

# Reverse Engineering for Beginners



Dennis Yurichev

---

# Reverse Engineering for Beginners

Dennis Yurichev  
<dennis(a)yurichev.com>



©2013-2015, Dennis Yurichev.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/>.

Text version (March 18, 2016).

The latest version (and Russian edition) of this text is accessible at [beginners.re](http://beginners.re). An e-book reader version is also available.

There is also a LITE-version (introductory short version), intended for those who want a very quick introduction to the basics of reverse engineering: [beginners.re](http://beginners.re)

You can also follow me on twitter to get information about updates of this text: @yurichev<sup>1</sup>, or subscribe to the mailing list<sup>2</sup>.

The cover was made by Andy Nechaevsky: [facebook](https://www.facebook.com/andynechaevsky).

---

<sup>1</sup>[twitter.com/yurichev](https://twitter.com/yurichev)

<sup>2</sup>[yurichev.com](http://yurichev.com)

---

# Call for translators!

You may want to help me with translation this work into languages other than English and Russian.

For those who are not afraid of TeX: [read here](#). For those who afraid, you may just open PDF file in OpenOffice and gradually rewrite each sentence. I'll cypypaste your work back to my LaTeX source code.

It's a tedious and boring work, so you probably may want to start with shortened [LITE version](#). There is even a better way: to my own experience, you can gain your motivation by translating short pieces of my book and posting them to your blog(s). I can publish URLs to these your posts here and also in my twitter ([@yurichev](#)).

Speed isn't important, because this is open-source project, after all. Your name will be mentioned as project contributor.

Korean, Chinese and Persian languages are reserved by publishers. As of March 2016, there are Brazilian Portuguese and German language teams working, drop me email, so I will connect you to them. All other attempts to translate pieces of these texts to other languages are highly welcomed!

English and Russian versions I do by myself, but my English is still that horrible, so I'm very grateful for any notes about grammar, etc. Even my Russian is also flawed, so I'm grateful for notes about Russian text as well!

So do not hesitate to contact me: `dennis(a)yurichev.com`.

# Abridged contents

I	Code patterns	1
II	Important fundamentals	430
III	Slightly more advanced examples	439
IV	Java	586
V	Finding important/interesting stuff in the code	624
VI	OS-specific	647
VII	Tools	701
VIII	Examples of real-world RE <sup>3</sup> tasks	707
IX	Examples of reversing proprietary file formats	821
X	Other things	852
XI	Books/blogs worth reading	870
	Afterword	875
	Appendix	877
	Acronyms used	907

---

<sup>3</sup>Reverse Engineering

# Contents

<b>I</b>	<b>Code patterns</b>	<b>1</b>
<b>1</b>	<b>A short introduction to the CPU</b>	<b>3</b>
1.1	A couple of words about different ISA <sup>4</sup> s	3
<b>2</b>	<b>The simplest Function</b>	<b>5</b>
2.1	x86	5
2.2	ARM	5
2.3	MIPS	6
2.3.1	A note about MIPS instruction/register names	6
<b>3</b>	<b>Hello, world!</b>	<b>7</b>
3.1	x86	7
3.1.1	MSVC	7
3.1.2	GCC	8
3.1.3	GCC: AT&T syntax	9
3.2	x86-64	10
3.2.1	MSVC—x86-64	10
3.2.2	GCC—x86-64	11
3.3	GCC—one more thing	12
3.4	ARM	12
3.4.1	Non-optimizing Keil 6/2013 (ARM mode)	13
3.4.2	Non-optimizing Keil 6/2013 (Thumb mode)	14
3.4.3	Optimizing Xcode 4.6.3 (LLVM) (ARM mode)	14
3.4.4	Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)	15
3.4.5	ARM64	17
3.5	MIPS	18
3.5.1	A word about the “global pointer”	18
3.5.2	Optimizing GCC	18
3.5.3	Non-optimizing GCC	20
3.5.4	Role of the stack frame in this example	21
3.5.5	Optimizing GCC: load it into GDB	21
3.6	Conclusion	22
3.7	Exercises	22
<b>4</b>	<b>Function prologue and epilogue</b>	<b>23</b>
4.1	Recursion	23
<b>5</b>	<b>Stack</b>	<b>24</b>
5.1	Why does the stack grow backwards?	24
5.2	What is the stack used for?	25
5.2.1	Save the function’s return address	25
5.2.2	Passing function arguments	26
5.2.3	Local variable storage	27
5.2.4	x86: <code>alloca()</code> function	27
5.2.5	(Windows) SEH	29
5.2.6	Buffer overflow protection	29
5.2.7	Automatic deallocation of data in stack	29
5.3	A typical stack layout	29
5.4	Noise in stack	29

---

<sup>4</sup>Instruction Set Architecture

5.4.1	MSVC 2013	33
5.5	Exercises	34
<b>6</b>	<b>printf() with several arguments</b>	<b>35</b>
6.1	x86	35
6.1.1	x86: 3 arguments	35
6.1.2	x64: 8 arguments	43
6.2	ARM	46
6.2.1	ARM: 3 arguments	46
6.2.2	ARM: 8 arguments	47
6.3	MIPS	51
6.3.1	3 arguments	51
6.3.2	8 arguments	53
6.4	Conclusion	57
6.5	By the way	58
<b>7</b>	<b>scanf()</b>	<b>59</b>
7.1	Simple example	59
7.1.1	About pointers	59
7.1.2	x86	60
7.1.3	MSVC + OllyDbg	62
7.1.4	x64	65
7.1.5	ARM	66
7.1.6	MIPS	67
7.2	Global variables	68
7.2.1	MSVC: x86	68
7.2.2	MSVC: x86 + OllyDbg	70
7.2.3	GCC: x86	71
7.2.4	MSVC: x64	71
7.2.5	ARM: Optimizing Keil 6/2013 (Thumb mode)	72
7.2.6	ARM64	73
7.2.7	MIPS	73
7.3	scanf() result checking	77
7.3.1	MSVC: x86	77
7.3.2	MSVC: x86: IDA	78
7.3.3	MSVC: x86 + OllyDbg	82
7.3.4	MSVC: x86 + Hiew	84
7.3.5	MSVC: x64	85
7.3.6	ARM	86
7.3.7	MIPS	87
7.3.8	Exercise	88
7.4	Exercise	88
<b>8</b>	<b>Accessing passed arguments</b>	<b>89</b>
8.1	x86	89
8.1.1	MSVC	89
8.1.2	MSVC + OllyDbg	90
8.1.3	GCC	90
8.2	x64	91
8.2.1	MSVC	91
8.2.2	GCC	92
8.2.3	GCC: uint64_t instead of int	93
8.3	ARM	94
8.3.1	Non-optimizing Keil 6/2013 (ARM mode)	94
8.3.2	Optimizing Keil 6/2013 (ARM mode)	95
8.3.3	Optimizing Keil 6/2013 (Thumb mode)	95
8.3.4	ARM64	95
8.4	MIPS	97
<b>9</b>	<b>More about results returning</b>	<b>98</b>
9.1	Attempt to use the result of a function returning <i>void</i>	98
9.2	What if we do not use the function result?	99
9.3	Returning a structure	99

<b>10 Pointers</b>	<b>101</b>
10.1 Global variables example	101
10.2 Local variables example	107
10.3 Conclusion	110
<b>11 GOTO operator</b>	<b>111</b>
11.1 Dead code	113
11.2 Exercise	114
<b>12 Conditional jumps</b>	<b>115</b>
12.1 Simple example	115
12.1.1 x86	115
12.1.2 ARM	126
12.1.3 MIPS	129
12.2 Calculating absolute value	132
12.2.1 Optimizing MSVC	132
12.2.2 Optimizing Keil 6/2013: Thumb mode	132
12.2.3 Optimizing Keil 6/2013: ARM mode	132
12.2.4 Non-optimizing GCC 4.9 (ARM64)	133
12.2.5 MIPS	133
12.2.6 Branchless version?	133
12.3 Ternary conditional operator	133
12.3.1 x86	134
12.3.2 ARM	135
12.3.3 ARM64	135
12.3.4 MIPS	136
12.3.5 Let's rewrite it in an if/else way	136
12.3.6 Conclusion	136
12.4 Getting minimal and maximal values	137
12.4.1 32-bit	137
12.4.2 64-bit	139
12.4.3 MIPS	141
12.5 Conclusion	141
12.5.1 x86	141
12.5.2 ARM	141
12.5.3 MIPS	142
12.5.4 Branchless	142
12.6 Exercise	142
<b>13 switch()/case/default</b>	<b>143</b>
13.1 Small number of cases	143
13.1.1 x86	143
13.1.2 ARM: Optimizing Keil 6/2013 (ARM mode)	153
13.1.3 ARM: Optimizing Keil 6/2013 (Thumb mode)	153
13.1.4 ARM64: Non-optimizing GCC (Linaro) 4.9	154
13.1.5 ARM64: Optimizing GCC (Linaro) 4.9	155
13.1.6 MIPS	155
13.1.7 Conclusion	156
13.2 A lot of cases	156
13.2.1 x86	156
13.2.2 ARM: Optimizing Keil 6/2013 (ARM mode)	163
13.2.3 ARM: Optimizing Keil 6/2013 (Thumb mode)	164
13.2.4 MIPS	166
13.2.5 Conclusion	167
13.3 When there are several case statements in one block	168
13.3.1 MSVC	168
13.3.2 GCC	169
13.3.3 ARM64: Optimizing GCC 4.9.1	170
13.4 Fall-through	171
13.4.1 MSVC x86	172
13.4.2 ARM64	173
13.5 Exercises	173
13.5.1 Exercise #1	173

<b>14 Loops</b>	<b>174</b>
14.1 Simple example	174
14.1.1 x86	174
14.1.2 x86: OllyDbg	178
14.1.3 x86: tracer	178
14.1.4 ARM	180
14.1.5 MIPS	183
14.1.6 One more thing	184
14.2 Memory blocks copying routine	184
14.2.1 Straight-forward implementation	184
14.2.2 ARM in ARM mode	185
14.2.3 MIPS	186
14.2.4 Vectorization	186
14.3 Conclusion	187
14.4 Exercises	188
<b>15 Simple C-strings processing</b>	<b>189</b>
15.1 strlen()	189
15.1.1 x86	189
15.1.2 ARM	196
15.1.3 MIPS	199
<b>16 Replacing arithmetic instructions to other ones</b>	<b>200</b>
16.1 Multiplication	200
16.1.1 Multiplication using addition	200
16.1.2 Multiplication using shifting	200
16.1.3 Multiplication using shifting, subtracting, and adding	201
16.2 Division	205
16.2.1 Division using shifts	205
16.3 Exercise	205
<b>17 Floating-point unit</b>	<b>206</b>
17.1 IEEE 754	206
17.2 x86	206
17.3 ARM, MIPS, x86/x64 SIMD	206
17.4 C/C++	206
17.5 Simple example	207
17.5.1 x86	207
17.5.2 ARM: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)	214
17.5.3 ARM: Optimizing Keil 6/2013 (Thumb mode)	215
17.5.4 ARM64: Optimizing GCC (Linaro) 4.9	215
17.5.5 ARM64: Non-optimizing GCC (Linaro) 4.9	216
17.5.6 MIPS	217
17.6 Passing floating point numbers via arguments	217
17.6.1 x86	218
17.6.2 ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)	218
17.6.3 ARM + Non-optimizing Keil 6/2013 (ARM mode)	219
17.6.4 ARM64 + Optimizing GCC (Linaro) 4.9	219
17.6.5 MIPS	220
17.7 Comparison example	221
17.7.1 x86	221
17.7.2 ARM	248
17.7.3 ARM64	251
17.7.4 MIPS	253
17.8 Stack, calculators and reverse Polish notation	253
17.9 x64	253
17.10 Exercises	253
<b>18 Arrays</b>	<b>254</b>
18.1 Simple example	254
18.1.1 x86	254
18.1.2 ARM	257
18.1.3 MIPS	260
18.2 Buffer overflow	261
18.2.1 Reading outside array bounds	261



18.2.2 Writing beyond array bounds	264
18.3 Buffer overflow protection methods	269
18.3.1 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)	270
18.4 One more word about arrays	272
18.5 Array of pointers to strings	272
18.5.1 x64	273
18.5.2 32-bit ARM	274
18.5.3 ARM64	275
18.5.4 MIPS	276
18.5.5 Array overflow	276
18.6 Multidimensional arrays	279
18.6.1 Two-dimensional array example	279
18.6.2 Access two-dimensional array as one-dimensional	280
18.6.3 Three-dimensional array example	282
18.6.4 More examples	285
18.7 Pack of strings as a two-dimensional array	285
18.7.1 32-bit ARM	287
18.7.2 ARM64	287
18.7.3 MIPS	288
18.7.4 Conclusion	288
18.8 Conclusion	289
18.9 Exercises	289
<b>19 Manipulating specific bit(s)</b>	<b>290</b>
19.1 Specific bit checking	290
19.1.1 x86	290
19.1.2 ARM	292
19.2 Setting and clearing specific bits	293
19.2.1 x86	294
19.2.2 ARM + Optimizing Keil 6/2013 (ARM mode)	299
19.2.3 ARM + Optimizing Keil 6/2013 (Thumb mode)	300
19.2.4 ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode)	300
19.2.5 ARM: more about the BIC instruction	300
19.2.6 ARM64: Optimizing GCC (Linaro) 4.9	300
19.2.7 ARM64: Non-optimizing GCC (Linaro) 4.9	301
19.2.8 MIPS	301
19.3 Shifts	301
19.4 Setting and clearing specific bits: FPU <sup>5</sup> example	301
19.4.1 A word about the XOR operation	302
19.4.2 x86	302
19.4.3 MIPS	304
19.4.4 ARM	304
19.5 Counting bits set to 1	306
19.5.1 x86	307
19.5.2 x64	315
19.5.3 ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode)	317
19.5.4 ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)	318
19.5.5 ARM64 + Optimizing GCC 4.9	318
19.5.6 ARM64 + Non-optimizing GCC 4.9	318
19.5.7 MIPS	319
19.6 Conclusion	321
19.6.1 Check for specific bit (known at compile stage)	321
19.6.2 Check for specific bit (specified at runtime)	321
19.6.3 Set specific bit (known at compile stage)	322
19.6.4 Set specific bit (specified at runtime)	322
19.6.5 Clear specific bit (known at compile stage)	322
19.6.6 Clear specific bit (specified at runtime)	323
19.7 Exercises	323
<b>20 Linear congruential generator</b>	<b>324</b>
20.1 x86	324
20.2 x64	325
20.3 32-bit ARM	326

---

<sup>5</sup>Floating-point unit

20.4 MIPS	326
20.4.1 MIPS relocations	327
20.5 Thread-safe version of the example	328
<b>21 Structures</b>	<b>329</b>
21.1 MSVC: SYSTEMTIME example	329
21.1.1 OllyDbg	331
21.1.2 Replacing the structure with array	331
21.2 Let's allocate space for a structure using malloc()	332
21.3 UNIX: struct tm	334
21.3.1 Linux	334
21.3.2 ARM	337
21.3.3 MIPS	338
21.3.4 Structure as a set of values	340
21.3.5 Structure as an array of 32-bit words	341
21.3.6 Structure as an array of bytes	342
21.4 Fields packing in structure	344
21.4.1 x86	344
21.4.2 ARM	348
21.4.3 MIPS	349
21.4.4 One more word	350
21.5 Nested structures	350
21.5.1 OllyDbg	352
21.6 Bit fields in a structure	352
21.6.1 CPUID example	352
21.6.2 Working with the float type as with a structure	356
21.7 Exercises	359
<b>22 Unions</b>	<b>360</b>
22.1 Pseudo-random number generator example	360
22.1.1 x86	361
22.1.2 MIPS	362
22.1.3 ARM (ARM mode)	363
22.2 Calculating machine epsilon	364
22.2.1 x86	365
22.2.2 ARM64	365
22.2.3 MIPS	366
22.2.4 Conclusion	366
22.3 Fast square root calculation	366
<b>23 Pointers to functions</b>	<b>368</b>
23.1 MSVC	369
23.1.1 MSVC + OllyDbg	371
23.1.2 MSVC + tracer	373
23.1.3 MSVC + tracer (code coverage)	375
23.2 GCC	375
23.2.1 GCC + GDB (with source code)	376
23.2.2 GCC + GDB (no source code)	377
<b>24 64-bit values in 32-bit environment</b>	<b>380</b>
24.1 Returning of 64-bit value	380
24.1.1 x86	380
24.1.2 ARM	380
24.1.3 MIPS	381
24.2 Arguments passing, addition, subtraction	381
24.2.1 x86	381
24.2.2 ARM	382
24.2.3 MIPS	383
24.3 Multiplication, division	384
24.3.1 x86	384
24.3.2 ARM	386
24.3.3 MIPS	387
24.4 Shifting right	388
24.4.1 x86	388
24.4.2 ARM	388

24.4.3 MIPS	389
24.5 Converting 32-bit value into 64-bit one	389
24.5.1 x86	389
24.5.2 ARM	389
24.5.3 MIPS	390
<b>25 SIMD</b>	<b>391</b>
25.1 Vectorization	391
25.1.1 Addition example	392
25.1.2 Memory copy example	397
25.2 SIMD strlen() implementation	401
<b>26 64 bits</b>	<b>404</b>
26.1 x86-64	404
26.2 ARM	410
26.3 Float point numbers	411
<b>27 Working with floating point numbers using SIMD</b>	<b>412</b>
27.1 Simple example	412
27.1.1 x64	412
27.1.2 x86	413
27.2 Passing floating point number via arguments	420
27.3 Comparison example	421
27.3.1 x64	421
27.3.2 x86	422
27.4 Calculating machine epsilon: x64 and SIMD	422
27.5 Pseudo-random number generator example revisited	423
27.6 Summary	423
<b>28 ARM-specific details</b>	<b>425</b>
28.1 Number sign (#) before number	425
28.2 Addressing modes	425
28.3 Loading a constant into a register	426
28.3.1 32-bit ARM	426
28.3.2 ARM64	426
28.4 Relocs in ARM64	427
<b>29 MIPS-specific details</b>	<b>429</b>
29.1 Loading constants into register	429
29.2 Further reading about MIPS	429
<b>II Important fundamentals</b>	<b>430</b>
<b>30 Signed number representations</b>	<b>432</b>
<b>31 Endianness</b>	<b>434</b>
31.1 Big-endian	434
31.2 Little-endian	434
31.3 Example	434
31.4 Bi-endian	435
31.5 Converting data	435
<b>32 Memory</b>	<b>436</b>
<b>33 CPU</b>	<b>437</b>
33.1 Branch predictors	437
33.2 Data dependencies	437
<b>34 Hash functions</b>	<b>438</b>
34.1 How one-way function works?	438
<b>III Slightly more advanced examples</b>	<b>439</b>
<b>35 Temperature converting</b>	<b>440</b>

35.1 Integer values	440
35.1.1 Optimizing MSVC 2012 x86	440
35.1.2 Optimizing MSVC 2012 x64	442
35.2 Floating-point values	442
<b>36 Fibonacci numbers</b>	<b>445</b>
36.1 Example #1	445
36.2 Example #2	448
36.3 Summary	451
<b>37 CRC32 calculation example</b>	<b>452</b>
<b>38 Network address calculation example</b>	<b>455</b>
38.1 calc_network_address()	456
38.2 form_IP()	457
38.3 print_as_IP()	458
38.4 form_netmask() and set_bit()	459
38.5 Summary	460
<b>39 Loops: several iterators</b>	<b>461</b>
39.1 Three iterators	461
39.2 Two iterators	462
39.3 Intel C++ 2011 case	463
<b>40 Duff's device</b>	<b>466</b>
<b>41 Division by 9</b>	<b>469</b>
41.1 x86	469
41.2 ARM	470
41.2.1 Optimizing Xcode 4.6.3 (LLVM) (ARM mode)	470
41.2.2 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)	471
41.2.3 Non-optimizing Xcode 4.6.3 (LLVM) and Keil 6/2013	471
41.3 MIPS	471
41.4 How it works	472
41.4.1 More theory	473
41.5 Getting the divisor	473
41.5.1 Variant #1	473
41.5.2 Variant #2	474
41.6 Exercise	474
<b>42 String to number conversion (atoi())</b>	<b>475</b>
42.1 Simple example	475
42.1.1 Optimizing MSVC 2013 x64	475
42.1.2 Optimizing GCC 4.9.1 x64	476
42.1.3 Optimizing Keil 6/2013 (ARM mode)	476
42.1.4 Optimizing Keil 6/2013 (Thumb mode)	477
42.1.5 Optimizing GCC 4.9.1 ARM64	477
42.2 A slightly advanced example	478
42.2.1 Optimizing GCC 4.9.1 x64	479
42.2.2 Optimizing Keil 6/2013 (ARM mode)	480
42.3 Exercise	481
<b>43 Inline functions</b>	<b>482</b>
43.1 Strings and memory functions	483
43.1.1 strcmp()	483
43.1.2 strlen()	485
43.1.3 strcpy()	485
43.1.4 memset()	485
43.1.5 memcpy()	487
43.1.6 memcmp()	489
43.1.7 IDA script	490
<b>44 C99 restrict</b>	<b>491</b>
<b>45 Branchless abs() function</b>	<b>494</b>
45.1 Optimizing GCC 4.9.1 x64	494

45.2 Optimizing GCC 4.9 ARM64	495
<b>46 Variadic functions</b>	<b>496</b>
46.1 Computing arithmetic mean	496
46.1.1 <i>cdecl</i> calling conventions	496
46.1.2 Register-based calling conventions	497
46.2 <i>vprintf()</i> function case	499
<b>47 Strings trimming</b>	<b>501</b>
47.1 x64: Optimizing MSVC 2013	502
47.2 x64: Non-optimizing GCC 4.9.1	503
47.3 x64: Optimizing GCC 4.9.1	504
47.4 ARM64: Non-optimizing GCC (Linaro) 4.9	505
47.5 ARM64: Optimizing GCC (Linaro) 4.9	506
47.6 ARM: Optimizing Keil 6/2013 (ARM mode)	507
47.7 ARM: Optimizing Keil 6/2013 (Thumb mode)	507
47.8 MIPS	508
<b>48 toupper() function</b>	<b>510</b>
48.1 x64	510
48.1.1 Two comparison operations	510
48.1.2 One comparison operation	511
48.2 ARM	512
48.2.1 GCC for ARM64	512
48.3 Summary	513
<b>49 Incorrectly disassembled code</b>	<b>514</b>
49.1 Disassembling from an incorrect start (x86)	514
49.2 How does random noise looks disassembled?	515
<b>50 Obfuscation</b>	<b>519</b>
50.1 Text strings	519
50.2 Executable code	520
50.2.1 Inserting garbage	520
50.2.2 Replacing instructions with bloated equivalents	520
50.2.3 Always executed/never executed code	520
50.2.4 Making a lot of mess	520
50.2.5 Using indirect pointers	521
50.3 Virtual machine / pseudo-code	521
50.4 Other things to mention	521
50.5 Exercise	521
<b>51 C++</b>	<b>522</b>
51.1 Classes	522
51.1.1 A simple example	522
51.1.2 Class inheritance	528
51.1.3 Encapsulation	531
51.1.4 Multiple inheritance	532
51.1.5 Virtual methods	535
51.2 ostream	538
51.3 References	539
51.4 STL	539
51.4.1 <code>std::string</code>	539
51.4.2 <code>std::list</code>	546
51.4.3 <code>std::vector</code>	555
51.4.4 <code>std::map</code> and <code>std::set</code>	562
<b>52 Negative array indices</b>	<b>572</b>
<b>53 Windows 16-bit</b>	<b>575</b>
53.1 Example#1	575
53.2 Example #2	575
53.3 Example #3	576
53.4 Example #4	577
53.5 Example #5	579

53.6 Example #6	583
53.6.1 Global variables	584
<b>IV Java</b>	<b>586</b>
<b>54 Java</b>	<b>587</b>
54.1 Introduction	587
54.2 Returning a value	587
54.3 Simple calculating functions	591
54.4 JVM <sup>6</sup> memory model	594
54.5 Simple function calling	594
54.6 Calling beep()	595
54.7 Linear congruential PRNG <sup>7</sup>	596
54.8 Conditional jumps	597
54.9 Passing arguments	599
54.10 Bitfields	600
54.11 Loops	601
54.12 switch()	603
54.13 Arrays	604
54.13.1 Simple example	604
54.13.2 Summing elements of array	605
54.13.3 main() function sole argument is array too	605
54.13.4 Pre-initialized array of strings	606
54.13.5 Variadic functions	608
54.13.6 Two-dimensional arrays	609
54.13.7 Three-dimensional arrays	610
54.13.8 Summary	611
54.14 Strings	611
54.14.1 First example	611
54.14.2 Second example	612
54.15 Exceptions	613
54.16 Classes	616
54.17 Simple patching	618
54.17.1 First example	618
54.17.2 Second example	620
54.18 Summary	623
<b>V Finding important/interesting stuff in the code</b>	<b>624</b>
<b>55 Identification of executable files</b>	<b>626</b>
55.1 Microsoft Visual C++	626
55.1.1 Name mangling	626
55.2 GCC	626
55.2.1 Name mangling	626
55.2.2 Cygwin	626
55.2.3 MinGW	626
55.3 Intel FORTRAN	627
55.4 Watcom, OpenWatcom	627
55.4.1 Name mangling	627
55.5 Borland	627
55.5.1 Delphi	627
55.6 Other known DLLs	628
<b>56 Communication with the outer world (win32)</b>	<b>629</b>
56.1 Often used functions in the Windows API	629
56.2 tracer: Intercepting all functions in specific module	630
<b>57 Strings</b>	<b>631</b>
57.1 Text strings	631
57.1.1 C/C++	631
57.1.2 Borland Delphi	631

---

<sup>6</sup>Java virtual machine

<sup>7</sup>Pseudorandom number generator

57.1.3 Unicode	632
57.1.4 Base64	634
57.2 Error/debug messages	635
57.3 Suspicious magic strings	635
<b>58 Calls to assert()</b>	<b>636</b>
<b>59 Constants</b>	<b>637</b>
59.1 Magic numbers	637
59.1.1 DHCP	638
59.2 Searching for constants	638
<b>60 Finding the right instructions</b>	<b>639</b>
<b>61 Suspicious code patterns</b>	<b>641</b>
61.1 XOR instructions	641
61.2 Hand-written assembly code	641
<b>62 Using magic numbers while tracing</b>	<b>643</b>
<b>63 Other things</b>	<b>644</b>
63.1 General idea	644
63.2 C++	644
63.3 Some binary file patterns	644
63.4 Memory “snapshots” comparing	645
63.4.1 Windows registry	646
63.4.2 Blink-comparator	646
<b>VI OS-specific</b>	<b>647</b>
<b>64 Arguments passing methods (calling conventions)</b>	<b>648</b>
64.1 cdecl	648
64.2 stdcall	648
64.2.1 Functions with variable number of arguments	649
64.3 fastcall	649
64.3.1 GCC regparm	650
64.3.2 Watcom/OpenWatcom	650
64.4 thiscall	650
64.5 x86-64	650
64.5.1 Windows x64	650
64.5.2 Linux x64	653
64.6 Return values of <i>float</i> and <i>double</i> type	653
64.7 Modifying arguments	653
64.8 Taking a pointer to function argument	654
<b>65 Thread Local Storage</b>	<b>656</b>
65.1 Linear congruential generator revisited	656
65.1.1 Win32	656
65.1.2 Linux	660
<b>66 System calls (syscall-s)</b>	<b>661</b>
66.1 Linux	661
66.2 Windows	662
<b>67 Linux</b>	<b>663</b>
67.1 Position-independent code	663
67.1.1 Windows	665
67.2 <i>LD_PRELOAD</i> hack in Linux	665
<b>68 Windows NT</b>	<b>668</b>
68.1 CRT (win32)	668
68.2 Win32 PE	671
68.2.1 Terminology	671
68.2.2 Base address	672
68.2.3 Subsystem	672

68.2.4 OS version . . . . .	672
68.2.5 Sections . . . . .	673
68.2.6 Relocations (relocs) . . . . .	673
68.2.7 Exports and imports . . . . .	674
68.2.8 Resources . . . . .	676
68.2.9 .NET . . . . .	676
68.2.10 TLS . . . . .	677
68.2.11 Tools . . . . .	677
68.2.12 Further reading . . . . .	677
68.3 Windows SEH . . . . .	677
68.3.1 Let's forget about MSVC . . . . .	677
68.3.2 Now let's get back to MSVC . . . . .	682
68.3.3 Windows x64 . . . . .	695
68.3.4 Read more about SEH . . . . .	699
68.4 Windows NT: Critical section . . . . .	699
<b>VII Tools</b>	<b>701</b>
<b>69 Disassembler</b>	<b>702</b>
69.1 IDA . . . . .	702
<b>70 Debugger</b>	<b>703</b>
70.1 OllyDbg . . . . .	703
70.2 GDB . . . . .	703
70.3 tracer . . . . .	703
<b>71 System calls tracing</b>	<b>704</b>
71.0.1 strace / dtruss . . . . .	704
<b>72 Decompilers</b>	<b>705</b>
<b>73 Other tools</b>	<b>706</b>
<b>VIII Examples of real-world RE tasks</b>	<b>707</b>
<b>74 Task manager practical joke (Windows Vista)</b>	<b>709</b>
74.1 Using LEA to load values . . . . .	711
<b>75 Color Lines game practical joke</b>	<b>713</b>
<b>76 Minesweeper (Windows XP)</b>	<b>717</b>
76.1 Exercises . . . . .	721
<b>77 Hand decompiling + Z3 SMT solver</b>	<b>722</b>
77.1 Hand decompiling . . . . .	722
77.2 Now let's use the Z3 SMT solver . . . . .	725
<b>78 Dongles</b>	<b>730</b>
78.1 Example #1: MacOS Classic and PowerPC . . . . .	730
78.2 Example #2: SCO OpenServer . . . . .	737
78.2.1 Decrypting error messages . . . . .	744
78.3 Example #3: MS-DOS . . . . .	746
<b>79 "QR9": Rubik's cube inspired amateur crypto-algorithm</b>	<b>752</b>
<b>80 SAP</b>	<b>779</b>
80.1 About SAP client network traffic compression . . . . .	779
80.2 SAP 6.0 password checking functions . . . . .	789
<b>81 Oracle RDBMS</b>	<b>793</b>
81.1 V\$VERSION table in the Oracle RDBMS . . . . .	793
81.2 X\$KSMRLU table in Oracle RDBMS . . . . .	800
81.3 V\$TIMER table in Oracle RDBMS . . . . .	802



<b>82 Handwritten assembly code</b>	<b>806</b>
82.1 EICAR test file	806
<b>83 Demos</b>	<b>808</b>
83.1 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10	808
83.1.1 Trixter's 42 byte version	808
83.1.2 My attempt to reduce Trixter's version: 27 bytes	809
83.1.3 Taking random memory garbage as a source of randomness	809
83.1.4 Conclusion	810
83.2 Mandelbrot set	811
83.2.1 Theory	812
83.2.2 Let's get back to the demo	817
83.2.3 My "fixed" version	819
<b>IX Examples of reversing proprietary file formats</b>	<b>821</b>
<b>84 Primitive XOR-encryption</b>	<b>822</b>
84.1 Norton Guide: simplest possible 1-byte XOR encryption	823
84.1.1 Entropy	824
84.2 Simplest possible 4-byte XOR encryption	826
84.2.1 Exercise	829
<b>85 Millenium game save file</b>	<b>830</b>
<b>86 Oracle RDBMS: .SYM-files</b>	<b>837</b>
<b>87 Oracle RDBMS: .MSB-files</b>	<b>846</b>
87.1 Summary	851
<b>X Other things</b>	<b>852</b>
<b>88 npad</b>	<b>853</b>
<b>89 Executable files patching</b>	<b>855</b>
89.1 Text strings	855
89.2 x86 code	855
<b>90 Compiler intrinsic</b>	<b>856</b>
<b>91 Compiler's anomalies</b>	<b>857</b>
<b>92 OpenMP</b>	<b>858</b>
92.1 MSVC	860
92.2 GCC	861
<b>93 Itanium</b>	<b>864</b>
<b>94 8086 memory model</b>	<b>867</b>
<b>95 Basic blocks reordering</b>	<b>868</b>
95.1 Profile-guided optimization	868
<b>XI Books/blogs worth reading</b>	<b>870</b>
<b>96 Books</b>	<b>871</b>
96.1 Windows	871
96.2 C/C++	871
96.3 x86 / x86-64	871
96.4 ARM	871
96.5 Cryptography	871
<b>97 Blogs</b>	<b>872</b>
97.1 Windows	872

**98 Other****873****Afterword****875****99 Questions?****875****Appendix****877****A x86****877**

A.1	Terminology	877
A.2	General purpose registers	877
A.2.1	RAX/EAX/AX/AL	877
A.2.2	RBX/EBX/BX/BL	878
A.2.3	RCX/ECX/CX/CL	878
A.2.4	RDY/EDX/DX/DL	878
A.2.5	RSI/ESI/SI/SIL	878
A.2.6	RDI/EDI/DI/DIL	878
A.2.7	R8/R8D/R8W/R8L	878
A.2.8	R9/R9D/R9W/R9L	879
A.2.9	R10/R10D/R10W/R10L	879
A.2.10	R11/R11D/R11W/R11L	879
A.2.11	R12/R12D/R12W/R12L	879
A.2.12	R13/R13D/R13W/R13L	879
A.2.13	R14/R14D/R14W/R14L	879
A.2.14	R15/R15D/R15W/R15L	879
A.2.15	RSP/ESP/SP/SPL	880
A.2.16	RBP/EBP/BP/BPL	880
A.2.17	RIP/EIP/IP	880
A.2.18	CS/DS/ES/SS/FS/GS	880
A.2.19	Flags register	880
A.3	FPU registers	881
A.3.1	Control Word	881
A.3.2	Status Word	882
A.3.3	Tag Word	882
A.4	SIMD registers	883
A.4.1	MMX registers	883
A.4.2	SSE and AVX registers	883
A.5	Debugging registers	883
A.5.1	DR6	883
A.5.2	DR7	883
A.6	Instructions	884
A.6.1	Prefixes	884
A.6.2	Most frequently used instructions	885
A.6.3	Less frequently used instructions	889
A.6.4	FPU instructions	893
A.6.5	Instructions having printable ASCII opcode	894

**B ARM****896**

B.1	Terminology	896
B.2	Versions	896
B.3	32-bit ARM (AArch32)	896
B.3.1	General purpose registers	896
B.3.2	Current Program Status Register (CPSR)	897
B.3.3	VFP (floating point) and NEON registers	897
B.4	64-bit ARM (AArch64)	897
B.4.1	General purpose registers	897
B.5	Instructions	898
B.5.1	Conditional codes table	898

**C MIPS****899**

C.1	Registers	899
C.1.1	General purpose registers GPR <sup>8</sup>	899

<sup>8</sup>General Purpose Registers

C.1.2	Floating-point registers . . . . .	899
C.2	Instructions . . . . .	899
C.2.1	Jump instructions . . . . .	900
<b>D</b>	<b>Some GCC library functions</b>	<b>901</b>
<b>E</b>	<b>Some MSVC library functions</b>	<b>902</b>
<b>F</b>	<b>Cheatsheets</b>	<b>903</b>
F.1	IDA . . . . .	903
F.2	OllyDbg . . . . .	903
F.3	MSVC . . . . .	904
F.4	GCC . . . . .	904
F.5	GDB . . . . .	904
<b>Acronyms used</b>		<b>907</b>
<b>Glossary</b>		<b>911</b>
<b>Index</b>		<b>913</b>
<b>Bibliography</b>		<b>919</b>

## Preface

There are several popular meanings of the term “[reverse engineering](#)”: 1) The reverse engineering of software: researching compiled programs; 2) The scanning of 3D structures and the subsequent digital manipulation required in order to duplicate them; 3) Recreating [DBMS](#)<sup>9</sup> structure. This book is about the first meaning.

### Topics discussed in-depth

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

### Topics touched upon

Oracle RDBMS ([81 on page 793](#)), Itanium ([93 on page 864](#)), copy-protection dongles ([78 on page 730](#)), LD\_PRELOAD ([67.2 on page 665](#)), stack overflow, [ELF](#)<sup>10</sup>, win32 PE file format ([68.2 on page 671](#)), x86-64 ([26.1 on page 404](#)), critical sections ([68.4 on page 699](#)), syscalls ([66 on page 661](#)), [TLS](#)<sup>11</sup>, position-independent code ([PIC](#)<sup>12</sup>) ([67.1 on page 663](#)), profile-guided optimization ([95.1 on page 868](#)), C++ STL ([51.4 on page 539](#)), OpenMP ([92 on page 858](#)), SEH ([68.3 on page 677](#)).

### Exercises and tasks

...are all moved to the separate website: <http://challenges.re>.

### About the author



Dennis Yurichev is an experienced reverse engineer and programmer. He can be contacted by email: [dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com), or on Skype: [dennis.yurichev](#).

### Praise for *Reverse Engineering for Beginners*

- “It’s very well done .. and for free .. amazing.”<sup>13</sup> Daniel Bilar, Siege Technologies, LLC.
- “... excellent and free”<sup>14</sup> Pete Finnigan, Oracle RDBMS security guru.
- “... book is interesting, great job!” Michael Sikorski, author of *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.
- “... my compliments for the very nice tutorial!” Herbert Bos, full professor at the Vrije Universiteit Amsterdam, co-author of *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, SAP Netweaver & Security specialist.
- “... reasonable intro to some of the techniques.”<sup>15</sup> Mike Stay, teacher at the Federal Law Enforcement Training Center, Georgia, US.

<sup>9</sup>Database management systems

<sup>10</sup>Executable file format widely used in \*NIX systems including Linux

<sup>11</sup>Thread Local Storage

<sup>12</sup>Position Independent Code: [67.1 on page 663](#)

<sup>13</sup>[twitter.com/daniel\\_bilar/status/436578617221742593](https://twitter.com/daniel_bilar/status/436578617221742593)

<sup>14</sup>[twitter.com/petefinnigan/status/400551705797869568](https://twitter.com/petefinnigan/status/400551705797869568)

<sup>15</sup>[reddit](#)

- “I love this book! I have several students reading it at the moment, plan to use it in graduate course.”<sup>16</sup> Sergey Bratus, Research Assistant Professor at the Computer Science Department at Dartmouth College
- “Dennis @Yurichev has published an impressive (and free!) book on reverse engineering”<sup>17</sup> Tanel Poder, Oracle RDBMS performance tuning expert .
- “This book is some kind of Wikipedia to beginners...” Archer, Chinese Translator, IT Security Researcher.

## Thanks

For patiently answering all my questions: Andrey “herm1t” Baranovich, Slava “Avid” Kazakov.

For sending me notes about mistakes and inaccuracies: Stanislav “Beaver” Bobrytskyy, Alexander Lysenko, Shell Rocket, Zhu Ruijin, Changmin Heo.

For helping me in other ways: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), Aliaksandr Autayeu.

For translating the book into Simplified Chinese: Antiy Labs ([antiy.cn](http://antiy.cn)) and Archer.

For translating the book into Korean: Byungho Min.

For proofreading: Alexander “Lstar” Chernenkiy, Vladimir Botov, Andrei Brazhuk, Mark “Logxen” Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen.

Vasil Kolev did a great amount of work in proofreading and correcting many mistakes.

For illustrations and cover art: Andy Nechaevsky.

Thanks also to all the folks on github.com who have contributed notes and corrections.

Many  $\LaTeX$  packages were used: I would like to thank the authors as well.

## Donors

Those who supported me during the time when I wrote significant part of the book:

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).

Thanks a lot to every donor!

## mini-FAQ

Q: Why should one learn assembly language these days?

A: Unless you are an [OS](#)<sup>18</sup> developer, you probably don’t need to code in assembly—modern compilers are much better at performing optimizations than humans <sup>19</sup>. Also, modern [CPU](#)<sup>20</sup>s are very complex devices and assembly knowledge doesn’t really help one to understand their internals. That being said, there are at least two areas where a good understanding of assembly can be helpful: First and foremost, security/malware research. It is also a good way to gain a better understanding of your compiled code whilst debugging.

This book is therefore intended for those who want to understand assembly language rather than to code in it, which is why there are many examples of compiler output contained within.

Q: I clicked on a hyperlink inside a PDF-document, how do I go back?

A: In Adobe Acrobat Reader click Alt+LeftArrow.

Q: Your book is huge! Is there anything shorter?

<sup>16</sup>[twitter.com/sergeybratus/status/505590326560833536](https://twitter.com/sergeybratus/status/505590326560833536)

<sup>17</sup>[twitter.com/TanelPoder/status/524668104065159169](https://twitter.com/TanelPoder/status/524668104065159169)

<sup>18</sup>Operating System

<sup>19</sup>A very good text about this topic: [\[Fog13b\]](#)

<sup>20</sup>Central processing unit

A: There is a shortened, lite version found here: <http://beginners.re/#lite>.

Q: I'm not sure if I should try to learn reverse engineering or not.

A: Perhaps, the average time to become familiar with the contents of the shortened LITE-version is 1-2 month(s). You may also try [reverse engineering challenges](#).

Q: May I print this book / use it for teaching?

A: Of course! That's why the book is licensed under the Creative Commons license.

Someone might also want to build one's own version of book—read [here](#) to find out more.

Q: Why this book is free? You've done great job. This is suspicious, as many other free things.

A: To my own experience, authors of technical literature do this mostly for self-advertisement purposes. It's not possible to gain any decent money from such work.

Q: How does one get a job in reverse engineering?

A: There are hiring threads that appear from time to time on reddit, devoted to RE<sup>21</sup> ([2013 Q3](#), [2014](#)). Try looking there.

A somewhat related hiring thread can be found in the "netsec" subreddit: [2014 Q2](#).

Q: I have a question...

A: Send it to me by email ([dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com)).

## About the Korean translation

In January 2015, the Acorn publishing company ([www.acornpub.co.kr](http://www.acornpub.co.kr)) in South Korea did a huge amount of work in translating and publishing my book (as it was in August 2014) into Korean.

It's now available at [their website](#).

The translator is Byungho Min ([twitter/tais9](#)).

The cover art was done by my artistic friend, Andy Nechaevsky: [facebook/andydinka](#).

They also hold the copyright to the Korean translation.

So, if you want to have a *real* book on your shelf in Korean and want to support my work, it is now available for purchase.

---

<sup>21</sup>[reddit.com/r/ReverseEngineering/](http://reddit.com/r/ReverseEngineering/)

## **Part I**

# **Code patterns**

When the author of this book first started learning C and, later, C++, he used to write small pieces of code, compile them, and then look at the assembly language output. This made it very easy for him to understand what was going on in the code that he had written. <sup>22</sup> He did it so many times that the relationship between the C/C++ code and what the compiler produced was imprinted deeply in his mind. It's easy to imagine instantly a rough outline of C code's appearance and function. Perhaps this technique could be helpful for others.

Sometimes ancient compilers are used here, in order to get the shortest (or simplest) possible code snippet.

## Exercises

When the author of this book studied assembly language, he also often compiled small C-functions and then rewrote them gradually to assembly, trying to make their code as short as possible. This probably is not worth doing in real-world scenarios today, because it's hard to compete with modern compilers in terms of efficiency. It is, however, a very good way to gain a better understanding of assembly.

Feel free, therefore, to take any assembly code from this book and try to make it shorter. However, don't forget to test what you have written.

## Optimization levels and debug information

Source code can be compiled by different compilers with various optimization levels. A typical compiler has about three such levels, where level zero means disable optimization. Optimization can also be targeted towards code size or code speed.

A non-optimizing compiler is faster and produces more understandable (albeit verbose) code, whereas an optimizing compiler is slower and tries to produce code that runs faster (but is not necessarily more compact).

In addition to optimization levels and direction, a compiler can include in the resulting file some debug information, thus producing code for easy debugging.

One of the important features of the 'debug' code is that it might contain links between each line of the source code and the respective machine code addresses. Optimizing compilers, on the other hand, tend to produce output where entire lines of source code can be optimized away and thus not even be present in the resulting machine code.

Reverse engineers can encounter either version, simply because some developers turn on the compiler's optimization flags and others do not. Because of this, we'll try to work on examples of both debug and release versions of the code featured in this book, where possible.

---

<sup>22</sup>In fact, he still does it when he can't understand what a particular bit of code does.



# Chapter 1

## A short introduction to the CPU

The **CPU** is the device that executes the machine code a program consists of.

### A short glossary:

**Instruction** : A primitive **CPU** command. The simplest examples include: moving data between registers, working with memory, primitive arithmetic operations . As a rule, each **CPU** has its own instruction set architecture (**ISA**).

**Machine code** : Code that the **CPU** directly processes. Each instruction is usually encoded by several bytes.

**Assembly language** : Mnemonic code and some extensions like macros that are intended to make a programmer's life easier.

**CPU register** : Each **CPU** has a fixed set of general purpose registers (**GPR**).  $\approx 8$  in x86,  $\approx 16$  in x86-64,  $\approx 16$  in ARM. The easiest way to understand a register is to think of it as an untyped temporary variable . Imagine if you were working with a high-level **PL**<sup>1</sup> and could only use eight 32-bit (or 64-bit) variables . Yet a lot can be done using just these!

One might wonder why there needs to be a difference between machine code and a **PL**. The answer lies in the fact that humans and **CPUs** are not alike— it is much easier for humans to use a high-level **PL** like C/C++, Java, Python, etc., but it is easier for a **CPU** to use a much lower level of abstraction. Perhaps it would be possible to invent a **CPU** that can execute high-level **PL** code, but it would be many times more complex than the **CPUs** we know of today. In a similar fashion, it is very inconvenient for humans to write in assembly language, due to it being so low-level and difficult to write in without making a huge number of annoying mistakes. The program that converts the high-level **PL** code into assembly is called a *compiler*.

### 1.1 A couple of words about different **ISAs**

The x86 **ISA** has always been one with variable-length opcodes, so when the 64-bit era came, the x64 extensions did not impact the **ISA** very significantly. In fact, the x86 **ISA** still contains a lot of instructions that first appeared in 16-bit 8086 CPU, yet are still found in the CPUs of today.

ARM is a **RISC**<sup>2</sup> **CPU** designed with constant-length opcode in mind, which had some advantages in the past. In the very beginning, all ARM instructions were encoded in 4 bytes<sup>3</sup>. This is now referred to as “ARM mode”.

Then they thought it wasn't as frugal as they first imagined. In fact, most used **CPU** instructions<sup>4</sup> in real world applications can be encoded using less information. They therefore added another **ISA**, called Thumb, where each instruction was encoded in just 2 bytes. This is now referred as “Thumb mode”. However, not *all* ARM instructions can be encoded in just 2 bytes, so the Thumb instruction set is somewhat limited. It is worth noting that code compiled for ARM mode and Thumb mode may of course coexist within one single program.

The ARM creators thought Thumb could be extended, giving rise to Thumb-2, which appeared in ARMv7. Thumb-2 still uses 2-byte instructions, but has some new instructions which have the size of 4 bytes. There is a common misconception that Thumb-2 is a mix of ARM and Thumb. This is incorrect. Rather, Thumb-2 was extended to fully support all processor features so it could compete with ARM mode—a goal that was clearly achieved, as the majority of applications for iPod/iPhone/iPad are compiled for the Thumb-2 instruction set (admittedly, largely due to the fact that Xcode does this by default). Later the 64-bit ARM came out. This **ISA** has 4-byte opcodes, and lacked the need of any additional Thumb mode. However,

<sup>1</sup>Programming language

<sup>2</sup>Reduced instruction set computing

<sup>3</sup>By the way, fixed-length instructions are handy because one can calculate the next (or previous) instruction address without effort. This feature will be discussed in the `switch()` operator ( 13.2.2 on page 163) section.

<sup>4</sup>These are MOV/PUSH/CALL/Jcc

the 64-bit requirements affected the [ISA](#), resulting in us now having three ARM instruction sets: ARM mode, Thumb mode (including Thumb-2) and ARM64. These [ISAs](#) intersect partially, but it can be said that they are different [ISAs](#), rather than variations of the same one. Therefore, we would try to add fragments of code in all three ARM [ISAs](#) in this book.

There are, by the way, many other [RISC ISAs](#) with fixed length 32-bit opcodes, such as MIPS, PowerPC and Alpha AXP.

## Chapter 2

# The simplest Function

The simplest possible function is arguably one that simply returns a constant value:

Here it is:

Listing 2.1: C/C++ Code

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

Lets compile it!

## 2.1 x86

Here's what both the optimizing GCC and MSVC compilers produce on the x86 platform:

Listing 2.2: Optimizing GCC/MSVC (assembly output)

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

There are just two instructions: the first places the value 123 into the EAX register, which is used by convention for storing the return value and the second one is RET, which returns execution to the [caller](#). The caller will take the result from the EAX register.

## 2.2 ARM

There are a few differences on the ARM platform:

Listing 2.3: Optimizing Keil 6/2013 (ARM mode) ASM Output

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

ARM uses the register R0 for returning the results of functions, so 123 is copied into R0.

The return address is not saved on the local stack in the ARM [ISA](#), but rather in the link register, so the BX LR instruction causes execution to jump to that address—effectively returning execution to the [caller](#).

