     edx, 1
    mov     ecx, DWORD PTR _i$[ebp]
    shl     edx, cl                ; EDX=EDX<<CL
    and     edx, DWORD PTR _a$[ebp]
    je      SHORT $LN1@f           ; результат исполнения инструкции AND был 0?
    ; тогда пропускаем следующие команды
    mov     eax, DWORD PTR _rt$[ebp]
    add     eax, 1                ; инкремент rt
    mov     DWORD PTR _rt$[ebp], eax
$LN1@f:
    jmp     SHORT $LN3@f
$LN2@f:
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f ENDP
```

OllyDbg

Загрузим этот пример в OllyDbg. Входное значение для функции пусть будет 0x12345678.

Для $i = 1$, мы видим, как i загружается в ECX:

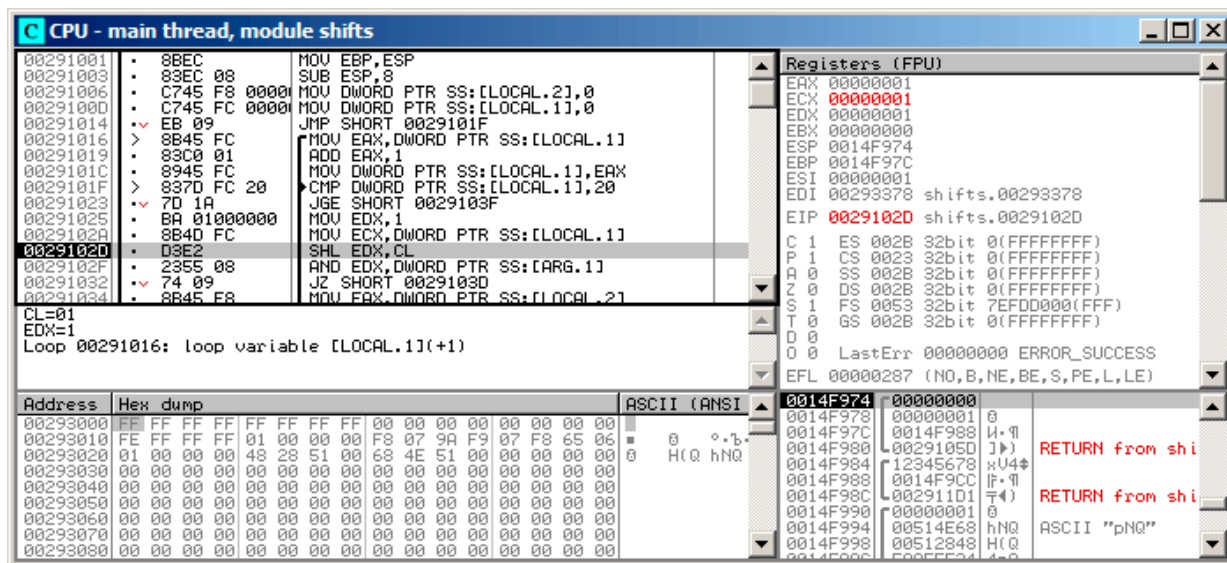


Рис. 20.5: OllyDbg: $i = 1$, i загружено в ECX

EDX содержит 1. Сейчас будет исполнена SHL.

SHL исполнилась:

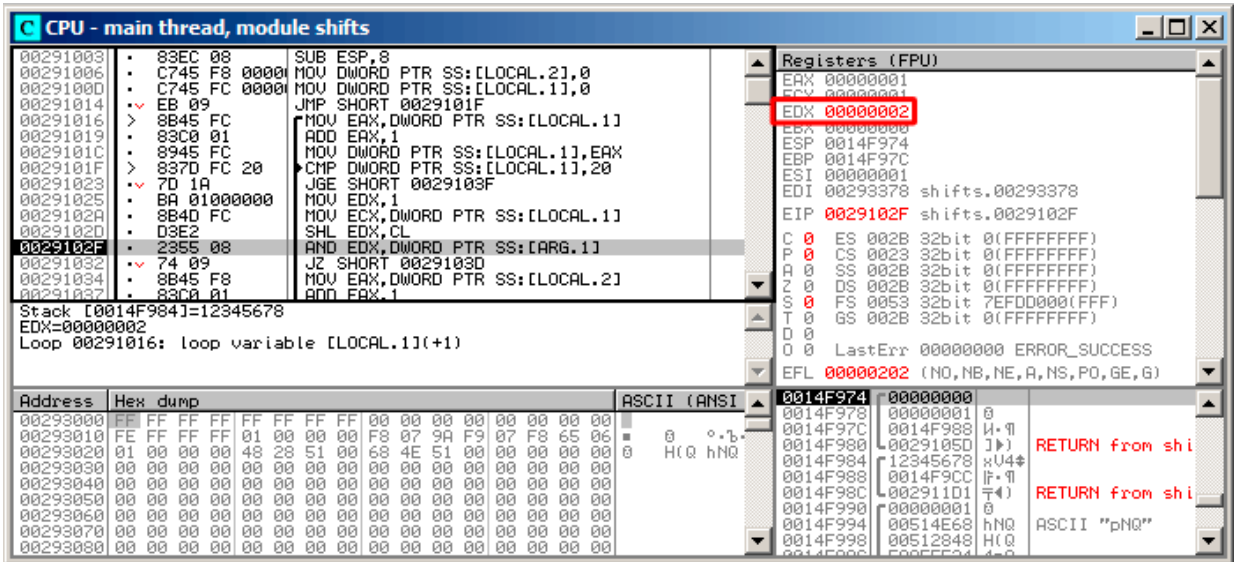


Рис. 20.6: OllyDbg: $i = 1$, $EDX = 1 \ll 1 = 2$

EDX содержит $1 \ll 1$ (или 2). Это битовая маска.

20.5. ПОДСЧЕТ ВЫСТАВЛЕННЫХ БИТ

AND устанавливает ZF в 1, что означает, что входное значение (0x12345678) умножается⁹ с 2 давая в результате 0:

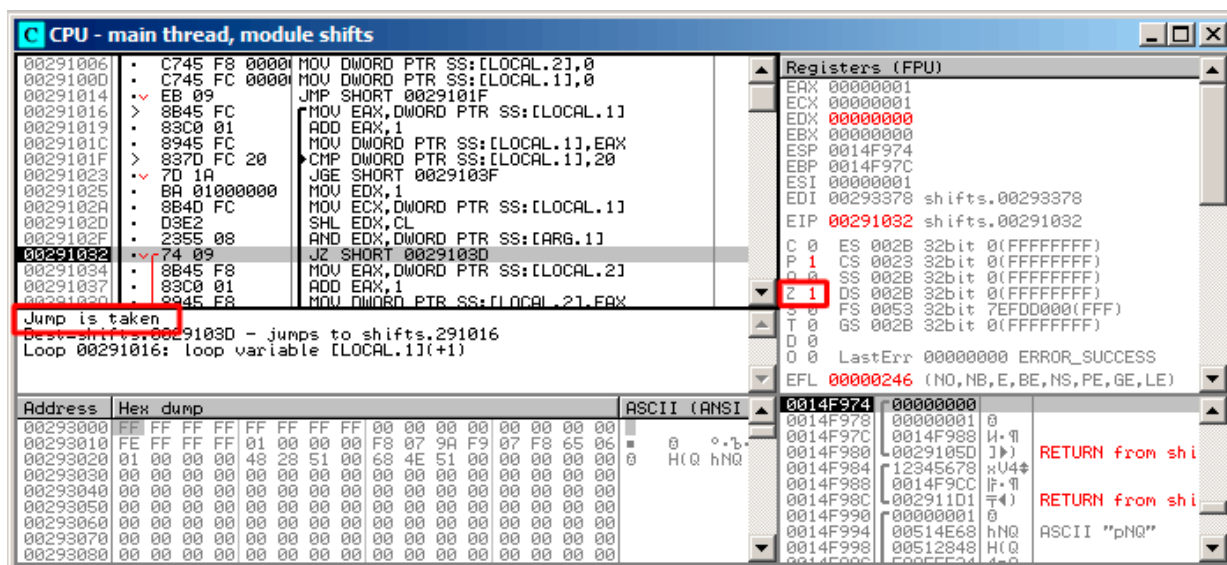


Рис. 20.7: OllyDbg: $i = 1$, есть ли этот бит во входном значении? Нет. (ZF =1)

Так что во входном значении соответствующего бита нет. Участок кода, увеличивающий счетчик бит на единицу, не будет исполнен: инструкция JZ обойдет его.

⁹Логическое «И»

20.5. ПОДСЧЕТ ВЫСТАВЛЕННЫХ БИТ

Немного потрассируем далее и *i* теперь 4.

SHL исполнилась:

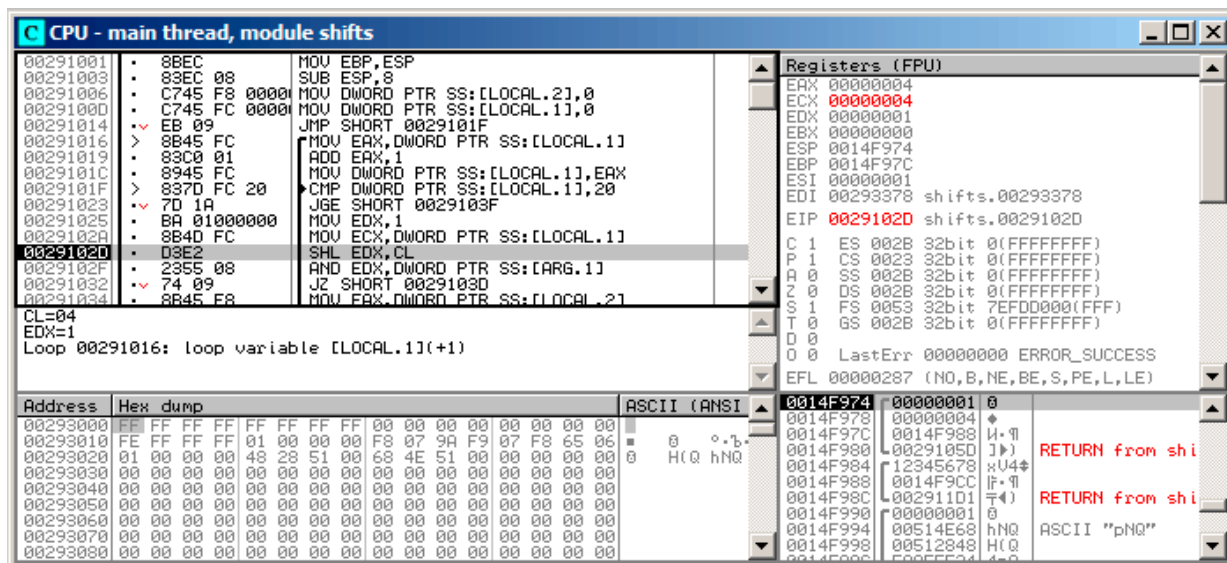


Рис. 20.8: OllyDbg: $i = 4$, i загружено в ECX

20.5. ПОДСЧЕТ ВЫСТАВЛЕННЫХ БИТ

EDX = 1 << 4 (или 0x10 или 16):

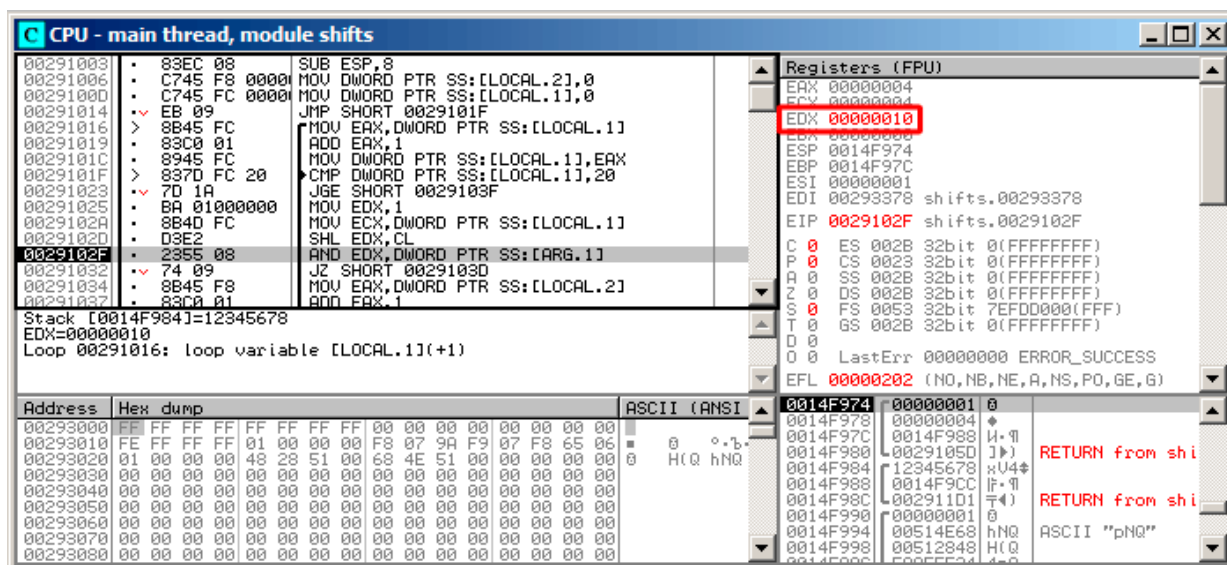


Рис. 20.9: OllyDbg: $i = 4$, EDX = 1 << 4 = 0x10

Это ещё одна битовая маска.

AND исполнилась:

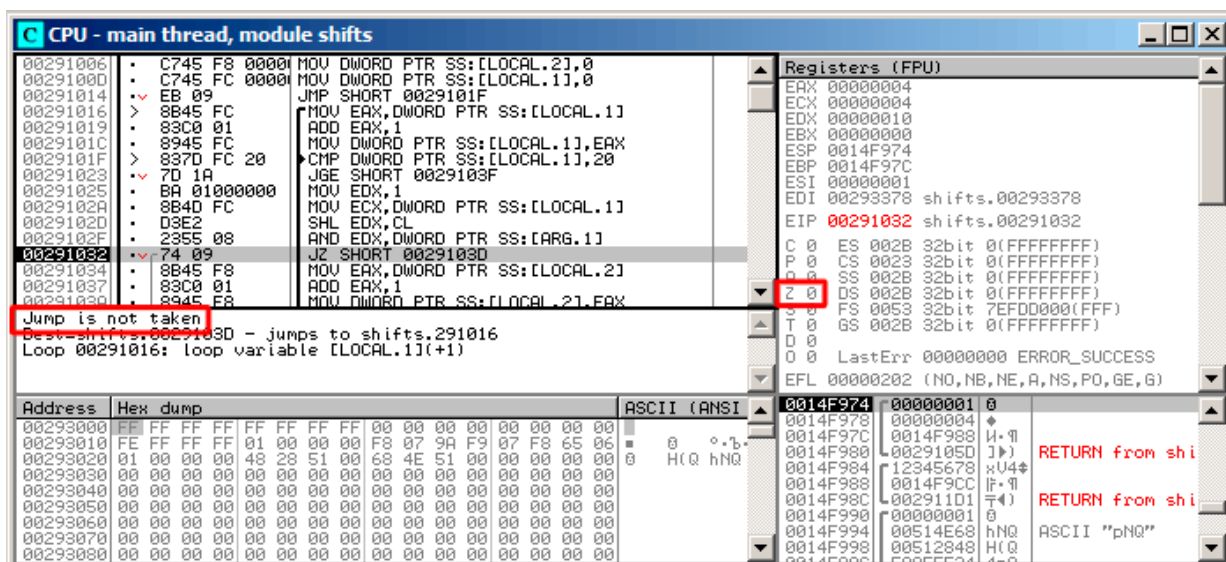


Рис. 20.10: OllyDbg: $i = 4$, есть ли этот бит во входном значении? Да. ($ZF = 0$)

ZF сейчас 0 потому что этот бит присутствует во входном значении. Действительно, $0x12345678 \& 0x10 = 0x10$. Этот бит считается: переход не сработает и счетчик бит будет увеличен на единицу.

Функция возвращает 13. Это количество установленных бит в значении $0x12345678$.

GCC

Скомпилируем то же и в GCC 4.4.1:

Листинг 20.28: GCC 4.4.1

```

f
public f
proc near

rt      = dword ptr -0Ch
i       = dword ptr -8
arg_0   = dword ptr  8

        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 10h
        mov     [ebp+rt], 0
        mov     [ebp+i], 0
        jmp     short loc_80483EF

loc_80483D0:
        mov     eax, [ebp+i]
        mov     edx, 1
        mov     ebx, edx
        mov     ecx, eax
        shl     ebx, cl
        mov     eax, ebx
        and     eax, [ebp+arg_0]
        test    eax, eax
        jz      short loc_80483EB
        add     [ebp+rt], 1

loc_80483EB:
        add     [ebp+i], 1

loc_80483EF:
        cmp     [ebp+i], 1Fh
        jle     short loc_80483D0
        mov     eax, [ebp+rt]
        add     esp, 10h
        pop     ebx

```

```

                pop     ebp
                retn
f:              endp

```

20.5.2. x64

Немного изменим пример, расширив его до 64-х бит:

```

#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit)      ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
};

```

Неоптимизирующий GCC 4.8.2

Пока всё просто.

Листинг 20.29: Неоптимизирующий GCC 4.8.2

```

f:
    push    rbp
    mov     rbp, rsp
    mov     QWORD PTR [rbp-24], rdi ; a
    mov     DWORD PTR [rbp-12], 0    ; rt=0
    mov     QWORD PTR [rbp-8], 0     ; i=0
    jmp     .L2
.L4:
    mov     rax, QWORD PTR [rbp-8]
    mov     rdx, QWORD PTR [rbp-24]
; RAX = i, RDX = a
    mov     ecx, eax
; ECX = i
    shr     rdx, cl
; RDX = RDX>>CL = a>>i
    mov     rax, rdx
; RAX = RDX = a>>i
    and     eax, 1
; EAX = EAX&1 = (a>>i)&1
    test    rax, rax
; последний бит был нулевым?
; пропустить следующую инструкцию ADD, если это было так.
    je     .L3
    add     DWORD PTR [rbp-12], 1    ; rt++
.L3:
    add     QWORD PTR [rbp-8], 1     ; i++
.L2:
    cmp     QWORD PTR [rbp-8], 63    ; i<63?
    jbe     .L4                     ; перейти на начало тела цикла, если это так
    mov     eax, DWORD PTR [rbp-12] ; возврат rt
    pop     rbp
    ret

```

Листинг 20.30: Оптимизирующий GCC 4.8.2

```

1 f:
2     xor     eax, eax           ; переменная rt будет находиться в регистре EAX
3     xor     ecx, ecx           ; переменная i будет находиться в регистре ECX
4 .L3:
5     mov     rsi, rdi           ; загрузить входное значение
6     lea     edx, [rax+1]       ; EDX=EAX+1
7 ; EDX здесь это "новая версия rt", которая будет записана в переменную rt, если последний бит был 1
8     shr     rsi, cl            ; RSI=RSI>>CL
9     and     esi, 1             ; ESI=ESI&1
10 ; последний бит был 1? Тогда записываем "новую версию rt" в EAX
11     cmovne  eax, edx
12     add     rcx, 1             ; RCX++
13     cmp     rcx, 64
14     jne     .L3
15     rep ret                    ; AKA fatret

```

Код более лаконичный, но содержит одну необычную вещь. Во всех примерах, что мы пока видели, инкремент значения переменной «rt» происходит после сравнения определенного бита с единицей, но здесь «rt» увеличивается на 1 до этого (строка 6), записывая новое значение в регистр **EDX**.

Затем, если последний бит был 1, инструкция **CMOVNE**¹⁰ (которая синонимична **CMOVNZ**¹¹) фиксирует новое значение «rt» копируя значение из **EDX** («предполагаемое значение rt») в **EAX** («текущее rt» которое будет возвращено в конце функции). Следовательно, инкремент происходит на каждом шаге цикла, т.е. 64 раза, вне всякой связи с входным значением.

Преимущество этого кода в том, что он содержит только один условный переход (в конце цикла) вместо двух (пропускающий инкремент «rt» и ещё одного в конце цикла).

И это может работать быстрее на современных CPU с предсказателем переходов: 34.1 (стр. 451).

Последняя инструкция это **REP RET** (опкод **F3 C3**) которая также называется **FATRET** в MSVC. Это оптимизированная версия **RET**, рекомендуемая AMD для вставки в конце функции, если **RET** идет сразу после условного перехода: [[Software Optimization Guide for AMD Family 16h Processors, (2013)]p.15]¹².

Оптимизирующий MSVC 2010

Листинг 20.31: MSVC 2010

```

a$ = 8
f PROC
; RCX = входное значение
xor     eax, eax
mov     edx, 1
lea     r8d, QWORD PTR [rax+64]
; R8D=64
npad    5
$LL4@f:
test    rdx, rcx
; не было такого бита во входном значении?
; тогда пропустить следующую инструкцию INC.
je      SHORT $LN3@f
inc     eax ; rt++
$LN3@f:
rol     rdx, 1 ; RDX=RDX<<1
dec     r8 ; R8--
jne     SHORT $LL4@f
fatret  0
f ENDP

```

Здесь используется инструкция **ROL** вместо **SHL**, которая на самом деле «rotate left» (прокручивать влево) вместо «shift left» (сдвиг влево), но здесь, в этом примере, она работает так же как и **SHL**.

¹⁰Conditional MOVe if Not Equal (**MOV** если не равно)

¹¹Conditional MOVe if Not Zero (**MOV** если не ноль)

¹²Больше об этом: <http://go.yurichev.com/17328>

20.5. ПОДСЧЕТ ВЫСТАВЛЕННЫХ БИТ

Об этих «прокручивающих» инструкциях больше читайте здесь: [A.6.3](#) (стр. 923).

R8 здесь считает от 64 до 0. Это как бы инвертированная переменная *i*.

Вот таблица некоторых регистров в процессе исполнения:

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
...	...
0x4000000000000000	2
0x8000000000000000	1

В конце видим инструкцию **FATRET**, которая была описана здесь: [20.5.2](#) (стр. 326).

Оптимизирующий MSVC 2012

Листинг 20.32: MSVC 2012

```
a$ = 8
f      PROC
; RCX = входное значение
      xor     eax, eax
      mov     edx, 1
      lea     r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
      npad    5
$LL4@f:
; проход 1 -----
      test    rdx, rcx
      je      SHORT $LN3@f
      inc     eax      ; rt++
$LN3@f:
      rol     rdx, 1   ; RDX=RDX<<1
; -----
; проход 2 -----
      test    rdx, rcx
      je      SHORT $LN11@f
      inc     eax      ; rt++
$LN11@f:
      rol     rdx, 1   ; RDX=RDX<<1
; -----
      dec     r8        ; R8--
      jne     SHORT $LL4@f
      fatret    0
f      ENDP
```

Оптимизирующий MSVC 2012 делает почти то же самое что и оптимизирующий MSVC 2010, но почему-то он генерирует 2 идентичных тела цикла и счетчик цикла теперь 32 вместо 64. Честно говоря, нельзя сказать, почему. Какой-то трюк с оптимизацией? Может быть, телу цикла лучше быть немного длиннее?

Так или иначе, такой код здесь уместен, чтобы показать, что результат компилятора иногда может быть очень странный и нелогичный, но прекрасно работающий, конечно же.

20.5.3. ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

Листинг 20.33: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

loc_2E54	MOV	R1, R0
	MOV	R0, #0
	MOV	R2, #1
	MOV	R3, R0
	TST	R1, R2, LSL R3 ; установить флаги в соответствии с R1 & (R2<<R3)
	ADD	R3, R3, #1 ; R3++
	ADDNE	R0, R0, #1 ; если флаг ZF сброшен TST, то R0++

20.5. ПОДСЧЕТ ВЫСТАВЛЕННЫХ БИТ

CMP	R3, #32
BNE	loc_2E54
BX	LR

TST это то же что и **TEST** в x86.

Как уже было указано (42.2.1 (стр. 485)), в режиме ARM нет отдельной инструкции для сдвигов.

Однако, модификаторами LSL (*Logical Shift Left*), LSR (*Logical Shift Right*), ASR (*Arithmetic Shift Right*), ROR (*Rotate Right*) и RRX (*Rotate Right with Extend*) можно дополнять некоторые инструкции, такие как **MOV**, **TST**, **CMP**, **ADD**, **SUB**, **RSB**¹³.

Эти модификаторы указывают, как сдвигать второй операнд, и на сколько.

Таким образом, инструкция «**TST R1, R2, LSL R3**» здесь работает как $R1 \wedge (R2 \ll R3)$.

20.5.4. ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

Почти такое же, только здесь применяется пара инструкций **LSL.W / TST** вместо одной **TST**, ведь в режиме Thumb нельзя добавлять модификатор **LSL** прямо в **TST**.

loc_2F7A	MOV	R1, R0
	MOVS	R0, #0
	MOV.W	R9, #1
	MOVS	R3, #0
	LSL.W	R2, R9, R3
	TST	R2, R1
	ADD.W	R3, R3, #1
	IT NE	
	ADDNE	R0, #1
	CMP	R3, #32
	BNE	loc_2F7A
	BX	LR

20.5.5. ARM64 + Оптимизирующий GCC 4.9

Возьмем 64-битный пример, который уже был здесь использован: 20.5.2 (стр. 325).

Листинг 20.34: Оптимизирующий GCC (Linaro) 4.8

```
f:
    mov     w2, 0           ; rt=0
    mov     x5, 1
    mov     w1, w2
.L2:
    lsl     x4, x5, x1       ; w4 = w5<<w1 = 1<<i
    add     w3, w2, 1        ; new_rt=rt+1
    tst     x4, x0           ; (1<<i) & a
    add     w1, w1, 1        ; i++
; результат TST был ненулевой?
; тогда w2=w3 или rt=new_rt.
; в противном случае: w2=w2 или rt=rt (холостая операция)
    csel    w2, w3, w2, ne
    cmp     w1, 64          ; i<64?
    bne     .L2             ; да
    mov     w0, w2          ; возврат rt
    ret
```

Результат очень похож на тот, что GCC сгенерировал для x64: 20.30 (стр. 326).

Инструкция **CSEL** это «Conditional SElect» (выбор при условии). Она просто выбирает одну из переменных, в зависимости от флагов выставленных **TST** и копирует значение в регистр **W2**, содержащий переменную «rt».

¹³Эти инструкции также называются «data processing instructions»

20.5.6. ARM64 + Неоптимизирующий GCC 4.9

И снова будем использовать 64-битный пример, который мы использовали ранее: [20.5.2](#) (стр. 325). Код более многословный, как обычно.

Листинг 20.35: Неоптимизирующий GCC (Linaro) 4.8

```
f:
    sub    sp, sp, #32
    str    x0, [sp,8]      ; сохранить значение "a" в Register Save Area
    str    wzr, [sp,24]    ; rt=0
    str    wzr, [sp,28]    ; i=0
    b      .L2

.L4:
    ldr    w0, [sp,28]
    mov    x1, 1
    lsl    x0, x1, x0      ; X0 = X1<<X0 = 1<<i
    mov    x1, x0
; X1 = 1<<1
    ldr    x0, [sp,8]
; X0 = a
    and    x0, x1, x0
; X0 = X1&X0 = (1<<i) & a
; X0 содержит ноль? тогда перейти на .L3, пропуская инкремент "rt"
    cmp    x0, xzr
    beq    .L3
; rt++
    ldr    w0, [sp,24]
    add    w0, w0, 1
    str    w0, [sp,24]

.L3:
; i++
    ldr    w0, [sp,28]
    add    w0, w0, 1
    str    w0, [sp,28]

.L2:
; i<=63? тогда перейти на .L4
    ldr    w0, [sp,28]
    cmp    w0, 63
    ble    .L4
; возврат rt
    ldr    w0, [sp,24]
    add    sp, sp, 32
    ret
```

20.5.7. MIPS**Неоптимизирующий GCC**

Листинг 20.36: Неоптимизирующий GCC 4.4.5 (IDA)

```
f:
; IDA не знает об именах переменных, мы присвоили их вручную:
rt      = -0x10
i       = -0xC
var_4   = -4
a       = 0

    addiu  $sp, -0x18
    sw     $fp, 0x18+var_4($sp)
    move   $fp, $sp
    sw     $a0, 0x18+a($fp)
; инициализировать переменные rt и i в ноль:
    sw     $zero, 0x18+rt($fp)
    sw     $zero, 0x18+i($fp)
; перейти на инструкции проверки цикла:
    b      loc_68
    or     $at, $zero ; branch delay slot, NOP
```

```

loc_20:
    li    $v1, 1
    lw    $v0, 0x18+i($fp)
    or     $at, $zero ; load delay slot, NOP
    sllv   $v0, $v1, $v0
; $v0 = 1<<i
    move   $v1, $v0
    lw    $v0, 0x18+a($fp)
    or     $at, $zero ; load delay slot, NOP
    and    $v0, $v1, $v0
; $v0 = a&(1<<i)
; a&(1<<i) равен нулю? тогда перейти на loc_58:
    beqz   $v0, loc_58
    or     $at, $zero
; переход не случился, это значит что a&(1<<i)!=0, так что инкрементируем "rt":
    lw    $v0, 0x18+rt($fp)
    or     $at, $zero ; load delay slot, NOP
    addiu  $v0, 1
    sw    $v0, 0x18+rt($fp)

loc_58:
; инкремент i:
    lw    $v0, 0x18+i($fp)
    or     $at, $zero ; load delay slot, NOP
    addiu  $v0, 1
    sw    $v0, 0x18+i($fp)

loc_68:
; загрузить i и сравнить его с 0x20 (32).
; перейти на loc_20, если это меньше чем 0x20 (32):
    lw    $v0, 0x18+i($fp)
    or     $at, $zero ; load delay slot, NOP
    slti   $v0, 0x20 # ' '
    bnez   $v0, loc_20
    or     $at, $zero ; branch delay slot, NOP
; эпилог функции. возврат rt:
    lw    $v0, 0x18+rt($fp)
    move   $sp, $fp ; load delay slot
    lw    $fp, 0x18+var_4($sp)
    addiu  $sp, 0x18 ; load delay slot
    jr     $ra
    or     $at, $zero ; branch delay slot, NOP

```

Это многословно: все локальные переменные расположены в локальном стеке и перезагружаются каждый раз, когда нужны. Инструкция **SLLV** это «Shift Word Left Logical Variable», она отличается от **SLL** только тем что количество бит для сдвига кодируется в **SLL** (и, следовательно, фиксировано), а **SLLV** берет количество из регистра.

Оптимизирующий GCC

Это более сжато. Здесь две инструкции сдвигов вместо одной. Почему? Можно заменить первую инструкцию **SLLV** на инструкцию безусловного перехода, передав управление прямо на вторую **SLLV**.

Но это ещё одна инструкция перехода в функции, а от них избавляться всегда выгодно: [34.1](#) (стр. [451](#)).

Листинг 20.37: Оптимизирующий GCC 4.4.5 (IDA)

```

f:
; $a0=a
; переменная rt будет находиться в $v0:
    move   $v0, $zero
; переменная i будет находиться в $v1:
    move   $v1, $zero
    li     $t0, 1
    li     $a3, 32
    sllv   $a1, $t0, $v1
; $a1 = $t0<<$v1 = 1<<i

loc_14:

```


20.6. Вывод

```
        and    $a1, $a0
; $a1 = a&(1<<i)
; инкремент i:
        addiu  $v1, 1
; переход на loc_28 если a&(1<<i)==0 и инкремент rt:
        beqz   $a1, loc_28
        addiu  $a2, $v0, 1
; если BEQZ не сработала, сохранить обновленную rt в $v0:
        move   $v0, $a2

loc_28:
; если i!=32, перейти на loc_14 а также подготовить следующее сдвинутое значение:
        bne    $v1, $a3, loc_14
        sllv   $a1, $t0, $v1
; возврат
        jr     $ra
        or     $at, $zero ; branch delay slot, NOP
```

20.6. Вывод

Инструкции сдвига, аналогичные операторам Си/Си++ $\ll$ и $\gg$, в x86 это **SHR / SHL** (для беззнаковых значений), **SAR / SHL** (для знаковых значений).

Инструкции сдвига в ARM это **LSR / LSL** (для беззнаковых значений), **ASR / LSL** (для знаковых значений).

Можно также добавлять суффикс сдвига для некоторых инструкций (которые называются «data processing instructions»).

20.6.1. Проверка определенного бита (известного на стадии компиляции)

Проверить, присутствует ли бит 1000000 (0x40) в значении в регистре:

Листинг 20.38: Си/Си++

```
if (input&0x40)
    ...
```

Листинг 20.39: x86

```
TEST REG, 40h
JNZ is_set
; бит не установлен
```

Листинг 20.40: x86

```
TEST REG, 40h
JZ is_cleared
; бит установлен
```

Листинг 20.41: ARM (Режим ARM)

```
TST REG, #0x40
BNE is_set
; бит не установлен
```

Иногда **AND** используется вместо **TEST**, но флаги выставляются точно также.

20.6.2. Проверка определенного бита (заданного во время исполнения)

Это обычно происходит при помощи вот такого фрагмента на Си/Си++ (сдвинуть значение на n бит вправо, затем отрезать самый младший бит):

Листинг 20.42: Си/Си++

```
if ((value>>n)&1)
    ....
```

20.6. ВЫВОД

Это обычно реализуется в x86-коде так:

Листинг 20.43: x86

```
; REG=input_value  
; CL=n  
SHR REG, CL  
AND REG, 1
```

Или (сдвинуть 1 n раз влево, изолировать этот же бит во входном значении и проверить, не ноль ли он):

Листинг 20.44: Си/Си++

```
if (value & (1<<n))  
....
```

Это обычно так реализуется в x86-коде:

Листинг 20.45: x86

```
; CL=n  
MOV REG, 1  
SHL REG, CL  
AND input_value, REG
```

20.6.3. Установка определенного бита (известного во время компиляции)

Листинг 20.46: Си/Си++

```
value=value|0x40;
```

Листинг 20.47: x86

```
OR REG, 40h
```

Листинг 20.48: ARM (Режим ARM) и ARM64

```
ORR R0, R0, #0x40
```

20.6.4. Установка определенного бита (заданного во время исполнения)

Листинг 20.49: Си/Си++

```
value=value|(1<<n);
```

Это обычно так реализуется в x86-коде:

Листинг 20.50: x86

```
; CL=n  
MOV REG, 1  
SHL REG, CL  
OR input_value, REG
```

20.6.5. Сброс определенного бита (известного во время компиляции)

Просто выполните операцию логического «И» (**AND**) с инвертированным значением:

Листинг 20.51: Си/Си++

```
value=value&(~0x40);
```

Листинг 20.52: x86

```
AND REG, 0FFFFFFBFh
```

Листинг 20.53: x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

Это на самом деле сохранение всех бит кроме одного.

В ARM в режиме ARM есть инструкция **BIC**, работающая как две инструкции **NOT** + **AND**:

Листинг 20.54: ARM (Режим ARM)

```
BIC R0, R0, #0x40
```

20.6.6. Сброс определенного бита (заданного во время исполнения)

Листинг 20.55: Си/Си++

```
value=value&~(1<<n));
```

Листинг 20.56: x86

```
; CL=n  
MOV REG, 1  
SHL REG, CL  
NOT REG  
AND input_value, REG
```

20.7. Упражнения

- <http://challenges.re/67>
- <http://challenges.re/68>
- <http://challenges.re/69>
- <http://challenges.re/70>

Глава 21

Линейный конгруэнтный генератор как генератор псевдослучайных чисел

Линейный конгруэнтный генератор, пожалуй, самый простой способ генерировать псевдослучайные числа. Он не в почете в наше время¹, но он настолько прост (только одно умножение, одно сложение и одна операция «И»), что мы можем использовать его в качестве примера.

```
#include <stdint.h>

// константы из книги Numerical Recipes
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}
```

Здесь две функции: одна используется для инициализации внутреннего состояния, а вторая вызывается собственно для генерации псевдослучайных чисел.

Мы видим что в алгоритме применяются две константы. Они взяты из [William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery, *Numerical Recipes*, (2007)]. Определим их используя выражение Си/Си++ `#define`. Это макрос. Разница между макросом в Си/Си++ и константой в том, что все макросы заменяются на значения препроцессором Си/Си++ и они не занимают места в памяти как переменные. А константы, напротив, это переменные только для чтения. Можно взять указатель (или адрес) переменной-константы, но это невозможно сделать с макросом.

Последняя операция «И» нужна, потому что согласно стандарту Си `my_rand()` должна возвращать значение в пределах 0..32767. Если вы хотите получать 32-битные псевдослучайные значения, просто уберите последнюю операцию «И».

21.1. x86

Листинг 21.1: Оптимизирующий MSVC 2013

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
```

¹Вихрь Мерсенна куда лучше

21.2. X64

```
_srand PROC
    mov     eax, DWORD PTR _init$[esp-4]
    mov     DWORD PTR _rand_state, eax
    ret     0
_srand ENDP

_TEXT SEGMENT
_rand PROC
    imul    eax, DWORD PTR _rand_state, 1664525
    add     eax, 1013904223          ; 3c6ef35fH
    mov     DWORD PTR _rand_state, eax
    and     eax, 32767              ; 00007fffH
    ret     0
_rand ENDP

_TEXT ENDS
```

Вот мы это и видим: обе константы встроены в код. Память для них не выделяется. Функция `my_srand()` просто копирует входное значение во внутреннюю переменную `rand_state`.

`my_rand()` берет её, вычисляет следующее состояние `rand_state`, обрезает его и оставляет в регистре EAX.

Неоптимизированная версия побольше:

Листинг 21.2: Неоптимизирующий MSVC 2013

```
_BSS SEGMENT
_rand_state DD 01H DUP (?)
_BSS ENDS

_init$ = 8
_srand PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _init$[ebp]
    mov     DWORD PTR _rand_state, eax
    pop     ebp
    ret     0
_srand ENDP

_TEXT SEGMENT
_rand PROC
    push    ebp
    mov     ebp, esp
    imul    eax, DWORD PTR _rand_state, 1664525
    mov     DWORD PTR _rand_state, eax
    mov     ecx, DWORD PTR _rand_state
    add     ecx, 1013904223          ; 3c6ef35fH
    mov     DWORD PTR _rand_state, ecx
    mov     eax, DWORD PTR _rand_state
    and     eax, 32767              ; 00007fffH
    pop     ebp
    ret     0
_rand ENDP

_TEXT ENDS
```

21.2. x64

Версия для x64 почти такая же, и использует 32-битные регистры вместо 64-битных (потому что мы работаем здесь с переменными типа `int`). Но функция `my_srand()` берет входной аргумент из регистра `ECX`, а не из стека:

Листинг 21.3: Оптимизирующий MSVC 2013 x64

```
_BSS SEGMENT
rand_state DD 01H DUP (?)
_BSS ENDS
```

21.3. 32-BIT ARM

```
init$ = 8
my_srand PROC
; ECX = входной аргумент
    mov     DWORD PTR rand_state, ecx
    ret     0
my_srand ENDP

_TEXT SEGMENT
my_rand PROC
    imul    eax, DWORD PTR rand_state, 1664525    ; 0019660dH
    add     eax, 1013904223                       ; 3c6ef35fH
    mov     DWORD PTR rand_state, eax
    and     eax, 32767                             ; 00007fffH
    ret     0
my_rand ENDP

_TEXT ENDS
```

GCC делает почти такой же код.

21.3. 32-bit ARM

Листинг 21.4: Оптимизирующий Keil 6/2013 (Режим ARM)

```
my_srand PROC
    LDR     r1, |L0.52| ; загрузить указатель на rand_state
    STR     r0, [r1, #0] ; сохранить rand_state
    BX      lr
ENDP

my_rand PROC
    LDR     r0, |L0.52| ; загрузить указатель на rand_state
    LDR     r2, |L0.56| ; загрузить RNG_a
    LDR     r1, [r0, #0] ; загрузить rand_state
    MUL     r1, r2, r1
    LDR     r2, |L0.60| ; загрузить RNG_c
    ADD     r1, r1, r2
    STR     r1, [r0, #0] ; сохранить rand_state
; AND с 0x7FFF:
    LSL     r0, r1, #17
    LSR     r0, r0, #17
    BX      lr
ENDP

|L0.52|
DCD        ||.data||
|L0.56|
DCD        0x0019660d
|L0.60|
DCD        0x3c6ef35f

        AREA ||.data||, DATA, ALIGN=2

rand_state
DCD        0x00000000
```

В ARM инструкцию невозможно встроить 32-битную константу, так что Keil-у приходится размещать их отдельно и дополнительно загружать. Вот еще что интересно: константу 0x7FFF также нельзя встроить. Поэтому Keil сдвигает `rand_state` влево на 17 бит и затем сдвигает вправо на 17 бит. Это аналогично Си/Си++-выражению $(rand\_state \ll 17) \gg 17$. Выглядит как бессмысленная операция, но тем не менее, что она делает это очищает старшие 17 бит, оставляя младшие 15 бит нетронутыми, и это наша цель, в конце концов.

Оптимизирующий Keil для режима Thumb делает почти такой же код.

Листинг 21.5: Оптимизирующий GCC 4.4.5 (IDA)

```

my_srand:
; записать $a0 в rand_state:
    lui    $v0, (rand_state >> 16)
    jr     $ra
    sw     $a0, rand_state

my_rand:
; загрузить rand_state в $v0:
    lui    $v1, (rand_state >> 16)
    lw     $v0, rand_state
    or     $at, $zero ; load delay slot
; умножить rand_state в $v0 на 1664525 (RNG_a):
    sll    $a1, $v0, 2
    sll    $a0, $v0, 4
    addu   $a0, $a1, $a0
    sll    $a1, $a0, 6
    subu   $a0, $a1, $a0
    addu   $a0, $v0
    sll    $a1, $a0, 5
    addu   $a0, $a1
    sll    $a0, 3
    addu   $v0, $a0, $v0
    sll    $a0, $v0, 2
    addu   $v0, $a0
; прибавить 1013904223 (RNG_c)
; инструкция LI объединена в IDA из LUI и ORI
    li     $a0, 0x3C6EF35F
    addu   $v0, $a0
; сохранить в rand_state:
    sw     $v0, (rand_state & 0xFFFF)($v1)
    jr     $ra
    andi   $v0, 0x7FFF ; branch delay slot

```

Ух, мы видим здесь только одну константу (0x3C6EF35F или 1013904223). Где же вторая (1664525)?

Похоже, умножение на 1664525 сделано только при помощи сдвигов и прибавлений! Проверим эту версию:

```

#define RNG_a 1664525

int f (int a)
{
    return a*RNG_a;
}

```

Листинг 21.6: Оптимизирующий GCC 4.4.5 (IDA)

```

f:
    sll    $v1, $a0, 2
    sll    $v0, $a0, 4
    addu   $v0, $v1, $v0
    sll    $v1, $v0, 6
    subu   $v0, $v1, $v0
    addu   $v0, $a0
    sll    $v1, $v0, 5
    addu   $v0, $v1
    sll    $v0, 3
    addu   $a0, $v0, $a0
    sll    $v0, $a0, 2
    jr     $ra
    addu   $v0, $a0, $v0 ; branch delay slot

```

Действительно!

21.4.1. Перемещения в MIPS («relocs»)

Ещё поговорим о том, как на самом деле происходят операции загрузки из памяти и запись в память. Листинги здесь были сделаны в IDA, которая убирает немного деталей.

Запустим objdump дважды: чтобы получить дизассемблированный листинг и список перемещений:

Листинг 21.7: Оптимизирующий GCC 4.4.5 (objdump)

```
# objdump -D rand_03.o

...

00000000 <my_srand>:
   0:  3c020000      lui      v0,0x0
   4:  03e00008      jr       ra
   8:  ac440000      sw       a0,0(v0)

0000000c <my_rand>:
   c:  3c030000      lui      v1,0x0
  10:  8c620000      lw       v0,0(v1)
  14:  00200825      move     at,at
  18:  00022880      sll      a1,v0,0x2
  1c:  00022100      sll      a0,v0,0x4
  20:  00a42021      addu     a0,a1,a0
  24:  00042980      sll      a1,a0,0x6
  28:  00a42023      subu     a0,a1,a0
  2c:  00822021      addu     a0,a0,v0
  30:  00042940      sll      a1,a0,0x5
  34:  00852021      addu     a0,a0,a1
  38:  000420c0      sll      a0,a0,0x3
  3c:  00821021      addu     v0,a0,v0
  40:  00022080      sll      a0,v0,0x2
  44:  00441021      addu     v0,v0,a0
  48:  3c043c6e      lui      a0,0x3c6e
  4c:  3484f35f      ori      a0,a0,0xf35f
  50:  00441021      addu     v0,v0,a0
  54:  ac620000      sw       v0,0(v1)
  58:  03e00008      jr       ra
  5c:  30427fff      andi     v0,v0,0x7fff

...

# objdump -r rand_03.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET      TYPE          VALUE
00000000 R_MIPS_HI16      .bss
00000008 R_MIPS_LO16      .bss
0000000c R_MIPS_HI16      .bss
00000010 R_MIPS_LO16      .bss
00000054 R_MIPS_LO16      .bss

...
```

Рассмотрим два перемещения для функции `my_srand()`. Первое, для адреса 0, имеет тип `R_MIPS_HI16`, и второе, для адреса 8, имеет тип `R_MIPS_LO16`. Это значит, что адрес начала сегмента `.bss` будет записан в инструкцию по адресу 0 (старшая часть адреса) и по адресу 8 (младшая часть адреса). Ведь переменная `rand_state` находится в самом начале сегмента `.bss`. Так что мы видим нули в операндах инструкций `LUI` и `SW` потому что там пока ничего нет — компилятор не знает, что туда записать. Линкер это исправит и старшая часть адреса будет записана в операнд инструкции `LUI` и младшая часть адреса — в операнд инструкции `SW`. `SW` просуммирует младшую часть адреса и то что находится в регистре `$V0` (там старшая часть).

Та же история и с функцией `my_rand()`: перемещение `R_MIPS_HI16` указывает линкеру записать старшую часть адреса сегмента `.bss` в инструкцию `LUI`. Так что старшая часть адреса переменной `rand_state` находится в регистре `$V1`. Инструкция `LW` по адресу 0x10 просуммирует старшую и младшую часть и загрузит значение переменной `rand_state` в

\$V1. Инструкция `SW` по адресу `0x54` также просуммирует и затем запишет новое значение в глобальную переменную `rand_state`.

IDA обрабатывает перемещения при загрузке, и таким образом эти детали скрываются. Но мы должны о них помнить.

21.5. Версия этого примера для многопоточной среды

Версия примера для многопоточной среды будет рассмотрена позже: [66.1](#) (стр. [680](#)).

Глава 22

Структуры

В принципе, структура в Си/Си++ это, с некоторыми допущениями, просто всегда лежащий рядом, и в той же последовательности, набор переменных, не обязательно одного типа ¹.

22.1. MSVC: Пример SYSTEMTIME

Возьмем, к примеру, структуру SYSTEMTIME² из win32 описывающую время.

Она объявлена так:

Листинг 22.1: WinBase.h

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Пишем на Си функцию для получения текущего системного времени:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

    return;
};
```

Что в итоге (MSVC 2010):

Листинг 22.2: MSVC 2010 /GS-

```
_t$ = -16 ; size = 16
_main PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 16
```

¹ АКА «гетерогенный контейнер»

² MSDN: [SYSTEMTIME structure](#)

22.1. MSVC: ПРИМЕР SYSTEMTIME

```
lea    eax, DWORD PTR _t$[ebp]
push   eax
call   DWORD PTR __imp__GetSystemTime@4
movzx  ecx, WORD PTR _t$[ebp+12] ; wSecond
push   ecx
movzx  edx, WORD PTR _t$[ebp+10] ; wMinute
push   edx
movzx  eax, WORD PTR _t$[ebp+8] ; wHour
push   eax
movzx  ecx, WORD PTR _t$[ebp+6] ; wDay
push   ecx
movzx  edx, WORD PTR _t$[ebp+2] ; wMonth
push   edx
movzx  eax, WORD PTR _t$[ebp] ; wYear
push   eax
push   OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
call   _printf
add    esp, 28
xor    eax, eax
mov    esp, ebp
pop    ebp
ret    0
_main  ENDP
```

Под структуру в стеке выделено 16 байт — именно столько будет `sizeof(WORD)*8` (в структуре 8 переменных с типом WORD).

Обратите внимание на тот факт, что структура начинается с поля `wYear`. Можно сказать, что в качестве аргумента для `GetSystemTime()`³ передается указатель на структуру SYSTEMTIME, но можно также сказать, что передается указатель на поле `wYear`, что одно и то же! `GetSystemTime()` пишет текущий год в тот WORD на который указывает переданный указатель, затем сдвигается на 2 байта вправо, пишет текущий месяц, и т.д., и т.д.

³MSDN: [GetSystemTime function](#)

22.1.1. OllyDbg

Компилируем этот пример в MSVC 2010 с ключами /GS- /MD и запускаем в OllyDbg. Открываем окна данных и стека по адресу, который передается в качестве первого аргумента в функцию `GetSystemTime()`, ждем пока эта функция исполнится, и видим следующее:

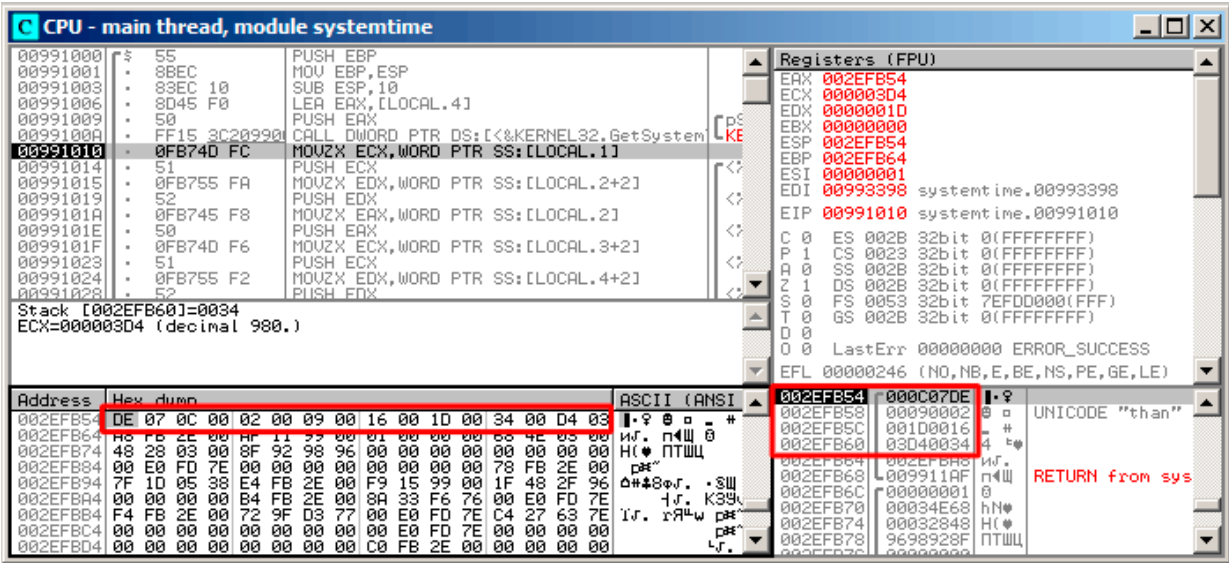


Рис. 22.1: OllyDbg: `GetSystemTime()` только что исполнялась

Точное системное время на моем компьютере, в которое исполнялась функция, это 9 декабря 2014, 22:29:52:

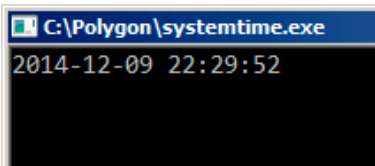


Рис. 22.2: OllyDbg: Вывод `printf()`

Таким образом, в окне данных мы видим следующие 16 байт:

```
DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 D4 03
```

Каждые два байта отражают одно поле структуры. А так как порядок байт (*endianness*) *little endian*, то в начале следует младший байт, затем старший. Следовательно, вот какие 16-битные числа сейчас записаны в памяти :

Шестнадцатеричное число	десятичное число	имя поля
0x07DE	2014	wYear
0x000C	12	wMonth
0x0002	2	wDayOfWeek
0x0009	9	wDay
0x0016	22	wHour
0x001D	29	wMinute
0x0034	52	wSecond
0x03D4	980	wMilliseconds

В окне стека, видны те же значения, только они сгруппированы как 32-битные значения .

Затем `printf()` просто берет нужные значения и выводит их на консоль .

Некоторые поля `printf()` не выводит (`wDayOfWeek` и `wMilliseconds`), но они находятся в памяти и доступны для использования.

22.1.2. Замена структуры массивом

Тот факт, что поля структуры — это просто переменные расположенные рядом, легко проиллюстрировать следующим образом. Глядя на описание структуры `SYSTEMTIME`, можно переписать этот простой пример так:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond */);

    return;
};
```

Компилятор немного ворчит:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types – from 'WORD [8]' to 'LPSYSTEMTIME'
```

Тем не менее, выдает такой код:

Листинг 22.3: Неоптимизирующий MSVC 2010

```
$SG78573 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_array$ = -16    ; size = 16
_main PROC
    push        ebp
    mov         ebp, esp
    sub         esp, 16
    lea         eax, DWORD PTR _array$[ebp]
    push        eax
    call        DWORD PTR __imp__GetSystemTime@4
    movzx       ecx, WORD PTR _array$[ebp+12] ; wSecond
    push        ecx
    movzx       edx, WORD PTR _array$[ebp+10] ; wMinute
    push        edx
    movzx       eax, WORD PTR _array$[ebp+8] ; wHour
    push        eax
    movzx       ecx, WORD PTR _array$[ebp+6] ; wDay
    push        ecx
    movzx       edx, WORD PTR _array$[ebp+2] ; wMonth
    push        edx
    movzx       eax, WORD PTR _array$[ebp] ; wYear
    push        eax
    push        OFFSET $SG78573
    call        _printf
    add         esp, 28
    xor         eax, eax
    mov         esp, ebp
    pop         ebp
    ret         0
_main ENDP
```

И это работает так же!

Любопытно что результат на ассемблере неотличим от предыдущего. Таким образом, глядя на этот код, никогда нельзя сказать с уверенностью, была ли там объявлена структура, либо просто набор переменных.

Тем не менее, никто в здравом уме делать так не будет. Потому что это неудобно. К тому же, иногда, поля в структуре могут меняться разработчиками, переставляться местами, и т.д.

С OllyDbg этот пример изучать не будем, потому что он будет точно такой же, как и в случае со структурой.

22.2. Выделяем место для структуры через malloc()

Однако, бывает и так, что проще хранить структуры не в стеке, а в [куче](#):

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
};
```

Скомпилируем на этот раз с оптимизацией (/Ox) чтобы было проще увидеть то, что нам нужно.

Листинг 22.4: Оптимизирующий MSVC

```
_main      PROC
    push    esi
    push    16
    call    _malloc
    add     esp, 4
    mov     esi, eax
    push    esi
    call    DWORD PTR __imp__GetSystemTime@4
    movzx   eax, WORD PTR [esi+12] ; wSecond
    movzx   ecx, WORD PTR [esi+10] ; wMinute
    movzx   edx, WORD PTR [esi+8] ; wHour
    push    eax
    movzx   eax, WORD PTR [esi+6] ; wDay
    push    ecx
    movzx   ecx, WORD PTR [esi+2] ; wMonth
    push    edx
    movzx   edx, WORD PTR [esi] ; wYear
    push    eax
    push    ecx
    push    edx
    push    OFFSET $SG78833
    call    _printf
    push    esi
    call    _free
    add     esp, 32
    xor     eax, eax
    pop     esi
    ret     0
_main      ENDP
```

Итак, `sizeof(SYSTEMTIME) = 16`, именно столько байт выделяется при помощи `malloc()`. Она возвращает указатель на только что выделенный блок памяти в `EAX`, который копируется в `ESI`. Win32 функция `GetSystemTime()` обязуется сохранить состояние `ESI`, поэтому здесь оно нигде не сохраняется и продолжает использоваться после вызова `GetSystemTime()`.

Новая инструкция — `MOVZX` (*Move with Zero eXtend*). Она нужна почти там же где и `MOVSX`, только всегда очищает остальные биты в 0. Дело в том, что `printf()` требует 32-битный тип `int`, а в структуре лежит `WORD` — это 16-битный беззнаковый тип. Поэтому копируя значение из `WORD` в `int`, нужно очистить биты от 16 до 31, иначе там будет просто случайный мусор, оставшийся от предыдущих действий с регистрами.

22.3. UNIX: STRUCT TM

В этом примере можно также представить структуру как массив 8-и WORD-ов:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};
```

Получим такое:

Листинг 22.5: Оптимизирующий MSVC

```
$SG78594 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
```

```
_main  PROC
        push    esi
        push    16
        call    _malloc
        add     esp, 4
        mov     esi, eax
        push    esi
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   eax, WORD PTR [esi+12]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
        push    ecx
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
        add     esp, 32
        xor     eax, eax
        pop     esi
        ret     0
_main  ENDP
```

И снова мы получаем идентичный код, неотличимый от предыдущего. Но и снова нужно отметить, что в реальности так лучше не делать, если только вы не знаете точно, что вы делаете.

22.3. UNIX: struct tm

22.3.1. Linux

В Линуксе, для примера, возьмем структуру `tm` из `time.h`:

```

#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
};

```

Компилируем при помощи GCC 4.4.1:

Листинг 22.6: GCC 4.4.1

```

main proc near
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFF0h
    sub     esp, 40h
    mov     dword ptr [esp], 0 ; первый аргумент для time()
    call    time
    mov     [esp+3Ch], eax
    lea     eax, [esp+3Ch] ; берем указатель на то что вернула time()
    lea     edx, [esp+10h] ; по ESP+10h будет начинаться структура struct tm
    mov     [esp+4], edx ; передаем указатель на начало структуры
    mov     [esp], eax ; передаем указатель на результат time()
    call    localtime_r
    mov     eax, [esp+24h] ; tm_year
    lea     edx, [eax+76Ch] ; edx=eax+1900
    mov     eax, offset format ; "Year: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+20h] ; tm_mon
    mov     eax, offset aMonthD ; "Month: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+1Ch] ; tm_mday
    mov     eax, offset aDayD ; "Day: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+18h] ; tm_hour
    mov     eax, offset aHourD ; "Hour: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+14h] ; tm_min
    mov     eax, offset aMinutesD ; "Minutes: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+10h]
    mov     eax, offset aSecondsD ; "Seconds: %d\n"
    mov     [esp+4], edx ; tm_sec
    mov     [esp], eax
    call    printf
    leave
    retn

```


22.3. UNIX: STRUCT TM

```
main endp
```

К сожалению, по какой-то причине, IDA не сформировала названия локальных переменных в стеке. Но так как мы уже опытные реверсеры :) то можем обойтись и без этого в таком простом примере.

Обратите внимание на `lea edx, [eax+76Ch]` — эта инструкция прибавляет `0x76C` (1900) к `EAX`, но не модифицирует флаги. См. также соответствующий раздел об инструкции `LEA` (A.6.2 (стр. 917)).

GDB

Попробуем загрузить пример в GDB ⁴:

Листинг 22.7: GDB

```
dennis@ubuntuvm:~/polygon$ date
Mon Jun  2 18:10:37 EEST 2014
dennis@ubuntuvm:~/polygon$ gcc GCC_tm.c -o GCC_tm
dennis@ubuntuvm:~/polygon$ gdb GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/GCC_tm...(no debugging symbols found)...done.
(gdb) b printf
Breakpoint 1 at 0x8048330
(gdb) run
Starting program: /home/dennis/polygon/GCC_tm

Breakpoint 1, __printf (format=0x80485c0 "Year: %d\n") at printf.c:29
29  printf.c: No such file or directory.
(gdb) x/20x $esp
0xbffff0dc: 0x080484c3      0x080485c0      0x000007de      0x00000000
0xbffff0ec: 0x08048301      0x538c93ed      0x00000025      0x0000000a
0xbffff0fc: 0x00000012      0x00000002      0x00000005      0x00000072
0xbffff10c: 0x00000001      0x00000098      0x00000001      0x00002a30
0xbffff11c: 0x0804b090      0x08048530      0x00000000      0x00000000
(gdb)
```

Мы легко находим нашу структуру в стеке. Для начала, посмотрим, как она объявлена в *time.h*:

Листинг 22.8: time.h

```
struct tm
{
    int    tm_sec;
    int    tm_min;
    int    tm_hour;
    int    tm_mday;
    int    tm_mon;
    int    tm_year;
    int    tm_wday;
    int    tm_yday;
    int    tm_isdst;
};
```

Обратите внимание что здесь 32-битные *int* вместо *WORD* в *SYSTEMTIME*. Так что, каждое поле занимает 32-битное слово.

Вот поля нашей структуры в стеке:

⁴Результат работы *date* немного подправлен в целях демонстрации. Конечно же, в реальности, нельзя так быстро запустить GDB, чтобы значение секунд осталось бы таким же.

```

0xbffff0dc:    0x080484c3    0x080485c0    0x000007de    0x00000000
0xbffff0ec:    0x08048301    0x538c93ed    0x00000025 sec  0x0000000a min
0xbffff0fc:    0x00000012 hour 0x00000002 mday 0x00000005 mon  0x00000072 year
0xbffff10c:    0x00000001 wday 0x00000098 yday 0x00000001 isdst0x00002a30
0xbffff11c:    0x0804b090    0x08048530    0x00000000    0x00000000

```

Либо же, в виде таблицы:

Шестнадцатеричное число	десятичное число	имя поля
0x00000025	37	tm_sec
0x0000000a	10	tm_min
0x00000012	18	tm_hour
0x00000002	2	tm_mday
0x00000005	5	tm_mon
0x00000072	114	tm_year
0x00000001	1	tm_wday
0x00000098	152	tm_yday
0x00000001	1	tm_isdst

Как и в примере с SYSTEMTIME (22.1 (стр. 340)), здесь есть и другие поля, готовые для использования, но в нашем примере они не используются, например, tm_wday, tm_yday, tm_isdst.

22.3.2. ARM

Оптимизирующий Keil 6/2013 (Режим Thumb)

Этот же пример:

Листинг 22.9: Оптимизирующий Keil 6/2013 (Режим Thumb)

```

var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer  = -0xC

    PUSH    {LR}
    MOVS    R0, #0          ; timer
    SUB     SP, SP, #0x34
    BL      time
    STR     R0, [SP,#0x38+timer]
    MOV     R1, SP          ; tp
    ADD     R0, SP, #0x38+timer ; timer
    BL      localtime_r
    LDR     R1, =0x76C
    LDR     R0, [SP,#0x38+var_24]
    ADDS    R1, R0, R1
    ADR     R0, aYearD      ; "Year: %d\n"
    BL      __2printf
    LDR     R1, [SP,#0x38+var_28]
    ADR     R0, aMonthD     ; "Month: %d\n"
    BL      __2printf
    LDR     R1, [SP,#0x38+var_2C]
    ADR     R0, aDayD       ; "Day: %d\n"
    BL      __2printf
    LDR     R1, [SP,#0x38+var_30]
    ADR     R0, aHourD      ; "Hour: %d\n"
    BL      __2printf
    LDR     R1, [SP,#0x38+var_34]
    ADR     R0, aMinutesD   ; "Minutes: %d\n"
    BL      __2printf
    LDR     R1, [SP,#0x38+var_38]
    ADR     R0, aSecondsD   ; "Seconds: %d\n"
    BL      __2printf
    ADD     SP, SP, #0x34

```

Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

IDA «узнала» структуру `tm` (потому что IDA «знает» типы аргументов библиотечных функций, таких как `localtime_r()`), поэтому показала здесь обращения к отдельным элементам структуры и присвоила им имена.

Листинг 22.10: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

```

var_38 = -0x38
var_34 = -0x34

    PUSH {R7,LR}
    MOV  R7, SP
    SUB  SP, SP, #0x30
    MOVS R0, #0 ; time_t *
    BLX  _time
    ADD  R1, SP, #0x38+var_34 ; struct tm *
    STR  R0, [SP,#0x38+var_38]
    MOV  R0, SP ; time_t *
    BLX  _localtime_r
    LDR  R1, [SP,#0x38+var_34.tm_year]
    MOV  R0, 0xF44 ; "Year: %d\n"
    ADD  R0, PC ; char *
    ADDW R1, R1, #0x76C
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34.tm_mon]
    MOV  R0, 0xF3A ; "Month: %d\n"
    ADD  R0, PC ; char *
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34.tm_mday]
    MOV  R0, 0xF35 ; "Day: %d\n"
    ADD  R0, PC ; char *
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34.tm_hour]
    MOV  R0, 0xF2E ; "Hour: %d\n"
    ADD  R0, PC ; char *
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34.tm_min]
    MOV  R0, 0xF28 ; "Minutes: %d\n"
    ADD  R0, PC ; char *
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34]
    MOV  R0, 0xF25 ; "Seconds: %d\n"
    ADD  R0, PC ; char *
    BLX  _printf
    ADD  SP, SP, #0x30
    POP  {R7,PC}

...

00000000 tm      struc ; (sizeof=0x2C, standard type)
00000000 tm_sec   DCD ?
00000004 tm_min   DCD ?
00000008 tm_hour  DCD ?
0000000C tm_mday  DCD ?
00000010 tm_mon   DCD ?
00000014 tm_year  DCD ?
00000018 tm_wday  DCD ?
0000001C tm_yday  DCD ?
00000020 tm_isdst DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone  DCD ? ; offset
0000002C tm      ends

```

22.3.3. MIPS

```

1 main:
2
3 ; IDA не в курсе имен полей структуры, мы назвали их так вручную:
4
5 var_40      = -0x40
6 var_38      = -0x38
7 seconds     = -0x34
8 minutes     = -0x30
9 hour        = -0x2C
10 day         = -0x28
11 month       = -0x24
12 year        = -0x20
13 var_4       = -4
14
15         lui    $gp, (__gnu_local_gp >> 16)
16         addiu  $sp, -0x50
17         la     $gp, (__gnu_local_gp & 0xFFFF)
18         sw     $ra, 0x50+var_4($sp)
19         sw     $gp, 0x50+var_40($sp)
20         lw     $t9, (time & 0xFFFF)($gp)
21         or     $at, $zero ; load delay slot, NOP
22         jalr   $t9
23         move   $a0, $zero ; branch delay slot, NOP
24         lw     $gp, 0x50+var_40($sp)
25         addiu  $a0, $sp, 0x50+var_38
26         lw     $t9, (localtime_r & 0xFFFF)($gp)
27         addiu  $a1, $sp, 0x50+seconds
28         jalr   $t9
29         sw     $v0, 0x50+var_38($sp) ; branch delay slot
30         lw     $gp, 0x50+var_40($sp)
31         lw     $a1, 0x50+year($sp)
32         lw     $t9, (printf & 0xFFFF)($gp)
33         la     $a0, $LC0          # "Year: %d\n"
34         jalr   $t9
35         addiu  $a1, 1900 ; branch delay slot
36         lw     $gp, 0x50+var_40($sp)
37         lw     $a1, 0x50+month($sp)
38         lw     $t9, (printf & 0xFFFF)($gp)
39         lui    $a0, ($LC1 >> 16) # "Month: %d\n"
40         jalr   $t9
41         la     $a0, ($LC1 & 0xFFFF) # "Month: %d\n" ; branch delay slot
42         lw     $gp, 0x50+var_40($sp)
43         lw     $a1, 0x50+day($sp)
44         lw     $t9, (printf & 0xFFFF)($gp)
45         lui    $a0, ($LC2 >> 16) # "Day: %d\n"
46         jalr   $t9
47         la     $a0, ($LC2 & 0xFFFF) # "Day: %d\n" ; branch delay slot
48         lw     $gp, 0x50+var_40($sp)
49         lw     $a1, 0x50+hour($sp)
50         lw     $t9, (printf & 0xFFFF)($gp)
51         lui    $a0, ($LC3 >> 16) # "Hour: %d\n"
52         jalr   $t9
53         la     $a0, ($LC3 & 0xFFFF) # "Hour: %d\n" ; branch delay slot
54         lw     $gp, 0x50+var_40($sp)
55         lw     $a1, 0x50+minutes($sp)
56         lw     $t9, (printf & 0xFFFF)($gp)
57         lui    $a0, ($LC4 >> 16) # "Minutes: %d\n"
58         jalr   $t9
59         la     $a0, ($LC4 & 0xFFFF) # "Minutes: %d\n" ; branch delay slot
60         lw     $gp, 0x50+var_40($sp)
61         lw     $a1, 0x50+seconds($sp)
62         lw     $t9, (printf & 0xFFFF)($gp)
63         lui    $a0, ($LC5 >> 16) # "Seconds: %d\n"
64         jalr   $t9
65         la     $a0, ($LC5 & 0xFFFF) # "Seconds: %d\n" ; branch delay slot
66         lw     $ra, 0x50+var_4($sp)
67         or     $at, $zero ; load delay slot, NOP
68         jr     $ra
69         addiu  $sp, 0x50

```

22.3. UNIX: STRUCT TM

```
70
71 $LC0:      .ascii "Year: %d\n"<0>
72 $LC1:      .ascii "Month: %d\n"<0>
73 $LC2:      .ascii "Day: %d\n"<0>
74 $LC3:      .ascii "Hour: %d\n"<0>
75 $LC4:      .ascii "Minutes: %d\n"<0>
76 $LC5:      .ascii "Seconds: %d\n"<0>
```

Это тот пример, где branch delay slot-ы могут нас запутать. Например, в строке 35 есть инструкция «addiu \$a1, 1900», добавляющая 1900 к числу года. Но она выполняется перед исполнением соответствующей JALR в строке 34, не забывайте.

22.3.4. Структура как набор переменных

Чтобы проиллюстрировать то что структура — это просто набор переменных, лежащих в одном месте, переделаем немного пример, еще раз заглянув в описание структуры *tm* : листинг.22.8.

```
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday, tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
};
```

N.B. В `localtime_r` передается указатель именно на `tm_sec` , т.е. на первый элемент «структуры».

В итоге, и этот компилятор поворчит:

Листинг 22.12: GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from incompatible pointer type [enabled ↵
↳ by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of type 'int *'
```

Тем не менее, сгенерирует такое:

Листинг 22.13: GCC 4.7.3

```
main      proc near
var_30    = dword ptr -30h
var_2C    = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec    = dword ptr -18h
tm_min    = dword ptr -14h
tm_hour   = dword ptr -10h
tm_mday   = dword ptr -0Ch
tm_mon    = dword ptr -8
tm_year   = dword ptr -4

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 30h
```

```

call    __main
mov     [esp+30h+var_30], 0 ; arg 0
call    time
mov     [esp+30h+unix_time], eax
lea     eax, [esp+30h+tm_sec]
mov     [esp+30h+var_2C], eax
lea     eax, [esp+30h+unix_time]
mov     [esp+30h+var_30], eax
call    localtime_r
mov     eax, [esp+30h+tm_year]
add     eax, 1900
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_30], offset aYearD ; "Year: %d\n"
call    printf
mov     eax, [esp+30h+tm_mon]
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
call    printf
mov     eax, [esp+30h+tm_mday]
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_30], offset aDayD ; "Day: %d\n"
call    printf
mov     eax, [esp+30h+tm_hour]
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
call    printf
mov     eax, [esp+30h+tm_min]
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
call    printf
mov     eax, [esp+30h+tm_sec]
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
call    printf
leave
retn
main    endp

```

Этот код почти идентичен уже рассмотренному, и нельзя сказать, была ли структура в оригинальном исходном коде либо набор переменных.

И это работает. Однако, в реальности так лучше не делать. Обычно, неоптимизирующий компилятор располагает переменные в локальном стеке в том же порядке, в котором они объявляются в функции. Тем не менее, никакой гарантии нет.

Кстати, какой-нибудь другой компилятор может предупредить, что переменные `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min`, но не `tm_sec`, используются без инициализации. Действительно, ведь компилятор не знает что они будут заполнены при вызове функции `localtime_r()`.

Мы выбрали именно этот пример для иллюстрации, потому что все члены структуры имеют тип `int`. Это не сработает, если поля структуры будут иметь размер 16 бит (`WORD`), как в случае со структурой `SYSTEMTIME` — `GetSystemTime()` заполнит их неверно (потому что локальные переменные выровнены по 32-битной границе). Читайте об этом в следующей секции: «Упаковка полей в структуре» (22.4 (стр. 355)).

Так что, структура — это просто набор переменных лежащих в одном месте, рядом. Можно было бы сказать, что структура — это инструкция компилятору, заставляющая его удерживать переменные в одном месте. Кстати, когда-то, в очень ранних версиях Си (перед 1972) структур не было вовсе [Dennis M. Ritchie, *The development of the C language*, (1993)]⁵.

Здесь нет примера с отладчиком: потому что он будет полностью идентичным тому, что вы уже видели.

22.3.5. Структура как массив 32-битных слов

```

#include <stdio.h>
#include <time.h>

void main()

```

⁵Также доступно здесь: <http://go.yurichev.com/17264>

22.3. UNIX: STRUCT TM

```
{
    struct tm t;
    time_t unix_time;
    int i;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        int tmp=((int*)&t)[i];
        printf ("0x%08X (%d)\n", tmp, tmp);
    };
};
```

Мы просто приводим (*cast*) указатель на структуру к массиву *int*-ов. И это работает! Запускаем пример 23:51:45 26-July-2014.

```
0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
0x00000001 (1)
```

Переменные здесь в том же порядке, в котором они перечислены в определении структуры: [22.8](#) (стр. [347](#)).

Вот как это компилируется:

Листинг 22.14: Оптимизирующий GCC 4.8.1

```
main      proc near
          push    ebp
          mov     ebp, esp
          push    esi
          push    ebx
          and     esp, 0FFFFFFF0h
          sub     esp, 40h
          mov     dword ptr [esp], 0 ; timer
          lea     ebx, [esp+14h]
          call    _time
          lea     esi, [esp+38h]
          mov     [esp+4], ebx      ; tp
          mov     [esp+10h], eax
          lea     eax, [esp+10h]
          mov     [esp], eax       ; timer
          call    _localtime_r
          nop
          lea     esi, [esi+0]     ; NOP
loc_80483D8:
; EBX здесь это указатель на структуру, ESI - указатель на её конец.
          mov     eax, [ebx]       ; загрузить 32-битное слово из массива
          add     ebx, 4           ; следующее поле в структуре
          mov     dword ptr [esp+4], offset a0x08xD ; "0x%08X (%d)\n"
          mov     dword ptr [esp], 1
          mov     [esp+0Ch], eax   ; передать значение в printf()
          mov     [esp+8], eax    ; передать значение в printf()
          call    __printf_chk
          cmp     ebx, esi        ; достигли конца структуры?
          jnz     short loc_80483D8 ; нет - тогда загрузить следующее значение
          lea     esp, [ebp-8]
          pop     ebx
          pop     esi
          pop     ebp
          retn
main      endp
```

22.3. UNIX: STRUCT TM

И действительно: место в локальном стеке в начале используется как структура, затем как массив.

Возможно даже модифицировать поля структуры через указатель.

И снова, это сомнительный хакерский способ, который не рекомендуется использовать в настоящем коде.

Упражнение

В качестве упражнения, попробуйте модифицировать (увеличить на 1) текущий номер месяца обращаясь со структурой как с массивом.

22.3.6. Структура как массив байт

Можно пойти еще дальше. Можно привести (*cast*) указатель к массиву байт и вывести его:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf ("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf ("\n");
    };
};
```

```
0x2D 0x00 0x00 0x00
0x33 0x00 0x00 0x00
0x17 0x00 0x00 0x00
0x1A 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0x72 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0xCE 0x00 0x00 0x00
0x01 0x00 0x00 0x00
```

Мы также запускаем этот пример в 23:51:45 26-July-2014 ⁶. Переменные точно такие же, как и в предыдущем выводе (22.3.5 (стр. 353)), и конечно, младший байт идет в самом начале, потому что это архитектура little-endian (32 (стр. 448)).

Листинг 22.15: Оптимизирующий GCC 4.8.1

```
main      proc near
          push    ebp
          mov     ebp, esp
          push    edi
          push    esi
          push    ebx
          and     esp, 0FFFFFFF0h
          sub     esp, 40h
          mov     dword ptr [esp], 0 ; timer
          lea     esi, [esp+14h]
          call    _time
          lea     edi, [esp+38h] ; struct end
          mov     [esp+4], esi ; tp
          mov     [esp+10h], eax
          lea     eax, [esp+10h]
```

⁶Время и дата такая же в целях демонстрации. Значения байт были подправлены.

22.4. УПАКОВКА ПОЛЕЙ В СТРУКТУРЕ

```
mov     [esp], eax      ; timer
call    _localtime_r
lea     esi, [esi+0]    ; NOP
; ESI здесь это указатель на структуру в локальном стеке. EDI это указатель на конец структуры.
loc_8048408:
xor     ebx, ebx        ; j=0

loc_804840A:
movzx   eax, byte ptr [esi+ebx] ; загрузить байт
add     ebx, 1          ; j=j+1
mov     dword ptr [esp+4], offset a0x02x ; "0x%02X "
mov     dword ptr [esp], 1
mov     [esp+8], eax     ; передать загруженный байт в printf()
call    __printf_chk
cmp     ebx, 4
jnz     short loc_804840A
; вывести символ перевода каретки (CR)
mov     dword ptr [esp], 0Ah ; c
add     esi, 4
call    _putchar
cmp     esi, edi        ; достигли конца структуры?
jnz     short loc_8048408 ; j=0
lea     esp, [ebp-0Ch]
pop     ebx
pop     esi
pop     edi
pop     ebp
retn
main    endp
```

22.4. Упаковка полей в структуре

Достаточно немаловажный момент, это упаковка полей в структурах⁷.

Возьмем простой пример:

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
};
```

Как видно, мы имеем два поля *char* (занимающий один байт) и еще два — *int* (по 4 байта).

⁷См. также: [Wikipedia: Выравнивание данных](#)

22.4.1. x86

Компилируется это все в:

Листинг 22.16: MSVC 2012 /GS- /Ob0

```

1  _tmp$ = -16
2  _main PROC
3      push    ebp
4      mov     ebp, esp
5      sub     esp, 16
6      mov     BYTE PTR _tmp$[ebp], 1      ; установить поле a
7      mov     DWORD PTR _tmp$[ebp+4], 2   ; установить поле b
8      mov     BYTE PTR _tmp$[ebp+8], 3    ; установить поле c
9      mov     DWORD PTR _tmp$[ebp+12], 4  ; установить поле d
10     sub     esp, 16                      ; выделить место для временной структуры
11     mov     eax, esp
12     mov     ecx, DWORD PTR _tmp$[ebp]    ; скопировать нашу структуру во временную
13     mov     DWORD PTR [eax], ecx
14     mov     edx, DWORD PTR _tmp$[ebp+4]
15     mov     DWORD PTR [eax+4], edx
16     mov     ecx, DWORD PTR _tmp$[ebp+8]
17     mov     DWORD PTR [eax+8], ecx
18     mov     edx, DWORD PTR _tmp$[ebp+12]
19     mov     DWORD PTR [eax+12], edx
20     call    _f
21     add     esp, 16
22     xor     eax, eax
23     mov     esp, ebp
24     pop     ebp
25     ret     0
26 _main ENDP
27
28 _s$ = 8 ; size = 16
29 ?f@@YAXUs@@@Z PROC ; f
30     push    ebp
31     mov     ebp, esp
32     mov     eax, DWORD PTR _s$[ebp+12]
33     push    eax
34     movsx   ecx, BYTE PTR _s$[ebp+8]
35     push    ecx
36     mov     edx, DWORD PTR _s$[ebp+4]
37     push    edx
38     movsx   eax, BYTE PTR _s$[ebp]
39     push    eax
40     push    OFFSET $SG3842
41     call    _printf
42     add     esp, 20
43     pop     ebp
44     ret     0
45 ?f@@YAXUs@@@Z ENDP ; f
46 _TEXT ENDS

```

Кстати, мы передаем всю структуру, но в реальности, как видно, структура в начале копируется во временную структуру (выделение места под нее в стеке происходит в строке 10, а все 4 поля, по одному, копируются в строках 12 ... 19), затем передается только указатель на нее (или адрес). Структура копируется, потому что неизвестно, будет ли функция `f()` модифицировать структуру или нет. И если да, то структура внутри `main()` должна остаться той же. Мы могли бы использовать указатели на Си/Си++, и итоговый код был бы почти такой же, только копирования не было бы.

Мы видим здесь что адрес каждого поля в структуре выравнивается по 4-байтной границе. Так что каждый `char` здесь занимает те же 4 байта что и `int`. Зачем? Затем что процессору удобнее обращаться по таким адресам и кэшировать данные из памяти.

Но это не экономично по размеру данных.

Попробуем скомпилировать тот же исходник с опцией (`/Zp1`) (`/Zp[n] pack structures on n-byte boundary`).

Листинг 22.17: MSVC 2012 /GS- /Zp1

```

1  _main PROC
2      push    ebp

```

22.4. УПАКОВКА ПОЛЕЙ В СТРУКТУРЕ

```
3  mov     ebp, esp
4  sub     esp, 12
5  mov     BYTE PTR _tmp$[ebp], 1      ; установить поле a
6  mov     DWORD PTR _tmp$[ebp+1], 2   ; установить поле b
7  mov     BYTE PTR _tmp$[ebp+5], 3     ; установить поле c
8  mov     DWORD PTR _tmp$[ebp+6], 4   ; установить поле d
9  sub     esp, 12                     ; выделить место для временной структуры
10 mov     eax, esp
11 mov     ecx, DWORD PTR _tmp$[ebp]   ; скопировать 10 байт
12 mov     DWORD PTR [eax], ecx
13 mov     edx, DWORD PTR _tmp$[ebp+4]
14 mov     DWORD PTR [eax+4], edx
15 mov     cx, WORD PTR _tmp$[ebp+8]
16 mov     WORD PTR [eax+8], cx
17 call    _f
18 add     esp, 12
19 xor     eax, eax
20 mov     esp, ebp
21 pop     ebp
22 ret     0
23 _main   ENDP
24
25 _TEXT   SEGMENT
26 _s$ = 8 ; size = 10
27 ?f@@YAXUs@@@Z PROC      ; f
28     push    ebp
29     mov     ebp, esp
30     mov     eax, DWORD PTR _s$[ebp+6]
31     push    eax
32     movsx   ecx, BYTE PTR _s$[ebp+5]
33     push    ecx
34     mov     edx, DWORD PTR _s$[ebp+1]
35     push    edx
36     movsx   eax, BYTE PTR _s$[ebp]
37     push    eax
38     push    OFFSET $SG3842
39     call    _printf
40     add     esp, 20
41     pop     ebp
42     ret     0
43 ?f@@YAXUs@@@Z ENDP      ; f
```

Теперь структура занимает 10 байт и все *char* занимают по байту. Что это дает? Экономия места. Недостаток — процессор будет обращаться к этим полям не так эффективно по скорости, как мог бы.

Структура так же копируется в `main()`. Но не по одному полю, а 10 байт, при помощи трех пар `MOV`. Почему не 4? Компилятор рассудил, что будет лучше скопировать 10 байт при помощи 3 пар `MOV`, чем копировать два 32-битных слова и два байта при помощи 4 пар `MOV`. Кстати, подобная реализация копирования при помощи `MOV` взамен вызова функции `memcpy()`, например, это очень распространенная практика, потому что это в любом случае работает быстрее чем вызов `memcpy()` — если речь идет о коротких блоках, конечно: 44.1.5 (стр. 501).

Как нетрудно догадаться, если структура используется много в каких исходниках и объектных файлах, все они должны быть откомпилированы с одним и тем же соглашением об упаковке структур.

Помимо ключа `MSVC /Zp`, указывающего, по какой границе упаковывать поля структур, есть также опция компилятора `#pragma pack`, её можно указывать прямо в исходнике. Это справедливо и для `MSVC`⁸ и `GCC`⁹.

Давайте теперь вернемся к `SYSTEMTIME`, которая состоит из 16-битных полей. Откуда наш компилятор знает что их надо паковать по однобайтной границе?

В файле `WinNT.h` попадаете такое:

Листинг 22.18: WinNT.h

```
#include "pshpack1.h"
```

И такое:

⁸MSDN: Working with Packing Structures

⁹Structure-Packing Pragma

```
#include "pshpack4.h"           // 4 byte packing is the default
```

Сам файл PshPack1.h выглядит так:

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

Собственно, так и задается компилятору, как паковать объявленные после `#pragma pack` структуры.

Попробуем в OllyDbg наш пример, где поля выровнены по умолчанию (4 байта) :

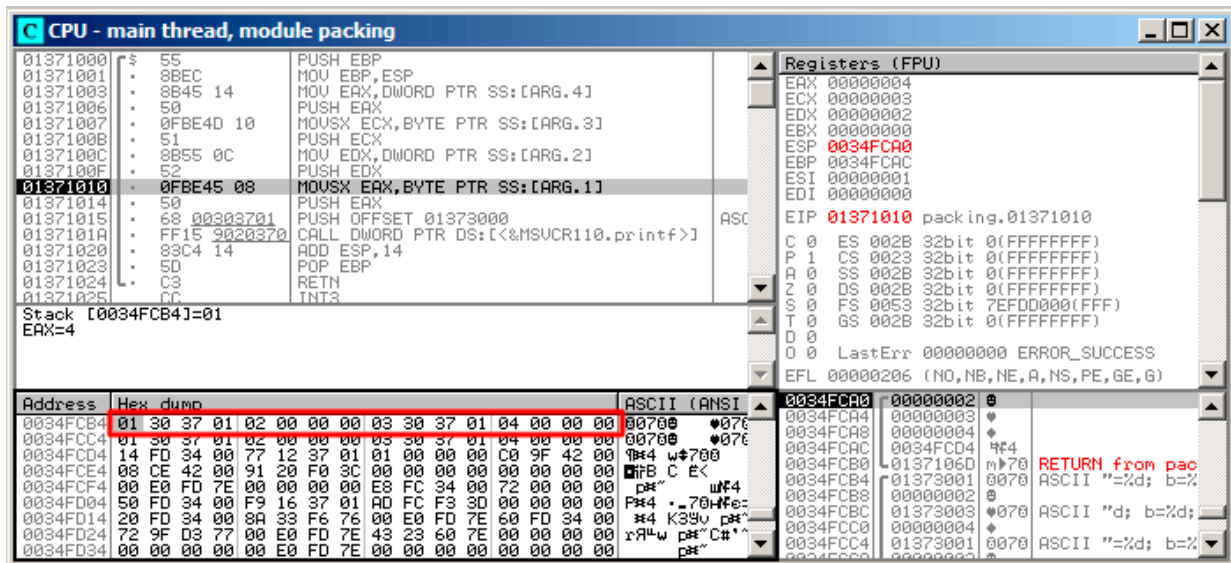


Рис. 22.3: OllyDbg: Перед исполнением `printf()`

В окне данных видим наши четыре поля. Вот только, откуда взялись случайные байты (0x30, 0x37, 0x01) рядом с первым (а) и третьим (с) полем? Если вернуться к листингу 22.16 (стр. 356), то увидите, что первое и третье поле имеет тип `char`, а следовательно, туда записывается только один байт, 1 и 3 соответственно (строки 6 и 8). Остальные три байта 32-битного слова не будут модифицироваться в памяти! А, следовательно, там остается случайный мусор. Этот мусор никак не будет влиять на работу `printf()`, потому что значения для нее готовятся при помощи инструкции `MOVEX`, которая загружает из памяти байты а не слова : листинг.22.16 (строки 34 и 38).

Кстати, здесь используется именно `MOVEX` (расширяющая знак), потому что тип `char` — знаковый по умолчанию в MSVC и GCC. Если бы здесь был тип `unsigned char` или `uint8_t`, то здесь была бы инструкция `MOVZX`.

OllyDbg + упаковка полей по границе в 1 байт

Здесь всё куда понятнее: 4 поля занимают 10 байт и значения сложены в памяти друг к другу

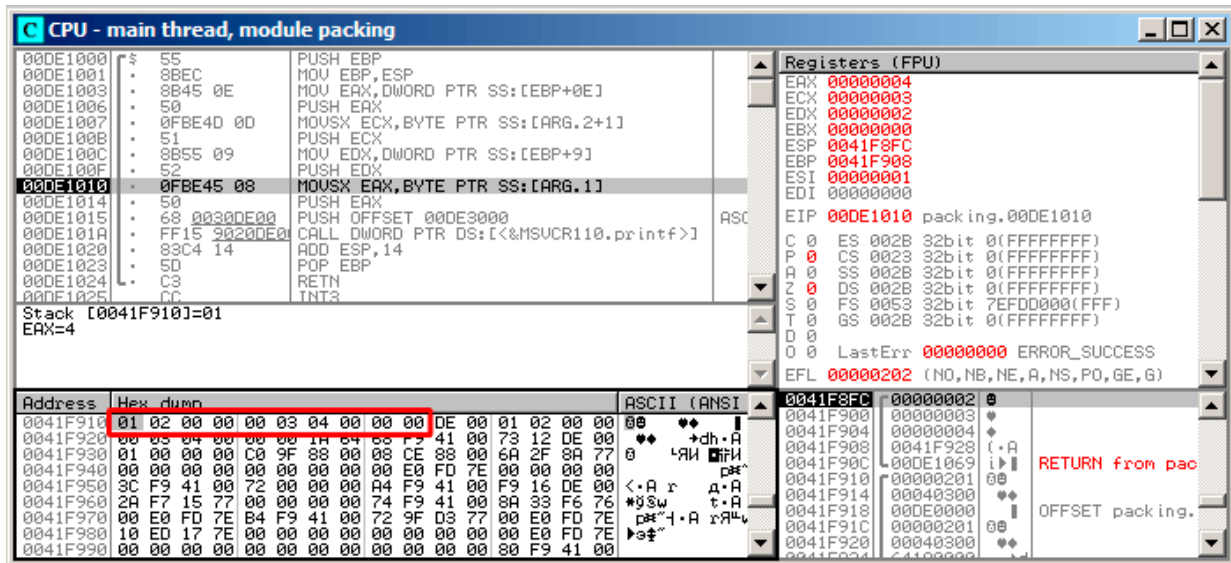


Рис. 22.4: OllyDbg: Перед исполнением `printf()`

22.4.2. ARM

Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 22.21: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
.text:0000003E      exit ; CODE XREF: f+16
.text:0000003E 05 B0      ADD     SP, SP, #0x14
.text:00000040 00 BD      POP     {PC}

.text:00000280      f
.text:00000280
.text:00000280      var_18 = -0x18
.text:00000280      a      = -0x14
.text:00000280      b      = -0x10
.text:00000280      c      = -0xC
.text:00000280      d      = -8
.text:00000280
.text:00000280 0F B5      PUSH    {R0-R3,LR}
.text:00000282 81 B0      SUB     SP, SP, #4
.text:00000284 04 98      LDR     R0, [SP,#16] ; d
.text:00000286 02 9A      LDR     R2, [SP,#8] ; b
.text:00000288 00 90      STR     R0, [SP]
.text:0000028A 68 46      MOV     R0, SP
.text:0000028C 03 7B      LDRB    R3, [R0,#12] ; c
.text:0000028E 01 79      LDRB    R1, [R0,#4] ; a
.text:00000290 59 A0      ADR     R0, aADBDCDDD ; "a=%d; b=%d; c=%d; d=%d\n"
.text:00000292 05 F0 AD FF  BL     __2printf
.text:00000296 D2 E6      B       exit
```

Как мы помним, здесь передается не указатель на структуру, а сама структура, а так как в ARM первые 4 аргумента функции передаются через регистры, то поля структуры передаются через `R0-R3`.

Инструкция `LDRB` загружает один байт из памяти и расширяет до 32-бит учитывая знак. Это то же что и инструкция `MOVSX` в x86. Она здесь применяется для загрузки полей `a` и `c` из структуры.

Еще что бросается в глаза, так это то что вместо эпилога функции, переход на эпилог другой функции! Действительно, то была совсем другая, не относящаяся к этой, функция, однако, она имела точно такой же эпилог (видимо, тоже хранила в

22.4. УПАКОВКА ПОЛЕЙ В СТРУКТУРЕ

стеке 5 локальных переменных ($5 * 4 = 0x14$). К тому же, она находится рядом (обратите внимание на адреса). Действительно, нет никакой разницы, какой эпилог исполнять, если он работает так же, как нам нужно. Keil решил использовать часть другой функции, вероятно, из-за экономии. Эпилог занимает 4 байта, а переход — только 2.

ARM + Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

Листинг 22.22: Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

```
var_C = -0xC

    PUSH    {R7,LR}
    MOV     R7, SP
    SUB     SP, SP, #4
    MOV     R9, R1 ; b
    MOV     R1, R0 ; a
    MOVW    R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
    SXTB    R1, R1 ; prepare a
    MOVT.W  R0, #0
    STR     R3, [SP,#0xC+var_C] ; place d to stack for printf()
    ADD     R0, PC ; format-string
    SXTB    R3, R2 ; prepare c
    MOV     R2, R9 ; b
    BLX     _printf
    ADD     SP, SP, #4
    POP     {R7,PC}
```

SXTB (*Signed Extend Byte*) это также аналог **MOVSX** в x86. Всё остальное — так же.

22.4.3. MIPS

Листинг 22.23: Оптимизирующий GCC 4.4.5 (IDA)

```
1 f:
2
3 var_18      = -0x18
4 var_10      = -0x10
5 var_4       = -4
6 arg_0       = 0
7 arg_4       = 4
8 arg_8       = 8
9 arg_C       = 0xC
10
11 ; $a0=s.a
12 ; $a1=s.b
13 ; $a2=s.c
14 ; $a3=s.d
15         lui     $gp, (__gnu_local_gp >> 16)
16         addiu   $sp, -0x28
17         la      $gp, (__gnu_local_gp & 0xFFFF)
18         sw      $ra, 0x28+var_4($sp)
19         sw      $gp, 0x28+var_10($sp)
20 ; prepare byte from 32-bit big-endian integer:
21         sra     $t0, $a0, 24
22         move    $v1, $a1
23 ; prepare byte from 32-bit big-endian integer:
24         sra     $v0, $a2, 24
25         lw      $t9, (printf & 0xFFFF)($gp)
26         sw      $a0, 0x28+arg_0($sp)
27         lui     $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d\n"
28         sw      $a3, 0x28+var_18($sp)
29         sw      $a1, 0x28+arg_4($sp)
30         sw      $a2, 0x28+arg_8($sp)
31         sw      $a3, 0x28+arg_C($sp)
32         la      $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d\n"
33         move    $a1, $t0
34         move    $a2, $v1
35         jalr    $t9
```

22.5. ВЛОЖЕННЫЕ СТРУКТУРЫ

```
36         move    $a3, $v0 ; branch delay slot
37         lw      $ra, 0x28+var_4($sp)
38         or      $at, $zero ; load delay slot, NOP
39         jr      $ra
40         addiu   $sp, 0x28 ; branch delay slot
41
42 $LC0:      .ascii "a=%d; b=%d; c=%d; d=%d\n"<0>
```

Поля структуры приходят в регистрах \$A0..\$A3 и затем перетасовываются в регистры \$A1..\$A4 для `printf()`. Но здесь есть две инструкции SRA («Shift Word Right Arithmetic»), которые готовят поля типа *char*. Почему? По умолчанию, MIPS это big-endian архитектура [32](#) (стр. [448](#)), и Debian Linux в котором мы работаем, также big-endian. Так что когда один байт расположен в 32-битном элементе структуры, он занимает биты 31..24. И когда переменную типа *char* нужно расширить до 32-битного значения, она должна быть сдвинута вправо на 24 бита. *char* это знаковый тип, так что здесь нужно использовать арифметический сдвиг вместо логического.

22.4.4. Еще кое-что

Передача структуры как аргумент функции (вместо передачи указателя на структуру) это то же что и передача всех полей структуры по одному. Если поля в структуре пакуются по умолчанию, то функцию `f()` можно переписать так:

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

И в итоге будет такой же код.

22.5. Вложенные структуры

Теперь, как насчет ситуаций, когда одна структура определена внутри другой структуры?

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};

void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
           s.a, s.b, s.c.a, s.c.b, s.d, s.e);
};

int main()
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
};
```


22.5. ВЛОЖЕННЫЕ СТРУКТУРЫ

...в этом случае, оба поля `inner_struct` просто будут располагаться между полями `a,b` и `d,e` в `outer_struct`.

Компилируем (MSVC 2010):

Листинг 22.24: Оптимизирующий MSVC 2010 /Ob0

```
$SG2802 DB      'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT        SEGMENT
_s$ = 8
_f          PROC
    mov     eax, DWORD PTR _s$[esp+16]
    movsx   ecx, BYTE PTR _s$[esp+12]
    mov     edx, DWORD PTR _s$[esp+8]
    push    eax
    mov     eax, DWORD PTR _s$[esp+8]
    push    ecx
    mov     ecx, DWORD PTR _s$[esp+8]
    push    edx
    movsx   edx, BYTE PTR _s$[esp+8]
    push    eax
    push    ecx
    push    edx
    push    OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call    _printf
    add     esp, 28
    ret     0
_f          ENDP

_s$ = -24
_main       PROC
    sub     esp, 24
    push    ebx
    push    esi
    push    edi
    mov     ecx, 2
    sub     esp, 24
    mov     eax, esp
    mov     BYTE PTR _s$[esp+60], 1
    mov     ebx, DWORD PTR _s$[esp+60]
    mov     DWORD PTR [eax], ebx
    mov     DWORD PTR [eax+4], ecx
    lea     edx, DWORD PTR [ecx+98]
    lea     esi, DWORD PTR [ecx+99]
    lea     edi, DWORD PTR [ecx+2]
    mov     DWORD PTR [eax+8], edx
    mov     BYTE PTR _s$[esp+76], 3
    mov     ecx, DWORD PTR _s$[esp+76]
    mov     DWORD PTR [eax+12], esi
    mov     DWORD PTR [eax+16], ecx
    mov     DWORD PTR [eax+20], edi
    call    _f
    add     esp, 24
    pop     edi
    pop     esi
    xor     eax, eax
    pop     ebx
    add     esp, 24
    ret     0
_main      ENDP
```

Очень любопытный момент в том, что глядя на этот код на ассемблере, мы даже не видим, что была использована какая-то еще другая структура внутри этой! Так что, пожалуй, можно сказать, что все вложенные структуры в итоге разворачиваются в одну, *линейную* или *одномерную* структуру.

Конечно, если заменить объявление `struct inner_struct c;` на `struct inner_struct *c;` (объявляя таким образом указатель), ситуация будет совсем иная.

22.5.1. OllyDbg

Загружаем пример в OllyDbg и смотрим на `outer_struct` в памяти:

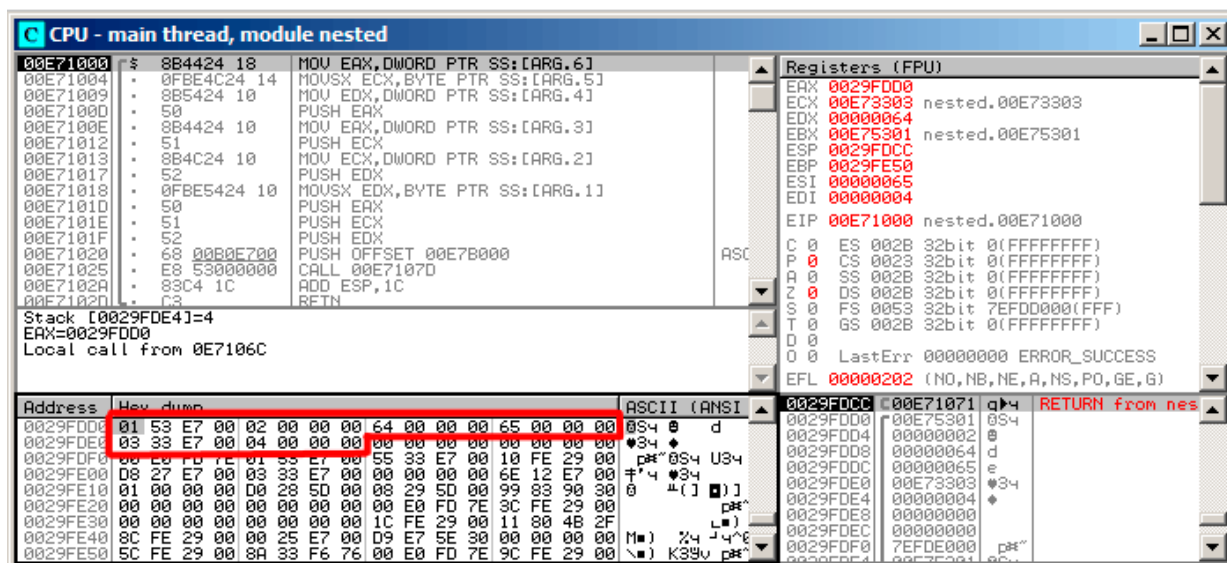


Рис. 22.5: OllyDbg: Перед исполнением `printf()`

Вот как расположены значения в памяти:

- `(outer_struct.a)` (байт) 1 + 3 байта случайного мусора;
- `(outer_struct.b)` (32-битное слово) 2;
- `(inner_struct.a)` (32-битное слово) 0x64 (100);
- `(inner_struct.b)` (32-битное слово) 0x65 (101);
- `(outer_struct.d)` (байт) 3 + 3 байта случайного мусора;
- `(outer_struct.e)` (32-битное слово) 4.

22.6. Работа с битовыми полями в структуре

22.6.1. Пример CPUID

Язык Си/Си++ позволяет указывать, сколько именно бит отвести для каждого поля структуры. Это удобно если нужно экономить место в памяти. К примеру, для переменной типа `bool` достаточно одного бита. Но, это не очень удобно, если нужна скорость.

Рассмотрим пример с инструкцией `CPUID`¹⁰. Эта инструкция возвращает информацию о том, какой процессор имеется в наличии и какие возможности он имеет.

Если перед исполнением инструкции в `EAX` будет 1, то `CPUID` вернет упакованную в `EAX` такую информацию о процессоре:

3:0 (4 бита)	Stepping
7:4 (4 бита)	Model
11:8 (4 бита)	Family
13:12 (2 бита)	Processor Type
19:16 (4 бита)	Extended Model
27:20 (8 бита)	Extended Family

MSVC 2010 имеет макрос для `CPUID`, а GCC 4.4.1 — нет. Поэтому для GCC сделаем эту функцию сами, используя его встроенный ассемблер¹¹.

¹⁰[wikipedia](#)

¹¹Подробнее о встроенном ассемблере GCC

```

#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
    asm volatile("cpuid":"=a"(*a), "=b"(*b), "=c"(*c), "=d"(*d):"a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b, 1);
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
};

```

После того как `CPUID` заполнит `EAX / EBX / ECX / EDX`, у нас они отразятся в массиве `b[]`. Затем, мы имеем указатель на структуру `CPUID_1_EAX`, и мы указываем его на значение `EAX` из массива `b[]`.

Иными словами, мы трактуем 32-битный `int` как структуру. Затем мы читаем отдельные биты из структуры.

MSVC

Компилируем в MSVC 2008 с опцией `/Ox`:

Листинг 22.25: Оптимизирующий MSVC 2008

```

_b$ = -16 ; size = 16
_main PROC
    sub     esp, 16
    push    ebx

    xor     ecx, ecx
    mov     eax, 1
    cpuid

```

22.6. РАБОТА С БИТОВЫМИ ПОЛЯМИ В СТРУКТУРЕ

```
push    esi
lea     esi, DWORD PTR _b$[esp+24]
mov     DWORD PTR [esi], eax
mov     DWORD PTR [esi+4], ebx
mov     DWORD PTR [esi+8], ecx
mov     DWORD PTR [esi+12], edx

mov     esi, DWORD PTR _b$[esp+24]
mov     eax, esi
and     eax, 15
push    eax
push    OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
call    _printf

mov     ecx, esi
shr     ecx, 4
and     ecx, 15
push    ecx
push    OFFSET $SG15436 ; 'model=%d', 0aH, 00H
call    _printf

mov     edx, esi
shr     edx, 8
and     edx, 15
push    edx
push    OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
call    _printf

mov     eax, esi
shr     eax, 12
and     eax, 3
push    eax
push    OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
call    _printf

mov     ecx, esi
shr     ecx, 16
and     ecx, 15
push    ecx
push    OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
call    _printf

shr     esi, 20
and     esi, 255
push    esi
push    OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
call    _printf
add     esp, 48
pop     esi

xor     eax, eax
pop     ebx

add     esp, 16
ret     0
_main   ENDP
```

Инструкция **SHR** сдвигает значение из **EAX** на то количество бит, которое нужно *пропустить*, то есть, мы игнорируем некоторые биты *справа*.

А инструкция **AND** очищает биты *слева* которые нам не нужны, или же, говоря иначе, она оставляет по маске только те биты в **EAX**, которые нам сейчас нужны.

MSVC + OllyDbg

Загрузим пример в OllyDbg и увидим, какие значения были установлены в EAX/EBX/ECX/EDX после исполнения CPUID:

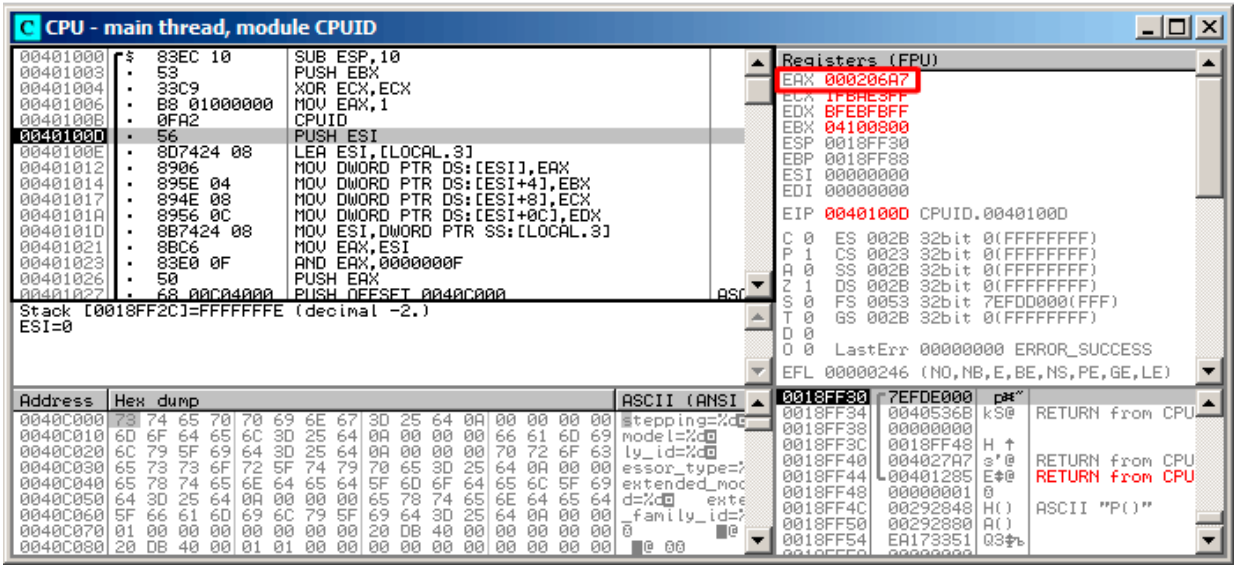


Рис. 22.6: OllyDbg: После исполнения CPUID

В EAX установлено 0x000206A7 (мой CPU – Intel Xeon E3-1220).
В двоичном виде это 00000000000000100000011010100111.

Вот как распределяются биты по полям в моем случае:

поле	в двоичном виде	в десятичном виде
reserved2	0000	0
extended_family_id	00000000	0
extended_model_id	0010	2
reserved1	00	0
processor_id	00	0
family_id	0110	6
model	1010	10
stepping	0111	7

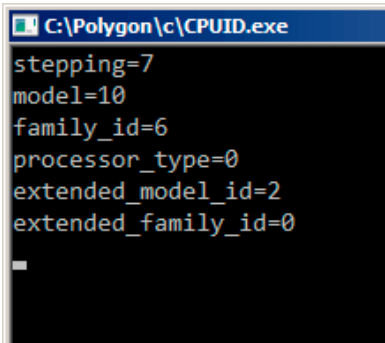


Рис. 22.7: OllyDbg: Результат работы

GCC

Попробуем GCC 4.4.1 с опцией -O3.

Листинг 22.26: Оптимизирующий GCC 4.4.1

```
main      proc near ; DATA XREF: _start+17
push     ebp
mov      ebp, esp
and      esp, 0FFFFFF0h
push     esi
```

```

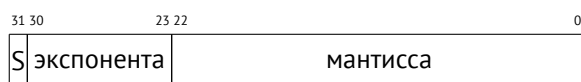
mov     esi, 1
push    ebx
mov     eax, esi
sub     esp, 18h
cuid
mov     esi, eax
and     eax, 0Fh
mov     [esp+8], eax
mov     dword ptr [esp+4], offset aSteppingD ; "stepping=%d\n"
mov     dword ptr [esp], 1
call    __printf_chk
mov     eax, esi
shr     eax, 4
and     eax, 0Fh
mov     [esp+8], eax
mov     dword ptr [esp+4], offset aModelD ; "model=%d\n"
mov     dword ptr [esp], 1
call    __printf_chk
mov     eax, esi
shr     eax, 8
and     eax, 0Fh
mov     [esp+8], eax
mov     dword ptr [esp+4], offset aFamily_idD ; "family_id=%d\n"
mov     dword ptr [esp], 1
call    __printf_chk
mov     eax, esi
shr     eax, 0Ch
and     eax, 3
mov     [esp+8], eax
mov     dword ptr [esp+4], offset aProcessor_type ; "processor_type=%d\n"
mov     dword ptr [esp], 1
call    __printf_chk
mov     eax, esi
shr     eax, 10h
shr     esi, 14h
and     eax, 0Fh
and     esi, 0FFh
mov     [esp+8], eax
mov     dword ptr [esp+4], offset aExtended_model ; "extended_model_id=%d\n"
mov     dword ptr [esp], 1
call    __printf_chk
mov     [esp+8], esi
mov     dword ptr [esp+4], offset unk_80486D0
mov     dword ptr [esp], 1
call    __printf_chk
add     esp, 18h
xor     eax, eax
pop     ebx
pop     esi
mov     esp, ebp
pop     ebp
retn
main     endp

```

Практически, то же самое. Единственное что стоит отметить это то, что GCC решил зачем-то объединить вычисление `extended_model_id` и `extended_family_id` в один блок, вместо того чтобы вычислять их перед соответствующим вызовом `printf()`.

22.6.2. Работа с типом float как со структурой

Как уже ранее указывалось в секции о FPU (18 (стр. 214)), и *float* и *double* содержат в себе *знак*, *мантиссу* и *экспоненту*. Однако, можем ли мы работать с этими полями напрямую? Попробуем с *float*.



(S – знак)

```

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // мантисса
    unsigned int exponent : 8; // экспонента + 0x3FF
    unsigned int sign : 1;      // бит знака
};

float f(float v)
{
    float f=v;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // установить отрицательный знак
    t.exponent=t.exponent+2; // умножить d на 2n (n здесь 2)

    memcpy (&f, &t, sizeof (float));

    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};

```

Структура `float_as_struct` занимает в памяти столько же места сколько и `float`, то есть 4 байта или 32 бита.

Далее мы выставляем во входящем значении отрицательный знак, а также прибавляя двойку к экспоненте, мы тем самым умножаем всё значение на 2^2 , то есть на 4.

Компилируем в MSVC 2008 без включенной оптимизации:

Листинг 22.27: Неоптимизирующий MSVC 2008

```

_t$ = -8 ; size = 4
_f$ = -4 ; size = 4
__in$ = 8 ; size = 4
?f@@YAMM@Z PROC ; f
    push    ebp
    mov     ebp, esp
    sub     esp, 8

    fld     DWORD PTR __in$[ebp]
    fstp    DWORD PTR _f$[ebp]

    push    4
    lea     eax, DWORD PTR _f$[ebp]
    push    eax
    lea     ecx, DWORD PTR _t$[ebp]
    push    ecx
    call    _memcpy
    add     esp, 12

    mov     edx, DWORD PTR _t$[ebp]
    or      edx, -2147483648 ; 80000000H – выставляем знак минус
    mov     DWORD PTR _t$[ebp], edx

    mov     eax, DWORD PTR _t$[ebp]
    shr     eax, 23          ; 00000017H – выкидываем мантиссу
    and     eax, 255         ; 000000ffH – оставляем здесь только экспоненту
    add     eax, 2           ; прибавляем к ней 2

```

22.6. РАБОТА С БИТОВЫМИ ПОЛЯМИ В СТРУКТУРЕ

```

and     eax, 255          ; 000000ffH
shl     eax, 23           ; 00000017H – поддвигаем результат на место бит 30:23
mov     ecx, DWORD PTR _t$[ebp]
and     ecx, -2139095041 ; 807fffffH – выкидываем экспоненту

; складываем оригинальное значение без экспоненты с новой только что вычисленной экспонентой
or      ecx, eax
mov     DWORD PTR _t$[ebp], ecx

push    4
lea     edx, DWORD PTR _t$[ebp]
push    edx
lea     eax, DWORD PTR _f$[ebp]
push    eax
call    _memcpy
add     esp, 12

fld     DWORD PTR _f$[ebp]

mov     esp, ebp
pop     ebp
ret     0
?f@@YAMM@Z ENDP ; f

```

Слегка избыточно. В версии скомпилированной с флагом `/Ox` нет вызовов `memcpy()`, там работа происходит сразу с переменной `f`. Но по неоптимизированной версии будет проще понять.

А что сделает GCC 4.4.1 с опцией `-O3`?

Листинг 22.28: Оптимизирующий GCC 4.4.1

```

; f(float)
public _Z1ff
_Z1ff proc near

var_4 = dword ptr -4
arg_0 = dword ptr 8

    push    ebp
    mov     ebp, esp
    sub     esp, 4
    mov     eax, [ebp+arg_0]
    or      eax, 80000000h ; выставить знак минуса
    mov     edx, eax
    and     eax, 807FFFFFh ; оставить в EAX только знак и мантиссу
    shr     edx, 23        ; подготовить экспоненту
    add     edx, 2         ; прибавить 2
    movzx   edx, dl        ; сбросить все биты кроме 7:0 в EAX в 0
    shl     edx, 23        ; подвинуть новую только что вычисленную экспоненту на свое место
    or      eax, edx       ; сложить новую экспоненту и оригинальное значение без экспоненты
    mov     [ebp+var_4], eax
    fld     [ebp+var_4]
    leave
    retn
_Z1ff endp

main public main
main proc near
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFF0h
    sub     esp, 10h
    fld     ds:dword_8048614 ; -4.936
    fstp    qword ptr [esp+8]
    mov     dword ptr [esp+4], offset asc_8048610 ; "%f\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    xor     eax, eax
    leave
    retn

```



```
main    endp
```

Да, функция `f()` в целом понятна. Однако, что интересно, еще при компиляции, не взирая на мешанину с полями структуры, GCC умудрился вычислить результат функции `f(1.234)` еще во время компиляции и сразу подставить его в аргумент для `printf()`!

22.7. Упражнения

- <http://challenges.re/71>
- <http://challenges.re/72>

Глава 23

Объединения (union)

union в Си/Си++ используется в основном для интерпертации переменной (или блока памяти) одного типа как переменной другого типа.

23.1. Пример генератора случайных чисел

Если нам нужны случайные значения с плавающей запятой в интервале от 0 до 1, самое простое это взять ПСЧ вроде Mersenne twister. Он выдает случайные беззнаковые 32-битные числа (иными словами, он выдает 32 случайных бита). Затем мы можем преобразовать это число в *float* и затем разделить на `RAND_MAX` (`0xFFFFFFFF` в данном случае) — полученное число будет в интервале от 0 до 1.

Но как известно, операция деления — это медленная операция. Да и вообще хочется избежать лишних операций с FPU. Сможем ли мы избежать деления?

Вспомним состав числа с плавающей запятой: это бит знака, биты мантиссы и биты экспоненты. Для получения случайного числа, нам нужно просто заполнить случайными битами все биты мантиссы!

Экспонента не может быть нулевой (иначе число с плавающей точкой будет денормализованным), так что в эти биты мы запишем `01111111` — это будет означать что экспонента равна единице. Далее заполняем мантиссу случайными битами, знак оставляем в виде 0 (что значит наше число положительное), и вуаля. Генерируемые числа будут в интервале от 1 до 2, так что нам еще нужно будет отнять единицу.

В моем примере¹ применяется очень простой линейный конгруэнтный генератор случайных чисел, выдающий 32-битные числа. Генератор инициализируется текущим временем в стиле UNIX.

Далее, тип *float* представляется в виде *union* — это конструкция Си/Си++ позволяющая интерпретировать часть памяти по-разному. В нашем случае, мы можем создать переменную типа *union* и затем обращаться к ней как к *float* или как к *uint32_t*. Можно сказать, что это хак, причем грязный.

Код целочисленного ПСЧ точно такой же, как мы уже рассматривали ранее: 21 (стр. 334). Так что и в скомпилированном виде этот код будет опущен.

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>

// определения, данные и ф-ции для целочисленного PRNG

// константы из книги Numerical Recipes
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;
uint32_t RNG_state; // глобальная переменная

void my_srand(uint32_t i)
{
    RNG_state=i;
};

uint32_t my_rand()
{
```

¹идея взята здесь: <http://go.yurichev.com/17308>

23.1. ПРИМЕР ГЕНЕРАТОРА СЛУЧАЙНЫХ ЧИСЕЛ

```
RNG_state=RNG_state*RNG_a+RNG_c;
return RNG_state;
};

// определения и ф-ции FPU PRNG:

union uint32_t_float
{
    uint32_t i;
    float f;
};

float float_rand()
{
    union uint32_t_float tmp;
    tmp.i=my_rand() & 0x007fffff | 0x3f800000;
    return tmp.f-1;
};

// тест

int main()
{
    my_srand(time(NULL)); // инициализация PRNG

    for (int i=0; i<100; i++)
        printf ("%f\n", float_rand());

    return 0;
};
```

23.1.1. x86

Листинг 23.1: Оптимизирующий MSVC 2010

```
$SG4238 DB      '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff000000000000r    ; 1

tv130 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
    push    ecx
    call    ?my_rand@@YAIXZ
; EAX=псевдослучайное значение
    and     eax, 8388607                      ; 007fffffH
    or      eax, 1065353216                    ; 3f800000H
; EAX=псевдослучайное значение & 0x007fffff | 0x3f800000
; сохранить его в локальном стеке
    mov     DWORD PTR _tmp$(esp+4), eax
; перезагрузить его как число с плавающей точкой:
    fld     DWORD PTR _tmp$(esp+4)
; вычесть 1.0:
    fsub    QWORD PTR __real@3ff0000000000000
; сохранить полученное значение в локальном стеке и перезагрузить его
    fstp    DWORD PTR tv130(esp+4) ; \ эти инструкции избыточны
    fld     DWORD PTR tv130(esp+4) ; /
    pop     ecx
    ret     0
?float_rand@@YAMXZ ENDP

_main PROC
    push    esi
    xor     eax, eax
    call    _time
    push    eax
    call    ?my_srand@@YAXI@Z
    add     esp, 4
```

23.1. ПРИМЕР ГЕНЕРАТОРА СЛУЧАЙНЫХ ЧИСЕЛ

```
    mov     esi, 100
$LL3@main:
    call    ?float_rand@@YAMXZ
    sub     esp, 8
    fstp    QWORD PTR [esp]
    push    OFFSET $SG4238
    call    _printf
    add     esp, 12
    dec     esi
    jne     SHORT $LL3@main
    xor     eax, eax
    pop     esi
    ret     0
_main     ENDP
```

Имена функций такие странные, потому что этот пример был скомпилирован как Си++, и это манглинг имен в Си++, мы будем рассматривать это позже: [52.1.1](#) (стр. [539](#)).

Если скомпилировать это в MSVC 2012, компилятор будет использовать SIMD-инструкции для FPU, читайте об этом здесь: [28.5](#) (стр. [437](#)).

23.1.2. MIPS

Листинг 23.2: Оптимизирующий GCC 4.4.5

```
float_rand:
var_10      = -0x10
var_4       = -4

    lui     $gp, (__gnu_local_gp >> 16)
    addiu   $sp, -0x20
    la      $gp, (__gnu_local_gp & 0xFFFF)
    sw      $ra, 0x20+var_4($sp)
    sw      $gp, 0x20+var_10($sp)
; вызвать my_rand():
    jal     my_rand
    or      $at, $zero ; branch delay slot, NOP
; $v0=32-битное псевдослучайное значение
    li      $v1, 0x7FFFFFFF
; $v1=0x7FFFFFFF
    and     $v1, $v0, $v1
; $v1=псевдослучайное значение & 0x7FFFFFFF
    lui     $a0, 0x3F80
; $a0=0x3F800000
    or      $v1, $a0
; $v1=псевдослучайное значение & 0x7FFFFFFF | 0x3F800000
; смысл этой инструкции всё так же трудно понять:
    lui     $v0, ($LC0 >> 16)
; загрузить 1.0 в $f0:
    lwc1    $f0, $LC0
; скопировать значение из $v1 в первый сопроцессор (в регистр $f2)
; это работает как побитовая копия, без всякого конвертирования
    mtc1    $v1, $f2
    lw      $ra, 0x20+var_4($sp)
; вычесть 1.0. оставить результат в $f0:
    sub.s   $f0, $f2, $f0
    jr      $ra
    addiu   $sp, 0x20 ; branch delay slot

main:
var_18      = -0x18
var_10      = -0x10
var_C       = -0xC
var_8       = -8
var_4       = -4
```

23.1. ПРИМЕР ГЕНЕРАТОРА СЛУЧАЙНЫХ ЧИСЕЛ

```
        lui    $gp, (__gnu_local_gp >> 16)
        addiu  $sp, -0x28
        la     $gp, (__gnu_local_gp & 0xFFFF)
        sw     $ra, 0x28+var_4($sp)
        sw     $s2, 0x28+var_8($sp)
        sw     $s1, 0x28+var_C($sp)
        sw     $s0, 0x28+var_10($sp)
        sw     $gp, 0x28+var_18($sp)
        lw     $t9, (time & 0xFFFF)($gp)
        or     $at, $zero ; load delay slot, NOP
        jalr   $t9
        move   $a0, $zero ; branch delay slot
        lui    $s2, ($LC1 >> 16) # "%f\n"
        move   $a0, $v0
        la     $s2, ($LC1 & 0xFFFF) # "%f\n"
        move   $s0, $zero
        jal    my_srand
        li     $s1, 0x64 # 'd' ; branch delay slot

loc_104:
        jal    float_rand
        addiu  $s0, 1
        lw     $gp, 0x28+var_18($sp)
; сконвертировать полученное из float_rand() значение в тип double (для printf()):
        cvt.d.s $f2, $f0
        lw     $t9, (printf & 0xFFFF)($gp)
        mfc1   $a3, $f2
        mfc1   $a2, $f3
        jalr   $t9
        move   $a0, $s2
        bne    $s0, $s1, loc_104
        move   $v0, $zero
        lw     $ra, 0x28+var_4($sp)
        lw     $s2, 0x28+var_8($sp)
        lw     $s1, 0x28+var_C($sp)
        lw     $s0, 0x28+var_10($sp)
        jr     $ra
        addiu  $sp, 0x28 ; branch delay slot

$LC1:    .ascii "%f\n"<0>
$LC0:    .float 1.0
```

Здесь снова зачем-то добавлена инструкция **LUI**, которая ничего не делает. Мы уже рассматривали этот артефакт ранее: [18.5.6](#) (стр. 226).

23.1.3. ARM (Режим ARM)

Листинг 23.3: Оптимизирующий GCC 4.6.3 (IDA)

```
float_rand
        STMFD  SP!, {R3,LR}
        BL     my_rand
; R0=псевдослучайное значение
        FLDS   S0, =1.0
; S0=1.0
        BIC    R3, R0, #0xFF000000
        BIC    R3, R3, #0x800000
        ORR    R3, R3, #0x3F800000
; R3=псевдослучайное значение & 0x007fffff | 0x3f800000
; копировать из R3 в FPU (регистр S15).
; это работает как побитовое копирование, без всякого конвертирования
        FMSR   S15, R3
; вычесть 1.0 и оставить результат в S0:
        FSUBS  S0, S15, S0
        LDMFD  SP!, {R3,PC}

flt_5C    DCFS 1.0
```

23.2. ВЫЧИСЛЕНИЕ МАШИННОГО ЭПСИЛОНА

```
main
    STMFD    SP!, {R4,LR}
    MOV     R0, #0
    BL      time
    BL      my_srand
    MOV     R4, #0x64 ; 'd'

loc_78
    BL      float_rand
; S0=псевдослучайное значение
    LDR     R0, =aF          ; "%f"
; сконvertировать значение типа float в значение типа double (это нужно для printf()):
    FCVTDS  D7, S0
; побитовое копирование из D7 в пару регистров R2/R3 (для printf()):
    FMRRD   R2, R3, D7
    BL      printf
    SUBS    R4, R4, #1
    BNE     loc_78
    MOV     R0, R4
    LDMFD   SP!, {R4,PC}

aF          DCB "%f",0xA,0
```

Мы также сделаем дамп в objdump и увидим что FPU-инструкции имеют немного другие имена чем в [IDA](#). Наверное, разработчики IDA и binutils пользовались разной документацией? Должно быть, будет полезно знать оба варианта названий инструкций.

Листинг 23.4: Оптимизирующий GCC 4.6.3 (objdump)

```
00000038 <float_rand>:
 38: e92d4008    push    {r3, lr}
 3c: ebfffffe    bl      10 <my_rand>
 40: ed9f0a05    vldr    s0, [pc, #20] ; 5c <float_rand+0x24>
 44: e3c034ff    bic     r3, r0, #-16777216 ; 0xff000000
 48: e3c33502    bic     r3, r3, #8388608 ; 0x800000
 4c: e38335fe    orr     r3, r3, #1065353216 ; 0x3f800000
 50: ee073a90    vmov    s15, r3
 54: ee370ac0    vsub.f32 s0, s15, s0
 58: e8bd8008    pop     {r3, pc}
 5c: 3f800000    svccc   0x00800000

00000000 <main>:
 0: e92d4010    push    {r4, lr}
 4: e3a00000    mov     r0, #0
 8: ebfffffe    bl      0 <time>
 c: ebfffffe    bl      0 <main>
10: e3a04064    mov     r4, #100 ; 0x64
14: ebfffffe    bl      38 <main+0x38>
18: e59f0018    ldr     r0, [pc, #24] ; 38 <main+0x38>
1c: eeb77ac0    vcvf.f64.f32 d7, s0
20: ec532b17    vmov    r2, r3, d7
24: ebfffffe    bl      0 <printf>
28: e2544001    subs    r4, r4, #1
2c: 1afffff8    bne     14 <main+0x14>
30: e1a00004    mov     r0, r4
34: e8bd8010    pop     {r4, pc}
38: 00000000    andeq   r0, r0, r0
```

Инструкции по адресам 0x5c в `float_rand()` и 0x38 в `main()` это (псевдо-)случайный мусор.

23.2. Вычисление машинного эпсилона

Машинный эпсилон — это самая маленькая гранула, с которой может работать FPU ². Чем больше бит выделено для числа с плавающей точкой, тем меньше машинный эпсилон. Это $2^{-23} = 1.19e-07$ для *float* и $2^{-52} = 2.22e-16$ для *double*. См.также: [статью в Wikipedia](#).

²В русскоязычной литературе встречается также термин «машинный ноль».

23.2. ВЫЧИСЛЕНИЕ МАШИННОГО ЭПСИЛОНА

Любопытно, что вычислить машинный эpsilon очень легко:

```
#include <stdio.h>
#include <stdint.h>

union uint_float
{
    uint32_t i;
    float f;
};

float calculate_machine_epsilon(float start)
{
    union uint_float v;
    v.f=start;
    v.i++;
    return v.f-start;
}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

Что мы здесь делаем это обходимся с мантиссой числа в формате IEEE 754 как с целочисленным числом и прибавляем единицу к нему. Итоговое число с плавающей точкой будет равно *starting_value + machine_epsilon*, так что нам нужно просто вычесть изначальное значение (используя арифметику с плавающей точкой) чтобы измерить, какое число отражает один бит в одинарной точности (*float*). *union* здесь нужен чтобы мы могли обращаться к числу в формате IEEE 754 как к обычному целочисленному. Прибавление 1 к нему на самом деле прибавляет 1 к *мантиссе* числа, хотя, нужно сказать, переполнение также возможно, что приведет к прибавлению единицы к экспоненте.

23.2.1. x86

Листинг 23.5: Оптимизирующий MSVC 2010

```
tv130 = 8
_v$ = 8
_start$ = 8
_calculate_machine_epsilon PROC
    fld     DWORD PTR _start$[esp-4]
    fst     DWORD PTR _v$[esp-4]      ; это лишняя инструкция
    inc     DWORD PTR _v$[esp-4]
    fsubr   DWORD PTR _v$[esp-4]
    fstp    DWORD PTR tv130[esp-4]    ; \ эта пара инструкций также лишняя
    fld     DWORD PTR tv130[esp-4]    ; /
    ret     0
_calculate_machine_epsilon ENDP
```

Вторая инструкция **FST** избыточна: нет необходимости сохранять входное значение в этом же месте (компилятор решил выделить переменную *v* в том же месте локального стека, где находится и входной аргумент). Далее оно инкрементируется при помощи **INC**, как если это обычная целочисленная переменная. Затем оно загружается в FPU как если это 32-битное число в формате IEEE 754, **FSUBR** делает остальную часть работы и результат в **ST0**. Последняя пара инструкций **FSTP** / **FLD** избыточна, но компилятор не оптимизировал её.

23.2.2. ARM64

Расширим этот пример до 64-бит:

```
#include <stdio.h>
#include <stdint.h>

typedef union
{
    uint64_t i;
    double d;
} uint_double;
```

23.3. БЫСТРОЕ ВЫЧИСЛЕНИЕ КВАДРАТНОГО КОРНЯ

```
double calculate_machine_epsilon(double start)
{
    uint_double v;
    v.d=start;
    v.i++;
    return v.d-start;
}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

В ARM64 нет инструкции для добавления числа к D-регистру в FPU, так что входное значение (пришедшее в D0) в начале копируется в GPR, инкрементируется, копируется в регистр FPU D1, затем происходит вычитание.

Листинг 23.6: Оптимизирующий GCC 4.9 ARM64

```
calculate_machine_epsilon:
    fmov    x0, d0      ; загрузить входное значение типа double в X0
    add     x0, x0, 1    ; X0++
    fmov    d1, x0      ; переместить его в регистр FPU
    fsub    d0, d1, d0   ; вычесть
    ret
```

Смотрите также этот пример скомпилированный под x64 с SIMD-инструкциями: 28.4 (стр. 436).

23.2.3. MIPS

Новая для нас здесь инструкция это MTC1 («Move To Coprocessor 1»), она просто переносит данные из GPR в регистры FPU.

Листинг 23.7: Оптимизирующий GCC 4.4.5 (IDA)

```
calculate_machine_epsilon:
    mfc1    $v0, $f12
    or      $at, $zero ; NOP
    addiu   $v1, $v0, 1
    mtc1    $v1, $f2
    jr      $ra
    sub.s   $f0, $f2, $f12 ; branch delay slot
```

23.2.4. Вывод

Трудно сказать, понадобится ли кому-то такая эквилибристика в реальном коде, но как уже было упомянуто много раз в этой книге, этот пример хорошо подходит для объяснения формата IEEE 754 и *union* в Си/Си++.

23.3. Быстрое вычисление квадратного корня

Вот где еще можно на практике применить трактовку типа *float* как целочисленного, это быстрое вычисление квадратного корня.

Листинг 23.8: Исходный код взят из Wikipedia: <http://go.yurichev.com/17364>

```
/* Assumes that float is in the IEEE 754 single precision floating point format
 * and that int is 32 bits. */
float sqrt_approx(float z)
{
    int val_int = *(int*)&z; /* Same bits, but as an int */
    /*
     * To justify the following code, prove that
     *
     * (((val_int / 2^m) - b) / 2) + b * 2^m = ((val_int - 2^m) / 2) + ((b + 1) / 2) * 2^m
     */
}
```


23.3. БЫСТРОЕ ВЫЧИСЛЕНИЕ КВАДРАТНОГО КОРНЯ

```
*
* where
*
* b = exponent bias
* m = number of mantissa bits
*
* .
*/

val_int -= 1 << 23; /* Subtract 2^m. */
val_int >>= 1; /* Divide by 2. */
val_int += 1 << 29; /* Add ((b + 1) / 2) * 2^m. */

return *(float*)&val_int; /* Interpret again as float */
}
```

В качестве упражнения, вы можете попробовать скомпилировать эту функцию и разобраться, как она работает.

Имеется также известный алгоритм быстрого вычисления $\frac{1}{\sqrt{x}}$. Алгоритм стал известным, вероятно потому, что был применен в Quake III Arena.

Описание алгоритма есть в Wikipedia: <http://go.yurichev.com/17361>.

Глава 24

Указатели на функции

Указатель на функцию, в целом, как и любой другой указатель, просто адрес, указывающий на начало функции в сегменте кода.

Это часто применяется для вызовов т.н. callback-функций ¹.

Известные примеры:

- `qsort()` ², `atexit()` ³ из стандартной библиотеки Си;
- сигналы в *NIX ОС⁴;
- запуск тредов: `CreateThread()` (win32), `pthread_create()` (POSIX);
- множество функций win32, например `EnumChildWindows()` ⁵.
- множество мест в ядре Linux, например, функции драйверов файловой системы вызываются через callback-и: <http://go.yurichev.com/17076>
- функции плагинов GCC также вызываются через callback-и: <http://go.yurichev.com/17077>
- Один из примеров указателей на функции это таблица в оконном менеджере «dwm» для Linux, описывающая шорт-каты.

Каждый шорт-кат имеет соответствующую функцию, которую нужно вызвать, если эта клавиша нажата: [GitHub](#). Как мы видим, с такой таблицей намного легче обходиться чем с большим выражением `switch()`.

Итак, функция `qsort()` это реализация алгоритма «быстрой сортировки». Функция может сортировать что угодно, любые типы данных, но при условии, что вы имеете функцию сравнения этих двух элементов данных и `qsort()` может вызывать её.

Эта функция сравнения может определяться так:

```
int (*compare)(const void *, const void *)
```

Воспользуемся немного модифицированным примером, который был найден [здесь](#):

```
1 /* ex3 Sorting ints with qsort */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int comp(const void * _a, const void * _b)
7 {
8     const int *a=(const int *)_a;
9     const int *b=(const int *)_b;
10
11     if (*a==*b)
12         return 0;
13     else
14         if (*a < *b)
```

¹[wikipedia](#)

²[wikipedia](#)

³<http://go.yurichev.com/17073>

⁴[wikipedia](#)

⁵MSDN

24.1. MSVC

```
15     return -1;
16     else
17         return 1;
18 }
19
20 int main(int argc, char* argv[])
21 {
22     int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
23     int i;
24
25     /* Sort the array */
26     qsort(numbers,10,sizeof(int),comp) ;
27     for (i=0;i<9;i++)
28         printf("Number = %d\n",numbers[ i ]) ;
29     return 0;
30 }
```

24.1. MSVC

Компилируем в MSVC 2010 (некоторые части убраны для краткости) с опцией /Ox :

Листинг 24.1: Оптимизирующий MSVC 2010: /GS- /MD

```
__a$ = 8 ; size = 4
__b$ = 12 ; size = 4
_comp PROC
    mov     eax, DWORD PTR __a$[esp-4]
    mov     ecx, DWORD PTR __b$[esp-4]
    mov     eax, DWORD PTR [eax]
    mov     ecx, DWORD PTR [ecx]
    cmp     eax, ecx
    jne     SHORT $LN4@comp
    xor     eax, eax
    ret     0
$LN4@comp:
    xor     edx, edx
    cmp     eax, ecx
    setge   dl
    lea     eax, DWORD PTR [edx+edx-1]
    ret     0
_comp ENDP

_numbers$ = -40 ; size = 40
_argc$ = 8 ; size = 4
_argv$ = 12 ; size = 4
_main PROC
    sub     esp, 40 ; 00000028H
    push    esi
    push    OFFSET _comp
    push    4
    lea     eax, DWORD PTR _numbers$[esp+52]
    push    10 ; 0000000aH
    push    eax
    mov     DWORD PTR _numbers$[esp+60], 1892 ; 00000764H
    mov     DWORD PTR _numbers$[esp+64], 45 ; 0000002dH
    mov     DWORD PTR _numbers$[esp+68], 200 ; 000000c8H
    mov     DWORD PTR _numbers$[esp+72], -98 ; ffffffff9eH
    mov     DWORD PTR _numbers$[esp+76], 4087 ; 00000ff7H
    mov     DWORD PTR _numbers$[esp+80], 5
    mov     DWORD PTR _numbers$[esp+84], -12345 ; fffffcfc7H
    mov     DWORD PTR _numbers$[esp+88], 1087 ; 0000043fH
    mov     DWORD PTR _numbers$[esp+92], 88 ; 00000058H
    mov     DWORD PTR _numbers$[esp+96], -100000 ; fffe7960H
    call    _qsort
    add     esp, 16 ; 00000010H
```

...

Ничего особо удивительного здесь мы не видим. В качестве четвертого аргумента, в `qsort()` просто передается адрес метки `_comp`, где собственно и располагается функция `comp()`, или, можно сказать, самая первая инструкция этой функции.

Как `qsort()` вызывает её?

Посмотрим в MSVCR80.DLL (эта DLL куда в MSVC вынесены функции из стандартных библиотек Си):

Листинг 24.2: MSVCR80.DLL

```
.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int (__cdecl *)(const void *, ↗
    ↳ const void *))
.text:7816CBF0      public _qsort
.text:7816CBF0      _qsort      proc near
.text:7816CBF0
.text:7816CBF0      lo          = dword ptr -104h
.text:7816CBF0      hi          = dword ptr -100h
.text:7816CBF0      var_FC      = dword ptr -0FCh
.text:7816CBF0      stkptr      = dword ptr -0F8h
.text:7816CBF0      lostk       = dword ptr -0F4h
.text:7816CBF0      histk       = dword ptr -7Ch
.text:7816CBF0      base        = dword ptr 4
.text:7816CBF0      num         = dword ptr 8
.text:7816CBF0      width       = dword ptr 0Ch
.text:7816CBF0      comp        = dword ptr 10h
.text:7816CBF0
.text:7816CBF0      sub         esp, 100h

....

.text:7816CCE0 loc_7816CCE0:                                ; CODE XREF: _qsort+B1
.text:7816CCE0      shr         eax, 1
.text:7816CCE2      imul        eax, ebp
.text:7816CCE5      add         eax, ebx
.text:7816CCE7      mov         edi, eax
.text:7816CCE9      push        edi
.text:7816CCEA      push        ebx
.text:7816CCEB      call        [esp+118h+comp]
.text:7816CCF2      add         esp, 8
.text:7816CCF5      test        eax, eax
.text:7816CCF7      jle         short loc_7816CD04
```

`comp` — это четвертый аргумент функции. Здесь просто передается управление по адресу, указанному в `comp`. Перед этим подготавливается два аргумента для функции `comp()`. Далее, проверяется результат её выполнения.

Вот почему использование указателей на функции — это опасно. Во-первых, если вызвать `qsort()` с неправильным указателем на функцию, то `qsort()`, дойдя до этого вызова, может передать управление неизвестно куда, процесс упадет, и эту ошибку можно будет найти не сразу.

Во-вторых, типизация callback-функции должна строго соблюдаться, вызов не той функции с не теми аргументами не того типа, может привести к плачевным результатам, хотя падение процесса это и не проблема, проблема — это найти ошибку, ведь компилятор на стадии компиляции может вас и не предупредить о потенциальных неприятностях.

24.1.1. MSVC + OllyDbg

Загрузим наш пример в OllyDbg и установим точку останова на функции `comp()`. Как значения сравниваются, мы можем увидеть во время самого первого вызова `comp()`:

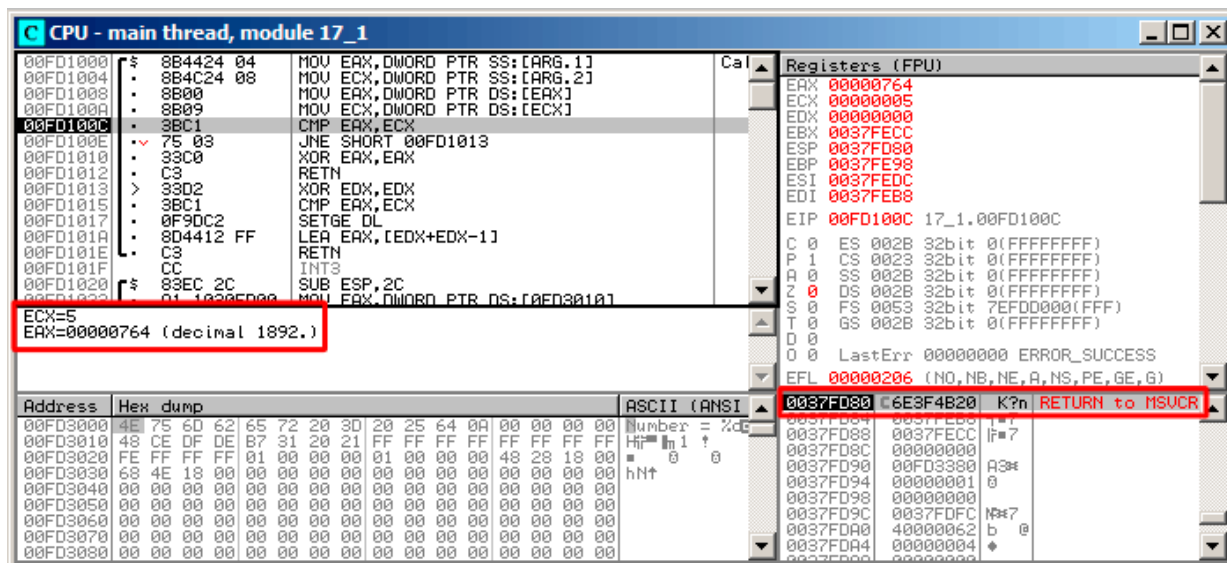


Рис. 24.1: OllyDbg: первый вызов `comp()`

Для удобства, OllyDbg показывает сравниваемые значения в окне под окном кода. Мы можем так же увидеть, что `SP` указывает на `RA` где находится место в функции `qsort()` (на самом деле, находится в `MSVCRT100.DLL`).

24.1. MSVC

Трассируя (F8) до инструкции `RETN` и нажав F8 еще один раз, мы возвращаемся в функцию `qsort()` :

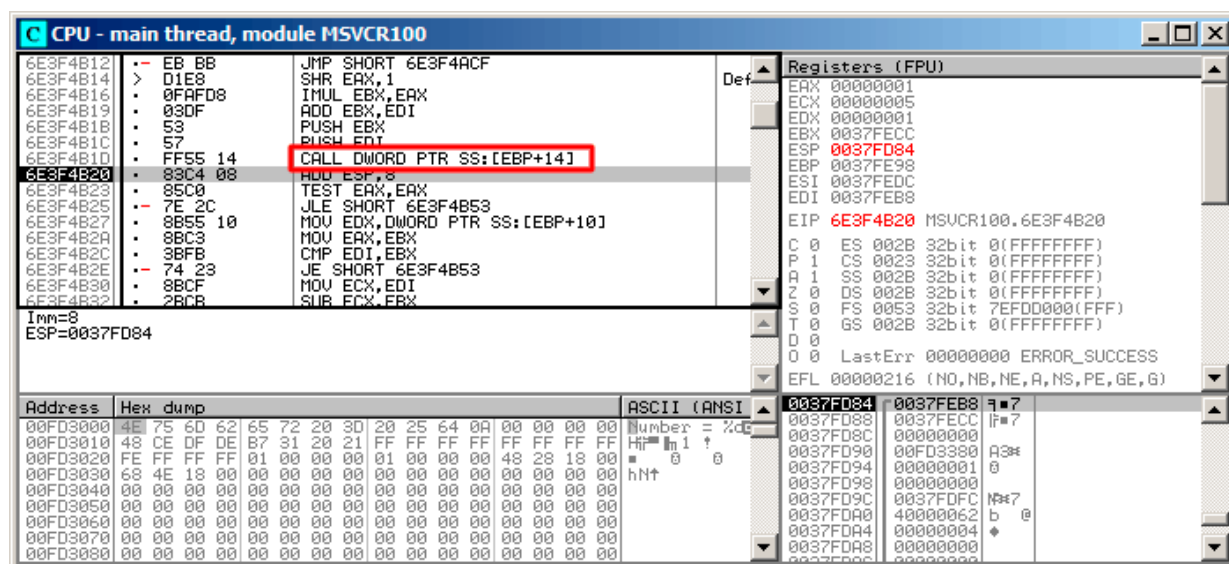


Рис. 24.2: OllyDbg: код в `qsort()` сразу после вызова `comp()`

Это был вызов функции сравнения.

24.1. MSVC

Вот также скриншот момента второго вызова функции `comp()` – теперь сравниваемые значения другие:

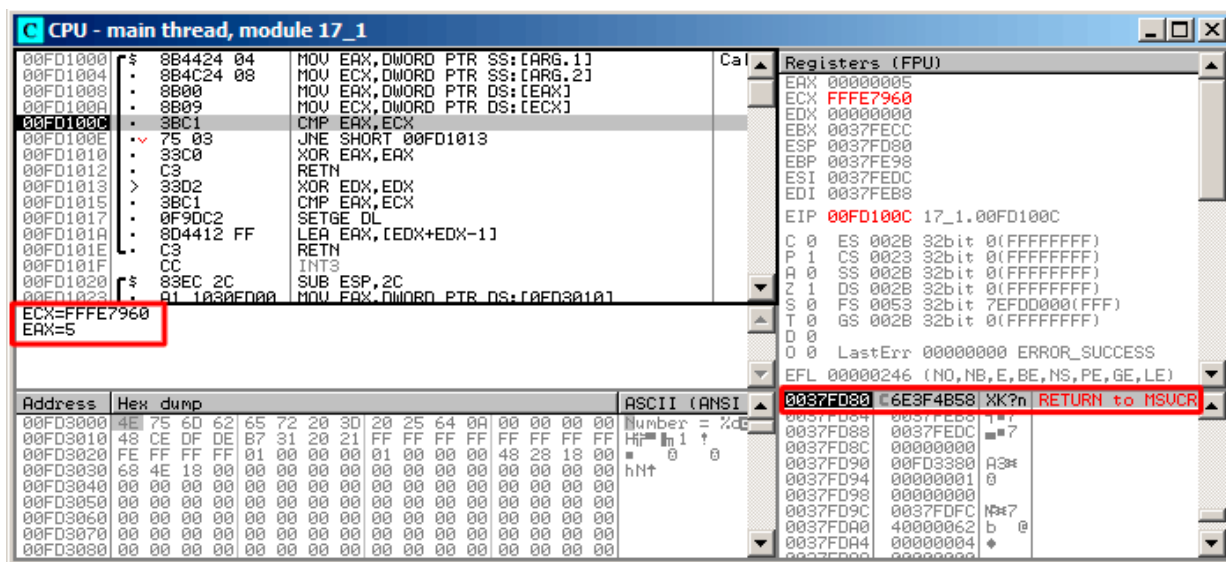


Рис. 24.3: OllyDbg: второй вызов `comp()`

24.1.2. MSVC + tracer

Посмотрим, какие пары сравниваются. Эти 10 чисел будут сортироваться: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

Найдем адрес первой инструкции `CMP` в `comp()` и это `0x0040100C` и мы ставим точку останова на ней:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

Получаем информацию о регистрах на точке останова:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xffffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

Отфильтруем `EAX` и `ECX` и получим:

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xffffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
```

24.1. MSVC

EAX=0xffffffff9e	ECX=0x00000005
EAX=0xffffffffc7	ECX=0xfffe7960
EAX=0x00000005	ECX=0xffffffffc7
EAX=0xffffffff9e	ECX=0x00000005
EAX=0xffffffffc7	ECX=0xfffe7960
EAX=0xffffffff9e	ECX=0xffffffffc7
EAX=0xffffffffc7	ECX=0xfffe7960
EAX=0x000000c8	ECX=0x00000ff7
EAX=0x0000002d	ECX=0x00000ff7
EAX=0x0000043f	ECX=0x00000ff7
EAX=0x00000058	ECX=0x00000ff7
EAX=0x00000764	ECX=0x00000ff7
EAX=0x000000c8	ECX=0x00000764
EAX=0x0000002d	ECX=0x00000764
EAX=0x0000043f	ECX=0x00000764
EAX=0x00000058	ECX=0x00000764
EAX=0x000000c8	ECX=0x00000058
EAX=0x0000002d	ECX=0x000000c8
EAX=0x0000043f	ECX=0x000000c8
EAX=0x000000c8	ECX=0x00000058
EAX=0x0000002d	ECX=0x000000c8
EAX=0x0000002d	ECX=0x00000058

Это 34 пары. Следовательно, алгоритму быстрой сортировки нужно 34 операции сравнения для сортировки этих 10-и чисел.

24.1.3. MSVC + tracer (code coverage)

Но можно также и воспользоваться возможностью tracer накапливать все возможные состояния регистров и показывать их в IDA.

Трассируем все инструкции в функции `comp()`:

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

Получем .idc-скрипт для загрузки в IDA и загружаем его:

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare      proc near                ; DATA XREF: _main+5↓o
.text:00401000
.text:00401000 arg_0          = dword ptr 4
.text:00401000 arg_4          = dword ptr 8
.text:00401000
.text:00401000 mov     eax, [esp+arg_0] ; [ESP+4]=0x45f7ec..0x45f810(step=4), L"?\x04?
.text:00401004 mov     ecx, [esp+arg_4] ; [ESP+8]=0x45f7ec..0x45f7f4(step=4), 0x45f7fc
.text:00401008 mov     eax, [eax]      ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff
.text:0040100a mov     ecx, [ecx]      ; [ECX]=5, 0x58, 0xc8, 0x764, 0xff7, 0xfffe7960
.text:0040100c cmp     eax, ecx        ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:0040100e jnz     short loc_401013 ; ZF=false
.text:00401010 xor     eax, eax
.text:00401012 retn
.text:00401013 ; -----
.text:00401013
.text:00401013 loc_401013:                ; CODE XREF: PtFuncCompare+E↑j
.text:00401013 xor     edx, edx
.text:00401015 cmp     eax, ecx        ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:00401017 setnl  dl,                ; SF=false,true OF=false
.text:0040101a lea     eax, [edx+edx-1]
.text:0040101e retn                ; EAX=1, 0xffffffff
.text:0040101e PtFuncCompare endp
.text:0040101f
```

Рис. 24.4: tracer и IDA. N.B.: некоторые значения обрезаны справа

Имя этой функции (PtFuncCompare) дала IDA — видимо, потому что видит что указатель на эту функцию передается в `qsort()`.

Мы видим, что указатели *a* и *b* указывают на разные места внутри массива, но шаг между указателями — 4, что логично, ведь в массиве хранятся 32-битные значения.

Видно, что инструкции по адресам `0x401010` и `0x401012` никогда не исполнялись (они и остались белыми): действительно, функция `comp()` никогда не возвращала 0, потому что в массиве нет одинаковых элементов.

24.2. GCC

Не слишком большая разница:

Листинг 24.3: GCC

```
lea     eax, [esp+40h+var_28]
mov     [esp+40h+var_40], eax
mov     [esp+40h+var_28], 764h
mov     [esp+40h+var_24], 2Dh
mov     [esp+40h+var_20], 0C8h
mov     [esp+40h+var_1C], 0FFFFFF9Eh
mov     [esp+40h+var_18], 0FF7h
mov     [esp+40h+var_14], 5
mov     [esp+40h+var_10], 0FFFCFC7h
mov     [esp+40h+var_C], 43Fh
mov     [esp+40h+var_8], 58h
mov     [esp+40h+var_4], 0FFFE7960h
mov     [esp+40h+var_34], offset comp
```

24.2. GCC

```
mov    [esp+40h+var_38], 4
mov    [esp+40h+var_3C], 0Ah
call   _qsort
```

Функция `comp()` :

```
comp    public comp
comp    proc near
arg_0    = dword ptr 8
arg_4    = dword ptr 0Ch

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_4]
        mov     ecx, [ebp+arg_0]
        mov     edx, [eax]
        xor     eax, eax
        cmp     [ecx], edx
        jnz     short loc_8048458
        pop     ebp
        retn

loc_8048458:
        setnl   al
        movzx   eax, al
        lea     eax, [eax+eax-1]
        pop     ebp
        retn
comp    endp
```

Реализация `qsort()` находится в `libc.so.6`, и представляет собой просто wrapper ⁶ для `qsort_r()`.

Она, в свою очередь, вызывает `quicksort()`, где есть вызовы определенной нами функции через переданный указатель:

Листинг 24.4: (файл `libc.so.6`, версия `glibc` – 2.10.1)

```
.text:0002DDF6      mov     edx, [ebp+arg_10]
.text:0002DDF9      mov     [esp+4], esi
.text:0002DDFD      mov     [esp], edi
.text:0002DE00      mov     [esp+8], edx
.text:0002DE04      call    [ebp+arg_C]
...
```

24.2.1. GCC + GDB (с исходными кодами)

Очевидно, у нас есть исходный код нашего примера на Си (24 (стр. 380)), так что мы можем установить точку останова (b) на номере строки (11-й – это номер строки где происходит первое сравнение). Нам также нужно скомпилировать наш пример с ключом `-g`, чтобы в исполняемом файле была полная отладочная информация. Мы можем так же вывести значения используя имена переменных (p): отладочная информация также содержит информацию о том, в каком регистре и/или элементе локального стека находится какая переменная.

Мы можем также увидеть стек (bt) и обнаружить что в Glibc используется какая-то вспомогательная функция с именем `msort_with_tmp()`.

Листинг 24.5: GDB-сессия

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
```

⁶понятие близкое к [thunk function](#)

24.2. GCC

```
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
11      if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17_1.c:11
11      if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c:307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)
```

24.2.2. GCC + GDB (без исходных кодов)

Но часто никаких исходных кодов нет вообще, так что мы можем дизассемблировать функцию `comp()` (`disas`), найти самую первую инструкцию `CMP` и установить точку останова (`b`) по этому адресу. На каждой точке останова мы будем видеть содержимое регистров (`info registers`). Информация из стека так же доступна (`bt`), но частичная: здесь нет номеров строк для функции `comp()`.

Листинг 24.6: GDB-сессия

```
dennis@ubuntuvvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
0x0804844d <+0>:  push    ebp
0x0804844e <+1>:  mov     ebp,esp
0x08048450 <+3>:  sub     esp,0x10
```

24.2. GCC

```

0x08048453 <+6>:    mov     eax,DWORD PTR [ebp+0x8]
0x08048456 <+9>:    mov     DWORD PTR [ebp-0x8],eax
0x08048459 <+12>:   mov     eax,DWORD PTR [ebp+0xc]
0x0804845c <+15>:   mov     DWORD PTR [ebp-0x4],eax
0x0804845f <+18>:   mov     eax,DWORD PTR [ebp-0x8]
0x08048462 <+21>:   mov     edx,DWORD PTR [eax]
0x08048464 <+23>:   mov     eax,DWORD PTR [ebp-0x4]
0x08048467 <+26>:   mov     eax,DWORD PTR [eax]
0x08048469 <+28>:   cmp     edx,eax
0x0804846b <+30>:   jne     0x8048474 <comp+39>
0x0804846d <+32>:   mov     eax,0x0
0x08048472 <+37>:   jmp     0x804848e <comp+65>
0x08048474 <+39>:   mov     eax,DWORD PTR [ebp-0x8]
0x08048477 <+42>:   mov     edx,DWORD PTR [eax]
0x08048479 <+44>:   mov     eax,DWORD PTR [ebp-0x4]
0x0804847c <+47>:   mov     eax,DWORD PTR [eax]
0x0804847e <+49>:   cmp     edx,eax
0x08048480 <+51>:   jge     0x8048489 <comp+60>
0x08048482 <+53>:   mov     eax,0xffffffff
0x08048487 <+58>:   jmp     0x804848e <comp+65>
0x08048489 <+60>:   mov     eax,0x1
0x0804848e <+65>:   leave
0x0804848f <+66>:   ret

```

End of assembler dump.

(gdb) b *0x08048469

Breakpoint 1 at 0x8048469

(gdb) run

Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

```

eax             0x2d      45
ecx             0xbffff0f8  -1073745672
edx             0x764     1892
ebx             0xb7fc0000  -1208221696
esp             0xbffffeeb8 0xbffffeeb8
ebp             0xbffffeec8 0xbffffeec8
esi             0xbffff0fc  -1073745668
edi             0xbffff010  -1073745904
eip             0x8048469   0x8048469 <comp+28>
eflags          0x286     [ PF SF IF ]
cs              0x73      115
ss              0x7b      123
ds              0x7b      123
es              0x7b      123
fs              0x0        0
gs              0x33      51

```

(gdb) c

Continuing.

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

```

eax             0xff7     4087
ecx             0xbffff104  -1073745660
edx             0xffffffff9e -98
ebx             0xb7fc0000  -1208221696
esp             0xbffffee58 0xbffffee58
ebp             0xbffffee68 0xbffffee68
esi             0xbffff108  -1073745656
edi             0xbffff010  -1073745904
eip             0x8048469   0x8048469 <comp+28>
eflags          0x282     [ SF IF ]
cs              0x73      115
ss              0x7b      123
ds              0x7b      123
es              0x7b      123
fs              0x0        0
gs              0x33      51

```

(gdb) c

Continuing.

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

eax	0xffffffff9e	-98
ecx	0xbffff100	-1073745664
edx	0xc8	200
ebx	0xb7fc0000	-1208221696
esp	0xbffffeeb8	0xbffffeeb8
ebp	0xbffffec8	0xbffffec8
esi	0xbffff104	-1073745660
edi	0xbffff010	-1073745904
eip	0x8048469	0x8048469 <comp+28>
eflags	0x286	[PF SF IF]
cs	0x73	115
ss	0x7b	123
ds	0x7b	123
es	0x7b	123
fs	0x0	0
gs	0x33	51

(gdb) bt

```
#0 0x08048469 in comp ()
#1 0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2 0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#3 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c:53
#4 0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#5 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c:53
#6 0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#7 __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8 0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c:307
#9 0x0804850d in main ()
```

Глава 25

64-битные значения в 32-битной среде

В среде, где GPR-ы 32-битные, 64-битные значения хранятся и передаются как пары 32-битных значений ¹.

25.1. Возврат 64-битного значения

```
#include <stdint.h>

uint64_t f ()
{
    return 0x1234567890ABCDEF;
};
```

25.1.1. x86

64-битные значения в 32-битной среде возвращаются из функций в паре регистров EDX : EAX .

Листинг 25.1: Оптимизирующий MSVC 2010

```
_f PROC
    mov     eax, -1867788817      ; 90abcdefH
    mov     edx, 305419896       ; 12345678H
    ret     0
_f ENDP
```

25.1.2. ARM

64-битное значение возвращается в паре регистров R0 - R1 — (R1 это старшая часть и R0 — младшая часть):

Листинг 25.2: Оптимизирующий Keil 6/2013 (Режим ARM)

```
||f|| PROC
    LDR     r0, |L0.12|
    LDR     r1, |L0.16|
    BX      lr
    ENDP

|L0.12|
    DCD     0x90abcdef

|L0.16|
    DCD     0x12345678
```

¹Кстати, в 16-битной среде, 32-битные значения передаются 16-битными парами точно так же: 54.4 (стр. 594)

25.1.3. MIPS

64-битное значение возвращается в паре регистров **V0** - **V1** (\$2-\$3) – (**V0** (\$2) это старшая часть и **V1** (\$3) – младшая часть):

Листинг 25.3: Оптимизирующий GCC 4.4.5 (assembly listing)

```
li    $3, -1867841536          # 0xfffffffff90ab0000
li    $2, 305397760            # 0x12340000
ori   $3, $3, 0xcdef
j     $31
ori   $2, $2, 0x5678
```

Листинг 25.4: Оптимизирующий GCC 4.4.5 (IDA)

```
lui    $v1, 0x90AB
lui    $v0, 0x1234
li     $v1, 0x90ABCDEF
jr     $ra
li     $v0, 0x12345678
```

25.2. Передача аргументов, сложение, вычитание

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
    return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
    printf ("%lld\n", f_add(12345678901234, 23456789012345));
#else
    printf ("%I64d\n", f_add(12345678901234, 23456789012345));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
    return a-b;
};
```

25.2.1. x86

Листинг 25.5: Оптимизирующий MSVC 2012 /Ob1

```
_a$ = 8      ; size = 8
_b$ = 16     ; size = 8
_f_add PROC
    mov     eax, DWORD PTR _a$[esp-4]
    add     eax, DWORD PTR _b$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    adc     edx, DWORD PTR _b$[esp]
    ret     0
_f_add ENDP

_f_add_test PROC
    push    5461          ; 00001555H
    push    1972608889    ; 75939f79H
    push    2874          ; 00000b3aH
    push    1942892530    ; 73ce2ff_subH
    call    _f_add
    push    edx
```

25.2. ПЕРЕДАЧА АРГУМЕНТОВ, СЛОЖЕНИЕ, ВЫЧИТАНИЕ

```
    push    eax
    push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
    call    _printf
    add     esp, 28
    ret     0
_f_add_test ENDP

_f_sub PROC
    mov     eax, DWORD PTR _a$[esp-4]
    sub     eax, DWORD PTR _b$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    sbb     edx, DWORD PTR _b$[esp]
    ret     0
_f_sub ENDP
```

В `f_add_test()` видно, как каждое 64-битное число передается двумя 32-битными значениями, сначала старшая часть, затем младшая.

Сложение и вычитание происходит также парами.

При сложении, в начале складываются младшие 32 бита. Если при сложении был перенос, выставляется флаг CF. Следующая инструкция `ADC` складывает старшие части чисел, но также прибавляет единицу если $CF = 1$.

Вычитание также происходит парами. Первый `SUB` может также включить флаг переноса CF, который затем будет проверяться в `SBB`: если флаг переноса включен, то от результата отнимется единица.

Легко увидеть, как результат работы `f_add()` затем передается в `printf()`.

Листинг 25.6: GCC 4.8.1 -O1 -fno-inline

```
_f_add:
    mov     eax, DWORD PTR [esp+12]
    mov     edx, DWORD PTR [esp+16]
    add     eax, DWORD PTR [esp+4]
    adc     edx, DWORD PTR [esp+8]
    ret

_f_add_test:
    sub     esp, 28
    mov     DWORD PTR [esp+8], 1972608889 ; 75939f79H
    mov     DWORD PTR [esp+12], 5461      ; 00001555H
    mov     DWORD PTR [esp], 1942892530   ; 73ce2ff_subH
    mov     DWORD PTR [esp+4], 2874       ; 00000b3aH
    call    _f_add
    mov     DWORD PTR [esp+4], eax
    mov     DWORD PTR [esp+8], edx
    mov     DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\\12\\0"
    call    _printf
    add     esp, 28
    ret

_f_sub:
    mov     eax, DWORD PTR [esp+4]
    mov     edx, DWORD PTR [esp+8]
    sub     eax, DWORD PTR [esp+12]
    sbb     edx, DWORD PTR [esp+16]
    ret
```

Код GCC почти такой же.

25.2.2. ARM

Листинг 25.7: Оптимизирующий Keil 6/2013 (Режим ARM)

```
f_add PROC
    ADDS    r0,r0,r2
    ADC     r1,r1,r3
    BX      lr
ENDP
```



```

f_sub PROC
    SUBS    r0,r0,r2
    SBC     r1,r1,r3
    BX      lr
    ENDP

f_add_test PROC
    PUSH    {r4,lr}
    LDR     r2,|L0.68| ; 0x75939f79
    LDR     r3,|L0.72| ; 0x00001555
    LDR     r0,|L0.76| ; 0x73ce2ff2
    LDR     r1,|L0.80| ; 0x00000b3a
    BL      f_add
    POP     {r4,lr}
    MOV     r2,r0
    MOV     r3,r1
    ADR     r0,|L0.84| ; "%I64d\n"
    B       __2printf
    ENDP

|L0.68|
DCD        0x75939f79
|L0.72|
DCD        0x00001555
|L0.76|
DCD        0x73ce2ff2
|L0.80|
DCD        0x00000b3a
|L0.84|
DCB        "%I64d\n",0

```

Первое 64-битное значение передается в паре регистров **R0** и **R1**, второе — в паре **R2** и **R3**. В ARM также есть инструкция **ADC** (учитывающая флаг переноса) и **SBC** («subtract with carry» — вычесть с переносом). Важная вещь: когда младшие части слагаются/вычитаются, используются инструкции **ADDS** и **SUBS** с суффиксом **-S**. Суффикс **-S** означает «set flags» (установить флаги), а флаги (особенно флаг переноса) это то что однозначно нужно последующим инструкциями **ADC** / **SBC**. А иначе инструкции без суффикса **-S** здесь вполне бы подошли (**ADD** и **SUB**).

25.2.3. MIPS

Листинг 25.8: Оптимизирующий GCC 4.4.5 (IDA)

```

f_add:
; $a0 – старшая часть a
; $a1 – младшая часть a
; $a2 – старшая часть b
; $a3 – младшая часть b
    addu    $v1, $a3, $a1 ; суммировать младшие части
    addu    $a0, $a2, $a0 ; суммировать старшие части
; будет ли перенос сгенерирован во время суммирования младших частей?
; установить $v0 в 1, если да
    sltu    $v0, $v1, $a3
    jr      $ra
; прибавить 1 к старшей части результата, если перенос должен был быть сгенерирован
    addu    $v0, $a0 ; branch delay slot
; $v0 – старшая часть результата
; $v1 – младшая часть результата

f_sub:
; $a0 – старшая часть a
; $a1 – младшая часть a
; $a2 – старшая часть b
; $a3 – младшая часть b
    subu    $v1, $a1, $a3 ; вычитать младшие части
    subu    $v0, $a0, $a2 ; вычитать старшие части
; будет ли перенос сгенерирован во время вычитания младших частей?
; установить $a0 в 1, если да
    sltu    $a1, $v1

```

25.3. УМНОЖЕНИЕ, ДЕЛЕНИЕ

```
        jr      $ra
; вычесть 1 из старшей части результата, если перенос должен был быть сгенерирован
        subu    $v0, $a1 ; branch delay slot
; $v0 – старшая часть результата
; $v1 – младшая часть результата

f_add_test:

var_10      = -0x10
var_4       = -4

        lui     $gp, (__gnu_local_gp >> 16)
        addiu   $sp, -0x20
        la      $gp, (__gnu_local_gp & 0xFFFF)
        sw      $ra, 0x20+var_4($sp)
        sw      $gp, 0x20+var_10($sp)
        lui     $a1, 0x73CE
        lui     $a3, 0x7593
        li      $a0, 0xB3A
        li      $a3, 0x75939F79
        li      $a2, 0x1555
        jal     f_add
        li      $a1, 0x73CE2FF2
        lw      $gp, 0x20+var_10($sp)
        lui     $a0, ($LC0 >> 16) # "%lld\n"
        lw      $t9, (printf & 0xFFFF)($gp)
        lw      $ra, 0x20+var_4($sp)
        la      $a0, ($LC0 & 0xFFFF) # "%lld\n"
        move    $a3, $v1
        move    $a2, $v0
        jr      $t9
        addiu   $sp, 0x20

$LC0:    .ascii "%lld\n"<0>
```

В MIPS нет регистра флагов, так что эта информация не присутствует после исполнения арифметических операций.

Так что здесь нет инструкций как **ADC** или **SBB** в x86. Чтобы получить информацию о том, был бы выставлен флаг переноса, происходит сравнение (используя инструкцию **SLTU**), которая выставляет целевой регистр в 1 или 0.

Эта 1 или 0 затем прибавляется к итоговому результату, или вычитается.

25.3. Умножение, деление

```
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
{
    return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b)
{
    return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b)
{
    return a % b;
};
```

25.3.1. x86

25.3. УМНОЖЕНИЕ, ДЕЛЕНИЕ

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __allmul ; long long multiplication (умножение значений типа long long)
    pop     ebp
    ret     0
_f_mul ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_div PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __aulldiv ; unsigned long long division (деление беззнаковых значений типа long long)
    pop     ebp
    ret     0
_f_div ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_rem PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __aullrem ; unsigned long long remainder (вычисление беззнакового остатка)
    pop     ebp
    ret     0
_f_rem ENDP
```

Умножение и деление — это более сложная операция, так что обычно, компилятор встраивает вызовы библиотечных функций, делающих это.

Значение этих библиотечных функций, здесь: [E](#) (стр. 933).

Листинг 25.10: Оптимизирующий GCC 4.8.1 -fno-inline

```
_f_mul:
    push    ebx
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+16]
    mov     ebx, DWORD PTR [esp+12]
    mov     ecx, DWORD PTR [esp+20]
    imul    ebx, eax
    imul    ecx, edx
    mul     edx
    add     ecx, ebx
```

25.3. УМНОЖЕНИЕ, ДЕЛЕНИЕ

```
    add    edx, ecx
    pop    ebx
    ret

_f_div:
    sub    esp, 28
    mov    eax, DWORD PTR [esp+40]
    mov    edx, DWORD PTR [esp+44]
    mov    DWORD PTR [esp+8], eax
    mov    eax, DWORD PTR [esp+32]
    mov    DWORD PTR [esp+12], edx
    mov    edx, DWORD PTR [esp+36]
    mov    DWORD PTR [esp], eax
    mov    DWORD PTR [esp+4], edx
    call   ___udivdi3 ; unsigned division (беззнаковое деление)
    add    esp, 28
    ret

_f_rem:
    sub    esp, 28
    mov    eax, DWORD PTR [esp+40]
    mov    edx, DWORD PTR [esp+44]
    mov    DWORD PTR [esp+8], eax
    mov    eax, DWORD PTR [esp+32]
    mov    DWORD PTR [esp+12], edx
    mov    edx, DWORD PTR [esp+36]
    mov    DWORD PTR [esp], eax
    mov    DWORD PTR [esp+4], edx
    call   ___umoddi3 ; unsigned modulo (беззнаковый остаток)
    add    esp, 28
    ret
```

GCC делает почти то же самое, тем не менее, встраивает код умножения прямо в функцию, посчитав что так будет эффективнее. У GCC другие имена библиотечных функций: [D](#) (стр. [932](#)).