It is worth noting that MOV is a misleading name for the instruction in both x86 and ARM [ISAs](#). The data is not in fact *moved*, but *copied*.

## 2.3 MIPS

There are two naming conventions used in the world of MIPS when naming registers: by number (from \$0 to \$31) or by pseudoname (\$V0, \$A0, etc). The GCC assembly output below lists registers by number:

Listing 2.4: Optimizing GCC 4.4.5 (assembly output)

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

...while [IDA<sup>1</sup>](#) does it—by their pseudonames:

Listing 2.5: Optimizing GCC 4.4.5 (IDA)

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

The \$2 (or \$V0) register is used to store the function’s return value. LI stands for “Load Immediate” and is the MIPS equivalent to MOV.

The other instruction is the jump instruction (J or JR) which returns the execution flow to the [caller](#), jumping to the address in the \$31 (or \$RA) register. This is the register analogous to [LR<sup>2</sup>](#) in ARM.

You might be wondering why positions of the the load instruction (LI) and the jump instruction (J or JR) are swapped. This is due to a [RISC](#) feature called “branch delay slot”. The reason this happens is a quirk in the architecture of some RISC [ISAs](#) and isn’t important for our purposes - we just need to remember that in MIPS, the instruction following a jump or branch instruction is executed *before* the jump/branch instruction itself. As a consequence, branch instructions always swap places with the instruction which must be executed beforehand.

### 2.3.1 A note about MIPS instruction/register names

Register and instruction names in the world of MIPS are traditionally written in lowercase. However, for the sake of consistency, we’ll stick to using uppercase letters, as it is the convention followed by all other [ISAs](#) featured this book.

<sup>1</sup>Interactive Disassembler and debugger developed by [Hex-Rays](#)

<sup>2</sup>Link Register

## Chapter 3

# Hello, world!

Let's use the famous example from the book "The C programming Language"[[Ker88](#)]:

```
#include <stdio.h>

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

### 3.1 x86

#### 3.1.1 MSVC

Let's compile it in MSVC 2010:

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

(/Fa option instructs the compiler to generate assembly listing file)

Listing 3.1: 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 produces assembly listings in Intel-syntax. The difference between Intel-syntax and AT&T-syntax will be discussed in [3.1.3 on page 9](#).

The compiler generated the file, `1.obj`, which is to be linked into `1.exe`. In our case, the file contains two segments: `CONST` (for data constants) and `_TEXT` (for code).

The string `hello, world` in C/C++ has type `const char[]` [[Str13](#), p176, 7.3.2], but it does not have its own name. The compiler needs to deal with the string somehow so it defines the internal name `$SG3830` for it.

That is why the example may be rewritten as follows:

```
#include <stdio.h>

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

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

Let's go back to the assembly listing. As we can see, the string is terminated by a zero byte, which is standard for C/C++ strings. More about C strings: [57.1.1 on page 631](#).

In the code segment, `_TEXT`, there is only one function so far: `main()`. The function `main()` starts with prologue code and ends with epilogue code (like almost any function)<sup>1</sup>.

After the function prologue we see the call to the `printf()` function: `CALL _printf`. Before the call the string address (or a pointer to it) containing our greeting is placed on the stack with the help of the `PUSH` instruction.

When the `printf()` function returns the control to the `main()` function, the string address (or a pointer to it) is still on the stack. Since we do not need it anymore, the [stack pointer](#) (the `ESP` register) needs to be corrected.

`ADD ESP, 4` means add 4 to the `ESP` register value. Why 4? Since this is a 32-bit program, we need exactly 4 bytes for address passing through the stack. If it was x64 code we would need 8 bytes. `ADD ESP, 4` is effectively equivalent to `POP register` but without using any register<sup>2</sup>.

For the same purpose, some compilers (like the Intel C++ Compiler) may emit `POP ECX` instead of `ADD` (e.g., such a pattern can be observed in the Oracle RDBMS code as it is compiled with the Intel C++ compiler). This instruction has almost the same effect but the `ECX` register contents will be overwritten. The Intel C++ compiler probably uses `POP ECX` since this instruction's opcode is shorter than `ADD ESP, x` (1 byte for `POP` against 3 for `ADD`).

Here is an example of using `POP` instead of `ADD` from Oracle RDBMS:

Listing 3.2: Oracle RDBMS 10.2 Linux (app.o file)

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

After calling `printf()`, the original C/C++ code contains the statement `return 0`—return 0 as the result of the `main()` function. In the generated code this is implemented by the instruction `XOR EAX, EAX`. `XOR` is in fact just “eXclusive OR”<sup>3</sup> but the compilers often use it instead of `MOV EAX, 0`—again because it is a slightly shorter opcode (2 bytes for `XOR` against 5 for `MOV`).

Some compilers emit `SUB EAX, EAX`, which means *SUBtract the value in the EAX from the value in EAX*, which, in any case, results in zero.

The last instruction `RET` returns the control to the [caller](#). Usually, this is C/C++ [CRT](#)<sup>4</sup> code, which, in turn, returns control to the [OS](#).

### 3.1.2 GCC

Now let's try to compile the same C/C++ code in the GCC 4.4.1 compiler in Linux: `gcc 1.c -o 1`. Next, with the assistance of the [IDA](#) disassembler, let's see how the `main()` function was created. [IDA](#), like `MSVC`, uses Intel-syntax<sup>5</sup>.

Listing 3.3: code in [IDA](#)

main	proc near
var_10	= dword ptr -10h
	push ebp
	mov ebp, esp
	and esp, 0FFFFFF0h
	sub esp, 10h
	mov eax, offset aHelloWorld ; "hello, world\n"

<sup>1</sup>You can read more about it in the section about function prologues and epilogues ([4 on page 23](#)).

<sup>2</sup>CPU flags, however, are modified

<sup>3</sup>[wikipedia](#)

<sup>4</sup>C runtime library : [68.1 on page 668](#)

<sup>5</sup>We could also have GCC produce assembly listings in Intel-syntax by applying the options `-S -masm=intel`.

```

    mov     [esp+10h+var_10], eax
    call    _printf
    mov     eax, 0
    leave
    retn
main      endp

```

The result is almost the same. The address of the `hello, world` string (stored in the data segment) is loaded in the EAX register first and then it is saved onto the stack. In addition, the function prologue contains `AND ESP, 0FFFFFFF0h` –this instruction aligns the ESP register value on a 16-byte boundary. This results in all values in the stack being aligned the same way (The CPU performs better if the values it is dealing with are located in memory at addresses aligned on a 4-byte or 16-byte boundary)<sup>6</sup>.

`SUB ESP, 10h` allocates 16 bytes on the stack. Although, as we can see hereafter, only 4 are necessary here.

This is because the size of the allocated stack is also aligned on a 16-byte boundary.

The string address (or a pointer to the string) is then stored directly onto the stack without using the `PUSH` instruction. `var_10` –is a local variable and is also an argument for `printf()`. Read about it below.

Then the `printf()` function is called.

Unlike MSVC, when GCC is compiling without optimization turned on, it emits `MOV EAX, 0` instead of a shorter opcode.

The last instruction, `LEAVE` –is the equivalent of the `MOV ESP, EBP` and `POP EBP` instruction pair –in other words, this instruction sets the [stack pointer](#) (ESP) back and restores the EBP register to its initial state. This is necessary since we modified these register values (ESP and EBP) at the beginning of the function (by executing `MOV EBP, ESP / AND ESP, ...`).

### 3.1.3 GCC: AT&T syntax

Let's see how this can be represented in assembly language AT&T syntax. This syntax is much more popular in the UNIX-world.

Listing 3.4: let's compile in GCC 4.7.3

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

We get this:

Listing 3.5: 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
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"

```

<sup>6</sup>[Wikipedia: Data structure alignment](#)

<code>.section</code>	<code>.note.GNU-stack,"",@progbits</code>
-----------------------	---

The listing contains many macros (beginning with dot). These are not interesting for us at the moment. For now, for the sake of simplification, we can ignore them (except the `.string` macro which encodes a null-terminated character sequence just like a C-string). Then we'll see this<sup>7</sup>:

Listing 3.6: 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
```

Some of the major differences between Intel and AT&T syntax are:

- Source and destination operands are written in opposite order.

In Intel-syntax: <instruction> <destination operand> <source operand>.

In AT&T syntax: <instruction> <source operand> <destination operand>.

Here is an easy way to memorise the difference: when you deal with Intel-syntax, you can imagine that there is an equality sign (=) between operands and when you deal with AT&T-syntax imagine there is a right arrow (→)<sup>8</sup>.

- AT&T: Before register names, a percent sign must be written (%) and before numbers a dollar sign (\$). Parentheses are used instead of brackets.
- AT&T: A suffix is added to instructions to define the operand size:
  - q – quad (64 bits)
  - l – long (32 bits)
  - w – word (16 bits)
  - b – byte (8 bits)

Let's go back to the compiled result: it is identical to what we saw in [IDA](#). With one subtle difference: 0FFFFFFF0h is presented as \$-16. It is the same thing: 16 in the decimal system is 0x10 in hexadecimal. -0x10 is equal to 0xFFFFFFF0 (for a 32-bit data type).

One more thing: the return value is to be set to 0 by using the usual MOV, not XOR. MOV just loads a value to a register. Its name is a misnomer (data is not moved but rather copied). In other architectures, this instruction is named "LOAD" or "STORE" or something similar.

## 3.2 x86-64

### 3.2.1 MSVC-x86-64

Let's also try 64-bit MSVC:

Listing 3.7: 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
```

<sup>7</sup>This GCC option can be used to eliminate "unnecessary" macros: `-fno-asynchronous-unwind-tables`

<sup>8</sup>By the way, in some C standard functions (e.g., `memcpy()`, `strcpy()`) the arguments are listed in the same way as in Intel-syntax: first the pointer to the destination memory block, and then the pointer to the source memory block.



```

main    ret    0
        ENDP

```

In x86-64, all registers were extended to 64-bit and now their names have an R- prefix. In order to use the stack less often (in other words, to access external memory/cache less often), there exists a popular way to pass function arguments via registers (fastcall: [64.3 on page 649](#)). I.e., a part of the function arguments is passed in registers, the rest—via the stack. In Win64, 4 function arguments are passed in the RCX, RDX, R8, R9 registers. That is what we see here: a pointer to the string for `printf()` is now passed not in the stack, but in the RCX register.

The pointers are 64-bit now, so they are passed in the 64-bit registers (which have the R- prefix). However, for backward compatibility, it is still possible to access the 32-bit parts, using the E- prefix.

This is how the RAX/EAX/AX/AL register looks like in x86-64:

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RAX <sup>x64</sup>							
				EAX			
						AX	
						AH	AL

The `main()` function returns an *int*-typed value, which is, in C/C++, for better backward compatibility and portability, still 32-bit, so that is why the EAX register is cleared at the function end (i.e., the 32-bit part of the register) instead of RAX.

There are also 40 bytes allocated in the local stack. This is called the “shadow space”, about which we are going to talk later: [8.2.1 on page 92](#).

### 3.2.2 GCC—x86-64

Let's also try GCC in 64-bit Linux:

Listing 3.8: GCC 4.4.6 x64

```

.string "hello, world\n"
main:
    sub    rsp, 8
    mov    edi, OFFSET FLAT:.LC0 ; "hello, world\n"
    xor    eax, eax ; number of vector registers passed
    call   printf
    xor    eax, eax
    add    rsp, 8
    ret

```

A method to pass function arguments in registers is also used in Linux, \*BSD and Mac OS X [\[Mit13\]](#). The first 6 arguments are passed in the RDI, RSI, RDX, RCX, R8, R9 registers, and the rest—via the stack.

So the pointer to the string is passed in EDI (the 32-bit part of the register). But why not use the 64-bit part, RDI?

It is important to keep in mind that all MOV instructions in 64-bit mode that write something into the lower 32-bit register part also clear the higher 32-bits [\[Int13\]](#). I.e., the MOV EAX, 011223344h writes a value into RAX correctly, since the higher bits will be cleared.

If we open the compiled object file (.o), we can also see all the instructions' opcodes<sup>9</sup>:

Listing 3.9: 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

```

As we can see, the instruction that writes into EDI at 0x4004D4 occupies 5 bytes. The same instruction writing a 64-bit value into RDI occupies 7 bytes. Apparently, GCC is trying to save some space. Besides, it can be sure that the data segment containing the string will not be allocated at the addresses higher than 4GiB.

We also see that the EAX register was cleared before the `printf()` function call. This is done because the number of used vector registers is passed in EAX in \*NIX systems on x86-64 ([\[Mit13\]](#)).

<sup>9</sup>This must be enabled in Options → Disassembly → Number of opcode bytes

### 3.3 GCC—one more thing

The fact that an *anonymous* C-string has *const* type ([3.1.1 on page 7](#)), and that C-strings allocated in constants segment are guaranteed to be immutable, has an interesting consequence: the compiler may use a specific part of the string.

Let's try this example:

```
#include <stdio.h>

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

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

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

Common C/C++-compilers (including MSVC) allocate two strings, but let's see what GCC 4.8.1 does:

Listing 3.10: GCC 4.8.1 + IDA listing

```
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

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
```

Indeed: when we print the “hello world” string, these two words are positioned in memory adjacently and `puts()` called from `f2()` function is not aware that this string is divided. In fact, it's not divided; it's divided only “virtually”, in this listing.

When `puts()` is called from `f1()`, it uses the “world” string plus a zero byte. `puts()` is not aware that there is something before this string!

This clever trick is often used by at least GCC and can save some memory.

### 3.4 ARM

For my experiments with ARM processors, several compilers were used:

- Popular in the embedded area: Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE (with the LLVM-GCC 4.2 compiler<sup>10</sup>).

<sup>10</sup>It is indeed so: Apple Xcode 4.6.3 uses open-source GCC as front-end compiler and LLVM code generator

- GCC 4.9 (Linaro) (for ARM64), available as win32-executables at <http://go.yurichev.com/17325>.

32-bit ARM code is used (including Thumb and Thumb-2 modes) in all cases in this book, if not mentioned otherwise. When we talk about 64-bit ARM here, we call it ARM64.

### 3.4.1 Non-optimizing Keil 6/2013 (ARM mode)

Let's start by compiling our example in Keil:

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

The *armcc* compiler produces assembly listings in Intel-syntax but it has high-level ARM-processor related macros<sup>11</sup>, but it is more important for us to see the instructions “as is” so let's see the compiled result in *IDA*.

Listing 3.11: Non-optimizing Keil 6/2013 (ARM mode) *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
```

In the example, we can easily see each instruction has a size of 4 bytes. Indeed, we compiled our code for ARM mode, not for Thumb.

The very first instruction, `STMFD SP!, {R4,LR}`<sup>12</sup>, works as an x86 PUSH instruction, writing the values of two registers (R4 and LR) into the stack. Indeed, in the output listing from the *armcc* compiler, for the sake of simplification, actually shows the `PUSH {r4,lr}` instruction. But that is not quite precise. The PUSH instruction is only available in Thumb mode. So, to make things less confusing, we're doing this in *IDA*.

This instruction first *decrements* the *SP*<sup>14</sup> so it points to the place in the stack that is free for new entries, then it saves the values of the R4 and LR registers at the address stored in the modified *SP*.

This instruction (like the PUSH instruction in Thumb mode) is able to save several register values at once which can be very useful. By the way, this has no equivalent in x86. It can also be noted that the STMFD instruction is a generalization of the PUSH instruction (extending its features), since it can work with any register, not just with *SP*. In other words, STMFD may be used for storing a set of registers at the specified memory address.

The `ADR R0, aHelloWorld` instruction adds or subtracts the value in the *PC*<sup>15</sup> register to the offset where the `hello, world` string is located. How is the PC register used here, one might ask? This is called “position-independent code”.<sup>16</sup> Such code can be executed at a non-fixed address in memory. In other words, this is *PC*-relative addressing. The ADR instruction takes into account the difference between the address of this instruction and the address where the string is located. This difference (offset) is always to be the same, no matter at what address our code is loaded by the *OS*. That's why all we need is to add the address of the current instruction (from *PC*) in order to get the absolute memory address of our C-string.

`BL __2printf`<sup>17</sup> instruction calls the `printf()` function. Here's how this instruction works:

- store the address following the BL instruction (0xC) into the *LR*;
- then pass the control to `printf()` by writing its address into the *PC* register.

When `printf()` finishes its execution it must have information about where it needs to return the control to. That's why each function passes control to the address stored in the *LR* register.

That is a difference between “pure” *RISC*-processors like ARM and *CISC*<sup>18</sup>-processors like x86, where the return address is usually stored on the stack<sup>19</sup>.

By the way, an absolute 32-bit address or offset cannot be encoded in the 32-bit BL instruction because it only has space for 24 bits. As we may remember, all ARM-mode instructions have a size of 4 bytes (32 bits). Hence, they can only be located

<sup>11</sup>e.g. ARM mode lacks PUSH/POP instructions

<sup>12</sup>*STMFD*<sup>13</sup>

<sup>14</sup>*stack pointer*. SP/ESP/RSP in x86/x64. SP in ARM.

<sup>15</sup>Program Counter. IP/EIP/RIP in x86/64. PC in ARM.

<sup>16</sup>Read more about it in relevant section ( [6.7.1 on page 663](#) )

<sup>17</sup>Branch with Link

<sup>18</sup>Complex instruction set computing

<sup>19</sup>Read more about this in next section ( [5 on page 24](#) )

on 4-byte boundary addresses. This implies that the last 2 bits of the instruction address (which are always zero bits) may be omitted. In summary, we have 26 bits for offset encoding. This is enough to encode  $current\_PC \pm \approx 32M$ .

Next, the `MOV R0, #020` instruction just writes 0 into the R0 register. That's because our C-function returns 0 and the return value is to be placed in the R0 register.

The last instruction `LDMFD SP!, R4, PC21` is an inverse instruction of `STMFD`. It loads values from the stack (or any other memory place) in order to save them into R4 and PC, and increments the stack pointer SP. It works like POP here.

N.B. The very first instruction `STMFD` saved the R4 and LR registers pair on the stack, but R4 and PC are restored during the `LDMFD` execution.

As we already know, the address of the place where each function must return control to is usually saved in the LR register. The very first instruction saves its value in the stack because the same register will be used by our `main()` function when calling `printf()`. In the function's end, this value can be written directly to the PC register, thus passing control to where our function was called.

Since `main()` is usually the primary function in C/C++, the control will be returned to the OS loader or to a point in a CRT, or something like that.

All that allows omitting the `BX LR` instruction at the end of the function.

DCB is an assembly language directive defining an array of bytes or ASCII strings, akin to the DB directive in the x86-assembly language.

### 3.4.2 Non-optimizing Keil 6/2013 (Thumb mode)

Let's compile the same example using Keil in Thumb mode:

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

We are getting (in IDA):

Listing 3.12: Non-optimizing Keil 6/2013 (Thumb mode) + 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
```

We can easily spot the 2-byte (16-bit) opcodes. This is, as was already noted, Thumb. The BL instruction, however, consists of two 16-bit instructions. This is because it is impossible to load an offset for the `printf()` function while using the small space in one 16-bit opcode. Therefore, the first 16-bit instruction loads the higher 10 bits of the offset and the second instruction loads the lower 11 bits of the offset. As was noted, all instructions in Thumb mode have a size of 2 bytes (or 16 bits). This implies it is impossible for a Thumb-instruction to be at an odd address whatsoever. Given the above, the last address bit may be omitted while encoding instructions. In summary, the BL Thumb-instruction can encode an address in  $current\_PC \pm \approx 2M$ .

As for the other instructions in the function: PUSH and POP work here just like the described `STMFD/LDMFD` only the SP register is not mentioned explicitly here. ADR works just like in the previous example. MOVS writes 0 into the R0 register in order to return zero.

### 3.4.3 Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Xcode 4.6.3 without optimization turned on produces a lot of redundant code so we'll study optimized output, where the instruction count is as small as possible, setting the compiler switch `-O3`.

Listing 3.13: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
__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
```

<sup>20</sup>MOVE

<sup>21</sup>LDMFD<sup>22</sup>

```

__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

```

The instructions STMFD and LDMFD are already familiar to us.

The MOV instruction just writes the number 0x1686 into the R0 register. This is the offset pointing to the “Hello world!” string.

The R7 register (as it is standardized in [App10]) is a frame pointer. More on that below.

The MOVT R0, #0 (MOVE Top) instruction writes 0 into higher 16 bits of the register. The issue here is that the generic MOV instruction in ARM mode may write only the lower 16 bits of the register. Remember, all instruction opcodes in ARM mode are limited in size to 32 bits. Of course, this limitation is not related to moving data between registers. That’s why an additional instruction MOVT exists for writing into the higher bits (from 16 to 31 inclusive). Its usage here, however, is redundant because the MOV R0, #0x1686 instruction above cleared the higher part of the register. This is probably a shortcoming of the compiler.

The ADD R0, PC, R0 instruction adds the value in the PC to the value in the R0, to calculate the absolute address of the “Hello world!” string. As we already know, it is “position-independent code” so this correction is essential here.

The BL instruction calls the puts() function instead of printf().

GCC replaced the first printf() call with puts(). Indeed: printf() with a sole argument is almost analogous to puts(). *Almost*, because the two functions are producing the same result only in case the string does not contain printf format identifiers starting with %. In case it does, the effect of these two functions would be different<sup>23</sup>.

Why did the compiler replace the printf() with puts()? Probably because puts() is faster<sup>24</sup>. Because it just passes characters to stdout without comparing every one of them with the % symbol.

Next, we see the familiar MOV R0, #0 instruction intended to set the R0 register to 0.

### 3.4.4 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

By default Xcode 4.6.3 generates code for Thumb-2 in this manner:

Listing 3.14: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```

__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

```

The BL and BLX instructions in Thumb mode, as we recall, are encoded as a pair of 16-bit instructions. In Thumb-2 these *surrogate* opcodes are extended in such a way so that new instructions may be encoded here as 32-bit instructions. That is obvious considering that the opcodes of the Thumb-2 instructions always begin with 0xFx or 0Ex. But in the IDA listing the opcode bytes are swapped because for ARM processor the instructions are encoded as follows: last byte comes first and after that comes the first one (for Thumb and Thumb-2 modes) or for instructions in ARM mode the fourth byte comes first, then the third, then the second and finally the first (due to different [endianness](#)).

So that is how bytes are located in IDA listings:

- for ARM and ARM64 modes: 4-3-2-1;
- for Thumb mode: 2-1;
- for 16-bit instructions pair in Thumb-2 mode: 2-1-4-3.

<sup>23</sup>It has also to be noted the puts() does not require a “\n” new line symbol at the end of a string, so we do not see it here.

<sup>24</sup>[ciselant.de/projects/gcc\\_printf/gcc\\_printf.html](http://ciselant.de/projects/gcc_printf/gcc_printf.html)

So as we can see, the MOVW, MOVT .W and BLX instructions begin with 0xFx.

One of the Thumb-2 instructions is MOVW R0, #0x13D8 –it stores a 16-bit value into the lower part of the R0 register, clearing the higher bits.

Also, MOVT .W R0, #0 works just like MOVT from the previous example only it works in Thumb-2.

Among the other differences, the BLX instruction is used in this case instead of the BL. The difference is that, besides saving the RA<sup>25</sup> in the LR register and passing control to the puts ( ) function, the processor is also switching from Thumb/Thumb-2 mode to ARM mode (or back). This instruction is placed here since the instruction to which control is passed looks like (it is encoded in ARM mode):

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

This is essentially a jump to the place where the address of puts ( ) is written in the imports' section.

So, the observant reader may ask: why not call puts ( ) right at the point in the code where it is needed?

Because it is not very space-efficient.

Almost any program uses external dynamic libraries (like DLL in Windows, .so in \*NIX or .dylib in Mac OS X). The dynamic libraries contain frequently used library functions, including the standard C-function puts ( ).

In an executable binary file (Windows PE .exe, ELF or Mach-O) an import section is present. This is a list of symbols (functions or global variables) imported from external modules along with the names of the modules themselves.

The OS loader loads all modules it needs and, while enumerating import symbols in the primary module, determines the correct addresses of each symbol.

In our case, \_\_imp\_puts is a 32-bit variable used by the OS loader to store the correct address of the function in an external library. Then the LDR instruction just reads the 32-bit value from this variable and writes it into the PC register, passing control to it.

So, in order to reduce the time the OS loader needs for completing this procedure, it is good idea to write the address of each symbol only once, to a dedicated place.

Besides, as we have already figured out, it is impossible to load a 32-bit value into a register while using only one instruction without a memory access. Therefore, the optimal solution is to allocate a separate function working in ARM mode with the sole goal of passing control to the dynamic library and then to jump to this short one-instruction function (the so-called **thunk function**) from the Thumb-code.

By the way, in the previous example (compiled for ARM mode) the control is passed by the BL to the same **thunk function**. The processor mode, however, is not being switched (hence the absence of an “X” in the instruction mnemonic).

### More about thunk-functions

Thunk-functions are hard to understand, apparently, because of a misnomer.

The simplest way to understand it as adaptors or convertors of one type of jack to another. For example, an adaptor allowing the insertion of a British power plug into an American wall socket, or vice-versa.

Thunk functions are also sometimes called *wrappers*.

Here are a couple more descriptions of these functions:

“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](#) )

<sup>25</sup>Return Address

### 3.4.5 ARM64

#### GCC

Let's compile the example using GCC 4.8.1 in ARM64:

Listing 3.15: Non-optimizing 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 ...
12
13 Contents of section .rodata:
14 400640 01000200 00000000 48656c6c 6f210a00 .....Hello!..

```

There are no Thumb and Thumb-2 modes in ARM64, only ARM, so there are 32-bit instructions only. The Register count is doubled: [B.4.1 on page 897](#). 64-bit registers have X- prefixes, while its 32-bit parts—W-.

The STP instruction (*Store Pair*) saves two registers in the stack simultaneously: X29 in X30. Of course, this instruction is able to save this pair at an arbitrary place in memory, but the SP register is specified here, so the pair is saved in the stack. ARM64 registers are 64-bit ones, each has a size of 8 bytes, so one needs 16 bytes for saving two registers.

The exclamation mark after the operand means that 16 is to be subtracted from SP first, and only then are values from register pair to be written into the stack. This is also called *pre-index*. About the difference between *post-index* and *pre-index* read here: [28.2 on page 425](#).

Hence, in terms of the more familiar x86, the first instruction is just an analogue to a pair of PUSH X29 and PUSH X30. X29 is used as FP<sup>26</sup> in ARM64, and X30 as LR, so that's why they are saved in the function prologue and restored in the function epilogue.

The second instruction copies SP in X29 (or FP). This is done to set up the function stack frame.

ADRP and ADD instructions are used to fill the address of the string “Hello!” into the X0 register, because the first function argument is passed in this register. There are no instructions, whatsoever, in ARM that can store a large number into a register (because the instruction length is limited to 4 bytes, read more about it here: [28.3.1 on page 426](#)). So several instructions must be utilised. The first instruction (ADRP) writes the address of the 4KiB page, where the string is located, into X0, and the second one (ADD) just adds the remainder to the address. More about that in: [28.4 on page 427](#).

$0x400000 + 0x648 = 0x400648$ , and we see our “Hello!” C-string in the .rodata data segment at this address.

puts() is called afterwards using the BL instruction. This was already discussed: [3.4.3 on page 15](#).

MOV writes 0 into W0. W0 is the lower 32 bits of the 64-bit X0 register:

High 32-bit part	low 32-bit part
X0	
	W0

The function result is returned via X0 and main() returns 0, so that's how the return result is prepared. But why use the 32-bit part? Because the *int* data type in ARM64, just like in x86-64, is still 32-bit, for better compatibility. So if a function returns a 32-bit *int*, only the lower 32 bits of X0 register have to be filled.

In order to verify this, let's change this example slightly and recompile it. Now main() returns a 64-bit value:

Listing 3.16: main() returning a value of uint64\_t type

```

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

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

```

<sup>26</sup>Frame Pointer



The result is the same, but that's how MOV at that line looks like now:

Listing 3.17: Non-optimizing GCC 4.8.1 + objdump

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

LDP (*Load Pair*) then restores the X29 and X30 registers. There is no exclamation mark after the instruction: this implies that the value is first loaded from the stack, and only then is [SP](#) increased by 16. This is called *post-index*.

A new instruction appeared in ARM64: RET. It works just as BX LR, only a special *hint* bit is added, informing the [CPU](#) that this is a return from a function, not just another jump instruction, so it can execute it more optimally.

Due to the simplicity of the function, optimizing GCC generates the very same code.

## 3.5 MIPS

### 3.5.1 A word about the “global pointer”

One important MIPS concept is the “global pointer”. As we may already know, each MIPS instruction has a size of 32 bits, so it's impossible to embed a 32-bit address into one instruction: a pair has to be used for this (like GCC did in our example for the text string address loading).

It's possible, however, to load data from the address in the range of  $register - 32768 \dots register + 32767$  using one single instruction (because 16 bits of signed offset could be encoded in a single instruction). So we can allocate some register for this purpose and also allocate a 64KiB area of most used data. This allocated register is called a “global pointer” and it points to the middle of the 64KiB area. This area usually contains global variables and addresses of imported functions like `printf()`, because the GCC developers decided that getting the address of some function must be as fast as a single instruction execution instead of two. In an ELF file this 64KiB area is located partly in sections `.sbss` (“small [BSS](#)<sup>27</sup>”) for uninitialized data and `.sdata` (“small data”) for initialized data.

This implies that the programmer may choose what data he/she wants to be accessed fast and place it into `.sdata/.sbss`.

Some old-school programmers may recall the MS-DOS memory model [94 on page 867](#) or the MS-DOS memory managers like XMS/EMS where all memory was divided in 64KiB blocks.

This concept is not unique to MIPS. At least PowerPC uses this technique as well.

### 3.5.2 Optimizing GCC

Lets consider the following example, which illustrates the “global pointer” concept.

Listing 3.18: Optimizing GCC 4.4.5 (assembly output)

```

1 $LC0:
2 ; \000 is zero byte in octal base:
3   .ascii "Hello, world!\012\000"
4 main:
5 ; function prologue.
6 ; set the GP:
7     lui     $28,%hi(__gnu_local_gp)
8     addiu   $sp,$sp,-32
9     addiu   $28,$28,%lo(__gnu_local_gp)
10 ; save the RA to the local stack:
11     sw      $31,28($sp)
12 ; load the address of the puts() function from the GP to $25:
13     lw      $25,%call16(puts)($28)
14 ; load the address of the text string to $4 ($a0):
15     lui     $4,%hi($LC0)
16 ; jump to puts(), saving the return address in the link register:
17     jalr    $25
18     addiu   $4,$4,%lo($LC0) ; branch delay slot
19 ; restore the RA:
20     lw      $31,28($sp)
21 ; copy 0 from $zero to $v0:
22     move    $2,$0
23 ; return by jumping to the RA:
24     j       $31

```

<sup>27</sup>Block Started by Symbol



```

25 ; function epilogue:
26     addiu    $sp,$sp,32 ; branch delay slot

```

As we see, the `$GP` register is set in the function prologue to point to the middle of this area. The `RA` register is also saved in the local stack. `puts()` is also used here instead of `printf()`. The address of the `puts()` function is loaded into `$25` using `LW` the instruction (“Load Word”). Then the address of the text string is loaded to `$4` using `LUI` (“Load Upper Immediate”) and `ADDIU` (“Add Immediate Unsigned Word”) instruction pair. `LUI` sets the high 16 bits of the register (hence “upper” word in instruction name) and `ADDIU` adds the lower 16 bits of the address. `ADDIU` follows `JALR` (remember *branch delay slots*?). The register `$4` is also called `$A0`, which is used for passing the first function argument<sup>28</sup>.

`JALR` (“Jump and Link Register”) jumps to the address stored in the `$25` register (address of `puts()`) while saving the address of the next instruction (`LW`) in `RA`. This is very similar to ARM. Oh, and one important thing is that the address saved in `RA` is not the address of the next instruction (because it’s in a *delay slot* and is executed before the jump instruction), but the address of the instruction after the next one (after the *delay slot*). Hence, `PC + 8` is written to `RA` during the execution of `JALR`, in our case, this is the address of the `LW` instruction next to `ADDIU`.

`LW` (“Load Word”) at line 20 restores `RA` from the local stack (this instruction is actually part of the function epilogue).

`MOVE` at line 22 copies the value from the `$0` (`$ZERO`) register to `$2` (`$V0`). MIPS has a *constant* register, which always holds zero. Apparently, the MIPS developers came up with the idea that zero is in fact the busiest constant in the computer programming, so let’s just use the `$0` register every time zero is needed. Another interesting fact is that MIPS lacks an instruction that transfers data between registers. In fact, `MOVE DST, SRC` is `ADD DST, SRC, $ZERO` ( $DST = SRC + 0$ ), which does the same. Apparently, the MIPS developers wanted to have a compact opcode table. This does not mean an actual addition happens at each `MOVE` instruction. Most likely, the `CPU` optimizes these pseudoinstructions and the `ALU`<sup>29</sup> is never used.

`J` at line 24 jumps to the address in `RA`, which is effectively performing a return from the function. `ADDIU` after `J` is in fact executed before `J` (remember *branch delay slots*?) and is part of the function epilogue.

Here is also a listing generated by `IDA`. Each register here has its own pseudoname:

Listing 3.19: Optimizing 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  ; function prologue.
7  ; set the 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 ; save the RA to the local stack:
12 .text:0000000C          sw     $ra, 0x20+var_4($sp)
13 ; save the GP to the local stack:
14 ; for some reason, this instruction is missing in the GCC assembly output:
15 .text:00000010          sw     $gp, 0x20+var_10($sp)
16 ; load the address of the puts() function from the GP to $t9:
17 .text:00000014          lw     $t9, (puts & 0xFFFF)($gp)
18 ; form the address of the text string in $a0:
19 .text:00000018          lui    $a0, ($LC0 >> 16) # "Hello, world!"
20 ; jump to puts(), saving the return address in the link register:
21 .text:0000001C          jalr   $t9
22 .text:00000020          la     $a0, ($LC0 & 0xFFFF) # "Hello, world!"
23 ; restore the RA:
24 .text:00000024          lw     $ra, 0x20+var_4($sp)
25 ; copy 0 from $zero to $v0:
26 .text:00000028          move   $v0, $zero
27 ; return by jumping to the RA:
28 .text:0000002C          jr     $ra
29 ; function epilogue:
30 .text:00000030          addiu  $sp, 0x20

```

The instruction at line 15 saves the `GP` value into the local stack, and this instruction is missing mysteriously from the GCC output listing, maybe by a GCC error<sup>30</sup>. The `GP` value has to be saved indeed, because each function can use its own 64KiB data window.

<sup>28</sup>The MIPS registers table is available in appendix C.1 on page 899

<sup>29</sup>Arithmetic logic unit

<sup>30</sup>Apparently, functions generating listings are not so critical to GCC users, so some unfixed errors may still exist.

The register containing the `puts()` address is called `$T9`, because registers prefixed with T- are called “temporaries” and their contents may not be preserved.

### 3.5.3 Non-optimizing GCC

Non-optimizing GCC is more verbose.

Listing 3.20: Non-optimizing GCC 4.4.5 (assembly output)

```

1 $LC0:
2     .ascii "Hello, world!\012\000"
3 main:
4 ; function prologue.
5 ; save the RA ($31) and FP in the stack:
6     addiu    $sp,$sp,-32
7     sw       $31,28($sp)
8     sw       $fp,24($sp)
9 ; set the FP (stack frame pointer):
10    move     $fp,$sp
11 ; set the GP:
12    lui      $28,%hi(__gnu_local_gp)
13    addiu    $28,$28,%lo(__gnu_local_gp)
14 ; load the address of the text string:
15    lui      $2,%hi($LC0)
16    addiu    $4,$2,%lo($LC0)
17 ; load the address of puts() using the GP:
18    lw       $2,%call16(puts)($28)
19    nop
20 ; call puts():
21    move     $25,$2
22    jalr     $25
23    nop ; branch delay slot
24
25 ; restore the GP from the local stack:
26    lw       $28,16($fp)
27 ; set register $2 ($V0) to zero:
28    move     $2,$0
29 ; function epilogue.
30 ; restore the SP:
31    move     $sp,$fp
32 ; restore the RA:
33    lw       $31,28($sp)
34 ; restore the FP:
35    lw       $fp,24($sp)
36    addiu    $sp,$sp,32
37 ; jump to the RA:
38    j        $31
39    nop ; branch delay slot

```

We see here that register FP is used as a pointer to the stack frame. We also see 3 `NOP`<sup>31</sup>s. The second and third of which follow the branch instructions.

Perhaps, the GCC compiler always adds `NOP`s (because of *branch delay slots*) after branch instructions and then, if optimization is turned on, maybe eliminates them. So in this case they are left here.

Here is also `IDA` listing:

Listing 3.21: Non-optimizing 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 ; function prologue.
8 ; save the RA and FP in the stack:
9 .text:00000000          addiu    $sp, -0x20
10 .text:00000004          sw       $ra, 0x20+var_4($sp)

```

<sup>31</sup>No OPeration

```

11 .text:00000008      sw      $fp, 0x20+var_8($sp)
12 ; set the FP (stack frame pointer):
13 .text:0000000C      move     $fp, $sp
14 ; set the GP:
15 .text:00000010      la       $gp, __gnu_local_gp
16 .text:00000018      sw      $gp, 0x20+var_10($sp)
17 ; load the address of the text string:
18 .text:0000001C      lui      $v0, (aHelloWorld >> 16) # "Hello, world!"
19 .text:00000020      addiu    $a0, $v0, (aHelloWorld & 0xFFFF) # "Hello, world!"
20 ; load the address of puts() using the GP:
21 .text:00000024      lw       $v0, (puts & 0xFFFF)($gp)
22 .text:00000028      or       $at, $zero ; NOP
23 ; call puts():
24 .text:0000002C      move     $t9, $v0
25 .text:00000030      jalr     $t9
26 .text:00000034      or       $at, $zero ; NOP
27 ; restore the GP from local stack:
28 .text:00000038      lw       $gp, 0x20+var_10($fp)
29 ; set register $2 ($V0) to zero:
30 .text:0000003C      move     $v0, $zero
31 ; function epilogue.
32 ; restore the SP:
33 .text:00000040      move     $sp, $fp
34 ; restore the RA:
35 .text:00000044      lw       $ra, 0x20+var_4($sp)
36 ; restore the FP:
37 .text:00000048      lw       $fp, 0x20+var_8($sp)
38 .text:0000004C      addiu    $sp, 0x20
39 ; jump to the RA:
40 .text:00000050      jr       $ra
41 .text:00000054      or       $at, $zero ; NOP

```

Interestingly, [IDA](#) recognized the LUI/ADDIU instructions pair and coalesced them into one LA (“Load Address”) pseudoinstruction at line 15. We may also see that this pseudoinstruction has a size of 8 bytes! This is a pseudoinstruction (or *macro*) because it’s not a real MIPS instruction, but rather a handy name for an instruction pair.

Another thing is that [IDA](#) doesn’t recognize [NOP](#) instructions, so here they are at lines 22, 26 and 41. It is OR \$AT, \$ZERO. Essentially, this instruction applies the OR operation to the contents of the \$AT register with zero, which is, of course, an idle instruction. MIPS, like many other [ISAs](#), doesn’t have a separate [NOP](#) instruction.

### 3.5.4 Role of the stack frame in this example

The address of the text string is passed in the register. Why setup a local stack anyway? The reason for this lies in the fact that the values of registers [RA](#) and [GP](#) have to be saved somewhere (because `printf()` is called), and the local stack is used for this purpose. If this was a [leaf function](#), it would have been possible to get rid of the function prologue and epilogue, for example: [2.3 on page 6](#).

### 3.5.5 Optimizing GCC: load it into GDB

Listing 3.22: sample GDB session

```

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
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)

```

## 3.6 Conclusion

The main difference between x86/ARM and x64/ARM64 code is that the pointer to the string is now 64-bits in length. Indeed, modern CPUs are now 64-bit due to both the reduced cost of memory and the greater demand for it by modern applications. We can add much more memory to our computers than 32-bit pointers are able to address. As such, all pointers are now 64-bit.

## 3.7 Exercises

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

## Chapter 4

# Function prologue and epilogue

A function prologue is a sequence of instructions at the start of a function. It often looks something like the following code fragment:

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

What these instructions do: save the value in the EBP register, set the value of the EBP register to the value of the ESP and then allocate space on the stack for local variables.

The value in the EBP stays the same over the period of the function execution and is to be used for local variables and arguments access. For the same purpose one can use ESP, but since it changes over time this approach is not too convenient.

The function epilogue frees the allocated space in the stack, returns the value in the EBP register back to its initial state and returns the control flow to the [callee](#):

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

Function prologues and epilogues are usually detected in disassemblers for function delimitation.

### 4.1 Recursion

Epilogues and prologues can negatively affect the recursion performance.

More about recursion in this book: [36.3 on page 451](#).

## Chapter 5

# Stack

The stack is one of the most fundamental data structures in computer science<sup>1</sup>.

Technically, it is just a block of memory in process memory along with the ESP or RSP register in x86 or x64, or the SP register in ARM, as a pointer within that block.

The most frequently used stack access instructions are PUSH and POP (in both x86 and ARM Thumb-mode). PUSH subtracts from ESP/RSP/SP 4 in 32-bit mode (or 8 in 64-bit mode) and then writes the contents of its sole operand to the memory address pointed by ESP/RSP/SP.

POP is the reverse operation: retrieve the data from the memory location that SP points to, load it into the instruction operand (often a register) and then add 4 (or 8) to the stack pointer.

After stack allocation, the stack pointer points at the bottom of the stack. PUSH decreases the stack pointer and POP increases it. The bottom of the stack is actually at the beginning of the memory allocated for the stack block. It seems strange, but that's the way it is.

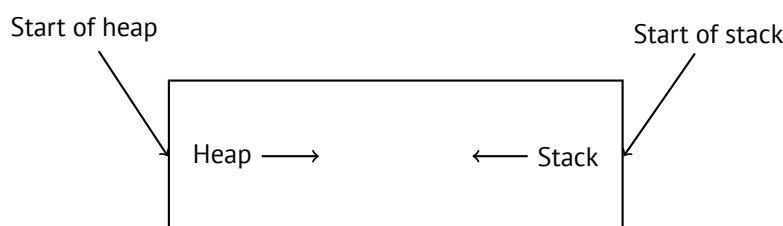
ARM supports both descending and ascending stacks.

For example the STMFD/LDMFD, STMED<sup>2</sup>/LDMED<sup>3</sup> instructions are intended to deal with a descending stack (grows downwards, starting with a high address and progressing to a lower one). The STMFA<sup>4</sup>/LDMFA<sup>5</sup>, STMEA<sup>6</sup>/LDMEA<sup>7</sup> instructions are intended to deal with an ascending stack (grows upwards, starting from a low address and progressing to a higher one).

### 5.1 Why does the stack grow backwards?

Intuitively, we might think that the stack grows upwards, i.e. towards higher addresses, like any other data structure.

The reason that the stack grows backward is probably historical. When the computers were big and occupied a whole room, it was easy to divide memory into two parts, one for the heap and one for the stack. Of course, it was unknown how big the heap and the stack would be during program execution, so this solution was the simplest possible.



In [RT74] we can read:

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

<sup>1</sup>[wikipedia.org/wiki/Call\\_stack](https://wikipedia.org/wiki/Call_stack)

<sup>2</sup>Store Multiple Empty Descending (ARM instruction)

<sup>3</sup>Load Multiple Empty Descending (ARM instruction)

<sup>4</sup>Store Multiple Full Ascending (ARM instruction)

<sup>5</sup>Load Multiple Full Ascending (ARM instruction)

<sup>6</sup>Store Multiple Empty Ascending (ARM instruction)

<sup>7</sup>Load Multiple Empty Ascending (ARM instruction)

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.

This reminds us how some students write two lecture notes using only one notebook: notes for the first lecture are written as usual, and notes for the second one are written from the end of notebook, by flipping it. Notes may meet each other somewhere in between, in case of lack of free space.

## 5.2 What is the stack used for?

### 5.2.1 Save the function's return address

#### x86

When calling another function with a `CALL` instruction, the address of the point exactly after the `CALL` instruction is saved to the stack and then an unconditional jump to the address in the `CALL` operand is executed.

The `CALL` instruction is equivalent to a `PUSH address_after_call / JMP operand` instruction pair.

`RET` fetches a value from the stack and jumps to it—that is equivalent to a `POP tmp / JMP tmp` instruction pair.

Overflowing the stack is straightforward. Just run eternal recursion:

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

MSVC 2008 reports the problem:

```
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
```

...but generates the right code anyway:

```
?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
```

... Also if we turn on the compiler optimization (`/Ox` option) the optimized code will not overflow the stack and will work *correctly*<sup>8</sup> instead:

```
?f@@YAXXZ PROC                                ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
    jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                                ; f
```

GCC 4.4.1 generates similar code in both cases without, however, issuing any warning about the problem.

<sup>8</sup>irony here

## ARM

ARM programs also use the stack for saving return addresses, but differently. As mentioned in “Hello, world!” (3.4 on page 12), the **RA** is saved to the **LR** (link register). If one needs, however, to call another function and use the **LR** register one more time, its value has to be saved. Usually it is saved in the function prologue. Often, we see instructions like `PUSH R4-R7, LR` along with this instruction in epilogue `POP R4-R7, PC` –thus register values to be used in the function are saved in the stack, including **LR**.

Nevertheless, if a function never calls any other function, in **RISC** terminology it is called a *leaf function*<sup>9</sup>. As a consequence, leaf functions do not save the **LR** register (because they don’t modify it). If such function is small and uses a small number of registers, it may not use the stack at all. Thus, it is possible to call leaf functions without using the stack, which can be faster than on older x86 machines because external RAM is not used for the stack<sup>10</sup>. This can be also useful for situations when memory for the stack is not yet allocated or not available.

Some examples of leaf functions: 8.3.2 on page 95, 8.3.3 on page 95, 19.17 on page 300, 19.33 on page 317, 19.5.4 on page 318, 15.4 on page 197, 15.2 on page 196, 17.3 on page 214.

## 5.2.2 Passing function arguments

The most popular way to pass parameters in x86 is called “cdecl”:

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

**Callee** functions get their arguments via the stack pointer.

Therefore, this is how the argument values are located in the stack before the execution of the `f()` function’s very first instruction:

ESP	return address
ESP+4	argument#1, marked in <b>IDA</b> as <code>arg_0</code>
ESP+8	argument#2, marked in <b>IDA</b> as <code>arg_4</code>
ESP+0xC	argument#3, marked in <b>IDA</b> as <code>arg_8</code>
...	...

For more information on other calling conventions see also section (64 on page 648). It is worth noting that nothing obliges programmers to pass arguments through the stack. It is not a requirement. One could implement any other method without using the stack at all.

For example, it is possible to allocate a space for arguments in the **heap**, fill it and pass it to a function via a pointer to this block in the **EAX** register. This will work<sup>11</sup>. However, it is a convenient custom in x86 and ARM to use the stack for this purpose.

By the way, the **callee** function does not have any information about how many arguments were passed. C functions with a variable number of arguments (like `printf()`) determine their number using format string specifiers (which begin with the `%` symbol). If we write something like

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

`printf()` will print 1234, and then two random numbers, which were lying next to it in the stack.

That’s why it is not very important how we declare the `main()` function: as `main()`, `main(int argc, char *argv[])` or `main(int argc, char *argv[], char *envp[])`.

In fact, the **CRT**-code is calling `main()` roughly as:

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

<sup>9</sup>[infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html)

<sup>10</sup>Some time ago, on PDP-11 and VAX, the `CALL` instruction (calling other functions) was expensive; up to 50% of execution time might be spent on it, so it was considered that having a big number of small functions is an **anti-pattern** [Ray03, Chapter 4, Part II].

<sup>11</sup>For example, in the “The Art of Computer Programming” book by Donald Knuth, in section 1.4.1 dedicated to subroutines [Knu98, section 1.4.1], we could read that one way to supply arguments to a subroutine is simply to list them after the `JMP` instruction passing control to subroutine. Knuth explains that this method was particularly convenient on IBM System/360.



If you declare `main()` as `main()` without arguments, they are, nevertheless, still present in the stack, but are not used. If you declare `main()` as `main(int argc, char *argv[])`, you will be able to use first two arguments, and the third will remain “invisible” for your function. Even more, it is possible to declare `main(int argc)`, and it will work.

### 5.2.3 Local variable storage

A function could allocate space in the stack for its local variables just by decreasing the [stack pointer](#) towards the stack bottom. Hence, it's very fast, no matter how many local variables are defined.

It is also not a requirement to store local variables in the stack. You could store local variables wherever you like, but traditionally this is how it's done.

### 5.2.4 x86: `alloca()` function

It is worth noting the `alloca()` function<sup>12</sup>.

This function works like `malloc()`, but allocates memory directly on the stack.

The allocated memory chunk does not need to be freed via a `free()` function call, since the function epilogue ([4 on page 23](#)) returns ESP back to its initial state and the allocated memory is just *dropped*.

It is worth noting how `alloca()` is implemented.