25.3.2. ARM

Keil для режима Thumb вставляет вызовы библиотечных функций:

Листинг 25.11: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
||f_mul|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_lmul
    POP     {r4,pc}
    ENDP

||f_div|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_uldivmod
    POP     {r4,pc}
    ENDP

||f_rem|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_uldivmod
    MOVS    r0,r2
    MOVS    r1,r3
    POP     {r4,pc}
    ENDP
```

Keil для режима ARM, тем не менее, может сгенерировать код для умножения 64-битных чисел:

Листинг 25.12: Оптимизирующий Keil 6/2013 (Режим ARM)

```
||f_mul|| PROC
    PUSH    {r4,lr}
    UMULL    r12,r4,r0,r2
    MLA      r1,r2,r1,r4
```

25.3. УМНОЖЕНИЕ, ДЕЛЕНИЕ

```
MLA    r1,r0,r3,r1
MOV    r0,r12
POP    {r4,pc}
ENDP

||f_div|| PROC
PUSH    {r4,lr}
BL      __aeabi_uldivmod
POP     {r4,pc}
ENDP

||f_rem|| PROC
PUSH    {r4,lr}
BL      __aeabi_uldivmod
MOV     r0,r2
MOV     r1,r3
POP     {r4,pc}
ENDP
```

25.3.3. MIPS

Оптимизирующий GCC для MIPS может генерировать код для 64-битного умножения, но для 64-битного деления приходится вызывать библиотечную функцию:

Листинг 25.13: Оптимизирующий GCC 4.4.5 (IDA)

```
f_mul:
    mult    $a2, $a1
    mflo    $v0
    or      $at, $zero ; NOP
    or      $at, $zero ; NOP
    mult    $a0, $a3
    mflo    $a0
    addu    $v0, $a0
    or      $at, $zero ; NOP
    multu   $a3, $a1
    mfhi    $a2
    mflo    $v1
    jr      $ra
    addu    $v0, $a2

f_div:
var_10      = -0x10
var_4       = -4

    lui     $gp, (__gnu_local_gp >> 16)
    addiu   $sp, -0x20
    la      $gp, (__gnu_local_gp & 0xFFFF)
    sw      $ra, 0x20+var_4($sp)
    sw      $gp, 0x20+var_10($sp)
    lw      $t9, (__udivdi3 & 0xFFFF)($gp)
    or      $at, $zero
    jalr    $t9
    or      $at, $zero
    lw      $ra, 0x20+var_4($sp)
    or      $at, $zero
    jr      $ra
    addiu   $sp, 0x20

f_rem:
var_10      = -0x10
var_4       = -4

    lui     $gp, (__gnu_local_gp >> 16)
    addiu   $sp, -0x20
    la      $gp, (__gnu_local_gp & 0xFFFF)
```

25.4. СДВИГ ВПРАВО

```
sw    $ra, 0x20+var_4($sp)
sw    $gp, 0x20+var_10($sp)
lw    $t9, (__umoddi3 & 0xFFFF)($gp)
or    $at, $zero
jalr  $t9
or    $at, $zero
lw    $ra, 0x20+var_4($sp)
or    $at, $zero
jr    $ra
addiu $sp, 0x20
```

Тут также много **НОП**-ов, это возможно заполнение delay slot-ов после инструкции умножения (она ведь работает медленнее прочих инструкций).

25.4. Сдвиг вправо

```
#include <stdint.h>

uint64_t f (uint64_t a)
{
    return a>>7;
};
```

25.4.1. x86

Листинг 25.14: Оптимизирующий MSVC 2012 /Ob1

```
_a$ = 8      ; size = 8
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    shrd    eax, edx, 7
    shr     edx, 7
    ret     0
_f ENDP
```

Листинг 25.15: Оптимизирующий GCC 4.8.1 -fno-inline

```
_f:
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+4]
    shrd    eax, edx, 7
    shr     edx, 7
    ret
```

Сдвиг происходит также в две операции: в начале сдвигается младшая часть, затем старшая. Но младшая часть сдвигается при помощи инструкции **SHRD**, она сдвигает значение в **EDX** на 7 бит, но подтягивает новые биты из **EAX**, т.е. из старшей части. Старшая часть сдвигается более известной инструкцией **SHR**: действительно, ведь освободившиеся биты в старшей части нужно просто заполнить нулями.

25.4.2. ARM

В ARM нет такой инструкции как **SHRD** в x86, так что компилятору Keil приходится всё это делать, используя простые сдвиги и операции «ИЛИ»:

Листинг 25.16: Оптимизирующий Keil 6/2013 (Режим ARM)

```
||f|| PROC
    LSR     r0,r0,#7
    ORR     r0,r0,r1,LSL #25
    LSR     r1,r1,#7
    BX      lr
    ENDP
```

```

||f|| PROC
    LSLS    r2,r1,#25
    LSRS    r0,r0,#7
    ORRS    r0,r0,r2
    LSRS    r1,r1,#7
    BX      lr
    ENDP

```

25.4.3. MIPS

GCC для MIPS реализует тот же алгоритм, что сделал Keil для режима Thumb:

Листинг 25.18: Оптимизирующий GCC 4.4.5 (IDA)

```

f:
    sll     $v0, $a0, 25
    srl     $v1, $a1, 7
    or      $v1, $v0, $v1
    jr      $ra
    srl     $v0, $a0, 7

```

25.5. Конвертирование 32-битного значения в 64-битное

```

#include <stdint.h>

int64_t f (int32_t a)
{
    return a;
};

```

25.5.1. x86

Листинг 25.19: Оптимизирующий MSVC 2012

```

_a$ = 8
_f    PROC
    mov     eax, DWORD PTR _a$[esp-4]
    cdq
    ret     0
_f    ENDP

```

Здесь появляется необходимость расширить 32-битное знаковое значение в 64-битное знаковое.

Конвертировать беззнаковые значения очень просто: нужно просто выставить в 0 все биты в старшей части. Но для знаковых типов это не подходит: знак числа должен быть скопирован в старшую часть числа-результата. Здесь это делает инструкция **CDQ**, она берет входное значение в **EAX**, расширяет его до 64-битного, и оставляет его в паре регистров **EDX:EAX**. Иными словами, инструкция **CDQ** узнает знак числа в **EAX** (просто берет самый старший бит в **EAX**) и в зависимости от этого, выставляет все 32 бита в **EDX** в 0 или в 1. Её работа в каком-то смысле напоминает работу инструкции **MOVSX**.

25.5.2. ARM

Листинг 25.20: Оптимизирующий Keil 6/2013 (Режим ARM)

```

||f|| PROC
    ASR     r1,r0,#31
    BX      lr
    ENDP

```

25.5. КОНВЕРТИРОВАНИЕ 32-БИТНОГО ЗНАЧЕНИЯ В 64-БИТНОЕ

Keil для ARM работает иначе: он просто сдвигает (арифметически) входное значение на 31 бит вправо. Как мы знаем, бит знака это **MSB**, и арифметический сдвиг копирует бит знака в «появляющихся» битах.

Так что после инструкции `ASR r1, r0, #31`, **R1** будет содержать 0xFFFFFFFF если входное значение было отрицательным, или 0 в противном случае. **R1** содержит старшую часть возвращаемого 64-битного значения. Другими словами, этот код просто копирует **MSB** (бит знака) из входного значения в **R0** во все биты старшей 32-битной части итогового 64-битного значения.

25.5.3. MIPS

GCC для MIPS делает то же, что сделал Keil для режима ARM:

Листинг 25.21: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
    sra    $v0, $a0, 31
    jr     $ra
    move   $v1, $a0
```


Глава 26

SIMD

SIMD это акроним: *Single Instruction, Multiple Data*.

Как можно судить по названию, это обработка множества данных исполняя только одну инструкцию.

Как и **FPU**, эта подсистема процессора выглядит так же отдельным процессором внутри x86.

SIMD в x86 начался с MMX. Появилось 8 64-битных регистров MM0-MM7.

Каждый MMX-регистр может содержать 2 32-битных значения, 4 16-битных или же 8 байт. Например, складывая значения двух MMX-регистров, можно складывать одновременно 8 8-битных значений.

Простой пример, это некий графический редактор, который хранит открытое изображение как двумерный массив. Когда пользователь меняет яркость изображения, редактору нужно, например, прибавить некий коэффициент ко всем пикселям, или отнять. Для простоты можно представить, что изображение у нас бело-серо-черное и каждый пиксель занимает один байт, то с помощью MMX можно менять яркость сразу у восьми пикселей. Кстати, вот причина почему в SIMD присутствуют инструкции с *насыщением* (*saturation*). Когда пользователь в графическом редакторе изменяет яркость, переполнение и антипереполнение (*underflow*) не нужны, так что в SIMD имеются, например, инструкции сложения, которые ничего не будут прибавлять если максимальное значение уже достигнуто, и т.д.

Когда MMX только появилось, эти регистры на самом деле располагались в FPU-регистрах. Можно было использовать либо FPU либо MMX в одно и то же время. Можно подумать, что Intel решило немного сэкономить на транзисторах, но на самом деле причина такого симбиоза проще — более старая **ОС** не знающая о дополнительных регистрах процессора не будет сохранять их во время переключения задач, а вот регистры FPU сохранять будет. Таким образом, процессор с MMX + старая **ОС** + задача, использующая возможности MMX = все это может работать вместе.

SSE — это расширение регистров до 128 бит, теперь уже отдельно от FPU.

AVX — расширение регистров до 256 бит.

Немного о практическом применении.

Конечно же, это копирование блоков в памяти (`memcpu`), сравнение (`memcmp`), и подобное.

Еще пример: имеется алгоритм шифрования DES, который берет 64-битный блок, 56-битный ключ, шифрует блок с ключом и образуется 64-битный результат. Алгоритм DES можно легко представить в виде очень большой электронной цифровой схемы, с проводами, элементами И, ИЛИ, НЕ.

Идея *bitslice DES*¹ — это обработка сразу группы блоков и ключей одновременно. Скажем, на x86 переменная типа *unsigned int* вмещает в себе 32 бита, так что там можно хранить промежуточные результаты сразу для 32-х блоков-ключей, используя 64+56 переменных типа *unsigned int*.

Существует утилита для перебора паролей/хешей Oracle RDBMS (которые основаны на алгоритме DES), реализующая алгоритм *bitslice DES* для SSE2 и AVX — и теперь возможно шифровать одновременно 128 или 256 блоков-ключей:

<http://go.yurichev.com/17313>

26.1. Векторизация

Векторизация² это когда у вас есть цикл, который берет на вход несколько массивов и выдает, например, один массив данных. Тело цикла берет некоторые элементы из входных массивов, что-то делает с ними и помещает в выходной. Векторизация — это обрабатывать несколько элементов одновременно.

¹<http://go.yurichev.com/17329>

²[Wikipedia: vectorization](http://en.wikipedia.org/wiki/vectorization)

26.1. ВЕКТОРИЗАЦИЯ

Векторизация — это не самая новая технология: автор сих строк видел её по крайней мере на линейке суперкомпьютеров Cray Y-MP от 1988, когда работал на его версии-«лайт» Cray Y-MP EL ³.

Например:

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

Этот фрагмент кода берет элементы из A и B, перемножает и сохраняет результат в C.

Если представить, что каждый элемент массива — это 32-битный *int*, то их можно загружать сразу по 4 из A в 128-битный XMM-регистр, из B в другой XMM-регистр и выполнив инструкцию *PMULLD* (*Перемножить запакованные знаковые DWORD и сохранить младшую часть результата*) и *PMULHW* (*Перемножить запакованные знаковые DWORD и сохранить старшую часть результата*), можно получить 4 64-битных **произведения** сразу.

Таким образом, тело цикла исполняется 1024/4 раза вместо 1024, что в 4 раза меньше, и, конечно, быстрее.

26.1.1. Пример сложения

Некоторые компиляторы умеют делать автоматическую векторизацию в простых случаях, например, Intel C++⁴.

Вот очень простая функция:

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
    for (int i=0; i<sz; i++)
        ar3[i]=ar1[i]+ar2[i];

    return 0;
};
```

Intel C++

Компилируем её при помощи Intel C++ 11.1.051 win32:

```
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

Имеем такое (в IDA):

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz      = dword ptr  4
ar1     = dword ptr  8
ar2     = dword ptr 0Ch
ar3     = dword ptr 10h

    push     edi
    push     esi
    push     ebx
    push     esi
    mov     edx, [esp+10h+sz]
    test    edx, edx
    jle     loc_15B
    mov     eax, [esp+10h+ar3]
    cmp     edx, 6
    jle     loc_143
    cmp     eax, [esp+10h+ar2]
    jbe     short loc_36
    mov     esi, [esp+10h+ar2]
    sub     esi, eax
    lea     ecx, ds:0[edx*4]
```

³Удаленно. Он находится в музее суперкомпьютеров: <http://go.yurichev.com/17081>

⁴Еще о том, как Intel C++ умеет автоматически векторизовать циклы: [Excerpt: Effective Automatic Vectorization](#)

26.1. ВЕКТОРИЗАЦИЯ

```
neg     esi
cmp     ecx, esi
jbe     short loc_55
```

```
loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
cmp     eax, [esp+10h+ar2]
jnb     loc_143
mov     esi, [esp+10h+ar2]
sub     esi, eax
lea     ecx, ds:0[edx*4]
cmp     esi, ecx
jb      loc_143
```

```
loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
cmp     eax, [esp+10h+ar1]
jbe     short loc_67
mov     esi, [esp+10h+ar1]
sub     esi, eax
neg     esi
cmp     ecx, esi
jbe     short loc_7F
```

```
loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
cmp     eax, [esp+10h+ar1]
jnb     loc_143
mov     esi, [esp+10h+ar1]
sub     esi, eax
cmp     esi, ecx
jb      loc_143
```

```
loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
mov     edi, eax          ; edi = ar1
and     edi, 0Fh          ; ar выровнен по 16-байтной границе?
jz      short loc_9A      ; да
test    edi, 3
jnz     loc_162
neg     edi
add     edi, 10h
shr     edi, 2
```

```
loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
lea     ecx, [edi+4]
cmp     edx, ecx
jl      loc_162
mov     ecx, edx
sub     ecx, edi
and     ecx, 3
neg     ecx
add     ecx, edx
test    edi, edi
jbe     short loc_D6
mov     ebx, [esp+10h+ar2]
mov     [esp+10h+var_10], ecx
mov     ecx, [esp+10h+ar1]
xor     esi, esi
```

```
loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
mov     edx, [ecx+esi*4]
add     edx, [ebx+esi*4]
mov     [eax+esi*4], edx
inc     esi
cmp     esi, edi
jb      short loc_C1
mov     ecx, [esp+10h+var_10]
mov     edx, [esp+10h+sz]
```

```
loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
mov     esi, [esp+10h+ar2]
lea     esi, [esi+edi*4] ; ar2+i*4 выровнен по 16-байтной границе?
test    esi, 0Fh
```

26.1. ВЕКТОРИЗАЦИЯ

```
jz      short loc_109    ; да!  
mov     ebx, [esp+10h+ar1]  
mov     esi, [esp+10h+ar2]
```

```
loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105  
        movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4  
        movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 не выровнен по 16-байтной границе, так что  
загружаем это в XMM0
```

```
        paddb xmm1, xmm0  
        movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4  
        add     edi, 4  
        cmp     edi, ecx  
        jnb     short loc_ED  
        jmp     short loc_127
```

```
loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3  
        mov     ebx, [esp+10h+ar1]  
        mov     esi, [esp+10h+ar2]
```

```
loc_111: ; CODE XREF: f(int,int *,int *,int *)+125  
        movdqu xmm0, xmmword ptr [ebx+edi*4]  
        paddb xmm0, xmmword ptr [esi+edi*4]  
        movdqa xmmword ptr [eax+edi*4], xmm0  
        add     edi, 4  
        cmp     edi, ecx  
        jnb     short loc_111
```

```
loc_127: ; CODE XREF: f(int,int *,int *,int *)+107  
        ; f(int,int *,int *,int *)+164  
        cmp     ecx, edx  
        jnb     short loc_15B  
        mov     esi, [esp+10h+ar1]  
        mov     edi, [esp+10h+ar2]
```

```
loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F  
        mov     ebx, [esi+ecx*4]  
        add     ebx, [edi+ecx*4]  
        mov     [eax+ecx*4], ebx  
        inc     ecx  
        cmp     ecx, edx  
        jnb     short loc_133  
        jmp     short loc_15B
```

```
loc_143: ; CODE XREF: f(int,int *,int *,int *)+17  
        ; f(int,int *,int *,int *)+3A ...  
        mov     esi, [esp+10h+ar1]  
        mov     edi, [esp+10h+ar2]  
        xor     ecx, ecx
```

```
loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159  
        mov     ebx, [esi+ecx*4]  
        add     ebx, [edi+ecx*4]  
        mov     [eax+ecx*4], ebx  
        inc     ecx  
        cmp     ecx, edx  
        jnb     short loc_14D
```

```
loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A  
        ; f(int,int *,int *,int *)+129 ...  
        xor     eax, eax  
        pop     ecx  
        pop     ebx  
        pop     esi  
        pop     edi  
        retn
```

```
loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C  
        ; f(int,int *,int *,int *)+9F  
        xor     ecx, ecx  
        jmp     short loc_127
```

```
?f@@YAHHPAH00@Z endp
```

Инструкции, имеющие отношение к SSE2 это:

- **MOVDQU** (*Move Unaligned Double Quadword*) — она просто загружает 16 байт из памяти в XMM-регистр .
- **PADDQ** (*Add Packed Integers*) — складывает сразу 4 пары 32-битных чисел и оставляет в первом операнде результат. Кстати, если произойдет переполнение, то исключения не произойдет и никакие флаги не установятся, запишутся просто младшие 32 бита результата. Если один из операндов **PADDQ** — адрес значения в памяти, то требуется чтобы адрес был выровнен по 16-байтной границе. Если он не выровнен, произойдет исключение ⁵.
- **MOVDQA** (*Move Aligned Double Quadword*) — тоже что и **MOVDQU**, только подразумевает что адрес в памяти выровнен по 16-байтной границе. Если он не выровнен, произойдет исключение. **MOVDQA** работает быстрее чем **MOVDQU**, но требует вышеозначенного.

Итак, эти SSE2-инструкции исполняются только в том случае если еще осталось просуммировать 4 пары переменных типа *int* плюс если указатель **ar3** выровнен по 16-байтной границе.

Более того, если еще и **ar2** выровнен по 16-байтной границе, то будет выполняться этот фрагмент кода:

```
movdqu xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddq xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

А иначе, значение из **ar2** загрузится в **XMM0** используя инструкцию **MOVDQU**, которая не требует выровненного указателя, зато может работать чуть медленнее:

```
movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 не выровнен по 16-байтной границе, так что загружаем это
в XMM0
paddq xmm1, xmm0
movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

А во всех остальных случаях, будет исполняться код, который был бы, как если бы не была включена поддержка SSE2.

GCC

Но и GCC умеет кое-что векторизировать⁶, если компилировать с опциями **-O3** и включить поддержку SSE2: **-msse2**.

Вот что вышло (GCC 4.4.1):

```
; f(int, int *, int *, int *)
    public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h
arg_C       = dword ptr  14h

    push    ebp
    mov     ebp, esp
    push    edi
    push    esi
    push    ebx
    sub     esp, 0Ch
    mov     ecx, [ebp+arg_0]
    mov     esi, [ebp+arg_4]
    mov     edi, [ebp+arg_8]
    mov     ebx, [ebp+arg_C]
    test    ecx, ecx
    jle     short loc_80484D8
    cmp     ecx, 6
```

⁵О выравнивании данных см. также: [Wikipedia: Выравнивание данных](http://en.cppreference.com/w/cpp/string/basic/basic_string_view)

⁶Подробнее о векторизации в GCC: <http://go.yurichev.com/17083>

26.1. ВЕКТОРИЗАЦИЯ

```
        lea     eax, [ebx+10h]
        ja      short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
            ; f(int,int *,int *,int *)+61 ...
            xor     eax, eax
            nop
            lea     esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
            mov     edx, [edi+eax*4]
            add     edx, [esi+eax*4]
            mov     [ebx+eax*4], edx
            add     eax, 1
            cmp     eax, ecx
            jnz     short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
            ; f(int,int *,int *,int *)+A5
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

            align 8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
            test    bl, 0Fh
            jnz     short loc_80484C1
            lea     edx, [esi+10h]
            cmp     ebx, edx
            jbe     loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
            lea     edx, [edi+10h]
            cmp     ebx, edx
            ja      short loc_8048503
            cmp     edi, eax
            jbe     short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
            mov     eax, ecx
            shr     eax, 2
            mov     [ebp+var_14], eax
            shl     eax, 2
            test    eax, eax
            mov     [ebp+var_10], eax
            jz      short loc_8048547
            mov     [ebp+var_18], ecx
            mov     ecx, [ebp+var_14]
            xor     eax, eax
            xor     edx, edx
            nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
            movdqu  xmm1, xmmword ptr [edi+eax]
            movdqu  xmm0, xmmword ptr [esi+eax]
            add     edx, 1
            paddb   xmm0, xmm1
            movdqa  xmmword ptr [ebx+eax], xmm0
            add     eax, 10h
            cmp     edx, ecx
            jb      short loc_8048520
            mov     ecx, [ebp+var_18]
            mov     eax, [ebp+var_10]
            cmp     ecx, eax
            jz      short loc_80484D8
```

26.1. ВЕКТОРИЗАЦИЯ

```
loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
    lea     edx, ds:0[eax*4]
    add     esi, edx
    add     edi, edx
    add     ebx, edx
    lea     esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
    mov     edx, [edi]
    add     eax, 1
    add     edi, 4
    add     edx, [esi]
    add     esi, 4
    mov     [ebx], edx
    add     ebx, 4
    cmp     ecx, eax
    jg      short loc_8048558
    add     esp, 0Ch
    xor     eax, eax
    pop     ebx
    pop     esi
    pop     edi
    pop     ebp
    retn

loc_8048578: ; CODE XREF: f(int,int *,int *,int *)+52
    cmp     eax, esi
    jnb     loc_80484C1
    jmp     loc_80484F8
_Z1fiPiS_S_ endp
```

Почти то же самое, хотя и не так дотошно, как Intel C++.

26.1.2. Пример копирования блоков

Вернемся к простому примеру `memcpy()` (15.2 (стр. 191)):

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
};
```

И вот что делает оптимизирующий GCC 4.9.1:

Листинг 26.1: Оптимизирующий GCC 4.9.1 x64

```
my_memcpy:
; RDI = адрес назначения
; RSI = исходный адрес
; RDX = размер блока
    test    rdx, rdx
    je      .L41
    lea     rax, [rdi+16]
    cmp     rsi, rax
    lea     rax, [rsi+16]
    setae   cl
    cmp     rdi, rax
    setae   al
    or      cl, al
    je      .L13
    cmp     rdx, 22
    jbe     .L13
    mov     rcx, rsi
    push    rbp
```

26.1. ВЕКТОРИЗАЦИЯ

```
push    rbx
neg     rcx
and     ecx, 15
cmp     rcx, rdx
cmova   rcx, rdx
xor     eax, eax
test    rcx, rcx
je      .L4
movzx   eax, BYTE PTR [rsi]
cmp     rcx, 1
mov     BYTE PTR [rdi], al
je      .L15
movzx   eax, BYTE PTR [rsi+1]
cmp     rcx, 2
mov     BYTE PTR [rdi+1], al
je      .L16
movzx   eax, BYTE PTR [rsi+2]
cmp     rcx, 3
mov     BYTE PTR [rdi+2], al
je      .L17
movzx   eax, BYTE PTR [rsi+3]
cmp     rcx, 4
mov     BYTE PTR [rdi+3], al
je      .L18
movzx   eax, BYTE PTR [rsi+4]
cmp     rcx, 5
mov     BYTE PTR [rdi+4], al
je      .L19
movzx   eax, BYTE PTR [rsi+5]
cmp     rcx, 6
mov     BYTE PTR [rdi+5], al
je      .L20
movzx   eax, BYTE PTR [rsi+6]
cmp     rcx, 7
mov     BYTE PTR [rdi+6], al
je      .L21
movzx   eax, BYTE PTR [rsi+7]
cmp     rcx, 8
mov     BYTE PTR [rdi+7], al
je      .L22
movzx   eax, BYTE PTR [rsi+8]
cmp     rcx, 9
mov     BYTE PTR [rdi+8], al
je      .L23
movzx   eax, BYTE PTR [rsi+9]
cmp     rcx, 10
mov     BYTE PTR [rdi+9], al
je      .L24
movzx   eax, BYTE PTR [rsi+10]
cmp     rcx, 11
mov     BYTE PTR [rdi+10], al
je      .L25
movzx   eax, BYTE PTR [rsi+11]
cmp     rcx, 12
mov     BYTE PTR [rdi+11], al
je      .L26
movzx   eax, BYTE PTR [rsi+12]
cmp     rcx, 13
mov     BYTE PTR [rdi+12], al
je      .L27
movzx   eax, BYTE PTR [rsi+13]
cmp     rcx, 15
mov     BYTE PTR [rdi+13], al
jne     .L28
movzx   eax, BYTE PTR [rsi+14]
mov     BYTE PTR [rdi+14], al
mov     eax, 15
.L4:
mov     r10, rdx
lea     r9, [rdx-1]
```


26.1. ВЕКТОРИЗАЦИЯ

```

sub    r10, rcx
lea    r8, [r10-16]
sub    r9, rcx
shr    r8, 4
add    r8, 1
mov    r11, r8
sal    r11, 4
cmp    r9, 14
jbe    .L6
lea    rbp, [rsi+rcx]
xor    r9d, r9d
add    rcx, rdi
xor    ebx, ebx

.L7:
movdqa xmm0, XMMWORD PTR [rbp+0+r9]
add    rbx, 1
movups XMMWORD PTR [rcx+r9], xmm0
add    r9, 16
cmp    rbx, r8
jb     .L7
add    rax, r11
cmp    r10, r11
je     .L1

.L6:
movzx  ecx, BYTE PTR [rsi+rax]
mov    BYTE PTR [rdi+rax], cl
lea    rcx, [rax+1]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+1+rax]
mov    BYTE PTR [rdi+1+rax], cl
lea    rcx, [rax+2]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+2+rax]
mov    BYTE PTR [rdi+2+rax], cl
lea    rcx, [rax+3]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+3+rax]
mov    BYTE PTR [rdi+3+rax], cl
lea    rcx, [rax+4]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+4+rax]
mov    BYTE PTR [rdi+4+rax], cl
lea    rcx, [rax+5]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+5+rax]
mov    BYTE PTR [rdi+5+rax], cl
lea    rcx, [rax+6]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+6+rax]
mov    BYTE PTR [rdi+6+rax], cl
lea    rcx, [rax+7]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+7+rax]
mov    BYTE PTR [rdi+7+rax], cl
lea    rcx, [rax+8]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+8+rax]
mov    BYTE PTR [rdi+8+rax], cl
lea    rcx, [rax+9]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+9+rax]

```

26.1. ВЕКТОРИЗАЦИЯ

```

mov     BYTE PTR [rdi+9+rax], cl
lea     rcx, [rax+10]
cmp     rdx, rcx
jbe     .L1
movzx   ecx, BYTE PTR [rsi+10+rax]
mov     BYTE PTR [rdi+10+rax], cl
lea     rcx, [rax+11]
cmp     rdx, rcx
jbe     .L1
movzx   ecx, BYTE PTR [rsi+11+rax]
mov     BYTE PTR [rdi+11+rax], cl
lea     rcx, [rax+12]
cmp     rdx, rcx
jbe     .L1
movzx   ecx, BYTE PTR [rsi+12+rax]
mov     BYTE PTR [rdi+12+rax], cl
lea     rcx, [rax+13]
cmp     rdx, rcx
jbe     .L1
movzx   ecx, BYTE PTR [rsi+13+rax]
mov     BYTE PTR [rdi+13+rax], cl
lea     rcx, [rax+14]
cmp     rdx, rcx
jbe     .L1
movzx   edx, BYTE PTR [rsi+14+rax]
mov     BYTE PTR [rdi+14+rax], dl
.L1:
pop     rbx
pop     rbp
.L41:
rep ret
.L13:
xor     eax, eax
.L3:
movzx   ecx, BYTE PTR [rsi+rax]
mov     BYTE PTR [rdi+rax], cl
add     rax, 1
cmp     rax, rdx
jne     .L3
rep ret
.L28:
mov     eax, 14
jmp     .L4
.L15:
mov     eax, 1
jmp     .L4
.L16:
mov     eax, 2
jmp     .L4
.L17:
mov     eax, 3
jmp     .L4
.L18:
mov     eax, 4
jmp     .L4
.L19:
mov     eax, 5
jmp     .L4
.L20:
mov     eax, 6
jmp     .L4
.L21:
mov     eax, 7
jmp     .L4
.L22:
mov     eax, 8
jmp     .L4
.L23:
mov     eax, 9
jmp     .L4

```

```
.L24:      mov     eax, 10
        jmp     .L4
.L25:      mov     eax, 11
        jmp     .L4
.L26:      mov     eax, 12
        jmp     .L4
.L27:      mov     eax, 13
        jmp     .L4
```

26.2. Реализация `strlen()` при помощи SIMD

Прежде всего, следует заметить, что SIMD-инструкции можно вставлять в Си/Си++ код при помощи специальных макросов⁷. В MSVC, часть находится в файле `intrin.h`.

Имеется возможность реализовать функцию `strlen()`⁸ при помощи SIMD-инструкций, работающий в 2-2.5 раза быстрее обычной реализации. Эта функция будет загружать в XMM-регистр сразу 16 байт и проверять каждый на ноль⁹.

```
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFF0) == (unsigned int)str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;

    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            _BitScanForward(&pos, mask);
            len += (size_t)pos;

            break;
        }
        s += sizeof(__m128i);
        len += sizeof(__m128i);
    };

    return len;
}
```

Компилируем в MSVC 2010 с опцией `/Ox`:

Листинг 26.2: Оптимизирующий MSVC 2010

```
_pos$75552 = -4          ; size = 4
_str$ = 8                ; size = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    push     ebp
    mov      ebp, esp
```

⁷MSDN: MMX, SSE, and SSE2 Intrinsics

⁸`strlen()` — стандартная функция Си для подсчета длины строки

⁹Пример базируется на исходнике отсюда: <http://go.yurichev.com/17330>.

26.2. РЕАЛИЗАЦИЯ `STRLEN()` ПРИ ПОМОЩИ `SIMD`

```
and     esp, -16                ; ffffffff0H
mov     eax, DWORD PTR _str$[ebp]
sub     esp, 12                 ; 0000000cH
push    esi
mov     esi, eax
and     esi, -16                ; ffffffff0H
xor     edx, edx
mov     ecx, eax
cmp     esi, eax
je      SHORT $LN4@strlen_sse
lea     edx, DWORD PTR [eax+1]
npad    3 ; выровнять следующую метку
$LL11@strlen_sse:
mov     cl, BYTE PTR [eax]
inc     eax
test    cl, cl
jne     SHORT $LL11@strlen_sse
sub     eax, edx
pop     esi
mov     esp, ebp
pop     ebp
ret     0
$LN4@strlen_sse:
movdqa  xmm1, XMMWORD PTR [eax]
pxor    xmm0, xmm0
pcmpeqb xmm1, xmm0
pmovmskb eax, xmm1
test    eax, eax
jne     SHORT $LN9@strlen_sse
$LL3@strlen_sse:
movdqa  xmm1, XMMWORD PTR [ecx+16]
add     ecx, 16                 ; 00000010H
pcmpeqb xmm1, xmm0
add     edx, 16                 ; 00000010H
pmovmskb eax, xmm1
test    eax, eax
je      SHORT $LL3@strlen_sse
$LN9@strlen_sse:
bsf     eax, eax
mov     ecx, eax
mov     DWORD PTR _pos$75552[esp+16], eax
lea     eax, DWORD PTR [ecx+edx]
pop     esi
mov     esp, ebp
pop     ebp
ret     0
?strlen_sse2@@YAIPBD@Z ENDP      ; strlen_sse2
```

Как это работает? Прежде всего, нужно определиться с целью этой ф-ции. Она вычисляет длину Си-строки, но можно сказать иначе — её задача это поиск нулевого байта, а затем вычисление его позиции относительно начала строки.

Итак, прежде всего, мы проверяем указатель `str`, выровнен ли он по 16-байтной границе. Если нет, то мы вызовем обычную реализацию `strlen()`.

Далее мы загружаем по 16 байт в регистр `XMM1` при помощи команды `MOVDQA`.

Наблюдательный читатель может спросить, почему в этом месте мы не можем использовать `MOVDQU`, которая может загружать откуда угодно не взирая на факт, выровнен ли указатель?

Да, можно было бы сделать вот как: если указатель выровнен, загружаем используя `MOVDQA`, иначе используем работающую чуть медленнее `MOVDQU`.

Однако здесь кроется не сразу заметная проблема, которая проявляется вот в чем:

В ОС линии [Windows NT](#) (и не только), память выделяется страницами по 4 KiB (4096 байт). Каждый win32-процесс якобы имеет в наличии 4 GiB, но на самом деле, только некоторые части этого адресного пространства присоединены к реальной физической памяти. Если процесс обратится к блоку памяти, которого не существует, сработает исключение. Так работает [VM](#)¹⁰.

¹⁰[wikipedia](#)

26.2. РЕАЛИЗАЦИЯ `STRLEN()` ПРИ ПОМОЩИ `SIMD`

Так вот, функция, читающая сразу по 16 байт, имеет возможность нечаянно вылезти за границу выделенного блока памяти. Предположим, ОС выделила программе 8192 (0x2000) байт по адресу 0x008c0000. Таким образом, блок занимает байты с адреса 0x008c0000 по 0x008c1fff включительно.

За этим блоком, то есть начиная с адреса 0x008c2000 нет вообще ничего, т.е. ОС не выделяла там память. Обращение к памяти начиная с этого адреса вызовет исключение.

И предположим, что программа хранит некую строку из, скажем, пяти символов почти в самом конце блока, что не является преступлением:

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	здесь случайный мусор
0x008c1fff	здесь случайный мусор

В обычных условиях, программа вызывает `strlen()` передав ей указатель на строку `'hello'` лежащую по адресу 0x008c1ff8. `strlen()` будет читать по одному байту до 0x008c1ffd, где ноль, и здесь она закончит работу.

Теперь, если мы напишем свою реализацию `strlen()` читающую сразу по 16 байт, с любого адреса, будь он выровнен по 16-байтной границе или нет, `MOVDQU` попытается загрузить 16 байт с адреса 0x008c1ff8 по 0x008c2008, и произойдет исключение. Это ситуация которой, конечно, хочется избежать.

Поэтому мы будем работать только с адресами, выровненными по 16 байт, что в сочетании со знанием что размер страницы ОС также, как правило, выровнен по 16 байт, даст некоторую гарантию что наша функция не будет пытаться читать из мест в невыделенной памяти.

Вернемся к нашей функции.

`_mm_setzero_si128()` – это макрос, генерирующий `pxor xmm0, xmm0` – инструкция просто обнуляет регистр `XMM0`.

`_mm_load_si128()` – это макрос для `MOVDQA`, он просто загружает 16 байт по адресу из указателя в `XMM1`.

`_mm_cmpeq_epi8()` – это макрос для `PCMPEQB`, это инструкция, которая побайтово сравнивает значения из двух XMM регистров.

И если какой-то из байт равен другому, то в результирующем значении будет выставлено на месте этого байта `0xff`, либо 0, если байты не были равны.

Например.

XMM1: 11223344556677880000000000000000

XMM0: 11ab3444007877881111111111111111

После исполнения `pcmpeqb xmm1, xmm0`, регистр `XMM1` содержит:

XMM1: ff0000ff0000ffff0000000000000000

Эта инструкция в нашем случае, сравнивает каждый 16-байтный блок с блоком состоящим из 16-и нулевых байт, выставленным в `XMM0` при помощи `pxor xmm0, xmm0`.

Следующий макрос `_mm_movemask_epi8()` – это инструкция `PMOVBMSKB`.

Она очень удобна как раз для использования в паре с `PCMPEQB`.

`pmovmskb eax, xmm1`

Эта инструкция выставит самый первый бит `EAX` в единицу, если старший бит первого байта в регистре `XMM1` является единицей. Иными словами, если первый байт в регистре `XMM1` является `0xff`, то первый бит в `EAX` будет также единицей, иначе нулем.

Если второй байт в регистре `XMM1` является `0xff`, то второй бит в `EAX` также будет единицей. Иными словами, инструкция отвечает на вопрос, «какие из байт в `XMM1` являются `0xff`?». В результате подготовит 16 бит и запишет в `EAX`. Остальные биты в `EAX` обнулятся.

Кстати, не забывайте также вот о какой особенности нашего алгоритма. На вход может прийти 16 байт вроде:

Глава 27

64 бита

27.1. x86-64

Это расширение x86-архитектуры до 64 бит.

С точки зрения начинающего reverse engineer-а, наиболее важные отличия от 32-битного x86 это:

- Почти все регистры (кроме FPU и SIMD) расширены до 64-бит и получили префикс R-. И еще 8 регистров добавлено. В итоге имеются эти GPR-ы: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

К ним также можно обращаться так же, как и прежде. Например, для доступа к младшим 32 битам RAX можно использовать EAX:

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RAX ^{x64}							
				EAX			
						AX	
						AH	AL

У новых регистров R8-R15 также имеются их младшие части: R8D-R15D (младшие 32-битные части), R8W-R15W (младшие 16-битные части), R8L-R15L (младшие 8-битные части).

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R8							
				R8D			
						R8W	
						R8L	

Удвоено количество SIMD-регистров: с 8 до 16: XMM0 - XMM15.

- В win64 передача всех параметров немного иная, это немного похоже на fastcall (65.3 (стр. 673)). Первые 4 аргумента записываются в регистры RCX, RDX, R8, R9, а остальные — в стек. Вызывающая функция также должна подготовить место из 32 байт чтобы вызываемая функция могла сохранить там первые 4 аргумента и использовать эти регистры по своему усмотрению. Короткие функции могут использовать аргументы прямо из регистров, но большие функции могут сохранять их значения на будущее.

Соглашение System V AMD64 ABI (Linux, *BSD, Mac OS X)[Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, System V Application Binary Interface. AMD64 Architecture Processor Supplement, (2013)]¹ также напоминает fastcall, использует 6 регистров RDI, RSI, RDX, RCX, R8, R9 для первых шести аргументов. Остальные передаются через стек.

См. также в соответствующем разделе о способах передачи аргументов через стек (65 (стр. 672)).

- int в Си/Си++ остается 32-битным для совместимости.
- Все указатели теперь 64-битные.

¹Также доступно здесь: <http://x86-64.org/documentation/abi.pdf>

На это иногда сетуют: ведь теперь для хранения всех указателей нужно в 2 раза больше места в памяти, в т.ч. и в кэш-памяти, не смотря на то что x64-процессоры могут адресовать только 48 бит внешней RAM².

Из-за того, что регистров общего пользования теперь вдвое больше, у компиляторов теперь больше свободного места для маневра, называемого [register allocation](#). Для нас это означает, что в итоговом коде будет меньше локальных переменных.

Для примера, функция вычисляющая первый S-бокс алгоритма шифрования DES, она обрабатывает сразу 32/64/128/256 значений, в зависимости от типа `DES_type` (`uint32`, `uint64`, `SSE2` или `AVX`), методом `bitslice DES` (больше об этом методе читайте здесь ([26](#) (стр. [403](#))))

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x5;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
```

²Random-access memory

27.1. X86-64

```
*out2 ^= x22;
x23 = x1 | x5;
x24 = x23 ^ x8;
x25 = x18 & ~x2;
x26 = a2 & ~x25;
x27 = x24 ^ x26;
x28 = x6 | x7;
x29 = x28 ^ x25;
x30 = x9 ^ x24;
x31 = x18 & ~x30;
x32 = a2 & x31;
x33 = x29 ^ x32;
x34 = a1 & x33;
x35 = x27 ^ x34;
*out4 ^= x35;
x36 = a3 & x28;
x37 = x18 & ~x36;
x38 = a2 | x3;
x39 = x37 ^ x38;
x40 = a3 | x31;
x41 = x24 & ~x37;
x42 = x41 | x3;
x43 = x42 & ~a2;
x44 = x40 ^ x43;
x45 = a1 & ~x44;
x46 = x39 ^ ~x45;
*out1 ^= x46;
x47 = x33 & ~x9;
x48 = x47 ^ x39;
x49 = x4 ^ x36;
x50 = x49 & ~x5;
x51 = x42 | x18;
x52 = x51 ^ a5;
x53 = a2 & ~x52;
x54 = x50 ^ x53;
x55 = a1 | x54;
x56 = x48 ^ ~x55;
*out3 ^= x56;
}
```

Здесь много локальных переменных. Конечно, далеко не все они будут в локальном стеке. Компилируем обычным MSVC 2008 с опцией `/Ox` :

Листинг 27.1: Оптимизирующий MSVC 2008

```
PUBLIC _s1
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_x6$ = -20 ; size = 4
_x3$ = -16 ; size = 4
_x1$ = -12 ; size = 4
_x8$ = -8 ; size = 4
_x4$ = -4 ; size = 4
_a1$ = 8 ; size = 4
_a2$ = 12 ; size = 4
_a3$ = 16 ; size = 4
_x33$ = 20 ; size = 4
_x7$ = 20 ; size = 4
_a4$ = 20 ; size = 4
_a5$ = 24 ; size = 4
tv326 = 28 ; size = 4
_x36$ = 28 ; size = 4
_x28$ = 28 ; size = 4
_a6$ = 28 ; size = 4
_out1$ = 32 ; size = 4
_x24$ = 36 ; size = 4
_out2$ = 36 ; size = 4
_out3$ = 40 ; size = 4
_out4$ = 44 ; size = 4
_s1 PROC
```

```

sub     esp, 20                      ; 00000014H
mov     edx, DWORD PTR _a5$[esp+16]
push    ebx
mov     ebx, DWORD PTR _a4$[esp+20]
push    ebp
push    esi
mov     esi, DWORD PTR _a3$[esp+28]
push    edi
mov     edi, ebx
not     edi
mov     ebp, edi
and     edi, DWORD PTR _a5$[esp+32]
mov     ecx, edx
not     ecx
and     ebp, esi
mov     eax, ecx
and     eax, esi
and     ecx, ebx
mov     DWORD PTR _x1$[esp+36], eax
xor     eax, ebx
mov     esi, ebp
or      esi, edx
mov     DWORD PTR _x4$[esp+36], esi
and     esi, DWORD PTR _a6$[esp+32]
mov     DWORD PTR _x7$[esp+32], ecx
mov     edx, esi
xor     edx, eax
mov     DWORD PTR _x6$[esp+36], edx
mov     edx, DWORD PTR _a3$[esp+32]
xor     edx, ebx
mov     ebx, esi
xor     ebx, DWORD PTR _a5$[esp+32]
mov     DWORD PTR _x8$[esp+36], edx
and     ebx, edx
mov     ecx, edx
mov     edx, ebx
xor     edx, ebp
or      edx, DWORD PTR _a6$[esp+32]
not     ecx
and     ecx, DWORD PTR _a6$[esp+32]
xor     edx, edi
mov     edi, edx
or      edi, DWORD PTR _a2$[esp+32]
mov     DWORD PTR _x3$[esp+36], ebp
mov     ebp, DWORD PTR _a2$[esp+32]
xor     edi, ebx
and     edi, DWORD PTR _a1$[esp+32]
mov     ebx, ecx
xor     ebx, DWORD PTR _x7$[esp+32]
not     edi
or      ebx, ebp
xor     edi, ebx
mov     ebx, edi
mov     edi, DWORD PTR _out2$[esp+32]
xor     ebx, DWORD PTR [edi]
not     eax
xor     ebx, DWORD PTR _x6$[esp+36]
and     eax, edx
mov     DWORD PTR [edi], ebx
mov     ebx, DWORD PTR _x7$[esp+32]
or      ebx, DWORD PTR _x6$[esp+36]
mov     edi, esi
or      edi, DWORD PTR _x1$[esp+36]
mov     DWORD PTR _x28$[esp+32], ebx
xor     edi, DWORD PTR _x8$[esp+36]
mov     DWORD PTR _x24$[esp+32], edi
xor     edi, ecx
not     edi
and     edi, edx
mov     ebx, edi

```

27.1. X86-64

```

and     ebx, ebp
xor     ebx, DWORD PTR _x28$[esp+32]
xor     ebx, eax
not     eax
mov     DWORD PTR _x33$[esp+32], ebx
and     ebx, DWORD PTR _a1$[esp+32]
and     eax, ebp
xor     eax, ebx
mov     ebx, DWORD PTR _out4$[esp+32]
xor     eax, DWORD PTR [ebx]
xor     eax, DWORD PTR _x24$[esp+32]
mov     DWORD PTR [ebx], eax
mov     eax, DWORD PTR _x28$[esp+32]
and     eax, DWORD PTR _a3$[esp+32]
mov     ebx, DWORD PTR _x3$[esp+36]
or      edi, DWORD PTR _a3$[esp+32]
mov     DWORD PTR _x36$[esp+32], eax
not     eax
and     eax, edx
or      ebx, ebp
xor     ebx, eax
not     eax
and     eax, DWORD PTR _x24$[esp+32]
not     ebp
or      eax, DWORD PTR _x3$[esp+36]
not     esi
and     ebp, eax
or      eax, edx
xor     eax, DWORD PTR _a5$[esp+32]
mov     edx, DWORD PTR _x36$[esp+32]
xor     edx, DWORD PTR _x4$[esp+36]
xor     ebp, edi
mov     edi, DWORD PTR _out1$[esp+32]
not     eax
and     eax, DWORD PTR _a2$[esp+32]
not     ebp
and     ebp, DWORD PTR _a1$[esp+32]
and     edx, esi
xor     eax, edx
or      eax, DWORD PTR _a1$[esp+32]
not     ebp
xor     ebp, DWORD PTR [edi]
not     ecx
and     ecx, DWORD PTR _x33$[esp+32]
xor     ebp, ebx
not     eax
mov     DWORD PTR [edi], ebp
xor     eax, ecx
mov     ecx, DWORD PTR _out3$[esp+32]
xor     eax, DWORD PTR [ecx]
pop     edi
pop     esi
xor     eax, ebx
pop     ebp
mov     DWORD PTR [ecx], eax
pop     ebx
add     esp, 20                ; 00000014H
ret     0
_s1     ENDP

```

5 переменных компилятору пришлось разместить в локальном стеке.

Теперь попробуем то же самое только в 64-битной версии MSVC 2008:

Листинг 27.2: Оптимизирующий MSVC 2008

```

a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88

```

27.1. X86-64

```
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1 PROC
$LN3:
    mov     QWORD PTR [rsp+24], rbx
    mov     QWORD PTR [rsp+32], rbp
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+8], rcx
    push    rsi
    push    rdi
    push    r12
    push    r13
    push    r14
    push    r15
    mov     r15, QWORD PTR a5$[rsp]
    mov     rcx, QWORD PTR a6$[rsp]
    mov     rbp, r8
    mov     r10, r9
    mov     rax, r15
    mov     rdx, rbp
    not     rax
    xor     rdx, r9
    not     r10
    mov     r11, rax
    and     rax, r9
    mov     rsi, r10
    mov     QWORD PTR x36$1$[rsp], rax
    and     r11, r8
    and     rsi, r8
    and     r10, r15
    mov     r13, rdx
    mov     rbx, r11
    xor     rbx, r9
    mov     r9, QWORD PTR a2$[rsp]
    mov     r12, rsi
    or      r12, r15
    not     r13
    and     r13, rcx
    mov     r14, r12
    and     r14, rcx
    mov     rax, r14
    mov     r8, r14
    xor     r8, rbx
    xor     rax, r15
    not     rbx
    and     rax, rdx
    mov     rdi, rax
    xor     rdi, rsi
    or      rdi, rcx
    xor     rdi, r10
    and     rbx, rdi
    mov     rcx, rdi
    or      rcx, r9
    xor     rcx, rax
    mov     rax, r13
    xor     rax, QWORD PTR x36$1$[rsp]
    and     rcx, QWORD PTR a1$[rsp]
    or      rax, r9
    not     rcx
    xor     rcx, rax
    mov     rax, QWORD PTR out2$[rsp]
    xor     rcx, QWORD PTR [rax]
    xor     rcx, r8
    mov     QWORD PTR [rax], rcx
    mov     rax, QWORD PTR x36$1$[rsp]
    mov     rcx, r14
```

```

or    rax, r8
or    rcx, r11
mov   r11, r9
xor   rcx, rdx
mov   QWORD PTR x36$1$(rsp), rax
mov   r8, rsi
mov   rdx, rcx
xor   rdx, r13
not   rdx
and   rdx, rdi
mov   r10, rdx
and   r10, r9
xor   r10, rax
xor   r10, rbx
not   rbx
and   rbx, r9
mov   rax, r10
and   rax, QWORD PTR a1$(rsp)
xor   rbx, rax
mov   rax, QWORD PTR out4$(rsp)
xor   rbx, QWORD PTR [rax]
xor   rbx, rcx
mov   QWORD PTR [rax], rbx
mov   rbx, QWORD PTR x36$1$(rsp)
and   rbx, rbp
mov   r9, rbx
not   r9
and   r9, rdi
or    r8, r11
mov   rax, QWORD PTR out1$(rsp)
xor   r8, r9
not   r9
and   r9, rcx
or    rdx, rbp
mov   rbp, QWORD PTR [rsp+80]
or    r9, rsi
xor   rbx, r12
mov   rcx, r11
not   rcx
not   r14
not   r13
and   rcx, r9
or    r9, rdi
and   rbx, r14
xor   r9, r15
xor   rcx, rdx
mov   rdx, QWORD PTR a1$(rsp)
not   r9
not   rcx
and   r13, r10
and   r9, r11
and   rcx, rdx
xor   r9, rbx
mov   rbx, QWORD PTR [rsp+72]
not   rcx
xor   rcx, QWORD PTR [rax]
or    r9, rdx
not   r9
xor   rcx, r8
mov   QWORD PTR [rax], rcx
mov   rax, QWORD PTR out3$(rsp)
xor   r9, r13
xor   r9, QWORD PTR [rax]
xor   r9, r8
mov   QWORD PTR [rax], r9
pop   r15
pop   r14
pop   r13
pop   r12
pop   rdi

```

27.2. ARM

```
pop    rsi
ret    0
s1     ENDP
```

Компилятор ничего не выделил в локальном стеке, а `x36` это синоним для `a5`.

Кстати, существуют процессоры с еще большим количеством [GPR](#), например, Itanium — 128 регистров.

27.2. ARM

64-битные инструкции появились в ARMv8.

27.3. Числа с плавающей запятой

О том как происходит работа с числами с плавающей запятой в x86-64, читайте здесь: [28](#) (стр. [425](#)).

Глава 28

Работа с числами с плавающей запятой используя SIMD

Разумеется, FPU остался в x86-совместимых процессорах в то время, когда ввели расширения SIMD .

SIMD-расширения (SSE2) позволяют удобнее работать с числами с плавающей запятой.

Формат чисел остается тот же (IEEE 754).

Так что современные компиляторы (включая те, что компилируют под x86-64) обычно используют SIMD-инструкции вместо FPU-инструкций.

Это, можно сказать, хорошая новость, потому что работать с ними легче .

Примеры будем использовать из секции о FPU : 18 (стр. 214).

28.1. Простой пример

```
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
};
```

28.1.1. x64

Листинг 28.1: Оптимизирующий MSVC 2012 x64

```
__real@4010666666666666 DQ 0401066666666666r    ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f      PROC
        divsd    xmm0, QWORD PTR __real@40091eb851eb851f
        mulsd    xmm1, QWORD PTR __real@4010666666666666
        addsd    xmm0, xmm1
        ret      0
f      ENDP
```

Собственно, входные значения с плавающей запятой передаются через регистры XMM0 - XMM3 , а остальные — через стек ¹.

¹MSDN: Parameter Passing

28.1. ПРОСТОЙ ПРИМЕР

a передается через `XMM0`, *b* — через `XMM1`. Но XMM-регистры (как мы уже знаем из секции о [SIMD: 26](#) (стр. 403)) 128-битные, а значения типа *double* — 64-битные, так что используется только младшая половина регистра.

`DIVSD` это SSE-инструкция, означает «Divide Scalar Double-Precision Floating-Point Values», и просто делит значение типа *double* на другое, лежащие в младших половинах операндов.

Константы закодированы компилятором в формате IEEE 754.

`MULSD` и `ADDSD` работают так же, только производят умножение и сложение.

Результат работы функции типа *double* функция оставляет в регистре `XMM0`.

Как работает неоптимизирующий MSVC:

Листинг 28.2: MSVC 2012 x64

```
__real@4010666666666666 DQ 0401066666666666r    ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14

a$ = 8
b$ = 16
f      PROC
    movsdx  QWORD PTR [rsp+16], xmm1
    movsdx  QWORD PTR [rsp+8], xmm0
    movsdx  xmm0, QWORD PTR a$[rsp]
    divsd   xmm0, QWORD PTR __real@40091eb851eb851f
    movsdx  xmm1, QWORD PTR b$[rsp]
    mulsd   xmm1, QWORD PTR __real@4010666666666666
    addsd   xmm0, xmm1
    ret     0
f      ENDP
```

Чуть более избыточно. Входные аргументы сохраняются в «shadow space» ([9.2.1](#) (стр. 96)), причем, только младшие половины регистров, т.е. только 64-битные значения типа *double*. Результат работы компилятора GCC точно такой же.

28.1.2. x86

Скомпилируем этот пример также и под x86. MSVC 2012 даже генерируя под x86, использует SSE2-инструкции:

Листинг 28.3: Неоптимизирующий MSVC 2012 x86

```
tv70 = -8      ; size = 8
_a$ = 8        ; size = 8
_b$ = 16       ; size = 8
_f      PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 8
    movsd   xmm0, QWORD PTR _a$[ebp]
    divsd   xmm0, QWORD PTR __real@40091eb851eb851f
    movsd   xmm1, QWORD PTR _b$[ebp]
    mulsd   xmm1, QWORD PTR __real@4010666666666666
    addsd   xmm0, xmm1
    movsd   QWORD PTR tv70[ebp], xmm0
    fld     QWORD PTR tv70[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f      ENDP
```

Листинг 28.4: Оптимизирующий MSVC 2012 x86

```
tv67 = 8       ; size = 8
_a$ = 8        ; size = 8
_b$ = 16       ; size = 8
_f      PROC
    movsd   xmm1, QWORD PTR _a$[esp-4]
    divsd   xmm1, QWORD PTR __real@40091eb851eb851f
    movsd   xmm0, QWORD PTR _b$[esp-4]
    mulsd   xmm0, QWORD PTR __real@4010666666666666
```


28.1. ПРОСТОЙ ПРИМЕР

```
    addsd    xmm1, xmm0
    movsd    QWORD PTR tv67[esp-4], xmm1
    fld      QWORD PTR tv67[esp-4]
    ret      0
_f      ENDP
```

Код почти такой же, правда есть пара отличий связанных с соглашениями о вызовах: 1) аргументы передаются не в XMM-регистрах, а через стек, как и прежде, в примерах с FPU ([18](#) (стр. [214](#))); 2) результат работы функции возвращается через `ST(0)` — для этого он через стек (через локальную переменную `tv`) копируется из XMM-регистра в `ST(0)`.

28.1. ПРОСТОЙ ПРИМЕР

Попробуем оптимизированный пример в OllyDbg:

The screenshot displays the OllyDbg interface with the following components:

- Assembly Window:** Shows assembly instructions for the CPU - main thread, module simple. Key instructions include:
 - `F20F104C24 0 MOVSD XMM1,QWORD PTR SS:[ESP+4]`
 - `F20F5E0D C02 DIUSD XMM1,QWORD PTR DS:[13320C0]`
 - `F20F104424 0 MOVSD XMM0,QWORD PTR SS:[ESP+0C]`
 - `F20F5905 D02 MULSD XMM0,QWORD PTR DS:[13320D0]`
 - `F20F58C8 ADDSD XMM1,XMM0`
 - `F20F114C24 0 MOVSD QWORD PTR SS:[ESP+4],XMM1`
 - `DD4424 04 FLD QWORD PTR SS:[ESP+4]`
 - `F20F1005 C82 MOVSD XMM0,QWORD PTR DS:[13320C8]`
 - `83EC 10 SUB ESP,10`
 - `F20F114424 0 MOVSD QWORD PTR SS:[ESP+8],XMM0`
 - `F20F1005 B82 MOVSD XMM0,QWORD PTR DS:[13320B8]`
 - `F20F110424 MOVSD QWORD PTR SS:[ESP],XMM0`
 - `E8 ADFFFFFF CALL 01331000`
 - `DD5C24 08 FSTP QWORD PTR SS:[ESP+8]`
 - `83C4 08 ADD ESP,8`
 - `68 00303301 PUSH OFFSET 01333000`
 - `FF15 00203300 CALL QWORD PTR DS:[<&MSUCR110.printf>]`
 - `83C4 0C ADD ESP,0C`
 - `33C0 XOR EAX,EAX`
 - `8B 405A0000 MOV EAX,5A40`
 - `66:3905 0000 CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEAD`
 - `74 04 JE SHORT 01331082`
 - `33C0 XOR EAX,EAX`
 - `EB 34 JMP SHORT 013310B6`
 - `8B0D 3C003300 MOV ECX,QWORD PTR DS:[133003C]`
 - `81B9 00003300 CMP DWORD PTR DS:[ECX+<STRUCT IMAGE_DOS`
 - `75 EA JNE SHORT 0133107E`
 - `8B 0B010000 MOV EAX,10B`
 - `66:3901 1300 CMP WORD PTR DS:[ECX+1330018],AX`
- Registers (FPU) Window:** Shows the state of floating-point registers. XMM0 and XMM1 contain floating-point values:
 - XMM0: 0.0
 - XMM1: 1.2000000000000000
 - XMM2: 0.0
 - XMM3: 0.0
 - XMM4: 0.0
 - XMM5: 0.0
 - XMM6: 0.0
 - XMM7: 0.0
- Memory Dump Window:** Shows a hex dump of memory starting at address 01333000. The dump includes hex values, ASCII characters, and comments like "RETURN from simple.01331000 to simple.013310B6".

Рис. 28.1: OllyDbg: MOVSD загрузила значение α в XMM1

CPU - main thread, module simple

01331000	MOVSD XMM1, QWORD PTR SS:[ESP+4]
01331006	DIVSD XMM1, QWORD PTR DS:[13320C0]
0133100E	MOVSD XMM0, QWORD PTR SS:[ESP+0C]
01331014	MULSD XMM0, QWORD PTR DS:[13320D0]
0133101C	ADDSD XMM1, XMM0
01331020	MOVSD QWORD PTR SS:[ESP+4], XMM1
01331026	FILD QWORD PTR SS:[ESP+4]
0133102A	RETN
0133102B	INT3
0133102C	INT3
0133102D	INT3
0133102E	INT3
0133102F	INT3
01331030	MOVSD XMM0, QWORD PTR DS:[13320C8]
01331038	SUB ESP, 10
0133103B	MOVSD QWORD PTR SS:[ESP+8], XMM0
01331041	MOVSD XMM0, QWORD PTR DS:[13320B8]
01331049	MOVSD QWORD PTR SS:[ESP], XMM0
0133104E	CALL 01331000
01331053	FSTP QWORD PTR SS:[ESP+8]
01331057	ADD ESP, 8
0133105A	PUSH OFFSET 01333000
0133105F	CALL QWORD PTR DS:[&MSUCR110.printf]
01331065	ADD ESP, 0C
01331068	XOR EAX, EAX
0133106A	RETN
0133106B	INT3
0133106C	INT3
0133106D	INT3
0133106E	INT3
0133106F	INT3
01331070	MOV EAX, 5A4D
01331075	CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEAD
0133107C	JE SHORT 01331082
0133107E	XOR EAX, EAX
01331080	JMP SHORT 013310B6
01331082	MOV ECK, DWORD PTR DS:[133003C]
01331088	CMP DWORD PTR DS:LECX+<STRUCT IMAGE_DOS
01331092	JNE SHORT 0133107E
01331094	MOV EAX, 10B
01331096	CMP WORD PTR DS:[ECX+1330018] AX

Registers (FPU)

EAX	68F88634	MSUCR110.__initenv
ECX	0066D530	
EDX	00000000	
EBX	00000000	
ESP	0017FBC4	
EBP	0017FC10	
ESI	00000001	
EDI	00000000	
EIP	0133100E	simple.0133100E
C 0	ES 002B	32bit 0(FFFFFFFF)
P 0	CS 0023	32bit 0(FFFFFFFF)
A 0	SS 002B	32bit 0(FFFFFFFF)
Z 0	DS 002B	32bit 0(FFFFFFFF)
S 0	FS 0053	32bit 7EFD0000(FFF)
T 0	GS 002B	32bit 0(FFFFFFFF)
D 0		
O 0	LastErr	00000000 ERROR_SUCCESS
EFL	00000202	(NO, NB, NE, A, NS, PO, GE, G)
ST0	empty	0.0
ST1	empty	0.0
ST2	empty	0.0
ST3	empty	0.0
ST4	empty	0.0
ST5	empty	0.0
ST6	empty	0.0
ST7	empty	0.0
FST	0000	Cond 0 0 0 0 Err 0 0 0 0 0 0 (GT)
FCW	027F	Prec NEAR, 53 Mask 1 1 1 1 1 1
Last cmd	0000:00000000	
XMM0		0.0 1.2000000000000000
XMM1		0.0 0.3821656050955414
XMM2		0.0 0.0
XMM3		0.0 0.0
XMM4		0.0 0.0
XMM5		0.0 0.0
XMM6		0.0 0.0
XMM7		0.0 0.0
MXCSR	00001FA0	FZ 0 DZ 0 Err 1 0 0 0 0 0
		Rnd NEAR Mask 1 1 1 1 1 1

Stack [0017FBC4]=3.4000000000000000

Address	Hex dump	ASCII (A)
01333000	25 66 0A 00 00 00 00 00 00 00 00 00 00 00 00 00	%f
01333010	01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0
01333020	FE FF FF FF FF FF FF FF BB 15 5B 87 44 EA A4 78	
01333030	00 00 00 00 00 00 00 00 01 00 00 00 00 AC 66 00	
01333040	30 D5 66 00 00 00 00 00 00 00 00 00 00 00 00 00	0ff
01333050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
01333060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
01333070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
01333080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
01333090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
01333100	00 00 00 00	

429

[illegible]

430

CPU - main thread, module simple

Address	Disassembly	Comment
01331000	MOVSD XMM1,QWORD PTR SS:[ESP+4]	
01331006	DIVSD XMM1,QWORD PTR DS:[13320C0]	Float 3.1400
0133100E	MOVSD XMM0,QWORD PTR SS:[ESP+0C]	
01331014	MULSD XMM0,QWORD PTR DS:[13320D0]	Float 4.1000
0133101C	ADDSD XMM1,XMM0	
01331020	MOVSD QWORD PTR SS:[ESP+4],XMM1	Float 0.0, 1
01331026	FLD QWORD PTR SS:[ESP+4]	
0133102A	RETN	
0133102B	INT3	
0133102C	CC	
0133102D	INT3	
0133102E	CC	
0133102F	INT3	
01331030	MOVSD XMM0,QWORD PTR DS:[13320C8]	Float 3.4000
01331038	SUB ESP,10	
0133103B	MOVSD QWORD PTR SS:[ESP+8],XMM0	
01331041	MOVSD XMM0,QWORD PTR DS:[13320B8]	Float 1.2000
01331049	MOVSD QWORD PTR SS:[ESP],XMM0	
0133104E	CALL 01331000	
01331053	FSTP QWORD PTR SS:[ESP+8]	
01331057	ADD ESP,8	
0133105A	PUSH OFFSET 01333000	
0133105F	CALL DWORD PTR DS:[<MSUCR110.printf>]	ASCII "%f"
01331065	ADD ESP,0C	
01331068	XOR EAX,EAX	
0133106A	CC	
0133106B	INT3	
0133106C	CC	
0133106D	INT3	
0133106E	CC	
0133106F	INT3	
01331070	MOV EAX,5A40	
01331075	CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEAD	
0133107C	JE SHORT 01331082	
0133107E	XOR EAX,EAX	
01331080	JMP SHORT 013310B6	
01331082	MOV ECX,QWORD PTR DS:[133003C]	
01331088	CMP DWORD PTR DS:[ECX+<STRUCT IMAGE_DOS	
01331092	JNE SHORT 0133107E	
01331094	MOV EAX,10B	
01331098	CMP WORD PTR DS:[ECX+1330018],AX	

XMM1=0.0, 14.32216560509554
Stack [0017FBC4]=1.2000000000000000

Registers (FPU)

Register	Value
EAX	68F88634 MSUCR110.__initenv
ECX	0066D530
EDX	00000000
EBX	00000000
ESP	0017FBC4
EBP	0017FC10
ESI	00000001
EDI	00000000
EIP	01331020 simple.01331020
C 0	ES 002B 32bit 0(FFFFFFFF)
P 0	CS 0023 32bit 0(FFFFFFFF)
A 0	SS 002B 32bit 0(FFFFFFFF)
Z 0	DS 002B 32bit 0(FFFFFFFF)
S 0	FS 0053 32bit 7EFD0000(FFF)
T 0	GS 002B 32bit 0(FFFFFFFF)
D 0	
O 0	LastErr 00000000 ERROR_SUCCESS
EFL	00000202 (NO,NB,NE,A,NS,PO,GE,G)
ST0	empty 0.0
ST1	empty 0.0
ST2	empty 0.0
ST3	empty 0.0
ST4	empty 0.0
ST5	empty 0.0
ST6	empty 0.0
ST7	empty 0.0
FST	0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 (GT)
FCW	027F Prec NEAR,53 Mask 1 1 1 1 1 1
Last cmd	0000:00000000
XMM0	0.0 13.940000000000000
XMM1	0.0 14.32216560509554
XMM2	0.0 0.0
XMM3	0.0 0.0
XMM4	0.0 0.0
XMM5	0.0 0.0
XMM6	0.0 0.0
XMM7	0.0 0.0
MXCSR	00001FA0 FZ 0 DZ 0 Err 1 0 0 0 0 0
	Rnd NEAR Mask 1 1 1 1 1 1

Address	Hex dump	ASCII (A)
01333000	25 66 0A 00 00 00 00 00 00 00 00 00 00 00 00 00	%f
01333010	01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0
01333020	FE FF FF FF FF FF FF FF BB 15 5B 87 44 EA AA 78	
01333030	00 00 00 00 00 00 00 00 01 00 00 00 00 AC 66 00	
01333040	30 D5 66 00 00 00 00 00 00 00 00 00 00 00 00 00	0ff
01333050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
01333060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
01333070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
01333080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
01333090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
013330F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

431

28.1. ПРОСТОЙ ПРИМЕР

The screenshot displays the OllyDbg interface with three main panes:

- Assembly Window:** Shows assembly instructions for the CPU thread. Key instructions include:
 - `F20F104C24 0 MOUSD XMM1,QWORD PTR SS:[ESP+4]`
 - `F20F5E00 C02 DIUSD XMM1,QWORD PTR DS:[13320C0]`
 - `F20F104424 0 MOUSD XMM0,QWORD PTR SS:[ESP+0C]`
 - `F20F5905 002 MULSD XMM0,QWORD PTR DS:[13320D0]`
 - `F20F5808 ADDSD XMM1,XMM0`
 - `F20F114C24 0 MOUSD QWORD PTR SS:[ESP+4],XMM1`
 - `DD4424 04 FLD QWORD PTR SS:[ESP+4]`
 - `0133102A C3 RETN`
 - `83EC 10 SUB ESP,10`
 - `F20F114424 0 MOUSD QWORD PTR SS:[ESP+8],XMM0`
 - `F20F1005 B82 MOUSD XMM0,QWORD PTR DS:[13320B8]`
 - `F20F110424 MOUSD QWORD PTR SS:[ESP],XMM0`
 - `E8 ADFFFFFF CALL 01331000`
 - `DD5C24 08 STP QWORD PTR SS:[ESP+8]`
 - `83C4 08 ADD ESP,8`
 - `68 00303301 PUSH OFFSET 01330000`
 - `FF15 30203300 CALL QWORD PTR DS:[<&MSUCR110.printf>]`
 - `83C4 0C ADD ESP,0C`
 - `33C0 XOR EAX,EAX`
 - `C3 RETN`
 - `B8 4D5A0000 MOV EAX,5A4D`
 - `66:3905 0000 CMP WORD PTR DS:[<STRUCT IMAGE_DOS_HEAD`
 - `74 04 JE SHORT 01331082`
 - `33C0 XOR EAX,EAX`
 - `EB 34 JMP SHORT 013310B6`
 - `8B0D 3C00330 MOV ECX,QWORD PTR DS:[133003C]`
 - `81B9 0000330 CMP QWORD PTR DS:[ECX+<STRUCT IMAGE_DOS`
 - `75 EA JNE SHORT 0133107E`
 - `B8 0B010000 MOV EAX,10B`
 - `66:3981 1800 CMP WORD PTR DS:[ECX+13300181] BY`
- Registers (FPU) Window:** Shows the state of the floating-point unit registers. ST(0) is highlighted as **valid 14.322165605095539930**. Other registers (ST1-ST7) are empty. The control word (C0) is 002B, and the status word (S0) is 0053.
- Stack Window:** Shows the current stack frame. The top of the stack is at `0017FBC0`, which corresponds to the `FLD` instruction. The stack contains various data, including the return address `01331053`.

Рис. 28.5: OllyDbg: `FLD` оставляет результат функции в `ST(0)`

Видно, что OllyDbg показывает XMM-регистры как пары чисел в формате *double*, но используется только младшая часть. Должно быть, OllyDbg показывает их именно так, потому что сейчас исполняются SSE2-инструкции с суффиксом `-SD`. Но конечно же, можно переключить отображение значений в регистрах и посмотреть содержимое как 4 *float*-числа или просто как 16 байт.

28.2. Передача чисел с плавающей запятой в аргументах

```
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

    return 0;
}
```

Они передаются в младших половинах регистров **XMM0** - **XMM3**.

Листинг 28.5: Оптимизирующий MSVC 2012 x64

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r  ; 1.54

main PROC
    sub     rsp, 40                                ; 00000028H
    movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
    movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
    call    pow
    lea     rcx, OFFSET FLAT:$SG1354
    movaps  xmm1, xmm0
    movd    rdx, xmm1
    call    printf
    xor     eax, eax
    add     rsp, 40                                ; 00000028H
    ret     0
main ENDP
```

Инструкция **MOVSDX** нет в документации от Intel и AMD (98.5 (стр. 903)), там она называется просто **MOVSD**. Таким образом, в процессорах x86 две инструкции с одинаковым именем (о второй: A.6.2 (стр. 918)). Возможно, в Microsoft решили избежать путаницы и переименовали инструкцию в **MOVSDX**. Она просто загружает значение в младшую половину XMM-регистра.

Функция **pow()** берет аргументы из **XMM0** и **XMM1**, и возвращает результат в **XMM0**. Далее он перекладывается в **RDX** для **printf()**. Почему? Может быть, это потому что **printf()** – функция с переменным количеством аргументов?

Листинг 28.6: Оптимизирующий GCC 4.4.6 x64

```
.LC2:
    .string "32.01 ^ 1.54 = %lf\n"
main:
    sub     rsp, 8
    movsd   xmm1, QWORD PTR .LC0[rip]
    movsd   xmm0, QWORD PTR .LC1[rip]
    call    pow
    ; результат сейчас в XMM0
    mov     edi, OFFSET FLAT:.LC2
    mov     eax, 1 ; количество переданных векторных регистров
    call    printf
    xor     eax, eax
    add     rsp, 8
    ret

.LC0:
    .long    171798692
    .long    1073259479
.LC1:
    .long    2920577761
    .long    1077936455
```

GCC работает понятнее. Значение для **printf()** передается в **XMM0**. Кстати, вот тот случай, когда в **EAX** для **printf()** записывается 1 – это значит, что будет передан один аргумент в векторных регистрах, так того требует стандарт [Michael

28.3. Пример с сравнением

```
#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;

    return b;
};

int main()
{
    printf ("%f\n", d_max (1.2, 3.4));
    printf ("%f\n", d_max (5.6, -4));
};
```

28.3.1. x64

Листинг 28.7: Оптимизирующий MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max PROC
    comisd    xmm0, xmm1
    ja       SHORT $LN2@d_max
    movaps    xmm0, xmm1
$LN2@d_max:
    fatret    0
d_max ENDP
```

Оптимизирующий MSVC генерирует очень понятный код.

Инструкция **COMISD** это «Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS». Собственно, это она и делает.

Неоптимизирующий MSVC генерирует более избыточно, но тоже всё понятно:

Листинг 28.8: MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max PROC
    movsdx    QWORD PTR [rsp+16], xmm1
    movsdx    QWORD PTR [rsp+8], xmm0
    movsdx    xmm0, QWORD PTR a$[rsp]
    comisd    xmm0, QWORD PTR b$[rsp]
    jbe       SHORT $LN1@d_max
    movsdx    xmm0, QWORD PTR a$[rsp]
    jmp       SHORT $LN2@d_max
$LN1@d_max:
    movsdx    xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
    fatret    0
d_max ENDP
```

А вот GCC 4.4.6 дошел в оптимизации дальше и применил инструкцию **MAXSD** («Return Maximum Scalar Double-Precision Floating-Point Value»), которая просто выбирает максимальное значение!

²Также доступно здесь: <http://x86-64.org/documentation/abi.pdf>


```
d_max:
    maxsd    xmm0, xmm1
    ret
```

28.3.2. x86

Скомпилируем этот пример в MSVC 2012 с включенной оптимизацией:

Листинг 28.10: Оптимизирующий MSVC 2012 x86

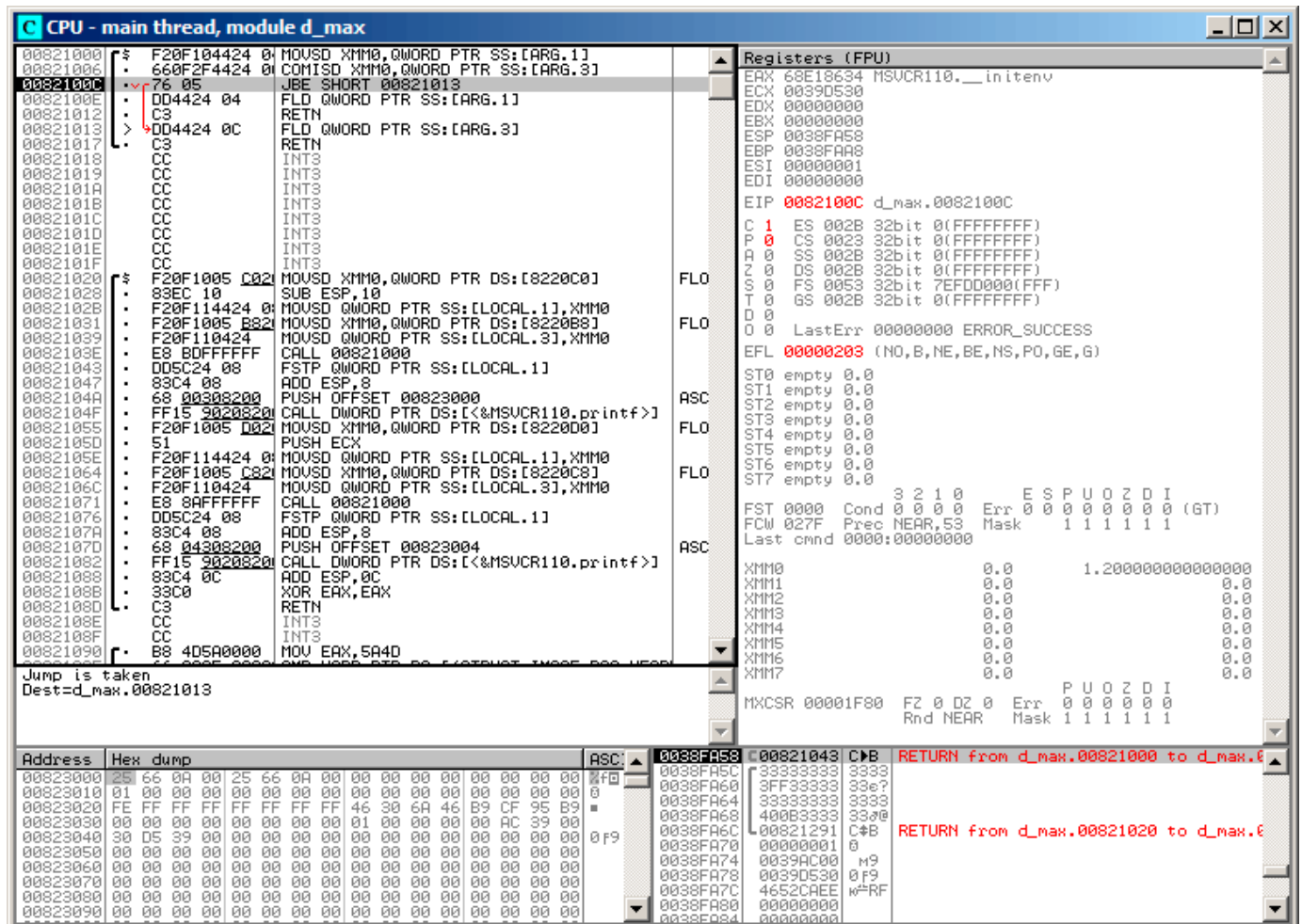
```

_a$ = 8      ; size = 8
_b$ = 16     ; size = 8
_d_max PROC
    movsd    xmm0, QWORD PTR _a$[esp-4]
    comisd   xmm0, QWORD PTR _b$[esp-4]
    jbe      SHORT $LN1@d_max
    fld      QWORD PTR _a$[esp-4]
    ret      0
$LN1@d_max:
    fld      QWORD PTR _b$[esp-4]
    ret      0
_d_max ENDP

```

Всё то же самое, только значения a и b берутся из стека, а результат функции оставляется в `ST(0)`.

Если загрузить этот пример в OllyDbg, увидим, как инструкция `COMISD` сравнивает значения и устанавливает/сбрасывает флаги `CF` и `PF`:

Рис. 28.6: OllyDbg: `COMISD` изменила флаги `CF` и `PF`

28.4. Вычисление машинного эпсилона: x64 и SIMD

Вернемся к примеру «вычисление машинного эпсилона» для *double* листинг.23.2.2.

Теперь скомпилируем его для x64:

```

v$ = 8
calculate_machine_epsilon PROC
    movsdx    QWORD PTR v$[rsp], xmm0
    movaps    xmm1, xmm0
    inc       QWORD PTR v$[rsp]
    movsdx    xmm0, QWORD PTR v$[rsp]
    subsd     xmm0, xmm1
    ret       0
calculate_machine_epsilon ENDP

```

Нет способа прибавить 1 к значению в 128-битном XMM-регистре, так что его нужно в начале поместить в память.

Впрочем, есть инструкция *ADDSD (Add Scalar Double-Precision Floating-Point Values)*, которая может прибавить значение к младшей 64-битной части XMM-регистра игнорируя старшую половину, но наверное MSVC 2012 пока недостаточно хорош для этого ³.

Так или иначе, значение затем перезагружается в XMM-регистр и происходит вычитание. *SUBSD* это «Subtract Scalar Double-Precision Floating-Point Values», т.е. операция производится над младшей 64-битной частью 128-битного XMM-регистра. Результат возвращается в регистре XMM0.

28.5. И снова пример генератора случайных чисел

Вернемся к примеру «пример генератора случайных чисел» листинг.23.1.

Если скомпилировать это в MSVC 2012, компилятор будет использовать SIMD-инструкции для FPU.

Листинг 28.12: Оптимизирующий MSVC 2012

```

__real@3f800000 DD 03f800000r ; 1

tv128 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
    push     ecx
    call     ?my_rand@@YAIKZ
; EAX=псевдослучайное значение
    and     eax, 8388607 ; 007ffffFH
    or      eax, 1065353216 ; 3f800000H
; EAX=псевдослучайное значение & 0x007ffffF | 0x3f800000
; сохранить его в локальном стеке:
    mov     DWORD PTR _tmp$(esp+4), eax
; перезагрузить его как число с плавающей точкой:
    movss   xmm0, DWORD PTR _tmp$(esp+4)
; вычесть 1.0:
    subss   xmm0, DWORD PTR __real@3f800000
; переместить значение в ST0 поместив его во временную переменную...
    movss   DWORD PTR tv128(esp+4), xmm0
; ... и затем перезагрузив её в ST0:
    fld     DWORD PTR tv128(esp+4)
    pop     ecx
    ret     0
?float_rand@@YAMXZ ENDP

```

У всех инструкций суффикс *-SS*, это означает «Scalar Single». «Scalar» означает что только одно значение хранится в регистре. «Single» означает что это тип *float*.

28.6. Итог

Во всех приведенных примерах, в XMM-регистрах используется только младшая половина регистра, там хранится значение в формате IEEE 754.

Собственно, все инструкции с суффиксом **-SD** («Scalar Double-Precision») — это инструкции для работы с числами с плавающей запятой в формате IEEE 754, хранящиеся в младшей 64-битной половине XMM-регистра.

³В качестве упражнения, вы можете попробовать переработать этот код, чтобы избавиться от использования локального стека.

28.6. ИТОГ

Всё удобнее чем это было в FPU, видимо, сказывается тот факт, что расширения SIMD развивались не так хаотично как FPU в прошлом. Стековая модель регистров не используется.

Если вы попытаетесь заменить в этих примерах *double* на *float*, то инструкции будут использоваться те же, только с суффиксом `-SS` («Scalar Single-Precision»), например, `MOVSS`, `COMISS`, `ADDSS`, и т.д.

«Scalar» означает что SIMD-регистр будет хранить только одно значение, вместо нескольких. Инструкции, работающие с несколькими значениями в регистре одновременно, имеют «Packed» в названии.

Нужно также обратить внимание, что SSE2-инструкции работают с 64-битными числами (*double*) в формате IEEE 754, в то время как внутреннее представление в FPU — 80-битные числа. Поэтому ошибок округления (*round-off error*) в FPU может быть меньше чем в SSE2, как следствие, можно сказать, работа с FPU может давать более точные результаты вычислений.

Глава 29

Кое-что специфичное для ARM

29.1. Знак номера (#) перед числом

Компилятор Keil, IDA и objdump предваряет все числа знаком номера («#»), например: листинг.15.1.4. Но когда GCC 4.9 выдает результат на языке ассемблера, он так не делает, например: листинг.40.3.

Так что листинги для ARM в этой книге в каком-то смысле перемешаны.

Трудно сказать, как правильнее. Должно быть, всякий должен придерживаться тех правил, которые приняты в той среде, в которой он работает.

29.2. Режимы адресации

В ARM64 возможна такая инструкция:

```
ldr    x0, [x29,24]
```

И это означает прибавить 24 к значению в X29 и загрузить значение по этому адресу. Обратите внимание что 24 внутри скобок. А если снаружи скобок, то весь смысл меняется:

```
ldr    w4, [x1],28
```

Это означает, загрузить значение по адресу в X1, затем прибавить 28 к X1.

ARM позволяет прибавлять некоторую константу к адресу, с которого происходит загрузка, либо вычитать. Причем, позволяет это делать до загрузки или после.

Такого режима адресации в x86 нет, но он есть в некоторых других процессорах, даже на PDP-11. Существует байка, что режимы пре-инкремента, пост-инкремента, пре-декремента и пост-декремента адреса в PDP-11, были «виновны» в появлении таких конструкций языка Си (который разрабатывался на PDP-11) как `*ptr++`, `++ptr`, `*ptr--`, `--ptr`. Кстати, это является трудно запоминаемой особенностью в Си. Дела обстоят так:

термин в Си	термин в ARM	выражение Си	как это работает
Пост-инкремент	post-indexed addressing	<code>*ptr++</code>	использовать значение <code>*ptr</code> , затем инкремент указателя <code>ptr</code>
Пост-декремент	post-indexed addressing	<code>*ptr--</code>	использовать значение <code>*ptr</code> , затем декремент указателя <code>ptr</code>
Пре-инкремент	pre-indexed addressing	<code>++ptr</code>	инкремент указателя <code>ptr</code> , затем использовать значение <code>*ptr</code>
Пре-декремент	pre-indexed addressing	<code>--ptr</code>	декремент указателя <code>ptr</code> , затем использовать значение <code>*ptr</code>

Pre-indexing маркируется как восклицательный знак в ассемблере ARM. Для примера, смотрите строку 2 в листинг.4.15.

Деннис Ритчи (один из создателей ЯП Си) указывал, что, это, вероятно, придумал Кен Томпсон (еще один создатель Си), потому что подобная возможность процессора имелась еще в PDP-7 http://yurichev.com/mirrors/C/c_dmr_

[postincrement.txt](#), [Dennis M. Ritchie, *The development of the C language*, (1993)]¹. Таким образом, компиляторы с ЯП Си на тот процессор, где это есть, могут использовать это.

Всё это очень удобно для работы с массивами.

29.3. Загрузка констант в регистр

29.3.1. 32-битный ARM

Как мы уже знаем, все инструкции имеют длину в 4 байта в режиме ARM и 2 байта в режиме Thumb. Как в таком случае записать в регистр 32-битное число, если его невозможно закодировать внутри одной инструкции?

Попробуем:

```
unsigned int f()
{
    return 0x12345678;
};
```

Листинг 29.1: GCC 4.6.3 -O3 Режим ARM

```
f:
    ldr    r0, .L2
    bx     lr
.L2:
    .word  305419896 ; 0x12345678
```

Т.е., значение `0x12345678` просто записано в памяти отдельно и загружается, если нужно. Но можно обойтись и без дополнительного обращения к памяти.

Листинг 29.2: GCC 4.6.3 -O3 -march=armv7-a (Режим ARM)

```
movw    r0, #22136      ; 0x5678
movt     r0, #4660       ; 0x1234
bx       lr
```

Видно, что число загружается в регистр по частям, в начале младшая часть (при помощи инструкции MOVW), затем старшая (при помощи MOVT).

Следовательно, нужно 2 инструкции в режиме ARM, чтобы записать 32-битное число в регистр. Это не так уж и страшно, потому что в реальном коде не так уж и много констант (кроме 0 и 1). Значит ли это, что это выполняется медленнее чем одна инструкция, как две инструкции? Вряд ли, наверняка современные процессоры ARM наверняка умеют распознавать такие последовательности и исполнять их быстро.

А IDA легко распознает подобные паттерны в коде и дизассемблирует эту функцию как:

```
MOV     R0, 0x12345678
BX      LR
```

29.3.2. ARM64

```
uint64_t f()
{
    return 0x12345678ABCDEF01;
};
```

Листинг 29.3: GCC 4.9.1 -O3

```
mov     x0, 61185      ; 0xef01
movk     x0, 0xabcd, lsl 16
movk     x0, 0x5678, lsl 32
movk     x0, 0x1234, lsl 48
ret
```

¹Также доступно здесь: <http://go.yurichev.com/17264>

29.4. РЕЛОКИ В ARM64

MOVK означает «MOV Keep», т.е. она записывает 16-битное значение в регистр, не трогая при этом остальные биты. Суффикс **LSL** сдвигает значение в каждом случае влево на 16, 32 и 48 бит. Сдвиг происходит перед загрузкой. Таким образом, нужно 4 инструкции, чтобы записать в регистр 64-битное значение.

Записать числа с плавающей точкой в регистр

Некоторые числа можно записывать в D-регистр при помощи только одной инструкции.

Например:

```
double a()
{
    return 1.5;
};
```

Листинг 29.4: GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a>:
 0: 1e6f1000      fmov    d0, #1.5000000000000000e+000
 4: d65f03c0      ret
```

Число 1.5 действительно было закодировано в 32-битной инструкции. Но как? В ARM64, инструкцию **FMOV** есть 8 бит для кодирования некоторых чисел с плавающей запятой. В [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)]² алгоритм называется **VFPEExpandImm()**. Это также называется *minifloat*³. Мы можем попробовать разные: 30.0 и 31.0 компилятору удастся закодировать, а 32.0 уже нет, для него приходится выделять 8 байт в памяти и записать его там в формате IEEE 754:

```
double a()
{
    return 32;
};
```

Листинг 29.5: GCC 4.9.1 -O3

```
a:
    ldr    d0, .LC0
    ret
.LC0:
    .word  0
    .word  1077936128
```

29.4. Релоки в ARM64

Как известно, в ARM64 инструкции 4-байтные, так что записать длинное число в регистр одной инструкцией нельзя. Тем не менее, файл может быть загружен по произвольному адресу в памяти, для этого релоки и нужны. Больше о них (в связи с Win32 PE): 69.2.6 (стр. 699).

В ARM64 принят следующий метод: адрес формируется при помощи пары инструкций: **ADRP** и **ADD**. Первая загружает в регистр адрес 4KiB-страницы, а вторая прибавляет остаток. Скомпилируем пример из «Hello, world!» (листинг.7) в GCC (Linaro) 4.9 под win32:

Листинг 29.6: GCC (Linaro) 4.9 и objdump объектного файла

```
...>aarch64-linux-gnu-gcc.exe hw.c -c
...>aarch64-linux-gnu-objdump.exe -d hw.o
...
0000000000000000 <main>:
 0: a9bf7bfd      stp     x29, x30, [sp,#-16]!
 4: 910003fd      mov     x29, sp
 8: 90000000      adrp    x0, 0 <main>
```

²Также доступно здесь: [http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_\(Issue_A.a\).pdf](http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf)

³wikipedia

29.4. РЕЛОКИ В ARM64

```

c: 91000000    add    x0, x0, #0x0
10: 94000000    bl     0 <printf>
14: 52800000    mov    w0, #0x0                                // #0
18: a8c17bfd    ldp    x29, x30, [sp],#16
1c: d65f03c0    ret

```

```
...>aarch64-linux-gnu-objdump.exe -r hw.o
```

...

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
000000000000000008	R_AARCH64_ADR_PREL_PG_HI21	.rodata
00000000000000000c	R_AARCH64_ADD_ABS_L012_NC	.rodata
000000000000000010	R_AARCH64_CALL26	printf

Итак, в этом объектом файле три релока.

- Самый первый берет адрес страницы, отсекает младшие 12 бит и записывает оставшиеся старшие 21 в битовые поля инструкции `ADRP`. Это потому что младшие 12 бит кодировать не нужно, и в `ADRP` выделено место только для 21 бит.
- Второй — 12 бит адреса, относительного от начала страницы, в поля инструкции `ADD`.
- Последний, 26-битный, накладывается на инструкцию по адресу `0x10`, где переход на функцию `printf()`. Все адреса инструкций в ARM64 (да и в ARM в режиме ARM) имеют нули в двух младших битах (потому что все инструкции имеют размер в 4 байта), так что нужно кодировать только старшие 26 бит из 28-битного адресного пространства ($\pm 128\text{MB}$).

В слинкованном исполняемом файле релоков в этих местах нет: потому что там уже точно известно, где будет находиться строка «Hello!», и в какой странице, а также известен адрес функции `puts()`. И поэтому там, в инструкциях `ADRP`, `ADD` и `BL`, уже проставлены нужные значения (их проставил линкер во время компоновки):

Листинг 29.7: objdump исполняемого файла

```
0000000000400590 <main>:
400590:    a9bf7bfd        stp     x29, x30, [sp,#-16]!
400594:    910003fd        mov     x29, sp
400598:    90000000        adrp    x0, 400000 <_init-0x3b8>
40059c:    91192000        add     x0, x0, #0x648
4005a0:    97fffffa0       bl      400420 <puts@plt>
4005a4:    52800000        mov     w0, #0x0 // #0
4005a8:    a8c17bfd        ldp     x29, x30, [sp],#16
4005ac:    d65f03c0        ret
```

...

Contents of section .rodata:

```
400640 01000200 00000000 48656c6c 6f210000 .....Hello!..
```

В качестве примера, попробуем дизассемблировать инструкцию BL вручную.

0x97ffffa0 это 1001011111111111111110100000b. В соответствии с [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)C5.6.26], imm26 это последние 26 бит: imm26 = 111111111111111110100000. Это 0x3FFFFA0, но MSB это 1, так что число отрицательное, мы можем вручную его конвертировать в удобный для нас вид. По правилам изменения знака (31 (стр. 447)), просто инвертируем все биты: (1011111=0x5F) и прибавляем 1 (0x5F+1=0x60). Так что число в знаковом виде: -0x60. Умножим -0x60 на 4 (потому что адрес записанный в опкоде разделен на 4): это -0x180. Теперь вычисляем адрес назначения: 0x4005a0 + (-0x180) = 0x400420 (пожалуйста заметьте: мы берем адрес инструкции BL, а не текущее значение PC, которое может быть другим!). Так что адрес в итоге 0x400420.

Больше о релоках связанных с ARM64: *[ELF for the ARM 64-bit Architecture (AArch64), (2013)]*⁴.

⁴Также доступно здесь: <http://go.yurichev.com/17288>

Глава 30

Кое-что специфичное для MIPS

30.1. Загрузка констант в регистр

```
unsigned int f()
{
    return 0x12345678;
};
```

В MIPS, так же как и в ARM, все инструкции имеют размер 32 бита, так что невозможно закодировать 32-битную константу в инструкцию. Так что это транслируется в две инструкции: первая загружает старшую часть 32-битного числа и вторая применяет операцию «ИЛИ», эффект от которой в том, что она просто выставляет младшие 16 бит целевого регистра:

Листинг 30.1: GCC 4.4.5 -O3 (вывод на ассемблере)

```
li    $2,305397760          # 0x12340000
j     $31
ori   $2,$2,0x5678 ; branch delay slot
```

IDA знает о таких часто встречающихся последовательностях, так что для удобства, она показывает последнюю инструкцию ORI как псевдоинструкцию LI, которая якобы загружает полное 32-битное значение в регистр \$V0.

Листинг 30.2: GCC 4.4.5 -O3 (IDA)

```
lui   $v0, 0x1234
jr    $ra
li    $v0, 0x12345678 ; branch delay slot
```

В выводе на ассемблере от GCC есть псевдоинструкция LUI, но на самом деле, там LUI («Load Upper Immediate»), загружающая 16-битное значение в старшую часть регистра.

30.2. Книги и прочие материалы о MIPS

[Dominic Sweetman, ITSee MIPS Run, Second Edition, (2010)].

Часть II

Важные фундаментальные вещи



Глава 31

Представление знака в числах

Методов представления чисел с знаком «плюс» или «минус» несколько¹, но в компьютерах обычно применяется метод «дополнительный код» или «two's complement».

Вот таблица некоторые значений байтов:

двоичное	шестнадцатеричное	беззнаковое	знаковое (дополнительный код)
01111111	0x7f	127	127
01111110	0x7e	126	126
...			
00000110	0x6	6	6
00000101	0x5	5	5
00000100	0x4	4	4
00000011	0x3	3	3
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	255	-1
11111110	0xfe	254	-2
11111101	0xfd	253	-3
11111100	0xfc	252	-4
11111011	0xfb	251	-5
11111010	0xfa	250	-6
...			
10000010	0x82	130	-126
10000001	0x81	129	-127
10000000	0x80	128	-128

Разница в подходе к знаковым/беззнаковым числам, собственно, нужна потому что, например, если представить `0xFFFFFFFF` и `0x00000002` как беззнаковое, то первое число (4294967294) больше второго (2). Если их оба представить как знаковые, то первое будет -2, которое, разумеется, меньше чем второе (2). Вот почему инструкции для условных переходов ([13](#) (стр. [120](#))) представлены в обеих версиях — и для знаковых сравнений (например, `JG`, `JL`) и для беззнаковых (`JA`, `JB`).

Для простоты, вот что нужно знать:

- Числа бывают знаковые и беззнаковые.
- Знаковые типы в Си/Си++:
 - `int64_t` (-9,223,372,036,854,775,808..9,223,372,036,854,775,807) (- 9.2.. 9.2 квинтиллионов) или `0x8000000000000000..0x7FFFFFFFFFFFFFFF`),
 - `int` (-2,147,483,648..2,147,483,647 (- 2.15.. 2.15Gb) или `0x80000000..0x7FFFFFFF`),
 - `char` (-128..127 или `0x80..0x7F`),
 - `ssize_t`.

Беззнаковые:

¹[wikipedia](#)

-
- `uint64_t` (0..18,446,744,073,709,551,615 (18 квинтиллионов) или `0..0xFFFFFFFFFFFFFFFF`),
 - `unsigned int` (0..4,294,967,295 (4.3Gb) или `0..0xFFFFFFFF`),
 - `unsigned char` (0..255 или `0..0xFF`),
 - `size_t`.
- У знаковых чисел знак определяется самым старшим битом: 1 означает «минус», 0 означает «плюс».
 - Преобразование в бóльшие типы данных обходится легко:
[25.5](#) (стр. [401](#)).
 - Изменить знак легко: просто инвертируйте все биты и прибавьте 1. Мы можем заметить, что число другого знака находится на другой стороне на том же расстоянии от нуля. Прибавление единицы необходимо из-за присутствия нуля посередине.
 - Инструкции сложения и вычитания работают одинаково хорошо и для знаковых и для беззнаковых значений. Но для операций умножения и деления, в x86 имеются разные инструкции: `IDIV / IMUL` для знаковых и `DIV / MUL` для беззнаковых.
 - Еще инструкции работающие с знаковыми числами: `CBW/CWD/CWDE/CDQ/CDQE` ([A.6.3](#) (стр. [920](#))), `MOVSX` ([16.1.1](#) (стр. [197](#))), `SAR` ([A.6.3](#) (стр. [924](#))).

Глава 32

Endianness (порядок байт)

Endianness (порядок байт) это способ представления чисел в памяти.

32.1. Big-endian (от старшего к младшему)

Число `0x12345678` представляется в памяти так:

адрес в памяти	значение байта
+0	0x12
+1	0x34
+2	0x56
+3	0x78

CPU с таким порядком включают в себя Motorola 68k, IBM POWER.

32.2. Little-endian (от младшего к старшему)

Число `0x12345678` представляется в памяти так:

адрес в памяти	значение байта
+0	0x78
+1	0x56
+2	0x34
+3	0x12

CPU с таким порядком байт включают в себя Intel x86.

32.3. Пример

Возьмем big-endian Linux для MIPS заинсталлированный в QEMU ¹.

И скомпилируем этот простой пример:

```
#include <stdio.h>

int main()
{
    int v, i;

    v=123;

    printf ("%02X %02X %02X %02X\n",
            *(char*)&v,
            (((char*)&v)+1),
            (((char*)&v)+2),
```

¹Доступен для скачивания здесь: <http://go.yurichev.com/17008>

32.4. BI-ENDIAN (ПЕРЕКЛЮЧАЕМЫЙ ПОРЯДОК)

```
        *(((char*)&v)+3));  
};
```

И запустим его:

```
root@debian-mips:~# ./a.out  
00 00 00 7B
```

Это оно и есть. 0x7B это 123 в десятичном виде. В little-endian-архитектуре, 7B это первый байт (вы можете это проверить в x86 или x86-64), но здесь он последний, потому что старший байт идет первым.

Вот почему имеются разные дистрибутивы Linux для MIPS («mips» (big-endian) и «mipsel» (little-endian)). Программа скомпилированная для одного соглашения об endiannes, не сможет работать в OS использующей другое соглашение.

Еще один пример связанный с big-endian в MIPS в этой книге: [22.4.3](#) (стр. [361](#)).

32.4. Bi-endian (переключаемый порядок)

CPU поддерживающие оба порядка, и его можно переключать, включают в себя ARM, PowerPC, SPARC, MIPS, [IA64²](#), и т.д.

32.5. Конвертирование

Инструкция `BSWAP` может использоваться для конвертирования.

Сетевые пакеты TCP/IP используют соглашение big-endian, вот почему программа, работающая на little-endian архитектуре должна конвертировать значения.

Обычно, используются функции `htonl()` и `htons()`.

Порядок байт big-endian в среде TCP/IP также называется, «network byte order», а порядок байт на компьютере «host byte order». На архитектуре Intel x86, и других little-endian архитектурах, «host byte order» это little-endian, а вот на IBM POWER это может быть big-endian, так что на последней, `htonl()` и `htons()` не меняют порядок байт.

²Intel Architecture 64 (Itanium): [95](#) (стр. [895](#))

Глава 33

Память

Есть три основных типа памяти:

- Глобальная память **АКА** «static memory allocation». Нет нужды явно выделять, выделение происходит просто при объявлении переменных/массивов глобально. Это глобальные переменные расположенные в сегменте данных или констант. Доступны глобально (поэтому считаются **анти-паттерном**). Не удобны для буферов/массивов, потому что должны иметь фиксированный размер. Переполнения буфера, случающиеся здесь, обычно перезаписывают переменные или буферы расположенные рядом в памяти. Пример в этой книге: [8.2](#) (стр. [71](#)).
- Стек **АКА** «allocate on stack», «выделить память в/на стеке». Выделение происходит просто при объявлении переменных/массивов локально в функции. Обычно это локальные для функции переменные. Иногда эти локальные переменные также доступны и для нисходящих функций (**callee**-функциям, если функция-**caller** передает указатель на переменную в функцию-**callee**). Выделение и освобождение очень быстрое, достаточно просто сдвига **SP**. Но также не удобно для буферов/массивов, потому что размер буфера фиксирован, если только не используется `alloca()` ([6.2.4](#) (стр. [29](#))) (или массив с переменной длиной). Переполнение буфера обычно перезаписывает важные структуры стека: [19.2](#) (стр. [271](#)).
- Куча (*heap*) **АКА** «dynamic memory allocation», «выделить память в куче». Выделение происходит при помощи вызова `malloc()/free()` или `new/delete` в Си++. Самый удобный метод: размер блока может быть задан во время исполнения. Изменение размера возможно (при помощи `realloc()`), но может быть медленным. Это самый медленный метод выделения памяти: аллокатор памяти должен поддерживать и обновлять все управляющие структуры во время выделения и освобождения. Переполнение буфера обычно перезаписывает все эти структуры. Выделения в куче также ведут к проблеме утечек памяти: каждый выделенный блок должен быть явно освобожден, но кто-то может забыть об этом, или делать это неправильно. Еще одна проблема — это «использовать после освобождения» — использовать блок памяти после того как `free()` был вызван на нем, это тоже очень опасно. Пример в этой книге: [22.2](#) (стр. [344](#)).

Глава 34

CPU

34.1. Предсказатели переходов

Некоторые современные компиляторы пытаются избавиться от инструкций условных переходов. Примеры в этой книге: [13.1.2](#) (стр. 131), [13.3](#) (стр. 139), [20.5.2](#) (стр. 325).

Это потому что предсказатель переходов далеко не всегда работает идеально, поэтому, компиляторы и стараются реже использовать переходы, если возможно.

Одна из возможностей — это условные инструкции в ARM (как ADRcc), а еще инструкция CMOVcc в x86.

34.2. Зависимости между данными

Современные процессоры способны исполнять инструкции одновременно ([OOE¹](#)), но для этого, внутри такой группы, результат одних не должен влиять на работу других. Следовательно, компилятор старается использовать инструкции с наименьшим влиянием на состояние процессора.

Вот почему инструкция `LEA` в x86 такая популярная — потому что она не модифицирует флаги процессора, а прочие арифметические инструкции — модифицируют.

¹Out-of-order execution

Глава 35

Хеш-функции

Простейший пример это CRC32, алгоритм «более мощный» чем простая контрольная сумма, для проверки целостности данных. Невозможно восстановить оригинальный текст из хеша, там просто меньше информации: ведь текст может быть очень длинным, но результат CRC32 всегда ограничен 32 битами. Но CRC32 не надежна в криптографическом смысле: известны методы как изменить текст таким образом, чтобы получить нужный результат. Криптографические хеш-функции защищены от этого.

Такие функции как MD5, SHA1, и т.д., широко используются для хеширования паролей для хранения их в базе. Действительно: БД форума в интернете может и не хранить пароли (иначе злоумышленник получивший доступ к БД сможет узнать все пароли), а только хеши. К тому же, скрипту интернет-форума вовсе не обязательно знать ваш пароль, он только должен сверить его хеш с тем что лежит в БД, и дать вам доступ если сверка проходит. Один из самых простых способов взлома — это просто перебирать все пароли и ждать пока результат будет такой же как тот что нам нужен. Другие методы намного сложнее.

35.1. Как работает односторонняя функция?

Односторонняя функция, это функция, которая способна превратить из одного значения другое, при этом невозможно (или трудно) проделать обратную операцию. Некоторые люди имеют трудности с пониманием, как это возможно. Рассмотрим очень простой пример.

У нас есть ряд из 10-и чисел в пределах 0..9, каждое встречается один раз, например:

4 6 0 1 3 5 7 8 9 2

Алгоритм простейшей односторонней функции выглядит так:

- возьми число на нулевой позиции (у нас это 4);
- возьми число на первой позиции (у нас это 6);
- обменяй местами числа на позициях 4 и 6.

Отметим числа на позициях 4 и 6:

4 6 0 1 3 5 7 8 9 2
^ ^

Меняем их местами и получаем результат:

4 6 0 1 7 5 3 8 9 2

Глядя на результат, и даже зная алгоритм функции, мы не можем однозначно восстановить изначальное положение чисел. Ведь первые два числа могли быть 0 и/или 1, и тогда именно они могли бы участвовать в обмене.

Это крайне упрощенный пример для демонстрации, настоящие односторонние функции могут быть значительно сложнее.

Часть III

Более сложные примеры

Глава 36

Конвертирование температуры

Еще один крайне популярный пример из книг по программированию для начинающих, это простейшая программа для конвертирования температуры по Фаренгейту в температуру по Цельсию.

$$C = \frac{5 \cdot (F - 32)}{9}$$

Мы также добавим простейшую обработку ошибок: 1) мы должны проверять правильность ввода пользователем; 2) мы должны проверять результат, не ниже ли он -273 по Цельсию (что, как мы можем помнить из школьных уроков физики, ниже абсолютного нуля).

Функция `exit()` заканчивает программу тут же, без возврата в вызывающую функцию.

36.1. Целочисленные значения

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int celsius, fahr;
    printf ("Enter temperature in Fahrenheit:\n");
    if (scanf ("%d", &fahr)!=1)
    {
        printf ("Error while parsing your input\n");
        exit(0);
    };

    celsius = 5 * (fahr-32) / 9;

    if (celsius<=-273)
    {
        printf ("Error: incorrect temperature!\n");
        exit(0);
    };
    printf ("Celsius: %d\n", celsius);
};
```

36.1.1. Оптимизирующий MSVC 2012 x86

Листинг 36.1: Оптимизирующий MSVC 2012 x86

\$SG4228 DB	'Enter temperature in Fahrenheit:', 0aH, 00H
\$SG4230 DB	'%d', 00H
\$SG4231 DB	'Error while parsing your input', 0aH, 00H
\$SG4233 DB	'Error: incorrect temperature!', 0aH, 00H
\$SG4234 DB	'Celsius: %d', 0aH, 00H

36.1. ЦЕЛОЧИСЛЕННЫЕ ЗНАЧЕНИЯ

```

_fahr$ = -4 ; size = 4
_main PROC
    push    ecx
    push    esi
    mov     esi, DWORD PTR __imp__printf
    push    OFFSET $SG4228 ; 'Enter temperature in Fahrenheit:'
    call    esi ; вызвать printf()
    lea     eax, DWORD PTR _fahr$[esp+12]
    push    eax
    push    OFFSET $SG4230 ; '%d'
    call    DWORD PTR __imp__scanf
    add     esp, 12 ; 0000000cH
    cmp     eax, 1
    je      SHORT $LN2@main
    push    OFFSET $SG4231 ; 'Error while parsing your input'
    call    esi ; вызвать printf()
    add     esp, 4
    push    0
    call    DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
    mov     eax, DWORD PTR _fahr$[esp+8]
    add     eax, -32 ; ffffffff0H
    lea     ecx, DWORD PTR [eax+eax*4]
    mov     eax, 954437177 ; 38e38e39H
    imul    ecx
    sar     edx, 1
    mov     eax, edx
    shr     eax, 31 ; 0000001fH
    add     eax, edx
    cmp     eax, -273 ; fffffeeFH
    jge     SHORT $LN10@main
    push    OFFSET $SG4233 ; 'Error: incorrect temperature!'
    call    esi ; вызвать printf()
    add     esp, 4
    push    0
    call    DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
    push    eax
    push    OFFSET $SG4234 ; 'Celsius: %d'
    call    esi ; вызвать printf()
    add     esp, 8
    ; возврат 0 - по стандарту C99
    xor     eax, eax
    pop     esi
    pop     ecx
    ret     0
$LN8@main:
_main ENDP

```

Что мы можем сказать об этом:

- Адрес функции `printf()` в начале загружается в регистр `ESI` так что последующие вызовы `printf()` происходят просто при помощи инструкции `CALL ESI`. Это очень популярная техника компиляторов, может присутствовать, если имеются несколько вызовов одной и той же функции в одном месте, и/или имеется свободный регистр для этого.
- Мы видим инструкцию `ADD EAX, -32` в том месте где от значения должно отняться 32. $EAX = EAX + (-32)$ эквивалентно $EAX = EAX - 32$ и как-то компилятор решил использовать `ADD` вместо `SUB`. Может быть оно того стоит, но сказать трудно.
- Инструкция `LEA` используются там, где нужно умножить значение на 5: `lea ecx, DWORD PTR [eax+eax*4]`. Да, $i+i*4$ эквивалентно $i*5$ и `LEA` работает быстрее чем `IMUL`. Кстати, пара инструкций `SHL EAX, 2 / ADD EAX, EAX` может быть использована здесь вместо `LEA` — некоторые компиляторы так и делают.
- Деление через умножение (42 (стр. 483)) также используется здесь.
- Функция `main()` возвращает 0 хотя `return 0` в конце функции отсутствует. В стандарте C99 [ISO/IEC 9899:TC3

36.2. ЧИСЛА С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

(C99 standard), (2007)5.1.2.2.3] указано что `main()` будет возвращать 0 в случае отсутствия выражения `return`. Это правило работает только для функции `main()`. И хотя, MSVC официально не поддерживает C99, может быть частично и поддерживает?

36.1.2. Оптимизирующий MSVC 2012 x64

Код почти такой же, хотя мы заметим инструкцию `INT 3` после каждого вызова `exit()`.

```
xor     ecx, ecx
call    QWORD PTR __imp_exit
int     3
```

`INT 3` это точка останова для отладчика.

Известно что функция `exit()` из тех, что никогда не возвращают управление ¹, так что если управление все же возвращается, значит происходит что-то крайне странное, и пришло время запускать отладчик.

36.2. Числа с плавающей запятой

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    double celsius, fahr;
    printf ("Enter temperature in Fahrenheit:\n");
    if (scanf ("%lf", &fahr)!=1)
    {
        printf ("Error while parsing your input\n");
        exit(0);
    };

    celsius = 5 * (fahr-32) / 9;

    if (celsius<-273)
    {
        printf ("Error: incorrect temperature!\n");
        exit(0);
    };
    printf ("Celsius: %lf\n", celsius);
};
```

MSVC 2010 x86 использует инструкции [FPU...](#)

Листинг 36.2: Оптимизирующий MSVC 2010 x86

```
$SG4038 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4040 DB      '%lf', 00H
$SG4041 DB      'Error while parsing your input', 0aH, 00H
$SG4043 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4044 DB      'Celsius: %lf', 0aH, 00H

__real@c071100000000000 DQ 0c07110000000000r    ; -273
__real@4022000000000000 DQ 0402200000000000r    ; 9
__real@4014000000000000 DQ 0401400000000000r    ; 5
__real@4040000000000000 DQ 0404000000000000r    ; 32

_fahr$ = -8                                     ; size = 8
_main PROC
    sub     esp, 8
    push    esi
    mov     esi, DWORD PTR __imp__printf
    push    OFFSET $SG4038                      ; 'Enter temperature in Fahrenheit:'
```

¹еще одна популярная из того же ряда это `longjmp()`

36.2. ЧИСЛА С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

```

call     esi                     ; вызвать printf()
lea      eax, DWORD PTR _fahr$[esp+16]
push     eax
push     OFFSET $SG4040          ; '%lf'
call     DWORD PTR __imp__scanf
add      esp, 12                  ; 0000000cH
cmp      eax, 1
je       SHORT $LN2@main
push     OFFSET $SG4041          ; 'Error while parsing your input'
call     esi                     ; вызвать printf()
add      esp, 4
push     0
call     DWORD PTR __imp__exit
$LN2@main:
fld      QWORD PTR _fahr$[esp+12]
fsub     QWORD PTR __real@4040000000000000 ; 32
fmul     QWORD PTR __real@4014000000000000 ; 5
fddiv    QWORD PTR __real@4022000000000000 ; 9
fld      QWORD PTR __real@c071100000000000 ; -273
fcomp    ST(1)
fnstsw   ax
test     ah, 65                  ; 00000041H
jne      SHORT $LN1@main
push     OFFSET $SG4043          ; 'Error: incorrect temperature!'
fstp     ST(0)
call     esi                     ; вызвать printf()
add      esp, 4
push     0
call     DWORD PTR __imp__exit
$LN1@main:
sub      esp, 8
fstp     QWORD PTR [esp]
push     OFFSET $SG4044          ; 'Celsius: %lf'
call     esi
add      esp, 12                  ; 0000000cH
; возврат 0 - по стандарту C99
xor      eax, eax
pop      esi
add      esp, 8
ret      0
$LN10@main:
_main    ENDP

```

...но MSVC от года 2012 использует инструкции **SIMD** вместо этого:

Листинг 36.3: Оптимизирующий MSVC 2010 x86

```

$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%lf', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %lf', 0aH, 00H
__real@c071100000000000 DQ 0c07110000000000r ; -273
__real@4040000000000000 DQ 0404000000000000r ; 32
__real@4022000000000000 DQ 0402200000000000r ; 9
__real@4014000000000000 DQ 0401400000000000r ; 5

_fahr$ = -8 ; size = 8
_main PROC
    sub     esp, 8
    push    esi
    mov     esi, DWORD PTR __imp__printf
    push    OFFSET $SG4228 ; 'Enter temperature in Fahrenheit:'
    call    esi ; вызвать printf()
    lea     eax, DWORD PTR _fahr$[esp+16]
    push    eax
    push    OFFSET $SG4230 ; '%lf'
    call    DWORD PTR __imp__scanf
    add     esp, 12 ; 0000000cH
    cmp     eax, 1
    je     SHORT $LN2@main

```

36.2. ЧИСЛА С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

```
    push    OFFSET $SG4231          ; 'Error while parsing your input'
    call    esi                     ; вызвать printf()
    add     esp, 4
    push    0
    call    DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
    movsd   xmm1, QWORD PTR _fahr$(esp+12]
    subsd   xmm1, QWORD PTR __real@4040000000000000 ; 32
    movsd   xmm0, QWORD PTR __real@c071100000000000 ; -273
    mulsd   xmm1, QWORD PTR __real@4014000000000000 ; 5
    divsd   xmm1, QWORD PTR __real@4022000000000000 ; 9
    comisd   xmm0, xmm1
    jbe     SHORT $LN1@main
    push    OFFSET $SG4233          ; 'Error: incorrect temperature!'
    call    esi                     ; вызвать printf()
    add     esp, 4
    push    0
    call    DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
    sub     esp, 8
    movsd   QWORD PTR [esp], xmm1
    push    OFFSET $SG4234          ; 'Celsius: %lf'
    call    esi                     ; вызвать printf()
    add     esp, 12                 ; 0000000cH
    ; возврат 0 - по стандарту C99
    xor     eax, eax
    pop     esi
    add     esp, 8
    ret     0
$LN8@main:
_main     ENDP
```

Конечно, **SIMD**-инструкции доступны и в x86-режиме, включая те что работают с числами с плавающей запятой. Их использовать в каком-то смысле проще, так что новый компилятор от Microsoft теперь применяет их .

Мы можем также заметить, что значение `-273` загружается в регистр **XMM0** слишком рано. И это нормально, потому что компилятор может генерировать инструкции далеко не в том порядке, в котором они появляются в исходном коде.

Глава 37

Числа Фибоначчи

Еще один часто используемый пример в учебниках по программированию это рекурсивная функция, генерирующая числа Фибоначчи ¹. Последовательность очень простая: каждое следующее число — это сумма двух предыдущих. Первые два числа — это единицы или 0, 1 и 1.

Начало последовательности:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181...

37.1. Пример #1

Реализация проста. Эта программа генерирует последовательность вплоть до 21.

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
    printf ("%d\n", a+b);
    if (a+b > limit)
        return;
    fib (b, a+b, limit);
};

int main()
{
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
};
```

Листинг 37.1: MSVC 2010 x86

```
_a$ = 8          ; size = 4
_b$ = 12         ; size = 4
_limit$ = 16     ; size = 4
_fib PROC
    push        ebp
    mov         ebp, esp
    mov         eax, DWORD PTR _a$[ebp]
    add         eax, DWORD PTR _b$[ebp]
    push        eax
    push        OFFSET $SG2643
    call        DWORD PTR __imp__printf
    add         esp, 8
    mov         ecx, DWORD PTR _a$[ebp]
    add         ecx, DWORD PTR _b$[ebp]
    cmp         ecx, DWORD PTR _limit$[ebp]
    jle         SHORT $LN1@fib
    jmp         SHORT $LN2@fib
$LN1@fib:
    mov         edx, DWORD PTR _limit$[ebp]
```

¹<http://go.yurichev.com/17332>

37.1. ПРИМЕР #1

```
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    call    _fib
    add     esp, 12
$LN2@fib:
    pop     ebp
    ret     0
_fib      ENDP

_main     PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2647 ; "0\n1\n1\n"
    call    DWORD PTR __imp__printf
    add     esp, 4
    push    20
    push    1
    push    1
    call    _fib
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0
_main     ENDP
```

Этим мы проиллюстрируем стековые фреймы.

37.1. ПРИМЕР #1

Загрузим пример в OllyDbg и дотрассируем до самого последнего вызова функции `f()` :

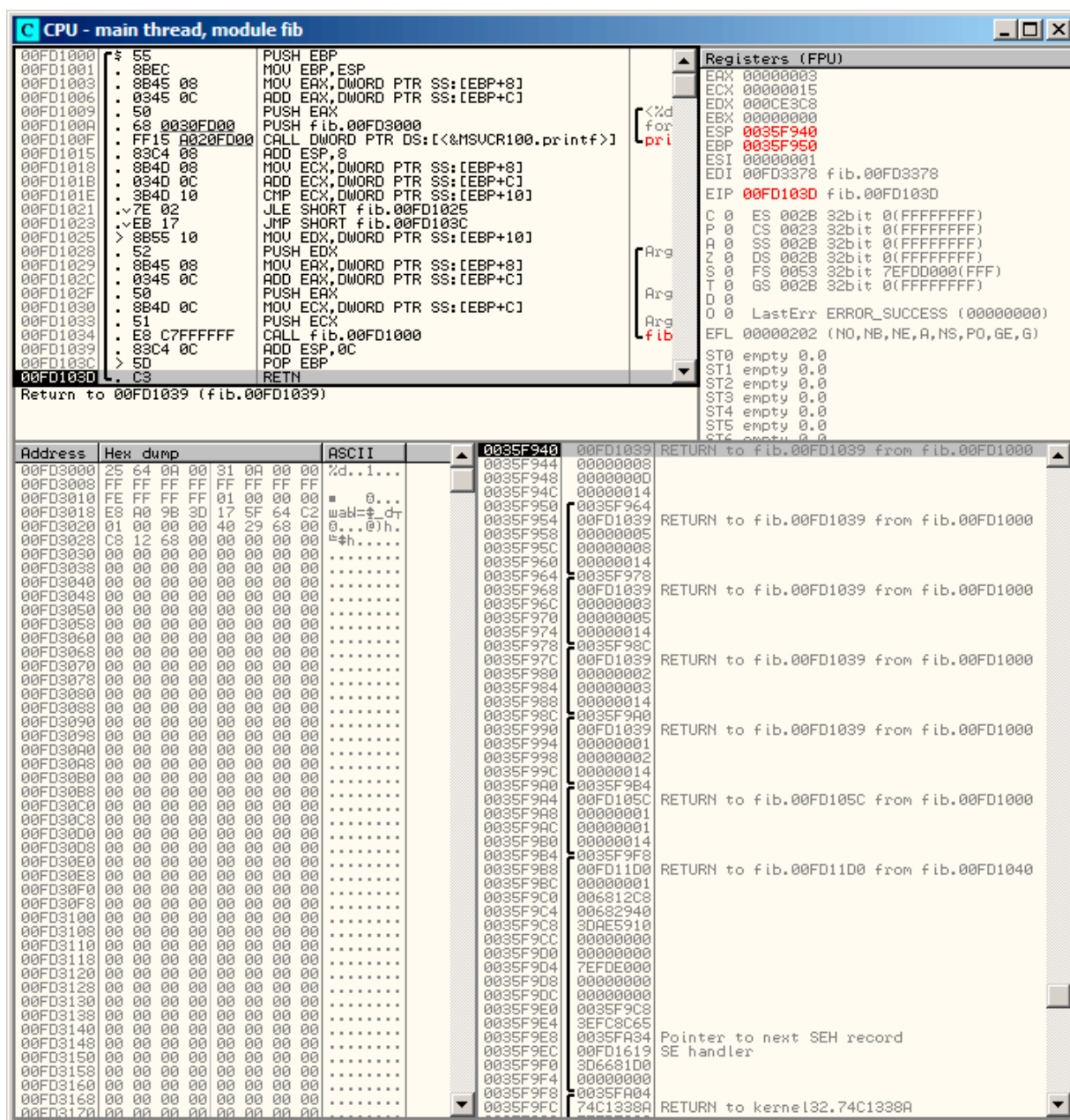


Рис. 37.1: OllyDbg: последний вызов `f()`

37.2. ПРИМЕР #2

Исследуем стек более пристально. Комментарии автора книги ²:

```
0035F940 00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F944 00000008 первый аргумент: a
0035F948 0000000D второй аргумент: b
0035F94C 00000014 третий аргумент: limit
0035F950 /0035F964 сохраненный регистр EBP
0035F954 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F958 |00000005 первый аргумент: a
0035F95C |00000008 второй аргумент: b
0035F960 |00000014 третий аргумент: limit
0035F964 |0035F978 сохраненный регистр EBP
0035F968 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F96C |00000003 первый аргумент: a
0035F970 |00000005 второй аргумент: b
0035F974 |00000014 третий аргумент: limit
0035F978 |0035F98C сохраненный регистр EBP
0035F97C |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F980 |00000002 первый аргумент: a
0035F984 |00000003 второй аргумент: b
0035F988 |00000014 третий аргумент: limit
0035F98C |0035F9A0 сохраненный регистр EBP
0035F990 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F994 |00000001 первый аргумент: a
0035F998 |00000002 второй аргумент: b
0035F99C |00000014 третий аргумент: limit
0035F9A0 |0035F9B4 сохраненный регистр EBP
0035F9A4 |00FD105C RETURN to fib.00FD105C from fib.00FD1000
0035F9A8 |00000001 первый аргумент: a \
0035F9AC |00000001 второй аргумент: b | подготовлено в main() для f1()
0035F9B0 |00000014 третий аргумент: limit /
0035F9B4 |0035F9F8 сохраненный регистр EBP
0035F9B8 |00FD11D0 RETURN to fib.00FD11D0 from fib.00FD1040
0035F9BC |00000001 main() первый аргумент: argc \
0035F9C0 |006812C8 main() второй аргумент: argv | подготовлено в CRT для main()
0035F9C4 |00682940 main() третий аргумент: envp /
```

Функция рекурсивная ³, поэтому стек выглядит как «бутерброд».

Мы видим, что аргумент *limit* всегда один и тот же (0x14 или 20), но аргументы *a* и *b* разные при каждом вызове.

Здесь также адреса *RA* и сохраненные значения *EBP*. OllyDbg способна определять EBP-фреймы, так что она тут нарисовала скобки. Значения внутри каждой скобки это *stack frame*, иными словами, место, которое каждая инкарнация функции может использовать для любых своих нужд. Можно сказать, каждая инкарнация функции не должна обращаться к элементам стека за пределами фрейма (не учитывая аргументов функции), хотя это и возможно технически. Обычно это так и есть, если только функция не содержит каких-то ошибок. Каждое сохраненное значение *EBP* это адрес предыдущего *stack frame*: это причина, почему некоторые отладчики могут легко делить стек на фреймы и выводить аргументы каждой функции.

Как видно, каждая инкарнация функции готовит аргументы для следующего вызова функции.

В самом конце мы видим 3 аргумента функции *main()*. *argc* равен 1 (да, действительно, ведь мы запустили эту программу без аргументов в командной строке).

Очень легко привести к переполнению стека: просто удалите (или закомментируйте) проверку предела и процесс упадет с исключением 0xC00000FD (переполнение стека.)

37.2. Пример #2

В моей функции есть некая избыточность, так что добавим переменную *next* и заменим на нее все «a+b»:

```
#include <stdio.h>

void fib (int a, int b, int limit)
{
    int next=a+b;
```

²Кстати, в OllyDbg можно отметить несколько элементов и скопировать их в клипбордин (Ctrl-C). Это было сделано для этого примера

³т.е. вызывающая сама себя

37.2. ПРИМЕР #2

```

    printf ("%d\n", next);
    if (next > limit)
        return;
    fib (b, next, limit);
};

int main()
{
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
};

```

Это результат работы неоптимизирующего MSVC, поэтому переменная *next* действительно находится в локальном стеке:

Листинг 37.2: MSVC 2010 x86

```

_next$ = -4      ; size = 4
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_limit$ = 16    ; size = 4
_fib PROC
    push     ebp
    mov     ebp, esp
    push     ecx
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    mov     DWORD PTR _next$[ebp], eax
    mov     ecx, DWORD PTR _next$[ebp]
    push     ecx
    push     OFFSET $SG2751 ; '%d'
    call    DWORD PTR __imp__printf
    add     esp, 8
    mov     edx, DWORD PTR _next$[ebp]
    cmp     edx, DWORD PTR _limit$[ebp]
    jle     SHORT $LN1@fib
    jmp     SHORT $LN2@fib
$LN1@fib:
    mov     eax, DWORD PTR _limit$[ebp]
    push     eax
    mov     ecx, DWORD PTR _next$[ebp]
    push     ecx
    mov     edx, DWORD PTR _b$[ebp]
    push     edx
    call    _fib
    add     esp, 12
$LN2@fib:
    mov     esp, ebp
    pop     ebp
    ret     0
_fib ENDP

_main PROC
    push     ebp
    mov     ebp, esp
    push     OFFSET $SG2753 ; "0\n1\n1\n"
    call    DWORD PTR __imp__printf
    add     esp, 4
    push     20
    push     1
    push     1
    call    _fib
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0
_main ENDP

```

37.2. ПРИМЕР #2

Загрузим OllyDbg снова:

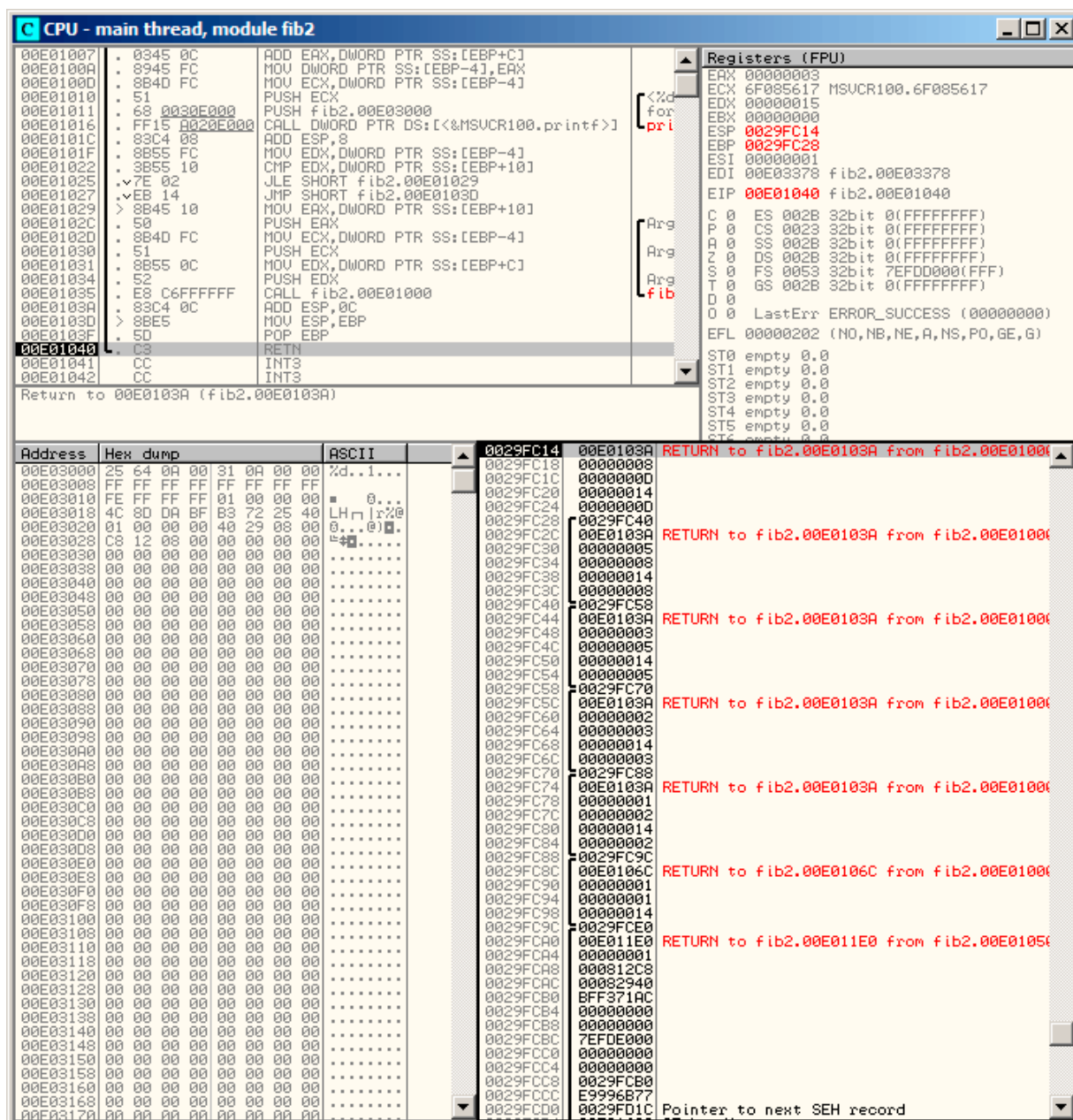


Рис. 37.2: OllyDbg: последний вызов `f()`

Теперь переменная *next* присутствует в каждом фрейме.

37.3. ИТОГ

Рассмотрим стек более пристально. Автор и здесь добавил туда своих комментариев:

0029FC14	00E0103A	RETURN to fib2.00E0103A from fib2.00E01000	
0029FC18	00000008	первый аргумент: a	
0029FC1C	0000000D	второй аргумент: b	
0029FC20	00000014	третий аргумент: limit	
0029FC24	0000000D	переменная "next"	
0029FC28	/0029FC40	сохраненный регистр EBP	
0029FC2C	00E0103A	RETURN to fib2.00E0103A from fib2.00E01000	
0029FC30	00000005	первый аргумент: a	
0029FC34	00000008	второй аргумент: b	
0029FC38	00000014	третий аргумент: limit	
0029FC3C	00000008	переменная "next"	
0029FC40	0029FC58	сохраненный регистр EBP	
0029FC44	00E0103A	RETURN to fib2.00E0103A from fib2.00E01000	
0029FC48	00000003	первый аргумент: a	
0029FC4C	00000005	второй аргумент: b	
0029FC50	00000014	третий аргумент: limit	
0029FC54	00000005	переменная "next"	
0029FC58	0029FC70	сохраненный регистр EBP	
0029FC5C	00E0103A	RETURN to fib2.00E0103A from fib2.00E01000	
0029FC60	00000002	первый аргумент: a	
0029FC64	00000003	второй аргумент: b	
0029FC68	00000014	третий аргумент: limit	
0029FC6C	00000003	переменная "next"	
0029FC70	0029FC88	сохраненный регистр EBP	
0029FC74	00E0103A	RETURN to fib2.00E0103A from fib2.00E01000	
0029FC78	00000001	первый аргумент: a	\
0029FC7C	00000002	второй аргумент: b	подготовлено в f1() для следующего вызова f1()
0029FC80	00000014	третий аргумент: limit	/
0029FC84	00000002	переменная "next"	
0029FC88	0029FC9C	сохраненный регистр EBP	
0029FC8C	00E0106C	RETURN to fib2.00E0106C from fib2.00E01000	
0029FC90	00000001	первый аргумент: a	\
0029FC94	00000001	второй аргумент: b	подготовлено в main() для f1()
0029FC98	00000014	третий аргумент: limit	/
0029FC9C	0029FCE0	сохраненный регистр EBP	
0029FCA0	00E011E0	RETURN to fib2.00E011E0 from fib2.00E01050	
0029FCA4	00000001	main() первый аргумент: argc	\
0029FCA8	000812C8	main() второй аргумент: argv	подготовлено в CRT для main()
0029FCAC	00082940	main() третий аргумент: envp	/

Значение переменной *next* вычисляется в каждой инкарнации функции, затем передается аргумент *b* в следующую инкарнацию.

37.3. Итог

Рекурсивные функции эстетически красивы, но технически могут ухудшать производительность из-за активного использования стека. Тот, кто пишет критические к времени исполнения участки кода, наверное, должен избегать применения там рекурсии.

Например, однажды автор этих строк написал функцию для поиска нужного узла в двоичном дереве. Рекурсивно она выглядела очень красиво, но из-за того, что при каждом вызове тратилось время на эпилог и пролог, все это работало в несколько раз медленнее чем та же функция, но без рекурсии.

Кстати, поэтому некоторые компиляторы функциональных ЯП⁴ (где рекурсия активно применяется) используют **хвостовую рекурсию**.

⁴LISP, Python, Lua, и т.д.

Глава 38

Пример вычисления CRC32

Это распространенный табличный способ вычисления хеша алгоритмом CRC32¹.

```
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stdint.h>
#include <string.h>

typedef unsigned long ub4;
typedef unsigned char ub1;

static const ub4 crctab[256] = {
    0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
    0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
    0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
    0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
    0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
    0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
    0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c,
    0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
    0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
    0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
    0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
    0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
    0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
    0x91646c97, 0xe6635c01, 0xb6b6b51f, 0x41c6c6162, 0x856530d8, 0xf262004e,
    0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
    0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
    0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0xa4adfa54, 0x3dd895d7,
    0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
    0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
    0xbe0b1010, 0xc90c0988, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
    0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
    0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
    0xeada5473, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
    0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
    0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
    0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
    0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
    0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
    0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55,
    0x316e8eef, 0x46699e79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
    0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
    0x2bb45a92, 0x5cb36a04, 0xc2d27ffa, 0xb5d50cf3, 0x2cd99e8b, 0x5bdeae1d,
    0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
    0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
    0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
    0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
    0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69,
    0x616bfff3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
    0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
```

¹Исходник взят тут: <http://go.yurichev.com/17327>


```

0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdeb9ec5, 0x47b2cf7f, 0x30b5ffe9,
0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcd70693,
0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
};

/* how to derive the values in crctab[] from polynomial 0xedb88320 */
void build_table()
{
    ub4 i, j;
    for (i=0; i<256; ++i) {
        j = i;
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        printf("0x%.8lx, ", j);
        if (i%6 == 5) printf("\n");
    }
}

/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
{
    ub4 i;
    const ub1 *k = key;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
    return hash;
}

/* To use, try "gcc -O crc.c -o crc; crc < crc.c" */
int main()
{
    char s[1000];
    while (gets(s)) printf("%.8lx\n", crc(s, strlen(s), 0));
    return 0;
}

```

Нас интересует функция `crc()`. Кстати, обратите внимание на два инициализатора в выражении `for()`: `hash=len, i=0`. Стандарт Си/Си++, конечно, допускает это. А в итоговом коде, вместо одной операции инициализации цикла, будет две.

Компилируем в MSVC с оптимизацией (/Ox). Для краткости, я приведу только функцию `crc()`, с некоторыми комментариями.

```

_key$ = 8          ; size = 4
_len$ = 12         ; size = 4
_hash$ = 16        ; size = 4
_crc PROC
    mov     edx, DWORD PTR _len$[esp-4]
    xor     ecx, ecx ; i будет лежать в регистре ECX
    mov     eax, edx
    test    edx, edx
    jbe     SHORT $LN1@crc
    push    ebx
    push    esi
    mov     esi, DWORD PTR _key$[esp+4] ; ESI = key
    push    edi
$LL3@crc:
; работаем с байтами используя 32-битные регистры. в EDI положим байт с адреса key+i

    movzx   edi, BYTE PTR [ecx+esi]
    mov     ebx, eax ; EBX = (hash = len)
    and     ebx, 255 ; EBX = hash & 0xff

```

```

; XOR EDI, EBX (EDI=EDI^EBX) – эта операция задействует все 32 бита каждого регистра
; но остальные биты (8-31) будут обнулены всегда, так что все ОК
; они обнулены потому что для EDI это было сделано инструкцией MOVZX выше
; а старшие биты EBX были сброшены инструкцией AND EBX, 255 (255 = 0xff)

xor    edi, ebx

; EAX=EAX>>8; образовавшиеся "из ниоткуда" биты в результате (биты 24-31) будут заполнены нулями
shr    eax, 8

; EAX=EAX^crctab[EDI*4] – выбираем элемент из таблицы crctab[] под номером EDI
xor    eax, DWORD PTR _crctab[edi*4]
inc    ecx          ; i++
cmp    ecx, edx      ; i<len ?
jnb    SHORT $LN1@crc ; да
pop    edi
pop    esi
pop    ebx
$LN1@crc:
ret    0
_crc   ENDP

```

Попробуем то же самое в GCC 4.4.1 с опцией `-O3`:

```

crc
public crc
proc near

key      = dword ptr 8
hash     = dword ptr 0Ch

    push    ebp
    xor     edx, edx
    mov     ebp, esp
    push    esi
    mov     esi, [ebp+key]
    push    ebx
    mov     ebx, [ebp+hash]
    test    ebx, ebx
    mov     eax, ebx
    jz      short loc_80484D3
    nop
    lea     esi, [esi+0] ; выравнивание; работает как NOP (ESI не меняется здесь)

loc_80484B8:
    mov     ecx, eax      ; сохранить предыдущее состояние хеша в ECX
    xor     al, [esi+edx]  ; AL=*(key+i)
    add     edx, 1         ; i++
    shr     ecx, 8         ; ECX=hash>>8
    movzx   eax, al        ; EAX=*(key+i)
    mov     eax, dword ptr ds:crctab[eax*4] ; EAX=crctab[EAX]
    xor     eax, ecx        ; hash=EAX^ECX
    cmp     ebx, edx
    ja      short loc_80484B8

loc_80484D3:
    pop     ebx
    pop     esi
    pop     ebp
    retn

crc
endp
\

```

GCC немного выровнял начало тела цикла по 8-байтной границе, для этого добавил `NOP` и `lea esi, [esi+0]` (что тоже *холостая операция*). Подробнее об этом смотрите в разделе о пад ([90](#) (стр. [884](#))).

Глава 39

Пример вычисления адреса сети

Как мы знаем, TCP/IP-адрес (IPv4) состоит из четырех чисел в пределах 0...255, т.е. 4 байта. 4 байта легко помещаются в 32-битную переменную, так что адрес хоста в IPv4, сетевая маска или адрес сети могут быть 32-битными числами.

С точки зрения пользователя, маска сети определяется четырьмя числами в формате вроде 255.255.255.0, но сетевые инженеры (сисадмины) используют более компактную нотацию (CIDR¹), вроде /8, /16, и т.д. Эта нотация просто определяет количество бит в сетевой маске, начиная с MSB.

Маска	Хосты	Свободно	Сетевая маска	В шестнадцатеричном виде	
/30	4	2	255.255.255.252	fffffffc	
/29	8	6	255.255.255.248	ffffff8	
/28	16	14	255.255.255.240	ffffff0	
/27	32	30	255.255.255.224	ffffffe0	
/26	64	62	255.255.255.192	ffffffc0	
/24	256	254	255.255.255.0	fffff00	сеть класса C
/23	512	510	255.255.254.0	ffffe00	
/22	1024	1022	255.255.252.0	ffffc00	
/21	2048	2046	255.255.248.0	ffff800	
/20	4096	4094	255.255.240.0	ffff000	
/19	8192	8190	255.255.224.0	ffffe000	
/18	16384	16382	255.255.192.0	ffffc000	
/17	32768	32766	255.255.128.0	ffff8000	
/16	65536	65534	255.255.0.0	ffff0000	сеть класса B
/8	16777216	16777214	255.0.0.0	ff000000	сеть класса A

Вот простой пример, вычисляющий адрес сети используя сетевую маску и адрес хоста.

```
#include <stdio.h>
#include <stdint.h>

uint32_t form_IP (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4)
{
    return (ip1<<24) | (ip2<<16) | (ip3<<8) | ip4;
};

void print_as_IP (uint32_t a)
{
    printf ("%d.%d.%d.%d\n",
        (a>>24)&0xFF,
        (a>>16)&0xFF,
        (a>>8)&0xFF,
        (a)&0xFF);
};

// bit=31..0
uint32_t set_bit (uint32_t input, int bit)
{
    return input|input|(1<<bit);
};
```

¹Classless Inter-Domain Routing

39.1. CALC_NETWORK_ADDRESS()

```
};

uint32_t form_netmask (uint8_t netmask_bits)
{
    uint32_t netmask=0;
    uint8_t i;

    for (i=0; i<netmask_bits; i++)
        netmask=set_bit(netmask, 31-i);

    return netmask;
};

void calc_network_address (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4, uint8_t netmask_bits)
{
    uint32_t netmask=form_netmask(netmask_bits);
    uint32_t ip=form_IP(ip1, ip2, ip3, ip4);
    uint32_t netw_adr;

    printf ("netmask=");
    print_as_IP (netmask);

    netw_adr=ip&netmask;

    printf ("network address=");
    print_as_IP (netw_adr);
};

int main()
{
    calc_network_address (10, 1, 2, 4, 24);    // 10.1.2.4, /24
    calc_network_address (10, 1, 2, 4, 8);     // 10.1.2.4, /8
    calc_network_address (10, 1, 2, 4, 25);    // 10.1.2.4, /25
    calc_network_address (10, 1, 2, 64, 26);   // 10.1.2.4, /26
};
```

39.1. calc_network_address()

Функция `calc_network_address()` самая простая: она просто умножает (логически, используя **AND**) адрес хоста на сетевую маску, в итоге давая адрес сети.

Листинг 39.1: Оптимизирующий MSVC 2012 /Ob0

```
1  _ip1$ = 8           ; size = 1
2  _ip2$ = 12          ; size = 1
3  _ip3$ = 16          ; size = 1
4  _ip4$ = 20          ; size = 1
5  _netmask_bits$ = 24 ; size = 1
6  _calc_network_address PROC
7      push     edi
8      push     DWORD PTR _netmask_bits$[esp]
9      call     _form_netmask
10     push     OFFSET $SG3045 ; 'netmask='
11     mov      edi, eax
12     call     DWORD PTR __imp__printf
13     push     edi
14     call     _print_as_IP
15     push     OFFSET $SG3046 ; 'network address='
16     call     DWORD PTR __imp__printf
17     push     DWORD PTR _ip4$[esp+16]
18     push     DWORD PTR _ip3$[esp+20]
19     push     DWORD PTR _ip2$[esp+24]
20     push     DWORD PTR _ip1$[esp+28]
21     call     _form_IP
22     and      eax, edi          ; network address = host address & netmask
23     push     eax
24     call     _print_as_IP
```

39.2. FORM_IP()

```
25     add     esp, 36
26     pop     edi
27     ret     0
28 _calc_network_address ENDP
```

На строке 22 мы видим самую важную инструкцию **AND** — так вычисляется адрес сети.

39.2. form_IP()

Функция **form_IP()** просто собирает все 4 байта в одно 32-битное значение.

Вот как это обычно происходит:

- Выделите переменную для возвращаемого значения. Обнулите её.
- Возьмите четвертый (самый младший) байт, сложите его (логически, инструкцией **OR**) с возвращаемым значением. Оно содержит теперь 4-й байт.
- Возьмите третий байт, сдвиньте его на 8 бит влево. Получится значение в виде **0x0000bb00**, где **bb** это третий байт. Сложите итоговое значение (логически, инструкцией **OR**) с возвращаемым значением. Возвращаемое значение пока что содержит **0x000000aa**, так что логическое сложение в итоге выдаст значение вида **0x0000bbaa**.
- Возьмите второй байт, сдвиньте его на 16 бит влево. Вы получите значение вида **0x00cc0000**, где **cc** это второй байт. Сложите (логически) результат и возвращаемое значение. Выходное значение содержит пока что **0x0000bbaa**, так что логическое сложение в итоге выдаст значение вида **0x00ccbbaa**.
- Возьмите первый байт, сдвиньте его на 24 бита влево. Вы получите значение вида **0xdd000000**, где **dd** это первый байт. Сложите (логически) результат и выходное значение. Выходное значение содержит пока что **0x00ccbbaa**, так что сложение выдаст в итоге значение вида **0xddccbbaa**.

И вот как работает неоптимизирующий MSVC 2012:

Листинг 39.2: Неоптимизирующий MSVC 2012

```
; определим ip1 как "dd", ip2 как "cc", ip3 как "bb", ip4 как "aa".
_ip1$ = 8      ; size = 1
_ip2$ = 12     ; size = 1
_ip3$ = 16     ; size = 1
_ip4$ = 20     ; size = 1
_form_IP PROC
    push     ebp
    mov     ebp, esp
    movzx   eax, BYTE PTR _ip1$[ebp]
    ; EAX=000000dd
    shl     eax, 24
    ; EAX=dd000000
    movzx   ecx, BYTE PTR _ip2$[ebp]
    ; ECX=000000cc
    shl     ecx, 16
    ; ECX=00cc0000
    or      eax, ecx
    ; EAX=ddcc0000
    movzx   edx, BYTE PTR _ip3$[ebp]
    ; EDX=000000bb
    shl     edx, 8
    ; EDX=0000bb00
    or      eax, edx
    ; EAX=ddccb000
    movzx   ecx, BYTE PTR _ip4$[ebp]
    ; ECX=000000aa
    or      eax, ecx
    ; EAX=ddccbbaa
    pop     ebp
    ret     0
_form_IP ENDP
```

Хотя, порядок операций другой, но, конечно, порядок роли не играет.

Листинг 39.3: Оптимизирующий MSVC 2012 /Ob0

```
; определим ip1 как "dd", ip2 как "cc", ip3 как "bb", ip4 как "aa".
_ip1$ = 8      ; size = 1
_ip2$ = 12     ; size = 1
_ip3$ = 16     ; size = 1
_ip4$ = 20     ; size = 1
_form_IP PROC
    movzx  eax, BYTE PTR _ip1$[esp-4]
    ; EAX=000000dd
    movzx  ecx, BYTE PTR _ip2$[esp-4]
    ; ECX=000000cc
    shl    eax, 8
    ; EAX=0000dd00
    or     eax, ecx
    ; EAX=0000ddcc
    movzx  ecx, BYTE PTR _ip3$[esp-4]
    ; ECX=000000bb
    shl    eax, 8
    ; EAX=00ddcc00
    or     eax, ecx
    ; EAX=00ddccbb
    movzx  ecx, BYTE PTR _ip4$[esp-4]
    ; ECX=000000aa
    shl    eax, 8
    ; EAX=ddccbb00
    or     eax, ecx
    ; EAX=ddccbbaa
    ret    0
_form_IP ENDP
```

Можно сказать, что каждый байт записывается в младшие 8 бит возвращаемого значения, и затем возвращаемое значение сдвигается на один байт влево на каждом шаге. Повторять 4 раза, для каждого байта.

Вот и всё! К сожалению, наверное, нет способа делать это иначе. Не существует более-менее популярных CPU или ISA, где имеется инструкция для сборки значения из бит или байт. Обычно всё это делает сдвигами бит и логическим сложением (OR).

39.3. print_as_IP()

`print_as_IP()` делает наоборот: расщепляет 32-битное значение на 4 байта.

Расщепление работает немного проще: просто сдвигайте входное значение на 24, 16, 8 или 0 бит, берите биты с нулевого по седьмой (младший байт), вот и всё:

Листинг 39.4: Неоптимизирующий MSVC 2012

```
_a$ = 8      ; size = 4
_print_as_IP PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    ; EAX=ddccbbaa
    and     eax, 255
    ; EAX=000000aa
    push    eax
    mov     ecx, DWORD PTR _a$[ebp]
    ; ECX=ddccbbaa
    shr     ecx, 8
    ; ECX=00ddccbb
    and     ecx, 255
    ; ECX=000000bb
    push    ecx
    mov     edx, DWORD PTR _a$[ebp]
    ; EDX=ddccbbaa
    shr     edx, 16
    ; EDX=0000ddcc
```

39.4. FORM_NETMASK() И SET_BIT()

```
and     edx, 255
; EDX=000000cc
push    edx
mov     eax, DWORD PTR _a$[ebp]
; EAX=ddccbbaa
shr     eax, 24
; EAX=000000dd
and     eax, 255 ; возможно, избыточная инструкция
; EAX=000000dd
push    eax
push    OFFSET $SG2973 ; '%d.%d.%d.%d'
call    DWORD PTR __imp__printf
add     esp, 20
pop     ebp
ret     0
_print_as_IP ENDP
```

Оптимизирующий MSVC 2012 делает почти всё то же самое, только без ненужных перезагрузок входного значения:

Листинг 39.5: Оптимизирующий MSVC 2012 /Ob0

```
_a$ = 8 ; size = 4
_print_as_IP PROC
mov     ecx, DWORD PTR _a$[esp-4]
; ECX=ddccbbaa
movzx   eax, cl
; EAX=000000aa
push    eax
mov     eax, ecx
; EAX=ddccbbaa
shr     eax, 8
; EAX=00ddccbb
and     eax, 255
; EAX=000000bb
push    eax
mov     eax, ecx
; EAX=ddccbbaa
shr     eax, 16
; EAX=0000ddcc
and     eax, 255
; EAX=000000cc
push    eax
; ECX=ddccbbaa
shr     ecx, 24
; ECX=000000dd
push    ecx
push    OFFSET $SG3020 ; '%d.%d.%d.%d'
call    DWORD PTR __imp__printf
add     esp, 20
ret     0
_print_as_IP ENDP
```

39.4. form_netmask() и set_bit()

`form_netmask()` делает сетевую маску из [CIDR](#)-нотации. Конечно, было бы куда эффективнее использовать здесь какую-то уже готовую таблицу, но мы рассматриваем это именно так, сознательно, для демонстрации битовых сдвигов. Мы также сделаем отдельную функцию `set_bit()`. Не очень хорошая идея выделять отдельную функцию для такой примитивной операции, но так будет проще понять, как это всё работает.

Листинг 39.6: Оптимизирующий MSVC 2012 /Ob0

```
_input$ = 8 ; size = 4
_bit$ = 12 ; size = 4
_set_bit PROC
mov     ecx, DWORD PTR _bit$[esp-4]
mov     eax, 1
shl     eax, cl
```

39.5. ИТОГ

```
    or     eax, DWORD PTR _input$[esp-4]
    ret    0
_set_bit ENDP

_netmask_bits$ = 8      ; size = 1
_form_netmask PROC
    push    ebx
    push    esi
    movzx   esi, BYTE PTR _netmask_bits$[esp+4]
    xor     ecx, ecx
    xor     bl, bl
    test    esi, esi
    jle     SHORT $LN9@form_netma
    xor     edx, edx
$LL3@form_netma:
    mov     eax, 31
    sub     eax, edx
    push    eax
    push    ecx
    call    _set_bit
    inc     bl
    movzx   edx, bl
    add     esp, 8
    mov     ecx, eax
    cmp     edx, esi
    jl      SHORT $LL3@form_netma
$LN9@form_netma:
    pop     esi
    mov     eax, ecx
    pop     ebx
    ret     0
_form_netmask ENDP
```

`set_bit()` примитивна: просто сдвигает единицу на нужное количество бит, затем складывает (логически) с входным значением «input». `form_netmask()` имеет цикл: он выставит столько бит (начиная с [MSB](#)), сколько передано в аргументе `netmask_bits`.

39.5. Итог

Вот и всё! Мы запускаем и видим:

```
netmask=255.255.255.0
network address=10.1.2.0
netmask=255.0.0.0
network address=10.0.0.0
netmask=255.255.255.128
network address=10.1.2.0
netmask=255.255.255.192
network address=10.1.2.64
```


Глава 40

Циклы: несколько итераторов

Часто, у цикла только один итератор, но в итоговом коде их может быть несколько.

Вот очень простой пример:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;

    // копировать из одного массива в другой по какой-то странной схеме
    for (i=0; i<cnt; i++)
        a1[i*3]=a2[i*7];
};
```

Здесь два умножения на каждой итерации, а это дорогая операция. Сможем ли мы оптимизировать это как-то? Да, если мы заметим, что индексы обоих массивов перескакивают на места, рассчитать которые мы можем легко и без умножения.

40.1. Три итератора

Листинг 40.1: Оптимизирующий MSVC 2013 x64

```
f      PROC
; RDX=a1
; RCX=a2
; R8=cnt
        test    r8, r8          ; cnt==0? тогда выйти
        je      SHORT $LN1@f
        npad    11
$LL3@f:
        mov     eax, DWORD PTR [rdx]
        lea     rcx, QWORD PTR [rcx+12]
        lea     rdx, QWORD PTR [rdx+28]
        mov     DWORD PTR [rcx-12], eax
        dec     r8
        jne     SHORT $LL3@f
$LN1@f:
        ret     0
f      ENDP
```

Теперь здесь три итератора: переменная *cnt* и два индекса, они увеличиваются на 12 и 28 на каждой итерации, указывая на новые элементы массивов. Мы можем переписать этот код на Си/Си++:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;
    size_t idx1=0; idx2=0;
```

40.2. ДВА ИТЕРАТОРА

```
// копировать из одного массива в другой по какой-то странной схеме
for (i=0; i<cnt; i++)
{
    a1[idx1]=a2[idx2];
    idx1+=3;
    idx2+=7;
};
};
```

Так что, ценой модификации трех итераторов на каждой итерации вместо одного, мы избавлены от двух операций умножения.

40.2. Два итератора

GCC 4.9 сделал еще больше, оставив только 2 итератора:

Листинг 40.2: Оптимизирующий GCC 4.9 x64

```
; RDI=a1
; RSI=a2
; RDX=cnt
f:
    test    rdx, rdx ; cnt==0? тогда выйти
    je      .L1
; вычислить адрес последнего элемента в "a2" и оставить его в RDX
    lea     rax, [0+rdx*4]
; RAX=RDX*4=cnt*4
    sal     rdx, 5
; RDX=RDX<<5=cnt*32
    sub     rdx, rax
; RDX=RDX-RAX=cnt*32-cnt*4=cnt*28
    add     rdx, rsi
; RDX=RDX+RSI=a2+cnt*28
.L3:
    mov     eax, DWORD PTR [rsi]
    add     rsi, 28
    add     rdi, 12
    mov     DWORD PTR [rdi-12], eax
    cmp     rsi, rdx
    jne     .L3
.L1:
    rep ret
```

Здесь больше нет переменной-счетчика: GCC рассудил, что она не нужна. Последний элемент массива *a2* вычисляется перед началом цикла (а это просто: $cnt * 7$), и при помощи этого цикл останавливается: просто исполняйте его пока второй индекс не сравняется с предвычисленным значением.

Об умножении используя сдвиги/сложения/вычитания, читайте здесь: [17.1.3](#) (стр. 208).

Этот код можно переписать на Си/Си++ вот так:

```
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;
    size_t idx1=0; idx2=0;
    size_t last_idx2=cnt*7;

    // копировать из одного массива в другой по какой-то странной схеме
    for (;;)
    {
        a1[idx1]=a2[idx2];
        idx1+=3;
        idx2+=7;
        if (idx2==last_idx2)
            break;
    }
}
```

40.3. СЛУЧАЙ INTEL C++ 2011

```
};  
};
```

GCC (Linaro) 4.9 для ARM64 делает тоже самое, только предвычисляет последний индекс массива *a1* вместо *a2*, а это, конечно, имеет тот же эффект:

Листинг 40.3: Оптимизирующий GCC (Linaro) 4.9 ARM64

```
; X0=a1  
; X1=a2  
; X2=cnt  
f:  
    cbz    x2, .L1          ; cnt==0? тогда выйти  
; вычислить последний элемент массива "a1"  
    add    x2, x2, x2, lsl 1  
; X2=X2+X2<<1=X2+X2*2=X2*3  
    mov    x3, 0  
    lsl    x2, x2, 2  
; X2=X2<<2=X2*4=X2*3*4=X2*12  
.L3:  
    ldr     w4, [x1],28      ; загружать по адресу в X1, прибавить 28 к X1 (пост-инкремент)  
    str     w4, [x0,x3]      ; записать по адресу в X0+X3=a1+X3  
    add     x3, x3, 12        ; сдвинуть X3  
    cmp     x3, x2            ; конец?  
    bne     .L3  
.L1:  
    ret
```

GCC 4.4.5 для MIPS делает то же самое:

Листинг 40.4: Оптимизирующий GCC 4.4.5 для MIPS (IDA)

```
; $a0=a1  
; $a1=a2  
; $a2=cnt  
f:  
; переход на код проверки в цикле:  
    beqz    $a2, locret_24  
; инициализировать счетчик (i) в 0:  
    move    $v0, $zero ; branch delay slot, NOP  
  
loc_8:  
; загрузить 32-битное слово в $a1  
    lw      $a3, 0($a1)  
; инкремент счетчика (i):  
    addiu    $v0, 1  
; проверка на конец (сравнить "i" в $v0 и "cnt" в $a2):  
    sltu     $v1, $v0, $a2  
; сохранить 32-битное слово в $a0:  
    sw       $a3, 0($a0)  
; прибавить 0x1C (28) к \a1 на каждой итерации:  
    addiu    $a1, 0x1C  
; перейти на тело цикла, если i<cnt:  
    bnez     $v1, loc_8  
; прибавить 0xC (12) к \a0 на каждой итерации:  
    addiu    $a0, 0xC ; branch delay slot  
  
locret_24:  
    jr       $ra  
    or       $at, $zero ; branch delay slot, NOP
```

40.3. Случай Intel C++ 2011

Оптимизации компилятора могут быть очень странными, но, тем не менее, корректными. Вот что делает Intel C++ 2011:

Листинг 40.5: Оптимизирующий Intel C++ 2011 x64

```
f      PROC
```

40.3. СЛУЧАЙ INTEL C++ 2011

```

; parameter 1: rcx = a1
; parameter 2: rdx = a2
; parameter 3: r8 = cnt
.B1.1::
    test    r8, r8
    jbe     exit
; Preds .B1.0
;8.14
; Prob 50%
;8.14
; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 ↵
    ↵ xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.2::
    cmp     r8, 6
    jbe     just_copy
; Preds .B1.1
;8.2
; Prob 50%
;8.2
; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 ↵
    ↵ xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.3::
    cmp     rcx, rdx
    jbe     .B1.5
; Preds .B1.2
;9.11
; Prob 50%
;9.11
; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 ↵
    ↵ xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.4::
    mov     r10, r8
    mov     r9, rcx
    shl     r10, 5
    lea     rax, QWORD PTR [r8*4]
    sub     r9, rdx
    sub     r10, rax
    cmp     r9, r10
    jge     just_copy2
; Preds .B1.3
;9.11
;9.11
;9.11
;9.11
;9.11
;9.11
;9.11
; Prob 50%
;9.11
; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 ↵
    ↵ xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.5::
    cmp     rdx, rcx
    jbe     just_copy
; Preds .B1.3 .B1.4
;9.11
; Prob 50%
;9.11
; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 ↵
    ↵ xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.6::
    mov     r9, rdx
    lea     rax, QWORD PTR [r8*8]
    sub     r9, rcx
    lea     r10, QWORD PTR [rax+r8*4]
    cmp     r9, r10
    jl      just_copy
; Preds .B1.5
;9.11
;9.11
;9.11
;9.11
;9.11
; Prob 50%
;9.11
; LOE rdx rcx rbx rbp rsi rdi r8 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 ↵
    ↵ xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
just_copy2::
; Preds .B1.4 .B1.6
; R8 = cnt
; RDX = a2
; RCX = a1
    xor     r10d, r10d
    xor     r9d, r9d
    xor     eax, eax
;8.2
;
;
; LOE rax rdx rcx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 ↵
    ↵ xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.8::
    mov     r11d, DWORD PTR [rax+rdx]
    inc     r10
    mov     DWORD PTR [r9+rcx], r11d
    add     r9, 12
    add     rax, 28
    cmp     r10, r8
    jb      .B1.8
    jmp     exit
; Preds .B1.8 just_copy2
;3.6
;8.2
;3.6
;8.2
;8.2
;8.2
;8.2
; Prob 82%
;8.2
; Prob 100%
;8.2
; LOE rax rdx rcx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 ↵
    ↵ xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
just_copy::
; Preds .B1.2 .B1.5 .B1.6
; R8 = cnt
; RDX = a2
; RCX = a1
    xor     r10d, r10d
    xor     r9d, r9d
    xor     eax, eax
;8.2
;
;

```

```

; LOE rax rdx rcx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 ↗
↘ xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
.B1.11::; Preds .B1.11 just_copy
    mov     r11d, DWORD PTR [rax+rdx]          ;3.6
    inc     r10                                ;8.2
    mov     DWORD PTR [r9+rcx], r11d          ;3.6
    add     r9, 12                             ;8.2
    add     rax, 28                            ;8.2
    cmp     r10, r8                            ;8.2
    jb      .B1.11                            ;8.2
; LOE rax rdx rcx rbx rbp rsi rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 ↗
↘ xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15
exit::; Preds .B1.11 .B1.8 .B1.1
    ret                                         ;10.1

```

В начале, принимаются какие-то решения, затем выполняется одна из процедур. Вероятно, это проверка, не пересекаются ли массивы. Это хорошо известный способ оптимизации процедур копирования блоков в памяти. Но копирующие процедуры одинаковые! Видимо, это ошибка оптимизатора Intel C++, который, тем не менее, генерирует работоспособный код.

Мы намеренно изучаем примеры такого кода в этой книге чтобы читатель мог понимать, что результаты работы компилятором иногда бывают крайне странными, но корректными, потому что когда компилятор тестировали, тесты прошли нормально.

Глава 41

Duff's device

Duff's device ¹ это развернутый цикл с возможностью перехода внутри цикла. Развернутый цикл реализован используя fallthrough-выражение switch().

Мы будем использовать здесь упрощенную версию кода Тома Даффа.

Скажем, нам нужно написать функцию, очищающую регион в памяти. Кто-то может подумать о простом цикле, очищающем байт за байтом. Это, очевидно, медленно, так как все современные компьютеры имеют намного более широкую шину памяти. Так что более правильный способ — это очищать регион в памяти блоками по 4 или 8 байт. Так как мы будем работать с 64-битным примером, мы будем очищать память блоками по 8 байт. Пока всё хорошо. Но что насчет хвоста? Функция очистки памяти будет также вызываться и для блоков с длиной не кратной 8.

Вот алгоритм:

- вычислить количество 8-байтных блоков, очистить их используя 8-байтный (64-битный) доступ к памяти;
- вычислить размер хвоста, очистить его используя 1-байтный доступ к памяти.

Второй шаг можно реализовать, используя простой цикл. Но давайте реализуем его используя развернутый цикл:

```
#include <stdint.h>
#include <stdio.h>

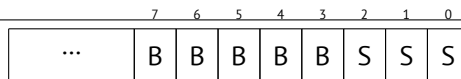
void bzero(uint8_t* dst, size_t count)
{
    int i;

    if (count & (~7))
        // обработать 8-байтные блоки
        for (i=0; i<count>>3; i++)
        {
            *(uint64_t*)dst=0;
            dst=dst+8;
        };

    // обработать хвост
    switch(count & 7)
    {
        case 7: *dst++ = 0;
        case 6: *dst++ = 0;
        case 5: *dst++ = 0;
        case 4: *dst++ = 0;
        case 3: *dst++ = 0;
        case 2: *dst++ = 0;
        case 1: *dst++ = 0;
        case 0: // ничего не делать
            break;
    }
}
```

В начале разберемся, как происходят вычисления. Размер региона в памяти приходит в 64-битном значении. И это значение можно разделить на две части:

¹[wikipedia](#)



(«В» это количество 8-байтных блоков и «S» это длина хвоста в байтах).

Если разделить размер входного блока в памяти на 8, то значение просто сдвигается на 3 бита вправо. Но для вычисления остатка, нам нужно просто изолировать младшие 3 бита! Так что количество 8-байтных блоков вычисляется как $count \gg 3$, а остаток как $count \& 7$.

В начале, нужно определить, будем ли мы вообще исполнять 8-байтную процедуру, так что нам нужно узнать, не больше ли $count$ чем 7. Мы делаем это очищая младшие 3 бита и сравнивая результат с нулем, потому что, всё что нам нужно узнать, это ответ на вопрос, содержит ли старшая часть значения $count$ ненулевые биты.

Конечно, это работает потому что 8 это 2^3 , так что деление на числа вида 2^n это легко. Это невозможно с другими числами.

А на самом деле, трудно сказать, стоит ли пользоваться такими хакерскими трюками, потому что они приводят к коду, который затем тяжело читать. С другой стороны, эти трюки очень популярны и практикующий программист, хотя может и не использовать их, всё же должен их понимать.

Так что первая часть простая: получить количество 8-байтных блоков и записать 64-битные нулевые значения в память.

Вторая часть — это развернутый цикл реализованный как fallthrough-выражение `switch()`. В начале, выразим на обычном русском языке, что мы хотим сделать. Мы должны «записать столько нулевых байт в память, сколько значение $count \& 7$ нам говорит». Если это 0, перейти на конец, больше ничего делать не нужно. Если это 1, перейти на место внутри выражения `switch()`, где произойдет только одна операция записи. Если это 2, перейти на другое место, где две операции записи будут исполнены, и т.д. 7 во входном значении приведет к тому что исполнятся все 7 операций. 8 здесь нет, потому что регион памяти размером в 8 байт будет обработан первой частью нашей функции.

Так что мы сделали развернутый цикл. Это однозначно работало быстрее обычных циклов на старых компьютерах (и наоборот, на современных процессорах короткие циклы работают быстрее развернутых). Может быть, это всё еще может иметь смысл на современных маломощных дешевых MCU².

Посмотрим, что делает оптимизирующий MSVC 2012:

```
dst$ = 8
count$ = 16
bzero PROC
    test    rdx, -8
    je      SHORT $LN11@bzero
; обработать 8-байтные блоки
    xor     r10d, r10d
    mov     r9, rdx
    shr     r9, 3
    mov     r8d, r10d
    test    r9, r9
    je      SHORT $LN11@bzero
    npad    5
$LL19@bzero:
    inc     r8d
    mov     QWORD PTR [rcx], r10
    add     rcx, 8
    movsxd  rax, r8d
    cmp     rax, r9
    jb      SHORT $LL19@bzero
$LN11@bzero:
; обработать хвост
    and     edx, 7
    dec     rdx
    cmp     rdx, 6
    ja      SHORT $LN9@bzero
    lea     r8, OFFSET FLAT:__ImageBase
    mov     eax, DWORD PTR $LN22@bzero[r8+rdx*4]
    add     rax, r8
    jmp     rax
$LN8@bzero:
    mov     BYTE PTR [rcx], 0
    inc     rcx
$LN7@bzero:
    mov     BYTE PTR [rcx], 0
```

²Microcontroller unit

```

        inc     rcx
$LN6@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN5@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN4@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN3@bzero:
        mov     BYTE PTR [rcx], 0
        inc     rcx
$LN2@bzero:
        mov     BYTE PTR [rcx], 0
$LN9@bzero:
        fatret  0
        npad    1
$LN22@bzero:
        DD      $LN2@bzero
        DD      $LN3@bzero
        DD      $LN4@bzero
        DD      $LN5@bzero
        DD      $LN6@bzero
        DD      $LN7@bzero
        DD      $LN8@bzero
bzero   ENDP

```

Первая часть функции выглядит для нас предсказуемо. Вторая часть — это просто развернутый цикл и переход передает управление на нужную инструкцию внутри него. Между парами инструкций `MOV / INC` никакого другого кода нет, так что исполнение продолжается до самого конца, исполняются столько пар, сколько нужно.

Кстати, мы можем заметить, что пара `MOV / INC` занимает какое-то фиксированное количество байт (3+3). Так что пара занимает 6 байт. Зная это, мы можем избавиться от таблицы переходов в `switch()`, мы можем просто умножить входное значение на 6 и перейти на `текущий_RIP + входное_значение * 6`. Это будет также быстрее, потому что не нужно будет загружать элемент из таблицы переходов (*jump table*). Может быть, 6 не самая подходящая константа для быстрого умножения, и может быть оно того и не стоит, но вы поняли идею³. Так в прошлом делали с развернутыми циклами олд-скульные демомейкеры.

³В качестве упражнения, вы можете попробовать переработать этот код и избавиться от таблицы переходов. Пару инструкций тоже можно переписать так что они будут занимать 4 байта или 8. 1 байт тоже возможен (используя инструкцию `STOSB`).

Глава 42

Деление на 9

Простая функция:

```
int f(int a)
{
    return a/9;
};
```

42.1. x86

...компилируется вполне предсказуемо:

Листинг 42.1: MSVC

```
_a$ = 8          ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    cdq     ; знаковое расширение EAX до EDX:EAX
    mov     ecx, 9
    idiv    ecx
    pop     ebp
    ret     0
_f ENDP
```

IDIV делит 64-битное число хранящееся в паре регистров **EDX:EAX** на значение в **ECX**. В результате, **EAX** будет содержать **частное**, а **EDX** — остаток от деления. Результат возвращается из функции через **EAX**, так что после операции деления, это значение не перекладывается больше никуда, оно уже там, где надо. Из-за того, что **IDIV** требует пару регистров **EDX:EAX**, то перед этим инструкция **CDQ** расширяет **EAX** до 64-битного значения учитывая знак, так же, как это делает **MOVSX**. Со включенной оптимизацией (**/Ox**) получается:

Листинг 42.2: Оптимизирующий MSVC

```
_a$ = 8          ; size = 4
_f PROC

    mov     ecx, DWORD PTR _a$[esp-4]
    mov     eax, 954437177 ; 38e38e39H
    imul    ecx
    sar     edx, 1
    mov     eax, edx
    shr     eax, 31        ; 0000001fH
    add     eax, edx
    ret     0
_f ENDP
```

Это — деление через умножение. Умножение конечно быстрее работает. Поэтому можно используя этот трюк ¹ создать код эквивалентный тому что мы хотим и работающий быстрее.

¹Читайте подробнее о делении через умножение в [Henry S. Warren, *Hacker's Delight*, (2002)10-3]

42.2. ARM

В оптимизации компиляторов, это также называется «strength reduction».

GCC 4.4.1 даже без включенной оптимизации генерирует примерно такой же код, как и MSVC с оптимизацией:

Листинг 42.3: Неоптимизирующий GCC 4.4.1

```
public f
f      proc near
arg_0  = dword ptr 8

      push    ebp
      mov     ebp, esp
      mov     ecx, [ebp+arg_0]
      mov     edx, 954437177 ; 38E38E39h
      mov     eax, ecx
      imul    edx
      sar     edx, 1
      mov     eax, ecx
      sar     eax, 1Fh
      mov     ecx, edx
      sub     ecx, eax
      mov     eax, ecx
      pop     ebp
      retn
f      endp
```

42.2. ARM

В процессоре ARM, как и во многих других «чистых» (pure) RISC-процессорах нет инструкции деления. Нет также возможности умножения на 32-битную константу одной инструкцией (вспомните что 32-битная константа просто не поместится в 32-битных опкод). При помощи этого любопытного трюка (или *хака*)², можно обойтись только тремя действиями: сложением, вычитанием и битовыми сдвигами (20 (стр. 300)).

Пример деления 32-битного числа на 10 из [Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)3.3 Division by a Constant]. На выходе и частное и остаток.

```
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
SUB    a2, a1, #10                ; keep (x-10) for later
SUB    a1, a1, a1, lsr #2
ADD    a1, a1, a1, lsr #4
ADD    a1, a1, a1, lsr #8
ADD    a1, a1, a1, lsr #16
MOV    a1, a1, lsr #3
ADD    a3, a1, a1, asl #2
SUBS   a2, a2, a3, asl #1          ; calc (x-10) - (x/10)*10
ADDPL  a1, a1, #1                 ; fix-up quotient
ADDMI  a2, a2, #10                ; fix-up remainder
MOV    pc, lr
```

42.2.1. Оптимизирующий Xcode 4.6.3 (LLVM) (Режим ARM)

```
__text:00002C58 39 1E 08 E3 E3 18 43 E3 MOV     R1, 0x38E38E39
__text:00002C60 10 F1 50 E7             SMMUL   R0, R0, R1
__text:00002C64 C0 10 A0 E1             MOV     R1, R0,ASR#1
__text:00002C68 A0 0F 81 E0             ADD     R0, R1, R0,LSR#31
__text:00002C6C 1E FF 2F E1             BX      LR
```

Этот код почти тот же, что сгенерирован MSVC и GCC в режиме оптимизации. Должно быть, LLVM использует тот же алгоритм для поиска констант.

²hack

42.3. MIPS

Наблюдательный читатель может спросить, как `MOV` записала в регистр сразу 32-битное число, ведь это невозможно в режиме ARM. Действительно невозможно, но как мы видим, здесь на инструкцию 8 байт вместо стандартных 4-х, на самом деле, здесь 2 инструкции. Первая инструкция загружает в младшие 16 бит регистра значение `0x8E39`, а вторая инструкция, на самом деле `MOVT`, загружающая в старшие 16 бит регистра значение `0x383E`. IDA легко распознала эту последовательность и для краткости, сократила всё это до одной «псевдо-инструкции».

Инструкция `SMMUL` (*Signed Most Significant Word Multiply*) умножает числа считая их знаковыми (signed) и оставляет в `R0` старшие 32 бита результата, не сохраняя младшие 32 бита.

Инструкция «`MOV R1, R0, ASR#1`» это арифметический сдвиг право на один бит.

«`ADD R0, R1, R0, LSR#31`» это $R0 = R1 + R0 \gg 31$

Дело в том, что в режиме ARM нет отдельных инструкций для битовых сдвигов. Вместо этого, некоторые инструкции (`MOV`, `ADD`, `SUB`, `RSB`)³ могут быть дополнены суффиксом, сдвигать ли второй операнд и если да, то на сколько и как. `ASR` означает *Arithmetic Shift Right*, `LSR` — *Logical Shift Right*.

42.2.2. Оптимизирующий Xcode 4.6.3 (LLVM) (Режим Thumb-2)

<code>MOV</code>	<code>R1, 0x38E38E39</code>
<code>SMMUL.W</code>	<code>R0, R0, R1</code>
<code>ASRS</code>	<code>R1, R0, #1</code>
<code>ADD.W</code>	<code>R0, R1, R0, LSR#31</code>
<code>BX</code>	<code>LR</code>

В режиме Thumb отдельные инструкции для битовых сдвигов есть, и здесь применяется одна из них — `ASRS` (арифметический сдвиг вправо).

42.2.3. Неоптимизирующий Xcode 4.6.3 (LLVM) и Keil 6/2013

Неоптимизирующий LLVM не занимается генерацией подобного кода, а вместо этого просто вставляет вызов библиотечной функции `__divsi3`.

А Keil во всех случаях вставляет вызов функции `__aeabi_idivmod`.

42.3. MIPS

По какой-то причине, оптимизирующий GCC 4.4.5 сгенерировал просто инструкцию деления:

Листинг 42.4: Оптимизирующий GCC 4.4.5 (IDA)

```
f:
    li      $v0, 9
    bnez    $v0, loc_10
    div     $a0, $v0 ; branch delay slot
    break   0x1C00 ; "break 7" в ассемблерном выводе и в objdump

loc_10:
    mflo    $v0
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP
```

И кстати, мы видим новую инструкцию: `BREAK`. Она просто генерирует исключение. В этом случае, исключение генерируется если делитель 0 (потому что в обычной математике нельзя делить на ноль). Но компилятор GCC наверное не очень хорошо оптимизировал код, и не заметил, что `$V0` не бывает нулем. Так что проверка осталась здесь. Так что если `$V0` будет каким-то образом 0, будет исполнена `BREAK`, сигнализирующая в ОС об исключении. В противном случае, исполняется `MFLO`, берущая результат деления из регистра `LO` и копирующая его в `$V0`.

Кстати, как мы уже можем знать, инструкция `MUL` оставляет старшую 32-битную часть результата в регистре `HI` и младшую 32-битную часть в `LO`. `DIV` оставляет результат в регистре `LO` и остаток в `HI`.

Если изменить выражение на «`a % 9`», вместо инструкции `MFLO` будет использована `MFHI`.

³Эти инструкции также называются «data processing instructions»

42.4. Как это работает

Вот как деление может быть заменено на умножение и деление на числа 2^n :

$$result = \frac{input}{divisor} = \frac{input \cdot \frac{2^n}{divisor}}{2^n} = \frac{input \cdot M}{2^n}$$

Где M это *magic*-коэффициент.

Как вычислить M :

$$M = \frac{2^n}{divisor}$$

Так что эти фрагменты кода обычно имеют форму :

$$result = \frac{input \cdot M}{2^n}$$

Деление на 2^n производится обычным битовым сдвигом вправо.

Если $n \geq 32$, то тогда сдвигается старшая часть **произведения** (в **EDX** или **RDX**).

n выбирается так, чтобы улучшить точность результата.

Если делать знаковое деление, знак результата умножения также добавляется к результату.

Посмотрите на разницу:

```
int f3_32_signed(int a)
{
    return a/3;
};

unsigned int f3_32_unsigned(unsigned int a)
{
    return a/3;
};
```

В беззнаковой версии функции, *magic*-коэффициент это **0xAAAAAAAB** и результат умножения делится на 2^{33} .

В знаковой версии функции, *magic*-коэффициент это **0x55555556** и результат умножения делится на 2^{32} . Впрочем здесь нет инструкции деления: результат просто берется из **EDX**.

Знак результата умножения также учитывается: старшие 32 бита результата сдвигаются на 31 (таким образом, оставляя знак в самом младшем бите **EAX**). 1 прибавляется к конечному результату, если знак отрицательный, для коррекции результата.

Листинг 42.5: Оптимизирующий MSVC 2012

```
_f3_32_unsigned PROC
    mov     eax, -1431655765      ; aaaaaaabH
    mul     DWORD PTR _a$[esp-4] ; беззнаковое умножение
; EDX=(input*0xaaaaaab)/2^32
    shr     edx, 1
; EDX=(input*0xaaaaaab)/2^33
    mov     eax, edx
    ret     0
_f3_32_unsigned ENDP

_f3_32_signed PROC
    mov     eax, 1431655766       ; 55555556H
    imul    DWORD PTR _a$[esp-4] ; знаковое умножение
; берем старшую часть произведения
; это всё равно что сдвинуть произведение на 32 бита вправо, либо разделить его на 2^32
    mov     eax, edx             ; EAX=EDX=(input*0x55555556)/2^32
    shr     eax, 31              ; 0000001fH
    add     eax, edx             ; прибавить 1 если знак отрицательный
    ret     0
_f3_32_signed ENDP
```

42.4.1. Больше теории

Это работает, потому что можно заменить деление на умножение вот так:

$$\frac{x}{c} = x \frac{1}{c}$$

$\frac{1}{c}$ называется *обратное число* и может быть предвычислено компилятором.

Но это для вещественных чисел. Что насчет целых чисел? Можно найти *обратное число по модулю* для целого числа в среде модульной арифметики ⁴. Регистры CPU подходят идеально: каждый ограничен 32 или 64-ю битами, так что практически любая арифметическая операция над регистрами это на самом деле операции по модулю 2^{32} или 2^{64} .

Читайте больше об этом в [Henry S. Warren, *Hacker's Delight*, (2002)10-3].

42.5. Определение делителя**42.5.1. Вариант #1**

Часто, код имеет вид:

```

mov    eax, MAGICAL_CONSTANT
imul   входное_значение
sar    edx, SHIFTING_COEFFICIENT ; знаковое деление на 2x при помощи арифметического сдвига
вправо
mov    eax, edx
shr    eax, 31
add    eax, edx

```

Определим 32-битную *magic*-коэффициент через M , коэффициент сдвига через C и делитель через D .

Делитель, который нам нужен это:

$$D = \frac{2^{32+C}}{M}$$

Например:

Листинг 42.6: Оптимизирующий MSVC 2012

```

mov    eax, 2021161081                ; 78787879H
imul   DWORD PTR _a$[esp-4]
sar    edx, 3
mov    eax, edx
shr    eax, 31                        ; 0000001fH
add    eax, edx

```

Это:

$$D = \frac{2^{32+3}}{2021161081}$$

Числа больше чем 32-битные, так что мы можем использовать Wolfram Mathematica для удобства:

Листинг 42.7: Wolfram Mathematica

```

In[1]:=N[2^(32+3)/2021161081]
Out[1]:=17.

```

Так что искомый делитель это 17.

При делении в x64, всё то же самое, только нужно использовать 2^{64} вместо 2^{32} :

```

uint64_t f1234(uint64_t a)
{
    return a/1234;
};

```

⁴Wikipedia

```
f1234 PROC
      mov     rax, 7653754429286296943          ; 6a37991a23aead6fH
      mul     rcx
      shr     rdx, 9
      mov     rax, rdx
      ret     0
f1234 ENDP
```

Листинг 42.9: Wolfram Mathematica

```
In[1]:=N[2^(64+9)/16^^6a37991a23aead6f]
Out[1]:=1234.
```

42.5.2. Вариант #2

Бывает также вариант с пропущенным арифметическим сдвигом, например:

```
mov     eax, 55555556h ; 1431655766
imul    ecx
mov     eax, edx
shr     eax, 1Fh
```

Метод определения делителя упрощается:

$$D = \frac{2^{32}}{M}$$

Для моего примера, это:

$$D = \frac{2^{32}}{1431655766}$$

Снова используем Wolfram Mathematica:

Листинг 42.10: Wolfram Mathematica

```
In[1]:=N[2^32/16^^55555556]
Out[1]:=3.
```

Искомый делитель это 3.

42.6. Упражнение

- <http://challenges.re/27>

Глава 43

Конверсия строки в число (atoi())

Попробуем реализовать стандартную функцию Си `atoi()`.

43.1. Простой пример

Это самый простой способ прочесть число, представленное в кодировке [ASCII](#)¹. Он не защищен от ошибок: символ отличный от цифры приведет к неверному результату.

```
#include <stdio.h>

int my_atoi (char *s)
{
    int rt=0;

    while (*s)
    {
        rt=rt*10 + (*s-'0');
        s++;
    };

    return rt;
};

int main()
{
    printf ("%d\n", my_atoi ("1234"));
    printf ("%d\n", my_atoi ("1234567890"));
};
```

То, что делает алгоритм это просто считывает цифры слева направо. Символ нуля в [ASCII](#) вычитается из каждой цифры. Цифры от «0» до «9» расположены по порядку в таблице [ASCII](#), так что мы даже можем и не знать точного значения символа «0». Всё что нам нужно знать это то что «0» минус «0» — это 0, а «9» минус «0» это 9, и т.д. Вычитание «0» от каждого символа в итоге дает число от 0 до 9 включительно. Любой другой символ, конечно, приведет к неверному результату! Каждая цифра добавляется к итоговому результату (в переменной «rt»), но итоговый результат также умножается на 10 на каждой цифре. Другими словами, на каждой итерации, результат сдвигается влево на одну позицию в десятичном виде. Самая последняя цифра прибавляется, но не сдвигается.

43.1.1. Оптимизирующий MSVC 2013 x64

Листинг 43.1: Оптимизирующий MSVC 2013 x64

```
s$ = 8
my_atoi PROC
; загрузить первый символ
    movzx    r8d, BYTE PTR [rcx]
; EAX выделен для переменной "rt"
; в начале там 0
```

¹American Standard Code for Information Interchange

43.1. ПРОСТОЙ ПРИМЕР

```
        xor     eax, eax
; первый символ - это нулевой байт, т.е. конец строки?
; тогда выходим.
        test    r8b, r8b
        je      SHORT $LN9@my_atoi
$LL2@my_atoi:
        lea     edx, DWORD PTR [rax+rax*4]
; EDX=RAX*RAX*4=rt+rt*4=rt*5
        movsx   eax, r8b
; EAX=входной символ
; загрузить следующий символ в R8D
        movzx   r8d, BYTE PTR [rcx+1]
; передвинуть указатель в RCX на следующий символ:
        lea     rcx, QWORD PTR [rcx+1]
        lea     eax, DWORD PTR [rax+rdx*2]
; EAX=RAX+RDX*2=входной символ + rt*5*2=входной символ + rt*10
; скорректировать цифру вычитая 48 (0x30 или '0')
        add     eax, -48                                ; ffffffff00000030H
; последний символ был нулем?
        test    r8b, r8b
; перейти на начало цикла, если нет
        jne     SHORT $LL2@my_atoi
$LN9@my_atoi:
        ret     0
my_atoi ENDP
```

Символы загружаются в двух местах: первый символ и все последующие символы. Это сделано для перегруппировки цикла. Здесь нет инструкции для умножения на 10, вместо этого две LEA делают это же. MSVC иногда использует инструкцию ADD с отрицательной константой вместо SUB. Это тот случай. Честно говоря, трудно сказать, чем это лучше, чем SUB. Но MSVC делает так часто.

43.1.2. Оптимизирующий GCC 4.9.1 x64

Оптимизирующий GCC 4.9.1 более краток, но здесь есть одна лишняя инструкция RET в конце. Одной было бы достаточно.

Листинг 43.2: Оптимизирующий GCC 4.9.1 x64

```
my_atoi:
; загрузить входной символ в EDX
        movsx   edx, BYTE PTR [rdi]
; EAX выделен для переменной "rt"
        xor     eax, eax
; выйти, если загруженный символ - это нулевой байт
        test    dl, dl
        je      .L4
.L3:
        lea     eax, [rax+rax*4]
; EAX=RAX*5=rt*5
; передвинуть указатель на следующий символ:
        add     rdi, 1
        lea     eax, [rdx-48+rax*2]
; EAX=входной символ - 48 + RAX*2 = входной символ - '0' + rt*10
; загрузить следующий символ:
        movsx   edx, BYTE PTR [rdi]
; перейти на начало цикла, если загруженный символ - это не нулевой байт
        test    dl, dl
        jne     .L3
        rep ret
.L4:
        rep ret
```

43.1.3. Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 43.3: Оптимизирующий Keil 6/2013 (Режим ARM)

43.1. ПРОСТОЙ ПРИМЕР

```
my_atoi PROC
; R1 будет содержать указатель на символ
    MOV     r1,r0
; R0 будет содержать переменную "rt"
    MOV     r0,#0
    B       |L0.28|
|L0.12|
    ADD     r0,r0,r0,LSL #2
; R0=R0+R0<<2=rt*5
    ADD     r0,r2,r0,LSL #1
; R0=входной символ + rt*5<<1 = входной символ + rt*10
; скорректировать, вычитая '0' из rt:
    SUB     r0,r0,#0x30
; сдвинуть указатель на следующий символ:
    ADD     r1,r1,#1
|L0.28|
; загрузить входной символ в R2
    LDRB    r2,[r1,#0]
; это нулевой байт? если нет, перейти на начало цикла.
    CMP     r2,#0
    BNE     |L0.12|
; выйти, если это нулевой байт.
; переменная "rt" всё еще в регистре R0, готовая для использования в вызывающей ф-ции
    BX      lr
    ENDP
```

43.1.4. Оптимизирующий Keil 6/2013 (Режим Thumb)

Листинг 43.4: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
my_atoi PROC
; R1 будет указателем на входной символ
    MOVS    r1,r0
; R0 выделен для переменной "rt"
    MOVS    r0,#0
    B       |L0.16|
|L0.6|
    MOVS    r3,#0xa
; R3=10
    MULS    r0,r3,r0
; R0=R3*R0=rt*10
; передвинуть указатель на следующий символ:
    ADDS    r1,r1,#1
; скорректировать, вычитая символ нуля:
    SUBS    r0,r0,#0x30
    ADDS    r0,r2,r0
; rt=R2+R0=входной символ + (rt*10 - '0')
|L0.16|
; загрузить входной символ в R2
    LDRB    r2,[r1,#0]
; это ноль?
    CMP     r2,#0
; перейти на тело цикла, если нет
    BNE     |L0.6|
; переменная rt сейчас в R0, готовая для использования в вызывающей ф-ции
    BX      lr
    ENDP
```

Интересно, из школьного курса математики мы можем помнить что порядок операций сложения и вычитания не играет роли. Это наш случай: в начале вычисляется выражение $rt * 10 - '0'$, затем к нему прибавляется значение входного символа. Действительно, результат тот же, но компилятор немного всё перегруппировал.

43.1.5. Оптимизирующий GCC 4.9.1 ARM64

Компилятор для ARM64 может использовать суффикс инструкции, задающий пре-инкремент:

```

my_atoi:
; загрузить входной символ в W1
    ldrb    w1, [x0]
    mov     x2, x0
; X2=адрес входной строки
; загруженный символ - 0?
; перейти на выход, если это так
; W1 будет содержать 0 в этом случае.
; он будет перезагружен в W0 на L4.
    cbz     w1, .L4
; W0 будет содержать переменную "rt"
; инициализировать её нулем:
    mov     w0, 0
.L3:
; вычесть 48 или '0' из входной переменной и оставить результат в W3:
    sub     w3, w1, #48
; загрузить следующий символ по адресу X2+1 в W1 с пре-инкрементом:
    ldrb     w1, [x2,1]!
    add     w0, w0, w0, lsl 2
; W0=W0+W0<<2=W0+W0*4=rt*5
    add     w0, w3, w0, lsl 1
; W0=входная цифра + W0<<1 = входная цифра + rt*5*2 = входная цифра + rt*10
; если только что загруженный символ - это нулевой байт, перейти на начало цикла
    cbnz    w1, .L3
; значение для возврата (rt) в W0, готовое для использования в вызывающей ф-ции
    ret
.L4:
    mov     w0, w1
    ret

```

43.2. Немного расширенный пример

Новый пример более расширенный, теперь здесь есть проверка знака «минус» в самом начале, и еще он может сообщать об ошибке если не-цифра была найдена во входной строке:

```

#include <stdio.h>

int my_atoi (char *s)
{
    int negative=0;
    int rt=0;

    if (*s=='-')
    {
        negative=1;
        s++;
    };

    while (*s)
    {
        if (*s<'0' || *s>'9')
        {
            printf ("Error! Unexpected char: '%c'\n", *s);
            exit(0);
        };
        rt=rt*10 + (*s-'0');
        s++;
    };

    if (negative)
        return -rt;
    return rt;
};

int main()
{

```

43.2. НЕМНОГО РАСШИРЕННЫЙ ПРИМЕР

```
printf ("%d\n", my_atoi ("1234"));
printf ("%d\n", my_atoi ("1234567890"));
printf ("%d\n", my_atoi ("-1234"));
printf ("%d\n", my_atoi ("-1234567890"));
printf ("%d\n", my_atoi ("-a1234567890")); // error
};
```

43.2.1. Оптимизирующий GCC 4.9.1 x64

Листинг 43.6: Оптимизирующий GCC 4.9.1 x64

```
.LC0:
    .string "Error! Unexpected char: '%c'\n"

my_atoi:
    sub    rsp, 8
    movsx  edx, BYTE PTR [rdi]
; проверка на знак минуса
    cmp    dl, 45 ; '-'
    je     .L22
    xor     esi, esi
    test   dl, dl
    je     .L20
.L10:
; ESI=0 здесь, если знака минуса не было, или 1 в противном случае
    lea     eax, [rdx-48]
; любой символ, отличающийся от цифры в результате даст беззнаковое число больше 9 после вычитания
; так что если это не число, перейти на L4, где будет просигнализировано об ошибке:
    cmp    al, 9
    ja     .L4
    xor     eax, eax
    jmp     .L6
.L7:
    lea     ecx, [rdx-48]
    cmp    cl, 9
    ja     .L4
.L6:
    lea     eax, [rax+rax*4]
    add     rdi, 1
    lea     eax, [rdx-48+rax*2]
    movsx  edx, BYTE PTR [rdi]
    test   dl, dl
    jne     .L7
; если знака минуса не было, пропустить инструкцию NEG
; а если был, то исполнить её.
    test   esi, esi
    je     .L18
    neg     eax
.L18:
    add     rsp, 8
    ret
.L22:
    movsx  edx, BYTE PTR [rdi+1]
    lea     rax, [rdi+1]
    test   dl, dl
    je     .L20
    mov     rdi, rax
    mov     esi, 1
    jmp     .L10
.L20:
    xor     eax, eax
    jmp     .L18
.L4:
; сообщить об ошибке. символ в EDX
    mov     edi, 1
    mov     esi, OFFSET FLAT:.LC0 ; "Error! Unexpected char: '%c'\n"
    xor     eax, eax
    call    __printf_chk
```

43.2. НЕМНОГО РАСШИРЕННЫЙ ПРИМЕР

```
xor    edi, edi
call   exit
```

Если знак «минус» был найден в начале строки, инструкция NEG будет исполнена в конце. Она просто меняет знак числа.

Еще кое-что надо отметить. Как среднестатистический программист будет проверять, является ли символ цифрой? Так же, как и у нас в исходном коде:

```
if (*s<'0' || *s>'9')
    ...
```

Здесь две операции сравнения. Но что интересно, так это то что мы можем заменить обе операции на одну: просто вычитайте «0» из значения символа, считается результат за беззнаковое значение (это важно) и проверьте, не больше ли он чем 9.

Например, скажем, строка на входе имеет символ точки («.»), которая имеет код 46 в таблице [ASCII](#). $46 - 48 = -2$ если считать результат за знаковое число. Действительно, символ точки расположен на два места раньше, чем символ «0» в таблице [ASCII](#). Но это 0xFFFFFFE (4294967294) если считать результат за беззнаковое значение, и это точно больше чем 9!

Компиляторы часто так делают, важно распознавать эти трюки.

Еще один пример подобного в этой книге: [49.1.2](#) (стр. [525](#)).

Оптимизирующий MSVC 2013 x64 применяет те же трюки.

43.2.2. Оптимизирующий Keil 6/2013 (Режим ARM)

Листинг 43.7: Оптимизирующий Keil 6/2013 (Режим ARM)

```
1 my_atoi PROC
2     PUSH    {r4-r6,lr}
3     MOV     r4,r0
4     LDRB    r0,[r0,#0]
5     MOV     r6,#0
6     MOV     r5,r6
7     CMP     r0,#0x2d '-'
8 ; R6 будет содержать 1 если минус был встречен, или 0 в противном случае
9     MOVEQ   r6,#1
10    ADDEQ   r4,r4,#1
11    B       |L0.80|
12 |L0.36|
13    SUB     r0,r1,#0x30
14    CMP     r0,#0xa
15    BCC     |L0.64|
16    ADR     r0,|L0.220|
17    BL      __2printf
18    MOV     r0,#0
19    BL      exit
20 |L0.64|
21    LDRB    r0,[r4],#1
22    ADD     r1,r5,r5,LSL #2
23    ADD     r0,r0,r1,LSL #1
24    SUB     r5,r0,#0x30
25 |L0.80|
26    LDRB    r1,[r4,#0]
27    CMP     r1,#0
28    BNE     |L0.36|
29    CMP     r6,#0
30 ; поменять знак в переменной результата
31    RSBNE   r0,r5,#0
32    MOVEQ   r0,r5
33    POP     {r4-r6,pc}
34    ENDP
35
36 |L0.220|
37    DCB     "Error! Unexpected char: '%c'\n",0
```

43.3. УПРАЖНЕНИЕ

В 32-битном ARM нет инструкции NEG, так что вместо этого используется операция «Reverse Subtraction» (строка 31). Она сработает если результат инструкции CMP (на строке 29) был «Not Equal» (не равно, отсюда суффикс -NE suffix). Что делает RSBNE это просто вычитает результирующее значение из нуля. Она работает, как и обычное вычитание, но меняет местами операнды. Вычитание любого числа из нуля это смена знака : $0 - x = -x$.

Код для режима Thumb почти такой же.

GCC 4.9 для ARM64 может использовать инструкцию NEG, доступную в ARM64.

43.3. Упражнение

Кстати, security research-еры часто имеют дело с непредсказуемым поведением программ во время обработки некорректных данных. Например, во время fuzzing-а. В качестве упражнения, вы можете попробовать ввести символы не относящиеся к числам и посмотреть, что случится. Попробуйте объяснить, что произошло, и почему.

Глава 44

Inline-функции

Inline-код это когда компилятор, вместо того чтобы генерировать инструкцию вызова небольшой функции, просто вставляет её тело прямо в это место.

Листинг 44.1: Простой пример

```
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
    return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
    int celsius=atol(argv[1]);
    printf ("%d\n", celsius_to_fahrenheit (celsius));
};
```

...это компилируется вполне предсказуемо, хотя, если включить оптимизации GCC (`-O3`), мы увидим:

Листинг 44.2: Оптимизирующий GCC 4.8.1

```
_main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 16
    call    __main
    mov     eax, DWORD PTR [ebp+12]
    mov     eax, DWORD PTR [eax+4]
    mov     DWORD PTR [esp], eax
    call    _atol
    mov     edx, 1717986919
    mov     DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d\12\0"
    lea     ecx, [eax+eax*8]
    mov     eax, ecx
    imul    edx
    sar     ecx, 31
    sar     edx
    sub     edx, ecx
    add     edx, 32
    mov     DWORD PTR [esp+4], edx
    call    _printf
    leave
    ret
```

(Здесь деление заменено умножением(42 (стр. 483)).)

Да, наша маленькая функция `celsius_to_fahrenheit()` была помещена прямо перед вызовом `printf()`. Почему? Это может быть быстрее чем исполнять код самой функции плюс затраты на вызов и возврат.

Современные оптимизирующие компиляторы самостоятельно выбирают функции для вставки. Но компилятор можно

дополнительно принудить развернуть некоторую функцию, если маркировать её ключевым словом «inline» в её определении.

44.1. Функции работы со строками и памятью

Другая очень частая оптимизация это вставка кода строковых функций таких как *strcpy()*, *strcmp()*, *strlen()*, *memset()*, *memset()*, *memset()*, и т.д.

Иногда это быстрее, чем вызывать отдельную функцию.

Это очень часто встречающиеся шаблонные вставки, которые желательно распознавать reverse engineer-ам «на глаз».

44.1.1. strcmp()

Листинг 44.3: пример с strcmp()

```
bool is_bool (char *s)
{
    if (strcmp (s, "true")==0)
        return true;
    if (strcmp (s, "false")==0)
        return false;

    assert(0);
};
```

Листинг 44.4: Оптимизирующий GCC 4.8.1

```
.LC0:
.string "true"
.LC1:
.string "false"
is_bool:
.LFB0:
    push    edi
    mov     ecx, 5
    push    esi
    mov     edi, OFFSET FLAT:.LC0
    sub     esp, 20
    mov     esi, DWORD PTR [esp+32]
    repz    cmpsb
    je      .L3
    mov     esi, DWORD PTR [esp+32]
    mov     ecx, 6
    mov     edi, OFFSET FLAT:.LC1
    repz    cmpsb
    seta    cl
    setb    dl
    xor     eax, eax
    cmp     cl, dl
    jne     .L8
    add     esp, 20
    pop     esi
    pop     edi
    ret
.L8:
    mov     DWORD PTR [esp], 0
    call    assert
    add     esp, 20
    pop     esi
    pop     edi
    ret
.L3:
    add     esp, 20
    mov     eax, 1
    pop     esi
    pop     edi
```

ret

Листинг 44.5: Оптимизирующий MSVC 2010

```

$SG3454 DB      'true', 00H
$SG3456 DB      'false', 00H

_s$ = 8          ; size = 4
?is_bool@@YA_NPAD@Z PROC ; is_bool
    push        esi
    mov         esi, DWORD PTR _s$[esp]
    mov         ecx, OFFSET $SG3454 ; 'true'
    mov         eax, esi
    npad        4 ; выровнять следующую метку
$LL6@is_bool:
    mov         dl, BYTE PTR [eax]
    cmp         dl, BYTE PTR [ecx]
    jne         SHORT $LN7@is_bool
    test        dl, dl
    je          SHORT $LN8@is_bool
    mov         dl, BYTE PTR [eax+1]
    cmp         dl, BYTE PTR [ecx+1]
    jne         SHORT $LN7@is_bool
    add         eax, 2
    add         ecx, 2
    test        dl, dl
    jne         SHORT $LL6@is_bool
$LN8@is_bool:
    xor         eax, eax
    jmp         SHORT $LN9@is_bool
$LN7@is_bool:
    sbb         eax, eax
    sbb         eax, -1
$LN9@is_bool:
    test        eax, eax
    jne         SHORT $LN2@is_bool

    mov         al, 1
    pop         esi

    ret         0
$LN2@is_bool:
    mov         ecx, OFFSET $SG3456 ; 'false'
    mov         eax, esi
$LL10@is_bool:
    mov         dl, BYTE PTR [eax]
    cmp         dl, BYTE PTR [ecx]
    jne         SHORT $LN11@is_bool
    test        dl, dl
    je          SHORT $LN12@is_bool
    mov         dl, BYTE PTR [eax+1]
    cmp         dl, BYTE PTR [ecx+1]
    jne         SHORT $LN11@is_bool
    add         eax, 2
    add         ecx, 2
    test        dl, dl
    jne         SHORT $LL10@is_bool
$LN12@is_bool:
    xor         eax, eax
    jmp         SHORT $LN13@is_bool
$LN11@is_bool:
    sbb         eax, eax
    sbb         eax, -1
$LN13@is_bool:
    test        eax, eax
    jne         SHORT $LN1@is_bool

    xor         al, al
    pop         esi

```



```

        ret     0
$LN1@is_bool:

        push    11
        push    OFFSET $SG3458
        push    OFFSET $SG3459
        call    DWORD PTR __imp__wassert
        add     esp, 12
        pop     esi

        ret     0
?is_bool@@YA_NPAD@Z ENDP ; is_bool

```

44.1.2. strlen()

Листинг 44.6: пример с strlen()

```

int strlen_test(char *s1)
{
    return strlen(s1);
};

```

Листинг 44.7: Оптимизирующий MSVC 2010

```

_s1$ = 8 ; size = 4
_strlen_test PROC
    mov     eax, DWORD PTR _s1$[esp-4]
    lea     edx, DWORD PTR [eax+1]
$LL3@strlen_tes:
    mov     cl, BYTE PTR [eax]
    inc     eax
    test    cl, cl
    jne     SHORT $LL3@strlen_tes
    sub     eax, edx
    ret     0
_strlen_test ENDP

```

44.1.3. strcpy()

Листинг 44.8: пример с strcpy()

```

void strcpy_test(char *s1, char *outbuf)
{
    strcpy(outbuf, s1);
};

```

Листинг 44.9: Оптимизирующий MSVC 2010

```

_s1$ = 8      ; size = 4
_outbuf$ = 12 ; size = 4
_strcpy_test PROC
    mov     eax, DWORD PTR _s1$[esp-4]
    mov     edx, DWORD PTR _outbuf$[esp-4]
    sub     edx, eax
    npad    6 ; выровнять следующую метку
$LL3@strcpy_tes:
    mov     cl, BYTE PTR [eax]
    mov     BYTE PTR [edx+eax], cl
    inc     eax
    test    cl, cl
    jne     SHORT $LL3@strcpy_tes
    ret     0
_strcpy_test ENDP

```

44.1.4. memset()**Пример#1**

Листинг 44.10: 32 байта

```
#include <stdio.h>

void f(char *out)
{
    memset(out, 0, 32);
};
```

Многие компиляторы не генерируют вызов `memset()` для коротких блоков, а просто вставляют набор **MOV**-ов:

Листинг 44.11: Оптимизирующий GCC 4.9.1 x64

```
f:
    mov     QWORD PTR [rdi], 0
    mov     QWORD PTR [rdi+8], 0
    mov     QWORD PTR [rdi+16], 0
    mov     QWORD PTR [rdi+24], 0
    ret
```

Кстати, это напоминает развернутые циклы: [15.1.4](#) (стр. 188).

Пример#2

Листинг 44.12: 67 байт

```
#include <stdio.h>

void f(char *out)
{
    memset(out, 0, 67);
};
```

Когда размер блока не кратен 4 или 8, разные компиляторы могут вести себя по-разному.

Например, MSVC 2012 продолжает вставлять **MOV**:

Листинг 44.13: Оптимизирующий MSVC 2012 x64

```
out$ = 8
f      PROC
    xor     eax, eax
    mov     QWORD PTR [rcx], rax
    mov     QWORD PTR [rcx+8], rax
    mov     QWORD PTR [rcx+16], rax
    mov     QWORD PTR [rcx+24], rax
    mov     QWORD PTR [rcx+32], rax
    mov     QWORD PTR [rcx+40], rax
    mov     QWORD PTR [rcx+48], rax
    mov     QWORD PTR [rcx+56], rax
    mov     WORD PTR [rcx+64], ax
    mov     BYTE PTR [rcx+66], al
    ret     0
f      ENDP
```

...а GCC использует **REP STOSQ**, полагая, что так будет короче, чем пачка **MOV** 's:

Листинг 44.14: Оптимизирующий GCC 4.9.1 x64

```
f:
    mov     QWORD PTR [rdi], 0
    mov     QWORD PTR [rdi+59], 0
    mov     rcx, rdi
    lea     rdi, [rdi+8]
    xor     eax, eax
```

44.1. ФУНКЦИИ РАБОТЫ СО СТРОКАМИ И ПАМЯТЬЮ

```
and    rdi, -8
sub    rcx, rdi
add    ecx, 67
shr    ecx, 3
rep stosq
ret
```

44.1.5. memcpy()

Короткие блоки

Если нужно скопировать немного байт, то, нередко, `memcpy()` заменяется на несколько инструкций `MOV`.

Листинг 44.15: пример с memcpy()

```
void memcpy_7(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 7);
};
```

Листинг 44.16: Оптимизирующий MSVC 2010

```
_inbuf$ = 8    ; size = 4
_outbuf$ = 12  ; size = 4
_memcpy_7 PROC
    mov     ecx, DWORD PTR _inbuf$[esp-4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR _outbuf$[esp-4]
    mov     DWORD PTR [eax+10], edx
    mov     dx, WORD PTR [ecx+4]
    mov     WORD PTR [eax+14], dx
    mov     cl, BYTE PTR [ecx+6]
    mov     BYTE PTR [eax+16], cl
    ret     0
_memcpy_7 ENDP
```

Листинг 44.17: Оптимизирующий GCC 4.8.1

```
memcpy_7:
    push    ebx
    mov     eax, DWORD PTR [esp+8]
    mov     ecx, DWORD PTR [esp+12]
    mov     ebx, DWORD PTR [eax]
    lea     edx, [ecx+10]
    mov     DWORD PTR [ecx+10], ebx
    movzx   ecx, WORD PTR [eax+4]
    mov     WORD PTR [edx+4], cx
    movzx   eax, BYTE PTR [eax+6]
    mov     BYTE PTR [edx+6], al
    pop     ebx
    ret
```

Обычно это происходит так: в начале копируются 4-байтные блоки, затем 16-битное слово (если нужно), затем последний байт (если нужно).

Точно так же при помощи `MOV` копируются структуры: [22.4.1](#) (стр. 357).

Длинные блоки

Здесь компиляторы ведут себя по-разному.

Листинг 44.18: пример с memcpy()

```
void memcpy_128(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 128);
};
```

44.1. ФУНКЦИИ РАБОТЫ СО СТРОКАМИ И ПАМЯТЬЮ

```
void memcpy_123(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 123);
};
```

При копировании 128 байт, MSVC может обойтись одной инструкцией **MOVSD** (ведь 128 кратно 4):

Листинг 44.19: Оптимизирующий MSVC 2010

```
_inbuf$ = 8          ; size = 4
_outbuf$ = 12        ; size = 4
_memcpy_128 PROC
    push    esi
    mov     esi, DWORD PTR _inbuf$[esp]
    push    edi
    mov     edi, DWORD PTR _outbuf$[esp+4]
    add     edi, 10
    mov     ecx, 32
    rep movsd
    pop     edi
    pop     esi
    ret     0
_memcpy_128 ENDP
```

При копировании 123-х байт, в начале копируется 30 32-битных слов при помощи **MOVSD** (это 120 байт), затем копируется 2 байта при помощи **MOVSW**, затем еще один байт при помощи **MOVSB**.

Листинг 44.20: Оптимизирующий MSVC 2010

```
_inbuf$ = 8          ; size = 4
_outbuf$ = 12        ; size = 4
_memcpy_123 PROC
    push    esi
    mov     esi, DWORD PTR _inbuf$[esp]
    push    edi
    mov     edi, DWORD PTR _outbuf$[esp+4]
    add     edi, 10
    mov     ecx, 30
    rep movsd
    movsw
    movsb
    pop     edi
    pop     esi
    ret     0
_memcpy_123 ENDP
```

GCC во всех случаях вставляет большую универсальную функцию, работающую для всех размеров блоков:

Листинг 44.21: Оптимизирующий GCC 4.8.1

```
memcpy_123:
.LFB3:
    push    edi
    mov     eax, 123
    push    esi
    mov     edx, DWORD PTR [esp+16]
    mov     esi, DWORD PTR [esp+12]
    lea     edi, [edx+10]
    test    edi, 1
    jne     .L24
    test    edi, 2
    jne     .L25
.L7:
    mov     ecx, eax
    xor     edx, edx
    shr     ecx, 2
    test    al, 2
    rep movsd
    je      .L8
```

44.1. ФУНКЦИИ РАБОТЫ СО СТРОКАМИ И ПАМЯТЬЮ

```
movzx    edx, WORD PTR [esi]
mov      WORD PTR [edi], dx
mov      edx, 2
.L8:
test     al, 1
je       .L5
movzx    eax, BYTE PTR [esi+edx]
mov      BYTE PTR [edi+edx], al
.L5:
pop      esi
pop      edi
ret
.L24:
movzx    eax, BYTE PTR [esi]
lea      edi, [edx+11]
add      esi, 1
test     edi, 2
mov      BYTE PTR [edx+10], al
mov      eax, 122
je       .L7
.L25:
movzx    edx, WORD PTR [esi]
add      edi, 2
add      esi, 2
sub      eax, 2
mov      WORD PTR [edi-2], dx
jmp      .L7
.LFE3:
```

Универсальные функции копирования блоков обычно работают по следующей схеме: вычислить, сколько 32-битных слов можно скопировать, затем сделать это при помощи `MOVSD`, затем скопировать остатки.

Более сложные функции копирования используют [SIMD](#) и учитывают выравнивание в памяти. Как пример функции `strlen()` использующую SIMD : [26.2](#) (стр. [413](#)).

44.1.6. `memcpy()`

Листинг 44.22: пример с `memcpy()`

```
void memcpy_1235(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 1235);
};
```

Для блоков разной длины, MSVC 2010 вставляет одну и ту же универсальную функцию:

Листинг 44.23: Оптимизирующий MSVC 2010

```
_buf1$ = 8      ; size = 4
_buf2$ = 12     ; size = 4
_memcpy_1235 PROC
    mov     edx, DWORD PTR _buf2$[esp-4]
    mov     ecx, DWORD PTR _buf1$[esp-4]
    push    esi
    push    edi
    mov     esi, 1235
    add     edx, 10
$LL4@memcpy_123:
    mov     eax, DWORD PTR [edx]
    cmp     eax, DWORD PTR [ecx]
    jne     SHORT $LN10@memcpy_123
    sub     esi, 4
    add     ecx, 4
    add     edx, 4
    cmp     esi, 4
    jae     SHORT $LL4@memcpy_123
$LN10@memcpy_123:
    movzx   edi, BYTE PTR [ecx]
    movzx   eax, BYTE PTR [edx]
```

44.1. ФУНКЦИИ РАБОТЫ СО СТРОКАМИ И ПАМЯТЬЮ

```
sub    eax, edi
jne    SHORT $LN7@memcmp_123
movzx  eax, BYTE PTR [edx+1]
movzx  edi, BYTE PTR [ecx+1]
sub    eax, edi
jne    SHORT $LN7@memcmp_123
movzx  eax, BYTE PTR [edx+2]
movzx  edi, BYTE PTR [ecx+2]
sub    eax, edi
jne    SHORT $LN7@memcmp_123
cmp    esi, 3
jbe    SHORT $LN6@memcmp_123
movzx  eax, BYTE PTR [edx+3]
movzx  ecx, BYTE PTR [ecx+3]
sub    eax, ecx
$LN7@memcmp_123:
sar    eax, 31
pop    edi
or     eax, 1
pop    esi
ret    0
$LN6@memcmp_123:
pop    edi
xor    eax, eax
pop    esi
ret    0
_memcmp_1235 ENDP
```

44.1.7. strcat()

Это ф-ция `strcat()` в том виде, в котором её сгенерировала MSVC 6.0. Здесь видны 3 части: 1) измерение длины исходной строки (первый `scasb`); 2) измерение длины целевой строки (второй `scasb`); 3) копирование исходной строки в конец целевой (пара `movsd` / `movsb`).

Листинг 44.24: `strcat()`

```
lea    edi, [src]
or     ecx, 0FFFFFFFFh
repne  scasb
not    ecx
sub    edi, ecx
mov    esi, edi
mov    edi, [dst]
mov    edx, ecx
or     ecx, 0FFFFFFFFh
repne  scasb
mov    ecx, edx
dec    edi
shr    ecx, 2
rep  movsd
mov    ecx, edx
and    ecx, 3
rep  movsb
```

44.1.8. Скрипт для IDA

Есть также небольшой скрипт для [IDA](#) для поиска и сворачивания таких очень часто попадающихся inline-функций: [GitHub](#).

Глава 45

C99 restrict

А вот причина, из-за которой программы на FORTRAN, в некоторых случаях, работают быстрее чем на Си.

```
void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    };
};
```

Это очень простой пример, в котором есть одна особенность : указатель на массив `update_me` может быть указателем на массив `sum`, `product`, или даже `sum_product` — ведь нет ничего криминального в том чтобы аргументам функции быть такими, верно?

Компилятор знает об этом, поэтому генерирует код, где в теле цикла будет 4 основных стадии:

- вычислить следующий `sum[i]`
- вычислить следующий `product[i]`
- вычислить следующий `update_me[i]`
- вычислить следующий `sum_product[i]` — на этой стадии придется снова загружать из памяти подсчитанные `sum[i]` и `product[i]`

Возможно ли оптимизировать последнюю стадию? Ведь подсчитанные `sum[i]` и `product[i]` не обязательно снова загружать из памяти, ведь мы их только что подсчитали. Можно, но компилятор не уверен, что на третьей стадии ничего не затерлось! Это называется «pointer aliasing», ситуация, когда компилятор не может быть уверен, что память на которую указывает какой-то указатель, не изменилась.

restrict в стандарте Си C99 [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.3/1] это обещание, данное компилятору программистом, что аргументы функции, отмеченные этим ключевым словом, всегда будут указывать на разные места в памяти и пересекаться не будут.

Если быть более точным, и описывать это формально, *restrict* показывает, что только данный указатель будет использоваться для доступа к этому объекту, больше никакой указатель для этого использоваться не будет. Можно даже сказать, что к всякому объекту, доступ будет осуществляться только через один единственный указатель, если он отмечен как *restrict*.

Добавим это ключевое слово к каждому аргументу-указателю:

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict product, int* restrict ↵
    ↵ sum_product,
    int* restrict update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
```

```

        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    };
};

```

Посмотрим результаты:

Листинг 45.1: GCC x64: f1()

```

f1:
    push    r15 r14 r13 r12 rbp rdi rsi rbx
    mov     r13, QWORD PTR 120[rsi]
    mov     rbp, QWORD PTR 104[rsi]
    mov     r12, QWORD PTR 112[rsi]
    test    r13, r13
    je      .L1
    add     r13, 1
    xor     ebx, ebx
    mov     edi, 1
    xor     r11d, r11d
    jmp     .L4
.L6:
    mov     r11, rdi
    mov     rdi, rax
.L4:
    lea     rax, 0[0+r11*4]
    lea     r10, [rcx+rax]
    lea     r14, [rdx+rax]
    lea     rsi, [r8+rax]
    add     rax, r9
    mov     r15d, DWORD PTR [r10]
    add     r15d, DWORD PTR [r14]
    mov     DWORD PTR [rsi], r15d ; сохранить в sum[]
    mov     r10d, DWORD PTR [r10]
    imul    r10d, DWORD PTR [r14]
    mov     DWORD PTR [rax], r10d ; сохранить в product[]
    mov     DWORD PTR [r12+r11*4], ebx ; сохранить в update_me[]
    add     ebx, 123
    mov     r10d, DWORD PTR [rsi] ; перезагрузить sum[i]
    add     r10d, DWORD PTR [rax] ; перезагрузить product[i]
    lea     rax, 1[rdi]
    cmp     rax, r13
    mov     DWORD PTR 0[rbp+r11*4], r10d ; сохранить в sum_product[]
    jne     .L6
.L1:
    pop     rbx rsi rdi rbp r12 r13 r14 r15
    ret

```

Листинг 45.2: GCC x64: f2()

```

f2:
    push    r13 r12 rbp rdi rsi rbx
    mov     r13, QWORD PTR 104[rsi]
    mov     rbp, QWORD PTR 88[rsi]
    mov     r12, QWORD PTR 96[rsi]
    test    r13, r13
    je      .L7
    add     r13, 1
    xor     r10d, r10d
    mov     edi, 1
    xor     eax, eax
    jmp     .L10
.L11:
    mov     rax, rdi
    mov     rdi, r11
.L10:
    mov     esi, DWORD PTR [rcx+rax*4]
    mov     r11d, DWORD PTR [rdx+rax*4]
    mov     DWORD PTR [r12+rax*4], r10d ; сохранить в update_me[]
    add     r10d, 123

```



```

    lea    ebx, [rsi+r11]
    imul   r11d, esi
    mov    DWORD PTR [r8+rax*4], ebx    ; сохранить в sum[]
    mov    DWORD PTR [r9+rax*4], r11d   ; сохранить в product[]
    add    r11d, ebx
    mov    DWORD PTR 0[rbp+rax*4], r11d ; сохранить в sum_product[]
    lea    r11, 1[rdi]
    cmp    r11, r13
    jne    .L11
.L7:
    pop    rbx rsi rdi rbp r12 r13
    ret

```

Разница между скомпилированной функцией `f1()` и `f2()` такая : в `f1()`, `sum[i]` и `product[i]` загружаются снова посреди тела цикла , а в `f2()` этого нет, используются уже подсчитанные значения , ведь мы «пообещали» компилятору, что никто и ничто не изменит значения в `sum[i]` и `product[i]` во время исполнения тела цикла, поэтому он «уверен», что значения из памяти можно не загружать снова . Очевидно, второй вариант работает быстрее.

Но что будет если указатели в аргументах функций все же будут пересекаться? Это на совести программиста, а результаты вычислений будут неверными.

Вернемся к FORTRAN. Компиляторы с этого ЯП, по умолчанию, все указатели считают таковыми, поэтому, когда не было возможности указать `restrict` в Си, то FORTRAN в этих случаях мог генерировать более быстрый код.

Насколько это практично? Там, где функция работает с несколькими большими блоками в памяти. Такого очень много в линейной алгебре, например. Очень много линейной алгебры используется на суперкомпьютерах/[HPC¹](#), возможно, поэтому, традиционно, там часто используется FORTRAN, до сих пор [Eugene Loh, *The Ideal HPC Programming Language*, (2010)]. Ну а когда итераций цикла не очень много, конечно, тогда прирост скорости может и не быть ощутимым.

¹High-Performance Computing

Глава 46

Функция *abs()* без переходов

Снова вернемся к уже рассмотренному ранее примеру [13.2](#) (стр. [137](#)) и спросим себя, возможно ли сделать версию этого кода под x86 без переходов?

```
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
};
```

И ответ положительный.

46.1. Оптимизирующий GCC 4.9.1 x64

Мы можем это увидеть если скомпилируем оптимизирующим GCC 4.9:

Листинг 46.1: Оптимизирующий GCC 4.9 x64

```
my_abs:
    mov     edx, edi
    mov     eax, edi
    sar     edx, 31
; EDX здесь 0xFFFFFFFF если знак входного значения -- минус
; EDX ноль если знак входного значения -- плюс (включая ноль)
; следующие две инструкции имеют эффект только если EDX равен 0xFFFFFFFF
; либо не работают, если EDX -- ноль
    xor     eax, edx
    sub     eax, edx
    ret
```

И вот как он работает:

Арифметически сдвигаем входное значение влево на 31. Арифметический сдвиг означает знаковое расширение, так что если MSB это 1, то все 32 бита будут заполнены единицами, либо нулями в противном случае. Другими словами, инструкция `SAR REG, 31` делает `0xFFFFFFFF` если знак был отрицательным либо 0 если положительным. После исполнения `SAR`, это значение у нас в `EDX`. Затем, если значение `0xFFFFFFFF` (т.е. знак отрицательный) входное значение инвертируется (потому что `XOR REG, 0xFFFFFFFF` работает как операция инвертирования всех бит). Затем, снова, если значение `0xFFFFFFFF` (т.е. знак отрицательный), 1 прибавляется к итоговому результату (потому что вычитание `-1` из значения это то же что и инкремент). Инвертирование всех бит и инкремент, это то, как меняется знак у значения в формате two's complement: [31](#) (стр. [446](#)).

Мы можем заметить, что последние две инструкции делают что-то если знак входного значения отрицательный. В противном случае (если знак положительный) они не делают ничего, оставляя входное значение нетронутым.

Алгоритм разъяснен в [Henry S. Warren, *Hacker's Delight*, (2002)2-4]. Трудно сказать, как именно GCC сгенерировал его, оптимизировал сам или просто нашел подходящий шаблон среди известных?

46.2. Оптимизирующий GCC 4.9 ARM64

GCC 4.9 для ARM64 делает почти то же, только использует полные 64-битные регистры. Здесь меньше инструкций, потому что входное значение может быть сдвинуто используя суффикс инструкции («asr») вместо отдельной инструкции.

Листинг 46.2: Оптимизирующий GCC 4.9 ARM64

```
my_abs:
; расширить входное 32-битное значение до 64-битного в регистре X0, учитывая знак:
    sxtw    x0, w0
    eor     x1, x0, x0, asr 63
; X1=X0^(X0>>63) (арифметический сдвиг)
    sub     x0, x1, x0, asr 63
; X0=X1-(X0>>63)=X0^(X0>>63)-(X0>>63) (все сдвиги -- арифметические)
    ret
```

Глава 47

Функции с переменным количеством аргументов (*variadic*)

Функции вроде `printf()` и `scanf()` могут иметь переменное количество аргументов. Как обращаться к аргументам?

47.1. Вычисление среднего арифметического

Представим, что нам нужно вычислить [среднее арифметическое](#), и по какой-то странной причине, нам нужно задать все числа в аргументах функции.

Но в Си/Си++ функции с переменным кол-вом аргументов невозможно определить кол-во аргументов, так что обозначим значение `-1` как конец списка.

Имеется стандартный заголовочный файл `stdarg.h`, который определяет макросы для работы с такими аргументами. Их так же используют функции `printf()` и `scanf()`.

```
#include <stdio.h>
#include <stdarg.h>

int arith_mean(int v, ...)
{
    va_list args;
    int sum=v, count=1, i;
    va_start(args, v);

    while(1)
    {
        i=va_arg(args, int);
        if (i==-1) // терминатор
            break;
        sum=sum+i;
        count++;
    }

    va_end(args);
    return sum/count;
};

int main()
{
    printf ("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* терминатор */));
};
```

Самый первый аргумент должен трактоваться как обычный аргумент. Остальные аргументы загружаются используя макрос `va_arg`, и затем суммируются.

Так что внутри?

47.1.1. Соглашение о вызовах *cdecl*

Листинг 47.1: Оптимизирующий MSVC 6.0

```

_v$ = 8
_arith_mean PROC NEAR
    mov     eax, DWORD PTR _v$[esp-4] ; загрузить первый аргумент в sum
    push    esi
    mov     esi, 1                    ; count=1
    lea     edx, DWORD PTR _v$[esp]  ; адрес первого аргумента
$L838:
    mov     ecx, DWORD PTR [edx+4]   ; загрузить следующий аргумент
    add     edx, 4                   ; сдвинуть указатель на следующий аргумент
    cmp     ecx, -1                  ; это -1?
    je      SHORT $L856              ; выйти, если это так
    add     eax, ecx                  ; sum = sum + загруженный аргумент
    inc     esi                       ; count++
    jmp     SHORT $L838
$L856:
; вычислить результат деления

    cdq
    idiv    esi
    pop     esi
    ret     0
_arith_mean ENDP

$SG851 DB      '%d', 0aH, 00H

_main PROC NEAR
    push    -1
    push    15
    push    10
    push    7
    push    2
    push    1
    call    _arith_mean
    push    eax
    push    OFFSET FLAT:$SG851 ; '%d'
    call    _printf
    add     esp, 32
    ret     0
_main ENDP

```

Аргументы, как мы видим, передаются в `main()` один за одним. Первый аргумент заталкивается в локальный стек первым. Терминатор (оконечивающее значение `-1`) заталкивается последним.

Функция `arith_mean()` берет первый аргумент и сохраняет его значение в переменной `sum`. Затем, она записывает адрес второго аргумента в регистр `EDX`, берет значение оттуда, прибавляет к `sum`, и делает это в бесконечном цикле, до тех пор, пока не встретится `-1`.

Когда встретится, сумма делится на число всех значений (исключая `-1`) и **частное** возвращается.

Так что, другими словами, я бы сказал, функция обходится с фрагментом стека как с массивом целочисленных значений, бесконечной длины. Теперь нам легче понять почему в соглашениях о вызовах *cdecl* первый аргумент заталкивается в стек последним. Потому что иначе будет невозможно найти первый аргумент, или, для функции вроде `printf()`, невозможно будет найти строку формата.

47.1.2. Соглашения о вызовах на основе регистров

Наблюдательный читатель может спросить, что насчет тех соглашений о вызовах, где первые аргументы передаются в регистрах? Посмотрим:

Листинг 47.2: Оптимизирующий MSVC 2012 x64

```

$SG3013 DB      '%d', 0aH, 00H

v$ = 8

```

47.1. ВЫЧИСЛЕНИЕ СРЕДНЕГО АРИФМЕТИЧЕСКОГО

```
arith_mean PROC
    mov     DWORD PTR [rsp+8], ecx    ; первый аргумент
    mov     QWORD PTR [rsp+16], rdx   ; второй аргумент
    mov     QWORD PTR [rsp+24], r8    ; третий аргумент
    mov     eax, ecx                  ; sum = первый аргумент
    lea     rcx, QWORD PTR v$[rsp+8] ; указатель на второй аргумент
    mov     QWORD PTR [rsp+32], r9    ; 4-й аргумент
    mov     edx, DWORD PTR [rcx]      ; загрузить второй аргумент
    mov     r8d, 1                    ; count=1
    cmp     edx, -1                   ; второй аргумент равен -1?
    je      SHORT $LN8@arith_mean    ; если так, то выход
$LL3@arith_mean:
    add     eax, edx                  ; sum = sum + загруженный аргумент
    mov     edx, DWORD PTR [rcx+8]    ; загрузить следующий аргумент
    lea     rcx, QWORD PTR [rcx+8]    ; сдвинуть указатель, чтобы он указывал на аргумент за следующим
    inc     r8d                       ; count++
    cmp     edx, -1                   ; загруженный аргумент равен -1?
    jne     SHORT $LL3@arith_mean    ; перейти на начал цикла, если нет
$LN8@arith_mean:
; вычислить результат деления
    cdq
    idiv    r8d
    ret     0
arith_mean ENDP

main PROC
    sub     rsp, 56
    mov     edx, 2
    mov     DWORD PTR [rsp+40], -1
    mov     DWORD PTR [rsp+32], 15
    lea     r9d, QWORD PTR [rdx+8]
    lea     r8d, QWORD PTR [rdx+5]
    lea     ecx, QWORD PTR [rdx-1]
    call    arith_mean
    lea     rcx, OFFSET FLAT:$SG3013
    mov     edx, eax
    call    printf
    xor     eax, eax
    add     rsp, 56
    ret     0
main ENDP
```

Мы видим, что первые 4 аргумента передаются в регистрах и еще два — в стеке. Функция `arith_mean()` в начале сохраняет эти 4 аргумента в *Shadow Space* и затем обходится с *Shadow Space* и стеком за ним как с единым непрерывным массивом!

Что насчет GCC? Тут немного неуклюже всё, потому что функция делится на две части: первая часть сохраняет регистры в «red zone», обрабатывает это пространство, а вторая часть функции обрабатывает стек:

Листинг 47.3: Оптимизирующий GCC 4.9.1 x64

```
arith_mean:
    lea     rax, [rsp+8]
; сохранить 6 входных регистров в "red zone" в локальном стеке
    mov     QWORD PTR [rsp-40], rsi
    mov     QWORD PTR [rsp-32], rdx
    mov     QWORD PTR [rsp-16], r8
    mov     QWORD PTR [rsp-24], rcx
    mov     esi, 8
    mov     QWORD PTR [rsp-64], rax
    lea     rax, [rsp-48]
    mov     QWORD PTR [rsp-8], r9
    mov     DWORD PTR [rsp-72], 8
    lea     rdx, [rsp+8]
    mov     r8d, 1
    mov     QWORD PTR [rsp-56], rax
    jmp     .L5
.L7:
; обработать сохраненные аргументы
    lea     rax, [rsp-48]
```

47.2. СЛУЧАЙ С ФУНКЦИЕЙ VPRINTF()

```
    mov     ecx, esi
    add     esi, 8
    add     rcx, rax
    mov     ecx, DWORD PTR [rcx]
    cmp     ecx, -1
    je      .L4
.L8:
    add     edi, ecx
    add     r8d, 1
.L5:
    ; решить, какую часть мы сейчас будем обрабатывать
    ; текущий номер аргумента меньше или равен 6?
    cmp     esi, 47
    jbe     .L7                ; нет, тогда обрабатываем сохраненные аргументы
    ; обрабатываем аргументы из стека
    mov     rcx, rdx
    add     rdx, 8
    mov     ecx, DWORD PTR [rcx]
    cmp     ecx, -1
    jne     .L8
.L4:
    mov     eax, edi
    cdq
    idiv    r8d
    ret

.LC1:
    .string "%d\n"
main:
    sub     rsp, 8
    mov     edx, 7
    mov     esi, 2
    mov     edi, 1
    mov     r9d, -1
    mov     r8d, 15
    mov     ecx, 10
    xor     eax, eax
    call    arith_mean
    mov     esi, OFFSET FLAT:.LC1
    mov     edx, eax
    mov     edi, 1
    xor     eax, eax
    add     rsp, 8
    jmp     __printf_chk
```

Кстати, похожее использование *Shadow Space* разбирается здесь: [65.8](#) (стр. 678).

47.2. Случай с функцией *vprintf()*

Многие программисты определяют свою собственную функцию для записи в лог, которая берет строку формата вида `printf()` + переменное количество аргументов.

Еще один популярный пример это функция `die()`, которая выводит некоторое сообщение и заканчивает работу. Нам нужен какой-то способ запаковать входные аргументы неизвестного количества и передать их в функцию `printf()`. Но как? Вот зачем нужны функции с «v» в названии. Одна из них это *vprintf()*: она берет строку формата и указатель на переменную типа `va_list`:

```
#include <stdlib.h>
#include <stdarg.h>

void die (const char * fmt, ...)
{
    va_list va;
    va_start (va, fmt);

    vprintf (fmt, va);
    exit(0);
}
```

};

При ближайшем рассмотрении, мы можем увидеть, что `va_list` это указатель на массив. Скомпилируем:

Листинг 47.4: Оптимизирующий MSVC 2010

```

_fmt$ = 8
_die PROC
    ; загрузить первый аргумент (строка формата)
    mov     ecx, DWORD PTR _fmt$[esp-4]
    ; установить указатель на второй аргумент
    lea     eax, DWORD PTR _fmt$[esp]
    push    eax                ; передать указатель
    push    ecx
    call    _vprintf
    add     esp, 8
    push    0
    call    _exit
$LN3@die:
    int     3
_die ENDP

```

Мы видим, что всё что наша функция делает это просто берет указатель на аргументы, передает его в *vprintf()*, и эта функция работает с ним, как с бесконечным массивом аргументов!

Листинг 47.5: Оптимизирующий MSVC 2012 x64

```

fmt$ = 48
die PROC
    ; сохранить первые 4 аргумента в Shadow Space
    mov     QWORD PTR [rsp+8], rcx
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+24], r8
    mov     QWORD PTR [rsp+32], r9
    sub     rsp, 40
    lea     rdx, QWORD PTR fmt$[rsp+8] ; передать указатель на первый аргумент
    ; RCX здесь всё еще указывает на первый аргумент (строку формата) ф-ции die()
    ; так что vprintf() возьмет его прямо из RCX
    call    vprintf
    xor     ecx, ecx
    call    exit
    int     3
die ENDP

```


Глава 48

Обрезка строк

Весьма востребованная операция со строками — это удаление некоторых символов в начале и/или конце строки.

В этом примере, мы будем работать с функцией, удаляющей все символы перевода строки ([CR¹](#)/[LF²](#)) в конце входной строки:

```
#include <stdio.h>
#include <string.h>

char* str_trim (char *s)
{
    char c;
    size_t str_len;

    // работать до тех пор, пока \r или \n находятся в конце строки
    // остановиться, если там какой-то другой символ, или если строка пустая
    // (на старте, или в результате наших действий)
    for (str_len=strlen(s); str_len>0 && (c=s[str_len-1]); str_len--)
    {
        if (c=='\r' || c=='\n')
            s[str_len-1]=0;
        else
            break;
    };
    return s;
};

int main()
{
    // тест

    // здесь применяется strdup() для копирования строк в сегмент данных,
    // потому что иначе процесс упадет в Linux,
    // где текстовые строки располагаются в константном сегменте данных,
    // и не могут модифицироваться.

    printf ("%s\n", str_trim (strdup("")));
    printf ("%s\n", str_trim (strdup("\n")));
    printf ("%s\n", str_trim (strdup("\r")));
    printf ("%s\n", str_trim (strdup("\n\r")));
    printf ("%s\n", str_trim (strdup("\r\n")));
    printf ("%s\n", str_trim (strdup("test1\r\n")));
    printf ("%s\n", str_trim (strdup("test2\n\r")));
    printf ("%s\n", str_trim (strdup("test3\n\r\n\r")));
    printf ("%s\n", str_trim (strdup("test4\n")));
    printf ("%s\n", str_trim (strdup("test5\r")));
    printf ("%s\n", str_trim (strdup("test6\r\r\r")));
};
```

Входной аргумент всегда возвращается на выходе, это удобно, когда вам нужно объединять функции обработки строк в цепочки, как это сделано здесь в функции `main()`.

¹Carriage return (возврат каретки) (13 или `\r` в Си/Си++)

²Line feed (подача строки) (10 или `\n` в Си/Си++)

48.1. X64: ОПТИМИЗИРУЮЩИЙ MSVC 2013

Вторая часть `for() ( str_len>0 && (c=s[str_len-1]) )` называется в Си/Си++ «short-circuit» (короткое замыкание) и это очень удобно: [Денис Юричев, *Заметки о языке программирования Си/Си++1.3.8*]. Компиляторы Си/Си++ гарантируют последовательное вычисление слева направо. Так что если первое условие не истинно после вычисления, второе никогда не будет вычисляться.

48.1. x64: Оптимизирующий MSVC 2013

Листинг 48.1: Оптимизирующий MSVC 2013 x64

```
s$ = 8
str_trim PROC

; RCX это первый аргумент функции, и он всегда будет указывать на строку
    mov     rdx, rcx
; это функция strlen() встроенная в код прямо здесь:
; установить RAX в 0xFFFFFFFFFFFFFFFF (-1)
    or      rax, -1
$LL14@str_trim:
    inc     rax
    cmp     BYTE PTR [rcx+rax], 0
    jne     SHORT $LL14@str_trim
; длина входной строки 0? тогда на выход:
    test    rax, rax
    je      SHORT $LN15@str_trim
; RAX содержит длину строки
    dec     rcx
; RCX = s-1
    mov     r8d, 1
    add     rcx, rax
; RCX = s-1+strlen(s), т.е., это адрес последнего символа в строке
    sub     r8, rdx
; R8 = 1-s
$LL6@str_trim:
; загрузить последний символ строки:
; перейти, если его код 13 или 10:
    movzx   eax, BYTE PTR [rcx]
    cmp     al, 13
    je      SHORT $LN2@str_trim
    cmp     al, 10
    jne     SHORT $LN15@str_trim
$LN2@str_trim:
; последний символ имеет код 13 или 10
; записываем ноль в этом месте:
    mov     BYTE PTR [rcx], 0
; декремент адреса последнего символа,
; так что он будет указывать на символ перед только что стертым:
    dec     rcx
    lea     rax, QWORD PTR [r8+rcx]
; RAX = 1 - s + адрес текущего последнего символа
; так мы определяем, достигли ли мы первого символа, и раз так, то нам нужно остановиться
    test    rax, rax
    jne     SHORT $LL6@str_trim
$LN15@str_trim:
    mov     rax, rdx
    ret     0
str_trim ENDP
```

В начале, MSVC вставил тело функции `strlen()` прямо в код, потому что решил, что так будет быстрее чем обычная работа `strlen()` + время на вызов её и возврат из нее. Это также называется *inlining*: 44 (стр. 496).

Первая инструкция функции `strlen()` вставленная здесь, это `OR RAX, 0xFFFFFFFFFFFFFFFF`. Трудно сказать, почему MSVC использовала `OR` вместо `MOV RAX, 0xFFFFFFFFFFFFFFFF`, но он делает это часто. И конечно, это эквивалентно друг другу: все биты просто выставляются, а все выставленные биты это -1 в дополнительном коде (two's complement): 31 (стр. 446).

Кто-то мог бы спросить, зачем вообще нужно использовать число -1 в функции `strlen()`? Вследствие оптимизации, конечно. Вот что сделал MSVC:

```
; RCX = указатель на входную строку
; RAX = текущая длина строки
      or      rax, -1
label:
      inc     rax
      cmp     BYTE PTR [rcx+rax], 0
      jne     SHORT label
; RAX = длина строки
```

Попробуйте написать короче, если хотите инициализировать счетчик нулем! Ну, например:

Листинг 48.3: Наша версия `strlen()`

```
; RCX = указатель на входную строку
; RAX = текущая длина строки
      xor     rax, rax
label:
      cmp     byte ptr [rcx+rax], 0
      jz      exit
      inc     rax
      jmp     label
exit:
; RAX = длина строки
```

Не получилось. Нам придется вводить дополнительную инструкцию JMP!

Что сделал MSVC 2013, так это передвинул инструкцию `INC` в место перед загрузкой символа. Если самый первый символ — нулевой, всё нормально, `RAX` содержит 0 в этот момент, так что итоговая длина строки будет 0.

Остальную часть функции проще понять.

48.2. x64: Неоптимизирующий GCC 4.9.1

```
str_trim:
      push    rbp
      mov     rbp, rsp
      sub     rsp, 32
      mov     QWORD PTR [rbp-24], rdi
; здесь начинается первая часть for()
      mov     rax, QWORD PTR [rbp-24]
      mov     rdi, rax
      call    strlen
      mov     QWORD PTR [rbp-8], rax ; str_len
; здесь заканчивается первая часть for()
      jmp     .L2
; здесь начинается тело for()
.L5:
      cmp     BYTE PTR [rbp-9], 13 ; c=='\r'?
      je      .L3
      cmp     BYTE PTR [rbp-9], 10 ; c=='\n'?
      jne     .L4
.L3:
      mov     rax, QWORD PTR [rbp-8] ; str_len
      lea     rdx, [rax-1] ; EDX=str_len-1
      mov     rax, QWORD PTR [rbp-24] ; s
      add     rax, rdx ; RAX=s+str_len-1
      mov     BYTE PTR [rax], 0 ; s[str_len-1]=0
; тело for() заканчивается здесь
; здесь начинается третья часть for()
      sub     QWORD PTR [rbp-8], 1 ; str_len--
; здесь заканчивается третья часть for()
.L2:
; здесь начинается вторая часть for()
      cmp     QWORD PTR [rbp-8], 0 ; str_len==0?
      je      .L4 ; тогда на выход
; проверить второе условие, и загрузить "с"
```

48.3. X64: ОПТИМИЗИРУЮЩИЙ GCC 4.9.1

```
    mov     rax, QWORD PTR [rbp-8]    ; RAX=str_len
    lea     rdx, [rax-1]              ; RDX=str_len-1
    mov     rax, QWORD PTR [rbp-24]   ; RAX=s
    add     rax, rdx                  ; RAX=s+str_len-1
    movzx   eax, BYTE PTR [rax]       ; AL=s[str_len-1]
    mov     BYTE PTR [rbp-9], al      ; записать загруженный символ в "с"
    cmp     BYTE PTR [rbp-9], 0       ; это ноль?
    jne     .L5                      ; да? тогда на выход
; здесь заканчивается вторая часть for()
.L4:
; возврат "s"
    mov     rax, QWORD PTR [rbp-24]
    leave
    ret
```

Комментарии — автора книги. После исполнения `strlen()`, управление передается на метку L2, и там проверяются два выражения, одно после другого. Второе никогда не будет проверяться, если первое выражение не истинно (`str_len==0`) (это «short-circuit»).

Теперь посмотрим на эту функцию в коротком виде:

- Первая часть `for()` (вызов `strlen()`)
- `goto L2`
- L5: Тело `for()`. переход на выход, если нужно
- Третья часть `for()` (декремент `str_len`)
- L2: Вторая часть `for()`: проверить первое выражение, затем второе. переход на начало тела цикла, или выход.
- L4:// выход
- `return s`

48.3. x64: Оптимизирующий GCC 4.9.1

```
str_trim:
    push    rbx
    mov     rbx, rdi
; RBX всегда будет "s"
    call    strlen
; проверить на str_len==0 и выйти, если это так
    test    rax, rax
    je      .L9
    lea     rdx, [rax-1]
; RDX всегда будет содержать значение str_len-1, но не str_len
; так что RDX будет скорее индексом буфера
    lea     rsi, [rbx+rdx]          ; RSI=s+str_len-1
    movzx   ecx, BYTE PTR [rsi]    ; загрузить символ
    test    cl, cl
    je      .L9                    ; выйти, если это ноль
    cmp     cl, 10
    je      .L4
    cmp     cl, 13                  ; выйти, если это не '\n' и не '\r'
    jne     .L9
.L4:
; это странная инструкция. нам здесь нужно RSI=s-1
; это можно сделать, используя MOV RSI, EBX / DEC RSI
; но это две инструкции между одной
    sub     rsi, rax
; RSI = s+str_len-1-str_len = s-1
; начало главного цикла
.L12:
    test    rdx, rdx
; записать ноль по адресу s-1+str_len-1+1 = s-1+str_len = s+str_len-1
    mov     BYTE PTR [rsi+1+rdx], 0
; проверка на str_len-1==0. выход, если да.
    je      .L9
```

48.4. ARM64: НЕОПТИМИЗИРУЮЩИЙ GCC (LINARO) 4.9

```
    sub    rdx, 1                ; эквивалент str_len--
; загрузить следующий символ по адресу s+str_len-1
    movzx  ecx, BYTE PTR [rbx+rdx]
    test   cl, cl                ; это ноль? тогда выход
    je     .L9
    cmp    cl, 10                ; это '\n'?
    je     .L12
    cmp    cl, 13                ; это '\r'?
    je     .L12
.L9:
; возврат "s"
    mov    rax, rbx
    pop    rbx
    ret
```

Тут более сложный результат. Код перед циклом выполняется только один раз, но также содержит проверку символов [CR/LF](#)! Зачем нужна это дублирование кода?

Обычная реализация главного цикла это — наверное, такая:

- (начало цикла) проверить символы [CR/LF](#), принять решения
- записать нулевой символ

Но GCC решил поменять местами эти два шага. Конечно, шаг *записать нулевой символ* не может быть первым, так что нужна еще одна проверка:

- обработать первый символ. сравнить его с [CR/LF](#), выйти если символ не равен [CR/LF](#)
- (начало цикла) записать нулевой символ
- проверить символы [CR/LF](#), принять решения

Теперь основной цикл очень короткий, а это очень хорошо для современных процессоров.

Код не использует переменную `str_len`, но `str_len-1`. Так что это больше похоже на индекс в буфере. Должно быть, GCC заметил, что выражение `str_len-1` используется дважды. Так что будет лучше выделить переменную, которая всегда содержит значение равное текущей длине строки минус 1, и уменьшать его на 1 (это тот же эффект, что и уменьшать переменную `str_len`).

48.4. ARM64: Неоптимизирующий GCC (Linaro) 4.9

Реализация простая и прямолинейная:

Листинг 48.4: Неоптимизирующий GCC (Linaro) 4.9

```
str_trim:
    stp    x29, x30, [sp, -48]!
    add    x29, sp, 0
    str    x0, [x29,24] ; скопировать входной аргумент в локальный стек
    ldr    x0, [x29,24] ; s
    bl     strlen
    str    x0, [x29,40] ; переменная str_len в локальном стеке
    b      .L2
; начало главного цикла
.L5:
    ldrb   w0, [x29,39]
; W0=c
    cmp    w0, 13        ; это '\r'?
    beq    .L3
    ldrb   w0, [x29,39]
; W0=c
    cmp    w0, 10        ; это '\n'?
    bne    .L4           ; перейти в конец, если нет
.L3:
    ldr    x0, [x29,40]
; X0=str_len
    sub    x0, x0, #1
; X0=str_len-1
    ldr    x1, [x29,24]
; X1=s
```

48.5. ARM64: ОПТИМИЗИРУЮЩИЙ GCC (LINARO) 4.9

```
        add     x0, x1, x0
; X0=s+str_len-1
        strb    wzr, [x0]      ; записать байт на s+str_len-1
; декремент str_len:
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        str     x0, [x29,40]
; сохранить X0 (или str_len-1) в локальном стеке
.L2:
        ldr     x0, [x29,40]
; str_len==0?
        cmp     x0, xzr
; перейти на выход, если да
        beq     .L4
        ldr     x0, [x29,40]
; X0=str_len
        sub     x0, x0, #1
; X0=str_len-1
        ldr     x1, [x29,24]
; X1=s
        add     x0, x1, x0
; X0=s+str_len-1
; загрузить байт по адресу s+str_len-1 в W0
        ldrb    w0, [x0]
        strb    w0, [x29,39] ; сохранить загруженный байт в "с"
        ldrb    w0, [x29,39] ; перезагрузить его
; это нулевой байт?
        cmp     w0, wzr
; перейти на конец, если это ноль, или на L5, если нет
        bne     .L5
.L4:
; возврат s
        ldr     x0, [x29,24]
        ldp     x29, x30, [sp], 48
        ret
```

48.5. ARM64: Оптимизирующий GCC (Linaro) 4.9

Это более продвинутая оптимизация. Первый символ загружается в самом начале и сравнивается с 10 (символ [LF](#)). Символы также загружаются и в главном цикле, для символов после первого. Это в каком смысле похоже на этот пример: [48.3](#) (стр. [518](#)).

Листинг 48.5: Оптимизирующий GCC (Linaro) 4.9

```
str_trim:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     x19, x0
; X19 всегда будет содержать значение "s"
        bl     strlen
; X0=str_len
        cbz     x0, .L9          ; перейти на L9 (выход), если str_len==0
        sub     x1, x0, #1
; X1=X0-1=str_len-1
        add     x3, x19, x1
; X3=X19+X1=s+str_len-1
        ldrb    w2, [x19,x1]     ; загрузить байт по адресу X19+X1=s+str_len-1
; W2=загруженный символ
        cbz     w2, .L9          ; это ноль? тогда перейти на выход
        cmp     w2, 10           ; это '\n'?
        bne     .L15
.L12:
; тело главного цикла. загруженный символ в этот момент всегда 10 или 13!
        sub     x2, x1, x0
```

48.6. ARM: ОПТИМИЗИРУЮЩИЙ KEIL 6/2013 (РЕЖИМ ARM)

```
; X2=X1-X0=str_len-1-str_len=-1
    add    x2, x3, x2
; X2=X3+X2=s+str_len-1+(-1)=s+str_len-2
    strb   wzr, [x2,1]    ; записать нулевой байт по адресу s+str_len-2+1=s+str_len-1
    cbz    x1, .L9        ; str_len-1==0? перейти на выход, если это так
    sub    x1, x1, #1     ; str_len--
    ldrb   w2, [x19,x1]   ; загрузить следующий символ по адресу X19+X1=s+str_len-1
    cmp    w2, 10         ; это '\n'?
    cbz    w2, .L9        ; перейти на выход, если это ноль
    beq    .L12           ; перейти на начало цикла, если это '\n'
.L15:
    cmp    w2, 13         ; это '\r'?
    beq    .L12           ; да, перейти на начало тела цикла
.L9:
; возврат "s"
    mov    x0, x19
    ldr    x19, [sp,16]
    ldp    x29, x30, [sp], 32
    ret
```

48.6. ARM: Оптимизирующий Keil 6/2013 (Режим ARM)

И снова, компилятор пользуется условными инструкциями в режиме ARM, поэтому код более компактный.

Листинг 48.6: Оптимизирующий Keil 6/2013 (Режим ARM)

```
str_trim PROC
    PUSH    {r4,lr}
; R0=s
    MOV     r4,r0
; R4=s
    BL      strlen      ; strlen() берет значение "s" из R0
; R0=str_len
    MOV     r3,#0
; R3 всегда будет содержать 0
|L0.16|
    CMP     r0,#0        ; str_len==0?
    ADDNE   r2,r4,r0     ; (если str_len!=0) R2=R4+R0=s+str_len
    LDRBNE  r1,[r2,#-1]  ; (если str_len!=0) R1=загрузить байт по адресу R2-1=s+str_len-1
    CMPNE   r1,#0        ; (если str_len!=0) сравнить загруженный байт с 0
    BEQ     |L0.56|      ; перейти на выход, если str_len==0 или если загруженный байт - это 0
    CMP     r1,#0xd      ; загруженный байт - это '\r'?
    CMPNE   r1,#0xa      ; (если загруженный байт - это не '\r') загруженный байт - это '\r'?
    SUBEQ   r0,r0,#1     ; (если загруженный байт - это '\r' или '\n') R0-- или str_len--
    STRBEQ  r3,[r2,#-1]  ; (если загруженный байт - это '\r' или '\n') записать R3 (ноль) по
адресу R2-1=s+str_len-1
    BEQ     |L0.16|      ; перейти на начало цикла, если загруженный байт был '\r' или '\n'
|L0.56|
; возврат "s"
    MOV     r0,r4
    POP     {r4,pc}
    ENDP
```

48.7. ARM: Оптимизирующий Keil 6/2013 (Режим Thumb)

В режиме Thumb куда меньше условных инструкций, так что код более простой. Но здесь есть одна странность со сдвигами на 0x20 и 0x19. Почему компилятор Keil сделал так? Честно говоря, трудно сказать. Возможно, это выверт процесса оптимизации компилятора. Тем не менее, код будет работать корректно.

Листинг 48.7: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
str_trim PROC
    PUSH    {r4,lr}
    MOVS    r4,r0
; R4=s
```

48.8. MIPS

```

        BL      strlen      ; strlen() берет значение "s" из R0
; R0=str_len
        MOVS    r3,#0
; R3 всегда будет содержать 0
        B       |L0.24|
|L0.12|
        CMP     r1,#0xd      ; загруженный байт - это '\r'?
        BEQ     |L0.20|
        CMP     r1,#0xa      ; загруженный байт - это '\n'?
        BNE     |L0.38|      ; перейти на выход, если нет
|L0.20|
        SUBS    r0,r0,#1      ; R0-- или str_len--
        STRB    r3,[r2,#0x1f] ; записать 0 по адресу R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1
|L0.24|
        CMP     r0,#0         ; str_len==0?
        BEQ     |L0.38|      ; да? тогда перейти на выход
        ADDS    r2,r4,r0      ; R2=R4+R0=s+str_len
        SUBS    r2,r2,#0x20    ; R2=R2-0x20=s+str_len-0x20
        LDRB    r1,[r2,#0x1f] ; загрузить байт по адресу R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1 в R1
        CMP     r1,#0         ; загруженный байт - это 0?
        BNE     |L0.12|      ; перейти на начало цикла, если это не 0
|L0.38|
; возврат "s"
        MOVS    r0,r4
        POP     {r4,pc}
        ENDP

```

48.8. MIPS

Листинг 48.8: Оптимизирующий GCC 4.4.5 (IDA)

```

str_trim:
; IDA не в курсе об именах переменных, мы присвоили их сами:
saved_GP      = -0x10
saved_S0      = -8
saved_RA      = -4

        lui     $gp, (__gnu_local_gp >> 16)
        addiu   $sp, -0x20
        la      $gp, (__gnu_local_gp & 0xFFFF)
        sw      $ra, 0x20+saved_RA($sp)
        sw      $s0, 0x20+saved_S0($sp)
        sw      $gp, 0x20+saved_GP($sp)
; вызов strlen(). адрес входной строки всё еще в $a0, strlen() возьмет его оттуда:
        lw      $t9, (strlen & 0xFFFF)($gp)
        or      $at, $zero ; load delay slot, NOP
        jalr    $t9
; адрес входной строки всё еще в $a0, переложить его в $s0:
        move    $s0, $a0 ; branch delay slot
; результат strlen() (т.е. длина строки) теперь в $v0
; перейти на выход, если $v0==0 (т.е. если длина строки это 0):
        beqz    $v0, exit
        or      $at, $zero ; branch delay slot, NOP
        addiu   $a1, $v0, -1
; $a1 = $v0-1 = str_len-1
        addu    $a1, $s0, $a1
; $a1 = адрес входной строки + $a1 = s+str_len-1
; загрузить байт по адресу $a1:
        lb      $a0, 0($a1)
        or      $at, $zero ; load delay slot, NOP
; загруженный байт - это ноль? перейти на выход, если это так:
        beqz    $a0, exit
        or      $at, $zero ; branch delay slot, NOP
        addiu   $v1, $v0, -2
; $v1 = str_len-2
        addu    $v1, $s0, $v1
; $v1 = $s0+$v1 = s+str_len-2

```


48.8. MIPS

```

        li      $a2, 0xD
; пропустить тело цикла:
        b       loc_6C
        li      $a3, 0xA    ; branch delay slot
loc_5C:
; загрузить следующий байт из памяти в $a0:
        lb      $a0, 0($v1)
        move    $a1, $v1
; $a1=s+str_len-2
; перейти на выход, если загруженный байт - это ноль:
        beqz    $a0, exit
; декремент str_len:
        addiu   $v1, -1    ; branch delay slot
loc_6C:
; в этот момент, $a0=загруженный байт, $a2=0xD (символ CR) и $a3=0xA (символ LF)
; загруженный байт - это CR? тогда перейти на loc_7C:
        beq     $a0, $a2, loc_7C
        addiu   $v0, -1    ; branch delay slot
; загруженный байт - это LF? перейти на выход, если это не LF:
        bne     $a0, $a3, exit
        or      $at, $zero ; branch delay slot, NOP
loc_7C:
; загруженный байт в этот момент это CR
; перейти на loc_5C (начало тела цикла) если str_len (в $v0) не ноль:
        bnez    $v0, loc_5C
; одновременно с этим, записать ноль в этом месте памяти:
        sb      $zero, 0($a1) ; branch delay slot
; метка "exit" была так названа мною:
exit:
        lw      $ra, 0x20+saved_RA($sp)
        move    $v0, $s0
        lw      $s0, 0x20+saved_S0($sp)
        jr      $ra
        addiu   $sp, 0x20    ; branch delay slot

```

Регистры с префиксом S- называются «saved temporaries», так что, значение \$S0 сохраняется в локальном стеке и восстанавливается во время выхода.

Глава 49

Функция toupper()

Еще одна очень востребованная функция конвертирует символ из строчного в заглавный, если нужно:

```
char toupper (char c)
{
    if(c>='a' && c<='z')
        return c-'a'+'A';
    else
        return c;
}
```

Выражение `'a'+'A'` оставлено в исходном коде для удобства чтения, конечно, оно оптимизируется ¹.

ASCII-код символа «a» это 97 (или 0x61), и 65 (или 0x41) для символа «A». Разница (или расстояние) между ними в ASCII-таблица это 32 (или 0x20).

Для лучшего понимания, читатель может посмотреть на стандартную 7-битную таблицу ASCII:

Characters in the coded character set ascii.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	C-@	C-a	C-b	C-c	C-d	C-e	C-f	C-g	C-h	TAB	C-j	C-k	C-l	RET	C-n	C-o
1x	C-p	C-q	C-r	C-s	C-t	C-u	C-v	C-w	C-x	C-y	C-z	ESC	C-\	C-]	C-^	C-_
2x		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Рис. 49.1: 7-битная таблица ASCII в Emacs

49.1. x64

49.1.1. Две операции сравнения

Неоптимизирующий MSVC прямолинеен: код проверят, находится ли входной символ в интервале [97..122] (или в интервале ['a'..'z']) и вычитает 32 в таком случае. Имеется также небольшой артефакт компилятора:

Листинг 49.1: Неоптимизирующий MSVC 2013 (x64)

```
1 c$ = 8
2 toupper PROC
3     mov     BYTE PTR [rsp+8], c1
4     movsx   eax, BYTE PTR c$[rsp]
5     cmp     eax, 97
6     jl      SHORT $LN2@toupper
7     movsx   eax, BYTE PTR c$[rsp]
```

¹ Впрочем, если быть дотошным, вполне могут до сих пор существовать компиляторы, которые не оптимизируют подобное и оставляют в коде.

49.1. X64

```
8      cmp     eax, 122
9      jg      SHORT $LN2@toupper
10     movsx   eax, BYTE PTR c$[rsp]
11     sub     eax, 32
12     jmp     SHORT $LN3@toupper
13     jmp     SHORT $LN1@toupper ; compiler artefact
14 $LN2@toupper:
15     movzx   eax, BYTE PTR c$[rsp] ; unnecessary casting
16 $LN1@toupper:
17 $LN3@toupper: ; compiler artefact
18     ret     0
19 toupper ENDP
```

Важно отметить что (на строке 3) входной байт загружается в 64-битный слот локального стека. Все остальные биты ([8..63]) не трогаются, т.е. содержат случайный шум (вы можете увидеть его в отладчике). Все инструкции работают только с байтами, так что всё нормально. Последняя инструкция `MOVZX` на строке 15 берет байт из локального стека и расширяет его до 32-битного *int*, дополняя нулями.

Неоптимизирующий GCC делает почти то же самое:

Листинг 49.2: Неоптимизирующий GCC 4.9 (x64)

```
toupper:
    push     rbp
    mov      rbp, rsp
    mov      eax, edi
    mov      BYTE PTR [rbp-4], al
    cmp      BYTE PTR [rbp-4], 96
    jle      .L2
    cmp      BYTE PTR [rbp-4], 122
    jg       .L2
    movzx    eax, BYTE PTR [rbp-4]
    sub      eax, 32
    jmp      .L3
.L2:
    movzx    eax, BYTE PTR [rbp-4]
.L3:
    pop      rbp
    ret
```

49.1.2. Одна операция сравнения

Оптимизирующий MSVC работает лучше, он генерирует только одну операцию сравнения:

Листинг 49.3: Оптимизирующий MSVC 2013 (x64)

```
toupper PROC
    lea      eax, DWORD PTR [rcx-97]
    cmp      al, 25
    ja       SHORT $LN2@toupper
    movsx    eax, cl
    sub      eax, 32
    ret      0
$LN2@toupper:
    movzx    eax, cl
    ret      0
toupper ENDP
```

Уже было описано, как можно заменить две операции сравнения на одну: [43.2.1](#) (стр. [494](#)).

Мы бы переписал это на Си/Си++ так:

```
int tmp=c-97;

if (tmp>25)
    return c;
else
    return c-32;
```

49.2. ARM

Переменная *tmp* должна быть знаковая. При помощи этого, имеем две операции вычитания в случае конверсии плюс одну операцию сравнения. В то время как оригинальный алгоритм использует две операции сравнения плюс одну операцию вычитания.

Оптимизирующий GCC даже лучше, он избавился от переходов (а это хорошо: [34.1](#) (стр. [451](#))) используя инструкцию CMOVcc:

Листинг 49.4: Оптимизирующий GCC 4.9 (x64)

```
1 toupper:
2     lea     edx, [rdi-97] ; 0x61
3     lea     eax, [rdi-32] ; 0x20
4     cmp     dl, 25
5     cmova   eax, edi
6     ret
```

На строке 3 код готовит уже сконвертированное значение заранее, как если бы конверсия всегда происходила. На строке 5 это значение в EAX заменяется нетронутым входным значением, если конверсия не нужна. И тогда это значение (конечно, неверное), просто выбрасывается. Вычитание с упреждением это цена, которую компилятор платит за отсутствие условных переходов.

49.2. ARM

Оптимизирующий Keil для режима ARM также генерирует только одну операцию сравнения:

Листинг 49.5: Оптимизирующий Keil 6/2013 (Режим ARM)

```
toupper PROC
    SUB     r1,r0,#0x61
    CMP     r1,#0x19
    SUBLS   r0,r0,#0x20
    ANDLS   r0,r0,#0xff
    BX      lr
    ENDP
```

SUBLS и ANDLS исполняются только если значение **R1** меньше чем 0x19 (или равно). Они и делают конверсию.

Оптимизирующий Keil для режима Thumb также генерирует только одну операцию сравнения:

Листинг 49.6: Оптимизирующий Keil 6/2013 (Режим Thumb)

```
toupper PROC
    MOVS    r1,r0
    SUBS    r1,r1,#0x61
    CMP     r1,#0x19
    BHI     |L0.14|
    SUBS    r0,r0,#0x20
    LSLS    r0,r0,#24
    LSRS    r0,r0,#24
|L0.14|
    BX      lr
    ENDP
```

Последние две инструкции LSLS и LSRS работают как **AND reg, 0xFF**: это аналог Си/Си++-выражения $(i \ll 24) \gg 24$. Очевидно, Keil для режима Thumb решил что две 2-байтных инструкции это короче чем код, загружающий константу 0xFF плюс инструкция AND.

49.2.1. GCC для ARM64

Листинг 49.7: Неоптимизирующий GCC 4.9 (ARM64)

```
toupper:
    sub     sp, sp, #16
    strb    w0, [sp,15]
    ldrb     w0, [sp,15]
    cmp     w0, 96
    bls     .L2
```

49.3. ИТОГ

```
    ldrb    w0, [sp,15]
    cmp     w0, 122
    bhi     .L2
    ldrb    w0, [sp,15]
    sub     w0, w0, #32
    uxtb    w0, w0
    b       .L3
.L2:
    ldrb    w0, [sp,15]
.L3:
    add     sp, sp, 16
    ret
```

Листинг 49.8: Оптимизирующий GCC 4.9 (ARM64)

```
toupper:
    uxtb    w0, w0
    sub     w1, w0, #97
    uxtb    w1, w1
    cmp     w1, 25
    bhi     .L2
    sub     w0, w0, #32
    uxtb    w0, w0
.L2:
    ret
```

49.3. Итог

Все эти оптимизации компиляторов очень популярны в наше время и практикующий reverse engineer обычно часто видит такие варианты кода.

Глава 50

Неверно дизассемблированный код

Практикующие reverse engineer-ы часто сталкиваются с неверно дизассемблированным кодом .

50.1. Дизассемблирование началось в неверном месте (x86)

В отличие от ARM и MIPS (где у каждой инструкции длина или 2 или 4 байта), x86-инструкции имеют переменную длину, так что, любой дизассемблер, начиная работу с середины x86-инструкции, может выдать неверные результаты.

Как пример:

```
add    [ebp-31F7Bh], cl
dec    dword ptr [ecx-3277Bh]
dec    dword ptr [ebp-2CF7Bh]
inc    dword ptr [ebx-7A76F33Ch]
fddiv  st(4), st
db 0FFh
dec    dword ptr [ecx-21F7Bh]
dec    dword ptr [ecx-22373h]
dec    dword ptr [ecx-2276Bh]
dec    dword ptr [ecx-22B63h]
dec    dword ptr [ecx-22F4Bh]
dec    dword ptr [ecx-23343h]
jmp    dword ptr [esi-74h]
xchg   eax, ebp
clc
std
db 0FFh
db 0FFh
mov     word ptr [ebp-214h], cs ; <- дизассемблер наконец нашел здесь правильный старт
mov     word ptr [ebp-238h], ds
mov     word ptr [ebp-23Ch], es
mov     word ptr [ebp-240h], fs
mov     word ptr [ebp-244h], gs
pushf
pop     dword ptr [ebp-210h]
mov     eax, [ebp+4]
mov     [ebp-218h], eax
lea     eax, [ebp+4]
mov     [ebp-20Ch], eax
mov     dword ptr [ebp-2D0h], 10001h
mov     eax, [eax-4]
mov     [ebp-21Ch], eax
mov     eax, [ebp+0Ch]
mov     [ebp-320h], eax
mov     eax, [ebp+10h]
mov     [ebp-31Ch], eax
mov     eax, [ebp+4]
mov     [ebp-314h], eax
call    ds:IsDebuggerPresent
mov     edi, eax
lea     eax, [ebp-328h]
```

50.2. КАК ВЫГЛЯДЯТ СЛУЧАЙНЫЕ ДАННЫЕ В ДИЗАССЕМБЛИРОВАННОМ ВИДЕ?

```
push    eax
call    sub_407663
pop     ecx
test    eax, eax
jnz     short loc_402D7B
```

В начале мы видим неверно дизассемблированные инструкции, но потом, так или иначе, дизассемблер находит верный след .

50.2. Как выглядят случайные данные в дизассемблированном виде?

Общее, что можно сразу заметить, это:

- Необычно большой разброс инструкций. Самые частые x86-инструкции это **PUSH** , **MOV** , **CALL** , но здесь мы видим инструкции из любых групп: **FPU**-инструкции, инструкции **IN** / **OUT** , редкие и системные инструкции, всё друг с другом смешано в одном месте. .
- Большие и случайные значения, смещения, immediates.
- Переходы с неверными смещениями часто имеют адрес перехода в середину другой инструкции .

Листинг 50.1: случайный шум (x86)

```
mov     bl, 0Ch
mov     ecx, 0D38558Dh
mov     eax, ds:2C869A86h
db      67h
mov     dl, 0CCh
insb
movsb
push    eax
xor     [edx-53h], ah
fcom    qword ptr [edi-45A0EF72h]
pop     esp
pop     ss
in      eax, dx
dec     ebx
push    esp
lds     esp, [esi-41h]
retf
rcl     dword ptr [eax], cl
mov     cl, 9Ch
mov     ch, 0DFh
push    cs
insb
mov     esi, 0D9C65E4Dh
imul    ebp, [ecx], 66h
pushf
sal     dword ptr [ebp-64h], cl
sub     eax, 0AC433D64h
out     8Ch, eax
pop     ss
sbb     [eax], ebx
aas
xchg    cl, [ebx+ebx*4+14B31Eh]
jecxz   short near ptr loc_58+1
xor     al, 0C6h
inc     edx
db      36h
pusha
stosb
test    [ebx], ebx
sub     al, 0D3h ; 'L'
pop     eax
stosb
```

```
loc_58: ; CODE XREF: seg000:0000004A
test    [esi], eax
inc     ebp
```

50.2. КАК ВЫГЛЯДЯТ СЛУЧАЙНЫЕ ДАННЫЕ В ДИЗАССЕМБЛИРОВАННОМ ВИДЕ?

```
das
db      64h
pop     ecx
das
hlt

pop     edx
out     0B0h, al
lods b
push    ebx
cdq
out     dx, al
sub     al, 0Ah
sti
outs d
add     dword ptr [edx], 96FCBE4Bh
and     eax, 0E537EE4Fh
inc     esp
stos d
cdq
push    ecx
in      al, 0CBh
mov     ds:0D114C45Ch, al
mov     esi, 659D1985h
```

Листинг 50.2: случайный шум (x86-64)

```
lea     esi, [rax+rdx*4+43558D29h]
loc_AF3: ; CODE XREF: seg000:0000000000000B46
rcl     byte ptr [rsi+rax*8+29BB423Ah], 1
lea     ecx, cs:0FFFFFFFB2A6780Fh
mov     al, 96h
mov     ah, 0CEh
push    rsp
lods    byte ptr [esi]

db      2Fh ; /

pop     rsp
db      64h
retf     0E993h

cmp     ah, [rax+4Ah]
movzx   rsi, dword ptr [rbp-25h]
push    4Ah
movzx   rdi, dword ptr [rdi+rdx*8]

db      9Ah

rcr     byte ptr [rax+1Dh], cl
lods d
xor     [rbp+6CF20173h], edx
xor     [rbp+66F8B593h], edx
push    rbx
sbb     ch, [rbx-0Fh]
stos d
int     87h
db      46h, 4Ch
out     33h, rax
xchg    eax, ebp
test    ecx, ebp
movsd
leave
push    rsp

db      16h

xchg    eax, esi
pop     rdi
```


50.2. КАК ВЫГЛЯДЯТ СЛУЧАЙНЫЕ ДАННЫЕ В ДИЗАССЕМБЛИРОВАННОМ ВИДЕ?

```
loc_B3D: ; CODE XREF: seg000:0000000000000B5F
mov     ds:93CA685DF98A90F9h, eax
jnz     short near ptr loc_AF3+6
out     dx, eax
cwde
mov     bh, 5Dh ; ']'
movsb
pop     rbp
```

Листинг 50.3: случайный шум (ARM (Режим ARM))

```
BLNE    0xFE16A9D8
BGE     0x1634D0C
SVCCS   0x450685
STRNVT  R5, [PC], #-0x964
LDCGE   p6, c14, [R0], #0x168
STCCSL  p9, c9, [LR], #0x14C
CMNHIP  PC, R10, LSL#22
FLDMIADNV LR!, {D4}
MCR     p5, 2, R2, c15, c6, 4
BLGE    0x1139558
BLGT    0xFF9146E4
STRNEB  R5, [R4], #0xCA2
STMNEIB R5, {R0, R4, R6, R7, R9-SP, PC}
STMIA   R8, {R0, R2-R4, R7, R8, R10, SP, LR}^
STRB    SP, [R8], PC, ROR#18
LDCCS   p9, c13, [R6], #0x1BC
LDRGE   R8, [R9], #0x66E
STRNEB  R5, [R8], #-0x8C3
STCCSL  p15, c9, [R7], #-0x84
RSBLS   LR, R2, R11, ASR LR
SVC GT  0x9B0362
SVC GT  0xA73173
STMNEB  R11!, {R0, R1, R4-R6, R8, R10, R11, SP}
STR     R0, [R3], #-0xCE4
LDCGT   p15, c8, [R1], #0x2CC
LDRCCB  R1, [R11], -R7, ROR#30
BLLT    0xFED9D58C
BL      0x13E60F4
LDMVSIB R3!, {R1, R4-R7}^
USATNE  R10, #7, SP, LSL#11
LDRGEB  LR, [R1], #0xE56
STRPLT  R9, [LR], #0x567
LDRLT   R11, [R1], #-0x29B
SVCNV   0x12DB29
MVNNVS  R5, SP, LSL#25
LDCL    p8, c14, [R12], #-0x288
STCNEL  p2, c6, [R6], #-0xBC!
SVCNV   0x2E5A2F
BLX     0x1A8C97E
TEQGE   R3, #0x1100000
STMLSIA R6, {R3, R6, R10, R11, SP}
BICPLS  R12, R2, #0x5800
BNE     0x7CC408
TEQGE   R2, R4, LSL#20
SUBS    R1, R11, #0x28C
BICVS   R3, R12, R7, ASR R0
LDRMI   R7, [LR], R3, LSL#21
BLMI    0x1A79234
STMVCDB R6, {R0-R3, R6, R7, R10, R11}
EORMI   R12, R6, #0xC5
MCR RCS p1, 0xF, R1, R3, c2
```

Листинг 50.4: случайный шум (ARM (Режим Thumb))

```
LSRS    R3, R6, #0x12
LDRH    R1, [R7], #0x2C
SUBS    R0, #0x55 ; 'U'
ADR     R1, loc_3C
```

50.2. КАК ВЫГЛЯДЯТ СЛУЧАЙНЫЕ ДАННЫЕ В ДИЗАССЕМБЛИРОВАННОМ ВИДЕ?

```
LDR    R2, [SP,#0x218]
CMP    R4, #0x86
SXTB   R7, R4
LDR    R4, [R1,#0x4C]
STR    R4, [R4,R2]
STR    R0, [R6,#0x20]
BGT    0xFFFFFFFF72
LDRH   R7, [R2,#0x34]
LDRSH  R0, [R2,R4]
LDRB   R2, [R7,R2]
```

```
DCB 0x17
DCB 0xED
```

```
STRB   R3, [R1,R1]
STR    R5, [R0,#0x6C]
LDMIA  R3, {R0-R5,R7}
ASRS   R3, R2, #3
LDR    R4, [SP,#0x2C4]
SVC    0xB5
LDR    R6, [R1,#0x40]
LDR    R5, =0xB2C5CA32
STMIA  R6, {R1-R4,R6}
LDR    R1, [R3,#0x3C]
STR    R1, [R5,#0x60]
BCC    0xFFFFFFFF70
LDR    R4, [SP,#0x1D4]
STR    R5, [R5,#0x40]
ORRS   R5, R7
```

```
loc_3C ; DATA XREF: ROM:00000006
B      0xFFFFFFFF98
```

Листинг 50.5: случайный шум (MIPS little endian)

```
lw      $t9, 0xCB3($t5)
sb      $t5, 0x3855($t0)
sltiu   $a2, $a0, -0x657A
ldr      $t4, -0x4D99($a2)
daddi    $s0, $s1, 0x50A4
lw      $s7, -0x2353($s4)
bgtzl   $a1, 0x17C5C

.byte 0x17
.byte 0xED
.byte 0x4B # K
.byte 0x54 # T

lwc2     $31, 0x66C5($sp)
lwu      $s1, 0x10D3($a1)
ldr      $t6, -0x204B($zero)
lwc1     $f30, 0x4DBE($s2)
daddiu   $t1, $s1, 0x6BD9
lwu      $s5, -0x2C64($v1)
cop0     0x13D642D
bne      $gp, $t4, 0xFFFF9EF0
lh       $ra, 0x1819($s1)
sdl      $fp, -0x6474($t8)
jal      0x78C0050
ori      $v0, $s2, 0xC634
blez     $gp, 0xFFFEA9D4
swl      $t8, -0x2CD4($s2)
sltiu    $a1, $k0, 0x685
sdc1     $f15, 0x5964($at)
sw       $s0, -0x19A6($a1)
sltiu    $t6, $a3, -0x66AD
lb       $t7, -0x4F6($t3)
sd       $fp, 0x4B02($a1)
```

50.2. КАК ВЫГЛЯДЯТ СЛУЧАЙНЫЕ ДАННЫЕ В ДИЗАСЕМБЛИРОВАННОМ ВИДЕ?

Также важно помнить, что хитрым образом написанный код для распаковки и дешифровки (включая саомодифицирующийся), также может выглядеть как случайный шум, тем не менее, он исполняется корректно.

Глава 51

Обфускация

Обфускация это попытка спрятать код (или его значение) от reverse engineer-а .

51.1. Текстовые строки

Как мы увидели в (58 (стр. 648)) текстовые строки могут быть крайне полезны. Знающие об этом программисты могут попытаться их спрятать так, чтобы их не было видно в IDA или любом шестнадцатеричном редакторе .

Вот простейший метод.

Вот как строка может быть сконструирована:

```
mov     byte ptr [ebx], 'h'
mov     byte ptr [ebx+1], 'e'
mov     byte ptr [ebx+2], 'l'
mov     byte ptr [ebx+3], 'l'
mov     byte ptr [ebx+4], 'o'
mov     byte ptr [ebx+5], ' '
mov     byte ptr [ebx+6], 'w'
mov     byte ptr [ebx+7], 'o'
mov     byte ptr [ebx+8], 'r'
mov     byte ptr [ebx+9], 'l'
mov     byte ptr [ebx+10], 'd'
```

Строка также может сравниваться с другой:

```
mov     ebx, offset username
cmp     byte ptr [ebx], 'j'
jnz     fail
cmp     byte ptr [ebx+1], 'o'
jnz     fail
cmp     byte ptr [ebx+2], 'h'
jnz     fail
cmp     byte ptr [ebx+3], 'n'
jnz     fail
jz      it_is_john
```

В обоих случаях, эти строки нельзя так просто найти в шестнадцатеричном редакторе.

Кстати, точно также со строками можно работать в тех случаях, когда строку нельзя разместить в сегменте данных, например, в PIC, или в шелл-коде.

Еще метод с использованием функции `sprintf()` для конструирования:

```
sprintf(buf, "%s%c%s%c%s", "hel", 'l', "o w", 'o', "rld");
```

Код выглядит ужасно, но как простейшая мера для анти-реверсинга, это может помочь.

Текстовые строки могут также присутствовать в зашифрованном виде, в таком случае, их использование будет предв-
рять вызов функции для дешифровки. Например: 78.2 (стр. 755).

51.2. Исполняемый код

51.2.1. Вставка мусора

Обфускация исполняемого кода — это вставка случайного мусора (между настоящим кодом), который исполняется, но не делает ничего полезного.

Просто пример:

Листинг 51.1: оригинальный код

```
add    eax, ebx
mul    ecx
```

Листинг 51.2: obfuscated code

```
xor     esi, 011223344h ; мусор
add     esi, eax        ; мусор
add     eax, ebx
mov     edx, eax        ; мусор
shl     edx, 4          ; мусор
mul     ecx
xor     esi, ecx        ; мусор
```

Здесь код-мусор использует регистры, которые не используются в настоящем коде (ESI и EDX). Впрочем, промежуточные результаты полученные при исполнении настоящего кода вполне могут использоваться кодом-мусором для большей путаницы — почему нет?

51.2.2. Замена инструкций на раздутые эквиваленты

- MOV op1, op2 может быть заменена на пару PUSH op2 / POP op1 .
- JMP label может быть заменена на пару PUSH label / RET . IDA не покажет ссылок на эту метку.
- CALL label может быть заменена на следующую тройку инструкций: PUSH label_after_CALL_instruction / PUSH
- PUSH op также можно заменить на пару инструкций: SUB ESP, 4 (или 8) / MOV [ESP], op .

51.2.3. Всегда исполняющийся/никогда не исполняющийся код

Если разработчик уверен, что в ESI всегда будет 0 в этом месте :

```
mov     esi, 1
...     ; какой-то не трогающий ESI код
dec     esi
...     ; какой-то не трогающий ESI код
cmp     esi, 0
jz      real_code
; фальшивый багаж
real_code:
```

Reverse engineer-у понадобится какое-то время чтобы с этим разобраться.

Это также называется *opaque predicate*.

Еще один пример (и снова разработчик уверен, что ESI — всегда ноль):

```
add     eax, ebx        ; реальный код
mul     ecx             ; реальный код
add     eax, esi        ; opaque predicate. вместо ADD тут может быть XOR, AND или SHL, и т.д.
```

51.2.4. Сделать побольше путаницы

```
instruction 1
instruction 2
instruction 3
```

Можно заменить на:

```
begin:      jmp     ins1_label

ins2_label: instruction 2
            jmp     ins3_label

ins3_label: instruction 3
            jmp     exit:

ins1_label: instruction 1
            jmp     ins2_label

exit:
```

51.2.5. Использование косвенных указателей

```
dummy_data1    db     100h dup (0)
message1       db     'hello world',0

dummy_data2    db     200h dup (0)
message2       db     'another message',0

func           proc
...
mov     eax, offset dummy_data1 ; PE or ELF reloc here
add     eax, 100h
push    eax
call    dump_string
...
mov     eax, offset dummy_data2 ; PE or ELF reloc here
add     eax, 200h
push    eax
call    dump_string
...
func         endp
```

IDA покажет ссылки на `dummy_data1` и `dummy_data2`, но не на сами текстовые строки.

К глобальным переменным и даже функциям можно обращаться так же .

51.3. Виртуальная машина / псевдо-код

Программист может также создать свой собственный ЯП или ISA и интерпретатор для него.

(Как версии Visual Basic перед 5.0, .NET or Java machines). Reverse engineer-у придется потратить какое-то время для понимания деталей всех инструкций в ISA. Ему также возможно придется писать что-то вроде дизассемблера/декомпилятора .

51.4. Еще кое-что

Моя попытка (хотя и слабая) пропатчить компилятор Tiny C чтобы он выдавал обфусцированный код : <http://go.yurichev.com/17220>.

Использование инструкции `MOV` для действительно сложных вещей : [Stephen Dolan, *mov is Turing-complete*, (2013)] ¹.

¹Также доступно здесь: <http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf>

51.5. Упражнение

- <http://challenges.re/29>

Глава 52

Си++

52.1. Классы

52.1.1. Простой пример

Внутреннее представление классов в Си++ почти такое же, как и представление структур.

Давайте попробуем простой пример с двумя переменными, двумя конструкторами и одним методом:

```
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // конструктор по умолчанию
    {
        v1=667;
        v2=999;
    };

    c(int a, int b) // конструктор
    {
        v1=a;
        v2=b;
    };

    void dump()
    {
        printf ("%d; %d\n", v1, v2);
    };
};

int main()
{
    class c c1;
    class c c2(5,6);

    c1.dump();
    c2.dump();

    return 0;
};
```

MSVC – x86

Вот как выглядит `main()` на ассемблере:


```

_c2$ = -16 ; size = 8
_c1$ = -8 ; size = 8
_main PROC
    push ebp
    mov  ebp, esp
    sub  esp, 16
    lea  ecx, DWORD PTR _c1$[ebp]
    call ??0c@@QAE@XZ ; c::c
    push 6
    push 5
    lea  ecx, DWORD PTR _c2$[ebp]
    call ??0c@@QAE@HH@Z ; c::c
    lea  ecx, DWORD PTR _c1$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    lea  ecx, DWORD PTR _c2$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP

```

Вот что происходит. Под каждый экземпляр класса *c* выделяется по 8 байт, столько же, сколько нужно для хранения двух переменных.

Для *c1* вызывается конструктор по умолчанию без аргументов `??0c@@QAE@XZ`. Для *c2* вызывается другой конструктор `??0c@@QAE@HH@Z` и передаются два числа в качестве аргументов.

А указатель на объект (*this* в терминологии Си++) передается в регистре `ECX`. Это называется *thiscall* (52.1.1 (стр. 539)) – метод передачи указателя на объект.

В данном случае, MSVC делает это через `ECX`. Необходимо помнить, что это не стандартизированный метод, и другие компиляторы могут делать это иначе, например, через первый аргумент функции (как GCC).

Почему у имен функций такие странные имена? Это [name mangling](#).

В Си++, у класса, может иметься несколько методов с одинаковыми именами, но аргументами разных типов – это полиморфизм. Ну и конечно, у разных классов могут быть методы с одинаковыми именами.

Name mangling позволяет закодировать имя класса + имя метода + типы всех аргументов метода в одной ASCII-строке, которая затем используется как внутреннее имя функции. Это все потому что ни компоновщик¹, ни загрузчик DLL ОС (мангленные имена могут быть среди экспортов/импортов в DLL), ничего не знают о Си++ или ООП².

Далее вызывается два раза `dump()`.

Теперь смотрим на код в конструкторах:

Листинг 52.2: MSVC

```

_this$ = -4 ; size = 4
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  DWORD PTR [eax], 667
    mov  ecx, DWORD PTR _this$[ebp]
    mov  DWORD PTR [ecx+4], 999
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
    pop  ebp
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_this$ = -4 ; size = 4
_a$ = 8 ; size = 4

```

¹linker

²Объектно-Ориентированное Программирование

52.1. КЛАССЫ

```
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov ebp, esp
    push ecx
    mov DWORD PTR _this$[ebp], ecx
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR _a$[ebp]
    mov DWORD PTR [eax], ecx
    mov edx, DWORD PTR _this$[ebp]
    mov eax, DWORD PTR _b$[ebp]
    mov DWORD PTR [edx+4], eax
    mov eax, DWORD PTR _this$[ebp]
    mov esp, ebp
    pop ebp
    ret 8
??0c@@QAE@HH@Z ENDP ; c::c
```

Конструкторы – это просто функции, они используют указатель на структуру в `ECX`, копируют его себе в локальную переменную, хотя это и не обязательно.

Из стандарта Си++ мы знаем (C++11 12.1) что конструкторы не должны возвращать значение. В реальности, внутри, конструкторы возвращают указатель на созданный объект, т.е. , *this*.

И еще метод `dump()` :

Листинг 52.3: MSVC

```
_this$ = -4 ; size = 4
?dump@@@QAE@XZ PROC ; c::dump, COMDAT
; _this$ = ecx
    push ebp
    mov ebp, esp
    push ecx
    mov DWORD PTR _this$[ebp], ecx
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR [eax+4]
    push ecx
    mov edx, DWORD PTR _this$[ebp]
    mov eax, DWORD PTR [edx]
    push eax
    push OFFSET ??_C@_07NJBDCIEC@?$_CFd?$_DL?5?$_CFd?6?$_AA@
    call _printf
    add esp, 12
    mov esp, ebp
    pop ebp
    ret 0
?dump@@@QAE@XZ ENDP ; c::dump
```

Все очень просто, `dump()` берет указатель на структуру состоящую из двух *int* через `ECX`, выдергивает оттуда две переменные и передает их в `printf()`.

А если скомпилировать с оптимизацией (`/Ox`), то кода будет намного меньше:

Листинг 52.4: MSVC

```
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    mov eax, ecx
    mov DWORD PTR [eax], 667
    mov DWORD PTR [eax+4], 999
    ret 0
??0c@@QAE@XZ ENDP ; c::c

_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    mov edx, DWORD PTR _b$[esp-4]
```

52.1. КЛАССЫ

```
mov     eax, ecx
mov     ecx, DWORD PTR _a$[esp-4]
mov     DWORD PTR [eax], ecx
mov     DWORD PTR [eax+4], edx
ret     8
??0c@@QAE@HH@Z ENDP ; c::c

?dump@c@@QAE@XZ PROC ; c::dump, COMDAT
; _this$ = ecx
mov     eax, DWORD PTR [ecx+4]
mov     ecx, DWORD PTR [ecx]
push    eax
push    ecx
push    OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
call    _printf
add     esp, 12
ret     0
?dump@c@@QAE@XZ ENDP ; c::dump
```

Вот и все. Единственное о чем еще нужно сказать, это о том, что в функции `main()`, когда вызывался второй конструктор с двумя аргументами, за ним не корректировался стек при помощи `add esp, X`. В то же время, в конце конструктора вместо `RET` имеется `RET 8`.

Это потому что здесь используется `thiscall` (52.1.1 (стр. 539)), который, вместе с `stdcall` (65.2 (стр. 672)) (все это — методы передачи аргументов через стек), предлагает вызываемой функции корректировать стек. Инструкция `ret X` сначала прибавляет `X` к `ESP`, затем передает управление вызывающей функции.

См. также в соответствующем разделе о способах передачи аргументов через стек (65 (стр. 672)).

Еще, кстати, нужно отметить, что именно компилятор решает, когда вызывать конструктор и деструктор — но это и так известно из основ языка Си++.

MSVC — x86-64

Как мы уже знаем, в x86-64 первые 4 аргумента функции передаются через регистры `RCX`, `RDX`, `R8`, `R9`, а остальные — через стек. Тем не менее, указатель на объект `this` передается через `RCX`, а первый аргумент метода — в `RDX`, и т.д. Здесь это видно во внутренностях метода `c(int a, int b)`:

Листинг 52.5: Оптимизирующий MSVC 2012 x64

```
; void dump()

?dump@c@@QEA@XZ PROC ; c::dump
mov     r8d, DWORD PTR [rcx+4]
mov     edx, DWORD PTR [rcx]
lea     rcx, OFFSET FLAT:??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@ ; '%d; %d'
jmp     printf
?dump@c@@QEA@XZ ENDP ; c::dump

; c(int a, int b)

??0c@@QEAA@HH@Z PROC ; c::c
mov     DWORD PTR [rcx], edx ; первый аргумент: a
mov     DWORD PTR [rcx+4], r8d ; второй аргумент: b
mov     rax, rcx
ret     0
??0c@@QEAA@HH@Z ENDP ; c::c

; конструктор по умолчанию

??0c@@QEAA@XZ PROC ; c::c
mov     DWORD PTR [rcx], 667
mov     DWORD PTR [rcx+4], 999
mov     rax, rcx
ret     0
??0c@@QEAA@XZ ENDP ; c::c
```

Тип `int` в x64 остается 32-битным ³, поэтому здесь используются 32-битные части регистров.

В методе `dump()` вместо `RET` мы видим `JMP printf`, этот *хак* мы рассматривали ранее : 14.1.1 (стр. 150).

GCC – x86

В GCC 4.4.1 всё почти так же, за исключением некоторых различий.

Листинг 52.6: GCC 4.4.1

```
public main
main proc near

var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_18 = dword ptr -18h
var_10 = dword ptr -10h
var_8  = dword ptr -8

    push ebp
    mov  ebp, esp
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Ev
    mov  [esp+20h+var_18], 6
    mov  [esp+20h+var_1C], 5
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Eii
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    mov  eax, 0
    leave
    retn
main endp
```

Здесь мы видим, что применяется иной *name mangling* характерный для стандартов GNU ⁴ Во-вторых, указатель на экземпляр передается как первый аргумент функции — конечно же, скрыто от программиста.

Это первый конструктор:

```
_ZN1cC1Ev      public _ZN1cC1Ev ; weak
               proc near          ; CODE XREF: main+10

arg_0          = dword ptr 8

               push  ebp
               mov   ebp, esp
               mov   eax, [ebp+arg_0]
               mov   dword ptr [eax], 667
               mov   eax, [ebp+arg_0]
               mov   dword ptr [eax+4], 999
               pop   ebp
               retn
_ZN1cC1Ev      endp
```

Он просто записывает два числа по указателю, переданному в первом (и единственном) аргументе.

Второй конструктор:

³Вероятно, так решили для упрощения портирования Си/Си++-кода на x64

⁴Еще о *name mangling* разных компиляторов: [Agner Fog, *Calling conventions* (2015)].

```

_ZN1cC1Eii      public _ZN1cC1Eii
_ZN1cC1Eii      proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     edx, [ebp+arg_4]
                mov     [eax], edx
                mov     eax, [ebp+arg_0]
                mov     edx, [ebp+arg_8]
                mov     [eax+4], edx
                pop     ebp
                retn
_ZN1cC1Eii      endp

```

Это функция, аналог которой мог бы выглядеть так:

```

void ZN1cC1Eii (int *obj, int a, int b)
{
    *obj=a;
    *(obj+1)=b;
};

```

...что, в общем, предсказуемо.

И функция `dump()` :

```

_ZN1c4dumpEv     public _ZN1c4dumpEv
_ZN1c4dumpEv     proc near

var_18           = dword ptr -18h
var_14           = dword ptr -14h
var_10           = dword ptr -10h
arg_0            = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                mov     eax, [ebp+arg_0]
                mov     edx, [eax+4]
                mov     eax, [ebp+arg_0]
                mov     eax, [eax]
                mov     [esp+18h+var_10], edx
                mov     [esp+18h+var_14], eax
                mov     [esp+18h+var_18], offset aDD ; "%d; %d\n"
                call    _printf
                leave
                retn
_ZN1c4dumpEv     endp

```

Эта функция *во внутреннем представлении* имеет один аргумент, через который передается указатель на объект⁵ (*this*).

Это можно переписать на Си:

```

void ZN1c4dumpEv (int *obj)
{
    printf ("%d; %d\n", *obj, *(obj+1));
};

```

Таким образом, если брать в учет только эти простые примеры, разница между MSVC и GCC в способе кодирования имен функций (*name mangling*) и передаче указателя на экземпляр класса (через `ECX` или через первый аргумент).

⁵экземпляр класса

Первые 6 аргументов, как мы уже знаем, передаются через 6 регистров `RDI`, `RSI`, `RDX`, `RCX`, `R8` and `R9` ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]⁶), а указатель на *this* через первый (`RDI`) что мы здесь и видим. Тип *int* 32-битный и здесь. Хак с `JMP` вместо `RET` используется и здесь.

Листинг 52.7: GCC 4.4.6 x64

```
; конструктор по умолчанию
_ZN1cC2Ev:
    mov     DWORD PTR [rdi], 667
    mov     DWORD PTR [rdi+4], 999
    ret

; c(int a, int b)
_ZN1cC2Eii:
    mov     DWORD PTR [rdi], esi
    mov     DWORD PTR [rdi+4], edx
    ret

; dump()
_ZN1c4dumpEv:
    mov     edx, DWORD PTR [rdi+4]
    mov     esi, DWORD PTR [rdi]
    xor     eax, eax
    mov     edi, OFFSET FLAT:.LC0 ; "%d; %d\n"
    jmp     printf
```

52.1.2. Наследование классов

О наследованных классах можно сказать, что это та же простая структура, которую мы уже рассмотрели, только расширяемая в наследуемых классах.

Возьмем очень простой пример:

```
#include <stdio.h>

class object
{
public:
    int color;
    object() { };
    object (int color) { this->color=color; };
    void print_color() { printf ("color=%d\n", color); };
};

class box : public object
{
private:
    int width, height, depth;
public:
    box(int color, int width, int height, int depth)
    {
        this->color=color;
        this->width=width;
        this->height=height;
        this->depth=depth;
    };
    void dump()
    {
        printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, ↵
        ↵ depth);
    };
};
```

⁶Также доступно здесь: <http://x86-64.org/documentation/abi.pdf>

52.1. КЛАССЫ

```
};

class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    };
    void dump()
    {
        printf ("this is sphere. color=%d, radius=%d\n", color, radius);
    };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    b.print_color();
    s.print_color();

    b.dump();
    s.dump();

    return 0;
};
```

Исследуя сгенерированный код для функций/методов `dump()`, а также `object::print_color()`, посмотрим, какая будет разметка памяти для структур-объектов (для 32-битного кода).

Итак, методы `dump()` разных классов сгенерированные MSVC 2008 с опциями `/Ox` и `/Ob0` ⁷

Листинг 52.8: Оптимизирующий MSVC 2008 /Ob0

```
??_C@_09GCED0LPA@color?$DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ; `string'
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
; _this$ = ecx
    mov     eax, DWORD PTR [ecx]
    push    eax

; 'color=%d', 0aH, 00H
    push    OFFSET ??_C@_09GCED0LPA@color?$DN?$CFd?6?$AA@
    call    _printf
    add     esp, 8
    ret     0
?print_color@object@@QAEXXZ ENDP ; object::print_color
```

Листинг 52.9: Оптимизирующий MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov     eax, DWORD PTR [ecx+12]
    mov     edx, DWORD PTR [ecx+8]
    push    eax
    mov     eax, DWORD PTR [ecx+4]
    mov     ecx, DWORD PTR [ecx]
    push    edx
    push    eax
    push    ecx

; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H ; `string'
    push    OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
```

⁷ опция `/Ob0` означает отмену inline expansion, ведь вставка компилятором тела функции/метода прямо в код где он вызывается, может затруднить наши эксперименты

52.1. КЛАССЫ

```
call _printf
add esp, 20
ret 0
?dump@box@@@QAEXXZ ENDP ; box::dump
```

Листинг 52.10: Оптимизирующий MSVC 2008 /Ob0

```
?dump@sphere@@@QAEXXZ PROC ; sphere::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+4]
mov ecx, DWORD PTR [ecx]
push eax
push ecx

; 'this is sphere. color=%d, radius=%d', 0aH, 00H
push OFFSET ??_C@_0CF@EFEDJLDC@this?5is?5sphere?4?5color?$DN?$CFd?0?5radius@
call _printf
add esp, 12
ret 0
?dump@sphere@@@QAEXXZ ENDP ; sphere::dump
```

Итак, разметка полей получается следующая:

(базовый класс *object*)

смещение	описание
+0x0	int color

(унаследованные классы)

box:

смещение	описание
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

sphere:

смещение	описание
+0x0	int color
+0x4	int radius

Посмотрим тело `main()` :

Листинг 52.11: Оптимизирующий MSVC 2008 /Ob0

```
PUBLIC _main
_TEXT SEGMENT
_s$ = -24 ; size = 8
_b$ = -16 ; size = 16
_main PROC
sub esp, 24
push 30
push 20
push 10
push 1
lea ecx, DWORD PTR _b$[esp+40]
call ??0box@@@QAE@HHHH@Z ; box::box
push 40
push 2
lea ecx, DWORD PTR _s$[esp+32]
call ??0sphere@@@QAE@HH@Z ; sphere::sphere
lea ecx, DWORD PTR _b$[esp+24]
call ?print_color@object@@@QAEXXZ ; object::print_color
lea ecx, DWORD PTR _s$[esp+24]
call ?print_color@object@@@QAEXXZ ; object::print_color
lea ecx, DWORD PTR _b$[esp+24]
call ?dump@box@@@QAEXXZ ; box::dump
lea ecx, DWORD PTR _s$[esp+24]
call ?dump@sphere@@@QAEXXZ ; sphere::dump
```


52.1. КЛАССЫ

```
xor    eax, eax
add    esp, 24
ret    0
_main  ENDP
```

Наследованные классы всегда должны добавлять свои поля после полей базового класса для того, чтобы методы базового класса могли продолжать работать со своими собственными полями.

Когда метод `object::print_color()` вызывается, ему в качестве `this` передается указатель и на объект типа `box` и на объект типа `sphere`, так как он может легко работать с классами `box` и `sphere`, потому что поле `color` в этих классах всегда стоит по тому же адресу (по смещению `0x0`).

Можно также сказать, что методу `object::print_color()` даже не нужно знать, с каким классом он работает, до тех пор, пока будет соблюдаться условие *закрепления* полей по тем же адресам, а это условие соблюдается всегда.

А если вы создадите класс-наследник класса `box`, например, то компилятор будет добавлять новые поля уже за полем `depth`, оставляя уже имеющиеся поля класса `box` по тем же адресам.

Так, метод `box::dump()` будет нормально работать обращаясь к полям `color/width/height/depth` всегда находящимся по известным адресам.

Код на GCC практически такой же, за исключением способа передачи `this` (он, как уже было указано, передается в первом аргументе, вместо регистра `ECX`).

52.1.3. Инкапсуляция

Инкапсуляция — это сокрытие данных в *private* секциях класса, например, чтобы разрешить доступ к ним только для методов этого класса, но не более.

Однако, маркируется ли как-нибудь в коде тот сам факт, что некоторое поле — приватное, а некоторое другое — нет?

Нет, никак не маркируется.

Попробуем простой пример:

```
#include <stdio.h>

class box
{
private:
    int color, width, height, depth;
public:
    box(int color, int width, int height, int depth)
    {
        this->color=color;
        this->width=width;
        this->height=height;
        this->depth=depth;
    };
    void dump()
    {
        printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, ↵
        ↵ depth);
    };
};
```

Снова скомпилируем в MSVC 2008 с опциями `/Ox` и `/Ob0` и посмотрим код метода `box::dump()` :

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
;_this$ = ecx
mov    eax, DWORD PTR [ecx+12]
mov    edx, DWORD PTR [ecx+8]
push   eax
mov    eax, DWORD PTR [ecx+4]
mov    ecx, DWORD PTR [ecx]
push   edx
push   eax
push   ecx
; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H
```

52.1. КЛАССЫ

```
push OFFSET ??_C@_ODG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
call _printf
add esp, 20
ret 0
?dump@box@@@QAEXXZ ENDP ; box::dump
```

Разметка полей в классе выходит такой:

смещение	описание
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

Все поля приватные и недоступные для модификации из других функций, но, зная эту разметку, сможем ли мы создать код модифицирующий эти поля?

Для этого добавим функцию `hack_oop_encapsulation()`, которая если обладает приведенным ниже телом, то просто не скомпилируется:

```
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // этот код не может быть скомпилирован:
                // "error C2248: 'box::width' : cannot access private member declared in class 'box'"
};
```

Тем не менее, если преобразовать тип `box` к типу *указатель на массив int*, и если модифицировать полученный массив *int*-ов, тогда всё получится.

```
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(o);
    ptr_to_object[1]=123;
};
```

Код этой функции довольно прост — можно сказать, функция берет на вход указатель на массив *int*-ов и записывает 123 во второй *int*:

```
?hack_oop_encapsulation@@YAXPAVbox@@@Z PROC ; hack_oop_encapsulation
mov eax, DWORD PTR _o$[esp-4]
mov DWORD PTR [eax+4], 123
ret 0
?hack_oop_encapsulation@@YAXPAVbox@@@Z ENDP ; hack_oop_encapsulation
```

Проверим, как это работает:

```
int main()
{
    box b(1, 10, 20, 30);

    b.dump();

    hack_oop_encapsulation(&b);

    b.dump();

    return 0;
};
```

Запускаем:

```
this is box. color=1, width=10, height=20, depth=30
this is box. color=1, width=123, height=20, depth=30
```

Выходит, инкапсуляция — это защита полей класса только на стадии компиляции. Компилятор ЯП Си++ не позволяет сгенерировать код прямо модифицирующий защищенные поля, тем не менее, используя *грязные трюки* — это вполне возможно.

52.1.4. Множественное наследование

Множественное наследование — это создание класса наследующего поля и методы от двух или более классов.

Снова напишем простой пример:

```
#include <stdio.h>

class box
{
    public:
        int width, height, depth;
        box() { };
        box(int width, int height, int depth)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. width=%d, height=%d, depth=%d\n", width, height, depth);
        };
        int get_volume()
        {
            return width * height * depth;
        };
};

class solid_object
{
    public:
        int density;
        solid_object() { };
        solid_object(int density)
        {
            this->density=density;
        };
        int get_density()
        {
            return density;
        };
        void dump()
        {
            printf ("this is solid_object. density=%d\n", density);
        };
};

class solid_box: box, solid_object
{
    public:
        solid_box (int width, int height, int depth, int density)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
            this->density=density;
        };
        void dump()
        {
            printf ("this is solid_box. width=%d, height=%d, depth=%d, density=%d\n", width, height, ↵
            ↵ depth, density);
        };
        int get_weight() { return get_volume() * get_density(); };
};

int main()
{
    box b(10, 20, 30);
    solid_object so(100);
```

52.1. КЛАССЫ

```
solid_box sb(10, 20, 30, 3);

b.dump();
so.dump();
sb.dump();
printf ("%d\n", sb.get_weight());

return 0;
};
```

Снова скомпилируем в MSVC 2008 с опциями `/Ox` и `/Ob0` и посмотрим код методов `box::dump()`, `solid_object::dump()` и `solid_box::dump()`:

Листинг 52.12: Оптимизирующий MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+8]
mov edx, DWORD PTR [ecx+4]
push eax
mov eax, DWORD PTR [ecx]
push edx
push eax
; 'this is box. width=%d, height=%d, depth=%d', 0aH, 00H
push OFFSET ??_C@_OCM@DIKPHDFI@this?5is?5box?4?5width?$DN?$CFd?0?5height?$DN?$CFd@
call _printf
add esp, 16
ret 0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Листинг 52.13: Оптимизирующий MSVC 2008 /Ob0

```
?dump@solid_object@@QAEXXZ PROC ; solid_object::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx]
push eax
; 'this is solid_object. density=%d', 0aH
push OFFSET ??_C@_OCC@KICFJINL@this?5is?5solid_object?4?5density?$DN?$CFd@
call _printf
add esp, 8
ret 0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump
```

Листинг 52.14: Оптимизирующий MSVC 2008 /Ob0

```
?dump@solid_box@@QAEXXZ PROC ; solid_box::dump, COMDAT
; _this$ = ecx
mov eax, DWORD PTR [ecx+12]
mov edx, DWORD PTR [ecx+8]
push eax
mov eax, DWORD PTR [ecx+4]
mov ecx, DWORD PTR [ecx]
push edx
push eax
push ecx
; 'this is solid_box. width=%d, height=%d, depth=%d, density=%d', 0aH
push OFFSET ??_C@_ODO@HNCNIHNN@this?5is?5solid_box?4?5width?$DN?$CFd?0?5hei@
call _printf
add esp, 20
ret 0
?dump@solid_box@@QAEXXZ ENDP ; solid_box::dump
```

Выходит, имеем такую разметку в памяти для всех трех классов:

класс `box`:

смещение	описание
+0x0	width
+0x4	height
+0x8	depth

52.1. КЛАССЫ

класс `solid_object`:

смещение	описание
+0x0	density

Можно сказать, что разметка класса `solid_box` объединённая:

Класс `solid_box`:

смещение	описание
+0x0	width
+0x4	height
+0x8	depth
+0xC	density

Код методов `box::get_volume()` и `solid_object::get_density()` тривиален:

Листинг 52.15: Оптимизирующий MSVC 2008 /Ob0

```
?get_volume@box@@QAEHXZ PROC ; box::get_volume, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx+8]
    imul eax, DWORD PTR [ecx+4]
    imul eax, DWORD PTR [ecx]
    ret 0
?get_volume@box@@QAEHXZ ENDP ; box::get_volume
```

Листинг 52.16: Оптимизирующий MSVC 2008 /Ob0

```
?get_density@solid_object@@QAEHXZ PROC ; solid_object::get_density, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx]
    ret 0
?get_density@solid_object@@QAEHXZ ENDP ; solid_object::get_density
```

А вот код метода `solid_box::get_weight()` куда интереснее:

Листинг 52.17: Оптимизирующий MSVC 2008 /Ob0

```
?get_weight@solid_box@@QAEHXZ PROC ; solid_box::get_weight, COMDAT
; _this$ = ecx
    push esi
    mov esi, ecx
    push edi
    lea ecx, DWORD PTR [esi+12]
    call ?get_density@solid_object@@QAEHXZ ; solid_object::get_density
    mov ecx, esi
    mov edi, eax
    call ?get_volume@box@@QAEHXZ ; box::get_volume
    imul eax, edi
    pop edi
    pop esi
    ret 0
?get_weight@solid_box@@QAEHXZ ENDP ; solid_box::get_weight
```

`get_weight()` просто вызывает два метода, но для `get_volume()` он передает просто указатель на `this`, а для `get_density()`, он передает указатель на `this` сдвинутый на 12 байт (либо `0xC` байт), а там, в разметке класса `solid_box`, как раз начинаются поля класса `solid_object`.

Так, метод `solid_object::get_density()` будет полагать что работает с обычным классом `solid_object`, а метод `box::get_volume()` будет работать только со своими тремя полями, полагая, что работает с обычным экземпляром класса `box`.

Таким образом, можно сказать, что экземпляр класса-наследника нескольких классов представляет в памяти просто *объединённый* класс, содержащий все унаследованные поля. А каждый унаследованный метод вызывается с передачей ему указателя на соответствующую часть структуры.

52.1.5. Виртуальные методы

И снова простой пример:

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        virtual void dump()
        {
            printf ("color=%d\n", color);
        };
};

class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, ↵
            ↵ depth);
        };
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
            this->radius=radius;
        };
        void dump()
        {
            printf ("this is sphere. color=%d, radius=%d\n", color, radius);
        };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    object *o1=&b;
    object *o2=&s;

    o1->dump();
    o2->dump();
    return 0;
};
```

У класса *object* есть виртуальный метод `dump()`, впоследствии заменяемый в классах-наследниках *box* и *sphere*.

Если в какой-то среде, где неизвестно, какого типа является экземпляр класса, как в функции `main()` в примере, вы-

52.1. КЛАССЫ

зывается виртуальный метод `dump()`, где-то должна сохраняться информация о том, какой же метод в итоге вызвать.

Скомпилируем в MSVC 2008 с опциями `/Ox` и `/Ob0` и посмотрим код функции `main()`:

```
_s$ = -32 ; size = 12
_b$ = -20 ; size = 20
_main PROC
    sub     esp, 32
    push    30
    push    20
    push    10
    push    1
    lea     ecx, DWORD PTR _b$[esp+48]
    call    ??0box@@QAE@HHHH@Z ; box::box
    push    40
    push    2
    lea     ecx, DWORD PTR _s$[esp+40]
    call    ??0sphere@@QAE@HH@Z ; sphere::sphere
    mov     eax, DWORD PTR _b$[esp+32]
    mov     edx, DWORD PTR [eax]
    lea     ecx, DWORD PTR _b$[esp+32]
    call    edx
    mov     eax, DWORD PTR _s$[esp+32]
    mov     edx, DWORD PTR [eax]
    lea     ecx, DWORD PTR _s$[esp+32]
    call    edx
    xor     eax, eax
    add     esp, 32
    ret     0
_main ENDP
```

Указатель на функцию `dump()` берется откуда-то из экземпляра класса (объекта). Где мог записаться туда адрес нового метода-функции? Только в конструкторах, больше нигде: ведь в функции `main()` ничего более не вызывается. ⁸

Посмотрим код конструктора класса `box`:

```
??_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@ ; box `RTTI Type Descriptor'
    DD      00H
    DB      '.?AVbox@', 00H

??_R1A@?0A@EA@box@@@8 DD FLAT:??_R0?AVbox@@@8 ; box::`RTTI Base Class Descriptor at (0,-1,0,64)'
    DD      01H
    DD      00H
    DD      0fffffffH
    DD      00H
    DD      040H
    DD      FLAT:??_R3box@@@8

??_R2box@@@8 DD FLAT:??_R1A@?0A@EA@box@@@8 ; box::`RTTI Base Class Array'
    DD      FLAT:??_R1A@?0A@EA@object@@@8

??_R3box@@@8 DD 00H ; box::`RTTI Class Hierarchy Descriptor'
    DD      00H
    DD      02H
    DD      FLAT:??_R2box@@@8

??_R4box@@6B@ DD 00H ; box::`RTTI Complete Object Locator'
    DD      00H
    DD      00H
    DD      FLAT:??_R0?AVbox@@@8
    DD      FLAT:??_R3box@@@8

??_7box@@6B@ DD FLAT:??_R4box@@6B@ ; box::`vtable'
    DD      FLAT:?dump@box@@UAEXXZ

_color$ = 8 ; size = 4
_width$ = 12 ; size = 4
_height$ = 16 ; size = 4
```

⁸Об указателях на функции читайте больше в соответствующем разделе: [\(24 \(стр. 380\)\)](#)

52.2. OSTREAM

```
_depth$ = 20 ; size = 4
??0box@@QAE@HHHH@Z PROC ; box::box, COMDAT
; _this$ = ecx
    push esi
    mov esi, ecx
    call ??0object@@QAE@XZ ; object::object
    mov eax, DWORD PTR _color$[esp]
    mov ecx, DWORD PTR _width$[esp]
    mov edx, DWORD PTR _height$[esp]
    mov DWORD PTR [esi+4], eax
    mov eax, DWORD PTR _depth$[esp]
    mov DWORD PTR [esi+16], eax
    mov DWORD PTR [esi], OFFSET ??_7box@@6B@
    mov DWORD PTR [esi+8], ecx
    mov DWORD PTR [esi+12], edx
    mov eax, esi
    pop esi
    ret 16
??0box@@QAE@HHHH@Z ENDP ; box::box
```

Здесь мы видим, что разметка класса немного другая: в качестве первого поля имеется указатель на некую таблицу `box::`vftable'` (название оставлено компилятором MSVC).

В этой таблице есть ссылка на таблицу с названием `box::`RTTI Complete Object Locator'` и еще ссылка на метод `box::dump()`. Итак, это называется таблица виртуальных методов и [RTTI](#)<sup>9</sup>. Таблица виртуальных методов хранит в себе адреса методов, а [RTTI](#) хранит информацию о типах вообще. Кстати, [RTTI](#)-таблицы — это именно те таблицы, информация из которых используются при вызове `dynamic_cast` и `typeid` в Си++. Вы можете увидеть, что здесь хранится даже имя класса в виде обычной строки. Так, какой-нибудь метод базового класса `object` может вызвать виртуальный метод `object::dump()` что в итоге вызовет нужный метод унаследованного класса, потому что информация о нем присутствует прямо в этой структуре класса.

Работа с этими таблицами и поиск адреса нужного метода, занимает какое-то время процессора, возможно, поэтому считается что работа с виртуальными методами медленна.

В сгенерированном коде от GCC [RTTI](#)-таблицы устроены чуть-чуть иначе.

52.2. ostream

Начнем снова с примера типа «hello world», на этот раз используя `ostream`:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Из практически любого учебника Си++, известно, что операцию « можно заменить для других типов. Что и делается в `ostream`. Видно, что в реальности вызывается `operator<<` для `ostream`:

Листинг 52.18: MSVC 2012 (reduced listing)

```
$SG37112 DB 'Hello, world!', 0aH, 00H

_main PROC
    push OFFSET $SG37112
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
    call ???$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; ↵
    ↵ std::operator<<<std::char_traits<char> >
    add esp, 8
    xor eax, eax
    ret 0
_main ENDP
```

Немного переделаем пример:

⁹Run-time type information


```
#include <iostream>

int main()
{
    std::cout << "Hello, " << "world!\n";
}
```

И снова, из многих учебников по Си++, известно, что результат каждого `operator<<` в ostream передается в следующий. Действительно:

Листинг 52.19: MSVC 2012

```
$SG37112 DB 'world!', 0aH, 00H
$SG37113 DB 'Hello, ', 00H

_main PROC
    push OFFSET $SG37113 ; 'Hello, '
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
    call ???$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; ↵
    ↵ std::operator<<<std::char_traits<char> >
    add esp, 8

    push OFFSET $SG37112 ; 'world!'
    push eax ; результат работы предыдущей ф-ции
    call ???$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; ↵
    ↵ std::operator<<<std::char_traits<char> >
    add esp, 8

    xor eax, eax
    ret 0
_main ENDP
```

Если переименовать название метода `operator<<` в `f()`, то этот код выглядел бы так:

```
f(f(std::cout, "Hello, "), "world!");
```

GCC генерирует практически такой же код как и MSVC.

52.3. References

References в Си++ это тоже указатели (11 (стр. 106)), но их называют *безопасными* (safe), потому что работая с ними, труднее сделать ошибку (C++11 8.3.2). Например, reference всегда должен указывать объект того же типа и не может быть NULL [Marshall Cline, C++ FAQ8.6]. Более того, reference нельзя менять, нельзя его заставить указывать на другой объект (reseat) [Marshall Cline, C++ FAQ8.5].

Если мы попробуем изменить пример с указателями (11 (стр. 106)) чтобы он использовал reference вместо указателей ...

```
void f2 (int x, int y, int & sum, int & product)
{
    sum=x+y;
    product=x*y;
};
```

...то выяснится, что скомпилированный код абсолютно такой же как и в примере с указателями (11 (стр. 106)):

Листинг 52.20: Оптимизирующий MSVC 2010

```
_x$ = 8 ; size = 4
_y$ = 12 ; size = 4
_sum$ = 16 ; size = 4
_product$ = 20 ; size = 4
?f2@@YAXHHAANO@Z PROC ; f2
    mov ecx, DWORD PTR _y$[esp-4]
    mov eax, DWORD PTR _x$[esp-4]
    lea edx, DWORD PTR [eax+ecx]
    imul eax, ecx
    mov ecx, DWORD PTR _product$[esp-4]
```

52.4. STL

```
push esi
mov     esi, DWORD PTR _sum$[esp]
mov     DWORD PTR [esi], edx
mov     DWORD PTR [ecx], eax
pop     esi
ret     0
?f2@@YAXHHAAH0@Z ENDP                ; f2
```

(Почему у функций в Си++ такие странные имена, описано здесь : [52.1.1](#) (стр. 539).)

Следовательно, references в С++ эффективны точно так же, как и обычные указатели.

52.4. STL

N.B.: все примеры здесь были проверены только в 32-битной среде . x64-версии не были проверены.

52.4.1. std::string

Как устроена структура

Многие строковые библиотеки [Денис Юричев, *Заметки о языке программирования Си/Си++2.2*] обеспечивают структуру содержащую ссылку на буфер собственно со строкой, переменная всегда содержащую длину строки (что очень удобно для массы функций [Денис Юричев, *Заметки о языке программирования Си/Си++2.2.1*]) и переменную содержащую текущий размер буфера. Строка в буфере обыкновенно оканчивается нулем: это для того чтобы указатель на буфер можно было передавать в функции требующие на вход обычную сишную ASCIIZ-строку.

Стандарт Си++ не описывает, как именно нужно реализовывать std::string, но как правило они реализованы как описано выше, с небольшими дополнениями .

Строки в Си++ это не класс (как, например, QString в Qt), а темплейт (basic_string), это сделано для того чтобы поддерживать строки содержащие разного типа символы: как минимум char и wchar_t.

Так что, std::string это класс с базовым типом char. А std::wstring это класс с базовым типом wchar_t.

MSVC

В реализации MSVC, вместо ссылки на буфер может содержаться сам буфер (если строка короче 16-и символов).

Это означает что каждая короткая строка будет занимать в памяти по крайней мере $16 + 4 + 4 = 24$ байт для 32-битной среды либо $16 + 8 + 8 = 32$ байта в 64-битной, а если строка длиннее 16-и символов, то прибавьте еще длину самой строки .

Листинг 52.21: пример для MSVC

```
#include <string>
#include <stdio.h>

struct std_string
{
    union
    {
        char buf[16];
        char* ptr;
    } u;
    size_t size;      // AKA 'Mysize' в MSVC
    size_t capacity; // AKA 'Myres' в MSVC
};

void dump_std_string(std::string s)
{
    struct std_string *p=(struct std_string*)&s;
    printf ("%s] size:%d capacity:%d\n", p->size>16 ? p->u.ptr : p->u.buf, p->size, p->capacity);
};

int main()
```

52.4. STL

```
{
    std::string s1="short string";
    std::string s2="string longer than 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // это работает без использования c_str()
    printf ("%s\n", &s1);
    printf ("%s\n", s2);
};
```

Собственно, из этого исходника почти всё ясно.

Несколько замечаний:

Если строка короче 16-и символов, то отдельный буфер для строки в [куче](#) выделяться не будет. Это удобно потому что на практике, действительно немало строк короткие. Вероятно, разработчики в Microsoft выбрали размер в 16 символов как разумный баланс.

Теперь очень важный момент в конце функции `main()`: мы не пользуемся методом `c_str()`, тем не менее, если это скомпилировать и запустить, то обе строки появятся в консоли!

Работает это вот почему.

В первом случае строка короче 16-и символов и в начале объекта `std::string` (его можно рассматривать просто как структуру) расположен буфер с этой строкой. `printf()` трактует указатель как указатель на массив символов оканчивающийся нулем и поэтому всё работает.

Вывод второй строки (длиннее 16-и символов) даже еще опаснее: это вообще типичная программистская ошибка (или опечатка), забыть дописать `c_str()`. Это работает потому что в это время в начале структуры расположен указатель на буфер. Это может надолго остаться незамеченным: до тех пор пока там не появится строка короче 16-и символов, тогда процесс упадет.

GCC

В реализации GCC в структуре есть еще одна переменная — `reference count`.

Интересно, что указатель на экземпляр класса `std::string` в GCC указывает не на начало самой структуры, а на указатель на буфер. В `libstdc++-v3\include\bits\basic_string.h` мы можем прочитать что это сделано для удобства отладки:

```
* The reason you want _M_data pointing to the character %array and
* not the _Rep is so that the debugger can see the string
* contents. (Probably we should add a non-inline member to get
* the _Rep for the debugger to use, so users can check the actual
* string length.)
```

[исходный код basic_string.h](#)

В нашем примере мы учитываем это:

Листинг 52.22: пример для GCC

```
#include <string>
#include <stdio.h>

struct std_string
{
    size_t length;
    size_t capacity;
    size_t refcount;
};

void dump_std_string(std::string s)
{
    char *p1=*(char**)&s; // обход проверки типов GCC
    struct std_string *p2=(struct std_string*)(p1-sizeof(struct std_string));
    printf ("[%s] size:%d capacity:%d\n", p1, p2->length, p2->capacity);
};
```

52.4. STL

```
int main()
{
    std::string s1="short string";
    std::string s2="string longer than 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // обход проверки типов GCC:
    printf ("%s\n", *(char**)&s1);
    printf ("%s\n", *(char**)&s2);
};
```

Нужны еще небольшие хаки чтобы симитировать типичную ошибку, которую мы уже видели выше, из-за более ужесточенной проверки типов в GCC, тем не менее, printf() работает и здесь без c_str() .

Чуть более сложный пример

```
#include <string>
#include <stdio.h>

int main()
{
    std::string s1="Hello, ";
    std::string s2="world!\n";
    std::string s3=s1+s2;

    printf ("%s\n", s3.c_str());
}
```

Листинг 52.23: MSVC 2012

```
$SG39512 DB 'Hello, ', 00H
$SG39514 DB 'world!', 0aH, 00H
$SG39581 DB '%s', 0aH, 00H

_s2$ = -72 ; size = 24
_s3$ = -48 ; size = 24
_s1$ = -24 ; size = 24
_main PROC
    sub     esp, 72

    push    7
    push    OFFSET $SG39512
    lea     ecx, DWORD PTR _s1$[esp+80]
    mov     DWORD PTR _s1$[esp+100], 15
    mov     DWORD PTR _s1$[esp+96], 0
    mov     BYTE PTR _s1$[esp+80], 0
    call    ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBIDI@Z ; std::↵
    ↵ basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    push    7
    push    OFFSET $SG39514
    lea     ecx, DWORD PTR _s2$[esp+80]
    mov     DWORD PTR _s2$[esp+100], 15
    mov     DWORD PTR _s2$[esp+96], 0
    mov     BYTE PTR _s2$[esp+80], 0
    call    ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBIDI@Z ; std::↵
    ↵ basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    lea     eax, DWORD PTR _s2$[esp+72]
    push    eax
    lea     eax, DWORD PTR _s1$[esp+76]
    push    eax
    lea     eax, DWORD PTR _s3$[esp+80]
    push    eax
    call    ??H$DU?$char_traits@D@std@@V?$allocator@D@1@std@@YA?AV?$basic_string@DU?↵
    ↵ $char_traits@D@std@@V?$allocator@D@2@@0@ABV10@0@Z ; std::operator+<char,std::char_traits<char>,↵
    ↵ std::allocator<char> >
```

```

; вставленный код метода (inlined) c_str():
cmp  DWORD PTR _s3$[esp+104], 16
lea  eax, DWORD PTR _s3$[esp+84]
cmovae eax, DWORD PTR _s3$[esp+84]

push eax
push OFFSET $SG39581
call _printf
add  esp, 20

cmp  DWORD PTR _s3$[esp+92], 16
jb   SHORT $LN119@main
push DWORD PTR _s3$[esp+72]
call ??3@YAXPAX@Z          ; operator delete
add  esp, 4
$LN119@main:
cmp  DWORD PTR _s2$[esp+92], 16
mov  DWORD PTR _s3$[esp+92], 15
mov  DWORD PTR _s3$[esp+88], 0
mov  BYTE PTR _s3$[esp+72], 0
jb   SHORT $LN151@main
push DWORD PTR _s2$[esp+72]
call ??3@YAXPAX@Z          ; operator delete
add  esp, 4
$LN151@main:
cmp  DWORD PTR _s1$[esp+92], 16
mov  DWORD PTR _s2$[esp+92], 15
mov  DWORD PTR _s2$[esp+88], 0
mov  BYTE PTR _s2$[esp+72], 0
jb   SHORT $LN195@main
push DWORD PTR _s1$[esp+72]
call ??3@YAXPAX@Z          ; operator delete
add  esp, 4
$LN195@main:
xor  eax, eax
add  esp, 72
ret  0
_main ENDP

```

Собственно, компилятор не конструирует строки статически: да в общем-то и как это возможно, если буфер с ней нужно хранить в [куче](#)? Вместо этого в сегменте данных хранятся обычные [ASCIIZ](#)-строки, а позже, во время выполнения, при помощи метода «assign», конструируются строки s1 и s2. При помощи `operator+`, создается строка s3.

Обратите внимание на то что вызов метода `c_str()` отсутствует, потому что его код достаточно короткий и компилятор вставил его прямо здесь: если строка короче 16-и байт, то в регистре EAX остается указатель на буфер, а если длиннее, то из этого же места достается адрес на буфер расположенный в [куче](#).

Далее следуют вызовы трех деструкторов, причем, они вызываются только если строка длиннее 16-и байт: тогда нужно освободить буфера в [куче](#). В противном случае, так как все три объекта `std::string` хранятся в стеке, они освобождаются автоматически после выхода из функции.

Следовательно, работа с короткими строками более быстрая из-за меньшего обращения к [куче](#).

Код на GCC даже проще (из-за того, что в GCC, как мы уже видели, не реализована возможность хранить короткую строку прямо в структуре):

Листинг 52.24: GCC 4.8.1

```

.LC0:
.string "Hello, "
.LC1:
.string "world!\n"
main:
push ebp
mov  ebp, esp
push edi
push esi
push ebx
and  esp, -16
sub  esp, 32

```

52.4. STL

```
lea ebx, [esp+28]
lea edi, [esp+20]
mov DWORD PTR [esp+8], ebx
lea esi, [esp+24]
mov DWORD PTR [esp+4], OFFSET FLAT:.LC0
mov DWORD PTR [esp], edi

call _ZNSsC1EPKcRKSaIcE

mov DWORD PTR [esp+8], ebx
mov DWORD PTR [esp+4], OFFSET FLAT:.LC1
mov DWORD PTR [esp], esi

call _ZNSsC1EPKcRKSaIcE

mov DWORD PTR [esp+4], edi
mov DWORD PTR [esp], ebx

call _ZNSsC1ERKSs

mov DWORD PTR [esp+4], esi
mov DWORD PTR [esp], ebx

call _ZNSs6appendERKSs

; вставленный код метода (inlined) c_str():
mov eax, DWORD PTR [esp+28]
mov DWORD PTR [esp], eax

call puts

mov eax, DWORD PTR [esp+28]
lea ebx, [esp+19]
mov DWORD PTR [esp+4], ebx
sub eax, 12
mov DWORD PTR [esp], eax
call _ZNSs4_Rep10_M_disposeERKSaIcE
mov eax, DWORD PTR [esp+24]
mov DWORD PTR [esp+4], ebx
sub eax, 12
mov DWORD PTR [esp], eax
call _ZNSs4_Rep10_M_disposeERKSaIcE
mov eax, DWORD PTR [esp+20]
mov DWORD PTR [esp+4], ebx
sub eax, 12
mov DWORD PTR [esp], eax
call _ZNSs4_Rep10_M_disposeERKSaIcE
lea esp, [ebp-12]
xor eax, eax
pop ebx
pop esi
pop edi
pop ebp
ret
```

Можно заметить, что в деструкторы передается не указатель на объект, а указатель на место за 12 байт (или 3 слова) перед ним, то есть, на настоящее начало структуры .

std::string как глобальная переменная

Опытные программисты на Си++ знают что глобальные переменные [STL](#)¹⁰-типов вполне можно объявлять.

Да, действительно:

```
#include <stdio.h>
#include <string>
```

¹⁰(Си++) Standard Template Library: [52.4](#) (стр. 556)

52.4. STL

```
std::string s="a string";

int main()
{
    printf ("%s\n", s.c_str());
};
```

Но как и где будет вызываться конструктор `std::string`?

На самом деле, эта переменная будет инициализирована даже перед началом `main()`.

Листинг 52.25: MSVC 2012: здесь конструируется глобальная переменная, а также регистрируется её деструктор

```
??_Es@YAXXZ PROC
    push 8
    push OFFSET $SG39512 ; 'a string'
    mov ecx, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBDI@Z ; std::↵
    ↵ basic_string<char,std::char_traits<char>,std::allocator<char> >::assign
    push OFFSET ??_Fs@YAXXZ ; `dynamic atexit destructor for 's'`
    call _atexit
    pop ecx
    ret 0
??_Es@YAXXZ ENDP
```

Листинг 52.26: MSVC 2012: здесь глобальная переменная используется в `main()`

```
$SG39512 DB 'a string', 00H
$SG39519 DB '%s', 0aH, 00H

_main PROC
    cmp DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 16
    mov eax, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    cmovae eax, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A
    push eax
    push OFFSET $SG39519 ; '%s'
    call _printf
    add esp, 8
    xor eax, eax
    ret 0
_main ENDP
```

Листинг 52.27: MSVC 2012: эта функция-деструктор вызывается перед выходом

```
??_Fs@YAXXZ PROC
    push ecx
    cmp DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 16
    jb SHORT $LN23@dynamic
    push esi
    mov esi, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A
    lea ecx, DWORD PTR $T2[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
    push OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    lea ecx, DWORD PTR $T2[esp+12]
    call ??$destroy@PAD@?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
    lea ecx, DWORD PTR $T1[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ
    push esi
    call ??3@YAXPAX@Z ; operator delete
    add esp, 4
    pop esi
$LN23@dynamic:
    mov DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 15
    mov DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+16, 0
    mov BYTE PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A, 0
    pop ecx
    ret 0
??_Fs@YAXXZ ENDP
```

52.4. STL

В реальности, из [CRT](#), еще до вызова `main()`, вызывается специальная функция, в которой перечислены все конструкторы подобных переменных. Более того: при помощи `atexit()` регистрируется функция, которая будет вызвана в конце работы программы: в этой функции компилятор собирает вызовы деструкторов всех подобных глобальных переменных.

GCC работает похожим образом:

Листинг 52.28: GCC 4.8.1

```
main:
    push ebp
    mov  ebp, esp
    and  esp, -16
    sub  esp, 16
    mov  eax, DWORD PTR s
    mov  DWORD PTR [esp], eax
    call puts
    xor  eax, eax
    leave
    ret
.LC0:
    .string "a string"
_GLOBAL__sub_I_s:
    sub  esp, 44
    lea  eax, [esp+31]
    mov  DWORD PTR [esp+8], eax
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], OFFSET FLAT:s
    call _ZN5sC1EPKcRKSaIcE
    mov  DWORD PTR [esp+8], OFFSET FLAT:__dso_handle
    mov  DWORD PTR [esp+4], OFFSET FLAT:s
    mov  DWORD PTR [esp], OFFSET FLAT:_ZN5sD1Ev
    call __cxa_atexit
    add  esp, 44
    ret
.LFE645:
    .size _GLOBAL__sub_I_s, .-_GLOBAL__sub_I_s
    .section .init_array,"aw"
    .align 4
    .long _GLOBAL__sub_I_s
    .globl s
    .bss
    .align 4
    .type s, @object
    .size s, 4
s:
    .zero 4
    .hidden __dso_handle
```

Но он не выделяет отдельной функции в которой будут собраны деструкторы: каждый деструктор передается в `atexit()` по одному.

52.4.2. `std::list`

Хорошо известный всем двусвязный список: каждый элемент имеет два указателя, на следующий и на предыдущий элементы.

Это означает что расход памяти увеличивается на 2 [слова](#) на каждый элемент (8 байт в 32-битной среде или 16 байт в 64-битной).

STL в Си++ просто добавляет указатели «next» и «previous» к той вашей структуре, которую вы желаете объединить в список.

Попробуем разобраться с примером в котором простая структура из двух переменных, мы объединим её в список.

Хотя и стандарт Си++ не указывает, как он должен быть реализован, реализации MSVC и GCC простые и похожи друг на друга, так что этот исходный код для обоих:

```
#include <stdio.h>
#include <list>
#include <iostream>
```



```

struct a
{
    int x;
    int y;
};

struct List_node
{
    struct List_node* _Next;
    struct List_node* _Prev;
    int x;
    int y;
};

void dump_List_node (struct List_node *n)
{
    printf ("ptr=0x%p _Next=0x%p _Prev=0x%p x=%d y=%d\n",
           n, n->_Next, n->_Prev, n->x, n->y);
};

void dump_List_vals (struct List_node* n)
{
    struct List_node* current=n;

    for (;;)
    {
        dump_List_node (current);
        current=current->_Next;
        if (current==n) // end
            break;
    };
};

void dump_List_val (unsigned int *a)
{
#ifdef _MSC_VER
    // в реализации GCC нет поля "size"
    printf ("_Myhead=0x%p, _Mysize=%d\n", a[0], a[1]);
#endif
    dump_List_vals ((struct List_node*)a[0]);
};

int main()
{
    std::list<struct a> l;

    printf ("* empty list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    struct a t1;
    t1.x=1;
    t1.y=2;
    l.push_front (t1);
    t1.x=3;
    t1.y=4;
    l.push_front (t1);
    t1.x=5;
    t1.y=6;
    l.push_back (t1);

    printf ("* 3-elements list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    std::list<struct a>::iterator tmp;
    printf ("node at .begin:\n");
    tmp=l.begin();
    dump_List_node ((struct List_node *)*(void**)&tmp);
    printf ("node at .end:\n");
    tmp=l.end();
}

```

52.4. STL

```
dump_List_node ((struct List_node *)*(void**)&tmp);

printf ("* let's count from the begin:\n");
std::list<struct a>::iterator it=l.begin();
printf ("1st element: %d %d\n", (*it).x, (*it).y);
it++;
printf ("2nd element: %d %d\n", (*it).x, (*it).y);
it++;
printf ("3rd element: %d %d\n", (*it).x, (*it).y);
it++;
printf ("element at .end(): %d %d\n", (*it).x, (*it).y);

printf ("* let's count from the end:\n");
std::list<struct a>::iterator it2=l.end();
printf ("element at .end(): %d %d\n", (*it2).x, (*it2).y);
it2--;
printf ("3rd element: %d %d\n", (*it2).x, (*it2).y);
it2--;
printf ("2nd element: %d %d\n", (*it2).x, (*it2).y);
it2--;
printf ("1st element: %d %d\n", (*it2).x, (*it2).y);

printf ("removing last element...\n");
l.pop_back();
dump_List_val((unsigned int*)(void*)&l);
};
```

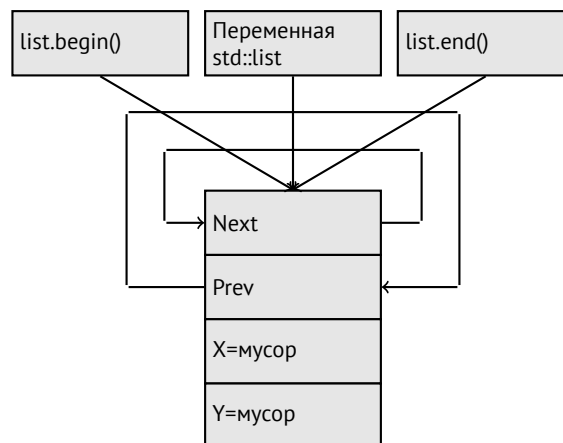
GCC

Начнем с GCC.

При запуске увидим длинный вывод, будем разбирать его по частям .

```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
```

Видим пустой список. Не смотря на то что он пуст, имеется один элемент с мусором ([АКА](#) узел-пустышка (*dummy node*)) в переменных *x* и *y*. Оба указателя «next» и «prev» указывают на себя :



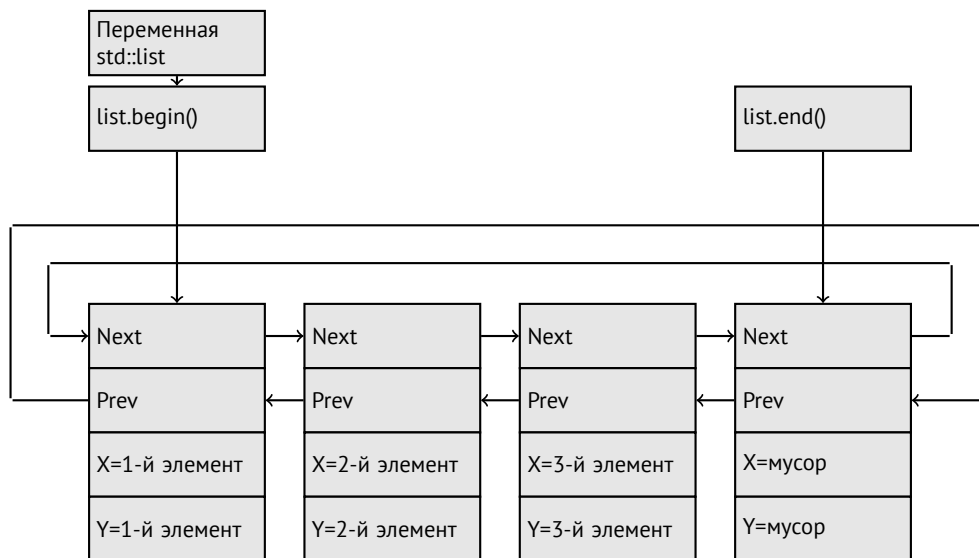
Это тот момент, когда итераторы .begin и .end равны друг другу .

Вставим 3 элемента и список в памяти будет представлен так:

```
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

Последний элемент всё еще на 0x0028fe90, он не будет передвинут куда-либо до самого уничтожения списка. Он все еще содержит случайный мусор в полях *x* и *y* (5 и 6) . Случайно совпало так, что эти значения точно такие же, как и в последнем элементе, но это не значит, что они имеют какое-то значение.

Вот как эти 3 элемента хранятся в памяти:



Переменная *l* всегда указывает на первый элемент.

Итераторы `.begin()` и `.end()` это не переменные, а функции, возвращающие указатели на соответствующие узлы.

Иметь элемент-пустышку (*dummy node* или *sentinel node*) это очень популярная практика в реализации двусвязных списков. Без него, многие операции были бы сложнее, и, следовательно, медленнее.

Итератор на самом деле это просто указатель на элемент. `list.begin()` и `list.end()` просто возвращают указатели.

```
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

Тот факт, что что последний элемент имеет указатель на первый и первый имеет указатель на последний, напоминает нам циркулярный список.

Это очень помогает: если иметь указатель только на первый элемент, т.е. тот что в переменной *l*, очень легко получить указатель на последний элемент, без необходимости обходить все элементы списка. Вставка элемента в конец списка также быстра благодаря этой особенности.

`operator--` и `operator++` просто выставляют текущее значение итератора на `current_node->prev` или `current_node->next`.

Обратные итераторы (`.rbegin`, `.rend`) работают точно так же, только наоборот.

`operator*` на итераторе просто возвращает указатель на место в структуре, где начинается пользовательская структура, т.е. указатель на самый первый элемент структуры (*x*).

Вставка в список и удаление очень просты: просто выделите новый элемент (или освободите) и исправьте все указатели так, чтобы они были верны.

Вот почему итератор может стать недействительным после удаления элемента: он может всё еще указывать на уже освобожденный элемент. Это также называется *dangling pointer*. И конечно же, информация из освобожденного элемента, на который указывает итератор, не может использоваться более.

В реализации GCC (по крайней мере 4.8.1) не сохраняется текущая длина списка: это выливается в медленный метод `.size()`: он должен пройти по всему списку считая элементы, просто потому что нет другого способа получить эту информацию. Это означает что эта операция $O(n)$, т.е. она работает тем медленнее, чем больше элементов в списке.

Листинг 52.29: Оптимизирующий GCC 4.8.1 -fno-inline-small-functions

```
main proc near
    push ebp
    mov  ebp, esp
    push esi
    push ebx
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  ebx, [esp+10h]
    mov  dword ptr [esp], offset s ; "* empty list:"
    mov  [esp+10h], ebx
    mov  [esp+14h], ebx
```

```

call puts
mov [esp], ebx
call _Z13dump_List_valPj ; dump_List_val(uint *)
lea esi, [esp+18h]
mov [esp+4], esi
mov [esp], ebx
mov dword ptr [esp+18h], 1 ; X нового элемента
mov dword ptr [esp+1Ch], 2 ; Y нового элемента
call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,std::allocator<a>>::push_front(a const&
↳ &)
mov [esp+4], esi
mov [esp], ebx
mov dword ptr [esp+18h], 3 ; X нового элемента
mov dword ptr [esp+1Ch], 4 ; Y нового элемента
call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,std::allocator<a>>::push_front(a const&
↳ &)
mov dword ptr [esp], 10h
mov dword ptr [esp+18h], 5 ; X нового элемента
mov dword ptr [esp+1Ch], 6 ; Y нового элемента
call _Znwj ; operator new(uint)
cmp eax, 0FFFFFFF8h
jz short loc_80002A6
mov ecx, [esp+1Ch]
mov edx, [esp+18h]
mov [eax+0Ch], ecx
mov [eax+8], edx

```

loc_80002A6: ; CODE XREF: main+86

```

mov [esp+4], ebx
mov [esp], eax
call _ZNSt8__detail15_List_node_base7_M_hookEPS0_ ; std::__detail::_List_node_base::_M_hook(std::
↳ __detail::_List_node_base*)
mov dword ptr [esp], offset a3ElementsList ; "* 3-elements list:"
call puts
mov [esp], ebx
call _Z13dump_List_valPj ; dump_List_val(uint *)
mov dword ptr [esp], offset aNodeAt_begin ; "node at .begin:"
call puts
mov eax, [esp+10h]
mov [esp], eax
call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
mov dword ptr [esp], offset aNodeAt_end ; "node at .end:"
call puts
mov [esp], ebx
call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
mov dword ptr [esp], offset aLetSCountFromT ; "* let's count from the begin:"
call puts
mov esi, [esp+10h]
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi] ; operator++: get ->next pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi] ; operator++: get ->next pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax

```

```

call __printf_chk
mov  eax, [esi] ; operator++: get ->next pointer
mov  edx, [eax+0Ch]
mov  [esp+0Ch], edx
mov  eax, [eax+8]
mov  dword ptr [esp+4], offset aElementAt_endD ; "element at .end(): %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call __printf_chk
mov  dword ptr [esp], offset aLetSCountFro_0 ; "* let's count from the end:"
call puts
mov  eax, [esp+1Ch]
mov  dword ptr [esp+4], offset aElementAt_endD ; "element at .end(): %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+0Ch], eax
mov  eax, [esp+18h]
mov  [esp+8], eax
call __printf_chk
mov  esi, [esp+14h]
mov  eax, [esi+0Ch]
mov  [esp+0Ch], eax
mov  eax, [esi+8]
mov  dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call __printf_chk
mov  esi, [esi+4] ; operator--: get ->prev pointer
mov  eax, [esi+0Ch]
mov  [esp+0Ch], eax
mov  eax, [esi+8]
mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call __printf_chk
mov  eax, [esi+4] ; operator--: get ->prev pointer
mov  edx, [eax+0Ch]
mov  [esp+0Ch], edx
mov  eax, [eax+8]
mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
mov  dword ptr [esp], 1
mov  [esp+8], eax
call __printf_chk
mov  dword ptr [esp], offset aRemovingLastEl ; "removing last element..."
call puts
mov  esi, [esp+14h]
mov  [esp], esi
call _ZNSt8__detail15_List_node_base9_M_unhookEv ; std::__detail::_List_node_base::_M_unhook(void)
mov  [esp], esi ; void *
call _ZdlPv ; operator delete(void *)
mov  [esp], ebx
call _Z13dump_List_valPj ; dump_List_val(uint *)
mov  [esp], ebx
call _ZNSt10_List_baseI1aSaISO_EE8_M_clearEv ; std::_List_base<a,std::allocator<a>>::_M_clear(void)
↳ )
lea  esp, [ebp-8]
xor  eax, eax
pop  ebx
pop  esi
pop  ebp
retn
main endp

```

Листинг 52.30: Весь вывод

```

* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6

```

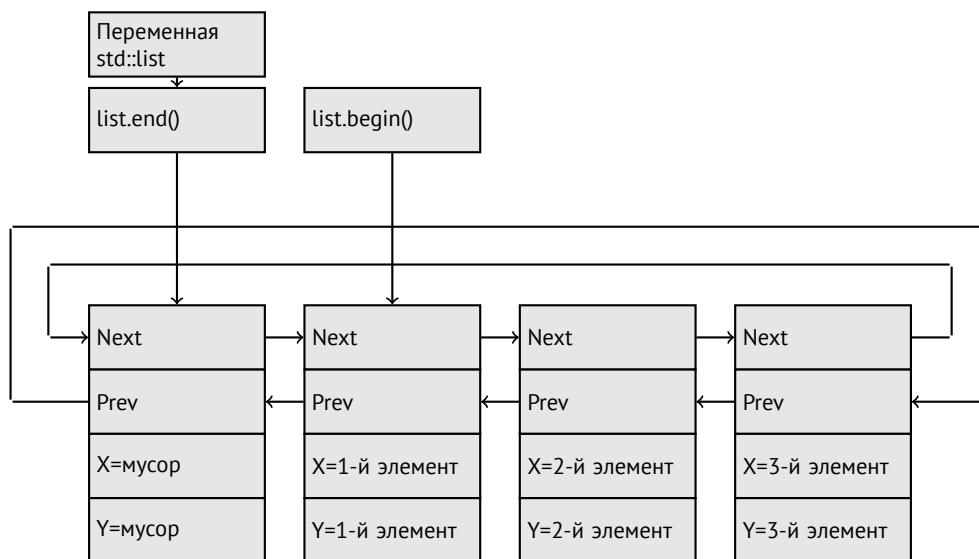
52.4. STL

```
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
* let's count from the begin:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 5 6
* let's count from the end:
element at .end(): 5 6
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x0028fe90 _Prev=0x000349a0 x=1 y=2
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034988 x=5 y=6
```

MSVC

Реализация MSVC (2012) точно такая же, только еще и сохраняет текущий размер списка. Это означает что метод `.size()` очень быстр ($O(1)$): просто прочитать одно значение из памяти. С другой стороны, переменная хранящая размер должна корректироваться при каждой вставке/удалении.

Реализация MSVC также немного отлична в смысле расстановки элементов:



У GCC его элемент-пустышка в самом конце списка, а у MSVC в самом начале.