In simple terms, this function just shifts ESP downwards toward the stack bottom by the number of bytes you need and sets ESP as a pointer to the *allocated* block. Let's try:

```
#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);
};
```

`_snprintf()` function works just like `printf()`, but instead of dumping the result into [stdout](#) (e.g., to terminal or console), it writes it to the `buf` buffer. Function `puts()` copies the contents of `buf` to [stdout](#). Of course, these two function calls might be replaced by one `printf()` call, but we have to illustrate small buffer usage.

#### MSVC

Let's compile (MSVC 2010):

Listing 5.1: 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
```

<sup>12</sup>In MSVC, the function implementation can be found in `alloca16.asm` and `chkstk.asm` in `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\src\intel`

```

push    esi
call    __snprintf

push    esi
call    _puts
add     esp, 28          ; 0000001cH

...

```

The sole `alloca()` argument is passed via `EAX` (instead of pushing it into the stack)<sup>13</sup>. After the `alloca()` call, `ESP` points to the block of 600 bytes and we can use it as memory for the `buf` array.

### GCC + Intel syntax

GCC 4.4.1 does the same without calling external functions:

Listing 5.2: GCC 4.7.3

```

.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                ; align pointer by 16-bit border
    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 + AT&T syntax

Let's see the same code, but in AT&T syntax:

Listing 5.3: 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

```

<sup>13</sup>It is because `alloca()` is rather a compiler intrinsic ([90 on page 856](#)) than a normal function.

One of the reasons we need a separate function instead of just a couple of instructions in the code, is because the [MSVC<sup>14</sup>](#) `alloca()` implementation also has code which reads from the memory just allocated, in order to let the [OS](#) map physical memory to this [VM<sup>15</sup>](#) region.

```

movl    -4(%ebp), %ebx
leave
ret

```

The code is the same as in the previous listing.

By the way, `movl $3, 20(%esp)` corresponds to `mov DWORD PTR [esp+20], 3` in Intel-syntax. In the AT&T syntax, the register+offset format of addressing memory looks like `offset(%register)`.

### 5.2.5 (Windows) SEH

[SEH<sup>16</sup>](#) records are also stored on the stack (if they are present)..

Read more about it: ( [68.3 on page 677](#)).

### 5.2.6 Buffer overflow protection

More about it here ( [18.2 on page 261](#)).

### 5.2.7 Automatic deallocation of data in stack

Perhaps, the reason for storing local variables and SEH records in the stack is that they are freed automatically upon function exit, using just one instruction to correct the stack pointer (it is often `ADD`). Function arguments, as we could say, are also deallocated automatically at the end of function. In contrast, everything stored in the *heap* must be deallocated explicitly.

## 5.3 A typical stack layout

A typical stack layout in a 32-bit environment at the start of a function, before the first instruction execution looks like this:

...	...
ESP-0xC	local variable #2, marked in <a href="#">IDA</a> as <code>var_8</code>
ESP-8	local variable #1, marked in <a href="#">IDA</a> as <code>var_4</code>
ESP-4	saved value of EBP
ESP	return address
ESP+4	argument#1, marked in <a href="#">IDA</a> as <code>arg_0</code>
ESP+8	argument#2, marked in <a href="#">IDA</a> as <code>arg_4</code>
ESP+0xC	argument#3, marked in <a href="#">IDA</a> as <code>arg_8</code>
...	...

## 5.4 Noise in stack

Often in this book “noise” or “garbage” values in the stack or memory are mentioned. Where do they come from? These are what was left in there after other functions’ executions. Short example:

```

#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();

```

<sup>16</sup>Structured Exception Handling : [68.3 on page 677](#)

```
f2();
};
```

Compiling...

Listing 5.4: Non-optimizing 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
    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
```

The compiler will grumble a little bit...

```
c:\Polygon\c>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
```

But when we run the compiled program...

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

Oh, what a weird thing! We did not set any variables in `f2()`. These are “ghosts” values, which are still in the stack.

Let's load the example into OllyDbg:

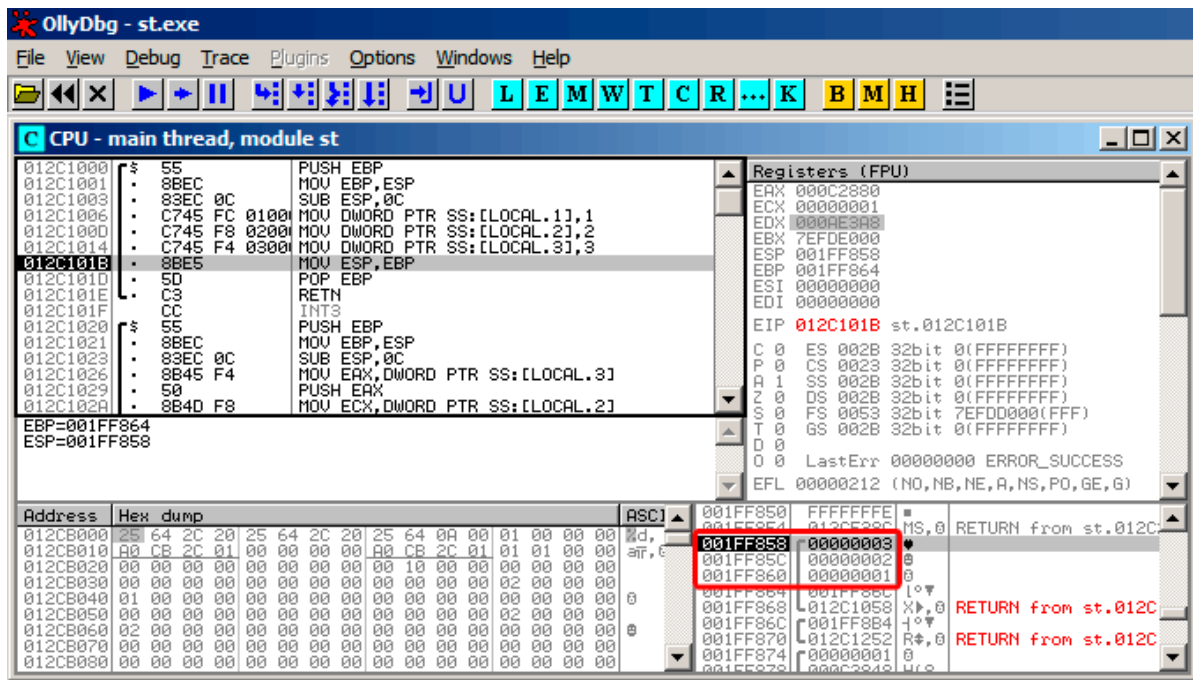


Figure 5.1: OllyDbg: f1()

When f1() assigns the variables *a*, *b* and *c*, their values are stored at the address 0x1FF860 and so on.

And when `f2()` executes:

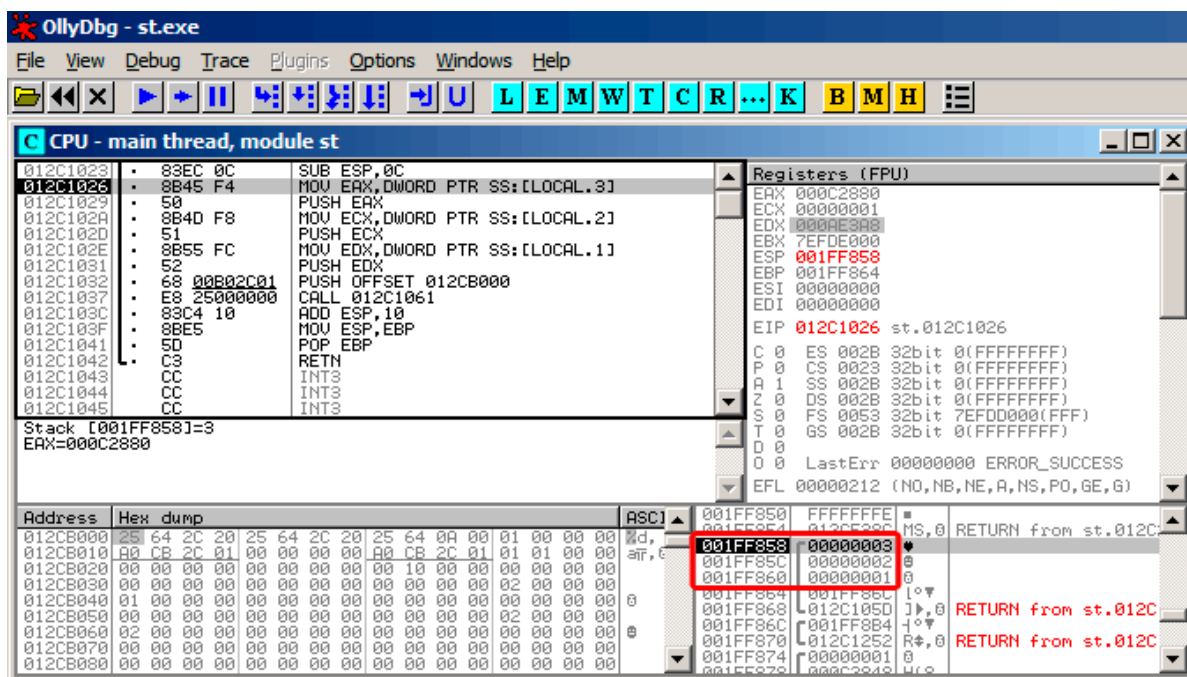


Figure 5.2: OllyDbg: `f2()`

... `a`, `b` and `c` of `f2()` are located at the same addresses! No one has overwritten the values yet, so at that point they are still untouched.

So, for this weird situation to occur, several functions have to be called one after another and `SP` has to be the same at each function entry (i.e., they have the same number of arguments). Then the local variables will be located at the same positions in the stack.

Summarizing, all values in the stack (and memory cells in general) have values left there from previous function executions. They are not random in the strict sense, but rather have unpredictable values.

Is there another option? It probably would be possible to clear portions of the stack before each function execution, but that's too much extra (and unnecessary) work.

### 5.4.1 MSVC 2013

The example was compiled by MSVC 2010. But the reader of this book made attempt to compile this example in MSVC 2013, ran it, and got all 3 numbers reversed:

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

Why?

I also compiled this example in MSVC 2013 and saw this:

Listing 5.5: 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
```

Unlike MSVC 2010, MSVC 2013 allocated a/b/c variables in function `f2()` in reverse order. And this is completely correct, because C/C++ standards has no rule, in which order local variables must be allocated in the local stack, if at all. The reason of difference is because MSVC 2010 has one way to do it, and MSVC 2013 has probably something changed inside of compiler guts, so it behaves slightly different.

## 5.5 Exercises

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



## Chapter 6

# printf() with several arguments

Now let's extend the *Hello, world!* ([3 on page 7](#)) example, replacing `printf()` in the `main()` function body with this:

```
#include <stdio.h>

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

## 6.1 x86

### 6.1.1 x86: 3 arguments

#### MSVC

When we compile it with MSVC 2010 Express we get:

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

Almost the same, but now we can see the `printf()` arguments are pushed onto the stack in reverse order. The first argument is pushed last.

By the way, variables of *int* type in 32-bit environment have 32-bit width, that is 4 bytes.

So, we have 4 arguments here.  $4 * 4 = 16$  —they occupy exactly 16 bytes in the stack: a 32-bit pointer to a string and 3 numbers of type *int*.

When the [stack pointer](#) (ESP register) has changed back by the `ADD ESP, X` instruction after a function call, often, the number of function arguments could be deduced by simply dividing X by 4.

Of course, this is specific to the *cdecl* calling convention, and only for 32-bit environment.

See also the calling conventions section ([64 on page 648](#)).

In certain cases where several functions return right after one another, the compiler could merge multiple “`ADD ESP, X`” instructions into one, after the last call:

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

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

Here is a real-world example:

Listing 6.1: x86

.text:100113E7	push	3	
.text:100113E9	call	sub_100018B0	; takes one argument (3)
.text:100113EE	call	sub_100019D0	; takes no arguments at all
.text:100113F3	call	sub_10006A90	; takes no arguments at all
.text:100113F8	push	1	
.text:100113FA	call	sub_100018B0	; takes one argument (1)
.text:100113FF	add	esp, 8	; drops two arguments from stack at once

## MSVC and OllyDbg

Now let's try to load this example in OllyDbg. It is one of the most popular user-land win32 debuggers. We can compile our example in MSVC 2012 with /MD option, which means to link with MSVCR\*.DLL, so we can see the imported functions clearly in the debugger.

Then load the executable in OllyDbg. The very first breakpoint is in `ntdll.dll`, press F9 (run). The second breakpoint is in `CRT`-code. Now we have to find the `main()` function.

Find this code by scrolling the code to the very top (MSVC allocates the `main()` function at the very beginning of the code section):

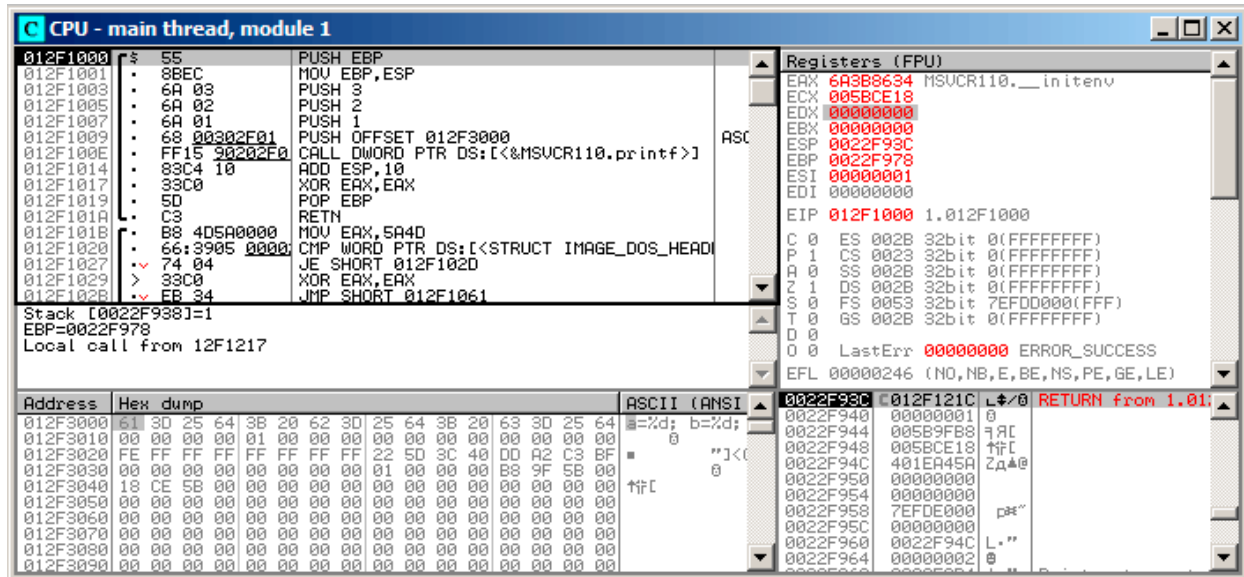


Figure 6.1: OllyDbg: the very start of the `main()` function

Click on the `PUSH EBP` instruction, press F2 (set breakpoint) and press F9 (run). We need to perform these actions in order to skip `CRT`-code, because we aren't really interested in it yet.

Press F8 (step over) 6 times, i.e. skip 6 instructions:

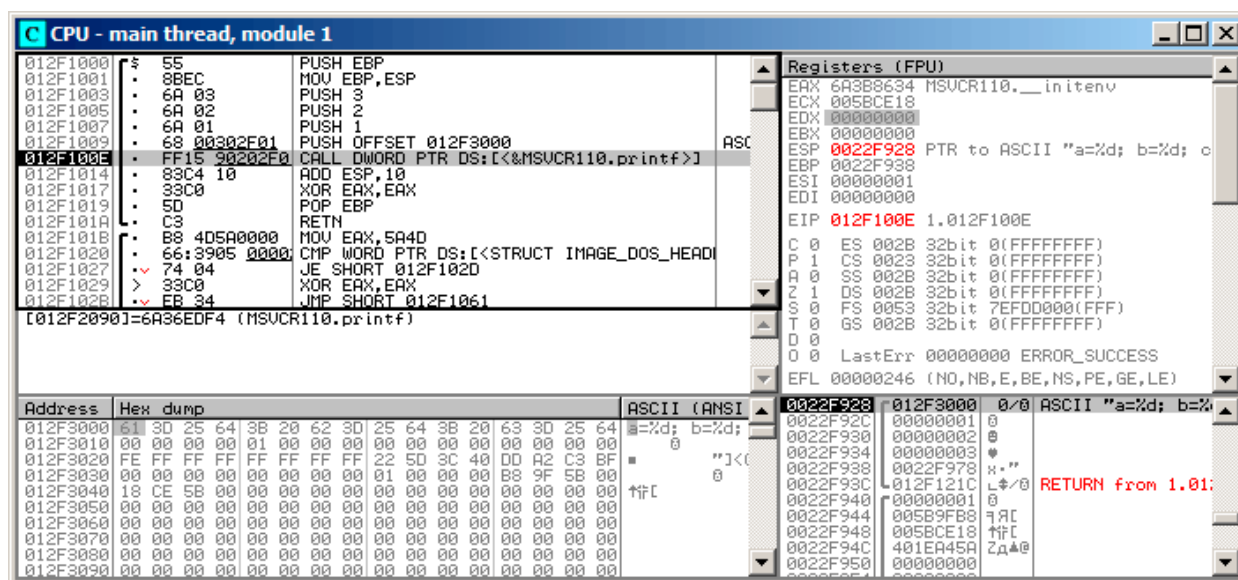


Figure 6.2: OllyDbg: before printf() execution

Now the PC points to the CALL printf instruction. OllyDbg, like other debuggers, highlights the value of the registers which were changed. So each time you press F8, EIP changes and its value is displayed in red. ESP changes as well, because the arguments values are pushed into the stack.

Where are the values in the stack? Take a look at the right bottom debugger window:

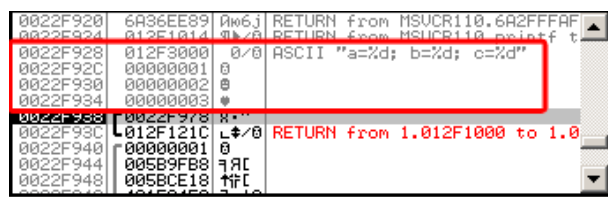


Figure 6.3: OllyDbg: stack after the argument values have been pushed (The red rectangular border was added by me in a graphics editor)

We can see 3 columns there: address in the stack, value in the stack and some additional OllyDbg comments. OllyDbg understands printf()-like strings, so it reports the string here and the 3 values attached to it.

It is possible to right-click on the format string, click on "Follow in dump", and the format string will appear in the debugger left-bottom window, which always displays some part of the memory. These memory values can be edited. It is possible to change the format string, in which case the result of our example would be different. It is not very useful in this particular case, but it could be good as an exercise so you start building a feel of how everything works here.

Press F8 (step over).

We see the following output in the console:

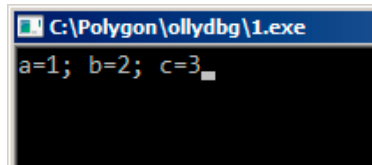


Figure 6.4: printf() function executed

Let's see how the registers and stack state have changed:

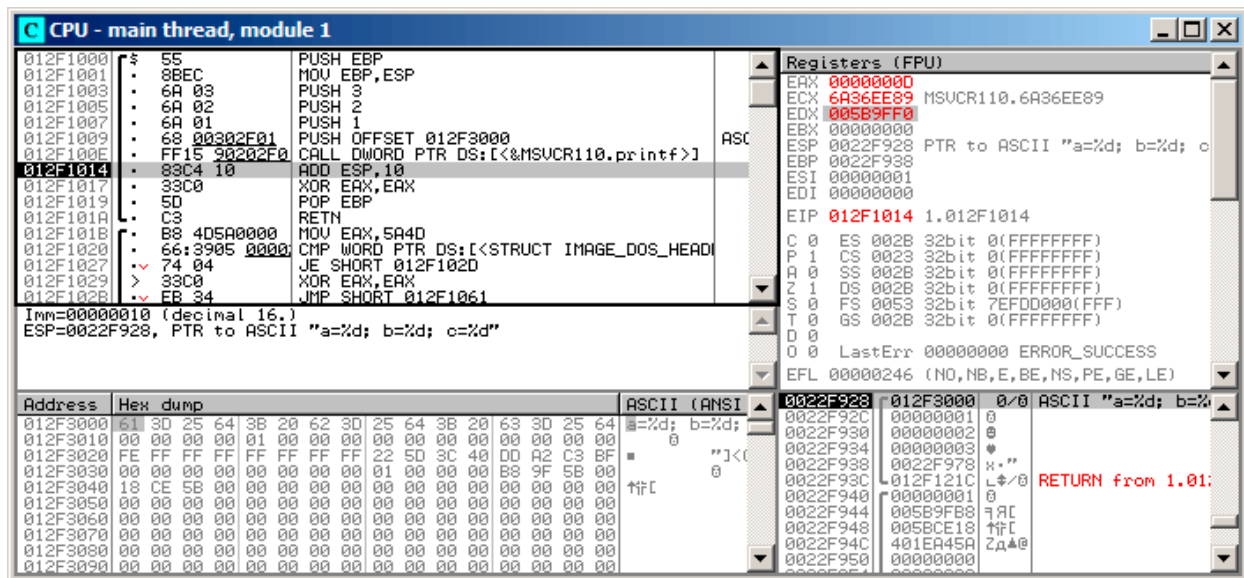


Figure 6.5: OllyDbg after printf() execution

Register EAX now contains 0xD (13). That is correct, since printf() returns the number of characters printed. The value of EIP has changed: indeed, now it contains the address of the instruction coming after CALL printf. ECX and EDX values have changed as well. Apparently, the printf() function's hidden machinery used them for its own needs.

A very important fact is that neither the ESP value, nor the stack state have been changed! We clearly see that the format string and corresponding 3 values are still there. This is indeed the *cdecl* calling convention behaviour: *callee* does not return ESP back to its previous value. The *caller* is responsible to do so.

Press F8 again to execute `ADD ESP, 10` instruction:

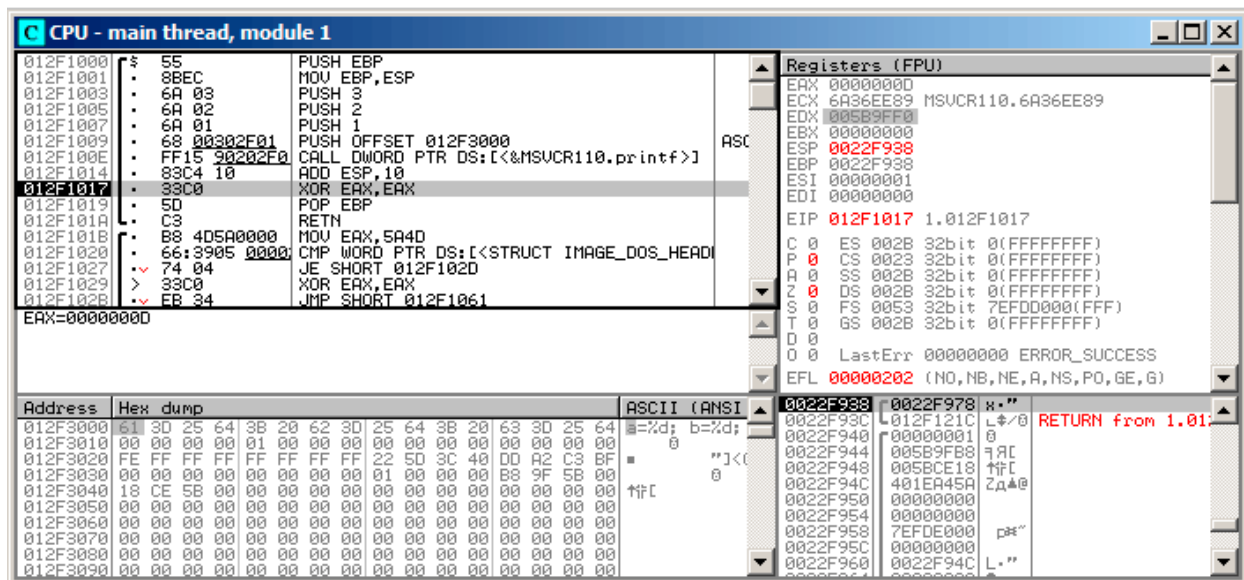


Figure 6.6: OllyDbg: after `ADD ESP, 10` instruction execution

ESP has changed, but the values are still in the stack! Yes, of course; no one needs to set these values to zeroes or something like that. Everything above the stack pointer (*SP*) is *noise* or *garbage* and has no meaning at all. It would be time consuming to clear the unused stack entries anyway, and no one really needs to.

## GCC

Now let's compile the same program in Linux using GCC 4.4.1 and take a look at what we have got in *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 aADBDCD ; "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
```

Its noticeable that the difference between the MSVC code and the GCC code is only in the way the arguments are stored on the stack. Here the GCC is working directly with the stack without the use of `PUSH/POP`.

## GCC and GDB

Let's try this example also in *GDB*<sup>1</sup> in Linux.

-g option instructs the compiler to include debug information in the executable file.

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

<sup>1</sup>GNU debugger

```
$ 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.
```

Listing 6.2: let's set breakpoint on printf()

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

Run. We don't have the printf() function source code here, so GDB can't show it, but may do so.

```
(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.
```

Print 10 stack elements. The most left column contains addresses on the stack.

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

The very first element is the RA (0x0804844a). We can verify this by disassembling the memory at this address:

```
(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
```

The two XCHG instructions are idle instructions, analogous to NOPs.

The second element (0x080484f0) is the format string address:

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

Next 3 elements (1, 2, 3) are the printf() arguments. The rest of the elements could be just "garbage" on the stack, but could also be values from other functions, their local variables, etc. We can ignore them for now.

Run "finish". The command instructs GDB to "execute all instructions until the end of the function". In this case: execute till the end of 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 shows what printf() returned in EAX (13). This is the number of characters printed out, just like in the OllyDbg example.

We also see "return 0;" and the information that this expression is in the 1.c file at the line 6. Indeed, the 1.c file is located in the current directory, and GDB finds the string there. How does GDB know which C-code line is being currently executed? This is due to the fact that the compiler, while generating debugging information, also saves a table of relations between source code line numbers and instruction addresses. GDB is a source-level debugger, after all.

Let's examine the registers. 13 in 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>
...
```

Let's disassemble the current instructions. The arrow points to the instruction to be executed next.

```
(gdb) disas
Dump of assembler code for function main:
0x0804841d <+0>:    push    %ebp
0x0804841e <+1>:    mov     %esp,%ebp
0x08048420 <+3>:    and     $0xffffffff0,%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
0x08048450 <+51>:   ret
End of assembler dump.
```

**GDB** uses AT&T syntax by default. It is possible to switch to Intel syntax:

```
(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,0xffffffff0
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
0x08048450 <+51>:   ret
End of assembler dump.
```

Execute next instruction. **GDB** shows ending bracket, meaning, it ends the block.

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

Let's examine the registers after the `MOV EAX, 0` instruction execution. Indeed EAX is zero at that point.

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000    -1208221696
esp            0xbffff120    0xbffff120
ebp            0xbffff138    0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844f      0x804844f <main+50>
...
```



### 6.1.2 x64: 8 arguments

To see how other arguments are passed via the stack, let's change our example again by increasing the number of arguments to 9 (`printf()` format string + 8 *int* variables):

```
#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

As it was mentioned earlier, the first 4 arguments has to be passed through the RCX, RDX, R8, R9 registers in Win64, while all the rest—via the stack. That is exactly what we see here. However, the MOV instruction, instead of PUSH, is used for preparing the stack, so the values are stored to the stack in a straightforward manner.

Listing 6.3: 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       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

          ; return 0
          xor     eax, eax

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

The observant reader may ask why are 8 bytes allocated for *int* values, when 4 is enough? Yes, one has to remember: 8 bytes are allocated for any data type shorter than 64 bits. This is established for the convenience's sake: it makes it easy to calculate the address of arbitrary argument. Besides, they are all located at aligned memory addresses. It is the same in the 32-bit environments: 4 bytes are reserved for all data types.

#### GCC

The picture is similar for x86-64 \*NIX OS-es, except that the first 6 arguments are passed through the RDI, RSI, RDX, RCX, R8, R9 registers. All the rest—via the stack. GCC generates the code storing the string pointer into EDI instead of RDI—we noted that previously: [3.2.2 on page 11](#).

We also noted earlier that the EAX register has been cleared before a `printf()` call: [3.2.2 on page 11](#).

Listing 6.4: Optimizing GCC 4.4.6 x64

```
.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:.LCO
xor     eax, eax ; number of vector registers passed
mov     DWORD PTR [rsp+16], 8
mov     DWORD PTR [rsp+8], 7
mov     DWORD PTR [rsp], 6
call    printf

; return 0

xor     eax, eax
add     rsp, 40
ret

```

## GCC + GDB

Let's try this example in [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.

```

Listing 6.5: let's set the breakpoint to `printf()`, and run

```

(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
29     printf.c: No such file or directory.

```

Registers RSI/RDX/RCX/R8/R9 have the expected values. RIP has the address of the very first instruction of the `printf()` function.

```

(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
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>
...

```

## Listing 6.6: let's inspect the format string

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

Let's dump the stack with the x/g command this time—g stands for *giant words*, i.e., 64-bit words.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576      0x0000000000000006
0x7fffffffdf48: 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000      0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78: 0x00007fffffe048      0x0000000100000000
```

The very first stack element, just like in the previous case, is the [RA](#). 3 values are also passed through the stack: 6, 7, 8. We also see that 8 is passed with the high 32-bits not cleared: 0x00007fff00000008. That's OK, because the values have *int* type, which is 32-bit. So, the high register or stack element part may contain "random garbage".

If you take a look at where the control will return after the `printf()` execution, [GDB](#) will show the entire `main()` function:

```
(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.
```

Let's finish executing `printf()`, execute the instruction zeroing EAX, and note that the EAX register has a value of exactly zero. RIP now points to the `LEAVE` instruction, i.e., the penultimate one in the `main()` function.

```
(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
↳ 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
rdx             0x7ffff7dd59f0 140737351866864
rsi             0x7fffffd9     2147483609
rdi             0x0          0
rbp             0x7fffffffdf60 0x7fffffffdf60
rsp             0x7fffffffdf40 0x7fffffffdf40
r8              0x7ffff7dd26a0 140737351853728
r9              0x7ffff7a60134 140737348239668
r10             0x7ffff7fd5b0 140737488344496
r11             0x7ffff7a95900 140737348458752
r12             0x400440 4195392
r13             0x7ffffffe040 140737488347200
r14             0x0          0
r15             0x0          0
```

```
rip          0x40057b 0x40057b <main+78>
...
```

## 6.2 ARM

### 6.2.1 ARM: 3 arguments

ARM's traditional scheme for passing arguments (calling convention) behaves as follows: the first 4 arguments are passed through the R0-R3 registers; the remaining arguments via the stack. This resembles the arguments passing scheme in fastcall ([64.3 on page 649](#)) or win64 ([64.5.1 on page 650](#)).

#### 32-bit ARM

##### Non-optimizing Keil 6/2013 (ARM mode)

Listing 6.7: Non-optimizing Keil 6/2013 (ARM mode)

```
.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}
```

So, the first 4 arguments are passed via the R0-R3 registers in this order: a pointer to the `printf()` format string in R0, then 1 in R1, 2 in R2 and 3 in R3.

The instruction at 0x18 writes 0 to R0—this is *return 0* C-statement.

There is nothing unusual so far.

Optimizing Keil 6/2013 generates the same code.

##### Optimizing Keil 6/2013 (Thumb mode)

Listing 6.8: Optimizing Keil 6/2013 (Thumb mode)

```
.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}
```

There is no significant difference from the non-optimized code for ARM mode.

##### Optimizing Keil 6/2013 (ARM mode) + let's remove return

Let's rework example slightly by removing *return 0*:

```
#include <stdio.h>

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

The result is somewhat unusual:

Listing 6.9: Optimizing Keil 6/2013 (ARM mode)

```
.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
```

This is the optimized (-O3) version for ARM mode and this time we see B as the last instruction instead of the familiar BL. Another difference between this optimized version and the previous one (compiled without optimization) is the lack of function prologue and epilogue (instructions preserving the R0 and LR registers values). The B instruction just jumps to another address, without any manipulation of the LR register, similar to JMP in x86. Why does it work? Because this code is, in fact, effectively equivalent to the previous. There are two main reasons: 1) neither the stack nor SP (the [stack pointer](#)) is modified; 2) the call to printf() is the last instruction, so there is nothing going on afterwards. On completion, the printf() function simply returns the control to the address stored in LR. Since the LR currently stores the address of the point from where our function was called then the control from printf() will be returned to that point. Therefore we do not need to save LR because we do not need to modify LR. And we do not need to modify LR because there are no other function calls except printf(). Furthermore, after this call we do not to do anything else! That is the reason such optimization is possible.

This optimization is often used in functions where the last statement is a call to another function.

A similar example is presented here: [13.1.1 on page 145](#).

## ARM64

### Non-optimizing GCC (Linaro) 4.9

Listing 6.10: Non-optimizing GCC (Linaro) 4.9

```
.LC1:
    .string "a=%d; b=%d; c=%d"
f2:
; save FP and LR in stack frame:
    stp     x29, x30, [sp, -16]!
; set stack frame (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
; restore FP and LR
    ldp     x29, x30, [sp], 16
    ret
```

The first instruction STP (Store Pair) saves FP (X29) and LR (X30) in the stack. The second ADD X29, SP, 0 instruction forms the stack frame. It is just writing the value of SP into X29.

Next, we see the familiar ADRP/ADD instruction pair, which forms a pointer to the string. *lo12* meaning low 12 bits, i.e., linker will write low 12 bits of LC1 address into the opcode of ADD instruction.

%d in printf() string format is a 32-bit *int*, so the 1, 2 and 3 are loaded into 32-bit register parts.

Optimizing GCC (Linaro) 4.9 generates the same code.

## 6.2.2 ARM: 8 arguments

Let's use again the example with 9 arguments from the previous section: [6.1.2 on page 43](#).

```
#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;
};

```

### Optimizing Keil 6/2013: ARM mode

```

.text:00000028      main
.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

```

This code can be divided into several parts:

- Function prologue:

The very first `STR LR, [SP,#var_4]!` instruction saves `LR` on the stack, because we are going to use this register for the `printf()` call. Exclamation mark at the end indicates *pre-index*. This implies that `SP` is to be decreased by 4 first, and then `LR` will be saved at the address stored in `SP`. This is similar to `PUSH` in x86. Read more about it at: [28.2 on page 425](#).

The second `SUB SP, SP, #0x14` instruction decreases `SP` (the [stack pointer](#)) in order to allocate 0x14 (20) bytes on the stack. Indeed, we need to pass 5 32-bit values via the stack to the `printf()` function, and each one occupies 4 bytes, which is exactly  $5 * 4 = 20$ . The other 4 32-bit values are to be passed through registers.

- Passing 5, 6, 7 and 8 via the stack: they are stored in the R0, R1, R2 and R3 registers respectively. Then, the `ADD R12, SP, #0x18+var_14` instruction writes the stack address where these 4 variables are to be stored, into the R12 register. `var_14` is an assembly macro, equal to -0x14, created by [IDA](#) to conveniently display the code accessing the stack. The `var_?` macros generated by [IDA](#) reflect local variables in the stack. So, `SP+4` is to be stored into the R12 register. The next `STMIA R12, R0-R3` instruction writes registers R0-R3 contents to the memory pointed by R12. `STMIA` abbreviates *Store Multiple Increment After*. “Increment After” implies that R12 is to be increased by 4 after each register value is written.
- Passing 4 via the stack: 4 is stored in R0 and then this value, with the help of the `STR R0, [SP,#0x18+var_18]` instruction is saved on the stack. `var_18` is -0x18, so the offset is to be 0, thus the value from the R0 register (4) is to be written to the address written in `SP`.
- Passing 1, 2 and 3 via registers:

The values of the first 3 numbers (a, b, c) (1, 2, 3 respectively) are passed through the R1, R2 and R3 registers right before the `printf()` call, and the other 5 values are passed via the stack:

- `printf()` call.
- Function epilogue:

The `ADD SP, SP, #0x14` instruction restores the `SP` pointer back to its former value, thus cleaning the stack. Of course, what was stored on the stack will stay there, but it will all be rewritten during the execution of subsequent functions.

The `LDR PC, [SP+4+var_4],#4` instruction loads the saved `LR` value from the stack into the `PC` register, thus causing the function to exit. There is no exclamation mark—indeed, `PC` is loaded first from the address stored in `SP`

$(4 + \text{var\_4} = 4 + (-4) = 0)$ , so this instruction is analogous to `LDR PC, [SP], #4`, and then `SP` is increased by 4. This is referred as *post-index*<sup>2</sup>. Why does `IDA` display the instruction like that? Because it wants to illustrate the stack layout and the fact that `var_4` is allocated for saving the `LR` value in the local stack. This instruction is somewhat similar to `POP PC` in `x86`<sup>3</sup>.

### Optimizing Keil 6/2013: Thumb mode

```
.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}
```

The output is almost like in the previous example. However, this is Thumb code and the values are packed into stack differently: 8 goes first, then 5, 6, 7, and 4 goes third.

### Optimizing Xcode 4.6.3 (LLVM): ARM mode

```
__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
```

<sup>2</sup>Read more about it: [28.2 on page 425](#).

<sup>3</sup>It is impossible to set `IP/EIP/RIP` value using `POP` in `x86`, but anyway, you got the analogy right.

```
__text:00002958 80 80 BD E8   LDMFD   SP!, {R7,PC}
```

Almost the same as what we have already seen, with the exception of STMFA (Store Multiple Full Ascending) instruction, which is a synonym of STMIB (Store Multiple Increment Before) instruction. This instruction increases the value in the **SP** register and only then writes the next register value into the memory, rather than performing those two actions in the opposite order.

Another thing that catches the eye is that the instructions are arranged seemingly random. For example, the value in the **R0** register is manipulated in three places, at addresses 0x2918, 0x2920 and 0x2928, when it would be possible to do it in one point. However, the optimizing compiler may have its own reasons on how to order the instructions so to achieve higher efficiency during the execution. Usually, the processor attempts to simultaneously execute instructions located side-by-side. For example, instructions like **MOVT R0, #0** and **ADD R0, PC, R0** cannot be executed simultaneously since they both modify the **R0** register. On the other hand, **MOVT R0, #0** and **MOV R2, #4** instructions can be executed simultaneously since the effects of their execution are not conflicting with each other. Presumably, the compiler tries to generate code in such a manner (wherever it is possible).

### Optimizing Xcode 4.6.3 (LLVM): Thumb-2 mode

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

The output is almost the same as in the previous example, with the exception that Thumb-instructions are used instead.

## ARM64

### Non-optimizing GCC (Linaro) 4.9

Listing 6.11: Non-optimizing GCC (Linaro) 4.9

```
.LC2:
.string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; grab more space in stack:
sub    sp, sp, #32
; save FP and LR in stack frame:
stp    x29, x30, [sp,16]
; set stack frame (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 ; 9th argument
```



```

    str    w1, [sp]          ; store 9th argument in the stack
    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
; restore FP and LR
    ldp     x29, x30, [sp,16]
    add     sp, sp, 32
    ret

```

The first 8 arguments are passed in X- or W-registers: [ARM13c]. A string pointer requires a 64-bit register, so it's passed in X0. All other values have a *int* 32-bit type, so they are stored in the 32-bit part of the registers (W-). The 9th argument (8) is passed via the stack. Indeed: it's not possible to pass large number of arguments through registers, because the number of registers is limited.

Optimizing GCC (Linaro) 4.9 generates the same code.

## 6.3 MIPS

### 6.3.1 3 arguments

#### Optimizing GCC 4.4.5

The main difference with the "Hello, world!" example is that in this case `printf()` is called instead of `puts()` and 3 more arguments are passed through the registers \$5...\$7 (or \$A0...\$A2).

That is why these registers are prefixed with A-, which implies they are used for function arguments passing.

Listing 6.12: Optimizing GCC 4.4.5 (assembly output)

```

$LC0:
    .ascii "a=%d; b=%d; c=%d\000"
main:
; function prologue:
    lui    $28,%hi(__gnu_local_gp)
    addiu   $sp,$sp,-32
    addiu   $28,$28,%lo(__gnu_local_gp)
    sw      $31,28($sp)
; load address of printf():
    lw      $25,%call16(printf)($28)
; load address of the text string and set 1st argument of printf():
    lui     $4,%hi($LC0)
    addiu   $4,$4,%lo($LC0)
; set 2nd argument of printf():
    li      $5,1                # 0x1
; set 3rd argument of printf():
    li      $6,2                # 0x2
; call printf():
    jalr    $25
; set 4th argument of printf() (branch delay slot):
    li      $7,3                # 0x3

; function epilogue:
    lw      $31,28($sp)
; set return value to 0:
    move     $2,$0
; return
    j       $31
    addiu   $sp,$sp,32 ; branch delay slot

```

Listing 6.13: Optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:
```

```

.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4          = -4
.text:00000000
; function prologue:
.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)
; load address of printf():
.text:00000014                lw     $t9, (printf & 0xFFFF)($gp)
; load address of the text string and set 1st argument of printf():
.text:00000018                la     $a0, $LC0          # "a=%d; b=%d; c=%d"
; set 2nd argument of printf():
.text:00000020                li     $a1, 1
; set 3rd argument of printf():
.text:00000024                li     $a2, 2
; call printf():
.text:00000028                jalr   $t9
; set 4th argument of printf() (branch delay slot):
.text:0000002C                li     $a3, 3
; function epilogue:
.text:00000030                lw     $ra, 0x20+var_4($sp)
; set return value to 0:
.text:00000034                move   $v0, $zero
; return
.text:00000038                jr     $ra
.text:0000003C                addiu  $sp, 0x20 ; branch delay slot

```

IDA has coalesced pair of LUI and ADDIU instructions into one LA pseudoinstruction. That's why there are no instruction at address 0x1C: because LA occupies 8 bytes.

### Non-optimizing GCC 4.4.5

Non-optimizing GCC is more verbose:

Listing 6.14: Non-optimizing GCC 4.4.5 (assembly output)

```

$LC0:
    .ascii "a=%d; b=%d; c=%d\000"
main:
; function prologue:
    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)
; load address of the text string:
    lui    $2,%hi($LC0)
    addiu  $2,$2,%lo($LC0)
; set 1st argument of printf():
    move   $4,$2
; set 2nd argument of printf():
    li     $5,1                # 0x1
; set 3rd argument of printf():
    li     $6,2                # 0x2
; set 4th argument of printf():
    li     $7,3                # 0x3
; get address of printf():
    lw     $2,%call16(printf)($28)
    nop
; call printf():
    move   $25,$2
    jalr   $25
    nop
; function epilogue:

```

```

        lw      $28,16($fp)
; set return value to 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; return
        j       $31
        nop

```

Listing 6.15: Non-optimizing 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
; function prologue:
.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)
; load address of the text string:
.text:0000001C                la      $v0, aADBDCD      # "a=%d; b=%d; c=%d"
; set 1st argument of printf():
.text:00000024                move    $a0, $v0
; set 2nd argument of printf():
.text:00000028                li      $a1, 1
; set 3rd argument of printf():
.text:0000002C                li      $a2, 2
; set 4th argument of printf():
.text:00000030                li      $a3, 3
; get address of printf():
.text:00000034                lw      $v0, (printf & 0xFFFF)($gp)
.text:00000038                or      $at, $zero
; call printf():
.text:0000003C                move    $t9, $v0
.text:00000040                jalr    $t9
.text:00000044                or      $at, $zero ; NOP
; function epilogue:
.text:00000048                lw      $gp, 0x20+var_10($fp)
; set return value to 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
; return
.text:00000060                jr      $ra
.text:00000064                or      $at, $zero ; NOP

```

### 6.3.2 8 arguments

Let's use again the example with 9 arguments from the previous section: [6.1.2 on page 43](#).