Листинг 52.31: Оптимизирующий MSVC 2012 /Fa2.asm /GS- /Ob1

```
_l$ = -16 ; size = 8
_t1$ = -8 ; size = 8
_main PROC
    sub esp, 16
    push ebx
    push esi
    push edi
    push 0
    push 0
    lea ecx, DWORD PTR _l$[esp+36]
    mov DWORD PTR _l$[esp+40], 0
    ; выделить первый "мусорный" элемент
    call ?_Buynode0@?$_List_alloc@$0A@U?$_List_base_types@Ua@@V?
    ↪ $allocator@Ua@@@std@@@std@@@std@@@QAEPAU?$_List_node@Ua@@@PAX@2@PAU32@0@Z ; std::_List_alloc<0,std
    ↪ ::_List_base_types<a,std::allocator<a> >::_Buynode0
    mov edi, DWORD PTR __imp__printf
    mov ebx, eax
```

```

push OFFSET $SG40685 ; '* empty list:'
mov  DWORD PTR _l$[esp+32], ebx
call edi ; printf
lea  eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
mov  esi, DWORD PTR [ebx]
add  esp, 8
lea  eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [esi+4]
lea  ecx, DWORD PTR _l$[esp+36]
push esi
mov  DWORD PTR _t1$[esp+40], 1 ; данные для нового узла
mov  DWORD PTR _t1$[esp+44], 2 ; данные для нового узла
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@std@@QAEPAU?↵
↵ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,std::allocator<a> >::_Buynode<a const ↵
↵ &>
mov  DWORD PTR [esi+4], eax
mov  ecx, DWORD PTR [eax+4]
mov  DWORD PTR _t1$[esp+28], 3 ; данные для нового узла
mov  DWORD PTR [ecx], eax
mov  esi, DWORD PTR [ebx]
lea  eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [esi+4]
lea  ecx, DWORD PTR _l$[esp+36]
push esi
mov  DWORD PTR _t1$[esp+44], 4 ; данные для нового узла
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@std@@QAEPAU?↵
↵ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,std::allocator<a> >::_Buynode<a const ↵
↵ &>
mov  DWORD PTR [esi+4], eax
mov  ecx, DWORD PTR [eax+4]
mov  DWORD PTR _t1$[esp+28], 5 ; данные для нового узла
mov  DWORD PTR [ecx], eax
lea  eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [ebx+4]
lea  ecx, DWORD PTR _l$[esp+36]
push ebx
mov  DWORD PTR _t1$[esp+44], 6 ; данные для нового узла
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@std@@QAEPAU?↵
↵ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,std::allocator<a> >::_Buynode<a const ↵
↵ &>
mov  DWORD PTR [ebx+4], eax
mov  ecx, DWORD PTR [eax+4]
push OFFSET $SG40689 ; '* 3-elements list:'
mov  DWORD PTR _l$[esp+36], 3
mov  DWORD PTR [ecx], eax
call edi ; printf
lea  eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
push OFFSET $SG40831 ; 'node at .begin:'
call edi ; printf
push DWORD PTR [ebx] ; взять поле следующего узла, на который указывает l
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40835 ; 'node at .end:'
call edi ; printf
push ebx ; pointer to the node $l$ variable points to!
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40839 ; '* let's count from the begin:'
call edi ; printf
mov  esi, DWORD PTR [ebx] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]

```

```

push OFFSET $SG40846 ; '1st element: %d %d'
call edi ; printf
mov esi, DWORD PTR [esi] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40848 ; '2nd element: %d %d'
call edi ; printf
mov esi, DWORD PTR [esi] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40850 ; '3rd element: %d %d'
call edi ; printf
mov eax, DWORD PTR [esi] ; operator++: get ->next pointer
add esp, 64
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
push OFFSET $SG40852 ; 'element at .end(): %d %d'
call edi ; printf
push OFFSET $SG40853 ; '* let's count from the end:'
call edi ; printf
push DWORD PTR [ebx+12] ; использовать поля x и y того узла, на который указывает переменная l
push DWORD PTR [ebx+8]
push OFFSET $SG40860 ; 'element at .end(): %d %d'
call edi ; printf
mov esi, DWORD PTR [ebx+4] ; operator--: get ->prev pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40862 ; '3rd element: %d %d'
call edi ; printf
mov esi, DWORD PTR [esi+4] ; operator--: get ->prev pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40864 ; '2nd element: %d %d'
call edi ; printf
mov eax, DWORD PTR [esi+4] ; operator--: get ->prev pointer
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
push OFFSET $SG40866 ; '1st element: %d %d'
call edi ; printf
add esp, 64
push OFFSET $SG40867 ; 'removing last element...'
call edi ; printf
mov edx, DWORD PTR [ebx+4]
add esp, 4

; prev=next?
; это единственный элемент, "мусор"?
; если да, не удаляем его!
cmp edx, ebx
je SHORT $LN349@main
mov ecx, DWORD PTR [edx+4]
mov eax, DWORD PTR [edx]
mov DWORD PTR [ecx], eax
mov ecx, DWORD PTR [edx]
mov eax, DWORD PTR [edx+4]
push edx
mov DWORD PTR [ecx+4], eax
call ??3@YAXPAX@Z ; operator delete
add esp, 4
mov DWORD PTR _l$[esp+32], 2
$LN349@main:
lea eax, DWORD PTR _l$[esp+28]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
mov eax, DWORD PTR [ebx]
add esp, 4
mov DWORD PTR [ebx], ebx
mov DWORD PTR [ebx+4], ebx
cmp eax, ebx
je SHORT $LN412@main

```


52.4. STL

```
$LL414@main:
    mov     esi, DWORD PTR [eax]
    push    eax
    call    ??3@YAXPAX@Z ; operator delete
    add     esp, 4
    mov     eax, esi
    cmp     esi, ebx
    jne     SHORT $LL414@main
$LN412@main:
    push    ebx
    call    ??3@YAXPAX@Z ; operator delete
    add     esp, 4
    xor     eax, eax
    pop     edi
    pop     esi
    pop     ebx
    add     esp, 16
    ret     0
_main ENDP
```

В отличие от GCC, код MSVC выделяет элемент-пустышку в самом начале функции при помощи функции «Buynode», она также используется и во время выделения остальных элементов (код GCC выделяет самый первый элемент в локальном стеке).

Листинг 52.32: Весь вывод

```
* empty list:
_Myhead=0x003CC258, _Mysize=0
ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y=4522072
* 3-elements list:
_Myhead=0x003CC258, _Mysize=3
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC2A0 _Prev=0x003CC288 x=1 y=2
ptr=0x003CC2A0 _Next=0x003CC258 _Prev=0x003CC270 x=5 y=6
node at .begin:
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
node at .end:
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
* let's count from the begin:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 6226002 4522072
* let's count from the end:
element at .end(): 6226002 4522072
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
_Myhead=0x003CC258, _Mysize=2
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC270 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC258 _Prev=0x003CC288 x=1 y=2
```

C++11 std::forward_list

Это то же самое что и std::list, но только односвязный список, т.е. имеющий только поле «next» в каждом элементе. Таким образом расход памяти меньше, но возможности идти по списку назад здесь нет.

52.4.3. std::vector

Мы бы назвали std::vector «безопасной оболочкой» (wrapper) PODT¹¹ массива в Си. Изнутри он очень похож на std::string (52.4.1 (стр. 556)): он имеет указатель на выделенный буфер, указатель на конец массива и указатель на конец выделенного буфера.

¹¹(Си++) Plain Old Data Type

52.4. STL

Элементы массива просто лежат в памяти впритык друг к другу, так же, как и в обычном массиве (19 (стр. 264)). В C++11 появился метод `.data()` возвращающий указатель на этот буфер, это похоже на `.c_str()` в `std::string`.

Выделенный буфер в *куче* может быть больше чем сам массив.

Реализации MSVC и GCC почти одинаковые, отличаются только имена полей в структуре ¹², так что здесь один исходник работающий для обоих компиляторов. И снова здесь Си-подобный код для вывода структуры `std::vector`:

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
    // MSVC names:
    int *Myfirst;
    int *Mylast;
    int *Myend;

    // структура в GCC такая же, а имена там: _M_start, _M_finish, _M_end_of_storage
};

void dump(struct vector_of_ints *in)
{
    printf ("_Myfirst=%p, _Mylast=%p, _Myend=%p\n", in->Myfirst, in->Mylast, in->Myend);
    size_t size=(in->Mylast-in->Myfirst);
    size_t capacity=(in->Myend-in->Myfirst);
    printf ("size=%d, capacity=%d\n", size, capacity);
    for (size_t i=0; i<size; i++)
        printf ("element %d: %d\n", i, in->Myfirst[i]);
};

int main()
{
    std::vector<int> c;
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(1);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(2);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(3);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(4);
    dump ((struct vector_of_ints*)(void*)&c);
    c.reserve (6);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(5);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(6);
    dump ((struct vector_of_ints*)(void*)&c);
    printf ("%d\n", c.at(5)); // с проверкой границ
    printf ("%d\n", c[8]); // operator[], без проверки границ
};
```

Примерный вывод программы скомпилированной в MSVC:

```
_Myfirst=00000000, _Mylast=00000000, _Myend=00000000
size=0, capacity=0
_Myfirst=0051CF48, _Mylast=0051CF4C, _Myend=0051CF4C
size=1, capacity=1
element 0: 1
_Myfirst=0051CF58, _Mylast=0051CF60, _Myend=0051CF60
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0051C278, _Mylast=0051C284, _Myend=0051C284
size=3, capacity=3
element 0: 1
```

¹²внутренности GCC: <http://go.yurichev.com/17086>

52.4. STL

```
element 1: 2
element 2: 3
_Myfirst=0051C290, _Mylast=0051C2A0, _Myend=0051C2A0
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B190, _Myend=0051B198
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B194, _Myend=0051B198
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0051B180, _Mylast=0051B198, _Myend=0051B198
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
6619158
```

Как можно заметить, выделенного буфера в самом начале функции `main()` пока нет. После первого вызова `push_back()` буфер выделяется. И далее, после каждого вызова `push_back()` и длина массива и вместимость буфера (*capacity*) увеличиваются. Но адрес буфера также меняется, потому что вызов функции `push_back()` перевыделяет буфер в куче каждый раз. Это дорогая операция, вот почему очень важно предсказать размер будущего массива и зарезервировать место для него при помощи метода `.reserve()`. Самое последнее число — это мусор: там нет элементов массива в этом месте, вот откуда это случайное число. Это иллюстрация того факта что метод `operator[]` в `std::vector` не проверяет индекс на правильность. Более медленный метод `.at()` с другой стороны, проверяет, и подкидывает исключение `std::out_of_range` в случае ошибки.

Давайте посмотрим код:

Листинг 52.33: MSVC 2012 /GS- /Ob1

```
$SG52650 DB '%d', 0aH, 00H
$SG52651 DB '%d', 0aH, 00H

_this$ = -4 ; size = 4
__Pos$ = 8 ; size = 4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z PROC ; std::vector<int,std::allocator<int> >::at, ↵
    ↳ COMDAT
; _this$ = ecx
    push ebp
    mov ebp, esp
    push ecx
    mov DWORD PTR _this$[ebp], ecx
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR _this$[ebp]
    mov edx, DWORD PTR [eax+4]
    sub edx, DWORD PTR [ecx]
    sar edx, 2
    cmp edx, DWORD PTR __Pos$[ebp]
    ja SHORT $LN1@at
    push OFFSET ??_C@_OBM@NMJKDPPO@invalid?5vector?$DMT?$D0?5subscript?$AA@
    call DWORD PTR __imp_?_Xout_of_range@std@YAXPBD@Z
$LN1@at:
    mov eax, DWORD PTR _this$[ebp]
```

```

mov ecx, DWORD PTR [eax]
mov edx, DWORD PTR __Pos$[ebp]
lea eax, DWORD PTR [ecx+edx*4]
$LN3@at:
mov esp, ebp
pop ebp
ret 4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ENDP ; std::vector<int,std::allocator<int> >::at

_c$ = -36 ; size = 12
$T1 = -24 ; size = 4
$T2 = -20 ; size = 4
$T3 = -16 ; size = 4
$T4 = -12 ; size = 4
$T5 = -8 ; size = 4
$T6 = -4 ; size = 4
_main PROC
push ebp
mov ebp, esp
sub esp, 36
mov DWORD PTR _c$[ebp], 0 ; Myfirst
mov DWORD PTR _c$[ebp+4], 0 ; Mylast
mov DWORD PTR _c$[ebp+8], 0 ; Myend
lea eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
mov DWORD PTR $T6[ebp], 1
lea ecx, DWORD PTR $T6[ebp]
push ecx
lea ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAE$$QAH@Z ; std::vector<int,std::allocator<
↳ int> >::push_back
lea edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
mov DWORD PTR $T5[ebp], 2
lea eax, DWORD PTR $T5[ebp]
push eax
lea ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAE$$QAH@Z ; std::vector<int,std::allocator<
↳ int> >::push_back
lea ecx, DWORD PTR _c$[ebp]
push ecx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
mov DWORD PTR $T4[ebp], 3
lea edx, DWORD PTR $T4[ebp]
push edx
lea ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAE$$QAH@Z ; std::vector<int,std::allocator<
↳ int> >::push_back
lea eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
mov DWORD PTR $T3[ebp], 4
lea ecx, DWORD PTR $T3[ebp]
push ecx
lea ecx, DWORD PTR _c$[ebp]
call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAE$$QAH@Z ; std::vector<int,std::allocator<
↳ int> >::push_back
lea edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
push 6
lea ecx, DWORD PTR _c$[ebp]
call ?reserve@?$vector@HV?$allocator@H@std@@@std@@QAEXI@Z ; std::vector<int,std::allocator<int> >

```

52.4. STL

```

↳ >::reserve
lea  eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T2[ebp], 5
lea  ecx, DWORD PTR $T2[ebp]
push ecx
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@$vector@HV?$allocator@H@std@@@std@@QAEX$QAH@Z ; std::vector<int,std::allocator<
↳ int> >::push_back
lea  edx, DWORD PTR _c$[ebp]
push edx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
mov  DWORD PTR $T1[ebp], 6
lea  eax, DWORD PTR $T1[ebp]
push eax
lea  ecx, DWORD PTR _c$[ebp]
call ?push_back@$vector@HV?$allocator@H@std@@@std@@QAEX$QAH@Z ; std::vector<int,std::allocator<
↳ int> >::push_back
lea  ecx, DWORD PTR _c$[ebp]
push ecx
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add  esp, 4
push 5
lea  ecx, DWORD PTR _c$[ebp]
call ?at?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ; std::vector<int,std::allocator<int> >::at
mov  edx, DWORD PTR [eax]
push edx
push OFFSET $SG52650 ; '%d'
call DWORD PTR __imp__printf
add  esp, 8
mov  eax, 8
shl  eax, 2
mov  ecx, DWORD PTR _c$[ebp]
mov  edx, DWORD PTR [ecx+eax]
push edx
push OFFSET $SG52651 ; '%d'
call DWORD PTR __imp__printf
add  esp, 8
lea  ecx, DWORD PTR _c$[ebp]
call ?_Tidy@$vector@HV?$allocator@H@std@@@std@@IAEXXZ ; std::vector<int,std::allocator<int> >::~
↳ _Tidy
xor  eax, eax
mov  esp, ebp
pop  ebp
ret  0
_main ENDP

```

Мы видим, как метод `.at()` проверяет границы и подкидывает исключение в случае ошибки. Число, которое выводит последний вызов `printf()` берется из памяти, без всяких проверок.

Читатель может спросить, почему бы не использовать переменные «size» и «capacity», как это сделано в `std::string`. Вероятно, это для более быстрой проверки границ.

Код генерируемый GCC почти такой же, в целом, но метод `.at()` вставлен прямо в код:

Листинг 52.34: GCC 4.8.1 -fno-inline-small-functions -O1

```

main proc near
    push ebp
    mov  ebp, esp
    push edi
    push esi
    push ebx
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    mov  dword ptr [esp+14h], 0
    mov  dword ptr [esp+18h], 0

```

```

mov     dword ptr [esp+1Ch], 0
lea     eax, [esp+14h]
mov     [esp], eax
call    _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov     dword ptr [esp+10h], 1
lea     eax, [esp+10h]
mov     [esp+4], eax
lea     eax, [esp+14h]
mov     [esp], eax
call    _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(int ↗
↵ const&)
lea     eax, [esp+14h]
mov     [esp], eax
call    _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov     dword ptr [esp+10h], 2
lea     eax, [esp+10h]
mov     [esp+4], eax
lea     eax, [esp+14h]
mov     [esp], eax
call    _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(int ↗
↵ const&)
lea     eax, [esp+14h]
mov     [esp], eax
call    _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov     dword ptr [esp+10h], 3
lea     eax, [esp+10h]
mov     [esp+4], eax
lea     eax, [esp+14h]
mov     [esp], eax
call    _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(int ↗
↵ const&)
lea     eax, [esp+14h]
mov     [esp], eax
call    _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov     dword ptr [esp+10h], 4
lea     eax, [esp+10h]
mov     [esp+4], eax
lea     eax, [esp+14h]
mov     [esp], eax
call    _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(int ↗
↵ const&)
lea     eax, [esp+14h]
mov     [esp], eax
call    _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov     ebx, [esp+14h]
mov     eax, [esp+1Ch]
sub     eax, ebx
cmp     eax, 17h
ja      short loc_80001CF
mov     edi, [esp+18h]
sub     edi, ebx
sar     edi, 2
mov     dword ptr [esp], 18h
call    _Znwj ; operator new(uint)
mov     esi, eax
test    edi, edi
jz      short loc_80001AD
lea     eax, ds:0[edi*4]
mov     [esp+8], eax ; n
mov     [esp+4], ebx ; src
mov     [esp], esi ; dest
call    memmove

```

loc_80001AD: ; CODE XREF: main+F8

```

mov     eax, [esp+14h]
test    eax, eax
jz      short loc_80001BD
mov     [esp], eax ; void *
call    _ZdlPv ; operator delete(void *)

```

52.4. STL

```

loc_80001BD: ; CODE XREF: main+117
    mov [esp+14h], esi
    lea eax, [esi+edi*4]
    mov [esp+18h], eax
    add esi, 18h
    mov [esp+1Ch], esi

loc_80001CF: ; CODE XREF: main+DD
    lea eax, [esp+14h]
    mov [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov dword ptr [esp+10h], 5
    lea eax, [esp+10h]
    mov [esp+4], eax
    lea eax, [esp+14h]
    mov [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(int ↗
↘ const&)
    lea eax, [esp+14h]
    mov [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov dword ptr [esp+10h], 6
    lea eax, [esp+10h]
    mov [esp+4], eax
    lea eax, [esp+14h]
    mov [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(int ↗
↘ const&)
    lea eax, [esp+14h]
    mov [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov eax, [esp+14h]
    mov edx, [esp+18h]
    sub edx, eax
    cmp edx, 17h
    ja short loc_8000246
    mov dword ptr [esp], offset aVector_m_range ; "vector::_M_range_check"
    call _ZSt20__throw_out_of_rangePKc ; std::__throw_out_of_range(char const*)

loc_8000246: ; CODE XREF: main+19C
    mov eax, [eax+14h]
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aD ; "%d\n"
    mov dword ptr [esp], 1
    call __printf_chk
    mov eax, [esp+14h]
    mov eax, [eax+20h]
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aD ; "%d\n"
    mov dword ptr [esp], 1
    call __printf_chk
    mov eax, [esp+14h]
    test eax, eax
    jz short loc_80002AC
    mov [esp], eax ; void *
    call _ZdlPv ; operator delete(void *)
    jmp short loc_80002AC

    mov ebx, eax
    mov edx, [esp+14h]
    test edx, edx
    jz short loc_80002A4
    mov [esp], edx ; void *
    call _ZdlPv ; operator delete(void *)

loc_80002A4: ; CODE XREF: main+1FE
    mov [esp], ebx
    call _Unwind_Resume

```

52.4. STL

```
loc_80002AC: ; CODE XREF: main+1EA
              ; main+1F4
              mov     eax, 0
              lea     esp, [ebp-0Ch]
              pop     ebx
              pop     esi
              pop     edi
              pop     ebp

locret_80002B8: ; DATA XREF: .eh_frame:08000510
               ; .eh_frame:080005BC
               retn
main endp
```

Метод `.reserve()` точно так же вставлен прямо в код `main()`. Он вызывает `new()` если буфер слишком мал для нового массива, вызывает `memmove()` для копирования содержимого буфера, и вызывает `delete()` для освобождения старого буфера.

Посмотрим, что выводит программа будучи скомпилированная GCC :

```
_Myfirst=0x(nil), _Mylast=0x(nil), _Myend=0x(nil)
size=0, capacity=0
_Myfirst=0x8257008, _Mylast=0x825700c, _Myend=0x825700c
size=1, capacity=1
element 0: 1
_Myfirst=0x8257018, _Mylast=0x8257020, _Myend=0x8257020
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0x8257028, _Mylast=0x8257034, _Myend=0x8257038
size=3, capacity=4
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0x8257028, _Mylast=0x8257038, _Myend=0x8257038
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257050, _Myend=0x8257058
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257054, _Myend=0x8257058
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0x8257040, _Mylast=0x8257058, _Myend=0x8257058
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
0
```

Мы можем заметить, что буфер растёт иначе чем в MSVC.

При помощи простых экспериментов становится ясно, что в реализации MSVC буфер увеличивается на ~50% каждый раз, когда он должен был увеличен, а у GCC он увеличивается на 100% каждый раз, т.е. удваивается .

52.4.4. std::map и std::set

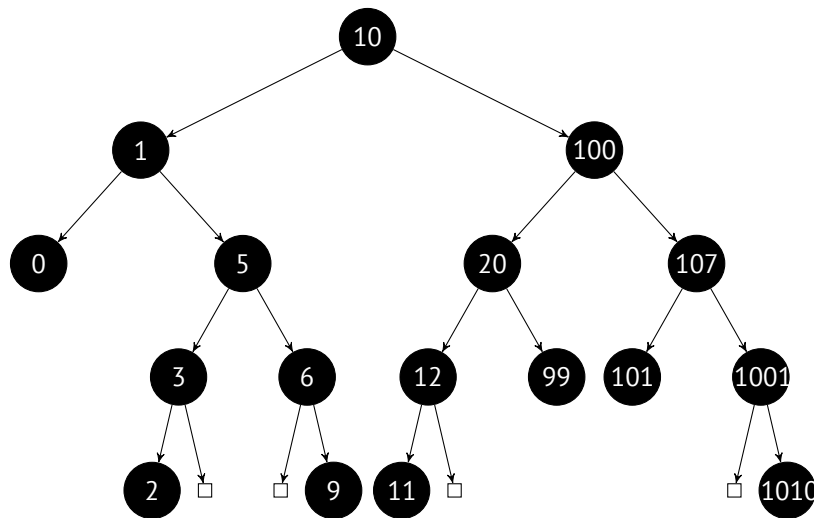
Двоичное дерево — это еще одна фундаментальная структура данных. Как следует из названия, это дерево, но у каждого узла максимум 2 связи с другими узлами. Каждый узел имеет ключ и/или значение.

Обычно, именно при помощи двоичных деревьев реализуются «словари» пар ключ-значение (АКА «ассоциативные массивы»).

Двоичные деревья имеют по крайней мере три важных свойства:

- Все ключи всегда хранятся в отсортированном виде.
- Могут храниться ключи любых типов. Алгоритмы для работы с двоичными деревьями не зависят от типа ключа, для работы им нужна только функция для сравнения ключей.
- Поиск заданного ключа относительно быстрый по сравнению со списками или массивами.

Очень простой пример: давайте сохраним вот эти числа в двоичном дереве : 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.



Все ключи меньше чем значение ключа узла, сохраняются по левой стороне. Все ключи больше чем значение ключа узла, сохраняются по правой стороне.

Таким образом, алгоритм для поиска нужного ключа прост: если искомое значение меньше чем значение текущего узла: двигаемся влево, если больше: двигаемся вправо, останавливаемся если они равны. Таким образом, алгоритм может искать числа, текстовые строки, и т.д., пользуясь только функцией сравнения ключей.

Все ключи имеют уникальные значения.

Учитывая это, нужно $\approx \log_2 n$ шагов для поиска ключа в сбалансированном дереве, содержащем n ключей. Это ≈ 10 шагов для ≈ 1000 ключей, или ≈ 13 шагов для ≈ 10000 ключей. Неплохо, но для этого дерево всегда должно быть сбалансировано: т.е. ключи должны быть равномерно распределены на всех ярусах. Операции вставки и удаления проводят дополнительную работу по обслуживанию дерева и сохранения его в сбалансированном состоянии.

Известно несколько популярных алгоритмов балансировки, включая AVL-деревья и красно-черные деревья. Последний дополняет узел значением «цвета» для упрощения балансировки, таким образом каждый узел может быть «красным» или «черным».

Реализации `std::map` и `std::set` обоих GCC и MSVC используют красно-черные деревья.

`std::set` содержит только ключи. `std::map` это «расширенная» версия `set`: здесь имеется еще и значение (value) на каждом узле.

MSVC

```

#include <map>
#include <set>
#include <string>
#include <iostream>

// структура не запакована!
struct tree_node
{

```

52.4. STL

```

struct tree_node *Left;
struct tree_node *Parent;
struct tree_node *Right;
char Color; // 0 - Red, 1 - Black
char Isnill;
//std::pair Myval;
unsigned int first; // называется Myval в std::set
const char *second; // отсутствует в std::set
};

struct tree_struct
{
    struct tree_node *Myhead;
    size_t Mysize;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse)
{
    printf ("ptr=0x%p Left=0x%p Parent=0x%p Right=0x%p Color=%d Isnill=%d\n",
            n, n->Left, n->Parent, n->Right, n->Color, n->Isnill);
    if (n->Isnill==0)
    {
        if (is_set)
            printf ("first=%d\n", n->first);
        else
            printf ("first=%d second=[%s]\n", n->first, n->second);
    }

    if (traverse)
    {
        if (n->Isnill==1)
            dump_tree_node (n->Parent, is_set, true);
        else
        {
            if (n->Left->Isnill==0)
                dump_tree_node (n->Left, is_set, true);
            if (n->Right->Isnill==0)
                dump_tree_node (n->Right, is_set, true);
        };
    };
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    if (is_set)
        printf ("%d\n", n->first);
    else
        printf ("%d [%s]\n", n->first, n->second);
    if (n->Left->Isnill==0)
    {
        printf (".*sL-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Left, is_set);
    };
    if (n->Right->Isnill==0)
    {
        printf (".*sR-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, Myhead=0x%p, Mysize=%d\n", m, m->Myhead, m->Mysize);
    dump_tree_node (m->Myhead, is_set, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->Myhead->Parent, is_set);
};

```

```

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";
    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *) (void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *) (void**)&it1, false, false);
    it1=m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *) (void**)&it1, false, false);

    // set

    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
    s.insert(1001);
    printf ("dumping s as set:\n");
    dump_map_and_set ((struct tree_struct *) (void*)&s, true);
    std::set<int>::iterator it2=s.begin();
    printf ("s.begin():\n");
    dump_tree_node ((struct tree_node *) (void**)&it2, true, false);
    it2=s.end();
    printf ("s.end():\n");
    dump_tree_node ((struct tree_node *) (void**)&it2, true, false);
};

```

Листинг 52.35: MSVC 2012

```

dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 Isnil=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0x005BB440 Color=1 Isnil=0
first=10 second=[ten]
ptr=0x005BB4C0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB520 Color=1 Isnil=0
first=1 second=[one]
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
ptr=0x005BB520 Left=0x005BB400 Parent=0x005BB4C0 Right=0x005BB4E0 Color=0 Isnil=0
first=5 second=[five]
ptr=0x005BB400 Left=0x005BB5A0 Parent=0x005BB520 Right=0x005BB3A0 Color=1 Isnil=0
first=3 second=[three]
ptr=0x005BB5A0 Left=0x005BB3A0 Parent=0x005BB400 Right=0x005BB3A0 Color=0 Isnil=0
first=2 second=[two]

```

52.4. STL

```
ptr=0x005BB4E0 Left=0x005BB3A0 Parent=0x005BB520 Right=0x005BB5C0 Color=1 Isnil=0
first=6 second=[six]
ptr=0x005BB5C0 Left=0x005BB3A0 Parent=0x005BB4E0 Right=0x005BB3A0 Color=0 Isnil=0
first=9 second=[nine]
ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB3C0 Right=0x005BB480 Color=1 Isnil=0
first=100 second=[one hundred]
ptr=0x005BB3E0 Left=0x005BB460 Parent=0x005BB440 Right=0x005BB500 Color=0 Isnil=0
first=20 second=[twenty]
ptr=0x005BB460 Left=0x005BB540 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 Isnil=0
first=12 second=[twelve]
ptr=0x005BB540 Left=0x005BB3A0 Parent=0x005BB460 Right=0x005BB3A0 Color=0 Isnil=0
first=11 second=[eleven]
ptr=0x005BB500 Left=0x005BB3A0 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 Isnil=0
first=99 second=[ninety-nine]
ptr=0x005BB480 Left=0x005BB420 Parent=0x005BB440 Right=0x005BB560 Color=0 Isnil=0
first=107 second=[one hundred seven]
ptr=0x005BB420 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB3A0 Color=1 Isnil=0
first=101 second=[one hundred one]
ptr=0x005BB560 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB580 Color=1 Isnil=0
first=1001 second=[one thousand one]
ptr=0x005BB580 Left=0x005BB3A0 Parent=0x005BB560 Right=0x005BB3A0 Color=0 Isnil=0
first=1010 second=[one thousand ten]
```

As a tree:

```
root----10 [ten]
    L-----1 [one]
        L-----0 [zero]
        R-----5 [five]
            L-----3 [three]
                L-----2 [two]
                R-----6 [six]
                    R-----9 [nine]
            R-----100 [one hundred]
                L-----20 [twenty]
                    L-----12 [twelve]
                        L-----11 [eleven]
                        R-----99 [ninety-nine]
                    R-----107 [one hundred seven]
                        L-----101 [one hundred one]
                        R-----1001 [one thousand one]
                            R-----1010 [one thousand ten]
```

m.begin():

```
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
```

m.end():

```
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 Isnil=1
```

dumping s as set:

```
ptr=0x0020FDFC, Myhead=0x005BB5E0, Mysize=6
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=1
ptr=0x005BB600 Left=0x005BB660 Parent=0x005BB5E0 Right=0x005BB620 Color=1 Isnil=0
first=123
ptr=0x005BB660 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB680 Color=1 Isnil=0
first=12
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=11
ptr=0x005BB680 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=100
ptr=0x005BB620 Left=0x005BB5E0 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=0
first=456
ptr=0x005BB6A0 Left=0x005BB5E0 Parent=0x005BB620 Right=0x005BB5E0 Color=0 Isnil=0
first=1001
```

As a tree:

```
root----123
    L-----12
        L-----11
        R-----100
    R-----456
        R-----1001
```

s.begin():

```
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
```

52.4. STL

```
first=11
s.end():
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnll=1
```

Структура не запакована, так что оба значения типа *char* занимают по 4 байта .

В `std::map`, `first` и `second` могут быть представлены как одно значение типа `std::pair`. `std::set` имеет только одно значение в этом месте структуры .

Текущий размер дерева всегда присутствует, как и в случае реализации `std::list` в MSVC (52.4.2 (стр. 568)).

Как и в случае с `std::list`, итераторы это просто указатели на узлы . Итератор `.begin()` указывает на минимальный ключ. Этот указатель нигде не сохранен (как в списках), минимальный ключ дерева нужно находить каждый раз . `operator--` и `operator++` перемещают указатель не текущий узел на узел-предшественник или узел-преемник, т.е. узлы содержащие предыдущий и следующий ключ . Алгоритмы для всех этих операций описаны в [Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, *Introduction to Algorithms, Third Edition*, (2009)].

Итератор `.end()` указывает на узел-пустышку, он имеет 1 в `Isnll`, что означает, что у узла нет ключа и/или значения. Так что его можно рассматривать как «landing zone» в HDD¹³. Поле «parent» узла-пустышки указывает на корневой узел, который служит как вершина дерева, и уже содержит информацию.

GCC

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse, bool dump_keys_and_values)
{
    printf ("ptr=0x%p M_left=0x%p M_parent=0x%p M_right=0x%p M_color=%d\n",
           n, n->M_left, n->M_parent, n->M_right, n->M_color);

    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (dump_keys_and_values)
    {
        if (is_set)
            printf ("key=%d\n", *(int*)point_after_struct);
        else
        {
            struct map_pair *p=(struct map_pair *)point_after_struct;
            printf ("key=%d value=[%s]\n", p->key, p->value);
        }
    }
}
```

¹³Hard disk drive

52.4. STL

```
;
if (traverse==false)
    return;

if (n->M_left)
    dump_tree_node (n->M_left, is_set, traverse, dump_keys_and_values);
if (n->M_right)
    dump_tree_node (n->M_right, is_set, traverse, dump_keys_and_values);
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (is_set)
        printf ("%d\n", *(int*)point_after_struct);
    else
    {
        struct map_pair *p=(struct map_pair *)point_after_struct;
        printf ("%d [%s]\n", p->key, p->value);
    }

    if (n->M_left)
    {
        printf (".*sL-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left, is_set);
    };
    if (n->M_right)
    {
        printf (".*sR-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, M_key_compare=0x%x, M_header=0x%p, M_node_count=%d\n",
           m, m->M_key_compare, &m->M_header, m->M_node_count);
    dump_tree_node (m->M_header.M_parent, is_set, true, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";
```

```

printf ("dumping m as map:\n");
dump_map_and_set ((struct tree_struct *) (void*)&m, false);

std::map<int, const char*>::iterator it1=m.begin();
printf ("m.begin():\n");
dump_tree_node ((struct tree_node *) (void*)&it1, false, false, true);
it1=m.end();
printf ("m.end():\n");
dump_tree_node ((struct tree_node *) (void*)&it1, false, false, false);

// set

std::set<int> s;
s.insert(123);
s.insert(456);
s.insert(11);
s.insert(12);
s.insert(100);
s.insert(1001);
printf ("dumping s as set:\n");
dump_map_and_set ((struct tree_struct *) (void*)&s, true);
std::set<int>::iterator it2=s.begin();
printf ("s.begin():\n");
dump_tree_node ((struct tree_node *) (void*)&it2, true, false, true);
it2=s.end();
printf ("s.end():\n");
dump_tree_node ((struct tree_node *) (void*)&it2, true, false, false);
};

```

Листинг 52.36: GCC 4.8.1

```

dumping m as map:
ptr=0x0028FE3C, M_key_compare=0x402b70, M_header=0x0028FE40, M_node_count=17
ptr=0x007A4988 M_left=0x007A4C00 M_parent=0x0028FE40 M_right=0x007A4B80 M_color=1
key=10 value=[ten]
ptr=0x007A4C00 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4C60 M_color=1
key=1 value=[one]
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 M_color=1
key=0 value=[zero]
ptr=0x007A4C60 M_left=0x007A4B40 M_parent=0x007A4C00 M_right=0x007A4C20 M_color=0
key=5 value=[five]
ptr=0x007A4B40 M_left=0x007A4CE0 M_parent=0x007A4C60 M_right=0x00000000 M_color=1
key=3 value=[three]
ptr=0x007A4CE0 M_left=0x00000000 M_parent=0x007A4B40 M_right=0x00000000 M_color=0
key=2 value=[two]
ptr=0x007A4C20 M_left=0x00000000 M_parent=0x007A4C60 M_right=0x007A4D00 M_color=1
key=6 value=[six]
ptr=0x007A4D00 M_left=0x00000000 M_parent=0x007A4C20 M_right=0x00000000 M_color=0
key=9 value=[nine]
ptr=0x007A4B80 M_left=0x007A49A8 M_parent=0x007A4988 M_right=0x007A4BC0 M_color=1
key=100 value=[one hundred]
ptr=0x007A49A8 M_left=0x007A4BA0 M_parent=0x007A4B80 M_right=0x007A4C40 M_color=0
key=20 value=[twenty]
ptr=0x007A4BA0 M_left=0x007A4C80 M_parent=0x007A49A8 M_right=0x00000000 M_color=1
key=12 value=[twelve]
ptr=0x007A4C80 M_left=0x00000000 M_parent=0x007A4BA0 M_right=0x00000000 M_color=0
key=11 value=[eleven]
ptr=0x007A4C40 M_left=0x00000000 M_parent=0x007A49A8 M_right=0x00000000 M_color=1
key=99 value=[ninety-nine]
ptr=0x007A4BC0 M_left=0x007A4B60 M_parent=0x007A4B80 M_right=0x007A4CA0 M_color=0
key=107 value=[one hundred seven]
ptr=0x007A4B60 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x00000000 M_color=1
key=101 value=[one hundred one]
ptr=0x007A4CA0 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x007A4CC0 M_color=1
key=1001 value=[one thousand one]
ptr=0x007A4CC0 M_left=0x00000000 M_parent=0x007A4CA0 M_right=0x00000000 M_color=0
key=1010 value=[one thousand ten]
As a tree:
root----10 [ten]

```

```

    L-----1 [one]
      L-----0 [zero]
      R-----5 [five]
        L-----3 [three]
        L-----2 [two]
        R-----6 [six]
        R-----9 [nine]
  R-----100 [one hundred]
    L-----20 [twenty]
      L-----12 [twelve]
      L-----11 [eleven]
      R-----99 [ninety-nine]
    R-----107 [one hundred seven]
      L-----101 [one hundred one]
      R-----1001 [one thousand one]
        R-----1010 [one thousand ten]

m.begin():
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 M_color=1
key=0 value=[zero]
m.end():
ptr=0x0028FE40 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4CC0 M_color=0

dumping s as set:
ptr=0x0028FE20, M_key_compare=0x8, M_header=0x0028FE24, M_node_count=6
ptr=0x007A1E80 M_left=0x01D5D890 M_parent=0x0028FE24 M_right=0x01D5D850 M_color=1
key=123
ptr=0x01D5D890 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8B0 M_color=1
key=12
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=11
ptr=0x01D5D8B0 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=100
ptr=0x01D5D850 M_left=0x00000000 M_parent=0x007A1E80 M_right=0x01D5D8D0 M_color=1
key=456
ptr=0x01D5D8D0 M_left=0x00000000 M_parent=0x01D5D850 M_right=0x00000000 M_color=0
key=1001
As a tree:
root----123
  L-----12
    L-----11
    R-----100
  R-----456
    R-----1001

s.begin():
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=11
s.end():
ptr=0x0028FE24 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8D0 M_color=0

```

Реализация в GCC очень похожа ¹⁴. Разница только в том, что здесь нет поля `Isnil`, так что структура занимает немного меньше места в памяти чем та что реализована в MSVC. Узел-пустышка — это также место, куда указывает итератор `.end()`, не имеющий ключа и/или значения.

Демонстрация перебалансировки (GCC)

Вот также демонстрация, показывающая нам как дерево может перебалансироваться после вставок.

Листинг 52.37: GCC

```

#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{

```

¹⁴<http://go.yurichev.com/17084>

52.4. STL

```
int key;
const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    printf ("%d\n", *(int*)point_after_struct);

    if (n->M_left)
    {
        printf (".*sL-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left);
    };
    if (n->M_right)
    {
        printf (".*sR-----", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right);
    };
};

void dump_map_and_set(struct tree_struct *m)
{
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent);
};

int main()
{
    std::set<int> s;
    s.insert(123);
    s.insert(456);
    printf ("123, 456 are inserted\n");
    dump_map_and_set ((struct tree_struct *) (void*)&s);
    s.insert(11);
    s.insert(12);
    printf ("\n");
    printf ("11, 12 are inserted\n");
    dump_map_and_set ((struct tree_struct *) (void*)&s);
    s.insert(100);
    s.insert(1001);
    printf ("\n");
    printf ("100, 1001 are inserted\n");
    dump_map_and_set ((struct tree_struct *) (void*)&s);
    s.insert(667);
    s.insert(1);
    s.insert(4);
    s.insert(7);
    printf ("\n");
    printf ("667, 1, 4, 7 are inserted\n");
    dump_map_and_set ((struct tree_struct *) (void*)&s);
    printf ("\n");
```

};

Листинг 52.38: GCC 4.8.1

```

123, 456 are inserted
root----123
      R-----456

11, 12 are inserted
root----123
      L-----11
            R-----12
      R-----456

100, 1001 are inserted
root----123
      L-----12
            L-----11
            R-----100
      R-----456
            R-----1001

667, 1, 4, 7 are inserted
root----12
      L-----4
            L-----1
            R-----11
                  L-----7
      R-----123
            L-----100
            R-----667
                  L-----456
                  R-----1001

```

Глава 53

Отрицательные индексы массивов

Возможно адресовать место в памяти *перед* массивом задавая отрицательный индекс, например, `array[-1]`.

Трудно сказать, зачем это вообще может понадобиться, известно только одно практическое применение этому. Элементы массивов в Си/Си++ индексируются, начиная с 0, но в некоторых ЯП, первый элемент индексируется через 1 (как минимум FORTRAN). Привычка у программистов может остаться, так что все еще возможно адресовать первый элемент через 1 в Си/Си++ используя этот трюк:

```
#include <stdio.h>

int main()
{
    int random_value=0x11223344;
    unsigned char array[10];
    int i;
    unsigned char *fakearray=&array[-1];

    for (i=0; i<10; i++)
        array[i]=i;

    printf ("first element %d\n", fakearray[1]);
    printf ("second element %d\n", fakearray[2]);
    printf ("last element %d\n", fakearray[10]);

    printf ("array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X\n",
        array[-1],
        array[-2],
        array[-3],
        array[-4]);
};
```

Листинг 53.1: Неоптимизирующий MSVC 2010

```
1 $SG2751 DB      'first element %d', 0aH, 00H
2 $SG2752 DB      'second element %d', 0aH, 00H
3 $SG2753 DB      'last element %d', 0aH, 00H
4 $SG2754 DB      'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]'
5             DB      ']=%02X', 0aH, 00H
6
7 _fakearray$ = -24          ; size = 4
8 _random_value$ = -20      ; size = 4
9 _array$ = -16             ; size = 10
10 _i$ = -4                  ; size = 4
11 _main PROC
12     push    ebp
13     mov     ebp, esp
14     sub     esp, 24
15     mov     DWORD PTR _random_value$[ebp], 287454020 ; 11223344H
16     ; установить fakearray[] на байт раньше перед array[]
17     lea     eax, DWORD PTR _array$[ebp]
18     add     eax, -1 ; eax=eax-1
19     mov     DWORD PTR _fakearray$[ebp], eax
20     mov     DWORD PTR _i$[ebp], 0
21     jmp     SHORT $LN3@main
```

```

22      ; заполнить array[] 0..9
23 $LN2@main:
24     mov     ecx, DWORD PTR _i$[ebp]
25     add     ecx, 1
26     mov     DWORD PTR _i$[ebp], ecx
27 $LN3@main:
28     cmp     DWORD PTR _i$[ebp], 10
29     jge     SHORT $LN1@main
30     mov     edx, DWORD PTR _i$[ebp]
31     mov     al, BYTE PTR _i$[ebp]
32     mov     BYTE PTR _array$(ebp+edx), al
33     jmp     SHORT $LN2@main
34 $LN1@main:
35     mov     ecx, DWORD PTR _fakearray$(ebp)
36     ; ecx=адрес fakearray[0], ecx+1 это fakearray[1] либо array[0]
37     movzx   edx, BYTE PTR [ecx+1]
38     push    edx
39     push    OFFSET $SG2751 ; 'first element %d'
40     call    _printf
41     add     esp, 8
42     mov     eax, DWORD PTR _fakearray$(ebp)
43     ; eax=адрес fakearray[0], eax+2 это fakearray[2] либо array[1]
44     movzx   ecx, BYTE PTR [eax+2]
45     push    ecx
46     push    OFFSET $SG2752 ; 'second element %d'
47     call    _printf
48     add     esp, 8
49     mov     edx, DWORD PTR _fakearray$(ebp)
50     ; edx=адрес fakearray[0], edx+10 это fakearray[10] либо array[9]
51     movzx   eax, BYTE PTR [edx+10]
52     push    eax
53     push    OFFSET $SG2753 ; 'last element %d'
54     call    _printf
55     add     esp, 8
56     ; отнять 4, 3, 2 и 1 от указателя array[0] чтобы найти значения, лежащие перед array[]
57     lea     ecx, DWORD PTR _array$(ebp)
58     movzx   edx, BYTE PTR [ecx-4]
59     push    edx
60     lea     eax, DWORD PTR _array$(ebp)
61     movzx   ecx, BYTE PTR [eax-3]
62     push    ecx
63     lea     edx, DWORD PTR _array$(ebp)
64     movzx   eax, BYTE PTR [edx-2]
65     push    eax
66     lea     ecx, DWORD PTR _array$(ebp)
67     movzx   edx, BYTE PTR [ecx-1]
68     push    edx
69     push    OFFSET $SG2754 ; 'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X'
70     call    _printf
71     add     esp, 20
72     xor     eax, eax
73     mov     esp, ebp
74     pop     ebp
75     ret     0
76 _main     ENDP

```

Так что у нас тут массив `array[]` из десяти элементов, заполненный байтами `0...9`. Затем у нас указатель `fakearray[]` указывающий на один байт перед `array[]`. `fakearray[1]` указывает точно на `array[0]`. Но нам все еще любопытно, что же находится перед `array[]`? Мы добавляем `random_value` перед `array[]` и установим её в `0x11223344`. Неоптимизирующий компилятор выделяет переменные в том же порядке, в котором они объявлены, так что да, 32-битная `random_value` находится точно перед массивом.

Запускаем, и:

```

first element 0
second element 1
last element 9
array[-1]=11, array[-2]=22, array[-3]=33, array[-4]=44

```

Фрагмент стека, который мы скопируем из окна стека в OllyDbg (включая комментарии автора):

Листинг 53.2: Неоптимизирующий MSVC 2010

CPU Stack	
Address	Value
001DFBCC	/001DFBD3 ; указатель fakearray
001DFBD0	11223344 ; random_value
001DFBD4	03020100 ; 4 байта array[]
001DFBD8	07060504 ; 4 байта array[]
001DFBDC	00CB0908 ; случайный мусор + 2 последних байта array[]
001DFBE0	0000000A ; последнее значение i после того как закончился цикл
001DFBE4	001DFC2C ; сохраненное значение EBP
001DFBE8	\00CB129D ; адрес возврата (RA)

Указатель на `fakearray[]` (`0x001DFBD3`) это действительно адрес `array[]` в стеке (`0x001DFBD4`), но минус 1 байт.

Трюк этот все-таки слишком хакерский и сомнительный. Вряд ли кто-то будет его использовать в своем коде, но для демонстрации, он здесь очень уместен.

Глава 54

Windows 16-bit

16-битные программы под Windows в наше время редки, хотя иногда можно поработать с ними, в смысле ретрокомпьютинга, либо которые защищенные донглами (78 (стр. 748)).

16-битные версии Windows были вплоть до 3.11. 96/98/ME также поддерживает 16-битный код, как и все 32-битные OS линейки [Windows NT](#). 64-битные версии [Windows NT](#) не поддерживают 16-битный код вообще .

Код напоминает тот что под MS-DOS.

Исполняемые файлы имеют NE-тип (так называемый «new executable»).

Все рассмотренные здесь примеры скомпилированы компилятором OpenWatcom 1.9 используя эти опции:

```
wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows example.c
```

54.1. Пример#1

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
    MessageBeep(MB_ICONEXCLAMATION);
    return 0;
};

WinMain          proc near
                 push    bp
                 mov     bp, sp
                 mov     ax, 30h ; '0'      ; MB_ICONEXCLAMATION constant
                 push    ax
                 call    MESSAGEBEEP
                 xor     ax, ax             ; return 0
                 pop     bp
                 retn    0Ah
WinMain          endp
```

Пока всё просто.

54.2. Пример #2

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
```

54.3. ПРИМЕР #3

```
    MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);
    return 0;
};
```

```
WinMain      proc near
              push    bp
              mov     bp, sp
              xor     ax, ax          ; NULL
              push    ax
              push    ds
              mov     ax, offset aHelloWorld ; 0x18. "hello, world"
              push    ax
              push    ds
              mov     ax, offset aCaption ; 0x10. "caption"
              push    ax
              mov     ax, 3           ; MB_YESNOCANCEL
              push    ax
              call    MESSAGEBOX
              xor     ax, ax          ; return 0
              pop     bp
              retn    0Ah
WinMain      endp

dseg02:0010 aCaption      db 'caption',0
dseg02:0018 aHelloWorld   db 'hello, world',0
```

Пара важных моментов: соглашение о передаче аргументов здесь **PASCAL** : оно указывает что самый первый аргумент должен передаваться первым (**MB_YESNOCANCEL**), а самый последний аргумент — последним (NULL). Это соглашение также указывает вызываемой функции восстановить **stack pointer**: поэтому инструкция **RETN** имеет аргумент **0Ah** означая что указатель нужно сдвинуть вперед на 10 байт во время возврата из функции . Это как stdcall (65.2 (стр. 672)), только аргументы передаются в «естественном» порядке.

Указатели передаются парами: сначала сегмент данных, потом указатель внутри сегмента . В этом примере только один сегмент, так что **DS** всегда указывает на сегмент данных в исполняемом файле.

54.3. Пример #3

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
    int result=MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);

    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK);
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)
        MessageBox (NULL, "you pressed no", "caption", MB_OK);

    return 0;
};
```

```
WinMain      proc near
              push    bp
              mov     bp, sp
              xor     ax, ax          ; NULL
              push    ax
              push    ds
              mov     ax, offset aHelloWorld ; "hello, world"
              push    ax
              push    ds
```

54.4. ПРИМЕР #4

```
mov     ax, offset aCaption ; "caption"
push    ax
mov     ax, 3                ; MB_YESNOCANCEL
push    ax
call    MESSAGEBOX
cmp     ax, 2                ; IDCANCEL
jnz     short loc_2F
xor     ax, ax
push    ax
push    ds
mov     ax, offset aYouPressedCanc ; "you pressed cancel"
jmp     short loc_49
loc_2F:
cmp     ax, 6                ; IDYES
jnz     short loc_3D
xor     ax, ax
push    ax
push    ds
mov     ax, offset aYouPressedYes ; "you pressed yes"
jmp     short loc_49
loc_3D:
cmp     ax, 7                ; IDNO
jnz     short loc_57
xor     ax, ax
push    ax
push    ds
mov     ax, offset aYouPressedNo ; "you pressed no"
loc_49:
push    ax
push    ds
mov     ax, offset aCaption ; "caption"
push    ax
xor     ax, ax
push    ax
call    MESSAGEBOX
loc_57:
xor     ax, ax
pop     bp
retn    0Ah
WinMain endp
```

Немного расширенная версия примера из предыдущей секции .

54.4. Пример #4

```
#include <windows.h>

int PASCAL func1 (int a, int b, int c)
{
    return a*b+c;
};

long PASCAL func2 (long a, long b, long c)
{
    return a*b+c;
};

long PASCAL func3 (long a, long b, long c, int d)
{
    return a*b+c-d;
};

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
```


54.4. ПРИМЕР #4

```
func1 (123, 456, 789);
func2 (600000, 700000, 800000);
func3 (600000, 700000, 800000, 123);
return 0;
```

```
};
```

```
func1      proc near

c          = word ptr 4
b          = word ptr 6
a          = word ptr 8

            push    bp
            mov     bp, sp
            mov     ax, [bp+a]
            imul    [bp+b]
            add     ax, [bp+c]
            pop     bp
            retn    6

func1      endp

func2      proc near

arg_0      = word ptr 4
arg_2      = word ptr 6
arg_4      = word ptr 8
arg_6      = word ptr 0Ah
arg_8      = word ptr 0Ch
arg_A      = word ptr 0Eh

            push    bp
            mov     bp, sp
            mov     ax, [bp+arg_8]
            mov     dx, [bp+arg_A]
            mov     bx, [bp+arg_4]
            mov     cx, [bp+arg_6]
            call    sub_B2 ; long 32-bit multiplication
            add     ax, [bp+arg_0]
            adc     dx, [bp+arg_2]
            pop     bp
            retn    12

func2      endp

func3      proc near

arg_0      = word ptr 4
arg_2      = word ptr 6
arg_4      = word ptr 8
arg_6      = word ptr 0Ah
arg_8      = word ptr 0Ch
arg_A      = word ptr 0Eh
arg_C      = word ptr 10h

            push    bp
            mov     bp, sp
            mov     ax, [bp+arg_A]
            mov     dx, [bp+arg_C]
            mov     bx, [bp+arg_6]
            mov     cx, [bp+arg_8]
            call    sub_B2 ; long 32-bit multiplication
            mov     cx, [bp+arg_2]
            add     cx, ax
            mov     bx, [bp+arg_4]
            adc     bx, dx          ; BX=high part, CX=low part
            mov     ax, [bp+arg_0]
            cwd                     ; AX=low part d, DX=high part d
            sub     cx, ax
            mov     ax, cx
            sbb     bx, dx
            mov     dx, bx
```

54.5. ПРИМЕР #5

```
func3      pop    bp
           retn   14
           endp

WinMain    proc near
           push   bp
           mov    bp, sp
           mov    ax, 123
           push   ax
           mov    ax, 456
           push   ax
           mov    ax, 789
           push   ax
           call   func1
           mov    ax, 9      ; high part of 600000
           push   ax
           mov    ax, 27C0h ; low part of 600000
           push   ax
           mov    ax, 0Ah    ; high part of 700000
           push   ax
           mov    ax, 0AE60h ; low part of 700000
           push   ax
           mov    ax, 0Ch    ; high part of 800000
           push   ax
           mov    ax, 3500h  ; low part of 800000
           push   ax
           call   func2
           mov    ax, 9      ; high part of 600000
           push   ax
           mov    ax, 27C0h ; low part of 600000
           push   ax
           mov    ax, 0Ah    ; high part of 700000
           push   ax
           mov    ax, 0AE60h ; low part of 700000
           push   ax
           mov    ax, 0Ch    ; high part of 800000
           push   ax
           mov    ax, 3500h  ; low part of 800000
           push   ax
           mov    ax, 7Bh    ; 123
           push   ax
           call   func3
           xor     ax, ax    ; return 0
           pop    bp
           retn   0Ah
WinMain    endp
```

32-битные значения (тип данных `long` означает 32-бита, а `int` здесь 16-битный) в 16-битном коде (и в MS-DOS и в Win16) передаются парами). Это так же как и 64-битные значения передаются в 32-битной среде (25 (стр. 392)).

`sub_B2 here` здесь это библиотечная функция написанная разработчиками компилятора, делающая «long multiplication», т.е. перемножает два 32-битных значения. Другие функции компиляторов делающие то же самое перечислены здесь : [E](#) (стр. 933), [D](#) (стр. 932).

Пара инструкций `ADD / ADC` используется для сложения этих составных значений : `ADD` может установить или сбросить флаг `CF`, а `ADC` будет использовать его после.

Пара инструкций `SUB / SBB` используется для вычитания: `SUB` может установить или сбросить флаг `CF`, `SBB` будет использовать его после.

32-битные значения возвращаются из функций в паре регистров `DX:AX`.

Константы так же передаются как пары в `WinMain()`.

Константа 123 типа `int` в начале конвертируется (учитывая знак) в 32-битное значение используя инструкция `CWD`.

54.5. Пример #5

```

#include <windows.h>

int PASCAL string_compare (char *s1, char *s2)
{
    while (1)
    {
        if (*s1!=*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    };
};

int PASCAL string_compare_far (char far *s1, char far *s2)
{
    while (1)
    {
        if (*s1!=*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    };
};

void PASCAL remove_digits (char *s)
{
    while (*s)
    {
        if (*s>='0' && *s<='9')
            *s='_';
        s++;
    };
};

char str[]="hello 1234 world";

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    string_compare ("asd", "def");
    string_compare_far ("asd", "def");
    remove_digits (str);
    MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
    return 0;
};

```

```

string_compare proc near

arg_0 = word ptr 4
arg_2 = word ptr 6

    push    bp
    mov     bp, sp
    push    si
    mov     si, [bp+arg_0]
    mov     bx, [bp+arg_2]

loc_12: ; CODE XREF: string_compare+21j
    mov     al, [bx]
    cmp     al, [si]
    jz      short loc_1C

```

54.5. ПРИМЕР #5

```
xor     ax, ax
jmp     short loc_2B

loc_1C: ; CODE XREF: string_compare+Ej
test    al, al
jz      short loc_22
jnz     short loc_27

loc_22: ; CODE XREF: string_compare+16j
mov     ax, 1
jmp     short loc_2B

loc_27: ; CODE XREF: string_compare+18j
inc     bx
inc     si
jmp     short loc_12

loc_2B: ; CODE XREF: string_compare+12j
        ; string_compare+1Dj
pop     si
pop     bp
retn    4
string_compare endp

string_compare_far proc near ; CODE XREF: WinMain+18p

arg_0 = word ptr 4
arg_2 = word ptr 6
arg_4 = word ptr 8
arg_6 = word ptr 0Ah

push    bp
mov     bp, sp
push    si
mov     si, [bp+arg_0]
mov     bx, [bp+arg_4]

loc_3A: ; CODE XREF: string_compare_far+35j
mov     es, [bp+arg_6]
mov     al, es:[bx]
mov     es, [bp+arg_2]
cmp     al, es:[si]
jz      short loc_4C
xor     ax, ax
jmp     short loc_67

loc_4C: ; CODE XREF: string_compare_far+16j
mov     es, [bp+arg_6]
cmp     byte ptr es:[bx], 0
jz      short loc_5E
mov     es, [bp+arg_2]
cmp     byte ptr es:[si], 0
jnz     short loc_63

loc_5E: ; CODE XREF: string_compare_far+23j
mov     ax, 1
jmp     short loc_67

loc_63: ; CODE XREF: string_compare_far+2Cj
inc     bx
inc     si
jmp     short loc_3A

loc_67: ; CODE XREF: string_compare_far+1Aj
```

54.5. ПРИМЕР #5

```
        ; string_compare_far+31j
    pop     si
    pop     bp
    retn    8
string_compare_far endp

remove_digits    proc near ; CODE XREF: WinMain+1Fp

arg_0 = word ptr 4

    push    bp
    mov     bp, sp
    mov     bx, [bp+arg_0]

loc_72: ; CODE XREF: remove_digits+18j
    mov     al, [bx]
    test    al, al
    jz      short loc_86
    cmp     al, 30h ; '0'
    jnb     short loc_83
    cmp     al, 39h ; '9'
    ja      short loc_83
    mov     byte ptr [bx], 2Dh ; '-'

loc_83: ; CODE XREF: remove_digits+Ej
        ; remove_digits+12j
    inc     bx
    jmp     short loc_72

loc_86: ; CODE XREF: remove_digits+Aj
    pop     bp
    retn    2
remove_digits    endp

WinMain proc near ; CODE XREF: start+EDp
    push    bp
    mov     bp, sp
    mov     ax, offset aAsd ; "asd"
    push    ax
    mov     ax, offset aDef ; "def"
    push    ax
    call    string_compare
    push    ds
    mov     ax, offset aAsd ; "asd"
    push    ax
    push    ds
    mov     ax, offset aDef ; "def"
    push    ax
    call    string_compare_far
    mov     ax, offset aHello1234World ; "hello 1234 world"
    push    ax
    call    remove_digits
    xor     ax, ax
    push    ax
    push    ds
    mov     ax, offset aHello1234World ; "hello 1234 world"
    push    ax
    push    ds
    mov     ax, offset aCaption ; "caption"
    push    ax
    mov     ax, 3 ; MB_YESNOCANCEL
    push    ax
    call    MESSAGEBOX
    xor     ax, ax
    pop     bp
    retn    0Ah
WinMain endp
```

Здесь мы можем увидеть разницу между указателями «near» и указателями «far» еще один ужасный артефакт сегмен-

54.6. ПРИМЕР #6

тированной памяти 16-битного 8086 .

Читайте больше об этом: [96](#) (стр. [898](#)).

Указатели «near» («близкие») это те которые указывают в пределах текущего сегмента . Поэтому, функция `string_compare()` берет на вход только 2 16-битных значения и работает с данными расположенными в сегменте, на который указывает `DS` (инструкция `mov al, [bx]` на самом деле работает как `mov al, ds:[bx]` — `DS` используется здесь неявно).

Указатели «far» (далекие) могут указывать на данные в другом сегменте памяти . Поэтому `string_compare_far()` берет на вход 16-битную пару как указатель, загружает старшую часть в сегментный регистр `ES` и обращается к данным через него (`mov al, es:[bx]`). Указатели «far» также используются в моем win16-примере касательно `MessageBox()` : [54.2](#) (стр. [592](#)). Действительно, ядро Windows должно знать, из какого сегмента данных читать текстовые строки, так что ему нужна полная информация.

Причина этой разницы в том, что компактная программа вполне может обойтись одним сегментом данных размером 64 килобайта, так что старшую часть указателя передавать не нужна (ведь она одинаковая везде) . Большие программы могут использовать несколько сегментов данных размером 64 килобайта, так что нужно указывать каждый раз, в каком сегменте расположены данные .

То же касается и сегментов кода. Компактная программа может расположиться в пределах одного 64kb-сегмента, тогда функции в ней будут вызываться инструкцией `CALL NEAR` , а возвращаться управление используя `RETN` . Но если сегментов кода несколько, тогда и адрес вызываемой функции будет задаваться парой, вызываться она будет используя `CALL FAR` , а возвращаться управление используя `RETF` .

Это то что задается в компиляторе указывая «memory model».

Компиляторы под MS-DOS и Win16 имели разные библиотеки под разные модели памяти: они отличались типами указателей для кода и данных.

54.6. Пример #6

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{
    struct tm *t;
    time_t unix_time;

    unix_time=time(NULL);

    t=localtime (&unix_time);

    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, t->tm_mon, t->tm_mday,
            t->tm_hour, t->tm_min, t->tm_sec);

    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};
```

WinMain	proc near
var_4	= word ptr -4
var_2	= word ptr -2
	push bp
	mov bp, sp
	push ax
	push ax
	xor ax, ax

```

call    time_
mov     [bp+var_4], ax    ; low part of UNIX time
mov     [bp+var_2], dx    ; high part of UNIX time
lea     ax, [bp+var_4]    ; take a pointer of high part
call    localtime_
mov     bx, ax            ; t
push    word ptr [bx]     ; second
push    word ptr [bx+2]   ; minute
push    word ptr [bx+4]   ; hour
push    word ptr [bx+6]   ; day
push    word ptr [bx+8]   ; month
mov     ax, [bx+0Ah]      ; year
add     ax, 1900
push    ax
mov     ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
push    ax
mov     ax, offset strbuf
push    ax
call    sprintf_
add     sp, 10h
xor     ax, ax            ; NULL
push    ax
push    ds
mov     ax, offset strbuf
push    ax
push    ds
mov     ax, offset aCaption ; "caption"
push    ax
xor     ax, ax            ; MB_OK
push    ax
call    MESSAGEBOX
xor     ax, ax
mov     sp, bp
pop     bp
retn    0Ah
WinMain endp

```

Время в формате UNIX это 32-битное значение, так что оно возвращается в паре регистров `DX:AX` и сохраняется в двух локальных 16-битных переменных. Потом указатель на эту пару передается в функцию `localtime()`. Функция `localtime()` имеет структуру `struct tm` расположенную у себя где-то внутри, так что только указатель на нее возвращается. Кстати, это также означает что функцию нельзя вызывать еще раз, пока её результаты не были использованы.

Для функций `time()` и `localtime()` используется Watcom-соглашение о вызовах: первые четыре аргумента передаются через регистры `AX`, `DX`, `BX` и `CX`, а остальные аргументы через стек. Функции, использующие это соглашение, маркируются символом подчеркивания в конце имени.

Для вызова функции `sprintf()` используется обычное соглашение `cdecl` (65.1 (стр. 672)) вместо `PASCAL` или Watcom, так что аргументы передаются привычным образом.

54.6.1. Глобальные переменные

Это тот же пример, только переменные теперь глобальные:

```

#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];
struct tm *t;
time_t unix_time;

int PASCAL WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow )
{

```

54.6. ПРИМЕР #6

```

    unix_time=time(NULL);

    t=localtime (&unix_time);

    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, t->tm_mon, t->tm_mday,
        t->tm_hour, t->tm_min, t->tm_sec);

    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};

```

```

unix_time_low    dw 0
unix_time_high   dw 0
t                dw 0

WinMain          proc near
    push        bp
    mov         bp, sp
    xor         ax, ax
    call        time_
    mov         unix_time_low, ax
    mov         unix_time_high, dx
    mov         ax, offset unix_time_low
    call        localtime_
    mov         bx, ax
    mov         t, ax                ; will not be used in future...
    push        word ptr [bx]        ; seconds
    push        word ptr [bx+2]      ; minutes
    push        word ptr [bx+4]      ; hour
    push        word ptr [bx+6]      ; day
    push        word ptr [bx+8]      ; month
    mov         ax, [bx+0Ah]         ; year
    add         ax, 1900
    push        ax
    mov         ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
    push        ax
    mov         ax, offset strbuf
    push        ax
    call        sprintf_
    add         sp, 10h
    xor         ax, ax                ; NULL
    push        ax
    push        ds
    mov         ax, offset strbuf
    push        ax
    push        ds
    mov         ax, offset aCaption ; "caption"
    push        ax
    xor         ax, ax                ; MB_OK
    push        ax
    call        MESSAGEBOX
    xor         ax, ax                ; return 0
    pop         bp
    retn        0Ah
WinMain          endp

```

`t` не будет использоваться, но компилятор создал код, записывающий в эту переменную. Потому что он не уверен, может быть это значение будет прочитано где-то в другом модуле.

Часть IV

Java

Глава 55

Java

55.1. Введение

Есть немало известных декомпиляторов для Java (или для JVM-байткода вообще) ¹.

Причина в том что декомпиляция JVM-байткода проще чем низкоуровневого x86-кода:

- Здесь намного больше информации о типах.
- Модель памяти в JVM более строгая и очерченная.
- Java-компилятор не делает никаких оптимизаций (это делает JVM JIT² во время исполнения), так что байткод в class-файлах легко читаем.

Когда знания JVM-байткода могут быть полезны?

- Мелкая/несложная работа по патчингу class-файлов без необходимости снова компилировать результаты декомпилятора.
- Анализ обфусцированного кода.
- Создание вашего собственного обфускатора.
- Создание кодегенератора компилятора (back-end), создающего код для JVM (как Scala, Clojure, итд ³).

Начнем с простых фрагментов кода. Если не указано иное, везде используется JDK 1.7.

Эта команда использовалась везде для декомпиляции class-файлов: `javap -c -verbose`

Эта книга использовалась мною для подготовки всех примеров: [Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] ⁴.

55.2. Возврат значения

Наверное, самая простая из всех возможных функций на Java это та, что возвращает некоторое значение. О, и мы не должны забывать что в Java нет «свободных» функций в общем смысле, это «методы». Каждый метод принадлежит какому-то классу, так что невозможно объявить метод вне какого-либо класса. Но мы все равно будем называть их «функциями», для простоты.

```
public class ret
{
    public static int main(String[] args)
    {
        return 0;
    }
}
```

Компилируем это:

¹Например, JAD: <http://varaneckas.com/jad/>

²Just-in-time compilation

³Полный список: http://en.wikipedia.org/wiki/List_of_JVM_languages

⁴Также доступно здесь: <https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf>; <http://docs.oracle.com/javase/specs/jvms/se7/html/>

```
javac ret.java
```

...и декомпилирую используя стандартную утилиту в Java:

```
javap -c -verbose ret.class
```

И получаем:

Листинг 55.1: JDK 1.7 (excerpt)

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
       0: iconst_0
       1: ireturn
```

Разработчики Java решили что 0 это самая используемая константа в программировании, так что здесь есть отдельная однобайтная инструкция `iconst_0`, заталкивающая 0 в стек ⁵. Здесь есть также `iconst_1` (заталкивающая 1), `iconst_2`, итд, вплоть до `iconst_5`. Есть также `iconst_m1` заталкивающая -1.

The stack is used in JVM for passing data to called functions and also for returning values. So `iconst_0` pushes 0 into the stack. `ireturn` returns an integer value (*i* in name mean *integer*) from the TOS⁶.

Немного перепишем наш пример, теперь возвращаем 1234:

```
public class ret
{
    public static int main(String[] args)
    {
        return 1234;
    }
}
```

...получаем:

Листинг 55.2: JDK 1.7 (excerpt)

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
       0: sipush      1234
       3: ireturn
```

`sipush` (*short integer*) заталкивает значение 1234 в стек. *short* в имени означает, что 16-битное значение будет заталкиваться в стек. Число 1234 действительно помещается в 16-битное значение.

Как насчет больших значений?

```
public class ret
{
    public static int main(String[] args)
    {
        return 12345678;
    }
}
```

Листинг 55.3: Constant pool

```
...
#2 = Integer      12345678
...
```

⁵Так же как и в MIPS, где для нулевой константы имеется отдельный регистр: 4.5.2 (стр. 21).

⁶Top Of Stack (вершина стека)

```
public static int main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
      0: ldc          #2              // int 12345678
      2: ireturn
```

Невозможно закодировать 32-битное число в опкоде какой-либо JVM-инструкции, разработчики не оставили такой возможности. Так что 32-битное число 12345678 сохранено в так называемом «constant pool» (пул констант), который, так скажем, является библиотекой наиболее используемых констант (включая строки, объекты, итд).

Этот способ передачи констант не уникален для JVM. MIPS, ARM и прочие RISC-процессоры не могут кодировать 32-битные числа в 32-битных опкодах, так что код для RISC-процессоров (включая MIPS и ARM) должен конструировать значения в несколько шагов, или держать их в сегменте данных: [29.3](#) (стр. 440), [30.1](#) (стр. 443).

Код для MIPS также традиционно имеет пул констант, называемый «literal pool», это сегменты с названиями «.lit4» (для хранения 32-битных чисел с плавающей точкой одинарной точности) и «.lit8» (для хранения 64-битных чисел с плавающей точкой двойной точности).

Попробуем некоторые другие типы данных!

Boolean:

```
public class ret
{
    public static boolean main(String[] args)
    {
        return true;
    }
}
```

```
public static boolean main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
      0: iconst_1
      1: ireturn
```

Этот JVM-байткод не отличается от того, что возвращает целочисленную 1. 32-битные слоты данных в стеке также используются для булевых значений, как в Си/Си++. Но нельзя использовать возвращаемое значение булевого типа как целочисленное и наоборот — информация о типах сохраняется в class-файлах и проверяется при запуске.

Та же история с 16-битным *short*:

```
public class ret
{
    public static short main(String[] args)
    {
        return 1234;
    }
}
```

```
public static short main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
      0: sipush        1234
      3: ireturn
```

...и *char*!

```
public class ret
{
    public static char main(String[] args)
    {
        return 'A';
    }
}
```

```
public static char main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
       0: bipush      65
       2: ireturn
```

`bipush` означает «push byte». Нужно сказать что *char* в Java, это 16-битный символ в кодировке UTF-16, и он эквивалентен *short*, но ASCII-код символа «А» это 65, и можно воспользоваться инструкцией для передачи байта в стек.

Попробуем также *byte*:

```
public class retc
{
    public static byte main(String[] args)
    {
        return 123;
    }
}
```

```
public static byte main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
       0: bipush      123
       2: ireturn
```

Кто-то может спросить, зачем заморачиваться использованием 16-битного типа *short*, который внутри все равно 32-битный *integer*? Зачем использовать тип данных *char*, если это то же самое что и тип *short*?

Ответ прост: для контроля типов данных и читабельности исходников. *char* может быть эквивалентом *short*, но мы быстро понимаем что это ячейка для символа в кодировке UTF-16, а не для какого-то другого целочисленного значения. Когда используем *short*, мы можем показать всем, что диапазон этой переменной ограничен 16-ю битами. Очень хорошая идея использовать тип *boolean* где нужно, вместо *int* для тех же целей, как это было в Си.

В Java есть также 64-битный целочисленный тип:

```
public class ret3
{
    public static long main(String[] args)
    {
        return 1234567890123456789L;
    }
}
```

Листинг 55.4: Constant pool

```
...
#2 = Long          1234567890123456789L
...
```

```
public static long main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       0: ldc2_w      #2          // long 1234567890123456789L
       3: lreturn
```

64-битное число также хранится в пуле констант, `ldc2_w` загружает его и `lreturn` (*long return*) возвращает его.

Инструкция `ldc2_w` также используется для загрузки чисел с плавающей точкой двойной точности (которые также занимают 64 бита) из пула констант:

```
public class ret
{
    public static double main(String[] args)
    {
        return 123.456d;
    }
}
```

55.3. ПРОСТАЯ ВЫЧИСЛЯЮЩАЯ ФУНКЦИЯ

```
}  
}
```

Листинг 55.5: Constant pool

```
...  
#2 = Double          123.456d  
...
```

```
public static double main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=2, locals=1, args_size=1  
    0: ldc2_w         #2              // double 123.456d  
    3: dreturn
```

`dreturn` означает «return double».

И наконец, числа с плавающей точкой одинарной точности:

```
public class ret  
{  
    public static float main(String[] args)  
    {  
        return 123.456f;  
    }  
}
```

Листинг 55.6: Constant pool

```
...  
#2 = Float           123.456f  
...
```

```
public static float main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=1, locals=1, args_size=1  
    0: ldc             #2              // float 123.456f  
    2: freturn
```

Используемая здесь инструкция `ldc` та же, что и для загрузки 32-битных целочисленных чисел из пула констант. `freturn` означает «return float».

А что насчет тех случаев, когда функция ничего не возвращает?

```
public class ret  
{  
    public static void main(String[] args)  
    {  
        return;  
    }  
}
```

```
public static void main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=0, locals=1, args_size=1  
    0: return
```

Это означает, что инструкция `return` используется для возврата управления без возврата какого-либо значения. Зная все это, по последней инструкции очень легко определить тип возвращаемого значения функции (или метода).

55.3. Простая вычисляющая функция

Продолжим с простой вычисляющей функцией.

```
public class calc
{
    public static int half(int a)
    {
        return a/2;
    }
}
```

Это тот случай, когда используется инструкция `iconst_2`:

```
public static int half(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
    0: iload_0
    1: iconst_2
    2: idiv
    3: ireturn
```

`iload_0` берет нулевой аргумент функции и заталкивает его в стек. `iconst_2` заталкивает в стек 2. Вот как выглядит стек после исполнения этих двух инструкций:

```
+---+
TOS ->| 2 |
+---+
    | a |
+---+
```

`idiv` просто берет два значения на вершине стека (`TOS`), делит одно на другое и оставляет результат на вершине (`TOS`):

```
+-----+
TOS ->| result |
+-----+
```

`ireturn` берет его и возвращает.

Продолжим с числами с плавающей запятой, двойной точности:

```
public class calc
{
    public static double half_double(double a)
    {
        return a/2.0;
    }
}
```

Листинг 55.7: Constant pool

```
...
#2 = Double          2.0d
...
```

```
public static double half_double(double);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=4, locals=2, args_size=1
    0: dload_0
    1: ldc2_w      #2          // double 2.0d
    4: ddiv
    5: dreturn
```

Почти то же самое, но инструкция `ldc2_w` используется для загрузки константы 2.0 из пула констант. Также, все три инструкции имеют префикс *d*, что означает, что они работают с переменными типа *double*.

Теперь перейдем к функции с двумя аргументами:

```
public class calc
{
    public static int sum(int a, int b)
    {
        return a+b;
    }
}
```

```
public static int sum(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2, args_size=2
    0: iload_0
    1: iload_1
    2: iadd
    3: ireturn
```

`iload_0` загружает первый аргумент функции (a), `iload_2` – второй (b). Вот так выглядит стек после исполнения обеих инструкций:

```

+---+
TOS ->| b |
+---+
    | a |
+---+
```

`iadd` складывает два значения и оставляет результат на [TOS](#):

```

+-----+
TOS ->| result |
+-----+
```

Расширим этот пример до типа данных *long*:

```
public static long lsum(long a, long b)
{
    return a+b;
}
```

...получим:

```
public static long lsum(long, long);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=4, locals=4, args_size=2
    0: lload_0
    1: lload_2
    2: ladd
    3: lreturn
```

Вторая инструкция `lload` берет второй аргумент из второго слота. Это потому что 64-битное значение *long* занимает ровно два 32-битных слота.

Немного более сложный пример:

```
public class calc
{
    public static int mult_add(int a, int b, int c)
    {
        return a*b+c;
    }
}
```

```
public static int mult_add(int, int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=3, args_size=3
```


55.4. МОДЕЛЬ ПАМЯТИ В JVM

```
0: iload_0
1: iload_1
2: imul
3: iload_2
4: iadd
5: ireturn
```

Первый шаг это умножение. Произведение остается на **TOS**:

```
+-----+
TOS ->| product |
+-----+
```

iload_2 загружает третий аргумент (с) в стек:

```
+-----+
TOS ->|   c   |
+-----+
| product |
+-----+
```

Теперь инструкция **iadd** может сложить два значения.

55.4. Модель памяти в JVM

x86 и другие низкоуровневые среды используют стек для передачи аргументов и как хранилище локальных переменных. **JVM** устроена немного иначе.

Тут есть:

- Массив локальных переменных (**LVA**⁷). Используется как хранилище для аргументов функций и локальных переменных. Инструкции вроде **iload_0** загружают значения оттуда. **istore** записывает значения туда. В начале идут аргументы функции: начиная с 0, или с 1 (если нулевой аргумент занят указателем *this*. Затем располагаются локальные переменные.

Каждый слот имеет размер 32 бита. Следовательно, значения типов *long* и *double* занимают два слота.

- Стек операндов (или просто «стек»). Используется для вычислений и для передачи аргументов во время вызова других функций. В отличие от низкоуровневых сред вроде x86, здесь невозможно работать со стеком без использования инструкций, которые явно заталкивают или вытаскивают значения туда/оттуда.
- Куча (heap). Используется как хранилище для объектов и массивов.

Эти 3 области изолированы друг от друга.

55.5. Простой вызов функций

Math.random() возвращает псевдослучайное число в пределах [0.0 ...1.0), но представим, по какой-то причине, нам нужна функция, возвращающая число в пределах [0.0 ...0.5):

```
public class HalfRandom
{
    public static double f()
    {
        return Math.random()/2;
    }
}
```

Листинг 55.8: Constant pool

```
...
#2 = Methodref          #18.#19          // java/lang/Math.random:()D
#3 = Double              2.0d
...
```

⁷ (Java) Local Variable Array (массив локальных переменных)

55.5. ПРОСТОЙ ВЫЗОВ ФУНКЦИЙ

```
#12 = Utf8          ()D
...
#18 = Class          #22          // java/lang/Math
#19 = NameAndType     #23:#12      // random:()D
#22 = Utf8            java/lang/Math
#23 = Utf8            random
```

```
public static double f();
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=0, args_size=0
      0: invokestatic  #2          // Method java/lang/Math.random:()D
      3: ldc2_w         #3          // double 2.0d
      6: ddiv
      7: dreturn
```

`invokestatic` вызывает функцию `Math.random()` и оставляет результат на [TOS](#). Затем результат делится на 2.0 и возвращается. Но как закодировано имя функции? Оно закодировано в пуле констант используя выражение `Methodref`. Оно определяет имена класса и метода. Первое поле `Methodref` указывает на выражение `Class`, которое, в свою очередь, указывает на обычную текстовую строку («java/lang/Math»). Второе выражение `Methodref` указывает на выражение `NameAndType`, которое также имеет две ссылки на строки. Первая строка это «random», это имя метода. Вторая строка это «()D», которая кодирует тип функции. Это означает что возвращаемый тип — *double* (отсюда *D* в строке). Благодаря этому 1) JVM проверяет корректность типов данных; 2) Java-декомпиляторы могут восстанавливать типы данных из class-файлов.

Наконец попробуем пример «Hello, world!»:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}
```

Листинг 55.9: Constant pool

```
...
#2 = Fieldref        #16.#17      // java/lang/System.out:Ljava/io/PrintStream;
#3 = String          #18          // Hello, World
#4 = Methodref        #19.#20      // java/io/PrintStream.println:(Ljava/lang/String;)V
...
#16 = Class           #23          // java/lang/System
#17 = NameAndType      #24:#25      // out:Ljava/io/PrintStream;
#18 = Utf8            Hello, World
#19 = Class           #26          // java/io/PrintStream
#20 = NameAndType      #27:#28      // println:(Ljava/lang/String;)V
...
#23 = Utf8            java/lang/System
#24 = Utf8            out
#25 = Utf8            Ljava/io/PrintStream;
#26 = Utf8            java/io/PrintStream
#27 = Utf8            println
#28 = Utf8            (Ljava/lang/String;)V
...
```

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
      0: getstatic  #2          // Field java/lang/System.out:Ljava/io/PrintStream;
      3: ldc        #3          // String Hello, World
      5: invokevirtual #4        // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      ↵ ; )V
      8: return
```

55.6. ВЫЗОВ BEEP()

`ldc` по смещению 3 берет указатель (или адрес) на строку «Hello, World» в пуле констант и заталкивает его в стек. В мире Java это называется *reference*, но это скорее указатель или просто адрес ⁸.

Уже знакомая нам инструкция `invokevirtual` берет информацию о функции (или методе) `println` из пула констант и вызывает её. Как мы можем знать, есть несколько методов `println`, каждый предназначен для каждого типа данных. В нашем случае, используется та версия `println`, которая для типа данных *String*.

Что насчет первой инструкции `getstatic`? Эта инструкция берет *reference* (или адрес) поля объекта `System.out` и заталкивает его в стек. Это значение работает как указатель *this* для метода `println`. Таким образом, внутри, метод `println` берет на вход два аргумента: 1) *this*, т.е. указатель на объект ⁹; 2) адрес строки «Hello, World».

Действительно, `println()` вызывается как метод в рамках инициализированного объекта `System.out`.

Для удобства, утилита `javap` пишет всю эту информацию в комментариях.

55.6. Вызов beep()

Вот простейший вызов двух функций без аргументов:

```
public static void main(String[] args)
{
    java.awt.Toolkit.getDefaultToolkit().beep();
};
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
       0: invokestatic #2          // Method java/awt/Toolkit.getDefaultToolkit():Ljava/awt/
    ↵ /Toolkit;
       3: invokevirtual #3          // Method java/awt/Toolkit.beep():V
       6: return
```

Первая `invokestatic` по смещению 0 вызывает `java.awt.Toolkit.getDefaultToolkit()`, которая возвращает *reference* (указатель) на объект класса `Toolkit`. Инструкция `invokevirtual` по смещению 3 вызывает метод `beep()` этого класса.

55.7. Линейный конгруэнтный ГПСЧ

Попробуем простой генератор псевдослучайных чисел, который мы однажды уже рассматривали в этой книге (21 (стр. 334)):

```
public class LCG
{
    public static int rand_state;

    public void my_srand (int init)
    {
        rand_state=init;
    }

    public static int RNG_a=1664525;
    public static int RNG_c=1013904223;

    public int my_rand ()
    {
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
    }
}
```

⁸О разнице между указателями и *reference* в C++: 52.3 (стр. 555).

⁹Или «экземпляр класса» в некоторой русскоязычной литературе.

55.8. УСЛОВНЫЕ ПЕРЕХОДЫ

Здесь пара полей класса, которые инициализируются в начале. Но как? В выводе `javap` мы можем найти конструктор класса:

```
static {};  
  flags: ACC_STATIC  
  Code:  
    stack=1, locals=0, args_size=0  
      0: ldc          #5              // int 1664525  
      2: putstatic     #3              // Field RNG_a:I  
      5: ldc          #6              // int 1013904223  
      7: putstatic     #4              // Field RNG_c:I  
     10: return
```

Так инициализируются переменные. `RNG_a` занимает третий слот в классе и `RNG_c` — четвертый, и `putstatic` записывает туда константы.

Функция `my_srand()` просто записывает входное значение в `rand_state`:

```
public void my_srand(int);  
  flags: ACC_PUBLIC  
  Code:  
    stack=1, locals=2, args_size=2  
      0: iload_1  
      1: putstatic     #2              // Field rand_state:I  
      4: return
```

`iload_1` берет входное значение и заталкивает его в стек. Но почему не `iload_0`? Это потому что эта функция может использовать поля класса, а переменная `this` также передается в эту функцию как нулевой аргумент. Поле `rand_state` занимает второй слот в классе, так что `putstatic` копирует переменную из `TOS` во второй слот.

Теперь `my_rand()`:

```
public int my_rand();  
  flags: ACC_PUBLIC  
  Code:  
    stack=2, locals=1, args_size=1  
      0: getstatic     #2              // Field rand_state:I  
      3: getstatic     #3              // Field RNG_a:I  
      6: imul  
      7: putstatic     #2              // Field rand_state:I  
     10: getstatic     #2              // Field rand_state:I  
     13: getstatic     #4              // Field RNG_c:I  
     16: iadd  
     17: putstatic     #2              // Field rand_state:I  
     20: getstatic     #2              // Field rand_state:I  
     23: sipush        32767  
     26: iand  
     27: ireturn
```

Она просто загружает все переменные из полей объекта, производит с ними операции и обновляет значение `rand_state`, используя инструкцию `putstatic`. По смещению 20, значение `rand_state` перезагружается снова (это потому что оно было выброшено из стека перед этим, инструкцией `putstatic`). Это выглядит как неэффективный код, но можете быть уверенными, `JVM` обычно достаточно хорош, чтобы хорошо оптимизировать подобные вещи.

55.8. Условные переходы

Перейдем к условным переходам.

```
public class abs  
{  
    public static int abs(int a)  
    {  
        if (a<0)  
            return -a;  
        return a;  
    }  
}
```

55.8. УСЛОВНЫЕ ПЕРЕХОДЫ

```
}  
}
```

```
public static int abs(int);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=1, locals=1, args_size=1  
    0: iload_0  
    1: ifge      7  
    4: iload_0  
    5: ineg  
    6: ireturn  
    7: iload_0  
    8: ireturn
```

`ifge` переходит на смещение 7 если значение на [TOS](#) больше или равно 0. Не забывайте, любая инструкция `ifXX` выталкивает значение (с которым будет производиться сравнение) из стека.

`ineg` просто меняет знак значения на [TOS](#).

Еще пример:

```
public static int min (int a, int b)  
{  
    if (a>b)  
        return b;  
    return a;  
}
```

Получаем:

```
public static int min(int, int);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=2, locals=2, args_size=2  
    0: iload_0  
    1: iload_1  
    2: if_icmple  7  
    5: iload_1  
    6: ireturn  
    7: iload_0  
    8: ireturn
```

`if_icmple` выталкивает два значения и сравнивает их. Если второе меньше первого (или равно), происходит переход на смещение 7.

Когда мы определяем функцию `max()` ...

```
public static int max (int a, int b)  
{  
    if (a>b)  
        return a;  
    return b;  
}
```

...итоговый код точно такой же, только последние инструкции `iload` (на смещениях 5 и 7) поменены местами:

```
public static int max(int, int);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=2, locals=2, args_size=2  
    0: iload_0  
    1: iload_1  
    2: if_icmple  7  
    5: iload_0  
    6: ireturn  
    7: iload_1  
    8: ireturn
```

Более сложный пример:

```
public class cond
{
    public static void f(int i)
    {
        if (i<100)
            System.out.print("<100");
        if (i==100)
            System.out.print("==100");
        if (i>100)
            System.out.print(">100");
        if (i==0)
            System.out.print("==0");
    }
}
```

```
public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
     0: iload_0
     1: bipush      100
     3: if_icmpge    14
     6: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
     9: ldc          #3          // String <100
    11: invokevirtual #4          // Method java/io/PrintStream.print:(Ljava/lang/String;)V
    14: iload_0
    15: bipush      100
    17: if_icmpne    28
    20: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
    23: ldc          #5          // String ==100
    25: invokevirtual #4          // Method java/io/PrintStream.print:(Ljava/lang/String;)V
    28: iload_0
    29: bipush      100
    31: if_icmple    42
    34: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
    37: ldc          #6          // String >100
    39: invokevirtual #4          // Method java/io/PrintStream.print:(Ljava/lang/String;)V
    42: iload_0
    43: ifne         54
    46: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
    49: ldc          #7          // String ==0
    51: invokevirtual #4          // Method java/io/PrintStream.print:(Ljava/lang/String;)V
    54: return
```

`if_icmpge` Вытаскивает два значения и сравнивает их. Если второй больше первого, происходит переход на смещение 14. `if_icmpne` и `if_icmple` работают одинаково, но используются разные условия.

По смещению 43 есть также инструкция `ifne`. Название неудачное, её было бы лучше назвать `ifnz` (переход если переменная на [TOS](#) не равна нулю). И вот что она делает: производит переход на смещение 54, если входное значение не ноль. Если ноль, управление передается на смещение 46, где выводится строка «==0».

Н.В.: В [JVM](#) нет беззнаковых типов данных, так что инструкции сравнения работают только с знаковыми целочисленными значениями.

55.9. Передача аргументов

Теперь расширим пример `min()/max()`:

```
public class minmax
{
    public static int min (int a, int b)
```

```

    {
        if (a>b)
            return b;
        return a;
    }

    public static int max (int a, int b)
    {
        if (a>b)
            return a;
        return b;
    }

    public static void main(String[] args)
    {
        int a=123, b=456;
        int max_value=max(a, b);
        int min_value=min(a, b);
        System.out.println(min_value);
        System.out.println(max_value);
    }
}

```

Вот код функции `main()`:

```

public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=5, args_size=1
     0: bipush      123
     2: istore_1
     3: sipush      456
     6: istore_2
     7: iload_1
     8: iload_2
     9: invokestatic #2          // Method max:(II)I
    12: istore_3
    13: iload_1
    14: iload_2
    15: invokestatic #3          // Method min:(II)I
    18: istore      4
    20: getstatic   #4          // Field java/lang/System.out:Ljava/io/PrintStream;
    23: iload      4
    25: invokevirtual #5        // Method java/io/PrintStream.println:(I)V
    28: getstatic   #4          // Field java/lang/System.out:Ljava/io/PrintStream;
    31: iload_3
    32: invokevirtual #5        // Method java/io/PrintStream.println:(I)V
    35: return

```

В другую функцию аргументы передаются передаются в стеке, а возвращаемое значение остается на [TOS](#).

55.10. Битовые поля

Все побитовые операции работают также, как и в любой другой [ISA](#):

```

    public static int set (int a, int b)
    {
        return a | 1<<b;
    }

    public static int clear (int a, int b)
    {
        return a & ~(1<<b);
    }

```

```

public static int set(int, int);
  flags: ACC_PUBLIC, ACC_STATIC

```

55.10. БИТОВЫЕ ПОЛЯ

```
Code:
  stack=3, locals=2, args_size=2
  0: iload_0
  1: iconst_1
  2: iload_1
  3: ishl
  4: ior
  5: ireturn
```

```
public static int clear(int, int);
  flags: ACC_PUBLIC, ACC_STATIC
```

```
Code:
  stack=3, locals=2, args_size=2
  0: iload_0
  1: iconst_1
  2: iload_1
  3: ishl
  4: iconst_m1
  5: ixor
  6: iand
  7: ireturn
```

`iconst_m1` загружает `-1` в стек, это то же что и значение `0xFFFFFFFF`. Операция XOR с `0xFFFFFFFF` в одном из операндов, это тот же эффект что инвертирование всех бит (?? (стр. ??)).

Попробуем также расширить все типы данных до 64-битного *long*:

```
public static long lset (long a, int b)
{
    return a | 1<<b;
}

public static long lclear (long a, int b)
{
    return a & ~(1<<b);
}
```

```
public static long lset(long, int);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
```

```
  stack=4, locals=3, args_size=2
  0: lload_0
  1: iconst_1
  2: iload_2
  3: ishl
  4: i2l
  5: lor
  6: lreturn
```

```
public static long lclear(long, int);
  flags: ACC_PUBLIC, ACC_STATIC
```

```
Code:
  stack=4, locals=3, args_size=2
  0: lload_0
  1: iconst_1
  2: iload_2
  3: ishl
  4: iconst_m1
  5: ixor
  6: i2l
  7: land
  8: lreturn
```

Код такой же, но используются инструкции с префиксом *l*, которые работают с 64-битными значениями. Так же, второй аргумент функции все еще имеет тип *int*, и когда 32-битное число в нем должно быть расширено до 64-битного значения, используется инструкция `i2l`, которая расширяет значение типа *integer* в значение типа *long*.

55.11. Циклы

```
public class Loop
{
    public static void main(String[] args)
    {
        for (int i = 1; i <= 10; i++)
        {
            System.out.println(i);
        }
    }
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2, args_size=1
        0: iconst_1
        1: istore_1
        2: iload_1
        3: bipush      10
        5: if_icmpgt    21
        8: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
       11: iload_1
       12: invokevirtual #3          // Method java/io/PrintStream.println:(I)V
       15: iinc         1, 1
       18: goto        2
       21: return
```

`iconst_1` загружает 1 в [TOS](#), `istore_1` сохраняет её в первом слоте [LVA](#). Почему не нулевой слот? Потому что функция `main()` имеет один аргумент (массив `String`), и указатель на него (или *reference*) сейчас в нулевом слоте.

Так что локальная переменная `i` всегда будет в первом слоте.

Инструкции по смещениям 3 и 5 сравнивают `i` с 10. Если `i` больше, управление передается на смещение 21, где функция заканчивает работу. Если нет, вызывается `println`. `i` перезагружается по смещению 11, для `println`. Кстати, мы вызываем метод `println` для типа данных *integer*, и мы видим это в комментариях: «(I)V» (*I* означает *integer* и *V* означает что возвращаемое значение имеет тип *void*).

Когда `println` заканчивается, `i` увеличивается на 1 по смещению 15. Первый операнд инструкции это номер слота (1), второй это число (1) для прибавления.

`goto` это просто GOTO, она переходит на начало цикла по смещению 2.

Перейдем к более сложному примеру:

```
public class Fibonacci
{
    public static void main(String[] args)
    {
        int limit = 20, f = 0, g = 1;

        for (int i = 1; i <= limit; i++)
        {
            f = f + g;
            g = f - g;
            System.out.println(f);
        }
    }
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=5, args_size=1
        0: bipush      20
        2: istore_1
        3: iconst_0
```

55.12. SWITCH()

```
4: istore_2
5: iconst_1
6: istore_3
7: iconst_1
8: istore      4
10: iload      4
12: iload_1
13: if_icmpgt   37
16: iload_2
17: iload_3
18: iadd
19: istore_2
20: iload_2
21: iload_3
22: isub
23: istore_3
24: getstatic   #2           // Field java/lang/System.out:Ljava/io/PrintStream;
27: iload_2
28: invokevirtual #3           // Method java/io/PrintStream.println:(I)V
31: iinc        4, 1
34: goto        10
37: return
```

Вот карта слотов в [LVA](#):

- 0 – единственный аргумент функции `main()`
- 1 – *limit*, всегда содержит 20
- 2 – *f*
- 3 – *g*
- 4 – *i*

Мы видим что компилятор Java расположил переменные в слотах [LVA](#) в точно таком же порядке, в котором переменные были определены в исходном коде.

Существуют отдельные инструкции `istore` для слотов 0, 1, 2, 3, но не 4 и более, так что здесь есть `istore` с дополнительным операндом по смещению 8, которая имеет номер слота в операнде. Та же история с `iload` по смещению 10.

Но не слишком ли это сомнительно, выделить целый слот для переменной *limit*, которая всегда содержит 20 (так что это по сути константа), и перезагружать её так часто? JIT-компилятор в [JVM](#) обычно достаточно хорош, чтобы всё это оптимизировать. Самостоятельное вмешательство в код, наверное, того не стоит.

55.12. switch()

Выражение `switch()` реализуется инструкцией `tableswitch`:

```
public static void f(int a)
{
    switch (a)
    {
        case 0: System.out.println("zero"); break;
        case 1: System.out.println("one\n"); break;
        case 2: System.out.println("two\n"); break;
        case 3: System.out.println("three\n"); break;
        case 4: System.out.println("four\n"); break;
        default: System.out.println("something unknown\n"); break;
    };
}
```

Проще не бывает:

```
public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
```

```

0: iload_0
1: tableswitch { // 0 to 4
    0: 36
    1: 47
    2: 58
    3: 69
    4: 80
    default: 91
}
36: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
39: ldc          #3          // String zero
41: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
↳ ;)V
44: goto         99
47: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
50: ldc          #5          // String one\n
52: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
↳ ;)V
55: goto         99
58: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
61: ldc          #6          // String two\n
63: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
↳ ;)V
66: goto         99
69: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
72: ldc          #7          // String three\n
74: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
↳ ;)V
77: goto         99
80: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
83: ldc          #8          // String four\n
85: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
↳ ;)V
88: goto         99
91: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
94: ldc          #9          // String something unknown\n
96: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
↳ ;)V
99: return

```

55.13. Массивы

55.13.1. Простой пример

Создадим массив из 10-и чисел и заполним его:

```

public static void main(String[] args)
{
    int a[]=new int[10];
    for (int i=0; i<10; i++)
        a[i]=i;
    dump (a);
}

```

```

public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=3, args_size=1
    0: bipush      10
    2: newarray    int
    4: astore_1
    5: iconst_0
    6: istore_2
    7: iload_2
    8: bipush      10
   10: if_icmpge   23

```

55.13. МАССИВЫ

```
13: aload_1
14: iload_2
15: iload_2
16: iastore
17: iinc      2, 1
20: goto      7
23: aload_1
24: invokestatic #4          // Method dump:([I)V
27: return
```

Инструкция `newarray` создает объект массива из 10 элементов типа `int`. Размер массива выставляется инструкцией `bipush` и остается на `TOS`. Тип массива выставляется в операнде инструкции `newarray`. После исполнения `newarray`, `reference` (или указатель) только что созданного в куче (heap) массива остается на `TOS`. `astore_1` сохраняет `reference` на него в первом слоте `LVA`. Вторая часть функции `main()` это цикл, сохраняющий значение `i` в соответствующий элемент массива. `aload_1` берет `reference` массива и сохраняет его в стеке. `iastore` затем сохраняет значение из стека в массив, `reference` на который в это время находится на `TOS`. Третья часть функции `main()` вызывает функцию `dump()`. Аргумент для нее готовится инструкцией `aload_1` (смещение 23).

Перейдем к функции `dump()`:

```
public static void dump(int a[])
{
    for (int i=0; i<a.length; i++)
        System.out.println(a[i]);
}
```

```
public static void dump(int[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=2, args_size=1
     0: iconst_0
     1: istore_1
     2: iload_1
     3: aload_0
     4: arraylength
     5: if_icmpge    23
     8: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
    11: aload_0
    12: iload_1
    13: iaload
    14: invokevirtual #3          // Method java/io/PrintStream.println:(I)V
    17: iinc         1, 1
    20: goto         2
    23: return
```

Входящий `reference` на массив в нулевом слоте. Выражение `a.length` в исходном коде конвертируется в инструкцию `arraylength`, она берет `reference` на массив и оставляет размер массива на `TOS`. Инструкция `iaload` по смещению 13 используется для загрузки элементов массива, она требует чтобы в стеке присутствовал `reference` на массив (подготовленный `aload_0` на 11), а также индекс (подготовленный `iload_1` по смещению 12).

Нужно сказать что инструкции с префиксом `a` могут быть неверно поняты, как инструкции работающие с массивами (`array`). Это неверно. Эти инструкции работают с `reference`-ами на объекты. А массивы и строки это тоже объекты.

55.13.2. Суммирование элементов массива

Еще один пример:

```
public class ArraySum
{
    public static int f (int[] a)
    {
        int sum=0;
        for (int i=0; i<a.length; i++)
            sum=sum+a[i];
    }
}
```

55.13. МАССИВЫ

```
        return sum;
    }
}
```

```
public static int f(int[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=3, args_size=1
     0: iconst_0
     1: istore_1
     2: iconst_0
     3: istore_2
     4: iload_2
     5: aload_0
     6: arraylength
     7: if_icmpge      22
    10: iload_1
    11: aload_0
    12: iload_2
    13: iaload
    14: iadd
    15: istore_1
    16: iinc           2, 1
    19: goto          4
    22: iload_1
    23: ireturn
```

Нулевой слот в [LVA](#) содержит указатель (*reference*) на входной массив. Первый слот [LVA](#) содержит локальную переменную *sum*.

55.13.3. Единственный аргумент `main()` это также массив

Будем использовать единственный аргумент `main()`, который массив строк:

```
public class UseArgument
{
    public static void main(String[] args)
    {
        System.out.print("Hi, ");
        System.out.print(args[1]);
        System.out.println(". How are you?");
    }
}
```

Нулевой аргумент это имя программы (как в Си/Си++, итд), так что первый аргумент это тот, что пользователь добавил первым.

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=1, args_size=1
     0: getstatic      #2          // Field java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #3          // String Hi,
     5: invokevirtual #4          // Method java/io/PrintStream.print:(Ljava/lang/String;)V
     8: getstatic      #2          // Field java/lang/System.out:Ljava/io/PrintStream;
    11: aload_0
    12: iconst_1
    13: aaload
    14: invokevirtual #4          // Method java/io/PrintStream.print:(Ljava/lang/String;)V
    17: getstatic      #2          // Field java/lang/System.out:Ljava/io/PrintStream;
    20: ldc           #5          // String . How are you?
    22: invokevirtual #6          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    25: return
```

55.13. МАССИВЫ

`aload_0` на 11 загружают *reference* на нулевой слот `LVA` (первый и единственный аргумент `main()`). `iconst_1` и `aaload` на 12 и 13 берут *reference* на первый (считая с 0) элемент массива. *Reference* на строковый объект на `TOS` по смещению 14, и оттуда он берется методом `println`.

55.13.4. Заранее инициализированный массив строк

```
class Month
{
    public static String[] months =
    {
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
    };

    public String get_month (int i)
    {
        return months[i];
    };
}
```

```
public java.lang.String get_month(int);
flags: ACC_PUBLIC
Code:
    stack=2, locals=2, args_size=2
        0: getstatic      #2          // Field months:[Ljava/lang/String;
        3: iload_1
        4: aaload
        5: areturn
```

`aaload` работает с массивом *reference*-ов. Строка в Java это объект, так что используются *a*-инструкции для работы с ними. `areturn` возвращает *reference* на объект `String`.

Как инициализируется массив `months[]`?

```
static {};
flags: ACC_STATIC
Code:
    stack=4, locals=0, args_size=0
        0: bipush      12
        2: anewarray   #3          // class java/lang/String
        5: dup
        6: iconst_0
        7: ldc        #4          // String January
        9: astore
       10: dup
       11: iconst_1
       12: ldc        #5          // String February
       14: astore
       15: dup
       16: iconst_2
       17: ldc        #6          // String March
       19: astore
       20: dup
       21: iconst_3
       22: ldc        #7          // String April
       24: astore
       25: dup
```

55.13. МАССИВЫ

```
26: iconst_4
27: ldc          #8          // String May
29: astore
30: dup
31: iconst_5
32: ldc          #9          // String June
34: astore
35: dup
36: bipush       6
38: ldc          #10         // String July
40: astore
41: dup
42: bipush       7
44: ldc          #11         // String August
46: astore
47: dup
48: bipush       8
50: ldc          #12         // String September
52: astore
53: dup
54: bipush       9
56: ldc          #13         // String October
58: astore
59: dup
60: bipush      10
62: ldc          #14         // String November
64: astore
65: dup
66: bipush      11
68: ldc          #15         // String December
70: astore
71: putstatic    #2          // Field months:[Ljava/lang/String;
74: return
```

`anewarray` создает новый массив *reference*-ов (отсюда префикс *a*). Тип объекта определяется в операнде `anewarray`, там текстовая строка «`java/lang/String`». `bipush 12` перед `anewarray` устанавливает размер массива. Новая для нас здесь инструкция: `dup`. Это очень известная инструкция в стековых компьютерах (включая ЯП Forth), которая делает дубликат значения на `TOS`. Она используется здесь для дублирования *reference*-а на массив, потому что инструкция `aastore` выталкивает из стека *reference* на массив, но последующая инструкция `aastore` снова нуждается в нем. Компилятор Java решил что лучше генерировать `dup` вместо генерации инструкции `getstatic` перед каждой операцией записи в массив (т.е. 11 раз).

`aastore` кладет *reference* (на строку) в массив по индексу взятому из `TOS`.

И наконец, `putstatic` кладет *reference* на только что созданный массив во второе поле нашего объекта, т.е. в поле *months*.

55.13.5. Функции с переменным кол-вом аргументов (variadic)

Функции с переменным кол-вом аргументов (variadic) на самом деле используют массивы:

```
public static void f(int... values)
{
    for (int i=0; i<values.length; i++)
        System.out.println(values[i]);
}

public static void main(String[] args)
{
    f (1,2,3,4,5);
}
```

```
public static void f(int...);
flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
Code:
    stack=3, locals=2, args_size=1
```

55.13. МАССИВЫ

```
0: iconst_0
1: istore_1
2: iload_1
3: aload_0
4: arraylength
5: if_icmpge    23
8: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;
11: aload_0
12: iload_1
13: iaload
14: invokevirtual #3          // Method java/io/PrintStream.println:(I)V
17: iinc         1, 1
20: goto         2
23: return
```

По смещению 3, `f()` просто берет массив переменных используя `aload_0`. Затем берет размер массива, итд.

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=1, args_size=1
      0: iconst_5
      1: newarray     int
      3: dup
      4: iconst_0
      5: iconst_1
      6: iastore
      7: dup
      8: iconst_1
      9: iconst_2
     10: iastore
     11: dup
     12: iconst_2
     13: iconst_3
     14: iastore
     15: dup
     16: iconst_3
     17: iconst_4
     18: iastore
     19: dup
     20: iconst_4
     21: iconst_5
     22: iastore
     23: invokestatic #4          // Method f:([I)V
     26: return
```

Массив конструируется в `main()` используя инструкцию `newarray`, затем он заполняется, и вызывается `f()`.

Кстати, объект массива не уничтожается в конце `main()`. В Java вообще нет деструкторов, потому что в JVM есть сборщик мусора (garbage collector), делающий это автоматически, когда считает нужным.

Как насчет метода `format()`? Он берет на вход два аргумента: строку и массив объектов:

```
public PrintStream format(String format, Object... args)
```

(<http://docs.oracle.com/javase/tutorial/java/data/numberformat.html>)

Посмотрим:

```
public static void main(String[] args)
{
    int i=123;
    double d=123.456;
    System.out.format("int: %d double: %f.%n", i, d);
}
```

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
```


55.13. МАССИВЫ

```
stack=7, locals=4, args_size=1
  0: bipush      123
  2: istore_1
  3: ldc2_w      #2          // double 123.456d
  6: dstore_2
  7: getstatic    #4          // Field java/lang/System.out:Ljava/io/PrintStream;
 10: ldc          #5          // String int: %d double: %f.%n
 12: iconst_2
 13: anewarray    #6          // class java/lang/Object
 16: dup
 17: iconst_0
 18: iload_1
 19: invokestatic #7          // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
↳ Integer;
 22: astore
 23: dup
 24: iconst_1
 25: dload_2
 26: invokestatic #8          // Method java/lang/Double.valueOf:(D)Ljava/lang/Double;
 29: astore
 30: invokevirtual #9          // Method java/io/PrintStream.format:(Ljava/lang/String;[Ljava/lang/Object;)Ljava/io/PrintStream;
↳ ;[Ljava/lang/Object;)Ljava/io/PrintStream;
 33: pop
 34: return
```

Так что в начале значения типов *int* и *double* конвертируются в объекты типов `Integer` и `Double` используя методы `valueOf`. Метод `format()` требует на входе объекты типа `Object`, а так как классы `Integer` и `Double` наследуются от корневого класса `Object`, они подходят как элементы во входном массиве. С другой стороны, массив всегда гомогенный, т.е. он не может содержать элементы разных типов, что делает невозможным хранение там значений типов *int* и *double*.

Массив объектов `Object` создается по смещению 13, объект `Integer` добавляется в массив по смещению 22, объект `Double` добавляется в массив по смещению 29.

Предпоследняя инструкция `pop` удаляет элемент на `TOS`, так что в момент исполнения `return`, стек пуст (или сбалансирован).

55.13.6. Двухмерные массивы

Двухмерные массивы в Java это просто одномерные массивы *reference*-в на другие одномерные массивы.

Создадим двухмерный массив:

```
public static void main(String[] args)
{
    int[][] a = new int[5][10];
    a[1][2]=3;
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=2, args_size=1
   0: iconst_5
   1: bipush      10
   3: multianewarray #2,  2          // class "[[I"
   7: astore_1
   8: aload_1
   9: iconst_1
  10: aaload
  11: iconst_2
  12: iconst_3
  13: iastore
  14: return
```

55.13. МАССИВЫ

Он создается при помощи инструкции `multianewarray`: тип объекта и размерность передаются в операндах. Размер массива (10*5) остается в стеке (используя инструкции `iconst_5` и `bipush`).

Reference на строку #1 загружается по смещению 10 (`iconst_1` и `aaload`). Выборка столбца происходит используя инструкцию `iconst_2` по смещению 11. Значение для записи устанавливается по смещению 12. `iastore` на 13 записывает элемент массива.

Как его прочитать?

```
public static int get12 (int[][] in)
{
    return in[1][2];
}
```

```
public static int get12(int[][]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=1, args_size=1
    0: aload_0
    1: iconst_1
    2: aaload
    3: iconst_2
    4: iaload
    5: ireturn
```

Reference на строку массива загружается по смещению 2, столбец устанавливается по смещению 3, `iaload` загружает элемент массива.

55.13.7. Трехмерные массивы

Трехмерные массивы это просто одномерные массивы *reference*-ов на одномерные массивы *reference*-ов.

```
public static void main(String[] args)
{
    int[][][] a = new int[5][10][15];

    a[1][2][3]=4;

    get_elem(a);
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=2, args_size=1
    0: iconst_5
    1: bipush      10
    3: bipush      15
    5: multianewarray #2, 3          // class "[[[I"
    9: astore_1
   10: aload_1
   11: iconst_1
   12: aaload
   13: iconst_2
   14: aaload
   15: iconst_3
   16: iconst_4
   17: iastore
   18: aload_1
   19: invokestatic #3              // Method get_elem:([[[I)I
   22: pop
   23: return
```

Чтобы найти нужный *reference*, теперь нужно две инструкции `aaload`:

```
public static int get_elem (int[][][] a)
{
    return a[1][2][3];
}
```

```
public static int get_elem(int[][][]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=1, args_size=1
    0: aload_0
    1: iconst_1
    2: aaload
    3: iconst_2
    4: aaload
    5: iconst_3
    6: iaload
    7: ireturn
```

55.13.8. Итоги

Возможно ли сделать переполнение буфера в Java? Нет, потому что длина массива всегда присутствует в объекте массива, границы массива контролируются и при попытке выйти за границы, сработает исключение.

В Java нет многомерных массивов в том смысле, как в Си/Си++, так что Java не очень подходит для быстрых научных вычислений.

55.14. Строки

55.14.1. Первый пример

Строки это объекты, и конструируются так же как и другие объекты (и массивы).

```
public static void main(String[] args)
{
    System.out.println("What is your name?");
    String input = System.console().readLine();
    System.out.println("Hello, "+input);
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=2, args_size=1
    0: getstatic      #2          // Field java/lang/System.out:Ljava/io/PrintStream;
    3: ldc           #3          // String What is your name?
    5: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    8: invokestatic  #5          // Method java/lang/System.console:()Ljava/io/Console;
   11: invokevirtual #6          // Method java/io/Console.readLine:()Ljava/lang/String;
   14: astore_1
   15: getstatic      #2          // Field java/lang/System.out:Ljava/io/PrintStream;
   18: new           #7          // class java/lang/StringBuilder
   21: dup
   22: invokespecial #8          // Method java/lang/StringBuilder."<init>":()V
   25: ldc           #9          // String Hello,
   27: invokevirtual #10         // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   30: aload_1
   31: invokevirtual #10         // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   34: invokevirtual #11         // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   37: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   40: ;)V
```

```
40: return
```

Метод `readLine()` вызывается по смещению 11, *reference* на строку (введенную пользователем) остается на [TOS](#). По смещению 14, *reference* на строку сохраняется в первом слоте [LVA](#). Строка введенная пользователем перезагружается по смещению 30 и складывается со строкой «Hello, » используя класс `StringBuilder`. Сконструированная строка затем выводится используя метод `println` по смещению 37.

55.14.2. Второй пример

Еще один пример:

```
public class strings
{
    public static char test (String a)
    {
        return a.charAt(3);
    };

    public static String concat (String a, String b)
    {
        return a+b;
    }
}
```

```
public static char test(java.lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
        0: aload_0
        1: iconst_3
        2: invokevirtual #2          // Method java/lang/String.charAt:(I)C
        5: ireturn
```

Складывание строк происходит при помощи класса `StringBuilder`:

```
public static java.lang.String concat(java.lang.String, java.lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2, args_size=2
        0: new          #3          // class java/lang/StringBuilder
        3: dup
        4: invokespecial #4          // Method java/lang/StringBuilder."<init>":()V
        7: aload_0
        8: invokevirtual #5          // Method java/lang/StringBuilder.append:(Ljava/lang/
↳ String;)Ljava/lang/StringBuilder;
       11: aload_1
       12: invokevirtual #5          // Method java/lang/StringBuilder.append:(Ljava/lang/
↳ String;)Ljava/lang/StringBuilder;
       15: invokevirtual #6          // Method java/lang/StringBuilder.toString:()Ljava/lang/
↳ String;
       18: areturn
```

Еще пример:

```
public static void main(String[] args)
{
    String s="Hello!";
    int n=123;
    System.out.println("s=" + s + " n=" + n);
}
```

И снова, строки создаются используя класс `StringBuilder` и его метод `append`, затем сконструированная строка передается в метод `println`:

```

public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=3, args_size=1
      0: ldc          #2             // String Hello!
      2: astore_1
      3: bipush        123
      5: istore_2
      6: getstatic     #3             // Field java/lang/System.out:Ljava/io/PrintStream;
      9: new           #4             // class java/lang/StringBuilder
     12: dup
     13: invokespecial #5             // Method java/lang/StringBuilder."<init>":()V
     16: ldc          #6             // String s=
     18: invokevirtual #7             // Method java/lang/StringBuilder.append:(Ljava/lang/
↳ String;)Ljava/lang/StringBuilder;
     21: aload_1
     22: invokevirtual #7             // Method java/lang/StringBuilder.append:(Ljava/lang/
↳ String;)Ljava/lang/StringBuilder;
     25: ldc          #8             // String n=
     27: invokevirtual #7             // Method java/lang/StringBuilder.append:(Ljava/lang/
↳ String;)Ljava/lang/StringBuilder;
     30: iload_2
     31: invokevirtual #9             // Method java/lang/StringBuilder.append:(I)Ljava/lang/
↳ StringBuilder;
     34: invokevirtual #10            // Method java/lang/StringBuilder.toString:()Ljava/lang/
↳ String;
     37: invokevirtual #11            // Method java/io/PrintStream.println:(Ljava/lang/String
↳ ;)V
     40: return

```

55.15. Исключения

Немного переделаем пример *Month* (55.13.4 (стр. 624)):

Листинг 55.10: IncorrectMonthException.java

```

public class IncorrectMonthException extends Exception
{
    private int index;

    public IncorrectMonthException(int index)
    {
        this.index = index;
    }
    public int getIndex()
    {
        return index;
    }
}

```

Листинг 55.11: Month2.java

```

class Month2
{
    public static String[] months =
    {
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
    }
}

```

```

        "December"
    };

    public static String get_month (int i) throws IncorrectMonthException
    {
        if (i<0 || i>11)
            throw new IncorrectMonthException(i);
        return months[i];
    };

    public static void main (String[] args)
    {
        try
        {
            System.out.println(get_month(100));
        }
        catch(IncorrectMonthException e)
        {
            System.out.println("incorrect month index: "+ e.getIndex());
            e.printStackTrace();
        }
    };
}

```

Котормко говоря, `IncorrectMonthException.class` имеет только конструктор объетка и один метод-акцессор.

Класс `IncorrectMonthException` наследуется от `Exception`, так что конструктор `IncorrectMonthException` в начале вызывает конструктор класса `Exception`, затем он перекладывает входящее значение в единственное поле класса `IncorrectMonthException`:

```

public IncorrectMonthException(int);
flags: ACC_PUBLIC
Code:
    stack=2, locals=2, args_size=2
    0: aload_0
    1: invokespecial #1           // Method java/lang/Exception."<init>":()V
    4: aload_0
    5: iload_1
    6: putfield      #2           // Field index:I
    9: return

```

`getIndex()` это просто акцессор. *Reference* (указатель) на `IncorrectMonthException` передается в нулевом слоте *LVA* (*this*), `aload_0` берет его, `getfield` загружает значение из объекта, `ireturn` возвращает его.

```

public int getIndex();
flags: ACC_PUBLIC
Code:
    stack=1, locals=1, args_size=1
    0: aload_0
    1: getfield      #2           // Field index:I
    4: ireturn

```

Посмотрим на `get_month()` в `Month2.class`:

Листинг 55.12: Month2.class

```

public static java.lang.String get_month(int) throws IncorrectMonthException;
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=1, args_size=1
    0: iload_0
    1: iflt          10
    4: iload_0
    5: bipush        11
    7: if_icmple     19
   10: new           #2           // class IncorrectMonthException
   13: dup
   14: iload_0

```

55.15. ИСКЛЮЧЕНИЯ

```

15: invokespecial #3          // Method IncorrectMonthException."<init>":(I)V
18: athrow
19: getstatic      #4          // Field months:[Ljava/lang/String;
22: iload_0
23: aaload
24: areturn

```

`iflt` по смещению 1 это *if less than* (если меньше, чем).

В случае неправильного индекса, создается новый объект при помощи инструкции `new` по смещению 10. Тип объекта передается как операнд инструкции (и это `IncorrectMonthException`). Затем вызывается его конструктор, в который передается индекс (через `TOS`) (по смещению 15). В то время как управление находится на смещении 18, объект уже создан, теперь инструкция `athrow` берет указатель (*reference*) на только что созданный объект и сигнализирует в `JVM`, чтобы тот нашел подходящий обработчик исключения.

Инструкция `athrow` не возвращает управление сюда, так что по смещению 19 здесь совсем другой *basic block*, не имеющий отношения к исключениям, сюда можно попасть со смещения 7.

Как работает обработчик? Take a look atПосмотрим на `main()` в `Month2.class`:

Листинг 55.13: Month2.class

```

public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=1
      0: getstatic      #5          // Field java/lang/System.out:Ljava/io/PrintStream;
      3: bipush         100
      5: invokestatic   #6          // Method get_month:(I)Ljava/lang/String;
      8: invokevirtual  #7          // Method java/io/PrintStream.println:(Ljava/lang/String)V
    ↪ ;)V
     11: goto          47
     14: astore_1
     15: getstatic      #5          // Field java/lang/System.out:Ljava/io/PrintStream;
     18: new            #8          // class java/lang/StringBuilder
     21: dup
     22: invokespecial  #9          // Method java/lang/StringBuilder."<init>":()V
     25: ldc            #10         // String incorrect month index:
     27: invokevirtual  #11         // Method java/lang/StringBuilder.append:(Ljava/lang/
    ↪ String;)Ljava/lang/StringBuilder;
     30: aload_1
     31: invokevirtual  #12         // Method IncorrectMonthException.getIndex:()I
     34: invokevirtual  #13         // Method java/lang/StringBuilder.append:(I)Ljava/lang/
    ↪ StringBuilder;
     37: invokevirtual  #14         // Method java/lang/StringBuilder.toString:()Ljava/lang/
    ↪ String;
     40: invokevirtual  #7          // Method java/io/PrintStream.println:(Ljava/lang/String)V
    ↪ ;)V
     43: aload_1
     44: invokevirtual  #15         // Method IncorrectMonthException.printStackTrace:()V
     47: return
  Exception table:
    from   to   target type
      0     11   14     Class IncorrectMonthException

```

Тут есть `Exception table`, которая определяет, что между смещениями 0 и 11 (включительно) может случиться исключение `IncorrectMonthException`, и если это произойдет, то нужно передать управление на смещение 14. Действительно, основная программа заканчивается на смещении 11. По смещению 14 начинается обработчик, и сюда невозможно попасть, здесь нет никаких условных/безусловных переходов в эту область. Но `JVM` передаст сюда управление в случае исключения. Самая первая `astore_1` (на 14) берет входящий указатель (*reference*) на объект исключения и сохраняет его в слоте 1 `LVA`. Позже, по смещению 31 будет вызван метод этого объекта (`getIndex()`). Указатель *reference* на текущий объект исключения передан немного раньше (смещение 30). Остальной код это просто код для манипуляции со строками: в начале значение возвращенное методом `getIndex()` конвертируется в строку используя метод `toString()`, затем эта строка прибавляется к текстовой строке «incorrect month index:» (как мы уже рассматривали ранее), затем вызываются `println()` и `printStackTrace()`. После того как `printStackTrace()` заканчивается, исключение уже обработано, мы можем возвращаться к нормальной работе. По смещению 47 есть `return`, который

55.16. КЛАССЫ

заканчивает работу функции `main()`, но там может быть любой другой код, который исполнится, если исключения не произошло.

Вот пример, как IDA показывает интервалы исключений:

Листинг 55.14: из какого-то случайного найденного на компьютере автора .class-файла

```
.catch java/io/FileNotFoundException from met001_335 to met001_360\
using met001_360
.catch java/io/FileNotFoundException from met001_185 to met001_214\
using met001_214
.catch java/io/FileNotFoundException from met001_181 to met001_192\
using met001_195
.catch java/io/FileNotFoundException from met001_155 to met001_176\
using met001_176
.catch java/io/FileNotFoundException from met001_83 to met001_129 using \
met001_129
.catch java/io/FileNotFoundException from met001_42 to met001_66 using \
met001_69
.catch java/io/FileNotFoundException from met001_begin to met001_37\
using met001_37
```

55.16. Классы

Простой класс:

Листинг 55.15: test.java

```
public class test
{
    public static int a;
    private static int b;

    public test()
    {
        a=0;
        b=0;
    }
    public static void set_a (int input)
    {
        a=input;
    }
    public static int get_a ()
    {
        return a;
    }
    public static void set_b (int input)
    {
        b=input;
    }
    public static int get_b ()
    {
        return b;
    }
}
```

Конструктор просто выставляет оба поля класса в нули:

```
public test();
flags: ACC_PUBLIC
Code:
    stack=1, locals=1, args_size=1
    0: aload_0
    1: invokespecial #1           // Method java/lang/Object."<init>":()V
    4: iconst_0
    5: putstatic    #2           // Field a:I
    8: iconst_0
    9: putstatic    #3           // Field b:I
   12: return
```


Сеттер **a**:

```
public static void set_a(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
      0: iload_0
      1: putstatic    #2          // Field a:I
      4: return
```

Геттер **a**:

```
public static int get_a();
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=0, args_size=0
      0: getstatic    #2          // Field a:I
      3: ireturn
```

Сеттер **b**:

```
public static void set_b(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
      0: iload_0
      1: putstatic    #3          // Field b:I
      4: return
```

Геттер **b**:

```
public static int get_b();
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=0, args_size=0
      0: getstatic    #3          // Field b:I
      3: ireturn
```

Здесь нет разницы между кодом, работающим для public-полей и private-полей. Но эта информация присутствует в `.class`-файле, и в любом случае невозможно иметь доступ к private-полям.

Создадим объект и вызовем метод:

Листинг 55.16: ex1.java

```
public class ex1
{
    public static void main(String[] args)
    {
        test obj=new test();
        obj.set_a (1234);
        System.out.println(obj.a);
    }
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2, args_size=1
      0: new          #2          // class test
      3: dup
      4: invokespecial #3          // Method test."<init>":()V
      7: astore_1
      8: aload_1
      9: pop
     10: sipush      1234
     13: invokestatic #4          // Method test.set_a:(I)V
     16: getstatic   #5          // Field java/lang/System.out:Ljava/io/PrintStream;
     19: aload_1
     20: pop
```

55.17. ПРОСТЕЙШАЯ МОДИФИКАЦИЯ

```
21: getstatic      #6          // Field test.a:I
24: invokevirtual #7          // Method java/io/PrintStream.println:(I)V
27: return
```

Инструкция `new` создает объект, но не вызывает конструктор (он вызывается по смещению 4). Метод `set_a()` вызывается по смещению 16. К полю `a` имеется доступ используя инструкцию `getstatic` по смещению 21.

55.17. Простейшая модификация

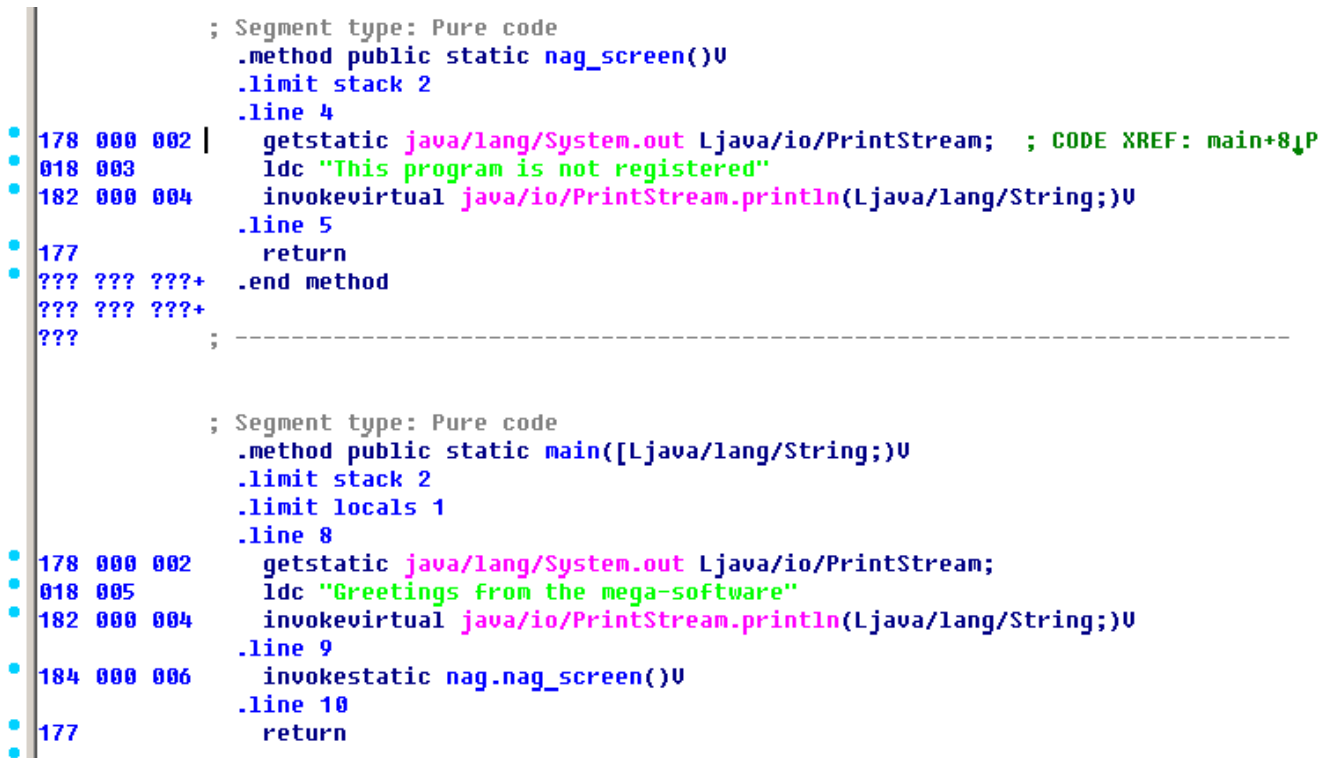
55.17.1. Первый пример

Перейдем к простой задаче модификации кода.

```
public class nag
{
    public static void nag_screen()
    {
        System.out.println("This program is not registered");
    };
    public static void main(String[] args)
    {
        System.out.println("Greetings from the mega-software");
        nag_screen();
    }
}
```

Как можно избавиться от печати строки «This program is not registered»?

Наконец загрузим .class-файл в IDA:



```
; Segment type: Pure code
.method public static nag_screen()V
.limit stack 2
.line 4
178 000 002 | getstatic java/lang/System.out Ljava/io/PrintStream; ; CODE XREF: main+8JP
018 003      ldc "This program is not registered"
182 000 004      invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
               .line 5
177          return
??? ??? ???+   .end method
??? ??? ???+
???           ; -----

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 1
.line 8
178 000 002      getstatic java/lang/System.out Ljava/io/PrintStream;
018 005          ldc "Greetings from the mega-software"
182 000 004      invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
               .line 9
184 000 006      invokestatic nag.nag_screen()V
               .line 10
177          return
```

Рис. 55.1: IDA

В начале заменим первый байт функции на 177 (это опкод инструкции `return`):

```

; Segment type: Pure code
.method public static nag_screen()V
.limit stack 2
.line 4

nag_screen:                                     ; CODE XREF: main+81P
return
000      0 ; 0x00
002      2 ; 0x02
018 003      ldc "This program is not registered"
182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 5
177      return
??? ??? ???+ .end method
??? ??? ???+
??? ; -----

```

Рис. 55.2: IDA

Но это не работает (JRE 1.7):

```

Exception in thread "main" java.lang.VerifyError: Expecting a stack map frame
Exception Details:
  Location:
    nag.nag_screen()V @1: nop
  Reason:
    Error exists in the bytecode
  Bytecode:
    0000000: b100 0212 03b6 0004 b1

    at java.lang.Class.getDeclaredMethods0(Native Method)
    at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
    at java.lang.Class.getMethod0(Class.java:2856)
    at java.lang.Class.getMethod(Class.java:1668)
    at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java:494)
    at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:486)

```

Вероятно, в JVM есть проверки связанные с картами стека.

ОК, попробуем пропатчить её иначе, удаляя вызов функции `nag()` :

```

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 1
.line 8
178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
018 005      ldc "Greetings from the mega-software"
182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 9
000      nop
000      nop
000      nop
.line 10
177      return
; =====

```

Рис. 55.3: IDA

0 это опкод инструкции `NOP`.

Теперь всё работает!

55.17.2. Второй пример

Еще один простой пример crackme:

```

public class password
{
    public static void main(String[] args)
    {
        System.out.println("Please enter the password");
        String input = System.console().readLine();
        if (input.equals("secret"))
            System.out.println("password is correct");
        else
            System.out.println("password is not correct");
    }
}

```

Загрузим в IDA:



```

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
.line 3
178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
018 003      ldc "Please enter the password"
182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 4
184 000 005    invokestatic java/lang/System.console()Ljava/io/Console;
182 000 006    invokevirtual java/io/Console.readLine()Ljava/lang/String;
076          astore_1 ; met002_slot001
.line 5
043          aload_1 ; met002_slot001
018 007      ldc "secret"
182 000 008    invokevirtual java/lang/String.equals(Ljava/lang/Object;)Z
153 000 014    ifeq met002_35
.line 6
178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
018 009      ldc "password is correct"
182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
167 000 011    goto met002_43
.line 8
met002_35:                                     ; CODE XREF: main+21fj
178 000 002    .stack use locals
          locals Object java/lang/String
          .end stack
018 010      getstatic java/lang/System.out Ljava/io/PrintStream;
018 010      ldc "password is not correct"
182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 9

```

Рис. 55.4: IDA

Видим здесь инструкцию `ifeq`, которая, собственно, всё и делает. Её имя означает *if equal*, и это не очень удачное название, её следовало бы назвать `ifz` (*if zero*), т.е. если значение на `TOS` ноль, тогда совершить переход. В нашем случае, переход происходит если пароль не верный (метод `equals` возвращает `False`, а это 0). Первое что приходит в голову это пропатчить эту инструкцию. В опкоде `ifeq` два байта, в которых закодировано смещение для перехода. Чтобы инструкция не работала, мы должны установить байт 3 на третьем байте (потому что 3 будет прибавляться к текущему смещению, и в итоге переход будет на следующую инструкцию, ведь длина инструкции `ifeq` это 3 байта):

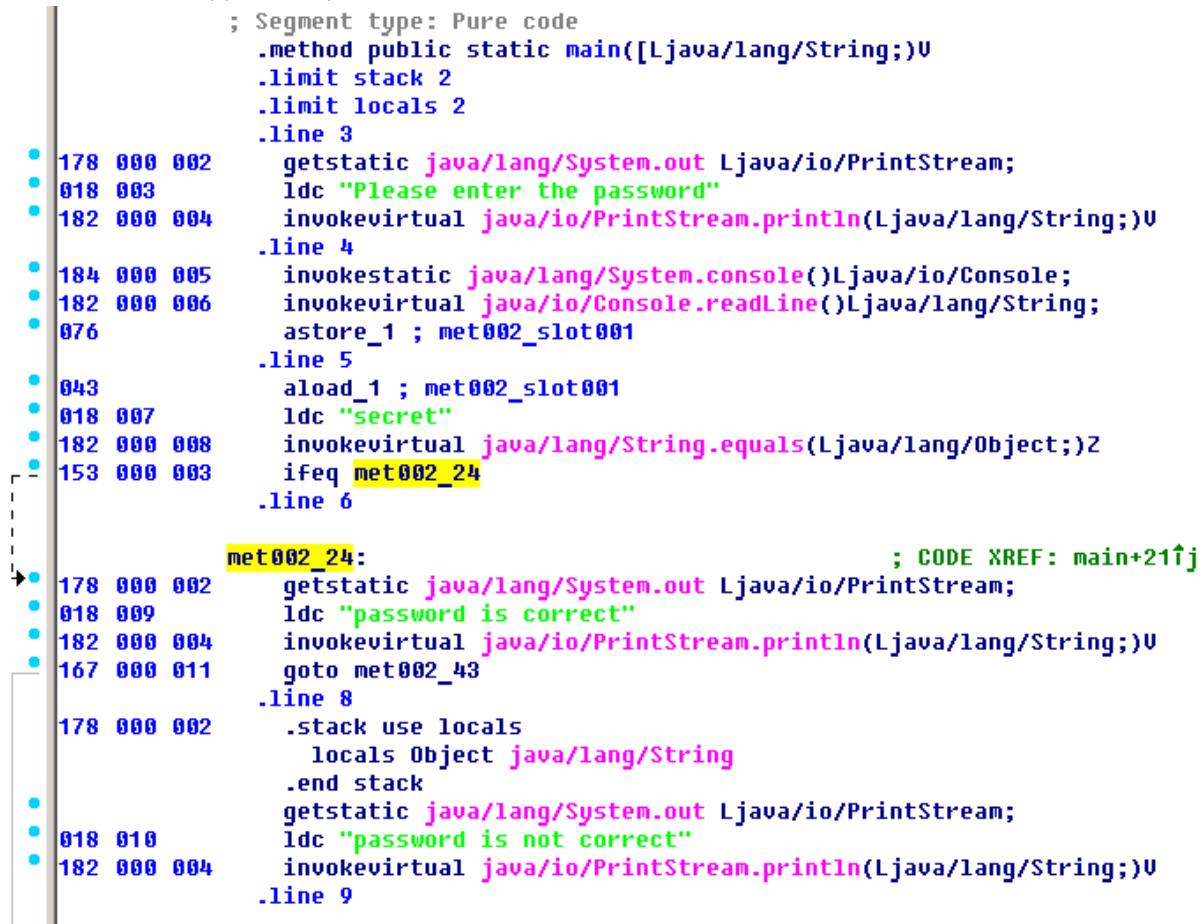


Рис. 55.5: IDA

Это не работает (JRE 1.7):

Exception in thread "main" java.lang.VerifyError: Expecting a stackmap frame at branch target 24
Exception Details:

Location:

password.main([Ljava/lang/String;)V @21: ifeq

Reason:

Expected stackmap frame at this location.

Bytecode:

```

00000000: b200 0212 03b6 0004 b800 05b6 0006 4c2b
00000010: 1207 b600 0899 0003 b200 0212 09b6 0004
00000020: a700 0bb2 0002 120a b600 04b1

```

Stackmap Table:

```

append_frame(@35,Object[#20])
same_frame(@43)

```

```

at java.lang.Class.getDeclaredMethods0(Native Method)
at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
at java.lang.Class.getMethod0(Class.java:2856)
at java.lang.Class.getMethod(Class.java:1668)
at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java:494)
at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:486)

```

Хотя, надо сказать, работает в JRE 1.6.

Мы также можем попробовать заменить все три байта опкода `ifeq` на нулевые байты (`NOP`), но это тоже не работает. Видимо, начиная с JRE 1.7, там появилось больше проверок карт стека.

ОК, заменим весь вызов метода `equals` на инструкцию `iconst_1` плюс набор `NOP`-ов:

```

; Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
.line 3
178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
018 003      ldc "Please enter the password"
182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 4
184 000 005    invokestatic java/lang/System.console()Ljava/io/Console;
182 000 006    invokevirtual java/io/Console.readLine()Ljava/lang/String;
076          astore_1 ; met002_slot001
.line 5
004          iconst_1
000          nop
000          nop
000          nop
000          nop
000          nop
000          nop
-- 153 000 014    ifeq met002_35
.line 6
178 000 002    getstatic java/lang/System.out Ljava/io/PrintStream;
018 009      ldc "password is correct"
182 000 004    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
167 000 011    goto met002_43
.line 8

met002_35:                                     ; CODE XREF: main+21↑j
178 000 002    .stack use locals
          locals Object java/lang/String
          .end stack

```

Рис. 55.6: IDA

1 будет всегда на **TOS** во время исполнения инструкции **ifeq**, так что **ifeq** никогда не совершит переход. Это работает.

55.18. Итоги

Чего не хватает в Java в сравнении с Си/Си++?

- Структуры: используйте классы.
- Объединения (union): используйте иерархии классов.
- Беззнаковые типы данных. Кстати, из-за этого реализовывать криптографические алгоритмы на Java немного труднее.
- Указатели на функции.

Часть V

Поиск в коде того что нужно

Современное ПО, в общем-то, минимализмом не отличается.

Но не потому, что программисты слишком много пишут, а потому что к исполняемым файлам обыкновенно прикомпилируют все подряд библиотеки. Если бы все вспомогательные библиотеки всегда выносили во внешние DLL, мир был бы иным. (Еще одна причина для Си++ — STL и прочие библиотеки шаблонов.)

Таким образом, очень полезно сразу понимать, какая функция из стандартной библиотеки или более-менее известной (как Boost¹⁰, libpng¹¹), а какая — имеет отношение к тому что мы пытаемся найти в коде.

Переписывать весь код на Си/Си++, чтобы разобраться в нем, безусловно, не имеет никакого смысла.

Одна из важных задач reverse engineer-а это быстрый поиск в коде того что собственно его интересует.

Дизассемблер IDA позволяет делать поиск как минимум строк, последовательностей байт, констант. Можно даже сделать экспорт кода в текстовый файл .lst или .asm и затем натравить на него `grep`, `awk`, и т.д.

Когда вы пытаетесь понять, что делает тот или иной код, это запросто может быть какая-то опенсорсная библиотека вроде libpng. Поэтому, когда находите константы, или текстовые строки, которые выглядят явно знакомыми, всегда полезно их *погуглить*. А если вы найдете искомый опенсорсный проект где это используется, то тогда будет достаточно будет просто сравнить вашу функцию с ней. Это решит часть проблем.

К примеру, если программа использует какие-то XML-файлы, первым шагом может быть установление, какая именно XML-библиотека для этого используется, ведь часто используется какая-то стандартная (или очень известная) вместо самодельной.

К примеру, автор этих строк однажды пытался разобраться как происходит компрессия/декомпрессия сетевых пакетов в SAP 6.0. Это очень большая программа, но к ней идет подробный .PDB-файл с отладочной информацией, и это очень удобно. Он в конце концов пришел к тому что одна из функций декомпрессирующая пакеты называется CsDecomprLZC(). Не сильно раздумывая, он решил погуглить и оказалось, что функция с таким же названием имеется в MaxDB (это опенсорсный проект SAP)¹².

<http://www.google.com/search?q=CsDecomprLZC>

Каково же было мое удивление, когда оказалось, что в MaxDB используется точно такой же алгоритм, скорее всего, с таким же исходником.

¹⁰<http://go.yurichev.com/17036>

¹¹<http://go.yurichev.com/17037>

¹²Больше об этом в соответствующей секции (80.1 (стр. 798))

Глава 56

Идентификация исполняемых файлов

56.1. Microsoft Visual C++

Версии MSVC и DLL которые могут быть импортированы:

Маркетинговая версия	Внутренняя версия	Версия CL.EXE	Импортируемые DLL	Дата выхода
6	6.0	12.00	msvcrt.dll, msvcp60.dll	June 1998
.NET (2002)	7.0	13.00	msvcr70.dll, msvcp70.dll	February 13, 2002
.NET 2003	7.1	13.10	msvcr71.dll, msvcp71.dll	April 24, 2003
2005	8.0	14.00	msvcr80.dll, msvcp80.dll	November 7, 2005
2008	9.0	15.00	msvcr90.dll, msvcp90.dll	November 19, 2007
2010	10.0	16.00	msvcr100.dll, msvcp100.dll	April 12, 2010
2012	11.0	17.00	msvcr110.dll, msvcp110.dll	September 12, 2012
2013	12.0	18.00	msvcr120.dll, msvcp120.dll	October 17, 2013

msvcr*.dll содержит функции связанные с Си++, так что если она импортируется, скорее всего, вы имеете дело с программой на Си++.

56.1.1. Name mangling

Имена обычно начинаются с символа `?`.

О [name mangling](#) в MSVC читайте также здесь: [52.1.1](#) (стр. 539).

56.2. GCC

Кроме компиляторов под *NIX, GCC имеется так же и для win32-окружения: в виде Cygwin и MinGW.

56.2.1. Name mangling

Имена обычно начинаются с символов `_Z`.

О [name mangling](#) в GCC читайте также здесь: [52.1.1](#) (стр. 539).

56.2.2. Cygwin

cygwin1.dll часто импортируется.

56.2.3. MinGW

msvcrt.dll может импортироваться.

56.3. Intel FORTRAN

libifcoremd.dll, libifportmd.dll и libiomp5md.dll (поддержка OpenMP) могут импортироваться.

В libifcoremd.dll много функций с префиксом `for_`, что значит FORTRAN.

56.4. Watcom, OpenWatcom

56.4.1. Name mangling

Имена обычно начинаются с символа `W`.

Например, так кодируется метод «method» класса «class» не имеющий аргументов и возвращающий `void` :

```
W?method$_class$n__v
```

56.5. Borland

Вот пример [name mangling](#) в Borland Delphi и C++Builder :

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiit1iiiiii
@TrueColorTo16BitN$qpviiiiit1iiiiii
@DIB24BitTo8BitBitmap$qpviiiiit1iiiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
@TrueBitmap@$bctr$qiilll
```

Имена всегда начинаются с символа `@` затем следует имя класса, имя метода и закодированные типы аргументов.

Эти имена могут присутствовать с импортах .exe, экспортах .dll, отладочной информации, и т.д.

Borland Visual Component Libraries (VCL) находятся в файлах .bpl вместо .dll, например , vcl50.dll, rtl60.dll.

Другие DLL которые могут импортироваться: BORLNDMM.DLL.

56.5.1. Delphi

Почти все исполняемые файлы имеют текстовую строку «Boolean» в самом начале сегмента кода, среди остальных имен типов .

Вот очень характерное для Delphi начало сегмента `CODE`, этот блок следует сразу за заголовком win32 PE-файла :

```
00000400 04 10 40 00 03 07 42 6f 6f 6c 65 61 6e 01 00 00 |..@...Boolean...|
00000410 00 00 01 00 00 00 00 10 40 00 05 46 61 6c 73 65 |.....@...False|
00000420 04 54 72 75 65 8d 40 00 2c 10 40 00 09 08 57 69 |.True.@...Wi|
00000430 64 65 43 68 61 72 03 00 00 00 00 ff ff 00 00 90 |deChar.....|
00000440 44 10 40 00 02 04 43 68 61 72 01 00 00 00 00 ff |D.@...Char.....|
00000450 00 00 00 90 58 10 40 00 01 08 53 6d 61 6c 6c 69 |...X.@...Smalli|
00000460 6e 74 02 00 80 ff ff ff 7f 00 00 90 70 10 40 00 |nt.....p.@|
00000470 01 07 49 6e 74 65 67 65 72 04 00 00 00 80 ff ff |..Integer.....|
00000480 ff 7f 8b c0 88 10 40 00 01 04 42 79 74 65 01 00 |.....@...Byte..|
00000490 00 00 00 ff 00 00 00 90 9c 10 40 00 01 04 57 6f |.....@...Wo|
000004a0 72 64 03 00 00 00 00 ff ff 00 00 90 b0 10 40 00 |rd.....@|
000004b0 01 08 43 61 72 64 69 6e 61 6c 05 00 00 00 00 ff |..Cardinal.....|
000004c0 ff ff ff 90 c8 10 40 00 10 05 49 6e 74 36 34 00 |.....@...Int64.|
000004d0 00 00 00 00 00 00 80 ff ff ff ff ff ff 7f 90 |.....|
000004e0 e4 10 40 00 04 08 45 78 74 65 6e 64 65 64 02 90 |..@...Extended..|
000004f0 f4 10 40 00 04 06 44 6f 75 62 6c 65 01 8d 40 00 |..@...Double..@|
00000500 04 11 40 00 04 08 43 75 72 72 65 6e 63 79 04 90 |..@...Currency..|
```

56.6. ДРУГИЕ ИЗВЕСТНЫЕ DLL

00000510	14 11 40 00 0a 06 73 74	72 69 6e 67 20 11 40 00	..@...string .@.
00000520	0b 0a 57 69 64 65 53 74	72 69 6e 67 30 11 40 00	..WideString0.@.
00000530	0c 07 56 61 72 69 61 6e	74 8d 40 00 40 11 40 00	..Variant.@.@.
00000540	0c 0a 4f 6c 65 56 61 72	69 61 6e 74 98 11 40 00	..OleVariant...@.
00000550	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
00000560	00 00 00 00 00 00 00 00	00 00 00 00 98 11 40 00	@.
00000570	04 00 00 00 00 00 00 00	18 4d 40 00 24 4d 40 00	M@.\$M@.
00000580	28 4d 40 00 2c 4d 40 00	20 4d 40 00 68 4a 40 00	(M@.,M@. M@.hJ@.
00000590	84 4a 40 00 c0 4a 40 00	07 54 4f 62 6a 65 63 74	.J@...J@..TObject
000005a0	a4 11 40 00 07 07 54 4f	62 6a 65 63 74 98 11 40	..@...TObject..@
000005b0	00 00 00 00 00 00 00 06	53 79 73 74 65 6d 00 00	System..
000005c0	c4 11 40 00 0f 0a 49 49	6e 74 65 72 66 61 63 65	..@...IInterface
000005d0	00 00 00 00 01 00 00 00	00 00 00 00 00 c0 00 00	
000005e0	00 00 00 00 46 06 53 79	73 74 65 6d 03 00 ff ff	F.System....
000005f0	f4 11 40 00 0f 09 49 44	69 73 70 61 74 63 68 c0	..@...IDispatch.
00000600	11 40 00 01 00 04 02 00	00 00 00 00 c0 00 00 00	.@.....
00000610	00 00 00 46 06 53 79 73	74 65 6d 04 00 ff ff 90	...F.System.....
00000620	cc 83 44 24 04 f8 e9 51	6c 00 00 83 44 24 04 f8	..D\$...Ql...D\$..
00000630	e9 6f 6c 00 00 83 44 24	04 f8 e9 79 6c 00 00 cc	.ol...D\$...yl...
00000640	cc 21 12 40 00 2b 12 40	00 35 12 40 00 01 00 00	.!.@.+.@.5.@....
00000650	00 00 00 00 00 00 00 00	00 c0 00 00 00 00 00 00	
00000660	46 41 12 40 00 08 00 00	00 00 00 00 00 8d 40 00	FA.@.....@.
00000670	bc 12 40 00 4d 12 40 00	00 00 00 00 00 00 00 00	..@.M.@.....
00000680	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
00000690	bc 12 40 00 0c 00 00 00	4c 11 40 00 18 4d 40 00	..@.....L.@..M@.
000006a0	50 7e 40 00 5c 7e 40 00	2c 4d 40 00 20 4d 40 00	P~@.\~@.,M@. M@.
000006b0	6c 7e 40 00 84 4a 40 00	c0 4a 40 00 11 54 49 6e	l~@..J@..J@..TIn
000006c0	74 65 72 66 61 63 65 64	4f 62 6a 65 63 74 8b c0	terfacedObject..
000006d0	d4 12 40 00 07 11 54 49	6e 74 65 72 66 61 63 65	..@...TInterface
000006e0	64 4f 62 6a 65 63 74 bc	12 40 00 a0 11 40 00 00	dObject..@...@..
000006f0	00 06 53 79 73 74 65 6d	00 00 8b c0 00 13 40 00	..System.....@.
00000700	11 0b 54 42 6f 75 6e 64	41 72 72 61 79 04 00 00	..TBoundArray...
00000710	00 00 00 00 00 03 00 00	00 6c 10 40 00 06 53 79	l.@..Sy
00000720	73 74 65 6d 28 13 40 00	04 09 54 44 61 74 65 54	stem(.@...TDateT
00000730	69 6d 65 01 ff 25 48 e0	c4 00 8b c0 ff 25 44 e0	ime..%H.....%D.

Первые 4 байта сегмента данных (DATA) в исполняемых файлах могут быть 00 00 00 00 , 32 13 8B C0 или FF FF FF FF . Эта информация может помочь при работе с запакованными/зашифрованными программами на Delphi.

56.6. Другие известные DLL

- vcomp*.dll – Реализация OpenMP от Microsoft.

Глава 57

Связь с внешним миром (win32)

Иногда, чтобы понять что делает та или иная функция, можно её не разбирать, а просто посмотреть на её входы и выходы. Так можно сэкономить время.

Обращения к файлам и реестру: для самого простого анализа может помочь утилита Process Monitor¹ от SysInternals.

Для анализа обращения программы к сети, может помочь Wireshark².

Затем всё-таки придётся смотреть внутрь.

Первое на что нужно обратить внимание, это какие функции из [API](#)³ ОС и какие функции стандартных библиотек используются.

Если программа поделена на главный исполняемый файл и группу DLL-файлов, то имена функций в этих DLL, бывает так, могут помочь.

Если нас интересует, что именно приводит к вызову `MessageBox()` с определенным текстом, то первое что можно попробовать сделать: найти в сегменте данных этот текст, найти ссылки на него, и найти, откуда может передаваться управление к интересующему нас вызову `MessageBox()`.

Если речь идет о компьютерной игре, и нам интересно какие события в ней более-менее случайны, мы можем найти функцию `rand()` или её заменитель (как алгоритм Mersenne twister), и посмотреть, из каких мест эта функция вызывается и что самое главное: как используется результат этой функции. Один пример: [76](#).

Но если это не игра, а `rand()` используется, то также весьма любопытно, зачем. Бывают неожиданные случаи вроде использования `rand()` в алгоритме для сжатия данных (для имитации шифрования): [blog.yurichev.com](#).

57.1. Часто используемые функции Windows API

Это функции которые можно увидеть в числе импортируемых. Но также нельзя забывать, что далеко не все они были использованы в коде написанном автором. Немалая часть может вызываться из библиотечных функций и CRT-кода.

- Работа с реестром (advapi32.dll): `RegEnumKeyEx`^{4 5}, `RegEnumValue`^{6 5}, `RegGetValue`^{7 5}, `RegOpenKeyEx`^{8 5}, `RegQueryValueEx`^{9 5}.
- Работа с текстовыми .ini-файлами (kernel32.dll): `GetPrivateProfileString`^{10 5}.
- Диалоговые окна (user32.dll): `MessageBox`^{11 5}, `MessageBoxEx`^{12 5}, `SetDlgItemText`^{13 5}, `GetDlgItemText`^{14 5}.

¹<http://go.yurichev.com/17301>

²<http://go.yurichev.com/17303>

³Application programming interface

⁴MSDN

⁵Может иметь суффикс -A для ASCII-версии и -W для Unicode-версии

⁶MSDN

⁷MSDN

⁸MSDN

⁹MSDN

¹⁰MSDN

¹¹MSDN

¹²MSDN

¹³MSDN

¹⁴MSDN

- Работа с ресурсами (69.2.8 (стр. 702)): (user32.dll): LoadMenu ^{15 5}.
- Работа с TCP/IP-сетью (ws2_32.dll): WSARcv ¹⁶, WSARecv ¹⁷.
- Работа с файлами (kernel32.dll): CreateFile ^{18 5}, ReadFile ¹⁹, ReadFileEx ²⁰, WriteFile ²¹, WriteFileEx ²².
- Высокоуровневая работа с Internet (wininet.dll): WinHttpOpen ²³.
- Проверка цифровой подписи исполняемого файла (wintrust.dll): WinVerifyTrust ²⁴.
- Стандартная библиотека MSVC (в случае динамического связывания) (msvcr*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

57.2. tracer: Перехват всех функций в отдельном модуле

В [tracer](#) есть поддержка точек останова INT3, хотя и срабатывающие только один раз, но зато их можно установить на все сразу функции в некоей DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Либо, поставим INT3-прерывание на все функции, имена которых начинаются с префикса `xml`:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

В качестве обратной стороны медали, такие прерывания срабатывают только один раз.

Tracer покажет вызов какой-либо функции, если он случится, но только один раз. Еще один недостаток — увидеть аргументы функции также нельзя.

Тем не менее, эта возможность очень удобна для тех ситуаций, когда вы знаете что некая программа использует некую DLL, но не знаете какие именно функции в этой DLL. И функций много.

Например, попробуем узнать, что использует `cygwin`-утилита `uptime`:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Так мы можем увидеть все функции из библиотеки `cygwin1.dll`, которые были вызваны хотя бы один раз, и откуда:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!0EP+0x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from uptime.exe!0EP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!0EP+0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!0EP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!0EP+0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!0EP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!0EP+0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!0EP+0x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!0EP+0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!0EP+0x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!0EP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!0EP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!0EP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!0EP+0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptime.exe!0EP+0x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!0EP+0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!0EP+0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!0EP+0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!0EP+0x4c3 (0x4014c3))
```

¹⁵MSDN

¹⁶MSDN

¹⁷MSDN

¹⁸MSDN

¹⁹MSDN

²⁰MSDN

²¹MSDN

²²MSDN

²³MSDN

²⁴MSDN

Глава 58

Строки

58.1. Текстовые строки

58.1.1. Си/Си++

Обычные строки в Си заканчиваются нулем (ASCIIZ-строки).

Причина, почему формат строки в Си именно такой (оканчивающийся нулем) вероятно историческая. В [Dennis M. Ritchie, *The Evolution of the Unix Time-sharing System*, (1979)] мы можем прочитать:

A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

Строки выглядят в Hiew или FAR Manager точно так же :

```
int main()
{
    printf ("Hello, world!\n");
};
```

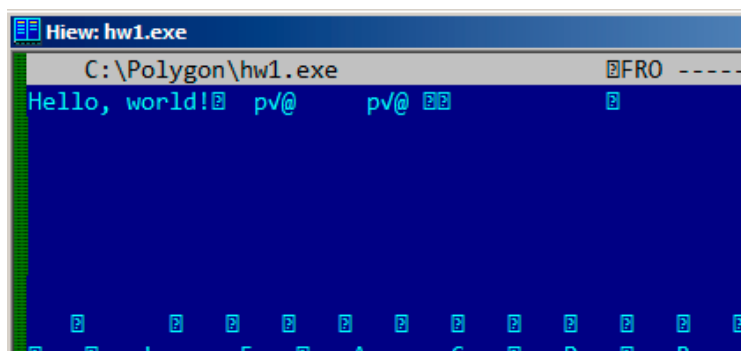


Рис. 58.1: Hiew

58.1.2. Borland Delphi

Когда кодируются строки в Pascal и Delphi, сама строка предваряется 8-битным или 32-битным значением, в котором закодирована длина строки.

Например:

Листинг 58.1: Delphi

```
CODE:00518AC8          dd 19h
CODE:00518ACC aLoading__Plea db 'Loading... , please wait.',0
...
```

```
CODE:00518AFC          dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run...',0
```

58.1.3. Unicode

Нередко уникодом называют все способы передачи символов, когда символ занимает 2 байта или 16 бит. Это распространенная терминологическая ошибка. Уникод — это стандарт, присваивающий номер каждому символу многих письменностей мира, но не описывающий способ кодирования.

Наиболее популярные способы кодирования: UTF-8 (наиболее часто используется в Интернете и *NIX-системах) и UTF-16LE (используется в Windows).

UTF-8

UTF-8 это один из очень удачных способов кодирования символов. Все символы латиницы кодируются так же, как и в ASCII-кодировке, а символы, выходящие за пределы ASCII-7-таблицы, кодируются несколькими байтами. 0 кодируется, как и прежде, нулевым байтом, так что все стандартные функции Си продолжают работать с UTF-8-строками так же как и с обычными строками.

Посмотрим, как символы из разных языков кодируются в UTF-8 и как это выглядит в FAR, в кодировке 437 ¹:

```
How much? 100€?

(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic): أنا قادر على أكل الزجاج و هذا لا يؤلمني.
(Hebrew): אני יכול לאכול זכוכית וזה לא מזיק לי.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```

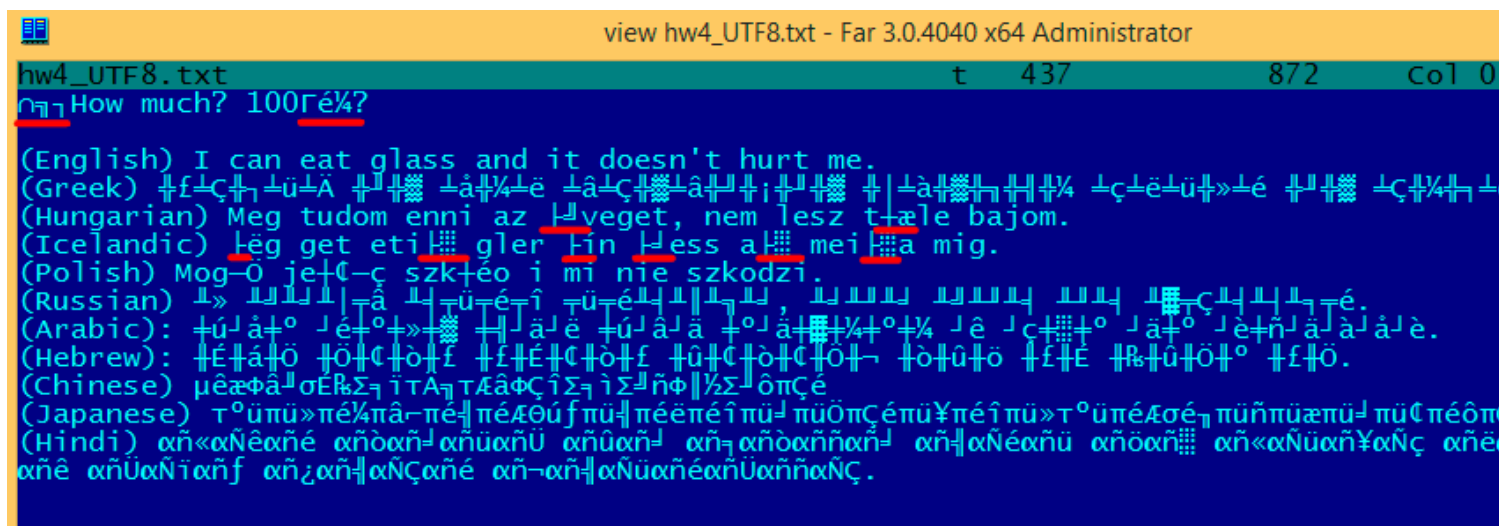


Рис. 58.2: FAR: UTF-8

Видно, что строка на английском языке выглядит точно так же, как и в ASCII-кодировке. В венгерском языке используются латиница плюс латинские буквы с диакритическими знаками. Здесь видно, что эти буквы кодируются несколькими байтами, они подчеркнуты красным. То же самое с исландским и польским языками. В самом начале имеется также символ валюты «Евро», который кодируется тремя байтами. Остальные системы письма здесь никак не связаны с латиницей

¹Пример и переводы на разные языки были взяты здесь: <http://go.yurichev.com/17304>

58.1. ТЕКСТОВЫЕ СТРОКИ

. По крайней мере о русском, арабском, иврите и хинди мы можем сказать, что здесь видны повторяющиеся байты, что не удивительно, ведь, обычно буквы из одной и той же системы письменности расположены в одной или нескольких таблицах уникада, поэтому часто их коды начинаются с одних и тех же цифр .

В самом начале, перед строкой «How much?», видны три байта, которые на самом деле BOM² . BOM показывает, какой способ кодирования будет сейчас использоваться.

UTF-16LE

Многие функции win32 в Windows имеют суффикс `-A` и `-W` . Первые функции работают с обычными строками, вторые с UTF-16LE-строками (*wide*). Во втором случае, каждый символ обычно хранится в 16-битной переменной типа *short* .

Строки с латинскими буквами выглядят в Hiew или FAR как перемежающиеся с нулевыми байтами :

```
int wmain()
{
    wprintf (L"Hello, world!\n");
};
```

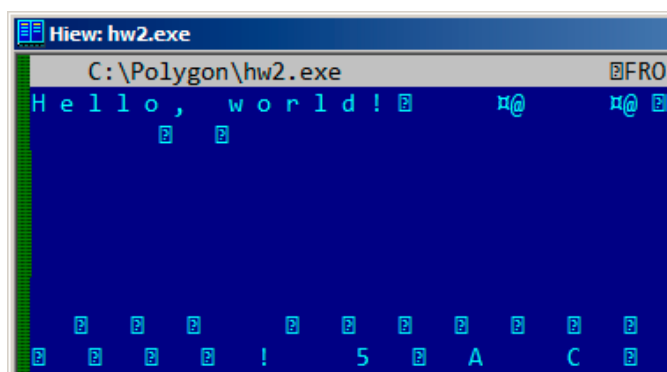


Рис. 58.3: Hiew

Подобное можно часто увидеть в системных файлах Windows NT:

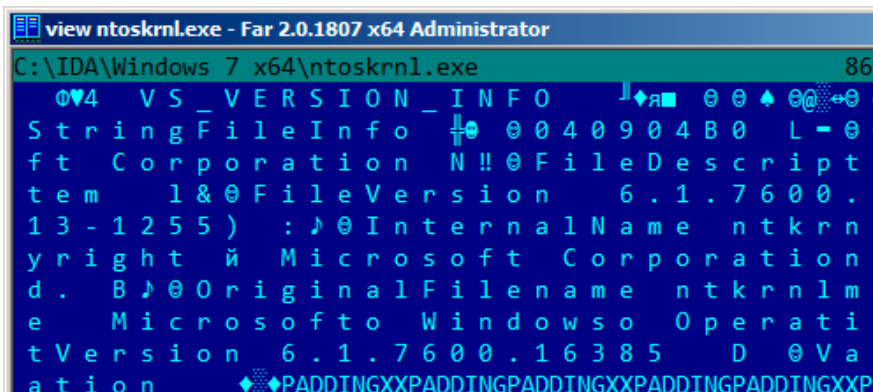


Рис. 58.4: Hiew

В IDA, уникадом называется именно строки с символами, занимающими 2 байта:

```
.data:0040E000 aHelloWorld:
.data:0040E000          unicode 0, <Hello, world!>
.data:0040E000          dw 0Ah, 0
```

Вот как может выглядеть строка на русском языке («И снова здравствуйте!») закодированная в UTF-16LE:

²Byte order mark

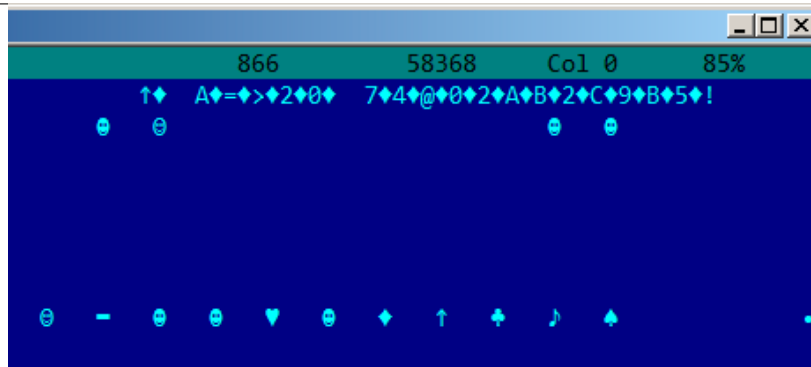


Рис. 58.5: Hiew: UTF-16LE

То что бросается в глаза — это то что символы перемежаются ромбиками (который имеет код 4) . Действительно, в уникоде кириллические символы находятся в четвертом блоке ³ . Таким образом, все кириллические символы в UTF-16LE находятся в диапазоне 0x400-0x4FF .

Вернемся к примеру, где одна и та же строка написана разными языками. Здесь посмотрим в кодировке UTF-16LE.

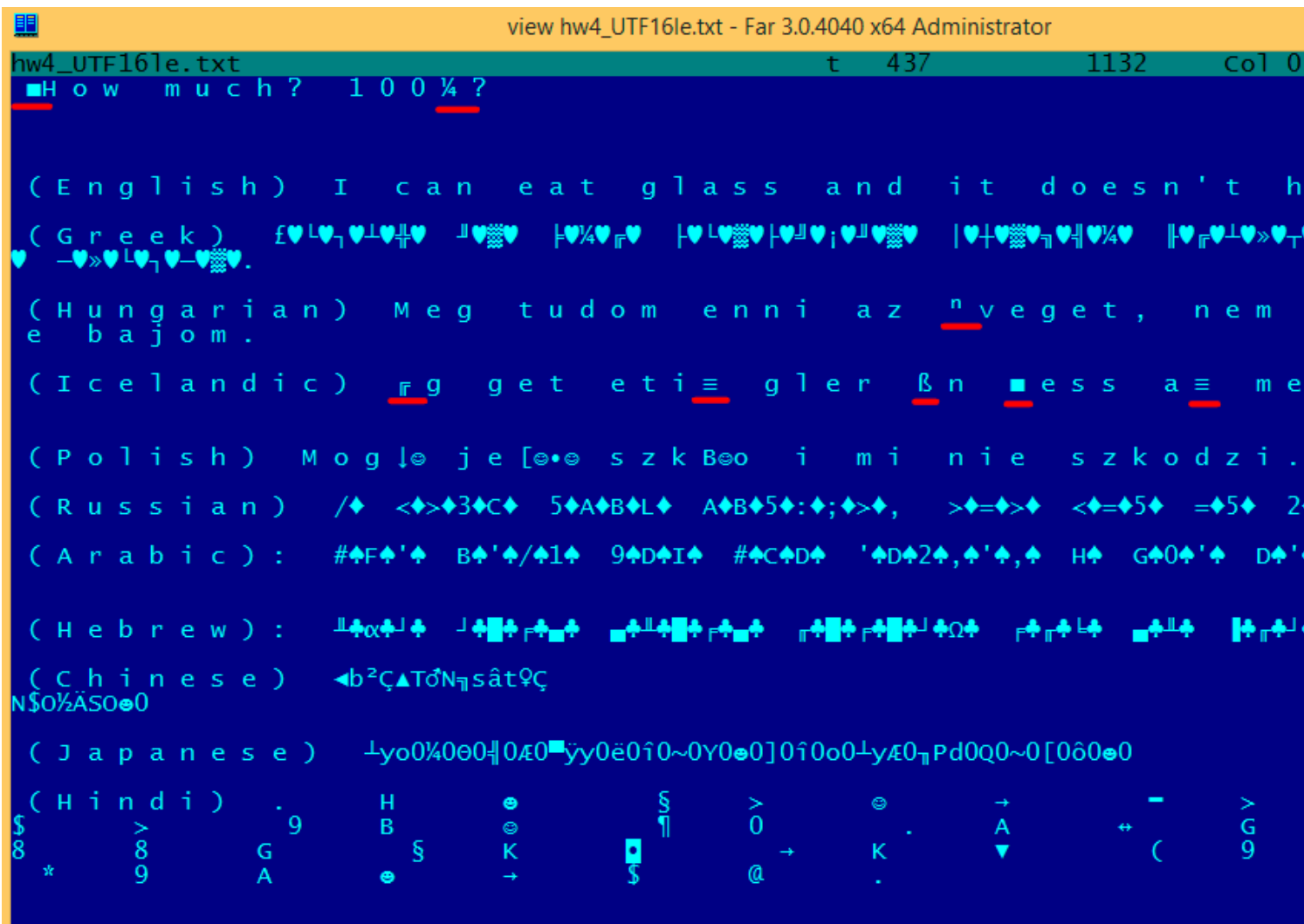


Рис. 58.6: FAR: UTF-16LE

Здесь мы также видим **BOM** в самом начале. Все латинские буквы перемежаются с нулевыми байтом. Некоторые буквы с диакритическими знаками (венгерский и исландский языки) также подчеркнуты красным.

³ wikipedia

58.1.4. Base64

Кодировка base64 очень популярна в тех случаях, когда нужно передать двоичные данные как текстовую строку. По сути, этот алгоритм кодирует 3 двоичных байта в 4 печатаемых символа: все 26 букв латинского алфавита (в обоих регистрах), цифры, знак плюса («+») и слэша («/»), в итоге это получается 64 символа.

Одна отличительная особенность строк в формате base64, это то что они часто (хотя и не всегда) заканчиваются одним или двумя символами знака равенства («=») для выравнивания, например:

```
AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp91A0uWs=
```

```
WVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp91A0uQ==
```

Так что знак равенства («=») никогда не встречается в середине строк закодированных в base64.

58.2. Сообщения об ошибках и отладочные сообщения

Очень сильно помогают отладочные сообщения, если они имеются. В некотором смысле, отладочные сообщения, это отчет о том, что сейчас происходит в программе. Зачастую, это `printf()`-подобные функции, которые пишут куда-нибудь в лог, а бывает так что и не пишут ничего, но вызовы остались, так как эта сборка — не отладочная, а *release*. Если в отладочных сообщениях дампятся значения некоторых локальных или глобальных переменных, это тоже может помочь, как минимум, узнать их имена. Например, в Oracle RDBMS одна из таких функций: `ksdwrt()`.

Осмысленные текстовые строки вообще очень сильно могут помочь. Дизассемблер [IDA](#) может сразу указать, из какой функции и из какого её места используется эта строка. Встречаются и смешные случаи ⁴.

Сообщения об ошибках также могут помочь найти то что нужно. В Oracle RDBMS сигнализация об ошибках проходит при помощи вызова некоторой группы функций.

Тут еще немного об этом : blog.yurichev.com.

Можно довольно быстро найти, какие функции сообщают о каких ошибках, и при каких условиях. Это, кстати, одна из причин, почему в защите софта от копирования, бывает так, что сообщение об ошибке заменяется невнятным кодом или номером ошибки. Мало кому приятно, если взломщик быстро поймет, из-за чего именно срабатывает защита от копирования, просто по сообщению об ошибке.

Один из примеров шифрования сообщений об ошибке, здесь: [78.2](#) (стр. [755](#)).

58.3. Подозрительные магические строки

Некоторые магические строки, используемые в бэкдорах выглядят очень подозрительно. Например, в домашних роутерах TP-Link WR740 был бэкдор ⁵. Бэкдор активировался при посещении следующего URL:

http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html.

Действительно, строка «userRpmNatDebugRpm26525557» присутствует в прошивке. Эту строку нельзя было наугадить до распространения информации о бэкдоре. Вы не найдете ничего такого ни в одном [RFC](#)⁶. Вы не найдете ни одного алгоритма, который бы использовал такие странные последовательности байт. И это не выглядит как сообщение об ошибке, или отладочное сообщение. Так что проверить использование подобных странных строк — это всегда хорошая идея.

Иногда такие строки кодируются при помощи `base64`⁷. Так что неплохая идея их всех декодировать и затем просмотреть глазами, пусть даже бегло.

Более точно, такой метод сокрытия бэкдоров называется «security through obscurity» (безопасность через запутанность).

⁴blog.yurichev.com

⁵<http://sekurak.pl/tp-link-httpftf-backdoor/>, на русском: <http://m.habrahabr.ru/post/172799/>

⁶Request for Comments

⁷Например, бэкдор в кабельном модеме Arris: <http://www.securitylab.ru/analytics/461497.php>

Глава 59

Вызовы assert()

Может также помочь наличие `assert()` в коде: обычно этот макрос оставляет название файла-исходника, номер строки, и условие.

Наиболее полезная информация содержится в `assert`-условии, по нему можно судить по именам переменных или именам полей структур. Другая полезная информация — это имена файлов, по их именам можно попытаться предположить, что там за код. Также, по именам файлов можно опознать какую-либо очень известную open-сорсную библиотеку.

Листинг 59.1: Пример информативных вызовов `assert()`

```
.text:107D4B29 mov dx, [ecx+42h]
.text:107D4B2D cmp edx, 1
.text:107D4B30 jz short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ; "td->td_planarconfig == PLANARCONFIG_CON"...
.text:107D4B41 call ds:_assert

...

.text:107D52CA mov edx, [ebp-4]
.text:107D52CD and edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30 ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert

...

.text:107D6759 mov cx, [eax+6]
.text:107D675D cmp ecx, 0Ch
.text:107D6760 jle short loc_107D677A
.text:107D6762 push 2D8h
.text:107D6767 push offset aLzw_c ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D6771 call ds:_assert
```

Полезно «гуглить» и условия и имена файлов, это может вывести вас к open-сорсной библиотечке. Например, если «погуглить» «`sp->lzw_nbits <= BITS_MAX`», это вполне предсказуемо выводит на openсорсный код, что-то связанное с LZW-компрессией.

Глава 60

Константы

Люди, включая программистов, часто используют круглые числа вроде 10, 100, 1000, в т.ч. и в коде.

Практикующие реверсеры, обычно, хорошо знают их в шестнадцатеричном представлении : 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

Иногда попадают константы 0xAAAAAAAA (10101010101010101010101010101010) и

0x55555555 (01010101010101010101010101010101) — это чередующиеся биты. Это помогает отличить некоторый сигнал от сигнала где все биты включены (1111 ...) или выключены (0000 ...). Например, константа 0x55AA используется как минимум в бут-секторе, MBR¹, и в ПЗУ² плат-расширений IBM-компьютеров.

Некоторые алгоритмы, особенно криптографические, используют хорошо различимые константы, которые при помощи IDA легко находить в коде.