```

#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;
};

```

**Optimizing GCC 4.4.5**

Only the first 4 arguments are passed in the \$A0 ...\$A3 registers, the rest are passed via the stack. This is the O32 calling convention (which is the most common one in the MIPS world). Other calling conventions (like N32) may use the registers for different purposes.

SW abbreviates “Store Word” (from register to memory). MIPS lacks instructions for storing a value into memory, so an instruction pair has to be used instead (LI/SW).

Listing 6.16: Optimizing GCC 4.4.5 (assembly output)

```

$LC0:
    .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; function prologue:
    lui     $28,%hi(__gnu_local_gp)
    addiu   $sp,$sp,-56
    addiu   $28,$28,%lo(__gnu_local_gp)
    sw      $31,52($sp)
; pass 5th argument in stack:
    li      $2,4                # 0x4
    sw      $2,16($sp)
; pass 6th argument in stack:
    li      $2,5                # 0x5
    sw      $2,20($sp)
; pass 7th argument in stack:
    li      $2,6                # 0x6
    sw      $2,24($sp)
; pass 8th argument in stack:
    li      $2,7                # 0x7
    lw      $25,%call16(sprintf)($28)
    sw      $2,28($sp)
; pass 1st argument in $a0:
    lui     $4,%hi($LC0)
; pass 9th argument in stack:
    li      $2,8                # 0x8
    sw      $2,32($sp)
    addiu   $4,$4,%lo($LC0)
; pass 2nd argument in $a1:
    li      $5,1                # 0x1
; pass 3rd argument in $a2:
    li      $6,2                # 0x2
; call printf():
    jalr    $25
; pass 4th argument in $a3 (branch delay slot):
    li      $7,3                # 0x3

; function epilogue:
    lw      $31,52($sp)
; set return value to 0:
    move    $2,$0
; return
    j       $31
    addiu   $sp,$sp,56 ; branch delay slot

```

Listing 6.17: Optimizing 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
; function prologue:
.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)
; pass 5th argument in stack:
.text:00000014      li      $v0, 4
.text:00000018      sw      $v0, 0x38+var_28($sp)
; pass 6th argument in stack:
.text:0000001C      li      $v0, 5
.text:00000020      sw      $v0, 0x38+var_24($sp)
; pass 7th argument in stack:
.text:00000024      li      $v0, 6
.text:00000028      sw      $v0, 0x38+var_20($sp)
; pass 8th argument in stack:
.text:0000002C      li      $v0, 7
.text:00000030      lw      $t9, (printf & 0xFFFF)($gp)
.text:00000034      sw      $v0, 0x38+var_1C($sp)
; prepare 1st argument in $a0:
.text:00000038      lui      $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d\
    ↪ ; g=%"...
; pass 9th argument in stack:
.text:0000003C      li      $v0, 8
.text:00000040      sw      $v0, 0x38+var_18($sp)
; pass 1st argument in $a1:
.text:00000044      la      $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d; e=%d; f\
    ↪ =%d; g=%"...
; pass 2nd argument in $a1:
.text:00000048      li      $a1, 1
; pass 3rd argument in $a2:
.text:0000004C      li      $a2, 2
; call printf():
.text:00000050      jalr     $t9
; pass 4th argument in $a3 (branch delay slot):
.text:00000054      li      $a3, 3
; function epilogue:
.text:00000058      lw      $ra, 0x38+var_4($sp)
; set return value to 0:
.text:0000005C      move     $v0, $zero
; return
.text:00000060      jr      $ra
.text:00000064      addiu    $sp, 0x38 ; branch delay slot

```

### Non-optimizing GCC 4.4.5

Non-optimizing GCC is more verbose:

Listing 6.18: Non-optimizing GCC 4.4.5 (assembly output)

```

$LC0:
    .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; function prologue:
    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)
; pass 5th argument in stack:
    li       $3,4                # 0x4
    sw       $3,16($sp)
; pass 6th argument in stack:
    li       $3,5                # 0x5
    sw       $3,20($sp)
; pass 7th argument in stack:
    li       $3,6                # 0x6
    sw       $3,24($sp)
; pass 8th argument in stack:
    li       $3,7                # 0x7

```

```

        sw      $3,28($sp)
; pass 9th argument in stack:
        li      $3,8                # 0x8
        sw      $3,32($sp)
; pass 1st argument in $a0:
        move    $4,$2
; pass 2nd argument in $a1:
        li      $5,1                # 0x1
; pass 3rd argument in $a2:
        li      $6,2                # 0x2
; pass 4th argument in $a3:
        li      $7,3                # 0x3
; call printf():
        lw      $2,%call16(printf)($28)
        nop
        move    $25,$2
        jalr    $25
        nop
; function epilogue:
        lw      $28,40($fp)
; set return value to 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,52($sp)
        lw      $fp,48($sp)
        addiu   $sp,$sp,56
; return
        j       $31
        nop

```

Listing 6.19: Non-optimizing 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_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; function prologue:
.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=%d; h=%d; i=%d; j=%d; k=%d; l=%d; m=%d; n=%d; o=%d; p=%d; q=%d; r=%d; s=%d; t=%d; u=%d; v=%d; w=%d; x=%d; y=%d; z=%d; \n"
; pass 5th argument in stack:
.text:00000024                li      $v1, 4
.text:00000028                sw      $v1, 0x38+var_28($sp)
; pass 6th argument in stack:
.text:0000002C                li      $v1, 5
.text:00000030                sw      $v1, 0x38+var_24($sp)
; pass 7th argument in stack:
.text:00000034                li      $v1, 6
.text:00000038                sw      $v1, 0x38+var_20($sp)
; pass 8th argument in stack:
.text:0000003C                li      $v1, 7
.text:00000040                sw      $v1, 0x38+var_1C($sp)
; pass 9th argument in stack:
.text:00000044                li      $v1, 8
.text:00000048                sw      $v1, 0x38+var_18($sp)
; pass 1st argument in $a0:
.text:0000004C                move    $a0, $v0
; pass 2nd argument in $a1:

```

```

.text:00000050      li      $a1, 1
; pass 3rd argument in $a2:
.text:00000054      li      $a2, 2
; pass 4th argument in $a3:
.text:00000058      li      $a3, 3
; call 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
; function epilogue:
.text:00000070      lw      $gp, 0x38+var_10($fp)
; set return value to 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
; return
.text:00000088      jr      $ra
.text:0000008C      or      $at, $zero ; NOP

```

## 6.4 Conclusion

Here is a rough skeleton of the function call:

Listing 6.20: x86

```

...
PUSH 3rd argument
PUSH 2nd argument
PUSH 1st argument
CALL function
; modify stack pointer (if needed)

```

Listing 6.21: x64 (MSVC)

```

MOV RCX, 1st argument
MOV RDX, 2nd argument
MOV R8, 3rd argument
MOV R9, 4th argument
...
PUSH 5th, 6th argument, etc (if needed)
CALL function
; modify stack pointer (if needed)

```

Listing 6.22: x64 (GCC)

```

MOV RDI, 1st argument
MOV RSI, 2nd argument
MOV RDX, 3rd argument
MOV RCX, 4th argument
MOV R8, 5th argument
MOV R9, 6th argument
...
PUSH 7th, 8th argument, etc (if needed)
CALL function
; modify stack pointer (if needed)

```

Listing 6.23: ARM

```

MOV R0, 1st argument
MOV R1, 2nd argument
MOV R2, 3rd argument
MOV R3, 4th argument
; pass 5th, 6th argument, etc, in stack (if needed)
BL function
; modify stack pointer (if needed)

```

Listing 6.24: ARM64

```
MOV X0, 1st argument
MOV X1, 2nd argument
MOV X2, 3rd argument
MOV X3, 4th argument
MOV X4, 5th argument
MOV X5, 6th argument
MOV X6, 7th argument
MOV X7, 8th argument
; pass 9th, 10th argument, etc, in stack (if needed)
BL CALL function
; modify stack pointer (if needed)
```

Listing 6.25: MIPS (O32 calling convention)

```
LI $4, 1st argument ; AKA $A0
LI $5, 2nd argument ; AKA $A1
LI $6, 3rd argument ; AKA $A2
LI $7, 4th argument ; AKA $A3
; pass 5th, 6th argument, etc, in stack (if needed)
LW temp_reg, address of function
JALR temp_reg
```

## 6.5 By the way

By the way, this difference between the arguments passing in x86, x64, fastcall, ARM and MIPS is a good illustration of the fact that the CPU is oblivious to how the arguments are passed to functions. It is also possible to create a hypothetical compiler able to pass arguments via a special structure without using stack at all.

MIPS \$A0 ...\$A3 registers are labelled this way only for convenience (that is in the O32 calling convention). Programmers may use any other register (well, maybe except \$ZERO) to pass data or use any other calling convention.

The [CPU](#) is not aware of calling conventions whatsoever.

We may also recall how newcomers assembly language programmers passing arguments into other functions: usually via registers, without any explicit order, or even via global variables. Of course, it works fine.



# Chapter 7

## scanf()

Now let's use `scanf()`.

### 7.1 Simple example

```
#include <stdio.h>

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

    scanf ("%d", &x);

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

    return 0;
};
```

It's not clever to use `scanf()` for user interactions nowadays. But we can, however, illustrate passing a pointer to a variable of type `int`.

#### 7.1.1 About pointers

Pointers are one of the fundamental concepts in computer science. Often, passing a large array, structure or object as an argument to another function is too expensive, while passing their address is much cheaper. In addition if the [callee](#) function needs to modify something in the large array or structure received as a parameter and return back the entire structure then the situation is close to absurd. So the simplest thing to do is to pass the address of the array or structure to the [callee](#) function, and let it change what needs to be changed.

A pointer in C/C++ is simply an address of some memory location.

In x86, the address is represented as a 32-bit number (i.e., it occupies 4 bytes), while in x86-64 it is a 64-bit number (occupying 8 bytes). By the way, that is the reason behind some people's indignation related to switching to x86-64—all pointers in the x64-architecture require twice as much space, including cache memory, which is "expensive" place.

It is possible to work with untyped pointers only, given some effort; e.g. the standard C function `memcpy()`, that copies a block from one memory location to another, takes 2 pointers of type `void*` as arguments, since it is impossible to predict the type of the data you would like to copy. Data types are not important, only the block size matters.

Pointers are also widely used when a function needs to return more than one value (we are going to get back to this later ([10 on page 101](#))). `scanf()` is such a case. Besides the fact that the function needs to indicate how many values were successfully read, it also needs to return all these values.

In C/C++ the pointer type is only needed for compile-time type checking. Internally, in the compiled code there is no information about pointer types at all.

## 7.1.2 x86

### MSVC

Here is what we get after compiling with 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

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

```

x is a local variable.

According to the C/C++ standard it must be visible only in this function and not from any other external scope. Traditionally, local variables are stored on the stack. There are probably other ways to allocate them, but in x86 that is the way it is.

The goal of the instruction following the function prologue, `PUSH ECX`, is not to save the ECX state (notice the absence of corresponding `POP ECX` at the function's end).

In fact it allocates 4 bytes on the stack for storing the x variable.

x is to be accessed with the assistance of the `_x$` macro (it equals to -4) and the EBP register pointing to the current frame.

Over the span of the function's execution, EBP is pointing to the current [stack frame](#) making it possible to access local variables and function arguments via `EBP+offset`.

It is also possible to use ESP for the same purpose, although that is not very convenient since it changes frequently. The value of the EBP could be perceived as a *frozen state* of the value in ESP at the start of the function's execution.

Here is a typical [stack frame](#) layout in 32-bit environment:

...	...
EBP-8	local variable #2, marked in <a href="#">IDA</a> as var_8
EBP-4	local variable #1, marked in <a href="#">IDA</a> as var_4
EBP	saved value of EBP
EBP+4	return address
EBP+8	argument#1, marked in <a href="#">IDA</a> as arg_0
EBP+0xC	argument#2, marked in <a href="#">IDA</a> as arg_4
EBP+0x10	argument#3, marked in <a href="#">IDA</a> as arg_8
...	...

The `scanf()` function in our example has two arguments.

The first one is a pointer to the string containing `%d` and the second is the address of the `x` variable.

First, the `x` variable's address is loaded into the `EAX` register by the `lea eax, DWORD PTR _x$[ebp]` instruction

`LEA` stands for *load effective address*, and is often used for forming an address ([A.6.2 on page 886](#)).

We could say that in this case `LEA` simply stores the sum of the `EBP` register value and the `_x$` macro in the `EAX` register.

This is the same as `lea eax, [ebp-4]`.

So, 4 is being subtracted from the `EBP` register value and the result is loaded in the `EAX` register. Next the `EAX` register value is pushed into the stack and `scanf()` is being called.

`printf()` is being called after that with its first argument – a pointer to the string: `You entered %d...\n`.

The second argument is prepared with: `mov ecx, [ebp-4]`. The instruction stores the `x` variable value and not its address, in the `ECX` register.

Next the value in the `ECX` is stored on the stack and the last `printf()` is being called.

### 7.1.3 MSVC + OllyDbg

Let's try this example in OllyDbg. Let's load it and keep pressing F8 (step over) until we reach our executable file instead of ntdll.dll. Scroll up until `main()` appears. Click on the first instruction (`PUSH EBP`), press F2 (set a breakpoint), then F9 (Run). The breakpoint will be triggered when `main()` begins.

Let's trace to the point where the address of the variable `x` is calculated:

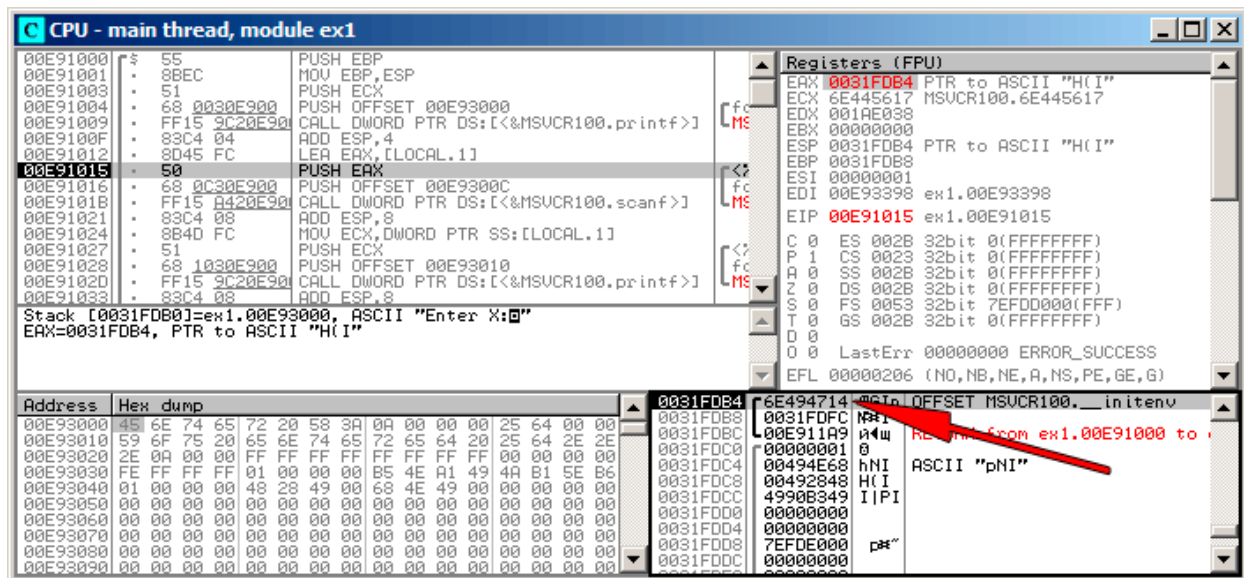


Figure 7.1: OllyDbg: The address of the local variable is calculated

Right-click the EAX in the registers window and then select "Follow in stack". This address will appear in the stack window. The red arrow has been added, pointing to the variable in the local stack. At that moment this location contains some garbage (0x6E494714). Now with the help of `PUSH` instruction the address of this stack element is going to be stored to the same stack on the next position. Let's trace with F8 until the `scanf()` execution completes. During the `scanf()` execution, we input, for example, 123, in the console window:

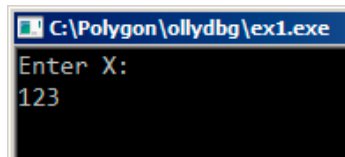


Figure 7.2: User input in the console window

scanf() completed its execution already:

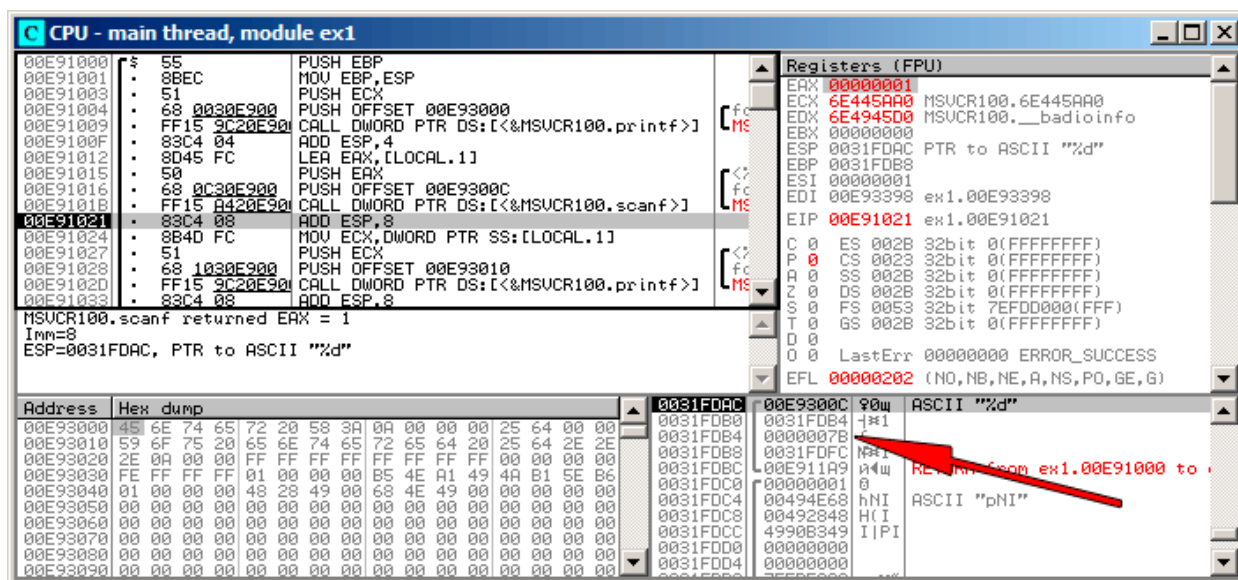


Figure 7.3: OllyDbg: scanf() executed

scanf() returns 1 in EAX, which implies that it has read successfully one value. If we look again at the stack element corresponding to the local variable it now contains 0x7B (123).

Later this value is copied from the stack to the ECX register and passed to `printf()`:

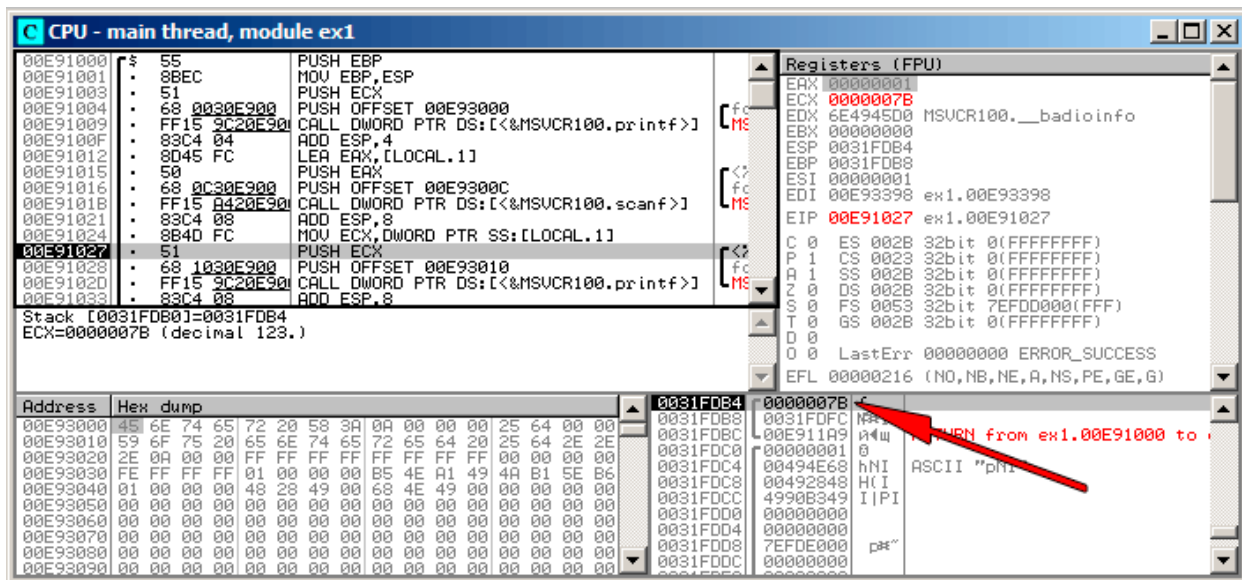


Figure 7.4: OllyDbg: preparing the value for passing to `printf()`

## GCC

Let's try to compile this code in GCC 4.4.1 under Linux:

```
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_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 replaced the `printf()` call with call to `puts()`. The reason for this was explained in ( [3.4.3 on page 15](#)).

As in the MSVC example—the arguments are placed on the stack using the `MOV` instruction.

## By the way

By the way, this simple example is a demonstration of the fact that compiler translates list of expressions in C/C++-block into sequential list of instructions. There are nothing between expressions in C/C++, and so in resulting machine code, there are nothing between, control flow slips from one expression to the next one.

**7.1.4 x64**

The picture here is similar with the difference that the registers, rather than the stack, are used for arguments passing.

**MSVC**

Listing 7.1: 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

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

```

**GCC**

Listing 7.2: Optimizing 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

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

```

## 7.1.5 ARM

### Optimizing Keil 6/2013 (Thumb mode)

```
.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}
```

In order for `scanf()` to be able to read item it needs a parameter—pointer to an *int*. *int* is 32-bit, so we need 4 bytes to store it somewhere in memory, and it fits exactly in a 32-bit register. A place for the local variable *x* is allocated in the stack and *IDA* has named it *var\_8*. It is not necessary, however, to allocate a such since *SP* (stack pointer) is already pointing to that space and it can be used directly. So, *SP*'s value is copied to the *R1* register and, together with the format-string, passed to `scanf()`. Later, with the help of the *LDR* instruction, this value is moved from the stack to the *R1* register in order to be passed to `printf()`.

## ARM64

Listing 7.3: Non-optimizing GCC 4.9.1 ARM64

```
1 .LC0:
2     .string "Enter X:"
3
4 .LC1:
5     .string "%d"
6
7 .LC2:
8     .string "You entered %d...\n"
9
10 scanf_main:
11 ; subtract 32 from SP, then save FP and LR in stack frame:
12     stp     x29, x30, [sp, -32]!
13 ; set stack frame (FP=SP)
14     add     x29, sp, 0
15 ; load pointer to the "Enter X:" string:
16     adrp    x0, .LC0
17     add     x0, x0, :lo12:LC0
18 ; X0=pointer to the "Enter X:" string
19 ; print it:
20     bl      puts
21 ; load pointer to the "%d" string:
22     adrp    x0, .LC1
23     add     x0, x0, :lo12:LC1
24 ; find a space in stack frame for "x" variable (X1=FP+28):
25     add     x1, x29, 28
26 ; X1=address of "x" variable
27 ; pass the address to scanf() and call it:
28     bl      __isoc99_scanf
29 ; load 32-bit value from the variable in stack frame:
30     ldr     w1, [x29,28]
31 ; W1=x
32 ; load pointer to the "You entered %d...\n" string
33 ; printf() will take text string from X0 and "x" variable from X1 (or W1)
34     adrp    x0, .LC2
35     add     x0, x0, :lo12:LC2
36     bl      printf
37 ; return 0
38     mov     w0, 0
39 ; restore FP and LR, then add 32 to SP:
40     ldp     x29, x30, [sp], 32
```



There is 32 bytes are allocated for stack frame, which is bigger than it needed. Perhaps, some memory aligning issue? The most interesting part is finding space for the  $x$  variable in the stack frame (line 22). Why 28? Somehow, compiler decided to place this variable at the end of stack frame instead of beginning. The address is passed to `scanf()`, which just stores the user input value in the memory at that address. This is 32-bit value of type *int*. The value is fetched at line 27 and then passed to `printf()`.

## 7.1.6 MIPS

A place in the local stack is allocated for the  $x$  variable, and it is to be referred as  $\$sp+24$ . Its address is passed to `scanf()`, and the user input values is loaded using the LW ("Load Word") instruction and then passed to `printf()`.

Listing 7.4: Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
    .ascii "Enter X:\000"
$LC1:
    .ascii "%d\000"
$LC2:
    .ascii "You entered %d...\012\000"
main:
; function prologue:
    lui    $28,%hi(__gnu_local_gp)
    addiu  $sp,$sp,-40
    addiu  $28,$28,%lo(__gnu_local_gp)
    sw     $31,36($sp)
; call puts():
    lw     $25,%call16(puts)($28)
    lui    $4,%hi($LC0)
    jalr   $25
    addiu  $4,$4,%lo($LC0) ; branch delay slot
; call scanf():
    lw     $28,16($sp)
    lui    $4,%hi($LC1)
    lw     $25,%call16(__isoc99_scanf)($28)
; set 2nd argument of scanf(), $a1=$sp+24:
    addiu  $5,$sp,24
    jalr   $25
    addiu  $4,$4,%lo($LC1) ; branch delay slot

; call printf():
    lw     $28,16($sp)
; set 2nd argument of printf(),
; load word at address $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

; function epilogue:
    lw     $31,36($sp)
; set return value to 0:
    move   $2,$0
; return:
    j      $31
    addiu  $sp,$sp,40 ; branch delay slot
```

IDA displays the stack layout as follows:

Listing 7.5: Optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
```

```

; function prologue:
.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)
; call 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
; call scanf():
.text:00000024      lw       $gp, 0x28+var_18($sp)
.text:00000028      lui      $a0, ($LC1 >> 16) # "%d"
.text:0000002C      lw       $t9, (__isoc99_scanf & 0xFFFF)($gp)
; set 2nd argument of 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"
; call printf():
.text:0000003C      lw       $gp, 0x28+var_18($sp)
; set 2nd argument of printf(),
; load word at address $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
; function epilogue:
.text:00000054      lw       $ra, 0x28+var_4($sp)
; set return value to 0:
.text:00000058      move     $v0, $zero
; return:
.text:0000005C      jr       $ra
.text:00000060      addiu    $sp, 0x28 ; branch delay slot

```

## 7.2 Global variables

What if the `x` variable from the previous example was not local but a global one? Then it would have been accessible from any point, not only from the function body. Global variables are considered [anti-pattern](#), but for the sake of the experiment, we could do this.

```

#include <stdio.h>

// now x is global variable
int x;

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

    scanf ("%d", &x);

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

    return 0;
};

```

### 7.2.1 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

```

In this case the `x` variable is defined in the `_DATA` segment and no memory is allocated in the local stack. It is accessed directly, not through the stack. Uninitialized global variables take no space in the executable file (indeed, why one needs to allocate space for variables initially set to zero?), but when someone accesses their address, the OS will allocate a block of zeroes there<sup>1</sup>.

Now let's explicitly assign a value to the variable:

```
int x=10; // default value
```

We got:

```

_DATA      SEGMENT
_x          DD          0aH

...

```

Here we see a value `0xA` of `DWORD` type (`DD` stands for `DWORD` = 32 bit) for this variable.

If you open the compiled `.exe` in [IDA](#), you can see the `x` variable placed at the beginning of the `_DATA` segment, and after it you can see text strings.

If you open the compiled `.exe` from the previous example in [IDA](#), where the value of `x` was not set, you would see something like this:

```

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

```

`_x` is marked with `?` with the rest of the variables that do not need to be initialized. This implies that after loading the `.exe` to the memory, a space for all these variables is to be allocated and filled with zeroes [[ISO07](#), 6.7.8p10]. But in the `.exe` file these uninitialized variables do not occupy anything. This is convenient for large arrays, for example.

<sup>1</sup>That is how a [VM](#) behaves

## 7.2.2 MSVC: x86 + OllyDbg

Things are even simpler here:

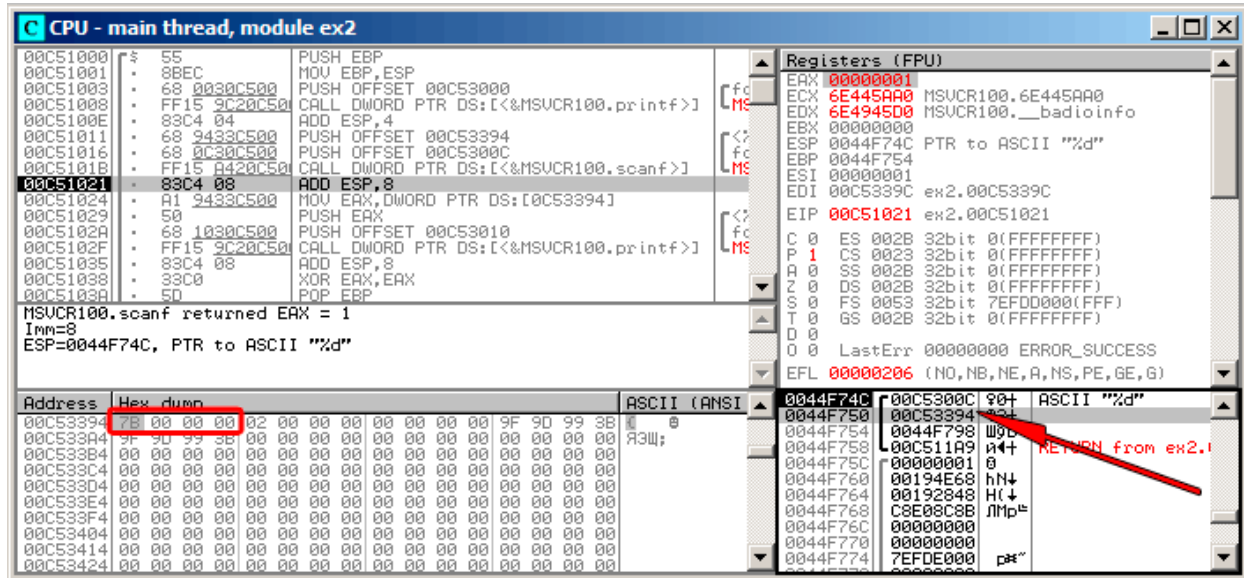


Figure 7.5: OllyDbg: after scanf() execution

The variable is located in the data segment. After the PUSH instruction (pushing the address of  $x$ ) gets executed, the address appears in the stack window. Right-click on that row and select “Follow in dump”. The variable will appear in the memory window on the left.

After we have entered 123 in the console, 0x7B appears in the memory window (see the highlighted screenshot regions).

But why is the first byte 7B? Thinking logically, 00 00 00 7B should be there. The cause for this is referred as [endianness](#), and x86 uses *little-endian*. This implies that the lowest byte is written first, and the highest written last. Read more about it at: [31 on page 434](#).

Back to the example, the 32-bit value is loaded from this memory address into EAX and passed to printf().

The memory address of  $x$  is 0x00C53394.

In OllyDbg we can review the process memory map (Alt-M) and we can see that this address is inside the `.data` PE-segment of our program:

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00070000	00007000			Heap	Map	R	R	C:\Windows\System32\locale.nls
00190000	00005000				Priv	RW	RW	
00200000	00007000				Priv	RW	Gua	Gua
0044C000	00001000				Priv	RW	Gua	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	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	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	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	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	.rsrc	Resources	Img	R	RWE	Cop
77E00000	00009000	ntdll	.data	Data	Img	RW	RWE	Cop

Figure 7.6: OllyDbg: process memory map

### 7.2.3 GCC: x86

The picture in Linux is near the same, with the difference that the uninitialized variables are located in the `_bss` segment. In `ELF` file this segment has the following attributes:

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

If you, however, initialise the variable with some value e.g. 10, it is to be placed in the `_data` segment, which has the following attributes:

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

### 7.2.4 MSVC: x64

Listing 7.6: 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:'
    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

        ; return 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main     ENDP
_TEXT    ENDS

```

The code is almost the same as in x86. Please note that the address of the *x* variable is passed to `scanf()` using a `LEA` instruction, while the variable's value is passed to the second `printf()` using a `MOV` instruction. `DWORD PTR` is a part of the assembly language (no relation to the machine code), indicating that the variable data size is 32-bit and the `MOV` instruction has to be encoded accordingly.

## 7.2.5 ARM: Optimizing Keil 6/2013 (Thumb mode)

```

.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

```

So, the *x* variable is now global and for this reason located in another segment, namely the data segment (*.data*). One could ask, why are the text strings located in the code segment (*.text*) and *x* is located right here? Because it is a variable and by definition its value could change. Moreover it could possibly change often. While text strings has constant type, they will not be changed, so they are located in the *.text* segment. The code segment might sometimes be located in a [ROM<sup>2</sup>](#) chip (remember, we now deal with embedded microelectronics, and memory scarcity is common here), and changeable

<sup>2</sup>Read-only memory

variables – in [RAM](#)<sup>3</sup>. It is not very economical to store constant variables in RAM when you have ROM. Furthermore, constant variables in RAM must be initialized, because after powering on, the RAM, obviously, contains random information.

Moving forward, we see a pointer to the `x` (`off_2C`) variable in the code segment, and that all operations with the variable occur via this pointer. That is because the `x` variable could be located somewhere far from this particular code fragment, so its address must be saved somewhere in close proximity to the code. The LDR instruction in Thumb mode can only address variables in a range of 1020 bytes from its location, and in ARM-mode – variables in range of  $\pm 4095$  bytes. And so the address of the `x` variable must be located somewhere in close proximity, because there is no guarantee that the linker would be able to accommodate the variable somewhere nearby the code, it may well be even in an external memory chip!

One more thing: if a variable is declared as *const*, the Keil compiler allocates it in the `.constdata` segment. Perhaps, thereafter, the linker could place this segment in ROM too, along with the code segment.

## 7.2.6 ARM64

Listing 7.7: Non-optimizing 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  ; save FP and LR in stack frame:
10     stp    x29, x30, [sp, -16]!
11  ; set stack frame (FP=SP)
12     add    x29, sp, 0
13  ; load pointer to the "Enter X:" string:
14     adrp   x0, .LC0
15     add    x0, x0, :lo12:LC0
16     bl     puts
17  ; load pointer to the "%d" string:
18     adrp   x0, .LC1
19     add    x0, x0, :lo12:LC1
20  ; form address of x global variable:
21     adrp   x1, x
22     add    x1, x1, :lo12:x
23     bl     __isoc99_scanf
24  ; form address of x global variable again:
25     adrp   x0, x
26     add    x0, x0, :lo12:x
27  ; load value from memory at this address:
28     ldr    w1, [x0]
29  ; load pointer to the "You entered %d...\n" string:
30     adrp   x0, .LC2
31     add    x0, x0, :lo12:LC2
32     bl     printf
33  ; return 0
34     mov    w0, 0
35  ; restore FP and LR:
36     ldp    x29, x30, [sp], 16
37     ret

```

In this case the `x` variable is declared as global and its address is calculated using the ADRP/ADD instruction pair (lines 21 and 25).

## 7.2.7 MIPS

### Uninitialized global variable

So now the `x` variable is global. Let's compile to executable file rather than object file and load it into [IDA](#). IDA displays the `x` variable in the `.sbss` ELF section (remember the "Global Pointer"? [3.5.1 on page 18](#)), since the variable is not initialized at the start.

<sup>3</sup>Random-access memory

Listing 7.8: Optimizing GCC 4.4.5 (IDA)

```

.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10      = -0x10
.text:004006C0 var_4      = -4
.text:004006C0
; function prologue:
.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)
; call 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
; call scanf():
.text:004006E4          lw       $gp, 0x20+var_10($sp)
.text:004006E8          lui      $a0, 0x40
.text:004006EC          la       $t9, __isoc99_scanf
; prepare address of x:
.text:004006F0          la       $a1, x
.text:004006F4          jalr     $t9 ; __isoc99_scanf
.text:004006F8          la       $a0, aD          # "%d"          ; branch delay slot
; call printf():
.text:004006FC          lw       $gp, 0x20+var_10($sp)
.text:00400700          lui      $a0, 0x40
; get address of x:
.text:00400704          la       $v0, x
.text:00400708          la       $t9, printf
; load value from "x" variable and pass it to printf() in $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
; function epilogue:
.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 reduces the amount of information, so we'll also do a listing using objdump and comment it:

Listing 7.9: Optimizing GCC 4.4.5 (objdump)

```

1 004006c0 <main>:
2 ; function prologue:
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 ; call 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 ; call scanf():
14 4006e4: 8fbc0010      lw      gp,16(sp)
15 4006e8: 3c040040      lui     a0,0x40
16 4006ec: 8f998038      lw      t9,-32712(gp)

```



```

17 ; prepare address of x:
18 4006f0:      8f858044      lw      a1,-32700(gp)
19 4006f4:      0320f809      jalr     t9
20 4006f8:      248408fc      addiu    a0,a0,2300 ; branch delay slot
21 ; call printf():
22 4006fc:      8fbc0010      lw      gp,16(sp)
23 400700:      3c040040      lui      a0,0x40
24 ; get address of x:
25 400704:      8f828044      lw      v0,-32700(gp)
26 400708:      8f99803c      lw      t9,-32708(gp)
27 ; load value from "x" variable and pass it to printf() in $a1:
28 40070c:      8c450000      lw      a1,0(v0)
29 400710:      0320f809      jalr     t9
30 400714:      24840900      addiu    a0,a0,2304 ; branch delay slot
31 ; function epilogue:
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 ; pack of NOPs used for aligning next function start on 16-byte boundary:
37 400728:      00200825      move     at,at
38 40072c:      00200825      move     at,at

```

Now we see the *x* variable address is read from a 64KiB data buffer using GP and adding negative offset to it (line 18). More than that, the addresses of the three external functions which are used in our example (`puts()`, `scanf()`, `printf()`), are also read from the 64KiB global data buffer using GP (lines 9, 16 and 26). GP points to the middle of the buffer, and such offset suggests that all three function's addresses, and also the address of the *x* variable, are all stored somewhere at the beginning of that buffer. That make sense, because our example is tiny.

Another thing worth mentioning is that the function ends with two **NOPs** (`MOVE $AT, $AT` – an idle instruction), in order to align next function's start on 16-byte boundary.

### Initialized global variable

Let's alter our example by giving the *x* variable a default value:

```
int x=10; // default value
```

Now IDA shows that the *x* variable is residing in the .data section:

Listing 7.10: Optimizing 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)
.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)
; prepare high part of x address:
.text:004006CC          lui      $s0, 0x41
.text:004006D0          la       $t9, __isoc99_scanf
.text:004006D4          lui      $a0, 0x40
; add low part of x address:
.text:004006D8          addiu    $a1, $s0, (x - 0x410000)
; now address of x is in $a1.
.text:004006DC          jalr     $t9 ; __isoc99_scanf
.text:004006E0          la       $a0, aD          # "%d"
.text:004006E4          lw       $gp, 0x20+var_10($sp)

```

```

; get a word from memory:
.text:004006E8      lw      $a1, x
; value of x is now in $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

```

Why not `.sdata`? Perhaps that this depends on some GCC option? Nevertheless, now `x` is in `.data`, which is a general memory area, and we can take a look how to work with variables there.

The variable's address must be formed using a pair of instructions. In our case those are LUI ("Load Upper Immediate") and ADDIU ("Add Immediate Unsigned Word").

Here is also the objdump listing for close inspection:

Listing 7.11: Optimizing GCC 4.4.5 (objdump)

```

004006a0 <main>:
4006a0: 3c1c0042      lui      gp,0x42
4006a4: 27bdffe0      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)
; prepare high part of x address:
4006cc: 3c100041      lui      s0,0x41
4006d0: 8f998038      lw       t9,-32712(gp)
4006d4: 3c040040      lui      a0,0x40
; add low part of x address:
4006d8: 26050920      addiu    a1,s0,2336
; now address of x is in $a1.
4006dc: 0320f809      jalr     t9
4006e0: 248408dc      addiu    a0,a0,2268
4006e4: 8fbc0010      lw       gp,16(sp)
; high part of x address is still in $s0.
; add low part to it and load a word from memory:
4006e8: 8e050920      lw       a1,2336(s0)
; value of x is now in $a1.
4006ec: 8f99803c      lw       t9,-32708(gp)
4006f0: 3c040040      lui      a0,0x40
4006f4: 0320f809      jalr     t9
4006f8: 248408e0      addiu    a0,a0,2272
4006fc: 8fbc001c      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

```

We see that the address is formed using LUI and ADDIU, but the high part of address is still in the `$S0` register, and it is possible to encode the offset in a LW ("Load Word") instruction, so one single LW is enough to load a value from the variable and pass it to `printf()`.

Registers holding temporary data are prefixed with T-, but here we also see some prefixed with S-, the contents of which is need to be preserved before use in other functions (i.e., "saved"). That is why the value of `$S0` was set at address `0x4006cc` and was used again at address `0x4006e8`, after the `scanf()` call. The `scanf()` function does not change its value.

## 7.3 scanf() result checking

As was noted before, it is slightly old-fashioned to use `scanf()` today. But if we have to, we need to at least check if `scanf()` finishes correctly without an error.

```
#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;
};
```

By standard, the `scanf()`<sup>4</sup> function returns the number of fields it has successfully read.

In our case, if everything goes fine and the user enters a number `scanf()` returns 1, or in case of error (or `EOF`<sup>5</sup>) – 0.

Let's add some C code to check the `scanf()` return value and print error message in case of an error.

This works as expected:

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

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

### 7.3.1 MSVC: x86

Here is what we get in the assembly output (MSVC 2010):

```
    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
```

The `caller` function (`main()`) needs the `callee` function (`scanf()`) result, so the `callee` returns it in the EAX register.

We check it with the help of the instruction `CMP EAX, 1` (*CoMPare*). In other words, we compare the value in the EAX register with 1.

<sup>4</sup>`scanf`, `wscanf`: [MSDN](#)

<sup>5</sup>End of file

A JNE conditional jump follows the CMP instruction. JNE stands for *Jump if Not Equal*.

So, if the value in the EAX register is not equal to 1, the CPU will pass the execution to the address mentioned in the JNE operand, in our case \$LN2@main. Passing the control to this address results in the CPU executing printf() with the argument What you entered? Huh?. But if everything is fine, the conditional jump is not be taken, and another printf() call is to be executed, with two arguments: 'You entered %d...' and the value of x.

Since in this case the second printf() has not to be executed, there is a JMP preceding it (unconditional jump). It passes the control to the point after the second printf() and just before the XOR EAX, EAX instruction, which implements return 0.

So, it could be said that comparing a value with another is *usually* implemented by CMP/Jcc instruction pair, where cc is *condition code*. CMP compares two values and sets processor flags<sup>6</sup>. Jcc checks those flags and decides to either pass the control to the specified address or not.

This could sound paradoxical, but the CMP instruction is in fact SUB (subtract). All arithmetic instructions set processor flags, not just CMP. If we compare 1 and 1,  $1 - 1$  is 0 so the ZF flag would be set (meaning that the last result was 0). In no other circumstances ZF can be set, except when the operands are equal. JNE checks only the ZF flag and jumps only if it is not set. JNE is in fact a synonym for JNZ (*Jump if Not Zero*). Assembler translates both JNE and JNZ instructions into the same opcode. So, the CMP instruction can be replaced with a SUB instruction and almost everything will be fine, with the difference that SUB alters the value of the first operand. CMP is *SUB without saving the result, but affecting flags*.

## 7.3.2 MSVC: x86: IDA

It is time to run IDA and try to do something in it. By the way, for beginners it is good idea to use /MD option in MSVC, which means that all these standard functions are not be linked with the executable file, but are to be imported from the MSVCRT\*.DLL file instead. Thus it will be easier to see which standard function are used and where.

While analysing code in IDA, it is very helpful to leave notes for oneself (and others). In instance, analysing this example, we see that JNZ is to be triggered in case of an error. So it is possible to move the cursor to the label, press “n” and rename it to “error”. Create another label—into “exit”. Here is my result:

```
.text:00401000 _main proc near
.text:00401000
.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
```

<sup>6</sup>x86 flags, see also: [wikipedia](http://wikipedia.org).

```
.text:00401050      retn
.text:00401050 _main endp
```

Now it is slightly easier to understand the code. However, it is not a good idea to comment on every instruction.

You could also hide(collapse) parts of a function in [IDA](#). To do that mark the block, then press “-” on the numerical pad and enter the text to be displayed instead.

Let’s hide two blocks and give them names:

```
.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
```

To expand previously collapsed parts of the code, use “+” on the numerical pad.

By pressing “space”, we can see how IDA represents a function as a graph:

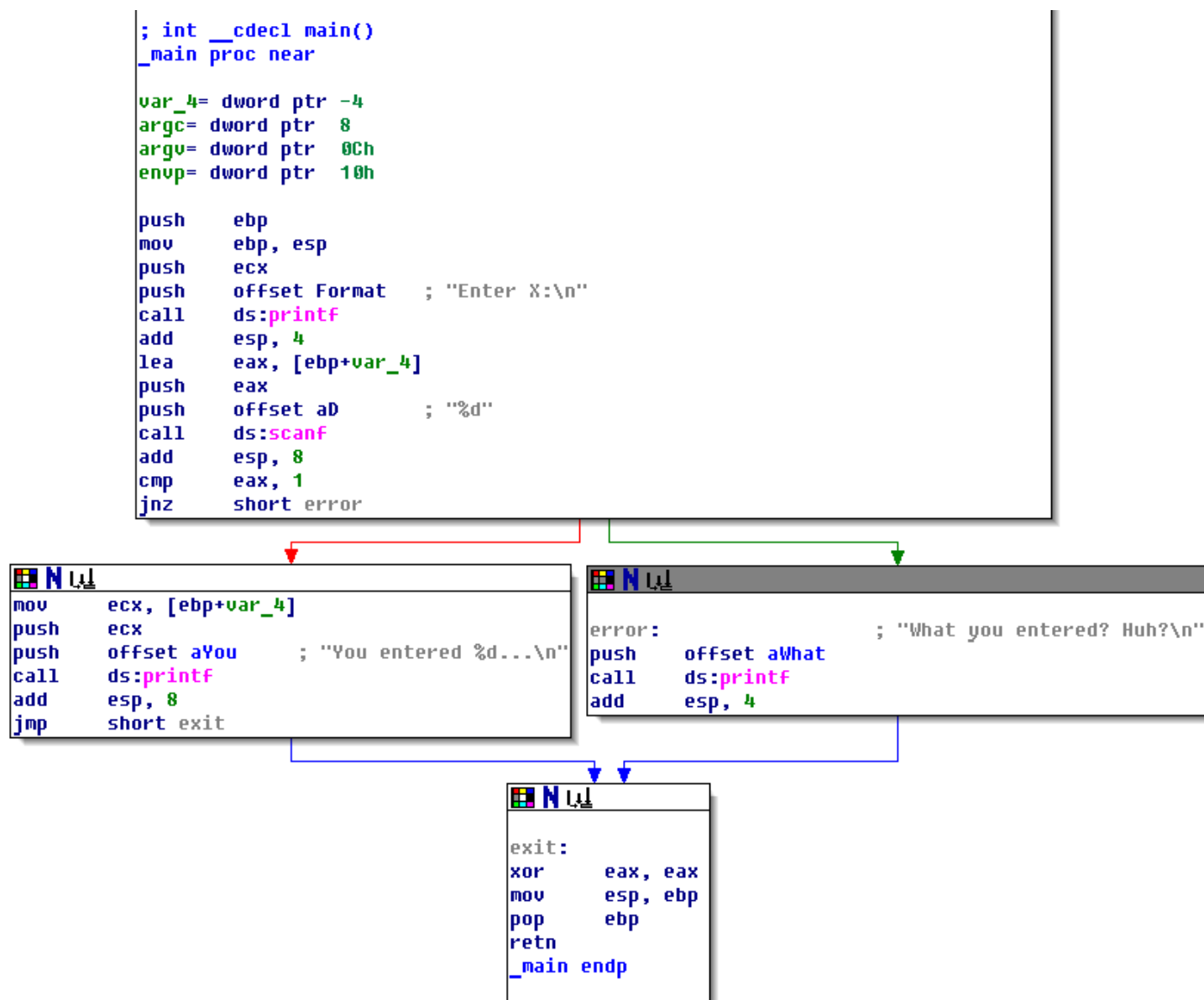


Figure 7.7: Graph mode in IDA

There are two arrows after each conditional jump: green and red. The green arrow points to the block which executes if the jump is triggered, and red if otherwise.

It is possible to fold nodes in this mode and give them names as well ("group nodes"). Let's do it for 3 blocks:

```

; 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

```

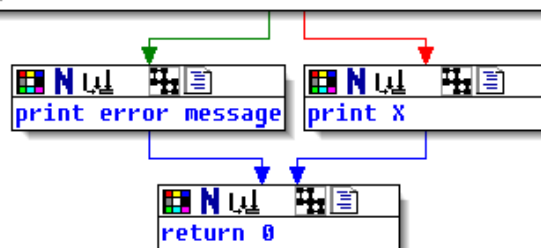


Figure 7.8: Graph mode in IDA with 3 nodes folded

That is very useful. It could be said that a very important part of the reverse engineers' job (and any other researcher as well) is to reduce the amount of information they deal with.

### 7.3.3 MSVC: x86 + OllyDbg

Let's try to hack our program in OllyDbg, forcing it to think `scanf()` always works without error.

When an address of a local variable is passed into `scanf()`, the variable initially contains some random garbage, in this case `0x6E494714`:

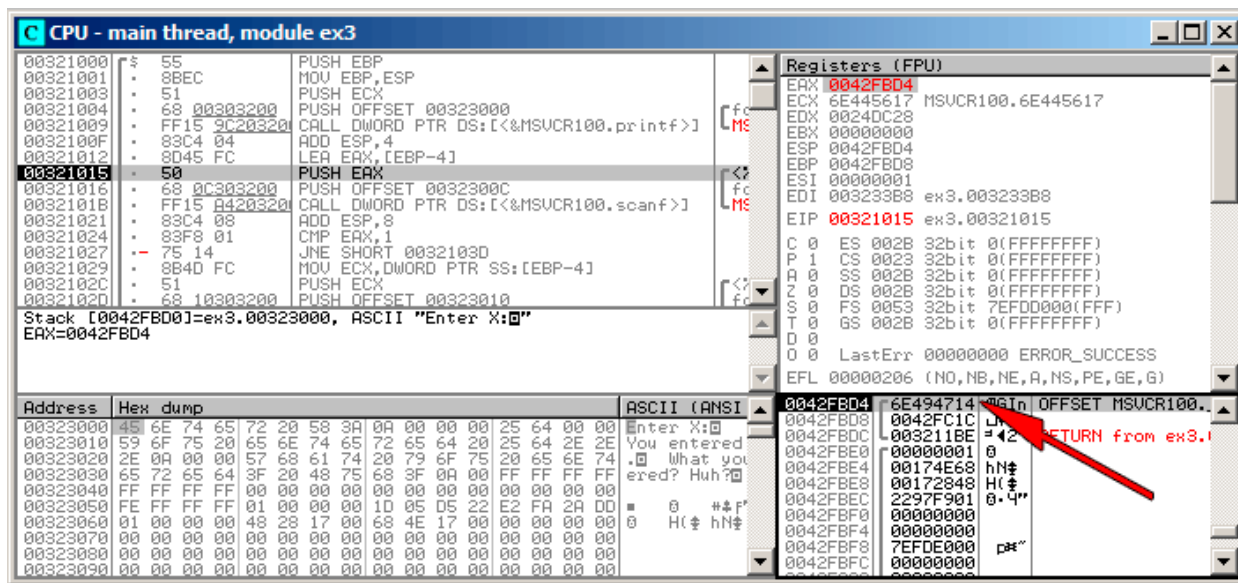


Figure 7.9: OllyDbg: passing variable address into `scanf()`



While `scanf()` executes, in the console we enter something that is definitely not a number, like “asdasd”. `scanf()` finishes with 0 in EAX, which indicates that an error has occurred:

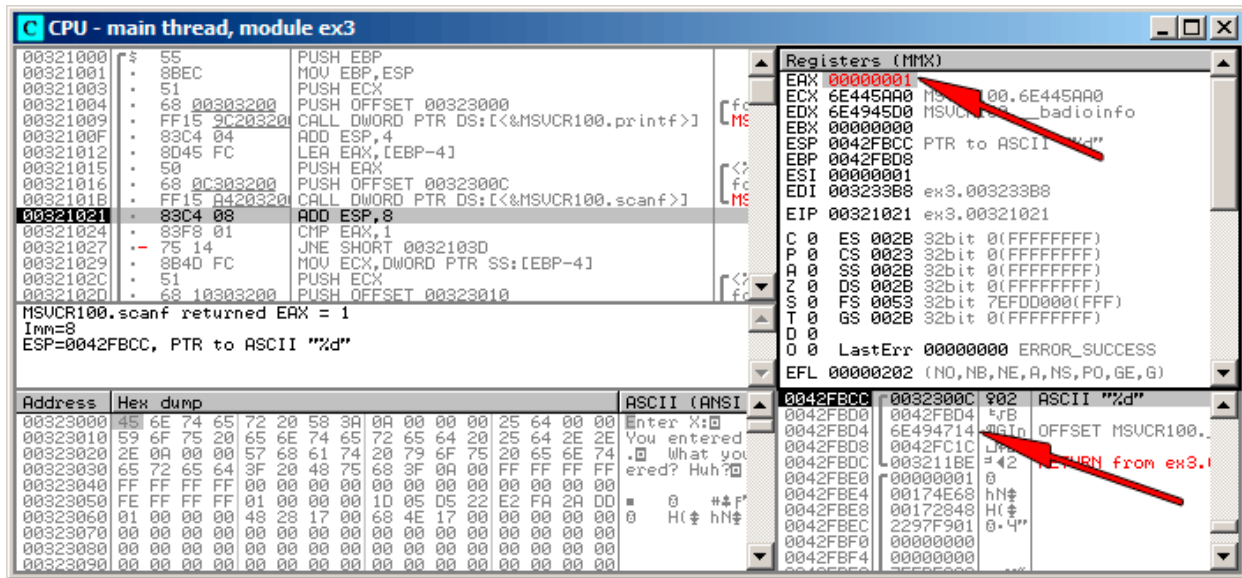


Figure 7.10: OllyDbg: `scanf()` returning error

We can also check the local variable in the stack and note that it has not changed. Indeed, what would `scanf()` write there? It simply did nothing except returning zero.