Например, алгоритм MD5³ инициализирует свои внутренние переменные так:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

Если в коде найти использование этих четырех констант подряд — очень высокая вероятность что эта функция имеет отношение к MD5.

Еще такой пример это алгоритмы CRC16/CRC32, часто, алгоритмы вычисления контрольной суммы по CRC используют заранее заполненные таблицы, вроде:

Листинг 60.1: linux/lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
u16 const crc16_table[256] = {
    0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
    0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
    0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
    ...
}
```

См. также таблицу CRC32: 38 (стр. 466).

60.1. Магические числа

Немало форматов файлов определяет стандартный заголовок файла где используются *магическое число* (magic number)⁴, один или даже несколько.

Скажем, все исполняемые файлы для Win32 и MS-DOS начинаются с двух символов «MZ»⁵.

В начале MIDI-файла должно быть «MThd». Если у нас есть использующая для чего-нибудь MIDI-файлы программа очень вероятно, что она будет проверять MIDI-файлы на правильность хотя бы проверяя первые 4 байта.

¹Master Boot Record

²Постоянное запоминающее устройство

³wikipedia

⁴wikipedia

⁵wikipedia

60.1. МАГИЧЕСКИЕ ЧИСЛА

Это можно сделать при помощи:

(*buf* указывает на начало загруженного в память файла)

```
cmp [buf], 0x6468544D ; "MThd"  
jnz _error_not_a_MIDI_file
```

...либо вызвав функцию сравнения блоков памяти `memcmp()` или любой аналогичный код, вплоть до инструкции `CMPSB` (A.6.3 (стр. 920)).

Найдя такое место мы получаем как минимум информацию о том, где начинается загрузка MIDI-файла, во-вторых, мы можем увидеть где располагается буфер с содержимым файла, и что еще оттуда берется, и как используется.

60.1.1. Даты

Часто, можно встретить число вроде `0x19861115`, которое явно выглядит как дата (1986-й год, 11-й месяц (ноябрь), 15-й день). Это может быть чей-то день рождения (программиста, его/её родственника, ребенка), либо какая-то другая важная дата. Дата может быть записана и в другом порядке, например `0x15111986`.

Известный пример это `0x19540119` (магическое число используемое в структуре суперблока UFS2), это день рождения Маршала Кирка МакКузика, видного разработчика FreeBSD.

Также, числа вроде таких очень популярны в любительской криптографии, например, это отрывок из внутренностей секретной функции донглы HASP3⁶:

```
void xor_pwd(void)  
{  
    int i;  
  
    pwd^=0x09071966;  
    for(i=0;i<8;i++)  
    {  
        al_buf[i]= pwd & 7; pwd = pwd >> 3;  
    }  
};  
  
void emulate_func2(unsigned short seed)  
{  
    int i, j;  
    for(i=0;i<8;i++)  
    {  
        ch[i] = 0;  
  
        for(j=0;j<8;j++)  
        {  
            seed *= 0x1989;  
            seed += 5;  
            ch[i] |= (tab[(seed>>9)&0x3f]) << (7-j);  
        }  
    }  
}
```

60.1.2. DHCP

Это касается также и сетевых протоколов. Например, сетевые пакеты протокола DHCP содержат так называемую *magic cookie*: `0x63538263`. Какой-либо код, генерирующий пакеты по протоколу DHCP где-то и как-то должен внедрять в пакет также и эту константу. Найдя её в коде мы сможем найти место где происходит это и не только это. Любая программа, получающая DHCP-пакеты, должна где-то как-то проверять *magic cookie*, сравнивая это поле пакета с константой.

Например, берем файл `dhcpcore.dll` из Windows 7 x64 и ищем эту константу. И находим, два раза: оказывается, эта константа используется в функциях с красноречивыми названиями `DhcpExtractOptionsForValidation()` и `DhcpExtractFullOptionsForValidation()`.

Листинг 60.2: `dhcpcore.dll` (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h ; DATA XREF: ↗  
↳ DhcpExtractOptionsForValidation+79
```

⁶<https://web.archive.org/web/20160311231616/http://www.woodmann.com/fravia/bayu3.htm>

60.2. ПОИСК КОНСТАНТ

```
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h ; DATA XREF: DhcpExtractFullOptions+97
```

А вот те места в функциях где происходит обращение к константам:

Листинг 60.3: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF6480875F mov     eax, [rsi]
.text:000007FF64808761 cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767 jnz     loc_7FF64817179
```

И:

Листинг 60.4: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF648082C7 mov     eax, [r12]
.text:000007FF648082CB cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1 jnz     loc_7FF648173AF
```

60.2. Поиск констант

В [IDA](#) это очень просто, Alt-B или Alt-I. А для поиска константы в большом количестве файлов, либо для поиска их в неисполняемых файлах, имеется небольшая утилита *binary grep*⁷.

⁷ [GitHub](#)

Глава 61

Поиск нужных инструкций

Если программа использует инструкции сопроцессора, и их не очень много, то можно попробовать вручную проверить отладчиком какую-то из них.

К примеру, нас может заинтересовать, при помощи чего Microsoft Excel считает результаты формул, введенных пользователем. Например, операция деления.

Если загрузить excel.exe (из Office 2010) версии 14.0.4756.1000 в [IDA](#), затем сделать полный листинг и найти все инструкции **FDIV** (но кроме тех, которые в качестве второго операнда используют константы — они, очевидно, не подходят нам):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...то окажется, что их всего 144.

Мы можем вводить в Excel строку вроде `=(1/3)` и проверить все эти инструкции.

Проверяя каждую инструкцию в отладчике или [tracer](#) (проверять эти инструкции можно по 4 за раз), окажется, что нам везет и срабатывает всего лишь 14-я по счету:

```
.text:3011E919 DC 33 fdiv qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=
FPU ST(0): 1.000000
```

В **ST(0)** содержится первый аргумент (1), второй содержится в **[EBX]**.

Следующая за **FDIV** инструкция (**FSTP**) записывает результат в память:

```
.text:3011E91B DD 1E fstp qword ptr [esi]
```

Если поставить breakpoint на ней, то мы можем видеть результат:

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

А также, в рамках пранка¹, модифицировать его на лету:

¹practical joke

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel показывает в этой ячейке 666, что окончательно убеждает нас в том, что мы нашли нужное место.

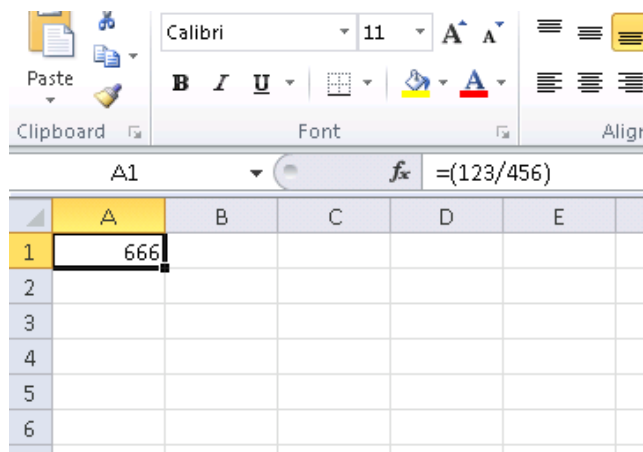


Рис. 61.1: Пранк сработал

Если попробовать ту же версию Excel, только x64, то окажется что там инструкций **FDIV** всего 12, причем нужная нам — третья по счету.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

Видимо, все дело в том, что много операций деления переменных типов *float* и *double* компилятор заменил на SSE-инструкции вроде **DIVSD**, коих здесь теперь действительно много (**DIVSD** присутствует в количестве 268 инструкций).

Глава 62

Подозрительные паттерны кода

62.1. Инструкции XOR

Инструкции вроде `XOR op, op` (например, `XOR EAX, EAX`) обычно используются для обнуления регистра, однако, если операнды разные, то применяется операция именно «исключающего или». Эта операция очень редко применяется в обычном программировании, но применяется очень часто в криптографии, включая любительскую. Особенно подозрительно, если второй операнд — это большое число. Это может указывать на шифрование, вычисление контрольной суммы, и т.д.

Одно из исключений из этого наблюдения о котором стоит сказать, то, что генерация и проверка значения «канарейки» (19.3 (стр. 279)) часто происходит, используя инструкцию `XOR`.

Этот AWK-скрипт можно использовать для обработки листингов (.lst) созданных IDA :

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if ($4!="esp") if ($4!="ebp") { ↵
  ↵ print $1, $2, tmp, ",", $4 } }' filename.lst
```

Нельзя также забывать, что если использовать подобный скрипт, то, возможно, он захватит и неверно дизассемблированный код (50 (стр. 528)).

62.2. Вручную написанный код на ассемблере

Современные компиляторы не генерируют инструкции `LOOP` и `RCL`. С другой стороны, эти инструкции хорошо знакомы кодерам, предпочитающим писать прямо на ассемблере. Подобные инструкции отмечены как (M) в списке инструкций в приложении: A.6 (стр. 915). Если такие инструкции встретились, можно сказать с какой-то вероятностью, что этот фрагмент кода написан вручную.

Также, пролог/эпилог функции обычно не встречается в ассемблерном коде, написанном вручную.

Как правило, в вручную написанном коде, нет никакого четкого метода передачи аргументов в функцию .

Пример из ядра Windows 2003 (файл ntoskrnl.exe):

```
MultiplyTest proc near                ; CODE XREF: Get386Stepping
xor          cx, cx
loc_620555:                            ; CODE XREF: MultiplyTest+E
push        cx
call        Multiply
pop         cx
jb          short locret_620563
loop        loc_620555
clc
locret_620563:                        ; CODE XREF: MultiplyTest+C
retn
MultiplyTest endp

Multiply     proc near                ; CODE XREF: MultiplyTest+5
mov         ecx, 81h
mov         eax, 417A000h
```

62.2. ВРУЧНУЮ НАПИСАННЫЙ КОД НА АСSEMBЛЕРЕ

```
mul    ecx
cmp    edx, 2
stc
jnz    short locret_62057F
cmp    eax, 0FE7A000h
stc
jnz    short locret_62057F
clc
locret_62057F:                ; CODE XREF: Multiply+10
                             ; Multiply+18
    retn
Multiply    endp
```

Действительно, если заглянуть в исходные коды [WRK¹](#) v1.2, данный код можно найти в файле *WRK-v1.2\base\ntos\ke\i386\cpu.asm*.

¹Windows Research Kernel

Глава 63

Использование magic numbers для трассировки

Нередко бывает нужно узнать, как используется то или иное значение, прочитанное из файла либо взятое из пакета, принятого по сети. Часто, ручное слежение за нужной переменной это трудный процесс. Один из простых методов (хотя и не полностью надежный на 100%) это использование вашей собственной *magic number*.

Это чем-то напоминает компьютерную томографию: пациенту перед сканированием вводят в кровь рентгеноконтрастный препарат, хорошо отсвечивающий в рентгеновских лучах. Известно, как кровь нормального человека расходится, например, по почкам, и если в этой крови будет препарат, то при томографии будет хорошо видно, достаточно ли хорошо кровь расходится по почкам и нет ли там камней, например, и прочих образований.

Мы можем взять 32-битное число вроде `0x0badf00d`, либо чью-то дату рождения вроде `0x11101979` и записать это, занимающее 4 байта число, в какое-либо место файла используемого исследуемой нами программой.

Затем, при трассировки этой программы, в том числе, при помощи [tracer](#) в режиме *code coverage*, а затем при помощи *grep* или простого поиска по текстовому файлу с результатами трассировки, мы можем легко увидеть, в каких местах кода использовалось это значение, и как.

Пример результата работы [tracer](#) в режиме *cc*, к которому легко применить утилиту *grep*:

```
0x150bf66 (_kziaia+0x14), e=      1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=      1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=      1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=      1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0xf1ac360
0x150bf78 (_kziaia+0x26), e=      1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

Это справедливо также и для сетевых пакетов. Важно только, чтобы наш *magic number* был как можно более уникален и не присутствовал в самом коде.

Помимо [tracer](#), такой эмулятор MS-DOS как DosBox, в режиме *heavydebug*, может писать в отчет информацию обо всех состояниях регистра на каждом шаге исполнения программы¹, так что этот метод может пригодиться и для исследования программ под DOS.

¹См. также мой пост в блоге об этой возможности в DosBox: blog.yurichev.com

Глава 64

Прочее

64.1. Общая идея

Нужно стараться как можно чаще ставить себя на место программиста и задавать себе вопрос, как бы вы сделали ту или иную вещь в этом случае и в этой программе.

64.2. Си++

[RTTI \(52.1.5 \(стр. 554\)\)](#)-информация также может быть полезна для идентификации классов в Си++.

64.3. Некоторые паттерны в бинарных файлах

Все примеры здесь были подготовлены в Windows с активной кодовой страницей 437 ¹ в консоли. Двоичные файлы внутри могут визуально выглядеть иначе если установлена другая кодовая страница.

¹<https://ru.wikipedia.org/wiki/CP437>

64.3.1. Массивы

Иногда мы можем легко заметить массив 16/32/64-битных значений визуально, в шестнадцатеричном редакторе.

Вот пример массива 16-битных значений. Мы видим что каждый первый байт в паре всегда равен 7 или 8, а второй выглядит случайным:

E:\...\3affacde09fe21c28f1543db51145b.dat		h	1252	2175000	Col 0	23%	21:25
000007CA70:	EF	07	C6	07	D6	07	26 08
000007CA80:	CC	07	AA	07	A2	07	AC 07
000007CA90:	09	08	CA	07	31	07	5E 07
000007CAA0:	E6	07	BD	07	D8	07	2F 08
000007CAB0:	B3	07	91	07	8B	07	97 07
000007CAC0:	03	08	CB	07	4C	07	61 07
000007CAD0:	E0	07	BB	07	DC	07	33 08
000007CAE0:	A4	07	84	07	81	07	90 07
000007CAF0:	FF	07	CD	07	65	07	69 07
000007CB00:	DE	07	BC	07	DF	07	33 08
000007CB10:	9F	07	82	07	81	07	93 07
000007CB20:	FE	07	CE	07	7E	07	78 07
000007CB30:	DE	07	BD	07	DF	07	32 08
000007CB40:	A1	07	87	07	88	07	9B 07
000007CB50:	02	08	CF	07	93	07	89 07
000007CB60:	E4	07	C0	07	DD	07	2D 08
000007CB70:	A9	07	90	07	91	07	A3 07
000007CB80:	04	08	D0	07	A7	07	9C 07
000007CB90:	E8	07	C7	07	DF	07	29 08
000007CBA0:	B4	07	9B	07	9B	07	AB 07
000007CBB0:	03	08	D5	07	BB	07	B3 07
000007CBC0:	EA	07	CD	07	E3	07	25 08
000007CBD0:	C1	07	A6	07	A5	07	B3 07
000007CBE0:	01	08	DC	07	CE	07	C8 07
		0C	08	CE	07	24	07 60 07
		E9	07	BF	07	D6	07 2C 08
		BC	07	9A	07	93	07 9E 07
		06	08	CB	07	3E	07 5E 07
		E1	07	BB	07	DB	07 32 08
		AA	07	89	07	84	07 91 07
		01	08	CC	07	57	07 64 07
		DE	07	BB	07	DE	07 34 08
		A0	07	81	07	7F	07 90 07
		FF	07	CE	07	70	07 6F 07
		DD	07	BC	07	E0	07 34 08
		9F	07	84	07	84	07 96 07
		FF	07	CE	07	87	07 7F 07
		E2	07	BF	07	DE	07 2F 08
		A4	07	8C	07	8D	07 9F 07
		03	08	CF	07	9C	07 92 07
		E6	07	C3	07	DD	07 2B 08
		AE	07	96	07	96	07 A7 07
		04	08	D3	07	B1	07 A7 07
		E8	07	CA	07	E1	07 27 08
		BB	07	A1	07	A0	07 AF 07
		03	08	D8	07	C4	07 BD 07
		EA	07	D1	07	E6	07 22 08
		C8	07	AD	07	AA	07 B7 07

Рис. 64.1: FAR: массив 16-битных значений

Для примера я использовал файл содержащий 12-канальный сигнал оцифрованный при помощи 16-битного ADC².

²Analog-to-digital converter

64.3. НЕКОТОРЫЕ ПАТТЕРНЫ В БИНАРНЫХ ФАЙЛАХ

А вот пример очень типичного MIPS-кода. Как мы наверное помним, каждая инструкция в MIPS (а также в ARM в режиме ARM, или ARM64) имеет длину 32 бита (или 4 байта), так что такой код это массив 32-битных значений. Глядя на этот скриншот, можно увидеть некий узор. Вертикальные красные линии добавлены для ясности:

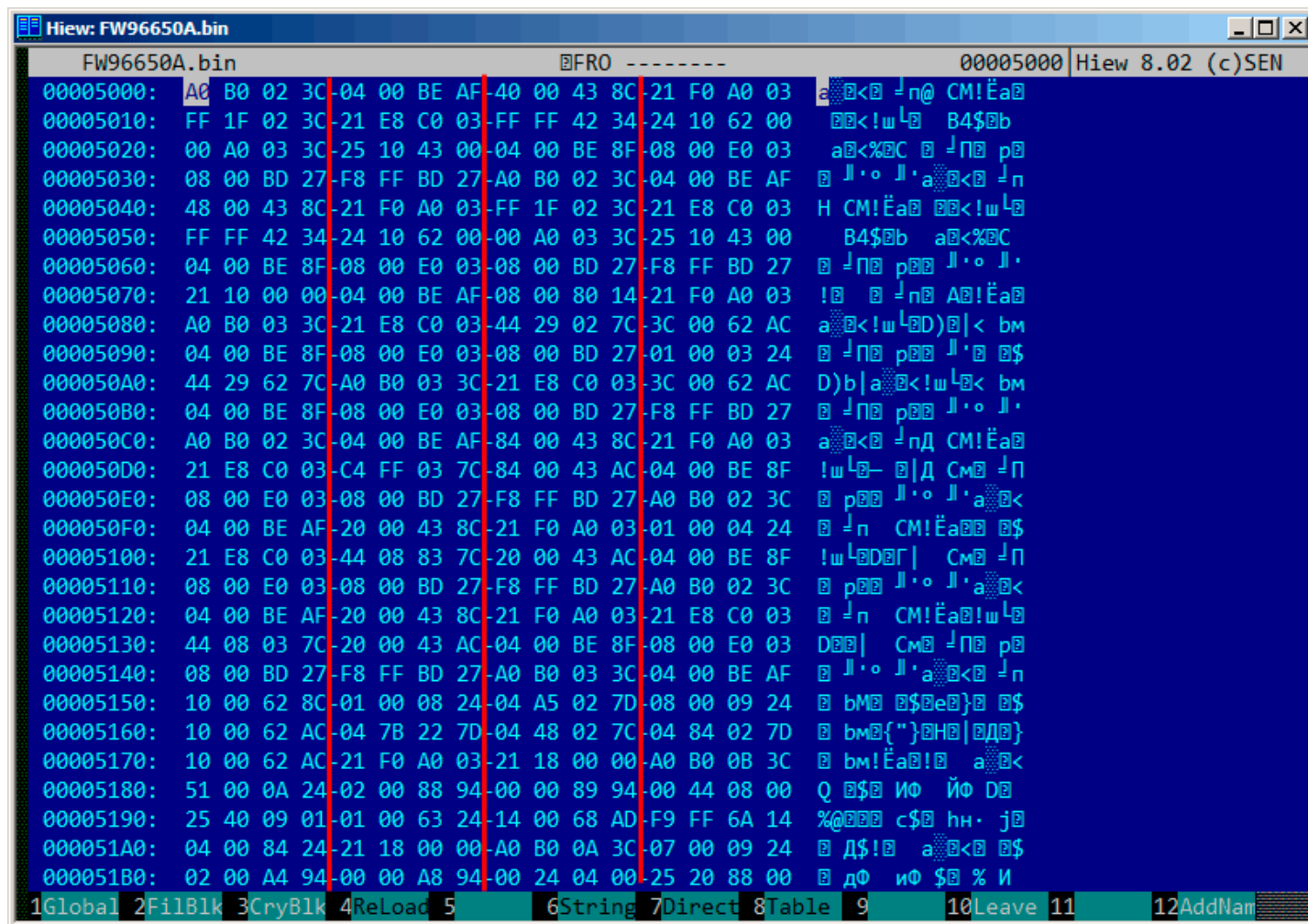


Рис. 64.2: Hiew: очень типичный код для MIPS

Еще пример таких файлов в этой книге: [88](#) (стр. [866](#)).

64.3.2. Разреженные файлы

Это разреженный файл, в котором данные разбросаны посреди почти пустого файла. Каждый символ пробела здесь на самом деле нулевой байт (который выглядит как пробел). Это файл для программирования FPGA (чип Altera Stratix GX). Конечно, такие файлы легко сжимаются, но подобные форматы очень популярны в научном и инженерном ПО, где быстрый доступ важен, а компактность — не очень.

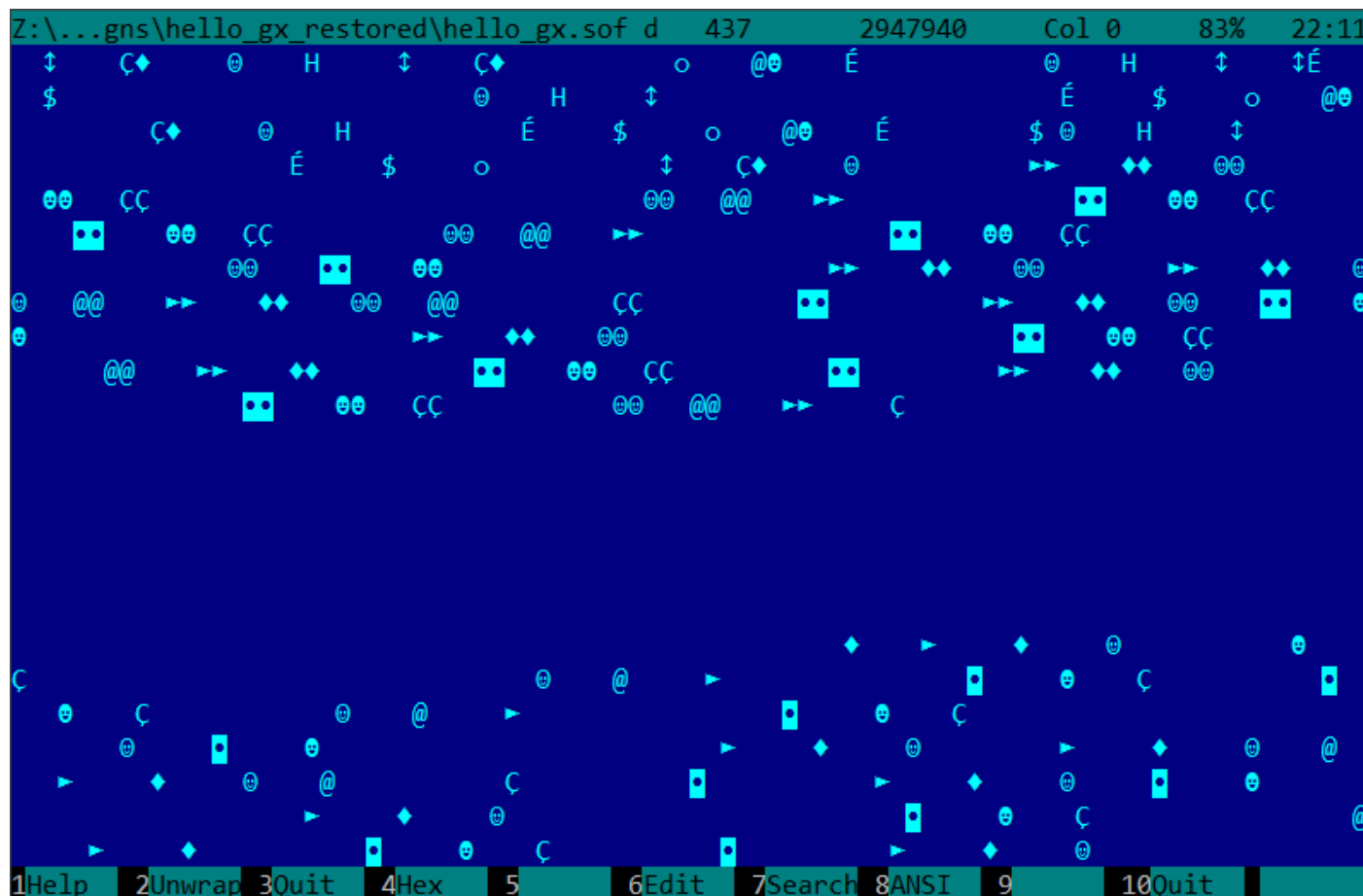


Рис. 64.3: FAR: Разреженный файл

64.3.3. Сжатый файл

Этот файл это просто некий сжатый архив. Он имеет довольно высокую энтропию и визуально выглядит просто хаотичным. Так выглядят сжатые и/или зашифрованные файлы.

Рис. 64.4: FAR: Сжатый файл


```
Z:\...\untu\ubuntu-15.10-desktop-i386.iso d    437    1226964992    Col 0    0%    21:59
```

```

PX$@mA      Am@      @      TF→@ñs@>↑, s@>↑, s@>↑, CEL@ $      $      φ
φ ` #      # @      @ s@>↑; @ @ @@ PX$@mA      Am@      @      TF→@ñs@>↑, s@
s@>↑, s@>↑, n %      % @      @ s@>↑; @ @ @+.DISKPX$@mA      Am@      @      TF
→@ñs@>↑% s@>↑, s@>↑ NM@@ .diskn &      & @      @ s@>↑; @ @ @+BOOT PX$@mA      Am@
@      TF→@ñs@>↑( s@>↑, s@>↑( NM@@ boot r (      ( @      @ s@>↑; @ @ @
+CASPER PX$@mA      Am@      @      TF→@ñs@>↑( s@>↑, s@>↑( NM@@ casper n )
) @      @ s@>↑; @ @ @+DISTSPX$@mA      Am@      @      TF→@ñs@>↑ s@>↑ s
s@>↑ NM@@ distr 1      1 @      @ s@>↑; @ @ @•INSTALLPX$@mA      Am@      @
TF→@ñs@>↑( s@>↑, s@>↑( NM@@ installv 2      2 H      H s@>↑; @ @ @ISOLINUX PX$@
mA      Am@      @      TF→@ñs@>↑( s@>↑( s@>↑( NM@@ isolinux z p@      @p`ñ
ñ s@>↑; @ @ MD5SUM.TXT PX$@$ü      ü$@      @      TF→@ñs@>↑: s@>↑; s@>↑
: NM@@ md5sum.txt n ;      ; @      @ s@>↑; @ @ @+PICS PX$@mA      Am@      @
TF→@ñs@>↑ s@>↑ s@>↑ NM@@ pics n <      < @      @ s@>↑; @ @ @+POOL PX$@mA
Am@      @      TF→@ñs@>↑ s@>↑ s@>↑ NM@@ pool r W      W @      @ s@>↑; @
@ @ •PRESEEDPX$@mA      Am@      @      TF→@ñs@>↑ s@>↑ s@>↑ NM@@ preseed
è S@      @Sa      as@>↑; @ @↓README.DISKDEFINES PX$@$ü      ü$@      @
TF→@ñs@>↑ s@>↑: s@>↑ NM±@ README.diskdefines x ≤      ≤      s@>↑; @ @ •UBUNT
U.PX$@mí      ím@      @      TF→@ñs@>↑ s@>↑ s@>↑ NM@@ ubuntuSL•@ @
```

```

1Help    2Unwrap    3Quit    4Hex    5    6Edit    7Search    8ANSI    9    10Quit
```

667

64.3.5. 32-битный x86 исполняемый код

Так выглядит 32-битный x86 исполняемый код. У него не очень высокая энтропия, потому что некоторые байты встречаются чаще других.



Рис. 64.6: FAR: Исполняемый 32-битных x86 код

64.3.6. Графические BMP-файлы

BMP-файлы не сжаты, так что каждый байт (или группа байт) описывают каждый пиксель. Я нашел эту картинку где-то внутри установленной Windows 8.1:



Рис. 64.7: Пример картинки

Вы видите что эта картинка имеет пиксели, которые вероятно не могут быть хорошо сжаты (в районе центра), но здесь есть длинные одноцветные линии вверху и внизу. Действительно, линии вроде этих выглядят как линии при просмотре этого файла:

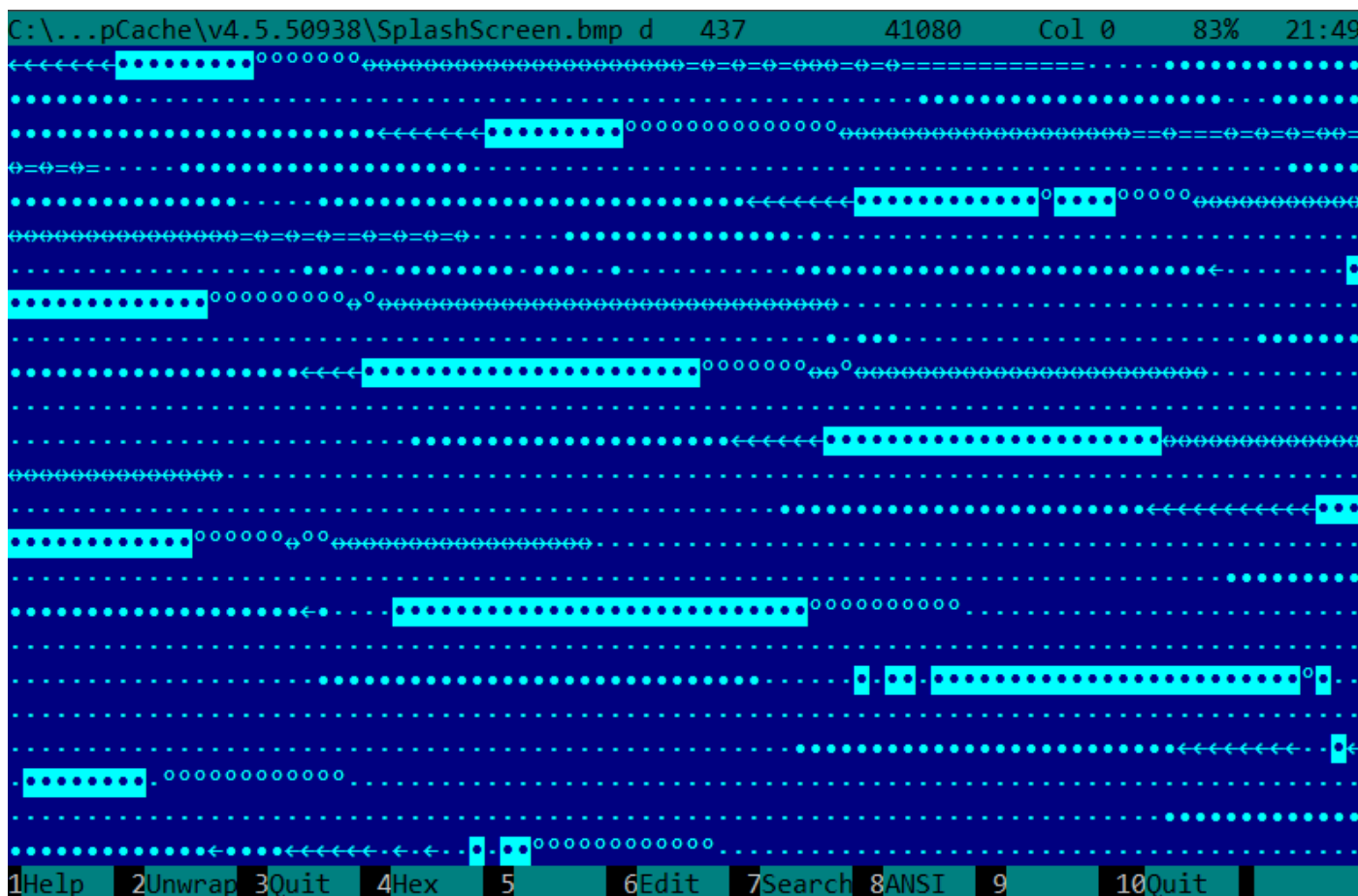


Рис. 64.8: Фрагмент BMP-файла

64.4. Сравнение «снимков» памяти

Метод простого сравнения двух снимков памяти для поиска изменений часто применялся для взлома игр на 8-битных компьютерах и взлома файлов с записанными рекордными очками.

К примеру, если вы имеете загруженную игру на 8-битном компьютере (где самой памяти не очень много, но игра занимает еще меньше), и вы знаете что сейчас у вас, условно, 100 пуль, вы можете сделать «снимок» всей памяти и

64.4. СРАВНЕНИЕ «СНИМКОВ» ПАМЯТИ

сохранить где-то. Затем просто стреляете куда угодно, у вас станет 99 пуль, сделать второй «снимок», и затем сравнить эти два снимка: где-то наверняка должен быть байт, который в начале был 100, а затем стал 99. Если учесть, что игры на тех маломощных домашних компьютерах обычно были написаны на ассемблере и подобные переменные там были глобальные, то можно с уверенностью сказать, какой адрес в памяти всегда отвечает за количество пуль. Если поискать в дизассемблированном коде игры все обращения по этому адресу, несложно найти код, отвечающий за уменьшение пуль и записать туда инструкцию `NOP` или несколько `NOP`-в, так мы получим игру в которой у игрока всегда будет 100 пуль, например. А так как игры на тех домашних 8-битных компьютерах всегда загружались по одним и тем же адресам, и версий одной игры редко когда было больше одной продолжительное время, то геймеры-энтузиасты знали, по какому адресу (используя инструкцию языка BASIC `POKE`) что записать после загрузки игры, чтобы хакнуть её. Это привело к появлению списков «читов» состоящих из инструкций `POKE`, публикуемых в журналах посвященным 8-битным играм. См. также: wikipedia.

Точно так же легко модифицировать файлы с сохраненными рекордами (кто сколько очков набрал), впрочем, это может сработать не только с 8-битными играми. Нужно заметить, какой у вас сейчас рекорд и где-то сохранить файл с очками. Затем, когда очков станет другое количество, просто сравнить два файла, можно даже DOS-утилитой `FC`⁴ (файлы рекордов, часто, бинарные). Где-то будут отличаться несколько байт, и легко будет увидеть, какие именно отвечают за количество очков. Впрочем, разработчики игр полностью осведомлены о таких хитростях и могут защититься от этого.

В каком-то смысле похожий пример в этой книге здесь : [87](#) (стр. [859](#)).

64.4.1. Реестр Windows

А еще можно вспомнить сравнение реестра Windows до инсталляции программы и после . Это также популярный метод поиска, какие элементы реестра программа использует. Наверное это причина, почему так популярны shareware-программы для очистки реестра в Windows.

64.4.2. Блинк-компаратор

Сравнение файлов или слепков памяти вообще, немного напоминает блинк-компаратор⁵: устройство, которое раньше использовали астрономы для поиска движущихся небесных объектов. Блинк-компаратор позволял быстро переключаться между двух отснятых в разное время кадров, и астроном мог увидеть разницу визуально. Кстати, при помощи блинк-компаратора, в 1930 был открыт Плутон.

⁴утилита MS-DOS для сравнения двух файлов побайтово

⁵<http://go.yurichev.com/17349>

Часть VI

Специфичное для ОС

Глава 65

Способы передачи аргументов при вызове функций

65.1. cdecl

Этот способ передачи аргументов через стек чаще всего используется в языках Си/Си++.

Вызывающая функция заталкивает в стек аргументы в обратном порядке: сначала последний аргумент в стек, затем предпоследний, и в самом конце — первый аргумент. Вызывающая функция должна также затем вернуть [указатель стека](#) в нормальное состояние, после возврата вызываемой функции.

Листинг 65.1: cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

65.2. stdcall

Это почти то же что и *cdecl*, за исключением того, что вызываемая функция сама возвращает `ESP` в нормальное состояние, выполнив инструкцию `RET x` вместо `RET`, где `x = количество_аргументов * sizeof(int)`¹. Вызывающая функция не будет корректировать [указатель стека](#), там нет инструкции `add esp, x`.

Листинг 65.2: stdcall

```
push arg3
push arg2
push arg1
call function

function:
... do something ...
ret 12
```

Этот способ используется почти везде в системных библиотеках win32, но не в win64 (о win64 смотрите ниже).

Например, мы можем взять функцию из 9.1 (стр. 93) и изменить её немного добавив модификатор `__stdcall`:

```
int __stdcall f2 (int a, int b, int c)
{
    return a*b+c;
};
```

Он будет скомпилирован почти так же как и 9.2 (стр. 93), но вы увидите `RET 12` вместо `RET`. `SP` не будет корректироваться в [вызывающей функции](#).

Как следствие, количество аргументов функции легко узнать из инструкции `RET n` просто разделите *n* на 4.

```

_a$ = 8                                ; size = 4
_b$ = 12                               ; size = 4
_c$ = 16                               ; size = 4
_f2@12 PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    imul    eax, DWORD PTR _b$[ebp]
    add     eax, DWORD PTR _c$[ebp]
    pop     ebp
    ret     12                          ; 0000000cH
_f2@12 ENDP

; ...
    push    3
    push    2
    push    1
    call    _f2@12
    push    eax
    push    OFFSET $SG81369
    call    _printf
    add     esp, 8

```

65.2.1. Функции с переменным количеством аргументов

Функции вроде `printf()`, должно быть, единственный случай функций в Си/Си++ с переменным количеством аргументов, но с их помощью можно легко проследить очень важную разницу между *cdecl* и *stdcall*. Начнем с того, что компилятор знает сколько аргументов было у `printf()`. Однако, вызываемая функция `printf()`, которая уже давно скомпилирована и находится в системной библиотеке MSVCRT.DLL (если говорить о Windows), не знает сколько аргументов ей передали, хотя может установить их количество по строке формата. Таким образом, если бы `printf()` была *stdcall*-функцией и возвращала [указатель стека](#) в первоначальное состояние подсчитав количество аргументов в строке формата, это была бы потенциально опасная ситуация, когда одна опечатка программиста могла бы вызывать неожиданные падения программы. Таким образом, для таких функций *stdcall* явно не подходит, а подходит *cdecl*.

65.3.fastcall

Это общее название для передачи некоторых аргументов через регистры, а всех остальных — через стек. На более старых процессорах, это работало потенциально быстрее чем *cdecl/stdcall* (ведь стек в памяти использовался меньше). Впрочем, на современных (намного более сложных) CPU, существенного выигрыша может и не быть.

Это не стандартизированный способ, поэтому разные компиляторы делают это по-своему. Разумеется, если у вас есть, скажем, две DLL, одна использует другую, и обе они собраны с *fastcall* но разными компиляторами, очень вероятно, будут проблемы.

MSVC и GCC передает первый и второй аргумент через `ECX` и `EDX` а остальные аргументы через стек.

[Указатель стека](#) должен быть возвращен в первоначальное состояние вызываемой функцией, как в случае *stdcall*.

Листинг 65.4:fastcall

```

push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
.. do something ..
ret 4

```

Например, мы можем взять функцию из 9.1 (стр. 93) и изменить её немного добавив модификатор `__fastcall`:

```

int __fastcall f3 (int a, int b, int c)
{
    return a*b+c;
}

```

};

Вот как он будет скомпилирован:

Листинг 65.5: Оптимизирующий MSVC 2010 /Ob0

```

_c$ = 8                                ; size = 4
@f3@12 PROC
; _a$ = ecx
; _b$ = edx
    mov     eax, ecx
    imul    eax, edx
    add     eax, DWORD PTR _c$[esp-4]
    ret     4
@f3@12 ENDP

; ...

    mov     edx, 2
    push    3
    lea     ecx, DWORD PTR [edx-1]
    call    @f3@12
    push    eax
    push    OFFSET $SG81390
    call    _printf
    add     esp, 8

```

Видно, что **вызываемая функция** сама возвращает **SP** при помощи инструкции **RETN** с операндом. Так что и здесь можно легко вычислять количество аргументов.

65.3.1. GCC regparm

Это в некотором роде, развитие *fastcall*². Опцией **-mregparm=x** можно указывать, сколько аргументов компилятор будет передавать через регистры. Максимально 3. В этом случае будут задействованы регистры **EAX**, **EDX** и **ECX**.

Разумеется, если аргументов у функции меньше трех, то будет задействована только часть регистров.

Вызывающая функция возвращает **указатель стека** в первоначальное состояние.

Для примера, см. (20.1.1 (стр. 301)).

65.3.2. Watcom/OpenWatcom

Здесь это называется «register calling convention». Первые 4 аргумента передаются через регистры **EAX**, **EDX**, **EBX** и **ECX**. Все остальные — через стек. Эти функции имеют символ подчеркивания, добавленный к концу имени функции, для отличия их от тех, которые имеют другой способ передачи аргументов.

65.4. thiscall

В Си++, это передача в функцию-метод указателя *this* на объект.

В MSVC указатель *this* обычно передается в регистре **ECX**.

В GCC указатель *this* обычно передается как самый первый аргумент. Таким образом, внутри будет видно: у всех функций-методов на один аргумент больше.

Для примера, см. (52.1.1 (стр. 539)).

²<http://go.yurichev.com/17040>

65.5. x86-64

65.5.1. Windows x64

В win64 метод передачи всех параметров немного похож на `fastcall`. Первые 4 аргумента записываются в регистры `RCX`, `RDX`, `R8`, `R9`, а остальные — в стек. Вызывающая функция также должна подготовить место из 32 байт или для четырех 64-битных значений, чтобы вызываемая функция могла сохранить там первые 4 аргумента. Короткие функции могут использовать переменные прямо из регистров, но большие могут сохранять их значения на будущее.

Вызывающая функция должна вернуть [указатель стека](#) в первоначальное состояние.

Это же соглашение используется и в системных библиотеках Windows x86-64 (вместо `stdcall` в win32).

Пример:

```
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
    printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main()
{
    f1(1,2,3,4,5,6,7);
};
```

Листинг 65.6: MSVC 2012 /Ob

```
$SG2937 DB      '%d %d %d %d %d %d %d', 0aH, 00H

main          PROC
              sub     rsp, 72                      ; 00000048H

              mov     DWORD PTR [rsp+48], 7
              mov     DWORD PTR [rsp+40], 6
              mov     DWORD PTR [rsp+32], 5
              mov     r9d, 4
              mov     r8d, 3
              mov     edx, 2
              mov     ecx, 1
              call    f1

              xor     eax, eax
              add     rsp, 72                      ; 00000048H
              ret     0
main          ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1          PROC
$LN3:
              mov     DWORD PTR [rsp+32], r9d
              mov     DWORD PTR [rsp+24], r8d
              mov     DWORD PTR [rsp+16], edx
              mov     DWORD PTR [rsp+8], ecx
              sub     rsp, 72                      ; 00000048H

              mov     eax, DWORD PTR g$[rsp]
              mov     DWORD PTR [rsp+56], eax
              mov     eax, DWORD PTR f$[rsp]
              mov     DWORD PTR [rsp+48], eax
              mov     eax, DWORD PTR e$[rsp]
              mov     DWORD PTR [rsp+40], eax
              mov     eax, DWORD PTR d$[rsp]
```

65.5. X86-64

```

mov     DWORD PTR [rsp+32], eax
mov     r9d, DWORD PTR c$[rsp]
mov     r8d, DWORD PTR b$[rsp]
mov     edx, DWORD PTR a$[rsp]
lea     rcx, OFFSET FLAT:$SG2937
call    printf

add     rsp, 72                                ; 00000048H
ret     0
f1      ENDP

```

Здесь мы легко видим, как 7 аргументов передаются: 4 через регистры и остальные 3 через стек . Код пролога функции f1() сохраняет аргументы в «scratch space» — место в стеке предназначенное именно для этого . Это делается потому что компилятор может быть не уверен, достаточно ли ему будет остальных регистров для работы исключая эти 4, которые иначе будут заняты аргументами до конца исполнения функции . Выделение «scratch space» в стеке лежит на ответственности вызывающей функции.

Листинг 65.7: Оптимизирующий MSVC 2012 /Ob

```

$SG2777 DB      '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        sub     rsp, 72                                ; 00000048H

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     DWORD PTR [rsp+32], r9d
        mov     r9d, r8d
        mov     r8d, edx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG2777
        call    printf

        add     rsp, 72                                ; 00000048H
        ret     0
f1      ENDP

main    PROC
        sub     rsp, 72                                ; 00000048H

        mov     edx, 2
        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        lea     r9d, QWORD PTR [rdx+2]
        lea     r8d, QWORD PTR [rdx+1]
        lea     ecx, QWORD PTR [rdx-1]
        mov     DWORD PTR [rsp+32], 5
        call    f1

        xor     eax, eax
        add     rsp, 72                                ; 00000048H
        ret     0
main    ENDP

```

Если компилировать этот пример с оптимизацией, то выйдет почти то же самое, только «scratch space» не используется, потому что незначем.

Обратите также внимание на то как MSVC 2012 оптимизирует примитивную загрузку значений в регистры используя

`LEA` (A.6.2 (стр. 917)) . `MOV` здесь был бы на 1 байт длиннее (5 вместо 4).

Еще один пример подобного: 75.1 (стр. 737).

Windows x64: Передача *this* (Си/Си++)

Указатель *this* передается через `RCX` , первый аргумент метода через `RDX` , и т.д. Для примера, см. также: 52.1.1 (стр. 541).

65.5.2. Linux x64

Метод передачи аргументов в Linux для x86-64 почти такой же, как и в Windows, но 6 регистров используется вместо 4 (`RDI` , `RSI` , `RDX` , `RCX` , `R8` , `R9`), и здесь нет «scratch space», но *callee* может сохранять значения регистров в стеке, если ему это нужно.

Листинг 65.8: Оптимизирующий GCC 4.7.3

```
.LC0:
    .string "%d %d %d %d %d %d %d\n"
f1:
    sub    rsp, 40
    mov    eax, DWORD PTR [rsp+48]
    mov    DWORD PTR [rsp+8], r9d
    mov    r9d, ecx
    mov    DWORD PTR [rsp], r8d
    mov    ecx, esi
    mov    r8d, edx
    mov    esi, OFFSET FLAT:.LC0
    mov    edx, edi
    mov    edi, 1
    mov    DWORD PTR [rsp+16], eax
    xor    eax, eax
    call   __printf_chk
    add    rsp, 40
    ret
main:
    sub    rsp, 24
    mov    r9d, 6
    mov    r8d, 5
    mov    DWORD PTR [rsp], 7
    mov    ecx, 4
    mov    edx, 3
    mov    esi, 2
    mov    edi, 1
    call   f1
    add    rsp, 24
    ret
```

Н.В.: здесь значения записываются в 32-битные части регистров (например `EAX`) а не в весь 64-битный регистр (`RAX`). Это связано с тем что в x86-64, запись в младшую 32-битную часть 64-битного регистра автоматически обнуляет старшие 32 бита. Вероятно, это так решили в AMD для упрощения портирования кода под x86-64.

65.6. Возвращение переменных типа *float*, *double*

Во всех соглашениях кроме Win64, переменная типа *float* или *double* возвращается через регистр FPU `ST(0)` .

В Win64 переменные типа *float* и *double* возвращаются в младших 16-и или 32-х битах регистра `XMM0` .

65.7. Модификация аргументов

Иногда программисты на Си/Си++ (и не только этих ЯП) задаются вопросом, что может случиться, если модифицировать аргументы? Ответ прост: аргументы хранятся в стеке, именно там и будет происходить модификация. А вызывающие функции не используют их после вызова функции (автор этих строк никогда не видел в своей практике обратного случая).

```
#include <stdio.h>

void f(int a, int b)
{
    a=a+b;
    printf ("%d\n", a);
};
```

Листинг 65.9: MSVC 2012

```
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    mov     DWORD PTR _a$[ebp], eax
    mov     ecx, DWORD PTR _a$[ebp]
    push    ecx
    push    OFFSET $SG2938 ; '%d', 0aH
    call    _printf
    add     esp, 8
    pop     ebp
    ret     0
_f ENDP
```

Следовательно, модифицировать аргументы функции можно запросто. Разумеется, если это не *references* в Си++ (52.3 (стр. 555)), и если вы не модифицируете данные по указателю, то эффект не будет распространяться за пределами текущей функции.

Теоретически, после возврата из *callee*, функция-*caller* могла бы получить модифицированный аргумент и использовать его как-то. Может быть, если бы она была написана на языке ассемблера. Но стандарты языков Си/Си++ не предлагают никакого способа достигнуть к ним.

65.8. Указатель на аргумент функции

...и даже более того, можно взять указатель на аргумент функции и передать его в другую функцию:

```
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
    modify_a (&a);
    printf ("%d\n", a);
};
```

Трудно понять, как это работает, пока мы не посмотрим на код:

Листинг 65.10: Оптимизирующий MSVC 2010

```
$SG2796 DB      '%d', 0aH, 00H

_a$ = 8
_f PROC
    lea     eax, DWORD PTR _a$[esp-4] ; just get the address of value in local stack
    push    eax                       ; and pass it to modify_a()
    call    _modify_a
    mov     ecx, DWORD PTR _a$[esp]   ; reload it from the local stack
    push    ecx                       ; and pass it to printf()
    push    OFFSET $SG2796 ; '%d'
    call    _printf
    add     esp, 12
    ret     0
_f ENDP
```

65.8. УКАЗАТЕЛЬ НА АРГУМЕНТ ФУНКЦИИ

Адрес места в стеке где была передана *a* просто передается в другую функцию. Она модифицирует переменную по этому адресу, и затем `printf()` выведет модифицированное значение.

Наблюдательный читатель может спросить, а что насчет тех соглашений о вызовах, где аргументы функции передаются в регистрах?

Это та ситуация где используется *Shadow Space*. Так что входящее значение копируется из регистра в *Shadow Space* в локальном стеке и затем это адрес передается в другую функцию:

Листинг 65.11: Оптимизирующий MSVC 2012 x64

```
$SG2994 DB      '%d', 0aH, 00H

a$ = 48
f      PROC
      mov     DWORD PTR [rsp+8], ecx    ; save input value in Shadow Space
      sub     rsp, 40
      lea     rcx, QWORD PTR a$[rsp]   ; get address of value and pass it to modify_a()
      call    modify_a
      mov     edx, DWORD PTR a$[rsp]   ; reload value from Shadow Space and pass it to printf()
      lea     rcx, OFFSET FLAT:$SG2994 ; '%d'
      call    printf
      add     rsp, 40
      ret     0
f      ENDP
```

GCC также записывает входное значение в локальный стек:

Листинг 65.12: Оптимизирующий GCC 4.9.1 x64

```
.LC0:
.string "%d\n"
f:
      sub     rsp, 24
      mov     DWORD PTR [rsp+12], edi   ; store input value to the local stack
      lea     rdi, [rsp+12]             ; take an address of the value and pass it to modify_a()
      call    modify_a
      mov     edx, DWORD PTR [rsp+12]   ; reload value from the local stack and pass it to printf()
      mov     esi, OFFSET FLAT:.LC0    ; '%d'
      mov     edi, 1
      xor     eax, eax
      call    __printf_chk
      add     rsp, 24
      ret
```

GCC для ARM64 делает то же самое, но это пространство здесь называется *Register Save Area*:

Листинг 65.13: Оптимизирующий GCC 4.9.1 ARM64

```
f:
      stp     x29, x30, [sp, -32]!
      add     x29, sp, 0                ; setup FP
      add     x1, x29, 32               ; calculate address of variable in Register Save Area
      str     w0, [x1,-4]!              ; store input value there
      mov     x0, x1                   ; pass address of variable to the modify_a()
      bl      modify_a
      ldr     w1, [x29,28]              ; load value from the variable and pass it to printf()
      adrp    x0, .LC0 ; '%d'
      add     x0, x0, :lo12:.LC0
      bl      printf                   ; call printf()
      ldp     x29, x30, [sp], 32
      ret
.LC0:
.string "%d\n"
```

Кстати, похожее использование *Shadow Space* разбирается здесь: [47.1.2](#) (стр. 511).

Глава 66

Thread Local Storage

Это область данных, отдельная для каждого треда. Каждый тред может хранить там то, что ему нужно. Один из известных примеров, это стандартная глобальная переменная в Си *errno*. Несколько тредов одновременно могут вызывать функции возвращающие код ошибки в *errno*, поэтому глобальная переменная здесь не будет работать корректно, для мультитредовых программ *errno* нужно хранить в [TLS](#).

В C++11 ввели модификатор *thread_local*, показывающий что каждый тред будет иметь свою версию этой переменной, и её можно инициализировать, и она расположена в [TLS](#)¹:

Листинг 66.1: C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
    std::cout << tmp << std::endl;
};
```

Компилируется в MinGW GCC 4.8.1, но не в MSVC 2012.

Если говорить о PE-файлах, то в исполняемом файле значение *tmp* будет размещено именно в секции отведенной [TLS](#).

66.1. Вернемся к линейному конгруэнтному генератору

Рассмотренный ранее [21](#) (стр. [334](#)) генератор псевдослучайных чисел имеет недостаток: он не пригоден для многопоточной среды, потому что переменная его внутреннего состояния может быть прочитана и/или модифицирована в разных потоках одновременно.

66.1.1. Win32

Неинициализированные данные в TLS

Одно из решений — это добавить модификатор `__declspec( thread )` к глобальной переменной, и теперь она будет выделена в [TLS](#) (строка 9):

```
1 #include <stdint.h>
2 #include <windows.h>
3 #include <winnt.h>
4
5 // from the Numerical Recipes book:
6 #define RNG_a 1664525
7 #define RNG_c 1013904223
8
9 __declspec( thread ) uint32_t rand_state;
```

¹ В C11 также есть поддержка тредов, хотя и опциональная

66.1. ВЕРНЕМСЯ К ЛИНЕЙНОМУ КОНГРУЭНТНОМУ ГЕНЕРАТОРУ

```
10
11 void my_srand (uint32_t init)
12 {
13     rand_state=init;
14 }
15
16 int my_rand ()
17 {
18     rand_state=rand_state*RNG_a;
19     rand_state=rand_state+RNG_c;
20     return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25     my_srand(0x12345678);
26     printf ("%d\n", my_rand());
27 };
```

Hiew показывает что в исполняемом файле теперь есть новая PE-секция: `.tls`.

Листинг 66.2: Оптимизирующий MSVC 2013 x86

```
_TLS      SEGMENT
_rand_state DD  01H DUP (?)
_TLS      ENDS

_DATA     SEGMENT
$SG84851 DB  '%d', 0aH, 00H
_DATA     ENDS
_TEXT     SEGMENT

_init$ = 8 ; size = 4
_my_srand PROC
; FS:0=address of TIB
    mov     eax, DWORD PTR fs:__tls_array ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
    mov     ecx, DWORD PTR __tls_index
    mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
    mov     eax, DWORD PTR _init$[esp-4]
    mov     DWORD PTR _rand_state[ecx], eax
    ret     0
_my_srand ENDP

_my_rand PROC
; FS:0=address of TIB
    mov     eax, DWORD PTR fs:__tls_array ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
    mov     ecx, DWORD PTR __tls_index
    mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
    imul    eax, DWORD PTR _rand_state[ecx], 1664525
    add     eax, 1013904223 ; 3c6ef35fH
    mov     DWORD PTR _rand_state[ecx], eax
    and     eax, 32767 ; 00007fffH
    ret     0
_my_rand ENDP

_TEXT     ENDS
```

`rand_state` теперь в `TLS`-сегменте и у каждого потока есть своя версия этой переменной.

Вот как к ней обращаться: загрузить адрес `TIB` из `FS:2Ch`, затем прибавить дополнительный индекс (если нужно), затем вычислить адрес `TLS`-сегмента.

Затем можно обращаться к переменной `rand_state` через регистр `ECX`, который указывает на свою область в каждом потоке.

Селектор `FS:` знаком любому reverse engineer-у, он всегда указывает на `TIB`, чтобы всегда можно было загружать данные специфичные для текущего потока.

В Win64 используется селектор **GS:** и адрес **TLS** теперь 0x58:

Листинг 66.3: Оптимизирующий MSVC 2013 x64

```

_TLS    SEGMENT
rand_state DD    01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
$SG85451 DB      '%d', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT

init$ = 8
my_srand PROC
    mov     edx, DWORD PTR _tls_index
    mov     rax, QWORD PTR gs:88 ; 58h
    mov     r8d, OFFSET FLAT:rand_state
    mov     rax, QWORD PTR [rax+rdx*8]
    mov     DWORD PTR [r8+rax], ecx
    ret     0
my_srand ENDP

my_rand PROC
    mov     rax, QWORD PTR gs:88 ; 58h
    mov     ecx, DWORD PTR _tls_index
    mov     edx, OFFSET FLAT:rand_state
    mov     rcx, QWORD PTR [rax+rcx*8]
    imul    eax, DWORD PTR [rcx+rdx], 1664525      ; 0019660dH
    add     eax, 1013904223                       ; 3c6ef35fH
    mov     DWORD PTR [rcx+rdx], eax
    and     eax, 32767                             ; 00007fffH
    ret     0
my_rand ENDP

_TEXT   ENDS

```

Инициализированные данные в TLS

Скажем, мы хотим, чтобы в переменной **rand_state** в самом начале было какое-то значение, и если программист забудет инициализировать генератор, то **rand_state** все же будет инициализирована какой-то константой (строка 9):

```

1  #include <stdint.h>
2  #include <windows.h>
3  #include <winnt.h>
4
5  // from the Numerical Recipes book:
6  #define RNG_a 1664525
7  #define RNG_c 1013904223
8
9  __declspec( thread ) uint32_t rand_state=1234;
10
11 void my_srand (uint32_t init)
12 {
13     rand_state=init;
14 }
15
16 int my_rand ()
17 {
18     rand_state=rand_state*RNG_a;
19     rand_state=rand_state+RNG_c;
20     return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25     printf ("%d\n", my_rand());
26 };

```


66.1. ВЕРНЕМСЯ К ЛИНЕЙНОМУ КОНГРУЭНТНОМУ ГЕНЕРАТОРУ

Код ничем не отличается от того, что мы уже видели, но вот что мы видим в IDA:

```
.tls:00404000 ; Segment type: Pure data
.tls:00404000 ; Segment permissions: Read/Write
.tls:00404000 _tls          segment para public 'DATA' use32
.tls:00404000          assume cs:_tls
.tls:00404000          ;org 404000h
.tls:00404000 TlsStart      db      0          ; DATA XREF: .rdata:TlsDirectory
.tls:00404001          db      0
.tls:00404002          db      0
.tls:00404003          db      0
.tls:00404004          dd 1234
.tls:00404008 TlsEnd        db      0          ; DATA XREF: .rdata:TlsEnd_ptr
...
```

Там 1234 и теперь, во время запуска каждого нового потока, новый TLS будет выделен для нового потока, и все эти данные, включая 1234, будут туда скопированы.

Вот типичный сценарий:

- Запустился поток А. TLS создался для него, 1234 скопировалось в `rand_state`.
- Функция `my_rand()` была вызвана несколько раз в потоке А. `rand_state` теперь содержит что-то неравное 1234.
- Запустился поток Б. TLS создался для него, 1234 скопировалось в `rand_state`, а в это же время, поток А имеет какое-то другое значение в этой переменной.

TLS-коллбэки

Но что если переменные в TLS должны быть установлены в значения, которые должны быть подготовлены каким-то необычным образом?

Скажем, у нас есть следующая задача: программист может забыть вызвать функцию `my_srand()` для инициализации ГПСЧ, но генератор должен быть инициализирован на старте чем-то по-настоящему случайным а не 1234.

Вот случай где можно применить TLS-коллбэки.

Нижеследующий код не очень портативный из-за хака, но тем не менее, вы поймете идею.

Мы здесь добавляем функцию (`tls_callback()`), которая вызывается *перед* стартом процесса и/или потока.

Функция будет инициализировать ГПСЧ значением возвращенным функцией `GetTickCount()`.

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
    my_srand (GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{

```

66.1. ВЕРНЕМСЯ К ЛИНЕЙНОМУ КОНГРУЭНТНОМУ ГЕНЕРАТОРУ

```
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    // rand_state is already initialized at the moment (using GetTickCount())
    printf ("%d\n", my_rand());
};
```

Посмотрим в IDA:

Листинг 66.4: Оптимизирующий MSVC 2013

```
.text:00401020 TlsCallback_0 proc near ; DATA XREF: .rdata:TlsCallbacks
.text:00401020 call ds:GetTickCount
.text:00401026 push eax
.text:00401027 call my_srand
.text:0040102C pop ecx
.text:0040102D retn 0Ch
.text:0040102D TlsCallback_0 endp

...

.rdata:004020C0 TlsCallbacks dd offset TlsCallback_0 ; DATA XREF: .rdata:TlsCallbacks_ptr
...

.rdata:00402118 TlsDirectory dd offset TlsStart
.rdata:0040211C TlsEnd_ptr dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h
```

TLS-коллбэки иногда используются в процедурах распаковки для запутывания их работы.

Некоторые люди могут быть в неведении что какой-то код уже был исполнен прямо перед [OEP²](#).

66.1.2. Linux

Вот как глобальная переменная локальная для потока определяется в GCC:

```
__thread uint32_t rand_state=1234;
```

Этот модификатор не стандартный для Си/Си++, он присутствует только в GCC

³.

Селектор **GS:** также используется для доступа к [TLS](#), но немного иначе:

Листинг 66.5: Оптимизирующий GCC 4.8.1 x86

```
.text:08048460 my_srand proc near
.text:08048460
.text:08048460 arg_0 = dword ptr 4
.text:08048460
.text:08048460 mov eax, [esp+arg_0]
.text:08048464 mov gs:0FFFFFFFCh, eax
.text:0804846A retn
.text:0804846A my_srand endp

.text:08048470 my_rand proc near
.text:08048470 imul eax, gs:0FFFFFFFCh, 19660Dh
.text:0804847B add eax, 3C6EF35Fh
.text:08048480 mov gs:0FFFFFFFCh, eax
```

²Original Entry Point

³<http://go.yurichev.com/17062>

66.1. ВЕРНЕМСЯ К ЛИНЕЙНОМУ КОНГРУЭНТНОМУ ГЕНЕРАТОРУ

```
.text:08048486          and     eax, 7FFFh  
.text:0804848B          retn  
.text:0804848B my_rand  endp
```

Еще об этом: [Ulrich Drepper, *ELF Handling For Thread-Local Storage*, (2013)]⁴.

⁴Также доступно здесь: <http://go.yurichev.com/17272>

Глава 67

Системные вызовы (syscall-ы)

Как известно, все работающие процессы в ОС делятся на две категории: имеющие полный доступ ко всему «железу» («kernel space») и не имеющие («user space»).

В первой категории ядро ОС и, обычно, драйвера .

Во второй категории всё прикладное ПО.

Например, ядро Linux в *kernel space*, но Glibc в *user space*.

Это разделение очень важно для безопасности ОС: очень важно чтобы никакой процесс не мог испортить что-то в других процессах или даже в самом ядре ОС . С другой стороны, падающий драйвер или ошибка внутри ядра ОС обычно приводит к kernel panic или BSOD¹.

Защита x86-процессора устроена так что возможно разделить всё на 4 слоя защиты (rings), но и в Linux, и в Windows, используются только 2 : ring0 («kernel space») и ring3 («user space»).

Системные вызовы (syscall-ы) это точка где соединяются вместе оба эти пространства. Это, можно сказать, самое главное API предоставляемое прикладному ПО.

В Windows NT таблица сисколлов находится в SSDT² .

Работа через syscall-ы популярна у авторов шеллкодов и вирусов, потому что там обычно бывает трудно определить адреса нужных функций в системных библиотеках, а syscall-ами проще пользоваться, хотя и придется писать больше кода из-за более низкого уровня абстракции этого API . Также нельзя еще забывать, что номера syscall-ов могут отличаться от версии к версии OS.

67.1. Linux

В Linux вызов syscall-а обычно происходит через `int 0x80` . В регистре `EAX` передается номер вызова, в остальных регистрах — параметры.

Листинг 67.1: Простой пример использования пары syscall-ов

```
section .text
global _start

_start:
    mov     edx,len ; buffer len
    mov     ecx,msg ; buffer
    mov     ebx,1   ; file descriptor. 1 is for stdout
    mov     eax,4   ; syscall number. 4 is for sys_write
    int     0x80

    mov     eax,1   ; syscall number. 1 is for sys_exit
    int     0x80

section .data

msg      db  'Hello, world!',0xa
len      equ $ - msg
```

¹Blue Screen of Death

²System Service Dispatch Table

Компиляция:

```
nasm -f elf32 1.s  
ld 1.o
```

Полный список syscall-ов в Linux: <http://go.yurichev.com/17319>.

Для перехвата и трассировки системных вызовов в Linux, можно применять `strace`([72](#) (стр. [730](#))).

67.2. Windows

Вызов происходит через `int 0x2e` либо используя специальную x86-инструкцию `SYSENTER`.

Полный список syscall-ов в Windows: <http://go.yurichev.com/17320>.

Смотрите также:

«Windows Syscall Shellcode» by Piotr Bania:

<http://go.yurichev.com/17321>.

Глава 68

Linux

68.1. Адресно-независимый код

Во время анализа динамических библиотек (.so) в Linux, часто можно заметить такой шаблонный код:

Листинг 68.1: libc-2.17.so x86

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near          ; CODE XREF: sub_17350+3
.text:0012D5E3                                     ; sub_173CC+4 ...
.text:0012D5E3             mov     ebx, [esp+0]
.text:0012D5E6             retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp

...

.text:000576C0 sub_576C0      proc near                ; CODE XREF: tmpfile+73
...

.text:000576C0             push    ebp
.text:000576C1             mov     ecx, large gs:0
.text:000576C8             push    edi
.text:000576C9             push    esi
.text:000576CA             push    ebx
.text:000576CB             call   __x86_get_pc_thunk_bx
.text:000576D0             add     ebx, 157930h
.text:000576D6             sub     esp, 9Ch

...

.text:000579F0             lea     eax, (a__gen_tempname - 1AF000h)[ebx] ; "__gen_tempname"
.text:000579F6             mov     [esp+0ACh+var_A0], eax
.text:000579FA             lea     eax, (a__SysdepsPosix - 1AF000h)[ebx] ; "../sysdeps/posix/↵
↵  tmpname.c"
.text:00057A00             mov     [esp+0ACh+var_A8], eax
.text:00057A04             lea     eax, (aInvalidKindIn_ - 1AF000h)[ebx] ; "! \"invalid KIND in ↵
↵  __gen_tempname\""
.text:00057A0A             mov     [esp+0ACh+var_A4], 14Ah
.text:00057A12             mov     [esp+0ACh+var_AC], eax
.text:00057A15             call   __assert_fail
```

Все указатели на строки корректируются при помощи некоторой константы из регистра **EBX**, которая вычисляется в начале каждой функции.

Это так называемый адресно-независимый код (**PIC**), он предназначен для исполнения будучи расположенным по любому адресу в памяти, вот почему он не содержит никаких абсолютных адресов в памяти.

PIC был очень важен в ранних компьютерных системах и важен сейчас во встраиваемых¹, не имеющих поддержки виртуальной памяти (все процессы расположены в одном непрерывном блоке памяти). Он до сих пор используется в *NIX системах для динамических библиотек, потому что динамическая библиотека может использоваться одновременно в нескольких процессах, будучи загружена в память только один раз. Но все эти процессы могут загрузить одну и ту же

¹embedded

68.1. АДРЕСНО-НЕЗАВИСИМЫЙ КОД

динамическую библиотеку по разным адресам, вот почему динамическая библиотека должна работать корректно, не привязываясь к абсолютным адресам.

Простой эксперимент:

```
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

Скомпилируем в GCC 4.7.3 и посмотрим итоговый файл .so в [IDA](#):

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Листинг 68.2: GCC 4.7.3

```
.text:00000440      public __x86_get_pc_thunk_bx
.text:00000440 __x86_get_pc_thunk_bx proc near          ; CODE XREF: _init_proc+4
.text:00000440                                         ; deregister_tm_clones+4 ...
.text:00000440      mov     ebx, [esp+0]
.text:00000443      retn
.text:00000443 __x86_get_pc_thunk_bx endp

.text:00000570      public f1
.text:00000570 f1      proc near
.text:00000570
.text:00000570 var_1C      = dword ptr -1Ch
.text:00000570 var_18      = dword ptr -18h
.text:00000570 var_14      = dword ptr -14h
.text:00000570 var_8       = dword ptr -8
.text:00000570 var_4       = dword ptr -4
.text:00000570 arg_0      = dword ptr 4
.text:00000570
.text:00000570      sub     esp, 1Ch
.text:00000570      mov     [esp+1Ch+var_8], ebx
.text:00000577      call    __x86_get_pc_thunk_bx
.text:0000057C      add     ebx, 1A84h
.text:00000582      mov     [esp+1Ch+var_4], esi
.text:00000586      mov     eax, ds:(global_variable_ptr - 2000h)[ebx]
.text:0000058C      mov     esi, [eax]
.text:0000058E      lea     eax, (aReturningD - 2000h)[ebx] ; "returning %d\n"
.text:00000594      add     esi, [esp+1Ch+arg_0]
.text:00000598      mov     [esp+1Ch+var_18], eax
.text:0000059C      mov     [esp+1Ch+var_1C], 1
.text:000005A3      mov     [esp+1Ch+var_14], esi
.text:000005A7      call    __printf_chk
.text:000005AC      mov     eax, esi
.text:000005AE      mov     ebx, [esp+1Ch+var_8]
.text:000005B2      mov     esi, [esp+1Ch+var_4]
.text:000005B6      add     esp, 1Ch
.text:000005B9      retn
.text:000005B9 f1      endp
```

Так и есть: указатели на строку «*returning %d\n*» и переменную *global_variable* корректируются при каждом исполнении функции.

Функция `__x86_get_pc_thunk_bx()` возвращает адрес точки после вызова самой себя (здесь: `0x57C`) в `EBX`. Это очень простой способ получить значение указателя на текущую инструкцию (`EIP`) в произвольном месте.

Константа `0x1A84` связана с разницей между началом этой функции и так называемой *Global Offset Table Procedure Linkage Table* (GOT PLT), секцией, сразу же за *Global Offset Table* (GOT), где находится указатель на *global_variable*. [IDA](#) показывает смещения уже обработанными, чтобы их было проще понимать, но на самом деле код такой:

```
.text:00000577      call    __x86_get_pc_thunk_bx
```

68.2. Трюк с LD_PRELOAD в Linux

```
.text:0000057C      add     ebx, 1A84h
.text:00000582      mov     [esp+1Ch+var_4], esi
.text:00000586      mov     eax, [ebx-0Ch]
.text:0000058C      mov     esi, [eax]
.text:0000058E      lea     eax, [ebx-1A30h]
```

Так что, `EBX` указывает на секцию `GOT PLT` и для вычисления указателя на `global_variable`, которая хранится в `GOT`, нужно вычесть `0xC`. А чтобы вычислить указатель на «`returning %d\n`», нужно вычесть `0x1A30`.

Кстати, вот зачем в AMD64 появилась поддержка адресации относительно `RIP`², просто для упрощения PIC-кода.

Скомпилируем тот же код на Си при помощи той же версии GCC, но для x64.

IDA упростит код на выходе убирая упоминания `RIP`, так что будем использовать `objdump` вместо нее:

```
0000000000000720 <f1>:
720:  48 8b 05 b9 08 20 00    mov     rax,QWORD PTR [rip+0x2008b9]      # 200fe0 <_DYNAMIC+0x1d0>
727:  53                     push    rbx
728:  89 fb                     mov     ebx,edi
72a:  48 8d 35 20 00 00 00    lea     rsi,[rip+0x20]                    # 751 <_fini+0x9>
731:  bf 01 00 00 00          mov     edi,0x1
736:  03 18                     add     ebx,DWORD PTR [rax]
738:  31 c0                     xor     eax,eax
73a:  89 da                     mov     edx,ebx
73c:  e8 df fe ff ff          call    620 <__printf_chk@plt>
741:  89 d8                     mov     eax,ebx
743:  5b                     pop     rbx
744:  c3                     ret
```

`0x2008b9` это разница между адресом инструкции по `0x720` и `global_variable`, а `0x20` это разница между инструкцией по `0x72A` и строкой «`returning %d\n`».

Как видно, необходимость очень часто пересчитывать адреса делает исполнение немного медленнее (хотя это и стало лучше в x64). Так что если вы заботитесь о скорости исполнения, то, наверное, нужно задуматься о статической компоновке (static linking) (см. [Agner Fog, *Optimizing software in C++* (2015)]).

68.1.1. Windows

Такой механизм не используется в Windows DLL. Если загрузчику в Windows приходится загружать DLL в другое место, он «патчит» DLL прямо в памяти (на местах `FIXUP`-ов) чтобы скорректировать все адреса. Это приводит к тому что загруженную один раз DLL нельзя использовать одновременно в разных процессах, желающих расположить её по разным адресам — потому что каждый загруженный в память экземпляр DLL *доводится* до того чтобы работать только по этим адресам.

68.2. Трюк с LD_PRELOAD в Linux

Это позволяет загружать свои динамические библиотеки перед другими, даже перед системными, такими как `libc.so.6`.

Что в свою очередь, позволяет «подставлять» написанные нами функции перед оригинальными из системных библиотек. Например, легко перехватывать все вызовы к `time()`, `read()`, `write()`, и т.д.

Попробуем узнать, сможем ли мы обмануть утилиту `uptime`. Как известно, она сообщает, как долго компьютер работает. При помощи `strace`([72](#) (стр. [730](#))), можно увидеть, что эту информацию утилита получает из файла `/proc/uptime`:

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY)      = 3
lseek(3, 0, SEEK_SET)               = 0
read(3, "416166.86 414629.38\n", 2047) = 20
...
```

Это не реальный файл на диске, это виртуальный файл, содержимое которого генерируется на лету в ядре Linux. Там просто два числа:

²указатель инструкций в AMD64


```
$ cat /proc/uptime
416690.91 415152.03
```

Из Wikipedia, можно узнать ³:

The first number is the total number of seconds the system has been up. The second number is how much of that time the machine has spent idle, in seconds.

Попробуем написать свою динамическую библиотеку, в которой будет `open()`, `read()`, `close()` с нужной нам функциональностью.

Во-первых, наш `open()` будет сравнивать имя открываемого файла с тем что нам нужно, и если да, то будет запоминать дескриптор открытого файла. Во-вторых, `read()`, если будет вызываться для этого дескриптора, будет подменять вывод, а в остальных случаях, будет вызывать настоящий `read()` из `libc.so.6`. А также `close()`, будет следить, закрывается ли файл за которым мы следим.

Для того чтобы найти адреса настоящих функций в `libc.so.6`, используем `dlopen()` и `dlsym()`.

Нам это нужно, потому что нам нужно передавать управление «настоящим» функциями.

С другой стороны, если бы мы перехватывали, скажем, `strcmp()`, и следили бы за всеми сравнениями строк в программе, то, наверное, `strcmp()` можно было бы и самому реализовать, не пользуясь настоящей функцией ⁴.

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool initied = false;

_Noreturn void die (const char * fmt, ...)
{
    va_list va;
    va_start (va, fmt);

    vprintf (fmt, va);
    exit(0);
};

static void find_original_functions ()
{
    if (initied)
        return;

    libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
    if (libc_handle==NULL)
        die ("can't open libc.so.6\n");

    open_ptr = dlsym (libc_handle, "open");
    if (open_ptr==NULL)
        die ("can't find open()\n");

    close_ptr = dlsym (libc_handle, "close");
    if (close_ptr==NULL)
        die ("can't find close()\n");

    read_ptr = dlsym (libc_handle, "read");
    if (read_ptr==NULL)
```

³[wikipedia](#)

⁴Например, посмотрите как обеспечивается простейший перехват `strcmp()` в статье ⁵ написанной Yong Huang

```

        die ("can't find read()\n");

    initied = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
    find_original_functions();

    int fd=(*open_ptr)(pathname, flags);
    if (strcmp(pathname, "/proc/uptime")==0)
        opened_fd=fd; // that's our file! record its file descriptor
    else
        opened_fd=0;
    return fd;
};

int close(int fd)
{
    find_original_functions();

    if (fd==opened_fd)
        opened_fd=0; // the file is not opened anymore
    return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
    find_original_functions();

    if (opened_fd!=0 && fd==opened_fd)
    {
        // that's our file!
        return snprintf (buf, count, "%d %d", 0x7fffffff, 0x7fffffff)+1;
    };
    // not our file, go to real read() function
    return (*read_ptr)(fd, buf, count);
};

```

([Исходный код на GitHub](#))

Компилируем как динамическую библиотеку:

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Запускаем *uptime*, подгружая нашу библиотеку перед остальными:

```
LD_PRELOAD=`pwd`/fool_uptime.so uptime
```

Видим такое:

```
01:23:02 up 24855 days,  3:14,  3 users,  load average: 0.00, 0.01, 0.05
```

Если переменная окружения *LD_PRELOAD* будет всегда указывать на путь и имя файла нашей библиотеки, то она будет загружаться для всех запускаемых программ.

Еще примеры:

- Перехват *time()* в Sun Solaris yurichev.com
- Очень простой перехват *strcmp()* (Yong Huang) <http://go.yurichev.com/17143>
- Kevin Pulo — Fun with LD_PRELOAD. Много примеров и идей. yurichev.com
- Перехват функций работы с файлами для компрессии и декомпрессии файлов на лету (zlibc). <http://go.yurichev.com/17146>

Глава 69

Windows NT

69.1. CRT (win32)

Начинается ли исполнение программы прямо с функции `main()` ? Нет, не начинается. Если открыть любой исполняемый файл в IDA или Hiew, то OEP указывает на какой-то совсем другой код. Это код, который делает некоторые приготовления перед тем как запустить ваш код . Он называется стартап-код или CRT-код (C RunTime).

Функция `main()` принимает на вход массив из параметров, переданных в командной строке, а также переменные окружения. Но в реальности в программу передается командная строка в виде простой строки, это именно CRT-код находит там пробелы и разрезает строку на части. CRT-код также готовит массив переменных окружения `envp` . В GUI¹-приложениях win32, вместо `main()` имеется функция `WinMain` со своими аргументами :

```
int CALLBACK WinMain(
    _In_ HINSTANCE hInstance,
    _In_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine,
    _In_ int nCmdShow
);
```

CRT-код готовит и их.

А также, число, возвращаемое функцией `main()` , это код ошибки возвращаемый программой . В CRT это значение передается в `ExitProcess()` , принимающей в качестве аргумента код ошибки .

Как правило, каждый компилятор имеет свой CRT-код.

Вот типичный для MSVC 2008 CRT-код.

```
1  __tmainCRTStartup proc near
2
3  var_24 = dword ptr -24h
4  var_20 = dword ptr -20h
5  var_1C = dword ptr -1Ch
6  ms_exc = CPPEH_RECORD ptr -18h
7
8      push    14h
9      push    offset stru_4092D0
10     call    __SEH_prolog4
11     mov     eax, 5A4Dh
12     cmp     ds:400000h, ax
13     jnz     short loc_401096
14     mov     eax, ds:40003Ch
15     cmp     dword ptr [eax+400000h], 4550h
16     jnz     short loc_401096
17     mov     ecx, 10Bh
18     cmp     [eax+400018h], cx
19     jnz     short loc_401096
20     cmp     dword ptr [eax+400074h], 0Eh
21     jbe     short loc_401096
```

¹Graphical user interface

69.1. CRT (WIN32)

```

22     xor     ecx, ecx
23     cmp     [eax+4000E8h], ecx
24     setnz   cl
25     mov     [ebp+var_1C], ecx
26     jmp     short loc_40109A
27
28
29 loc_401096: ; CODE XREF: __tmainCRTStartup+18
30             ; __tmainCRTStartup+29 ...
31     and     [ebp+var_1C], 0
32
33 loc_40109A: ; CODE XREF: __tmainCRTStartup+50
34     push    1
35     call    __heap_init
36     pop     ecx
37     test    eax, eax
38     jnz     short loc_4010AE
39     push    1Ch
40     call    _fast_error_exit
41     pop     ecx
42
43 loc_4010AE: ; CODE XREF: __tmainCRTStartup+60
44     call    __mtinit
45     test    eax, eax
46     jnz     short loc_4010BF
47     push    10h
48     call    _fast_error_exit
49     pop     ecx
50
51 loc_4010BF: ; CODE XREF: __tmainCRTStartup+71
52     call    sub_401F2B
53     and     [ebp+ms_exc.disabled], 0
54     call    __ioinit
55     test    eax, eax
56     jge     short loc_4010D9
57     push    1Bh
58     call    __amsg_exit
59     pop     ecx
60
61 loc_4010D9: ; CODE XREF: __tmainCRTStartup+8B
62     call    ds:GetCommandLineA
63     mov     dword_40B7F8, eax
64     call    __crtGetEnvironmentStringsA
65     mov     dword_40AC60, eax
66     call    __setargv
67     test    eax, eax
68     jge     short loc_4010FF
69     push    8
70     call    __amsg_exit
71     pop     ecx
72
73 loc_4010FF: ; CODE XREF: __tmainCRTStartup+B1
74     call    __setenvp
75     test    eax, eax
76     jge     short loc_401110
77     push    9
78     call    __amsg_exit
79     pop     ecx
80
81 loc_401110: ; CODE XREF: __tmainCRTStartup+C2
82     push    1
83     call    __cinit
84     pop     ecx
85     test    eax, eax
86     jz      short loc_401123
87     push    eax
88     call    __amsg_exit
89     pop     ecx
90
91 loc_401123: ; CODE XREF: __tmainCRTStartup+D6

```

69.1. CRT (WIN32)

```
92     mov     eax, envp
93     mov     dword_40AC80, eax
94     push    eax           ; envp
95     push    argv          ; argv
96     push    argc          ; argc
97     call    _main
98     add     esp, 0Ch
99     mov     [ebp+var_20], eax
100    cmp     [ebp+var_1C], 0
101    jnz     short $LN28
102    push    eax           ; uExitCode
103    call    $LN32
104
105 $LN28:      ; CODE XREF: ___tmainCRTStartup+105
106     call    __cexit
107     jmp     short loc_401186
108
109
110 $LN27:      ; DATA XREF: .rdata:stru_4092D0
111     mov     eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for function 401044
112     mov     ecx, [eax]
113     mov     ecx, [ecx]
114     mov     [ebp+var_24], ecx
115     push    eax
116     push    ecx
117     call    __XcptFilter
118     pop     ecx
119     pop     ecx
120
121 $LN24:
122     retn
123
124
125 $LN14:      ; DATA XREF: .rdata:stru_4092D0
126     mov     esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for function 401044
127     mov     eax, [ebp+var_24]
128     mov     [ebp+var_20], eax
129     cmp     [ebp+var_1C], 0
130     jnz     short $LN29
131     push    eax           ; int
132     call    __exit
133
134
135 $LN29:      ; CODE XREF: ___tmainCRTStartup+135
136     call    __c_exit
137
138 loc_401186: ; CODE XREF: ___tmainCRTStartup+112
139     mov     [ebp+ms_exc.disabled], 0FFFFFFFh
140     mov     eax, [ebp+var_20]
141     call    __SEH_epilog4
142     retn
```

Здесь можно увидеть по крайней мере вызов функции `GetCommandLineA()` (строка 62), затем `setargv()` (строка 66) и `setenvp()` (строка 74), которые, видимо, заполняют глобальные переменные-указатели `argc`, `argv`, `envp`.

В итоге, вызывается `main()` с этими аргументами (строка 97).

Также имеются вызовы функций с говорящими именами вроде `heap_init()` (строка 35), `ioinit()` (строка 54).

Куча действительно инициализируется в CRT. Если вы попытаетесь использовать `malloc()` в программе без CRT, программа упадет с такой ошибкой:

```
runtime error R6030
- CRT not initialized
```

Инициализация глобальных объектов в Си++ происходит до вызова `main()`, именно в CRT : 52.4.1 (стр. 560).

Значение, возвращаемое из `main()` передается или в `cexit()`, или же в `$LN32`, которая далее вызывает `doexit()`.

69.2. WIN32 PE

Можно ли обойтись без CRT? Можно, если вы знаете что делаете.

В линкере от MSVC точка входа задается опцией `/ENTRY`.

```
#include <windows.h>

int main()
{
    MessageBox (NULL, "hello, world", "caption", MB_OK);
};
```

Компилируем в MSVC 2008.

```
cl no_crt.c user32.lib /link /entry:main
```

Получаем вполне работающий .exe размером 2560 байт, внутри которого есть только PE-заголовок, инструкции, вызывающие `MessageBox`, две строки в сегменте данных, импортируемая из `user32.dll` функция `MessageBox`, и более ничего.

Это работает, но вы уже не сможете вместо `main()` написать `WinMain` с его четырьмя аргументами. Вернее, если быть точным, написать-то сможете, но доступа к этим аргументам не будет, потому что они не подготовлены на момент исполнения.

Кстати, можно еще короче сделать .exe если уменьшить выравнивание PE²-секций (которое, по умолчанию, 4096 байт).

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

Линкер скажет:

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```

Получим .exe размером 720 байт. Он запускается в Windows 7 x86, но не x64 (там выдает ошибку при загрузке). При желании, размер можно еще сильнее ужать, но, как видно, возникают проблемы с совместимостью с разными версиями Windows.

69.2. Win32 PE

PE это формат исполняемых файлов, принятый в Windows.

Разница между .exe, .dll, и .sys в том, что у .exe и .sys обычно нет экспортов, только импорты.

У DLL³, как и у всех PE-файлов, есть точка входа (OEP) (там располагается функция `DllMain()`), но обычно эта функция ничего не делает.

.sys это обычно драйвера устройств.

Для драйверов, Windows требует, чтобы контрольная сумма в PE-файле была проставлена и была верной⁴.

А начиная с Windows Vista, файлы драйверов должны быть также подписаны при помощи электронной подписи, иначе они не будут загружаться.

В начале всякого PE-файла есть крохотная DOS-программа, выводящая на консоль сообщение вроде «This program cannot be run in DOS mode.» — если запустить эту программу в DOS либо Windows 3.1 (ОС не знающие о PE-формате), выведется это сообщение.

69.2.1. Терминология

- Модуль — это отдельный файл, .exe или .dll.
- Процесс — это некая загруженная в память и работающая программа. Как правило состоит из одного .exe-файла и массы .dll-файлов.
- Память процесса — память с которой работает процесс. У каждого процесса — своя. Там обычно имеются загруженные модули, память стека, кучи, итд.

²Portable Executable

³Dynamic-link library

⁴Например, Hiew(74 (стр. 732)) умеет её подсчитывать

- **VA**⁵ — это адрес, который будет использоваться в самой программе во время исполнения.
- Базовый адрес (модуля) — это адрес, по которому модуль должен быть загружен в пространство процесса. Загрузчик **OC** может его изменить, если этот базовый адрес уже занят другим модулем, загруженным перед ним.
- **RVA**⁶ — это **VA**-адрес минус базовый адрес. Многие адреса в таблицах PE-файла используют **RVA**-адреса.
- **IAT**⁷ — массив адресов импортированных символов⁸. Иногда, директория `IMAGE_DIRECTORY_ENTRY_IAT` указывает на **IAT**. Важно отметить, что **IDA** (по крайней мере 6.1) может выделить псевдо-секцию с именем `.idata` для **IAT**, даже если **IAT** является частью совсем другой секции!
- **INT**⁹ — массив имен символов для импортирования¹⁰.

69.2.2. Базовый адрес

Дело в том, что несколько авторов модулей могут готовить DLL-файлы для других, и нет возможности договориться о том, какие адреса и кому будут отведены.

Поэтому, если у двух необходимых для загрузки процесса DLL одинаковые базовые адреса, одна из них будет загружена по этому базовому адресу, а вторая — по другому свободному месту в памяти процесса, и все виртуальные адреса во второй DLL будут скорректированы.

Очень часто линкер в **MSVC** генерирует .exe-файлы с базовым адресом `0x400000`¹¹, и с секцией кода начинающейся с `0x401000`. Это значит, что **RVA** начала секции кода — `0x1000`. А **DLL** часто генерируются **MSVC**-линкером с базовым адресом `0x10000000`¹².

Помимо всего прочего, есть еще одна причина намеренно загружать модули по разным адресам, а точнее, по случайным. Это **ASLR**¹³¹⁴.

Дело в том, что некий шеллкод, пытающийся исполниться на зараженной системе, должен вызывать какие-то системные функции, а следовательно, знать их адреса.

И в старых **OC** (в линейке **Windows NT**: до **Windows Vista**), системные DLL (такие как `kernel32.dll`, `user32.dll`) загружались все время по одним и тем же адресам, а если еще и вспомнить, что версии этих DLL редко менялись, то адреса отдельных функций, можно сказать, фиксированы и шеллкод может вызывать их напрямую.

Чтобы избежать этого, методика **ASLR** загружает и вашу программу, и все модули ей необходимые, по случайным адресам, разным при каждом запуске.

В PE-файлах, поддержка **ASLR** отмечается выставлением флага `IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` (см: [Mark Russinovich, *Microsoft Windows Internals*]).

69.2.3. Subsystem

Имеется также поле *subsystem*, обычно это:

- **native**¹⁵ (.sys-драйвер),
- **console** (консольное приложение) или
- **GUI** (не консольное).

⁵Virtual Address

⁶Relative Virtual Address

⁷Import Address Table

⁸Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]

⁹Import Name Table

¹⁰Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]

¹¹Причина выбора такого адреса описана здесь: [MSDN](#)

¹²Это можно изменять опцией `/BASE` в линкере

¹³Address Space Layout Randomization

¹⁴[wikipedia](#)

¹⁵Что означает, что модуль использует Native API а не Win32

69.2.4. Версия ОС

PE-файле также задает минимальный номер версии Windows, необходимый для загрузки модуля.

Соответствие номеров версий в файле и кодовых наименований Windows, можно посмотреть здесь¹⁶.

Например, [MSVC 2005](#) еще компилирует .exe-файлы запускающиеся на Windows NT4 (версия 4.00), а вот [MSVC 2008](#) уже нет (генерируемые файлы имеют версию 5.00, для запуска необходима как минимум Windows 2000).

[MSVC 2012](#) по умолчанию генерирует .exe-файлы версии 6.00, для запуска нужна как минимум Windows Vista. Хотя, изменив настройки компиляции¹⁷, можно заставить генерировать и под Windows XP.

69.2.5. Секции

Разделение на секции присутствует, по-видимому, во всех форматах исполняемых файлов.

Придумано это для того, чтобы отделить код от данных, а данные — от константных данных.

- На секции кода будет стоять флаг `IMAGE_SCN_CNT_CODE` или `IMAGE_SCN_MEM_EXECUTE` — это исполняемый код.
- На секции данных — флаги `IMAGE_SCN_CNT_INITIALIZED_DATA`, `IMAGE_SCN_MEM_READ` и `IMAGE_SCN_MEM_WRITE`.
- На пустой секции с неинициализированными данными — `IMAGE_SCN_CNT_UNINITIALIZED_DATA`, `IMAGE_SCN_MEM_READ` и `IMAGE_SCN_MEM_WRITE`.
- А на секции с константными данными, то есть, защищенными от записи, могут быть флаги `IMAGE_SCN_CNT_INITIALIZED_DATA` и `IMAGE_SCN_MEM_READ` без `IMAGE_SCN_MEM_WRITE`. Если попытаться записать что-то в эту секцию, процесс упадет.

В PE-файле можно задавать название для секции, но это не важно. Часто (но не всегда) секция кода называется `.text`, секция данных — `.data`, константных данных — `.rdata` (*readable data*). Еще популярные имена секций:

- `.idata` — секция импортов. [IDA](#) может создавать псевдо-секцию с этим же именем: [69.2.1](#) (стр. 697).
- `.edata` — секция экспортов (редко встречается)
- `.pdata` — секция содержащая информацию об исключениях в Windows NT для MIPS, [IA64](#) и [x64](#): [69.3.3](#) (стр. 721)
- `.reloc` — секция релоков
- `.bss` — неинициализированные данные
- `.tls` — thread local storage ([TLS](#))
- `.rsrc` — ресурсы
- `.CRT` — может присутствует в бинарных файлах, скомпилированных очень старыми версиями [MSVC](#)

Запаковщики/зашифровщики PE-файлов часто затирают имена секций, или меняют на свои.

В [MSVC](#) можно объявлять данные в произвольно названной секции¹⁸.

Некоторые компиляторы и линкеры могут добавлять также секцию с отладочными символами и вообще отладочной информацией (например, MinGW).

Хотя это не так в современных версиях [MSVC](#) (там принято отладочную информацию сохранять в отдельных [PDB](#)-файлах).

Вот как PE-секция описывается в файле:

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
```

¹⁶[wikipedia](#)

¹⁷[MSDN](#)

¹⁸[MSDN](#)


```

DWORD PointerToLinenumbers;
WORD   NumberOfRelocations;
WORD   NumberOfLinenumbers;
DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;

```

19

Еще немного терминологии: *PointerToRawData* называется «Offset» в Hiew и *VirtualAddress* называется «RVA» там же.

69.2.6. Релоки

Также известны как FIXUP-ы (по крайней мере в Hiew).

Это также присутствует почти во всех форматах загружаемых и исполняемых файлов ²⁰.

Исключения это динамические библиотеки явно скомпилированные с PIC или любой другой PIC-код.

Зачем они нужны? Как видно, модули могут загружаться по другим базовым адресам, но как же тогда работать с глобальными переменными, например?

Ведь нужно обращаться к ним по адресу. Одно из решений — это адресно-независимый код (68.1 (стр. 688)). Но это далеко не всегда удобно.

Поэтому имеется таблица релоков. Там просто перечислены адреса мест в модуле подлежащими коррекции при загрузке по другому базовому адресу.

Например, по 0x410000 лежит некая глобальная переменная, и вот как обеспечивается её чтение:

```

A1 00 00 41 00      mov      eax, [000410000]

```

Базовый адрес модуля 0x400000, а RVA глобальной переменной 0x10000.

Если загружать модуль по базовому адресу 0x500000, нужно чтобы адрес этой переменной в этой инструкции стал 0x510000.

Как видно, адрес переменной закодирован в самой инструкции MOV, после байта 0xA1.

Поэтому адрес четырех байт, после 0xA1, записывается в таблицу релоков.

Если модуль загружается по другому базовому адресу, загрузчик ОС обходит все адреса в таблице, находит каждое 32-битное слово по этому адресу, отнимает от него настоящий, оригинальный базовый адрес (в итоге получается RVA), и прибавляет к нему новый базовый адрес.

А если модуль загружается по своему оригинальному базовому адресу, ничего не происходит.

Так можно обходиться со всеми глобальными переменными.

Релоки могут быть разных типов, однако в Windows для x86-процессоров, тип обычно IMAGE_REL_BASED_HIGHLOW.

Кстати, релоки маркируются темным в Hiew, например: илл.8.12.

OllyDbg подчеркивает места в памяти, к которым будут применены релоки, например: илл.14.11.

69.2.7. Экспорты и импорты

Как известно, любая исполняемая программа должна как-то пользоваться сервисами ОС и прочими DLL-библиотеками.

Можно сказать, что нужно связывать функции из одного модуля (обычно DLL) и места их вызовов в другом модуле (.exe-файл или другая DLL).

Для этого, у каждой DLL есть «экспорты», это таблица функций плюс их адреса в модуле.

А у .exe-файла, либо DLL, есть «импорты», это таблица функций требующихся для исполнения включая список имен DLL-файлов.

Загрузчик ОС, после загрузки основного .exe-файла, проходит по таблице импортов: загружает дополнительные DLL-файлы, находит имена функций среди экспортов в DLL и прописывает их адреса в IAT в головном .exe-модуле.

¹⁹MSDN

²⁰Даже .exe-файлы в MS-DOS

Как видно, во время загрузки, загрузчику нужно много сравнивать одни имена функций с другими, а сравнение строк — это не очень быстрая процедура, так что, имеется также поддержка «ординалов» или «hint»-ов, это когда в таблице импортов проставлены номера функций вместо их имен.

Так их быстрее находить в загружаемой DLL. В таблице экспортов ординалы присутствуют всегда.

К примеру, программы использующие библиотеки [MFC²¹](#), обычно загружают mfc*.dll по ординалам, и в таких программах, в [INT](#), нет имен функций [MFC](#).

При загрузке такой программы в [IDA](#), она спросит у вас путь к файлу mfc*.dll, чтобы установить имена функций. Если в [IDA](#) не указать путь к этой DLL, то вместо имен функций будет что-то вроде *mfc80_123*.

Секция импортов

Под таблицу импортов и всё что с ней связано иногда отводится отдельная секция (с названием вроде `.idata`), но это не обязательно.

Импорты — это запутанная тема еще и из-за терминологической путаницы. Попробуем собрать всё в одно место.

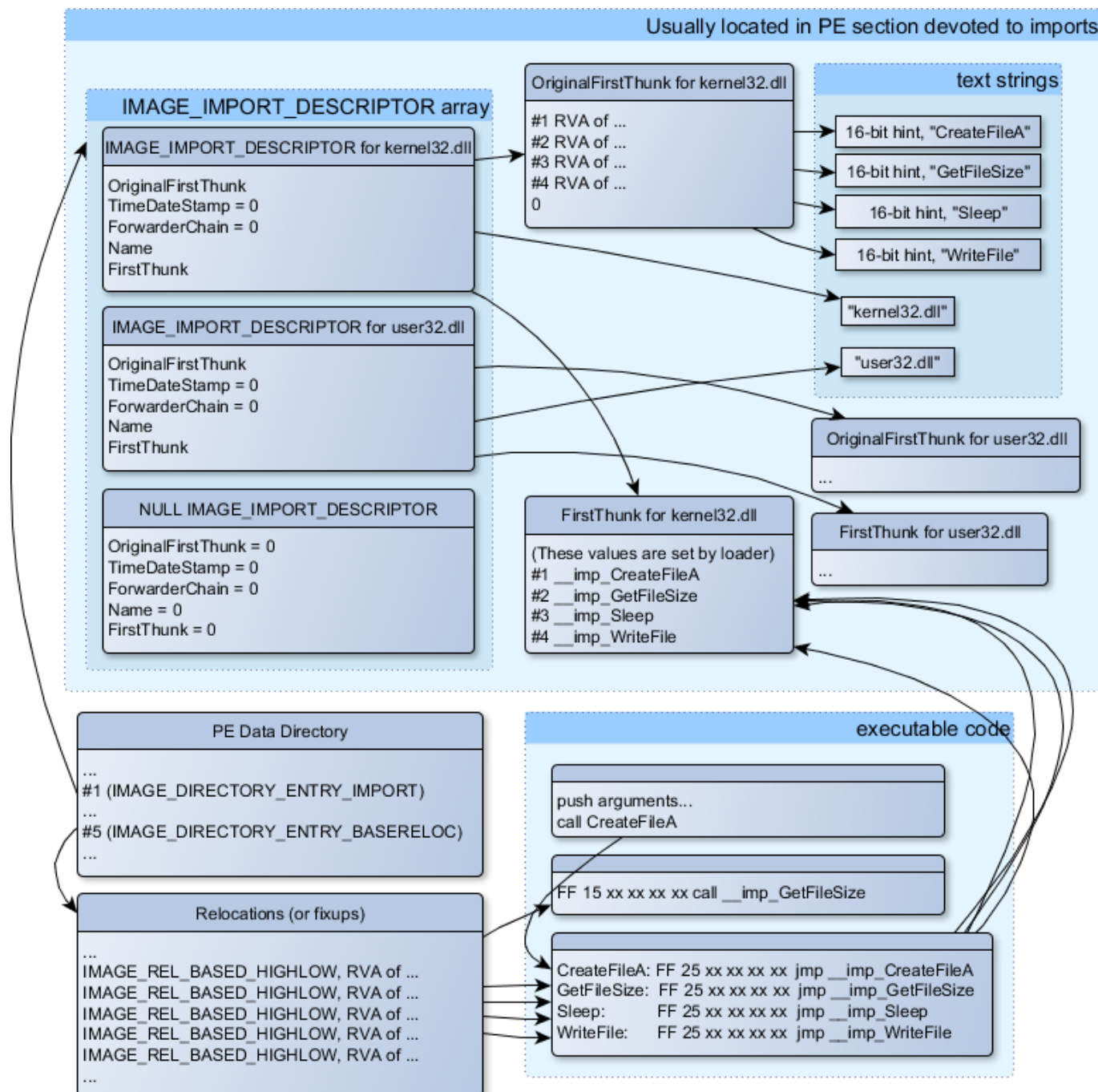


Рис. 69.1: схема, объединяющая все структуры в PE-файлы, связанные с импортами

²¹Microsoft Foundation Classes

Самая главная структура — это массив *IMAGE_IMPORT_DESCRIPTOR*. Каждый элемент на каждую импортируемую DLL.

У каждого элемента есть *RVA*-адрес текстовой строки (имя DLL) (*Name*).

OriginalFirstThunk это *RVA*-адрес таблицы *INT*. Это массив *RVA*-адресов, каждый из которых указывает на текстовую строку где записано имя функции. Каждую строку предваряет 16-битное число («hint») — «ординал» функции.

Если при загрузке удастся найти функцию по ординалу, тогда сравнение текстовых строк не будет происходить. Массив оканчивается нулем.

Есть также указатель на таблицу *IAT* с названием *FirstThunk*, это просто *RVA*-адрес места, где загрузчик будет проставлять адреса найденных функций.

Места где загрузчик проставляет адреса, *IDA* именует их так: *__imp_CreateFileA*, etc.

Есть по крайней мере два способа использовать адреса, предоставленные загрузчиком.

- В коде будут просто инструкции вроде *call __imp_CreateFileA*, а так как, поле с адресом импортируемой функции это как бы глобальная переменная, то в таблице релоков добавляется адрес (плюс 1 или 2) в инструкции *call*, на случай если модуль будет загружен по другому базовому адресу.

Но как видно, это приводит к увеличению таблицы релоков. Ведь вызовов импортируемой функции у вас в модуле может быть очень много. К тому же, чем больше таблица релоков, тем дольше загрузка.

- На каждую импортируемую функцию выделяется только один переход на импортируемую функцию используя инструкцию *JMP* плюс релок на эту инструкцию. Такие места-«переходники» называются также «thunk»-ами. А все вызовы импортируемой функции это просто инструкция *CALL* на соответствующий «thunk». В данном случае, дополнительные релоки не нужны, потому что эти *CALL*-ы имеют относительный адрес, и корректировать их не надо.

Оба этих два метода могут комбинироваться. Вероятно, линкер создает отдельный «thunk», если вызовов слишком много, но по умолчанию — не создает.

Кстати, массив адресов функций, на который указывает *FirstThunk*, не обязательно может быть в секции *IAT*. К примеру, автор сих строк написал утилиту *PE_add_import*²² для добавления импорта в уже существующий .exe-файл. Раньше, в прошлых версиях утилиты, на месте функции, вместо которой вы хотите подставить вызов в другую DLL, моя утилита вписывала такой код:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

При этом, *FirstThunk* указывает прямо на первую инструкцию. Иными словами, загрузчик, загружая *yourdll.dll*, прописывает адрес функции *function* прямо в коде.

Надо также отметить что обычно секция кода защищена от записи, так что, моя утилита добавляет флаг *IMAGE_SCN_MEM_WRITE* для секции кода. Иначе при загрузке такой программы, она упадет с ошибкой 5 (access denied).

Может возникнуть вопрос: а что если я поставлю программу с набором DLL, которые никогда не будут меняться (в т.ч., адреса всех функций в этих DLL), может как-то можно ускорить процесс загрузки?

Да, можно прописать адреса импортируемых функций в массивы *FirstThunk* заранее. Для этого в структуре *IMAGE_IMPORT_DESCRIPTOR* имеется поле *Timestamp*. И если там присутствует какое-то значение, то загрузчик сверяет это значение с датой-временем DLL-файла. И если они равны, то загрузчик больше ничего не делает, и загрузка может происходить быстрее.

Это называется «old-style binding»²³. В Windows SDK для этого имеется утилита *BIND.EXE*. Для ускорения загрузки вашей программы, Matt Pietrek в *Matt Pietrek, An In-Depth Look into the Win32 Portable Executable File Format, (2002)*²⁴, предлагает делать binding сразу после инсталляции вашей программы на компьютере конечного пользователя.

Запаковщики/зашифровщики PE-файлов могут также сжимать/шифровать таблицу импортов. В этом случае, загрузчик Windows, конечно же, не загрузит все нужные DLL. Поэтому распаковщик/расшифровщик делает это сам, при помощи вызовов *LoadLibrary()* и *GetProcAddress()*. Вот почему в запакованных файлах эти две функции часто присутствуют в *IAT*.

В стандартных DLL входящих в состав Windows, часто, *IAT* находится в самом начале PE-файла.

Возможно это для оптимизации. Ведь .exe-файл при загрузке не загружается в память весь (вспомните что инсталляторы огромного размера подозрительно быстро запускаются), он «мапится» (map), и подгружается в память частями по мере обращения к этой памяти.

²²yurichev.com

²³MSDN. Существует также «new-style binding».

²⁴Также доступно здесь: <http://go.yurichev.com/17318>

И возможно в Microsoft решили что так будет быстрее.

69.2.8. Ресурсы

Ресурсы в PE-файле — это набор иконок, картинок, текстовых строк, описаний диалогов. Возможно, их в свое время решили отделить от основного кода, чтобы все эти вещи были многоязычными, и было проще выбирать текст или картинку того языка, который установлен в ОС.

В качестве побочного эффекта, их легко редактировать и сохранять обратно в исполняемый файл, даже не обладая специальными знаниями, например, редактором ResHack (69.2.11 (стр. 702)).

69.2.9. .NET

Программы на .NET компилируются не в машинный код, а в свой собственный байткод. Собственно, в .exe-файлы байткод вместо обычного кода, однако, точка входа (OEP) указывает на крохотный фрагмент x86-кода:

```
jmp     mscoree.dll!_CorExeMain
```

А в mscoree.dll и находится .NET-загрузчик, который уже сам будет работать с PE-файлом.

Так было в ОС до Windows XP. Начиная с XP, загрузчик ОС уже сам определяет, что это .NET-файл и запускает его не исполняя этой инструкции JMP

²⁵.

69.2.10. TLS

Эта секция содержит в себе инициализированные данные для TLS(66 (стр. 680)) (если нужно). При старте нового треда, его TLS-данные инициализируются данными из этой секции.

Помимо всего прочего, спецификация PE-файла предусматривает инициализацию TLS-секции, т.н., TLS callbacks. Если они присутствуют, то они будут вызваны перед тем как передать управление на главную точку входа (OEP). Это широко используется запаковщиками/зашифровщиками PE-файлов.

69.2.11. Инструменты

- objdump (имеется в сугwin) для вывода всех структур PE-файла.
- Hiew(74 (стр. 732)) как редактор.
- refile — Python-библиотека для работы с PE-файлами ²⁶.
- ResHack AKA Resource Hacker — редактор ресурсов ²⁷.
- PE_add_import²⁸ — простая утилита для добавления символа/-ов в таблицу импортов PE-файла.
- PE_patcher²⁹ — простая утилита для модификации PE-файлов.
- PE_search_str_refs³⁰ — простая утилита для поиска функции в PE-файле, где используется некая текстовая строка.

69.2.12. Further reading

- Daniel Pistelli — The .NET File Format ³¹

²⁵MSDN

²⁶<http://go.yurichev.com/17052>

²⁷<http://go.yurichev.com/17052>

²⁸<http://go.yurichev.com/17049>

²⁹yurichev.com

³⁰yurichev.com

³¹<http://go.yurichev.com/17056>

69.3. Windows SEH

69.3.1. Забудем на время о MSVC

SEH в Windows предназначен для обработки исключений, тем не менее, с Си++ и ООП он никак не связан. Здесь мы рассмотрим SEH изолированно от Си++ и расширений MSVC.

Каждый процесс имеет цепочку SEH-обработчиков, и адрес последнего записан в TIB. Когда происходит исключение (деление на ноль, обращение по неверному адресу в памяти, пользовательское исключение, поднятое при помощи `RaiseException()`), ОС находит последний обработчик в TIB и вызывает его, передав ему информацию о состоянии CPU в момент исключения (все значения регистров, итд.). Обработчик выясняет, то ли это исключение, для которого он создавался?

Если да, то он обрабатывает исключение. Если нет, то показывает ОС что он не может его обработать и ОС вызывает следующий обработчик в цепочке, и так до тех пор, пока не найдется обработчик способный обработать исключение.

В самом конце цепочки находится стандартный обработчик, показывающий всем очень известное окно, сообщающее что процесс упал, сообщает также состояние CPU в момент падения и позволяет собрать и отправить информацию обработчикам в Microsoft.

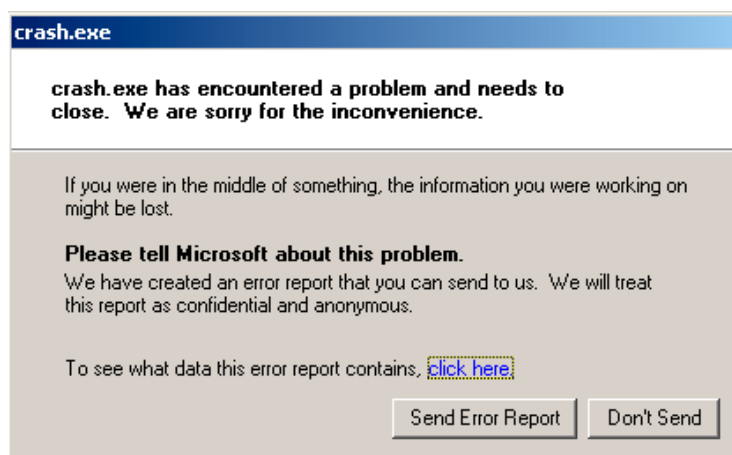


Рис. 69.2: Windows XP

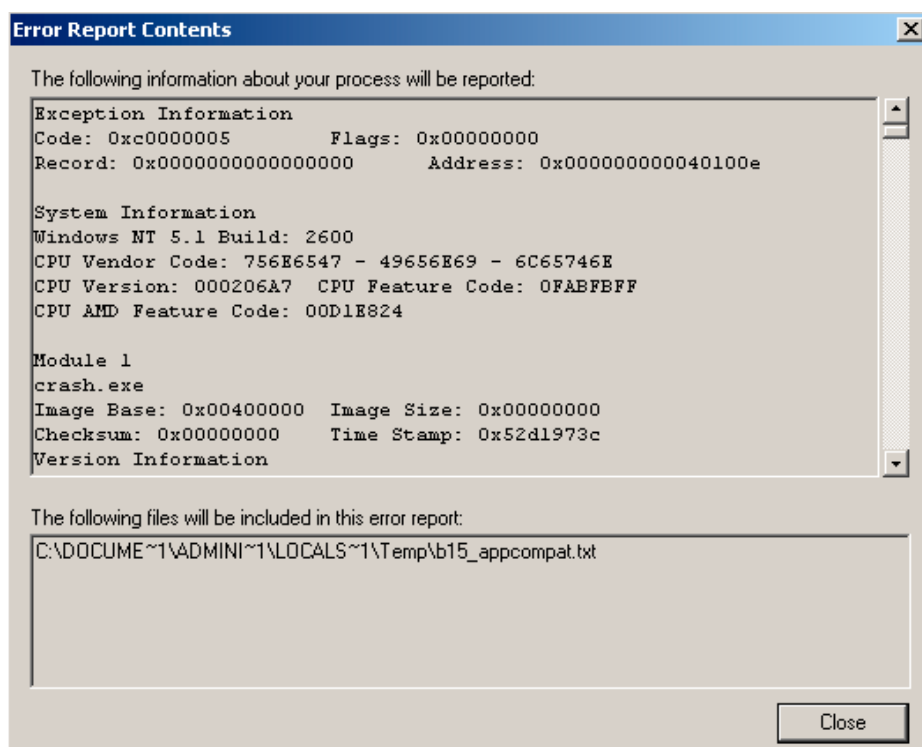


Рис. 69.3: Windows XP

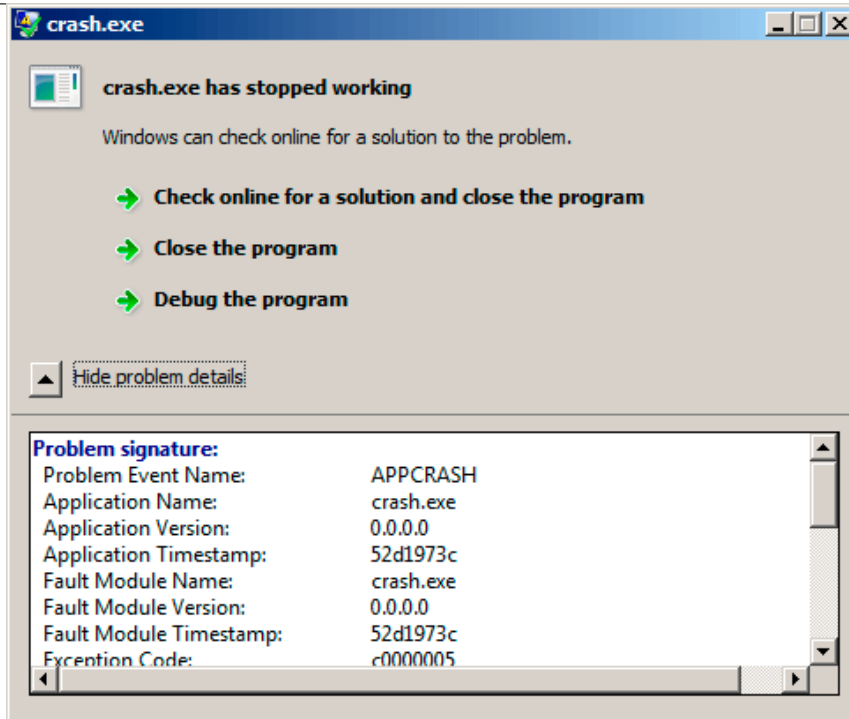


Рис. 69.4: Windows 7

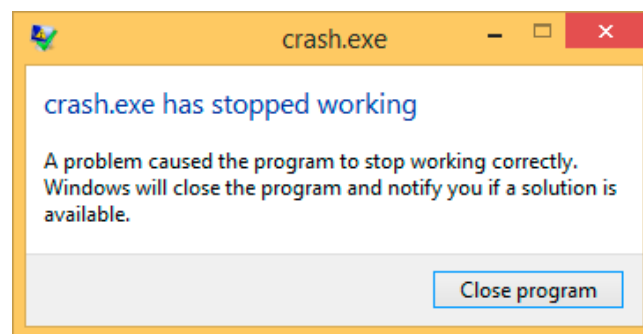


Рис. 69.5: Windows 8.1

Раньше этот обработчик назывался Dr. Watson ³².

Кстати, некоторые разработчики делают свой собственный обработчик, отправляющий информацию о падении программы им самим. Он регистрируется при помощи функции `SetUnhandledExceptionFilter()` и будет вызван если ОС не знает как иначе обработать исключение.

А, например, Oracle RDBMS в этом случае генерирует огромные дампы, содержащие всю возможную информацию и состоянии CPU и памяти.

Попробуем написать свой примитивный обработчик исключений. Этот пример основан на примере из [Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)]³³. Он должен компилироваться с опцией SAFESSEH: `cl seh1.cpp /link /safeseh:no`. Подробнее об опции SAFESSEH здесь: [MSDN](#).

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
```

³²[wikipedia](#)

³³Также доступно здесь: <http://go.yurichev.com/17293>

```

unsigned i;

printf ("%s\n", __FUNCTION__);
printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->ExceptionCode);
printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->ExceptionFlags);
printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord->ExceptionAddress);

if (ExceptionRecord->ExceptionCode==0xE1223344)
{
    printf ("That's for us\n");
    // yes, we "handled" the exception
    return ExceptionContinueExecution;
}
else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION)
{
    printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);
    // will it be possible to 'fix' it?
    printf ("Trying to fix wrong pointer address\n");
    ContextRecord->Eax=(DWORD)&new_value;
    // yes, we "handled" the exception
    return ExceptionContinueExecution;
}
else
{
    printf ("We do not handle this\n");
    // someone else's problem
    return ExceptionContinueSearch;
};
}

int main()
{
    DWORD handler = (DWORD)except_handler; // take a pointer to our handler

    // install exception handler
    __asm
    {
        push    handler           // make EXCEPTION_REGISTRATION record:
        push    FS:[0]            // address of handler function
        mov     FS:[0],ESP        // address of previous handler
        // add new EXCEPTION_REGISTRATION
    }

    RaiseException (0xE1223344, 0, 0, NULL);

    // now do something very bad
    int* ptr=NULL;
    int val=0;
    val=*ptr;
    printf ("val=%d\n", val);

    // deinstall exception handler
    __asm
    {
        mov     eax,[ESP]         // remove our EXCEPTION_REGISTRATION record
        mov     FS:[0], EAX       // get pointer to previous record
        add     esp, 8            // install previous record
        // clean our EXCEPTION_REGISTRATION off stack
    }

    return 0;
}

```

Сегментный регистр FS: в win32 указывает на [TIB](#). Самый первый элемент [TIB](#) это указатель на последний обработчик в цепочке. Мы сохраняем его в стеке и записываем туда адрес своего обработчика. Эта структура называется `_EXCEPTION_REGISTRATION`, это простейший односвязный список, и эти элементы хранятся прямо в стеке.

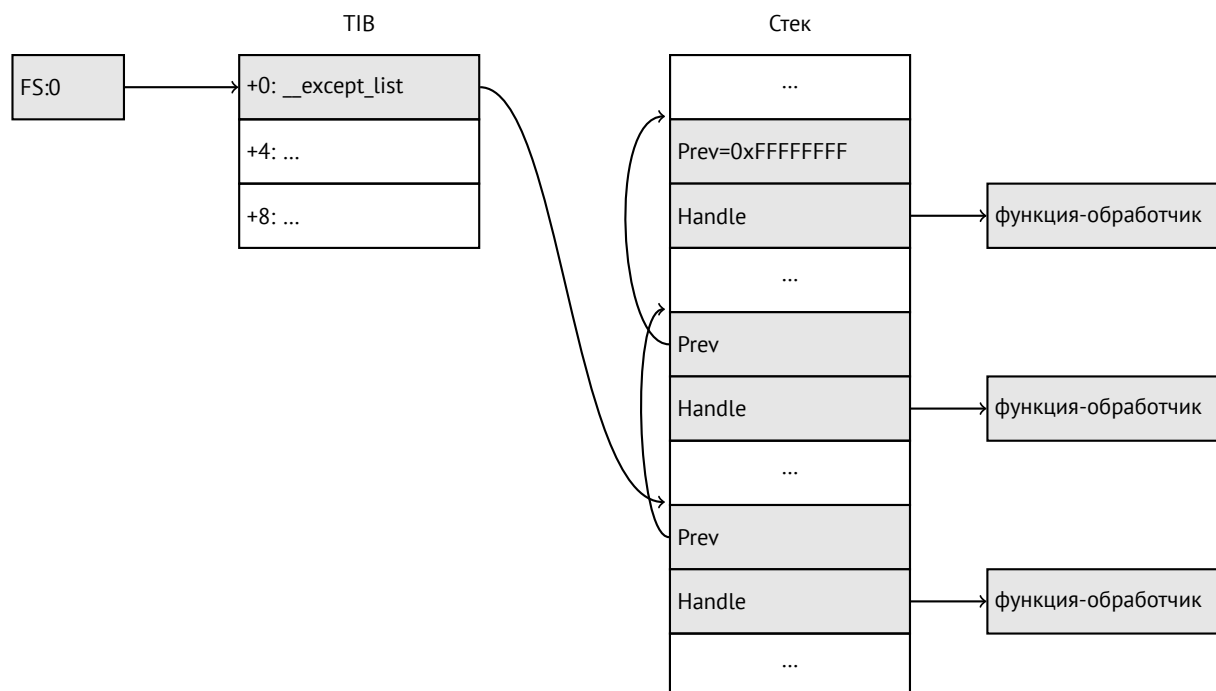
Листинг 69.1: MSVC/VC/crt/src/exsup.inc

```

_EXCEPTION_REGISTRATION struc
prev    dd    ?
handler dd    ?

```


Так что каждое поле «handler» указывает на обработчик, а каждое поле «prev» указывает на предыдущую структуру в стеке. Самая последняя структура имеет `0xFFFFFFFF` (-1) в поле «prev».



После инсталляции своего обработчика, вызываем `RaiseException()`³⁴. Это пользовательские исключения. Обработчик проверяет код. Если код `0xE1223344`, то он возвращает `ExceptionContinueExecution`, что сигнализирует системе что обработчик скорректировал состояние CPU (обычно это регистры EIP/ESP) и что ОС может возобновить исполнение треда. Если вы немного измените код так что обработчик будет возвращать `ExceptionContinueSearch`, то ОС будет вызывать остальные обработчики в цепочке, и вряд ли найдется тот, кто обработает ваше исключение, ведь информации о нем (вернее, его коде) ни у кого нет. Вы увидите стандартное окно Windows о падении процесса.

Какова разница между системными исключениями и пользовательскими? Вот системные:

как определен в WinBase.h	как определен в ntstatus.h	численное значение
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION	0xC0000005
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT	0x80000002
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT	0x80000003
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP	0x80000004
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT	0xC000008F
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION	0xC0000090
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW	0xC0000091
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK	0xC0000092
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW	0xC0000093
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW	0xC0000095
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR	0xC0000006
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION	0xC000001D
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW	0xC00000FD
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION	0xC0000026
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION	0x80000001
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE	0xC0000008
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK	0xC0000194
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT	0xC000013A

³⁴MSDN

69.3. WINDOWS SEH

Так определяется код:

31	29	28	27	16	15	0
S	U	0		Facility code		Error code

S это код статуса: 11 — ошибка; 10 — предупреждение; 01 — информация; 00 — успех. U — является ли этот код пользовательским, а не системным.

Вот почему мы выбрали 0xE1223344 — E₁₆ (1110₂) 0xE (1110_b) означает что это 1) пользовательское исключение; 2) ошибка. Хотя, если быть честным, этот пример нормально работает и без этих старших бит.

Далее мы пытаемся прочитать значение из памяти по адресу 0. Конечно, в win32 по этому адресу обычно ничего нет, и сработает исключение. Однако, первый обработчик, который будет заниматься этим делом — ваш, и он узнает об этом первым, проверяя код на соответствие с константой `EXCEPTION_ACCESS_VIOLATION`.

А если заглянуть в то что получилось на ассемблере, то можно увидеть, что код читающий из памяти по адресу 0, выглядит так:

Листинг 69.2: MSVC 2010

```
...
xor     eax, eax
mov     eax, DWORD PTR [eax] ; exception will occur here
push    eax
push    OFFSET msg
call    _printf
add     esp, 8
...
```

Возможно ли «на лету» исправить ошибку и предложить программе исполняться далее? Да, наш обработчик может изменить значение в `EAX` и предложить ОС исполнить эту же инструкцию еще раз. Что мы и делаем. `printf()` напечатает 1234, потому что после работы нашего обработчика, `EAX` будет не 0, а будет содержать адрес глобальной переменной `new_value`. Программа будет исполняться далее.

Собственно, вот что происходит: срабатывает защита менеджера памяти в CPU, он останавливает работу треда, отыскивает в ядре Windows обработчик исключений, тот, в свою очередь, начинает вызывать обработчики из цепочки SEH, по одному.

Мы компилируем это всё в MSVC 2010, но конечно же, нет никакой гарантии что для указателя будет использован именно регистр `EAX`.

Этот трюк с подменой адреса эффектно выглядит, и мы рассматриваем его здесь для наглядной иллюстрации работы SEH.

Тем не менее, трудно припомнить, применяется ли где-то подобное на практике для исправления ошибок «на лету».

Почему SEH-записи хранятся именно в стеке а не в каком-то другом месте? Вероятно, потому что ОС не нужно заботиться об освобождении этой информации, эти записи просто пропадают как ненужные когда функция заканчивает работу.

Это чем-то похоже на `alloca()`: (6.2.4 (стр. 29)).

69.3.2. Теперь вспомним MSVC

Должно быть, программистам Microsoft были нужны исключения в Си, но не в Си++, так что они добавили нестандартное расширение Си в MSVC³⁵. Оно не связано с исключениями в Си++.

```
__try
{
    ...
}
__except(filter code)
{
    handler code
}
```

Блок «finally» может присутствовать вместо код обработчика:

³⁵MSDN

```

__try
{
    ...
}
__finally
{
    ...
}

```

Код-фильтр — это выражение, отвечающее на вопрос, соответствует ли код этого обработчика к поднятому исключению. Если ваш код слишком большой и не помещается в одно выражение, отдельная функция-фильтр может быть определена.

Таких конструкций много в ядре Windows. Вот несколько примеров оттуда ([WRK](#)):

Листинг 69.3: WRK-v1.2/base/ntos/ob/obwait.c

```

try {

    KeReleaseMutant( (PKMUTANT)SignalObject,
                     MUTANT_INCREMENT,
                     FALSE,
                     TRUE );

} except((GetExceptionCode () == STATUS_ABANDONED ||
         GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
        EXCEPTION_EXECUTE_HANDLER :
        EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();

    goto WaitExit;
}

```

Листинг 69.4: WRK-v1.2/base/ntos/cache/cachesub.c

```

try {

    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                 UserBuffer,
                 MorePages ?
                 (PAGE_SIZE - PageOffset) :
                 (ReceivedLength - PageOffset) );

} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
                                     &Status ) ) {

```

Вот пример кода-фильтра:

Листинг 69.5: WRK-v1.2/base/ntos/cache/copysup.c

```

LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
)

/*++

Routine Description:

    This routine serves as a exception filter and has the special job of
    extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
    beneath us.

Arguments:

    ExceptionPointer – A pointer to the exception record that contains
                      the real Io Status.

    ExceptionCode – A pointer to an NTSTATUS that is to receive the real
                   status.

```

Return Value:

EXCEPTION_EXECUTE_HANDLER

--*/

```

{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
        (ExceptionPointer->ExceptionRecord->NumberParameters >= 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->ExceptionRecord->ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

    return EXCEPTION_EXECUTE_HANDLER;
}

```

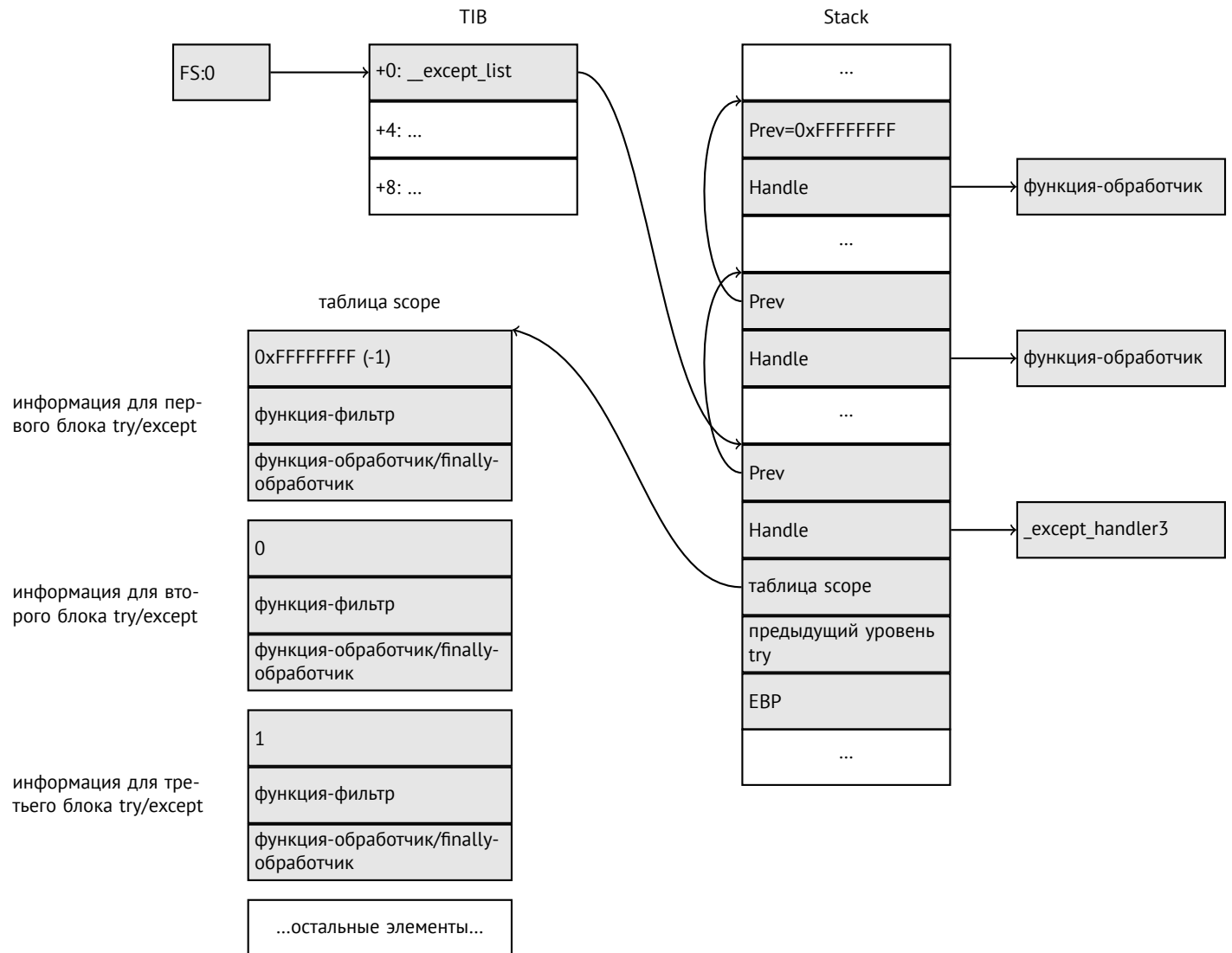
Внутри, SEH это расширение исключений поддерживаемых OS.

Но функция обработчик теперь или `_except_handler3` (для SEH3) или `_except_handler4` (для SEH4). Код обработчика от MSVC, расположен в его библиотеках, или же в `msvc*.dll`. Очень важно понимать, что SEH это специфичное для MSVC. Другие win32-компиляторы могут предлагать что-то совершенно другое.

SEH3

SEH3 имеет `_except_handler3` как функцию-обработчик, и расширяет структуру `_EXCEPTION_REGISTRATION` добавляя указатель на *scope table* и переменную *previous try level*. SEH4 расширяет *scope table* добавляя еще 4 значения связанных с защитой от переполнения буфера.

scope table это таблица, состоящая из указателей на код фильтра и обработчика, для каждого уровня вложенности *try/except*.



И снова, очень важно понимать, что OS заботится только о полях *prev/handle*, и больше ничего. Это работа функции `_except_handler3` читать другие поля, читать *scope table* и решать, какой обработчик исполнять и когда.

Исходный код функции `_except_handler3` закрыт. Хотя, Sanos OS, имеющая слой совместимости с win32, имеет некоторые функции написанные заново, которые в каком-то смысле эквивалентны тем что в Windows³⁶. Другие попытки реализации имеются в Wine³⁷ и ReactOS³⁸.

Если указатель *filter* ноль, *handler* указывает на код *finally*.

Во время исполнения, значение *previous try level* в стеке меняется, чтобы функция `_except_handler3` знала о текущем уровне вложенности, чтобы знать, какой элемент таблицы *scope table* использовать.

SEH3: пример с одним блоком try/except

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
    }
```

³⁶<http://go.yurichev.com/17058>

³⁷[GitHub](#)

³⁸<http://go.yurichev.com/17060>

69.3. WINDOWS SEH

```

    *p = 13;    // causes an access violation exception;
    printf("hello #2!\n");
}
__except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
{
    printf("access violation, can't recover\n");
}
}

```

Листинг 69.6: MSVC 2003

```

$SG74605 DB    'hello #1!', 0aH, 00H
$SG74606 DB    'hello #2!', 0aH, 00H
$SG74608 DB    'access violation, can't recover', 0aH, 00H
_DATA    ENDS

; scope table:
CONST    SEGMENT
$T74622  DD     0fffffffH    ; previous try level
         DD     FLAT:$L74617 ; filter
         DD     FLAT:$L74618 ; handler

CONST    ENDS
_TEXT    SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28    ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC NEAR
    push    ebp
    mov     ebp, esp
    push    -1                ; previous try level
    push    OFFSET FLAT:$T74622 ; scope table
    push    OFFSET FLAT:__except_handler3 ; handler
    mov     eax, DWORD PTR fs:__except_list
    push    eax                ; prev
    mov     DWORD PTR fs:__except_list, esp
    add     esp, -16
; 3 registers to be saved:
    push    ebx
    push    esi
    push    edi
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push    OFFSET FLAT:$SG74605 ; 'hello #1!'
    call    _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    push    OFFSET FLAT:$SG74606 ; 'hello #2!'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; previous try level
    jmp     SHORT $L74616

; filter code:
$L74617:
$L74627:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74621[ebp], eax
    mov     eax, DWORD PTR $T74621[ebp]
    sub     eax, -1073741819; c0000005H
    neg     eax
    sbb     eax, eax
    inc     eax
$L74619:
$L74626:

```

69.3. WINDOWS SEH

```
ret    0

; handler code:
$L74618:
mov     esp, DWORD PTR __$SEHRec$[ebp]
push    OFFSET FLAT:$SG74608 ; 'access violation, can't recover'
call    _printf
add     esp, 4
mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; setting previous try level back to -1
$L74616:
xor     eax, eax
mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
mov     DWORD PTR fs:__except_list, ecx
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
ret     0
_main   ENDP
_TEXT   ENDS
END
```

Здесь мы видим, как структура SEH конструируется в стеке. *Scope table* расположена в сегменте `CONST` — действительно, эти поля не будут меняться. Интересно, как меняется переменная *previous try level*. Исходное значение `0xFFFFFFFF` (−1). Момент, когда тело `try` открывается, обозначен инструкцией, записывающей 0 в эту переменную. В момент, когда тело `try` закрывается, −1 возвращается в нее назад. Мы также видим адреса кода фильтра и обработчика. Так мы можем легко увидеть структуру конструкций *try/except* в функции.

Так как код инициализации SEH-структур в прологе функций может быть общим для нескольких функций, иногда компилятор вставляет в прологе вызов функции `SEH_prolog()`, которая всё это делает. А код для деинициализации SEH в функции `SEH_epilog()`.

Запустим этот пример в [tracer](#):

```
tracer.exe -l:2.exe --dump-seh
```

Листинг 69.7: tracer.exe output

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0x60) handler=0x401088 (2.exe!↵
↳ main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.exe!mainCRTStartup+0x18d) handler=0↵
↳ x401545 (2.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xffffffff GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffff EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!__safe_se_handler_table+0x20↵
↳ ) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16)
```

Мы видим, что цепочка SEH состоит из 4-х обработчиков.

Первые два расположены в нашем примере. Два? Но ведь мы же сделали только один? Да, второй был установлен в `CRT`-функции `_mainCRTStartup()`, и судя по всему, он обрабатывает как минимум исключения связанные с `FPU`. Его код можно посмотреть в инсталляции MSVC: `crt/src/winxf1tr.c`.

69.3. WINDOWS SEH

Третий это SEH4 в ntdll.dll, и четвертый это обработчик, не имеющий отношения к MSVC, расположенный в ntdll.dll, имеющий «говорящее» название функции.

Как видно, в цепочке присутствуют обработчики трех типов: один не связан с MSVC вообще (последний) и два связанных с MSVC: SEH3 и SEH4.

SEH3: пример с двумя блоками try/except

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    }
};

int main()
{
    int* p = NULL;
    __try
    {
        __try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
            printf ("0x112233 raised. now let's crash\n");
            *p = 13;    // causes an access violation exception;
        }
        __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
            EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(), GetExceptionInformation()))
    {
        // the filter_user_exceptions() function answering to the question
        // "is this exception belongs to this block?"
        // if yes, do the follow:
        printf("user exception caught\n");
    }
}
```

Теперь здесь два блока `try`. Так что *scope table* теперь содержит два элемента, один элемент на каждый блок. *Previous try level* меняется вместе с тем, как исполнение доходит до очередного `try`-блока, либо выходит из него.

Листинг 69.8: MSVC 2003

```
$SG74606 DB    'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB    'yes, that is our exception', 0aH, 00H
$SG74610 DB    'not our exception', 0aH, 00H
$SG74617 DB    'hello!', 0aH, 00H
$SG74619 DB    '0x112233 raised. now let's crash', 0aH, 00H
$SG74621 DB    'access violation, can't recover', 0aH, 00H
$SG74623 DB    'user exception caught', 0aH, 00H

_code$ = 8    ; size = 4
_ep$ = 12    ; size = 4
```

69.3. WINDOWS SEH

```

_filter_user_exceptions PROC NEAR
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _code$[ebp]
    push    eax
    push    OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call    _printf
    add     esp, 8
    cmp     DWORD PTR _code$[ebp], 1122867; 00112233H
    jne     $L74607
    push    OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call    _printf
    add     esp, 4
    mov     eax, 1
    jmp     SHORT $L74605
$L74607:
    push    OFFSET FLAT:$SG74610 ; 'not our exception'
    call    _printf
    add     esp, 4
    xor     eax, eax
$L74605:
    pop     ebp
    ret     0
_filter_user_exceptions ENDP

; scope table:
CONST      SEGMENT
$T74644     DD      0fffffffH    ; previous try level for outer block
            DD      FLAT:$L74634 ; outer block filter
            DD      FLAT:$L74635 ; outer block handler
            DD      00H          ; previous try level for inner block
            DD      FLAT:$L74638 ; inner block filter
            DD      FLAT:$L74639 ; inner block handler
CONST      ENDS

$T74643 = -36      ; size = 4
$T74642 = -32      ; size = 4
_p$ = -28          ; size = 4
__$SEHRec$ = -24   ; size = 24
_main      PROC NEAR
    push    ebp
    mov     ebp, esp
    push    -1 ; previous try level
    push    OFFSET FLAT:$T74644
    push    OFFSET FLAT:__except_handler3
    mov     eax, DWORD PTR fs:__except_list
    push    eax
    mov     DWORD PTR fs:__except_list, esp
    add     esp, -20
    push    ebx
    push    esi
    push    edi
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block entered. set previous try level to 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block entered. set previous try level to 1
    push    OFFSET FLAT:$SG74617 ; 'hello!'
    call    _printf
    add     esp, 4
    push    0
    push    0
    push    0
    push    1122867      ; 00112233H
    call    DWORD PTR __imp__RaiseException@16
    push    OFFSET FLAT:$SG74619 ; '0x112233 raised. now let's crash'
    call    _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level back to 0

```


69.3. WINDOWS SEH

```
    jmp     SHORT $L74615

; inner block filter:
$L74638:
$L74650:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74643[ebp], eax
    mov     eax, DWORD PTR $T74643[ebp]
    sub     eax, -1073741819; c0000005H
    neg     eax
    sbb     eax, eax
    inc     eax
$L74640:
$L74648:
    ret     0

; inner block handler:
$L74639:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET FLAT:$SG74621 ; 'access violation, can't recover'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level back to 0

$L74615:
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block exited, set previous try level back to ↗
    ↘ -1
    jmp     SHORT $L74633

; outer block filter:
$L74634:
$L74651:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74642[ebp], eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    push    ecx
    mov     edx, DWORD PTR $T74642[ebp]
    push    edx
    call    _filter_user_exceptions
    add     esp, 8
$L74636:
$L74649:
    ret     0

; outer block handler:
$L74635:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET FLAT:$SG74623 ; 'user exception caught'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks exited. set previous try level back to ↗
    ↘ -1
$L74633:
    xor     eax, eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov     DWORD PTR fs:__except_list, ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP
```

Если установить точку останова на функцию `printf()` вызываемую из обработчика, мы можем увидеть, что добавился

69.3. WINDOWS SEH

еще один SEH-обработчик. Наверное, это еще какая-то дополнительная механика, скрытая внутри процесса обработки исключений. Тут мы также видим *scope table* состоящую из двух элементов.

```
tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh
```

Листинг 69.9: tracer.exe output

```
(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.dll!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0xb0) handler=0x40113b (3.exe!↵
↳ main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.exe!main+0x78) handler=0x401100 (3.exe!↵
↳ main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.exe!mainCRTStartup+0x18d) handler=0↵
↳ x401621 (3.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header: GSCookieOffset=0xffffffff GSCookieXOROffset=0x0
EHCookieOffset=0xffffffff EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!__safe_se_handler_table+0x20↵
↳ ) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16)
```

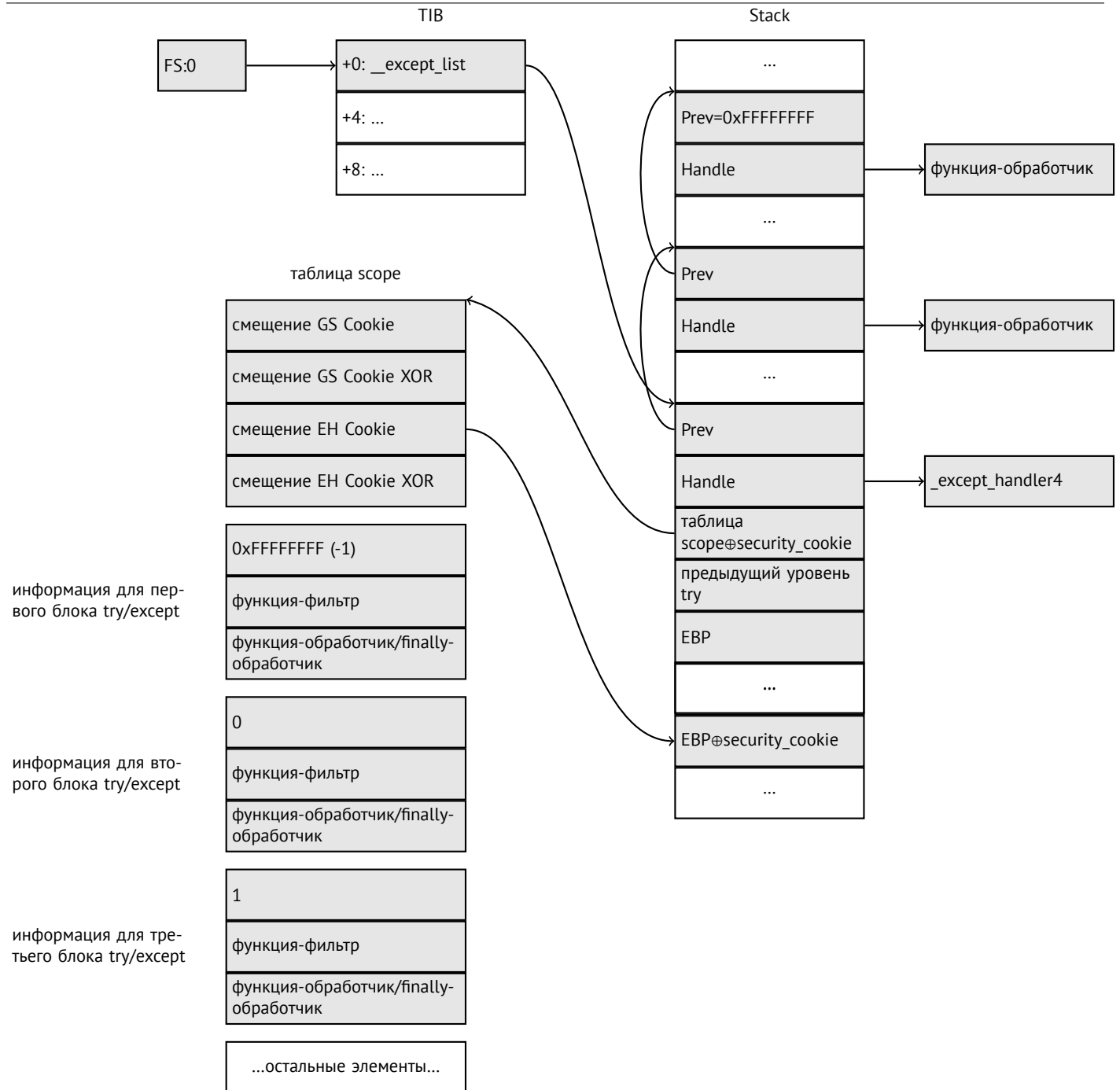
SEH4

Во время атаки переполнения буфера (19.2 (стр. 271)) адрес *scope table* может быть перезаписан, так что начиная с MSVC 2005, SEH3 был дополнен защитой от переполнения буфера, до SEH4. Указатель на *scope table* теперь [про-XOR-ен](#) с *security cookie*.

Scope table расширена, теперь имеет заголовок, содержащий 2 указателя на *security cookies*. Каждый элемент имеет смещение внутри стека на другое значение: это адрес [фрейма](#) (EBP) также [про-XOR-енный](#) с *security_cookie* расположенный в стеке. Это значение будет прочитано во время обработки исключения и проверено на правильность.

Security cookie в стеке случайное каждый раз, так что атакующий, как мы надеемся, не может предсказать его.

Изначальное значение *previous try level* это -2 в SEH4 вместо -1.



Оба примера скомпилированные в MSVC 2012 с SEH4:

Листинг 69.10: MSVC 2012: one try block example

```
$SG85485 DB 'hello #1!', 0aH, 00H
$SG85486 DB 'hello #2!', 0aH, 00H
$SG85488 DB 'access violation, can't recover', 0aH, 00H

; scope table:
xdata$x SEGMENT
__sehtable$__main DD 0fffffffEH ; GS Cookie Offset
DD 00H ; GS Cookie XOR Offset
DD 0ffffffcH ; EH Cookie Offset
DD 00H ; EH Cookie XOR Offset
DD 0fffffffEH ; previous try level
DD FLAT:$LN12@main ; filter
DD FLAT:$LN8@main ; handler
xdata$x ENDS

$T2 = -36 ; size = 4
_p$ = -32 ; size = 4
tv68 = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24
```

69.3. WINDOWS SEH

```

_main PROC
    push    ebp
    mov     ebp, esp
    push    -2
    push    OFFSET __sehtable$_main
    push    OFFSET __except_handler4
    mov     eax, DWORD PTR fs:0
    push    eax
    add     esp, -20
    push    ebx
    push    esi
    push    edi
    mov     eax, DWORD PTR __security_cookie
    xor     DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor     eax, ebp
    push    eax ; ebp ^ security_cookie
    lea     eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov     DWORD PTR fs:0, eax
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push    OFFSET $SG85485 ; 'hello #1!'
    call    _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    push    OFFSET $SG85486 ; 'hello #2!'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp     SHORT $LN6@main

; filter:
$LN7@main:
$LN12@main:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T2[ebp], eax
    cmp     DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne     SHORT $LN4@main
    mov     DWORD PTR tv68[ebp], 1
    jmp     SHORT $LN5@main
$LN4@main:
    mov     DWORD PTR tv68[ebp], 0
$LN5@main:
    mov     eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
    ret     0

; handler:
$LN8@main:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET $SG85488 ; 'access violation, can't recover'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor     eax, eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov     DWORD PTR fs:0, ecx
    pop     ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret     0
_main ENDP

```

```

$SG85486 DB 'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB 'yes, that is our exception', 0aH, 00H
$SG85490 DB 'not our exception', 0aH, 00H
$SG85497 DB 'hello!', 0aH, 00H
$SG85499 DB '0x112233 raised. now let''s crash', 0aH, 00H
$SG85501 DB 'access violation, can''t recover', 0aH, 00H
$SG85503 DB 'user exception caught', 0aH, 00H

xdata$x SEGMENT
__sehtable$__main DD 0fffffffEH ; GS Cookie Offset
                  DD 00H ; GS Cookie XOR Offset
                  DD 0fffffffC8H ; EH Cookie Offset
                  DD 00H ; EH Cookie Offset
                  DD 0fffffffEH ; previous try level for outer block
                  DD FLAT:$LN19@main ; outer block filter
                  DD FLAT:$LN9@main ; outer block handler
                  DD 00H ; previous try level for inner block
                  DD FLAT:$LN18@main ; inner block filter
                  DD FLAT:$LN13@main ; inner block handler
xdata$x ENDS

$T2 = -40 ; size = 4
$T3 = -36 ; size = 4
_p$ = -32 ; size = 4
tv72 = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24
_main PROC
    push ebp
    mov ebp, esp
    push -2 ; initial previous try level
    push OFFSET __sehtable$__main
    push OFFSET __except_handler4
    mov eax, DWORD PTR fs:0
    push eax ; prev
    add esp, -24
    push ebx
    push esi
    push edi
    mov eax, DWORD PTR __security_cookie
    xor DWORD PTR __SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor eax, ebp ; ebp ^ security_cookie
    push eax
    lea eax, DWORD PTR __SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov DWORD PTR fs:0, eax
    mov DWORD PTR __SEHRec$[ebp], esp
    mov DWORD PTR _p$[ebp], 0
    mov DWORD PTR __SEHRec$[ebp+20], 0 ; entering outer try block, setting previous try level=0
    mov DWORD PTR __SEHRec$[ebp+20], 1 ; entering inner try block, setting previous try level=1
    push OFFSET $SG85497 ; 'hello!'
    call _printf
    add esp, 4
    push 0
    push 0
    push 0
    push 1122867 ; 00112233H
    call DWORD PTR __imp__RaiseException@16
    push OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
    call _printf
    add esp, 4
    mov eax, DWORD PTR _p$[ebp]
    mov DWORD PTR [eax], 13
    mov DWORD PTR __SEHRec$[ebp+20], 0 ; exiting inner try block, set previous try level back to 0
    jmp SHORT $LN2@main

; inner block filter:
$LN12@main:
$LN18@main:
    mov ecx, DWORD PTR __SEHRec$[ebp+4]
    mov edx, DWORD PTR [ecx]

```

69.3. WINDOWS SEH

```

mov     eax, DWORD PTR [edx]
mov     DWORD PTR $T3[ebp], eax
cmp     DWORD PTR $T3[ebp], -1073741819 ; c0000005H
jne     SHORT $LN5@main
mov     DWORD PTR tv72[ebp], 1
jmp     SHORT $LN6@main
$LN5@main:
mov     DWORD PTR tv72[ebp], 0
$LN6@main:
mov     eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
ret     0

; inner block handler:
$LN13@main:
mov     esp, DWORD PTR __$SEHRec$[ebp]
push    OFFSET $SG85501 ; 'access violation, can't recover'
call    _printf
add     esp, 4
mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, setting previous try level back ↗
↳ to 0
$LN2@main:
mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level back to ↗
↳ -2
jmp     SHORT $LN7@main

; outer block filter:
$LN8@main:
$LN19@main:
mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
mov     edx, DWORD PTR [ecx]
mov     eax, DWORD PTR [edx]
mov     DWORD PTR $T2[ebp], eax
mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
push    ecx
mov     edx, DWORD PTR $T2[ebp]
push    edx
call    _filter_user_exceptions
add     esp, 8
$LN10@main:
$LN17@main:
ret     0

; outer block handler:
$LN9@main:
mov     esp, DWORD PTR __$SEHRec$[ebp]
push    OFFSET $SG85503 ; 'user exception caught'
call    _printf
add     esp, 4
mov     DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level back to ↗
↳ -2
$LN7@main:
xor     eax, eax
mov     ecx, DWORD PTR __$SEHRec$[ebp+8]
mov     DWORD PTR fs:0, ecx
pop     ecx
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
ret     0
_main   ENDP

_code$ = 8 ; size = 4
_ep$ = 12 ; size = 4
_filter_user_exceptions PROC
push    ebp
mov     ebp, esp

```

69.3. WINDOWS SEH

```
mov     eax, DWORD PTR _code$[ebp]
push    eax
push    OFFSET $SG85486 ; 'in filter. code=0x%08X'
call    _printf
add     esp, 8
cmp     DWORD PTR _code$[ebp], 1122867 ; 00112233H
jne     SHORT $LN2@filter_use
push    OFFSET $SG85488 ; 'yes, that is our exception'
call    _printf
add     esp, 4
mov     eax, 1
jmp     SHORT $LN3@filter_use
jmp     SHORT $LN3@filter_use
$LN2@filter_use:
push    OFFSET $SG85490 ; 'not our exception'
call    _printf
add     esp, 4
xor     eax, eax
$LN3@filter_use:
pop     ebp
ret     0
_filter_user_exceptions ENDP
```

Вот значение *cookies*: **Cookie Offset** это разница между адресом записанного в стеке значения ЕВР и значения $EBP \oplus security\_cookie$ в стеке. **Cookie XOR Offset** это дополнительная разница между значением $EBP \oplus security\_cookie$ и тем что записано в стеке. Если это уравнение не верно, то процесс остановится из-за разрушения стека:

$$security\_cookie \oplus (CookieXOROffset + address\_of\_saved\_EBP) == stack[address\_of\_saved\_EBP + CookieOffset]$$

Если **Cookie Offset** равно -2, это значит, что оно не присутствует.

Проверка *cookies* также реализована в моем [tracer](#), смотрите [GitHub](#) для деталей.

Возможность переключиться назад на SEH3 все еще присутствует в компиляторах после (и включая) MSVC 2005, нужно включить опцию `/GS-`, впрочем, **CRT**-код будет продолжать использовать SEH4.

69.3.3. Windows x64

Как видно, это не самая быстрая штука, устанавливать SEH-структуры в каждом прологе функции. Еще одна проблема производительности — это менять переменную *previous try level* много раз в течении исполнении функции. Так что в x64 всё сильно изменилось, теперь все указатели на **try**-блоки, функции фильтров и обработчиков, теперь записаны в другом PE-сегменте `.pdata`, откуда обработчик исключений **OC** берет всю информацию.

Вот два примера из предыдущей секции, скомпилированных для x64:

Листинг 69.12: MSVC 2012

```
$SG86276 DB      'hello #1!', 0aH, 00H
$SG86277 DB      'hello #2!', 0aH, 00H
$SG86279 DB      'access violation, can't recover', 0aH, 00H

pdata  SEGMENT
$pdata$main DD   imagerel $LN9
          DD     imagerel $LN9+61
          DD     imagerel $unwind$main
pdata  ENDS
pdata  SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
          DD     imagerel main$filt$0+32
          DD     imagerel $unwind$main$filt$0
pdata  ENDS
xdata  SEGMENT
$unwind$main DD  020609H
          DD     030023206H
          DD     imagerel __C_specific_handler
          DD     01H
          DD     imagerel $LN9+8
          DD     imagerel $LN9+40
```

69.3. WINDOWS SEH

```

        DD      imagerel main$filt$0
        DD      imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN9:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
        call    printf
        mov     DWORD PTR [rbx], 13
        lea     rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
        call    printf
        jmp     SHORT $LN8@main
$LN6@main:
        lea     rcx, OFFSET FLAT:$SG86279 ; 'access violation, can''t recover'
        call    printf
        npad    1 ; align next label
$LN8@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main     ENDP
_TEXT   ENDS

text$x   SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN5@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN7@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP
text$x   ENDS

```

Листинг 69.13: MSVC 2012

```

$SG86277 DB      'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB      'yes, that is our exception', 0aH, 00H
$SG86281 DB      'not our exception', 0aH, 00H
$SG86288 DB      'hello!', 0aH, 00H
$SG86290 DB      '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB      'access violation, can''t recover', 0aH, 00H
$SG86294 DB      'user exception caught', 0aH, 00H

pdata    SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
        DD      imagerel $LN6+73
        DD      imagerel $unwind$filter_user_exceptions
$pdata$main DD      imagerel $LN14
        DD      imagerel $LN14+95
        DD      imagerel $unwind$main
pdata    ENDS
pdata    SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32

```


69.3. WINDOWS SEH

```

        DD      imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
        DD      imagerel main$filt$1+30
        DD      imagerel $unwind$main$filt$1
pdata    ENDS

xdata    SEGMENT
$unwind$filter_user_exceptions DD 020601H
        DD      030023206H
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      02H
        DD      imagerel $LN14+8
        DD      imagerel $LN14+59
        DD      imagerel main$filt$0
        DD      imagerel $LN14+59
        DD      imagerel $LN14+8
        DD      imagerel $LN14+74
        DD      imagerel main$filt$1
        DD      imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
        DD      050023206H
$unwind$main$filt$1 DD 020601H
        DD      050023206H
xdata    ENDS

_TEXT    SEGMENT
main     PROC
$LN14:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86288 ; 'hello!'
        call    printf
        xor     r9d, r9d
        xor     r8d, r8d
        xor     edx, edx
        mov     ecx, 1122867 ; 00112233H
        call    QWORD PTR __imp_RaiseException
        lea     rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let's crash'
        call    printf
        mov     DWORD PTR [rbx], 13
        jmp     SHORT $LN13@main
$LN11@main:
        lea     rcx, OFFSET FLAT:$SG86292 ; 'access violation, can't recover'
        call    printf
        npad    1 ; align next label
$LN13@main:
        jmp     SHORT $LN9@main
$LN7@main:
        lea     rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
        call    printf
        npad    1 ; align next label
$LN9@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main     ENDP

text$x   SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN10@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H

```

69.3. WINDOWS SEH

```
    sete    cl
    mov     eax, ecx
$LN12@main$filt$:
    add     rsp, 32
    pop     rbp
    ret     0
    int     3
main$filt$0 ENDP

main$filt$1 PROC
    push    rbp
    sub     rsp, 32
    mov     rbp, rdx
$LN6@main$filt$:
    mov     rax, QWORD PTR [rcx]
    mov     rdx, rcx
    mov     ecx, DWORD PTR [rax]
    call    filter_user_exceptions
    npad    1 ; align next label
$LN8@main$filt$:
    add     rsp, 32
    pop     rbp
    ret     0
    int     3
main$filt$1 ENDP
text$x ENDS

_TEXT SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
    push    rbx
    sub     rsp, 32
    mov     ebx, ecx
    mov     edx, ecx
    lea     rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x%08X'
    call    printf
    cmp     ebx, 1122867; 00112233H
    jne     SHORT $LN2@filter_use
    lea     rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our exception'
    call    printf
    mov     eax, 1
    add     rsp, 32
    pop     rbx
    ret     0
$LN2@filter_use:
    lea     rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
    call    printf
    xor     eax, eax
    add     rsp, 32
    pop     rbx
    ret     0
filter_user_exceptions ENDP
_TEXT ENDS
```

Смотрите [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)]³⁹ для более детального описания.

Помимо информации об исключениях, секция `.pdata` также содержит начала и концы почти всех функций, так что эту информацию можно использовать в каких-либо утилитах, предназначенных для автоматизации анализа.

³⁹ Также доступно здесь: <http://go.yurichev.com/17294>

69.3.4. Больше о SEH

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)]⁴⁰, [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)]⁴¹.

69.4. Windows NT: Критические секции

Критические секции в любой ОС очень важны в мультитредовой среде, используются в основном для обеспечения гарантии что только один тред будет иметь доступ к данным в один момент времени, блокируя остальные треды и прерывания.

Вот как объявлена структура `CRITICAL_SECTION` объявлена в линейке OS Windows NT:

Листинг 69.14: (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    // The following three fields control entering and exiting the critical
    // section for the resource
    //

    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;           // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;          // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Вот как работает функция `EnterCriticalSection()`:

Листинг 69.15: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlEnterCriticalSection@4
var_C          = dword ptr -0Ch
var_8          = dword ptr -8
var_4          = dword ptr -4
arg_0          = dword ptr 8

    mov     edi, edi
    push    ebp
    mov     ebp, esp
    sub     esp, 0Ch
    push    esi
    push    edi
    mov     edi, [ebp+arg_0]
    lea     esi, [edi+4] ; LockCount
    mov     eax, esi
    lock btr dword ptr [eax], 0
    jnb     wait ; jump if CF=0

loc_7DE922DD:
    mov     eax, large fs:18h
    mov     ecx, [eax+24h]
    mov     [edi+0Ch], ecx
    mov     dword ptr [edi+8], 1
    pop     edi
    xor     eax, eax
    pop     esi
    mov     esp, ebp
    pop     ebp
    retn    4

... skipped
```

⁴⁰Также доступно здесь: <http://go.yurichev.com/17293>

⁴¹Также доступно здесь: <http://go.yurichev.com/17294>

69.4. WINDOWS NT: КРИТИЧЕСКИЕ СЕКЦИИ

Самая важная инструкция в этом фрагменте кода — это **BTR** (с префиксом **LOCK**): нулевой бит сохраняется в флаге CF и очищается в памяти. Это **атомарная операция**, блокирующая доступ всех остальных процессоров к этому значению в памяти (обратите внимание на префикс **LOCK** перед инструкцией **BTR**). Если бит в **LockCount** является 1, хорошо, сбросить его и вернуться из функции: мы в критической секции. Если нет — критическая секция уже занята другим тредом, тогда ждем.

Ожидание там сделано через вызов `WaitForSingleObject()`.

А вот как работает функция `LeaveCriticalSection()`:

Листинг 69.16: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlLeaveCriticalSection@4 proc near
arg_0          = dword ptr 8

    mov     edi, edi
    push    ebp
    mov     ebp, esp
    push    esi
    mov     esi, [ebp+arg_0]
    add     dword ptr [esi+8], 0FFFFFFFFh ; RecursionCount
    jnz     short loc_7DE922B2
    push    ebx
    push    edi
    lea     edi, [esi+4] ; LockCount
    mov     dword ptr [esi+0Ch], 0
    mov     ebx, 1
    mov     eax, edi
    lock xadd [eax], ebx
    inc     ebx
    cmp     ebx, 0FFFFFFFFh
    jnz     loc_7DEA8EB7

loc_7DE922B0:
    pop     edi
    pop     ebx

loc_7DE922B2:
    xor     eax, eax
    pop     esi
    pop     ebp
    retn    4

... skipped
```

XADD это «обменять и прибавить». В данном случае, это значит прибавить 1 к значению в **LockCount**, сохранить результат в регистре **EBX**, и в то же время 1 записывается в **LockCount**. Эта операция также атомарная, потому что также имеет префикс **LOCK**, что означает, что другие CPU или ядра CPU в системе не будут иметь доступа к этой ячейке памяти.

Префикс **LOCK** очень важен: два тред, каждый из которых работает на разных CPU или ядрах CPU, могут попытаться одновременно войти в критическую секцию, одновременно модифицируя значение в памяти, и это может привести к непредсказуемым результатам.

Часть VII

Инструменты

Глава 70

Дизассемблер

70.1. IDA

Старая бесплатная версия доступна для скачивания ¹.

Краткий справочник горячих клавиш: [F.1](#) (стр. [934](#))

¹hex-rays.com/products/ida/support/download_freeware.shtml

Глава 71

Отладчик

71.1. OllyDbg

Очень популярный отладчик пользовательской среды win32:

ollydbg.de.

Краткий справочник горячих клавиш: [F.2](#) (стр. 934)

71.2. GDB

Не очень популярный отладчик у реверсеров, тем не менее, крайне удобный.

Некоторые команды: [F.5](#) (стр. 935).

71.3. tracer

Автор часто использует *tracer* ¹ вместо отладчика.

Со временем, автор этих строк отказался использовать отладчик, потому что всё что ему нужно от него это иногда посмотреть какие-либо аргументы какой-либо функции во время исполнения или состояние регистров в определенном месте. Каждый раз загружать отладчик для этого это слишком, поэтому родилась очень простая утилита *tracer*. Она консольная, запускается из командной строки, позволяет перехватывать исполнение функций, ставить точки останова на произвольные места, смотреть состояние регистров, модифицировать их, и т.д.

Но для учебы очень полезно трассировать код руками в отладчике, наблюдать как меняются значения регистров (например, как минимум классический SoftICE, OllyDbg, WinDbg подсвечивают измененные регистры), флагов, данные, менять их самому, смотреть реакцию, и т.д.

¹yurichev.com

Глава 72

Трассировка системных вызовов

72.0.1. strace / dtruss

Позволяет показать, какие системные вызовы (syscalls([67](#) (стр. [686](#)))) прямо сейчас вызывает процесс.

Например:

```
# strace df -h

...

access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"... , 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb75b3000
```

В Mac OS X для этого же имеется dtruss.

В Cygwin также есть strace, впрочем, насколько известно, он показывает результаты только для .exe-файлов скомпилированных для среды самого cygwin.

Глава 73

Декомпиляторы

Пока существует только один публично доступный декомпилятор в Си высокого качества: Hex-Rays:

hex-rays.com/products/decompiler/

Глава 74

Прочие инструменты

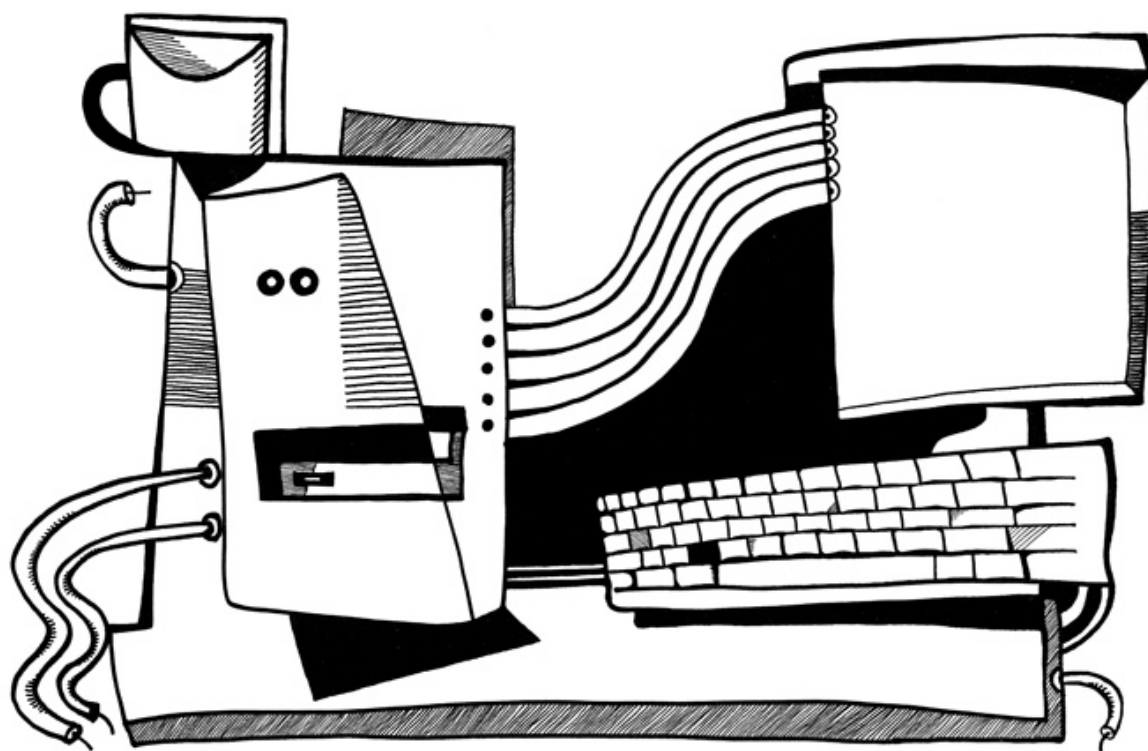
- Microsoft Visual Studio Express¹: Усеченная бесплатная версия Visual Studio, пригодная для простых экспериментов. Некоторые полезные опции: [F.3](#) (стр. 935).
- Hiew²: для мелкой модификации кода в исполняемых файлах.
- binary grep: небольшая утилита для поиска констант (либо просто последовательности байт) в большом количестве файлов, включая неисполняемые: [GitHub](#).

¹visualstudio.com/en-US/products/visual-studio-express-vs

²hiew.ru

Часть VIII

Примеры из практики



Глава 75

Шутка с task manager (Windows Vista)

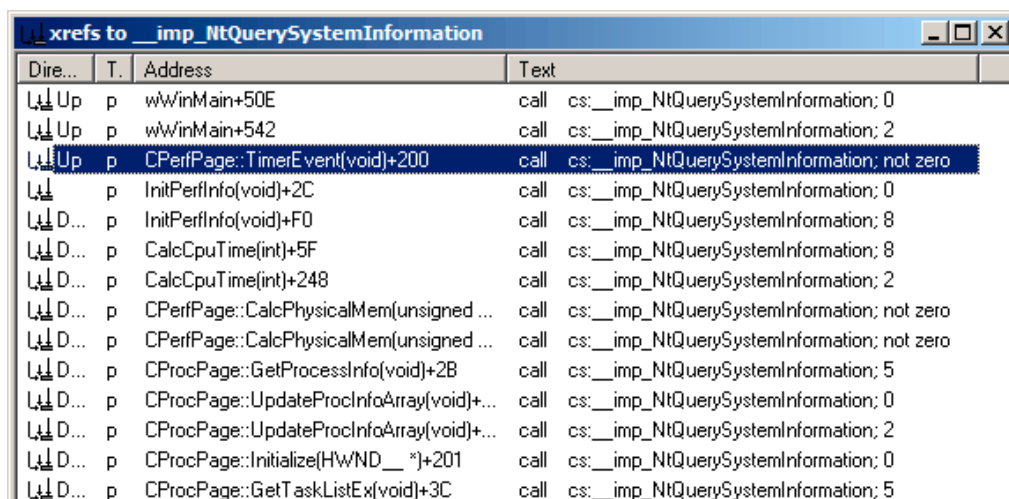
Посмотрим, сможем ли мы немного хакнуть Task Manager, чтобы он находил больше ядер в CPU, чем присутствует.

В начале задумаемся, откуда Task Manager знает количество ядер? В win32 имеется функция `GetSystemInfo()`, при помощи которой можно узнать. Но она не импортируется в `taskmgr.exe`. Есть еще одна в NTAPI, `NtQuerySystemInformation`, которая используется в `taskmgr.exe` в ряде мест. Чтобы узнать количество ядер, нужно вызвать эту функцию с константой `SystemBasicInformation` в первом аргументе (а это ноль ¹).

Второй аргумент должен указывать на буфер, который примет всю информацию.

Так что нам нужно найти все вызовы функции `NtQuerySystemInformation(0, ?, ?, ?)`. Откроем `taskmgr.exe` в IDA. Что всегда хорошо с исполняемыми файлами от Microsoft, это то что IDA может скачать соответствующий PDB-файл именно для этого файла и добавить все имена функций. Видимо, Task Manager написан на Си++ и некоторые функции и классы имеют говорящие за себя имена. Тут есть классы `CAdapter`, `CNetPage`, `CPerfPage`, `CProcInfo`, `CProcPage`, `CSvcPage`, `CTaskPage`, `CUserPage`. Должно быть, каждый класс соответствует каждой вкладке в Task Manager.

Пройдемся по всем вызовам и добавим комментарий с числом, передающимся как первый аргумент. В некоторых местах напишем «not zero», потому что значение в тех местах однозначно не ноль, но что-то другое (больше об этом во второй части главы). А мы все-таки ищем ноль передаваемый как аргумент.



Dire...	T.	Address	Text
Up	p	wWinMain+50E	call cs:__imp_NtQuerySystemInformation; 0
Up	p	wWinMain+542	call cs:__imp_NtQuerySystemInformation; 2
Up	p	CPerfPage::TimerEvent(void)+200	call cs:__imp_NtQuerySystemInformation; not zero
	p	InitPerfInfo(void)+2C	call cs:__imp_NtQuerySystemInformation; 0
D...	p	InitPerfInfo(void)+F0	call cs:__imp_NtQuerySystemInformation; 8
D...	p	CalcCpuTime(int)+5F	call cs:__imp_NtQuerySystemInformation; 8
D...	p	CalcCpuTime(int)+248	call cs:__imp_NtQuerySystemInformation; 2
D...	p	CPerfPage::CalcPhysicalMem(unsigned ...	call cs:__imp_NtQuerySystemInformation; not zero
D...	p	CPerfPage::CalcPhysicalMem(unsigned ...	call cs:__imp_NtQuerySystemInformation; not zero
D...	p	CProcPage::GetProcessInfo(void)+2B	call cs:__imp_NtQuerySystemInformation; 5
D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 0
D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 2
D...	p	CProcPage::Initialize(HWND_*)+201	call cs:__imp_NtQuerySystemInformation; 0
D...	p	CProcPage::GetTaskListEx(void)+3C	call cs:__imp_NtQuerySystemInformation; 5

Рис. 75.1: IDA: вызовы функции `NtQuerySystemInformation()`

Да, имена действительно говорящие сами за себя.

Когда мы внимательно изучим каждое место, где вызывается `NtQuerySystemInformation(0, ?, ?, ?)`, то быстро найдем то что нужно в функции `InitPerfInfo()`:

Листинг 75.1: `taskmgr.exe` (Windows Vista)

```
.text:10000B4B3      xor     r9d, r9d
.text:10000B4B6      lea     rdx, [rsp+0C78h+var_C58] ; buffer
.text:10000B4BB      xor     ecx, ecx
```

¹MSDN

```

.text:10000B4BD      lea     ebp, [r9+40h]
.text:10000B4C1      mov     r8d, ebp
.text:10000B4C4      call   cs:__imp_NtQuerySystemInformation ; 0
.text:10000B4CA      xor     ebx, ebx
.text:10000B4CC      cmp     eax, ebx
.text:10000B4CE      jge     short loc_10000B4D7
.text:10000B4D0
.text:10000B4D0      loc_10000B4D0:                                     ; CODE XREF: InitPerfInfo(void)+97
.text:10000B4D0                                     ; InitPerfInfo(void)+AF
.text:10000B4D0      xor     al, al
.text:10000B4D2      jmp     loc_10000B5EA
.text:10000B4D7      ; -----
.text:10000B4D7      loc_10000B4D7:                                     ; CODE XREF: InitPerfInfo(void)+36
.text:10000B4D7      mov     eax, [rsp+0C78h+var_C50]
.text:10000B4DB      mov     esi, ebx
.text:10000B4DD      mov     r12d, 3E80h
.text:10000B4E3      mov     cs:?g_PageSize@@3KA, eax ; ulong g_PageSize
.text:10000B4E9      shr     eax, 0Ah
.text:10000B4EC      lea     r13, __ImageBase
.text:10000B4F3      imul    eax, [rsp+0C78h+var_C4C]
.text:10000B4F8      cmp     [rsp+0C78h+var_C20], bpl
.text:10000B4FD      mov     cs:?g_MEMMax@@@3_JA, rax ; __int64 g_MEMMax
.text:10000B504      movzx   eax, [rsp+0C78h+var_C20] ; number of CPUs
.text:10000B509      cmova   eax, ebp
.text:10000B50C      cmp     al, bpl
.text:10000B50E      mov     cs:?g_cProcessors@@@3EA, al ; uchar g_cProcessors

```

`g_cProcessors` это глобальная переменная и это имя присвоено IDA в соответствии с [PDB](#)-файлом, скачанным с сервера символов Microsoft.

Байт берется из `var_C20`. И `var_C58` передается в `NtQuerySystemInformation()` как указатель на принимающий буфер. Разница между `0xC20` и `0xC58` это `0x38` (56). Посмотрим на формат структуры, который можно найти в MSDN:

```

typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION;

```

Это система x64, так что каждый `PVOID` занимает здесь 8 байт. Так что все *reserved*-поля занимают $24+4*8 = 56$. О да, это значит, что `var_C20` в локальном стеке это именно поле `NumberOfProcessors` структуры `SYSTEM_BASIC_INFORMATION`.

Проверим нашу догадку. Скопируем `taskmgr.exe` из `C:\Windows\System32` в какую-нибудь другую папку (чтобы *Windows Resource Protection* не пыталась восстанавливать измененный `taskmgr.exe`).

Откроем его в Hiew и найдем это место:

```

01` 0000B4F8: 40386C2458      cmp     [rsp][058], bpl
01` 0000B4FD: 48890544A00100  mov     [00000001`00025548], rax
01` 0000B504: 0FB6442458      movzx   eax, b, [rsp][058]
01` 0000B509: 0F47C5          cmova   eax, ebp
01` 0000B50C: 3AC3           cmp     al, bpl
01` 0000B50E: 880574950100    mov     [00000001`00024A88], al
01` 0000B514: 7645           jbe     .00000001`0000B55B --03
01` 0000B516: 488BFB         mov     rdi, rbx
01` 0000B519: 498BD4         smov    rdx, r12
01` 0000B51C: 8BCD          mov     ecx, ebp

```

Рис. 75.2: Hiew: найдем это место

Заменим инструкцию `MOVZX` на нашу. Сделаем вид что у нас 64 ядра процессора. Добавим дополнительную инструкцию `NOP` (потому что наша инструкция короче чем та что там сейчас):

75.1. ИСПОЛЬЗОВАНИЕ LEA ДЛЯ ЗАГРУЗКИ ЗНАЧЕНИЙ

00`0000A8F8: 40386C2458	cmp	[rsp][058],bp1
00`0000A8FD: 48890544A00100	mov	[000024948],rax
00`0000A904: 66B84000	mov	ax,00040 ; '@'
00`0000A908: 90	pop	
00`0000A909: 0F47C5	cmova	eax,ebp
00`0000A90C: 3AC3	cmp	al,b1
00`0000A90E: 880574950100	mov	[000023E88],a1
00`0000A914: 7645	jbe	0000A95B
00`0000A916: 488BFB	mov	rdi,rbx
00`0000A919: 498BD4	mov	rdx,r12
00`0000A91C: 8BCD	mov	ecx,ebp

Рис. 75.3: Hiew: меняем инструкцию

И это работает! Конечно же, данные в графиках неправильные. Иногда, Task Manager даже показывает общую загрузку CPU более 100%.

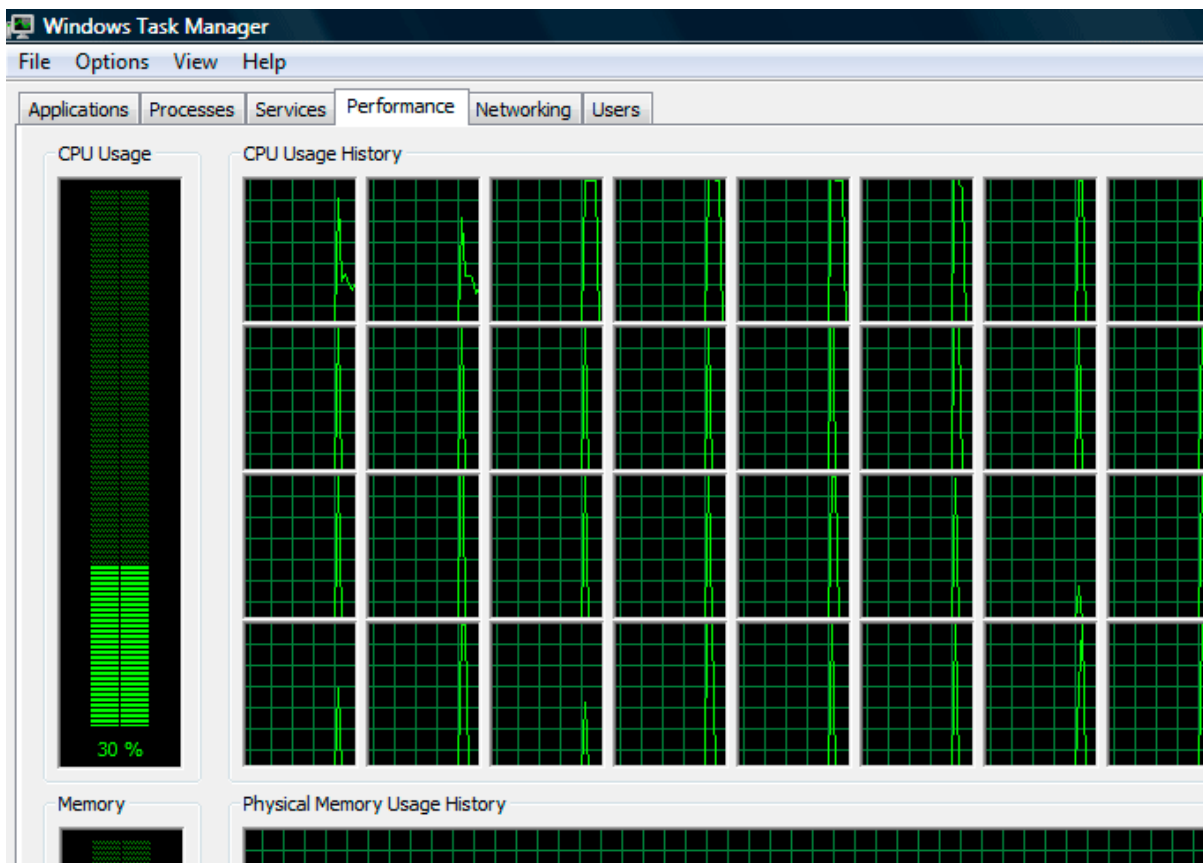


Рис. 75.4: Обманутый Windows Task Manager

Самое большое число, при котором Task Manager не падает, это 64. Должно быть, Task Manager в Windows Vista не тестировался на компьютерах с большим количеством ядер. И, наверное, там есть внутри какие-то статичные структуры данных, ограниченные до 64-х ядер.

75.1. Использование LEA для загрузки значений

Иногда, **LEA** используется в **taskmgr.exe** вместо **MOV** для установки первого аргумента **NtQuerySystemInformation()** :

Листинг 75.2: taskmgr.exe (Windows Vista)

```
xor    r9d, r9d
div     dword ptr [rsp+4C8h+WndClass.lpfWndProc]
lea     rdx, [rsp+4C8h+VersionInformation]
lea     ecx, [r9+2]      ; put 2 to ECX
mov     r8d, 138h
mov     ebx, eax
```

75.1. ИСПОЛЬЗОВАНИЕ LEA ДЛЯ ЗАГРУЗКИ ЗНАЧЕНИЙ

```
; ECX=SystemPerformanceInformation
    call    cs:__imp_NtQuerySystemInformation ; 2

    ...

    mov     r8d, 30h
    lea     r9, [rsp+298h+var_268]
    lea     rdx, [rsp+298h+var_258]
    lea     ecx, [r8-2Dh]    ; put 3 to ECX
; ECX=SystemTimeOfDayInformation
    call    cs:__imp_NtQuerySystemInformation ; not zero

    ...

    mov     rbp, [rsi+8]
    mov     r8d, 20h
    lea     r9, [rsp+98h+arg_0]
    lea     rdx, [rsp+98h+var_78]
    lea     ecx, [r8+2Fh]    ; put 0x4F to ECX
    mov     [rsp+98h+var_60], ebx
    mov     [rsp+98h+var_68], rbp
; ECX=SystemSuperfetchInformation
    call    cs:__imp_NtQuerySystemInformation ; not zero
```

Вероятно, [MSVC](#) сделал так, потому что код инструкции `LEA` короче чем `MOV REG, 5` (было бы 5 байт вместо 4).

`LEA` со смещением в пределах $-128..127$ (смещение будет занимать 1 байт в опкоде) с 32-битными регистрами даже еще короче (из-за отсутствия REX-префикса) — 3 байта.

Еще один пример подобного: [65.5.1](#) (стр. [676](#)).

Глава 76

Шутка с игрой Color Lines

Это очень популярная игра с большим количеством реализаций. Возьмем одну из них, с названием BallTriX, от 1997, доступную бесплатно на <http://go.yurichev.com/17311>¹. Вот как она выглядит:

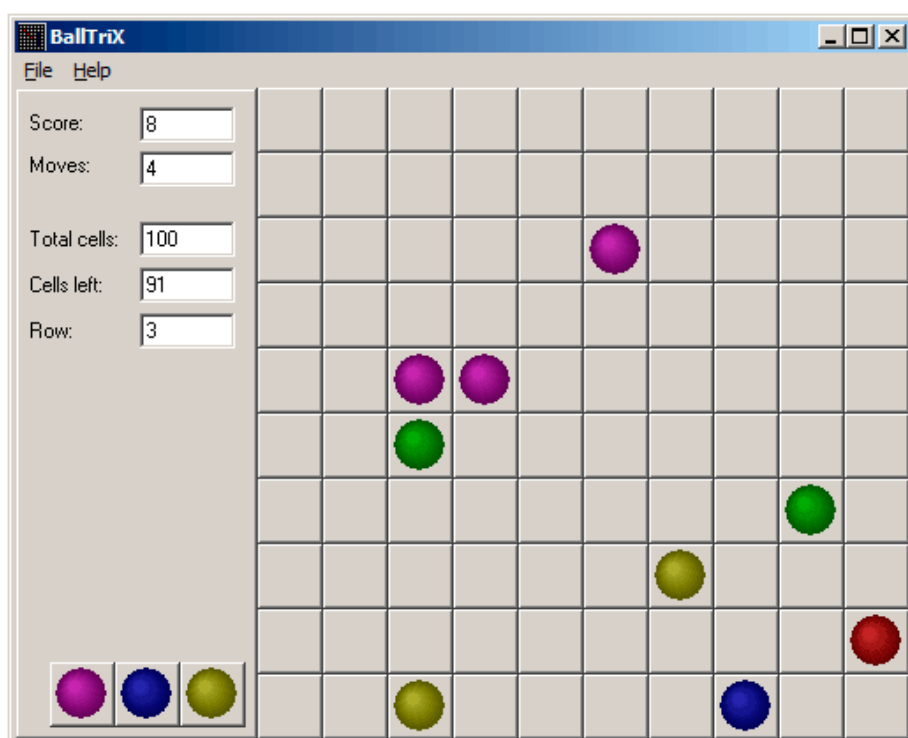


Рис. 76.1: Обычный вид игры

¹Или на <http://go.yurichev.com/17365> или <http://go.yurichev.com/17366>.

Посмотрим, сможем ли мы найти генератор псевдослучайных чисел и и сделать с ним одну шутку. IDA быстро распознает стандартную функцию `_rand` в `balltrix.exe` по адресу `0x00403DA0`. IDA также показывает, что она вызывается только из одного места:

```
.text:00402C9C sub_402C9C      proc near                ; CODE XREF: sub_402ACA+52
.text:00402C9C                                     ; sub_402ACA+64 ...
.text:00402C9C
.text:00402C9C arg_0          = dword ptr 8
.text:00402C9C
.text:00402C9C      push     ebp
.text:00402C9D      mov      ebp, esp
.text:00402C9F      push     ebx
.text:00402CA0      push     esi
.text:00402CA1      push     edi
.text:00402CA2      mov      eax, dword_40D430
.text:00402CA7      imul     eax, dword_40D440
.text:00402CAE      add      eax, dword_40D5C8
.text:0040CB4      mov      ecx, 32000
.text:0040CB9      cdq
.text:0040CBA      idiv     ecx
.text:0040CBC      mov      dword_40D440, edx
.text:0040CC2      call     _rand
.text:0040CC7      cdq
.text:0040CC8      idiv     [ebp+arg_0]
.text:0040CCB      mov      dword_40D430, edx
.text:0040CD1      mov      eax, dword_40D430
.text:0040CD6      jmp      $+5
.text:0040CDB      pop      edi
.text:0040CDC      pop      esi
.text:0040CDD      pop      ebx
.text:0040CDE      leave
.text:0040CDF      retn
.text:0040CDF sub_402C9C      endp
```

Назовем её «random». Пока не будем концентрироваться на самом коде функции.

Эта функция вызывается из трех мест.

Вот первые два:

```
.text:00402B16      mov      eax, dword_40C03C ; 10 here
.text:00402B1B      push     eax
.text:00402B1C      call     random
.text:00402B21      add      esp, 4
.text:00402B24      inc      eax
.text:00402B25      mov      [ebp+var_C], eax
.text:00402B28      mov      eax, dword_40C040 ; 10 here
.text:00402B2D      push     eax
.text:00402B2E      call     random
.text:00402B33      add      esp, 4
```

Вот третье:

```
.text:00402BBB      mov      eax, dword_40C058 ; 5 here
.text:00402BC0      push     eax
.text:00402BC1      call     random
.text:00402BC6      add      esp, 4
.text:00402BC9      inc      eax
```

Так что у функции только один аргумент. 10 передается в первых двух случаях и 5 в третьем. Мы также можем заметить, что размер доски 10×10 и здесь 5 возможных цветов. Это оно! Стандартная функция `rand()` возвращает число в пределах `0..0x7FFF` и это неудобно, так что многие программисты пишут свою функцию, возвращающую случайное число в некоторых заданных пределах. В нашем случае, предел это $0..n - 1$ и n передается как единственный аргумент в функцию. Мы можем быстро проверить это в отладчике.

Сделаем так, чтобы третий вызов функции всегда возвращал ноль. В начале заменим три инструкции (`PUSH/CALL/ADD`) на `NOPs`. Затем добавим инструкцию `XOR EAX, EAX`, для очистки регистра `EAX`.

```
.00402BB8: 83C410      add      esp,010
.00402BBB: A158C04000 mov      eax,[00040C058]
```

.00402BC0:	31C0	xor	eax, eax
.00402BC2:	90	nop	
.00402BC3:	90	nop	
.00402BC4:	90	nop	
.00402BC5:	90	nop	
.00402BC6:	90	nop	
.00402BC7:	90	nop	
.00402BC8:	90	nop	
.00402BC9:	40	inc	eax
.00402BCA:	8B4DF8	mov	ecx, [ebp] [-8]
.00402BCD:	8D0C49	lea	ecx, [ecx][ecx]*2
.00402BD0:	8B15F4D54000	mov	edx, [00040D5F4]

Что мы сделали, это заменили вызов функции `random()` на код, всегда возвращающий ноль.

Теперь запустим:

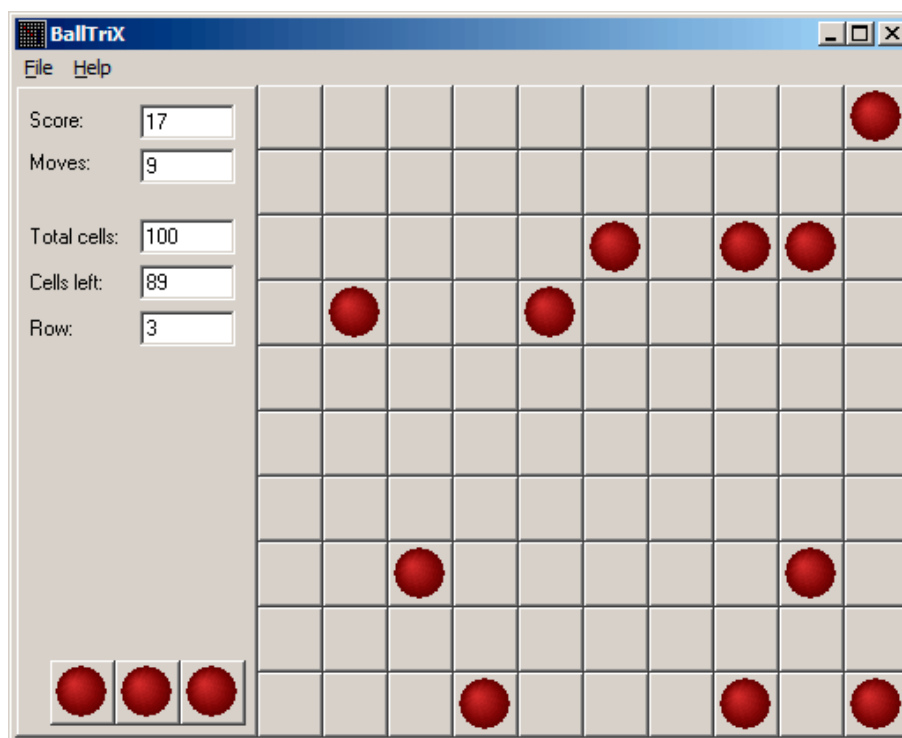


Рис. 76.2: Шутка сработала

О да, это работает².

Но почему аргументы функции `random()` это глобальные переменные? Это просто потому что в настройках игры можно изменять размер доски, так что эти параметры не фиксированы. 10 и 5 это просто значения по умолчанию.

²Автор этой книги однажды сделал это как шутку для его сотрудников, в надежде что они перестанут играть. Надежды не оправдались.

Глава 77

Сапёр (Windows XP)

Для тех, кто не очень хорошо играет в Сапёра (Minesweeper), можно попробовать найти все скрытые мины в отладчике.

Как мы знаем, Сапёр располагает мины случайным образом, так что там должен быть генератор случайных чисел или вызов стандартной функции Си `rand()`. Вот что хорошо в реверсинге продуктов от Microsoft, так это то что часто есть PDB-файл со всеми символами (имена функций, и т.д.). Когда мы загружаем `winmine.exe` в IDA, она скачивает PDB файл именно для этого исполняемого файла и добавляет все имена.

И вот оно, только один вызов `rand()` в этой функции:

```
.text:01003940 ; __stdcall Rnd(x)
.text:01003940 _Rnd@4      proc near                ; CODE XREF: StartGame()+53
.text:01003940                                     ; StartGame()+61
.text:01003940
.text:01003940 arg_0          = dword ptr 4
.text:01003940
.text:01003940      call     ds:__imp__rand
.text:01003946      cdq
.text:01003947      idiv     [esp+arg_0]
.text:0100394B      mov     eax, edx
.text:0100394D      retn     4
.text:0100394D _Rnd@4      endp
```

Так её назвала IDA и это было имя данное ей разработчиками Сапёра.

Функция очень простая:

```
int Rnd(int limit)
{
    return rand() % limit;
};
```

(В PDB-файле не было имени «limit»; это мы назвали этот аргумент так, вручную.)

Так что она возвращает случайное число в пределах от нуля до заданного предела.

`Rnd()` вызывается только из одного места, это функция с названием `StartGame()`, и как видно, это именно тот код, что расставляет мины:

```
.text:010036C7      push     _xBoxMac
.text:010036CD      call     _Rnd@4          ; Rnd(x)
.text:010036D2      push     _yBoxMac
.text:010036D8      mov     esi, eax
.text:010036DA      inc     esi
.text:010036DB      call     _Rnd@4          ; Rnd(x)
.text:010036E0      inc     eax
.text:010036E1      mov     ecx, eax
.text:010036E3      shl     ecx, 5           ; ECX=ECX*32
.text:010036E6      test    _rgBlk[ecx+esi], 80h
.text:010036EE      jnz     short loc_10036C7
.text:010036F0      shl     eax, 5           ; EAX=EAX*32
.text:010036F3      lea     eax, _rgBlk[eax+esi]
.text:010036FA      or      byte ptr [eax], 80h
.text:010036FD      dec     _cBombStart
```

```
.text:01003703          jnz     short loc_10036C7
```

Сапёр позволяет задать размеры доски, так что X (xBoxMac) и Y (yBoxMac) это глобальные переменные. Они передаются в `Rnd()` и генерируются случайные координаты. Мина устанавливается инструкцией `OR` на `0x010036FA`. И если она уже была установлена до этого (это возможно, если пара функций `Rnd()` сгенерирует пару, которая уже была сгенерирована), тогда `TEST` и `JNZ` на `0x010036E6` перейдет на повторную генерацию пары.

`cBombStart` это глобальная переменная, содержащая количество мин. Так что это цикл.

Ширина двумерного массива это 32 (мы можем это вывести, глядя на инструкцию `SHL`, которая умножает одну из координат на 32).

Размер глобального массива `rgBlk` можно легко узнать по разнице между меткой `rgBlk` в сегменте данных и следующей известной меткой. Это `0x360` (864):

```
.data:01005340 _rgBlk      db 360h dup(?)      ; DATA XREF: MainWndProc(x,x,x,x)+574
.data:01005340          ; DisplayBlk(x,x)+23
.data:010056A0 _Preferences dd ?              ; DATA XREF: FixMenus()+2
...
```

$864/32 = 27$.

Так что размер массива $27 * 32$? Это близко к тому что мы знаем: если попытаемся установить размер доски в установках Сапёра на $100 * 100$, то он установит размер $24 * 30$. Так что это максимальный размер доски здесь. И размер массива фиксирован для доски любого размера.

Посмотрим на всё это в OllyDbg. Запустим Сапёр, присоединим (attach) OllyDbg к нему и увидим содержимое памяти по адресу где массив `rgBlk` (`0x01005340`)¹.

Так что у нас выходит такой дамп памяти массива:

Address	Hex	dump
01005340	10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F	
01005350	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
01005360	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F	
01005370	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
01005380	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F	
01005390	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
010053A0	10 0F 0F 0F 0F 0F 0F 0F 8F 0F 10 0F 0F 0F 0F 0F	
010053B0	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
010053C0	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F	
010053D0	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
010053E0	10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F	
010053F0	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
01005400	10 0F 0F 8F 0F 0F 8F 0F 0F 0F 10 0F 0F 0F 0F 0F	
01005410	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
01005420	10 8F 0F 0F 8F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F	
01005430	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
01005440	10 8F 0F 0F 0F 0F 8F 0F 0F 8F 10 0F 0F 0F 0F 0F	
01005450	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
01005460	10 0F 0F 0F 0F 8F 0F 0F 0F 8F 10 0F 0F 0F 0F 0F	
01005470	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
01005480	10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F	
01005490	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
010054A0	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
010054B0	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	
010054C0	0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F	

OllyDbg, как и любой другой шестнадцатеричный редактор, показывает 16 байт на строку. Так что каждая 32-байтная строка массива занимает ровно 2 строки.

Это уровень для начинающих (доска 9*9).

Тут еще какая-то квадратная структура, заметная визуально (байты `0x10`).

Нажмем «Run» в OllyDbg чтобы разморозить процесс Сапёра, потом нажмем в случайное место окна Сапёра, попадаем на mine, но теперь видны все мины:

¹Все адреса здесь для Сапёра под Windows XP SP3 English. Они могут отличаться для других сервис-паков.



Рис. 77.1: Мины

Сравнивая места с минами и дампы, мы можем обнаружить что 0x10 это граница, 0x0F — пустой блок, 0x8F — мина.

Теперь добавим комментариев и также заключим все байты 0x8F в квадратные скобки:

```
border:
01005340  10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #1:
01005360  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005370  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #2:
01005380  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005390  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #3:
010053A0  10 0F 0F 0F 0F 0F 0F 0F 0F[8F]0F 10 0F 0F 0F 0F 0F
010053B0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #4:
010053C0  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053D0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #5:
010053E0  10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
010053F0  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #6:
01005400  10 0F 0F[8F]0F 0F[8F]0F 0F 0F 10 0F 0F 0F 0F 0F
01005410  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #7:
01005420  10[8F]0F 0F[8F]0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005430  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #8:
01005440  10[8F]0F 0F 0F 0F[8F]0F 0F[8F]10 0F 0F 0F 0F 0F
01005450  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
line #9:
01005460  10 0F 0F 0F 0F[8F]0F 0F 0F[8F]10 0F 0F 0F 0F 0F
01005470  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
border:
01005480  10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005490  0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
```

Теперь уберем все байты связанные с границами (0x10) и всё что за ними:

```
0F 0F 0F 0F 0F 0F 0F 0F 0F
0F 0F 0F 0F 0F 0F 0F 0F 0F
0F 0F 0F 0F 0F 0F 0F[8F]0F
0F 0F 0F 0F 0F 0F 0F 0F 0F
0F 0F 0F 0F 0F 0F 0F 0F 0F
0F 0F[8F]0F 0F[8F]0F 0F 0F
[8F]0F 0F[8F]0F 0F 0F 0F 0F
[8F]0F 0F 0F 0F[8F]0F 0F[8F]
0F 0F 0F 0F[8F]0F 0F 0F[8F]
```

Да, это всё мины, теперь это очень хорошо видно, в сравнении со скриншотом.

Вот что интересно, это то что мы можем модифицировать массив прямо в OllyDbg. Уберем все мины заменив все байты 0x8F на 0x0F, и вот что получится в Сапёре:

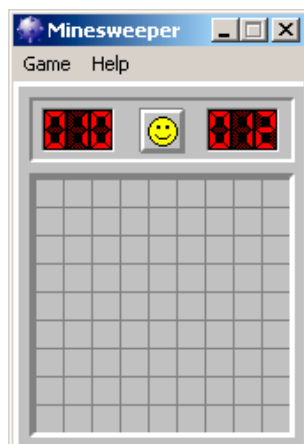


Рис. 77.2: Все мины убраны в отладчике

Также уберем их все и добавим их в первом ряду:

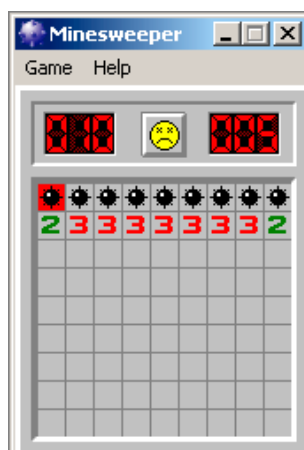


Рис. 77.3: Мины, установленные в отладчике

Отладчик не очень удобен для подсматривания (а это была наша изначальная цель), так что напишем маленькую утилиту для показа содержимого доски:

```
// Windows XP MineSweeper cheater
// written by dennis(a)yurichev.com for http://beginners.re/ book
#include <windows.h>
#include <assert.h>
#include <stdio.h>

int main (int argc, char * argv[])
{
    int i, j;
    HANDLE h;
    DWORD PID, address, rd;
    BYTE board[27][32];

    if (argc!=3)
    {
        printf ("Usage: %s <PID> <address>\n", argv[0]);
        return 0;
    };

    assert (argv[1]!=NULL);
    assert (argv[2]!=NULL);

    assert (sscanf (argv[1], "%d", &PID)==1);
    assert (sscanf (argv[2], "%x", &address)==1);
```



```

h=OpenProcess (PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE, FALSE, PID);

if (h==NULL)
{
    DWORD e=GetLastError();
    printf ("OpenProcess error: %08X\n", e);
    return 0;
};

if (ReadProcessMemory (h, (LPVOID)address, board, sizeof(board), &rd)!=TRUE)
{
    printf ("ReadProcessMemory() failed\n");
    return 0;
};

for (i=1; i<26; i++)
{
    if (board[i][0]==0x10 && board[i][1]==0x10)
        break; // end of board
    for (j=1; j<31; j++)
    {
        if (board[i][j]==0x10)
            break; // board border
        if (board[i][j]==0x8F)
            printf ("*");
        else
            printf (" ");
    };
    printf ("\n");
};

CloseHandle (h);
};

```

Просто установите [PID](#)^{2 3} и адрес массива ([0x01005340](#) для Windows XP SP3 English) и она покажет его ⁴.

Она подключается к win32-процессу по [PID](#)-у и просто читает из памяти процесса по этому адресу.

77.1. Упражнения

- Почему байты описывающие границы (0x10) присутствуют вообще? Зачем они нужны, если они вообще не видимы в интерфейсе Сапёра? Как можно обойтись без них?
- Как выясняется, здесь больше возможных значений (для открытых блоков, для тех на которых игрок установил флажок, и т.д.). Попробуйте найти значение каждого.
- Измените мою утилиту так, чтобы она в запущенном процессе Сапёра убирала все мины, или расставляла их в соответствии с каким-то заданным шаблоном.
- Измените мою утилиту так, чтобы она работала без задаваемого адреса массива и без [PDB](#)-файла. Да, вполне возможно автоматически найти информацию о доске в сегменте данных в запущенном процессе Сапёра.

²ID программы/процесса

³PID можно увидеть в Task Manager (это можно включить в «View → Select Columns»)

⁴Скомпилированная версия здесь: beginners.re

Глава 78

Донглы

Автор этих строк иногда делал замену [донглам](#) или «эмуляторы донглов» и здесь немного примеров, как это происходит. Об одном неопisanном здесь случае с Rockey и Z3 вы также можете прочитать здесь : http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf.

78.1. Пример #1: MacOS Classic и PowerPC

Вот пример программы для MacOS Classic ¹, для PowerPC. Компания, разработавшая этот продукт, давно исчезла, так что (легальный) пользователь боялся того что донгла может сломаться.

Если запустить программу без подключенной донглы, можно увидеть окно с надписью "Invalid Security Device". Мне повезло потому что этот текст можно было легко найти внутри исполняемого файла .

Представим, что мы не знакомы ни с Mac OS Classic, ни с PowerPC, но всё-таки попробуем.

IDA открывает исполняемый файл легко, показывая его тип как "PEF (Mac OS or Be OS executable)" (действительно, это стандартный тип файлов в Mac OS Classic).

В поисках текстовой строки с сообщением об ошибке, мы попадаем на этот фрагмент кода:

```
...
seg000:000C87FC 38 60 00 01      li      %r3, 1
seg000:000C8800 48 03 93 41      bl      check1
seg000:000C8804 60 00 00 00      nop
seg000:000C8808 54 60 06 3F      clrlwi. %r0, %r3, 24
seg000:000C880C 40 82 00 40      bne     OK
seg000:000C8810 80 62 9F D8      lwz     %r3, TC_aInvalidSecurityDevice
...
```

Да, это код PowerPC. Это очень типичный процессор для RISC 1990-х . Каждая инструкция занимает 4 байта (как и в MIPS и ARM) и их имена немного похожи на имена инструкций MIPS .

`check1()` это имя которое мы дадим этой функции немного позже. `BL` это инструкция *Branch Link* т.е. предназначенная для вызова подпрограмм. Самое важное место — это инструкция `BNE`, срабатывающая, если проверка наличия донглы прошла успешно, либо не срабатывающая в случае ошибки: и тогда адрес текстовой строки с сообщением об ошибке будет загружен в регистр `r3` для последующей передачи в функцию отображения диалогового окна .

Из [Steve Zucker, SunSoft and Kari Karhi, IBM, *SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement*, (1995)]² мы узнаем, что регистр `r3` используется для возврата значений (и еще `r4` если значение 64-битное).

Еще одна, пока что неизвестная инструкция `CLRLWI` . Из [PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, (2000)]³ мы узнаем, что эта инструкция одновременно и очищает и загружает. В нашем случае, она очищает 24 старших бита из значения в `r3` и записывает всё это в `r0`, так что это аналог `MOVZX` в x86 (16.1.1 (стр. 198)), но также устанавливает флаги, так что `BNE` может проверить их потом.

Посмотрим внутри `check1()` :

¹MacOS перед тем как перейти на UNIX

²Также доступно здесь: http://yurichev.com/mirrors/PowerPC/elfspec_ppc.pdf

³Также доступно здесь: http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf

```

seg000:00101B40      check1: # CODE XREF: seg000:00063E7Cp
seg000:00101B40      # sub_64070+160p ...
seg000:00101B40
seg000:00101B40      .set arg_8, 8
seg000:00101B40
seg000:00101B40 7C 08 02 A6      mflr    %r0
seg000:00101B44 90 01 00 08      stw     %r0, arg_8(%sp)
seg000:00101B48 94 21 FF C0      stwu    %sp, -0x40(%sp)
seg000:00101B4C 48 01 6B 39      bl      check2
seg000:00101B50 60 00 00 00      nop
seg000:00101B54 80 01 00 48      lwz     %r0, 0x40+arg_8(%sp)
seg000:00101B58 38 21 00 40      addi    %sp, %sp, 0x40
seg000:00101B5C 7C 08 03 A6      mtlr    %r0
seg000:00101B60 4E 80 00 20      blr
seg000:00101B60      # End of function check1

```

Как можно увидеть в [IDA](#), эта функция вызывается из многих мест в программе, но только значение в регистре r3 проверяется сразу после каждого вызова. Всё что эта функция делает это только вызывает другую функцию, так что это [thunk function](#): здесь присутствует и пролог функции и эпилог, но регистр r3 не трогается, так что `check1()` возвращает то, что возвращает `check2()`.

[BLR](#)⁴ это похоже возврат из функции, но так как IDA делает всю разметку функций автоматически, наверное, мы можем пока не интересоваться этим. Так как это типичный [RISC](#), похоже, подпрограммы вызываются, используя [link register](#), точно как в ARM.

Функция `check2()` более сложная:

```

seg000:00118684      check2: # CODE XREF: check1+Cp
seg000:00118684
seg000:00118684      .set var_18, -0x18
seg000:00118684      .set var_C, -0xC
seg000:00118684      .set var_8, -8
seg000:00118684      .set var_4, -4
seg000:00118684      .set arg_8, 8
seg000:00118684
seg000:00118684 93 E1 FF FC      stw     %r31, var_4(%sp)
seg000:00118688 7C 08 02 A6      mflr    %r0
seg000:0011868C 83 E2 95 A8      lwz     %r31, off_1485E8 # dword_24B704
seg000:00118690      .using dword_24B704, %r31
seg000:00118690 93 C1 FF F8      stw     %r30, var_8(%sp)
seg000:00118694 93 A1 FF F4      stw     %r29, var_C(%sp)
seg000:00118698 7C 7D 1B 78      mr      %r29, %r3
seg000:0011869C 90 01 00 08      stw     %r0, arg_8(%sp)
seg000:001186A0 54 60 06 3E      clrlwi  %r0, %r3, 24
seg000:001186A4 28 00 00 01      cmplwi  %r0, 1
seg000:001186A8 94 21 FF B0      stwu    %sp, -0x50(%sp)
seg000:001186AC 40 82 00 0C      bne     loc_1186B8
seg000:001186B0 38 60 00 01      li      %r3, 1
seg000:001186B4 48 00 00 6C      b       exit
seg000:001186B8
seg000:001186B8      loc_1186B8: # CODE XREF: check2+28j
seg000:001186B8 48 00 03 D5      bl      sub_118A8C
seg000:001186BC 60 00 00 00      nop
seg000:001186C0 3B C0 00 00      li      %r30, 0
seg000:001186C4
seg000:001186C4      skip:      # CODE XREF: check2+94j
seg000:001186C4 57 C0 06 3F      clrlwi. %r0, %r30, 24
seg000:001186C8 41 82 00 18      beq     loc_1186E0
seg000:001186CC 38 61 00 38      addi    %r3, %sp, 0x50+var_18
seg000:001186D0 80 9F 00 00      lwz     %r4, dword_24B704
seg000:001186D4 48 00 C0 55      bl      .RBEFINDNEXT
seg000:001186D8 60 00 00 00      nop
seg000:001186DC 48 00 00 1C      b       loc_1186F8
seg000:001186E0
seg000:001186E0      loc_1186E0: # CODE XREF: check2+44j
seg000:001186E0 80 BF 00 00      lwz     %r5, dword_24B704
seg000:001186E4 38 81 00 38      addi    %r4, %sp, 0x50+var_18

```

⁴(PowerPC) Branch to Link Register

78.1. ПРИМЕР #1: MACOS CLASSIC И POWERPC

```
seg000:001186E8 38 60 08 C2    li      %r3, 0x1234
seg000:001186EC 48 00 BF 99    bl      .RBEFINDFIRST
seg000:001186F0 60 00 00 00    nop
seg000:001186F4 3B C0 00 01    li      %r30, 1
seg000:001186F8
seg000:001186F8          loc_1186F8: # CODE XREF: check2+58j
seg000:001186F8 54 60 04 3F    clrlwi. %r0, %r3, 16
seg000:001186FC 41 82 00 0C    beq     must_jump
seg000:00118700 38 60 00 00    li      %r3, 0          # error
seg000:00118704 48 00 00 1C    b       exit
seg000:00118708
seg000:00118708          must_jump: # CODE XREF: check2+78j
seg000:00118708 7F A3 EB 78    mr      %r3, %r29
seg000:0011870C 48 00 00 31    bl      check3
seg000:00118710 60 00 00 00    nop
seg000:00118714 54 60 06 3F    clrlwi. %r0, %r3, 24
seg000:00118718 41 82 FF AC    beq     skip
seg000:0011871C 38 60 00 01    li      %r3, 1
seg000:00118720
seg000:00118720          exit:      # CODE XREF: check2+30j
seg000:00118720          # check2+80j
seg000:00118720 80 01 00 58    lwz     %r0, 0x50+arg_8(%sp)
seg000:00118724 38 21 00 50    addi    %sp, %sp, 0x50
seg000:00118728 83 E1 FF FC    lwz     %r31, var_4(%sp)
seg000:0011872C 7C 08 03 A6    mtlr    %r0
seg000:00118730 83 C1 FF F8    lwz     %r30, var_8(%sp)
seg000:00118734 83 A1 FF F4    lwz     %r29, var_C(%sp)
seg000:00118738 4E 80 00 20    blr
seg000:00118738          # End of function check2
```

Снова повезло: имена некоторых функций оставлены в исполняемом файле (в символах в отладочной секции? Трудно сказать до тех пор, пока мы не знакомы с этим форматом файлов, может быть это что-то вроде PE-экспортов (69.2.7 (стр. 699)))? как например `.RBEFINDNEXT()` and `.RBEFINDFIRST()`. В итоге, эти функции вызывают другие функции с именами вроде `.GetNextDeviceViaUSB()`, `.USBSendPKT()`, так что они явно работают с каким-то USB-устройством.

Тут даже есть функция с названием `.GetNextEve3Device()` — звучит знакомо, в 1990-х годах была донгла Sentinel Eve3 для ADB-порта (присутствующих на Макинтошах).

В начале посмотрим на то как устанавливается регистр `r3` одновременно игнорируя всё остальное. Мы знаем, что «хорошее» значение в `r3` должно быть не нулевым, а нулевой `r3` приведет к выводу диалогового окна с сообщением об ошибке.

В функции имеются две инструкции `li %r3, 1` и одна `li %r3, 0` (*Load Immediate*, т.е. загрузить значение в регистр). Самая первая инструкция находится на `0x001186B0` — и честно говоря, трудно заранее понять, что это означает.

А вот то что мы видим дальше понять проще: вызывается `.RBEFINDFIRST()` и в случае ошибки, 0 будет записан в `r3` и мы перейдем на `exit`, а иначе будет вызвана функция `check3()` — если и она будет выполнена с ошибкой, будет вызвана `.RBEFINDNEXT()` вероятно, для поиска другого USB-устройства.

Н.В.: `clrlwi. %r0, %r3, 16` это аналог того что мы уже видели, но она очищает 16 старших бит, т.е., `.RBEFINDFIRST()` вероятно возвращает 16-битное значение.

B (означает *branch*) — безусловный переход.

BEQ это обратная инструкция от **BNE**.

Посмотрим на `check3()`:

```
seg000:0011873C          check3: # CODE XREF: check2+88p
seg000:0011873C
seg000:0011873C          .set var_18, -0x18
seg000:0011873C          .set var_C, -0xC
seg000:0011873C          .set var_8, -8
seg000:0011873C          .set var_4, -4
seg000:0011873C          .set arg_8, 8
seg000:0011873C
seg000:0011873C 93 E1 FF FC    stw     %r31, var_4(%sp)
seg000:00118740 7C 08 02 A6    mflr    %r0
seg000:00118744 38 A0 00 00    li      %r5, 0
```

78.1. ПРИМЕР #1: MACOS CLASSIC И POWERPC

```

seg000:00118748 93 C1 FF F8 stw    %r30, var_8(%sp)
seg000:0011874C 83 C2 95 A8 lwz    %r30, off_1485E8 # dword_24B704
seg000:00118750          .using dword_24B704, %r30
seg000:00118750 93 A1 FF F4 stw    %r29, var_C(%sp)
seg000:00118754 3B A3 00 00 addi   %r29, %r3, 0
seg000:00118758 38 60 00 00 li     %r3, 0
seg000:0011875C 90 01 00 08 stw    %r0, arg_8(%sp)
seg000:00118760 94 21 FF B0 stwu   %sp, -0x50(%sp)
seg000:00118764 80 DE 00 00 lwz    %r6, dword_24B704
seg000:00118768 38 81 00 38 addi   %r4, %sp, 0x50+var_18
seg000:0011876C 48 00 C0 5D bl     .RBEREAD
seg000:00118770 60 00 00 00 nop
seg000:00118774 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118778 41 82 00 0C beq    loc_118784
seg000:0011877C 38 60 00 00 li     %r3, 0
seg000:00118780 48 00 02 F0 b      exit
seg000:00118784
seg000:00118784          loc_118784: # CODE XREF: check3+3Cj
seg000:00118784 A0 01 00 38 lhz    %r0, 0x50+var_18(%sp)
seg000:00118788 28 00 04 B2 cmplwi %r0, 0x1100
seg000:0011878C 41 82 00 0C beq    loc_118798
seg000:00118790 38 60 00 00 li     %r3, 0
seg000:00118794 48 00 02 DC b      exit
seg000:00118798
seg000:00118798          loc_118798: # CODE XREF: check3+50j
seg000:00118798 80 DE 00 00 lwz    %r6, dword_24B704
seg000:0011879C 38 81 00 38 addi   %r4, %sp, 0x50+var_18
seg000:001187A0 38 60 00 01 li     %r3, 1
seg000:001187A4 38 A0 00 00 li     %r5, 0
seg000:001187A8 48 00 C0 21 bl     .RBEREAD
seg000:001187AC 60 00 00 00 nop
seg000:001187B0 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:001187B4 41 82 00 0C beq    loc_1187C0
seg000:001187B8 38 60 00 00 li     %r3, 0
seg000:001187BC 48 00 02 B4 b      exit
seg000:001187C0
seg000:001187C0          loc_1187C0: # CODE XREF: check3+78j
seg000:001187C0 A0 01 00 38 lhz    %r0, 0x50+var_18(%sp)
seg000:001187C4 28 00 06 4B cmplwi %r0, 0x09AB
seg000:001187C8 41 82 00 0C beq    loc_1187D4
seg000:001187CC 38 60 00 00 li     %r3, 0
seg000:001187D0 48 00 02 A0 b      exit
seg000:001187D4
seg000:001187D4          loc_1187D4: # CODE XREF: check3+8Cj
seg000:001187D4 4B F9 F3 D9 bl     sub_B7BAC
seg000:001187D8 60 00 00 00 nop
seg000:001187DC 54 60 06 3E clrlwi. %r0, %r3, 24
seg000:001187E0 2C 00 00 05 cmpwi  %r0, 5
seg000:001187E4 41 82 01 00 beq    loc_1188E4
seg000:001187E8 40 80 00 10 bge    loc_1187F8
seg000:001187EC 2C 00 00 04 cmpwi  %r0, 4
seg000:001187F0 40 80 00 58 bge    loc_118848
seg000:001187F4 48 00 01 8C b      loc_118980
seg000:001187F8
seg000:001187F8          loc_1187F8: # CODE XREF: check3+ACj
seg000:001187F8 2C 00 00 0B cmpwi  %r0, 0xB
seg000:001187FC 41 82 00 08 beq    loc_118804
seg000:00118800 48 00 01 80 b      loc_118980
seg000:00118804
seg000:00118804          loc_118804: # CODE XREF: check3+C0j
seg000:00118804 80 DE 00 00 lwz    %r6, dword_24B704
seg000:00118808 38 81 00 38 addi   %r4, %sp, 0x50+var_18
seg000:0011880C 38 60 00 08 li     %r3, 8
seg000:00118810 38 A0 00 00 li     %r5, 0
seg000:00118814 48 00 BF B5 bl     .RBEREAD
seg000:00118818 60 00 00 00 nop
seg000:0011881C 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118820 41 82 00 0C beq    loc_11882C
seg000:00118824 38 60 00 00 li     %r3, 0
seg000:00118828 48 00 02 48 b      exit

```

78.1. ПРИМЕР #1: MACOS CLASSIC И POWERPC

```

seg000:0011882C
seg000:0011882C      loc_11882C: # CODE XREF: check3+E4j
seg000:0011882C A0 01 00 38  lhz      %r0, 0x50+var_18(%sp)
seg000:00118830 28 00 11 30  cmplwi   %r0, 0xFE40
seg000:00118834 41 82 00 0C  beq      loc_118840
seg000:00118838 38 60 00 00  li       %r3, 0
seg000:0011883C 48 00 02 34  b        exit
seg000:00118840
seg000:00118840      loc_118840: # CODE XREF: check3+F8j
seg000:00118840 38 60 00 01  li       %r3, 1
seg000:00118844 48 00 02 2C  b        exit
seg000:00118848
seg000:00118848      loc_118848: # CODE XREF: check3+B4j
seg000:00118848 80 DE 00 00  lwz      %r6, dword_24B704
seg000:0011884C 38 81 00 38  addi     %r4, %sp, 0x50+var_18
seg000:00118850 38 60 00 0A  li       %r3, 0xA
seg000:00118854 38 A0 00 00  li       %r5, 0
seg000:00118858 48 00 BF 71  bl       .RBEREAD
seg000:0011885C 60 00 00 00  nop
seg000:00118860 54 60 04 3F  clrlwi.  %r0, %r3, 16
seg000:00118864 41 82 00 0C  beq      loc_118870
seg000:00118868 38 60 00 00  li       %r3, 0
seg000:0011886C 48 00 02 04  b        exit
seg000:00118870
seg000:00118870      loc_118870: # CODE XREF: check3+128j
seg000:00118870 A0 01 00 38  lhz      %r0, 0x50+var_18(%sp)
seg000:00118874 28 00 03 F3  cmplwi   %r0, 0xA6E1
seg000:00118878 41 82 00 0C  beq      loc_118884
seg000:0011887C 38 60 00 00  li       %r3, 0
seg000:00118880 48 00 01 F0  b        exit
seg000:00118884
seg000:00118884      loc_118884: # CODE XREF: check3+13Cj
seg000:00118884 57 BF 06 3E  clrlwi   %r31, %r29, 24
seg000:00118888 28 1F 00 02  cmplwi   %r31, 2
seg000:0011888C 40 82 00 0C  bne      loc_118898
seg000:00118890 38 60 00 01  li       %r3, 1
seg000:00118894 48 00 01 DC  b        exit
seg000:00118898
seg000:00118898      loc_118898: # CODE XREF: check3+150j
seg000:00118898 80 DE 00 00  lwz      %r6, dword_24B704
seg000:0011889C 38 81 00 38  addi     %r4, %sp, 0x50+var_18
seg000:001188A0 38 60 00 0B  li       %r3, 0xB
seg000:001188A4 38 A0 00 00  li       %r5, 0
seg000:001188A8 48 00 BF 21  bl       .RBEREAD
seg000:001188AC 60 00 00 00  nop
seg000:001188B0 54 60 04 3F  clrlwi.  %r0, %r3, 16
seg000:001188B4 41 82 00 0C  beq      loc_1188C0
seg000:001188B8 38 60 00 00  li       %r3, 0
seg000:001188BC 48 00 01 B4  b        exit
seg000:001188C0
seg000:001188C0      loc_1188C0: # CODE XREF: check3+178j
seg000:001188C0 A0 01 00 38  lhz      %r0, 0x50+var_18(%sp)
seg000:001188C4 28 00 23 1C  cmplwi   %r0, 0x1C20
seg000:001188C8 41 82 00 0C  beq      loc_1188D4
seg000:001188CC 38 60 00 00  li       %r3, 0
seg000:001188D0 48 00 01 A0  b        exit
seg000:001188D4
seg000:001188D4      loc_1188D4: # CODE XREF: check3+18Cj
seg000:001188D4 28 1F 00 03  cmplwi   %r31, 3
seg000:001188D8 40 82 01 94  bne      error
seg000:001188DC 38 60 00 01  li       %r3, 1
seg000:001188E0 48 00 01 90  b        exit
seg000:001188E4
seg000:001188E4      loc_1188E4: # CODE XREF: check3+A8j
seg000:001188E4 80 DE 00 00  lwz      %r6, dword_24B704
seg000:001188E8 38 81 00 38  addi     %r4, %sp, 0x50+var_18
seg000:001188EC 38 60 00 0C  li       %r3, 0xC
seg000:001188F0 38 A0 00 00  li       %r5, 0
seg000:001188F4 48 00 BE D5  bl       .RBEREAD
seg000:001188F8 60 00 00 00  nop

```

78.1. ПРИМЕР #1: MACOS CLASSIC И POWERPC

```

seg000:001188FC 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:00118900 41 82 00 0C  beq     loc_11890C
seg000:00118904 38 60 00 00  li      %r3, 0
seg000:00118908 48 00 01 68  b       exit
seg000:0011890C
seg000:0011890C          loc_11890C: # CODE XREF: check3+1C4j
seg000:0011890C A0 01 00 38  lhz     %r0, 0x50+var_18(%sp)
seg000:00118910 28 00 1F 40  cmplwi  %r0, 0x40FF
seg000:00118914 41 82 00 0C  beq     loc_118920
seg000:00118918 38 60 00 00  li      %r3, 0
seg000:0011891C 48 00 01 54  b       exit
seg000:00118920
seg000:00118920          loc_118920: # CODE XREF: check3+1D8j
seg000:00118920 57 BF 06 3E  clrlwi  %r31, %r29, 24
seg000:00118924 28 1F 00 02  cmplwi  %r31, 2
seg000:00118928 40 82 00 0C  bne     loc_118934
seg000:0011892C 38 60 00 01  li      %r3, 1
seg000:00118930 48 00 01 40  b       exit
seg000:00118934
seg000:00118934          loc_118934: # CODE XREF: check3+1ECj
seg000:00118934 80 DE 00 00  lwz     %r6, dword_24B704
seg000:00118938 38 81 00 38  addi    %r4, %sp, 0x50+var_18
seg000:0011893C 38 60 00 0D  li      %r3, 0xD
seg000:00118940 38 A0 00 00  li      %r5, 0
seg000:00118944 48 00 BE 85  bl      .RBEREAD
seg000:00118948 60 00 00 00  nop
seg000:0011894C 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:00118950 41 82 00 0C  beq     loc_11895C
seg000:00118954 38 60 00 00  li      %r3, 0
seg000:00118958 48 00 01 18  b       exit
seg000:0011895C
seg000:0011895C          loc_11895C: # CODE XREF: check3+214j
seg000:0011895C A0 01 00 38  lhz     %r0, 0x50+var_18(%sp)
seg000:00118960 28 00 07 CF  cmplwi  %r0, 0xFC7
seg000:00118964 41 82 00 0C  beq     loc_118970
seg000:00118968 38 60 00 00  li      %r3, 0
seg000:0011896C 48 00 01 04  b       exit
seg000:00118970
seg000:00118970          loc_118970: # CODE XREF: check3+228j
seg000:00118970 28 1F 00 03  cmplwi  %r31, 3
seg000:00118974 40 82 00 F8  bne     error
seg000:00118978 38 60 00 01  li      %r3, 1
seg000:0011897C 48 00 00 F4  b       exit
seg000:00118980
seg000:00118980          loc_118980: # CODE XREF: check3+B8j
seg000:00118980          # check3+C4j
seg000:00118980 80 DE 00 00  lwz     %r6, dword_24B704
seg000:00118984 38 81 00 38  addi    %r4, %sp, 0x50+var_18
seg000:00118988 3B E0 00 00  li      %r31, 0
seg000:0011898C 38 60 00 04  li      %r3, 4
seg000:00118990 38 A0 00 00  li      %r5, 0
seg000:00118994 48 00 BE 35  bl      .RBEREAD
seg000:00118998 60 00 00 00  nop
seg000:0011899C 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:001189A0 41 82 00 0C  beq     loc_1189AC
seg000:001189A4 38 60 00 00  li      %r3, 0
seg000:001189A8 48 00 00 C8  b       exit
seg000:001189AC
seg000:001189AC          loc_1189AC: # CODE XREF: check3+264j
seg000:001189AC A0 01 00 38  lhz     %r0, 0x50+var_18(%sp)
seg000:001189B0 28 00 1D 6A  cmplwi  %r0, 0xAED0
seg000:001189B4 40 82 00 0C  bne     loc_1189C0
seg000:001189B8 3B E0 00 01  li      %r31, 1
seg000:001189BC 48 00 00 14  b       loc_1189D0
seg000:001189C0
seg000:001189C0          loc_1189C0: # CODE XREF: check3+278j
seg000:001189C0 28 00 18 28  cmplwi  %r0, 0x2818
seg000:001189C4 41 82 00 0C  beq     loc_1189D0
seg000:001189C8 38 60 00 00  li      %r3, 0
seg000:001189CC 48 00 00 A4  b       exit

```

78.1. ПРИМЕР #1: MACOS CLASSIC И POWERPC

```

seg000:001189D0
seg000:001189D0          loc_1189D0: # CODE XREF: check3+280j
seg000:001189D0          # check3+288j
seg000:001189D0 57 A0 06 3E  clrlwi  %r0, %r29, 24
seg000:001189D4 28 00 00 02  cmplwi  %r0, 2
seg000:001189D8 40 82 00 20  bne     loc_1189F8
seg000:001189DC 57 E0 06 3F  clrlwi. %r0, %r31, 24
seg000:001189E0 41 82 00 10  beq     good2
seg000:001189E4 48 00 4C 69  bl      sub_11D64C
seg000:001189E8 60 00 00 00  nop
seg000:001189EC 48 00 00 84  b        exit
seg000:001189F0
seg000:001189F0          good2:      # CODE XREF: check3+2A4j
seg000:001189F0 38 60 00 01  li      %r3, 1
seg000:001189F4 48 00 00 7C  b        exit
seg000:001189F8
seg000:001189F8          loc_1189F8: # CODE XREF: check3+29Cj
seg000:001189F8 80 DE 00 00  lwz     %r6, dword_24B704
seg000:001189FC 38 81 00 38  addi    %r4, %sp, 0x50+var_18
seg000:00118A00 38 60 00 05  li      %r3, 5
seg000:00118A04 38 A0 00 00  li      %r5, 0
seg000:00118A08 48 00 BD C1  bl      .RBEREAD
seg000:00118A0C 60 00 00 00  nop
seg000:00118A10 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:00118A14 41 82 00 0C  beq     loc_118A20
seg000:00118A18 38 60 00 00  li      %r3, 0
seg000:00118A1C 48 00 00 54  b        exit
seg000:00118A20
seg000:00118A20          loc_118A20: # CODE XREF: check3+2D8j
seg000:00118A20 A0 01 00 38  lhz     %r0, 0x50+var_18(%sp)
seg000:00118A24 28 00 11 D3  cmplwi  %r0, 0xD300
seg000:00118A28 40 82 00 0C  bne     loc_118A34
seg000:00118A2C 3B E0 00 01  li      %r31, 1
seg000:00118A30 48 00 00 14  b        good1
seg000:00118A34
seg000:00118A34          loc_118A34: # CODE XREF: check3+2ECj
seg000:00118A34 28 00 1A EB  cmplwi  %r0, 0xEBA1
seg000:00118A38 41 82 00 0C  beq     good1
seg000:00118A3C 38 60 00 00  li      %r3, 0
seg000:00118A40 48 00 00 30  b        exit
seg000:00118A44
seg000:00118A44          good1:      # CODE XREF: check3+2F4j
seg000:00118A44          # check3+2FCj
seg000:00118A44 57 A0 06 3E  clrlwi  %r0, %r29, 24
seg000:00118A48 28 00 00 03  cmplwi  %r0, 3
seg000:00118A4C 40 82 00 20  bne     error
seg000:00118A50 57 E0 06 3F  clrlwi. %r0, %r31, 24
seg000:00118A54 41 82 00 10  beq     good
seg000:00118A58 48 00 4B F5  bl      sub_11D64C
seg000:00118A5C 60 00 00 00  nop
seg000:00118A60 48 00 00 10  b        exit
seg000:00118A64
seg000:00118A64          good:      # CODE XREF: check3+318j
seg000:00118A64 38 60 00 01  li      %r3, 1
seg000:00118A68 48 00 00 08  b        exit
seg000:00118A6C
seg000:00118A6C          error:      # CODE XREF: check3+19Cj
seg000:00118A6C          # check3+238j ...
seg000:00118A6C 38 60 00 00  li      %r3, 0
seg000:00118A70
seg000:00118A70          exit:      # CODE XREF: check3+44j
seg000:00118A70          # check3+58j ...
seg000:00118A70 80 01 00 58  lwz     %r0, 0x50+arg_8(%sp)
seg000:00118A74 38 21 00 50  addi    %sp, %sp, 0x50
seg000:00118A78 83 E1 FF FC  lwz     %r31, var_4(%sp)
seg000:00118A7C 7C 08 03 A6  mtlr    %r0
seg000:00118A80 83 C1 FF F8  lwz     %r30, var_8(%sp)
seg000:00118A84 83 A1 FF F4  lwz     %r29, var_C(%sp)
seg000:00118A88 4E 80 00 20  blr
seg000:00118A88          # End of function check3

```


78.2. ПРИМЕР #2: SCO OPENSERVR

Здесь много вызовов `.RBEREAD()`. Эта функция вероятно читает какие-то значения из донглы, которые потом сравниваются здесь при помощи `CMPLWI`.

Мы также видим в регистр `r3` записывается перед каждым вызовом `.RBEREAD()` одно из этих значений: 0, 1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Вероятно адрес в памяти или что-то в этом роде?

Да, действительно, если погуглить имена этих функций, можно легко найти документацию к Sentinel Eve3!

Наверное, уже и не нужно изучать остальные инструкции PowerPC: всё что делает эта функция это просто вызывает `.RBEREAD()`, сравнивает его результаты с константами и возвращает 1 если результат сравнения положительный или 0 в другом случае.

Всё ясно: `check1()` должна всегда возвращать 1 или иное ненулевое значение. Но так как мы не очень уверены в своих знаниях инструкций PowerPC, будем осторожны и пропатчим переходы в `check2` на адресах `0x001186FC` и `0x00118718`.

На `0x001186FC` мы записываем байты `0x48` и `0` таким образом превращая инструкцию `BEQ` в инструкцию `B` (безусловный переход): Мы можем заметить этот опкод прямо в коде даже без обращения к *[PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, (2000)]*⁵.

На `0x00118718` мы записываем байт `0x60` и еще 3 нулевых байта, таким образом превращая её в инструкцию `NOP`: Этот опкод мы также можем подсмотреть прямо в коде.

И всё заработало без подключенной донглы.

Резюмируя, такие простые модификации можно делать в IDA даже с минимальными знаниями ассемблера.

78.2. Пример #2: SCO OpenServer

Древняя программа для SCO OpenServer от 1997 разработанная давно исчезнувшей компанией.

Специальный драйвер донглы устанавливается в системе, он содержит такие текстовые строки: «Copyright 1989, Rainbow Technologies, Inc., Irvine, CA» и «Sentinel Integrated Driver Ver. 3.0».

После инсталляции драйвера, в `/dev` появляются такие устройства:

```
/dev/rbsl8
/dev/rbsl9
/dev/rbsl10
```

Без подключенной донглы, программа сообщает об ошибке, но сообщение об ошибке не удается найти в исполняемых файлах.

Еще раз спасибо IDA, она легко загружает исполняемые файлы формата COFF использующиеся в SCO OpenServer.

Попробуем также поискать строку «rbsl», и действительно, её можно найти в таком фрагменте кода:

```
.text:00022AB8      public SSQC
.text:00022AB8 SSQC  proc near ; CODE XREF: SSQ+7p
.text:00022AB8
.text:00022AB8 var_44 = byte ptr -44h
.text:00022AB8 var_29 = byte ptr -29h
.text:00022AB8 arg_0  = dword ptr  8
.text:00022AB8
.text:00022AB8      push    ebp
.text:00022AB8      mov     ebp, esp
.text:00022AB9      sub     esp, 44h
.text:00022ABB      push    edi
.text:00022ABE      mov     edi, offset unk_4035D0
.text:00022AC4      push    esi
.text:00022AC5      mov     esi, [ebp+arg_0]
.text:00022AC8      push    ebx
.text:00022AC9      push    esi
.text:00022ACA      call    strlen
.text:00022ACF      add     esp, 4
.text:00022AD2      cmp     eax, 2
.text:00022AD7      jnz     loc_22BA4
.text:00022ADD      inc     esi
```

⁵Также доступно здесь: http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf

78.2. ПРИМЕР #2: SCO OPENSERVR

```
.text:00022ADE      mov     al, [esi-1]
.text:00022AE1      movsx   eax, al
.text:00022AE4      cmp     eax, '3'
.text:00022AE9      jz      loc_22B84
.text:00022AEF      cmp     eax, '4'
.text:00022AF4      jz      loc_22B94
.text:00022AFA      cmp     eax, '5'
.text:00022AFF      jnz     short loc_22B6B
.text:00022B01      movsx   ebx, byte ptr [esi]
.text:00022B04      sub     ebx, '0'
.text:00022B07      mov     eax, 7
.text:00022B0C      add     eax, ebx
.text:00022B0E      push    eax
.text:00022B0F      lea     eax, [ebp+var_44]
.text:00022B12      push    offset aDevSlD ; "/dev/sl%d"
.text:00022B17      push    eax
.text:00022B18      call    nl_sprintf
.text:00022B1D      push    0 ; int
.text:00022B1F      push    offset aDevRbsl8 ; char *
.text:00022B24      call    _access
.text:00022B29      add     esp, 14h
.text:00022B2C      cmp     eax, 0FFFFFFFh
.text:00022B31      jz      short loc_22B48
.text:00022B33      lea     eax, [ebx+7]
.text:00022B36      push    eax
.text:00022B37      lea     eax, [ebp+var_44]
.text:00022B3A      push    offset aDevRbslD ; "/dev/rbsl%d"
.text:00022B3F      push    eax
.text:00022B40      call    nl_sprintf
.text:00022B45      add     esp, 0Ch
.text:00022B48
.text:00022B48 loc_22B48: ; CODE XREF: SSQC+79j
.text:00022B48      mov     edx, [edi]
.text:00022B4A      test    edx, edx
.text:00022B4C      jle     short loc_22B57
.text:00022B4E      push    edx ; int
.text:00022B4F      call    _close
.text:00022B54      add     esp, 4
.text:00022B57
.text:00022B57 loc_22B57: ; CODE XREF: SSQC+94j
.text:00022B57      push    2 ; int
.text:00022B59      lea     eax, [ebp+var_44]
.text:00022B5C      push    eax ; char *
.text:00022B5D      call    _open
.text:00022B62      add     esp, 8
.text:00022B65      test    eax, eax
.text:00022B67      mov     [edi], eax
.text:00022B69      jge     short loc_22B78
.text:00022B6B
.text:00022B6B loc_22B6B: ; CODE XREF: SSQC+47j
.text:00022B6B      mov     eax, 0FFFFFFFh
.text:00022B70      pop     ebx
.text:00022B71      pop     esi
.text:00022B72      pop     edi
.text:00022B73      mov     esp, ebp
.text:00022B75      pop     ebp
.text:00022B76      retn
.text:00022B78
.text:00022B78 loc_22B78: ; CODE XREF: SSQC+B1j
.text:00022B78      pop     ebx
.text:00022B79      pop     esi
.text:00022B7A      pop     edi
.text:00022B7B      xor     eax, eax
.text:00022B7D      mov     esp, ebp
.text:00022B7F      pop     ebp
.text:00022B80      retn
.text:00022B84
.text:00022B84 loc_22B84: ; CODE XREF: SSQC+31j
.text:00022B84      mov     al, [esi]
.text:00022B86      pop     ebx
```

78.2. ПРИМЕР #2: SCO OPENSERVR

```
.text:00022B87      pop     esi
.text:00022B88      pop     edi
.text:00022B89      mov     ds:byte_407224, al
.text:00022B8E      mov     esp, ebp
.text:00022B90      xor     eax, eax
.text:00022B92      pop     ebp
.text:00022B93      retn
.text:00022B94
.text:00022B94 loc_22B94: ; CODE XREF: SSQC+3Cj
.text:00022B94      mov     al, [esi]
.text:00022B96      pop     ebx
.text:00022B97      pop     esi
.text:00022B98      pop     edi
.text:00022B99      mov     ds:byte_407225, al
.text:00022B9E      mov     esp, ebp
.text:00022BA0      xor     eax, eax
.text:00022BA2      pop     ebp
.text:00022BA3      retn
.text:00022BA4
.text:00022BA4 loc_22BA4: ; CODE XREF: SSQC+1Fj
.text:00022BA4      movsx   eax, ds:byte_407225
.text:00022BAB      push    esi
.text:00022BAC      push    eax
.text:00022BAD      movsx   eax, ds:byte_407224
.text:00022BB4      push    eax
.text:00022BB5      lea     eax, [ebp+var_44]
.text:00022BB8      push    offset a46CCS ; "46%c%c%s"
.text:00022BBD      push    eax
.text:00022BBE      call    nl_sprintf
.text:00022BC3      lea     eax, [ebp+var_44]
.text:00022BC6      push    eax
.text:00022BC7      call    strlen
.text:00022BCC      add     esp, 18h
.text:00022BCF      cmp     eax, 1Bh
.text:00022BD4      jle     short loc_22BDA
.text:00022BD6      mov     [ebp+var_29], 0
.text:00022BDA
.text:00022BDA loc_22BDA: ; CODE XREF: SSQC+11Cj
.text:00022BDA      lea     eax, [ebp+var_44]
.text:00022BDD      push    eax
.text:00022BDE      call    strlen
.text:00022BE3      push    eax ; unsigned int
.text:00022BE4      lea     eax, [ebp+var_44]
.text:00022BE7      push    eax ; void *
.text:00022BE8      mov     eax, [edi]
.text:00022BEA      push    eax ; int
.text:00022BEB      call    _write
.text:00022BF0      add     esp, 10h
.text:00022BF3      pop     ebx
.text:00022BF4      pop     esi
.text:00022BF5      pop     edi
.text:00022BF6      mov     esp, ebp
.text:00022BF8      pop     ebp
.text:00022BF9      retn
.text:00022BFA      db 0Eh dup(90h)
.text:00022BFA SSQC endp
```

Действительно, должна же как-то программа обмениваться информацией с драйвером .

Единственное место где вызывается функция `SSQC()` это [thunk function](#):

```
.text:0000DBE8      public SSQ
.text:0000DBE8 SSQ      proc near ; CODE XREF: sys_info+A9p
.text:0000DBE8                  ; sys_info+CBp ...
.text:0000DBE8
.text:0000DBE8 arg_0    = dword ptr 8
.text:0000DBE8
.text:0000DBE8      push    ebp
.text:0000DBE9      mov     ebp, esp
.text:0000DBEB      mov     edx, [ebp+arg_0]
```

78.2. ПРИМЕР #2: SCO OPENSERVR

```
.text:0000DBEE      push    edx
.text:0000DBEF      call    SSQC
.text:0000DBF4      add     esp, 4
.text:0000DBF7      mov     esp, ebp
.text:0000DBF9      pop     ebp
.text:0000DBFA      retn
.text:0000DBFB SSQ   endp
```

А вот SSQ() вызывается по крайней мере из двух разных функций .

Одна из них:

```
.data:0040169C _51_52_53      dd offset aPressAnyKeyT_0 ; DATA XREF: init_sys+392r
.data:0040169C                                     ; sys_info+A1r
.data:0040169C                                     ; "PRESS ANY KEY TO CONTINUE: "
.data:004016A0      dd offset a51          ; "51"
.data:004016A4      dd offset a52          ; "52"
.data:004016A8      dd offset a53          ; "53"

...

.data:004016B8 _3C_or_3E      dd offset a3c          ; DATA XREF: sys_info:loc_D67Br
.data:004016B8                                     ; "3C"
.data:004016BC      dd offset a3e          ; "3E"

; these names we gave to the labels:
.data:004016C0 answers1      dd 6B05h          ; DATA XREF: sys_info+E7r
.data:004016C4      dd 3D87h
.data:004016C8 answers2      dd 3Ch          ; DATA XREF: sys_info+F2r
.data:004016CC      dd 832h
.data:004016D0 _C_and_B      db 0Ch          ; DATA XREF: sys_info+BAr
.data:004016D0      ; sys_info:OKr
.data:004016D1 byte_4016D1    db 0Bh          ; DATA XREF: sys_info+FDr
.data:004016D2      db 0

...

.text:0000D652      xor     eax, eax
.text:0000D654      mov     al, ds:ctl_port
.text:0000D659      mov     ecx, _51_52_53[eax*4]
.text:0000D660      push    ecx
.text:0000D661      call    SSQ
.text:0000D666      add     esp, 4
.text:0000D669      cmp     eax, 0FFFFFFFh
.text:0000D66E      jz      short loc_D6D1
.text:0000D670      xor     ebx, ebx
.text:0000D672      mov     al, _C_and_B
.text:0000D677      test    al, al
.text:0000D679      jz      short loc_D6C0
.text:0000D67B      loc_D67B: ; CODE XREF: sys_info+106j
.text:0000D67B      mov     eax, _3C_or_3E[ebx*4]
.text:0000D682      push    eax
.text:0000D683      call    SSQ
.text:0000D688      push    offset a4g          ; "4G"
.text:0000D68D      call    SSQ
.text:0000D692      push    offset a0123456789 ; "0123456789"
.text:0000D697      call    SSQ
.text:0000D69C      add     esp, 0Ch
.text:0000D69F      mov     edx, answers1[ebx*4]
.text:0000D6A6      cmp     eax, edx
.text:0000D6A8      jz      short OK
.text:0000D6AA      mov     ecx, answers2[ebx*4]
.text:0000D6B1      cmp     eax, ecx
.text:0000D6B3      jz      short OK
.text:0000D6B5      mov     al, byte_4016D1[ebx]
.text:0000D6BB      inc     ebx
.text:0000D6BC      test    al, al
.text:0000D6BE      jnz     short loc_D67B
.text:0000D6C0
```

78.2. ПРИМЕР #2: SCO OPENSERVR

```
.text:0000D6C0 loc_D6C0: ; CODE XREF: sys_info+C1j
.text:0000D6C0          inc     ds:ctl_port
.text:0000D6C6          xor     eax, eax
.text:0000D6C8          mov     al, ds:ctl_port
.text:0000D6CD          cmp     eax, edi
.text:0000D6CF          jle     short loc_D652
.text:0000D6D1
.text:0000D6D1 loc_D6D1: ; CODE XREF: sys_info+98j
.text:0000D6D1          ; sys_info+B6j
.text:0000D6D1          mov     edx, [ebp+var_8]
.text:0000D6D4          inc     edx
.text:0000D6D5          mov     [ebp+var_8], edx
.text:0000D6D8          cmp     edx, 3
.text:0000D6DB          jle     loc_D641
.text:0000D6E1
.text:0000D6E1 loc_D6E1: ; CODE XREF: sys_info+16j
.text:0000D6E1          ; sys_info+51j ...
.text:0000D6E1          pop     ebx
.text:0000D6E2          pop     edi
.text:0000D6E3          mov     esp, ebp
.text:0000D6E5          pop     ebp
.text:0000D6E6          retn
.text:0000D6E8 OK:      ; CODE XREF: sys_info+F0j
.text:0000D6E8          ; sys_info+FBj
.text:0000D6E8          mov     al, _C_and_B[ebx]
.text:0000D6EE          pop     ebx
.text:0000D6EF          pop     edi
.text:0000D6F0          mov     ds:ctl_model, al
.text:0000D6F5          mov     esp, ebp
.text:0000D6F7          pop     ebp
.text:0000D6F8          retn
.text:0000D6F8 sys_info endp
```

« 3C » и « 3E » — это звучит знакомо: когда-то была донгла Sentinel Pro от Rainbow без памяти, предоставляющая только одну секретную крипто-хеширующую функцию.

О том, что такое хэш-функция, было описано здесь: [35](#) (стр. [452](#)).

Но вернемся к нашей программе. Так что программа может только проверить подключена ли донгла или нет. Никакой больше информации в такую донглу без памяти записать нельзя. Двухсимвольные коды — это команды (можно увидеть, как они обрабатываются в функции `SSQC()`) а все остальные строки хешируются внутри донглы превращаясь в 16-битное число. Алгоритм был секретный, так что нельзя было написать замену драйверу или сделать электронную копию донглы идеально эмулирующую алгоритм. С другой стороны, всегда можно было перехватить все обращения к ней и найти те константы, с которыми сравнивается результат хеширования. Но надо сказать, вполне возможно создать устойчивую защиту от копирования базирующуюся на секретной хеш-функции: пусть она шифрует все файлы с которыми ваша программа работает.

Но вернемся к нашему коду.

Коды 51/52/53 используются для выбора номера принтерного LPT-порта. 3х/4х используются для выбора «family» так донглы Sentinel Pro можно отличать друг от друга: ведь более одной донглы может быть подключено к LPT-порту.

Единственная строка, передающаяся в хеш-функцию это "0123456789". Затем результат сравнивается с несколькими правильными значениями.

Если результат правилен, 0xC или 0xB будет записано в глобальную переменную `ctl_model`.

Еще одна строка для хеширования: "PRESS ANY KEY TO CONTINUE:", но результат не проверяется. Трудно сказать, зачем это, может быть по ошибке ⁶.

Давайте посмотрим, где проверяется значение глобальной переменной `ctl_mode`.

Одно из таких мест:

```
.text:0000D708 prep_sys proc near ; CODE XREF: init_sys+46Ap
.text:0000D708
.text:0000D708 var_14 = dword ptr -14h
.text:0000D708 var_10 = byte ptr -10h
.text:0000D708 var_8 = dword ptr -8
.text:0000D708 var_2 = word ptr -2
```

⁶Это очень странное чувство: находить ошибки в столь древнем ПО.

78.2. ПРИМЕР #2: SCO OPENSERVR

```
.text:0000D708
.text:0000D708      push    ebp
.text:0000D709      mov     eax, ds:net_env
.text:0000D70E      mov     ebp, esp
.text:0000D710      sub     esp, 1Ch
.text:0000D713      test    eax, eax
.text:0000D715      jnz     short loc_D734
.text:0000D717      mov     al, ds:ctl_model
.text:0000D71C      test    al, al
.text:0000D71E      jnz     short loc_D77E
.text:0000D720      mov     [ebp+var_8], offset aIeCvulnv0kgT_ ; "Ie-cvulnvV\\b0KG]T_"
.text:0000D727      mov     edx, 7
.text:0000D72C      jmp     loc_D7E7

...

.text:0000D7E7 loc_D7E7: ; CODE XREF: prep_sys+24j
.text:0000D7E7      ; prep_sys+33j
.text:0000D7E7      push    edx
.text:0000D7E8      mov     edx, [ebp+var_8]
.text:0000D7EB      push    20h
.text:0000D7ED      push    edx
.text:0000D7EE      push    16h
.text:0000D7F0      call    err_warn
.text:0000D7F5      push    offset station_sem
.text:0000D7FA      call    ClosSem
.text:0000D7FF      call    startup_err
```

Если оно 0, шифрованное сообщение об ошибке будет передано в функцию дешифрования, и оно будет показано.

Функция дешифровки сообщений об ошибке похоже применяет простой [xoring](#) :

```
.text:0000A43C err_warn      proc near      ; CODE XREF: prep_sys+E8p
.text:0000A43C      ; prep_sys2+2Fp ...
.text:0000A43C
.text:0000A43C var_55      = byte ptr -55h
.text:0000A43C var_54      = byte ptr -54h
.text:0000A43C arg_0       = dword ptr 8
.text:0000A43C arg_4       = dword ptr 0Ch
.text:0000A43C arg_8       = dword ptr 10h
.text:0000A43C arg_C       = dword ptr 14h
.text:0000A43C
.text:0000A43C      push    ebp
.text:0000A43D      mov     ebp, esp
.text:0000A43F      sub     esp, 54h
.text:0000A442      push    edi
.text:0000A443      mov     ecx, [ebp+arg_8]
.text:0000A446      xor     edi, edi
.text:0000A448      test    ecx, ecx
.text:0000A44A      push    esi
.text:0000A44B      jle     short loc_A466
.text:0000A44D      mov     esi, [ebp+arg_C] ; key
.text:0000A450      mov     edx, [ebp+arg_4] ; string
.text:0000A453
.text:0000A453 loc_A453:      ; CODE XREF: err_warn+28j
.text:0000A453      xor     eax, eax
.text:0000A455      mov     al, [edx+edi]
.text:0000A458      xor     eax, esi
.text:0000A45A      add     esi, 3
.text:0000A45D      inc     edi
.text:0000A45E      cmp     edi, ecx
.text:0000A460      mov     [ebp+edi+var_55], al
.text:0000A464      jl      short loc_A453
.text:0000A466
.text:0000A466 loc_A466:      ; CODE XREF: err_warn+Fj
.text:0000A466      mov     [ebp+edi+var_54], 0
.text:0000A46B      mov     eax, [ebp+arg_0]
.text:0000A46E      cmp     eax, 18h
.text:0000A473      jnz     short loc_A49C
.text:0000A475      lea     eax, [ebp+var_54]
```

78.2. ПРИМЕР #2: SCO OPENSERVR

```
.text:0000A478      push     eax
.text:0000A479      call    status_line
.text:0000A47E      add     esp, 4
.text:0000A481
.text:0000A481 loc_A481:                                ; CODE XREF: err_warn+72j
.text:0000A481      push    50h
.text:0000A483      push    0
.text:0000A485      lea     eax, [ebp+var_54]
.text:0000A488      push    eax
.text:0000A489      call    memset
.text:0000A48E      call    pcvt_refresh
.text:0000A493      add     esp, 0Ch
.text:0000A496      pop     esi
.text:0000A497      pop     edi
.text:0000A498      mov     esp, ebp
.text:0000A49A      pop     ebp
.text:0000A49B      retn
.text:0000A49C
.text:0000A49C loc_A49C:                                ; CODE XREF: err_warn+37j
.text:0000A49C      push    0
.text:0000A49E      lea     eax, [ebp+var_54]
.text:0000A4A1      mov     edx, [ebp+arg_0]
.text:0000A4A4      push    edx
.text:0000A4A5      push    eax
.text:0000A4A6      call    pcvt_lputs
.text:0000A4AB      add     esp, 0Ch
.text:0000A4AE      jmp     short loc_A481
.text:0000A4AE err_warn      endp
```

Вот почему не получилось найти сообщение об ошибке в исполняемых файлах, потому что оно было зашифровано, это очень популярная практика.

Еще один вызов хеширующей функции передает строку «offln» и сравнивает результат с константами 0xFE81 и 0x12A9. Если результат не сходится, происходит работа с какой-то функцией timer() (может быть для ожидания плохого подключения донглы и нового запроса?), затем дешифрует еще одно сообщение об ошибке и выводит его.

```
.text:0000DA55 loc_DA55:                                ; CODE XREF: sync_sys+24Cj
.text:0000DA55      push    offset aOffln ; "offln"
.text:0000DA5A      call    SSQ
.text:0000DA5F      add     esp, 4
.text:0000DA62      mov     dl, [ebx]
.text:0000DA64      mov     esi, eax
.text:0000DA66      cmp     dl, 0Bh
.text:0000DA69      jnz     short loc_DA83
.text:0000DA6B      cmp     esi, 0FE81h
.text:0000DA71      jz      OK
.text:0000DA77      cmp     esi, 0FFFFFF8EFh
.text:0000DA7D      jz      OK
.text:0000DA83
.text:0000DA83 loc_DA83:                                ; CODE XREF: sync_sys+201j
.text:0000DA83      mov     cl, [ebx]
.text:0000DA85      cmp     cl, 0Ch
.text:0000DA88      jnz     short loc_DA9F
.text:0000DA8A      cmp     esi, 12A9h
.text:0000DA90      jz      OK
.text:0000DA96      cmp     esi, 0FFFFFFF5h
.text:0000DA99      jz      OK
.text:0000DA9F
.text:0000DA9F loc_DA9F:                                ; CODE XREF: sync_sys+220j
.text:0000DA9F      mov     eax, [ebp+var_18]
.text:0000DAA2      test    eax, eax
.text:0000DAA4      jz      short loc_DAB0
.text:0000DAA6      push    24h
.text:0000DAA8      call    timer
.text:0000DAAD      add     esp, 4
.text:0000DAB0
.text:0000DAB0 loc_DAB0:                                ; CODE XREF: sync_sys+23Cj
.text:0000DAB0      inc     edi
.text:0000DAB1      cmp     edi, 3
.text:0000DAB4      jle     short loc_DA55
```

78.2. ПРИМЕР #2: SCO OPENSERVR

```
.text:0000DAB6      mov     eax, ds:net_env
.text:0000DABB      test    eax, eax
.text:0000DABD      jz      short error

...

.text:0000DAF7 error:                                ; CODE XREF: sync_sys+255j
.text:0000DAF7                                ; sync_sys+274j ...
.text:0000DAF7      mov     [ebp+var_8], offset encrypted_error_message2
.text:0000DAFE      mov     [ebp+var_C], 17h ; decrypting key
.text:0000DB05      jmp     decrypt_end_print_message

...

; this name we gave to label:
.text:0000D9B6 decrypt_end_print_message:            ; CODE XREF: sync_sys+29Dj
.text:0000D9B6                                ; sync_sys+2ABj
.text:0000D9B6      mov     eax, [ebp+var_18]
.text:0000D9B9      test    eax, eax
.text:0000D9BB      jnz     short loc_D9FB
.text:0000D9BD      mov     edx, [ebp+var_C] ; key
.text:0000D9C0      mov     ecx, [ebp+var_8] ; string
.text:0000D9C3      push    edx
.text:0000D9C4      push    20h
.text:0000D9C6      push    ecx
.text:0000D9C7      push    18h
.text:0000D9C9      call   err_warn
.text:0000D9CE      push    0Fh
.text:0000D9D0      push    190h
.text:0000D9D5      call   sound
.text:0000D9DA      mov     [ebp+var_18], 1
.text:0000D9E1      add     esp, 18h
.text:0000D9E4      call   pc_v_kbhit
.text:0000D9E9      test    eax, eax
.text:0000D9EB      jz      short loc_D9FB

...

; this name we gave to label:
.data:00401736 encrypted_error_message2 db 74h, 72h, 78h, 43h, 48h, 6, 5Ah, 49h, 4Ch, 2 dup(47h)
.data:00401736      db 51h, 4Fh, 47h, 61h, 20h, 22h, 3Ch, 24h, 33h, 36h, 76h
.data:00401736      db 3Ah, 33h, 31h, 0Ch, 0, 0Bh, 1Fh, 7, 1Eh, 1Ah
```

Заставить работать программу без донглы довольно просто: просто пропатчить все места после инструкций **CMP** где происходят соответствующие сравнения.

Еще одна возможность — это написать свой драйвер для SCO OpenServer, содержащий таблицу возможных вопросов и ответов, все те что имеются в прогаме.

78.2.1. Дешифровка сообщений об ошибке

Кстати, мы также можем дешифровать все сообщения об ошибке. Алгоритм, находящийся в функции **err_warn()** действительно, крайне прост:

Листинг 78.1: Функция дешифровки

```
.text:0000A44D      mov     esi, [ebp+arg_C] ; key
.text:0000A450      mov     edx, [ebp+arg_4] ; string
.text:0000A453 loc_A453:
.text:0000A453      xor     eax, eax
.text:0000A455      mov     al, [edx+edi] ; загружаем байт для дешифровки
.text:0000A458      xor     eax, esi      ; дешифруем его
.text:0000A45A      add     esi, 3        ; изменяем ключ для следующего байта
.text:0000A45D      inc     edi
.text:0000A45E      cmp     edi, ecx
.text:0000A460      mov     [ebp+edi+var_55], al
.text:0000A464      jl      short loc_A453
```


78.2. ПРИМЕР #2: SCO OPENSERVR

Как видно, не только сама строка поступает на вход, но также и ключ для дешифровки:

```
.text:0000DAF7 error:                                ; CODE XREF: sync_sys+255j
.text:0000DAF7                                ; sync_sys+274j ...
.text:0000DAF7      mov      [ebp+var_8], offset encrypted_error_message2
.text:0000DAFE      mov      [ebp+var_C], 17h ; decrypting key
.text:0000DB05      jmp      decrypt_end_print_message

...

; this name we gave to label manually:
.text:0000D9B6 decrypt_end_print_message:          ; CODE XREF: sync_sys+29Dj
.text:0000D9B6                                ; sync_sys+2ABj
.text:0000D9B6      mov      eax, [ebp+var_18]
.text:0000D9B9      test     eax, eax
.text:0000D9BB      jnz      short loc_D9FB
.text:0000D9BD      mov      edx, [ebp+var_C] ; key
.text:0000D9C0      mov      ecx, [ebp+var_8] ; string
.text:0000D9C3      push     edx
.text:0000D9C4      push     20h
.text:0000D9C6      push     ecx
.text:0000D9C7      push     18h
.text:0000D9C9      call     err_warn
```

Алгоритм это очень простой **xoring**: каждый байт XOR-ится с ключом, но ключ увеличивается на 3 после обработки каждого байта.

Напишем небольшой скрипт на Python для проверки наших догадок:

Листинг 78.2: Python 3.x

```
#!/usr/bin/python
import sys

msg=[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A]

key=0x17
tmp=key
for i in msg:
    sys.stdout.write ("%c" % (i^tmp))
    tmp=tmp+3
sys.stdout.flush()
```

И он выводит: «check security device connection». Так что да, это дешифрованное сообщение.

Здесь есть также и другие сообщения, с соответствующими ключами. Но надо сказать, их можно дешифровать и без ключей. В начале, мы можем заметить, что ключ — это просто байт. Это потому что самая важная часть функции дешифровки (XOR) оперирует байтами. Ключ находится в регистре **ESI**, но только младшие 8 бит (т.е. байт) регистра используются. Следовательно, ключ может быть больше 255, но его значение будет округляться.

И как следствие, мы можем попробовать обычный перебор всех ключей в диапазоне 0..255. Мы также можем пропускать сообщения содержащие непечатные символы.

Листинг 78.3: Python 3.x

```
#!/usr/bin/python
import sys, curses.ascii

msgs=[
[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A],

[0x49, 0x65, 0x2D, 0x63, 0x76, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0x5C,
8, 0x4F, 0x4B, 0x47, 0x5D, 0x54, 0x5F, 0x1D, 0x26, 0x2C, 0x33,
0x27, 0x28, 0x6F, 0x72, 0x75, 0x78, 0x7B, 0x7E, 0x41, 0x44],

[0x45, 0x61, 0x31, 0x67, 0x72, 0x79, 0x68, 0x52, 0x4A, 0x52, 0x50,
0x0C, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x2B,
0x39, 0x39, 0x3C, 0x38, 0x79, 0x3A, 0x30, 0x17, 0x0B, 0x0C],
```

78.3. ПРИМЕР #3: MS-DOS

```
[0x40, 0x64, 0x79, 0x75, 0x7F, 0x6F, 0x0, 0x4C, 0x40, 0x9, 0x4D, 0x5A,
0x46, 0x5D, 0x57, 0x49, 0x57, 0x3B, 0x21, 0x23, 0x6A, 0x38, 0x23,
0x36, 0x24, 0x2A, 0x7C, 0x3A, 0x1A, 0x6, 0x0D, 0x0E, 0x0A, 0x14,
0x10],

[0x72, 0x7C, 0x72, 0x79, 0x76, 0x0,
0x50, 0x43, 0x4A, 0x59, 0x5D, 0x5B, 0x41, 0x41, 0x1B, 0x5A,
0x24, 0x32, 0x2E, 0x29, 0x28, 0x70, 0x20, 0x22, 0x38, 0x28, 0x36,
0x0D, 0x0B, 0x48, 0x4B, 0x4E]]

def is_string_printable(s):
    return all(list(map(lambda x: curses.ascii.isprint(x), s)))

cnt=1
for msg in msgs:
    print ("message #%d" % cnt)
    for key in range(0,256):
        result=[]
        tmp=key
        for i in msg:
            result.append (i^tmp)
            tmp=tmp+3
        if is_string_printable (result):
            print ("key=", key, "value=", "".join(list(map(chr, result))))
    cnt=cnt+1
```

И мы получим:

Листинг 78.4: Results

```
message #1
key= 20 value= `eb^h%|`hudw|_af{n~f%ljmSbnwlpk
key= 21 value= ajc]i"}cawtgv{^bgto}g"millcmvkqh
key= 22 value= bkd\j#rbbvsfuz!cdudh|d#bhomdlujni
key= 23 value= check security device connection
key= 24 value= lifbl!pd|tqhsx#ejwjbb!`nQofbshlo
message #2
key= 7 value= No security device found
key= 8 value= An#rbbvsVuz!cdudhd#ghtme?!#!'!#!
message #3
key= 7 value= Bk<waoqNUpu$`yrea\wmpusj,bkIjh
key= 8 value= Mj?vfnrOjqv%gxqd``_vwlstlk/clHii
key= 9 value= Lm>ugasLkvw&fgpgag^uvcwml.`mwhj
key= 10 value= Ol!td`tmhwx'efwfbf!tubuvnm!anvok
key= 11 value= No security device station found
key= 12 value= In#rjbvsnuz!{duhdd#r{`whho#gPtme
message #4
key= 14 value= Number of authorized users exceeded
key= 15 value= Ovlmdq!hg#`juknuhydk!vrbsp!Zy`dbefe
message #5
key= 17 value= check security device station
key= 18 value= `ijbh!td`tmhwx'efwfbf!tubuVnm!'!
```

Тут есть какой-то мусор, но мы можем быстро отыскать сообщения на английском языке!

Кстати, так как алгоритм использует простой **XOR**, та же функция может использоваться и для шифрования сообщения. Если нужно, мы можем зашифровать наши собственные сообщения, и пропатчить программу вставив их.

78.3. Пример #3: MS-DOS

Еще одна очень старая программа для MS-DOS от 1995 также разработанная давно исчезнувшей компанией .

Во времена перед DOS-экстендерами, всё ПО для MS-DOS рассчитывалось на процессоры 8086 или 80286, так что в своей массе весь код был 16-битным . 16-битный код в основном такой же, какой вы уже видели в этой книге, но все регистры 16-битные, и доступно меньше инструкций .

78.3. ПРИМЕР #3: MS-DOS

Среда MS-DOS не могла иметь никаких драйверов, и ПО работало с «голым» железом через порты, так что здесь вы можете увидеть инструкции `OUT / IN`, которые в наше время присутствуют в основном только в драйверах (в современных OS нельзя обращаться напрямую к портам из [user mode](#)).

Учитывая это, ПО для MS-DOS должно работать с донглой обращаясь к принтерному LPT-порту напрямую. Так что мы можем просто поискать эти инструкции. И да, вот они :

```
seg030:0034          out_port proc far ; CODE XREF: sent_pro+22p
seg030:0034                      ; sent_pro+2Ap ...
seg030:0034
seg030:0034          arg_0      = byte ptr 6
seg030:0034
seg030:0034 55          push     bp
seg030:0035 8B EC          mov     bp, sp
seg030:0037 8B 16 7E E7      mov     dx, _out_port ; 0x378
seg030:003B 8A 46 06          mov     al, [bp+arg_0]
seg030:003E EE          out     dx, al
seg030:003F 5D          pop     bp
seg030:0040 CB          retf
seg030:0040          out_port endp
```

(Все имена меток в этом примере даны мною).

Функция `out_port()` вызывается только из одной функции :

```
seg030:0041          sent_pro proc far ; CODE XREF: check_dongle+34p
seg030:0041
seg030:0041          var_3      = byte ptr -3
seg030:0041          var_2      = word ptr -2
seg030:0041          arg_0      = dword ptr 6
seg030:0041
seg030:0041 C8 04 00 00          enter    4, 0
seg030:0045 56          push     si
seg030:0046 57          push     di
seg030:0047 8B 16 82 E7      mov     dx, _in_port_1 ; 0x37A
seg030:004B EC          in     al, dx
seg030:004C 8A D8          mov     bl, al
seg030:004E 80 E3 FE          and     bl, 0FEh
seg030:0051 80 CB 04          or     bl, 4
seg030:0054 8A C3          mov     al, bl
seg030:0056 88 46 FD          mov     [bp+var_3], al
seg030:0059 80 E3 1F          and     bl, 1Fh
seg030:005C 8A C3          mov     al, bl
seg030:005E EE          out     dx, al
seg030:005F 68 FF 00          push    0FFh
seg030:0062 0E          push    cs
seg030:0063 E8 CE FF          call    near ptr out_port
seg030:0066 59          pop     cx
seg030:0067 68 D3 00          push    0D3h
seg030:006A 0E          push    cs
seg030:006B E8 C6 FF          call    near ptr out_port
seg030:006E 59          pop     cx
seg030:006F 33 F6          xor     si, si
seg030:0071 EB 01          jmp     short loc_359D4
seg030:0073
seg030:0073          loc_359D3: ; CODE XREF: sent_pro+37j
seg030:0073 46          inc     si
seg030:0074
seg030:0074          loc_359D4: ; CODE XREF: sent_pro+30j
seg030:0074 81 FE 96 00          cmp     si, 96h
seg030:0078 7C F9          jl     short loc_359D3
seg030:007A 68 C3 00          push    0C3h
seg030:007D 0E          push    cs
seg030:007E E8 B3 FF          call    near ptr out_port
seg030:0081 59          pop     cx
seg030:0082 68 C7 00          push    0C7h
seg030:0085 0E          push    cs
seg030:0086 E8 AB FF          call    near ptr out_port
seg030:0089 59          pop     cx
seg030:008A 68 D3 00          push    0D3h
seg030:008D 0E          push    cs
```

78.3. ПРИМЕР #3: MS-DOS

```

seg030:008E E8 A3 FF      call    near ptr out_port
seg030:0091 59          pop     cx
seg030:0092 68 C3 00      push    0C3h
seg030:0095 0E          push    cs
seg030:0096 E8 9B FF      call    near ptr out_port
seg030:0099 59          pop     cx
seg030:009A 68 C7 00      push    0C7h
seg030:009D 0E          push    cs
seg030:009E E8 93 FF      call    near ptr out_port
seg030:00A1 59          pop     cx
seg030:00A2 68 D3 00      push    0D3h
seg030:00A5 0E          push    cs
seg030:00A6 E8 8B FF      call    near ptr out_port
seg030:00A9 59          pop     cx
seg030:00AA BF FF FF      mov     di, 0FFFFh
seg030:00AD EB 40          jmp     short loc_35A4F
seg030:00AF
seg030:00AF loc_35A0F: ; CODE XREF: sent_pro+BDj
seg030:00AF BE 04 00      mov     si, 4
seg030:00B2
seg030:00B2 loc_35A12: ; CODE XREF: sent_pro+ACj
seg030:00B2 D1 E7          shl     di, 1
seg030:00B4 8B 16 80 E7      mov     dx, _in_port_2 ; 0x379
seg030:00B8 EC          in     al, dx
seg030:00B9 A8 80          test    al, 80h
seg030:00BB 75 03          jnz     short loc_35A20
seg030:00BD 83 CF 01      or      di, 1
seg030:00C0
seg030:00C0 loc_35A20: ; CODE XREF: sent_pro+7Aj
seg030:00C0 F7 46 FE 08+    test    [bp+var_2], 8
seg030:00C5 74 05          jz      short loc_35A2C
seg030:00C7 68 D7 00      push    0D7h ; '+'
seg030:00CA EB 0B          jmp     short loc_35A37
seg030:00CC
seg030:00CC loc_35A2C: ; CODE XREF: sent_pro+84j
seg030:00CC 68 C3 00      push    0C3h
seg030:00CF 0E          push    cs
seg030:00D0 E8 61 FF      call    near ptr out_port
seg030:00D3 59          pop     cx
seg030:00D4 68 C7 00      push    0C7h
seg030:00D7
seg030:00D7 loc_35A37: ; CODE XREF: sent_pro+89j
seg030:00D7 0E          push    cs
seg030:00D8 E8 59 FF      call    near ptr out_port
seg030:00DB 59          pop     cx
seg030:00DC 68 D3 00      push    0D3h
seg030:00DF 0E          push    cs
seg030:00E0 E8 51 FF      call    near ptr out_port
seg030:00E3 59          pop     cx
seg030:00E4 8B 46 FE      mov     ax, [bp+var_2]
seg030:00E7 D1 E0          shl     ax, 1
seg030:00E9 89 46 FE      mov     [bp+var_2], ax
seg030:00EC 4E          dec     si
seg030:00ED 75 C3          jnz     short loc_35A12
seg030:00EF
seg030:00EF loc_35A4F: ; CODE XREF: sent_pro+6Cj
seg030:00EF C4 5E 06      les     bx, [bp+arg_0]
seg030:00F2 FF 46 06      inc     word ptr [bp+arg_0]
seg030:00F5 26 8A 07      mov     al, es:[bx]
seg030:00F8 98          cbw
seg030:00F9 89 46 FE      mov     [bp+var_2], ax
seg030:00FC 0B C0          or      ax, ax
seg030:00FE 75 AF          jnz     short loc_35A0F
seg030:0100 68 FF 00      push    0FFh
seg030:0103 0E          push    cs
seg030:0104 E8 2D FF      call    near ptr out_port
seg030:0107 59          pop     cx
seg030:0108 8B 16 82 E7      mov     dx, _in_port_1 ; 0x37A
seg030:010C EC          in     al, dx
seg030:010D 8A C8          mov     cl, al

```

78.3. ПРИМЕР #3: MS-DOS

```

seg030:010F 80 E1 5F      and     cl, 5Fh
seg030:0112 8A C1        mov     al, cl
seg030:0114 EE          out     dx, al
seg030:0115 EC          in      al, dx
seg030:0116 8A C8        mov     cl, al
seg030:0118 F6 C1 20     test    cl, 20h
seg030:011B 74 08        jz      short loc_35A85
seg030:011D 8A 5E FD     mov     bl, [bp+var_3]
seg030:0120 80 E3 DF     and     bl, 0DFh
seg030:0123 EB 03        jmp     short loc_35A88
seg030:0125
seg030:0125             loc_35A85: ; CODE XREF: sent_pro+DAj
seg030:0125 8A 5E FD     mov     bl, [bp+var_3]
seg030:0128
seg030:0128             loc_35A88: ; CODE XREF: sent_pro+E2j
seg030:0128 F6 C1 80     test    cl, 80h
seg030:012B 74 03        jz      short loc_35A90
seg030:012D 80 E3 7F     and     bl, 7Fh
seg030:0130
seg030:0130             loc_35A90: ; CODE XREF: sent_pro+EAj
seg030:0130 8B 16 82 E7     mov     dx, _in_port_1 ; 0x37A
seg030:0134 8A C3        mov     al, bl
seg030:0136 EE          out     dx, al
seg030:0137 8B C7        mov     ax, di
seg030:0139 5F          pop     di
seg030:013A 5E          pop     si
seg030:013B C9          leave
seg030:013C CB          retf
seg030:013C             sent_pro endp

```

Это также «хеширующая» донгла Sentinel Pro как и в предыдущем примере . Это заметно по тому что текстовые строки передаются и здесь, 16-битные значения также возвращаются и сравниваются с другими.

Так вот как происходит работа с Sentinel Pro через порты . Адрес выходного порта обычно 0x378, т.е. принтерного порта, данные для него во времена перед USB отправлялись прямо сюда . Порт однонаправленный, потому что когда его разрабатывали, никто не мог предположить, что кому-то понадобится получать информацию из принтера ⁷ . Единственный способ получить информацию из принтера это регистр статуса на порту 0x379, он содержит такие биты как «paper out», «ask», «busy» — так принтер может сигнализировать о том, что он готов или нет, и о том, есть ли в нем бумага . Так что донгла возвращает информацию через какой-то из этих бит, по одному биту на каждой итерации .

`_in_port_2` содержит адрес статуса (0x379) и `_in_port_1` содержит адрес управляющего регистра (0x37A).

Судя по всему, донгла возвращает информацию только через флаг «busy» на `seg030:00B9` : каждый бит записывается в регистре `DI` позже возвращаемый в самом конце функции .

Что означают все эти отсылаемые в выходной порт байты? Трудно сказать. Возможно, команды донглы. Но честно говоря, нам и не обязательно знать: нашу задачу можно легко решить и без этих знаний .

Вот функция проверки донглы:

```

00000000 struct_0      struc ; (sizeof=0x1B)
00000000 field_0        db 25 dup(?)          ; string(C)
00000019 _A             dw ?
0000001B struct_0      ends

dseg:3CBC 61 63 72 75+_Q struct_0 <'hello', 01122h>
dseg:3CBC 6E 00 00 00+   ; DATA XREF: check_dongle+2E0

... skipped ...

dseg:3E00 63 6F 66 66+   struct_0 <'coffee', 7EB7h>
dseg:3E1B 64 6F 67 00+   struct_0 <'dog', 0FFADh>
dseg:3E36 63 61 74 00+   struct_0 <'cat', 0FF5Fh>
dseg:3E51 70 61 70 65+   struct_0 <'paper', 0FFDFh>
dseg:3E6C 63 6F 6B 65+   struct_0 <'coke', 0F568h>
dseg:3E87 63 6C 6F 63+   struct_0 <'clock', 55EAh>
dseg:3EA2 64 69 72 00+   struct_0 <'dir', 0FFAEh>
dseg:3EBD 63 6F 70 79+   struct_0 <'copy', 0F557h>

```

⁷Если учитывать только Centronics и не учитывать последующий стандарт IEEE 1284 — в нем из принтера можно получать информацию .

```

seg030:0145          check_dongle proc far ; CODE XREF: sub_3771D+3EP
seg030:0145
seg030:0145          var_6 = dword ptr -6
seg030:0145          var_2 = word ptr -2
seg030:0145
seg030:0145 C8 06 00 00      enter    6, 0
seg030:0149 56              push     si
seg030:014A 66 6A 00      push     large 0          ; newtime
seg030:014D 6A 00      push     0              ; cmd
seg030:014F 9A C1 18 00+    call     _biostime
seg030:0154 52              push     dx
seg030:0155 50              push     ax
seg030:0156 66 58      pop      eax
seg030:0158 83 C4 06      add     sp, 6
seg030:015B 66 89 46 FA      mov     [bp+var_6], eax
seg030:015F 66 3B 06 D8+    cmp     eax, _expiration
seg030:0164 7E 44      jle     short loc_35B0A
seg030:0166 6A 14      push     14h
seg030:0168 90              nop
seg030:0169 0E              push     cs
seg030:016A E8 52 00      call    near ptr get_rand
seg030:016D 59              pop      cx
seg030:016E 8B F0      mov     si, ax
seg030:0170 6B C0 1B      imul    ax, 1Bh
seg030:0173 05 BC 3C      add     ax, offset _Q
seg030:0176 1E              push     ds
seg030:0177 50              push     ax
seg030:0178 0E              push     cs
seg030:0179 E8 C5 FE      call    near ptr sent_pro
seg030:017C 83 C4 04      add     sp, 4
seg030:017F 89 46 FE      mov     [bp+var_2], ax
seg030:0182 8B C6      mov     ax, si
seg030:0184 6B C0 12      imul    ax, 18
seg030:0187 66 0F BF C0      movsx   eax, ax
seg030:018B 66 8B 56 FA      mov     edx, [bp+var_6]
seg030:018F 66 03 D0      add     edx, eax
seg030:0192 66 89 16 D8+    mov     _expiration, edx
seg030:0197 8B DE      mov     bx, si
seg030:0199 6B DB 1B      imul    bx, 27
seg030:019C 8B 87 D5 3C      mov     ax, _Q._A[bx]
seg030:01A0 3B 46 FE      cmp     ax, [bp+var_2]
seg030:01A3 74 05      jz      short loc_35B0A
seg030:01A5 B8 01 00      mov     ax, 1
seg030:01A8 EB 02      jmp     short loc_35B0C
seg030:01AA
seg030:01AA          loc_35B0A: ; CODE XREF: check_dongle+1Fj
seg030:01AA                      ; check_dongle+5Ej
seg030:01AA 33 C0      xor     ax, ax
seg030:01AC
seg030:01AC          loc_35B0C: ; CODE XREF: check_dongle+63j
seg030:01AC 5E              pop      si
seg030:01AD C9              leave
seg030:01AE CB              retf
seg030:01AE          check_dongle endp

```

А так как эта функция может вызываться слишком часто, например, перед выполнением каждой важной возможности ПО, а обращение к донгле вообще-то медленное (и из-за медленного принтерного порта, и из-за медленного **MCU** в донгле), так что они, наверное, добавили возможность пропускать проверку донглы слишком часто, используя текущее время в функции `biostime()` .

Функция `get_rand()` использует стандартную функцию Си :

```

seg030:01BF          get_rand proc far ; CODE XREF: check_dongle+25p
seg030:01BF
seg030:01BF          arg_0      = word ptr 6
seg030:01BF
seg030:01BF 55              push     bp
seg030:01C0 8B EC      mov     bp, sp

```

78.3. ПРИМЕР #3: MS-DOS

```
seg030:01C2 9A 3D 21 00+    call    _rand
seg030:01C7 66 0F BF C0      movsx   eax, ax
seg030:01CB 66 0F BF 56+    movsx   edx, [bp+arg_0]
seg030:01D0 66 0F AF C2      imul    eax, edx
seg030:01D4 66 BB 00 80+    mov     ebx, 8000h
seg030:01DA 66 99          cdq
seg030:01DC 66 F7 FB      idiv    ebx
seg030:01DF 5D          pop     bp
seg030:01E0 CB          retf
seg030:01E0                get_rand endp
```

Так что текстовая строка выбирается случайно, отправляется в донглу и результат хеширования сверяется с корректным значением .

Текстовые строки, похоже, составлялись так же случайно, во время разработки ПО.

И вот как вызывается главная процедура проверки донглы :

```
seg033:087B 9A 45 01 96+    call    check_dongle
seg033:0880 0B C0          or      ax, ax
seg033:0882 74 62          jz      short OK
seg033:0884 83 3E 60 42+    cmp     word_620E0, 0
seg033:0889 75 5B          jnz     short OK
seg033:088B FF 06 60 42      inc     word_620E0
seg033:088F 1E          push    ds
seg033:0890 68 22 44      push    offset aTrupcRequiresA ; "This Software Requires a Software Lock\n"
seg033:0893 1E          push    ds
seg033:0894 68 60 E9      push    offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+    call    _strcpy
seg033:089C 83 C4 08      add     sp, 8
seg033:089F 1E          push    ds
seg033:08A0 68 42 44      push    offset aPleaseContactA ; "Please Contact ..."
seg033:08A3 1E          push    ds
seg033:08A4 68 60 E9      push    offset byte_6C7E0 ; dest
seg033:08A7 9A CD 64 00+    call    _strcat
```

Заставить работать программу без донглы очень просто: просто заставить функцию `check_dongle()` возвращать всегда 0 .

Например, вставив такой код в самом её начале:

```
mov ax,0
retf
```

Наблюдательный читатель может заметить, что функция Си `strcpy()` имеет 2 аргумента, но здесь мы видим, что передается 4 :

```
seg033:088F 1E          push    ds
seg033:0890 68 22 44      push    offset aTrupcRequiresA ; "This Software Requires a Software Lock\n"
seg033:0893 1E          push    ds
seg033:0894 68 60 E9      push    offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+    call    _strcpy
seg033:089C 83 C4 08      add     sp, 8
```

Это связано с моделью памяти в MS-DOS. Об этом больше читайте здесь : [96](#) (стр. 898).

Так что, `strcpy()` , как и любая другая функция принимающая указатель (-и) в аргументах, работает с 16-битными парами .

Вернемся к нашему примеру. `DS` сейчас указывает на сегмент данных размещенный в исполняемом файле, там, где хранится текстовая строка.

В функции `sent_pro()` каждый байт строки загружается на `seg030:00EF` : инструкция `LES` загружает из переданного аргумента пару ES:BX одновременно . `MOV` на `seg030:00F5` загружает байт из памяти, на который указывает пара ES:BX.

Глава 79

«QR9»: Любительская криптосистема, вдохновленная кубиком Рубика

Любительские криптосистемы иногда встречаются довольно странные.

Однажды автора сих строк попросили разобраться с одним таким любительским криптоалгоритмом встроенным в утилиту для шифрования, исходный код которой был утерян¹.

Вот листинг этой утилиты для шифрования, полученный при помощи [IDA](#):

```
.text:00541000 set_bit      proc near          ; CODE XREF: rotate1+42
.text:00541000                                     ; rotate2+42 ...
.text:00541000
.text:00541000 arg_0      = dword ptr  4
.text:00541000 arg_4      = dword ptr  8
.text:00541000 arg_8      = dword ptr  0Ch
.text:00541000 arg_C      = byte ptr  10h
.text:00541000
.text:00541000      mov     al, [esp+arg_C]
.text:00541004      mov     ecx, [esp+arg_8]
.text:00541008      push    esi
.text:00541009      mov     esi, [esp+4+arg_0]
.text:0054100D      test    al, al
.text:0054100F      mov     eax, [esp+4+arg_4]
.text:00541013      mov     dl, 1
.text:00541015      jz      short loc_54102B
.text:00541017      shl     dl, cl
.text:00541019      mov     cl, cube64[eax+esi*8]
.text:00541020      or      cl, dl
.text:00541022      mov     cube64[eax+esi*8], cl
.text:00541029      pop     esi
.text:0054102A      retn
.text:0054102B loc_54102B:                                     ; CODE XREF: set_bit+15
.text:0054102B      shl     dl, cl
.text:0054102D      mov     cl, cube64[eax+esi*8]
.text:00541034      not     dl
.text:00541036      and     cl, dl
.text:00541038      mov     cube64[eax+esi*8], cl
.text:0054103F      pop     esi
.text:00541040      retn
.text:00541040 set_bit      endp
.text:00541040
.text:00541041      align 10h
.text:00541050
.text:00541050 ; ===== S U B R O U T I N E =====
.text:00541050
.text:00541050
.text:00541050 get_bit      proc near          ; CODE XREF: rotate1+16
.text:00541050                                     ; rotate2+16 ...
.text:00541050
.text:00541050 arg_0      = dword ptr  4
```

¹Он также получил разрешение от клиента на публикацию деталей алгоритма


```

.text:00541050 arg_4      = dword ptr 8
.text:00541050 arg_8      = byte ptr 0Ch
.text:00541050
.text:00541050      mov     eax, [esp+arg_4]
.text:00541054      mov     ecx, [esp+arg_0]
.text:00541058      mov     al, cube64[eax+ecx*8]
.text:0054105F      mov     cl, [esp+arg_8]
.text:00541063      shr     al, cl
.text:00541065      and     al, 1
.text:00541067      retn
.text:00541067 get_bit      endp
.text:00541067
.text:00541068      align 10h
.text:00541070
.text:00541070 ; ===== S U B R O U T I N E =====
.text:00541070
.text:00541070
.text:00541070 rotate1      proc near          ; CODE XREF: rotate_all_with_password+8E
.text:00541070
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0      = dword ptr 4
.text:00541070
.text:00541070      sub     esp, 40h
.text:00541073      push    ebx
.text:00541074      push    ebp
.text:00541075      mov     ebp, [esp+48h+arg_0]
.text:00541079      push    esi
.text:0054107A      push    edi
.text:0054107B      xor     edi, edi          ; EDI is loop1 counter
.text:0054107D      lea     ebx, [esp+50h+internal_array_64]
.text:00541081
.text:00541081 first_loop1_begin:          ; CODE XREF: rotate1+2E
.text:00541081      xor     esi, esi          ; ESI is loop2 counter
.text:00541083
.text:00541083 first_loop2_begin:          ; CODE XREF: rotate1+25
.text:00541083      push    ebp          ; arg_0
.text:00541084      push    esi
.text:00541085      push    edi
.text:00541086      call    get_bit
.text:0054108B      add     esp, 0Ch
.text:0054108E      mov     [ebx+esi], al      ; store to internal array
.text:00541091      inc     esi
.text:00541092      cmp     esi, 8
.text:00541095      jnl     short first_loop2_begin
.text:00541097      inc     edi
.text:00541098      add     ebx, 8
.text:0054109B      cmp     edi, 8
.text:0054109E      jnl     short first_loop1_begin
.text:005410A0      lea     ebx, [esp+50h+internal_array_64]
.text:005410A4      mov     edi, 7          ; EDI is loop1 counter, initial state is 7
.text:005410A9
.text:005410A9 second_loop1_begin:          ; CODE XREF: rotate1+57
.text:005410A9      xor     esi, esi          ; ESI is loop2 counter
.text:005410AB
.text:005410AB second_loop2_begin:          ; CODE XREF: rotate1+4E
.text:005410AB      mov     al, [ebx+esi]      ; value from internal array
.text:005410AE      push    eax
.text:005410AF      push    ebp          ; arg_0
.text:005410B0      push    edi
.text:005410B1      push    esi
.text:005410B2      call    set_bit
.text:005410B7      add     esp, 10h
.text:005410BA      inc     esi          ; increment loop2 counter
.text:005410BB      cmp     esi, 8
.text:005410BE      jnl     short second_loop2_begin
.text:005410C0      dec     edi          ; decrement loop2 counter
.text:005410C1      add     ebx, 8
.text:005410C4      cmp     edi, 0FFFFFFFh
.text:005410C7      jg      short second_loop1_begin
.text:005410C9      pop     edi

```

```

.text:005410CA      pop     esi
.text:005410CB      pop     ebp
.text:005410CC      pop     ebx
.text:005410CD      add     esp, 40h
.text:005410D0      retn
.text:005410D0 rotate1      endp
.text:005410D0
.text:005410D1      align 10h
.text:005410E0
.text:005410E0 ; ===== S U B R O U T I N E =====
.text:005410E0
.text:005410E0
.text:005410E0 rotate2      proc near      ; CODE XREF: rotate_all_with_password+7A
.text:005410E0
.text:005410E0 internal_array_64= byte ptr -40h
.text:005410E0 arg_0      = dword ptr 4
.text:005410E0
.text:005410E0      sub     esp, 40h
.text:005410E3      push    ebx
.text:005410E4      push    ebp
.text:005410E5      mov     ebp, [esp+48h+arg_0]
.text:005410E9      push    esi
.text:005410EA      push    edi
.text:005410EB      xor     edi, edi      ; loop1 counter
.text:005410ED      lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:      ; CODE XREF: rotate2+2E
.text:005410F1      xor     esi, esi      ; loop2 counter
.text:005410F3
.text:005410F3 loc_5410F3:      ; CODE XREF: rotate2+25
.text:005410F3      push    esi      ; loop2
.text:005410F4      push    edi      ; loop1
.text:005410F5      push    ebp      ; arg_0
.text:005410F6      call    get_bit
.text:005410FB      add     esp, 0Ch
.text:005410FE      mov     [ebx+esi], al ; store to internal array
.text:00541101      inc     esi      ; increment loop1 counter
.text:00541102      cmp     esi, 8
.text:00541105      jnl     short loc_5410F3
.text:00541107      inc     edi      ; increment loop2 counter
.text:00541108      add     ebx, 8
.text:0054110B      cmp     edi, 8
.text:0054110E      jnl     short loc_5410F1
.text:00541110      lea     ebx, [esp+50h+internal_array_64]
.text:00541114      mov     edi, 7      ; loop1 counter is initial state 7
.text:00541119
.text:00541119 loc_541119:      ; CODE XREF: rotate2+57
.text:00541119      xor     esi, esi      ; loop2 counter
.text:0054111B
.text:0054111B loc_54111B:      ; CODE XREF: rotate2+4E
.text:0054111B      mov     al, [ebx+esi] ; get byte from internal array
.text:0054111E      push    eax
.text:0054111F      push    edi      ; loop1 counter
.text:00541120      push    esi      ; loop2 counter
.text:00541121      push    ebp      ; arg_0
.text:00541122      call    set_bit
.text:00541127      add     esp, 10h
.text:0054112A      inc     esi      ; increment loop2 counter
.text:0054112B      cmp     esi, 8
.text:0054112E      jnl     short loc_54111B
.text:00541130      dec     edi      ; decrement loop2 counter
.text:00541131      add     ebx, 8
.text:00541134      cmp     edi, 0FFFFFFFh
.text:00541137      jg     short loc_541119
.text:00541139      pop     edi
.text:0054113A      pop     esi
.text:0054113B      pop     ebp
.text:0054113C      pop     ebx
.text:0054113D      add     esp, 40h
.text:00541140      retn

```

```

.text:00541140 rotate2      endp
.text:00541140
.text:00541141          align 10h
.text:00541150
.text:00541150 ; ===== S U B R O U T I N E =====
.text:00541150
.text:00541150 rotate3      proc near          ; CODE XREF: rotate_all_with_password+66
.text:00541150
.text:00541150 var_40      = byte ptr -40h
.text:00541150 arg_0      = dword ptr  4
.text:00541150
.text:00541150          sub     esp, 40h
.text:00541153          push    ebx
.text:00541154          push    ebp
.text:00541155          mov     ebp, [esp+48h+arg_0]
.text:00541159          push    esi
.text:0054115A          push    edi
.text:0054115B          xor     edi, edi
.text:0054115D          lea     ebx, [esp+50h+var_40]
.text:00541161
.text:00541161 loc_541161:          ; CODE XREF: rotate3+2E
.text:00541161          xor     esi, esi
.text:00541163
.text:00541163 loc_541163:          ; CODE XREF: rotate3+25
.text:00541163          push    esi
.text:00541164          push    ebp
.text:00541165          push    edi
.text:00541166          call    get_bit
.text:0054116B          add     esp, 0Ch
.text:0054116E          mov     [ebx+esi], al
.text:00541171          inc     esi
.text:00541172          cmp     esi, 8
.text:00541175          jnl     short loc_541163
.text:00541177          inc     edi
.text:00541178          add     ebx, 8
.text:0054117B          cmp     edi, 8
.text:0054117E          jnl     short loc_541161
.text:00541180          xor     ebx, ebx
.text:00541182          lea     edi, [esp+50h+var_40]
.text:00541186
.text:00541186 loc_541186:          ; CODE XREF: rotate3+54
.text:00541186          mov     esi, 7
.text:0054118B
.text:0054118B loc_54118B:          ; CODE XREF: rotate3+4E
.text:0054118B          mov     al, [edi]
.text:0054118D          push    eax
.text:0054118E          push    ebx
.text:0054118F          push    ebp
.text:00541190          push    esi
.text:00541191          call    set_bit
.text:00541196          add     esp, 10h
.text:00541199          inc     edi
.text:0054119A          dec     esi
.text:0054119B          cmp     esi, 0FFFFFFFh
.text:0054119E          jg      short loc_54118B
.text:005411A0          inc     ebx
.text:005411A1          cmp     ebx, 8
.text:005411A4          jnl     short loc_541186
.text:005411A6          pop     edi
.text:005411A7          pop     esi
.text:005411A8          pop     ebp
.text:005411A9          pop     ebx
.text:005411AA          add     esp, 40h
.text:005411AD          retn
.text:005411AD rotate3      endp
.text:005411AD
.text:005411AE          align 10h
.text:005411B0
.text:005411B0 ; ===== S U B R O U T I N E =====

```

```

.text:005411B0
.text:005411B0
.text:005411B0 rotate_all_with_password proc near      ; CODE XREF: crypt+1F
.text:005411B0                                     ; decrypt+36
.text:005411B0
.text:005411B0 arg_0          = dword ptr 4
.text:005411B0 arg_4          = dword ptr 8
.text:005411B0
.text:005411B0          mov     eax, [esp+arg_0]
.text:005411B4          push    ebp
.text:005411B5          mov     ebp, eax
.text:005411B7          cmp     byte ptr [eax], 0
.text:005411BA          jz      exit
.text:005411C0          push    ebx
.text:005411C1          mov     ebx, [esp+8+arg_4]
.text:005411C5          push    esi
.text:005411C6          push    edi
.text:005411C7
.text:005411C7 loop_begin:                                     ; CODE XREF: rotate_all_with_password+9F
.text:005411C7          movsx   eax, byte ptr [ebp+0]
.text:005411CB          push    eax                                     ; C
.text:005411CC          call   _tolower
.text:005411D1          add     esp, 4
.text:005411D4          cmp     al, 'a'
.text:005411D6          jl      short next_character_in_password
.text:005411D8          cmp     al, 'z'
.text:005411DA          jg      short next_character_in_password
.text:005411DC          movsx   ecx, al
.text:005411DF          sub     ecx, 'a'
.text:005411E2          cmp     ecx, 24
.text:005411E5          jle     short skip_subtracting
.text:005411E7          sub     ecx, 24
.text:005411EA
.text:005411EA skip_subtracting:                               ; CODE XREF: rotate_all_with_password+35
.text:005411EA          mov     eax, 55555556h
.text:005411EF          imul   ecx
.text:005411F1          mov     eax, edx
.text:005411F3          shr     eax, 1Fh
.text:005411F6          add     edx, eax
.text:005411F8          mov     eax, ecx
.text:005411FA          mov     esi, edx
.text:005411FC          mov     ecx, 3
.text:00541201          cdq
.text:00541202          idiv   ecx
.text:00541204          sub     edx, 0
.text:00541207          jz      short call_rotate1
.text:00541209          dec     edx
.text:0054120A          jz      short call_rotate2
.text:0054120C          dec     edx
.text:0054120D          jnz     short next_character_in_password
.text:0054120F          test    ebx, ebx
.text:00541211          jle     short next_character_in_password
.text:00541213          mov     edi, ebx
.text:00541215
.text:00541215 call_rotate3:                                     ; CODE XREF: rotate_all_with_password+6F
.text:00541215          push    esi
.text:00541216          call   rotate3
.text:0054121B          add     esp, 4
.text:0054121E          dec     edi
.text:0054121F          jnz     short call_rotate3
.text:00541221          jmp     short next_character_in_password
.text:00541223
.text:00541223 call_rotate2:                                     ; CODE XREF: rotate_all_with_password+5A
.text:00541223          test    ebx, ebx
.text:00541225          jle     short next_character_in_password
.text:00541227          mov     edi, ebx
.text:00541229
.text:00541229 loc_541229:                                     ; CODE XREF: rotate_all_with_password+83
.text:00541229          push    esi
.text:0054122A          call   rotate2

```

```

.text:0054122F      add     esp, 4
.text:00541232      dec     edi
.text:00541233      jnz     short loc_541229
.text:00541235      jmp     short next_character_in_password
.text:00541237
.text:00541237 call_rotate1:      ; CODE XREF: rotate_all_with_password+57
.text:00541237      test    ebx, ebx
.text:00541239      jle     short next_character_in_password
.text:0054123B      mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:      ; CODE XREF: rotate_all_with_password+97
.text:0054123D      push    esi
.text:0054123E      call    rotate1
.text:00541243      add     esp, 4
.text:00541246      dec     edi
.text:00541247      jnz     short loc_54123D
.text:00541249
.text:00541249 next_character_in_password:      ; CODE XREF: rotate_all_with_password+26
.text:00541249      ; rotate_all_with_password+2A ...
.text:00541249      mov     al, [ebp+1]
.text:0054124C      inc     ebp
.text:0054124D      test    al, al
.text:0054124F      jnz     loop_begin
.text:00541255      pop     edi
.text:00541256      pop     esi
.text:00541257      pop     ebx
.text:00541258
.text:00541258 exit:      ; CODE XREF: rotate_all_with_password+A
.text:00541258      pop     ebp
.text:00541259      retn
.text:00541259 rotate_all_with_password endp
.text:00541259
.text:0054125A      align 10h
.text:00541260
.text:00541260 ; ===== S U B R O U T I N E =====
.text:00541260
.text:00541260
.text:00541260 crypt      proc near      ; CODE XREF: crypt_file+8A
.text:00541260
.text:00541260 arg_0      = dword ptr 4
.text:00541260 arg_4      = dword ptr 8
.text:00541260 arg_8      = dword ptr 0Ch
.text:00541260
.text:00541260      push    ebx
.text:00541261      mov     ebx, [esp+4+arg_0]
.text:00541265      push    ebp
.text:00541266      push    esi
.text:00541267      push    edi
.text:00541268      xor     ebp, ebp
.text:0054126A
.text:0054126A loc_54126A:      ; CODE XREF: crypt+41
.text:0054126A      mov     eax, [esp+10h+arg_8]
.text:0054126E      mov     ecx, 10h
.text:00541273      mov     esi, ebx
.text:00541275      mov     edi, offset cube64
.text:0054127A      push    1
.text:0054127C      push    eax
.text:0054127D      rep movsd
.text:0054127F      call    rotate_all_with_password
.text:00541284      mov     eax, [esp+18h+arg_4]
.text:00541288      mov     edi, ebx
.text:0054128A      add     ebp, 40h
.text:0054128D      add     esp, 8
.text:00541290      mov     ecx, 10h
.text:00541295      mov     esi, offset cube64
.text:0054129A      add     ebx, 40h
.text:0054129D      cmp     ebp, eax
.text:0054129F      rep movsd
.text:005412A1      jl      short loc_54126A
.text:005412A3      pop     edi

```

```

.text:005412A4      pop     esi
.text:005412A5      pop     ebp
.text:005412A6      pop     ebx
.text:005412A7      retn
.text:005412A7 crypt      endp
.text:005412A7
.text:005412A8      align 10h
.text:005412B0
.text:005412B0 ; ===== S U B R O U T I N E =====
.text:005412B0
.text:005412B0
.text:005412B0 ; int __cdecl decrypt(int, int, void *Src)
.text:005412B0 decrypt      proc near      ; CODE XREF: decrypt_file+99
.text:005412B0
.text:005412B0 arg_0          = dword ptr 4
.text:005412B0 arg_4          = dword ptr 8
.text:005412B0 Src          = dword ptr 0Ch
.text:005412B0
.text:005412B0      mov     eax, [esp+Src]
.text:005412B4      push    ebx
.text:005412B5      push    ebp
.text:005412B6      push    esi
.text:005412B7      push    edi
.text:005412B8      push    eax      ; Src
.text:005412B9      call    __strdup
.text:005412BE      push    eax      ; Str
.text:005412BF      mov     [esp+18h+Src], eax
.text:005412C3      call    __strrev
.text:005412C8      mov     ebx, [esp+18h+arg_0]
.text:005412CC      add     esp, 8
.text:005412CF      xor     ebp, ebp
.text:005412D1
.text:005412D1 loc_5412D1:      ; CODE XREF: decrypt+58
.text:005412D1      mov     ecx, 10h
.text:005412D6      mov     esi, ebx
.text:005412D8      mov     edi, offset cube64
.text:005412DD      push    3
.text:005412DF      rep movsd
.text:005412E1      mov     ecx, [esp+14h+Src]
.text:005412E5      push    ecx
.text:005412E6      call    rotate_all_with_password
.text:005412EB      mov     eax, [esp+18h+arg_4]
.text:005412EF      mov     edi, ebx
.text:005412F1      add     ebp, 40h
.text:005412F4      add     esp, 8
.text:005412F7      mov     ecx, 10h
.text:005412FC      mov     esi, offset cube64
.text:00541301      add     ebx, 40h
.text:00541304      cmp     ebp, eax
.text:00541306      rep movsd
.text:00541308      jnl     short loc_5412D1
.text:0054130A      mov     edx, [esp+10h+Src]
.text:0054130E      push    edx      ; Memory
.text:0054130F      call    _free
.text:00541314      add     esp, 4
.text:00541317      pop     edi
.text:00541318      pop     esi
.text:00541319      pop     ebp
.text:0054131A      pop     ebx
.text:0054131B      retn
.text:0054131B decrypt      endp
.text:0054131B
.text:0054131C      align 10h
.text:00541320
.text:00541320 ; ===== S U B R O U T I N E =====
.text:00541320
.text:00541320
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file      proc near      ; CODE XREF: _main+42
.text:00541320

```

```

.text:00541320 Str          = dword ptr 4
.text:00541320 Filename    = dword ptr 8
.text:00541320 password    = dword ptr 0Ch
.text:00541320
.text:00541320          mov     eax, [esp+Str]
.text:00541324          push    ebp
.text:00541325          push    offset Mode      ; "rb"
.text:0054132A          push    eax              ; Filename
.text:0054132B          call    _fopen           ; open file
.text:00541330          mov     ebp, eax
.text:00541332          add     esp, 8
.text:00541335          test    ebp, ebp
.text:00541337          jnz     short loc_541348
.text:00541339          push    offset Format      ; "Cannot open input file!\n"
.text:0054133E          call    _printf
.text:00541343          add     esp, 4
.text:00541346          pop     ebp
.text:00541347          retn
.text:00541348
.text:00541348 loc_541348:          ; CODE XREF: crypt_file+17
.text:00541348          push    ebx
.text:00541349          push    esi
.text:0054134A          push    edi
.text:0054134B          push    2              ; Origin
.text:0054134D          push    0              ; Offset
.text:0054134F          push    ebp              ; File
.text:00541350          call    _fseek
.text:00541355          push    ebp              ; File
.text:00541356          call    _ftell         ; get file size
.text:0054135B          push    0              ; Origin
.text:0054135D          push    0              ; Offset
.text:0054135F          push    ebp              ; File
.text:00541360          mov     [esp+2Ch+Str], eax
.text:00541364          call    _fseek         ; rewind to start
.text:00541369          mov     esi, [esp+2Ch+Str]
.text:0054136D          and     esi, 0FFFFFFC0h ; reset all lowest 6 bits
.text:00541370          add     esi, 40h       ; align size to 64-byte border
.text:00541373          push    esi              ; Size
.text:00541374          call    _malloc
.text:00541379          mov     ecx, esi
.text:0054137B          mov     ebx, eax         ; allocated buffer pointer -> to EBX
.text:0054137D          mov     edx, ecx
.text:0054137F          xor     eax, eax
.text:00541381          mov     edi, ebx
.text:00541383          push    ebp              ; File
.text:00541384          shr     ecx, 2
.text:00541387          rep stosd
.text:00541389          mov     ecx, edx
.text:0054138B          push    1              ; Count
.text:0054138D          and     ecx, 3
.text:00541390          rep stosb              ; memset (buffer, 0, aligned_size)
.text:00541392          mov     eax, [esp+38h+Str]
.text:00541396          push    eax              ; ElementSize
.text:00541397          push    ebx              ; DstBuf
.text:00541398          call    _fread         ; read file
.text:0054139D          push    ebp              ; File
.text:0054139E          call    _fclose
.text:005413A3          mov     ecx, [esp+44h+password]
.text:005413A7          push    ecx              ; password
.text:005413A8          push    esi              ; aligned size
.text:005413A9          push    ebx              ; buffer
.text:005413AA          call    crypt            ; do crypt
.text:005413AF          mov     edx, [esp+50h+Filename]
.text:005413B3          add     esp, 40h
.text:005413B6          push    offset aWb       ; "wb"
.text:005413BB          push    edx              ; Filename
.text:005413BC          call    _fopen
.text:005413C1          mov     edi, eax
.text:005413C3          push    edi              ; File
.text:005413C4          push    1              ; Count

```

```

.text:005413C6      push     3                ; Size
.text:005413C8      push     offset aQr9      ; "QR9"
.text:005413CD      call    _fwrite          ; write file signature
.text:005413D2      push     edi              ; File
.text:005413D3      push     1               ; Count
.text:005413D5      lea     eax, [esp+30h+Str]
.text:005413D9      push     4               ; Size
.text:005413DB      push     eax              ; Str
.text:005413DC      call    _fwrite          ; write original file size
.text:005413E1      push     edi              ; File
.text:005413E2      push     1               ; Count
.text:005413E4      push     esi              ; Size
.text:005413E5      push     ebx              ; Str
.text:005413E6      call    _fwrite          ; write encrypted file
.text:005413EB      push     edi              ; File
.text:005413EC      call    _fclose
.text:005413F1      push     ebx              ; Memory
.text:005413F2      call    _free
.text:005413F7      add     esp, 40h
.text:005413FA      pop     edi
.text:005413FB      pop     esi
.text:005413FC      pop     ebx
.text:005413FD      pop     ebp
.text:005413FE      retn
.text:005413FE crypt_file      endp
.text:005413FE
.text:005413FF      align 10h
.text:00541400
.text:00541400 ; ===== S U B R O U T I N E =====
.text:00541400
.text:00541400
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)
.text:00541400 decrypt_file      proc near                ; CODE XREF: _main+6E
.text:00541400
.text:00541400 Filename      = dword ptr 4
.text:00541400 arg_4        = dword ptr 8
.text:00541400 Src          = dword ptr 0Ch
.text:00541400
.text:00541400      mov     eax, [esp+Filename]
.text:00541404      push     ebx
.text:00541405      push     ebp
.text:00541406      push     esi
.text:00541407      push     edi
.text:00541408      push     offset aRb      ; "rb"
.text:0054140D      push     eax              ; Filename
.text:0054140E      call    _fopen
.text:00541413      mov     esi, eax
.text:00541415      add     esp, 8
.text:00541418      test    esi, esi
.text:0054141A      jnz     short loc_54142E
.text:0054141C      push     offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421      call    _printf
.text:00541426      add     esp, 4
.text:00541429      pop     edi
.text:0054142A      pop     esi
.text:0054142B      pop     ebp
.text:0054142C      pop     ebx
.text:0054142D      retn
.text:0054142E
.text:0054142E loc_54142E:                ; CODE XREF: decrypt_file+1A
.text:0054142E      push     2               ; Origin
.text:00541430      push     0               ; Offset
.text:00541432      push     esi              ; File
.text:00541433      call    _fseek
.text:00541438      push     esi              ; File
.text:00541439      call    _ftell
.text:0054143E      push     0               ; Origin
.text:00541440      push     0               ; Offset
.text:00541442      push     esi              ; File
.text:00541443      mov     ebp, eax

```



```

.text:00541445      call     _fseek
.text:0054144A      push    ebp                ; Size
.text:0054144B      call    _malloc
.text:00541450      push    esi                ; File
.text:00541451      mov     ebx, eax
.text:00541453      push    1                  ; Count
.text:00541455      push    ebp                ; ElementSize
.text:00541456      push    ebx                ; DstBuf
.text:00541457      call    _fread
.text:0054145C      push    esi                ; File
.text:0054145D      call    _fclose
.text:00541462      add     esp, 34h
.text:00541465      mov     ecx, 3
.text:0054146A      mov     edi, offset aQr9_0 ; "QR9"
.text:0054146F      mov     esi, ebx
.text:00541471      xor     edx, edx
.text:00541473      repe    cmpsb
.text:00541475      jz      short loc_541489
.text:00541477      push    offset aFileIsNotCrypt ; "File is not encrypted!\n"
.text:0054147C      call    _printf
.text:00541481      add     esp, 4
.text:00541484      pop     edi
.text:00541485      pop     esi
.text:00541486      pop     ebp
.text:00541487      pop     ebx
.text:00541488      retn
.text:00541489
.text:00541489 loc_541489:                ; CODE XREF: decrypt_file+75
.text:00541489      mov     eax, [esp+10h+Src]
.text:0054148D      mov     edi, [ebx+3]
.text:00541490      add     ebp, 0FFFFFFF9h
.text:00541493      lea     esi, [ebx+7]
.text:00541496      push    eax                ; Src
.text:00541497      push    ebp                ; int
.text:00541498      push    esi                ; int
.text:00541499      call    decrypt
.text:0054149E      mov     ecx, [esp+1Ch+arg_4]
.text:005414A2      push    offset aWb_0        ; "wb"
.text:005414A7      push    ecx                ; Filename
.text:005414A8      call    _fopen
.text:005414AD      mov     ebp, eax
.text:005414AF      push    ebp                ; File
.text:005414B0      push    1                  ; Count
.text:005414B2      push    edi                ; Size
.text:005414B3      push    esi                ; Str
.text:005414B4      call    _fwrite
.text:005414B9      push    ebp                ; File
.text:005414BA      call    _fclose
.text:005414BF      push    ebx                ; Memory
.text:005414C0      call    _free
.text:005414C5      add     esp, 2Ch
.text:005414C8      pop     edi
.text:005414C9      pop     esi
.text:005414CA      pop     ebp
.text:005414CB      pop     ebx
.text:005414CC      retn
.text:005414CC decrypt_file      endp

```

Все имена функций и меток даны мною в процессе анализа.

Начнем с самого верха. Вот функция, берущая на вход два имени файла и пароль.

```

.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file      proc near
.text:00541320
.text:00541320 Str                = dword ptr 4
.text:00541320 Filename            = dword ptr 8
.text:00541320 password            = dword ptr 0Ch
.text:00541320

```

Открыть файл и сообщить об ошибке в случае ошибки:

```
.text:00541320      mov     eax, [esp+Str]
.text:00541324      push    ebp
.text:00541325      push    offset Mode      ; "rb"
.text:0054132A      push    eax               ; Filename
.text:0054132B      call    _fopen           ; open file
.text:00541330      mov     ebp, eax
.text:00541332      add     esp, 8
.text:00541335      test    ebp, ebp
.text:00541337      jnz     short loc_541348
.text:00541339      push    offset Format     ; "Cannot open input file!\n"
.text:0054133E      call    _printf
.text:00541343      add     esp, 4
.text:00541346      pop     ebp
.text:00541347      retn
.text:00541348      loc_541348:
```

Узнать размер файла используя `fseek()` / `ftell()`:

```
.text:00541348      push    ebx
.text:00541349      push    esi
.text:0054134A      push    edi
.text:0054134B      push    2                ; Origin
.text:0054134D      push    0                ; Offset
.text:0054134F      push    ebp              ; File

; переместить текущую позицию файла на конец
.text:00541350      call    _fseek
.text:00541355      push    ebp              ; File
.text:00541356      call    _ftell           ; узнать текущую позицию
.text:0054135B      push    0                ; Origin
.text:0054135D      push    0                ; Offset
.text:0054135F      push    ebp              ; File
.text:00541360      mov     [esp+2Ch+Str], eax

; переместить текущую позицию файла на начало
.text:00541364      call    _fseek
```

Этот фрагмент кода вычисляет длину файла, выровненную по 64-байтной границе. Это потому что этот алгоритм шифрования работает только с блоками размерами 64 байта. Работает очень просто: разделить длину файла на 64, забыть об остатке, прибавить 1, умножить на 64. Следующий код удаляет остаток от деления, как если бы это значение уже было разделено на 64 и добавляет 64. Это почти то же самое.

```
.text:00541369      mov     esi, [esp+2Ch+Str]
; сбросить в ноль младшие 6 бит
.text:0054136D      and     esi, 0FFFFFFC0h
; выровнять размер по 64-байтной границе
.text:00541370      add     esi, 40h
```

Выделить буфер с выровненным размером:

```
.text:00541373      push    esi               ; Size
.text:00541374      call    _malloc
```

Вызвать `memset()`, т.е. очистить выделенный буфер².

```
.text:00541379      mov     ecx, esi
.text:0054137B      mov     ebx, eax          ; указатель на выделенный буфер -> в EBX
.text:0054137D      mov     edx, ecx
.text:0054137F      xor     eax, eax
.text:00541381      mov     edi, ebx
.text:00541383      push    ebp              ; File
.text:00541384      shr     ecx, 2
.text:00541387      rep     stosd
.text:00541389      mov     ecx, edx
.text:0054138B      push    1                ; Count
```

² `malloc()` + `memset()` можно было бы заменить на `calloc()`

```
.text:0054138D and     ecx, 3
.text:00541390 rep stosb      ; memset (buffer, 0, выровненный_размер)
```

Чтение файла используя стандартную функцию Си `fread()`.

```
.text:00541392      mov     eax, [esp+38h+Str]
.text:00541396      push    eax             ; ElementSize
.text:00541397      push    ebx             ; DstBuf
.text:00541398      call    _fread          ; read file
.text:0054139D      push    ebp             ; File
.text:0054139E      call    _fclose
```

Вызов `crypt()`. Эта функция берет на вход буфер, длину буфера (выровненную) и строку пароля.

```
.text:005413A3      mov     ecx, [esp+44h+password]
.text:005413A7      push    ecx             ; password
.text:005413A8      push    esi             ; aligned size
.text:005413A9      push    ebx             ; buffer
.text:005413AA      call    crypt           ; do crypt
```

Создать выходной файл. Кстати, разработчик забыл вставить проверку, создан ли файл успешно! Результат открытия файла, впрочем, проверяется.

```
.text:005413AF      mov     edx, [esp+50h+Filename]
.text:005413B3      add     esp, 40h
.text:005413B6      push    offset aWb      ; "wb"
.text:005413BB      push    edx             ; Filename
.text:005413BC      call    _fopen
.text:005413C1      mov     edi, eax
```

Теперь хэндл созданного файла в регистре `EDI`. Записываем сигнатуру «QR9».

```
.text:005413C3      push    edi             ; File
.text:005413C4      push    1              ; Count
.text:005413C6      push    3              ; Size
.text:005413C8      push    offset aQr9     ; "QR9"
.text:005413CD      call    _fwrite         ; write file signature
```

Записываем настоящую длину файла (не выровненную):

```
.text:005413D2      push    edi             ; File
.text:005413D3      push    1              ; Count
.text:005413D5      lea     eax, [esp+30h+Str]
.text:005413D9      push    4              ; Size
.text:005413DB      push    eax             ; Str
.text:005413DC      call    _fwrite         ; write original file size
```

Записываем зашифрованный буфер:

```
.text:005413E1      push    edi             ; File
.text:005413E2      push    1              ; Count
.text:005413E4      push    esi             ; Size
.text:005413E5      push    ebx             ; Str
.text:005413E6      call    _fwrite         ; write encrypted file
```

Закрывать файл и освободить выделенный буфер:

```
.text:005413EB      push    edi             ; File
.text:005413EC      call    _fclose
.text:005413F1      push    ebx             ; Memory
.text:005413F2      call    _free
.text:005413F7      add     esp, 40h
.text:005413FA      pop     edi
.text:005413FB      pop     esi
.text:005413FC      pop     ebx
.text:005413FD      pop     ebp
.text:005413FE      retn
.text:005413FE crypt_file endp
```

Переписанный на Си код:

```
void crypt_file(char *fin, char* fout, char *pw)
{
    FILE *f;
    int flen, flen_aligned;
    BYTE *buf;

    f=fopen(fin, "rb");

    if (f==NULL)
    {
        printf ("Cannot open input file!\n");
        return;
    };

    fseek (f, 0, SEEK_END);
    flen=ftell (f);
    fseek (f, 0, SEEK_SET);

    flen_aligned=(flen&0xFFFFFC0)+0x40;

    buf=(BYTE*)malloc (flen_aligned);
    memset (buf, 0, flen_aligned);

    fread (buf, flen, 1, f);

    fclose (f);

    crypt (buf, flen_aligned, pw);

    f=fopen(fout, "wb");

    fwrite ("QR9", 3, 1, f);
    fwrite (&flen, 4, 1, f);
    fwrite (buf, flen_aligned, 1, f);

    fclose (f);

    free (buf);
};
```

Процедура дешифрования почти такая же:

```
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)
.text:00541400 decrypt_file      proc near
.text:00541400
.text:00541400 Filename          = dword ptr  4
.text:00541400 arg_4              = dword ptr  8
.text:00541400 Src                = dword ptr  0Ch
.text:00541400
.text:00541400          mov     eax, [esp+Filename]
.text:00541404          push    ebx
.text:00541405          push    ebp
.text:00541406          push    esi
.text:00541407          push    edi
.text:00541408          push    offset aRb      ; "rb"
.text:0054140D          push    eax          ; Filename
.text:0054140E          call   _fopen
.text:00541413          mov     esi, eax
.text:00541415          add     esp, 8
.text:00541418          test    esi, esi
.text:0054141A          jnz     short loc_54142E
.text:0054141C          push    offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421          call   _printf
.text:00541426          add     esp, 4
.text:00541429          pop     edi
.text:0054142A          pop     esi
.text:0054142B          pop     ebp
.text:0054142C          pop     ebx
.text:0054142D          retn
```

```

.text:0054142E
.text:0054142E loc_54142E:
.text:0054142E      push     2           ; Origin
.text:00541430      push     0           ; Offset
.text:00541432      push     esi          ; File
.text:00541433      call    _fseek
.text:00541438      push     esi          ; File
.text:00541439      call    _ftell
.text:0054143E      push     0           ; Origin
.text:00541440      push     0           ; Offset
.text:00541442      push     esi          ; File
.text:00541443      mov     ebp, eax
.text:00541445      call    _fseek
.text:0054144A      push     ebp          ; Size
.text:0054144B      call    _malloc
.text:00541450      push     esi          ; File
.text:00541451      mov     ebx, eax
.text:00541453      push     1           ; Count
.text:00541455      push     ebp          ; ElementSize
.text:00541456      push     ebx          ; DstBuf
.text:00541457      call    _fread
.text:0054145C      push     esi          ; File
.text:0054145D      call    _fclose

```

Проверяем сигнатуру (первые 3 байта):

```

.text:00541462      add     esp, 34h
.text:00541465      mov     ecx, 3
.text:0054146A      mov     edi, offset aQr9_0 ; "QR9"
.text:0054146F      mov     esi, ebx
.text:00541471      xor     edx, edx
.text:00541473      repe cmpsb
.text:00541475      jz      short loc_541489

```

Сообщить об ошибке если сигнатура отсутствует:

```

.text:00541477      push    offset aFileIsNotCrypt ; "File is not encrypted!\n"
.text:0054147C      call    _printf
.text:00541481      add     esp, 4
.text:00541484      pop     edi
.text:00541485      pop     esi
.text:00541486      pop     ebp
.text:00541487      pop     ebx
.text:00541488      retn
.text:00541489
.text:00541489 loc_541489:

```

Вызвать `decrypt()`.

```

.text:00541489      mov     eax, [esp+10h+Src]
.text:0054148D      mov     edi, [ebx+3]
.text:00541490      add     ebp, 0FFFFFFF9h
.text:00541493      lea     esi, [ebx+7]
.text:00541496      push    eax           ; Src
.text:00541497      push    ebp           ; int
.text:00541498      push    esi           ; int
.text:00541499      call    decrypt
.text:0054149E      mov     ecx, [esp+1Ch+arg_4]
.text:005414A2      push    offset aWb_0   ; "wb"
.text:005414A7      push    ecx           ; Filename
.text:005414A8      call    _fopen
.text:005414AD      mov     ebp, eax
.text:005414AF      push    ebp           ; File
.text:005414B0      push    1             ; Count
.text:005414B2      push    edi           ; Size
.text:005414B3      push    esi           ; Str
.text:005414B4      call    _fwrite
.text:005414B9      push    ebp           ; File
.text:005414BA      call    _fclose
.text:005414BF      push    ebx           ; Memory

```

```

.text:005414C0      call    _free
.text:005414C5      add     esp, 2Ch
.text:005414C8      pop     edi
.text:005414C9      pop     esi
.text:005414CA      pop     ebp
.text:005414CB      pop     ebx
.text:005414CC      retn
.text:005414CC decrypt_file endp

```

Переписанный на Си код:

```

void decrypt_file(char *fin, char* fout, char *pw)
{
    FILE *f;
    int real_flen, flen;
    BYTE *buf;

    f=fopen(fin, "rb");

    if (f==NULL)
    {
        printf ("Cannot open input file!\n");
        return;
    };

    fseek (f, 0, SEEK_END);
    flen=ftell (f);
    fseek (f, 0, SEEK_SET);

    buf=(BYTE*)malloc (flen);

    fread (buf, flen, 1, f);

    fclose (f);

    if (memcmp (buf, "QR9", 3)!=0)
    {
        printf ("File is not encrypted!\n");
        return;
    };

    memcpy (&real_flen, buf+3, 4);

    decrypt (buf+(3+4), flen-(3+4), pw);

    f=fopen(fout, "wb");

    fwrite (buf+(3+4), real_flen, 1, f);

    fclose (f);

    free (buf);
};

```

ОК, посмотрим глубже.

Функция `crypt()`:

```

.text:00541260 crypt      proc near
.text:00541260
.text:00541260 arg_0      = dword ptr  4
.text:00541260 arg_4      = dword ptr  8
.text:00541260 arg_8      = dword ptr  0Ch
.text:00541260
.text:00541260      push    ebx
.text:00541261      mov     ebx, [esp+4+arg_0]
.text:00541265      push    ebp
.text:00541266      push    esi
.text:00541267      push    edi
.text:00541268      xor     ebp, ebp

```

```
.text:0054126A
.text:0054126A loc_54126A:
```

Этот фрагмент кода копирует часть входного буфера во внутренний буфер, который мы позже назовем «cube64». Длина в регистре ECX. MOVSD означает *скопировать 32-битное слово*, так что, 16 32-битных слов это как раз 64 байта.

```
.text:0054126A      mov     eax, [esp+10h+arg_8]
.text:0054126E      mov     ecx, 10h
.text:00541273      mov     esi, ebx    ; EBX is pointer within input buffer
.text:00541275      mov     edi, offset cube64
.text:0054127A      push    1
.text:0054127C      push    eax
.text:0054127D      rep     movsd
```

Вызвать rotate_all_with_password() :

```
.text:0054127F      call    rotate_all_with_password
```

Скопировать зашифрованное содержимое из «cube64» назад в буфер :

```
.text:00541284      mov     eax, [esp+18h+arg_4]
.text:00541288      mov     edi, ebx
.text:0054128A      add     ebp, 40h
.text:0054128D      add     esp, 8
.text:00541290      mov     ecx, 10h
.text:00541295      mov     esi, offset cube64
.text:0054129A      add     ebx, 40h    ; add 64 to input buffer pointer
.text:0054129D      cmp     ebp, eax    ; EBP contain amount of encrypted data.
.text:0054129F      rep     movsd
```

Если EBP не больше чем длина во входном аргументе, тогда переходим к следующему блоку.

```
.text:005412A1      jle     short loc_54126A
.text:005412A3      pop     edi
.text:005412A4      pop     esi
.text:005412A5      pop     ebp
.text:005412A6      pop     ebx
.text:005412A7      retn
.text:005412A7 crypt      endp
```

Реконструированная функция crypt() :

```
void crypt (BYTE *buf, int sz, char *pw)
{
    int i=0;

    do
    {
        memcpy (cube, buf+i, 8*8);
        rotate_all (pw, 1);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);
};
```

ОК, углубимся в функцию rotate_all_with_password(). Она берет на вход два аргумента: строку пароля и число. В функции crypt(), число 1 используется и в decrypt() (где rotate_all_with_password() функция вызывается также), число 3.

```
.text:005411B0 rotate_all_with_password proc near
.text:005411B0
.text:005411B0 arg_0      = dword ptr 4
.text:005411B0 arg_4      = dword ptr 8
.text:005411B0
.text:005411B0      mov     eax, [esp+arg_0]
.text:005411B4      push    ebp
.text:005411B5      mov     ebp, eax
```

Проверяем символы в пароле. Если это ноль, выходим:

```
.text:005411B7      cmp     byte ptr [eax], 0
.text:005411BA      jz      exit
.text:005411C0      push   ebx
.text:005411C1      mov     ebx, [esp+8+arg_4]
.text:005411C5      push   esi
.text:005411C6      push   edi
.text:005411C7      loop_begin:
```

Вызываем `tolower()`, стандартную функцию Си.

```
.text:005411C7      movsx   eax, byte ptr [ebp+0]
.text:005411CB      push    eax                ; C
.text:005411CC      call    _tolower
.text:005411D1      add     esp, 4
```

Хмм, если пароль содержит символ не из латинского алфавита, он пропускается! Действительно, если мы запускаем утилиту для шифрования используя символы не латинского алфавита, похоже, они просто игнорируются.

```
.text:005411D4      cmp     al, 'a'
.text:005411D6      jl      short next_character_in_password
.text:005411D8      cmp     al, 'z'
.text:005411DA      jg      short next_character_in_password
.text:005411DC      movsx   ecx, al
```

Отнимем значение «а» (97) от символа.

```
.text:005411DF      sub     ecx, 'a' ; 97
```

После вычитания, тут будет 0 для «а», 1 для «b», и так далее. И 25 для «z».

```
.text:005411E2      cmp     ecx, 24
.text:005411E5      jle     short skip_subtracting
.text:005411E7      sub     ecx, 24
```

Похоже, символы «у» и «z» также исключительные. После этого фрагмента кода, «у» становится 0, а «z» — 1. Это значит, что 26 латинских букв становятся значениями в интервале 0..23, (всего 24).

```
.text:005411EA
.text:005411EA skip_subtracting:                ; CODE XREF: rotate_all_with_password+35
```

Это, на самом деле, деление через умножение. Читайте об этом больше в секции «Деление на 9» ([42](#) (стр. 483)).

Это код, на самом деле, делит значение символа пароля на 3.

```
.text:005411EA      mov     eax, 55555556h
.text:005411EF      imul    ecx
.text:005411F1      mov     eax, edx
.text:005411F3      shr     eax, 1Fh
.text:005411F6      add     edx, eax
.text:005411F8      mov     eax, ecx
.text:005411FA      mov     esi, edx
.text:005411FC      mov     ecx, 3
.text:00541201      cdq
.text:00541202      idiv    ecx
```

EDX — остаток от деления.

```
.text:00541204 sub     edx, 0
.text:00541207 jz      short call_rotate1 ; если остаток 0, перейти к rotate1
.text:00541209 dec     edx
.text:0054120A jz      short call_rotate2 ; .. если он 1, перейти к rotate2
.text:0054120C dec     edx
.text:0054120D jnz     short next_character_in_password
.text:0054120F test    ebx, ebx
.text:00541211 jle     short next_character_in_password
.text:00541213 mov     edi, ebx
```


Если остаток 2, вызываем `rotate3()`. `EDX` это второй аргумент функции `rotate_all_with_password()`. Как мы уже заметили, 1 это для шифрования, 3 для дешифрования. Так что здесь цикл, функции `rotate1/2/3` будут вызываться столько же раз, сколько значение переменной в первом аргументе.

```
.text:00541215 call_rotate3:
.text:00541215      push    esi
.text:00541216      call    rotate3
.text:0054121B      add     esp, 4
.text:0054121E      dec     edi
.text:0054121F      jnz     short call_rotate3
.text:00541221      jmp     short next_character_in_password
.text:00541223
.text:00541223 call_rotate2:
.text:00541223      test    ebx, ebx
.text:00541225      jle     short next_character_in_password
.text:00541227      mov     edi, ebx
.text:00541229
.text:00541229 loc_541229:
.text:00541229      push    esi
.text:0054122A      call    rotate2
.text:0054122F      add     esp, 4
.text:00541232      dec     edi
.text:00541233      jnz     short loc_541229
.text:00541235      jmp     short next_character_in_password
.text:00541237
.text:00541237 call_rotate1:
.text:00541237      test    ebx, ebx
.text:00541239      jle     short next_character_in_password
.text:0054123B      mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:
.text:0054123D      push    esi
.text:0054123E      call    rotate1
.text:00541243      add     esp, 4
.text:00541246      dec     edi
.text:00541247      jnz     short loc_54123D
.text:00541249
```

Достать следующий символ из строки пароля.

```
.text:00541249 next_character_in_password:
.text:00541249      mov     al, [ebp+1]
```

[Инкремент](#) указателя на символ в строке пароля:

```
.text:0054124C      inc     ebp
.text:0054124D      test    al, al
.text:0054124F      jnz     loop_begin
.text:00541255      pop     edi
.text:00541256      pop     esi
.text:00541257      pop     ebx
.text:00541258
.text:00541258 exit:
.text:00541258      pop     ebp
.text:00541259      retn
.text:00541259 rotate_all_with_password endp
```

Реконструированный код на Си:

```
void rotate_all (char *pwd, int v)
{
    char *p=pwd;

    while (*p)
    {
        char c=*p;
        int q;

        c=tolower (c);
```

```

        if (c>='a' && c<='z')
        {
            q=c-'a';
            if (q>24)
                q-=24;

            int quotient=q/3;
            int remainder=q % 3;

            switch (remainder)
            {
                case 0: for (int i=0; i<v; i++) rotate1 (quotient); break;
                case 1: for (int i=0; i<v; i++) rotate2 (quotient); break;
                case 2: for (int i=0; i<v; i++) rotate3 (quotient); break;
            };

        };

        p++;
    };
};

```

Углубимся еще дальше и исследуем функции rotate1/2/3. Каждая функция вызывает еще две. В итоге мы назовем их `set_bit()` и `get_bit()`.

Начнем с `get_bit()`:

```

.text:00541050 get_bit      proc near
.text:00541050
.text:00541050 arg_0        = dword ptr  4
.text:00541050 arg_4        = dword ptr  8
.text:00541050 arg_8        = byte ptr  0Ch
.text:00541050
.text:00541050          mov     eax, [esp+arg_4]
.text:00541054          mov     ecx, [esp+arg_0]
.text:00541058          mov     al, cube64[ecx*8]
.text:0054105F          mov     cl, [esp+arg_8]
.text:00541063          shr     al, cl
.text:00541065          and     al, 1
.text:00541067          retn
.text:00541067 get_bit      endp

```

...иными словами: подсчитать индекс в массиве cube64: $arg\_4 + arg\_0 * 8$. Затем сдвинуть байт из массива вправо на количество бит заданных в `arg_8`. Изолировать самый младший бит и вернуть его

Посмотрим другую функцию, `set_bit()`:

```

.text:00541000 set_bit      proc near
.text:00541000
.text:00541000 arg_0        = dword ptr  4
.text:00541000 arg_4        = dword ptr  8
.text:00541000 arg_8        = dword ptr  0Ch
.text:00541000 arg_C        = byte ptr  10h
.text:00541000
.text:00541000          mov     al, [esp+arg_C]
.text:00541004          mov     ecx, [esp+arg_8]
.text:00541008          push    esi
.text:00541009          mov     esi, [esp+4+arg_0]
.text:0054100D          test    al, al
.text:0054100F          mov     eax, [esp+4+arg_4]
.text:00541013          mov     dl, 1
.text:00541015          jz      short loc_54102B

```

`DL` тут равно 1. Сдвигаем эту единицу на количество, указанное в `arg_8`. Например, если в `arg_8` число 4, тогда значение в `DL` станет 0x10 или 1000b в двоичной системе счисления.

```

.text:00541017          shl     dl, cl
.text:00541019          mov     cl, cube64[ecx+esi*8]

```

Вытащить бит из массива и явно выставить его.

```
.text:00541020          or      cl, dl
```

Сохранить его назад:

```
.text:00541022          mov     cube64[eax+esi*8], cl
.text:00541029          pop     esi
.text:0054102A          retn
.text:0054102B
.text:0054102B loc_54102B:
.text:0054102B          shl     dl, cl
```

Если arg_C не ноль...

```
.text:0054102D          mov     cl, cube64[eax+esi*8]
```

...инвертировать DL. Например, если состояние DL после сдвига 0x10 или 1000b в двоичной системе, здесь будет 0xEF после инструкции **NOT** или 11101111b в двоичной системе.

```
.text:00541034          not     dl
```

Эта инструкция сбрасывает бит, иными словами, она сохраняет все биты в **CL** которые также выставлены в **DL** кроме тех в **DL**, что были сброшены. Это значит, что если в **DL**, например, 11101111b в двоичной системе, все биты будут сохранены кроме пятого (считая с младшего бита).

```
.text:00541036          and     cl, dl
```

Сохранить его назад

```
.text:00541038          mov     cube64[eax+esi*8], cl
.text:0054103F          pop     esi
.text:00541040          retn
.text:00541040 set_bit    endp
```

Это почти то же самое что и **get_bit()**, кроме того, что если arg_C ноль, тогда функция сбрасывает указанный бит в массиве, либо же, в противном случае, выставляет его в 1.

Мы также знаем что размер массива 64. Первые два аргумента и у **set_bit()** и у **get_bit()** могут быть представлены как двумерные координаты. Таким образом, массив – это матрица 8*8.

Представление на Си всего того, что мы уже знаем:

```
#define IS_SET(flag, bit)    ((flag) & (bit))
#define SET_BIT(var, bit)    ((var) |= (bit))
#define REMOVE_BIT(var, bit) ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int shift, int bit)
{
    if (bit)
        SET_BIT (cube[x][y], 1<<shift);
    else
        REMOVE_BIT (cube[x][y], 1<<shift);
};

bool get_bit (int x, int y, int shift)
{
    if ((cube[x][y]>>shift)&1==1)
        return 1;
    return 0;
};
```

Теперь вернемся к функциям rotate1/2/3.