Let's try to “hack” our program. Right-click on EAX, Among the options there is “Set to 1”. This is what we need.

We now have 1 in EAX, so the following check is to be executed as intended, and `printf()` will print the value of the variable in the stack.

When we run the program (F9) we can see the following in the console window:

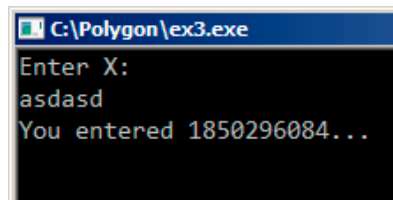


Figure 7.11: console window

Indeed, 1850296084 is a decimal representation of the number in the stack (0x6E494714)!

### 7.3.4 MSVC: x86 + Hiew

This can also be used as a simple example of executable file patching. We may try to patch the executable so the program would always print the input, no matter what we enter.

Assuming that the executable is compiled against external `MSVCR*.DLL` (i.e., with `/MD` option)<sup>7</sup>, we see the `main()` function at the beginning of the `.text` section. Let's open the executable in Hiew and find the beginning of the `.text` section (Enter, F8, F6, Enter, Enter).

We can see this:

Hiew: ex3.exe  
 C:\Polygon\ollydbg\ex3.exe    FRO -----    a32 PE .00401000 | Hiew 8.02 (c)SEN

Address	Disassembly	Comment
00401000:	55	push ebp
00401001:	8BEC	mov ebp, esp
00401003:	51	push ecx
00401004:	6800304000	push 000403000 ; 'Enter X: ' --E1
00401009:	FF1594204000	call printf
0040100F:	83C404	add esp, 4
00401012:	8D45FC	lea eax, [ebp] [-4]
00401015:	50	push eax
00401016:	680C304000	push 00040300C --E2
0040101B:	FF158C204000	call scanf
00401021:	83C408	add esp, 8
00401024:	83F801	cmp eax, 1
00401027:	7514	jnz .00040103D --E3
00401029:	8B4DFC	mov ecx, [ebp] [-4]
0040102C:	51	push ecx
0040102D:	6810304000	push 000403010 ; 'You entered %d...' --E4
00401032:	FF1594204000	call printf
00401038:	83C408	add esp, 8
0040103B:	EB0E	jmps .00040104B --E5
0040103D:	6824304000	3push 000403024 ; 'What you entered? Huh?' --E6
00401042:	FF1594204000	call printf
00401048:	83C404	add esp, 4
0040104B:	33C0	5xor eax, eax
0040104D:	8BE5	mov esp, ebp
0040104F:	5D	pop ebp
00401050:	C3	retn ;
00401051:	B84D5A0000	mov eax, 00005A4D ; ' ZM'

1Global 2FillBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 9byte 10Leave 11Naked 12AddNam

Figure 7.12: Hiew: `main()` function

Hiew finds `ASCIIZ`<sup>8</sup> strings and displays them, as it does with the imported functions' names.

<sup>7</sup>that's what also called "dynamic linking"

<sup>8</sup>ASCII Zero (null-terminated ASCII string)

Move the cursor to address .00401027 (where the JNZ instruction, we have to bypass, is located), press F3, and then type “9090”, meaning two **NOPs**):

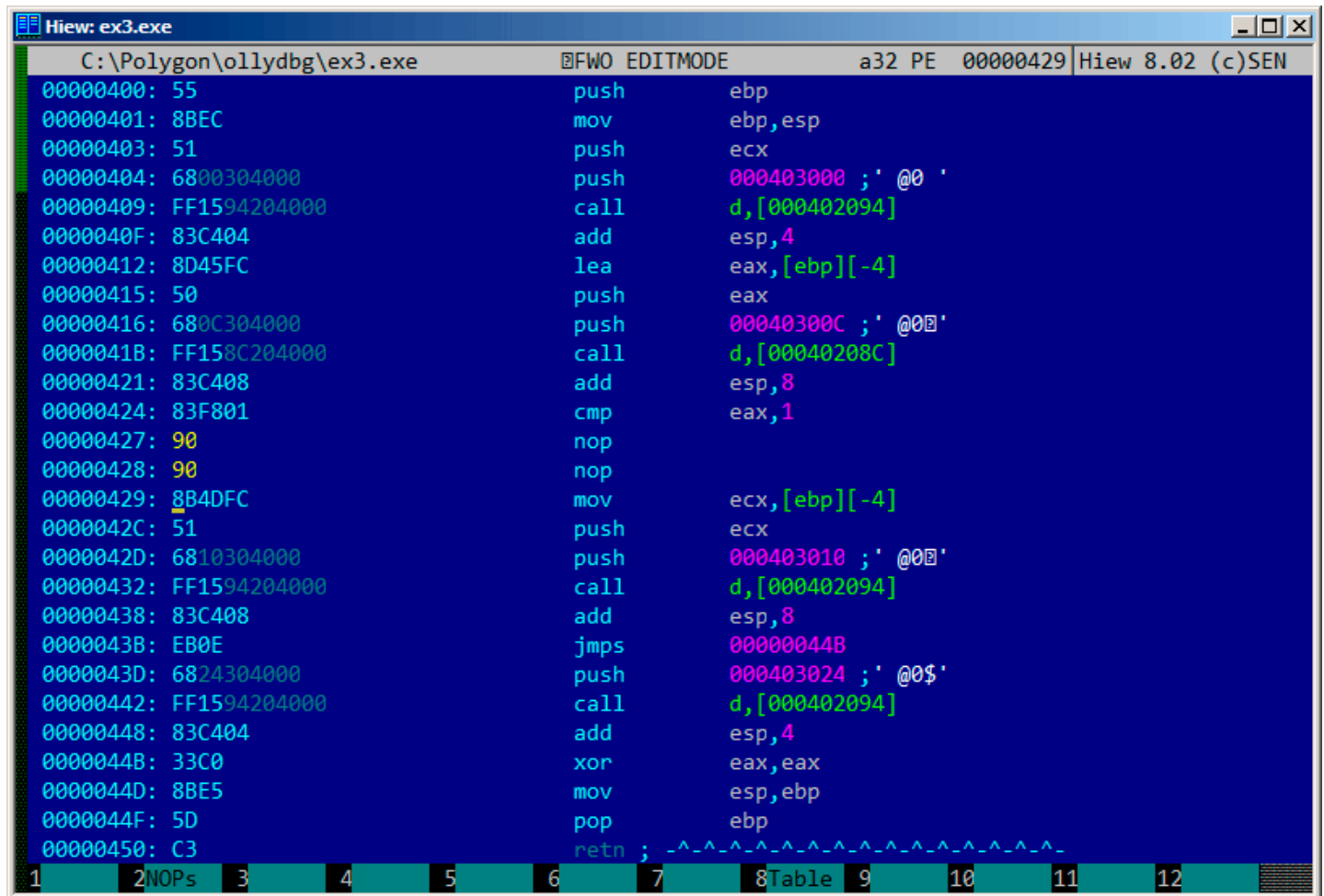


Figure 7.13: Hiew: replacing JNZ with two NOPs

Then press F9 (update). Now the executable is saved to the disk. It will behave as we wanted.

Two **NOPs** are probably not the most æsthetic approach. Another way to patch this instruction is to write just 0 to the second opcode byte (**jump offset**), so that JNZ will always jump to the next instruction.

We could also do the opposite: replace first byte with EB while not touching the second byte ([jump offset](#)). We would get an unconditional jump that is always triggered. In this case the error message would be printed every time, no matter the input.

### 7.3.5 MSVC: x64

Since we work here with *int*-typed variables, which are still 32-bit in x86-64, we see how the 32-bit part of the registers (prefixed with E-) are used here as well. While working with pointers, however, 64-bit register parts are used, prefixed with R-.

Listing 7.12: 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
$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:
    ; return 0
    xor    eax, eax
    add    rsp, 56
    ret    0
main      ENDP
_TEXT     ENDS
END

```

### 7.3.6 ARM

#### ARM: Optimizing Keil 6/2013 (Thumb mode)

Listing 7.13: Optimizing Keil 6/2013 (Thumb mode)

```

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

```

The new instructions here are `CMP` and `BEQ`<sup>9</sup>.

`CMP` is analogous to the x86 instruction with the same name, it subtracts one of the arguments from the other and updates the conditional flags if needed.

`BEQ` jumps to another address if the operands were equal to each other, or, if the result of the last computation was 0, or if the Z flag is 1. It behaves as `JZ` in x86.

Everything else is simple: the execution flow forks in two branches, then the branches converge at the point where 0 is written into the R0 as a function return value, and then the function ends.

#### ARM64

<sup>9</sup>(PowerPC, ARM) Branch if Equal

Listing 7.14: Non-optimizing 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 ; save FP and LR in stack frame:
11     stp    x29, x30, [sp, -32]!
12 ; set stack frame (FP=SP)
13     add    x29, sp, 0
14 ; load pointer to the "Enter X:" string:
15     adrp   x0, .LC0
16     add    x0, x0, :lo12:LC0
17     bl     puts
18 ; load pointer to the "%d" string:
19     adrp   x0, .LC1
20     add    x0, x0, :lo12:LC1
21 ; calculate address of x variable in the local stack
22     add    x1, x29, 28
23     bl     __isoc99_scanf
24 ; scanf() returned result in W0.
25 ; check it:
26     cmp    w0, 1
27 ; BNE is Branch if Not Equal
28 ; so if W0<>0, jump to L2 will be occurred
29     bne    .L2
30 ; at this moment W0=1, meaning no error
31 ; load x value from the local stack
32     ldr    w1, [x29,28]
33 ; load pointer to the "You entered %d...\n" string:
34     adrp   x0, .LC2
35     add    x0, x0, :lo12:LC2
36     bl     printf
37 ; skip the code, which print the "What you entered? Huh?" string:
38     b      .L3
39 .L2:
40 ; load pointer to the "What you entered? Huh?" string:
41     adrp   x0, .LC3
42     add    x0, x0, :lo12:LC3
43     bl     puts
44 .L3:
45 ; return 0
46     mov    w0, 0
47 ; restore FP and LR:
48     ldp    x29, x30, [sp], 32
49     ret

```

Code flow in this case forks with the use of CMP/BNE (Branch if Not Equal) instructions pair.

### 7.3.7 MIPS

Listing 7.15: Optimizing 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
.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()` returns the result of its work in register `$V0`. It is checked at address `0x004006E4` by comparing the values in `$V0` with `$V1` (1 was stored in `$V1` earlier, at `0x004006DC`). `BEQ` stands for “Branch Equal”. If the two values are equal (i.e., success), the execution jumps to address `0x0040070C`.

### 7.3.8 Exercise

As we can see, the `JNE/JNZ` instruction can be easily replaced by the `JE/JZ` and vice versa (or `BNE` by `BEQ` and vice versa). But then the basic blocks must also be swapped. Try to do this in some of the examples.

## 7.4 Exercise

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

## Chapter 8

# Accessing passed arguments

Now we figured out that the [caller](#) function is passing arguments to the [callee](#) via the stack. But how does the [callee](#) access them?

Listing 8.1: simple example

```
#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;
};
```

## 8.1 x86

### 8.1.1 MSVC

Here is what we get after compilation (MSVC 2010 Express):

Listing 8.2: 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 ; 3rd argument
    push    2 ; 2nd argument
    push    1 ; 1st argument
    call    _f
    add     esp, 12
    push    eax
    push    OFFSET $SG2463 ; '%d', 0aH, 00H
```

```

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

```

What we see is that the `main()` function pushes 3 numbers onto the stack and calls `f(int, int, int)`. Argument access inside `f()` is organized with the help of macros like: `_a$ = 8`, in the same way as local variables, but with positive offsets (addressed with *plus*). So, we are addressing the *outer* side of the [stack frame](#) by adding the `_a$` macro to the value in the EBP register.

Then the value of `a` is stored into EAX. After `IMUL` instruction execution, the value in EAX is a [product](#) of the value in EAX and the content of `_b`. After that, `ADD` adds the value in `_c` to EAX. The value in EAX does not need to be moved: it is already where it must be. On returning to [caller](#), it takes the EAX value and use it as an argument to `printf()`.

### 8.1.2 MSVC + OllyDbg

Let's illustrate this in OllyDbg. When we trace to the first instruction in `f()` that uses one of the arguments (first one), we see that EBP is pointing to the [stack frame](#), which is marked with a red rectangle. The first element of the [stack frame](#) is the saved value of EBP, the second one is [RA](#), the third is the first function argument, then the second and third ones. To access the first function argument, one needs to add exactly 8 (2 32-bit words) to EBP.

OllyDbg is aware about this, so it has added comments to the stack elements like "RETURN from" and "Arg1 = ...", etc.

N.B.: Function arguments are not members of the function's stack frame, they are rather members of the stack frame of the [caller](#) function. Hence, OllyDbg marked "Arg" elements as members of another stack frame.

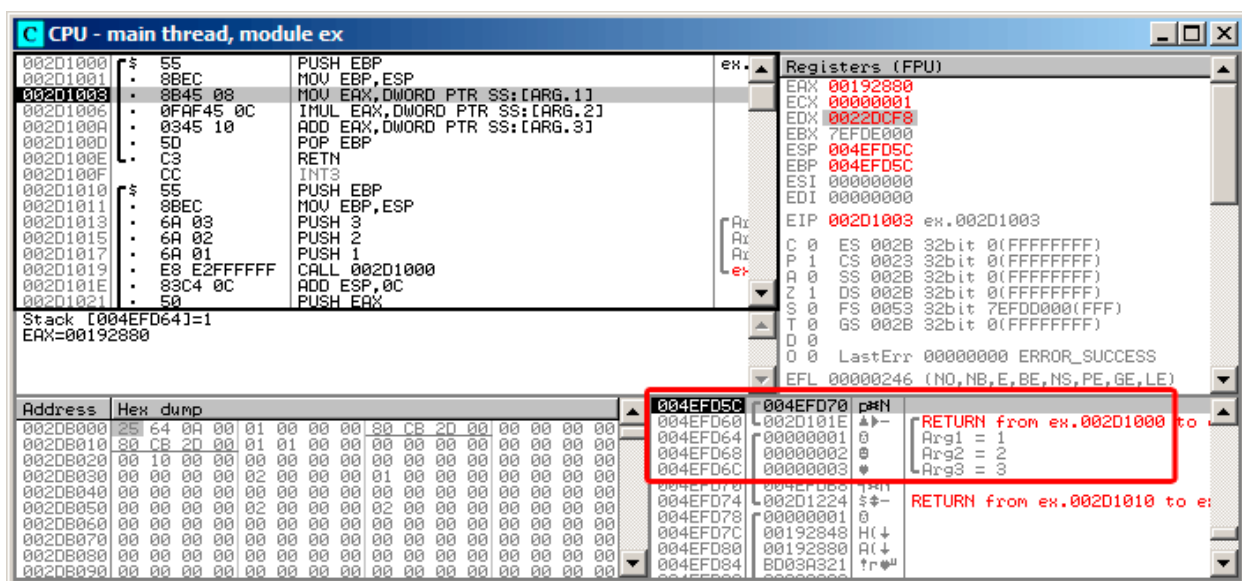


Figure 8.1: OllyDbg: inside of `f()` function

### 8.1.3 GCC

Let's compile the same in GCC 4.4.1 and see the results in [IDA](#):

Listing 8.3: GCC 4.4.1

```

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] ; 1st argument

```



```

    imul    eax, [ebp+arg_4] ; 2nd argument
    add     eax, [ebp+arg_8] ; 3rd argument
    pop     ebp
    retn
f
endp

main      public main
          proc near

var_10    = 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 ; 3rd argument
          mov     [esp+10h+var_C], 2 ; 2nd argument
          mov     [esp+10h+var_10], 1 ; 1st argument
          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

```

The result is almost the same with some minor differences discussed earlier.

The [stack pointer](#) is not set back after the two function calls(f and printf), because the penultimate LEAVE ([A.6.2 on page 886](#)) instruction takes care of this at the end.

## 8.2 x64

The story is a bit different in x86-64. Function arguments (first 4 or first 6 of them) are passed in registers i.e. the [callee](#) reads them from registers instead of reading them from the stack.

### 8.2.1 MSVC

Optimizing MSVC:

Listing 8.4: Optimizing 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'
          mov     edx, eax
          call    printf
          xor     eax, eax
          add     rsp, 40
          ret     0
main      ENDP

f         PROC
          ; ECX - 1st argument
          ; EDX - 2nd argument
          ; R8D - 3rd argument
          imul    ecx, edx

```

```

    lea    eax, DWORD PTR [r8+rcx]
    ret    0
f        ENDP

```

As we can see, the compact function `f()` takes all its arguments from the registers. The LEA instruction here is used for addition, apparently the compiler considered it faster than ADD. LEA is also used in the `main()` function to prepare the first and third `f()` arguments. The compiler must have decided that this would work faster than the usual way of loading values into a register using MOV instruction.

Let's take a look at the non-optimizing MSVC output:

Listing 8.5: MSVC 2012 x64

```

f                proc near
; shadow space:
arg_0            = dword ptr 8
arg_8            = dword ptr 10h
arg_10           = dword ptr 18h

                ; ECX - 1st argument
                ; EDX - 2nd argument
                ; R8D - 3rd argument
                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 ; 3rd argument
                mov     edx, 2 ; 2nd argument
                mov     ecx, 1 ; 1st argument
                call    f
                mov     edx, eax
                lea     rcx, $SG2931 ; "%d\n"
                call    printf

                ; return 0
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp

```

It looks somewhat puzzling because all 3 arguments from the registers are saved to the stack for some reason. This is called “shadow space”<sup>1</sup>: every Win64 may (but is not required to) save all 4 register values there. This is done for two reasons: 1) it is too lavish to allocate a whole register (or even 4 registers) for an input argument, so it will be accessed via stack; 2) the debugger is always aware where to find the function arguments at a break<sup>2</sup>.

So, some large functions can save their input arguments in the “shadows space” if they need to use them during execution, but some small functions (like ours) may not do this.

It is a [caller](#) responsibility to allocate “shadow space” in the stack.

## 8.2.2 GCC

Optimizing GCC generates more or less understandable code:

Listing 8.6: Optimizing GCC 4.4.6 x64

```

f:
    ; EDI - 1st argument
    ; ESI - 2nd argument

```

<sup>1</sup>MSDN

<sup>2</sup>MSDN

```

; EDX - 3rd argument
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 ; number of vector registers passed
call    printf
xor     eax, eax
add     rsp, 8
ret

```

Non-optimizing GCC:

Listing 8.7: GCC 4.4.6 x64

```

f:
; EDI - 1st argument
; ESI - 2nd argument
; EDX - 3rd argument
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 ; number of vector registers passed
call    printf
mov     eax, 0
leave
ret

```

There are no “shadow space” requirements in System V\*NIX[Mit13], but the [callee](#) may need to save its arguments somewhere in case of registers shortage.

### 8.2.3 GCC: `uint64_t` instead of `int`

Our example works with 32-bit *int*, that is why 32-bit register parts are used (prefixed by E-).

It can be altered slightly in order to use 64-bit values:

```

#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;
};
```

Listing 8.8: Optimizing 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 ; 3rd argument
      mov     rsi, 1111111122222222h ; 2nd argument
      mov     rdi, 1122334455667788h ; 1st argument
      call    f
      mov     edi, offset format ; "%lld\n"
      mov     rsi, rax
      xor     eax, eax ; number of vector registers passed
      call    _printf
      xor     eax, eax
      add     rsp, 8
      retn
main   endp
```

The code is the same, but this time the *full size* registers (prefixed by R-) are used.

## 8.3 ARM

### 8.3.1 Non-optimizing Keil 6/2013 (ARM mode)

```
.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
...
.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}
```

The `main()` function simply calls two other functions, with three values passed to the first one –(`f()`).

As was noted before, in ARM the first 4 values are usually passed in the first 4 registers (R0-R3).

The `f()` function, as it seems, uses the first 3 registers (R0-R2) as arguments.

The *MLA* (*Multiply Accumulate*) instruction multiplies its first two operands (R3 and R1), adds the third operand (R2) to the product and stores the result into the zeroth register (R0), via which, by standard, functions return values.

Multiplication and addition at once<sup>3</sup> (*Fused multiply-add*) is a very useful operation. By the way, there was no such instruction in x86 before FMA-instructions appeared in SIMD<sup>4</sup>.

<sup>3</sup>Wikipedia: [Multiply-accumulate operation](#)

<sup>4</sup>wikipedia

The very first `MOV R3, R0` instruction is, apparently, redundant (a single `MLA` instruction could be used here instead). The compiler has not optimized it, since this is non-optimizing compilation.

The `BX` instruction returns the control to the address stored in the `LR` register and, if necessary, switches the processor mode from Thumb to ARM or vice versa. This can be necessary since, as we can see, function `f()` is not aware from what kind of code it may be called, ARM or Thumb. Thus, if it gets called from Thumb code, `BX` is not only returns control to the calling function, but also switches the processor mode to Thumb. Or not switch, if the function was called from ARM code [ARM12, A2.3.2].

### 8.3.2 Optimizing Keil 6/2013 (ARM mode)

```
.text:00000098          f
.text:00000098 91 20 20 E0          MLA    R0, R1, R0, R2
.text:0000009C 1E FF 2F E1          BX     LR
```

And here is the `f()` function compiled by the Keil compiler in full optimization mode (`-O3`). The `MOV` instruction was optimized out (or reduced) and now `MLA` uses all input registers and also places the result right into `R0`, exactly where the calling function will read and use it.

### 8.3.3 Optimizing Keil 6/2013 (Thumb mode)

```
.text:0000005E 48 43          MULS   R0, R1
.text:00000060 80 18          ADDS   R0, R0, R2
.text:00000062 70 47          BX     LR
```

The `MLA` instruction is not available in Thumb mode, so the compiler generates the code doing these two operations (multiplication and addition) separately. First the `MULS` instruction multiplies `R0` by `R1`, leaving the result in register `R1`. The second instruction (`ADDS`) adds the result and `R2` leaving the result in register `R0`.

### 8.3.4 ARM64

#### Optimizing GCC (Linaro) 4.9

Everything here is simple. `MADD` is just an instruction doing fused multiply/add (similar to the `MLA` we already saw). All 3 arguments are passed in the 32-bit parts of X-registers. Indeed, the argument types are 32-bit *int*'s. The result is returned in `W0`.

Listing 8.9: Optimizing GCC (Linaro) 4.9

```
f:
    madd    w0, w0, w1, w2
    ret

main:
; save FP and LR to stack frame:
    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
; return 0
    mov     w0, 0
; restore FP and LR
    ldp     x29, x30, [sp], 16
    ret

.LC7:
    .string "%d\n"
```

Let's also extend all data types to 64-bit `uint64_t` and test:

```
#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

.LC8:
    .string "%lld\n"
```

The `f()` function is the same, only the whole 64-bit X-registers are now used. Long 64-bit values are loaded into the registers by parts, this is also described here: [28.3.1 on page 426](#).

### Non-optimizing GCC (Linaro) 4.9

The non-optimizing compiler is more redundant:

```
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
```

The code saves its input arguments in the local stack, in case someone (or something) in this function needs using the `W0...W2` registers. This prevents overwriting the original function arguments, which may be needed again in the future. This is called *Register Save Area*. [\[ARM13c\]](#) The callee, however, is not obliged to save them. This is somewhat similar to “Shadow Space”: [8.2.1 on page 92](#).

Why did the optimizing GCC 4.9 drop this argument saving code? Because it did some additional optimizing work and concluded that the function arguments will not be needed in the future and also that the registers `W0...W2` will not be used.

We also see a `MUL/ADD` instruction pair instead of single a `MADD`.

## 8.4 MIPS

Listing 8.10: Optimizing 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
; result in $v0 upon return

.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)
; set c:
.text:00000024      li      $a2, 3
; set a:
.text:00000028      li      $a0, 1
.text:0000002C      jal     f
; set b:
.text:00000030      li      $a1, 2      ; branch delay slot
; result in $v0 now
.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
; take result of f() function and pass it as a second argument to 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
```

The first four function arguments are passed in four registers prefixed by A-.

There are two special registers in MIPS: HI and LO which are filled with the 64-bit result of the multiplication during the execution of the MULT instruction. These registers are accessible only by using the MFLO and MFHI instructions. MFLO here takes the low-part of the multiplication result and stores it into \$V0. So the high 32-bit part of the multiplication result is dropped (the HI register content is not used). Indeed: we work with 32-bit *int* data types here.

Finally, ADDU (“Add Unsigned”) adds the value of the third argument to the result.

There are two different addition instructions in MIPS: ADD and ADDU. The difference between them is not related to signedness, but to exceptions. ADD can raise an exception on overflow, which is sometimes useful<sup>5</sup> and supported in Ada PL, for instance. ADDU does not raise exceptions on overflow. Since C/C++ does not support this, in our example we see ADDU instead of ADD.

The 32-bit result is left in \$V0.

There is a new instruction for us in main(): JAL (“Jump and Link”). The difference between JAL and JALR is that a relative offset is encoded in the first instruction, while JALR jumps to the absolute address stored in a register (“Jump and Link Register”). Both f() and main() functions are located in the same object file, so the relative address of f() is known and fixed.

<sup>5</sup><http://go.yurichev.com/17326>

## Chapter 9

# More about results returning

In x86, the result of function execution is usually returned<sup>1</sup> in the EAX register. If it is byte type or a character (*char*), then the lowest part of register EAX (AL) is used. If a function returns a *float* number, the FPU register ST(0) is used instead. In ARM, the result is usually returned in the R0 register.

### 9.1 Attempt to use the result of a function returning *void*

So, what if the `main()` function return value was declared of type *void* and not *int*?

The so-called startup-code is calling `main()` roughly as follows:

```
push envp
push argv
push argc
call main
push eax
call exit
```

In other words:

```
exit(main(argv,argv,envp));
```

If you declare `main()` as *void*, nothing is to be returned explicitly (using the *return* statement), then something random, that was stored in the EAX register at the end of `main()` becomes the sole argument of the `exit()` function. Most likely, there will be a random value, left from your function execution, so the exit code of program is pseudorandom.

We can illustrate this fact. Please note that here the `main()` function has a *void* return type:

```
#include <stdio.h>

void main()
{
    printf ("Hello, world!\n");
};
```

Let's compile it in Linux.

GCC 4.8.1 replaced `printf()` with `puts()` (we have seen this before: [3.4.3 on page 15](#)), but that's OK, since `puts()` returns the number of characters printed out, just like `printf()`. Please notice that EAX is not zeroed before `main()`'s end. This implies that the value of EAX at the end of `main()` contains what `puts()` has left there.

Listing 9.1: GCC 4.8.1

```
.LC0:
    .string "Hello, world!"
main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 16
```

<sup>1</sup>See also: MSDN: Return Values (C++): [MSDN](#)



```

mov     DWORD PTR [esp], OFFSET FLAT:.LC0
call    puts
leave   esp, 4
ret

```

Let's write a bash script that shows the exit status:

Listing 9.2: tst.sh

```

#!/bin/sh
./hello_world
echo $?

```

And run it:

```

$ tst.sh
Hello, world!
14

```

14 is the number of characters printed.

## 9.2 What if we do not use the function result?

`printf()` returns the count of characters successfully output, but the result of this function is rarely used in practice. It is also possible to call a function whose essence is in returning a value, and not use it:

```

int f()
{
    // skip first 3 random values
    rand();
    rand();
    rand();
    // and use 4th
    return rand();
};

```

The result of the `rand()` function is left in EAX, in all four cases. But in the first 3 cases, the value in EAX is just thrown away.

## 9.3 Returning a structure

Let's go back to the fact that the return value is left in the EAX register. That is why old C compilers cannot create functions capable of returning something that does not fit in one register (usually *int*), but if one needs it, one has to return information via pointers passed as function's arguments. So, usually, if a function needs to return several values, it returns only one, and all the rest—via pointers. Now it has become possible to return, let's say, an entire structure, but that is still not very popular. If a function has to return a large structure, the [caller](#) must allocate it and pass a pointer to it via the first argument, transparently for the programmer. That is almost the same as to pass a pointer in the first argument manually, but the compiler hides it.

Small example:

```

struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;
}

```

```
    return rt;
};
```

...what we got (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], ecx
    mov     DWORD PTR [eax+8], ecx
    ret     0
?get_some_values@@YA?AUs@@H@Z ENDP          ; get_some_values
```

The macro name for internal passing of pointer to a structure here is \$T3853.

This example can be rewritten using the C99 language extensions:

```
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};
};
```

Listing 9.3: 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
```

As we see, the function is just filling the structure's fields allocated by the caller function, as if a pointer to the structure was passed. So there are no performance drawbacks.

# Chapter 10

## Pointers

Pointers are often used to return values from functions (recall `scanf( )` case ([7 on page 59](#))). For example, when a function needs to return two values.

### 10.1 Global variables example

```
#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);
};
```

This compiles to:

Listing 10.1: Optimizing 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
    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
```

Let's see this in OllyDbg:

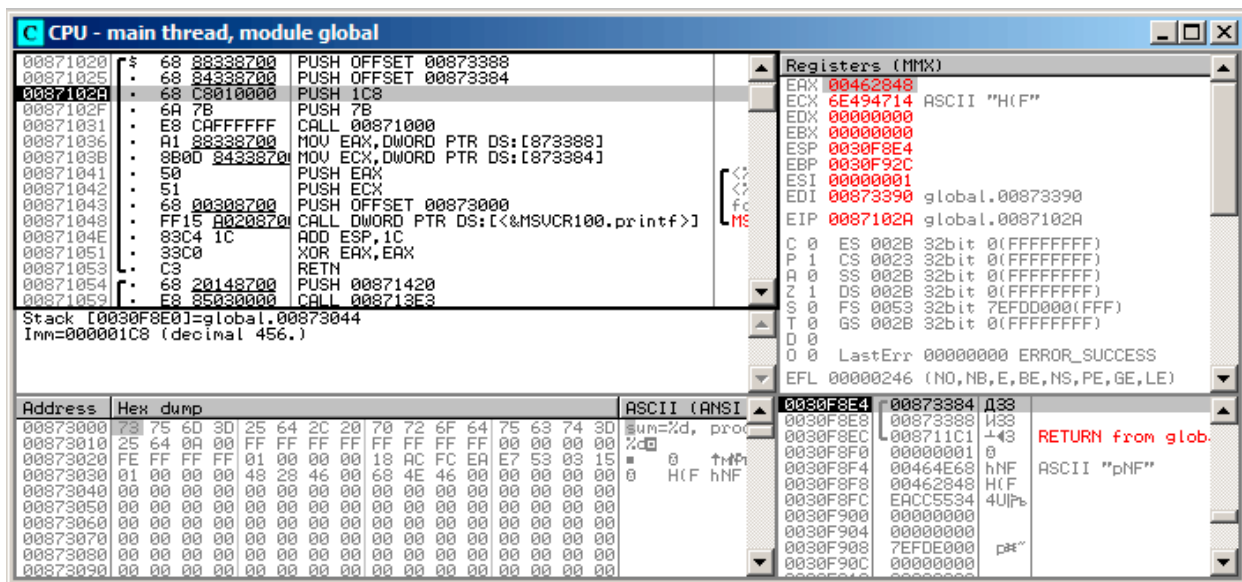
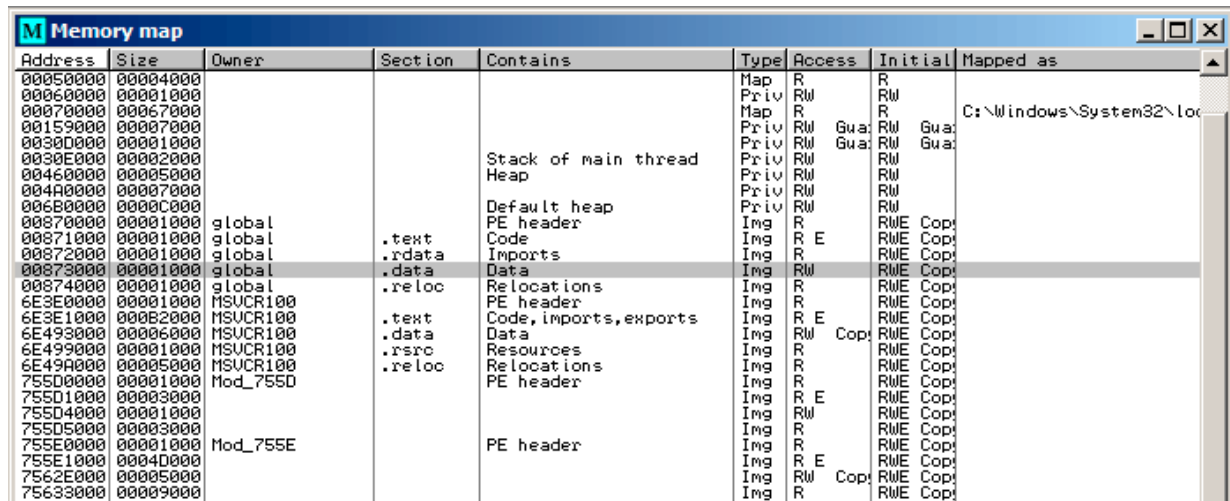


Figure 10.1: OllyDbg: global variables addresses are passed to f1()

First, global variables' addresses are passed to f1(). We can click "Follow in dump" on the stack element, and we can see the place in the data segment allocated for the two variables.

These variables are zeroed, because non-initialized data (from [BSS](#)) is cleared before the execution begins: [\[ISO07, 6.7.8p10\]](#). They reside in the data segment, we can verify this by pressing Alt-M and reviewing the 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	C:\Windows\System32\loc
00159000	00007000				Priv	RW	Gua: RW	Gua: RW
0030D000	00001000				Priv	RW	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	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	RWE Cop	
75633000	00009000				Img	R	RWE Cop	

Figure 10.2: OllyDbg: memory map

Let's trace (F7) to the start of `f1()`:

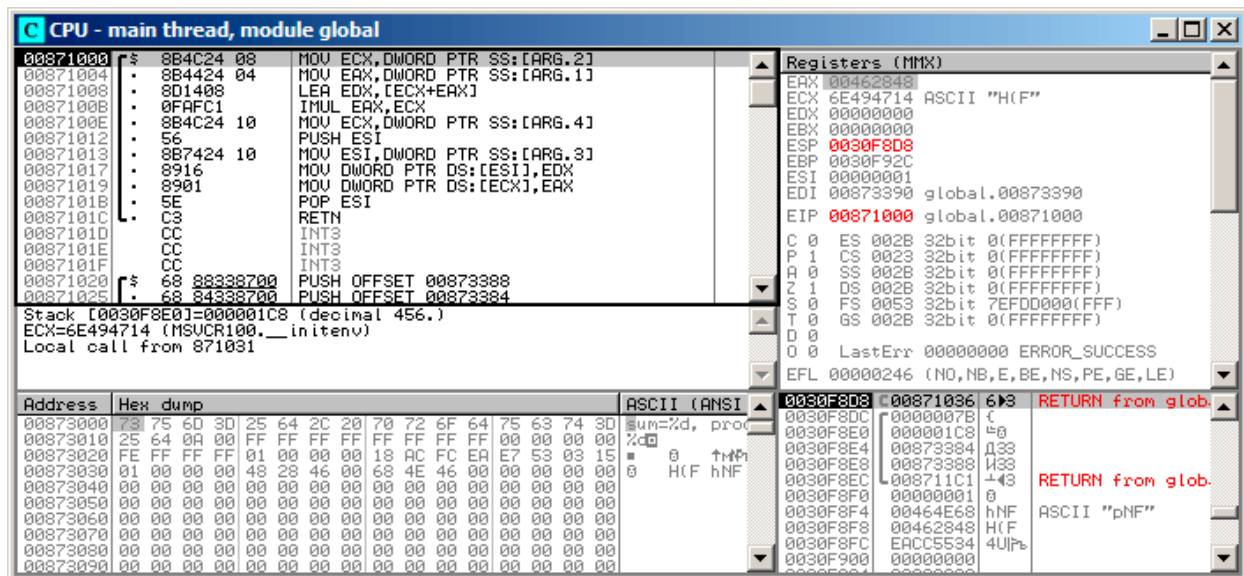


Figure 10.3: OllyDbg: `f1()` starts

Two values are visible in the stack 456 (0x1C8) and 123 (0x7B), and also the addresses of the two global variables.

Let's trace until the end of `f1()`. In the left bottom window we see how the results of the calculation appear in the global variables:

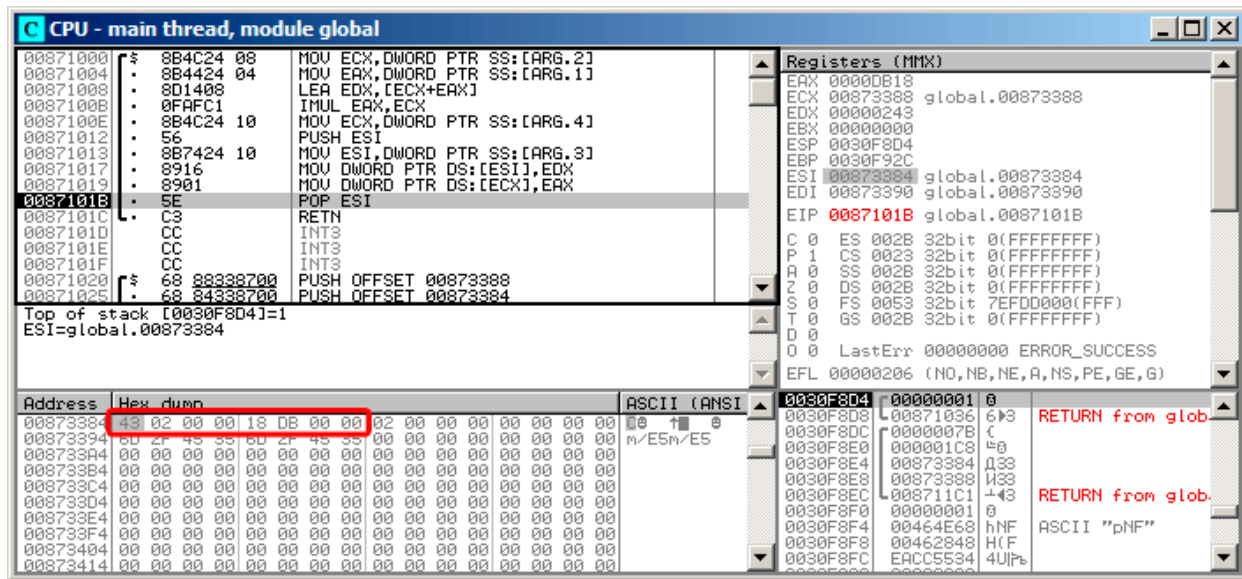


Figure 10.4: OllyDbg: `f1()` execution completed



Now the global variables' values are loaded into registers ready for passing to `printf()` (via the stack):

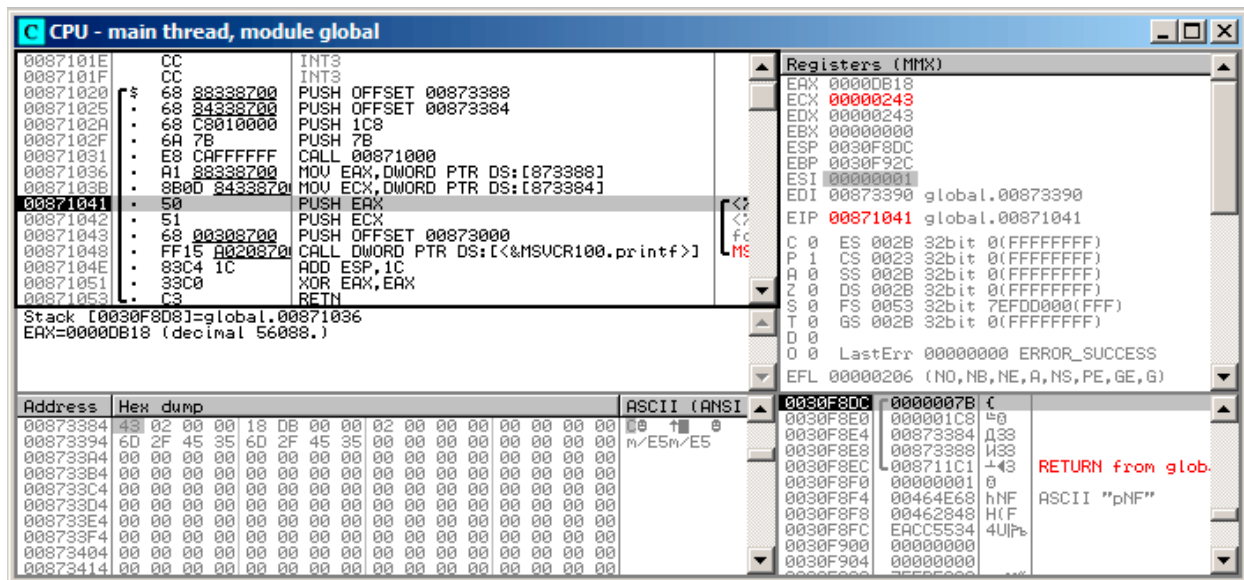


Figure 10.5: OllyDbg: global variables' addresses are passed into `printf()`

## 10.2 Local variables example

Let's rework our example slightly:

Listing 10.2: now the sum and product variables are local

```

void main()
{
    int sum, product; // now variables are local in this function

    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
};

```

`f1()` code will not change. Only the code of `main()` will do:

Listing 10.3: Optimizing 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

```

Let's look again with OllyDbg. The addresses of the local variables in the stack are 0x2EF854 and 0x2EF858. We see how these are pushed into the stack:

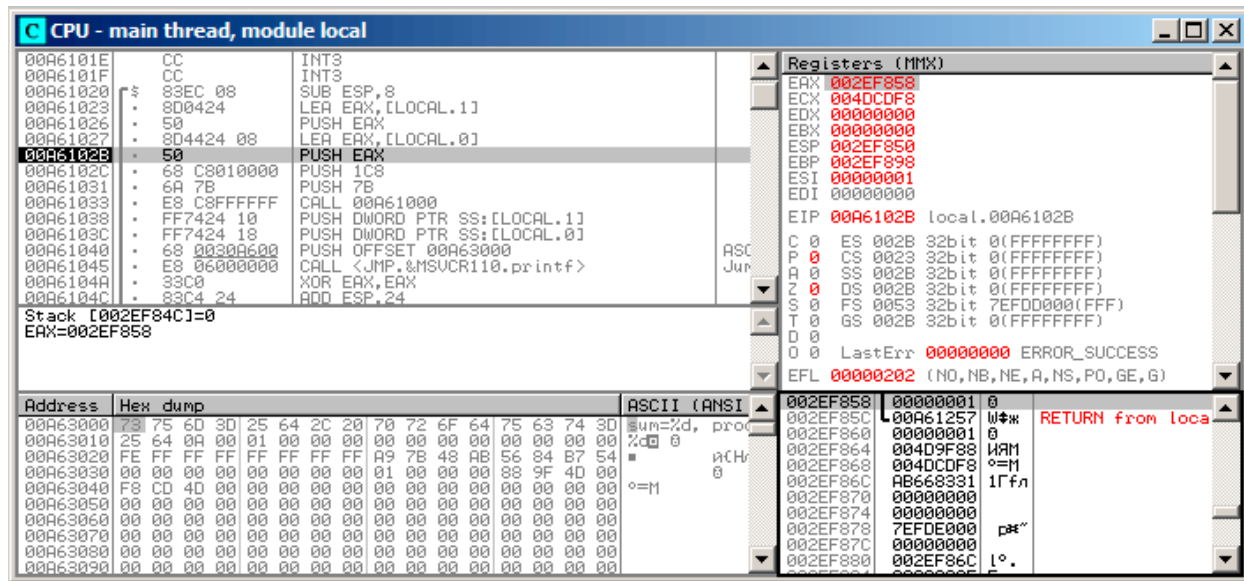


Figure 10.6: OllyDbg: local variables' addresses are pushed into the stack

f1() starts. So far there is only random garbage in the stack at 0x2EF854 and 0x2EF858:

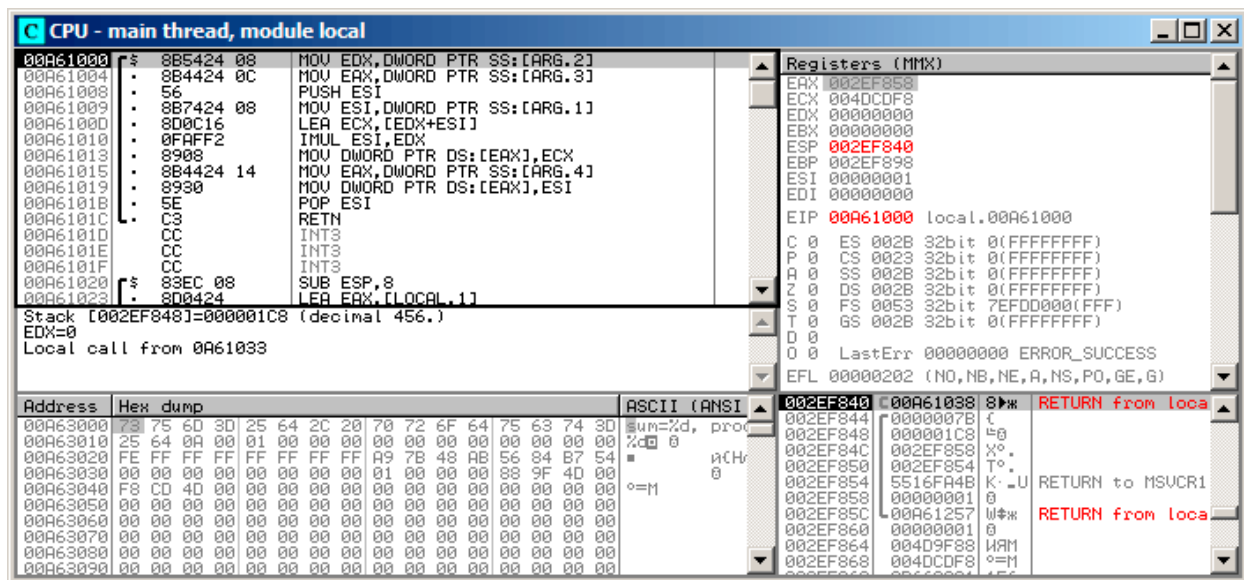


Figure 10.7: OllyDbg: f1() starting

f1() completes:

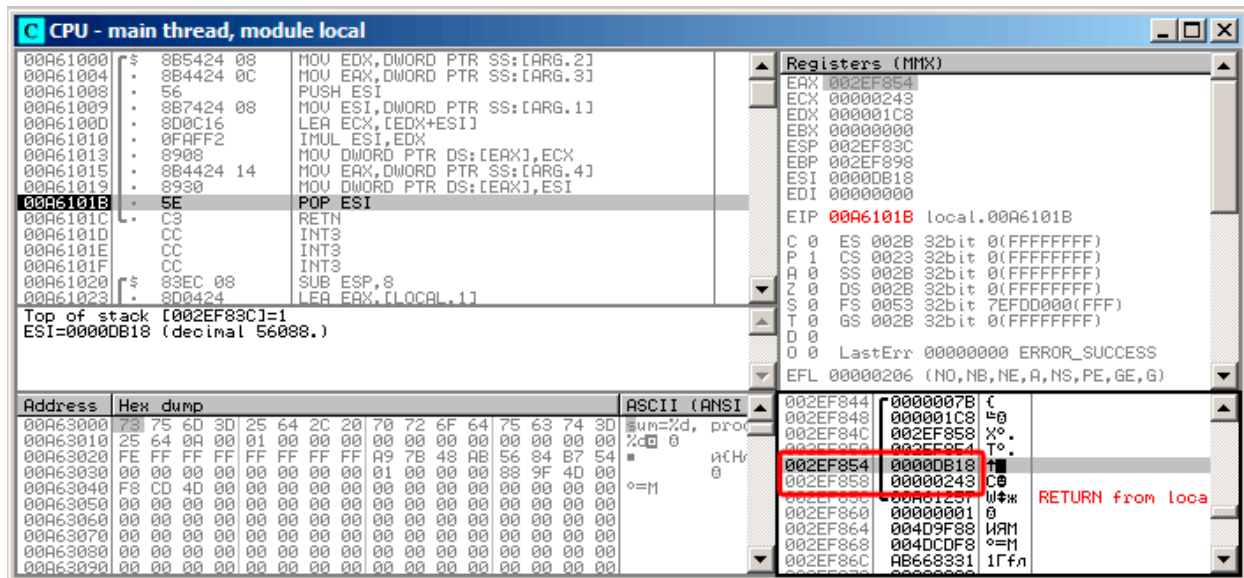


Figure 10.8: OllyDbg: f1() completes execution

We now find 0xDB18 and 0x243 at addresses 0x2EF854 and 0x2EF858. These values are the f1() results.