```
.text:00541070 rotate1    proc near
.text:00541070
```

Выделение внутреннего массива размером 64 байта в локальном стеке:

```
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0          = dword ptr  4
.text:00541070
.text:00541070          sub     esp, 40h
.text:00541073          push    ebx
.text:00541074          push    ebp
.text:00541075          mov     ebp, [esp+48h+arg_0]
.text:00541079          push    esi
.text:0054107A          push    edi
.text:0054107B          xor     edi, edi      ; EDI is loop1 counter
```

EBX указывает на внутренний массив

```
.text:0054107D          lea     ebx, [esp+50h+internal_array_64]
.text:00541081
```

Здесь два вложенных цикла:

```
.text:00541081 first_loop1_begin:
.text:00541081 xor     esi, esi      ; ESI это счетчик второго цикла
.text:00541083
.text:00541083 first_loop2_begin:
.text:00541083 push    ebp          ; arg_0
.text:00541084 push    esi          ; счетчик первого цикла
.text:00541085 push    edi          ; счетчик второго цикла
.text:00541086 call    get_bit
.text:0054108B add     esp, 0Ch
.text:0054108E mov     [ebx+esi], al ; записываем во внутренний массив
.text:00541091 inc     esi          ; инкремент счетчика первого цикла
.text:00541092 cmp     esi, 8
.text:00541095 jl     short first_loop2_begin
.text:00541097 inc     edi          ; инкремент счетчика второго цикла

; инкремент указателя во внутреннем массиве на 8 на каждой итерации первого цикла
.text:00541098 add     ebx, 8
.text:0054109B cmp     edi, 8
.text:0054109E jl     short first_loop1_begin
```

Мы видим, что оба счетчика циклов в интервале 0..7. Также, они используются как первый и второй аргумент **get_bit()**.

Третий аргумент **get_bit()** это единственный аргумент **rotate1()**. То что возвращает **get_bit()** будет сохранено во внутреннем массиве.

Снова приготовить указатель на внутренний массив:

```
.text:005410A0 lea     ebx, [esp+50h+internal_array_64]
.text:005410A4 mov     edi, 7      ; EDI здесь счетчик первого цикла, значение на старте - 7
.text:005410A9
.text:005410A9 second_loop1_begin:
.text:005410A9 xor     esi, esi      ; ESI - счетчик второго цикла
.text:005410AB
.text:005410AB second_loop2_begin:
.text:005410AB mov     al, [ebx+esi] ; EN(value from internal array)значение из внутреннего массива
.text:005410AE push    eax
.text:005410AF push    ebp          ; arg_0
.text:005410B0 push    edi          ; счетчик первого цикла
.text:005410B1 push    esi          ; счетчик второго цикла
.text:005410B2 call    set_bit
.text:005410B7 add     esp, 10h
.text:005410BA inc     esi          ; инкремент счетчика второго цикла
.text:005410BB cmp     esi, 8
.text:005410BE jl     short second_loop2_begin
.text:005410C0 dec     edi          ; декремент счетчика первого цикла
.text:005410C1 add     ebx, 8      ; инкремент указателя во внутреннем массиве
.text:005410C4 cmp     edi, 0FFFFFFFh
.text:005410C7 jg     short second_loop1_begin
.text:005410C9 pop     edi
.text:005410CA pop     esi
.text:005410CB pop     ebp
.text:005410CC pop     ebx
```

```
.text:005410CD    add     esp, 40h
.text:005410D0    retn
.text:005410D0 rotate1      endp
```

...этот код помещает содержимое из внутреннего массива в глобальный массив cube используя функцию `set_bit()`, но, в обратном порядке! Теперь счетчик первого цикла в интервале 7 до 0, уменьшается на 1 на каждой итерации!

Представление кода на Си выглядит так:

```
void rotate1 (int v)
{
    bool tmp[8][8]; // internal array
    int i, j;

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            tmp[i][j]=get_bit (i, j, v);

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            set_bit (j, 7-i, v, tmp[x][y]);
};
```

Не очень понятно, но если мы посмотрим в функцию `rotate2()` :

```
.text:005410E0 rotate2 proc near
.text:005410E0
.text:005410E0 internal_array_64 = byte ptr -40h
.text:005410E0 arg_0 = dword ptr 4
.text:005410E0
.text:005410E0     sub     esp, 40h
.text:005410E3     push    ebx
.text:005410E4     push    ebp
.text:005410E5     mov     ebp, [esp+48h+arg_0]
.text:005410E9     push    esi
.text:005410EA     push    edi
.text:005410EB     xor     edi, edi          ; счетчик первого цикла
.text:005410ED     lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:
.text:005410F1     xor     esi, esi          ; счетчик второго цикла
.text:005410F3
.text:005410F3 loc_5410F3:
.text:005410F3     push    esi              ; счетчик второго цикла
.text:005410F4     push    edi              ; счетчик первого цикла
.text:005410F5     push    ebp              ; arg_0
.text:005410F6     call    get_bit
.text:005410FB     add     esp, 0Ch
.text:005410FE     mov     [ebx+esi], al     ; записать во внутренний массив
.text:00541101     inc     esi              ; инкремент счетчика первого цикла
.text:00541102     cmp     esi, 8
.text:00541105     jl      short loc_5410F3
.text:00541107     inc     edi              ; инкремент счетчика второго цикла
.text:00541108     add     ebx, 8
.text:0054110B     cmp     edi, 8
.text:0054110E     jl      short loc_5410F1
.text:00541110     lea     ebx, [esp+50h+internal_array_64]
.text:00541114     mov     edi, 7           ; первоначальное значение счетчика первого цикла - 7
.text:00541119
.text:00541119 loc_541119:
.text:00541119     xor     esi, esi          ; счетчик второго цикла
.text:0054111B
.text:0054111B loc_54111B:
.text:0054111B     mov     al, [ebx+esi]     ; взять байт из внутреннего массива
.text:0054111E     push    eax
.text:0054111F     push    edi              ; счетчик первого цикла
.text:00541120     push    esi              ; счетчик второго цикла
.text:00541121     push    ebp              ; arg_0
.text:00541122     call    set_bit
.text:00541127     add     esp, 10h
```

```
.text:0054112A    inc     esi             ;
.text:0054112B    cmp     esi, 8
.text:0054112E    jnl     short loc_54111B
.text:00541130    dec     edi             ;
.text:00541131    add     ebx, 8
.text:00541134    cmp     edi, 0FFFFFFFh
.text:00541137    jg      short loc_541119
.text:00541139    pop     edi
.text:0054113A    pop     esi
.text:0054113B    pop     ebp
.text:0054113C    pop     ebx
.text:0054113D    add     esp, 40h
.text:00541140    retn
.text:00541140 rotate2 endp
```

Почти то же самое, за исключением иного порядка аргументов в `get_bit()` и `set_bit()`. Перепишем это на Си-подобный код:

```
void rotate2 (int v)
{
    bool tmp[8][8]; // internal array
    int i, j;

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            tmp[i][j]=get_bit (v, i, j);

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            set_bit (v, j, 7-i, tmp[i][j]);
};
```

Перепишем так же функцию `rotate3()` :

```
void rotate3 (int v)
{
    bool tmp[8][8];
    int i, j;

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            tmp[i][j]=get_bit (i, v, j);

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            set_bit (7-j, v, i, tmp[i][j]);
};
```

Теперь всё проще. Если мы представим cube64 как трехмерный куб $8*8*8$, где каждый элемент это бит, то `get_bit()` и `set_bit()` просто берут на вход координаты бита.

Функции `rotate1/2/3` просто поворачивают все биты на определенной плоскости. Три функции, каждая на каждую сторону куба и аргумент `v` выставляет плоскость в интервале 0..7

Может быть, автор алгоритма думал о кубике Рубика $8*8*8$ ³?!

Да, действительно.

Рассмотрим функцию `decrypt()` , вот её переписанная версия:

```
void decrypt (BYTE *buf, int sz, char *pw)
{
    char *p=strdup (pw);
    strrev (p);
    int i=0;

    do
    {
```

³[wikipedia](https://ru.wikipedia.org/wiki/Кубик_Рубика)

```

        memcpy (cube, buf+i, 8*8);
        rotate_all (p, 3);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);

    free (p);
};

```

Почти то же самое что и `crypt()`, но строка пароля разворачивается стандартной функцией Си `strrev()`⁴ и `rotate_all()` вызывается с аргументом 3.

Это значит, что, в случае дешифровки, `rotate1/2/3` будут вызываться трижды.

Это почти кубик Рубика! Если вы хотите вернуть его состояние назад, делайте то же самое в обратном порядке и направлении! Чтобы вернуть эффект от поворота плоскости по часовой стрелке, нужно повернуть её же против часовой стрелки, либо же трижды по часовой стрелке.

`rotate1()`, вероятно, поворот «лицевой» плоскости. `rotate2()`, вероятно, поворот «верхней» плоскости. `rotate3()`, вероятно, поворот «левой» плоскости.

Вернемся к ядру функции `rotate_all()`

```

q=c-'a';
if (q>24)
    q-=24;

int quotient=q/3; // in range 0..7
int remainder=q % 3;

switch (remainder)
{
    case 0: for (int i=0; i<v; i++) rotate1 (quotient); break; // front
    case 1: for (int i=0; i<v; i++) rotate2 (quotient); break; // top
    case 2: for (int i=0; i<v; i++) rotate3 (quotient); break; // left
};

```

Так понять проще: каждый символ пароля определяет сторону (одну из трех) и плоскость (одну из восьми). $3 \cdot 8 = 24$, вот почему два последних символа латинского алфавита переопределяются так чтобы алфавит состоял из 24-х элементов.

Алгоритм очевидно слаб: в случае коротких паролей, в бинарном редакторе файлов можно будет увидеть, что в зашифрованных файлах остались незашифрованные символы.

Весь исходный код в реконструированном виде:

```

#include <windows.h>

#include <stdio.h>
#include <assert.h>

#define IS_SET(flag, bit)      ((flag) & (bit))
#define SET_BIT(var, bit)      ((var) |= (bit))
#define REMOVE_BIT(var, bit)   ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int z, bool bit)
{
    if (bit)
        SET_BIT (cube[x][y], 1<<z);
    else
        REMOVE_BIT (cube[x][y], 1<<z);
};

bool get_bit (int x, int y, int z)
{
    if ((cube[x][y]>>z)&1==1)
        return true;
}

```

⁴MSDN

```

        return false;
};

void rotate_f (int row)
{
    bool tmp[8][8];
    int x, y;

    for (x=0; x<8; x++)
        for (y=0; y<8; y++)
            tmp[x][y]=get_bit (x, y, row);

    for (x=0; x<8; x++)
        for (y=0; y<8; y++)
            set_bit (y, 7-x, row, tmp[x][y]);
};

void rotate_t (int row)
{
    bool tmp[8][8];
    int y, z;

    for (y=0; y<8; y++)
        for (z=0; z<8; z++)
            tmp[y][z]=get_bit (row, y, z);

    for (y=0; y<8; y++)
        for (z=0; z<8; z++)
            set_bit (row, z, 7-y, tmp[y][z]);
};

void rotate_l (int row)
{
    bool tmp[8][8];
    int x, z;

    for (x=0; x<8; x++)
        for (z=0; z<8; z++)
            tmp[x][z]=get_bit (x, row, z);

    for (x=0; x<8; x++)
        for (z=0; z<8; z++)
            set_bit (7-z, row, x, tmp[x][z]);
};

void rotate_all (char *pwd, int v)
{
    char *p=pwd;

    while (*p)
    {
        char c=*p;
        int q;

        c=tolower (c);

        if (c>='a' && c<='z')
        {
            q=c-'a';
            if (q>24)
                q-=24;

            int quotient=q/3;
            int remainder=q % 3;

            switch (remainder)
            {
                case 0: for (int i=0; i<v; i++) rotate_f (quotient); break;
                case 1: for (int i=0; i<v; i++) rotate_t (quotient); break;
                case 2: for (int i=0; i<v; i++) rotate_l (quotient); break;
            }
        }
        p++;
    }
}

```



```

        };
        };
        p++;
    };
};

void crypt (BYTE *buf, int sz, char *pw)
{
    int i=0;

    do
    {
        memcpy (cube, buf+i, 8*8);
        rotate_all (pw, 1);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);
};

void decrypt (BYTE *buf, int sz, char *pw)
{
    char *p=strdup (pw);
    strrev (p);
    int i=0;

    do
    {
        memcpy (cube, buf+i, 8*8);
        rotate_all (p, 3);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);

    free (p);
};

void crypt_file(char *fin, char* fout, char *pw)
{
    FILE *f;
    int flen, flen_aligned;
    BYTE *buf;

    f=fopen(fin, "rb");

    if (f==NULL)
    {
        printf ("Cannot open input file!\n");
        return;
    };

    fseek (f, 0, SEEK_END);
    flen=ftell (f);
    fseek (f, 0, SEEK_SET);

    flen_aligned=(flen&0xFFFFFC0)+0x40;

    buf=(BYTE*)malloc (flen_aligned);
    memset (buf, 0, flen_aligned);

    fread (buf, flen, 1, f);

    fclose (f);

    crypt (buf, flen_aligned, pw);

    f=fopen(fout, "wb");

```

```

    fwrite ("QR9", 3, 1, f);
    fwrite (&flen, 4, 1, f);
    fwrite (buf, flen_aligned, 1, f);

    fclose (f);

    free (buf);
};

void decrypt_file(char *fin, char* fout, char *pw)
{
    FILE *f;
    int real_flen, flen;
    BYTE *buf;

    f=fopen(fin, "rb");

    if (f==NULL)
    {
        printf ("Cannot open input file!\n");
        return;
    };

    fseek (f, 0, SEEK_END);
    flen=ftell (f);
    fseek (f, 0, SEEK_SET);

    buf=(BYTE*)malloc (flen);

    fread (buf, flen, 1, f);

    fclose (f);

    if (memcmp (buf, "QR9", 3)!=0)
    {
        printf ("File is not encrypted!\n");
        return;
    };

    memcpy (&real_flen, buf+3, 4);

    decrypt (buf+(3+4), flen-(3+4), pw);

    f=fopen(fout, "wb");

    fwrite (buf+(3+4), real_flen, 1, f);

    fclose (f);

    free (buf);
};

// run: input output 0/1 password
// 0 for encrypt, 1 for decrypt

int main(int argc, char *argv[])
{
    if (argc!=5)
    {
        printf ("Incorrect parameters!\n");
        return 1;
    };

    if (strcmp (argv[3], "0")==0)
        crypt_file (argv[1], argv[2], argv[4]);
    else
        if (strcmp (argv[3], "1")==0)
            decrypt_file (argv[1], argv[2], argv[4]);
        else

```

```
        printf ("Wrong param %s\n", argv[3]);  
  
    return 0;  
};
```

Глава 80

SAP

80.1. Касательно сжимания сетевого трафика в клиенте SAP

(Трассировка связи между переменной окружения `TDW_NOCOMPRESS` SAPGUI¹ до «назойливого всплывающего окна» и самой функции сжатия данных.)

Известно, что сетевой трафик между SAPGUI и SAP по умолчанию не шифруется, а сжимается (читайте здесь² и здесь³).

Известно также что если установить переменную окружения `TDW_NOCOMPRESS` в 1, можно выключить сжатие сетевых пакетов.

Но вы увидите окно, которое нельзя будет закрыть :

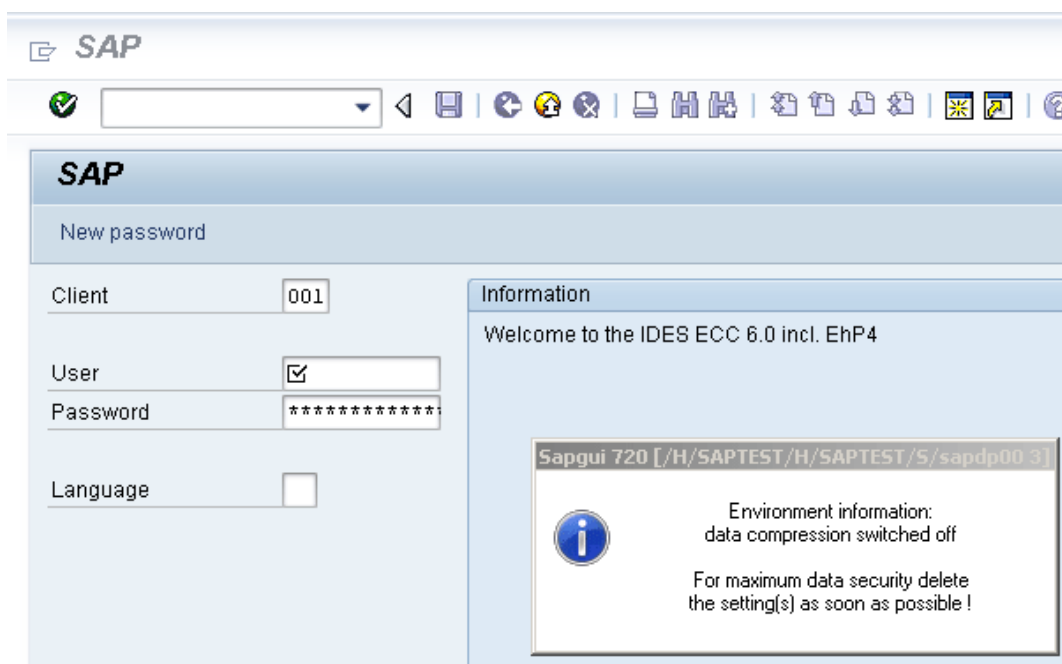


Рис. 80.1: Скриншот

Посмотрим, сможем ли мы как-то убрать это окно .

Но в начале давайте посмотрим, что мы уже знаем. Первое: мы знаем, что переменная окружения `TDW_NOCOMPRESS` проверяется где-то внутри клиента SAPGUI. Второе: строка вроде «data compression switched off» также должна где-то присутствовать. При помощи файлового менеджера FAR⁴ мы можем найти обе эти строки в файле `SAPguilib.dll`.

Так что давайте откроем файл `SAPguilib.dll` в IDA и поищем там строку «`TDW_NOCOMPRESS`». Да, она присутствует и имеется только одна ссылка на эту строку.

¹GUI-клиент от SAP

²<http://go.yurichev.com/17221>

³blog.yurichev.com

⁴<http://go.yurichev.com/17347>

80.1. КАСАТЕЛЬНО СЖИМАНИЯ СЕТЕВОГО ТРАФФИКА В КЛИЕНТЕ SAP

Мы увидим такой фрагмент кода (все смещения верны для версии SAPGUI 720 win32, SAPguilib.dll версия файла 7200,1,0,9009):

```
.text:6440D51B      lea     eax, [ebp+2108h+var_211C]
.text:6440D51E      push    eax                ; int
.text:6440D51F      push    offset aTdw_nocompress ; "TDW_NOCOMPRESS"
.text:6440D524      mov     byte ptr [edi+15h], 0
.text:6440D528      call    chk_env
.text:6440D52D      pop     ecx
.text:6440D52E      pop     ecx
.text:6440D52F      push    offset byte_64443AF8
.text:6440D534      lea     ecx, [ebp+2108h+var_211C]

; demangled name: int ATL::CStringT::Compare(char const *)const
.text:6440D537      call    ds:mfc90_1603
.text:6440D53D      test    eax, eax
.text:6440D53F      jz      short loc_6440D55A
.text:6440D541      lea     ecx, [ebp+2108h+var_211C]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:6440D544      call    ds:mfc90_910
.text:6440D54A      push    eax                ; Str
.text:6440D54B      call    ds:atoi
.text:6440D551      test    eax, eax
.text:6440D553      setnz   al
.text:6440D556      pop     ecx
.text:6440D557      mov     [edi+15h], al
```

Строка возвращаемая функцией `chk_env()` через второй аргумент, обрабатывается далее строковыми функциями MFC, затем вызывается `atoi()`⁵. После этого, число сохраняется в `edi+15h`.

Обратите также внимание на функцию `chk_env` (это мы так назвали её вручную):

```
.text:64413F20 ; int __cdecl chk_env(char *VarName, int)
.text:64413F20 chk_env      proc near
.text:64413F20
.text:64413F20 DstSize      = dword ptr -0Ch
.text:64413F20 var_8          = dword ptr -8
.text:64413F20 DstBuf        = dword ptr -4
.text:64413F20 VarName        = dword ptr 8
.text:64413F20 arg_4          = dword ptr 0Ch
.text:64413F20
.text:64413F20      push    ebp
.text:64413F21      mov     ebp, esp
.text:64413F23      sub     esp, 0Ch
.text:64413F26      mov     [ebp+DstSize], 0
.text:64413F2D      mov     [ebp+DstBuf], 0
.text:64413F34      push    offset unk_6444C88C
.text:64413F39      mov     ecx, [ebp+arg_4]

; (demangled name) ATL::CStringT::operator=(char const *)
.text:64413F3C      call    ds:mfc90_820
.text:64413F42      mov     eax, [ebp+VarName]
.text:64413F45      push    eax                ; VarName
.text:64413F46      mov     ecx, [ebp+DstSize]
.text:64413F49      push    ecx                ; DstSize
.text:64413F4A      mov     edx, [ebp+DstBuf]
.text:64413F4D      push    edx                ; DstBuf
.text:64413F4E      lea     eax, [ebp+DstSize]
.text:64413F51      push    eax                ; ReturnSize
.text:64413F52      call    ds:getenv_s
.text:64413F58      add     esp, 10h
.text:64413F5B      mov     [ebp+var_8], eax
.text:64413F5E      cmp     [ebp+var_8], 0
.text:64413F62      jz      short loc_64413F68
.text:64413F64      xor     eax, eax
.text:64413F66      jmp     short loc_64413F6C
.text:64413F68
```

⁵Стандартная функция Си, конвертирующая число в строке в число

```

.text:64413F68 loc_64413F68:
.text:64413F68          cmp     [ebp+DstSize], 0
.text:64413F6C          jnz     short loc_64413F72
.text:64413F6E          xor     eax, eax
.text:64413F70          jmp     short loc_64413FBC
.text:64413F72
.text:64413F72 loc_64413F72:
.text:64413F72          mov     ecx, [ebp+DstSize]
.text:64413F75          push    ecx
.text:64413F76          mov     ecx, [ebp+arg_4]

; demangled name: ATL::CStringT<char, 1>::Preallocate(int)
.text:64413F79          call    ds:mfc90_2691
.text:64413F7F          mov     [ebp+DstBuf], eax
.text:64413F82          mov     edx, [ebp+VarName]
.text:64413F85          push    edx                ; VarName
.text:64413F86          mov     eax, [ebp+DstSize]
.text:64413F89          push    eax                ; DstSize
.text:64413F8A          mov     ecx, [ebp+DstBuf]
.text:64413F8D          push    ecx                ; DstBuf
.text:64413F8E          lea     edx, [ebp+DstSize]
.text:64413F91          push    edx                ; ReturnSize
.text:64413F92          call    ds:getenv_s
.text:64413F98          add     esp, 10h
.text:64413F9B          mov     [ebp+var_8], eax
.text:64413F9E          push    0FFFFFFFFh
.text:64413FA0          mov     ecx, [ebp+arg_4]

; demangled name: ATL::CStringT::ReleaseBuffer(int)
.text:64413FA3          call    ds:mfc90_5835
.text:64413FA9          cmp     [ebp+var_8], 0
.text:64413FAD          jz      short loc_64413FB3
.text:64413FAF          xor     eax, eax
.text:64413FB1          jmp     short loc_64413FBC
.text:64413FB3
.text:64413FB3 loc_64413FB3:
.text:64413FB3          mov     ecx, [ebp+arg_4]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:64413FB6          call    ds:mfc90_910
.text:64413FBC
.text:64413FBC loc_64413FBC:
.text:64413FBC
.text:64413FBC          mov     esp, ebp
.text:64413FBE          pop     ebp
.text:64413FBF          retn
.text:64413FBF chk_env          endp

```

Да. Функция `getenv_s()`⁶ это безопасная версия функции `getenv()`⁷ в MSVC.

Тут также имеются манипуляции со строками при помощи функций из MFC.

Множество других переменных окружения также проверяются. Здесь список всех переменных проверяемых SAPGUI а также сообщение записываемое им в лог-файл, если переменная включена:

⁶MSDN

⁷Стандартная функция Си, возвращающая значение переменной окружения

80.1. КАСАТЕЛЬНО СЖИМАНИЯ СЕТЕВОГО ТРАФФИКА В КЛИЕНТЕ SAP

DPTRACE	"GUI-OPTION: Trace set to %d"
TDW_HEXDUMP	"GUI-OPTION: Hexdump enabled"
TDW_WORKDIR	"GUI-OPTION: working directory '%s'"
TDW_SPLASHSRCEENOFF	"GUI-OPTION: Splash Screen Off" / "GUI-OPTION: Splash Screen On"
TDW_REPLYTIMEOUT	"GUI-OPTION: reply timeout %d milliseconds"
TDW_PLAYBACKTIMEOUT	"GUI-OPTION: PlaybackTimeout set to %d milliseconds"
TDW_NOCOMPRESS	"GUI-OPTION: no compression read"
TDW_EXPERT	"GUI-OPTION: expert mode"
TDW_PLAYBACKPROGRESS	"GUI-OPTION: PlaybackProgress"
TDW_PLAYBACKNETTRAFFIC	"GUI-OPTION: PlaybackNetTraffic"
TDW_PLAYLOG	"GUI-OPTION: /PlayLog is YES, file %s"
TDW_PLAYTIME	"GUI-OPTION: /PlayTime set to %d milliseconds"
TDW_LOGFILE	"GUI-OPTION: TDW_LOGFILE '%s'"
TDW_WAN	"GUI-OPTION: WAN - low speed connection enabled"
TDW_FULLMENU	"GUI-OPTION: FullMenu enabled"
SAP_CP / SAP_CODEPAGE	"GUI-OPTION: SAP_CODEPAGE '%d'"
UPDOWNLOAD_CP	"GUI-OPTION: UPDOWNLOAD_CP '%d'"
SNC_PARTNERNAME	"GUI-OPTION: SNC name '%s'"
SNC_QOP	"GUI-OPTION: SNC_QOP '%s'"
SNC_LIB	"GUI-OPTION: SNC is set to: %s"
SAPGUI_INPLACE	"GUI-OPTION: environment variable SAPGUI_INPLACE is on"

Настройки для каждой переменной записываются в массив через указатель в регистре **EDI**. **EDI** выставляется перед вызовом функции :

```
.text:6440EE00      lea     edi, [ebp+2884h+var_2884] ; options here like +0x15...
.text:6440EE03      lea     ecx, [esi+24h]
.text:6440EE06      call   load_command_line
.text:6440EE0B      mov     edi, eax
.text:6440EE0D      xor     ebx, ebx
.text:6440EE0F      cmp     edi, ebx
.text:6440EE11      jz      short loc_6440EE42
.text:6440EE13      push    edi
.text:6440EE14      push    offset aSapguiStoppedA ; "Sapgui stopped after commandline ↵
        ↵ interp"...
.text:6440EE19      push    dword_644F93E8
.text:6440EE1F      call   FEWTraceError
```

А теперь, можем ли мы найти строку «*data record mode switched on*»? Да, и есть только одна ссылка на эту строку в функции

CDwsGui::PrepareInfoWindow(). Откуда мы узнали имена классов/методов? Здесь много специальных отладочных вызовов, пишущих в лог-файл вроде:

```
.text:64405160      push    dword ptr [esi+2854h]
.text:64405166      push    offset aCdwsGuiPrepare ; "\nCDwsGui::PrepareInfoWindow: sapgui ↵
        ↵ env"...
.text:6440516B      push    dword ptr [esi+2848h]
.text:64405171      call    dbg
.text:64405176      add     esp, 0Ch
```

...или:

```
.text:6440237A      push    eax
.text:6440237B      push    offset aCClientStart_6 ; "CClient::Start: set shortcut user to ↵
        ↵ '%\"/>
```

Они *очень* полезны.

Посмотрим содержимое функции «назойливого всплывающего окна» :

```
.text:64404F4F CDwsGui__PrepareInfoWindow proc near
.text:64404F4F
.text:64404F4F pvParam      = byte ptr -3Ch
.text:64404F4F var_38        = dword ptr -38h
.text:64404F4F var_34        = dword ptr -34h
.text:64404F4F rc          = tagRECT ptr -2Ch
```

```

.text:64404F4F cy          = dword ptr -1Ch
.text:64404F4F h          = dword ptr -18h
.text:64404F4F var_14     = dword ptr -14h
.text:64404F4F var_10     = dword ptr -10h
.text:64404F4F var_4      = dword ptr -4
.text:64404F4F
.text:64404F4F          push    30h
.text:64404F51          mov     eax, offset loc_64438E00
.text:64404F56          call    __EH_prolog3
.text:64404F5B          mov     esi, ecx          ; ECX is pointer to object
.text:64404F5D          xor     ebx, ebx
.text:64404F5F          lea     ecx, [ebp+var_14]
.text:64404F62          mov     [ebp+var_10], ebx

; demangled name: ATL::CStringT(void)
.text:64404F65          call    ds:mfc90_316
.text:64404F6B          mov     [ebp+var_4], ebx
.text:64404F6E          lea     edi, [esi+2854h]
.text:64404F74          push   offset aEnvironmentInf ; "Environment information:\n"
.text:64404F79          mov     ecx, edi

; demangled name: ATL::CStringT::operator=(char const *)
.text:64404F7B          call    ds:mfc90_820
.text:64404F81          cmp     [esi+38h], ebx
.text:64404F84          mov     ebx, ds:mfc90_2539
.text:64404F8A          jbe     short loc_64404FA9
.text:64404F8C          push   dword ptr [esi+34h]
.text:64404F8F          lea     eax, [ebp+var_14]
.text:64404F92          push   offset aWorkingDirecto ; "working directory: '%s'\n"
.text:64404F97          push   eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404F98          call    ebx ; mfc90_2539
.text:64404F9A          add     esp, 0Ch
.text:64404F9D          lea     eax, [ebp+var_14]
.text:64404FA0          push   eax
.text:64404FA1          mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FA3          call    ds:mfc90_941
.text:64404FA9
.text:64404FA9 loc_64404FA9:
.text:64404FA9          mov     eax, [esi+38h]
.text:64404FAC          test    eax, eax
.text:64404FAE          jbe     short loc_64404FD3
.text:64404FB0          push   eax
.text:64404FB1          lea     eax, [ebp+var_14]
.text:64404FB4          push   offset aTraceLevelDAct ; "trace level %d activated\n"
.text:64404FB9          push   eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404FBA          call    ebx ; mfc90_2539
.text:64404FBC          add     esp, 0Ch
.text:64404FBF          lea     eax, [ebp+var_14]
.text:64404FC2          push   eax
.text:64404FC3          mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FC5          call    ds:mfc90_941
.text:64404FCB          xor     ebx, ebx
.text:64404FCD          inc     ebx
.text:64404FCE          mov     [ebp+var_10], ebx
.text:64404FD1          jmp     short loc_64404FD6
.text:64404FD3
.text:64404FD3 loc_64404FD3:
.text:64404FD3          xor     ebx, ebx
.text:64404FD5          inc     ebx
.text:64404FD6
.text:64404FD6 loc_64404FD6:
.text:64404FD6          cmp     [esi+38h], ebx

```


80.1. КАСАТЕЛЬНО СЖИМАНИЯ СЕТЕВОГО ТРАФФИКА В КЛИЕНТЕ SAP

```
.text:64404FD9      jbe      short loc_64404FF1
.text:64404FDB      cmp      dword ptr [esi+2978h], 0
.text:64404FE2      jz       short loc_64404FF1
.text:64404FE4      push     offset aHexdumpInTrace ; "hexdump in trace activated\n"
.text:64404FE9      mov      ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FEB      call     ds:mfc90_945
.text:64404FF1
.text:64404FF1 loc_64404FF1:
.text:64404FF1
.text:64404FF1      cmp      byte ptr [esi+78h], 0
.text:64404FF5      jz       short loc_64405007
.text:64404FF7      push     offset aLoggingActivat ; "logging activated\n"
.text:64404FFC      mov      ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FFE      call     ds:mfc90_945
.text:64405004      mov      [ebp+var_10], ebx
.text:64405007
.text:64405007 loc_64405007:
.text:64405007      cmp      byte ptr [esi+3Dh], 0
.text:6440500B      jz       short bypass
.text:6440500D      push     offset aDataCompressio ; "data compression switched off\n"
.text:64405012      mov      ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014      call     ds:mfc90_945
.text:6440501A      mov      [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
.text:6440501D      mov      eax, [esi+20h]
.text:64405020      test     eax, eax
.text:64405022      jz       short loc_6440503A
.text:64405024      cmp      dword ptr [eax+28h], 0
.text:64405028      jz       short loc_6440503A
.text:6440502A      push     offset aDataRecordMode ; "data record mode switched on\n"
.text:6440502F      mov      ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405031      call     ds:mfc90_945
.text:64405037      mov      [ebp+var_10], ebx
.text:6440503A
.text:6440503A loc_6440503A:
.text:6440503A
.text:6440503A      mov      ecx, edi
.text:6440503C      cmp      [ebp+var_10], ebx
.text:6440503F      jnz      loc_64405142
.text:64405045      push     offset aForMaximumData ; "\nFor maximum data security delete\
    ↵ nthe s"...

; demangled name: ATL::CStringT::operator+=(char const *)
.text:6440504A      call     ds:mfc90_945
.text:64405050      xor      edi, edi
.text:64405052      push     edi ; fWinIni
.text:64405053      lea      eax, [ebp+pvParam]
.text:64405056      push     eax ; pvParam
.text:64405057      push     edi ; uiParam
.text:64405058      push     30h ; uiAction
.text:6440505A      call     ds:SystemParametersInfoA
.text:64405060      mov      eax, [ebp+var_34]
.text:64405063      cmp      eax, 1600
.text:64405068      jle      short loc_64405072
.text:6440506A      cdq
.text:6440506B      sub      eax, edx
.text:6440506D      sar      eax, 1
.text:6440506F      mov      [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:
.text:64405072      push     edi ; hWnd
```

80.1. КАСАТЕЛЬНО СЖИМАНИЯ СЕТЕВОГО ТРАФФИКА В КЛИЕНТЕ SAP

```
.text:64405073      mov     [ebp+cy], 0A0h
.text:6440507A      call    ds:GetDC
.text:64405080      mov     [ebp+var_10], eax
.text:64405083      mov     ebx, 12Ch
.text:64405088      cmp     eax, edi
.text:6440508A      jz      loc_64405113
.text:64405090      push    11h          ; i
.text:64405092      call    ds:GetStockObject
.text:64405098      mov     edi, ds:SelectObject
.text:6440509E      push    eax          ; h
.text:6440509F      push    [ebp+var_10] ; hdc
.text:644050A2      call    edi ; SelectObject
.text:644050A4      and     [ebp+rc.left], 0
.text:644050A8      and     [ebp+rc.top], 0
.text:644050AC      mov     [ebp+h], eax
.text:644050AF      push    401h          ; format
.text:644050B4      lea     eax, [ebp+rc]
.text:644050B7      push    eax          ; lprc
.text:644050B8      lea     ecx, [esi+2854h]
.text:644050BE      mov     [ebp+rc.right], ebx
.text:644050C1      mov     [ebp+rc.bottom], 0B4h

; demangled name: ATL::CStringT::GetLength(void)
.text:644050C8      call    ds:mfc90_3178
.text:644050CE      push    eax          ; cchText
.text:644050CF      lea     ecx, [esi+2854h]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:644050D5      call    ds:mfc90_910
.text:644050DB      push    eax          ; lpchText
.text:644050DC      push    [ebp+var_10] ; hdc
.text:644050DF      call    ds:DrawTextA
.text:644050E5      push    4            ; nIndex
.text:644050E7      call    ds:GetSystemMetrics
.text:644050ED      mov     ecx, [ebp+rc.bottom]
.text:644050F0      sub     ecx, [ebp+rc.top]
.text:644050F3      cmp     [ebp+h], 0
.text:644050F7      lea     eax, [eax+ecx+28h]
.text:644050FB      mov     [ebp+cy], eax
.text:644050FE      jz      short loc_64405108
.text:64405100      push    [ebp+h]      ; h
.text:64405103      push    [ebp+var_10] ; hdc
.text:64405106      call    edi ; SelectObject
.text:64405108
.text:64405108 loc_64405108:
.text:64405108      push    [ebp+var_10] ; hDC
.text:6440510B      push    0            ; hWnd
.text:6440510D      call    ds:ReleaseDC
.text:64405113
.text:64405113 loc_64405113:
.text:64405113      mov     eax, [ebp+var_38]
.text:64405116      push    80h          ; uFlags
.text:6440511B      push    [ebp+cy]      ; cy
.text:6440511E      inc     eax
.text:6440511F      push    ebx          ; cx
.text:64405120      push    eax          ; Y
.text:64405121      mov     eax, [ebp+var_34]
.text:64405124      add     eax, 0FFFFFFD4h
.text:64405129      cdq
.text:6440512A      sub     eax, edx
.text:6440512C      sar     eax, 1
.text:6440512E      push    eax          ; X
.text:6440512F      push    0            ; hWndInsertAfter
.text:64405131      push    dword ptr [esi+285Ch] ; hWnd
.text:64405137      call    ds:SetWindowPos
.text:6440513D      xor     ebx, ebx
.text:6440513F      inc     ebx
.text:64405140      jmp     short loc_6440514D
.text:64405142
.text:64405142 loc_64405142:
```

80.1. КАСАТЕЛЬНО СЖИМАНИЯ СЕТЕВОГО ТРАФФИКА В КЛИЕНТЕ SAP

```
.text:64405142      push     offset byte_64443AF8

; demangled name: ATL::CStringT::operator=(char const *)
.text:64405147      call     ds:mfc90_820
.text:6440514D
.text:6440514D loc_6440514D:
.text:6440514D      cmp     dword_6450B970, ebx
.text:64405153      jnl     short loc_64405188
.text:64405155      call     sub_6441C910
.text:6440515A      mov     dword_644F858C, ebx
.text:64405160      push    dword ptr [esi+2854h]
.text:64405166      push    offset aCdwsguiPrepare ; "\nCDwsGui::PrepareInfoWindow: sapgui ↵
        ↵ env"...
.text:6440516B      push    dword ptr [esi+2848h]
.text:64405171      call     dbg
.text:64405176      add     esp, 0Ch
.text:64405179      mov     dword_644F858C, 2
.text:64405183      call     sub_6441C920
.text:64405188
.text:64405188 loc_64405188:
.text:64405188      or      [ebp+var_4], 0FFFFFFFh
.text:6440518C      lea     ecx, [ebp+var_14]

; demangled name: ATL::CStringT::~CStringT()
.text:6440518F      call     ds:mfc90_601
.text:64405195      call     __EH_epilog3
.text:6440519A      retn
.text:6440519A CDwsGui__PrepareInfoWindow endp
```

ECX в начале функции содержит в себе указатель на объект (потому что это тип функции thiscall (52.1.1 (стр. 539))). В нашем случае, класс имеет тип, очевидно, *CDwsGui*. В зависимости от включенных опций в объекте, разные сообщения добавляются к итоговому сообщению.

Если переменная по адресу `this+0x3D` не ноль, компрессия сетевых пакетов будет выключена :

```
.text:64405007 loc_64405007:
.text:64405007      cmp     byte ptr [esi+3Dh], 0
.text:6440500B      jz      short bypass
.text:6440500D      push    offset aDataCompressio ; "data compression switched off\n"
.text:64405012      mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014      call     ds:mfc90_945
.text:6440501A      mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
```

Интересно, что в итоге, состояние переменной `var_10` определяет, будет ли показано сообщение вообще :

```
.text:6440503C      cmp     [ebp+var_10], ebx
.text:6440503F      jnz     exit ; пропустить отрисовку

; добавить строки "For maximum data security delete" / "the setting(s) as soon as possible !":

.text:64405045      push    offset aForMaximumData ; "\nFor maximum data security delete\↵
        ↵ nthe s"...
.text:6440504A      call     ds:mfc90_945 ; ATL::CStringT::operator+=(char const *)
.text:64405050      xor     edi, edi
.text:64405052      push    edi           ; fWinIni
.text:64405053      lea     eax, [ebp+pvParam]
.text:64405056      push    eax           ; pvParam
.text:64405057      push    edi           ; uiParam
.text:64405058      push    30h           ; uiAction
.text:6440505A      call     ds:SystemParametersInfoA
.text:64405060      mov     eax, [ebp+var_34]
.text:64405063      cmp     eax, 1600
.text:64405068      jle     short loc_64405072
.text:6440506A      cdq
.text:6440506B      sub     eax, edx
.text:6440506D      sar     eax, 1
```

80.1. КАСАТЕЛЬНО СЖИМАНИЯ СЕТЕВОГО ТРАФФИКА В КЛИЕНТЕ SAP

```
.text:6440506F      mov     [ebp+var_34], eax
.text:64405072
.text:64405072  loc_64405072:

начинает рисовать:

.text:64405072      push    edi                ; hWnd
.text:64405073      mov     [ebp+cy], 0A0h
.text:6440507A      call    ds:GetDC
```

Давайте проверим нашу теорию на практике.

JNZ в этой строке ...

```
.text:6440503F      jnz     exit ; пропустить отрисовку
```

...заменим просто на **JMP** и получим SAPGUI работающим без этого назойливого всплывающего окна!

Копнем немного глубже и проследим связь между смещением 0x15 в `load_command_line()` (Это мы дали имя этой функции) и переменной `this+0x3D` в `CDwsGui::PrepareInfoWindow`. Уверены ли мы что это одна и та же переменная?

Начинаем искать все места где в коде используется константа `0x15`. Для таких небольших программ как SAPGUI, это иногда срабатывает. Вот первое что находим:

```
.text:64404C19  sub_64404C19  proc near
.text:64404C19
.text:64404C19  arg_0        = dword ptr 4
.text:64404C19
.text:64404C19      push    ebx
.text:64404C1A      push    ebp
.text:64404C1B      push    esi
.text:64404C1C      push    edi
.text:64404C1D      mov     edi, [esp+10h+arg_0]
.text:64404C21      mov     eax, [edi]
.text:64404C23      mov     esi, ecx ; ESI/ECX are pointers to some unknown object.
.text:64404C25      mov     [esi], eax
.text:64404C27      mov     eax, [edi+4]
.text:64404C2A      mov     [esi+4], eax
.text:64404C2D      mov     eax, [edi+8]
.text:64404C30      mov     [esi+8], eax
.text:64404C33      lea     eax, [edi+0Ch]
.text:64404C36      push    eax
.text:64404C37      lea     ecx, [esi+0Ch]

; demangled name: ATL::CStringT::operator=(class ATL::CStringT ... &)
.text:64404C3A      call    ds:mfc90_817
.text:64404C40      mov     eax, [edi+10h]
.text:64404C43      mov     [esi+10h], eax
.text:64404C46      mov     al, [edi+14h]
.text:64404C49      mov     [esi+14h], al
.text:64404C4C      mov     al, [edi+15h] ; copy byte from 0x15 offset
.text:64404C4F      mov     [esi+15h], al ; to 0x15 offset in CDwsGui object
```

Эта функция вызывается из функции с названием `CDwsGui::CopyOptions`! И снова спасибо отладочной информации.

Но настоящий ответ находится в функции `CDwsGui::Init()`:

```
.text:6440B0BF  loc_6440B0BF:
.text:6440B0BF      mov     eax, [ebp+arg_0]
.text:6440B0C2      push    [ebp+arg_4]
.text:6440B0C5      mov     [esi+284h], eax
.text:6440B0CB      lea     eax, [esi+28h] ; ESI is pointer to CDwsGui object
.text:6440B0CE      push    eax
.text:6440B0CF      call    CDwsGui__CopyOptions
```

Теперь ясно: массив заполняемый в `load_command_line()` на самом деле расположен в классе `CDwsGui` но по адресу `this+0x28`. `0x15 + 0x28` это `0x3D`. ОК, мы нашли место, куда наша переменная копируется.

Посмотрим также и другие места, где используется смещение `0x3D`. Одно из таких мест находится в функции `CDwsGui::SapguiRun` (и снова спасибо отладочным вызовам):

```
.text:64409D58      cmp     [esi+3Dh], bl    ; ESI is pointer to CDwsGui object
.text:64409D5B      lea     ecx, [esi+2B8h]
.text:64409D61      setz    al
.text:64409D64      push    eax              ; arg_10 of CConnectionContext::CreateNetwork
.text:64409D65      push    dword ptr [esi+64h]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:64409D68      call    ds:mfc90_910
.text:64409D68      ; no arguments
.text:64409D6E      push    eax
.text:64409D6F      lea     ecx, [esi+2BCh]

; demangled name: const char* ATL::CStringT::operator PCWSTR
.text:64409D75      call    ds:mfc90_910
.text:64409D75      ; no arguments
.text:64409D7B      push    eax
.text:64409D7C      push    esi
.text:64409D7D      lea     ecx, [esi+8]
.text:64409D80      call    CConnectionContext__CreateNetwork
```

Проверим нашу идею. Заменяем `setz al` здесь на `xor eax, eax / nop`, убираем переменную окружения `TDW_NOCOMPRESS` и запускаем SAPGUI. Ух! Назойливого окна больше нет (как и ожидалось: ведь переменной окружения также нет), но в Wireshark мы видим, что сетевые пакеты больше не сжимаются! Очевидно, это то самое место где флаг отражающий сжатие пакетов выставляется в объекте `CConnectionContext`.

Так что, флаг сжатия передается в пятом аргументе функции `CConnectionContext::CreateNetwork`. Внутри этой функции, вызывается еще одна:

```
...
.text:64403476      push    [ebp+compression]
.text:64403479      push    [ebp+arg_C]
.text:6440347C      push    [ebp+arg_8]
.text:6440347F      push    [ebp+arg_4]
.text:64403482      push    [ebp+arg_0]
.text:64403485      call    CNetwork__CNetwork
```

Флаг отвечающий за сжатие здесь передается в пятом аргументе для конструктора `CNetwork::CNetwork`.

И вот как конструктор `CNetwork` выставляет некоторые флаги в объекте `CNetwork` в соответствии с пятым аргументом и еще какую-то переменную, возможно, также отвечающую за сжатие сетевых пакетов.

```
.text:64411DF1      cmp     [ebp+compression], esi
.text:64411DF7      jz      short set_EAX_to_0
.text:64411DF9      mov     al, [ebx+78h]    ; another value may affect compression?
.text:64411DFC      cmp     al, '3'
.text:64411DFE      jz      short set_EAX_to_1
.text:64411E00      cmp     al, '4'
.text:64411E02      jnz     short set_EAX_to_0
.text:64411E04      set_EAX_to_1:
.text:64411E04      xor     eax, eax
.text:64411E06      inc     eax              ; EAX -> 1
.text:64411E07      jmp     short loc_64411E0B
.text:64411E09      set_EAX_to_0:
.text:64411E09      xor     eax, eax
.text:64411E09      ; EAX -> 0
.text:64411E0B      loc_64411E0B:
.text:64411E0B      mov     [ebx+3A4h], eax ; EBX is pointer to CNetwork object
```

Теперь мы знаем, что флаг отражающий сжатие данных сохраняется в классе `CNetwork` по адресу `this+0x3A4`.

Поискем теперь значение `0x3A4` в `SAPguilib.dll`. Находим второе упоминание этого значения в функции `CDwsGui::OnClientMessage` (бесконечная благодарность отладочной информации):

```
.text:64406F76      loc_64406F76:
.text:64406F76      mov     ecx, [ebp+7728h+var_7794]
.text:64406F79      cmp     dword ptr [ecx+3A4h], 1
```

80.2. ФУНКЦИИ ПРОВЕРКИ ПАРОЛЯ В SAP 6.0

```
.text:64406F80      jnz      compression_flag_is_zero
.text:64406F86      mov      byte ptr [ebx+7], 1
.text:64406F8A      mov      eax, [esi+18h]
.text:64406F8D      mov      ecx, eax
.text:64406F8F      test     eax, eax
.text:64406F91      ja       short loc_64406FFF
.text:64406F93      mov      ecx, [esi+14h]
.text:64406F96      mov      eax, [esi+20h]
.text:64406F99      loc_64406F99:
.text:64406F99      push     dword ptr [edi+2868h] ; int
.text:64406F9F      lea      edx, [ebp+7728h+var_77A4]
.text:64406FA2      push     edx ; int
.text:64406FA3      push     30000 ; int
.text:64406FA8      lea      edx, [ebp+7728h+Dst]
.text:64406FAB      push     edx ; Dst
.text:64406FAC      push     ecx ; int
.text:64406FAD      push     eax ; Src
.text:64406FAE      push     dword ptr [edi+28C0h] ; int
.text:64406FB4      call     sub_644055C5 ; actual compression routine
.text:64406FB9      add      esp, 1Ch
.text:64406FBC      cmp      eax, 0FFFFFFF6h
.text:64406FBF      jz       short loc_64407004
.text:64406FC1      cmp      eax, 1
.text:64406FC4      jz       loc_6440708C
.text:64406FCA      cmp      eax, 2
.text:64406FCD      jz       short loc_64407004
.text:64406FCF      push     eax
.text:64406FD0      push     offset aCompressionErr ; "compression error [rc = %d]- program
        ↪ wi"...
.text:64406FD5      push     offset aGui_err_compre ; "GUI_ERR_COMPRESS"
.text:64406FDA      push     dword ptr [edi+28D0h]
.text:64406FE0      call     SapPcTxtRead
```

Заглянем в функцию `sub_644055C5`. Всё что в ней мы находим это вызов `memcmp()` и еще какую-то функцию, названную `IDA sub_64417440`.

И теперь заглянем в `sub_64417440`. Увидим там:

```
.text:6441747C      push     offset aErrorCsrcompre ; "\nERROR: CsRCompress: invalid handle"
.text:64417481      call     eax ; dword_644F94C8
.text:64417483      add      esp, 4
```

Voilà! Мы находим функцию которая собственно и сжимает сетевые пакеты. Как уже было видно в прошлом ⁸, эта функция используется в SAP и в open-сорсном проекте MaxDB. Так что эта функция доступна в виде исходников.

Последняя проверка:

```
.text:64406F79      cmp      dword ptr [ecx+3A4h], 1
.text:64406F80      jnz      compression_flag_is_zero
```

Заменяем `JNZ` на безусловный переход `JMP`. Уберем переменную окружения `TDW_NOCOMPRESS`. Вуаля! В Wireshark мы видим, что сетевые пакеты, исходящие от клиента, не сжаты. Ответы сервера, впрочем, сжаты.

Так что мы нашли связь между переменной окружения и местом где функция сжатия данных вызывается, а также может быть отключена.

80.2. Функции проверки пароля в SAP 6.0

Когда автор этой книги в очередной раз вернулся к своему SAP 6.0 IDES установленному в виртуальной машине VMware, он обнаружил что забыл пароль, впрочем, затем он вспомнил его, но теперь получаем такую ошибку: *«Password logon no longer possible - too many failed attempts»*, потому что были потрачены все попытки на то, чтобы вспомнить его.

Первая очень хорошая новость состоит в том, что с SAP поставляется полный `PDB`-файл `disp+work.pdb`, он содержит все: имена функций, структуры, типы, локальные переменные, имена аргументов, и т.д. Какой щедрый подарок!

⁸<http://go.yurichev.com/17312>

80.2. ФУНКЦИИ ПРОВЕРКИ ПАРОЛЯ В SAP 6.0

Существует утилита TYPEINFODUMP⁹ для дампа содержимого PDB-файлов во что-то более читаемое и grep-абельное.

Вот пример её работы: информация о функции + её аргументах + её локальных переменных:

```
FUNCTION ThVmcsysEvent
  Address:      10143190  Size:      675 bytes  Index:      60483  TypeIndex:      60484
  Type: int NEAR_C ThVmcsysEvent (unsigned int, unsigned char, unsigned short*)
Flags: 0
PARAMETER events
  Address: Reg335+288  Size:      4 bytes  Index:      60488  TypeIndex:      60489
  Type: unsigned int
Flags: d0
PARAMETER opcode
  Address: Reg335+296  Size:      1 bytes  Index:      60490  TypeIndex:      60491
  Type: unsigned char
Flags: d0
PARAMETER serverName
  Address: Reg335+304  Size:      8 bytes  Index:      60492  TypeIndex:      60493
  Type: unsigned short*
Flags: d0
STATIC_LOCAL_VAR func
  Address:      12274af0  Size:      8 bytes  Index:      60495  TypeIndex:      60496
  Type: wchar_t*
Flags: 80
LOCAL_VAR admhead
  Address: Reg335+304  Size:      8 bytes  Index:      60498  TypeIndex:      60499
  Type: unsigned char*
Flags: 90
LOCAL_VAR record
  Address: Reg335+64   Size:     204 bytes  Index:      60501  TypeIndex:      60502
  Type: AD_RECORD
Flags: 90
LOCAL_VAR adlen
  Address: Reg335+296  Size:      4 bytes  Index:      60508  TypeIndex:      60509
  Type: int
Flags: 90
```

А вот пример дампа структуры:

```
STRUCT DBSL_STMTID
Size: 120 Variables: 4 Functions: 0 Base classes: 0
MEMBER moduletype
  Type: DBSL_MODULETYPE
  Offset:      0  Index:      3  TypeIndex:      38653
MEMBER module
  Type: wchar_t module[40]
  Offset:      4  Index:      3  TypeIndex:      831
MEMBER stmtnum
  Type: long
  Offset:      84  Index:      3  TypeIndex:      440
MEMBER timestamp
  Type: wchar_t timestamp[15]
  Offset:      88  Index:      3  TypeIndex:      6612
```

Bay!

Вторая хорошая новость: *отладочные* вызовы, коих здесь очень много, очень полезны.

Здесь вы можете увидеть глобальную переменную *ct_level*¹⁰, отражающую уровень трассировки.

В *disp+work.exe* очень много таких отладочных вставок:

```
cmp     cs:ct_level, 1
jl      short loc_1400375DA
call    DpLock
lea     rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov     edx, 4Eh          ; line
call    CTrcSaveLocation
mov     r8, cs:func_48
```

⁹<http://go.yurichev.com/17038>

¹⁰Еще об уровне трассировки: <http://go.yurichev.com/17039>

80.2. ФУНКЦИИ ПРОВЕРКИ ПАРОЛЯ В SAP 6.0

```
mov     rcx, cs:hdl      ; hdl
lea     rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov     r9d, ebx
call    DpTrcErr
call    DpUnlock
```

Если текущий уровень трассировки выше или равен заданному в этом коде порогу, отладочное сообщение будет записано в лог-файл вроде *dev_w0*, *dev_disp* и прочие файлы *dev**.

Попробуем grep-ать файл недавно полученный при помощи утилиты TYPEINFODUMP:

```
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

Получаем:

```
FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
FUNCTION password_is_initial
FUNCTION rcui::AgiPassword::IsVisible
FUNCTION password_distance_ok
FUNCTION get_password_downwards_compatibility
FUNCTION dySignUnSkipUserPassword
FUNCTION rcui::AgiPassword::GetTypeNames
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$2
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$0
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$1
FUNCTION usm_set_password
FUNCTION rcui::AgiPassword::TraceTo
FUNCTION days_since_last_password_change
FUNCTION rsecgrp_generate_random_password
FUNCTION rcui::AgiPassword::`scalar deleting destructor'
FUNCTION password_attempt_limit_exceeded
FUNCTION handle_incorrect_password
FUNCTION `rcui::AgiPassword::`scalar deleting destructor'::`1'::dtor$1
FUNCTION calculate_new_password_hash
FUNCTION shift_password_to_history
FUNCTION rcui::AgiPassword::GetType
FUNCTION found_password_in_history
FUNCTION `rcui::AgiPassword::`scalar deleting destructor'::`1'::dtor$0
FUNCTION rcui::AgiObj::IsaPassword
FUNCTION password_idle_check
FUNCTION SlicHwPasswordForDay
FUNCTION rcui::AgiPassword::IsaPassword
FUNCTION rcui::AgiPassword::AgiPassword
FUNCTION delete_user_password
FUNCTION usm_set_user_password
FUNCTION Password_API
FUNCTION get_password_change_for_SSO
FUNCTION password_in_USR40
FUNCTION rsec_agrp_abap_generate_random_password
```

Попробуем так же искать отладочные сообщения содержащие слова «password» и «locked». Одна из таких это строка «user was locked by subsequently failed password logon attempts» на которую есть ссылка в функции *password_attempt_limit_exceeded()*.

Другие строки, которые эта найденная функция может писать в лог-файл это : «password logon attempt will be rejected immediately (preventing dictionary attacks)», «failed-logon lock: expired (but not removed due to 'read-only' operation)», «failed-logon lock: expired => removed».

Немного поэкспериментировав с этой функцией, мы быстро понимаем что проблема именно в ней. Она вызывается из функции *chkpass()* — одна из функций проверяющих пароль.

В начале, давайте убедимся, что мы на верном пути:

Запускаем [tracer](#):

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chkpass,args:3,unicode
```



```
PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L"Brewered1", 0x41) (called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2236|TID=2248|(0) disp+work.exe!chckpass -> 0x35
```

Функции вызываются так: *syssigni()* -> *DylSigni()* -> *dychkurs()* -> *usrexist()* -> *chckpass()*.

Число 0x35 возвращается из *chckpass()* в этом месте:

```
.text:00000001402ED567 loc_1402ED567: ; CODE XREF: chckpass+B4
.text:00000001402ED567 mov rcx, rbx ; usr02
.text:00000001402ED56A call password_idle_check
.text:00000001402ED56F cmp eax, 33h
.text:00000001402ED572 jz loc_1402EDB4E
.text:00000001402ED578 cmp eax, 36h
.text:00000001402ED57B jz loc_1402EDB3D
.text:00000001402ED581 xor edx, edx ; usr02_readonly
.text:00000001402ED583 mov rcx, rbx ; usr02
.text:00000001402ED586 call password_attempt_limit_exceeded
.text:00000001402ED58B test al, al
.text:00000001402ED58D jz short loc_1402ED5A0
.text:00000001402ED58F mov eax, 35h
.text:00000001402ED594 add rsp, 60h
.text:00000001402ED598 pop r14
.text:00000001402ED59A pop r12
.text:00000001402ED59C pop rdi
.text:00000001402ED59D pop rsi
.text:00000001402ED59E pop rbx
.text:00000001402ED59F ret
```

Отлично, давайте проверим:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!password_attempt_limit_exceeded,args:4,unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0x257758, 0) (called from 0x1402ed58b (disp+work.exe!chckpass+0xeb))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0, 0) (called from 0x1402e9794 (disp+work.exe!chnghpass+0xe4))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Великолепно! Теперь мы можем успешно залогиниться.

Кстати, мы можем сделать вид что вообще забыли пароль, заставляя *chckpass()* всегда возвращать ноль, и этого достаточно для отключения проверки пароля:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!chckpass (0x202c770, L"bogus", 0x41) (called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2744|TID=360|(0) disp+work.exe!chckpass -> 0x35
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Что еще можно сказать, бегло анализируя функцию *password_attempt_limit_exceeded()*, это то, что в начале можно увидеть следующий вызов:

```
lea rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call sapgparam
test rax, rax
jz short loc_1402E19DE
movzx eax, word ptr [rax]
cmp ax, 'N'
jz short loc_1402E19D4
cmp ax, 'n'
jz short loc_1402E19D4
cmp ax, '0'
jnz short loc_1402E19DE
```

80.2. ФУНКЦИИ ПРОВЕРКИ ПАРОЛЯ В SAP 6.0

Очевидно, функция *sapgpam()* используется чтобы узнать значение какой-либо переменной конфигурации. Эта функция может вызываться из 1768 разных мест. Вероятно, при помощи этой информации, мы можем легко находить те места кода, на которые влияют определенные переменные конфигурации.

Замечательно! Имена функций очень понятны, куда понятнее чем в Oracle RDBMS. По всей видимости, процесс *disp+work* весь написан на Си++. Вероятно, он был переписан не так давно?

Глава 81

Oracle RDBMS

81.1. Таблица V\$VERSION в Oracle RDBMS

Oracle RDBMS 11.2 это очень большая программа, основной модуль `oracle.exe` содержит около 124 тысячи функций. Для сравнения, ядро Windows 7 x64 (`ntoskrnl.exe`) — около 11 тысяч функций, а ядро Linux 3.9.8 (с драйверами по умолчанию) — 31 тысяч функций.

Начнем с одного простого вопроса. Откуда Oracle RDBMS берет информацию, когда мы в SQL*Plus пишем вот такой вот простой запрос:

```
SQL> select * from V$VERSION;
```

И получаем:

BANNER

```
-----  
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 – Production  
PL/SQL Release 11.2.0.1.0 – Production  
CORE      11.2.0.1.0      Production  
TNS for 32-bit Windows: Version 11.2.0.1.0 – Production  
NLSRTL Version 11.2.0.1.0 – Production
```

Начнем. Где в самом Oracle RDBMS мы можем найти строку `V$VERSION` ?

Для win32-версии, эта строка имеется в файле `oracle.exe`, это легко увидеть. Но мы также можем использовать объектные (.o) файлы от версии Oracle RDBMS для Linux, потому что в них сохраняются имена функций и глобальных переменных, а в `oracle.exe` для win32 этого нет.

Итак, строка `V$VERSION` имеется в файле `kqf.o`, в самой главной Oracle-библиотеке `libserver11.a`.

Ссылка на эту текстовую строку имеется в таблице `kqfvw`, размещенной в этом же файле `kqf.o`:

Листинг 81.1: kqf.o

```
.rodata:0800C4A0 kqfvw      dd 0Bh                      ; DATA XREF: kqfchk:loc_8003A6D  
.rodata:0800C4A0          ; kqfgbn+34  
.rodata:0800C4A4          dd offset _2__STRING_10102_0 ; "GV$WAITSTAT"  
.rodata:0800C4A8          dd 4  
.rodata:0800C4AC          dd offset _2__STRING_10103_0 ; "NULL"  
.rodata:0800C4B0          dd 3  
.rodata:0800C4B4          dd 0  
.rodata:0800C4B8          dd 195h  
.rodata:0800C4BC          dd 4  
.rodata:0800C4C0          dd 0  
.rodata:0800C4C4          dd 0FFFFFFC1CBh  
.rodata:0800C4C8          dd 3  
.rodata:0800C4CC          dd 0  
.rodata:0800C4D0          dd 0Ah  
.rodata:0800C4D4          dd offset _2__STRING_10104_0 ; "V$WAITSTAT"  
.rodata:0800C4D8          dd 4  
.rodata:0800C4DC          dd offset _2__STRING_10103_0 ; "NULL"
```

81.1. ТАБЛИЦА V\$VERSION В ORACLE RDBMS

```
.rodata:0800C4E0      dd 3
.rodata:0800C4E4      dd 0
.rodata:0800C4E8      dd 4Eh
.rodata:0800C4EC      dd 3
.rodata:0800C4F0      dd 0
.rodata:0800C4F4      dd 0FFFFFFC003h
.rodata:0800C4F8      dd 4
.rodata:0800C4FC      dd 0
.rodata:0800C500      dd 5
.rodata:0800C504      dd offset _2__STRING_10105_0 ; "GV$BH"
.rodata:0800C508      dd 4
.rodata:0800C50C      dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C510      dd 3
.rodata:0800C514      dd 0
.rodata:0800C518      dd 269h
.rodata:0800C51C      dd 15h
.rodata:0800C520      dd 0
.rodata:0800C524      dd 0FFFFFFC1EDh
.rodata:0800C528      dd 8
.rodata:0800C52C      dd 0
.rodata:0800C530      dd 4
.rodata:0800C534      dd offset _2__STRING_10106_0 ; "V$BH"
.rodata:0800C538      dd 4
.rodata:0800C53C      dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C540      dd 3
.rodata:0800C544      dd 0
.rodata:0800C548      dd 0F5h
.rodata:0800C54C      dd 14h
.rodata:0800C550      dd 0
.rodata:0800C554      dd 0FFFFFFC1EEh
.rodata:0800C558      dd 5
.rodata:0800C55C      dd 0
```

Кстати, нередко, при изучении внутренностей Oracle RDBMS, появляется вопрос, почему имена функций и глобальных переменных такие странные. Вероятно, дело в том, что Oracle RDBMS очень старый продукт сам по себе и писался на Си еще в 1980-х. А в те времена стандарт Си гарантировал поддержку имен переменных длиной только до шести символов включительно: «6 significant initial characters in an external identifier»¹

Вероятно, таблица `kqfviw` содержащая в себе многие (а может даже и все) view с префиксом V\$, это служебные view (fixed views), присутствующие всегда. Бегло оценив цикличность данных, мы легко видим, что в каждом элементе таблицы `kqfviw` 12 полей 32-битных полей. В IDA легко создать структуру из 12-и элементов и применить её ко всем элементам таблицы. Для версии Oracle RDBMS 11.2, здесь 1023 элемента в таблице, то есть, здесь описываются 1023 всех возможных *fixed view*. Позже, мы еще вернемся к этому числу.

Как видно, мы не очень много можем узнать чисел в этих полях. Самое первое число всегда равно длине строки-названия view (без терминирующего поля). Это справедливо для всех элементов. Но эта информация не очень полезна.

Мы также знаем, что информацию обо всех fixed views можно получить из *fixed view* под названием `V$FIXED_VIEW_DEFINITION` (кстати, информация для этого view также берется из таблиц `kqfviw` и `kqfvip`). Кстати, там тоже 1023 элемента. Совпадение? Нет.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';
```

```
VIEW_NAME
```

```
VIEW_DEFINITION
```

```
V$VERSION
```

```
select BANNER from GV$VERSION where inst_id = USERENV('Instance')
```

Итак, `V$VERSION` это как бы *thunk view* для другого, с названием `GV$VERSION`, который, в свою очередь:

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$VERSION';
```

```
VIEW_NAME
```

¹Draft ANSI C Standard (ANSI X3J11/88-090) (May 13, 1988) (yurichev.com)

81.1. ТАБЛИЦА V\$VERSION В ORACLE RDBMS

VIEW_DEFINITION

GV\$VERSION

```
select inst_id, banner from x$version
```

Таблицы с префиксом X\$ в Oracle RDBMS — это также служебные таблицы, они не документированы, не могут изменяться пользователем, и обновляются динамически.

Попробуем поискать текст `select BANNER from GV$VERSION where inst_id = USERENV('Instance')` в файле `kqf.o` и находим ссылку на него в таблице `kqfvip`:

Листинг 81.2: kqf.o

```
rodata:080185A0 kqfvip          dd offset _2__STRING_11126_0 ; DATA XREF: kqfgvcn+18
.rodata:080185A0                                ; kqfgvt+F
.rodata:080185A0                                ; "select inst_id,decode(indx,1,'data bloc
↳ "...
.rodata:080185A4          dd offset kqfv459_c_0
.rodata:080185A8          dd 0
.rodata:080185AC          dd 0
...
.rodata:08019570          dd offset _2__STRING_11378_0 ; "select  BANNER from GV$VERSION where ↵
↳ in"...
.rodata:08019574          dd offset kqfv133_c_0
.rodata:08019578          dd 0
.rodata:0801957C          dd 0
.rodata:08019580          dd offset _2__STRING_11379_0 ; "select inst_id,decode(bitand(cfflg,1)↵
↳ ,0"...
.rodata:08019584          dd offset kqfv403_c_0
.rodata:08019588          dd 0
.rodata:0801958C          dd 0
.rodata:08019590          dd offset _2__STRING_11380_0 ; "select  STATUS , NAME, ↵
↳ IS_RECOVERY_DEST"...
.rodata:08019594          dd offset kqfv199_c_0
```

Таблица, по всей видимости, имеет 4 поля в каждом элементе. Кстати, здесь так же 1023 элемента, уже знакомое нам число. Второе поле указывает на другую таблицу, содержащую поля этого *fixed view*. Для `V$VERSION`, эта таблица только из двух элементов, первый это 6 и второй это строка `BANNER` (число 6 это длина строки) и далее *терминирующий* элемент содержащий 0 и нулевую Си-строку:

Листинг 81.3: kqf.o

```
.rodata:080BBAC4 kqfv133_c_0    dd 6                                ; DATA XREF: .rodata:08019574
.rodata:080BBAC8          dd offset _2__STRING_5017_0 ; "BANNER"
.rodata:080BBACC          dd 0
.rodata:080BBAD0          dd offset _2__STRING_0_0
```

Объединив данные из таблиц `kqfviw` и `kqfvip`, мы получим SQL-запросы, которые исполняются, когда пользователь хочет получить информацию из какого-либо *fixed view*.

Напишем программу `oracle tables`², которая собирает всю эту информацию из объектных файлов от Oracle RDBMS под Linux. Для `V$VERSION`, мы можем найти следующее:

Листинг 81.4: Результат работы oracle tables

```
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xffffc085 0x4
kqfvip_element.statement: [select  BANNER from GV$VERSION where inst_id = USERENV('Instance')]
kqfvip_element.params:
[BANNER]
```

и:

Листинг 81.5: Результат работы oracle tables

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xffffc192 0x1
```

²yurichev.com

81.1. ТАБЛИЦА V\$VERSION В ORACLE RDBMS

```
kqfvip_element.statement: [select inst_id, banner from x$version]
kqfvip_element.params:
[INST_ID] [BANNER]
```

Fixed view **GV\$VERSION** отличается от **V\$VERSION** тем, что содержит еще и поле отражающее идентификатор *instance*. Но так или иначе, мы теперь упираемся в таблицу **X\$VERSION**. Как и прочие X\$-таблицы, она не документирована, однако, мы можем отсюда что-то прочесть:

```
SQL> select * from x$version;

ADDR          INDX    INST_ID
-----
BANNER
-----

ODBAF574          0          1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
...
```

Эта таблица содержит дополнительные поля вроде **ADDR** и **INDX**.

Бегло листая содержимое файла **kqf.o** в **IDA** мы можем увидеть еще одну таблицу где есть ссылка на строку **X\$VERSION**, это **kqftab**:

Листинг 81.6: kqf.o

```
.rodata:0803CAC0      dd 9                ; element number 0x1f6
.rodata:0803CAC4      dd offset _2__STRING_13113_0 ; "X$VERSION"
.rodata:0803CAC8      dd 4
.rodata:0803CACC      dd offset _2__STRING_13114_0 ; "kqvt"
.rodata:0803CAD0      dd 4
.rodata:0803CAD4      dd 4
.rodata:0803CAD8      dd 0
.rodata:0803CADC      dd 4
.rodata:0803CAE0      dd 0Ch
.rodata:0803CAE4      dd 0FFFFC075h
.rodata:0803CAE8      dd 3
.rodata:0803CAEC      dd 0
.rodata:0803CAF0      dd 7
.rodata:0803CAF4      dd offset _2__STRING_13115_0 ; "X$KQFSZ"
.rodata:0803CAF8      dd 5
.rodata:0803CAFC      dd offset _2__STRING_13116_0 ; "kqfsz"
.rodata:0803CB00      dd 1
.rodata:0803CB04      dd 38h
.rodata:0803CB08      dd 0
.rodata:0803CB0C      dd 7
.rodata:0803CB10      dd 0
.rodata:0803CB14      dd 0FFFFC09Dh
.rodata:0803CB18      dd 2
.rodata:0803CB1C      dd 0
```

Здесь очень много ссылок на названия X\$-таблиц, вероятно, на все те что имеются в Oracle RDBMS этой версии. Но мы снова упираемся в то что не имеем достаточно информации. Не ясно, что означает строка **kqvt**. Вообще, префикс **kq** может означать *kernel* и *query*. **v**, может быть, *version*, а **t** — *type*? Сказать трудно.

Таблицу с очень похожим названием мы можем найти в **kqf.o**:

Листинг 81.7: kqf.o

```
.rodata:0808C360 kqvt_c_0      kqftap_param <4, offset _2__STRING_19_0, 917h, 0, 0, 0, 4, 0, 0>
.rodata:0808C360                                     ; DATA XREF: .rodata:08042680
.rodata:0808C360                                     ; "ADDR"
.rodata:0808C384      kqftap_param <4, offset _2__STRING_20_0, 0B02h, 0, 0, 0, 4, 0, 0> ; "↵
↵ INDX"
.rodata:0808C3A8      kqftap_param <7, offset _2__STRING_21_0, 0B02h, 0, 0, 0, 4, 0, 0> ; "↵
↵ INST_ID"
.rodata:0808C3CC      kqftap_param <6, offset _2__STRING_5017_0, 601h, 0, 0, 0, 50h, 0, 0> ↵
↵ ; "BANNER"
```

81.1. ТАБЛИЦА V\$VERSION В ORACLE RDBMS

```
.rodata:0808C3F0          kqftap_param <0, offset _2_STRING_0_0, 0, 0, 0, 0, 0, 0, 0>
```

Она содержит информацию об именах полей в таблице `X$VERSION`. Единственная ссылка на эту таблицу имеется в таблице `kqftap`:

Листинг 81.8: kqf.o

```
.rodata:08042680          kqftap_element <0, offset kqvt_c_0, offset kqvrow, 0> ; element 0x1f6
```

Интересно что здесь этот элемент проходит так же под номером `0x1f6` (502-й), как и ссылка на строку `X$VERSION` в таблице `kqftab`. Вероятно, таблицы `kqftap` и `kqftab` дополняют друг друга, как и `kqfvip` и `kqfviw`. Мы также видим здесь ссылку на функцию с названием `kqvrow()`. А вот это уже кое-что!

Сделаем так чтобы наша программа `oracle tables`³ могла дампит и эти таблицы. Для `X$VERSION` получается:

Листинг 81.9: Результат работы oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0xfffffc075 0x3
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[BANNER] ?: 0x601 0x0 0x0 0x0 0x50 0x0 0x0
kqftap_element.fn1=kqvrow
kqftap_element.fn2=NULL
```

При помощи `tracer`, можно легко проверить, что эта функция вызывается 6 раз кряду (из функции `qerfxFetch()`) при получении строк из `X$VERSION`.

Запустим `tracer` в режиме `cc` (он добавит комментарий к каждой исполненной инструкции):

```
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc
```

```
_kqvrow_ proc near
var_7C      = byte ptr -7Ch
var_18      = dword ptr -18h
var_14      = dword ptr -14h
Dest        = dword ptr -10h
var_C       = dword ptr -0Ch
var_8       = dword ptr -8
var_4       = dword ptr -4
arg_8       = dword ptr 10h
arg_C       = dword ptr 14h
arg_14      = dword ptr 1Ch
arg_18      = dword ptr 20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES

        push    ebp
        mov     ebp, esp
        sub     esp, 7Ch
        mov     eax, [ebp+arg_14] ; [EBP+1Ch]=1
        mov     ecx, TlsIndex    ; [69AEB08h]=0
        mov     edx, large fs:2Ch
        mov     edx, [edx+ecx*4] ; [EDX+ECX*4]=0xc98c938
        cmp     eax, 2           ; EAX=1
        mov     eax, [ebp+arg_8] ; [EBP+10h]=0xcdf554
        jz      loc_2CE1288
        mov     ecx, [eax]       ; [EAX]=0..5
        mov     [ebp+var_4], edi ; EDI=0xc98c938

loc_2CE10F6: ; CODE XREF: _kqvrow_+10A
            ; _kqvrow_+1A9
        cmp     ecx, 5           ; ECX=0..5
        ja      loc_56C11C7
        mov     edi, [ebp+arg_18] ; [EBP+20h]=0
```

³yurichev.com

81.1. ТАБЛИЦА V\$VERSION В ORACLE RDBMS

```

mov     [ebp+var_14], edx ; EDX=0xc98c938
mov     [ebp+var_8], ebx ; EBX=0
mov     ebx, eax          ; EAX=0xcdfe554
mov     [ebp+var_C], esi ; ESI=0xcdfe248

loc_2CE110D: ; CODE XREF: _kqvrow_+29E00E6
mov     edx, ds:off_628B09C[ecx*4] ; [ECX*4+628B09Ch]=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a
↳ x2ce11f6, 0x2ce1236, 0x2ce127a
jmp     edx                ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a
↳ x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
push    offset aXKqvvsBuffer ; "x$kqvvs buffer"
mov     ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
xor     edx, edx
mov     esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
push    edx                ; EDX=0
push    edx                ; EDX=0
push    50h
push    ecx                ; ECX=0x8a172b4
push    dword ptr [esi+10494h] ; [ESI+10494h]=0xc98cd58
call    _kghalf            ; tracing nested maximum level (1) reached, skipping this CALL
mov     esi, ds:__imp__vsnnun ; [59771A8h]=0x61bc49e0
mov     [ebp+Dest], eax ; EAX=0xce2ffb0
mov     [ebx+8], eax      ; EAX=0xce2ffb0
mov     [ebx+4], eax      ; EAX=0xce2ffb0
mov     edi, [esi]        ; [ESI]=0xb200100
mov     esi, ds:__imp__vsenstr ; [597D6D4h]=0x65852148, "- Production"
push    esi                ; ESI=0x65852148, "- Production"
mov     ebx, edi           ; EDI=0xb200100
shr     ebx, 18h           ; EBX=0xb200100
mov     ecx, edi           ; EDI=0xb200100
shr     ecx, 14h           ; ECX=0xb200100
and     ecx, 0Fh           ; ECX=0xb2
mov     edx, edi           ; EDI=0xb200100
shr     edx, 0Ch           ; EDX=0xb200100
movzx   edx, dl            ; DL=0
mov     eax, edi           ; EDI=0xb200100
shr     eax, 8             ; EAX=0xb200100
and     eax, 0Fh           ; EAX=0xb2001
and     edi, 0FFh          ; EDI=0xb200100
push    edi                ; EDI=0
mov     edi, [ebp+arg_18] ; [EBP+20h]=0
push    eax                ; EAX=1
mov     eax, ds:__imp__vsnbab ; [597D6D8h]=0x65852100, "Oracle Database 11g Enterprise
↳ Edition Release %d.%d.%d.%d %s"
push    edx                ; EDX=0
push    ecx                ; ECX=2
push    ebx                ; EBX=0xb
mov     ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
push    eax                ; EAX=0x65852100, "Oracle Database 11g Enterprise Edition Release %d
↳ .%d.%d.%d.%d %s"
mov     eax, [ebp+Dest] ; [EBP-10h]=0xce2ffb0
push    eax                ; EAX=0xce2ffb0
call    ds:__imp__sprintf ; op1=MSVCR80.dll!sprintf tracing nested maximum level (1) reached
↳ , skipping this CALL
add     esp, 38h
mov     dword ptr [ebx], 1

loc_2CE1192: ; CODE XREF: _kqvrow_+FB
; _kqvrow_+128 ...
test    edi, edi           ; EDI=0
jnz     __VInfreq__kqvrow
mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
mov     eax, ebx           ; EBX=0xcdfe554
mov     ebx, [ebp+var_8] ; [EBP-8]=0
lea     eax, [eax+4]        ; [EAX+4]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production", "
↳ Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production", "PL/SQL Release
↳ 11.2.0.1.0 - Production", "TNS for 32-bit Windows: Version 11.2.0.1.0 - Production"

```



```

loc_2CE11A8: ; CODE XREF: _kqvrow_+29E00F6
        mov     esp, ebp
        pop     ebp
        retn                    ; EAX=0xcdfe558

loc_2CE11AC: ; DATA XREF: .rdata:0628B0A0
        mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release ✓
↳ 11.2.0.1.0 - Production"
        mov     dword ptr [ebx], 2
        mov     [ebx+4], edx    ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release ✓
↳ 11.2.0.1.0 - Production"
        push    edx            ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release ✓
↳ 11.2.0.1.0 - Production"
        call    _kkxvsn        ; tracing nested maximum level (1) reached, skipping this CALL
        pop     ecx
        mov     edx, [ebx+4]    ; [EBX+4]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
        movzx   ecx, byte ptr [edx] ; [EDX]=0x50
        test    ecx, ecx        ; ECX=0x50
        jnz     short loc_2CE1192
        mov     edx, [ebp+var_14]
        mov     esi, [ebp+var_C]
        mov     eax, ebx
        mov     ebx, [ebp+var_8]
        mov     ecx, [eax]
        jmp     loc_2CE10F6

loc_2CE11DB: ; DATA XREF: .rdata:0628B0A4
        push    0
        push    50h
        mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
        mov     [ebx+4], edx    ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
        push    edx            ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
        call    _lmxver        ; tracing nested maximum level (1) reached, skipping this CALL
        add     esp, 0Ch
        mov     dword ptr [ebx], 3
        jmp     short loc_2CE1192

loc_2CE11F6: ; DATA XREF: .rdata:0628B0A8
        mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0
        mov     [ebp+var_18], 50h
        mov     [ebx+4], edx    ; EDX=0xce2ffb0
        push    0
        call    _npinli        ; tracing nested maximum level (1) reached, skipping this CALL
        pop     ecx
        test    eax, eax        ; EAX=0
        jnz     loc_56C11DA
        mov     ecx, [ebp+var_14] ; [EBP-14h]=0xc98c938
        lea     edx, [ebp+var_18] ; [EBP-18h]=0x50
        push    edx            ; EDX=0xd76c93c
        push    dword ptr [ebx+8] ; [EBX+8]=0xce2ffb0
        push    dword ptr [ecx+13278h] ; [ECX+13278h]=0xacce190
        call    _nrtnsvrs      ; tracing nested maximum level (1) reached, skipping this CALL
        add     esp, 0Ch

loc_2CE122B: ; CODE XREF: _kqvrow_+29E0118
        mov     dword ptr [ebx], 4
        jmp     loc_2CE1192

loc_2CE1236: ; DATA XREF: .rdata:0628B0AC
        lea     edx, [ebp+var_7C] ; [EBP-7Ch]=1
        push    edx            ; EDX=0xd76c8d8
        push    0
        mov     esi, [ebx+8]    ; [EBX+8]=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 - ✓
↳ Production"
        mov     [ebx+4], esi    ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 - ✓
↳ Production"
        mov     ecx, 50h
        mov     [ebp+var_18], ecx ; ECX=0x50
        push    ecx            ; ECX=0x50

```

81.2. ТАБЛИЦА X\$KSMLRU В ORACLE RDBMS

```

push     esi                ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 - ↗
↳ Production"
call     _lxvers            ; tracing nested maximum level (1) reached, skipping this CALL
add      esp, 10h
mov      edx, [ebp+var_18] ; [EBP-18h]=0x50
mov      dword ptr [ebx], 5
test     edx, edx          ; EDX=0x50
jnz      loc_2CE1192
mov      edx, [ebp+var_14]
mov      esi, [ebp+var_C]
mov      eax, ebx
mov      ebx, [ebp+var_8]
mov      ecx, 5
jmp      loc_2CE10F6

loc_2CE127A: ; DATA XREF: .rdata:0628B0B0
mov      edx, [ebp+var_14] ; [EBP-14h]=0xc98c938
mov      esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
mov      edi, [ebp+var_4] ; [EBP-4]=0xc98c938
mov      eax, ebx          ; EBX=0xcdfe554
mov      ebx, [ebp+var_8] ; [EBP-8]=0

loc_2CE1288: ; CODE XREF: _kqvrow_+1F
mov      eax, [eax+8]      ; [EAX+8]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
test     eax, eax          ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
jz       short loc_2CE12A7
push     offset aXKqvvsBuffer ; "x$kqvvs buffer"
push     eax                ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
mov      eax, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
push     eax                ; EAX=0x8a172b4
push     dword ptr [edx+10494h] ; [EDX+10494h]=0xc98cd58
call     _kghfrf            ; tracing nested maximum level (1) reached, skipping this CALL
add      esp, 10h

loc_2CE12A7: ; CODE XREF: _kqvrow_+1C1
xor      eax, eax
mov      esp, ebp
pop      ebp
retn                                ; EAX=0
_kqvrow_ endp

```

Так можно легко увидеть, что номер строки таблицы задается извне. Сама функция возвращает строку, формируя её так:

Строка 1	Использует глобальные переменные <code>vsenstr</code> , <code>vsnum</code> , <code>vsban</code> . Вызывает <code>sprintf()</code> .
Строка 2	Вызывает <code>kkxvs()</code> .
Строка 3	Вызывает <code>lmxver()</code> .
Строка 4	Вызывает <code>npinli()</code> , <code>nrtnsvrs()</code> .
Строка 5	Вызывает <code>lxvers()</code> .

Так вызываются соответствующие функции для определения номеров версий отдельных модулей.

81.2. Таблица X\$KSMLRU в Oracle RDBMS

В заметке *Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]* упоминается некая служебная таблица:

There is a fixed table called X\$KSMLRU that tracks allocations in the shared pool that cause other objects in the shared pool to be aged out. This fixed table can be used to identify what is causing the large allocation.

If many objects are being periodically flushed from the shared pool then this will cause response time problems and will likely cause library cache latch contention problems when the objects are reloaded into the shared pool.

One unusual thing about the X\$KSMLRU fixed table is that the contents of the fixed table are erased whenever someone selects from the fixed table. This is done since the fixed table stores only the largest allocations that have occurred. The values are reset after being selected so that subsequent large allocations

can be noted even if they were not quite as large as others that occurred previously. Because of this resetting, the output of selecting from this table should be carefully kept since it cannot be retrieved back after the query is issued.

Однако, как можно легко убедиться, эта системная таблица очищается всякий раз, когда кто-то делает запрос к ней. Сможем ли мы найти причину, почему это происходит? Если вернуться к уже рассмотренным таблицам `kqftab` и `kqftap` полученных при помощи `oracle tables`⁴, содержащим информацию о X\$-таблицах, мы узнаем что для того чтобы подготавливать строки этой таблицы, вызывается функция `ksmlrs()`:

Листинг 81.10: Результат работы `oracle tables`

```
kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0xffffc0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRDUR] ?: 0xb02 0x0 0x0 0x0 0x4 0x4 0x0
kqftap_param.name=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0x0
kqftap_param.name=[KSMLRCOM] ?: 0x501 0x0 0x0 0x0 0x14 0xc 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
kqftap_param.name=[KSMLRNUM] ?: 0x2 0x0 0x0 0x0 0x4 0x24 0x0
kqftap_param.name=[KSMLRHON] ?: 0x501 0x0 0x0 0x0 0x20 0x28 0x0
kqftap_param.name=[KSMLROHV] ?: 0xb02 0x0 0x0 0x0 0x4 0x48 0x0
kqftap_param.name=[KSMLRSES] ?: 0x17 0x0 0x0 0x0 0x4 0x4c 0x0
kqftap_param.name=[KSMLRADU] ?: 0x2 0x0 0x0 0x0 0x4 0x50 0x0
kqftap_param.name=[KSMLRNID] ?: 0x2 0x0 0x0 0x0 0x4 0x54 0x0
kqftap_param.name=[KSMLRNSD] ?: 0x2 0x0 0x0 0x0 0x4 0x58 0x0
kqftap_param.name=[KSMLRNCD] ?: 0x2 0x0 0x0 0x0 0x4 0x5c 0x0
kqftap_param.name=[KSMLRNED] ?: 0x2 0x0 0x0 0x0 0x4 0x60 0x0
kqftap_element.fn1=ksmlrs
kqftap_element.fn2=NULL
```

Действительно, при помощи `tracer` легко убедиться, что эта функция вызывается каждый раз, когда мы обращаемся к таблице `X$KSMLRU`.

Здесь есть ссылки на функции `ksmsplu_sp()` и `ksmsplu_jp()`, каждая из которых в итоге вызывает `ksmsplu()`. В конце функции `ksmsplu()` мы видим вызов `memset()`:

Листинг 81.11: `ksm.o`

```
...
.text:00434C50 loc_434C50:                                ; DATA XREF: .rdata:off_5E50EA8
.text:00434C50      mov     edx, [ebp-4]
.text:00434C53      mov     [eax], esi
.text:00434C55      mov     esi, [edi]
.text:00434C57      mov     [eax+4], esi
.text:00434C5A      mov     [edi], eax
.text:00434C5C      add     edx, 1
.text:00434C5F      mov     [ebp-4], edx
.text:00434C62      jnz     loc_434B7D
.text:00434C68      mov     ecx, [ebp+14h]
.text:00434C6B      mov     ebx, [ebp-10h]
.text:00434C6E      mov     esi, [ebp-0Ch]
.text:00434C71      mov     edi, [ebp-8]
.text:00434C74      lea     eax, [ecx+8Ch]
.text:00434C7A      push    370h                ; Size
.text:00434C7F      push    0                   ; Val
.text:00434C81      push    eax                 ; Dst
.text:00434C82      call    __intel_fast_memset
.text:00434C87      add     esp, 0Ch
.text:00434C8A      mov     esp, ebp
.text:00434C8C      pop     ebp
.text:00434C8D      retn
.text:00434C8D _ksmsplu      endp
```

⁴yurichev.com

81.3. ТАБЛИЦА V\$TIMER В ORACLE RDBMS

Такие конструкции (`memset (block, 0, size)`) очень часто используются для простого обнуления блока памяти. Мы можем попробовать рискнуть, заблокировав вызов `memset()` и посмотреть, что будет?

Запускаем `tracer` со следующей опцией: поставить точку останова на `0x434C7A` (там, где начинается передача параметров для функции `memset()`) так, чтобы `tracer` в этом месте установил указатель инструкций процессора (`EIP`) на место, где уже произошла очистка переданных параметров в `memset()` (по адресу `0x434C8A`): Можно сказать, при помощи этого, мы симулируем безусловный переход с адреса `0x434C7A` на `0x434C8A`.

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0x00434C8A)
```

(Важно: все эти адреса справедливы только для win32-версии Oracle RDBMS 11.2)

Действительно, после этого мы можем обращаться к таблице `X$KSMLRU` сколько угодно, и она уже не очищается!

Не делайте этого дома ("Разрушители легенд") Не делайте этого на своих production-серверах.

Впрочем, это не обязательно полезное или желаемое поведение системы, но как эксперимент по поиску нужного кода, нам это подошло!

81.3. Таблица V\$TIMER в Oracle RDBMS

`V$TIMER` это еще один служебный *fixed view*, отражающий какое-то часто меняющееся значение:

`V$TIMER` displays the elapsed time in hundredths of a second. Time is measured since the beginning of the epoch, which is operating system specific, and wraps around to 0 again whenever the value overflows four bytes (roughly 497 days).

(Из документации Oracle RDBMS ⁵)

Интересно что периоды разные в Oracle для Win32 и для Linux. Сможем ли мы найти функцию, отвечающую за генерирование этого значения?

Как видно, эта информация, в итоге, берется из системной таблицы `X$KSUTM`.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$TIMER';

VIEW_NAME
-----
VIEW_DEFINITION
-----

V$TIMER
select  HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$TIMER';

VIEW_NAME
-----
VIEW_DEFINITION
-----

GV$TIMER
select inst_id,ksutmtim from x$ksutm
```

Здесь мы упираемся в небольшую проблему, в таблицах `kqftab` / `kqftap` нет указателей на функцию, которая бы генерировала значение :

Листинг 81.12: Результат работы oracle tables

```
kqftab_element.name: [X$KSUTM] ? : [ksutm] 0x1 0x4 0x4 0x0 0xffffc09b 0x3
kqftap_param.name=[ADDR] ? : 0x10917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ? : 0x20b02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ? : 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSUTMTIM] ? : 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e
```

⁵<http://go.yurichev.com/17088>

81.3. ТАБЛИЦА V\$TIMER В ORACLE RDBMS

```
kqftap_element.fn1=NULL  
kqftap_element.fn2=NULL
```

Попробуем в таком случае просто поискать строку `KSUTMTIM`, и находим ссылку на нее в такой функции:

```
kqfd_DRN_ksutm_c proc near          ; DATA XREF: .rodata:0805B4E8  
  
arg_0      = dword ptr  8  
arg_8      = dword ptr 10h  
arg_C      = dword ptr 14h  
  
            push     ebp  
            mov      ebp, esp  
            push     [ebp+arg_C]  
            push     offset ksugtm  
            push     offset _2_STRING_1263_0 ; "KSUTMTIM"  
            push     [ebp+arg_8]  
            push     [ebp+arg_0]  
            call     kqfd_cfui_drain  
            add      esp, 14h  
            mov      esp, ebp  
            pop      ebp  
            retn  
kqfd_DRN_ksutm_c endp
```

Сама функция `kqfd_DRN_ksutm_c()` упоминается в таблице `kqfd_tab_registry_0` вот так:

```
dd offset _2_STRING_62_0 ; "X$KSUTM"  
dd offset kqfd_OPN_ksutm_c  
dd offset kqfd_tabl_fetch  
dd 0  
dd 0  
dd offset kqfd_DRN_ksutm_c
```

Упоминается также некая функция `ksugtm()`. Посмотрим, что там (в Linux x86):

Листинг 81.13: `ksu.o`

```
ksugtm      proc near  
  
var_1C      = byte ptr -1Ch  
arg_4       = dword ptr  0Ch  
  
            push     ebp  
            mov      ebp, esp  
            sub      esp, 1Ch  
            lea      eax, [ebp+var_1C]  
            push     eax  
            call     slgcs  
            pop      ecx  
            mov      edx, [ebp+arg_4]  
            mov      [edx], eax  
            mov      eax, 4  
            mov      esp, ebp  
            pop      ebp  
            retn  
ksugtm      endp
```

В win32-версии тоже самое.

Искомая ли эта функция? Попробуем узнать:

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0x4
```

Пробуем несколько раз:

```
SQL> select * from V$TIMER;  
  
HSECS  
-----
```

```

27294929

SQL> select * from V$TIMER;

      HSECS
-----
27295006

SQL> select * from V$TIMER;

      HSECS
-----
27295167

```

Листинг 81.14: вывод [tracer](#)

```

TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+0xfad ↗
↳ (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                "8."
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: D1 7C A0 01                                ".|.."
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+0xfad ↗
↳ (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                "8."
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: 1E 7D A0 01                                ".}.."
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+0xfad ↗
↳ (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                "8."
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: BF 7D A0 01                                ".}.."

```

Действительно — значение то, что мы видим в SQL*Plus, и оно возвращается через второй аргумент.

Посмотрим, что в функции `slgcs()` (Linux x86):

```

slgcs      proc near
var_4      = dword ptr -4
arg_0      = dword ptr 8

        push    ebp
        mov     ebp, esp
        push    esi
        mov     [ebp+var_4], ebx
        mov     eax, [ebp+arg_0]
        call    $+5
        pop     ebx
        nop
        ; PIC mode
        mov     ebx, offset _GLOBAL_OFFSET_TABLE_
        mov     dword ptr [eax], 0
        call    sltrgtime64 ; PIC mode
        push    0
        push    0Ah
        push    edx
        push    eax
        call    __udivdi3 ; PIC mode
        mov     ebx, [ebp+var_4]
        add     esp, 10h
        mov     esp, ebp
        pop     ebp
        retn
slgcs      endp

```

(это просто вызов `sltrgtime64()` и деление его результата на 10 ([42](#) (стр. [483](#))))

И в win32-версии:

```

_slgcs      proc near      ; CODE XREF: _dbgefghTtElResetCount+15
                                ; _dbgerRunActions+1528
    db      66h
    nop
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+8]
    mov     dword ptr [eax], 0
    call    ds:__imp__GetTickCount@0 ; GetTickCount()
    mov     edx, eax
    mov     eax, 0CCCCCDh
    mul     edx
    shr     edx, 3
    mov     eax, edx
    mov     esp, ebp
    pop     ebp
    retn
_slgcs      endp

```

Это просто результат `GetTickCount()` ⁶ поделенный на 10 ([42](#) (стр. [483](#))).

Вуаля! Вот почему в win32-версии и версии Linux x86 разные результаты, потому что они получаются разными системными функциями [ОС](#).

Drain по-английски *дренаж, отток, водосток*. Таким образом, возможно имеется ввиду *подключение* определенного столбца системной таблице к функции.

Добавим поддержку таблицы `kqfd_tab_registry_0` в `oracle tables`⁷, теперь мы можем видеть, при помощи каких функций, столбцы в системных таблицах *подключаются* к значениям, например:

```

[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksusg_c]

```

OPN, возможно, *open*, а *DRN*, вероятно, означает *drain*.

⁷yurichev.com

Глава 82

Вручную написанный на ассемблере код

82.1. Тестовый файл EICAR

Этот .COM-файл предназначен для тестирования антивирусов, его можно запустить в MS-DOS и он выведет такую строку: «EICAR-STANDARD-ANTIVIRUS-TEST-FILE!»¹.

Он примечателен тем, что он полностью состоит только из печатных ASCII-символов, следовательно, его можно набрать в любом текстовом редакторе:

```
X50!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Попробуем его разобрать:

```
; изначальное состояние: SP=0FFFEh, SS:[SP]=0
0100 58          pop      ax
; AX=0, SP=0
0101 35 4F 21     xor      ax, 214Fh
; AX = 214Fh and SP = 0
0104 50          push     ax
; AX = 214Fh, SP = FFFEh and SS:[FFFE] = 214Fh
0105 25 40 41     and      ax, 4140h
; AX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
0108 50          push     ax
; AX = 140h, SP = FFFCh, SS:[FFFC] = 140h and SS:[FFFE] = 214Fh
0109 5B          pop      bx
; AX = 140h, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010A 34 5C        xor      al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010C 50          push     ax
010D 5A          pop      dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFEh and SS:[FFFE] = 214Fh
010E 58          pop      ax
; AX = 214Fh, BX = 140h, DX = 11Ch and SP = 0
010F 35 34 28     xor      ax, 2834h
; AX = 97Bh, BX = 140h, DX = 11Ch and SP = 0
0112 50          push     ax
0113 5E          pop      si
; AX = 97Bh, BX = 140h, DX = 11Ch, SI = 97Bh and SP = 0
0114 29 37        sub      [bx], si
0116 43          inc      bx
0117 43          inc      bx
0118 29 37        sub      [bx], si
011A 7D 24        jge      short near ptr word_10140
011C 45 49 43 ...  db 'EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$'
0140 48 2B        word_10140 dw 2B48h ; CD 21 (INT 21) будет здесь
0142 48 2A        dw 2A48h ; CD 20 (INT 20) будет здесь
0144 0D          db 0Dh
0145 0A          db 0Ah
```

Добавим везде комментарии, показывающие состояние регистров и стека после каждой инструкции.

¹[wikipedia](#)

82.1. ТЕСТОВЫЙ ФАЙЛ EICAR

Собственно, все эти инструкции нужны только для того чтобы исполнить следующий код:

```
B4 09    MOV AH, 9
BA 1C 01  MOV DX, 11Ch
CD 21     INT 21h
CD 20     INT 20h
```

`INT 21h` с функцией 9 (переданной в `AH`) просто выводит строку, адрес которой передан в `DS:DX`. Кстати, строка должна быть завершена символом '\$'. Вероятно, это наследие [CP/M](#) и эта функция в DOS осталась для совместимости.

`INT 20h` возвращает управление в DOS.

Но, как видно, далеко не все опкоды этих инструкций печатные. Так что основная часть EICAR-файла это:

- подготовка нужных значений регистров (AH и DX);
- подготовка в памяти опкодов для INT 21 и INT 20;
- исполнение INT 21 и INT 20.

Кстати, подобная техника широко используется для создания шеллкодов, где нужно создать x86-код, который будет нужно передать в виде текстовой строки .

Здесь также список всех x86-инструкций с печатаемыми опкодами: [A.6.5](#) (стр. [925](#)).

Глава 83

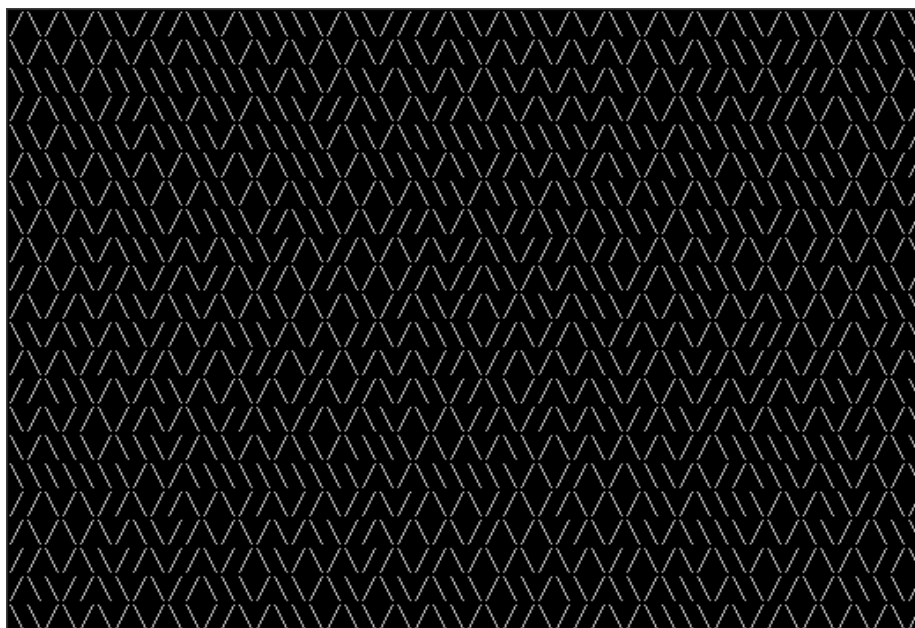
Демо

Демо (или демомейкинг) были великолепным упражнением в математике, программировании компьютерной графики и очень плотному программированию на ассемблере вручную.

83.1. 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10

Все примеры здесь для .COM-файлов под MS-DOS.

В [Nick Montfort et al, *10 PRINT CHR\$(205.5+RND(1)); : GOTO 10*, (The MIT Press:2012)]¹ можно прочитать об одном из простейших генераторов случайных лабиринтов. Он просто бесконечно и случайно печатает символ слэша или обратный слэша, выдавая в итоге что-то вроде :



Здесь несколько известных реализаций для 16-битного x86.

83.1.1. Версия 42-х байт от Trixter

Листинг взят с его сайта², но комментарии — мои.

00000000:	B001	mov	al,1	; установить видеорежим 40x25
00000002:	CD10	int	010	
00000004:	30FF	xor	bh,bh	; установить видеостраницу для вызова int 10h
00000006:	B9D007	mov	cx,007D0	; вывод 2000 символов
00000009:	31C0	xor	ax,ax	
0000000B:	9C	pushf		; сохранить флаги

¹Также доступно здесь: <http://go.yurichev.com/17286>

²<http://go.yurichev.com/17305>

83.1. 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10

```
; узнать случайное число из чипа таймера
0000000C: FA          cli          ; запретить прерывания
0000000D: E643        out          043,al    ; записать 0 в порт 43h
; прочитать 16-битное значение из порта 40h
0000000F: E440        in           al,040
00000011: 88C4        mov          ah,al
00000013: E440        in           al,040
00000015: 9D          popf          ; разрешить прерывания возвращая значение флага IF
00000016: 86C4        xchg         ah,al
; здесь мы имеем псевдослучайное 16-битное значение
00000018: D1E8        shr          ax,1
0000001A: D1E8        shr          ax,1
; в CF сейчас находится второй бит из значения
0000001C: B05C        mov          al,05C ; '\'
; если CF=1, пропускаем следующую инструкцию
0000001E: 7202        jc           00000022
; если CF=0, загружаем в регистр AL другой символ
00000020: B02F        mov          al,02F ; '/'
; вывод символа
00000022: B40E        mov          ah,00E
00000024: CD10        int          010
00000026: E2E1        loop         00000009 ; цикл в 2000 раз
00000028: CD20        int          020      ; возврат в DOS
```

Псевдослучайное число на самом деле это время, прошедшее со старта системы, получаемое из чипа таймера 8253, это значение увеличивается на единицу 18.2 раза в секунду.

Записывая ноль в порт 43h, мы имеем ввиду что команда это «выбрать счетчик 0», "counter latch", "двоичный счетчик" (а не значение BCD³).

Прерывания снова разрешаются при помощи инструкции POPF, которая также возвращает флаг IF.

Инструкцию IN нельзя использовать с другими регистрами кроме AL, поэтому здесь перетасовка.

83.1.2. Моя попытка укоротить версию Trixter: 27 байт

Мы можем сказать, что мы используем таймер не для того чтобы получить точное время, но псевдослучайное число, так что мы можем не тратить время (и код) на запрещение прерываний. Еще можно сказать, что так как мы берем бит из младшей 8-битной части, то мы можем считывать только её.

Немного укоротим код и выходит 27 байт:

```
00000000: B9D007 mov cx,007D0 ; вывести только 2000 символов
00000003: 31C0 xor ax,ax ; команда чипу таймера
00000005: E643 out 043,al
00000007: E440 in al,040 ; читать 8 бит из таймера
00000009: D1E8 shr ax,1 ; переместить второй бит в флаг CF
0000000B: D1E8 shr ax,1
0000000D: B05C mov al,05C ; подготовить '\'
0000000F: 7202 jc 00000013
00000011: B02F mov al,02F ; подготовить '/'
; вывести символ на экран
00000013: B40E mov ah,00E
00000015: CD10 int 010
00000017: E2EA loop 00000003
; выход в DOS
00000019: CD20 int 020
```

83.1.3. Использование случайного мусора в памяти как источника случайных чисел

Так как это MS-DOS, защиты памяти здесь нет вовсе, так что мы можем читать с какого угодно адреса. И даже более того: простая инструкция LODSB будет читать байт по адресу DS:SI, но это не проблема если правильные значения не установлены в регистры, пусть она читает 1) случайные байты; 2) из случайного места в памяти!

Так что на странице Trixter-а⁴ можно найти предложение использовать LODSB без всякой инициализации.

³Binary-coded decimal

⁴<http://go.yurichev.com/17305>

83.1. 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10

Есть также предложение использовать инструкцию `SCASB` вместо, потому что она выставляет флаги в соответствии с прочитанным значением.

Еще одна идея насчет минимизации кода — это использовать прерывание DOS `INT 29h` которое просто печатает символ на экране из регистра `AL`.

Это то что сделали Peter Ferrie и Андрей «herm1t» Баранович (11 и 10 байт)⁵:

Листинг 83.1: Андрей «herm1t» Баранович: 11 байт

```
00000000: B05C      mov     al,05C      ; '\'
; читать байт в AL из случайного места в памяти
00000002: AE        scasb
; PF = четность(AL - случайный_байт) = четность(5Ch - случайный_байт)
00000003: 7A02      jp      00000007
00000005: B02F      mov     al,02F      ; '/'
00000007: CD29      int     029        ; вывод AL на экран
00000009: EBF5      jmp     00000000 ; бесконечный цикл
```

`SCASB` также использует значение в регистре `AL`, она вычитает значение случайного байта в памяти из `5Ch` в `AL`. `JP` это редкая инструкция, здесь она используется для проверки флага четности (PF), который вычисляется по формуле в листинге. Как следствие, выводимый символ определяется не каким-то конкретным битом из случайного байта в памяти, а суммой бит, и это (надеюсь) сделает результат более распределенным.

Можно сделать еще короче, если использовать недокументированную x86-инструкцию `SALC` (АКА `SETALC`) («Set AL CF»). Она появилась в `CPU` и выставляет `AL` в `0xFF` если `CF` это 1 или 0 если наоборот.

Листинг 83.2: Peter Ferrie: 10 байт

```
; AL в этом месте имеет случайное значение
00000000: AE        scasb
; CF устанавливается по результату вычитания случайного байта памяти из AL.
; так что он здесь случаен, в каком-то смысле
00000001: D6        setalc
; AL выставляется в 0xFF если CF=1 или в 0 если наоборот
00000002: 242D      and     al,02D      ; '-'
; AL здесь 0x2D либо 0
00000004: 042F      add     al,02F      ; '/'
; AL здесь 0x5C либо 0x2F
00000006: CD29      int     029        ; вывести AL на экране
00000008: EBF6      jmps    00000000 ; бесконечный цикл
```

Так что можно избавиться и от условных переходов. ASCII-код обратного слэша («\») это `0x5C` и `0x2F` для слэша («/»). Так что нам нужно конвертировать один (псевдослучайный) бит из флага `CF` в значение `0x5C` или `0x2F`.

Это делается легко: применяя операцию «И» ко всем битам в `AL` (где все 8 бит либо выставлены, либо сброшены) с `0x2D` мы имеем просто 0 или `0x2D`.

Прибавляя значение `0x2F` к этому значению, мы получаем `0x5C` или `0x2F`. И просто выводим это на экран.

83.1.4. Вывод

Также стоит отметить, что результат может быть разным в эмуляторе DOSBox, `Windows NT` и даже MS-DOS, из-за разных условий: чип таймера может эмулироваться по-разному, изначальные значения регистров также могут быть разными.

⁵<http://go.yurichev.com/17087>

83.2. Множество Мандельброта

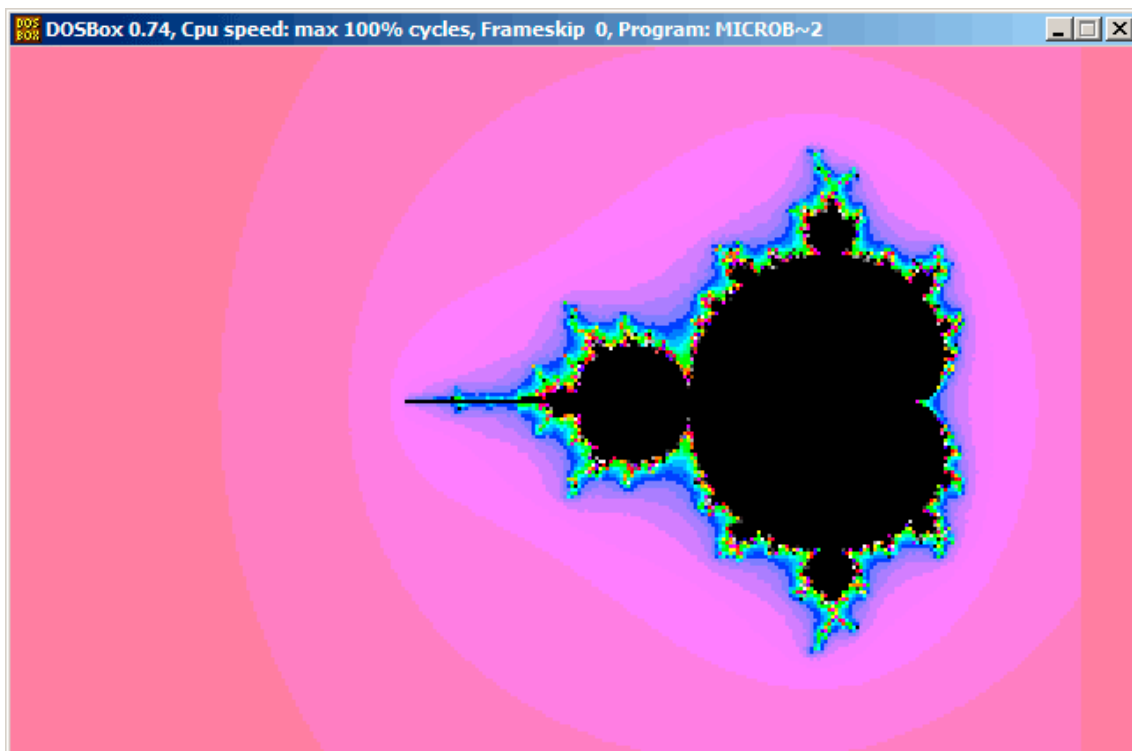
You know, if you magnify the coastline, it still looks like a coastline, and a lot of other things have this property. Nature has recursive algorithms that it uses to generate clouds and Swiss cheese and things like that.

Дональд Кнут, интервью в 1993

Множество Мандельброта это фрактал, характерное свойство которого это самоподобие. При увеличении картинки, вы видите, что этот характерный узор повторяется бесконечно.

Вот демо⁶ написанное автором по имени «Sir_Lagsalot» в 2009, рисующее множество Мандельброта, и это программа для x86 с размером файла всего 64 байта. Там только 30 16-битных x86-инструкций.

Вот что она рисует:



Попробуем разобраться, как она работает.

⁶Можно скачать [здесь](#),

83.2.1. Теория**Немного о комплексных числах**

Комплексное число состоит из двух чисел (вещественная (Re) и мнимая (Im)).

Комплексная плоскость — это двумерная плоскость, где любое комплексное число может быть расположено: вещественная часть — это одна координата и мнимая — вторая.

Некоторые базовые правила, которые нам понадобятся:

- Сложение: $(a + bi) + (c + di) = (a + c) + (b + d)i$

Другими словами:

$$\text{Re}(\text{sum}) = \text{Re}(a) + \text{Re}(b)$$

$$\text{Im}(\text{sum}) = \text{Im}(a) + \text{Im}(b)$$

- Умножение: $(a + bi)(c + di) = (ac - bd) + (bc + ad)i$

Другими словами:

$$\text{Re}(\text{product}) = \text{Re}(a) \cdot \text{Re}(c) - \text{Re}(b) \cdot \text{Re}(d)$$

$$\text{Im}(\text{product}) = \text{Im}(b) \cdot \text{Im}(c) + \text{Im}(a) \cdot \text{Im}(d)$$

- Возведение в квадрат: $(a + bi)^2 = (a + bi)(a + bi) = (a^2 - b^2) + (2ab)i$

Другими словами:

$$\text{Re}(\text{square}) = \text{Re}(a)^2 - \text{Im}(a)^2$$

$$\text{Im}(\text{square}) = 2 \cdot \text{Re}(a) \cdot \text{Im}(a)$$

Как нарисовать множество Мандельброта

Множество Мандельброта — это набор точек, для которых рекурсивное соотношение $z_{n+1} = z_n^2 + c$ (где z и c это комплексные числа и c это начальное значение) не стремится к бесконечности.

Простым русским языком:

- Перечисляем все точки на экране.
- Проверяем, является ли эта точка в множестве Мандельброта.
- Вот как проверить:
 - Представим точку как комплексное число.
 - Возведем в квадрат.
 - Прибавим значение точки в самом начале.
 - Вышло за пределы? Прерываемся, если да.
 - Передвигаем точку в новое место, координаты которого только что вычислили.
 - Повторять всё это некое разумное количество итераций.
- Двигающаяся точка в итоге не вышла за пределы? Тогда рисуем точку.
- Двигающаяся точка в итоге вышла за пределы?
 - (Для черно-белого изображения) ничего не рисуем.
 - (Для цветного изображения) преобразуем количество итераций в какой-нибудь цвет. Так что цвет будет показывать, с какой скоростью точка вышла за пределы.

Вот алгоритмы для комплексных и обычных целочисленных чисел (на языке, отдаленно напоминающем Python):

Листинг 83.3: Для комплексных чисел

```
def check_if_is_in_set(P):
    P_start=P
    iterations=0

    while True:
```

83.2. МНОЖЕСТВО МАНДЕЛЬБРОТА

```
    if (P>bounds):
        break
    P=P^2+P_start
    if iterations > max_iterations:
        break
    iterations++

    return iterations

# черно-белое
for each point on screen P:
    if check_if_is_in_set (P) < max_iterations:
        нарисовать точку

# цветное
for each point on screen P:
    iterations = if check_if_is_in_set (P)
    преобразовать количество итераций в цвет
    нарисовать цветную точку
```

Целочисленная версия, это версия где все операции над комплексными числами заменены на операции с целочисленными, в соответствии с изложенными ранее правилами.

Листинг 83.4: Для целочисленных чисел

```
def check_if_is_in_set(X, Y):
    X_start=X
    Y_start=Y
    iterations=0

    while True:
        if (X^2 + Y^2 > bounds):
            break
        new_X=X^2 - Y^2 + X_start
        new_Y=2*X*Y + Y_start
        if iterations > max_iterations:
            break
        iterations++

    return iterations

# черно-белое
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        if check_if_is_in_set (X,Y) < max_iterations:
            нарисовать точку на X, Y

# цветное
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        iterations = if check_if_is_in_set (X,Y)
        преобразовать количество итераций в цвет
        нарисовать цветную точку на X,Y
```

Вот также исходный текст на C#, который есть в статье в Wikipedia⁷, но мы немного изменим его, чтобы он выдавал количество итераций, вместо некоторого символа⁸:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Mnoj
{
    class Program
    {
        static void Main(string[] args)
```

⁷[wikipedia](#)

⁸Здесь также и исполняемый файл: [beginners.re](#)

```

{
    double realCoord, imagCoord;
    double realTemp, imagTemp, realTemp2, arg;
    int iterations;
    for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
    {
        for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
        {
            iterations = 0;
            realTemp = realCoord;
            imagTemp = imagCoord;
            arg = (realCoord * realCoord) + (imagCoord * imagCoord);
            while ((arg < 2*2) && (iterations < 40))
            {
                realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp) - realCoord;
                imagTemp = (2 * realTemp * imagTemp) - imagCoord;
                realTemp = realTemp2;
                arg = (realTemp * realTemp) + (imagTemp * imagTemp);
                iterations += 1;
            }
            Console.WriteLine("{0,2:D} ", iterations);
        }
        Console.WriteLine("\n");
    }
    Console.ReadKey();
}
}
}

```

Вот файл с результатом, который слишком широкий, чтобы привести его здесь:

beginners.re.

Максимальное число итераций 40, так что если вы видите 40 в этом файле, это означает что точка ходила 40 итераций, но так и не вышла за пределы. Номер n меньше 40 означает что эта точка оставалась внутри пределов только n итераций, и затем вышла наружу.

83.2. МНОЖЕСТВО МАНДЕЛЬБРОТА

Вот здесь есть неплохая демонстрация: <http://go.yurichev.com/17309>, она показывает визуально, как определенная точка движется по плоскости на каждой итерации. Вот два скриншота.

В начале кликаем внутри желтой области, и увидим траекторию (зеленые линии), которая в итоге загручивается в какой-то точке внутри:

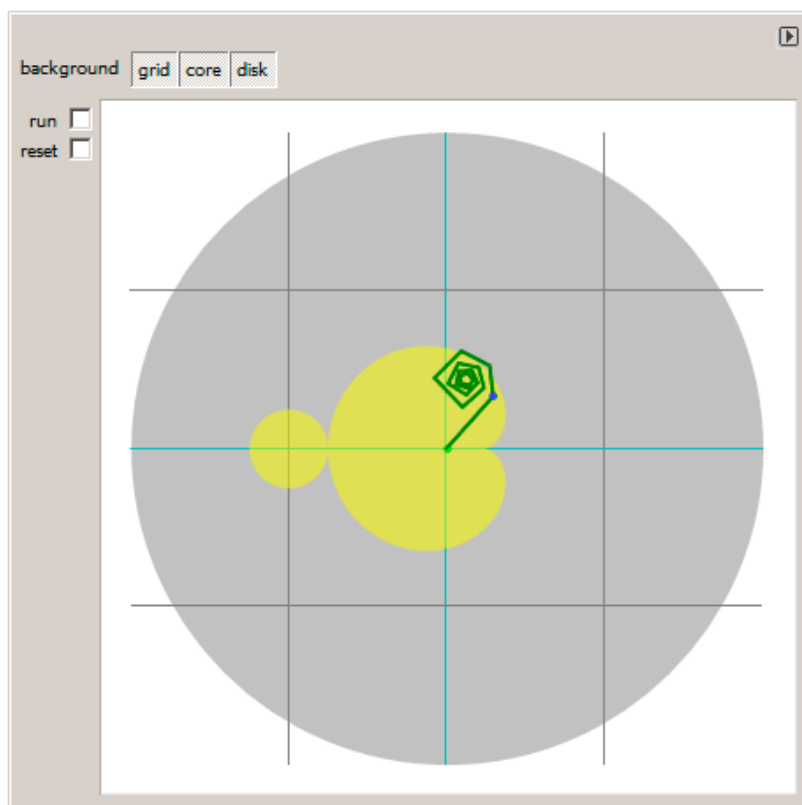


Рис. 83.1: Клик внутри желтой области

Это значит, что точка на которой кликнули, находится внутри множества Мандельброта.

83.2. МНОЖЕСТВО МАНДЕЛЬБРОТА

Затем кликаем снаружи желтой области, и мы видим более хаотичные движения точки, которая быстро выходит за пределы:

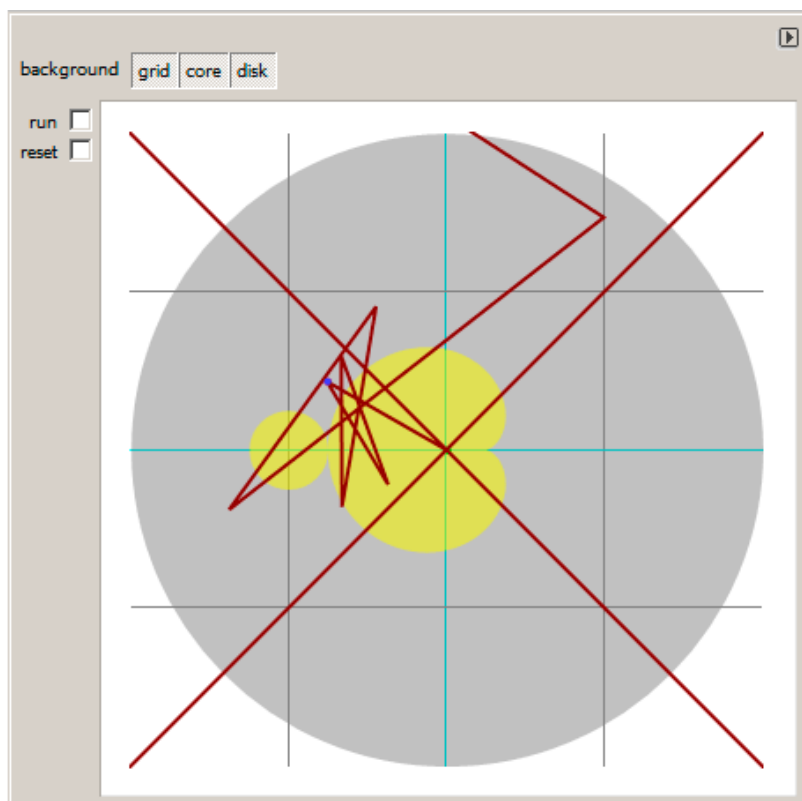


Рис. 83.2: Клик снаружи желтой области

Это значит, что эта точка не принадлежит множеству Мандельброта.

Другая неплохая демонстрация там: <http://go.yurichev.com/17310>.

83.2.2. Вернемся к демо

Демо, хотя и крошечная (только 64 байта или 30 инструкций), реализует общий алгоритм, изложенный здесь, но с некоторыми трюками.

Исходный код можно скачать, так что вот он, но также снабдим его своими комментариями:

Листинг 83.5: Исходный код с комментариями

```

1  ; X это столбец на экране
2  ; Y это строка на экране
3
4
5  ; X=0, Y=0                X=319, Y=0
6  ; +----->
7  ; |
8  ; |
9  ; |
10 ; |
11 ; |
12 ; |
13 ; v
14 ; X=0, Y=199              X=319, Y=199
15
16
17 ; переключиться в графический видеорежим VGA 320*200*256
18 mov al,13h
19 int 10h
20 ; в самом начале BX равен 0
21 ; в самом начале DI равен 0xFFFFE
22 ; DS:BX (или DS:0) указывает на Program Segment Prefix в этот момент
23 ; ... первые 4 байта которого этого CD 20 FF 9F
24 les ax,[bx]
25 ; ES:AX=9FFF:20CD
26
27 FillLoop:
28 ; установить DX в 0. CWD работает так: DX:AX = sign_extend(AX).
29 ; AX здесь 0x20CD (в начале) или меньше 320 (когда вернемся после цикла),
30 ; так что DX всегда будет 0.
31 cwd
32 mov ax,di
33 ; AX это текущий указатель внутри VGA-буфера
34 ; разделить текущий указатель на 320
35 mov cx,320
36 div cx
37 ; DX (start_X) – остаток (столбец: 0..319); AX – результат (строка: 0..199)
38 sub ax,100
39 ; AX=AX-100, так что AX (start_Y) сейчас в пределах -100..99
40 ; DX в пределах 0..319 или 0x0000..0x013F
41 dec dh
42 ; DX сейчас в пределах 0xFF00..0x003F (-256..63)
43
44 xor bx,bx
45 xor si,si
46 ; BX (temp_X)=0; SI (temp_Y)=0
47
48 ; получить максимальное количество итераций
49 ; CX всё еще 320 здесь, так что это будет максимальным количеством итераций
50 MandelLoop:
51 mov bp,si      ; BP = temp_Y
52 imul si,bx     ; SI = temp_X*temp_Y
53 add si,si      ; SI = SI*2 = (temp_X*temp_Y)*2
54 imul bx,bx     ; BX = BX^2 = temp_X^2
55 jo MandelBreak ; переполнение?
56 imul bp,bp     ; BP = BP^2 = temp_Y^2
57 jo MandelBreak ; переполнение?
58 add bx,bp      ; BX = BX+BP = temp_X^2 + temp_Y^2
59 jo MandelBreak ; переполнение?
60 sub bx,bp      ; BX = BX-BP = temp_X^2 + temp_Y^2 - temp_Y^2 = temp_X^2
61 sub bx,bp      ; BX = BX-BP = temp_X^2 - temp_Y^2
62

```

83.2. МНОЖЕСТВО МАНДЕЛЬБРОТА

```
63 ; скорректировать масштаб:
64 sar bx,6      ; BX=BX/64
65 add bx,dx     ; BX=BX+start_X
66 ; здесь temp_X = temp_X^2 - temp_Y^2 + start_X
67 sar si,6      ; SI=SI/64
68 add si,ax     ; SI=SI+start_Y
69 ; здесь temp_Y = (temp_X*temp_Y)*2 + start_Y
70
71 loop MandelLoop
72
73 MandelBreak:
74 ; CX=итерации
75 xchg ax,cx
76 ; AX=итерации. записать AL в VGA-буфер на ES:[DI]
77 stosb
78 ; stosb также инкрементирует DI, так что DI теперь указывает на следующую точку в VGA-буфере
79 ; всегда переходим, так что это вечный цикл
80 jmp FillLoop
```

Алгоритм:

- Переключаемся в режим VGA 320*200 256 цветов. $320 * 200 = 64000$ (0xFA00). Каждый пиксель кодируется одним байтом, так что размер буфера 0xFA00 байт. Он адресуется здесь при помощи пары регистров ES:DI.

ES должен быть здесь 0xA000, потому что это сегментный адрес видеобуфера, но запись числа 0xA000 в ES потребует по крайней мере 4 байта (PUSH 0A000h / POP ES). О 16-битной модели памяти в MS-DOS, читайте больше тут: [96](#) (стр. [898](#)).

Учитывая, что BX здесь 0, и Program Segment Prefix находится по нулевому адресу, 2-байтная инструкция LES AX, [BX] запишет 0x20CD в AX и 0x9FFF в ES. Так что программа начнет рисовать на 16 пикселей (или байт) перед видеобуфером. Но это MS-DOS, здесь нет защиты памяти, так что запись происходит в самый конец обычной памяти, а там, как правило, ничего важного нет. Вот почему вы видите красную полосу шириной 16 пикселей справа. Вся картинка сдвинута влево на 16 пикселей. Это цена экономии 2-х байт.

- Вечный цикл, обрабатывающий каждый пиксель. Наверное, самый общий метод обойти все точки на экране это два цикла: один для X-координаты, второй для Y-координаты. Но тогда вам придется перемножать координаты для поиска байта в видеобуфере VGA. Автор этого демо решил сделать наоборот: перебирать все байты в видеобуфере при помощи одного цикла вместо двух и затем получать координаты текущей точки при помощи деления. В итоге координаты такие: X в пределах -256..63 и Y в пределах -100..99. Вы можете увидеть на скриншоте что картинка как бы сдвинута в правую часть экрана. Это потому что самая большая черная дыра в форме сердца обычно появляется на координатах 0,0 и они здесь сдвинуты вправо. Мог ли автор просто отнять 160 от X, чтобы получилось значение в пределах -160..159? Да, но инструкция SUB DX, 160 занимает 4 байта, тогда как DEC DH — 2 байта (которая отнимает 0x100 (256) от DX). Так что картинка сдвинута ценой экономии еще 2-х байт.
 - Проверить, является ли текущая точка внутри множества Мандельброта. Алгоритм такой же, как и описанный здесь.
 - Цикл организуется инструкцией LOOP, которая использует регистр CX как счетчик. Автор мог бы установить число итераций на какое-то число, но не сделал этого: потому что 320 уже находится в CX (было установлено на строке 35), и это итак подходящее число как число максимальных итераций. Мы здесь экономим немного места, не загружая другое значение в регистр CX.
 - Здесь используется IMUL вместо MUL, потому что мы работаем с знаковыми значениями: помните, что координаты 0,0 должны быть где-то рядом с центром экрана. Тоже самое и с SAR (арифметический сдвиг для знаковых значений): она используется вместо SHR.
 - Еще одна идея — это упростить проверку пределов. Нам бы пришлось проверять пару координат, т.е. две переменных. Что делает автор это трижды проверяет на переполнение: две операции возведения в квадрат и одно прибавление. Действительно, мы ведь используем 16-битные регистры, содержащие знаковые значения в пределах -32768..32767, так что если любая из координат больше чем 32767 в процессе умножения, точка однозначно вышла за пределы, и мы переходим на метку MandelBreak.
 - Здесь также имеется деление на 64 (при помощи инструкции SAR). 64 задает масштаб. Попробуйте увеличить значение и вы получите более увеличенную картинку, или уменьшить для меньшей.
- Мы находимся на метке MandelBreak, есть только две возможности попасть сюда: цикл закончился с CX=0 (точка внутри множества Мандельброта); или потому что произошло переполнение (CX все еще содержит какое-то значение). Записываем 8-битную часть CX (CL) в видеобуфер. Палитра по умолчанию грубая, тем не менее, 0 это черный: поэтому видим черные дыры в местах где точки внутри множества Мандельброта. Палитру можно инициализировать в начале программы, но не забывайте, это всего лишь программа на 64 байта!

83.2. МНОЖЕСТВО МАНДЕЛЬБРОТА

- Программа работает в вечном цикле, потому что дополнительная проверка, где остановится, или пользовательский интерфейс, это дополнительные инструкции.

Еще оптимизационные трюки:

- 1-байтная CWD используется здесь для обнуления DX вместо двухбайтной `XOR DX, DX` или даже трехбайтной `MOV DX, 0`.
- 1-байтная `XCHG AX, CX` используется вместо двухбайтной `MOV AX, CX`. Текущее значение в AX все равно уже не нужно.
- DI (позиция в видеобufferе) не инициализирована, и будет 0xFFFFE в начале ⁹. Это нормально, потому что программа работает бесконечно для всех DI в пределах 0..0xFFFF, и пользователь не может увидеть, что работала началась за экраном (последний пиксель видеобufferа 320*200 имеет адрес 0xF9FF). Так что некоторая часть работы на самом деле происходит за экраном. А иначе понадобятся дополнительные инструкции для установки DI в 0; добавить проверку на конец буфера.

83.2.3. Моя «исправленная» версия

Листинг 83.6: Моя «исправленная» версия

```
1  org 100h
2  mov al,13h
3  int 10h
4
5  ; установить палитру
6  mov dx, 3c8h
7  mov al, 0
8  out dx, al
9  mov cx, 100h
10 inc dx
11 100:
12 mov al, cl
13 shl ax, 2
14 out dx, al ; красный
15 out dx, al ; зеленый
16 out dx, al ; синий
17 loop 100
18
19 push 0a000h
20 pop es
21
22 xor di, di
23
24 FillLoop:
25 cwd
26 mov ax,di
27 mov cx,320
28 div cx
29 sub ax,100
30 sub dx,160
31
32 xor bx,bx
33 xor si,si
34
35 MandelLoop:
36 mov bp,si
37 imul si,bx
38 add si,si
39 imul bx,bx
40 jo MandelBreak
41 imul bp,bp
42 jo MandelBreak
43 add bx,bp
44 jo MandelBreak
45 sub bx,bp
46 sub bx,bp
```

⁹Больше о состояниях регистров на старте: <http://go.yurichev.com/17004>

83.2. МНОЖЕСТВО МАНДЕЛЬБРОТА

```
47
48 sar bx,6
49 add bx,dx
50 sar si,6
51 add si,ax
52
53 loop MandelLoop
54
55 MandelBreak:
56 xchg ax,cx
57 stosb
58 cmp di, 0FA00h
59 jb FillLoop
60
61 ; дождаться нажатия любой клавиши
62 xor ax,ax
63 int 16h
64 ; установить текстовый видеорежим
65 mov ax, 3
66 int 10h
67 ; выход
68 int 20h
```

Автор сих строк попытался исправить все эти странности: теперь палитра плавная черно-белая, видеобуфер на правильном месте (строки 19..20), картинка рисуется в центре экрана (строка 30), программа в итоге заканчивается и ждет, пока пользователь нажмет какую-нибудь клавишу (строки 58..68). Но теперь она намного больше: 105 байт (или 54 инструкции) ¹⁰.

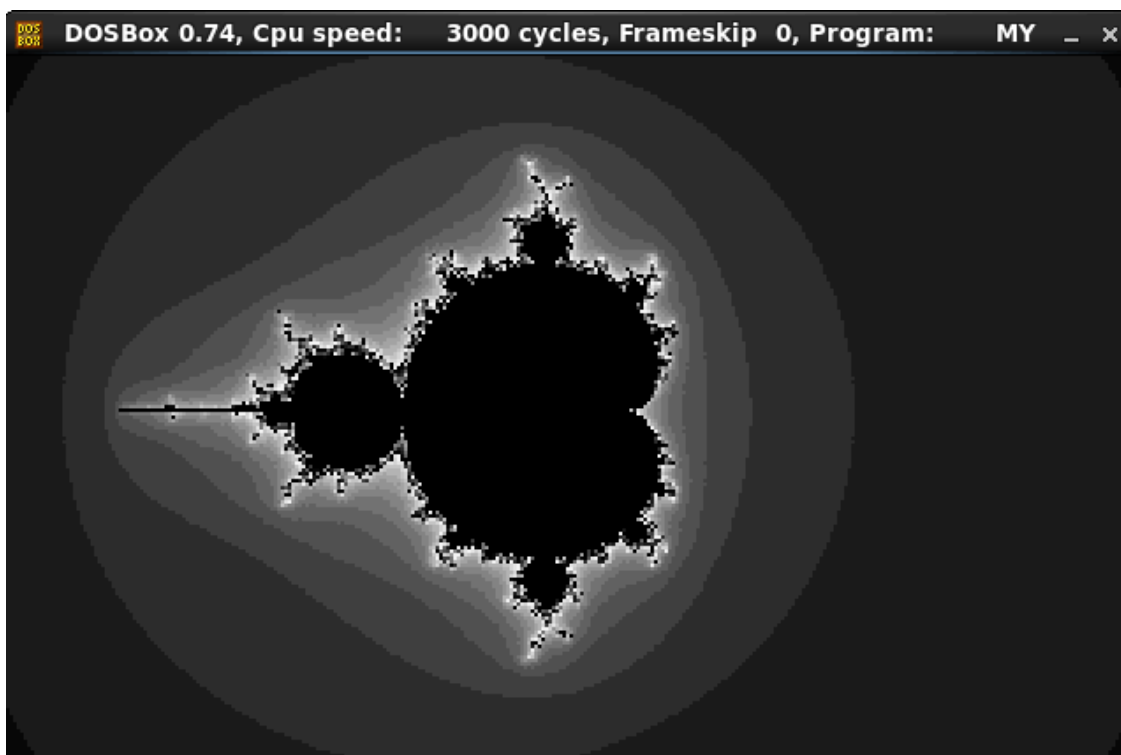


Рис. 83.3: Моя «исправленная» версия

¹⁰Можете поэкспериментировать и сами: скачайте DosBox и NASM и компилируйте так: `nasm fiole.asm -fbin -o file.com`

Глава 84

”Прикуп” в игре ”Марьяж”

Знал бы прикуп — жил бы в Сочи.

Поговорка.

”Марьяж” — старая и довольно популярная версия игры в ”Преферанс” под DOS.

Играют три игрока, каждому раздается по 10 карт, остальные 2 остаются в т.н. ”прикупе”. Начинаются торги, во время которых ”прикуп” скрыт. Он открывается после того, как один из игроков сделает ”заказ”.

Знание карт в ”прикупе” обычно имеет решающее преимущество.

Вот так в игре выглядит состояние ”торгов”, и ”прикуп” посредине, скрытый:



Рис. 84.1: ”Торги”

Попробуем ”подсмотреть” карты в ”прикупе” в этой игре.

Для начала — что мы знаем? Игра под DOS, датируется 1997-м годом. IDA показывает имена стандартных функций вроде `@GetImage$q7Integert1t1t1m3Any` — это ”манглинг” типичный для Borland Pascal, что позволяет сделать вывод, что сама игра написана на Паскале и скомпилирована Borland Pascal-ем.

Файлов около 10-и и некоторые имеют текстовую строку в заголовке "Marriage Image Library" — вероятно, это библиотеки спрайтов.

В IDA можно увидеть что используется функция `@PutImage$q7Integert1m3Any4Word`, которая, собственно, рисует некий спрайт на экране. Она вызывается по крайней мере из 8-и мест. Чтобы узнать что происходит в каждом из этих 8-и мест, мы можем блокировать работу каждой функции и смотреть, что будет происходить. Например, первая ф-ция имеет адрес `seg002:062E`, и она заканчивается инструкцией `retf 0Eh` на `seg002:102A`. Это означает что метод вызовов ф-ций в Borland Pascal под DOS схож с `stdcall` — вызываемая ф-ция должна сама возвращать стек в состояние до того как началась передача аргументов. В самом начале этой ф-ции вписываем инструкцию `"retf 0eh"`, либо 3 байта: `CA 0E 00`. Запускаем "Марьяж" и внешне вроде бы ничего не изменилось.

Переходим ко второй ф-ции, которая активно использует `@PutImage$q7Integert1m3Any4Word`. Она находится по адресу `seg008:0AB5` и заканчивается инструкцией `retf 0Ah`. Вписываем эту инструкцию в самом начале и запускаем:



Рис. 84.2: Карт нет

Карт не видно вообще. И видимо, эта функция их отображает, мы её заблокировали, и теперь карт не видно. Назовем эту ф-цию в IDA `draw_card()`. Помимо `@PutImage$q7Integert1m3Any4Word`, в этой ф-ции вызываются также ф-ции `@SetColor$q4Word`, `@SetFillStyle$q4Wordt1`, `@Bar$q7Integert1t1t1`, `@OutTextXY$q7Integert16String`.

Сама ф-ция `draw_cards()` (её название мы дали ей сами только что) вызывается из 4-х мест. Попробуем точно также "блокировать" каждую ф-цию.

Когда я "блокирую" вторую, по адресу `seg008:0DF3` и запускаю программу, вижу такое:

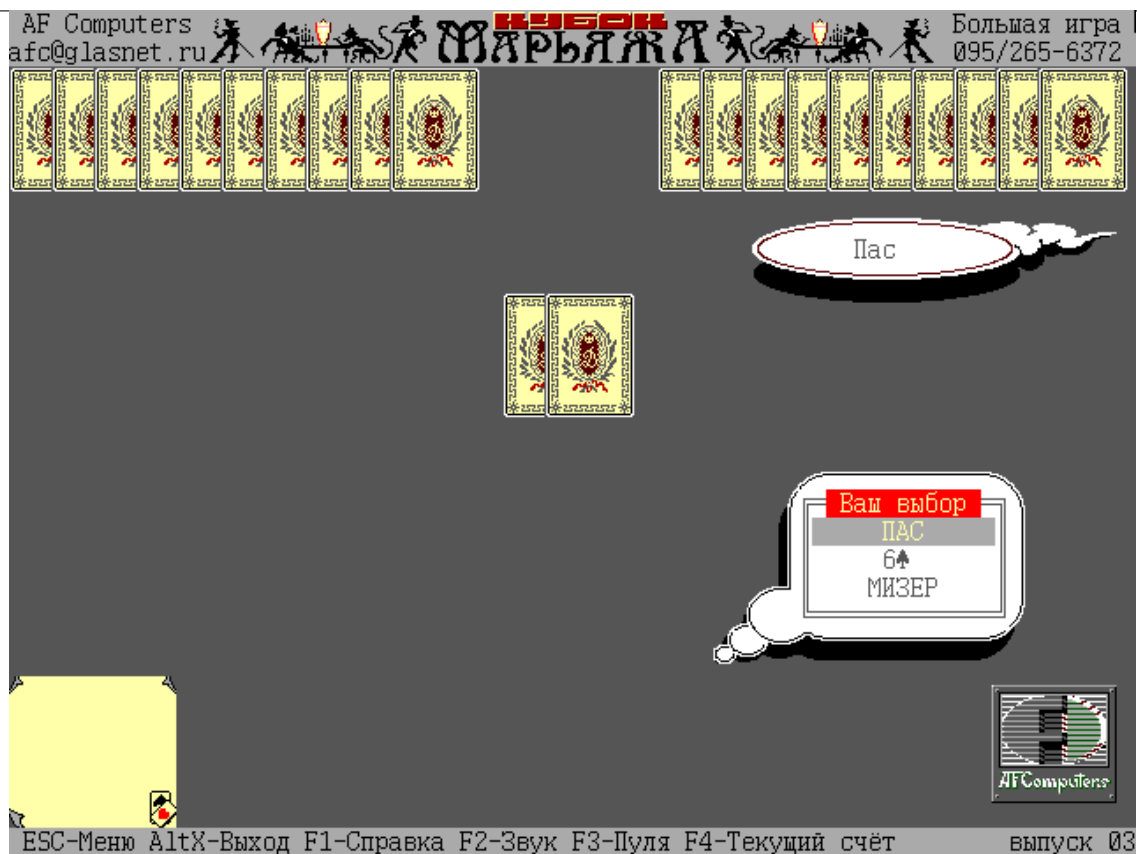


Рис. 84.3: Все карты кроме карт игрока

Видны все карты, кроме карт игрока. Видимо, эта функция рисует карты игрока.

Я переименовываю её в IDA в `draw_players_cards()`.

Четвертая ф-ция, вызывающая `draw_cards()`, находится по адресу `seg008:16B3`, и когда я её "блокирую", я вижу в игре такое:



Рис. 84.4: "Прикупа" нет

Все карты есть, кроме "прикупа". Более того, эта ф-ция вызывает только `draw_cards()`, и только 2 раза. Видимо эта ф-ция и отображает карты "прикупа". Будем рассматривать её внимательнее.

```

seg008:16B3 draw_prikup      proc far                      ; CODE XREF: seg010:00B0
seg008:16B3                                     ; sub_15098+6
seg008:16B3
seg008:16B3 var_E           = word ptr -0Eh
seg008:16B3 var_C           = word ptr -0Ch
seg008:16B3 arg_0           = byte ptr 6
seg008:16B3
seg008:16B3                enter    0Eh, 0
seg008:16B7                mov     al, byte_2C0EA
seg008:16BA                xor     ah, ah
seg008:16BC                imul    ax, 23h
seg008:16BF                mov     [bp+var_C], ax
seg008:16C2                mov     al, byte_2C0EB
seg008:16C5                xor     ah, ah
seg008:16C7                imul    ax, 0Ah
seg008:16CA                mov     [bp+var_E], ax
seg008:16CD                cmp     [bp+arg_0], 0
seg008:16D1                jnz     short loc_1334A
seg008:16D3                cmp     byte_2BB08, 0
seg008:16D8                jz      short loc_13356
seg008:16DA                loc_1334A:                                ; CODE XREF: draw_prikup+1E
seg008:16DA                mov     al, byte ptr word_32084
seg008:16DD                mov     byte_293AD, al
seg008:16E0                mov     al, byte ptr word_32086
seg008:16E3                mov     byte_293AC, al
seg008:16E6                loc_13356:                                ; CODE XREF: draw_prikup+25
seg008:16E6                mov     al, byte_293AC
seg008:16E9                xor     ah, ah
seg008:16EB                push    ax
seg008:16EC                mov     al, byte_293AD
seg008:16EF                xor     ah, ah
seg008:16F1                push    ax
seg008:16F2                push    [bp+var_C]
seg008:16F5                push    [bp+var_E]
seg008:16F8                cmp     [bp+arg_0], 0
seg008:16FC                jnz     short loc_13379
seg008:16FE                cmp     byte_2BB08, 0
seg008:1703                jnz     short loc_13379
seg008:1705                mov     al, 0
seg008:1707                jmp     short loc_1337B
seg008:1709 ; -----
seg008:1709                loc_13379:                                ; CODE XREF: draw_prikup+49
seg008:1709                                     ; draw_prikup+50
seg008:1709                mov     al, 1
seg008:170B                loc_1337B:                                ; CODE XREF: draw_prikup+54
seg008:170B                push    ax
seg008:170C                push    cs
seg008:170D                call    near ptr draw_card
seg008:1710                mov     al, byte_2C0EA
seg008:1713                xor     ah, ah
seg008:1715                mov     si, ax
seg008:1717                shl     ax, 1
seg008:1719                add     ax, si
seg008:171B                add     ax, [bp+var_C]
seg008:171E                mov     [bp+var_C], ax
seg008:1721                cmp     [bp+arg_0], 0
seg008:1725                jnz     short loc_1339E
seg008:1727                cmp     byte_2BB08, 0
seg008:172C                jz      short loc_133AA
seg008:172E                loc_1339E:                                ; CODE XREF: draw_prikup+72
seg008:172E                mov     al, byte ptr word_32088
seg008:1731                mov     byte_293AD, al

```

```

seg008:1734      mov     al, byte ptr word_3208A
seg008:1737      mov     byte_293AC, al
seg008:173A
seg008:173A loc_133AA:      ; CODE XREF: draw_prikup+79
seg008:173A      mov     al, byte_293AC
seg008:173D      xor     ah, ah
seg008:173F      push    ax
seg008:1740      mov     al, byte_293AD
seg008:1743      xor     ah, ah
seg008:1745      push    ax
seg008:1746      push    [bp+var_C]
seg008:1749      push    [bp+var_E]
seg008:174C      cmp     [bp+arg_0], 0
seg008:1750      jnz     short loc_133CD
seg008:1752      cmp     byte_2BB08, 0
seg008:1757      jnz     short loc_133CD
seg008:1759      mov     al, 0
seg008:175B      jmp     short loc_133CF
seg008:175D ; -----
seg008:175D
seg008:175D loc_133CD:      ; CODE XREF: draw_prikup+9D
seg008:175D      ; draw_prikup+A4
seg008:175D      mov     al, 1
seg008:175F
seg008:175F loc_133CF:      ; CODE XREF: draw_prikup+A8
seg008:175F      push    ax
seg008:1760      push    cs
seg008:1761      call   near ptr draw_card ; prikup #2
seg008:1764      leave
seg008:1765      retf    2
seg008:1765 draw_prikup      endp

```

Интересно посмотреть, как именно вызывается `draw_prikup()`. У нее только один аргумент.

Иногда она вызывается с аргументом 1:

```

...
seg010:084C      push    1
seg010:084E      call   draw_prikup
...

```

А иногда с аргументом 0, причем вот в таком контексте, где уже есть другая знакомая функция:

```

seg010:0067      push    1
seg010:0069      mov     al, byte_31F41
seg010:006C      push    ax
seg010:006D      call   sub_12FDC
seg010:0072      push    1
seg010:0074      mov     al, byte_31F41
seg010:0077      push    ax
seg010:0078      call   draw_players_cards
seg010:007D      push    2
seg010:007F      mov     al, byte_31F42
seg010:0082      push    ax
seg010:0083      call   sub_12FDC
seg010:0088      push    2
seg010:008A      mov     al, byte_31F42
seg010:008D      push    ax
seg010:008E      call   draw_players_cards
seg010:0093      push    3
seg010:0095      mov     al, byte_31F43
seg010:0098      push    ax
seg010:0099      call   sub_12FDC
seg010:009E      push    3
seg010:00A0      mov     al, byte_31F43
seg010:00A3      push    ax
seg010:00A4      call   draw_players_cards
seg010:00A9      call   sub_1257A
seg010:00AE      push    0
seg010:00B0      call   draw_prikup

```

```
seg010:00B5      mov     byte_2BB95, 0
```

Так что единственный аргумент у `draw_prikup()` может быть или 0 или 1, т.е., это, возможно, булевый тип. На что он влияет внутри самой ф-ции? При ближайшем рассмотрении видно, что входящий 0 или 1 передается в `draw_card()`, т.е., у последней тоже есть булевый аргумент. Помимо всего прочего, если передается 1, то по адресам `seg008:16DA` и `seg008:172E` копируются несколько байт из одной группы глобальных переменных в другую.

Эксперимент: здесь 4 раза сравнивается единственный аргумент с 0 и далее следует `JNZ`. Что если сравнение будет происходить с 1, и, таким образом, работа функции `draw_prikup()` будет обратной? Патчим и запускаем:



Рис. 84.5: "Прикуп" открыт

"Прикуп" открыт, но когда я делаю "заказ", и, по логике вещей, "прикуп" теперь должен стать открытым, он наоборот становится закрытым:



Рис. 84.6: "Прикуп" закрыт

Всё ясно: если аргумент `draw_prikup()` нулевой, то карты рисуются рубашкой вверх, если 1, то открытые. Этот же аргумент передается в `draw_card()` — эта ф-ция может рисовать и открытые и закрытые карты.

Пропатчить "Марьяж" теперь легко, достаточно исправить все условные переходы так, как будто бы в ф-цию всегда приходит 1 в аргументе и тогда "прикуп" всегда будет открыт.

Но что за байты копируются в `seg008:16DA` и `seg008:172E`? Я попробовал забить инструкции копирования `MOV` `NOP` - ами — "прикуп" вообще перестал отображаться.

Тогда я сделал так, чтобы всегда записывалась 1:

```

...
00004B5A: B001          mov     al,1
00004B5C: 90             nop
00004B5D: A26D08        mov     [0086D],al
00004B60: B001          mov     al,1
00004B62: 90             nop
00004B63: A26C08        mov     [0086C],al
...

```

Тогда "прикуп" отображается как два пиковых туза. А если первый байт — 2, а второй — 1, получается трефовый туз. Видимо так и кодируется масть карты, а затем и сама карта. А `draw_card()` затем считывает эту информацию из пары глобальных переменных. А копируется она тоже из глобальных переменных, где собственно и находится состояние карт у игроков и в прикупе после случайной тасовки. Но нельзя забывать что если мы сделаем так, что в "прикупе" всегда будет 2 пиковых туза, это будет только на экране так отображаться, а в памяти состояние карт останется таким же, как и после тасовки.

Всё понятно: автор решил сделать одну ф-цию для отрисовки и закрытого и открытого прикупа, поэтому нам, можно сказать, повезло. Могло быть труднее: в самом начале рисовались бы просто две рубашки карт, а открытый прикуп только потом.

Я также пробовал сделать шутку-пранк: во время торгов одна карта "прикупа" открыта, а вторая закрыта, а после "заказа", наоборот, первая закрыта, а вторая открывается. В качестве упражнения, вы можете попробовать сделать так.

Еще кое-что: чтобы сделать прикуп открытым, ведь можно же найти место где вызывается `draw_prikup()` и поменять 0 на 1. Можно, только это место не в головной `marriage.exe`, а в `marriage.000`, а это DOS-овский оверлей (начинается с сигнатуры "FBOV").

В качестве упражнения, можно попробовать подсматривать состояние всех карт, и у обоих игроков. Для этого нужно отладчиком смотреть состояние глобальной памяти рядом с тем, откуда считываются обе карты прикупа.

Файлы:

оригинальная версия: <http://beginners.re/examples/marriage/original.zip>,

пропатченная мною версия: <http://beginners.re/examples/marriage/patched.zip> (все 4 условных перехода после `cmp [bp+arg_0], 0` заменены на `JMP`).

Глава 85

Другие примеры

Здесь также был пример с Z3 и ручной декомпиляцией. Он (временно) перемещен сюда: http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf.

Часть IX

Примеры разбора закрытых (proprietary) форматов файлов

Глава 86

Примитивное XOR-шифрование

86.1. Norton Guide: простейшее однобайтное XOR-шифрование

Norton Guide¹ был популярен во времена MS-DOS, это была резидентная программа, работающая как гипертекстовый справочник.

Базы данных Norton Guide это файлы с расширением .ng, содержимое которых выглядит как зашифрованное:

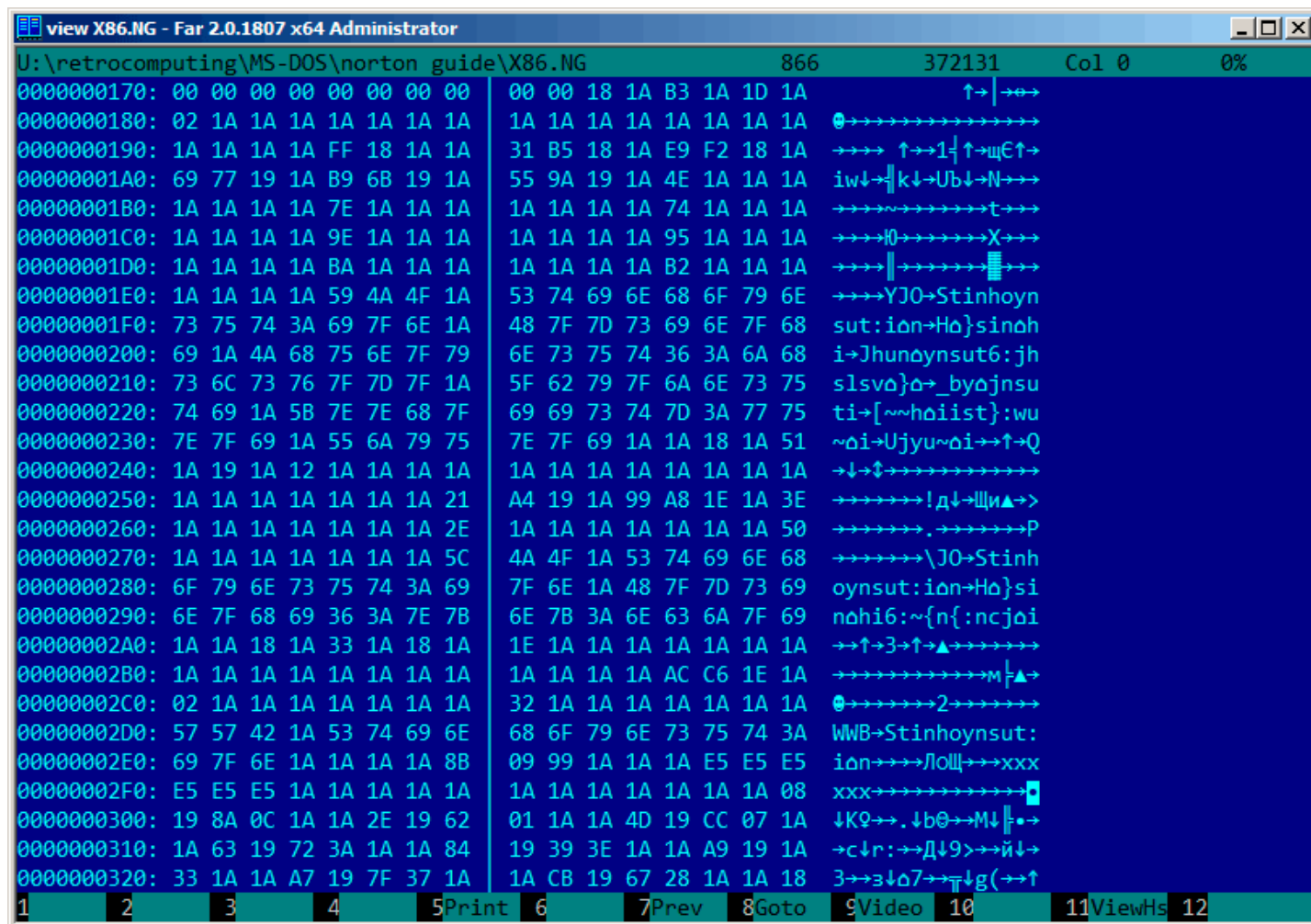


Рис. 86.1: Очень типичный вид

Почему мы думаем, что зашифрованное а не сжатое? Мы видим, как слишком часто попадаетея байт 0x1A (который выглядит как «→»), в сжатом файле такого не было бы никогда. Во-вторых, мы видим длинные части состоящие только из латинских букв, они выглядят как строки на незнакомом языке.

¹wikipedia

86.1. NORTON GUIDE: ПРОСТЕЙШЕЕ ОДНОБАЙТНОЕ XOR-ШИФРОВАНИЕ

Из-за того, что байт 0x1A слишком часто встречается, мы можем попробовать расшифровать файл, полагая что он зашифрован простейшим XOR-шифрованием. Применяем XOR с константой 0x1A к каждому байту в Hiew и мы можем видеть знакомые текстовые строки на английском:

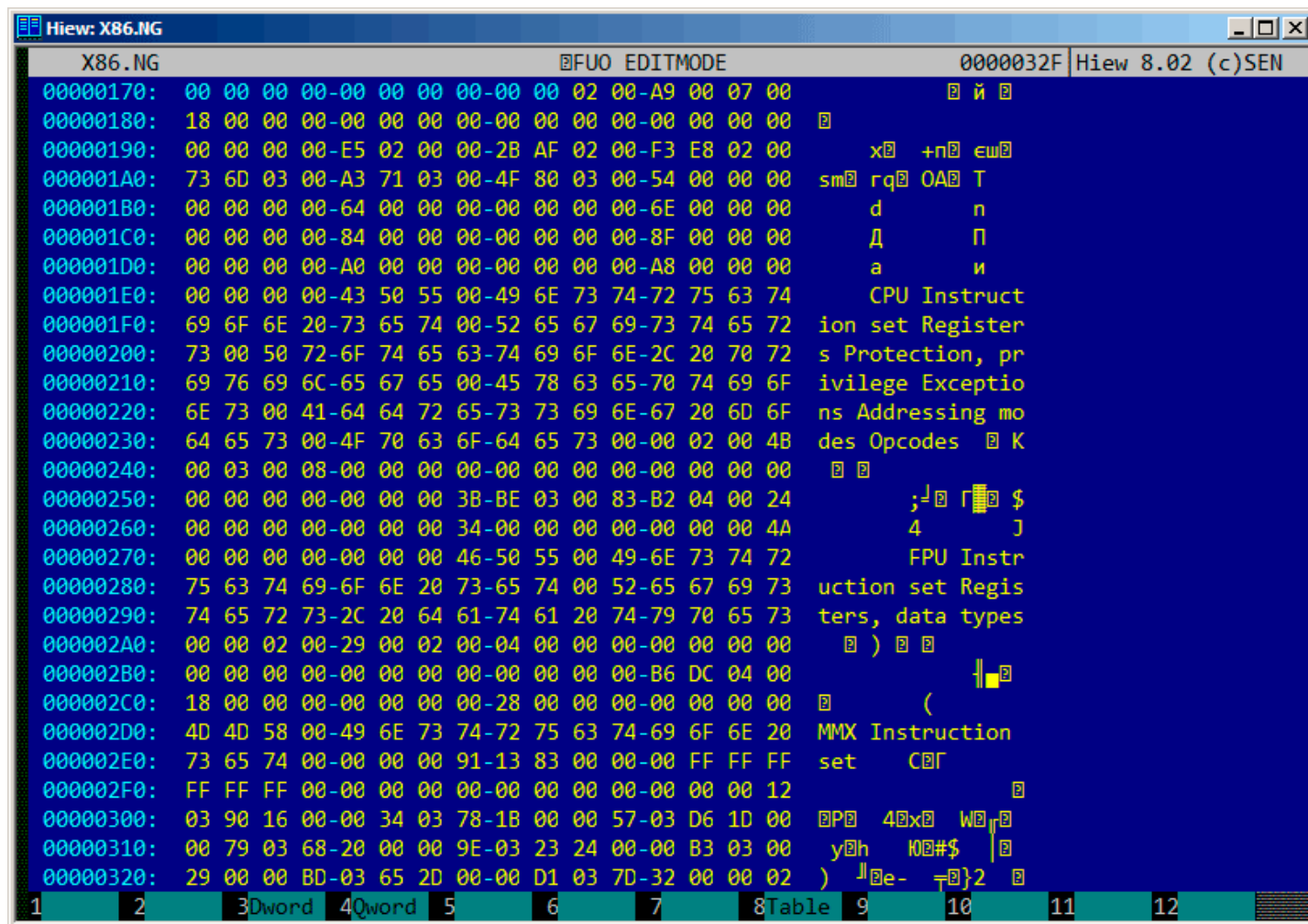


Рис. 86.2: Hiew применение XOR с 0x1A

XOR-шифрование с одним константным байтом это самый простой способ шифрования, который, тем не менее, иногда встречается.

Теперь понятно почему байт 0x1A так часто встречался: потому что в файле очень много нулевых байт и в зашифрованном виде они везде были заменены на 0x1A.

Но эта константа могла быть другой. В таком случае, можно было бы попробовать перебрать все 256 комбинаций, и посмотреть содержимое «на глаз», а 256 — это совсем немного.

Больше о формате файлов Norton Guide: <http://go.yurichev.com/17317>.

86.1.1. Энтропия

Очень важное свойство подобного примитивного шифрования в том, что информационная энтропия зашифрованного/-дешифрованного блока точно такая же. Вот мой анализ в Wolfram Mathematica 10.

Листинг 86.1: Wolfram Mathematica 10

```
In[1]:= input = BinaryReadList["X86.NG"];

In[2]:= Entropy[2, input] // N
Out[2]= 5.62724

In[3]:= decrypted = Map[BitXor[#, 16^^1A] &, input];

In[4]:= Export["X86_decrypted.NG", decrypted, "Binary"];
```

```
In[5]:= Entropy[2, decrypted] // N
Out[5]= 5.62724

In[6]:= Entropy[2, ExampleData[{"Text", "ShakespearesSonnets"}]] // N
Out[6]= 4.42366
```

Что мы здесь делаем это загружаем файл, вычисляем его энтропию, дешифруем его, сохраняем, снова вычисляем энтропию (точно такая же!). Mathematica дает возможность анализировать некоторые хорошо известные англоязычные тексты. Так что мы вычисляем энтропию сонnetов Шейкспира, и она близка к энтропии анализируемого нами файла. Анализируемый нами файл состоит из предложений на английском языке, которые близки к языку Шейкспира. И применение побайтового XOR к тексту на английском языке не меняет энтропию.

Хотя, это не будет справедливо когда файл зашифрован при помощи XOR шаблоном длинее одного байта.

Файл, который мы анализировали, можно скачать здесь: <http://go.yurichev.com/17350>.

Еще кое-что о базе энтропии

Wolfram Mathematica вычисляет энтропию с базой e (основание натурального логарифма), а утилита UNIX *ent*² использует базу 2. Так что мы явно указываем базу 2 в команде `Entropy`, чтобы Mathematica давала те же результаты, что и утилита *ent*.

²<http://www.fourmilab.ch/random/>

Hiw: kernel32.dll

C:\tmp2\kernel32.dll PE .7DD60000 Hiw 8.02 (c)SEN

MZP 0 0 1 @

e run in DOS mode.

\$ 0g0YMeMveMveD~xe |0veM0we>0veD~feL0veD~yeE0veD~ie0veD~Geo0veD~teL0veD~chel0veRichM0ve

PE LEB EBOS p 0!000 0 Y20 0 r} 0 0 0 0 0 0 0

0 o00 0 @0 0 0 0 0 0 p 0 0 \$й0 İ0 0 (0 0 бн 40 8

050 @ 0 È

.data 00 0 0 0 @ L.rsrc (0 0 0 0 @ @.reloc бн 0

0 0 @ B

1 2 3 4ReLoad 5 6String 7Direct 8Table 9 10Leave 11 12

855

Вот он же, но «зашифрованный» 4-байтным ключом:

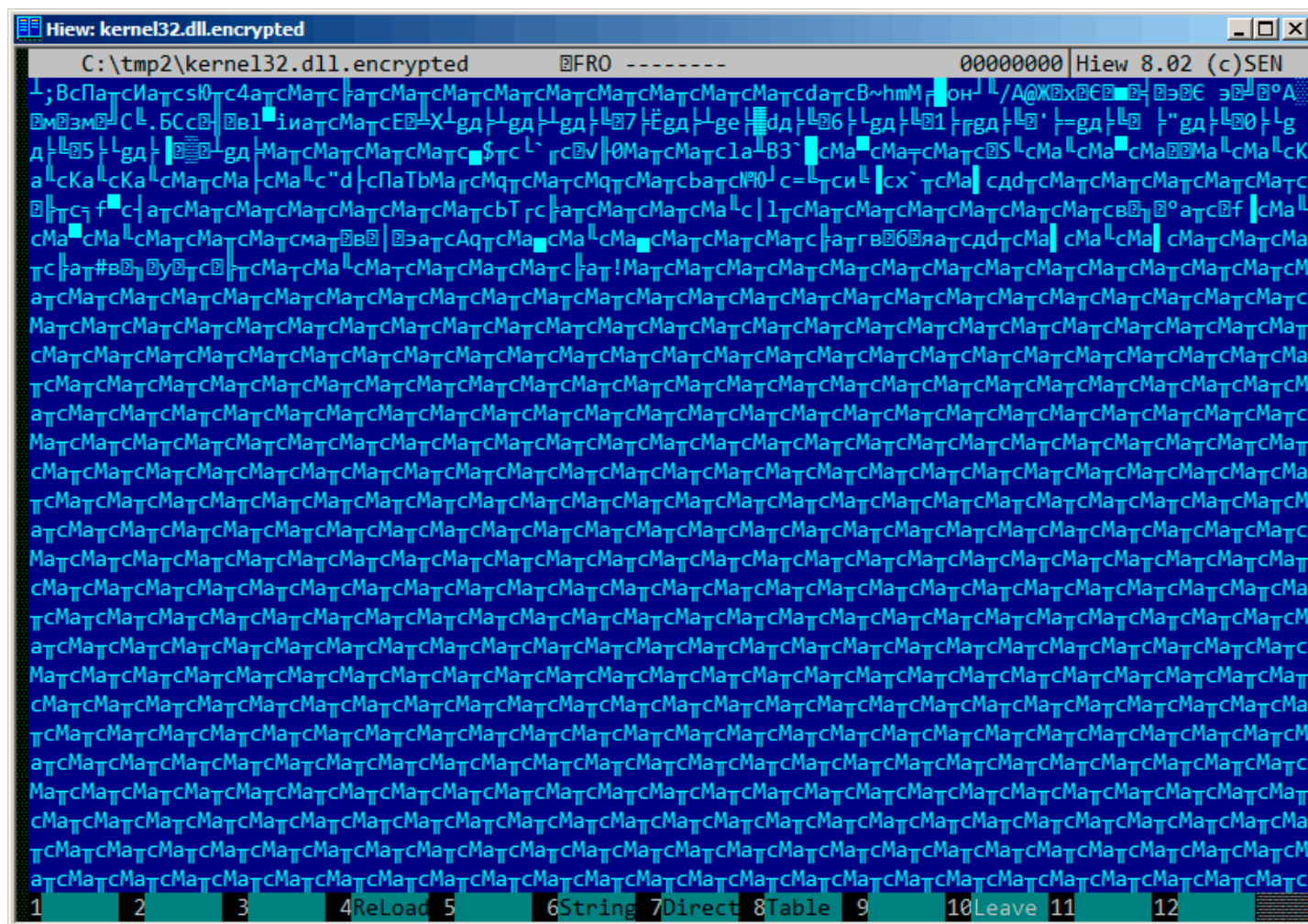


Рис. 86.4: «Зашифрованный» файл

Очень легко увидеть повторяющиеся 4 символа. Ведь в заголовке PE-файла много длинных нулевых областей, из-за которых ключ становится видимым.

Вот начало РЕ-заголовка в 16-ричном виде:



И вот он же, «зашифрованный»:

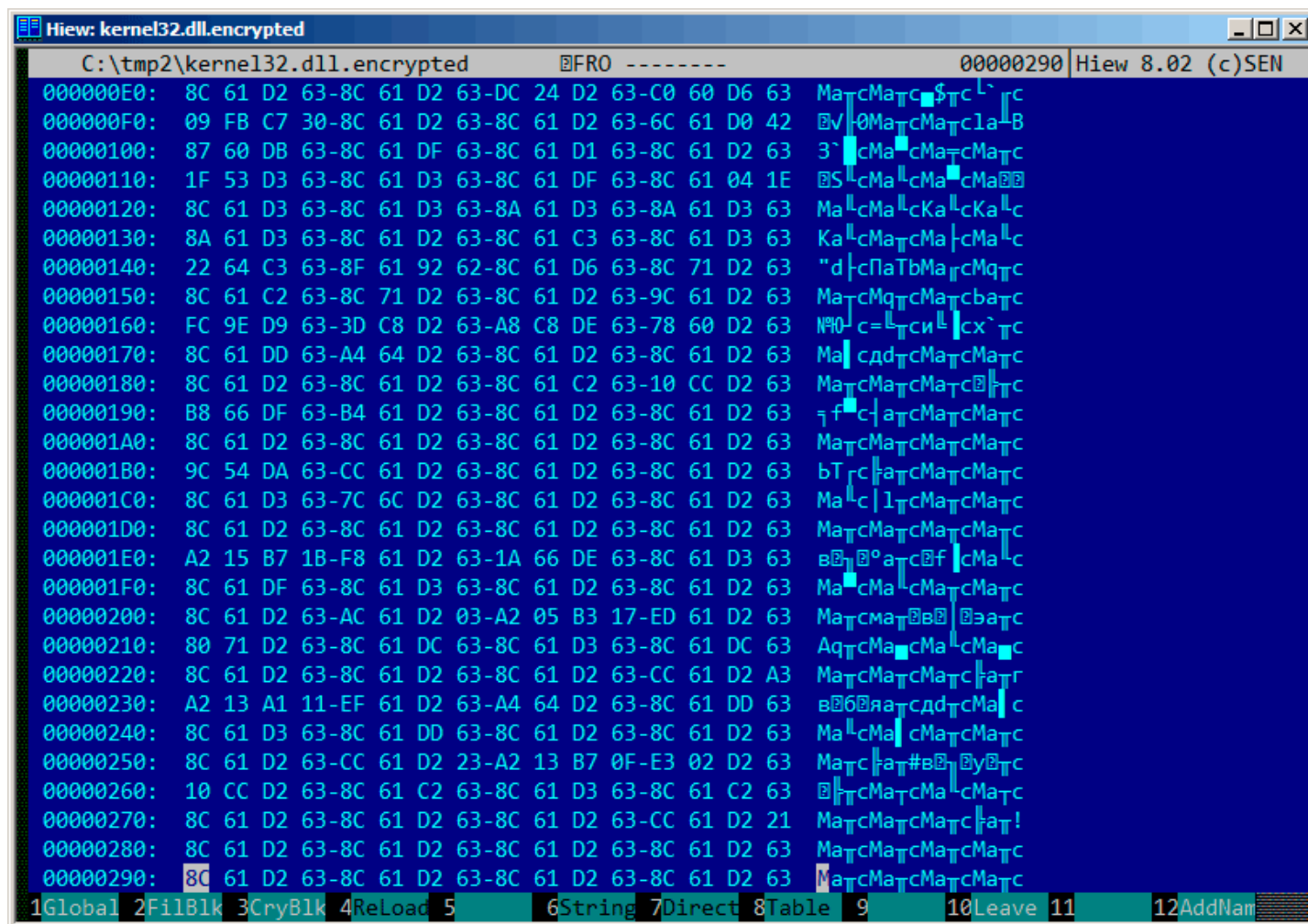


Рис. 86.6: «Зашифрованный» PE-заголовок

Легко увидеть визуально, что ключ это следующие 4 байта : 8C 61 D2 63 . Используя эту информацию, довольно легко расшифровать весь файл.

Таким образом, важно помнить эти свойства PE-файлов: 1) в PE-заголовке много нулевых областей; 2) все PE-секции дополняются нулями до границы страницы (4096 байт), так что после всех секций обычно имеются длинные нулевые области.

Некоторые другие форматы файлов могут также иметь длинные нулевые области. Это очень типично для файлов, используемых научным и инженерным ПО.

Для тех, кто самостоятельно хочет изучить эти файлы, то их можно скачать здесь: <http://go.yurichev.com/17352>.

86.2.1. Упражнение

- <http://challenges.re/50>

Глава 87

Файл сохранения состояния в игре Millenium

Игра «Millenium Return to Earth» под DOS довольно древняя (1991), позволяющая добывать ресурсы, строить корабли, снаряжать их на другие планеты, и т.д. ¹.

Как и многие другие игры, она позволяет сохранять состояние игры в файл.

Посмотрим, сможем ли мы найти что-нибудь в нем.

¹Её можно скачать бесплатно [здесь](#)

В игре есть шахта. Шахты на некоторых планетах работают быстрее, на некоторых других – медленнее. Набор ресурсов также разный.

Здесь видно, какие ресурсы добыты в этот момент:



Рис. 87.1: Шахта: первое состояние

Сохраним состояние игры. Это файл размером 9538 байт.

Подождем несколько «дней» здесь в игре и теперь в шахте добыто больше ресурсов:



Рис. 87.2: Шахта: второе состояние

Снова сохраним состояние игры.

Теперь просто попробуем сравнить оба файла побайтово используя простую утилиту FC под DOS/Windows:

```
...> FC /b 2200save.i.v1 2200SAVE.I.V2

Comparing files 2200save.i.v1 and 2200SAVE.I.V2
00000016: 0D 04
00000017: 03 04
0000001C: 1F 1E
00000146: 27 3B
00000BDA: 0E 16
00000BDC: 66 9B
00000BDE: 0E 16
00000BE0: 0E 16
00000BE6: DB 4C
00000BE7: 00 01
00000BE8: 99 E8
00000BEC: A1 F3
00000BEE: 83 C7
00000BFB: A8 28
00000BFD: 98 18
00000BFF: A8 28
00000C01: A8 28
00000C07: D8 58
00000C09: E4 A4
00000C0D: 38 B8
00000C0F: E8 68
...
```

Вывод здесь неполный, там было больше отличий, но мы обрежем результат до самого интересного.

В первой версии у нас было 14 единиц водорода (hydrogen) и 102 — кислорода (oxygen). Во второй версии у нас 22 и 155 единиц соответственно. Если эти значения сохраняются в файл, мы должны увидеть разницу. И она действительно есть. Там 0x0E (14) на позиции 0xBDA и это значение 0x16 (22) в новой версии файла. Это, наверное, водород. Там также 0x66 (102) на позиции 0xBDC в старой версии и 0x9B (155) в новой версии файла. Это, наверное, кислород.

Обе версии файла доступны на сайте, для тех кто хочет их изучить (или поэкспериментировать): beginners.re.

Новую версию файла откроем в Hiew и отметим значения, связанные с ресурсами, добытыми на шахте в игре:

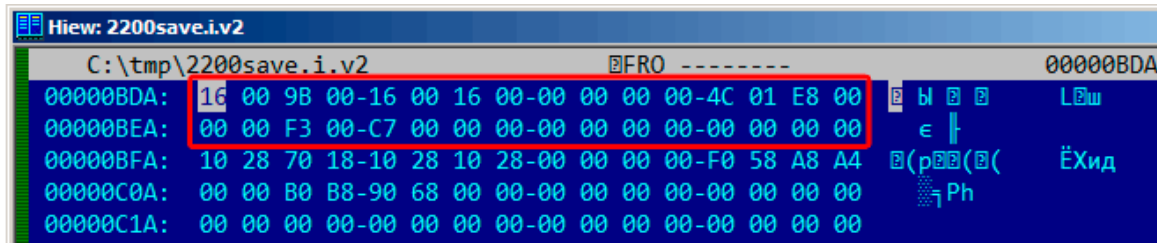


Рис. 87.3: Hiew: первое состояние

Проверим каждое, и это они. Это явно 16-битные значения: не удивительно для 16-битной программы под DOS, где *int* имел длину в 16 бит.

Проверим наши предположения. Запишем 1234 (0x4D2) на первой позиции (это должен быть водород):

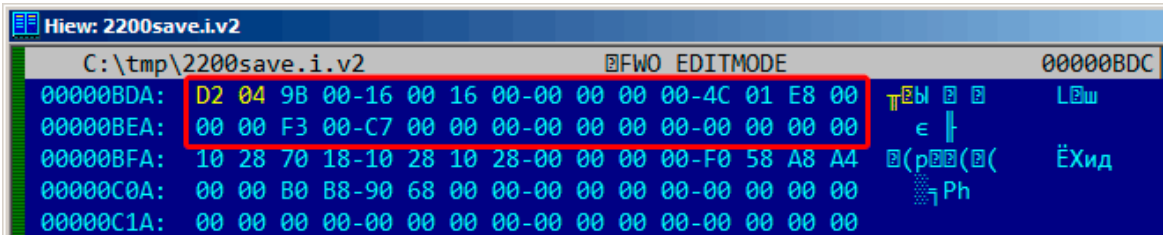


Рис. 87.4: Hiew: запишем там (0x4D2)

Затем загрузим измененный файл в игру и посмотрим на статистику в шахте:



Рис. 87.5: Проверим значение водорода

Так что да, это оно.

Попробуем пройти игру как можно быстрее, установим максимальные значения везде:

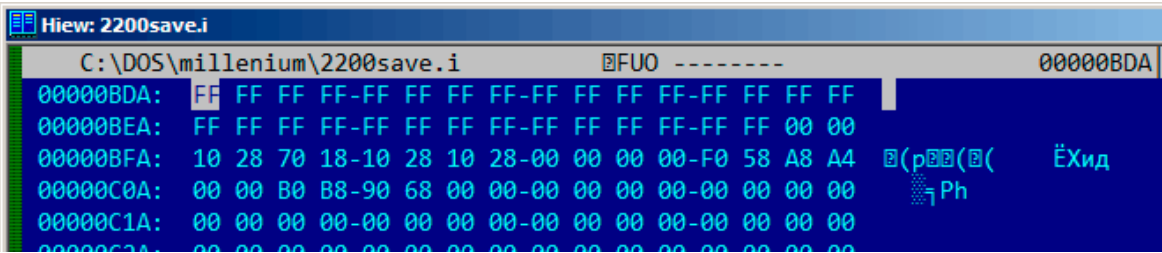


Рис. 87.6: Hiew: установим максимальные значения

0xFFFF это 65535, так что да, у нас много ресурсов теперь:



Рис. 87.7: Все ресурсы теперь действительно 65535 (0xFFFF)

Пропустим еще несколько «дней» в игре и видим что-то неладное! Некоторых ресурсов стало меньше:



Рис. 87.8: Переполнение переменных ресурсов

Это просто переполнение. Разработчик игры вероятно никогда не думал, что значения ресурсов будут такими большими, так что, здесь, наверное, нет проверок на переполнение, но шахта в игре «работает», ресурсы добавляются, отсюда и переполнение. Вероятно, не нужно было жадничать.

Здесь наверняка еще какие-то значения в этом файле.

Так что это очень простой способ читинга в играх. Файл с таблицей очков также можно легко модифицировать.

Еще насчет сравнения файлов и снимков памяти: [64.4](#) (стр. [669](#)).

Глава 88

Oracle RDBMS: .SYM-файлы

Когда процесс в Oracle RDBMS терпит серьезную ошибку (crash), он записывает массу информации в лог-файлы, включая состояние стека, вроде:

```
----- Call Stack Trace -----
calling      call      entry      argument values in hex
location     type      point      (? means dubious value)
-----
_kqvwrow()           00000000
_opifch2()+2729      CALLptr   00000000      23D4B914 E47F264 1F19AE2
                                     EB1C8A8 1
_kpoal8()+2832       CALLrel   _opifch2()     89 5 EB1CC74
_opiodr()+1248       CALLreg   00000000      5E 1C EB1F0A0
_ttcpip()+1051       CALLreg   00000000      5E 1C EB1F0A0 0
_opitsk()+1404       CALL???   00000000      C96C040 5E EB1F0A0 0 EB1ED30
                                     EB1F1CC 53E52E 0 EB1F1F8
_opiino()+980        CALLrel   _opitsk()      0 0
_opiodr()+1248       CALLreg   00000000      3C 4 EB1FBF4
_opidrv()+1201       CALLrel   _opiodr()      3C 4 EB1FBF4 0
_sou2o()+55          CALLrel   _opidrv()      3C 4 EB1FBF4
_opimai_real()+124   CALLrel   _sou2o()       EB1FC04 3C 4 EB1FBF4
_opimai()+125        CALLrel   _opimai_real() 2 EB1FC2C
_OracleThreadStart@ CALLrel   _opimai()      2 EB1FF6C 7C88A7F4 EB1FC34 0
4()+830              EB1FD04
77E6481C             CALLreg   00000000      E41FF9C 0 0 E41FF9C 0 EB1FFC4
00000000             CALL???   00000000
```

Но конечно, для этого исполняемые файлы Oracle RDBMS должны содержать некоторую отладочную информацию, либо тар-файлы с информацией о символах или что-то в этом роде.

Oracle RDBMS для Windows NT содержит информацию о символах в файлах с расширением .SYM, но его формат закрыт. (Простые текстовые файлы — это хорошо, но они требуют дополнительной обработки (парсинга), и из-за этого доступ к ним медленнее.)

Посмотрим, сможем ли мы разобрать его формат. Выберем самый короткий файл `orawtc8.sym`, поставляемый с файлом `orawtc8.dll` в Oracle 8.1.7 ¹.

¹Будем использовать древнюю версию Oracle RDBMS сознательно, из-за более короткого размера его модулей

Вот я открываю этот файл в Hiew:

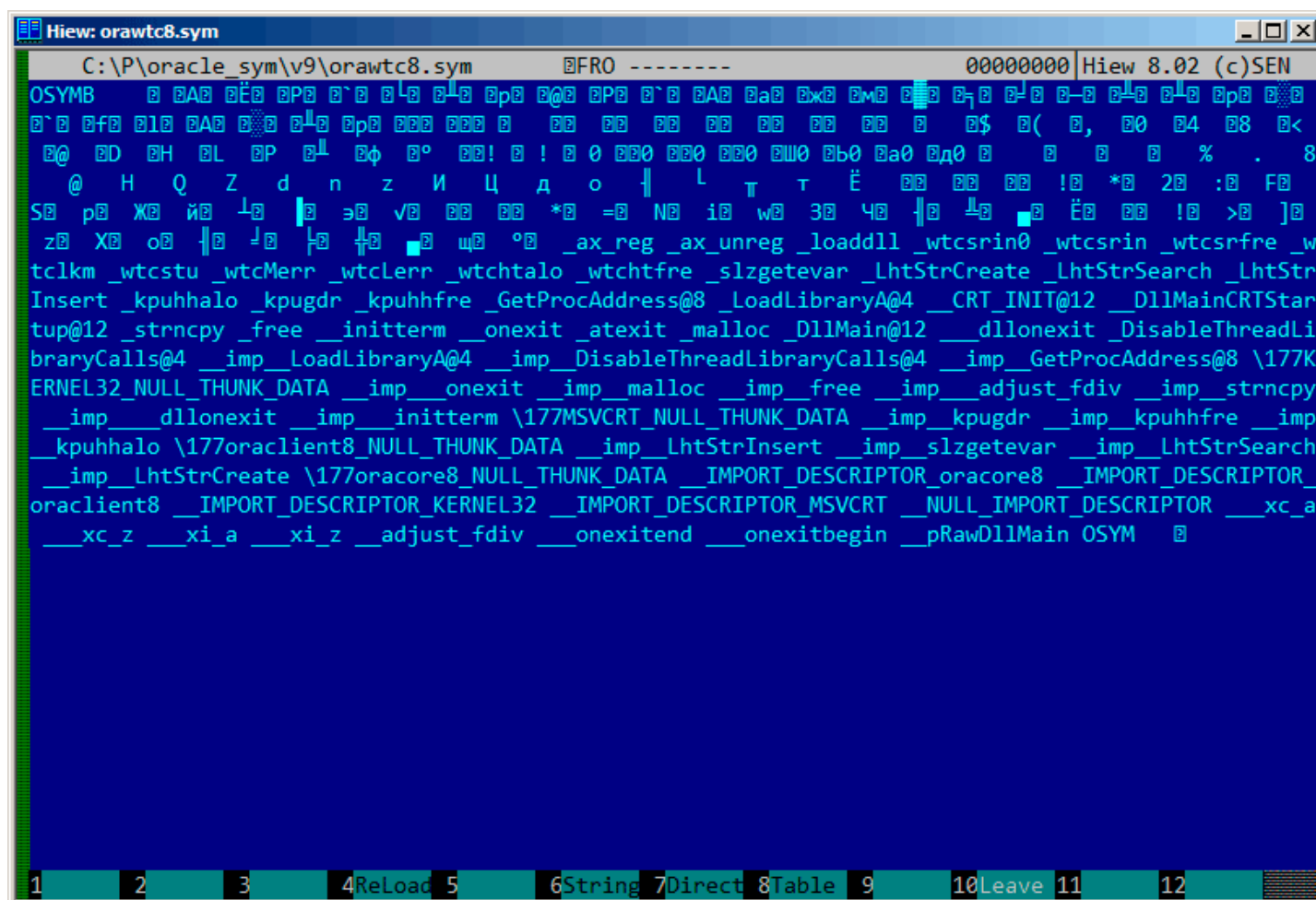


Рис. 88.1: Весь файл в Hiew

Сравнивая этот файл с другими .SYM-файлами, мы можем быстро заметить, что **OSYM** всегда является заголовком (и концом), так что это, наверное, сигнатура файла.