## 10.3 Conclusion

f1() could return pointers to any place in memory, located anywhere. This is in essence the usefulness of the pointers. By the way, C++ *references* work exactly the same way. Read more about them: ([51.3 on page 539](#)).

# Chapter 11

## GOTO operator

The GOTO operator is generally considered as anti-pattern. [Dij68], Nevertheless, it can be used reasonably [Knu74], [Yur13, p. 1.3.2].

Here is a very simple example:

```
#include <stdio.h>

int main()
{
    printf ("begin\n");
    goto exit;
    printf ("skip me!\n");
exit:
    printf ("end\n");
};
```

Here is what we have got in MSVC 2012:

Listing 11.1: 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
```

The *goto* statement has been simply replaced by a JMP instruction, which has the same effect: unconditional jump to another place.

The second `printf()` could be executed only with human intervention, by using a debugger or by patching the code.

This could also be useful as a simple patching exercise. Let's open the resulting executable in Hiew:

```

Hiew: goto.exe
C:\Polygon\goto.exe  FRO ----- a32 PE .00401000
.00401000: 55          push     ebp
.00401001: 8BEC       mov     ebp,esp
.00401003: 6800304000 push     000403000 ;'begin' --01
.00401008: FF1590204000 call    printf
.0040100E: 83C404     add     esp,4
.00401011: EB0E       jmps     .000401021 --02
.00401013: 6808304000 push     000403008 ;'skip me!' --03
.00401018: FF1590204000 call    printf
.0040101E: 83C404     add     esp,4
.00401021: 6814304000 2push    000403014 --04
.00401026: FF1590204000 call    printf
.0040102C: 83C404     add     esp,4
.0040102F: 33C0       xor     eax,eax
.00401031: 5D         pop     ebp
.00401032: C3         retn    ; ^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-

```

Figure 11.1: Hiew

Place the cursor to address `JMP (0x410)`, press `F3` (edit), press zero twice, so the opcode becomes `EB 00`:

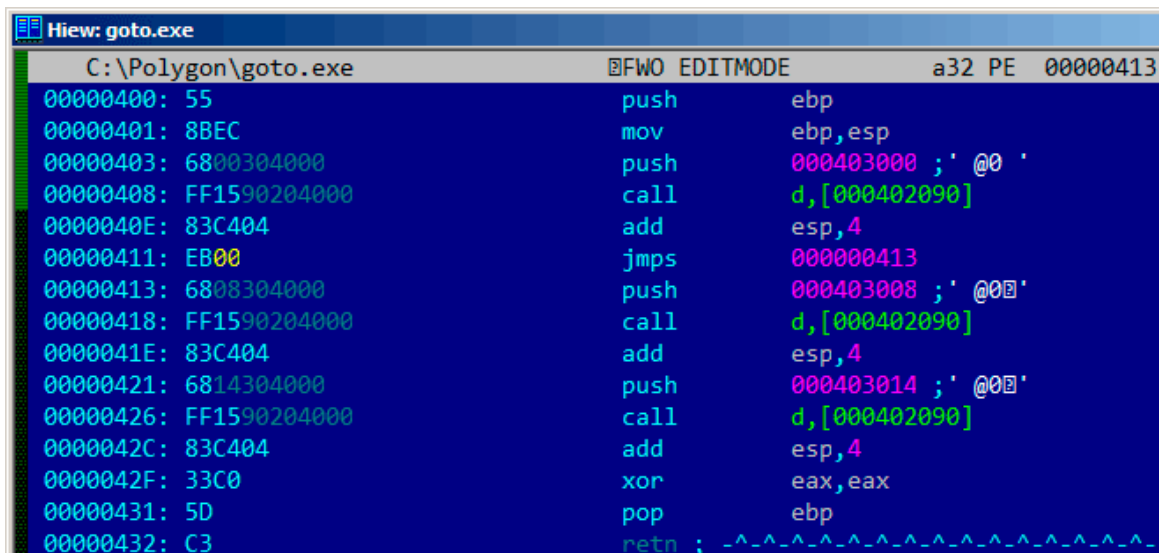


Figure 11.2: Hiew

The second byte of the `JMP` opcode denotes the relative offset for the jump, 0 means the point right after the current instruction. So now `JMP` not skipping the second `printf()` call.

Press F9 (save) and exit. Now if we run the executable we should see this:

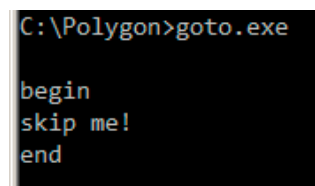


Figure 11.3: Patched executable output

The same result could be achieved by replacing the `JMP` instruction with 2 `NOP` instructions. `NOP` has an opcode of `0x90` and length of 1 byte, so we need 2 instructions as `JMP` replacement (which is 2 bytes in size).

## 11.1 Dead code

The second `printf()` call is also called “dead code” in compiler terms. This means that the code will never be executed. So when you compile this example with optimizations, the compiler removes “dead code”, leaving no trace of it:

Listing 11.2: Optimizing 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

```

However, the compiler forgot to remove the “skip me!” string.

## 11.2 Exercise

Try to achieve the same result using your favorite compiler and debugger.



## Chapter 12

# Conditional jumps

### 12.1 Simple example

```
#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;
};
```

#### 12.1.1 x86

##### x86 + MSVC

Here is how the `f_signed()` function looks like:

Listing 12.1: Non-optimizing 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
    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

```

The first instruction, JLE, stands for *Jump if Less or Equal*. In other words, if the second operand is larger or equal to the first one, the control flow will be passed to the specified in the instruction address or label. If this condition does not trigger because the second operand is smaller than the first one, the control flow would not be altered and the first `printf()` would be executed. The second check is JNE: *Jump if Not Equal*. The control flow will not change if the operands are equal.

The third check is JGE: *Jump if Greater or Equal*—jump if the first operand is larger than the second or if they are equal. So, if all three conditional jumps are triggered, none of the `printf()` calls would be executed whatsoever. This is impossible without special intervention.

Now let's take a look at the `f_unsigned()` function. The `f_unsigned()` function is the same as `f_signed()`, with the exception that the JBE and JAE instructions are used instead of JLE and JGE, as follows:

Listing 12.2: 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

```

As already mentioned, the branch instructions are different: JBE—*Jump if Below or Equal* and JAE—*Jump if Above or Equal*. These instructions (JA/JAE/JB/JBE) differ from JG/JGE/JL/JLE in the fact that they work with unsigned numbers.

See also the section about signed number representations ([30 on page 432](#)). That is why if we see JG/JL in use instead of JA/JB or vice-versa, we can be almost sure that the variables are signed or unsigned, respectively.

Here is also the `main()` function, where there is nothing much new to us:

Listing 12.3: 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
```

## x86 + MSVC + OllyDbg

We can see how flags are set by running this example in OllyDbg. Let's begin with `f_unsigned()`, which works with unsigned numbers. `CMP` is executed thrice here, but for the same arguments, so the flags are the same each time.

Result of the first comparison:

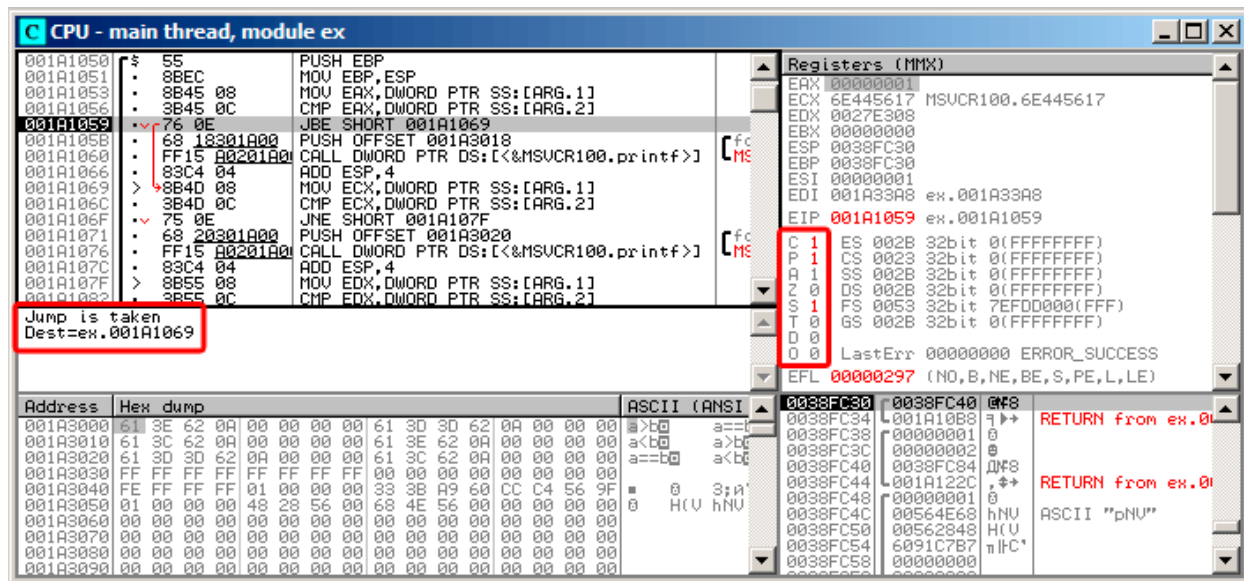


Figure 12.1: OllyDbg: `f_unsigned()`: first conditional jump

So, the flags are: `C=1`, `P=1`, `A=1`, `Z=0`, `S=1`, `T=0`, `D=0`, `O=0`. They are named with one character for brevity in OllyDbg.

OllyDbg gives a hint that the (`JBE`) jump is to be triggered now. Indeed, if we take a look into [\[Int13\]](#), we can read there that `JBE` is triggering if `CF=1` or `ZF=1`. The condition is true here, so the jump is triggered.

The next conditional jump:

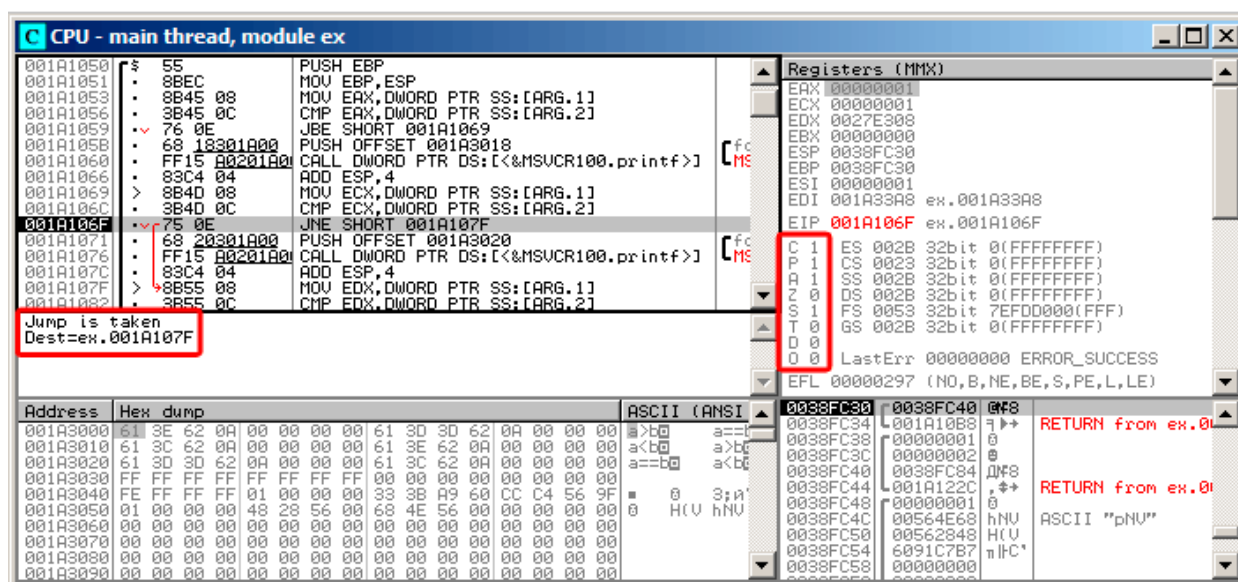


Figure 12.2: OllyDbg: `f_unsigned()`: second conditional jump

OllyDbg gives a hint that `JNZ` is to be triggered now. Indeed, `JNZ` triggering if `ZF=0` (zero flag).

The third conditional jump, JNB:

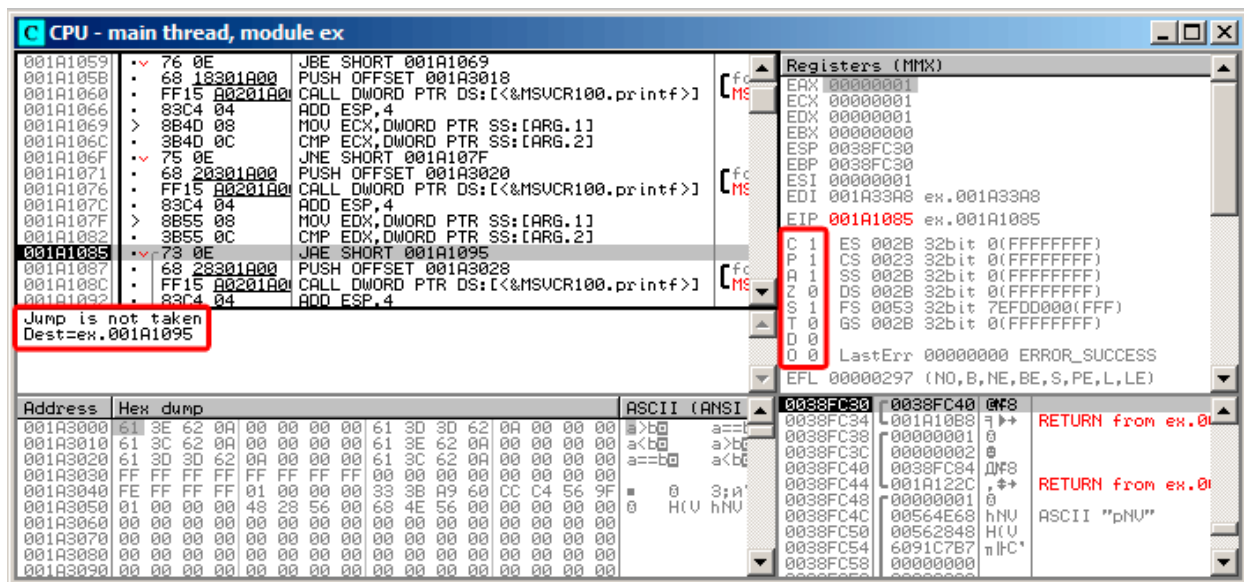


Figure 12.3: OllyDbg: f\_unsigned(): third conditional jump

In [Int13] we can see that JNB triggers if CF=0 (carry flag). That is not true in our case, so the third `printf()` will execute.

Now let's review the `f_signed()` function, which works with signed values, in OllyDbg.

Flags are set in the same way: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

The first conditional jump JLE is to be triggered:

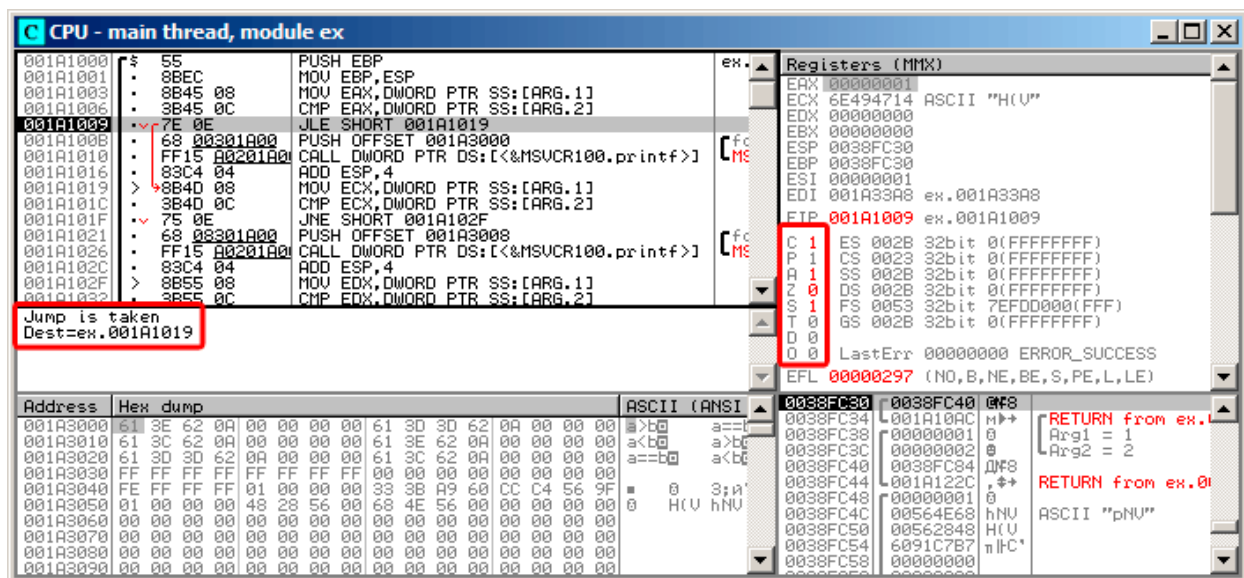


Figure 12.4: OllyDbg: `f_signed()`: first conditional jump

In [Int13] we find that this instruction is triggered if `ZF=1` or `SF≠OF`. `SF≠OF` in our case, so the jump triggers.

The second JNZ conditional jump triggering: if ZF=0 (zero flag):

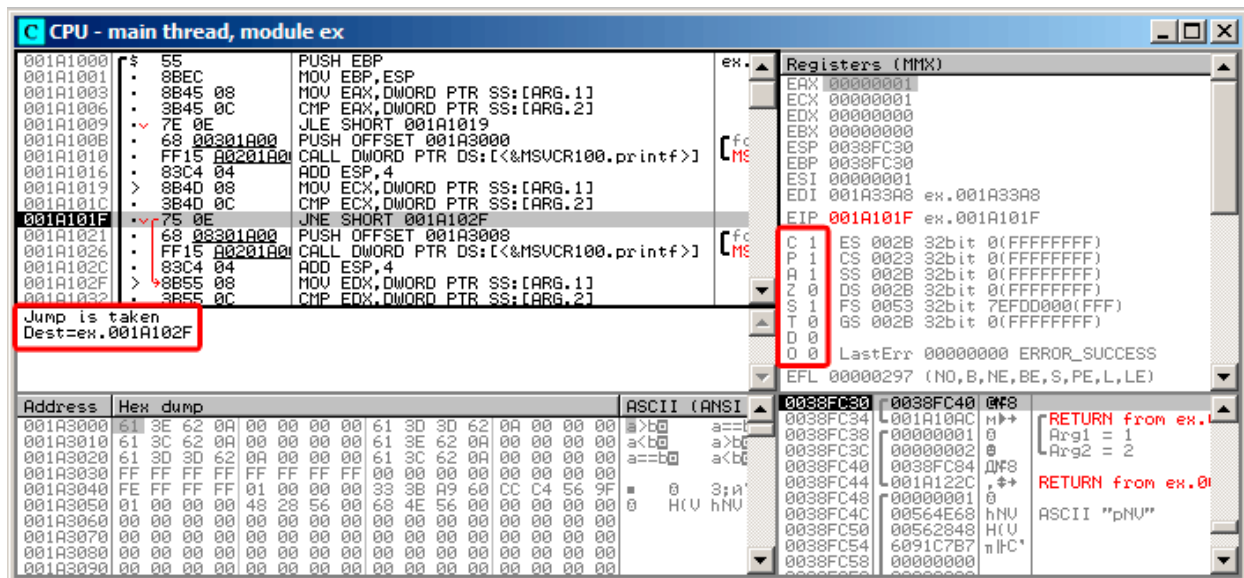


Figure 12.5: OllyDbg: `f_signed()`: second conditional jump



The third conditional jump JGE will not trigger because it would only do so if SF=OF, and that is not true in our case:

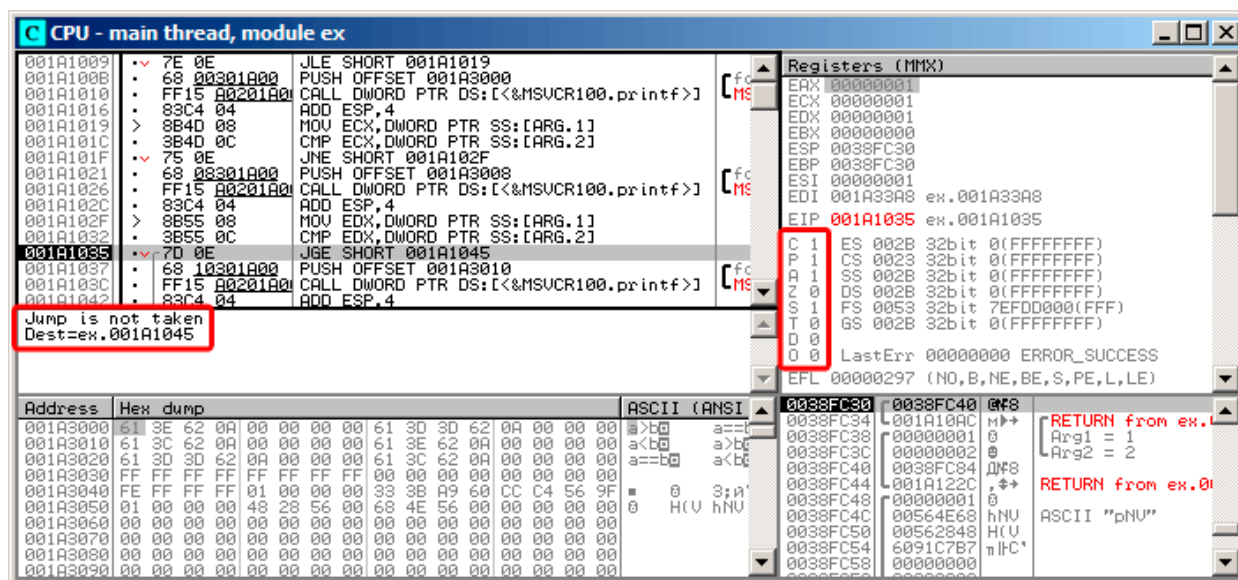


Figure 12.6: OllyDbg: f\_signed(): third conditional jump

## x86 + MSVC + Hiew

We can try to patch the executable file in a way that the `f_unsigned()` function would always print “a==b”, no matter the input values. Here is how it looks in Hiew:

```

C:\Polygon\ollydbg\7_1.exe  FRO -----  a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 55          push     ebp
.00401001: 8BEC       mov      ebp,esp
.00401003: 8B4508     mov      eax,[ebp+8]
.00401006: 3B450C     cmp      eax,[ebp+0C]
.00401009: 7E0D      jle      .000401018 --E1
.0040100B: 680B04000 push     00040B000 --E2
.00401010: E8AA00000 call     .0004010BF --E3
.00401015: 83C404     add      esp,4
.00401018: 8B4D08     1mov     ecx,[ebp+8]
.0040101B: 3B4D0C     cmp      ecx,[ebp+0C]
.0040101E: 750D      jnz      .00040102D --E4
.00401020: 680B04000 push     00040B008 ; 'a==b' --E5
.00401025: E89500000 call     .0004010BF --E3
.0040102A: 83C404     add      esp,4
.0040102D: 8B5508     4mov     edx,[ebp+8]
.00401030: 3B550C     cmp      edx,[ebp+0C]
.00401033: 7D0D      jge      .000401042 --E6
.00401035: 6810B04000 push     00040B010 --E7
.0040103A: E88000000 call     .0004010BF --E3
.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 2FillBlk 3CryBlk 4Reload 5OrdLdr 6String 7Direct 8Table 9Ibyte 10Leave 11Naked 12AddNam

```

Figure 12.7: Hiew: `f_unsigned()` function

Essentially, we need to accomplish three tasks:

- force the first jump to always trigger;
- force the second jump to never trigger;
- force the third jump to always trigger.

Thus we can direct the code flow to always pass through the second `printf()`, and output “a==b”.

Three instructions (or bytes) has to be patched:

- The first jump becomes `JMP`, but the [jump offset](#) would remain the same.
- The second jump might be triggered sometimes, but in any case it will jump to the next instruction, because, we set the [jump offset](#) to 0. In these instructions the [jump offset](#) is added to the address for the next instruction. So if the offset is 0, the jump will transfer the control to the next instruction.
- The third jump we replace with `JMP` just as we do with the first one, so it will always trigger.

Here is the modified code:

```

C:\Polygon\ollydbg\7_1.exe  FPU EDITMODE  a32 PE  00000434  Hiew 8.02 (c)SEN
00000400: 55      push     ebp
00000401: 8BEC    mov      ebp, esp
00000403: 8B4508  mov      eax, [ebp+8]
00000406: 3B450C  cmp      eax, [ebp+0C]
00000409: EB0D    jmps     00000418
0000040B: 680B040000 push    00040B000 ; '@'
00000410: E8AA000000 call     000004BF
00000415: 83C404  add      esp, 4
00000418: 8B4D08  mov      ecx, [ebp+8]
0000041B: 3B4D0C  cmp      ecx, [ebp+0C]
0000041E: 7500    jnz      00000420
00000420: 680B040000 push    00040B000 ; '@'
00000425: E895000000 call     000004BF
0000042A: 83C404  add      esp, 4
0000042D: 8B5508  mov      edx, [ebp+8]
00000430: 3B550C  cmp      edx, [ebp+0C]
00000433: EB0D    jmps     00000442
00000435: 6810B04000 push    00040B010 ; '@'
0000043A: E880000000 call     000004BF
0000043F: 83C404  add      esp, 4
00000442: 5D      pop      ebp
00000443: C3      retn ; ^.^.^.^.^.^.^.^.^.^.^.^.^.^.^.^
00000444: CC      int      3
00000445: CC      int      3
00000446: CC      int      3
00000447: CC      int      3
00000448: CC      int      3

```

Figure 12.8: Hiew: let's modify the `f_unsigned()` function

If we miss to change any of these jumps, then several `printf()` calls may execute, while we want to execute only one.

### Non-optimizing GCC

Non-optimizing GCC 4.4.1 produces almost the same code, but with `puts()` (3.4.3 on page 15) instead of `printf()`.

### Optimizing GCC

An observant reader may ask, why execute CMP several times, if the flags has the same values after each execution? Perhaps optimizing MSVC can not do this, but optimizing GCC 4.8.1 can go deeper:

Listing 12.4: 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
.L7:
    mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"

```

jmp	puts
-----	------

We also see `JMP puts` here instead of `CALL puts / RETN`. This kind of trick will have explained later: [13.1.1 on page 144](#).

This type of x86 code is somewhat rare. MSVC 2012 as it seems, can't generate such code. On the other hand, assembly language programmers are fully aware of the fact that `Jcc` instructions can be stacked. So if you see such stacking somewhere, it is highly probable that the code was hand-written.

The `f_unsigned()` function is not that aesthetically short:

Listing 12.5: 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      ; this instruction could be removed
    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
```

Nevertheless, there are two `CMP` instructions instead of three. So optimization algorithms of GCC 4.8.1 are probably not perfect yet.

## 12.1.2 ARM

### 32-bit ARM

#### Optimizing Keil 6/2013 (ARM mode)

Listing 12.6: Optimizing Keil 6/2013 (ARM mode)

```
.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"
.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
```

Many instructions in ARM mode could be executed only when specific flags are set. E.g. this is often used when comparing numbers.

For instance, the ADD instruction is in fact named ADDAL internally, where AL stands for *Always*, i.e., execute always. The predicates are encoded in 4 high bits of the 32-bit ARM instructions (*condition field*). The B instruction for unconditional jumping is in fact conditional and encoded just like any other conditional jump, but has AL in the *condition field*, and it implies *execute Always*, ignoring flags.

The ADRGT instruction works just like ADR but executes only in case the previous CMP instruction finds one of the numbers greater than the another, while comparing the two (*Greater Than*).

The next BLGT instruction behaves exactly as BL and is triggered only if the result of the comparison was the same (*Greater Than*). ADRGT writes a pointer to the string a>b\n into R0 and BLGT calls printf(). therefore, these instructions with suffix -GT are to execute only in case the value in R0 (which is *a*) is bigger than the value in R4 (which is *b*).

Moving forward we see the ADREQ and BLEQ instructions. They behave just like ADR and BL, but are to be executed only if operands were equal to each other during the last comparison. Another CMP is located before them (because the printf() execution may have tampered the flags).

Then we see LDMGEFD, this instruction works just like LDMFD<sup>1</sup>, but is triggered only when one of the values is greater or equal than the other (*Greater or Equal*).

The LDMGEFD SP!, {R4-R6,PC} instruction acts like a function epilogue, but it will be triggered only if  $a \geq b$ , and only then the function execution will finish.

But if that condition is not satisfied, i.e.,  $a < b$ , then the control flow will continue to the next "LDMFD SP!, {R4-R6,LR}" instruction, which is one more function epilogue. This instruction restores not only the R4-R6 registers state, but also LR instead of PC, thus, it does not returns from the function.

The last two instructions call printf() with the string «a<b\n» as a sole argument. We already examined an unconditional jump to the printf() function instead of function return in «printf() with several arguments» section (6.2.1 on page 46).

f\_unsigned is similar, only the ADRHI, BLHI, and LDMCSFD instructions are used there, these predicates (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) are analogous to those examined before, but for unsigned values.

There is not much new in the main() function for us:

Listing 12.7: 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
```

That is how you can get rid of conditional jumps in ARM mode.

Why is this so good? Read here: [33.1 on page 437](#).

There is no such feature in x86, except the CMOVcc instruction, it is the same as MOV, but triggered only when specific flags are set, usually set by CMP.

## Optimizing Keil 6/2013 (Thumb mode)

Listing 12.8: Optimizing Keil 6/2013 (Thumb mode)

```
.text:00000072          f_signed ; CODE XREF: main+6
.text:00000072 70 B5      PUSH     {R4-R6,LR}
```

<sup>1</sup>LDMFD

```
.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
```

Only B instructions in Thumb mode may be supplemented by *condition codes*, so the Thumb code looks more ordinary.

BLE is a normal conditional jump *Less than or Equal*, BNE—*Not Equal*, BGE—*Greater than or Equal*.

f\_unsigned is similar, only other instructions are used while dealing with unsigned values: BLS (*Unsigned lower or same*) and BCS (*Carry Set (Greater than or equal)*).

#### ARM64: Optimizing GCC (Linaro) 4.9

Listing 12.9: f\_signed()

```
f_signed:
; W0=a, W1=b
    cmp     w0, w1
    bgt     .L19      ; Branch if Greater Than (a>b)
    beq     .L20      ; Branch if Equal (a==b)
    bge     .L15      ; Branch if Greater than or Equal (a>=b) (impossible here)
; 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:
; impossible here
    ret
.L20:
    adrp    x0, .LC10      ; "a==b"
    add     x0, x0, :lo12:LC10
    b       puts
```

Listing 12.10: f\_unsigned()

```
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 (a>b)
    cmp     w19, w1
    beq     .L26      ; Branch if Equal (a==b)
.L23:
    bcc     .L27      ; Branch if Carry Clear (if less than) (a<b)
; function epilogue, impossible to be here
```

```

        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

.L26:
        ldr    x19, [sp,16]
        adrp   x0, .LC10      ; "a==b"
        ldp    x29, x30, [sp], 48
        add    x0, x0, :lo12:.LC10
        b      puts

```

The comments were added by the author of this book. What is striking is that the compiler is not aware that some conditions are not possible at all, so there is dead code at some places, which can never be executed.

### Exercise

Try to optimize these functions manually for size, removing redundant instructions, without adding new ones.

## 12.1.3 MIPS

One distinctive MIPS feature is the absence of flags. Apparently, it was done to simplify the analysis of data dependencies. There are instructions similar to SETcc in x86: SLT (“Set on Less Than”: signed version) and SLTU (unsigned version). These instructions sets destination register value to 1 if the condition is true or to 0 if otherwise.

The destination register is then checked using BEQ (“Branch on Equal”) or BNE (“Branch on Not Equal”) and a jump may occur. So, this instruction pair has to be used in MIPS for comparison and branch.

Let’s first start with the signed version of our function:

Listing 12.11: Non-optimizing 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
.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)
; store input values into local stack:
.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

```

```

; this is pseudoinstruction. in fact, "slt $v0,$v0,$v1" is there.
; so $v0 will be set to 1 if $v0<$v1 (b<a) or to 0 if otherwise:
.text:00000030          slt     $v0, $v1
; jump to loc_5c, if condition is not true.
; this is pseudoinstruction. in fact, "beq $v0,$zero,loc_5c" is there:
.text:00000034          beqz    $v0, loc_5C
; print "a>b" and finish
.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
; check if a==b, jump to loc_90 if its not true':
.text:00000068          bne     $v1, $v0, loc_90
.text:0000006C          or      $at, $zero ; branch delay slot, NOP
; condition is true, so print "a==b" and finish:
.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
; check if $v1<$v0 (a<b), set $v0 to 1 if condition is true:
.text:0000009C          slt     $v0, $v1, $v0
; if condition is not true (i.e., $v0==0), jump to loc_c8:
.text:000000A0          beqz    $v0, loc_C8
.text:000000A4          or      $at, $zero ; branch delay slot, NOP
; condition is true, print "a<b" and finish
.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
; all 3 conditions were false, so just finish:
.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” is reduced by IDA to its shorter form “SLT REG0, REG1”. We also see there BEQZ pseudoinstruction (“Branch if Equal to Zero”), which are in fact “BEQ REG, \$ZERO, LABEL”.

The unsigned version is just the same, but SLTU (unsigned version, hence “U” in name) is used instead of SLT:

Listing 12.12: Non-optimizing 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          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          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
.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

```

## 12.2 Calculating absolute value

A simple function:

```
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
};
```

### 12.2.1 Optimizing MSVC

This is how the code is usually generated:

Listing 12.13: Optimizing MSVC 2012 x64

```
i$ = 8
my_abs PROC
; ECX = input
    test    ecx, ecx
; check for sign of input value
; skip NEG instruction if sign is positive
    jns     SHORT $LN2@my_abs
; negate value
    neg     ecx
$LN2@my_abs:
; prepare result in EAX:
    mov     eax, ecx
    ret     0
my_abs ENDP
```

GCC 4.9 does mostly the same.

### 12.2.2 Optimizing Keil 6/2013: Thumb mode

Listing 12.14: Optimizing Keil 6/2013: Thumb mode

```
my_abs PROC
    CMP     r0,#0
; is input value equal to zero or greater than zero?
; skip RSBS instruction then
    BGE     |L0.6|
; subtract input value from 0:
    RSBS    r0,r0,#0
|L0.6|
    BX      lr
ENDP
```

ARM lacks a negate instruction, so the Keil compiler uses the “Reverse Subtract” instruction, which just subtracts with reversed operands.

### 12.2.3 Optimizing Keil 6/2013: ARM mode

It is possible to add condition codes to some instructions in ARM mode, so that is what the Keil compiler does:

Listing 12.15: Optimizing Keil 6/2013: ARM mode

```
my_abs PROC
    CMP     r0,#0
; execute "Reverse Subtract" instruction only if input value is less than 0:
    RSBLT   r0,r0,#0
    BX      lr
ENDP
```

Now there are no conditional jumps and this is good: [33.1 on page 437](#).

## 12.2.4 Non-optimizing GCC 4.9 (ARM64)

ARM64 has instruction NEG for negating:

Listing 12.16: Optimizing GCC 4.9 (ARM64)

```
my_abs:
    sub    sp, sp, #16
    str    w0, [sp,12]
    ldr    w0, [sp,12]
; compare input value with contents of WZR register
; (which always holds zero)
    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
```

## 12.2.5 MIPS

Listing 12.17: Optimizing GCC 4.4.5 (IDA)

```
my_abs:
; jump if $a0<0:
        bltz    $a0, locret_10
; just return input value ($a0) in $v0:
        move    $v0, $a0
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP
locret_10:
; negate input value and store it in $v0:
        jr      $ra
; this is pseudoinstruction. in fact, this is "subu $v0,$zero,$a0" ($v0=0-$a0)
        negu    $v0, $a0
```

Here we see a new instruction: BLTZ (“Branch if Less Than Zero”). There is also the NEGU pseudoinstruction, which just does subtraction from zero. The “U” suffix in both SUBU and NEGU implies that no exception to be raised in case of integer overflow.

## 12.2.6 Branchless version?

You could have also a branchless version of this code. This we will review later: [45 on page 494](#).

## 12.3 Ternary conditional operator

The ternary conditional operator in C/C++ has the following syntax:

```
expression ? expression : expression
```

Here is an example:

```
const char* f (int a)
{
    return a==10 ? "it is ten" : "it is not ten";
};
```

**12.3.1 x86**

Old and non-optimizing compilers generate assembly code just as if an `if/else` statement was used:

Listing 12.18: Non-optimizing MSVC 2008

```
$SG746 DB      'it is ten', 00H
$SG747 DB      'it is not ten', 00H

tv65 = -4 ; this will be used as a temporary variable
_a$ = 8
_f      PROC
    push     ebp
    mov      ebp, esp
    push     ecx
; compare input value with 10
    cmp      DWORD PTR _a$[ebp], 10
; jump to $LN3@f if not equal
    jne      SHORT $LN3@f
; store pointer to the string into temporary variable:
    mov      DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; jump to exit
    jmp      SHORT $LN4@f
$LN3@f:
; store pointer to the string into temporary variable:
    mov      DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; this is exit. copy pointer to the string from temporary variable to EAX.
    mov      eax, DWORD PTR tv65[ebp]
    mov      esp, ebp
    pop      ebp
    ret      0
_f      ENDP
```

Listing 12.19: Optimizing MSVC 2008

```
$SG792 DB      'it is ten', 00H
$SG793 DB      'it is not ten', 00H

_a$ = 8 ; size = 4
_f      PROC
; compare input value with 10
    cmp      DWORD PTR _a$[esp-4], 10
    mov      eax, OFFSET $SG792 ; 'it is ten'
; jump to $LN4@f if equal
    je       SHORT $LN4@f
    mov      eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
    ret      0
_f      ENDP
```

Newer compilers are more concise:

Listing 12.20: Optimizing MSVC 2012 x64

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
f      PROC
; load pointers to the both strings
    lea      rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
    lea      rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; compare input value with 10
    cmp      ecx, 10
; if equal, copy value from RDX ("it is ten")
; if not, do nothing. pointer to the string "it is not ten" is still in RAX as for now.
    cmov     rax, rdx
    ret      0
f      ENDP
```

Optimizing GCC 4.8 for x86 also uses the `CMOVcc` instruction, while the non-optimizing GCC 4.8 uses conditional jumps.

### 12.3.2 ARM

Optimizing Keil for ARM mode also uses the conditional instructions `ADRCc`:

Listing 12.21: Optimizing Keil 6/2013 (ARM mode)

```
f PROC
; compare input value with 10
    CMP    r0,#0xa
; if comparison result is Equal, copy pointer to the "it is ten" string into R0
    ADREQ  r0,|L0.16| ; "it is ten"
; if comparison result is Not Equal, copy pointer to the "it is not ten" string into 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
```

Without manual intervention, the two instructions `ADREQ` and `ADRNE` cannot be executed in the same run.

Optimizing Keil for Thumb mode needs to use conditional jump instructions, since there are no load instructions that support conditional flags:

Listing 12.22: Optimizing Keil 6/2013 (Thumb mode)

```
f PROC
; compare input value with 10
    CMP    r0,#0xa
; jump to |L0.8| if Equal
    BEQ    |L0.8|
    ADR     r0,|L0.12| ; "it is not ten"
    BX     lr
|L0.8|
    ADR     r0,|L0.28| ; "it is ten"
    BX     lr
ENDP

|L0.12|
    DCB    "it is not ten",0
|L0.28|
    DCB    "it is ten",0
```

### 12.3.3 ARM64

Optimizing GCC (Linaro) 4.9 for ARM64 also uses conditional jumps:

Listing 12.23: Optimizing GCC (Linaro) 4.9

```
f:
    cmp     x0, 10
    beq     .L3          ; branch if equal
    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"
```

That is because ARM64 does not have a simple load instruction with conditional flags, like `ADRCc` in 32-bit ARM mode or `CMOVcc` in x86 [ARM13a, p390, C5.5]. It has, however, “Conditional SElect” instruction (`CSEL`), but GCC 4.9 does not seem to be smart enough to use it in such piece of code.

### 12.3.4 MIPS

Unfortunately, GCC 4.4.5 for MIPS is not very smart, either:

Listing 12.24: Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
    .ascii "it is not ten\000"
$LC1:
    .ascii "it is ten\000"
f:
    li      $2,10                # 0xa
; compare $a0 and 10, jump if equal:
    beq     $4,$2,$L2
    nop ; branch delay slot

; leave address of "it is not ten" string in $v0 and return:
    lui     $2,%hi($LC0)
    j       $31
    addiu   $2,$2,%lo($LC0)

$L2:
; leave address of "it is ten" string in $v0 and return:
    lui     $2,%hi($LC1)
    j       $31
    addiu   $2,$2,%lo($LC1)
```

### 12.3.5 Let's rewrite it in an if/else way

```
const char* f (int a)
{
    if (a==10)
        return "it is ten";
    else
        return "it is not ten";
};
```

Interestingly, optimizing GCC 4.8 for x86 was also able to use CMOVcc in this case:

Listing 12.25: Optimizing GCC 4.8

```
.LC0:
    .string "it is ten"
.LC1:
    .string "it is not ten"
f:
.LFB0:
; compare input value with 10
    cmp     DWORD PTR [esp+4], 10
    mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
    mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; if comparison result is Not Equal, copy EDX value to EAX
; if not, do nothing
    cmovne  eax, edx
    ret
```

Optimizing Keil in ARM mode generates code identical to listing [12.21](#).

But the optimizing MSVC 2012 is not that good (yet).

### 12.3.6 Conclusion

Why optimizing compilers try to get rid of conditional jumps? Read here about it: [33.1 on page 437](#).

## 12.4 Getting minimal and maximal values

### 12.4.1 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;
};
```

Listing 12.26: Non-optimizing MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
; compare A and B:
    cmp     eax, DWORD PTR _b$[ebp]
; jump, if A is greater or equal to B:
    jge     SHORT $LN2@my_min
; reload A to EAX if otherwise and jump to exit
    mov     eax, DWORD PTR _a$[ebp]
    jmp     SHORT $LN3@my_min
    jmp     SHORT $LN3@my_min ; this is redundant JMP
$LN2@my_min:
; return B
    mov     eax, DWORD PTR _b$[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$[ebp]
; compare A and B:
    cmp     eax, DWORD PTR _b$[ebp]
; jump if A is less or equal to B:
    jle     SHORT $LN2@my_max
; reload A to EAX if otherwise and jump to exit
    mov     eax, DWORD PTR _a$[ebp]
    jmp     SHORT $LN3@my_max
    jmp     SHORT $LN3@my_max ; this is redundant JMP
$LN2@my_max:
; return B
    mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
    pop     ebp
    ret     0
_my_max ENDP
```

These two functions differ only in the conditional jump instruction: JGE (“Jump if Greater or Equal”) is used in the first one and JLE (“Jump if Less or Equal”) in the second.

There is one unneeded JMP instruction in each function, which MSVC probably left by mistake.

## Branchless

ARM for Thumb mode reminds us of x86 code:

Listing 12.27: Optimizing Keil 6/2013 (Thumb mode)

```
my_max PROC
; R0=A
; R1=B
; compare A and B:
    CMP    r0,r1
; branch if A is greater then B:
    BGT    |L0.6|
; otherwise (A<=B) return R1 (B):
    MOVS   r0,r1
|L0.6|
; return
    BX     lr
ENDP

my_min PROC
; R0=A
; R1=B
; compare A and B:
    CMP    r0,r1
; branch if A is less then B:
    BLT    |L0.14|
; otherwise (A>=B) return R1 (B):
    MOVS   r0,r1
|L0.14|
; return
    BX     lr
ENDP
```

The functions differ in the branching instruction: BGT and BLT.

It's possible to use conditional suffixes in ARM mode, so the code is shorter. MOVcc is to be executed only if the condition is met:

Listing 12.28: Optimizing Keil 6/2013 (ARM mode)

```
my_max PROC
; R0=A
; R1=B
; compare A and B:
    CMP    r0,r1
; return B instead of A by placing B in R0
; this instruction will trigger only if A<=B (hence, LE - Less or Equal)
; if instruction is not triggered (in case of A>B), A is still in R0 register
    MOVLE  r0,r1
    BX     lr
ENDP

my_min PROC
; R0=A
; R1=B
; compare A and B:
    CMP    r0,r1
; return B instead of A by placing B in R0
; this instruction will trigger only if A>=B (hence, GE - Greater or Equal)
; if instruction is not triggered (in case of A<B), A value is still in R0 register
    MOVGE  r0,r1
    BX     lr
ENDP
```

Optimizing GCC 4.8.1 and optimizing MSVC 2013 can use CMOVcc instruction, which is analogous to MOVcc in ARM:

Listing 12.29: Optimizing MSVC 2013



```

my_max:
    mov     edx, DWORD PTR [esp+4]
    mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; compare A and B:
    cmp     edx, eax
; if A>=B, load A value into EAX
; the instruction idle if otherwise (if A<B)
    cmovge  eax, edx
    ret

my_min:
    mov     edx, DWORD PTR [esp+4]
    mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; compare A and B:
    cmp     edx, eax
; if A<=B, load A value into EAX
; the instruction idle if otherwise (if A>B)
    cmovle  eax, edx
    ret

```

## 12.4.2 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;
};

```

There is some unneeded value shuffling, but the code is comprehensible:

Listing 12.30: Non-optimizing 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

```

## Branchless

No need to load function arguments from the stack, as they are already in the registers:

Listing 12.31: Optimizing GCC 4.9.1 x64

```

my_max:
; RDI=A
; RSI=B
; compare A and B:
    cmp    rdi, rsi
; prepare B in RAX for return:
    mov    rax, rsi
; if A>=B, put A (RDI) in RAX for return.
; this instruction is idle if otherwise (if A<B)
    cmovge rax, rdi
    ret

my_min:
; RDI=A
; RSI=B
; compare A and B:
    cmp    rdi, rsi
; prepare B in RAX for return:
    mov    rax, rsi
; if A<=B, put A (RDI) in RAX for return.
; this instruction is idle if otherwise (if A>B)
    cmovle rax, rdi
    ret

```

MSVC 2013 does almost the same.

ARM64 has the CSEL instruction, which works just as MOVcc in ARM or CMOVcc in x86, just the name is different: “Conditional SElect”.

Listing 12.32: Optimizing GCC 4.9.1 ARM64

```

my_max:
; X0=A
; X1=B
; compare A and B:
    cmp    x0, x1
; select X0 (A) to X0 if X0>=X1 or A>=B (Greater or Equal)
; select X1 (B) to X0 if A<B
    csel   x0, x0, x1, ge
    ret

my_min:
; X0=A
; X1=B
; compare A and B:
    cmp    x0, x1
; select X0 (A) to X0 if X0<=X1 or A<=B (Less or Equal)
; select X1 (B) to X0 if A>B
    csel   x0, x0, x1, le
    ret

```

### 12.4.3 MIPS

Unfortunately, GCC 4.4.5 for MIPS is not that good:

Listing 12.33: Optimizing GCC 4.4.5 (IDA)

```
my_max:
; set $v1 $a1<$a0:
        slt     $v1, $a1, $a0
; jump, if $a1<$a0:
        beqz    $v1, locret_10
; this is branch delay slot
; prepare $a1 in $v0 in case of branch triggered:
        move    $v0, $a1
; no branch triggered, prepare $a0 in $v0:
        move    $v0, $a0

locret_10:
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP

; the min() function is same, but input operands in SLT instruction are swapped:
my_min:
        slt     $v1, $a0, $a1
        beqz    $v1, locret_28
        move    $v0, $a1
        move    $v0, $a0

locret_28:
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP
```

Do not forget about the *branch delay slots*: the first MOVE is executed *before* BEQZ, the second MOVE is executed only if the branch wasn't taken.

## 12.5 Conclusion

### 12.5.1 x86

Here's the rough skeleton of a conditional jump:

Listing 12.34: x86

```
CMP register, register/value
Jcc true ; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

### 12.5.2 ARM

Listing 12.35: ARM

```
CMP register, register/value
Bcc true ; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

### 12.5.3 MIPS

Listing 12.36: Check for zero

```
BEQZ REG, label
...
```

Listing 12.37: Check for less than zero:

```
BLTZ REG, label
...
```

Listing 12.38: Check for equal values

```
BEQ REG1, REG2, label
...
```

Listing 12.39: Check for non-equal values

```
BNE REG1, REG2, label
...
```

Listing 12.40: Check for less than, greater than (signed)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

Listing 12.41: Check for less than, greater than (unsigned)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

### 12.5.4 Branchless

If the body of a condition statement is very short, the conditional move instruction can be used: MOVcc in ARM (in ARM mode), CSEL in ARM64, CMOVcc in x86.

#### ARM

It's possible to use conditional suffixes in ARM mode for some instructions:

Listing 12.42: ARM (ARM mode)

```
CMP register, register/value
instr1_cc ; some instruction will be executed if condition code is true
instr2_cc ; some other instruction will be executed if other condition code is true
... etc...
```

Of course, there is no limit for the number of instructions with conditional code suffixes, as long as the CPU flags are not modified by any of them.

Thumb mode has the IT instruction, allowing to add conditional suffixes to the next four instructions. Read more about it: [17.7.2 on page 249](#).

Listing 12.43: ARM (Thumb mode)

```
CMP register, register/value
ITEEE EQ ; set these suffixes: if-then-else-else-else
instr1   ; instruction will be executed if condition is true
instr2   ; instruction will be executed if condition is false
instr3   ; instruction will be executed if condition is false
instr4   ; instruction will be executed if condition is false
```

## 12.6 Exercise

(ARM64) Try rewriting the code in listing [12.23](#) by removing all conditional jump instructions and using the CSEL instruction.

## Chapter 13

# switch()/case/default

### 13.1 Small number of cases

```
#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
};
```

#### 13.1.1 x86

##### Non-optimizing MSVC

Result (MSVC 2010):

Listing 13.1: 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:
    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

```

Our function with a few cases in `switch()` is in fact analogous to this construction:

```

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");
};

```

If we work with `switch()` with a few cases it is impossible to be sure if it was a real `switch()` in the source code, or just a pack of `if()` statements. This implies that `switch()` is like syntactic sugar for a large number of nested `if()`s.

There is nothing especially new to us in the generated code, with the exception of the compiler moving input variable `a` to a temporary local variable `tv64`<sup>1</sup>.

If we compile this in GCC 4.4.1, we'll get almost the same result, even with maximal optimization turned on (`-O3` option).

### Optimizing MSVC

Now let's turn on optimization in MSVC (`/Ox`): `cl 1.c /Fa1.asm /Ox`

Listing 13.2: 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
$LN4@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
    jmp     _printf
_f    ENDP

```

<sup>1</sup>Local variables in stack are prefixed with `tv`—that's how MSVC names internal variables for its needs

Here we can see some dirty hacks.

First: the value of *a* is placed in EAX and 0 is subtracted from it. Sounds absurd, but it is done to check if the value in EAX was 0. If yes, the ZF flag is to be set (e.g. subtracting from 0 is 0) and the first conditional jump JE (*Jump if Equal* or synonym JZ –*Jump if Zero*) is to be triggered and control flow is to be passed to the \$LN4@f label, where the 'zero' message is being printed. If the first jump doesn't get triggered, 1 is subtracted from the input value and if at some stage the result is 0, the corresponding jump is to be triggered.

And if no jump gets triggered at all, the control flow passes to printf() with string argument 'something unknown'.

Second: we see something unusual for us: a string pointer is placed into the *a* variable, and then printf() is called not via CALL, but via JMP. There is a simple explanation for that: the caller pushes a value to the stack and calls our function via CALL. CALL itself pushes the return address (RA) to the stack and does an unconditional jump to our function address. Our function at any point of execution (since it do not contain any instruction that moves the stack pointer) has the following stack layout:

- ESP—points to RA
- ESP+4—points to the *a* variable

On the other side, when we need to call printf() here we need exactly the same stack layout, except for the first printf() argument, which needs to point to the string. And that is what our code does.

It replaces the function's first argument with the address of the string and jumps to printf(), as if we didn't call our function f(), but directly printf(). printf() prints a string to stdout and then executes the RET instruction, which POPs RA from the stack and control flow is returned not to f() but rather to f()'s callee, bypassing the end of the f() function.

All this is possible because printf() is called right at the end of the f() function in all cases. In some way, it is similar to the longjmp()<sup>2</sup> function. And of course, it is all done for the sake of speed.

A similar case with the ARM compiler is described in “printf() with several arguments” section, here ( [6.2.1 on page 46](#) ).

---

<sup>2</sup>[wikipedia](#)

## OllyDbg

Since this example is tricky, let's trace it in OllyDbg.

OllyDbg can detect such switch() constructs, and it can add some useful comments. EAX is 2 in the beginning, that's the function's input value:

**CPU - main thread, module few**

Address	Hex dump	Assembly	Comment
00FF1000	8B4424 04	MOV EAX,DWORD PTR SS:[ARG.1]	
00FF1004	83E8 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	C74424 04 18	MOV DWORD PTR SS:[ARG.1],OFFSET 00FF3018	ASCII "something unknown"
00FF1017	FF25 0020FF00	JMP DWORD PTR DS:[&MSUCR100.printf]	
00FF101D	C74424 04 10	MOV DWORD PTR SS:[ARG.1],OFFSET 00FF3010	ASCII "two", case 2 of
00FF1025	FF25 0020FF00	JMP DWORD PTR DS:[&MSUCR100.printf]	
00FF102B	C74424 04 08	MOV DWORD PTR SS:[ARG.1],OFFSET 00FF3008	ASCII "one", case 1 of
00FF1033	FF25 0020FF00	JMP DWORD PTR DS:[&MSUCR100.printf]	
00FF1039	C74424 04 00	MOV DWORD PTR SS:[ARG.1],OFFSET 00FF3000	ASCII "zero", case 0 of
00FF1047	FF25 0020FF00	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=0  
EAX=2

**Registers (MMX)**

Register	Value	Comment
EAX	00000002	
ECX	6E494714	ASCII "H( \""
EDX	00000000	
EBX	00000000	
ESP	001EF84C	
EBP	001EF894	
ESI	00000001	
EDI	00FF33A8	few.00FF33A8
EIP	00FF1004	few.00FF1004

**Hex dump**

Address	Hex dump	ASCII (ANSI - Cy)
00FF3000	74 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
00FF3020	67 20 75 6E 6B 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
00FF3030	FF FF FF FF 00 00 00 00 34 54 75 46 CB AB 8A B3	
00FF3040	FE FF FF FF 01 00 00 00 00 00 00 00 00 00 00 00	
00FF3050	01 00 00 00 48 23 2A 00 63 4E 2A 00 00 00 00 00	
00FF3060	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	
00FF3080	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	
00FF30A0	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	
00FF30C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

**Registers (MMX)**

Register	Value	Comment
EAX	00000002	
ECX	00FF11CA	
EDX	00000001	
EBX	002A4E68	hN# ASCII "pN"
ESP	002A2848	H(
EBP	466BACA0	ankF
ESI	00000000	
EDI	00000000	
EIP	001EF870	p#
ECX	001EF874	
EDX	001EF878	
EBX	001EF87C	d°
ESP	001EF880	D3389310
EIP	001EF884	Printer to ne

Figure 13.1: OllyDbg: EAX now contain the first (and only) function argument



0 is subtracted from 2 in EAX. Of course, EAX still contains 2. But the ZF flag is now 0, indicating that the resulting value is non-zero:

**CPU - main thread, module few**

Address	Hex dump	Assembly	Comment
00FF1000	8B4424 04	MOV EAX, DWORD PTR SS:[ARG.1]	
00FF1004	83E8 00	SUB EAX, 0	Switch (cases 0..2, 4 ex)
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	C74424 04 18	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF301	ASCII "something unknown"
00FF1011	FF25 0020FE0	JMP DWORD PTR DS:[<MSUCR100.printf>]	
00FF1017	C74424 04 10	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF301	ASCII "two", case 2 of
00FF101D	FF25 0020FE0	JMP DWORD PTR DS:[<MSUCR100.printf>]	
00FF1025	C74424 04 08	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF300	ASCII "one", case 1 of
00FF102B	FF25 0020FE0	JMP DWORD PTR DS:[<MSUCR100.printf>]	
00FF1033	C74424 04 00	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF300	ASCII "zero", case 0 of
00FF1039	FF25 0020FE0	JMP DWORD PTR DS:[<MSUCR100.printf>]	
00FF1041	CC	INT3	
00FF1047	CC	INT3	
00FF1048	CC	INT3	
00FF1049	CC	INT3	
00FF104A	CC	INT3	
00FF104B	CC	INT3	
00FF104C	CC	INT3	

Jump is not taken  
Dest=few.00FF1039

**Registers (MMX)**

Register	Value	Comment
EAX	00000002	
ECX	6E494714	ASCII "H(*)"
EDX	00000000	
EBX	00000000	
ESP	001EF84C	
EBP	001EF894	
ESI	00000001	
EDI	00FF33A8	few.00FF33A8
EIP	00FF1007	few.00FF1007
CS	002B	32bit 0(FFFFFFFF)
DS	002B	32bit 0(FFFFFFFF)
SS	002B	32bit 0(FFFFFFFF)
ES	002B	32bit 0(FFFFFFFF)
FS	0053	32bit 7EFD0000(FFF)
GS	002B	32bit 0(FFFFFFFF)
LastErr	00000000	ERROR_SUCCESS
EFL	00000202	(NO, NB, NE, A, NS, PO, GE, G)
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
00FF3000	74 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 63 6E	two something
00FF3020	67 20 75 6E 6B 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
00FF3030	FF FF FF FF 01 00 00 00 34 54 75 46 CB AB 3A B9	
00FF3040	FE FF FF FF 01 00 00 00 68 4E 2A 00 00 00 00 00	
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	
00FF30D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

**001EF84C 00FF1057**

Address	Hex dump	ASCII
001EF850	00000002	
001EF854	00FF11CA	
001EF858	00000001	
001EF85C	002A4E68	hN*
001EF860	002A2848	H(*)
001EF864	466BACA0	amkF
001EF868	00000000	
001EF86C	00000000	
001EF870	7EFD0000	p#
001EF874	00000000	
001EF878	00000000	
001EF87C	001EF864	d°
001EF880	D389310	Y8L
001EF884	001EF800	Pointer to

Figure 13.2: OllyDbg: SUB executed

DEC is executed and EAX now contains 1. But 1 is non-zero, so the ZF flag is still 0:

**CPU - main thread, module few**

Address	Hex dump	Assembly	Comment
00FF1000	8B4424 04	MOV EAX,DWORD PTR SS:[ARG.1]	
00FF1004	83E8 00	SUB EAX,0	
00FF1007	74 30	JZ SHORT 00FF1039	Switch (cases 0..2, 4 ex
00FF1009	48	DEC EAX	
00FF100A	74 1F	JZ SHORT 00FF102B	
00FF100C	48	DEC EAX	
00FF100D	74 0E	JZ SHORT 00FF101D	
00FF100F	C74424 04 18	MOV DWORD PTR SS:[ARG.1],OFFSET 00FF3018	ASCII "something unknown"
00FF1017	FF25 0020FF01	JMP DWORD PTR DS:[<&MSUCR100.printf>]	
00FF101D	C74424 04 10	MOV DWORD PTR SS:[ARG.1],OFFSET 00FF3010	ASCII "two", case 2 of
00FF1025	FF25 0020FF01	JMP DWORD PTR DS:[<&MSUCR100.printf>]	
00FF102B	C74424 04 08	MOV DWORD PTR SS:[ARG.1],OFFSET 00FF3008	ASCII "one", case 1 of
00FF1033	FF25 0020FF01	JMP DWORD PTR DS:[<&MSUCR100.printf>]	
00FF1039	C74424 04 00	MOV DWORD PTR SS:[ARG.1],OFFSET 00FF3000	ASCII "zero", case 0 of
00FF1041	FF25 0020FF01	JMP DWORD PTR DS:[<&MSUCR100.printf>]	
00FF1047	CC	INT3	
00FF1048	CC	INT3	
00FF1049	CC	INT3	
00FF104A	CC	INT3	
00FF104B	CC	INT3	
00FF104C	CC	INT3	

Jump is not taken  
Dest=few.00FF102B

**Registers (MMX)**

Register	Value	Comment
EAX	00000001	
ECX	6E394714	ASCII "H(*)"
EDX	00000000	
EBX	00000000	
ESP	001EF84C	
EBP	001EF894	
ESI	00000001	
EDI	00FF33A8	few.00FF33A8
EIP	00FF100A	few.00FF100A
CS	002B	32bit 0(FFFFFFFF)
DS	002B	32bit 0(FFFFFFFF)
FS	0053	32bit 7EFD0000(FFF)
GS	002B	32bit 0(FFFFFFFF)
LastErr	00000000	ERROR_SUCCESS
EFL	00000202	(NO,NB,NE,A,NS,PO,GE,G
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
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
00FF3020	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
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 4TuFipnki
00FF3050	01 00 00 00 48 28 2A 00 68 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	

**Registers (MMX)**

Register	Value	Comment
EAX	00000001	
ECX	00000002	
EDX	00FF11CA	RETURN from
EBX	00000001	
ESP	002A4E68	hN* ASCII "pN*"
EBP	002A2848	H(*
ESI	466BAC00	ankF
EDI	00000000	
EIP	7EFD0000	pN*
CS	002B	32bit 0(FFFFFFFF)
DS	002B	32bit 0(FFFFFFFF)
FS	0053	32bit 7EFD0000(FFF)
GS	002B	32bit 0(FFFFFFFF)
LastErr	00000000	ERROR_SUCCESS
EFL	00000202	(NO,NB,NE,A,NS,PO,GE,G
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
001EF84C	00000002	RETURN from
001EF854	00FF11CA	RETURN from
001EF858	00000001	
001EF85C	002A4E68	hN* ASCII "pN*"
001EF860	002A2848	H(*
001EF864	466BAC00	ankF
001EF868	00000000	
001EF86C	00000000	
001EF870	7EFD0000	pN*
001EF874	00000000	
001EF878	00000000	
001EF87C	001EF864	d°
001EF880	0389310	y8u
001EF884	001EF800	Pointer to

Figure 13.3: OllyDbg: first DEC executed

Next DEC is executed. EAX is finally 0 and the ZF flag gets set, because the result is zero:

**CPU - main thread, module few**

Address	Hex dump	ASCII (ANSI - Cy)
00FF1000	7A 65 72 6F 0A 00 00 00 6F 6E 65 0A 00 00 00 00	zero one
00FF1010	74 77 6F 0A 00 00 00 00 73 6F 6D 65 74 68 69 6E	two something unknown
00FF1020	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF FF	
00FF1030	FF FF FF FF 01 00 00 00 00 00 00 00 00 00 00 00	
00FF1040	FE FF FF FF 01 00 00 00 34 54 75 46 CB AB 8A B9	
00FF1050	01 00 00 00 48 28 2A 00 68 4E 2A 00 00 00 00 00	
00FF1060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF1070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF1080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF1090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF10A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF10B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF10C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF10D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

**Registers (MMX)**

Register	Value	Comment
EAX	00000000	
ECX	6E494714	ASCII "H(*)"
EDX	00000000	
EBX	00000000	
ESP	001EF84C	
EBP	001EF894	
ESI	00000001	
EDI	00FF33A8	few.00FF33A8
EIP	00FF100D	few.00FF100D
CS	002B	32bit 0(FFFFFFFF)
DS	002B	32bit 0(FFFFFFFF)
FS	0053	32bit 7EFD0000(FFF)
GS	002B	32bit 0(FFFFFFFF)
LastErr	00000000	ERROR_SUCCESS
EFL	00000246	(NO,NB,E,BE,NS,PE,GE,LE)

**Jump is taken**  
Dest=few.00FF101D

Figure 13.4: OllyDbg: second DEC executed

OllyDbg shows that this jump is to be taken now.

A pointer to the string “two” is to be written into the stack now:

**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 00 20 FF 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF1011	C7 44 24 04 10	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3010: ASCII "two", case 2 of
00FF1012	FF 25 00 20 FF 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF1013	C7 44 24 04 08	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3008: ASCII "one", case 1 of
00FF1014	FF 25 00 20 FF 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF1015	C7 44 24 04 00	MOV DWORD PTR SS:[ARG.1], OFFSET 00FF3000: ASCII "zero", case 0 of
00FF1016	FF 25 00 20 FF 00	JMP DWORD PTR DS:[<&MSUCR100.printf>]
00FF1017	CC	INT3
00FF1018	CC	INT3
00FF1019	CC	INT3
00FF101A	CC	INT3
00FF101B	CC	INT3
00FF101C	CC	INT3

Imm=few.00FF3010, ASCII "two"  
Stack [001EF850]=2  
Jump from 0FF100D

**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	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
00FF3020	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
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 4TuFjnKl
00FF3050	01 00 00 00 48 28 2A 00 68 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	

**RETURN from**

Address	Hex dump	ASCII (ANSI - Cy)
001EF84C	00 00 00 00 00 00 00 00	
001EF850	00 00 00 00 00 00 00 00	
001EF854	00 00 00 00 00 00 00 00	
001EF858	00 00 00 00 00 00 00 00	
001EF85C	00 2A 4E 68	hN*
001EF860	00 2A 28 48	H(* hN*
001EF864	46 6B AC A0	ankF
001EF868	00 00 00 00	
001EF86C	00 00 00 00	
001EF870	7E FD E0 00	p&"
001EF874	00 00 00 00	
001EF878	00 00 00 00	
001EF87C	00 1E F8 64	d°
001EF880	03 89 31 10	y8u
001EF884	00 1E F8 00	Pointer to p

Figure 13.5: OllyDbg: pointer to the string is to be written at the place of the first argument

Please note: the current argument of the function is 2 and 2 is now in the stack at the address 0x001EF850.

MOV writes the pointer to the string at address 0x001EF850 (see the stack window). Then, jump happens. This is the first instruction of the `printf()` function in MSVCR100.DLL (This example was compiled with /MD switch):

**CPU - main thread, module MSVCR100**

Address	Hex dump	ASCII (ANSI - Cy)
00FF3000	74 65 72 6F 0A 00 00 00 6F 6E 65 0A 00 00 00 00	two one
00FF3010	74 77 6F 0A 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 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	4TuF7nKl
00FF3050	01 00 00 00 48 23 2A 00 68 4E 2A 00 00 00 00 00	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	

**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	00FF11C4	RETURN from few
001EF858	00000001	0
001EF85C	002A4E68	hN*
001EF860	002A2848	H(*
001EF864	466BAC00	ankF
001EF868	00000000	
001EF86C	00000000	
001EF870	7EFDE000	p*
001EF874	00000000	
001EF878	00000000	
001EF87C	001EF864	d*
001EF880	03389310	y8
001EF884	001EF800	Printer to next

Figure 13.6: OllyDbg: first instruction of `printf()` in MSVCR100.DLL

Now `printf()` treats the string at 0x00FF3010 as its only argument and prints the string.

This is the last instruction of `printf()`:

**CPU - main thread, module MSVCR100**

Address	Disassembly	Comment
6E4455D0	PUSH EAX	
6E4455D1	PUSH ESI	
6E4455D2	PUSH DWORD PTR SS:[EBP+8]	
6E4455D3	CALL __iob_func	
6E4455D4	ADD EAX,EBX	
6E4455D5	PUSH EAX	
6E4455D6	CALL 6E45C71D	
6E4455D7	MOV DWORD PTR SS:[EBP-1C],EAX	
6E4455D8	CALL __iob_func	
6E4455D9	ADD EAX,EBX	
6E4455DA	PUSH EAX	
6E4455DB	PUSH EAX	
6E4455DC	PUSH EDI	
6E4455DD	CALL 6E4006AC	
6E4455DE	ADD ESP,18	
6E4455DF	MOV DWORD PTR SS:[EBP-4],-2	
6E4455E0	CALL 6E445618	
6E4455E1	MOV EAX,DWORD PTR SS:[EBP-1C]	
6E4455E2	CALL 6E3F0995	
6E4455E3	RET	
6E4455E4	CALL __iob_func	
6E4455E5	ADD EAX,20	

Top of stack [001EF84C]=few.00FF1057

MSVCR100.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 somethin
00FF3020	67 20 75 6E 68 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
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 4TuF7nKj
00FF3050	01 00 00 00 48 28 2A 00 68 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	

**Registers (MMX)**

Register	Value	Comment
EAX	00000004	
ECX	6E445617	MSVCR100.6E445617
EDX	0009DC88	
EBX	00000000	
ESP	001EF84C	
EBP	001EF894	
ESI	00000001	
EDI	00FF33A8	few.00FF33A8
EIP	6E445617	MSVCR100.6E445617

**Stack**

Register	Value	Comment
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)	
O 0	LastErr 00000000 ERROR_SUCCESS	
EFL	00000246 (NO,NB,E,BE,NS,PE,GE,LE)	

**MMX**

Register	Value	Comment
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 Address**

Address	Hex dump	ASCII (ANSI - Cy)
001EF84C	00FF1057	RETURN from f
001EF850	00FF3010	ASCII "two"
001EF854	00FF11CA	RETURN from f
001EF858	00000001	hN*
001EF85C	002A4E68	ASCII "pN"
001EF860	002A2848	H(* hN*
001EF864	466BAC00	ankF
001EF868	00000000	
001EF86C	00000000	
001EF870	7EFD0000	pN
001EF874	00000000	
001EF878	00000000	
001EF87C	001EF864	d*
001EF880	0339310	us
001EF884	001EF800	Pointer to pe

Figure 13.7: OllyDbg: last instruction of `printf()` in `MSVCR100.DLL`

The string "two" was just printed to the console window.



Now let's press F7 or F8 (step over) and return...not to `f()`, but rather to `main()`:

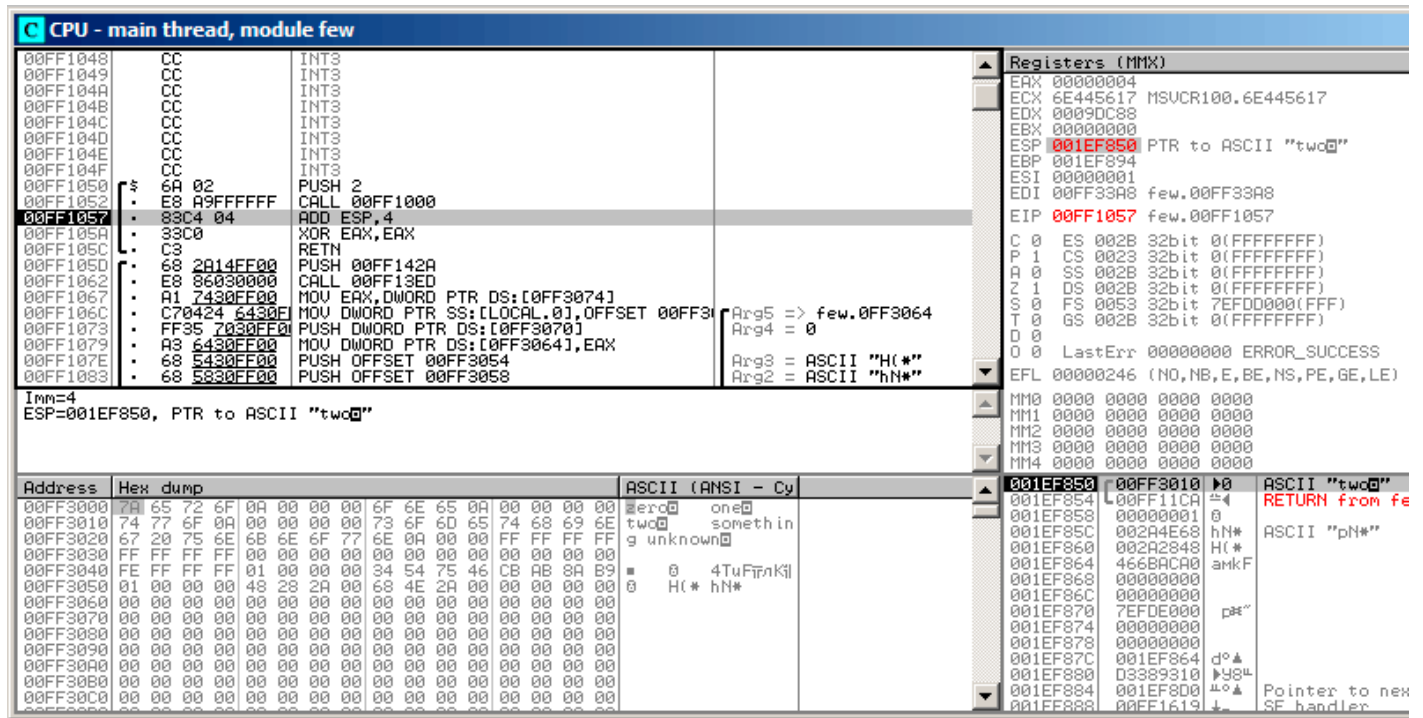


Figure 13.8: OllyDbg: return to `main()`

Yes, the jump was direct, from the guts of `printf()` to `main()`. Because `RA` in the stack points not to some place in `f()`, but rather to `main()`. And `CALL 0x00FF1000` was the actual instruction which called `f()`.

### 13.1.2 ARM: Optimizing Keil 6/2013 (ARM mode)

```
.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    ADRNE   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
```

Again, by investigating this code we cannot say if it was a `switch()` in the original source code, or just a pack of `if()` statements.

Anyway, we see here predicated instructions again (like `ADREQ` (*Equal*)) which is triggered only in case `R0 = 0`, and then loads the address of the string `«zero\n»` into `R0`. The next instruction `BEQ` redirects control flow to `loc_170`, if `R0 = 0`.

An astute reader may ask, will `BEQ` trigger correctly since `ADREQ` before it has already filled the `R0` register with another value? Yes, it will since `BEQ` checks the flags set by the `CMP` instruction, and `ADREQ` does not modify any flags at all.

The rest of the instructions are already familiar to us. There is only one call to `printf()`, at the end, and we have already examined this trick here (6.2.1 on page 46). In the end, there are three paths to `printf()`.

The last instruction, `CMP R0, #2`, is needed to check if `a = 2`. If it is not true, then `ADRNE` loads a pointer to the string `«something unknown\n»` into `R0`, since `a` was already checked to be equal to 0 or 1, and we can sure that the `a` variable is not equal to these numbers at this point. And if `R0 = 2`, a pointer to the string `«two\n»` will be loaded by `ADREQ` into `R0`.

### 13.1.3 ARM: Optimizing Keil 6/2013 (Thumb mode)

```

.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}

```

As was already mentioned, it is not possible to add conditional predicates to most instructions in Thumb mode, so the Thumb-code here is somewhat similar to the easily understandable x86 [CISC](#)-style code.

### 13.1.4 ARM64: Non-optimizing 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      ; jump to default label
    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
    b       .L32
.L35:
    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

```

The type of the input value is *int*, hence register W0 is used to hold it instead of the whole X0 register. The string pointers are passed to `puts()` using an `ADRP/ADD` instructions pair just like it was demonstrated in the “Hello, world!” example: [3.4.5 on page 17](#).

### 13.1.5 ARM64: Optimizing GCC (Linaro) 4.9

```

f12:
        cmp     w0, 1
        beq     .L31
        cmp     w0, 2
        beq     .L32
        cbz     w0, .L35
; default case
        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

```

Better optimized piece of code. *CBZ (Compare and Branch on Zero)* instruction does jump if W0 is zero. There is also a direct jump to `puts()` instead of calling it, like it was explained before: [13.1.1 on page 144](#).

### 13.1.6 MIPS

Listing 13.3: Optimizing GCC 4.4.5 (IDA)

```

f:
        lui     $gp, (__gnu_local_gp >> 16)
; is it 1?
        li      $v0, 1
        beq     $a0, $v0, loc_60
        la      $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; is it 2?
        li      $v0, 2
        beq     $a0, $v0, loc_4C
        or      $at, $zero ; branch delay slot, NOP
; jump, if not equal to 0:
        bnez    $a0, loc_38
        or      $at, $zero ; branch delay slot, NOP
; zero case:
        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

```

The function always ends with calling `puts()`, so here we see a jump to `puts()` (JR: “Jump Register”) instead of “jump and link”. We talked about this earlier: [13.1.1 on page 144](#).

We also often see NOP instructions after LW ones. This is “load delay slot”: another *delay slot* in MIPS. An instruction next to LW may execute at the moment while LW loads value from memory. However, the next instruction must not use the result of LW. Modern MIPS CPUs have a feature to wait if the next instruction uses result of LW, so this is somewhat outdated, but GCC still adds NOPs for older MIPS CPUs. In general, it can be ignored.

### 13.1.7 Conclusion

A *switch()* with few cases is indistinguishable from an *if/else* construction, for example: [listing 13.1.1](#).

## 13.2 A lot of cases

If a `switch()` statement contains a lot of cases, it is not very convenient for the compiler to emit too large code with a lot of JE/JNE instructions.

```

#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
};

```

### 13.2.1 x86

#### Non-optimizing MSVC

We get (MSVC 2010):

## Listing 13.4: 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
    npad    2 ; align next label
$LN11@f:
    DD      $LN6@f ; 0
    DD      $LN5@f ; 1
    DD      $LN4@f ; 2
    DD      $LN3@f ; 3
    DD      $LN2@f ; 4
_f ENDP

```

What we see here is a set of `printf()` calls with various arguments. All they have not only addresses in the memory of the process, but also internal symbolic labels assigned by the compiler. All these labels are also mentioned in the `$LN11@f` internal table.

At the function start, if *a* is greater than 4, control flow is passed to label `$LN1@f`, where `printf()` with argument 'something unknown' is called.

But if the value of *a* is less or equals to 4, then it gets multiplied by 4 and added with the `$LN11@f` table address. That is how an address inside the table is constructed, pointing exactly to the element we need. For example, let's say *a* is equal to 2.  $2 * 4 = 8$  (all table elements are addresses in a 32-bit process and that is why all elements are 4 bytes wide). The address of the `$LN11@f` table + 8 is the table element where the `$LN4@f` label is stored. `JMP` fetches the `$LN4@f` address from the table and jumps to it.

This table is sometimes called *jumptable* or *branch table*<sup>3</sup>.

<sup>3</sup>The whole method was once called *computed GOTO* in early versions of FORTRAN: [wikipedia](https://en.wikipedia.org/wiki/Computed_goto). Not quite relevant these days, but what a term!

Then the corresponding `printf()` is called with argument 'two'. Literally, the `jmp DWORD PTR $LN11@f[ecx*4]` instruction implies *jump to the DWORD that is stored at address  $\$LN11@f + ecx * 4$* .

`npad` ([88 on page 853](#)) is assembly language macro that aligning the next label so that it is to be stored at an address aligned on a 4 byte (or 16 byte) boundary. This is very suitable for the processor since it is able to fetch 32-bit values from memory through the memory bus, cache memory, etc, in a more effective way if it is aligned.

## OllyDbg

Let's try this example in OllyDbg. The input value of the function (2) is loaded into EAX:

**CPU - main thread, module lot**

Address	Hex dump	ASCII (ANSI - Cy)
010B1000	55 8B EC	PUSH EBP
010B1001	51	MOV EBP, ESP
010B1003	51	PUSH ECX
010B1004	8B 45 08	MOV EAX, DWORD PTR SS:[EBP+8]
010B1007	8B 45 FC	MOV EDI, DWORD PTR SS:[EBP-4], EAX
010B100A	83 7D FC 04	CMP DWORD PTR SS:[EBP-4], 4
010B100E	77 5A	JA SHORT 010B106A
010B1010	8B 4D FC	MOV ECX, DWORD PTR SS:[EBP-4]
010B1013	FF 24 8D 7C 10 00	JMP DWORD PTR DS:[ECX*4+010B107C]
010B101F	68 00 30 00 00 01	PUSH OFFSET 010B3000
010B1025	FF 15 00 20 00 00	CALL DWORD PTR DS:[<&MSUCR100.printf>]
010B1028	83 C4 04	ADD ESP, 4
010B102B	EB 4E	JMP SHORT 010B1078
010B102A	68 00 30 00 00 01	PUSH OFFSET 010B3000
010B102F	FF 15 00 20 00 00	CALL DWORD PTR DS:[<&MSUCR100.printf>]
010B1035	83 C4 04	ADD ESP, 4
010B1038	EB 3E	JMP SHORT 010B1078
010B103A	68 00 30 00 00 01	PUSH OFFSET 010B3010
010B103F	FF 15 00 20 00 00	CALL DWORD PTR DS:[<&MSUCR100.printf>]
010B1045	83 C4 04	ADD ESP, 4
010B1048	EB 2E	JMP SHORT 010B1078

**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 7EFDD000(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)
010B3000	65 72 6F 0A 00 00 00 00 6F 6E 65 0A 00 00 00 00	zero one
010B3010	74 77 6F 0A 00 00 00 00 74 63 72 65 65 0A 00 00	two three
010B3020	66 6F 75 72 0A 00 00 00 73 6F 6D 65 74 63 69 6E	four somethin
010B3030	67 20 75 6E 6B 6E 6F 77 6E 0A 00 00 FF FF FF FF	g unknown
010B3040	FF FF FF FF 01 00 00 00 9A E2 68 1D 65 1D 97 E2	
010B3050	01 00 00 00 48 28 03 00 68 4E 03 00 00 00 00 00	
010B3060	00 00 00 00 00 00 00 00 00 00 00 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 6E494714 76In OFFSET MSUCR100**

Address	Hex dump	ASCII (ANSI - Cy)
003CFDA8	003CFDB8	76In
003CFDB8	010B109A	RETURN from lot
003CFDB4	00000002	
003CFDB8	003CFDFC	76In
003CFDBC	010B120E	RETURN from lot
003CFDC0	00000001	
003CFDC4	00034E68	hN
003CFDC8	00032848	H
003CFDCC	1D541F66	fT
003CFDD0	00000000	
003CFDD4	00000000	
003CFDD8	7EFDE000	pH
003CFDDC	00000000	
003CFDE0	00000000	

Figure 13.9: OllyDbg: function's input value is loaded in EAX

The input value is checked, is it bigger than 4? If not, the “default” jump is not taken:

**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
010B100A	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:[&MSVCR100.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:[&MSVCR100.printf]
010B1035	83C4 04	ADD ESP,4
010B1038	EB 3E	JMP SHORT 010B1078
010B103A	68 10300001	PUSH OFFSET 010B3010
010B103F	FF15 00200000	CALL DWORD PTR DS:[&MSVCR100.printf]
010B1045	83C4 04	ADD ESP,4
010B1048	EB 2E	JMP SHORT 010B1078

**Registers (MMX)**

EAX	00000002
ECX	6E494714 MSVCR100.__initenv
EDX	00000000
EBX	00000000
ESP	003CFD08
EBP	003CFD0C
ESI	00000001
EDI	010B33B8 lot.010B33B8
EIP	010B100E lot.010B100E
C	1 ES 002B 32bit 0(FFFFFFFF)
P	0 CS 0023 32bit 0(FFFFFFFF)
A	1 SS 002B 32bit 0(FFFFFFFF)
Z	0 DS 002B 32bit 0(FFFFFFFF)
S	1 FS 0053 32bit 7EFD0000(FFF)
T	0 GS 002B 32bit 0(FFFFFFFF)
D	0
O	0 LastErr 00000000 ERROR_SUCCESS
EFL	00000293 (NO,B,NE,BE,S,PO,L,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

**Jump is not taken**  
**Dest=010B106A**

Address	Hex dump	ASCII (ANSI - Cy)
010B3000	74 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 somethin
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	b'th#e#4t
010B3060	01 00 00 00 48 28 03 00 68 4E 03 00 00 00 00 00	H( hN
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	

**003CFD08 00000002**

**003CFD0C 010B109A** RETURN from

**003CFD0E 00000002**

**003CFD0F 003CFD0C** RETURN from

**003CFD10 010B120E**

**003CFD11 00000001**

**003CFD12 00034E68** hN

**003CFD13 00032848** H(

**003CFD14 1D541F66** fT#

**003CFD15 00000000**

**003CFD16 00000000**

**003CFD17 7EFD0000** c#

**003CFD18 00000000**

**003CFD19 00000000**

Figure 13.10: OllyDbg: 2 is no bigger than 4: no jump is taken

Here we see a jumptable:

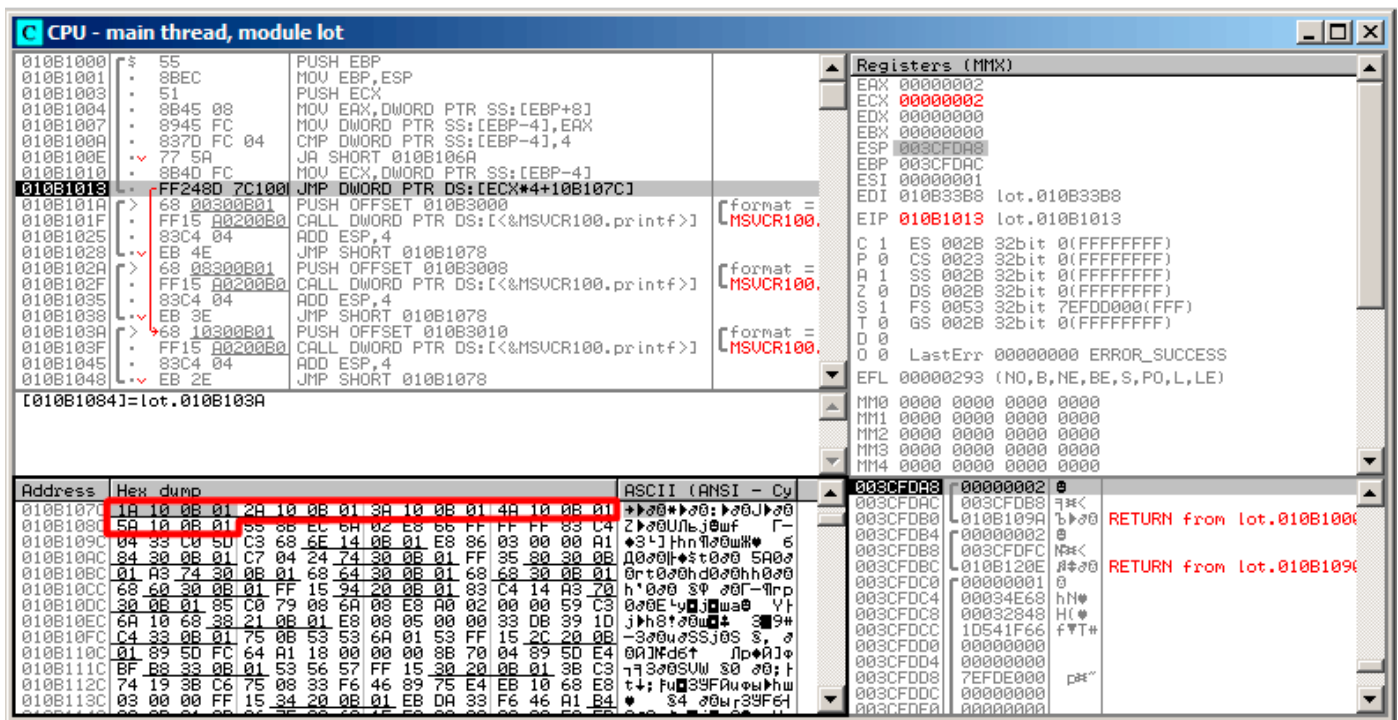


Figure 13.11: OllyDbg: calculating destination address using jumptable

Here we've clicked "Follow in Dump" → "Address constant", so now we see the *jumptable* in the data window. These are 5 32-bit values<sup>4</sup>. ECX is now 2, so the second element (counting from zero) of the table is to be used. It's also possible to click "Follow in Dump" → "Memory address" and OllyDbg will show the element addressed by the JMP instruction. That's 0x010B103A.

<sup>4</sup>They are underlined by OllyDbg because these are also FIXUPs: 68.2.6 on page 673, we are going to come back to them later

After the jump we are at 0x010B103A: the code printing “two” will now be executed:

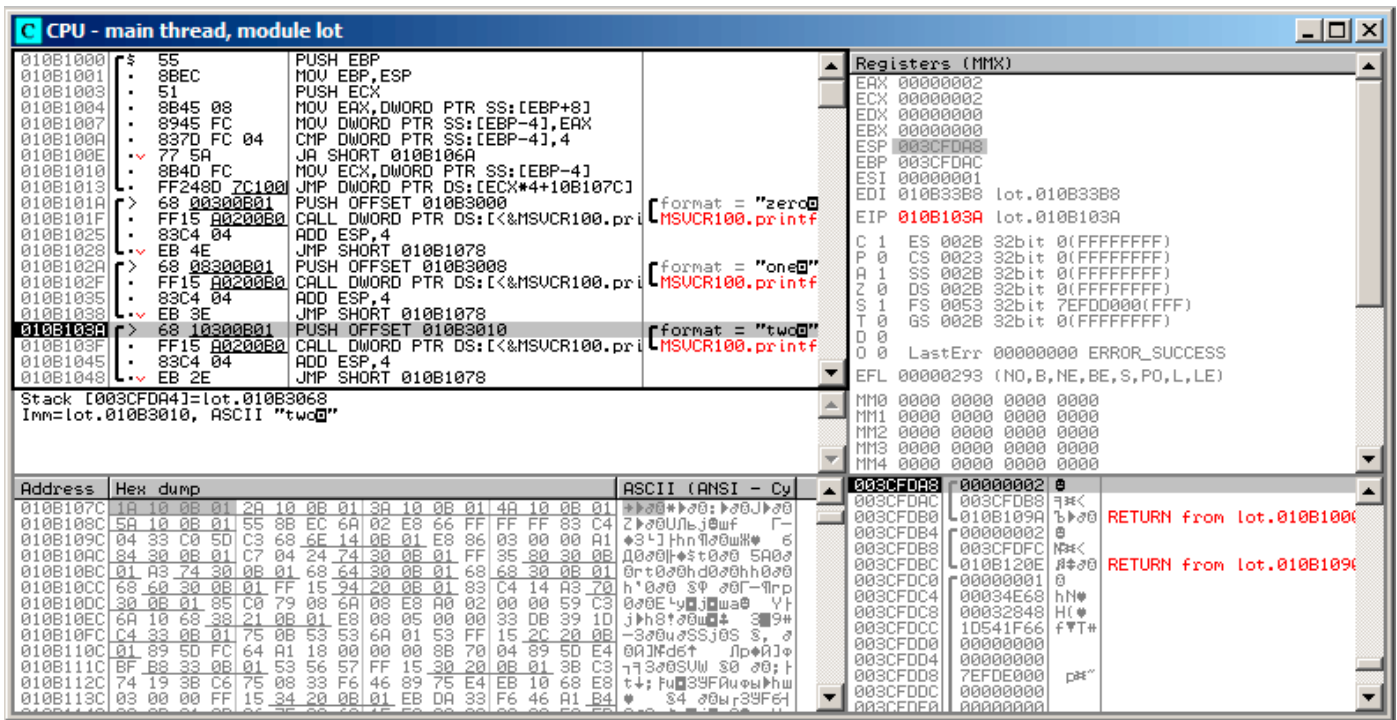


Figure 13.12: OllyDbg: now we at the case: label

## Non-optimizing GCC

Let's see what GCC 4.4.1 generates:

Listing 13.5: 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

```



```

    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

```

It is almost the same, with a little nuance: argument `arg_0` is multiplied by 4 by shifting it to left by 2 bits (it is almost the same as multiplication by 4) ( [16.2.1 on page 205](#)). Then the address of the label is taken from the `off_804855C` array, stored in `EAX`, and then `JMP EAX` does the actual jump.

### 13.2.2 ARM: Optimizing Keil 6/2013 (ARM mode)

Listing 13.6: Optimizing Keil 6/2013 (ARM mode)

```

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    ; jumptable 00000178 default case

00000180
00000180          loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA      B       zero_case      ; jumptable 00000178 case 0

00000184
00000184          loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA      B       one_case       ; jumptable 00000178 case 1

00000188
00000188          loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA      B       two_case      ; jumptable 00000178 case 2

0000018C
0000018C          loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA      B       three_case     ; jumptable 00000178 case 3

00000190
00000190          loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA      B       four_case     ; jumptable 00000178 case 4

00000194
00000194          zero_case ; CODE XREF: f2+4
00000194          ; f2:loc_180
00000194 EC 00 8F E2      ADR     R0, aZero      ; jumptable 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      ; jumptable 00000178 case 1

```

```

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

```

This code makes use of the ARM mode feature in which all instructions have a fixed size of 4 bytes.

Let's keep in mind that the maximum value for *a* is 4 and any greater value will cause «something unknown» string to be printed.

The first `CMP R0, #5` instruction compares the input value of *a* with 5.

The next `ADDCC PC, PC, R0, LSL#2`<sup>5</sup> instruction is being executed only if  $R0 < 5$  (*CC=Carry clear/Less than*). Consequently, if `ADDCC` does not trigger (it is a  $R0 \geq 5$  case), a jump to `default_case` label will occur.

But if  $R0 < 5$  and `ADDCC` triggers, the following is to be happen:

The value in *R0* is multiplied by 4. In fact, `LSL#2` at the instruction's suffix stands for "shift left by 2 bits". But as we will see later ( [16.2.1 on page 205](#)) in section "Shifts", shift left by 2 bits is equivalent to multiplying by 4.

Then we add  $R0 * 4$  to the current value in *PC*, thus jumping to one of the *B (Branch)* instructions located below.

At the moment of the execution of `ADDCC`, the value in *PC* is 8 bytes ahead (0x180) than the address at which the `ADDCC` instruction is located (0x178), or, in other words, 2 instructions ahead.

This is how the pipeline in ARM processors works: when `ADDCC` is executed, the processor at the moment is beginning to process the instruction after the next one, so that is why *PC* points there. This has to be memorized.

If  $a = 0$ , then is to be added to the value in *PC*, and the actual value of the *PC* will be written into *PC* (which is 8 bytes ahead) and a jump to the label `loc_180` will happen, which is 8 bytes ahead of the point where the `ADDCC` instruction is.

If  $a = 1$ , then  $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 12 = 0x184$  will be written to *PC*, which is the address of the `loc_184` label.

With every 1 added to *a*, the resulting *PC* is increased by 4. 4 is the instruction length in ARM mode and also, the length of each *B* instruction, of which there are 5 in row.

Each of these five *B* instructions passes control further, to what was programmed in the `switch()`. Pointer loading of the corresponding string occurs there,etc.

### 13.2.3 ARM: Optimizing Keil 6/2013 (Thumb mode)

Listing 13.7: Optimizing Keil 6/2013 (Thumb mode)

```

000000F6          EXPORT f2
000000F6          f2
000000F6 10 B5      PUSH      {R4,LR}

```

<sup>5</sup>ADD—addition

```

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
00000106          zero_case ; CODE XREF: f2+4
00000106 8D A0          ADR      R0, aZero ; jumtable 000000FA case 0
00000108 06 E0          B       loc_118

0000010A
0000010A          one_case ; CODE XREF: f2+4
0000010A 8E A0          ADR      R0, aOne ; jumtable 000000FA case 1
0000010C 04 E0          B       loc_118

0000010E
0000010E          two_case ; CODE XREF: f2+4
0000010E 8F A0          ADR      R0, aTwo ; jumtable 000000FA case 2
00000110 02 E0          B       loc_118

00000112
00000112          three_case ; CODE XREF: f2+4
00000112 90 A0          ADR      R0, aThree ; jumtable 000000FA case 3
00000114 00 E0          B       loc_118

00000116
00000116          four_case ; CODE XREF: f2+4
00000116 91 A0          ADR      R0, aFour ; jumtable 000000FA case 4
00000118
00000118          loc_118 ; CODE XREF: f2+12
00000118          ; f2+16
00000118 06 F0 6A F8      BL      __2printf
0000011C 10 BD          POP     {R4,PC}

0000011E
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

```

One cannot be sure that all instructions in Thumb and Thumb-2 modes has the same size. It can even be said that in these modes the instructions have variable lengths, just like in x86.

So there is a special table added that contains information about how much cases are there (not including default-case), and an offset for each with a label to which control must be passed in the corresponding case.

A special function is present here in order to deal with the table and pass control, named `__ARM_common_switch8_thumb`. It starts with `BX PC`, whose function is to switch the processor to ARM-mode. Then you see the function for table processing. It is too complex to describe it here now, so let's omit it.

It is interesting to note that the function uses the `LR` register as a pointer to the table. Indeed, after calling of this function, `LR` contains the address after `BL __ARM_common_switch8_thumb` instruction, where the table starts.

It is also worth noting that the code is generated as a separate function in order to reuse it, so the compiler not generates the same code for every switch() statement.

IDA successfully perceived it as a service function and a table, and added comments to the labels like jumptable 000000FA case 0.

## 13.2.4 MIPS

Listing 13.8: Optimizing GCC 4.4.5 (IDA)

```
f:
    lui    $gp, (__gnu_local_gp >> 16)
; jump to loc_24 if input value is lesser than 5:
    sltiu  $v0, $a0, 5
    bnez   $v0, loc_24
    la     $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; input value is greater or equal to 5.
; print "something unknown" and finish:
    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
; load address of jumptable
; LA is pseudoinstruction, LUI and ADDIU pair are there in fact:
    la     $v0, off_120
; multiply input value by 4:
    sll    $a0, 2
; sum up multiplied value and jumptable address:
    addu   $a0, $v0, $a0
; load element from jumptable:
    lw     $v0, 0($a0)
    or     $at, $zero ; NOP
; jump to the address we got in jumptable:
    jr     $v0
    or     $at, $zero ; branch delay slot, NOP

sub_44:
                                # DATA XREF: .rodata:0000012C
; print "three" and finish
    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
; print "four" and finish
    lui    $a0, ($LC4 >> 16) # "four"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC4 & 0xFFFF) # "four" ; branch delay slot

sub_6C:
                                # DATA XREF: .rodata:off_120
; print "zero" and finish
    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
; print "one" and finish
    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
; print "two" and finish
        lui     $a0, ($LC2 >> 16) # "two"
        lw      $t9, (puts & 0xFFFF)($gp)
        or      $at, $zero ; NOP
        jr      $t9
        la      $a0, ($LC2 & 0xFFFF) # "two" ; branch delay slot

; may be placed in .rodata section:
off_120: .word sub_6C
        .word sub_80
        .word sub_94
        .word sub_44
        .word sub_58

```

The new instruction for us is SLTIU (“Set on Less Than Immediate Unsigned”). This is the same as SLTU (“Set on Less Than Unsigned”), but “I” stands for “immediate”, i.e., a number has to be specified in the instruction itself.