Мы также видим, что в общем-то, формат файла это: OSYM + какие-то бинарные данные + текстовые строки разделенные нулем + OSYM. Строки – это, очевидно, имена функций и глобальных переменных.

Отметим сигнатуры OSYM и строки здесь:

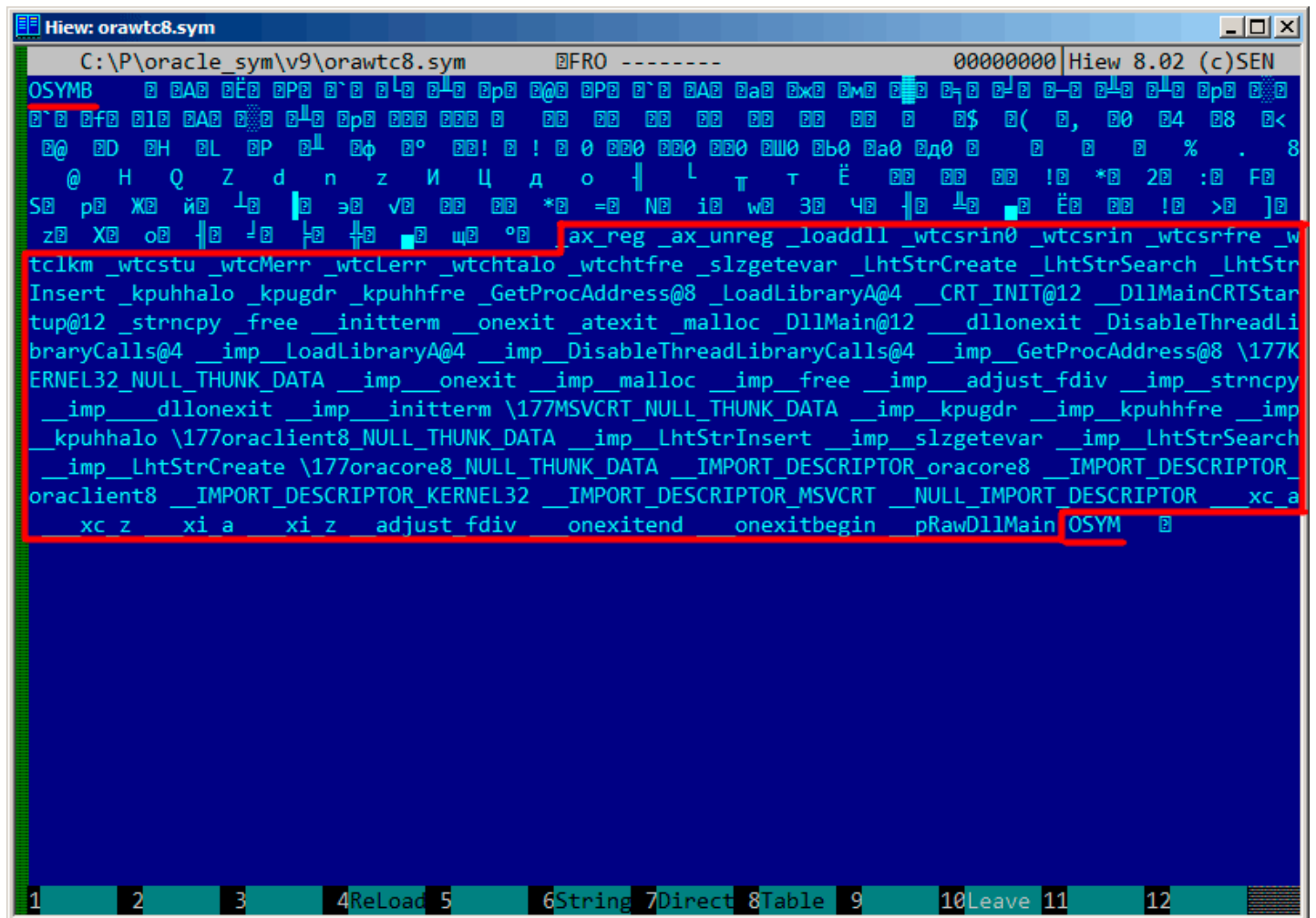


Рис. 88.2: Сигнатура OSYM и текстовые строки

Посмотрим. В Hiew отметим весь блок со строками (исключая оконечивающую сигнатуру OSYM) и сохраним его в отдельный файл. Затем запустим UNIX-утилиты *strings* и *wc* для подсчета текстовых строк:

```
strings strings_block | wc -l
66
```

Так что здесь 66 текстовых строк. Запомните это число.

Можно сказать, что в общем, как правило, количество *чего-либо* часто сохраняется в бинарном файле отдельно. Это действительно так, мы можем найти значение 66 (0x42) в самом начале файла, прямо после сигнатуры OSYM:

```
$ hexdump -C orawtc8.sym
00000000  4f 53 59 4d 42 00 00 00 00 10 00 10 80 10 00 10 |OSYMB.....|
00000010  f0 10 00 10 50 11 00 10 60 11 00 10 c0 11 00 10 |....P...`.....|
00000020  d0 11 00 10 70 13 00 10 40 15 00 10 50 15 00 10 |....p...@...P...|
00000030  60 15 00 10 80 15 00 10 a0 15 00 10 a6 15 00 10 |`.....|
....
```

Конечно, 0x42 здесь это не байт, но скорее всего, 32-битное значение, упакованное как little-endian, поэтому мы видим 0x42 и затем как минимум 3 байта.

Почему мы полагаем, что оно 32-битное? Потому что файлы с символами в Oracle RDBMS могут быть очень большими. *oracle.sym* для главного исполняемого файла *oracle.exe* (версия 10.2.0.4) содержит 0x3A38E (238478) символов. 16-битного значения тут недостаточно.

Проверим другие .SYM-файлы как этот и это подтвердит нашу догадку: значение после 32-битной сигнатуры OSYM всегда отражает количество текстовых строк в файле.

Это общая особенность почти всех бинарных файлов: заголовок с сигнатурой плюс некоторая дополнительная информация о файле.

Рассмотрим бинарный блок поближе. Снова используя `Niew`, сохраним блок начиная с адреса 8 (т.е. после 32-битного значения, отражающего количество) до блока со строками, в отдельный файл.

Посмотрим этот блок в Hiew:

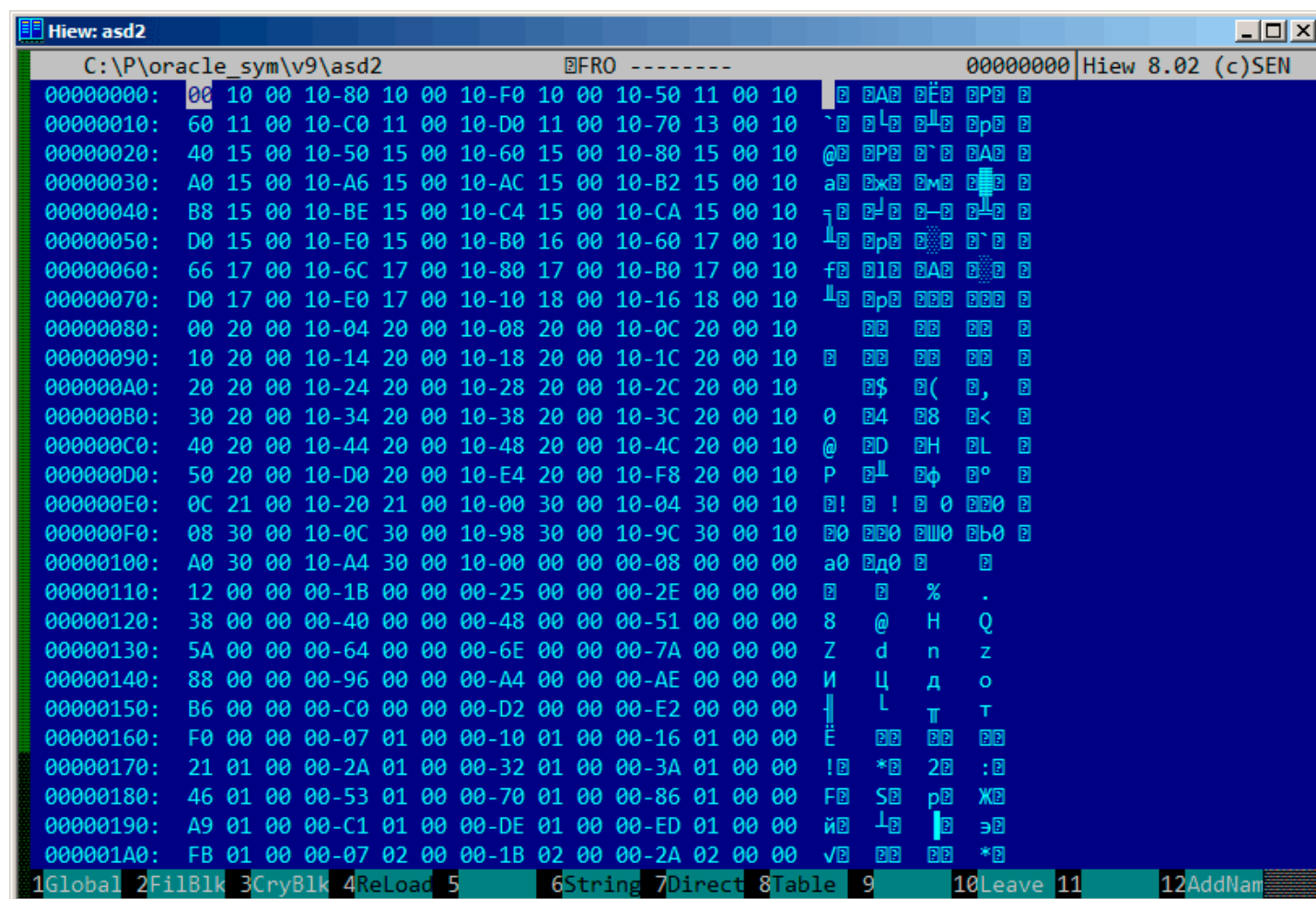


Рис. 88.3: Бинарный блок

Тут явно есть какая-то структура.

Добавим красные линии, чтобы разделить блок:



Рис. 88.4: Структура бинарного блока

Hiew, как и многие другие шестнадцатеричные редакторы, показывает 16 байт на строку. Так что структура явно видна: здесь 4 32-битных значения на строку.

Эта структура видна визуально потому что некоторые значения здесь (вплоть до адреса 0x104) всегда в виде 0x1000xxxx, так что начинаются с байт 0x10 и 0. Другие значения (начинающиеся на 0x108) всегда в виде 0x0000xxxx, так что начинаются с двух нулевых байт.

Посмотрим на этот блок как на массив 32-битных значений:

Листинг 88.1: первый столбец — это адрес

```
$ od -v -t x4 binary_block
0000000 10001000 10001080 100010f0 10001150
0000020 10001160 100011c0 100011d0 10001370
0000040 10001540 10001550 10001560 10001580
0000060 100015a0 100015a6 100015ac 100015b2
0000100 100015b8 100015be 100015c4 100015ca
0000120 100015d0 100015e0 100016b0 10001760
0000140 10001766 1000176c 10001780 100017b0
0000160 100017d0 100017e0 10001810 10001816
0000200 10002000 10002004 10002008 1000200c
0000220 10002010 10002014 10002018 1000201c
0000240 10002020 10002024 10002028 1000202c
0000260 10002030 10002034 10002038 1000203c
0000300 10002040 10002044 10002048 1000204c
0000320 10002050 100020d0 100020e4 100020f8
0000340 1000210c 10002120 10003000 10003004
0000360 10003008 1000300c 10003098 1000309c
0000400 100030a0 100030a4 00000000 00000008
```

```

0000420 00000012 0000001b 00000025 0000002e
0000440 00000038 00000040 00000048 00000051
0000460 0000005a 00000064 0000006e 0000007a
0000500 00000088 00000096 000000a4 000000ae
0000520 000000b6 000000c0 000000d2 000000e2
0000540 000000f0 00000107 00000110 00000116
0000560 00000121 0000012a 00000132 0000013a
0000600 00000146 00000153 00000170 00000186
0000620 000001a9 000001c1 000001de 000001ed
0000640 000001fb 00000207 0000021b 0000022a
0000660 0000023d 0000024e 00000269 00000277
0000700 00000287 00000297 000002b6 000002ca
0000720 000002dc 000002f0 00000304 00000321
0000740 0000033e 0000035d 0000037a 00000395
0000760 000003ae 000003b6 000003be 000003c6
0001000 000003ce 000003dc 000003e9 000003f8
0001020

```

Здесь 132 значения, а это 66×2 . Может быть здесь 2 32-битных значения на каждый символ, а может быть здесь два массива? Посмотрим.

Значения, начинающиеся с `0x1000` могут быть адресами. В конце концов, этот .SYM-файл для DLL, а базовый адрес для DLL в win32 это `0x10000000`, и сам код обычно начинается по адресу `0x10001000`.

Когда открываем файл `orawtc8.dll` в [IDA](#), базовый адрес другой, но тем не менее, первая функция это:

```

.text:60351000 sub_60351000    proc near
.text:60351000
.text:60351000 arg_0          = dword ptr 8
.text:60351000 arg_4          = dword ptr 0Ch
.text:60351000 arg_8          = dword ptr 10h
.text:60351000
.text:60351000                push    ebp
.text:60351001                mov     ebp, esp
.text:60351003                mov     eax, dword_60353014
.text:60351008                cmp     eax, 0FFFFFFFFh
.text:6035100B                jnz     short loc_6035104F
.text:6035100D                mov     ecx, hModule
.text:60351013                xor     eax, eax
.text:60351015                cmp     ecx, 0FFFFFFFFh
.text:60351018                mov     dword_60353014, eax
.text:6035101D                jnz     short loc_60351031
.text:6035101F                call    sub_603510F0
.text:60351024                mov     ecx, eax
.text:60351026                mov     eax, dword_60353014
.text:6035102B                mov     hModule, ecx
.text:60351031
.text:60351031 loc_60351031:                ; CODE XREF: sub_60351000+1D
.text:60351031                test    ecx, ecx
.text:60351033                jbe     short loc_6035104F
.text:60351035                push    offset ProcName ; "ax_reg"
.text:6035103A                push    ecx              ; hModule
.text:6035103B                call    ds:GetProcAddress
...

```

Ух ты, «ax_reg» звучит знакомо. Действительно, это самая первая строка в блоке строк! Так что имя этой функции, похоже «ax_reg».

Вторая функция:

```

.text:60351080 sub_60351080    proc near
.text:60351080
.text:60351080 arg_0          = dword ptr 8
.text:60351080 arg_4          = dword ptr 0Ch
.text:60351080
.text:60351080                push    ebp
.text:60351081                mov     ebp, esp
.text:60351083                mov     eax, dword_60353018
.text:60351088                cmp     eax, 0FFFFFFFFh
.text:6035108B                jnz     short loc_603510CF
.text:6035108D                mov     ecx, hModule

```

```

.text:60351093      xor     eax, eax
.text:60351095      cmp     ecx, 0FFFFFFFFh
.text:60351098      mov     dword_60353018, eax
.text:6035109D      jnz     short loc_603510B1
.text:6035109F      call    sub_603510F0
.text:603510A4      mov     ecx, eax
.text:603510A6      mov     eax, dword_60353018
.text:603510AB      mov     hModule, ecx
.text:603510B1
.text:603510B1 loc_603510B1:                                ; CODE XREF: sub_60351080+1D
.text:603510B1      test    ecx, ecx
.text:603510B3      jbe     short loc_603510CF
.text:603510B5      push    offset aAx_unreg ; "ax_unreg"
.text:603510BA      push    ecx                ; hModule
.text:603510BB      call    ds:GetProcAddress
...

```

Строка «ax_unreg» также это вторая строка в строке блок! Адрес начала второй функции это **0x60351080**, а второе значение в бинарном блоке это **10001080**. Так что это адрес, но для DLL с базовым адресом по умолчанию.

Мы можем быстро проверить и убедиться, что первые 66 значений в массиве (т.е. первая половина) это просто адреса функций в DLL, включая некоторые метки, и т.д. Хорошо, что же тогда остальная часть массива? Остальные 66 значений, начинающиеся с **0x0000**? Они похожи в пределах **[0...0x3F8]**. И не похоже, что это битовые поля: ряд чисел возрастает. Последняя шестнадцатеричная цифра выглядит как случайная, так что, не похоже, что это адрес чего-либо (в противном случае, он бы делился, может быть, на 4 или 8 или 0x10).

Спросим себя: что еще разработчики Oracle RDBMS хранили бы здесь, в этом файле? Случайная догадка: это может быть адрес текстовой строки (название функции). Это можно легко проверить, и да, каждое число — это просто позиция первого символа в блоке строк.

Вот и всё! Всё закончено.

Напишем утилиту для конвертирования .SYM-файлов в IDA-скрипт, так что сможем загружать .idc-скрипт и он выставит имена функций:

```

#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include <assert.h>
#include <malloc.h>
#include <fcntl.h>
#include <string.h>

int main (int argc, char *argv[])
{
    uint32_t sig, cnt, offset;
    uint32_t *d1, *d2;
    int      h, i, remain, file_len;
    char      *d3;
    uint32_t array_size_in_bytes;

    assert (argv[1]); // file name
    assert (argv[2]); // additional offset (if needed)

    // additional offset
    assert (sscanf (argv[2], "%X", &offset)==1);

    // get file length
    assert ((h=open (argv[1], _O_RDONLY | _O_BINARY, 0))!=-1);
    assert ((file_len=lseek (h, 0, SEEK_END))!=-1);
    assert (lseek (h, 0, SEEK_SET)!=-1);

    // read signature
    assert (read (h, &sig, 4)==4);
    // read count
    assert (read (h, &cnt, 4)==4);

    assert (sig==0x4D59534F); // OSYM

    // skip timestamp (for 11g)

```



```

//_lseek (h, 4, 1);

array_size_in_bytes=cnt*sizeof(uint32_t);

// load symbol addresses array
d1=(uint32_t*)malloc (array_size_in_bytes);
assert (d1);
assert (read (h, d1, array_size_in_bytes)==array_size_in_bytes);

// load string offsets array
d2=(uint32_t*)malloc (array_size_in_bytes);
assert (d2);
assert (read (h, d2, array_size_in_bytes)==array_size_in_bytes);

// calculate strings block size
remain=file_len-(8+4)-(cnt*8);

// load strings block
assert (d3=(char*)malloc (remain));
assert (read (h, d3, remain)==remain);

printf ("#include <idc.idc>\n\n");
printf ("static main() {\n");

for (i=0; i<cnt; i++)
    printf ("\tMakeName(0x%08X, \"%s\");\n", offset + d1[i], &d3[d2[i]]);

printf ("}\n");

close (h);
free (d1); free (d2); free (d3);
};

```

Пример его работы:

```

#include <idc.idc>

static main() {
    MakeName(0x60351000, "_ax_reg");
    MakeName(0x60351080, "_ax_unreg");
    MakeName(0x603510F0, "_loaddll");
    MakeName(0x60351150, "_wtcsrin0");
    MakeName(0x60351160, "_wtcsrin");
    MakeName(0x603511C0, "_wtcsrfre");
    MakeName(0x603511D0, "_wtclkm");
    MakeName(0x60351370, "_wtcstu");
    ...
}

```

Файлы, использованные в этом примере, здесь: beginners.re.

Глава 89

Oracle RDBMS: .MSB-файлы

Работая над решением задачи, всегда полезно
знать ответ.

Законы Мерфи, правило точности

Это бинарный файл, содержащий сообщения об ошибках вместе с их номерами. Давайте попробуем понять его формат и найти способ распаковать его.

В Oracle RDBMS имеются файлы с сообщениями об ошибках в текстовом виде, так что мы можем сравнивать файлы: текстовый и запакованный бинарный ¹.

Это начало файла ORAUS.MSG без ненужных комментариев:

Листинг 89.1: Начало файла ORAUS.MSG без комментариев

```
00000, 00000, "normal, successful completion"
00001, 00000, "unique constraint (%s.%s) violated"
00017, 00000, "session requested to set trace event"
00018, 00000, "maximum number of sessions exceeded"
00019, 00000, "maximum number of session licenses exceeded"
00020, 00000, "maximum number of processes (%s) exceeded"
00021, 00000, "session attached to some other process; cannot switch session"
00022, 00000, "invalid session ID; access denied"
00023, 00000, "session references process private memory; cannot detach session"
00024, 00000, "logins from more than one process not allowed in single-process mode"
00025, 00000, "failed to allocate %s"
00026, 00000, "missing or invalid session ID"
00027, 00000, "cannot kill current session"
00028, 00000, "your session has been killed"
00029, 00000, "session is not a user session"
00030, 00000, "User session ID does not exist."
00031, 00000, "session marked for kill"
...
```

Первое число — это код ошибки. Второе это, вероятно, могут быть дополнительные флаги.

¹Текстовые файлы с открытым кодом в Oracle RDBMS имеются не для каждого .MSB-файла, вот почему мы будем работать над его форматом

Давайте откроем бинарный файл ORAUS.MSB и найдем эти текстовые строки. И вот они:

```
Hiew: oraus.msb
C:\tmp\oraus.msb
000013B9 Hiew 8.02 (c)SEN
normal, successful completionunique constraint (%s.%s) viol
atedsession requested to set trace eventmaximum number of sessions exceededmaximum number of session
n licenses exceededmaximum number of processes (%s) exceededsession attached to some other process;
cannot switch sessioninvalid session ID; access deniedsession references process private memory; c
annot detach sessionlogins from more than one process not allowed in single-process mode P
e B Э | r i ! . " Z# }$ || юfailed to allocate %smissing o
r invalid session IDcannot kill current sessionyour session has been killedsession is not a user se
ssionUser session ID does not exist.session marked for killinvalid session migration passwordcurren
t session has empty migration passwordcannot %s in current PL/SQL sessionLICENSE_MAX_USERS cannot b
e less than current number of usersmaximum number of recursive SQL levels (%s) exceeded
% D & A ' ( ) * 3+ J2 a3 b4 L .cannot switch to a sessio
n belonging to a different server groupCannot create session: server group belongs to another user
rror during periodic actionactive time limit exceeded - call abortedactive time limit exceeded - se
ssion terminatedUnknown Service name %sremote operation failedoperating system error occurred while
obtaining an enqueuetimeout occurred while waiting for a resourcemaximum number of enqueue resourc
es (%s) exceeded 5 > 6 a 7 r 8 | 9 : 9; q< 0= L Emaximum number
of enqueues exceededresource busy and acquire with NOWAIT specified or timeout expiredmaximum numb
er of DML locks exceededDDL lock on object '%s.%s' is already held in an incompatible modemaximum n
umber of temporary table locks exceededDB_BLOCK_SIZE must be %s to mount this database (not %s)maxi
mum number of DB_FILES exceededdeadlock detected while waiting for resourceanother instance has a d
ifferent DML_LOCKS setting > D ? z @ 6 A | C . D 0E 1F 00G |H
D DML full-table lock cannot be acquired; DML_LOCKS is 0maximum number of log files exceeded
%object is too large to allocate on this O/S (%s,%s,%s)initialization of FIXED_DATE failedinvalid
value %s for parameter %s; must be at least %sinvalid value %s for parameter %s, must be between %
s and %scannot acquire lock -- table locks disabled for %scommand %s is not validprocess number mus
t be between 1 and %sprocess "%s" is not active I P J ~ K Ы L Т М Ъ N ч O
P Q R hS иT | Ecommand %s takes between %s and %s argument(s)no process has
1 2 3 4ReLoad 5 6String 7Direct 8Table 9 10Leave 11 12
```

Рис. 89.1: Hiew: первый блок

Мы видим текстовые строки (включая те, с которых начинается файл ORAUS.MSG) перемежаемые с какими-то бинарными значениями. Мы можем довольно быстро обнаружить что главная часть бинарного файла поделена на блоки размером 0x200 (512) байт.

Посмотрим содержимое первого блока:

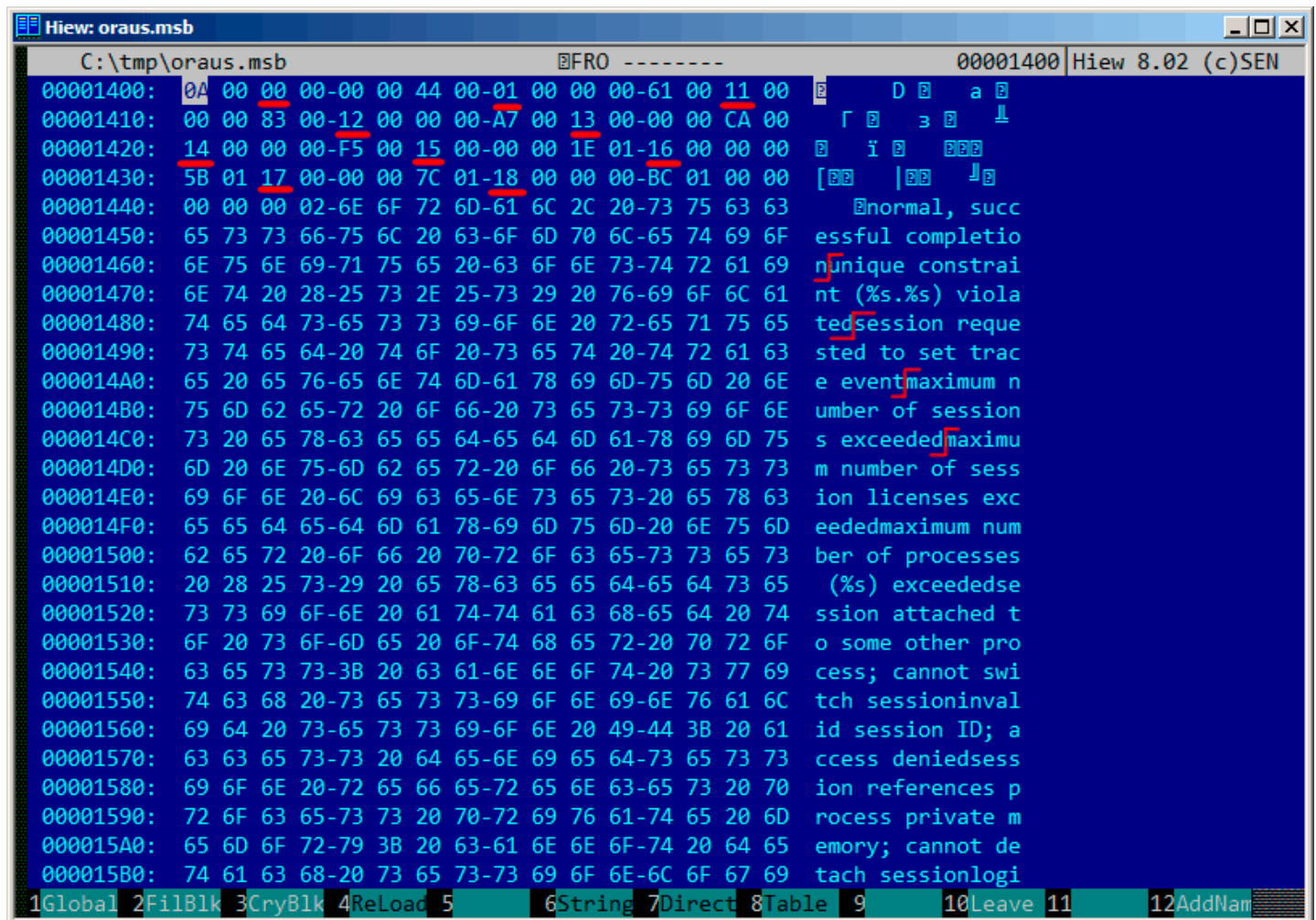


Рис. 89.2: Hiew: первый блок

Мы видим тексты первых сообщений об ошибках. Что мы видим еще, так это то, что здесь нет нулевых байтов между сообщениями. Это значит, что это не оканчивающиеся нулем Си-строки. Как следствие, длина каждого сообщения об ошибке должна быть как-то закодирована. Попробуем также найти номера ошибок. Файл ORAUS.MSG начинается с таких: 0, 1, 17 (0x11), 18 (0x12), 19 (0x13), 20 (0x14), 21 (0x15), 22 (0x16), 23 (0x17), 24 (0x18)... Найдем эти числа в начале блока и отметим их красными линиями. Период между кодами ошибок 6 байт. Это значит, здесь, наверное, 6 байт информации выделено для каждого сообщения об ошибке.

Первое 16-битное значение (здесь 0xA или 10) означает количество сообщений в блоке: это можно проверить глядя на другие блоки. Действительно: сообщения об ошибках имеют произвольный размер. Некоторые длиннее, некоторые короче. Но длина блока всегда фиксирована, следовательно, никогда не знаешь, сколько сообщений можно запаковать в каждый блок.

Как мы уже отметили, так как это не оканчивающиеся нулем Си-строки, длина строки должна быть закодирована где-то. Длина первой строки «normal, successful completion» это 29 (0x1D) байт. Длина второй строки «unique constraint (%s.%s) violated» это 34 (0x22) байт. Мы не можем отыскать этих значений (0x1D или/и 0x22) в блоке.

А вот еще кое-что. Oracle RDBMS должен как-то определять позицию строки, которую он должен загрузить, верно? Первая строка «normal, successful completion» начинается с позиции 0x1444 (если считать с начала бинарного файла) или с 0x44 (от начала блока). Вторая строка «unique constraint (%s.%s) violated» начинается с позиции 0x1461 (от начала файла) или с 0x61 (считая с начала блока). Эти числа (0x44 и 0x61) нам знакомы! Мы их можем легко отыскать в начале блока.

Так что, каждый 6-байтный блок это:

- 16-битный номер ошибки;
- 16-битный ноль (может быть, дополнительные флаги);
- 16-битная начальная позиция текстовой строки внутри текущего блока.

Мы можем быстро проверить остальные значения чтобы удостовериться в своей правоте. И здесь еще последний «пустой» 6-байтный блок с нулевым номером ошибки и начальной позицией за последним символом последнего сообщения об ошибке. Может быть именно так и определяется длина сообщения? Мы просто перебираем 6-байтные блоки в поисках нужного номера ошибки, затем мы узнаем позицию текстовой строки, затем мы узнаем позицию следующей текстовой строки глядя на следующий 6-байтный блок! Так мы определяем границы строки! Этот метод позволяет сэкономить место в файле не записывая длину строки! Нельзя сказать, что экономия памяти большая, но это интересный трюк..

Вернемся к заголовку .MSB-файла:

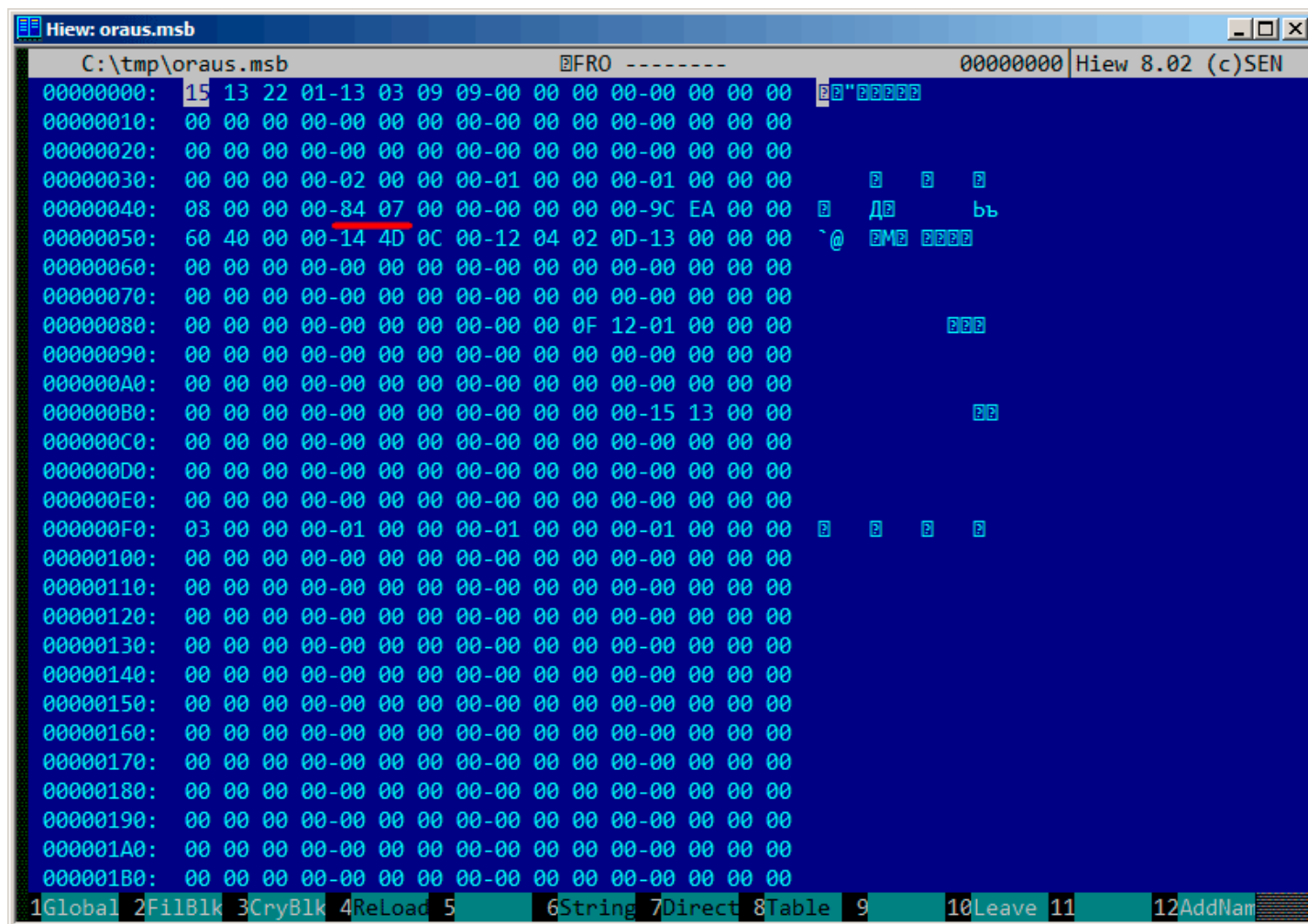


Рис. 89.3: Hiew: заголовок файла

Теперь мы можем быстро найти количество блоков (отмечено красным). Проверяем другие .MSB-файлы и оказывается что это справедливо для всех. Здесь есть много других значений, но мы не будем разбираться с ними, так как наша задача (утилита для распаковки) уже решена. А если бы мы писали упаковщик .MSB-файлов, тогда нам наверное пришлось бы понять, зачем нужны остальные.

89.1. Вывод

Тут еще есть таблица после заголовка, вероятно, содержащая 16-битные значения:

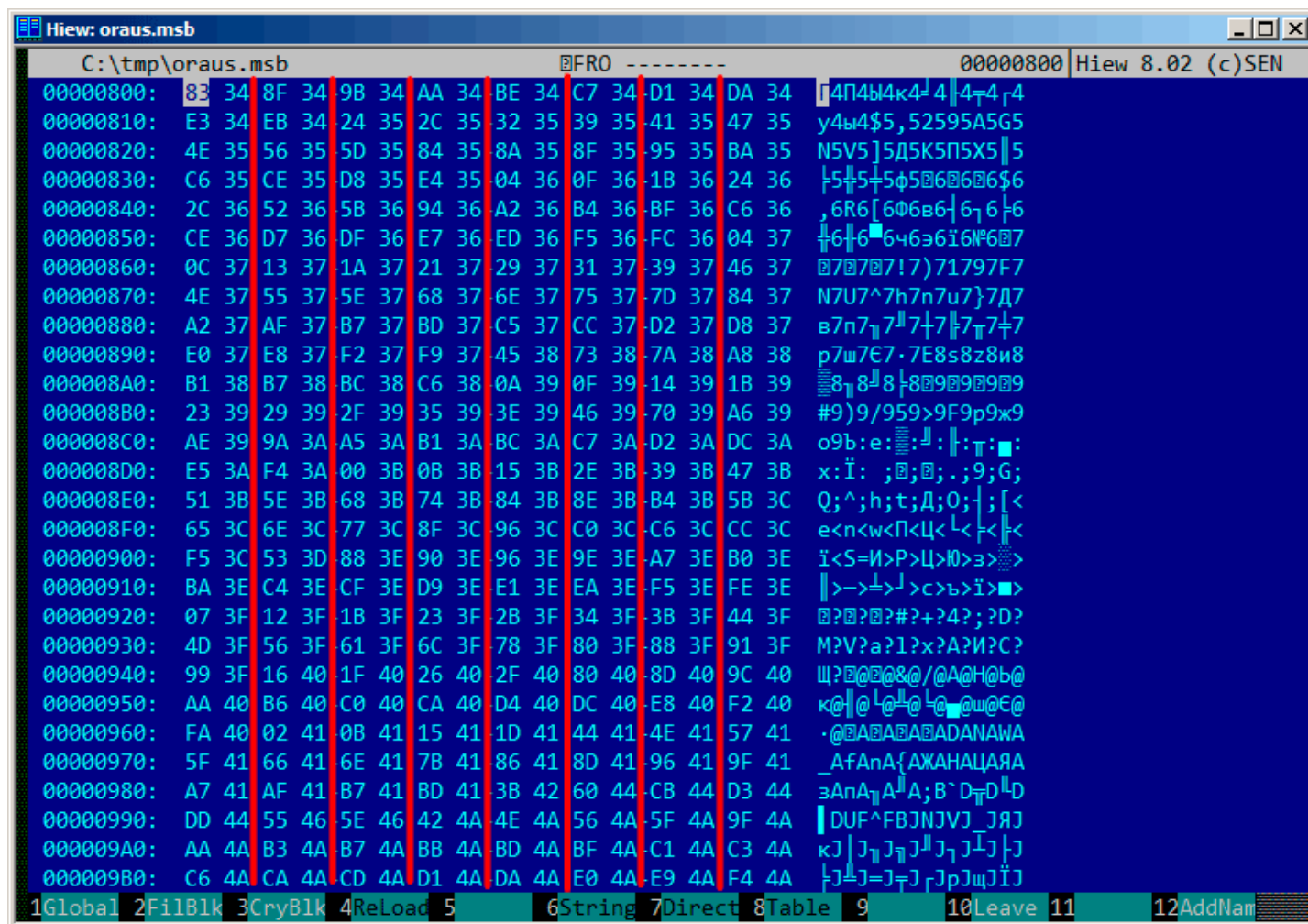


Рис. 89.4: Hiew: таблица last_errnos

Их длина может быть определена визуально (здесь нарисованы красные линии). Когда мы сдублировали эти значения, мы обнаружили, что каждое 16-битное число — это последний код ошибки для каждого блока.

Так вот как Oracle RDBMS быстро находит сообщение об ошибке:

- загружает таблицу, которую мы назовем last_errnos (содержащую последний номер ошибки для каждого блока);
- находит блок содержащий код ошибки, полагая что все коды ошибок увеличиваются и внутри каждого блока и также в файле;
- загружает соответствующий блок;
- перебирает 6-байтные структуры, пока не найдется соответствующий номер ошибки;
- находит позицию первого символа из текущего 6-байтного блока;
- находит позицию последнего символа из следующего 6-байтного блока;
- загружает все символы сообщения в этих пределах.

Это программа на Си которую мы написали для распаковки .MSB-файлов : beginners.re.

И еще два файла которые были использованы в этом примере (Oracle RDBMS 11.1.0.6): beginners.re, beginners.re.

89.1. Вывод

Этот метод, наверное, слишком олд-скульный для современных компьютеров. Возможно, формат этого файла был разработан в середине 1980-х кем-то, кто программировал для мэйнфреймов, учитывая экономию памяти и места на дисках.

89.1. *ВЫВОД*

Тем не менее, это интересная (хотя и простая) задача на разбор проприетарного формата файла без заглядывания в код Oracle RDBMS.

Часть X

Прочее

Глава 90

npad

Это макрос в ассемблере, для выравнивания некоторой метки по некоторой границе.

Это нужно для тех *нагруженных* меток, куда чаще всего передается управление, например, начало тела цикла. Для того чтобы процессор мог эффективнее вытягивать данные или код из памяти, через шину с памятью, кэширование, и т.д.

Взято из `listing.inc` (MSVC):

Это, кстати, любопытный пример различных вариантов `NOP`-ов. Все эти инструкции не дают никакого эффекта, но отличаются разной длиной.

Цель в том, чтобы была только одна инструкция, а не набор NOP-ов, считается что так лучше для производительности CPU.

```
;; LISTING.INC
;;
;; This file contains assembler macros and is included by the files created
;; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

;; non destructive nops
npad macro size
if size eq 1
    nop
else
if size eq 2
    mov edi, edi
else
if size eq 3
    ; lea ecx, [ecx+00]
    DB 8DH, 49H, 00H
else
if size eq 4
    ; lea esp, [esp+00]
    DB 8DH, 64H, 24H, 00H
else
if size eq 5
    add eax, DWORD PTR 0
else
if size eq 6
    ; lea ebx, [ebx+00000000]
    DB 8DH, 9BH, 00H, 00H, 00H, 00H
else
if size eq 7
    ; lea esp, [esp+00000000]
    DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
else
if size eq 8
    ; jmp .+8; .npad 6
    DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
else
if size eq 9
```


Глава 91

Модификация исполняемых файлов

91.1. Текстовые строки

Сишные строки проще всего модифицировать (если они не зашифрованы) в любом шестнадцатеричном редакторе. Эта техника доступна даже для тех, кто вовсе не разбирается в машинном коде и форматах исполняемых файлов. Новая строка не должна быть длиннее старой, потому что имеется риск затереть какую-то другую переменную или код. Используя этот метод, очень много ПО было *локализовано* во времена MS-DOS, как минимум, в странах бывшего СССР, в 80-х и 90-х. Отсюда наличие очень странных аббревиатур и сокращений в *локализованном* ПО: там просто не было места для более длинных строк.

В строках в Delphi, длина строки также должна быть поправлена, если нужно.

91.2. x86-код

Часто необходимые задачи:

- Часто нужно просто запретить исполнение какой-либо инструкции. И чаще всего, это можно сделать, заполняя её байтом `0x90` (`NOP`).
- Условные переходы, имеющие опкод вроде `74 xx` (`JZ`), так же могут быть заполнены двумя `NOP`-ами. Также возможно запретить исполнение условного перехода записав 0 во второй байт (*jump offset*).
- Еще одна часто необходимая задача это сделать условный переход всегда срабатывающим: это возможно при помощи записи `0xEB` вместо опкода, это значит `JMP`.
- Исполнение функции может быть запрещено, если записать `RETN` (`0xC3`) в её начале. Это справедливо для всех функций кроме `stdcall` (65.2 (стр. 672)). При модификации функций `stdcall`, нужно в начале определить количество аргументов (например, отыскав `RETN` в этой функции), и использовать `RETN` с 16-битным аргументом (`0xC2`).
- Иногда, запрещенная функция должна возвращать 0 или 1. Это можно сделать при помощи `MOV EAX, 0` или `MOV EAX, 1`, но это слишком многословно. Способ получше это `XOR EAX, EAX` (2 байта `0x31 0xC0`) или `XOR EAX, EAX / INC EAX` (3 байта `0x31 0xC0 0x40`).

ПО может быть защищено от модификаций. Эта защита чаще всего реализуется путем чтения кода и вычисления контрольной суммы. Следовательно, код должен быть прочитан перед тем как защита сработает. Это можно определить установив точку останова на чтение памяти.

В `tracer` имеется опция `BPM` для этого.

Релоки в исполняемых PE-файлах (69.2.6 (стр. 699)) не должны быть тронуты, потому что загрузчик Windows перезапишет ваш новый код. (Они выделяются серым в Niew, например: илл.8.12). В качестве последней меры, можно записать `JMP` для обхода релока, либо же придется модифицировать таблицу релоков.

Глава 92

Compiler intrinsic

Специфичная для компилятора функция не являющаяся обычной библиотечной функцией. Компилятор вместо её вызова генерирует определенный машинный код. Нередко, это псевдофункции для определенной инструкции [CPU](#).

Например, в языках Си/Си++ нет операции циклического сдвига, а во многих [CPU](#) она есть. Чтобы программисту были доступны эти инструкции, в MSVC есть псевдофункции `_rotl()` и `_rotr()`¹, которые компилятором напрямую транслируются в x86-инструкции `ROL` / `ROR`.

Еще один пример это функции позволяющие генерировать SSE-инструкции прямо в коде.

Полный список intrinsics от MSVC: [MSDN](#).

¹[MSDN](#)

Глава 93

Аномалии компиляторов

Intel C++ 10.1 которым скомпилирован Oracle RDBMS 11.2 Linux86, может сгенерировать два **JZ** идущих подряд, причем на второй **JZ** нет ссылки ниоткуда. Второй **JZ** таким образом, не имеет никакого смысла.

Листинг 93.1: kdli.o из libserver11.a

```
.text:08114CF1          loc_8114CF1: ; CODE XREF: __PGOSF539_kdlimemSer+89A
.text:08114CF1          ; __PGOSF539_kdlimemSer+3994
.text:08114CF1 8B 45 08          mov     eax, [ebp+arg_0]
.text:08114CF4 0F B6 50 14      movzx   edx, byte ptr [eax+14h]
.text:08114CF8 F6 C2 01          test    dl, 1
.text:08114CFB 0F 85 17 08 00 00 jnz     loc_8115518
.text:08114D01 85 C9            test    ecx, ecx
.text:08114D03 0F 84 8A 00 00 00 jz      loc_8114D93
.text:08114D09 0F 84 09 08 00 00 jz      loc_8115518
.text:08114D0F 8B 53 08          mov     edx, [ebx+8]
.text:08114D12 89 55 FC          mov     [ebp+var_4], edx
.text:08114D15 31 C0            xor     eax, eax
.text:08114D17 89 45 F4          mov     [ebp+var_C], eax
.text:08114D1A 50              push    eax
.text:08114D1B 52              push    edx
.text:08114D1C E8 03 54 00 00    call    len2nbytes
.text:08114D21 83 C4 08          add     esp, 8
```

Листинг 93.2: оттуда же

```
.text:0811A2A5          loc_811A2A5: ; CODE XREF: kdliSerLengths+11C
.text:0811A2A5          ; kdliSerLengths+1C1
.text:0811A2A5 8B 7D 08          mov     edi, [ebp+arg_0]
.text:0811A2A8 8B 7F 10          mov     edi, [edi+10h]
.text:0811A2AB 0F B6 57 14      movzx   edx, byte ptr [edi+14h]
.text:0811A2AF F6 C2 01          test    dl, 1
.text:0811A2B2 75 3E            jnz     short loc_811A2F2
.text:0811A2B4 83 E0 01          and     eax, 1
.text:0811A2B7 74 1F            jz      short loc_811A2D8
.text:0811A2B9 74 37            jz      short loc_811A2F2
.text:0811A2BB 6A 00            push    0
.text:0811A2BD FF 71 08          push    dword ptr [ecx+8]
.text:0811A2C0 E8 5F FE FF FF    call    len2nbytes
```

Возможно, это ошибка его кодегенератора, не выявленная тестами (ведь результирующий код и так работает нормально).

Еще подобные аномалии компиляторов в этой книге : [20.2.4](#) (стр. 310), [40.3](#) (стр. 477), [48.7](#) (стр. 521), [19.7](#) (стр. 297), [13.4.1](#) (стр. 143), [20.5.2](#) (стр. 327).

В этой книге здесь приводятся подобные случаи для того, чтобы легче было понимать, что подобные ошибки компиляторов все же имеют место быть, и не следует ломать голову над тем, почему он сгенерировал такой странный код.

Глава 94

OpenMP

OpenMP это один из простейших способов распараллелить работу простого алгоритма .

В качестве примера, попытаемся написать программу для вычисления криптографического *nonce* . В моем простейшем примере, *nonce* это число, добавляемое к нешифрованному тексту, чтобы получить хэш с какой-то особенностью . Например, на одной из стадии, протокол Bitcoin требует найти такую *nonce*, чтобы в результате хэширования подряд шли определенное количество нулей. Это еще называется «proof of work» ¹ (т.е. система доказывает, что она произвела какие-то очень ресурсоёмкие вычисления и затратила время на это).

Мой пример не связан с Bitcoin, он будет пытаться добавлять числа к строке «hello, world!_» чтобы найти такое число, при котором строка вида «hello, world!_<number>» после хеширования алгоритмом SHA512 будет содержать как минимум 3 нулевых байта в начале.

Ограничимся перебором всех чисел в интервале 0..INT32_MAX-1 (т.е., 0x7FFFFFFE или 2147483646).

Алгоритм очень простой:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "sha512.h"

int found=0;
int32_t checked=0;

int32_t* __min;
int32_t* __max;

time_t start;

#ifdef __GNUC__
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
{
    uint8_t buf[32];
    struct sha512_ctx ctx;
    uint8_t res[64];

    // update statistics
    int t=omp_get_thread_num();

    if (__min[t]==-1)
        __min[t]=nonce;
    if (__max[t]==-1)
        __max[t]=nonce;

    __min[t]=min(__min[t], nonce);
    __max[t]=max(__max[t], nonce);
}
```

¹[wikipedia](#)

```

// idle if valid nonce found
if (found)
    return;

memset (buf, 0, sizeof(buf));
sprintf (buf, "hello, world!_%d", nonce);

sha512_init_ctx (&ctx);
sha512_process_bytes (buf, strlen(buf), &ctx);
sha512_finish_ctx (&ctx, &res);
if (res[0]==0 && res[1]==0 && res[2]==0)
{
    printf ("found (thread %d): [%s]. seconds spent=%d\n", t, buf, time(NULL)-start);
    found=1;
};
#pragma omp atomic
checked++;

#pragma omp critical
if ((checked % 100000)==0)
    printf ("checked=%d\n", checked);
};

int main()
{
    int32_t i;
    int threads=omp_get_max_threads();
    printf ("threads=%d\n", threads);

    __min=(int32_t*)malloc(threads*sizeof(int32_t));
    __max=(int32_t*)malloc(threads*sizeof(int32_t));
    for (i=0; i<threads; i++)
        __min[i]=__max[i]=-1;

    start=time(NULL);

    #pragma omp parallel for
    for (i=0; i<INT32_MAX; i++)
        check_nonce (i);

    for (i=0; i<threads; i++)
        printf ("__min[%d]=0x%08x __max[%d]=0x%08x\n", i, __min[i], i, __max[i]);

    free(__min); free(__max);
};

```

`check_nonce()` просто добавляет число к строке, хеширует алгоритмом SHA512 и проверяет 3 нулевых байта в начале.

Очень важная часть кода — это:

```

#pragma omp parallel for
for (i=0; i<INT32_MAX; i++)
    check_nonce (i);

```

Да, вот настолько просто, без `#pragma` мы просто вызываем `check_nonce()` для каждого числа от 0 до `INT32_MAX` (`0x7fffffff` или `2147483647`). С `#pragma`, компилятор добавляет специальный код, который разрежет интервал цикла на меньшие интервалы, чтобы запустить их на доступных ядрах CPU ².

Пример может быть скомпилирован ³ в MSVC 2012:

```
cl openmp_example.c sha512.obj /openmp /O1 /Zi /Faopenmp_example.asm
```

Или в GCC:

```
gcc -fopenmp 2.c sha512.c -S -masm=intel
```

²Н.В.: Это намеренно упрощенный пример, но на практике, применение OpenMP может быть труднее и сложнее

³файлы sha512.c|h и u64.h можно взять из библиотеки OpenSSL : <http://go.yurichev.com/17324>

94.1. MSVC

Вот как MSVC 2012 генерирует главный цикл:

Листинг 94.1: MSVC 2012

```

push    OFFSET _main$omp$1
push    0
push    1
call    __vcomp_fork
add     esp, 16                                ; 00000010H

```

Функции с префиксом `vcomp` связаны с OpenMP и находятся в файле `vcomp*.dll`. Так что тут запускается группа тредов.

Посмотрим на `_main$omp$1`:

Листинг 94.2: MSVC 2012

```

$T1 = -8                                ; size = 4
$T2 = -4                                ; size = 4
_main$omp$1 PROC                        ; COMDAT
    push    ebp
    mov     ebp, esp
    push    ecx
    push    ecx
    push    esi
    lea     eax, DWORD PTR $T2[ebp]
    push    eax
    lea     eax, DWORD PTR $T1[ebp]
    push    eax
    push    1
    push    1
    push    2147483646                    ; 7fffffffH
    push    0
    call    __vcomp_for_static_simple_init
    mov     esi, DWORD PTR $T1[ebp]
    add     esp, 24                        ; 00000018H
    jmp     SHORT $LN6@main$omp$1
$LL2@main$omp$1:
    push    esi
    call    _check_nonce
    pop     ecx
    inc     esi
$LN6@main$omp$1:
    cmp     esi, DWORD PTR $T2[ebp]
    jle     SHORT $LL2@main$omp$1
    call    __vcomp_for_static_end
    pop     esi
    leave
    ret     0
_main$omp$1 ENDP

```

Эта функция будет запущена n раз параллельно, где n это число ядер CPU. `vcomp_for_static_simple_init()` вычисляет интервал для конструкта `for()` для текущего треда, в зависимости от текущего номера треда. Значения начала и конца цикла записаны в локальных переменных `$T1` и `$T2`. Вы также можете заметить `7fffffffH` (или `2147483646`) как аргумент для функции `vcomp_for_static_simple_init()` это количество итераций всего цикла, оно будет поделено на равные части.

Потом мы видим новый цикл с вызовом функции `check_nonce()` делающей всю работу.

Добавил также немного кода в начале функции `check_nonce()` для сбора статистики, с какими аргументами эта функция вызывалась.

Вот что мы видим если запустим:

```

threads=4
...
checked=2800000
checked=3000000
checked=3200000

```

94.1. MSVC

```
checked=3300000
found (thread 3): [hello, world!_1611446522]. seconds spent=3
__min[0]=0x00000000 __max[0]=0x1fffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7fffffff
```

Да, результат правильный, первые 3 байта это нули:

```
C:\...\sha512sum test
000000f4a8fac5a4ed38794da4c1e39f54279ad5d9bb3c5465cdf57adaf60403
df6e3fe6019f5764fc9975e505a7395fed780fee50eb38dd4c0279cb114672e2 *test
```

Оно требует $\approx 2..3$ секунды на 4-х ядерном Intel Xeon E3-1220 3.10 GHz. В task manager мы видим 5 тредов: один главный тред + 4 запущенных. Никаких оптимизаций не было сделано, чтобы оставить этот пример в как можно более простом виде. Но, наверное, этот алгоритм может работать быстрее. У моего CPU 4 ядра, вот почему OpenMP запустил именно 4 тредов.

Глядя на таблицу статистики, можно легко увидеть, что цикл был разделен очень точно на 4 равных части. Ну хорошо, почти равных, если не учитывать последний бит.

Имеются также прагмы и для [атомарных операций](#).

Посмотрим, как вот этот код будет скомпилирован:

```
#pragma omp atomic
checked++;

#pragma omp critical
if ((checked % 100000)==0)
    printf ("checked=%d\n", checked);
```

Листинг 94.3: MSVC 2012

```
push    edi
push    OFFSET _checked
call    __vcomp_atomic_add_i4
; Line 55
push    OFFSET _$vcomp$critsect$
call    __vcomp_enter_critsect
add     esp, 12                ; 0000000cH
; Line 56
mov     ecx, DWORD PTR _checked
mov     eax, ecx
cdq
mov     esi, 100000            ; 000186a0H
idiv    esi
test    edx, edx
jne     SHORT $LN1@check_nonc
; Line 57
push    ecx
push    OFFSET ??_C@_0M@NPNHLI00@checked?$DN?$CFd?6?$AA@
call    _printf
pop     ecx
pop     ecx
$LN1@check_nonc:
push    DWORD PTR _$vcomp$critsect$
call    __vcomp_leave_critsect
pop     ecx
```

Как выясняется, функция `vcomp_atomic_add_i4()` в `vcomp*.dll` это просто крохотная функция имеющая инструкцию `LOCK XADD` ⁴.

`vcomp_enter_critsect()` в конце концов вызывает функцию win32 API `EnterCriticalSection()` ⁵.

⁴О префиксе LOCK читайте больше : [A.6.1](#) (стр. 915)

⁵О критических секциях читайте больше тут: [69.4](#) (стр. 725)

94.2. GCC

GCC 4.8.1 выдает программу показывающую точно такую же таблицу со статистикой , так что, реализация GCC делит цикл на части точно так же .

Листинг 94.4: GCC 4.8.1

```

mov     edi, OFFSET FLAT:main._omp_fn.0
call    GOMP_parallel_start
mov     edi, 0
call    main._omp_fn.0
call    GOMP_parallel_end

```

В отличие от реализации MSVC, то, что делает код GCC, это запускает 3 треда, но также запускает четвертый прямо в текущем треде. Так что здесь всего 4 треда а не 5 как в случае с MSVC.

Вот функция `main._omp_fn.0` :

Листинг 94.5: GCC 4.8.1

```

main._omp_fn.0:
    push    rbp
    mov     rbp, rsp
    push    rbx
    sub     rsp, 40
    mov     QWORD PTR [rbp-40], rdi
    call    omp_get_num_threads
    mov     ebx, eax
    call    omp_get_thread_num
    mov     esi, eax
    mov     eax, 2147483647 ; 0x7FFFFFFF
    cdq
    idiv    ebx
    mov     ecx, eax
    mov     eax, 2147483647 ; 0x7FFFFFFF
    cdq
    idiv    ebx
    mov     eax, edx
    cmp     esi, eax
    jl      .L15
.L18:
    imul    esi, ecx
    mov     edx, esi
    add     eax, edx
    lea     ebx, [rax+rcx]
    cmp     eax, ebx
    jge     .L14
    mov     DWORD PTR [rbp-20], eax
.L17:
    mov     eax, DWORD PTR [rbp-20]
    mov     edi, eax
    call    check_nonce
    add     DWORD PTR [rbp-20], 1
    cmp     DWORD PTR [rbp-20], ebx
    jl      .L17
    jmp     .L14
.L15:
    mov     eax, 0
    add     ecx, 1
    jmp     .L18
.L14:
    add     rsp, 40
    pop     rbx
    pop     rbp
    ret

```

Здесь мы видим это деление явно: вызывая `omp_get_num_threads()` и `omp_get_thread_num()` мы получаем количество запущенных тредов, а также номер текущего треда , и затем определяем интервал цикла. И затем запускаем `check_nonce()` .

94.2. GCC

GCC также вставляет инструкцию **LOCK ADD** прямо в том месте кода, где MSVC сгенерировал вызов отдельной функции в DLL :

Листинг 94.6: GCC 4.8.1

```
lock add     DWORD PTR checked[rip], 1
call        GOMP_critical_start
mov         ecx, DWORD PTR checked[rip]
mov         edx, 351843721
mov         eax, ecx
imul        edx
sar         edx, 13
mov         eax, ecx
sar         eax, 31
sub         edx, eax
mov         eax, edx
imul        eax, eax, 100000
sub         ecx, eax
mov         eax, ecx
test        eax, eax
jne         .L7
mov         eax, DWORD PTR checked[rip]
mov         esi, eax
mov         edi, OFFSET FLAT:.LC2 ; "checked=%d\n"
mov         eax, 0
call        printf
.L7:
call        GOMP_critical_end
```

Функции с префиксом GOMP это часть библиотеки GNU OpenMP. В отличие от vcomp*.dll, её исходный код свободно доступен : [GitHub](#).

Глава 95

Itanium

Еще одна очень интересная архитектура (хотя и почти провальная) это Intel Itanium (IA64) . Другие OOE-процессоры сами решают, как переставлять инструкции и исполнять их параллельно, EPIC¹ это была попытка сдвинуть эти решения на компилятор: дать ему возможность самому группировать инструкции во время компиляции.

Это вылилось в очень сложные компиляторы

Вот один пример IA64-кода: простой криптоалгоритм из ядра Linux :

Листинг 95.1: Linux kernel 3.2.0.4

```
#define TEA_ROUNDS          32
#define TEA_DELTA           0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
    u32 y, z, n, sum = 0;
    u32 k0, k1, k2, k3;
    struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
    const __le32 *in = (const __le32 *)src;
    __le32 *out = (__le32 *)dst;

    y = le32_to_cpu(in[0]);
    z = le32_to_cpu(in[1]);

    k0 = ctx->KEY[0];
    k1 = ctx->KEY[1];
    k2 = ctx->KEY[2];
    k3 = ctx->KEY[3];

    n = TEA_ROUNDS;

    while (n-- > 0) {
        sum += TEA_DELTA;
        y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1);
        z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3);
    }

    out[0] = cpu_to_le32(y);
    out[1] = cpu_to_le32(z);
}
```

И вот как он был скомпилирован:

Листинг 95.2: Linux Kernel 3.2.0.4 для Itanium 2 (McKinley)

0090	tea_encrypt:	
0090 08 80 80 41 00 21	adds r16 = 96, r32	// ptr to ctx->KEY[2]
0096 80 C0 82 00 42 00	adds r8 = 88, r32	// ptr to ctx->KEY[0]
009C 00 00 04 00	nop.i 0	
00A0 09 18 70 41 00 21	adds r3 = 92, r32	// ptr to ctx->KEY[1]
00A6 F0 20 88 20 28 00	ld4 r15 = [r34], 4	// load z
00AC 44 06 01 84	adds r32 = 100, r32;;	// ptr to ctx->KEY[3]

¹Explicitly parallel instruction computing

00B0 08 98 00 20 10 10	ld4 r19 = [r16]	// r19=k2
00B6 00 01 00 00 42 40	mov r16 = r0	// r0 always contain zero
00BC 00 08 CA 00	mov.i r2 = ar.lc	// save lc register
00C0 05 70 00 44 10 10 9E FF FF FF 7F 20	ld4 r14 = [r34]	// load y
00CC 92 F3 CE 6B	movl r17 = 0xFFFFFFFF9E3779B9;;	// TEA_DELTA
00D0 08 00 00 00 01 00	nop.m 0	
00D6 50 01 20 20 20 00	ld4 r21 = [r8]	// r21=k0
00DC F0 09 2A 00	mov.i ar.lc = 31	// TEA_ROUNDS is 32
00E0 0A A0 00 06 10 10	ld4 r20 = [r3];;	// r20=k1
00E6 20 01 80 20 20 00	ld4 r18 = [r32]	// r18=k3
00EC 00 00 04 00	nop.i 0	
00F0		
00F0	loc_F0:	
00F0 09 80 40 22 00 20	add r16 = r16, r17	// r16=sum, r17=TEA_DELTA
00F6 D0 71 54 26 40 80	shladd r29 = r14, 4, r21	// r14=y, r21=k0
00FC A3 70 68 52	extr.u r28 = r14, 5, 27;;	
0100 03 F0 40 1C 00 20	add r30 = r16, r14	
0106 B0 E1 50 00 40 40	add r27 = r28, r20;;	// r20=k1
010C D3 F1 3C 80	xor r26 = r29, r30;;	
0110 0B C8 6C 34 0F 20	xor r25 = r27, r26;;	
0116 F0 78 64 00 40 00	add r15 = r15, r25	// r15=z
011C 00 00 04 00	nop.i 0;;	
0120 00 00 00 00 01 00	nop.m 0	
0126 80 51 3C 34 29 60	extr.u r24 = r15, 5, 27	
012C F1 98 4C 80	shladd r11 = r15, 4, r19	// r19=k2
0130 0B B8 3C 20 00 20	add r23 = r15, r16;;	
0136 A0 C0 48 00 40 00	add r10 = r24, r18	// r18=k3
013C 00 00 04 00	nop.i 0;;	
0140 0B 48 28 16 0F 20	xor r9 = r10, r11;;	
0146 60 B9 24 1E 40 00	xor r22 = r23, r9	
014C 00 00 04 00	nop.i 0;;	
0150 11 00 00 00 01 00	nop.m 0	
0156 E0 70 58 00 40 A0	add r14 = r14, r22	
015C A0 FF FF 48	br.cloop.sptk.few loc_F0;;	
0160 09 20 3C 42 90 15	st4 [r33] = r15, 4	// store z
0166 00 00 00 02 00 00	nop.m 0	
016C 20 08 AA 00	mov.i ar.lc = r2;;	// restore lc register
0170 11 00 38 42 90 11	st4 [r33] = r14	// store y
0176 00 00 00 02 00 80	nop.i 0	
017C 08 00 84 00	br.ret.sptk.many b0;;	

Прежде всего, все инструкции [IA64](#) сгруппированы в пакки (bundle) из трех инструкций . Каждая пачка имеет размер 16 байт (128 бит) и состоит из template-кода (5 бит) и трех инструкций (41 бит на каждую). [IDA](#) показывает пакки как 6+6+4 байт — вы можете легко заметить эту повторяющуюся структуру .

Все 3 инструкции каждой пачки обычно исполняются одновременно, если только у какой-то инструкции нет «стоп-бита» .

Вероятно, инженеры Intel и HP собрали статистику наиболее встречающихся шаблонных сочетаний инструкций и решили ввести типы пачек ([AKA](#) «templates»): код пачки определяет типы инструкций в пачке . Их всего 12. Например, нулевой тип это [MII](#) , что означает: первая инструкция это Memory (загрузка или запись в память), вторая и третья это I (инструкция, работающая с целочисленными значениями). Еще один пример, тип 0x1d: [MFB](#) : первая инструкция это Memory (загрузка или запись в память), вторая это Float (инструкция, работающая с [FPU](#)), третья это Branch (инструкция перехода).

Если компилятор не может подобрать подходящую инструкцию в соответствующее место пачки, он может вставить [NOP](#): вы можете здесь увидеть инструкции [nop.i](#) ([NOP](#) на том месте где должна была бы находиться целочисленная инструкция) или [nop.m](#) (инструкция обращения к памяти должна была находиться здесь). Если вручную писать на ассемблере, [NOP](#)-ы могут вставляться автоматически .

И это еще не все. Пакки тоже могут быть объединены в группы . Каждая пачка может иметь «стоп-бит», так что все следующие друг за другом пакки вплоть до той, что имеет стоп-бит, могут быть исполнены одновременно . На практике, Itanium 2 может исполнять 2 пакки одновременно, таким образом, исполнять 6 инструкций одновременно .

Так что все инструкции внутри пачки и группы не могут мешать друг другу (т.е. не должны иметь data hazard-ов) . А если это так, то результаты будут непредсказуемые.

На ассемблере, каждый стоп-бит маркируется как две точки с запятой (; ;) после инструкции. Так, инструкции на [90-ас] могут быть исполнены одновременно: они не мешают друг другу . Следующая группа: [b0-cc].

Мы также видим стоп-бит на 10с. Следующая инструкция на 110 также имеет стоп-бит. Это значит, что эти инструкции должны исполняться изолированно от всех остальных (как в CISC). Действительно: следующая инструкция на 110 использует результат, полученный от предыдущей (значение в регистре r26), так что они не могут исполняться одновременно. Должно быть, компилятор не смог найти лучший способ распараллелить инструкции, или, иными словами, загрузить CPU насколько это возможно, отсюда так много стоп-битов и NOP-ов. Писать на ассемблере вручную это также очень трудная задача: программист должен группировать инструкции вручную.

У программиста остается возможность добавлять стоп-биты к каждой инструкции, но это сведет на нет всю мощность Itanium, ради которой он создавался.

Интересные примеры написания IA64-кода вручную можно найти в исходниках ядра Linux :

<http://go.yurichev.com/17322>.

Еще пара вводных статей об ассемблере Itanium : [Mike Burrell, *Writing Efficient Itanium 2 Assembly Code* (2010)]², [papasutra of haquebright, *WRITING SHELLCODE FOR IA-64* (2001)]³.

Еще одна интересная особенность Itanium это *speculative execution* (исполнение инструкций заранее, когда еще не известно, нужно ли это) и бит NaT («not a thing»), отдаленно напоминающий NaN-числа :

MSDN.

²Также доступно здесь: <http://yurichev.com/mirrors/RE/itanium.pdf>

³Также доступно здесь: <http://phrack.org/issues/57/5.html>

Глава 96

Модель памяти в 8086

Разбирая 16-битные программы для MS-DOS или Win16 (78.3 (стр. 764) или 54.5 (стр. 596)), мы можем увидеть, что указатель состоит из двух 16-битных значений. Что это означает? О да, еще один дивный артефакт MS-DOS и 8086 .

8086/8088 был 16-битным процессором, но мог адресовать 20-битное адресное пространство (таким образом мог адресовать 1MB внешней памяти) . Внешняя адресное пространство было разделено между RAM (максимум 640KB), ПЗУ, окна для видеопамяти, EMS-карт, и т.д.

Припомним также что 8086/8088 был на самом деле наследником 8-битного процессора 8080 . Процессор 8080 имел 16-битное адресное пространство, т.е. мог адресовать только 64KB. И возможно в расчете на портирование старого ПО¹, 8086 может поддерживать 64-килобайтные окна, одновременно много таких, расположенных внутри одномегабайтного адресного пространства . Это, в каком-то смысле, игрушечная виртуализация . Все регистры 8086 16-битные, так что, чтобы адресовать больше, специальные сегментные регистры (CS, DS, ES, SS) были введены . Каждый 20-битный указатель вычисляется, используя значения из пары состоящей из сегментного регистра и адресного регистра (например DS:BX) вот так

$$real\_address = (segment\_register \ll 4) + address\_register$$

Например, окно памяти для графики (EGA², VGA³) на старых IBM PC-совместимых компьютерах имело размер 64KB. Для доступа к нему, значение 0xA000 должно быть записано в один из сегментных регистров, например, в DS. Тогда DS:0 будет адресовать самый первый байт видеопамяти, а DS:0xFFFF — самый последний байт. А реальный адрес на 20-битной адресной шине, на самом деле будет от 0xA0000 до 0xAFFFF.

Программа может содержать жесткопривязанные адреса вроде 0x1234, но ОС может иметь необходимость загрузить программу по другим адресам, так что она пересчитывает значения для сегментных регистров так, что программа будет нормально работать, не обращая внимания на то, в каком месте памяти она была расположена.

Так что, любой указатель в окружении старой MS-DOS на самом деле состоял из адреса сегмента и адреса внутри сегмента, т.е. из двух 16-битных значений. 20-битного значения было бы достаточно для этого, хотя, тогда пришлось бы вычислять адреса слишком часто: так что передача большего количества информации в стеке — это более хороший баланс между экономией места и удобством .

Кстати, из-за всего этого, не было возможным выделить блок памяти больше чем 64KB.

В 80286 сегментные регистры получили новую роль селекторов, имеющих немного другую функцию.

Когда появился процессор 80386 и компьютеры с большей памятью, MS-DOS была всё еще популярна, так что появились DOS-экстендеры: на самом деле это уже был шаг к «серьезным» ОС, они переключали CPU в защищенный режим и предлагали куда лучшее API для программ, которые всё еще предполагалось запускать в MS-DOS. Широко известные примеры это DOS/4GW (игра DOOM была скомпилирована под него), Phar Lap, PMODE

Кстати, точно такой же способ адресации памяти был и в 16-битной линейке Windows 3.x, перед Win32 .

¹Автор не уверен на 100% здесь

²Enhanced Graphics Adapter

³Video Graphics Array

Глава 97

Перестановка basic block-ов

97.1. Profile-guided optimization

Этот метод оптимизации кода может перемещать некоторые [basic block](#)-и в другую секцию исполняемого бинарного файла .

Очевидно, в функции есть места которые исполняются чаще всего (например, тела циклов) и реже всего (например, код обработки ошибок, обработчики исключений) .

Компилятор добавляет дополнительный (instrumentation) код в исполняемый файл, затем разработчик запускает его с тестами для сбора статистики. Затем компилятор, при помощи собранной статистики, приготавливает итоговый исполняемый файл где весь редко исполняемый код перемещен в другую секцию .

В результате, весь часто исполняемый код функции становится компактным, что очень важно для скорости исполнения и кэш-памяти .

Пример из Oracle RDBMS, который скомпилирован при помощи Intel C++ :

Листинг 97.1: orageneric11.dll (win32)

```
public _skgfsync
_skgfsync    proc near
; address 0x6030D86A

        db      66h
        nop
        push    ebp
        mov     ebp, esp
        mov     edx, [ebp+0Ch]
        test    edx, edx
        jz      short loc_6030D884
        mov     eax, [edx+30h]
        test    eax, 400h
        jnz     __VInfreq__skgfsync ; write to log
continue:
        mov     eax, [ebp+8]
        mov     edx, [ebp+10h]
        mov     dword ptr [eax], 0
        lea     eax, [edx+0Fh]
        and     eax, 0FFFFFFFCh
        mov     ecx, [eax]
        cmp     ecx, 45726963h
        jnz     error ; exit with error
        mov     esp, ebp
        pop     ebp
        retn
_skgfsync    endp

...

; address 0x60B953F0
__VInfreq__skgfsync:
```

97.1. PROFILE-GUIDED OPTIMIZATION

```
mov     eax, [edx]
test    eax, eax
jz      continue
mov     ecx, [ebp+10h]
push    ecx
mov     ecx, [ebp+8]
push    ecx
push    ecx
push    offset ... ; "skgfsync(se=0x%x, ctx=0x%x, iov=0x%x)\n"
push    dword ptr [edx+4]
call    dword ptr [eax] ; write to log
add     esp, 14h
jmp     continue

error:
mov     edx, [ebp+8]
mov     dword ptr [edx], 69AAh ; 27050 "function called with invalid FIB/IOV structure"
↪ "
mov     eax, [eax]
mov     [edx+4], eax
mov     dword ptr [edx+8], 0FA4h ; 4004
mov     esp, ebp
pop     ebp
retn
; END OF FUNCTION CHUNK FOR _skgfsync
```

Расстояние между двумя адресами приведенных фрагментов кода почти 9 МБ.

Весь редко исполняемый код помещен в конце секции кода DLL-файла, среди редко исполняемых частей прочих функций. Эта часть функции была отмечена компилятором Intel C++ префиксом `VInfreg`. Мы видим часть функции которая записывает в лог-файл (вероятно, в случае ошибки или предупреждения, или чего-то в этом роде) которая, наверное, не исполнялась слишком часто, когда разработчики Oracle собирали статистику (если вообще исполнялась). Basic block записывающий в лог-файл, в конце концов возвращает управление в «горячую» часть функции

Другая «редкая» часть — это `basic block` возвращающий код ошибки 27050.

В ELF-файлах для Linux весь редко исполняемый код перемещается компилятором Intel C++ в другую секцию (`text.unlikely`) оставляя весь «горячий» код в секции `text.hot`.

С точки зрения reverse engineer-а, эта информация может помочь разделить функцию на её основу и части, отвечающие за обработку ошибок.

Часть XI

Что стоит почитать

Глава 98

Книги и прочие материалы

98.1. Reverse Engineering

- Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, (2005)
- Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, (2014)
- Michael Sikorski, Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, (2012)
- Chris Eagle, *IDA Pro Book*, (2011)

Русскоязычным читателям также можно порекомендовать книги Криса Касперски.

98.2. Windows

- Mark Russinovich, *Microsoft Windows Internals*

Блоги:

- [Microsoft: Raymond Chen](#)
- [nynaeve.net](#)

98.3. Си/Си++

98.4. Си/Си++

- [Брайан Керниган, Деннис Ритчи, *Язык программирования Си*, (1988)]
- *ISO/IEC 9899:TC3 (C C99 standard)*, (2007)¹
- Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)
- Стандарт Си++11²
- Agner Fog, *Optimizing software in C++* (2015)³
- Marshall Cline, *C++ FAQ*⁴
- Денис Юричев, *Заметки о языке программирования Си/Си++*⁵

¹Также доступно здесь: <http://go.yurichev.com/17274>

²Также доступно здесь: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf>.

³Также доступно здесь: http://agner.org/optimize/optimizing_cpp.pdf.

⁴Также доступно здесь: <http://go.yurichev.com/17291>

⁵Также доступно здесь: <http://yurichev.com/C-book.html>

98.5. x86 / x86-64

- Документация от Intel⁶
- Документация от AMD⁷
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)⁸
- Agner Fog, *Calling conventions* (2015)⁹
- [Intel® 64 and IA-32 Architectures Optimization Reference Manual, (2014)]
- [Software Optimization Guide for AMD Family 16h Processors, (2013)]

98.6. ARM

- Документация от ARM¹⁰
- ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition, (2012)
- [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)]¹¹
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)¹²

98.7. Java

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] ¹³.

98.8. UNIX

[Eric S. Raymond, *The Art of UNIX Programming*, (2003)]

98.9. Криптография

98.10. Cryptography

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*¹⁴.

⁶Также доступно здесь: <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

⁷Также доступно здесь: <http://developer.amd.com/resources/developer-guides-manuals/>

⁸Также доступно здесь: <http://agner.org/optimize/microarchitecture.pdf>

⁹Также доступно здесь: http://www.agner.org/optimize/calling_conventions.pdf

¹⁰Также доступно здесь: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html>

¹¹Также доступно здесь: [http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_\(Issue_A.a\).pdf](http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf)

¹²Также доступно здесь: <http://go.yurichev.com/17273>

¹³Также доступно здесь: <https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf>; <http://docs.oracle.com/javase/specs/jvms/se7/html/>

¹⁴Также доступно здесь: <https://crypto.stanford.edu/~dabo/cryptobook/>

Глава 99

Прочее

Henry S. Warren, *Hacker's Delight*, (2002).

Имеются два отличных субреддита на reddit.com посвященных RE¹: reddit.com/r/ReverseEngineering/ и reddit.com/r/remath

Имеется также часть сайта Stack Exchange посвященная RE:

reverseengineering.stackexchange.com.

На IRC есть канал `##re` на FreeNode².

¹Reverse Engineering

²freenode.net

Послесловие

Глава 100

Вопросы?

Совершенно по любым вопросам вы можете не раздумывая писать автору: `<dennis(a)yurichev.com>` Есть идеи о том, что ещё можно добавить в эту книгу? Пожалуйста, присылайте мне информацию о замеченных ошибках (включая грамматические), итд.

Автор много работает над книгой, поэтому номера страниц, листингов, итд, очень часто меняются. Пожалуйста, в своих письмах мне не ссылайтесь на номера страниц и листингов. Есть метод проще: сделайте скриншот страницы, затем в графическом редакторе подчеркните место, где вы видите ошибку, и отправьте автору. Так он может исправить её намного быстрее. Ну а если вы знакомы с git и $\text{\LaTeX}$, вы можете исправить ошибку прямо в исходных текстах:

[GitHub](#).

Не бойтесь побеспокоить меня написав мне о какой-то мелкой ошибке, даже если вы не очень уверены. Я всё-таки пишу для начинающих, поэтому мнение и комментарии именно начинающих очень важны для моей работы.

Приложение

Приложение А

x86

А.1. Терминология

Общее для 16-bit (8086/80286), 32-bit (80386, и т.д.), 64-bit.

byte 8-бит. Для определения переменных и массива байт используется директива ассемблера DB . Байты передаются в 8-битных частях регистров : **AL/BL/CL/DL/AH/BH/CH/DH/SIL/DIL/R*L** .

word 16-бит. —директива ассемблера DW . Слова передаются в 16-битных частях регистров: **AX/BX/CX/DX/SI/DI/R*W** .

double word («dword») 32-бит. —директива ассемблера DD . Двойные слова передаются в регистрах (x86) или в 32-битных частях регистров (x64). В 16-битном коде, двойные слова передаются в парах 16-битных регистров .

quad word («qword») 64-бит. —директива ассемблера DQ . В 32-битной среде, учетверенные слова передаются в парах 32-битных регистров .

tbyte (10 байт) 80-бит или 10 байт (используется для регистров IEEE 754 FPU).

paragraph (16 байт) — термин был популярен в среде MS-DOS .

Типы данных с той же шириной (BYTE, WORD, DWORD) точно такие же и в Windows [API](#).

А.2. Регистры общего пользования

Ко многим регистрам можно обращаться как к частям размером в байт или 16-битное слово . Это всё — наследие от более старых процессоров Intel (вплоть до 8-битного 8080), все еще поддерживаемое для обратной совместимости. Старые 8-битные процессоры 8080 имели 16-битные регистры, разделенные на две части. Программы, написанные для 8080 имели доступ к младшему байту 16-битного регистра, к старшему байту или к целому 16-битному регистру. Вероятно, эта возможность была оставлена в 8086 для более простого портирования. В [RISC](#) процессорах, такой возможности, как правило, нет.

Регистры, имеющие префикс R- появились только в x86-64, а префикс E- — в 80386. Таким образом, R-регистры 64-битные, а E-регистры — 32-битные.

В x86-64 добавили еще 8 [GPR](#): R8-R15 .

N.B.: В документации от Intel, для обращения к самому младшему байту к имени регистра нужно добавлять суффикс *L*: *R8L*, но [IDA](#) называет эти регистры добавляя суффикс *B*: *R8B* .

А.2.1. RAX/EAX/AX/AL

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RAX ^{x64}							
				EAX			
						AX	
						AH	AL

[AKA](#) аккумулятор. Результат функции обычно возвращается через этот регистр .

A.2.2. RBX/EBX/BX/BL

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RBX ^{x64}							
				EBX			
						BX	
						BH	BL

A.2.3. RCX/ECX/CX/CL

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RCX ^{x64}							
				ECX			
						CX	
						CH	CL

AKA счетчик: используется в этой роли в инструкциях с префиксом REP и в инструкциях сдвига (SHL/SHR/RxL/RxR).

A.2.4. RDX/EDX/DX/DI

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RDX ^{x64}							
				EDX			
						DX	
						DH	DL

A.2.5. RSI/ESI/SI/SIL

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RSI ^{x64}							
				ESI			
						SI	
						SIL ^{x64}	

AKA «source index». Используется как источник в инструкциях REP MOVsX, REP CMPSX.

A.2.6. RDI/EDI/DI/DIL

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RDI ^{x64}							
				EDI			
						DI	
						DIL ^{x64}	

AKA «destination index». Используется как указатель на место назначения в инструкции REP MOVsX, REP STOSX.

A.2.7. R8/R8D/R8W/R8L

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R8							
				R8D			
						R8W	
						R8L	

A.2.8. R9/R9D/R9W/R9L

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R9							
				R9D			
						R9W	
							R9L

A.2.9. R10/R10D/R10W/R10L

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R10							
				R10D			
						R10W	
							R10L

A.2.10. R11/R11D/R11W/R11L

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R11							
				R11D			
						R11W	
							R11L

A.2.11. R12/R12D/R12W/R12L

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R12							
				R12D			
						R12W	
							R12L

A.2.12. R13/R13D/R13W/R13L

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R13							
				R13D			
						R13W	
							R13L

A.2.13. R14/R14D/R14W/R14L

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R14							
				R14D			
						R14W	
							R14L

A.2.14. R15/R15D/R15W/R15L

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
R15							
				R15D			
						R15W	
							R15L

A.2.15. RSP/ESP/SP/SPL

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RSP							
				ESP			
						SP	
							SPL

[AKA stack pointer](#). Обычно всегда указывает на текущий стек, кроме тех случаев, когда он не инициализирован.

A.2.16. RBP/EBP/BP/BPL

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RBP							
				EBP			
						BP	
							BPL

[AKA frame pointer](#). Обычно используется для доступа к локальным переменным функции и аргументам, Больше о нем : (8.1.2 (стр. 63)).

A.2.17. RIP/EIP/IP

Номер байта:							
7-й	6-й	5-й	4-й	3-й	2-й	1-й	0-й
RIP ^{x64}							
				EIP			
						IP	

[AKA «instruction pointer»](#) ¹. Обычно всегда указывает на инструкцию, которая сейчас будет исполняться Напрямую модифицировать регистр нельзя, хотя можно делать так (что равноценно):

```
MOV EAX, ...
JMP EAX
```

Либо:

```
PUSH value
RET
```

A.2.18. CS/DS/ES/SS/FS/GS

16-битные регистры, содержащие селектор кода (CS), данных (DS), стека (SS).

FS в win32 указывает на [TLS](#), а в Linux на эту роль был выбран GS . Это сделано для более быстрого доступа к [TLS](#) и прочим структурам там вроде [TIB](#) .

В прошлом эти регистры использовались как сегментные регистры ([96](#) (стр. 898)).

¹Иногда называется также «program counter»

А.2.19. Регистр флагов

АКА EFLAGS.

Бит (маска)	Аббревиатура (значение)	Описание
0 (1)	CF (Carry)	Флаг переноса. Инструкции CLC/STC/CMC используются для установки/сброса/инвертирования этого флага
2 (4)	PF (Parity)	Флаг четности (18.7.1 (стр. 230)).
4 (0x10)	AF (Adjust)	Существует только для работы с числами закодированными в BCD
6 (0x40)	ZF (Zero)	Выставляется в 0 если результат последней операции был 0.
7 (0x80)	SF (Sign)	Флаг знака.
8 (0x100)	TF (Trap)	Применяется при отладке. Если включен, то после исполнения каждой инструкции будет сгенерировано исключение.
9 (0x200)	IF (Interrupt enable)	Разрешены ли прерывания. Инструкции CLI/STI используются для установки/сброса этого флага
10 (0x400)	DF (Direction)	Задается направление для инструкций REP MOVSB, REP CMPSB, REP LODSB, REP SCASB. Инструкции CLD/STD используются для установки/сброса этого флага
11 (0x800)	OF (Overflow)	Переполнение.
12, 13 (0x3000)	IOPL (I/O privilege level) ⁸⁰²⁸⁶	
14 (0x4000)	NT (Nested task) ⁸⁰²⁸⁶	
16 (0x10000)	RF (Resume) ⁸⁰³⁸⁶	Применяется при отладке. Если включить, CPU проигнорирует хардварную точку останова в DRx.
17 (0x20000)	VM (Virtual 8086 mode) ⁸⁰³⁸⁶	
18 (0x40000)	AC (Alignment check) ⁸⁰⁴⁸⁶	
19 (0x80000)	VIF (Virtual interrupt) ^{Pentium}	
20 (0x100000)	VIP (Virtual interrupt pending) ^{Pentium}	
21 (0x200000)	ID (Identification) ^{Pentium}	

Остальные флаги зарезервированы.

А.3. FPU регистры

8 80-битных регистров работающих как стек: ST(0)-ST(7). N.B.: IDA называет ST(0) просто ST. Числа хранятся в формате IEEE 754.

Формат значения *long double*:

79	78	64	63	62	0
S	экспонента			I	мантисса

(S — знак, I — целочисленная часть)

А.3.1. Регистр управления

Регистр, при помощи которого можно задавать поведение FPU.

A.3. FPU РЕГИСТРЫ

Бит	Аббревиатура (значение)	Описание
0	IM (Invalid operation Mask)	
1	DM (Denormalized operand Mask)	
2	ZM (Zero divide Mask)	
3	OM (Overflow Mask)	
4	UM (Underflow Mask)	
5	PM (Precision Mask)	
7	IEM (Interrupt Enable Mask)	Разрешение исключений, по умолчанию 1 (запрещено)
8, 9	PC (Precision Control)	Управление точностью 00 – 24 бита (REAL4) 10 – 53 бита (REAL8) 11 – 64 бита (REAL10)
10, 11	RC (Rounding Control)	Управление округлением 00 – (по умолчанию) округлять к ближайшему 01 – округлять к $-\infty$ 10 – округлять к $+\infty$ 11 – округлять к 0
12	IC (Infinity Control)	0 – (по умолчанию) считать $+\infty$ и $-\infty$ за беззнаковое 1 – учитывать и $+\infty$ и $-\infty$

Флагами PM, UM, OM, ZM, DM, IM задается, генерировать ли исключения в случае соответствующих ошибок .

A.3.2. Регистр статуса

Регистр только для чтения.

Бит	Аббревиатура (значение)	Описание
15	B (Busy)	Работает ли сейчас FPU (1) или закончил и результаты готовы (0)
14	C3	
13, 12, 11	TOP	указывает, какой сейчас регистр является нулевым
10	C2	
9	C1	
8	C0	
7	IR (Interrupt Request)	
6	SF (Stack Fault)	
5	P (Precision)	
4	U (Underflow)	
3	O (Overflow)	
2	Z (Zero)	
1	D (Denormalized)	
0	I (Invalid operation)	

Биты SF, P, U, O, Z, D, I сигнализируют об исключениях .

О C3, C2, C1, C0 читайте больше: (18.7.1 (стр. 230)).

N.B.: когда используется регистр ST(x), FPU прибавляет x к TOP по модулю 8 и получается номер внутреннего регистра.

A.3.3. Tag Word

Этот регистр отражает текущее содержимое регистров чисел .

Бит	Аббревиатура (значение)
15, 14	Tag(7)
13, 12	Tag(6)
11, 10	Tag(5)
9, 8	Tag(4)
7, 6	Tag(3)
5, 4	Tag(2)
3, 2	Tag(1)
1, 0	Tag(0)

Каждый тэг содержит информацию о физическом регистре FPU (R(x)), но не логическом (ST(x)).

Для каждого тэга:

- 00 — Регистр содержит ненулевое значение
- 01 — Регистр содержит 0
- 10 — Регистр содержит специальное число (NaN^2 , ∞ , или денормализованное число)
- 11 — Регистр пуст

A.4. SIMD регистры

A.4.1. MMX регистры

8 64-битных регистров: MM0..MM7.

A.4.2. SSE и AVX регистры

SSE: 8 128-битных регистров: XMM0..XMM7. В x86-64 добавлено еще 8 регистров: XMM8..XMM15.

AVX это расширение всех регистры до 256 бит.

A.5. Отладочные регистры

Применяются для работы с т.н. hardware breakpoints.

- DR0 — адрес точки останова #1
- DR1 — адрес точки останова #2
- DR2 — адрес точки останова #3
- DR3 — адрес точки останова #4
- DR6 — здесь отображается причина останова
- DR7 — здесь можно задать типы точек останова

A.5.1. DR6

Бит (маска)	Описание
0 (1)	B0 — сработала точка останова #1
1 (2)	B1 — сработала точка останова #2
2 (4)	B2 — сработала точка останова #3
3 (8)	B3 — сработала точка останова #4
13 (0x2000)	BD — была попытка модифицировать один из регистров DRx. может быть выставлен если бит GD выставлен.
14 (0x4000)	BS — точка останова типа single step (флаг TF был выставлен в EFLAGS) . Наивысший приоритет. Другие биты также могут быть выставлены .
15 (0x8000)	BT (task switch flag)

N.B. Точка останова single step это срабатывающая после каждой инструкции . Может быть включена выставлением флага TF в EFLAGS ([A.2.19](#) (стр. [912](#))).

A.5.2. DR7

В этом регистре задаются типы точек останова.

²Not a Number

А.6. ИНСТРУКЦИИ

Бит (маска)	Описание
0 (1)	L0 — разрешить точку останова #1 для текущей задачи
1 (2)	G0 — разрешить точку останова #1 для всех задач
2 (4)	L1 — разрешить точку останова #2 для текущей задачи
3 (8)	G1 — разрешить точку останова #2 для всех задач
4 (0x10)	L2 — разрешить точку останова #3 для текущей задачи
5 (0x20)	G2 — разрешить точку останова #3 для всех задач
6 (0x40)	L3 — разрешить точку останова #4 для текущей задачи
7 (0x80)	G3 — разрешить точку останова #4 для всех задач
8 (0x100)	LE — не поддерживается, начиная с P6
9 (0x200)	GE — не поддерживается, начиная с P6
13 (0x2000)	GD — исключение будет вызвано если какая-либо инструкция MOV попытается модифицировать один из DRx-регистров
16,17 (0x30000)	точка останова #1: R/W — тип
18,19 (0xC0000)	точка останова #1: LEN — длина
20,21 (0x300000)	точка останова #2: R/W — тип
22,23 (0xC00000)	точка останова #2: LEN — длина
24,25 (0x3000000)	точка останова #3: R/W — тип
26,27 (0xC000000)	точка останова #3: LEN — длина
28,29 (0x30000000)	точка останова #4: R/W — тип
30,31 (0xC0000000)	точка останова #4: LEN — длина

Так задается тип точки останова (R/W):

- 00 — исполнение инструкции
- 01 — запись в память
- 10 — обращения к I/O-портам (недоступно из user-mode)
- 11 — обращение к памяти (чтение или запись)

N.B.: отдельного типа для чтения из памяти действительно нет .

Так задается длина точки останова (LEN):

- 00 — 1 байт
- 01 — 2 байта
- 10 — не определено для 32-битного режима, 8 байт для 64-битного
- 11 — 4 байта

А.6. Инструкции

Инструкции, отмеченные как (M) обычно не генерируются компилятором: если вы видите её, вероятно, это вручную написанный фрагмент кода, либо это т.н. compiler intrinsic ([92](#) (стр. [887](#))).

Только наиболее используемые инструкции перечислены здесь . Обращайтесь к ?? (стр. ??) для полной документации.

Нужно ли заучивать опкоды инструкций на память? Нет, только те, которые часто используются для модификации кода ([91.2](#) (стр. [886](#))). Остальные запоминать нет смысла.

А.6.1. Префиксы

LOCK используется чтобы предоставить эксклюзивный доступ к памяти в многопроцессорной среде . Для упрощения, можно сказать, что когда выполняется инструкция с этим префиксом, остальные процессоры в системе останавливаются. Чаще все это используется для критических секций, семафоров, мьютексов. Обычно используется с ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. Читайте больше о критических секциях ([69.4](#) (стр. [725](#))).

REP используется с инструкциями MOVsx и STOSx: инструкция будет исполняться в цикле, счетчик расположен в регистре CX/ECX/RCX . Для более детального описания, читайте больше об инструкциях MOVsx ([А.6.2](#) (стр. [918](#))) и STOSx ([А.6.2](#) (стр. [919](#))).

Работа инструкций с префиксом REP зависит от флага DF, он задает направление .

REPE/REPNE (AKA REPZ/REPNZ) используется с инструкциями CMPSx и SCASx: инструкция будет исполняться в цикле, счетчик расположен в регистре CX / ECX / RCX . Выполнение будет прервано если ZF будет 0 (REPE) либо если ZF будет 1 (REPNE) .

Для более детального описания, читайте больше об инструкциях CMPSx (A.6.3 (стр. 920)) и SCASx (A.6.2 (стр. 919)).

Работа инструкций с префиксами REPE/REPNE зависит от флага DF, он задает направление .

А.6.2. Наиболее часто используемые инструкции

Их можно заучить в первую очередь.

ADC (*add with carry*) сложить два значения, **инкремент** если выставлен флаг CF. ADC часто используется для складывания больших значений, например, складывания двух 64-битных значений в 32-битной среде используя две инструкции ADD и ADC, например:

```
; работа с 64-битными значениями: прибавить val1 к val2.
; .lo означает младшие 32 бита, .hi - старшие
ADD val1.lo, val2.lo
ADC val1.hi, val2.hi ; использовать CF выставленный или очищенный в предыдущей инструкции
```

Еще один пример: 25 (стр. 392).

ADD сложить два значения

AND логическое «И»

CALL вызвать другую функцию: PUSH address_after_CALL_instruction; JMP label

CMP сравнение значений и установка флагов, то же что и SUB , но только без записи результата

DEC decrement. В отличие от других арифметических инструкций, INC не модифицирует флаг CF.

IMUL умножение с учетом знаковых значений

INC increment. В отличие от других арифметических инструкций, INC не модифицирует флаг CF.

JCXZ, JECXZ, JRCXZ (M) переход если CX/ECX/RCX=0

JMP перейти на другой адрес. ОPCODE имеет т.н. jump offset.

Jcc (где cc — condition code)

Немало этих инструкций имеют синонимы (отмечены с AKA), это сделано для удобства . Синонимичные инструкции транслируются в один и тот же OPCODE . ОPCODE имеет т.н. jump offset.

JAE AKA JNC: переход если больше или равно (беззнаковый): CF=0

JA AKA JNBE: переход если больше (беззнаковый): CF=0 и ZF=0

JBE переход если меньше или равно (беззнаковый): CF=1 или ZF=1

JB AKA JC: переход если меньше (беззнаковый): CF=1

JC AKA JB: переход если CF=1

JE AKA JZ: переход если равно или ноль: ZF=1

JGE переход если больше или равно (знаковый): SF=OF

JG переход если больше (знаковый): ZF=0 и SF=OF

JLE переход если меньше или равно (знаковый): ZF=1 или SF≠OF

JL переход если меньше (знаковый): SF≠OF

JNAE AKA JC: переход если не больше или равно (беззнаковый) CF=1

JNA переход если не больше (беззнаковый) CF=1 и ZF=1

JNBE переход если не меньше или равно (беззнаковый): CF=0 и ZF=0

JNB AKA JNC: переход если не меньше (беззнаковый): CF=0

JNC AKA JAE: переход если CF=0, синонимично JNB.

JNE AKA JNZ: переход если не равно или не ноль: ZF=0

JNGE переход если не больше или равно (знаковый): SF≠OF

JNG переход если не больше (знаковый): ZF=1 или SF≠OF

JNLE переход если не меньше (знаковый): ZF=0 и SF=OF

JNL переход если не меньше (знаковый): SF=OF

JNO переход если не переполнение: OF=0

JNS переход если флаг SF сброшен

JNZ AKA JNE: переход если не равно или не ноль: ZF=0

JO переход если переполнение: OF=1

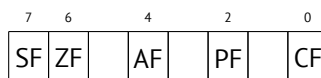
JPO переход если сброшен флаг PF (Jump Parity Odd)

JP AKA JPE: переход если выставлен флаг PF

JS переход если выставлен флаг SF

JZ AKA JE: переход если равно или ноль: ZF=1

LAHF скопировать некоторые биты флагов в AH:



LEAVE аналог команд **MOV ESP, EBP** и **POP EBP** — то есть возврат [указателя стека](#) и регистра **EBP** в первоначальное состояние.

LEA (*Load Effective Address*) сформировать адрес

Это инструкция, которая задумывалась вовсе не для складывания и умножения чисел, а для формирования адреса например, из указателя на массив и прибавления индекса к нему ³.

То есть, разница между **MOV** и **LEA** в том, что **MOV** формирует адрес в памяти и загружает значение из памяти, либо записывает его туда, а **LEA** только формирует адрес.

Тем не менее, её можно использовать для любых других вычислений.

LEA удобна тем, что производимые ею вычисления не модифицируют флаги **CPU**. Это может быть очень важно для **OOE** процессоров (чтобы было меньше зависимостей между данными).

```
int f(int a, int b)
{
    return a*8+b;
};
```

Листинг A.1: Оптимизирующий MSVC 2010

```
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_f PROC
    mov     eax, DWORD PTR _b$[esp-4]
    mov     ecx, DWORD PTR _a$[esp-4]
    lea     eax, DWORD PTR [eax+ecx*8]
    ret     0
_f ENDP
```

Intel C++ использует LEA даже больше:

```
int f1(int a)
{
    return a*13;
};
```

Листинг A.2: Intel C++ 2011

```
_f1 PROC NEAR
    mov     ecx, DWORD PTR [4+esp] ; ecx = a
    lea     edx, DWORD PTR [ecx+ecx*8] ; edx = a*9
    lea     eax, DWORD PTR [edx+ecx*4] ; eax = a*9 + a*4 = a*13
    ret
```

³См. также: wikipedia

Эти две инструкции вместо одной IMUL будут работать быстрее.

MOVSБ/MOVSW/MOVSД/MOVSQ скопировать байт/ 16-битное слово/ 32-битное слово/ 64-битное слово на который указывает SI/ESI/RSI куда указывает DI/EDI/RDI.

Вместе с префиксом REP, инструкция выполняется в цикле, счетчик находится в регистре CX/ECX/RX: это работает как метсру() в Си. Если размер блока известен компилятору на стадии компиляции, метсру() часто компилируется в короткий фрагмент кода использующий REP MOVSB, иногда даже несколько инструкций .

Эквивалент метсру(EDI, ESI, 15):

```
; скопировать 15 байт из ESI в EDI
CLD      ; установить направление на "вперед"
MOV ECX, 3
REP MOVSD ; скопировать 12 байт
MOVSW    ; скопировать еще 2 байта
MOVSB    ; скопировать оставшийся байт
```

(Вероятно, так быстрее чем копировать 15 байт используя просто одну REP MOVSB).

MOVSB загрузить с расширением знака см. также: (16.1.1 (стр. 197))

MOVZB загрузить и очистить все остальные биты см. также: (16.1.1 (стр. 198))

MOV загрузить значение. эта инструкция была названа неудачно (данные не перемещаются, а копируются), что является результатом путаницы: в других архитектурах эта же инструкция называется «LOAD» и/или «STORE» или что-то в этом роде.

Важно: если в 32-битном режиме при помощи MOV записывать младшую 16-байтную часть регистра, то старшие 16 бит останутся такими же. Но если в 64-битном режиме модифицировать 32-битную часть регистра, то старшие 32 бита обнуляются.

Вероятно, это сделано для упрощения портирования кода под x86-64.

MUL умножение с учетом беззнаковых значений.

NEG смена знака: $op = -op$

NOP **NOP**. Её опкод 0x90, что на самом деле это холостая инструкция **XCHG EAX, EAX**. Это значит, что в x86 (как и во многих **RISC**) нет отдельной **NOP**-инструкции . В этой книге есть по крайней мере один листинг, где GDB отображает NOP как 16-битную инструкцию XCHG: 7.1.1 (стр. 43).

Еще примеры подобных операций: (90 (стр. 884)).

NOP может быть сгенерировать компилятором для выравнивания меток по 16-байтной границе . Другое очень популярное использование **NOP** это вставка её вручную (патчинг) на месте какой-либо инструкции вроде условного перехода, чтобы запретить её исполнение.

NOT $op1: op1 = \neg op1$. логическое «НЕ» Особенность — инструкция не меняет флаги.

OR логическое «ИЛИ»

POP взять значение из стека: `value=SS:[ESP]; ESP=ESP+4 (или 8)`

PUSH записать значение в стек: `ESP=ESP-4 (или 8); SS:[ESP]=value`

RET возврат из процедуры: `POP tmp; JMP tmp`.

В реальности, RET это макрос ассемблера, в среде Windows и *NIX транслирующийся в RETN («return near») ибо, во времена MS-DOS, где память адресовалась немного иначе (96 (стр. 898)), в RETF («return far»).

RET может иметь операнд. Тогда его работа будет такой : `POP tmp; ADD ESP op1; JMP tmp`. **RET** с операндом обычно завершает функции с соглашением о вызовах *stdcall*, см. также : 65.2 (стр. 672).

SAHF скопировать биты из AH в флаги CPU:

7	6	4	2	0
SF	ZF	AF	PF	CF

SBB (*subtraction with borrow*) вычесть одно значение из другого, **декремент** результата если флаг CF выставлен. SBB часто используется для вычитания больших значений, например, для вычитания двух 64-битных значений в 32-битной среде используя инструкции SUB и SBB, например:

```
; работа с 64-битными значениями: вычесть val2 из val1
; .lo означает младшие 32 бита, .hi - старшие
SUB val1.lo, val2.lo
```

SBB val1.hi, val2.hi ; использовать CF выставленный или очищенный в предыдущей инструкции

Еще один пример: 25 (стр. 392).

SCASB/SCASW/SCASD/SCASQ (M) сравнить байт/ 16-битное слово/ 32-битное слово/ 64-битное слово, записанное в AX/EAX/RAX со значением, адрес которого находится в DI/EDI/RDI. Выставить флаги так же, как это делает CMP.

Эта инструкция часто используется с префиксом REPNE: продолжать сканировать буфер до тех пор, пока не встретится специальное значение, записанное в AX/EAX/RAX. Отсюда «NE» в REPNE: продолжать сканирование если сравниваемые значения не равны и остановиться если равны.

Она часто используется как стандартная функция Си strlen(), для определения длины ASCIIZ-строки:

Пример:

```
lea    edi, string
mov    ecx, 0FFFFFFFh ; сканировать 232 - 1 байт, т.е. почти "бесконечно"
xor    eax, eax        ; конец строки это 0
repne scasb
add    edi, 0FFFFFFFh ; скорректировать

; теперь EDI указывает на последний символ в ASCIIZ-строке.

; узнать длину строки
; сейчас ECX = -1-strlen

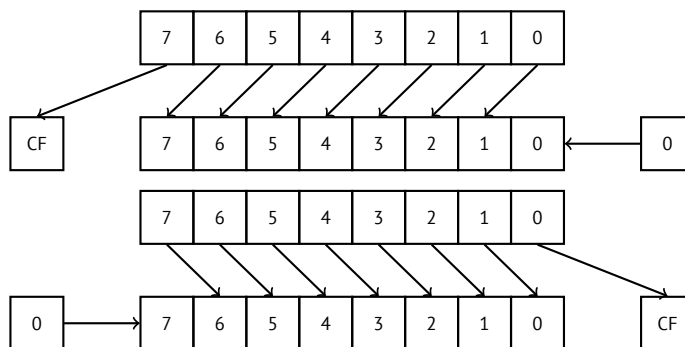
not    ecx
dec    ecx

; теперь в ECX хранится длина строки
```

Если использовать другое значение AX/EAX/RAX, функция будет работать как стандартная функция Си memchr(), т.е. для поиска определенного байта.

SHL сдвинуть значение влево

SHR сдвинуть значение вправо:



Эти инструкции очень часто применяются для умножения и деления на 2^n . Еще одно очень частое применение это работа с битовыми полями: 20 (стр. 300).

SHRD ор1, ор2, ор3: сдвинуть значение в ор2 вправо на ор3 бит, подтягивая биты из ор1.

Пример: 25 (стр. 392).

STOSB/STOSW/STOSD/STOSQ записать байт/ 16-битное слово/ 32-битное слово/ 64-битное слово из AX/EAX/RAX в место, адрес которого находится в DI/EDI/RDI.

Вместе с префиксом REP, инструкция будет исполняться в цикле, счетчик будет находится в регистре CX/ECX/RAX: это работает как memset() в Си. Если размер блока известен компилятору на стадии компиляции, memset() часто компилируется в короткий фрагмент кода использующий REP STOSx, иногда даже несколько инструкций.

Эквивалент memset(EDI, 0xAA, 15):

```
; записать 15 байт 0xAA в EDI
CLD    ; установить направление на "вперед"
MOV    EAX, 0AAAAAAAAh
MOV    ECX, 3
REP    STOSD    ; записать 12 байт
STOSW    ; записать еще 2 байта
STOSB    ; записать оставшийся байт
```

A.6. ИНСТРУКЦИИ

(Вероятно, так быстрее чем заполнять 15 байт используя просто одну REP STOSB).

SUB вычесть одно значение из другого. часто встречающийся вариант `SUB reg, reg` означает обнуление *reg*.

TEST то же что и AND, но без записи результатов, см. также: 20 (стр. 300)

XCHG обменять местами значения в операндах

XOR *op1, op2*: `XOR4` значений. $op1 = op1 \oplus op2$. Часто встречающийся вариант `XOR reg, reg` означает обнуление регистра *reg*. См. также: ?? (стр. ??).

A.6.3. Реже используемые инструкции

BSF *bit scan forward*, см. также: 26.2 (стр. 416)

BSR *bit scan reverse*

BSWAP (*byte swap*), смена [порядка байт](#) в значении.

BTC *bit test and complement*

BTR *bit test and reset*

BTS *bit test and set*

BT *bit test*

CBW/CWD/CWDE/CDQ/CDQE Расширить значение учитывая его знак:

CBW конвертировать байт в AL в слово в AX

CWD конвертировать слово в AX в двойное слово в DX:AX

CWDE конвертировать слово в AX в двойное слово в EAX

CDQ конвертировать двойное слово в EAX в четверное слово в EDX:EAX

CDQE (x64) конвертировать двойное слово в EAX в четверное слово в RAX

Эти инструкции учитывают знак значения, расширяя его в старшую часть выходного значения. См. также: 25.5 (стр. 401).

Интересно узнать, что эти инструкции назывались **SEX** (*Sign EXtend*), как Stephen P. Morse (один из создателей Intel 8086 CPU) пишет в [Stephen P. Morse, *The 8086 Primer*, (1980)]⁵:

The process of stretching numbers by extending the sign bit is called sign extension. The 8086 provides instructions (Fig. 3.29) to facilitate the task of sign extension. These instructions were initially named SEX (sign extend) but were later renamed to the more conservative CBW (convert byte to word) and CWD (convert word to double word).

CLD сбросить флаг DF.

CLI (M) сбросить флаг IF

CMC (M) инвертировать флаг CF

CMOVcc условный MOV: загрузить значение если условие верно. Коды точно такие же, как и в инструкциях Jcc (A.6.2 (стр. 916)).

CMPSB/CMPSW/CMPSD/CMPSQ (M) сравнить байт/ 16-битное слово/ 32-битное слово/ 64-битное слово из места, адрес которого находится в SI/ESI/RSI со значением, адрес которого находится в DI/EDI/RDI. Выставить флаги так же, как это делает **CMP**.

Вместе с префиксом REPE, инструкция будет исполняться в цикле, счетчик будет находится в регистре CX/ECX/RCX, процесс будет продолжаться пока флаг ZF=0 (т.е. до тех пор, пока все сравниваемые значения равны, отсюда «Е» в REPE).

Это работает как `memcmp()` в Си.

Пример из ядра Windows NT ([WRK v1.2](#)):

⁴eXclusive OR (исключающее «ИЛИ»)

⁵Также доступно здесь: <https://archive.org/details/The8086Primer>

```

; ULONG
; RtlCompareMemory (
;     IN PVOID Source1,
;     IN PVOID Source2,
;     IN ULONG Length
; )
;
; Routine Description:
;
;     This function compares two blocks of memory and returns the number
;     of bytes that compared equal.
;
; Arguments:
;
;     Source1 (esp+4) – Supplies a pointer to the first block of memory to
;     compare.
;
;     Source2 (esp+8) – Supplies a pointer to the second block of memory to
;     compare.
;
;     Length (esp+12) – Supplies the Length, in bytes, of the memory to be
;     compared.
;
; Return Value:
;
;     The number of bytes that compared equal is returned as the function
;     value. If all bytes compared equal, then the length of the original
;     block of memory is returned.
;
;--

RcmSource1      equ     [esp+12]
RcmSource2      equ     [esp+16]
RcmLength       equ     [esp+20]

CODE_ALIGNMENT
cPublicProc _RtlCompareMemory,3
cPublicFpo 3,0

        push     esi                ; save registers
        push     edi                ;
        cld                     ; clear direction
        mov      esi,RcmSource1     ; (esi) -> first block to compare
        mov      edi,RcmSource2     ; (edi) -> second block to compare
;
; Compare dwords, if any.
;
rcm10:  mov      ecx,RcmLength       ; (ecx) = length in bytes
        shr      ecx,2              ; (ecx) = length in dwords
        jz       rcm20              ; no dwords, try bytes
        repe     cmpsd              ; compare dwords
        jnz     rcm40              ; mismatch, go find byte
;
; Compare residual bytes, if any.
;
rcm20:  mov      ecx,RcmLength       ; (ecx) = length in bytes
        and      ecx,3              ; (ecx) = length mod 4
        jz       rcm30              ; 0 odd bytes, go do dwords
        repe     cmpsb              ; compare odd bytes
        jnz     rcm50              ; mismatch, go report how far we got
;
; All bytes in the block match.
;

```

```

rcm30:  mov     eax,RcmLength      ; set number of matching bytes
        pop     edi              ; restore registers
        pop     esi              ;
        stdRET  _RtlCompareMemory

;
;  When we come to rcm40, esi (and edi) points to the dword after the
;  one which caused the mismatch. Back up 1 dword and find the byte.
;  Since we know the dword didn't match, we can assume one byte won't.
;
rcm40:  sub     esi,4              ; back up
        sub     edi,4              ; back up
        mov     ecx,5              ; ensure that ecx doesn't count out
        repe    cmpsb             ; find mismatch byte

;
;  When we come to rcm50, esi points to the byte after the one that
;  did not match, which is TWO after the last byte that did match.
;
rcm50:  dec     esi              ; back up
        sub     esi,RcmSource1    ; compute bytes that matched
        mov     eax,esi           ;
        pop     edi              ; restore registers
        pop     esi              ;
        stdRET  _RtlCompareMemory

stdENDP _RtlCompareMemory

```

N.B.: эта функция использует сравнение 32-битных слов (CMPSD) если длина блоков кратна 4-м байтам, либо побайтовое сравнение (CMPSB) если не кратна .

CPUID получить информацию о доступных возможностях [CPU](#) . см. также: (22.6.1 (стр. 364)).

DIV деление с учетом беззнаковых значений

IDIV деление с учетом знаковых значений

INT (M): `INT x` аналогична `PUSHF; CALL dword ptr [x*4]` в 16-битной среде. Она активно использовалась в MS-DOS, работая как сисколл. Аргументы записывались в регистры AX/BX/CX/DX/SI/DI и затем происходил переход на таблицу векторов прерываний (расположенную в самом начале адресного пространства) . Она была очень популярна потому что имела короткий опкод (2 байта) и программе использующая сервисы MS-DOS не нужно было заморачиваться узнавая адреса всех функций этих сервисов . Обработчик прерываний возвращал управление назад при помощи инструкции `IRET` .

Самое используемое прерывание в MS-DOS было 0x21, там была основная часть его [API](#) . См. также: [Ralf Brown *Ralf Brown's Interrupt List*], самый крупный список всех известных прерываний и вообще там много информации о MS-DOS .

Во времена после MS-DOS, эта инструкция все еще использовалась как сискол, и в Linux и в Windows (67 (стр. 686)), но позже была заменена инструкцией `SYSENTER` или `SYSCALL` .

INT 3 (M): эта инструкция стоит немного в стороне от `INT` , она имеет собственный 1-байтный опкод (`0xCC`), и активно используется в отладке. Часто, отладчик просто записывает байт `0xCC` по адресу в памяти где устанавливается точка останова, и когда исключение поднимается, оригинальный байт будет восстановлен и оригинальная инструкция по этому адресу исполнена заново.

В [Windows NT](#), исключение `EXCEPTION_BREAKPOINT` поднимается, когда [CPU](#) исполняет эту инструкцию. Это отладочное событие может быть перехвачено и обработано отладчиком, если он загружен . Если он не загружен, Windows предложит запустить один из зарегистрированных в системе отладчиков . Если [MSVS](#)⁶ установлена, его отладчик может быть загружен и подключен к процессу. В целях защиты от [reverse engineering](#), множество анти-отладочных методов проверяют целостность загруженного кода.

В [MSVC](#) есть [compiler intrinsic](#) для этой инструкции: `__debugbreak()` ⁷.

В win32 также имеется функция в `kernel32.dll` с названием `DebugBreak()` ⁸, которая также исполняет `INT 3` .

⁶Microsoft Visual Studio

⁷[MSDN](#)

⁸[MSDN](#)

А.6. ИНСТРУКЦИИ

IN (M) получить данные из порта. Эту инструкцию обычно можно найти в драйверах OS либо в старом коде для MS-DOS, например (78.3 (стр. 764)).

IRET : использовалась в среде MS-DOS для возврата из обработчика прерываний, после того как он был вызван при помощи инструкции INT . Эквивалентна `POP tmp; POPF; JMP tmp` .

LOOP (M) **декремент** CX/ECX/RCX, переход если он всё еще не ноль.

OUT (M) послать данные в порт. Эту инструкцию обычно можно найти в драйверах OS либо в старом коде для MS-DOS, например (78.3 (стр. 764)).

POPA (M) восстанавливает значения регистров (R)EDI, (R)ESI, (R)EBP, (R)EBX, (R)EDX, (R)ECX, (R)EAX из стека.

POPCNT population count. Считает количество бит выставленных в 1 в значении . AKA «hamming weight». AKA «NSA instruction» из-за слухов:

This branch of cryptography is fast-paced and very politically charged. Most designs are secret; a majority of military encryptions systems in use today are based on LFSRs. In fact, most Cray computers (Cray 1, Cray X-MP, Cray Y-MP) have a rather curious instruction generally known as “population count.” It counts the 1 bits in a register and can be used both to efficiently calculate the Hamming distance between two binary words and to implement a vectorized version of a LFSR. I’ve heard this called the canonical NSA instruction, demanded by almost all computer contracts.

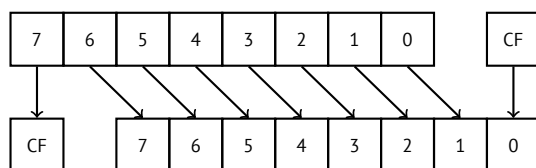
[Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)]

POPF восстановить флаги из стека (AKA регистр EFLAGS)

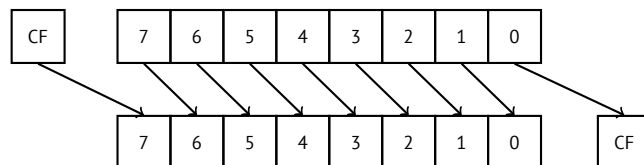
PUSHA (M) сохраняет значения регистров (R)EAX, (R)ECX, (R)EDX, (R)EBX, (R)EBP, (R)ESI, (R)EDI в стеке.

PUSHF сохранить в стеке флаги (AKA регистр EFLAGS)

RCL (M) вращать биты налево через флаг CF:

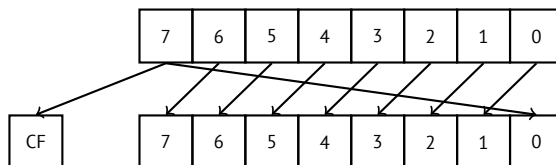


RCR (M) вращать биты направо через флаг CF:

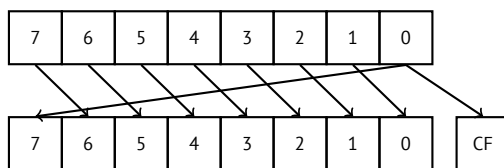


ROL/ROR (M) циклический сдвиг

ROL: вращать налево:



ROR: вращать направо:



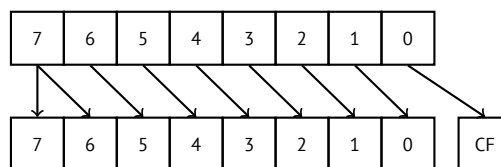
Не смотря на то что многие CPU имеют эти инструкции, в Си/Си++ нет соответствующих операций, так что компиляторы с этих ЯП обычно не генерируют код использующий эти инструкции.

Чтобы программисту были доступны эти инструкции, в MSVC есть псевдофункции (compiler intrinsics) `_rotl()` и `_rotr()`⁹, которые транслируются компилятором напрямую в эти инструкции .

⁹MSDN

SAL Арифметический сдвиг влево, синонимично **SHL**

SAR Арифметический сдвиг вправо



Таким образом, бит знака всегда остается на месте **MSB**.

SETcc op: загрузить 1 в op (только байт) если условие верно или 0 если наоборот. Коды точно такие же, как и в инструкциях Jcc (A.6.2 (стр. 916)).

STC (M) установить флаг CF

STD (M) установить флаг DF. Эта инструкция не генерируется компиляторами и вообще редкая. Например, она может быть найдена в файле `ntoskrnl.exe` (ядро Windows) в написанных вручную функциях копирования памяти.

STI (M) установить флаг IF

SYSCALL (AMD) вызов сисколла (67 (стр. 686))

SYSENTER (Intel) вызов сисколла (67 (стр. 686))

UD2 (M) неопределенная инструкция, вызывает исключение. Применяется для тестирования.

A.6.4. Инструкции FPU

-R в названии инструкции обычно означает что операнды поменяны местами, -P означает что один элемент выталкивается из стека после исполнения инструкции, -PP означает что выталкиваются два элемента.

-P инструкции часто бывают полезны, когда нам уже больше не нужно хранить значение в FPU-стеке после операции.

FABS заменить значение в ST(0) на абсолютное значение ST(0)

FADD op: ST(0)=op+ST(0)

FADD ST(0), ST(i): ST(0)=ST(0)+ST(i)

FADDP ST(1)=ST(0)+ST(1); вытолкнуть один элемент из стека, таким образом, складываемые значения в стеке заменяются суммой

FCHS ST(0)=-ST(0)

FCOM сравнить ST(0) с ST(1)

FCOM op: сравнить ST(0) с op

FCOMP сравнить ST(0) с ST(1); вытолкнуть один элемент из стека

FCOMPP сравнить ST(0) с ST(1); вытолкнуть два элемента из стека

FDIVR op: ST(0)=op/ST(0)

FDIVR ST(i), ST(j): ST(i)=ST(j)/ST(i)

FDIVRP op: ST(0)=op/ST(0); вытолкнуть один элемент из стека

FDIVRP ST(i), ST(j): ST(i)=ST(j)/ST(i); вытолкнуть один элемент из стека

FDIV op: ST(0)=ST(0)/op

FDIV ST(i), ST(j): ST(i)=ST(i)/ST(j)

FDIVP ST(1)=ST(0)/ST(1); вытолкнуть один элемент из стека, таким образом, делимое и делитель в стеке заменяются частным

FILD op: сконвертировать целочисленный op и затолкнуть его в стек.

FIST op: конвертировать ST(0) в целочисленное op

FISTP op: конвертировать ST(0) в целочисленное op; вытолкнуть один элемент из стека

FLD1 затолкнуть 1 в стек

FLDCW op: загрузить FPU control word (A.3 (стр. 912)) из 16-bit op.

FLDZ затолкнуть ноль в стек

FLD op: затолкнуть op в стек.

FMUL op: $ST(0)=ST(0)*op$

FMUL ST(i), ST(j): $ST(i)=ST(i)*ST(j)$

FMULP op: $ST(0)=ST(0)*op$; вытолкнуть один элемент из стека

FMULP ST(i), ST(j): $ST(i)=ST(i)*ST(j)$; вытолкнуть один элемент из стека

FSINCOS : tmp=ST(0); ST(1)=sin(tmp); ST(0)=cos(tmp)

FSQRT : $ST(0) = \sqrt{ST(0)}$

FSTCW op: записать FPU control word (A.3 (стр. 912)) в 16-bit op после проверки ожидающих исключений.

FNSTCW op: записать FPU control word (A.3 (стр. 912)) в 16-bit op.

FSTSW op: записать FPU status word (A.3.2 (стр. 913)) в 16-bit op после проверки ожидающих исключений.

FNSTSW op: записать FPU status word (A.3.2 (стр. 913)) в 16-bit op.

FST op: копировать ST(0) в op

FSTP op: копировать ST(0) в op; вытолкнуть один элемент из стека

FSUBR op: $ST(0)=op-ST(0)$

FSUBR ST(0), ST(i): $ST(0)=ST(i)-ST(0)$

FSUBRP ST(1)=ST(0)-ST(1); вытолкнуть один элемент из стека, таким образом, складываемые значения в стеке заменяются разностью

FSUB op: $ST(0)=ST(0)-op$

FSUB ST(0), ST(i): $ST(0)=ST(0)-ST(i)$

FSUBP ST(1)=ST(1)-ST(0); вытолкнуть один элемент из стека, таким образом, складываемые значения в стеке заменяются разностью

FUCOM ST(i): сравнить ST(0) и ST(i)

FUCOM сравнить ST(0) и ST(1)

FUCOMP сравнить ST(0) и ST(1); вытолкнуть один элемент из стека.

FUCOMPP сравнить ST(0) и ST(1); вытолкнуть два элемента из стека.

Инструкция работает так же, как и FCOM, за тем исключением что исключение срабатывает только если один из операндов SNaN, но числа QNaN нормально обрабатываются.

FXCH ST(i) обменять местами значения в ST(0) и ST(i)

FXCH обменять местами значения в ST(0) и ST(1)

А.6.5. Инструкции с печатаемым ASCII-опкодом

(В 32-битном режиме).

Это может пригодиться для создания шеллкодов. См. также: 82.1 (стр. 826).

ASCII-символ	шестнадцатеричный код	x86-инструкция
0	30	XOR
1	31	XOR
2	32	XOR
3	33	XOR
4	34	XOR
5	35	XOR
7	37	AAA
8	38	CMP
9	39	CMP
:	3a	CMP
;	3b	CMP
<	3c	CMP
=	3d	CMP

?	3f	AAS
@	40	INC
A	41	INC
B	42	INC
C	43	INC
D	44	INC
E	45	INC
F	46	INC
G	47	INC
H	48	DEC
I	49	DEC
J	4a	DEC
K	4b	DEC
L	4c	DEC
M	4d	DEC
N	4e	DEC
O	4f	DEC
P	50	PUSH
Q	51	PUSH
R	52	PUSH
S	53	PUSH
T	54	PUSH
U	55	PUSH
V	56	PUSH
W	57	PUSH
X	58	POP
Y	59	POP
Z	5a	POP
[	5b	POP
\	5c	POP
]	5d	POP
^	5e	POP
˘	5f	POP
˘	60	PUSHA
a	61	POPA
f	66	(в 32-битном режиме) переключиться на 16-битный размер операнда
g	67	(в 32-битном режиме) переключиться на 16-битный размер адреса
h	68	PUSH
i	69	IMUL
j	6a	PUSH
k	6b	IMUL
p	70	JO
q	71	JNO
r	72	JB
s	73	JAЕ
t	74	JE
u	75	JNE
v	76	JBE
w	77	JA
x	78	JS
y	79	JNS
z	7a	JP

В итоге: AAA, AAS, CMP, DEC, IMUL, INC, JA, JAE, JB, JBE, JE, JNE, JNO, JNS, JO, JP, JS, POP, POPA, PUSH, PUSHA, XOR.

Приложение В

ARM

В.1. Терминология

ARM изначально разрабатывался как 32-битный [CPU](#), поэтому *слово* здесь, в отличие от x86, 32-битное.

byte 8-бит. Для определения переменных и массива байт используется директива ассемблера DCB .

halfword 16-бит. —”—директива ассемблера DCW .

word 32-бит. —”—директива ассемблера DCD .

doubleword 64-бит.

quadword 128-бит.

В.2. Версии

- ARMv4: появился режим Thumb.
- ARMv6: использовался в iPhone 1st gen., iPhone 3G (Samsung 32-bit RISC ARM 1176JZ(F)-S поддерживающий Thumb-2)
- ARMv7: появился Thumb-2 (2003). Использовался в iPhone 3GS, iPhone 4, iPad 1st gen. (ARM Cortex-A8), iPad 2 (Cortex-A9), iPad 3rd gen.
- ARMv7s: Добавлены новые инструкции. Использовался в iPhone 5, iPhone 5c, iPad 4th gen. (Apple A6).
- ARMv8: 64-битный процессор, [AKA](#) ARM64 [AKA](#) AArch64. Использовался в iPhone 5S, iPad Air (Apple A7). В 64-битном режиме, режима Thumb больше нет, только режим ARM (4-байтные инструкции).

В.3. 32-битный ARM (AArch32)

В.3.1. Регистры общего пользования

- R0 — результат функции обычно возвращается через R0
- R1...R12 — [GPRs](#)
- R13 — [AKA](#) SP ([stack pointer](#))
- R14 — [AKA](#) LR ([link register](#))
- R15 — [AKA](#) PC (program counter)

R0 - R3 называются также «scratch registers»: аргументы функции обычно передаются через них, и эти значения не обязательно восстанавливать перед выходом из функции .

В.3.2. Current Program Status Register (CPSR)

Бит	Описание
0..4	M – processor mode
5	T – Thumb state
6	F – FIQ disable
7	I – IRQ disable
8	A – imprecise data abort disable
9	E – data endianness
10..15, 25, 26	IT – if-then state
16..19	GE – greater-than-or-equal-to
20..23	DNM – do not modify
24	J – Java state
27	Q – sticky overflow
28	V – overflow
29	C – carry/borrow/extend
30	Z – zero bit
31	N – negative/less than

В.3.3. Регистры VFP (для чисел с плавающей точкой) и NEON

0..31 ^{bits}	32..64	65..96	97..127
Q0 ^{128 bits}			
D0 ^{64 bits}		D1	
S0 ^{32 bits}	S1	S2	S3

S-регистры 32-битные, используются для хранения чисел с одинарной точностью .

D-регистры 64-битные, используются для хранения чисел с двойной точностью .

D- и S-регистры занимают одно и то же место в памяти CPU — можно обращаться к D-регистрам через S-регистры (хотя это и бессмысленно) .

Точно также, **NEON** Q-регистры имеют размер 128 бит и занимают то же физическое место в памяти CPU что и остальные регистры, предназначенные для чисел с плавающей точкой .

В VFP присутствует 32 S-регистров: S0..S31 .

В VFPv2 были добавлены 16 D-регистров, которые занимают то же место что и S0..S31 .

В VFPv3 (**NEON** или «Advanced SIMD») добавили еще 16 D-регистров, в итоге это D0..D31, но регистры D16..D31 не делят место с другими S-регистрами.

В **NEON** или «Advanced SIMD» были добавлены также 16 128-битных Q-регистров, делящих место с регистрами D0..D31.

В.4. 64-битный ARM (AArch64)**В.4.1. Регистры общего пользования**

Количество регистров было удвоено со времен AArch32.

- X0 — результат функции обычно возвращается через X0
- X0...X7 — Здесь передаются аргументы функции.
- X8
- X9...X15 — временные регистры, вызываемая функция может их использовать и не восстанавливать их.
- X16
- X17
- X18
- X19...X29 — вызываемая функция может их использовать, но должна восстанавливать их по завершению.
- X29 — используется как **FP** (как минимум в GCC)

- X30 — «Procedure Link Register» AKA LR (link register).
- X31 — регистр, всегда содержащий ноль AKA XZR или «Zero Register». Его 32-битная часть называется WZR.
- SP, больше не регистр общего пользования.

См.также: [Procedure Call Standard for the ARM 64-bit Architecture (AArch64), (2013)]¹.

32-битная часть каждого X-регистра также доступна как W-регистр (W0, W1, и т.д.).

Старшие 32 бита	младшие 32 бита
X0	
	W0

В.5. Инструкции

В ARM имеется также для некоторых инструкций суффикс -S, указывающий, что эта инструкция будет модифицировать флаги. Инструкции без этого суффикса не модифицируют флаги. Например, инструкция ADD в отличие от ADDS сложит два числа, но флаги не изменит. Такие инструкции удобно использовать между CMP где выставляются флаги и, например, инструкциями перехода, где флаги используются. Они также лучше в смысле анализа зависимостей данных (data dependency analysis) (потому что меньшее количество регистров модифицируется во время исполнения).

В.5.1. Таблица условных кодов

Код	Описание	Флаги
EQ	равно	Z == 1
NE	не равно	Z == 0
CS AKA HS (Higher or Same)	перенос / беззнаковое, больше или равно	C == 1
CC AKA LO (LOwer)	нет переноса / беззнаковое, меньше чем	C == 0
MI	минус, отрицательный знак / меньше чем	N == 1
PL	плюс, положительный знак или ноль / больше чем или равно	N == 0
VS	переполнение	V == 1
VC	нет переполнения	V == 0
HI	беззнаковое, больше чем /	C == 1 and Z == 0
LS	беззнаковое, меньше или равно /	C == 0 or Z == 1
GE	знаковое, больше чем или равно /	N == V
LT	знаковое, меньше чем /	N != V
GT	знаковое, больше чем /	Z == 0 and N == V
LE	знаковое, меньше чем или равно /	Z == 1 or N != V
None / AL	Всегда	Любые

¹Также доступно здесь: <http://go.yurichev.com/17287>

Приложение С

MIPS

С.1. Регистры

(Соглашение о вызовах O32)

С.1.1. Регистры общего пользования GPR

Номер	Псевдоимя	Описание
\$0	\$ZERO	Всегда ноль. Запись в этот регистр работает как холостая инструкция (NOP).
\$1	\$AT	Используется как временный регистр для ассемблерных макросов и псевдоинструкций.
\$2 ...\$3	\$V0 ...\$V1	Здесь возвращается результат функции.
\$4 ...\$7	\$A0 ...\$A3	Аргументы функции.
\$8 ...\$15	\$T0 ...\$T7	Используется для временных данных.
\$16 ...\$23	\$S0 ...\$S7	Используется для временных данных*.
\$24 ...\$25	\$T8 ...\$T9	Используется для временных данных.
\$26 ...\$27	\$K0 ...\$K1	Зарезервировано для ядра ОС.
\$28	\$GP	Глобальный указатель**.
\$29	\$SP	SP*.
\$30	\$FP	FP*.
\$31	\$RA	RA.
n/a	PC	PC.
n/a	HI	старшие 32 бита результата умножения или остаток от деления***.
n/a	LO	младшие 32 бита результата умножения или результат деления***.

С.1.2. Регистры для работы с числами с плавающей точкой

Название	Описание
\$F0..\$F1	Здесь возвращается результат функции.
\$F2..\$F3	Не используется.
\$F4..\$F11	Используется для временных данных.
\$F12..\$F15	Первые два аргумента функции.
\$F16..\$F19	Используется для временных данных.
\$F20..\$F31	Используется для временных данных*.

* — Callee должен сохранять.

** — Callee должен сохранять (кроме PIC-кода).

*** — доступны используя инструкции MFHI и MFLO.

С.2. Инструкции

Есть три типа инструкций:

- Тип R: имеющие 3 регистра. R-инструкции обычно имеют такой вид:


```
instruction destination, source1, source2
```

Важно помнить что если первый и второй регистр один и тот же, IDA может показать инструкцию в сокращенной форме:

```
instruction destination/source1, source2
```

Это немного напоминает Интеловский синтаксис ассемблера x86.

- Тип I: имеющие 2 регистра и 16-битное «immediate»-значение.
- Тип J: инструкции перехода, имеют 26 бит для кодирования смещения.

С.2.1. Инструкции перехода

Какая разница между инструкциями начинающихся с B- (BEQ, B, и т.д.) и с J- (JAL, JALR, и т.д.)?

B-инструкции имеют тип I, так что, смещение в этих инструкциях кодируется как 16-битное значение. Инструкции JR и JALR имеют тип R, и они делают переход по абсолютному адресу указанному в регистре. J и JAL имеют тип J, так что смещение кодируется как 26-битное значение.

Коротко говоря, в B-инструкциях можно кодировать условие (B на самом деле это псевдоинструкция для `BEQ $ZERO, $ZERO, L` а в J-инструкциях нельзя.

Приложение D

Некоторые библиотечные функции GCC

имя	значение
__divdi3	знаковое деление
__moddi3	остаток от знакового деления
__udivdi3	беззнаковое деление
__umoddi3	остаток от беззнакового деления

Приложение E

Некоторые библиотечные функции MSVC

`ll` в имени функции означает «long long», т.е. 64-битный тип данных .

имя	значение
<code>__alldiv</code>	знаковое деление
<code>__allmul</code>	умножение
<code>__allrem</code>	остаток от знакового деления
<code>__allshl</code>	сдвиг влево
<code>__allshr</code>	знаковый сдвиг вправо
<code>__aulldiv</code>	беззнаковое деление
<code>__aullrem</code>	остаток от беззнакового деления
<code>__aullshr</code>	беззнаковый сдвиг вправо

Процедуры умножения и сдвига влево, одни и те же и для знаковых чисел, и для беззнаковых, поэтому здесь только одна функция для каждой операции .

Исходные коды этих функций можно найти в установленной [MSVS](#), в `VC/crt/src/intel/*.asm` .

Приложение F

Cheatsheets

F.1. IDA

Краткий справочник горячих клавиш:

клавиша	значение
Space	переключать между листингом и просмотром кода в виде графа
C	конвертировать в код
D	конвертировать в данные
A	конвертировать в строку
*	конвертировать в массив
U	сделать неопределенным
O	сделать смещение из операнда
H	сделать десятичное число
R	сделать символ
B	сделать двоичное число
Q	сделать шестнадцатеричное число
N	переименовать идентификатор
?	калькулятор
G	переход на адрес
:	добавить комментарий
Ctrl-X	показать ссылки на текущую функцию, метку, переменную (в т.ч., в стеке)
X	показать ссылки на функцию, метку, переменную, и т.д.
Alt-I	искать константу
Ctrl-I	искать следующее вхождение константы
Alt-B	искать последовательность байт
Ctrl-B	искать следующее вхождение последовательности байт
Alt-T	искать текст (включая инструкции, и т.д.)
Ctrl-T	искать следующее вхождение текста
Alt-P	редактировать текущую функцию
Enter	перейти к функции, переменной, и т.д.
Esc	вернуться назад
Num -	свернуть функцию или отмеченную область
Num +	снова показать функцию или область

Сворачивание функции или области может быть удобно чтобы прятать те части функции, чья функция вам стала уже ясна . это используется в моем скрипте¹ для сворачивания некоторых очень часто используемых фрагментов inline-кода .

F.2. OllyDbg

Краткий справочник горячих клавиш:

¹[GitHub](#)

хот-кей	значение
F7	трассировать внутрь
F8	сделать шаг, не входя в функцию
F9	запуск
Ctrl-F2	перезапуск

F.3. MSVC

Некоторые полезные опции, которые были использованы в книге .

опция	значение
/O1	оптимизация по размеру кода
/Ob0	не заменять вызовы inline-функций их кодом
/Ox	максимальная оптимизация
/GS-	отключить проверки переполнений буфера
/Fa(file)	генерировать листинг на ассемблере
/Zi	генерировать отладочную информацию
/Zp(n)	паковать структуры по границе в n байт
/MD	выходной исполняемый файл будет использовать <code>MSVCR*.DLL</code>

Кое-какая информация о версиях MSVC: [56.1](#) (стр. [643](#)).

F.4. GCC

Некоторые полезные опции, которые были использованы в книге.

опция	значение
-Os	оптимизация по размеру кода
-O3	максимальная оптимизация
-regparm=	как много аргументов будет передаваться через регистры
-o file	задать имя выходного файла
-g	генерировать отладочную информацию в итоговом исполняемом файле
-S	генерировать листинг на ассемблере
-masm=intel	генерировать листинг в синтаксисе Intel
-fno-inline	не вставлять тело функции там, где она вызывается

F.5. GDB

Некоторые команды, которые были использованы в книге:

опция	значение
break filename.c:number	установить точку останова на номере строки в исходном файле
break function	установить точку останова на функции
break *address	установить точку останова на адресе
b	—
p variable	вывести значение переменной
run	запустить
r	—
cont	продолжить исполнение
c	—
bt	вывести стек
set disassembly-flavor intel	установить Intel-синтаксис
disas	disassemble current function
disas function	дизассемблировать функцию
disas function,+50	disassemble portion
disas \$eip,+0x10	—
disas/r	дизассемблировать с опкодами
info registers	вывести все регистры
info float	вывести FPU-регистры
info locals	вывести локальные переменные (если известны)
x/w ...	вывести память как 32-битные слова
x/w \$rdi	вывести память как 32-битные слова по адресу, на который указывает RDI
x/10w ...	вывести 10 слов памяти
x/s ...	вывести строку из памяти
x/i ...	трактовать память как код
x/10c ...	вывести 10 символов
x/b ...	вывести байты
x/h ...	вывести 16-битные полуслова
x/g ...	вывести 64-битные слова
finish	исполнять до конца функции
next	следующая инструкция (не заходить в функции)
step	следующая инструкция (заходить в функции)
set step-mode on	не использовать информацию о номерах строк при использовании команды step
frame n	переключить фрейм стека
info break	список точек останова
del n	удалить точку останова
set args ...	установить аргументы командной строки

Список принятых сокращений

ОС Операционная Система	xxi
ООП Объектно-Ориентированное Программирование	539
ЯП Язык Программирования	4
ГПСЧ Генератор псевдослучайных чисел	xiii
ПЗУ Постоянное запоминающее устройство	654
АЛУ Арифметико-логическое устройство	21
PID ID программы/процесса	747
LF Line feed (подача строки) (10 или '\n' в Си/Си++)	515
CR Carriage return (возврат каретки) (13 или '\r' в Си/Си++)	515
LIFO Last In First Out (последним вошел, первым вышел)	26
RA Адрес возврата	6
PE Portable Executable	696
SP stack pointer . SP/ESP/RSP в x86/x64. SP в ARM.	15
DLL Dynamic-link library	696
PC Program Counter. IP/EIP/RIP в x86/64. PC в ARM.	15
LR Link Register	6
IDA Интерактивный дизассемблер и отладчик, разработан Hex-Rays	7
IAT Import Address Table	697
INT Import Name Table	697
RVA Relative Virtual Address	697
VA Virtual Address	697
OEP Original Entry Point	684
MSVC Microsoft Visual C++	
MSVS Microsoft Visual Studio	922
ASLR Address Space Layout Randomization	697
MFC Microsoft Foundation Classes	700

TLS Thread Local Storage	xix
AKA Also Known As - (Также известный как)	26
CRT C runtime library	9
CPU Central processing unit	xxi
FPU Floating-point unit.....	viii
CISC Complex instruction set computing	15
RISC Reduced instruction set computing	4
GUI Graphical user interface.....	693
RTTI Run-time type information.....	554
BSS Block Started by Symbol.....	20
SIMD Single instruction, multiple data	191
BSOD Blue Screen of Death	686
ISA Instruction Set Architecture (Архитектура набора команд)	iv
HPC High-Performance Computing	507
SEH Structured Exception Handling	31
ELF Формат исполняемых файлов, использующийся в Linux и некоторых других *NIX	xix
TIB Thread Information Block	280
PIC Position Independent Code: 68.1 (стр. 688)	xix
NAN Not a Number	914
NOP No OPeration	22
BEQ (PowerPC, ARM) Branch if Equal	90
BNE (PowerPC, ARM) Branch if Not Equal	204
BLR (PowerPC) Branch to Link Register	749
XOR eXclusive OR (исключающее «ИЛИ»)	920
MCU Microcontroller unit	481
RAM Random-access memory	418

EGA Enhanced Graphics Adapter	898
VGA Video Graphics Array	898
API Application programming interface	646
ASCII American Standard Code for Information Interchange	489
ASCIIZ ASCII Zero (ASCII-строка заканчивающаяся нулем)	88
IA64 Intel Architecture 64 (Itanium): 95 (стр. 895)	449
EPIC Explicitly parallel instruction computing	895
OOE Out-of-order execution	451
MSB Most significant bit/byte (самый старший бит/байт)	312
LSB Least significant bit/byte (самый младший бит/байт)	
STL (Си++) Standard Template Library: 52.4 (стр. 556)	560
PODT (Си++) Plain Old Data Type	571
HDD Hard disk drive	583
VM Virtual Memory (виртуальная память)	
WRK Windows Research Kernel	660
GPR General Purpose Registers (регистры общего пользования)	xviii
SSDT System Service Dispatch Table	686
RE Reverse Engineering	904
BCD Binary-coded decimal	829
BOM Byte order mark	650
GDB GNU debugger	42
FP Frame Pointer	19
MBR Master Boot Record	654
JPE Jump Parity Even (инструкция x86)	235
CIDR Classless Inter-Domain Routing	469
STMFD Store Multiple Full Descending (инструкция ARM)	

LDMFD Load Multiple Full Descending (инструкция ARM)

STMED Store Multiple Empty Descending (инструкция ARM)	26
LDMED Load Multiple Empty Descending (инструкция ARM)	26
STMFA Store Multiple Full Ascending (инструкция ARM)	26
LDMFA Load Multiple Full Ascending (инструкция ARM)	26
STMEA Store Multiple Empty Ascending (инструкция ARM)	26
LDMEA Load Multiple Empty Ascending (инструкция ARM)	26
APSR (ARM) Application Program Status Register	258
FPSCR (ARM) Floating-Point Status and Control Register	258
RFC Request for Comments	652
TOS Top Of Stack (вершина стека)	605
LVA (Java) Local Variable Array (массив локальных переменных)	611
JVM Java virtual machine	xiii
JIT Just-in-time compilation	604
CDFS Compact Disc File System	xiv
CD Compact Disc	
ADC Analog-to-digital converter	663
EOF End of file (конец файла)	81

Glossary

вещественное число числа, которые могут иметь точку. в Си/Си++ это *float* и *double* . [212](#)

декремент Уменьшение на 1. [181](#), [198](#), [439](#), [916](#), [918](#), [923](#)

инкремент Увеличение на 1. [181](#), [185](#), [187](#), [198](#), [787](#), [916](#)

интегральный тип данных обычные числа, но не вещественные. могут использоваться для передачи булевых типов и перечислений (enumerations) . [228](#)

произведение Результат умножения. [94](#), [223](#), [404](#), [430](#), [486](#)

среднее арифметическое сумма всех значений, разделенная на их количество. [510](#)

указатель стека Регистр указывающий на место в стеке . [9](#), [10](#), [15](#), [26](#), [29](#), [37](#), [49](#), [50](#), [69](#), [95](#), [593](#), [672–675](#), [911](#), [917](#), [927](#), [938](#)

хвостовая рекурсия Это когда компилятор или интерпретатор превращает рекурсию (с которой возможно это проделать, т.е. *хвостовую*) в итерацию для эффективности : [wikipedia](#). [465](#)

частное Результат деления. [212](#), [216](#), [218](#), [219](#), [223](#), [429](#), [483](#), [511](#)

anti-pattern Нечто широко известное как плохое решение . [28](#), [71](#), [450](#)

atomic operation «*ατομος*» означает «неделимый» в греческом языке, так что атомарная операция — это операция которая гарантированно не будет прервана другими тредами . [726](#), [892](#)

basic block группа инструкций, не имеющая инструкций переходов, а также не имеющая переходов в середину блока извне. В [IDA](#) он выглядит как просто список инструкций без строк-разрывов . [633](#), [899](#), [900](#)

callee Вызываемая функция. [62](#), [95](#), [98](#), [450](#), [674](#), [677](#), [678](#), [930](#)

caller Функция вызывающая другую функцию. [7](#), [41](#), [97](#), [450](#), [672](#), [678](#)

compiler intrinsic Специфичная для компилятора функция не являющаяся обычной библиотечной функцией. Компилятор вместо её вызова генерирует определенный машинный код. Нередко, это псевдофункции для определенной инструкции [CPU](#). Читайте больше: ([92](#) (стр. [887](#))). [922](#)

CP/M Control Program for Microcomputers: очень простая дисковая [ОС](#) использовавшаяся перед [MS-DOS](#). [827](#)

dongle Небольшое устройство подключаемое к LPT-порту для принтера (в прошлом) или к USB . Исполняло функции security token-а, имела память и, иногда, секретную (крипто-)хеширующую функцию. [748](#)

endianness Порядок байт: [32](#) (стр. [448](#)). [17](#), [74](#), [342](#), [920](#)

GiB Гибибайт: 2^{30} или 1024 мегабайт или 1073741824 байт . [13](#)

heap (куча) обычно, большой кусок памяти предоставляемый [ОС](#), так что прикладное ПО может делить его как захочет. [malloc\(\)](#)/[free\(\)](#) работают с кучей . [26](#), [344](#), [557](#), [559](#), [572](#), [573](#), [695](#), [696](#)

jump offset Часть опкода JMP или Jcc инструкции, просто прибавляется к адресу следующей инструкции, и так вычисляется новый [PC](#). Может быть отрицательным. [129](#), [916](#)

leaf function Функция не вызывающая больше никаких функций . [23](#), [28](#)

link register (RISC) Регистр в котором обычно записан адрес возврата. Это позволяет вызывать leaf-функции без использования стека, т.е. быстрее . [28](#), [749](#), [927](#), [929](#)

loop unwinding Это когда вместо организации цикла на n итераций, компилятор генерирует n копий тела цикла, для экономии на инструкциях, обеспечивающих сам цикл . [183](#)

name mangling применяется как минимум в Си++, где компилятору нужно закодировать имя класса, метода и типы аргументов в одной строке, которая будет внутренним именем функции. читайте также здесь : [52.1.1](#) (стр. [539](#)). [539](#), [643](#), [644](#)

NaN не число: специальные случаи чисел с плавающей запятой, обычно сигнализирующие об ошибках . [231](#), [253](#), [897](#)

NEON AKA «Advanced SIMD» — SIMD от ARM. [928](#)

NOP «no operation», холостая инструкция. [670](#)

NTAPI API доступное только в линии Windows NT. Большею частью не документировано Microsoft-ом. [735](#)

PDB (Win32) Файл с отладочной информацией, обычно просто имена функций, но иногда имена аргументов функций и локальных переменных . [642](#), [698](#), [735](#), [736](#), [743](#), [747](#), [808](#), [809](#)

POKE Инструкция языка BASIC записывающая байт по определенному адресу . [670](#)

register allocator Функция компилятора распределяющая локальные переменные по регистрам процессора . [197](#), [302](#), [418](#)

reverse engineering процесс понимания как устроена некая вещь, иногда, с целью клонирования оной . [v](#), [922](#)

security cookie Случайное значение, разное при каждом исполнении. Читайте больше об этом тут : [19.3](#) (стр. [279](#)). [716](#)

stack frame Часть стека, в которой хранится информация, связанная с текущей функцией: локальные переменные, аргументы функции, [RA](#), и т.д.. [63](#), [94](#), [462](#), [716](#)

stdout standard output. [17](#), [30](#), [151](#)

thunk function Крохотная функция делающая только одно: вызывающая другую функцию . [18](#), [388](#), [749](#), [757](#)

tracer Моя простейшая утилита для отладки. Читайте больше об этом тут : [71.3](#) (стр. [729](#)). [185](#)–[187](#), [647](#), [657](#), [661](#), [712](#), [721](#), [810](#), [817](#), [821](#), [822](#), [824](#), [886](#)

user mode Режим CPU с ограниченными возможностями в котором он исполняет прикладное ПО. ср. [kernel mode](#). [765](#)

Windows NT Windows NT, 2000, XP, Vista, 7, 8. [289](#), [414](#), [592](#), [650](#), [686](#), [697](#), [725](#), [830](#), [922](#)

word (слово) тип данных помещающийся в [GPR](#). В компьютерах старше персональных, память часто измерялась не в байтах, а в словах. [562](#)

xoring нередко применяемое в английском языке, означает применение операции XOR . [716](#), [760](#), [763](#)

Предметный указатель

.NET, 702

Ada, 102

Alpha AXP, 5

AMD, 677

Angry Birds, 259

Apollo Guidance Computer, 206

ARM, 204, 531, 748, 927

armel, 223

armhf, 223

Condition codes, 132

D-регистры, 222, 928

Data processing instructions, 485

DCB, 15

hard float, 223

if-then block, 259

Leaf function, 28

Optional operators

ASR, 328, 485

LSL, 268, 295, 328, 441

LSR, 328, 485

ROR, 328

RRX, 328

S-регистры, 222, 928

soft float, 223

Инструкции

ADC, 395

ADD, 16, 101, 132, 188, 316, 328, 485, 929

ADDAL, 132

ADDCC, 170

ADDS, 99, 395, 929

ADR, 15, 132

ADRcc, 132, 159, 451

ADRP/ADD pair, 19, 50, 78, 286, 298, 441

ANDcc, 526

ASR, 331

ASRS, 310, 485

B, 49, 132, 133

Bcc, 91, 92, 144

BCS, 133, 261

BEQ, 90, 159

BGE, 133

BIC, 310, 315, 333

BL, 15–19, 132, 442

BLcc, 132

BLE, 133

BLS, 133

BLT, 188

BLX, 17

BNE, 133

BX, 99, 171

CMP, 90, 91, 132, 159, 170, 188, 328, 929

CSEL, 141, 146, 148, 328

EOR, 315

FCMPE, 261

FCSEL, 261

FMOV, 441

FMRS, 316

FSTP, 242

IT, 148, 259, 282

LDMccFD, 132

LDMEA, 26

LDMED, 26

LDMFA, 26

LDMFD, 15, 26, 132

LDP, 20

LDR, 51, 69, 77, 268, 285, 439

LDR.W, 295

LDRB, 360

LDRB.W, 204

LDRSB, 204

LSL, 328, 331

LSL.W, 328

LSLR, 526

LSLS, 269, 315, 526

LSR, 331

LSRS, 315

MADD, 100

MLA, 99

MOV, 6, 15, 16, 328, 484

MOVcc, 144, 148

MOVK, 441

MOVT, 16, 484

MOVT.W, 17

MOVW, 17

MUL, 101

MULS, 99

MVNS, 204

NEG, 495

ORR, 310

POP, 14–16, 26, 28

PUSH, 16, 26, 28

RET, 20

RSB, 138, 295, 328, 495

SBC, 395

SMMUL, 485

STMEA, 26

STMED, 26

STMFA, 26, 52

STMFD, 14, 26

STMIA, 51

STMIB, 52

STP, 19, 50

STR, 50, 268

SUB, 50, 295, 328

SUBcc, 526

- SUBEQ, [205](#)
- SUBS, [395](#)
- SXTB, [361](#)
- SXTW, [298](#)
- TEST, [197](#)
- TST, [303](#), [328](#)
- VADD, [223](#)
- VDIV, [223](#)
- VLDR, [223](#)
- VMOV, [223](#), [258](#)
- VMOVGT, [258](#)
- VMRS, [258](#)
- VMUL, [223](#)
- XOR, [138](#), [316](#)
- Конвейер, [170](#)
- Переключение режимов, [99](#), [171](#)
- Регистры
 - APSR, [258](#)
 - FPSCR, [258](#)
 - Link Register, [15](#), [28](#), [49](#), [172](#), [927](#)
 - R0, [103](#), [927](#)
 - scratch registers, [204](#), [927](#)
 - X0, [928](#)
 - Z, [90](#), [928](#)
- Режим ARM, [4](#)
- Режим Thumb-2, [4](#), [171](#), [258](#), [260](#)
- Режим Thumb, [4](#), [133](#), [171](#)
- Режимы адресации, [439](#)
- переключение режимов, [17](#)
- ARM64
 - lo12, [50](#)
- ASLR, [697](#)
- AWK, [659](#)
- Base64, [652](#)
- base64, [652](#)
- bash, [104](#)
- BASIC
 - POKE, [670](#)
- binary grep, [656](#), [732](#)
- BIND.EXE, [701](#)
- binutils, [376](#)
- Bitcoin, [889](#)
- Borland C++Builder, [644](#)
- Borland Delphi, [644](#), [648](#), [886](#)
- BSoD, [686](#)
- BSS, [698](#)
- C11, [680](#)
- Callbacks, [380](#)
- Canary, [279](#)
- cdecl, [37](#), [672](#)
- COFF, [755](#)
- column-major order, [289](#)
- Compiler intrinsic, [30](#), [887](#)
- CRC32, [452](#), [466](#)
- CRT, [693](#), [712](#)
- Cygwin, [643](#), [647](#), [702](#), [730](#)
- DES, [403](#), [418](#)
- dlopen(), [691](#)
- dlsym(), [691](#)
- DOSBox, [830](#)
- DosBox, [661](#)
- double, [215](#), [677](#)
- Doubly linked list, [562](#)
- dtruss, [730](#)
- Duff's device, [480](#)
- EICAR, [826](#)
- ELF, [75](#)
- Error messages, [652](#)
- fastcall, [12](#), [60](#), [301](#), [673](#)
- float, [215](#), [677](#)
- Forth, [625](#)
- FORTTRAN, [289](#), [505](#), [589](#), [644](#)
- FreeBSD, [655](#)
- Function epilogue, [25](#), [49](#), [51](#), [132](#), [360](#), [659](#)
- Function prologue, [10](#), [25](#), [28](#), [50](#), [279](#), [659](#)
- Fused multiply-add, [99](#), [100](#)
- Fuzzing, [495](#)
- GCC, [643](#), [932](#), [935](#)
- GDB, [23](#), [42](#), [46](#), [278](#), [388](#), [389](#), [729](#), [935](#)
- Glibc, [388](#), [686](#)
- HASP, [655](#)
- Hiew, [88](#), [129](#), [648](#), [699](#), [702](#), [886](#)
- IDA, [82](#), [376](#), [504](#), [636](#), [650](#), [873](#), [934](#)
 - var ?, [51](#), [69](#)
- IEEE 754, [214](#), [312](#), [372](#), [425](#), [908](#)
- Inline code, [189](#), [309](#), [496](#), [545](#), [575](#)
- Integer overflow, [102](#)
- Intel
 - 8080, [204](#)
 - 8086, [204](#), [309](#), [764](#)
 - Модель памяти, [599](#), [898](#)
 - 8253, [829](#)
 - 80286, [764](#), [898](#)
 - 80386, [309](#), [898](#)
 - 80486, [214](#)
 - FPU, [214](#)
- Intel C++, [9](#), [404](#), [888](#), [899](#), [917](#)
- iPod/iPhone/iPad, [14](#)
- Itanium, [895](#)
- Java, [604](#)
- jumptable, [164](#), [171](#)
- Keil, [14](#)
- kernel panic, [686](#)
- kernel space, [686](#)
- LD_PRELOAD, [690](#)
- Linux, [302](#), [688](#), [813](#)
 - libc.so.6, [301](#), [388](#)
- LLVM, [14](#)
- long double, [215](#)
- Loop unwinding, [183](#)
- Mac OS Classic, [748](#)
- Mac OS X, [730](#)
- MD5, [452](#), [654](#)
- MFC, [700](#), [800](#)
- MIDI, [654](#)
- MinGW, [643](#)
- minifloat, [441](#)
- MIPS, [5](#), [532](#), [664](#), [698](#), [748](#)
 - Branch delay slot, [7](#)

- Global Pointer, [295](#)
- Load delay slot, [162](#)
- O32, [56](#), [60](#), [61](#), [930](#)
- Глобальный указатель, [20](#)
- Инструкции
 - ADD, [102](#)
 - ADD.D, [225](#)
 - ADDIU, [21](#), [80](#), [81](#)
 - ADDU, [102](#)
 - AND, [311](#)
 - BC1F, [262](#)
 - BC1T, [262](#)
 - BEQ, [92](#), [134](#)
 - BLTZ, [139](#)
 - BNE, [134](#)
 - BNEZ, [173](#)
 - BREAK, [485](#)
 - C.LT.D, [262](#)
 - DIV.D, [225](#)
 - J, [7](#), [21](#)
 - JAL, [102](#)
 - JALR, [21](#), [102](#)
 - JR, [162](#)
 - LB, [193](#)
 - LBU, [193](#)
 - LI, [443](#)
 - LUI, [21](#), [80](#), [81](#), [226](#), [314](#), [443](#)
 - LW, [21](#), [70](#), [81](#), [162](#)
 - LWC1, [225](#)
 - MFC1, [229](#)
 - MFHI, [102](#), [485](#), [930](#)
 - MFLO, [102](#), [485](#), [930](#)
 - MTC1, [378](#)
 - MUL.D, [225](#)
 - MULT, [102](#)
 - NOR, [206](#)
 - OR, [23](#)
 - ORI, [311](#), [443](#)
 - SB, [193](#)
 - SLL, [173](#), [208](#), [330](#)
 - SLLV, [330](#)
 - SLT, [134](#)
 - SLTIU, [173](#)
 - SLTU, [134](#), [136](#), [173](#)
 - SRL, [212](#)
 - SUBU, [139](#)
 - SW, [56](#)
- Псевдоинструкции
 - B, [191](#)
 - BEQZ, [136](#)
 - L.D, [225](#)
 - LA, [23](#)
 - LI, [7](#)
 - MOVE, [21](#), [79](#)
 - NEGU, [139](#)
 - NOP, [23](#), [79](#)
 - NOT, [206](#)
- Регистры
 - FCCR, [262](#)
 - HI, [485](#)
 - LO, [485](#)
- MS-DOS, [280](#), [596](#), [654](#), [661](#), [670](#), [696](#), [764](#), [826](#), [828](#), [859](#), [886](#), [898](#), [908](#), [918](#), [922](#), [923](#)
- DOS extenders, [898](#)
- MSVC, [933](#), [935](#)
- Name mangling, [539](#)
- Native API, [697](#)
- NEC V20, [830](#)
- objdump, [376](#), [690](#), [702](#)
- OEP, [696](#), [702](#)
- OllyDbg, [39](#), [65](#), [74](#), [94](#), [107](#), [123](#), [165](#), [185](#), [199](#), [217](#), [232](#), [243](#), [266](#), [273](#), [276](#), [290](#), [319](#), [342](#), [359](#), [360](#), [364](#), [367](#), [383](#), [699](#), [729](#), [934](#)
- opaque predicate, [535](#)
- OpenMP, [645](#), [889](#)
- OpenWatcom, [644](#), [674](#)
- Oracle RDBMS, [9](#), [403](#), [652](#), [704](#), [813](#), [820](#), [822](#), [866](#), [876](#), [888](#), [899](#)
- Page (memory), [414](#)
- Pascal, [648](#)
- PDP-11, [439](#)
- PowerPC, [5](#), [20](#), [748](#)
- puts() вместо printf(), [17](#), [67](#), [103](#), [130](#)
- Quake III Arena, [379](#)
- Raspberry Pi, [14](#)
- ReactOS, [710](#)
- Register allocation, [418](#)
- Relocation, [18](#)
- row-major order, [289](#)
- RVA, [697](#)
- SAP, [642](#), [808](#)
- SCO OpenServer, [755](#)
- Scratch space, [676](#)
- Security cookie, [279](#), [716](#)
- Security through obscurity, [652](#)
- SHA1, [452](#)
- SHA512, [889](#)
- Shadow space, [96](#), [98](#), [426](#)
- Shellcode, [534](#), [686](#), [697](#), [827](#), [925](#)
- Signed numbers, [122](#), [446](#)
- SIMD, [425](#), [503](#)
- SSE, [425](#)
- SSE2, [425](#)
- stdcall, [672](#), [886](#)
- strace, [690](#), [730](#)
- syscall, [301](#), [686](#), [730](#)
- TCP/IP, [449](#)
- thiscall, [539](#), [541](#), [674](#)
- thunk-функции, [18](#), [701](#), [749](#), [757](#)
- TLS, [280](#), [680](#), [698](#), [702](#), [911](#)
 - Callbacks, [702](#)
 - Коллбэки, [683](#)
- tracer, [185](#), [385](#), [387](#), [647](#), [657](#), [661](#), [712](#), [721](#), [729](#), [810](#), [817](#), [821–823](#), [886](#)
- UFS2, [655](#)
- Unicode, [649](#)
- Unrolled loop, [189](#), [282](#), [500](#)
- uptime, [690](#)
- USB, [750](#)
- user space, [686](#)
- UTF-16LE, [649](#), [650](#)
- UTF-8, [649](#)

VA, 697

Watcom, 644

Windows, 725

API, 908

IAT, 697

INT, 697

KERNEL32.DLL, 300

MSVCR80.DLL, 382

NTAPI, 735

ntoskrnl.exe, 813

PDB, 642, 698, 735, 743, 808

Structured Exception Handling, 31, 703

TIB, 280, 703, 911

Win32, 300, 650, 690, 696, 898

GetProcAddress, 701

LoadLibrary, 701

Ordinal, 699

RaiseException(), 703

SetUnhandledExceptionFilter(), 704

Windows 2000, 698

Windows 3.x, 592, 898

Windows NT4, 698

Windows Vista, 696, 735

Windows XP, 698, 702, 743

Wine, 710

Wolfram Mathematica, 487, 488, 853

x86

AVX, 403

FPU, 912

MMX, 403

SSE, 403

SSE2, 403

Инструкции

AAA, 926

AAS, 926

ADC, 394, 596, 916

ADD, 9, 37, 94, 490, 596, 916

ADDSD, 426

ADDSS, 438

ADRCc, 140

AND, 10, 301, 304, 318, 331, 366, 916, 920

BSF, 416, 920

BSR, 920

BSWAP, 449, 920

BT, 920

BTC, 313, 920

BTR, 313, 726, 920

BTS, 313, 920

CALL, 9, 27, 529, 701, 916

CBW, 447, 920

CDQ, 401, 447, 920

CDQE, 447, 920

CLD, 920

CLI, 920

CMC, 920

CMOVcc, 133, 140, 142, 144, 148, 451, 920

CMP, 82, 916, 926

CMPSB, 655, 920

CMPSD, 920

CMPSQ, 920

CMPSW, 920

COMISD, 434

COMISS, 438

CUID, 364, 922

CWD, 447, 596, 839, 920

CWDE, 447, 920

DEC, 198, 916, 926

DIV, 447, 922

DIVSD, 426, 658

FABS, 924

FADD, 924

FADDP, 216, 222, 924

FATRET, 326, 327

FCHS, 924

FCMOVcc, 255

FCOM, 242, 253, 924

FCOMP, 230, 924

FCOMPP, 924

FDIV, 216, 657, 924

FDIVP, 216, 924

FDIVR, 222, 924

FDIVRP, 924

FILD, 924

FIST, 924

FISTP, 924

FLD, 226, 230, 925

FLD1, 924

FLDCW, 924

FLDZ, 924

FMUL, 216, 925

FMULP, 925

FNSTCW, 925

FNSTSW, 230, 253, 925

FSINCOS, 925

FSQRT, 925

FST, 925

FSTCW, 925

FSTP, 226, 925

FSTSW, 925

FSUB, 925

FSUBP, 925

FSUBR, 925

FSUBRP, 925

FUCOM, 253, 925

FUCOMI, 255

FUCOMP, 925

FUCOMPP, 253, 925

FWAIT, 214

FXCH, 925

IDIV, 447, 483, 922

IMUL, 94, 297, 447, 916, 926

IN, 529, 764, 829, 922

INC, 198, 886, 916, 926

INT, 827, 922

INT3, 647

IRET, 922, 923

JA, 122, 254, 446, 916, 926

JAE, 121, 916, 926

JB, 122, 446, 916, 926

JBE, 121, 916, 926

JC, 916

Jcc, 92, 143

JCXZ, 916

JE, 151, 916, 926

JECXZ, 916

JG, 122, 446, 916

JGE, 121, 916

JL, 122, 446, 916
 JLE, 121, 916
 JMP, 27, 49, 701, 886, 916
 JNA, 916
 JNAE, 916
 JNB, 916
 JNBE, 254, 916
 JNC, 916
 JNE, 82, 121, 916, 926
 JNG, 916
 JNGE, 916
 JNL, 916
 JNLE, 916
 JNO, 916, 926
 JNS, 916, 926
 JNZ, 916
 JO, 916, 926
 JP, 231, 830, 916, 926
 JPO, 916
 JRCXZ, 916
 JS, 916, 926
 JZ, 90, 151, 888, 916
 LAHF, 917
 LEA, 64, 96, 347, 451, 455, 468, 490, 676, 738, 917
 LEAVE, 10, 917
 LES, 769, 838
 LOCK, 725
 LODSB, 829
 LOOP, 181, 195, 659, 838, 923
 MAXSD, 434
 MOV, 6, 9, 11, 500, 501, 529, 699, 886, 918
 MOVDQA, 407
 MOVDQU, 407
 MOVSB, 918
 MOVSD, 433, 502, 785, 918
 MOVSDX, 433
 MOVSQ, 918
 MOVSS, 438
 MOVSW, 918
 MOVSX, 197, 204, 359–361, 447, 918
 MOVXSD, 284
 MOVZX, 198, 344, 748, 918
 MUL, 447, 918
 MULSD, 426
 NEG, 494, 918
 NOP, 468, 884, 886, 918
 NOT, 202, 204, 789, 918
 OR, 304, 516, 918
 OUT, 529, 764, 923
 PADDD, 407
 PCMPEQB, 415
 PLMULHW, 404
 PLMULLD, 404
 PMOVMSKB, 415
 POP, 9, 26, 27, 918, 926
 POPA, 923, 926
 POPCNT, 923
 POPF, 829, 923
 PUSH, 9, 10, 26, 27, 63, 529, 918, 926
 PUSHA, 923, 926
 PUSHF, 923
 PXOR, 415
 RCL, 659, 923
 RCR, 923

RET, 6, 9, 27, 279, 541, 593, 886, 918
 ROL, 326, 887, 923
 ROR, 887, 923
 SAHF, 253, 918
 SAL, 923
 SALC, 830
 SAR, 331, 447, 508, 838, 924
 SBB, 394, 918
 SCASB, 829, 830, 919
 SCASD, 919
 SCASQ, 919
 SCASW, 919
 SETALC, 830
 SETcc, 134, 198, 254, 924
 SHL, 208, 265, 331, 919
 SHR, 212, 331, 366, 919
 SHRD, 400, 919
 STC, 924
 STD, 924
 STI, 924
 STOSB, 482, 919
 STOSD, 919
 STOSQ, 500, 919
 STOSW, 919
 SUB, 9, 10, 82, 151, 490, 916, 920
 SYSCALL, 922, 924
 SYSENTER, 687, 922, 924
 TEST, 197, 301, 303, 331, 920
 UD2, 924
 XADD, 726
 XCHG, 918, 920
 XOR, 9, 82, 202, 508, 659, 760, 886, 920, 926

Префиксы

LOCK, 726, 915
 REP, 915, 918, 919
 REPE/REPNE, 915
 REPNE, 919

Регистры

AH, 917, 918
 CS, 898
 DR6, 914
 DR7, 914
 DS, 898
 EAX, 82, 103
 EBP, 63, 94
 ECX, 539
 ES, 838, 898
 ESP, 37, 63
 FS, 681
 GS, 280, 681, 684
 JMP, 169
 RIP, 690
 SS, 898
 ZF, 82, 301
 Флаги, 82, 123, 912

Флаги

CF, 916, 918, 920, 923, 924
 DF, 920, 924
 IF, 920, 924

x86-64, 12, 45, 62, 68, 89, 95, 417, 425, 530, 675, 690, 908, 914

Xcode, 14

Аномалии компиляторов, 143, 226, 297, 310, 327, 477, 521,

- 888
- Базовый адрес, 697
- Глобальные переменные, 71
- Двоичное дерево, 579
- Динамически подгружаемые библиотеки, 18
- Использование gperf, 187, 260, 642, 657, 661, 810
- Компоновщик, 77, 539
- Конвейер RISC, 132
- Не-числа (NaNs), 253
- ОЗУ, 76
- ООП
- Полиморфизм, 539
- Обратная польская запись, 263
- ПЗУ, 76, 77
- Переполнение буфера, 271, 278, 716
- Режим Thumb-2, 17
- Рекурсия, 25, 27, 465
- Tail recursion, 465
- Сборщик мусора, 626
- Си++, 812
- C++11, 572, 680
- ostream, 554
- References, 555
- RTTI, 554
- STL, 642
- std::forward_list, 571
- std::list, 562
- std::map, 579
- std::set, 579
- std::string, 556
- std::vector, 571
- исключения, 707
- Синтаксис AT&T, 11, 31
- Синтаксис Intel, 11, 14
- Синтаксический сахар, 150
- Стандартная библиотека Си
- alloca(), 29, 282, 450, 707
- assert(), 288, 653
- atexit(), 562
- atoi(), 489, 799
- calloc(), 780
- close(), 691
- exit(), 454
- free(), 450
- fseek(), 780
- ftell(), 780
- getenv(), 800
- localtime(), 601
- localtime_r(), 351
- longjmp(), 151
- malloc(), 344, 450
- memchr(), 919
- memcmp(), 503, 655, 920
- memcpy(), 11, 62, 501, 918
- memset(), 500, 821, 919
- open(), 691
- pow(), 226
- puts(), 17
- qsort(), 380
- rand(), 334, 646, 740, 743, 768
- read(), 691
- realloc(), 450
- scanf(), 62
- strcat(), 504
- strcmp(), 497, 691
- strcpy(), 11, 499, 769
- strlen(), 196, 413, 499, 516, 919
- time(), 601
- tolower(), 786
- toupper(), 524
- va_arg, 510
- va_list, 513
- vprintf, 513
- Стек, 26, 93, 151
- Переполнение стека, 27
- Стековый фрейм, 63
- Хеш-функции, 452
- Элементы языка Си
- C99, 105
- bool, 300
- restrict, 505
- variable length arrays, 282
- const, 8, 77
- for, 181, 467
- if, 120, 150
- return, 9, 82, 104
- switch, 149, 150, 159
- while, 196
- Пост-декремент, 439
- Пост-инкремент, 439
- Пре-декремент, 439
- Пре-инкремент, 439
- Указатели, 62, 69, 106, 380, 417
- Энтропия, 853
- адресно-независимый код, 15, 688