BNEZ is “Branch if Not Equal to Zero”.

Code is very close to the other [ISAs](#). SLL (“Shift Word Left Logical”) does multiplication by 4. MIPS is a 32-bit CPU after all, so all addresses in the *jump table* are 32-bit ones.

### 13.2.5 Conclusion

Rough skeleton of *switch()*:

Listing 13.9: x86

```

MOV REG, input
CMP REG, 4 ; maximal number of cases
JA default
SHL REG, 2 ; find element in table. shift for 3 bits in x64.
MOV REG, jump_table[REG]
JMP REG

case1:
    ; do something
    JMP exit
case2:
    ; do something
    JMP exit
case3:
    ; do something
    JMP exit
case4:
    ; do something
    JMP exit
case5:
    ; do something
    JMP exit

default:
    ...

exit:
    ....

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5

```

The jump to the address in the jump table may also be implemented using this instruction: `JMP jump_table[REG*4]`. Or `JMP jump_table[REG*8]` in x64.

A *jump table* is just array of pointers, like the one described later: [18.5 on page 272](#).

## 13.3 When there are several *case* statements in one block

Here is a very widespread construction: several *case* statements for a single block:

```
#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);
};
```

It's too wasteful to generate a block for each possible case, so what is usually done is to generate each block plus some kind of dispatcher.

### 13.3.1 MSVC

Listing 13.10: Optimizing 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      npad     2 ; align $LN11@f table on 16-byte boundary
31 $LN11@f:
32      DD      $LN5@f ; print '1, 2, 7, 10'
33      DD      $LN4@f ; print '3, 4, 5'
34      DD      $LN3@f ; print '8, 9, 21'
35      DD      $LN2@f ; print '22'
36      DD      $LN1@f ; print '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

```

We see two tables here: the first table (\$LN10@f) is an index table, and the second one (\$LN11@f) is an array of pointers to blocks.

First, the input value is used as an index in the index table (line 13).

Here is a short legend for the values in the table: 0 is the first *case* block (for values 1, 2, 7, 10), 1 is the second one (for values 3, 4, 5), 2 is the third one (for values 8, 9, 21), 3 is the fourth one (for value 22), 4 is for the default block.

There we get an index for the second table of code pointers and we jump to it (line 14).

What is also worth noting is that there is no case for input value 0. That's why we see the DEC instruction at line 10, and the table starts at  $a = 1$ , because there is no need to allocate a table element for  $a = 0$ .

This is a very widespread pattern.

So why is this economical? Why isn't it possible to make it as before ( [13.2.1 on page 162](#)), just with one table consisting of block pointers? The reason is that the elements in index table are 8-bit, hence it's all more compact.

### 13.3.2 GCC

GCC does the job in the way we already discussed ( [13.2.1 on page 162](#)), using just one table of pointers.

### 13.3.3 ARM64: Optimizing GCC 4.9.1

There is no code to be triggered if the input value is 0, so GCC tries to make the jump table more compact and so it starts at 1 as an input value.

GCC 4.9.1 for ARM64 uses an even cleverer trick. It's able to encode all offsets as 8-bit bytes. Let's recall that all ARM64 instructions have a size of 4 bytes. GCC uses the fact that all offsets in my tiny example are in close proximity to each other. So the jump table consisting of single bytes.

Listing 13.11: Optimizing GCC 4.9.1 ARM64

```
f14:
; input value in W0
    sub    w0, w0, #1
    cmp    w0, 21
; branch if less or equal (unsigned):
    bls    .L9
.L2:
; print "default":
    adrp   x0, .LC4
    add    x0, x0, :lo12:LC4
    b      puts
.L9:
; load jumtable address to X1:
    adrp   x1, .L4
    add    x1, x1, :lo12:L4
; W0=input_value-1
; load byte from the table:
    ldrb   w0, [x1,w0,uxtw]
; load address of the Lrtx label:
    adr    x1, .Lrtx4
; multiply table element by 4 (by shifting 2 bits left) and add (or subtract) to the address of
    Lrtx:
    add    x0, x1, w0, sxtb #2
; jump to the calculated address:
    br     x0
; this label is pointing in code (text) segment:
.Lrtx4:
    .section      .rodata
; everything after ".section" statement is allocated in the read-only data (rodata) segment:
.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
    .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
; everything after ".text" statement is allocated in the code (text) segment:
.L7:
; print "22"
    adrp   x0, .LC3
    add    x0, x0, :lo12:LC3
    b      puts
.L6:
```



```

; print "8, 9, 21"
    adrp    x0, .LC2
    add     x0, x0, :lo12:.LC2
    b       puts
.L5:
; print "3, 4, 5"
    adrp    x0, .LC1
    add     x0, x0, :lo12:.LC1
    b       puts
.L3:
; print "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"

```

Let's compile this example to object file and open it in [IDA](#). Here is the jump table:

Listing 13.12: jumtable 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
.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

```

So in case of 1, 9 is to be multiplied by 4 and added to the address of Lrtx4 label. In case of 22, 0 is to be multiplied by 4, resulting in 0. Right after the Lrtx4 label is the L7 label, where you can find the code that prints “22”. There is no jump table in the code segment, it's allocated in a separate .rodata section (there is no special need to place it in the code section).

There are also negative bytes (0xF7), they are used for jumping back to the code that prints the “default” string (at .L2).

## 13.4 Fall-through

Another very popular usage of `switch()` is the fall-through. Here is a small example:

```

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 };

```

If *type* = 1 (R), *read* is to be set to 1, if *type* = 2 (W), *write* is to be set to 2. In case of *type* = 3 (RW), both *read* and *write* is to be set to 1.

The code at line 14 is executed in two cases: if *type* = *RW* or if *type* = *W*. There is no “break” for “case *RW*”x and that’s OK.

### 13.4.1 MSVC x86

Listing 13.13: 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
    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
----	------

The code mostly resembles what is in the source. There are no jumps between labels \$LN4@f and \$LN3@f: so when code flow is at \$LN4@f, *read* is first set to 1, then *write*. This is why it's called fall-through: code flow falls through one piece of code (setting *read*) to another (setting *write*). If *type* = *W*, we land at \$LN3@f, so no code setting *read* to 1 is executed.

## 13.4.2 ARM64

Listing 13.14: 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] ; set "read" and "write" local variables to zero
    str    wzr, [x29,40]
    ldr    w0, [x29,28] ; load "type" argument
    cmp    w0, 2          ; type=W?
    beq    .L3
    cmp    w0, 3          ; type=RW?
    beq    .L4
    cmp    w0, 1          ; type=R?
    beq    .L5
    b      .L6            ; otherwise...
.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
    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] ; load "read"
    ldr    w2, [x29,40] ; load "write"
    bl     printf
    ldp    x29, x30, [sp], 48
    ret
```

Merely the same thing. There are no jumps between labels .L4 and .L3.

## 13.5 Exercises

### 13.5.1 Exercise #1

It's possible to rework the C example in [13.2 on page 156](#) in such way that the compiler can produce even smaller code, but will work just the same. Try to achieve it.

# Chapter 14

## Loops

### 14.1 Simple example

#### 14.1.1 x86

There is a special LOOP instruction in x86 instruction set for checking the value in register ECX and if it is not 0, to [decrement](#) ECX and pass control flow to the label in the LOOP operand. Probably this instruction is not very convenient, and there are no any modern compilers which emit it automatically. So, if you see this instruction somewhere in code, it is most likely that this is a manually written piece of assembly code.

In C/C++ loops are usually constructed using `for()`, `while()` or `do/while()` statements.

Let's start with `for()`.

This statement defines loop initialization (set loop counter to initial value), loop condition (is the counter bigger than a limit?), what is done at each iteration ([increment/decrement](#)) and of course loop body.

```
for (initialization; condition; at each iteration)
{
    loop_body;
}
```

The generated code is consisting of four parts as well.

Let's start with a simple example:

```
#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;
};
```

Result (MSVC 2010):

Listing 14.1: MSVC 2010

```
_i$ = -4
_main PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     DWORD PTR _i$[ebp], 2    ; loop initialization
    jmp     SHORT $LN3@main
```

```

$LN2@main:
    mov     eax, DWORD PTR _i$[ebp] ; here is what we do after each iteration:
    add     eax, 1                   ; add 1 to (i) value
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 10  ; this condition is checked *before* each iteration
    jge     SHORT $LN1@main         ; if (i) is biggest or equals to 10, lets finish loop'
    mov     ecx, DWORD PTR _i$[ebp] ; loop body: call printing_function(i)
    push    ecx
    call    _printing_function
    add     esp, 4
    jmp     SHORT $LN2@main         ; jump to loop begin
$LN1@main:
                                ; loop end
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP

```

As we see, nothing special.

GCC 4.4.1 emits almost the same code, with one subtle difference:

Listing 14.2: 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 ; (i) initializing
    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 ; (i) increment

loc_8048476:
    cmp     [esp+20h+var_4], 9
    jle     short loc_8048465 ; if i<=9, continue loop
    mov     eax, 0
    leave
    retn
main      endp

```

Now let's see what we get with optimization turned on (/Ox):

Listing 14.3: Optimizing 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

```

What happens here is that space for the *i* variable is not allocated in the local stack anymore, but uses an individual register for it, ESI. This is possible in such small functions where there aren't many local variables.

One very important thing is that the *f()* function must not change the value in ESI. Our compiler is sure here. And if the compiler decides to use the ESI register in *f()* too, its value would have to be saved at the function's prologue and restored at the function's epilogue, almost like in our listing: please note `PUSH ESI/POP ESI` at the function start and end.

Let's try GCC 4.4.1 with maximal optimization turned on (-O3 option):

Listing 14.4: Optimizing GCC 4.4.1

```
main      proc near
var_10    = dword ptr -10h

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        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
```

Huh, GCC just unwound our loop.

[Loop unwinding](#) has an advantage in the cases when there aren't much iterations and we could cut some execution time by removing all loop support instructions. On the other side, the resulting code is obviously larger.

Big unrolled loops are not recommended in modern times, because bigger functions may require bigger cache footprint<sup>1</sup>.

OK, let's increase the maximum value of the *i* variable to 100 and try again. GCC does:

Listing 14.5: GCC

```
main      public main
          proc near
var_20    = dword ptr -20h

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        push    ebx
        mov     ebx, 2      ; i=2
        sub     esp, 1Ch

; aligning label loc_80484D0 (loop body begin) by 16-byte border:
        nop

loc_80484D0:
; pass (i) as first argument to printing_function():
        mov     [esp+20h+var_20], ebx
        add     ebx, 1      ; i++
        call    printing_function
```

<sup>1</sup>A very good article about it: [\[Dre07\]](#). Another recommendations about loop unrolling from Intel are here : [\[Int14, p. 3.4.1.7\]](#).

```
    cmp     ebx, 64h ; i==100?
    jnz     short loc_80484D0 ; if not, continue
    add     esp, 1Ch
    xor     eax, eax ; return 0
    pop     ebx
    mov     esp, ebp
    pop     ebp
    retn
main      endp
```

It is quite similar to what MSVC 2010 with optimization (/Ox) produce, with the exception that the EBX register is allocated for the *i* variable. GCC is sure this register will not be modified inside of the *f()* function, and if it will, it will be saved at the function prologue and restored at epilogue, just like here in the *main()* function.

### 14.1.2 x86: OllyDbg

Let's compile our example in MSVC 2010 with `/Ox` and `/Ob0` options and load it into OllyDbg.

It seems that OllyDbg is able to detect simple loops and show them in square brackets, for convenience:

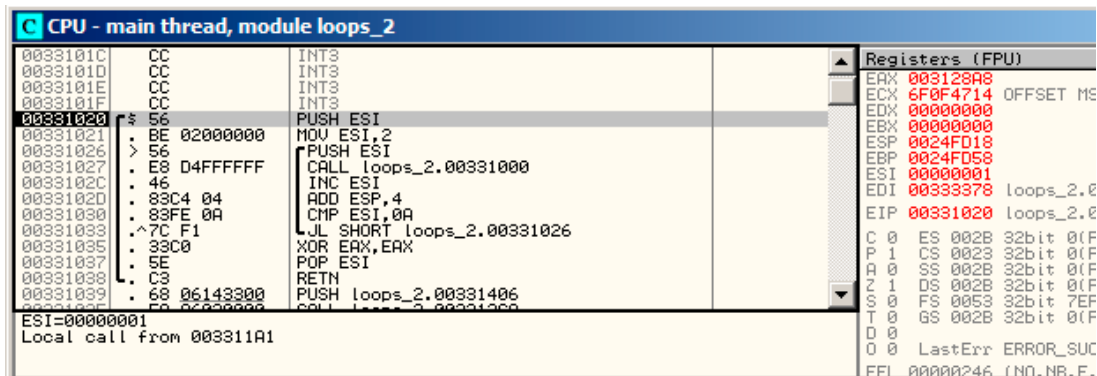


Figure 14.1: OllyDbg: `main()` begin

By tracing (F8 – step over) we see ESI [incrementing](#). Here, for instance,  $ESI = i = 6$ :

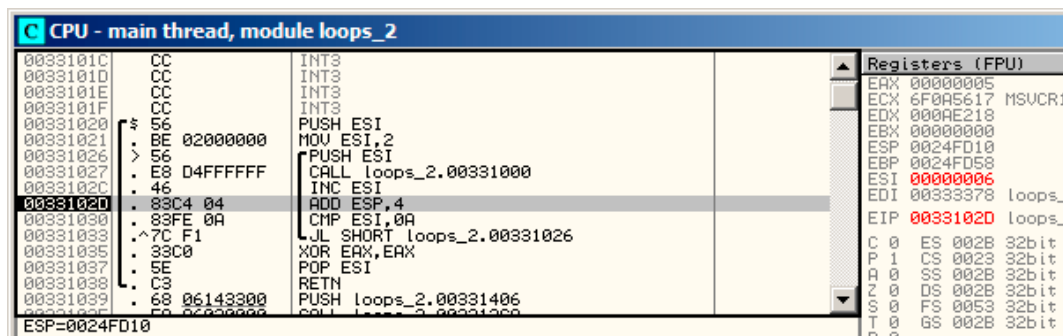


Figure 14.2: OllyDbg: loop body just executed with  $i = 6$

9 is the last loop value. That's why JL is not triggering after the [increment](#), and the function will finish:

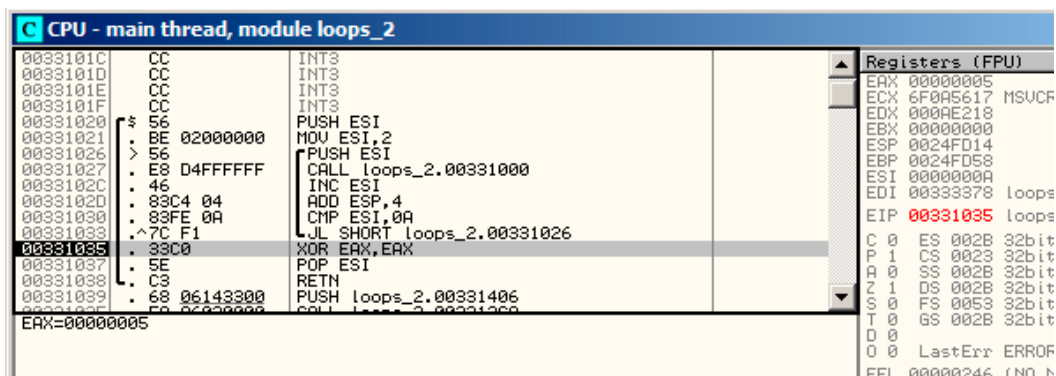


Figure 14.3: OllyDbg:  $ESI = 10$ , loop end

### 14.1.3 x86: tracer

As we might see, it is not very convenient to trace manually in the debugger. That's a reason we will try [tracer](#).

We open compiled example in [IDA](#), find the address of the instruction `PUSH ESI` (passing the sole argument to `f()`), which is `0x401026` for this case and we run the [tracer](#):

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX just sets a breakpoint at the address and tracer will then print the state of the registers.

In the `tracer.log` This is what we see:



```

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 PF 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)

```

We see how the value of ESI register changes from 2 to 9.

Even more than that, the [tracer](#) can collect register values for all addresses within the function. This is called *trace* there. Every instruction gets traced, all interesting register values are recorded. Then, an [IDA](#).idc-script is generated, that adds comments. So, in the [IDA](#) we've learned that the `main()` function address is `0x00401020` and we run:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF stands for set breakpoint on function.

As a result, we get the `loops_2.exe.idc` and `loops_2.exe_clear.idc` scripts.

We load `loops_2.exe.idc` into [IDA](#) and see:

```
.text:00401020
.text:00401020 ; ===== S U B R O U T I N E =====
.text:00401020
.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
```

Figure 14.4: [IDA](#) with `.idc`-script loaded

We see that `ESI` can be from 2 to 9 at the start of the loop body, but from 3 to 0xA (10) after the increment. We can also see that `main()` is finishing with 0 in `EAX`.

[tracer](#) also generates `loops_2.exe.txt`, that contains information about how many times each instruction was executed and register values:

Listing 14.6: `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
```

We can use `grep` here.

## 14.1.4 ARM

### Non-optimizing Keil 6/2013 (ARM mode)

```
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
loc_368     ; CODE XREF: main+8
```

```

CMP    R4, #0xA
BLT    loc_35C
MOV    R0, #0
LDMFD  SP!, {R4,PC}

```

Iteration counter  $i$  is to be stored in the R4 register.

The “MOV R4, #2” instruction just initializes  $i$ .

The “MOV R0, R4” and “BL printing\_function” instructions compose the body of the loop, the first instruction preparing the argument for  $f()$  function and the second calling the function.

The “ADD R4, R4, #1” instruction just adds 1 to the  $i$  variable at each iteration.

“CMP R4, #0xA” compares  $i$  with 0xA (10). The next instruction BLT (*Branch Less Than*) jumps if  $i$  is less than 10.

Otherwise, 0 is to be written into R0 (since our function returns 0) and function execution finishes.

### Optimizing Keil 6/2013 (Thumb mode)

```

_main
    PUSH    {R4,LR}
    MOVS    R4, #2

loc_132
    MOVS    R0, R4
    BL      printing_function
    ADDS    R4, R4, #1
    CMP     R4, #0xA
    BLT     loc_132
    MOVS    R0, #0
    POP     {R4,PC}
; CODE XREF: _main+E

```

Practically the same.

### Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```

_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}

```

In fact, this was in my `f()` function:

```
void printing_function(int i)
{
    printf ("%d\n", i);
};
```

So, LLVM not just *unrolled* the loop, but also *inlined* my very simple function `f()`, and inserted its body 8 times instead of calling it. This is possible when the function is so simple (like mine) and when it is not called too much (like here).

#### ARM64: Optimizing GCC 4.9.1

Listing 14.7: Optimizing GCC 4.9.1

```
printing_function:
; prepare second argument of printf():
    mov     w1, w0
; load address of the "f(%d)\n" string
    adrp    x0, .LC0
    add     x0, x0, :lo12:LC0
; just branch here instead of branch with link and return:
    b       printf
main:
; save FP and LR in the local stack:
    stp     x29, x30, [sp, -32]!
; set up stack frame:
    add     x29, sp, 0
; save contents of X19 register in the local stack:
    str     x19, [sp,16]
; we will use W19 register as counter.
; set initial value of 2 to it:
    mov     w19, 2
.L3:
; prepare first argument of printing_function():
    mov     w0, w19
; increment counter register.
    add     w19, w19, 1
; W0 here still holds value of counter value before increment.
    bl     printing_function
; is it end?
    cmp     w19, 10
; no, jump to the loop body begin:
    bne     .L3
; return 0
    mov     w0, 0
; restore contents of X19 register:
    ldr     x19, [sp,16]
; restore FP and LR values:
    ldp     x29, x30, [sp], 32
    ret
.LC0:
    .string "f(%d)\n"
```

#### ARM64: Non-optimizing GCC 4.9.1

Listing 14.8: Non-optimizing GCC 4.9.1 -fno-inline

```
printing_function:
; prepare second argument of printf():
    mov     w1, w0
; load address of the "f(%d)\n" string
    adrp    x0, .LC0
    add     x0, x0, :lo12:LC0
; just branch here instead of branch with link and return:
    b       printf
main:
; save FP and LR in the local stack:
```

```

        stp    x29, x30, [sp, -32]!
; set up stack frame:
        add    x29, sp, 0
; save contents of X19 register in the local stack:
        str    x19, [sp,16]
; we will use W19 register as counter.
; set initial value of 2 to it:
        mov    w19, 2
.L3:
; prepare first argument of printing_function():
        mov    w0, w19
; increment counter register.
        add    w19, w19, 1
; W0 here still holds value of counter value before increment.
        bl     printing_function
; is it end?
        cmp    w19, 10
; no, jump to the loop body begin:
        bne    .L3
; return 0
        mov    w0, 0
; restore contents of X19 register:
        ldr    x19, [sp,16]
; restore FP and LR values:
        ldp    x29, x30, [sp], 32
        ret
.LC0:
        .string "f(%d)\n"

```

## 14.1.5 MIPS

Listing 14.9: Non-optimizing GCC 4.4.5 (IDA)

```

main:
; IDA is not aware of variable names in local stack
; We gave them names manually:
i      = -0x10
saved_FP = -8
saved_RA = -4

; function prologue:
        addiu   $sp, -0x28
        sw      $ra, 0x28+saved_RA($sp)
        sw      $fp, 0x28+saved_FP($sp)
        move    $fp, $sp
; initialize counter at 2 and store this value in local stack
        li      $v0, 2
        sw      $v0, 0x28+i($fp)
; pseudoinstruction. "BEQ $ZERO, $ZERO, loc_9C" there in fact:
        b       loc_9C
        or      $at, $zero ; branch delay slot, NOP
# -----

loc_80:                                     # CODE XREF: main+48
; load counter value from local stack and call printing_function():
        lw      $a0, 0x28+i($fp)
        jal     printing_function
        or      $at, $zero ; branch delay slot, NOP
; load counter, increment it, store it back:
        lw      $v0, 0x28+i($fp)
        or      $at, $zero ; NOP
        addiu   $v0, 1
        sw      $v0, 0x28+i($fp)

loc_9C:                                     # CODE XREF: main+18
; check counter, is it 10?
        lw      $v0, 0x28+i($fp)

```

```

        or      $at, $zero ; NOP
        slti    $v0, 0xA
; if it is less than 10, jump to loc_80 (loop body begin):
        bnez    $v0, loc_80
        or      $at, $zero ; branch delay slot, NOP
; finishing, return 0:
        move    $v0, $zero
; function epilogue:
        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

```

The instruction that's new to us is “B”. It is actually the pseudoinstruction (BEQ).

### 14.1.6 One more thing

In the generated code we can see: after initializing *i*, the body of the loop is not to be executed, as the condition for *i* is checked first, and only after that loop body can be executed. And that is correct. Because, if the loop condition is not met at the beginning, the body of the loop must not be executed. This is possible in the following case:

```

for (i=0; i<total_entries_to_process; i++)
    loop_body;

```

If *total\_entries\_to\_process* is 0, the body of the loop must not be executed at all. This is why the condition is checked before the execution.

However, an optimizing compiler may swap the condition check and loop body, if it is sure that the situation described here is not possible (like in the case of our very simple example and Keil, Xcode (LLVM), MSVC in optimization mode).

## 14.2 Memory blocks copying routine

Real-world memory copy routines may copy 4 or 8 bytes at each iteration, use [SIMD<sup>2</sup>](#), vectorization, etc. But for the sake of simplicity, this example is the simplest possible.

```

#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];
};

```

### 14.2.1 Straight-forward implementation

Listing 14.10: GCC 4.9 x64 optimized for size (-Os)

```

my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block

; initialize counter (i) at 0
    xor     eax, eax
.L2:
; all bytes copied? exit then:
    cmp     rax, rdx
    je      .L5
; load byte at RSI+i:
    mov     cl, BYTE PTR [rsi+rax]

```

<sup>2</sup>Single instruction, multiple data

```

; store byte at RDI+i:
    mov     BYTE PTR [rdi+rax], cl
    inc     rax ; i++
    jmp     .L2
.L5:
    ret

```

Listing 14.11: GCC 4.9 ARM64 optimized for size (-Os)

```

my_memcpy:
; X0 = destination address
; X1 = source address
; X2 = size of block

; initialize counter (i) at 0
    mov     x3, 0
.L2:
; all bytes copied? exit then:
    cmp     x3, x2
    beq     .L5
; load byte at X1+i:
    ldrb     w4, [x1,x3]
; store byte at X1+i:
    strb     w4, [x0,x3]
    add     x3, x3, 1 ; i++
    b       .L2
.L5:
    ret

```

Listing 14.12: Optimizing Keil 6/2013 (Thumb mode)

```

my_memcpy PROC
; R0 = destination address
; R1 = source address
; R2 = size of block

    PUSH     {r4,lr}
; initialize counter (i) at 0
    MOVS     r3,#0
; condition checked at the end of function, so jump there:
    B        |L0.12|
|L0.6|
; load byte at R1+i:
    LDRB     r4,[r1,r3]
; store byte at R1+i:
    STRB     r4,[r0,r3]
; i++
    ADDS     r3,r3,#1
|L0.12|
; i<size?
    CMP      r3,r2
; jump to the loop begin if its so:
    BCC      |L0.6|
    POP      {r4,pc}
ENDP

```

## 14.2.2 ARM in ARM mode

Keil in ARM mode takes full advantage of conditional suffixes:

Listing 14.13: Optimizing Keil 6/2013 (ARM mode)

```

my_memcpy PROC
; R0 = destination address
; R1 = source address
; R2 = size of block

; initialize counter (i) at 0

```

```

        MOV     r3,#0
|L0.4|
; all bytes copied?
        CMP     r3,r2
; the following block is executed only if "less than" condition,
; i.e., if R2<R3 or i<size.
; load byte at R1+i:
        LDRBCC  r12,[r1,r3]
; store byte at R1+i:
        STRBCC  r12,[r0,r3]
; i++
        ADDCC   r3,r3,#1
; the last instruction of the "conditional block".
; jump to loop begin if i<size
; do nothing otherwise (i.e., if i>=size)
        BCC     |L0.4|
; return
        BX      lr
        ENDP

```

That's why there is only one branch instruction instead of 2.

### 14.2.3 MIPS

Listing 14.14: GCC 4.4.5 optimized for size (-Os) (IDA)

```

my_memcpy:
; jump to loop check part:
        b       loc_14
; initialize counter (i) at 0
; it will always reside in $v0:
        move    $v0, $zero ; branch delay slot

loc_8:
; load byte as unsigned at address in $t0 to $v1:
        lbu     $v1, 0($t0)
; increment counter (i):
        addiu   $v0, 1
; store byte at $a3
        sb      $v1, 0($a3)

loc_14:
; check if counter (i) in $v0 is still less then 3rd function argument ("cnt" in $a2):
        sltu    $v1, $v0, $a2
; form address of byte in source block:
        addu    $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; jump to loop body if counter sill less then "cnt":
        bnez    $v1, loc_8
; form address of byte in destination block ($a3 = $a0+$v0 = dst+i):
        addu    $a3, $a0, $v0 ; branch delay slot
; finish if BNEZ wasnt triggered:
        jr      $ra
        or      $at, $zero ; branch delay slot, NOP

```

Here we have two new instructions: LBU (“Load Byte Unsigned”) and SB (“Store Byte”). Just like in ARM, all MIPS registers are 32-bit wide, there are no byte-wide parts like in x86. So when dealing with single bytes, we have to allocate whole 32-bit registers for them. LBU loads a byte and clears all other bits (“Unsigned”). On the other hand, LB (“Load Byte”) instruction sign-extends the loaded byte to a 32-bit value. SB just writes a byte from lowest 8 bits of register to memory.

### 14.2.4 Vectorization

Optimizing GCC can do much more on this example: [25.1.2 on page 397](#).



## 14.3 Conclusion

Rough skeleton of loop from 2 to 9 inclusive:

Listing 14.15: x86

```

mov [counter], 2 ; initialization
jmp check
body:
; loop body
; do something here
; use counter variable in local stack
add [counter], 1 ; increment
check:
cmp [counter], 9
jle body

```

The increment operation may be represented as 3 instructions in non-optimized code:

Listing 14.16: x86

```

MOV [counter], 2 ; initialization
JMP check
body:
; loop body
; do something here
; use counter variable in local stack
MOV REG, [counter] ; increment
INC REG
MOV [counter], REG
check:
CMP [counter], 9
JLE body

```

If the body of the loop is short, a whole register can be dedicated to the counter variable:

Listing 14.17: x86

```

MOV EBX, 2 ; initialization
JMP check
body:
; loop body
; do something here
; use counter in EBX, but do not modify it!
INC EBX ; increment
check:
CMP EBX, 9
JLE body

```

Some parts of the loop may be generated by compiler in different order:

Listing 14.18: x86

```

MOV [counter], 2 ; initialization
JMP label_check
label_increment:
ADD [counter], 1 ; increment
label_check:
CMP [counter], 10
JGE exit
; loop body
; do something here
; use counter variable in local stack
JMP label_increment
exit:

```

Usually the condition is checked *before* loop body, but the compiler may rearrange it in a way that the condition is checked *after* loop body. This is done when the compiler is sure that the condition is always *true* on the first iteration, so the body of the loop is to be executed at least once:

Listing 14.19: x86

```
    MOV REG, 2 ; initialization
body:
    ; loop body
    ; do something here
    ; use counter in REG, but do not modify it!
    INC REG ; increment
    CMP REG, 10
    JL body
```

Using the `LOOP` instruction. This is rare, compilers are not using it. When you see it, it's a sign that this piece of code is hand-written:

Listing 14.20: x86

```
    ; count from 10 to 1
    MOV ECX, 10
body:
    ; loop body
    ; do something here
    ; use counter in ECX, but do not modify it!
    LOOP body
```

ARM. The `R4` register is dedicated to counter variable in this example:

Listing 14.21: ARM

```
    MOV R4, 2 ; initialization
    B check
body:
    ; loop body
    ; do something here
    ; use counter in R4, but do not modify it!
    ADD R4,R4, #1 ; increment
check:
    CMP R4, #10
    BLT body
```

## 14.4 Exercises

- <http://challenges.re/54>
- <http://challenges.re/55>
- <http://challenges.re/56>
- <http://challenges.re/57>

## Chapter 15

# Simple C-strings processing

### 15.1 strlen()

Let's talk about loops one more time. Often, the `strlen()` function<sup>1</sup> is implemented using a `while()` statement. Here is how it is done in the MSVC standard libraries:

```
int my_strlen (const char * str)
{
    const char *eos = str;

    while( *eos++ ) ;

    return( eos - str - 1 );
}

int main()
{
    // test
    return my_strlen("hello!");
};
```

#### 15.1.1 x86

##### Non-optimizing MSVC

Let's compile:

```
_eos$ = -4 ; size = 4
_str$ = 8 ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp] ; place pointer to string from "str"
    mov     DWORD PTR _eos$[ebp], eax ; place it to local variable "eos"
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp] ; ECX=eos

    ; take 8-bit byte from address in ECX and place it as 32-bit value to EDX with sign extension

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp] ; EAX=eos
    add     eax, 1 ; increment EAX
    mov     DWORD PTR _eos$[ebp], eax ; place EAX back to "eos"
    test    edx, edx ; EDX is zero?
    je      SHORT $LN1@strlen_ ; yes, then finish loop
    jmp     SHORT $LN2@strlen_ ; continue loop
$LN1@strlen_:
```

<sup>1</sup>counting the characters in a string in the C language

```

; here we calculate the difference between two pointers

mov     eax, DWORD PTR _eos$[ebp]
sub     eax, DWORD PTR _str$[ebp]
sub     eax, 1                      ; subtract 1 and return result
mov     esp, ebp
pop     ebp
ret     0
_strlen_ ENDP

```

We get two new instructions here: `MOVSX` and `TEST`.

The first one—`MOVSX`—takes a byte from an address in memory and stores the value in a 32-bit register. `MOVSX` stands for *MOV with Sign-Extend*. `MOVSX` sets the rest of the bits, from the 8th to the 31st, to 1 if the source byte is *negative* or to 0 if is *positive*.

And here is why.

By default, the *char* type is signed in MSVC and GCC. If we have two values of which one is *char* and the other is *int*, (*int* is signed too), and if the first value contain -2 (coded as 0xFE) and we just copy this byte into the *int* container, it makes 0x000000FE, and this from the point of signed *int* view is 254, but not -2. In signed *int*, -2 is coded as 0xFFFFFEE. So if we need to transfer 0xFE from a variable of *char* type to *int*, we need to identify its sign and extend it. That is what `MOVSX` does.

You can also read about it in “*Signed number representations*” section ( [30 on page 432](#)).

It’s hard to say if the compiler needs to store a *char* variable in `EDX`, it could just take a 8-bit register part (for example `DL`). Apparently, the compiler’s [register allocator](#) works like that.

Then we see `TEST EDX, EDX`. You can read more about the `TEST` instruction in the section about bit fields ( [19 on page 290](#)). Here this instruction just checks if the value in `EDX` equals to 0.

## Non-optimizing GCC

Let’s try GCC 4.4.1:

```

strlen      public strlen
proc near
eos
arg_0       = dword ptr -4
            = 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

```

The result is almost the same as in MSVC, but here we see `MOVZX` instead of `MOVSX`. `MOVZX` stands for *MOV with Zero-Extend*. This instruction copies a 8-bit or 16-bit value into a 32-bit register and sets the rest of the bits to 0. In fact, this instruction is convenient only because it enable us to replace this instruction pair: `xor eax, eax / mov al, [...]`.

On the other hand, it is obvious that the compiler could produce this code: `mov al, byte ptr [eax] / test al, al`—it is almost the same, however, the highest bits of the EAX register will contain random noise. But let's think it is compiler's drawback—it cannot produce more understandable code. Strictly speaking, the compiler is not obliged to emit understandable (to humans) code at all.

The next new instruction for us is `SETNZ`. Here, if AL doesn't contain zero, `test al, al` sets the ZF flag to 0, but `SETNZ`, if `ZF==0` (NZ stands for *not zero*) sets AL to 1. Speaking in natural language, *if AL is not zero, let's jump to loc\_80483F0*. The compiler emits some redundant code, but let's not forget that the optimizations are turned off.

## Optimizing MSVC

Now let's compile all this in MSVC 2012, with optimizations turned on (/Ox):

Listing 15.1: Optimizing MSVC 2012 /Ob0

```
_str$ = 8 ; size = 4
_strlen PROC
    mov     edx, DWORD PTR _str$[esp-4] ; EDX -> pointer to the string
    mov     eax, edx ; move to EAX
$LL2@strlen:
    mov     cl, BYTE PTR [eax] ; CL = *EAX
    inc     eax ; EAX++
    test    cl, cl ; CL==0?
    jne     SHORT $LL2@strlen ; no, continue loop
    sub     eax, edx ; calculate pointers difference
    dec     eax ; decrement EAX
    ret     0
_strlen ENDP
```

Now it is all simpler. Needless to say, the compiler could use registers with such efficiency only in small functions with a few local variables.

`INC/DEC`— are [increment/decrement](#) instructions, in other words: add or subtract 1 to/from a variable.

## Optimizing MSVC + OllyDbg

We can try this (optimized) example in OllyDbg. Here is the first iteration:

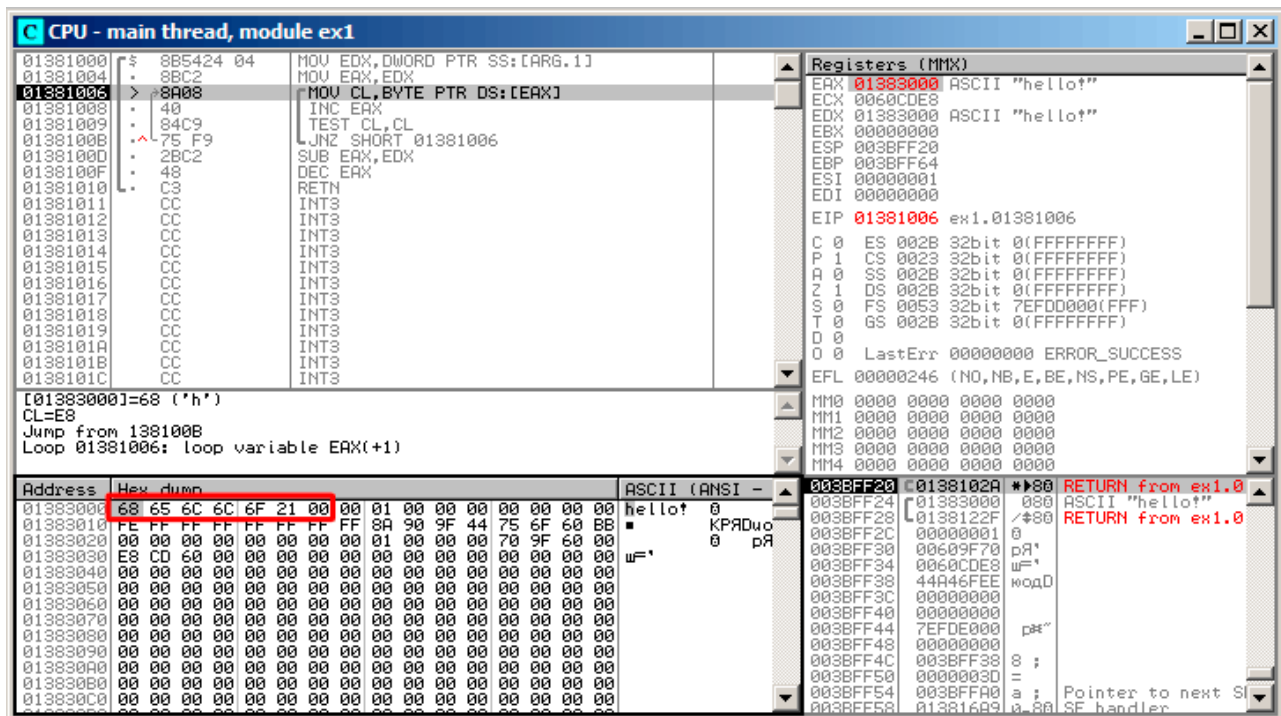


Figure 15.1: OllyDbg: first iteration start

We see that OllyDbg found a loop and, for convenience, *wrapped* its instructions in brackets. By clicking the right button on EAX, we can choose “Follow in Dump” and the memory window scrolls to the right place. Here we can see the string “hello!” in memory. There is at least one zero byte after it and then random garbage. If OllyDbg sees a register with a valid address in it, that points to some string, it is shown as a string.

Let's press F8 (step over) a few times, to get to the start of the body of the loop:

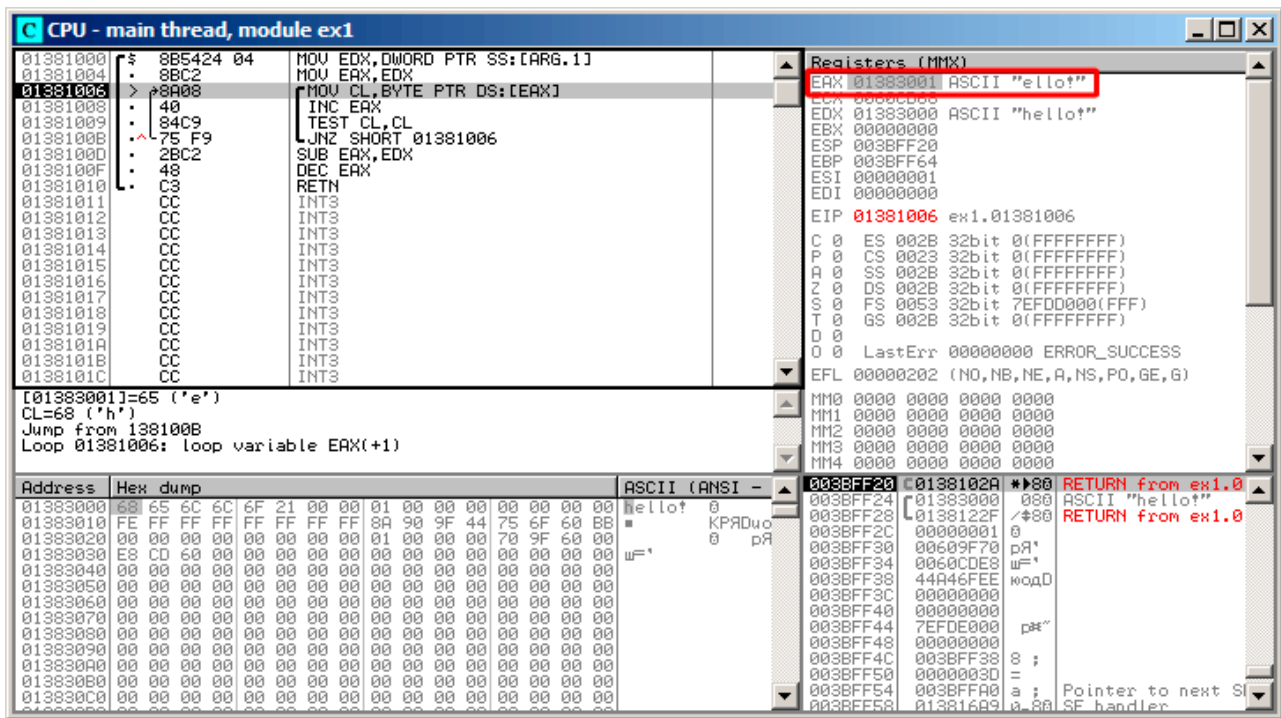


Figure 15.2: OllyDbg: second iteration start

We see that EAX contains the address of the second character in the string.

We have to press F8 enough number of times in order to escape from the loop:

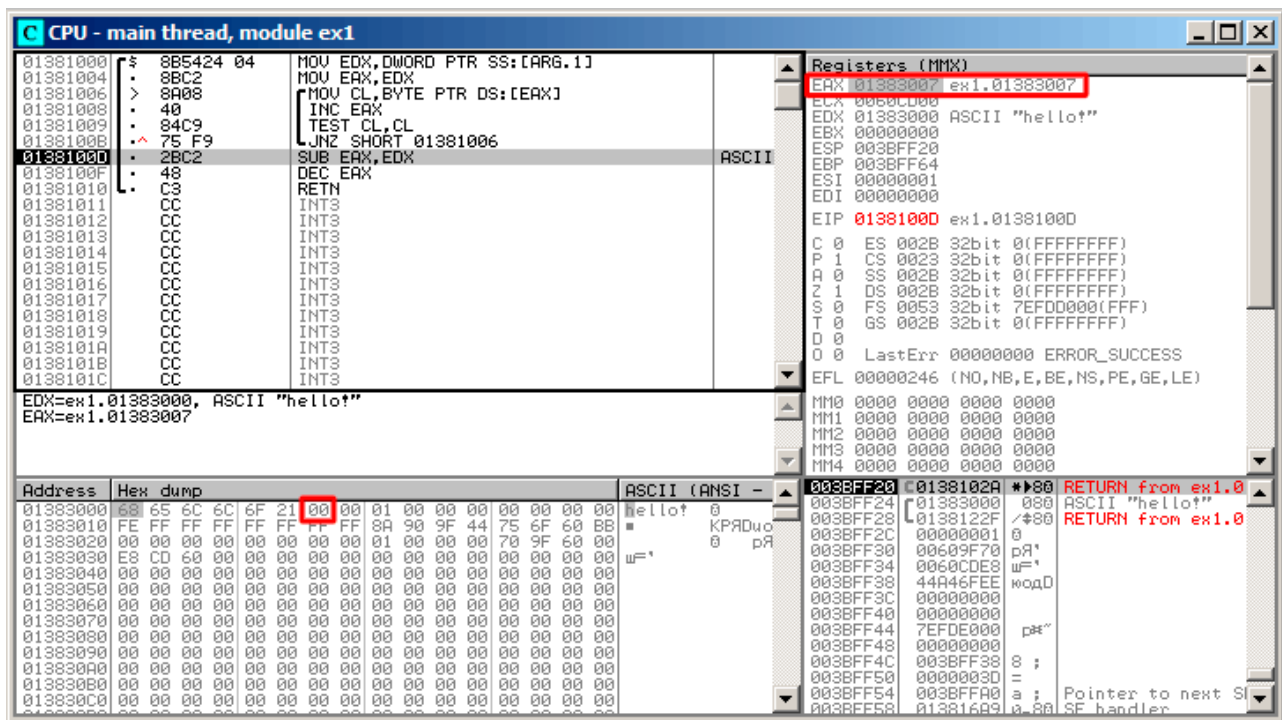


Figure 15.3: OllyDbg: pointers difference to be calculated now

We see that EAX now contains the address of zero byte that's right after the string. Meanwhile, EDX hasn't changed, so it still pointing to the start of the string. The difference between these two addresses is being calculated now.



The SUB instruction just got executed:

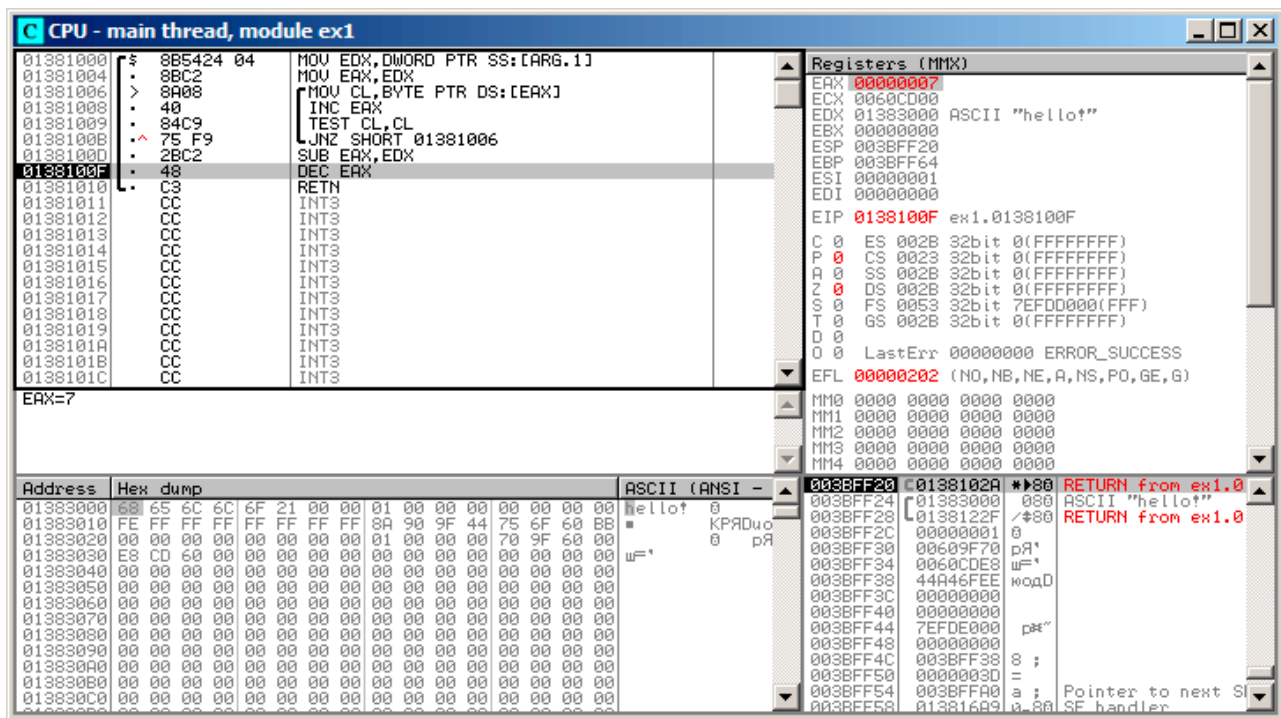


Figure 15.4: OllyDbg: EAX to be decremented now

The difference of pointers is in the EAX register now—7. Indeed, the length of the “hello!” string is 6, but with the zero byte included—7. But `strlen()` must return the number of non-zero characters in the string. So the decrement executes and then the function returns.

## Optimizing GCC

Let’s check GCC 4.4.1 with optimizations turned on (-O3 key):

```

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

```

Here GCC is almost the same as MSVC, except for the presence of `MOVZX`.

However, here `MOVZX` could be replaced with `mov dl, byte ptr [eax]`.

Probably it is simpler for GCC’s code generator to *remember* the whole 32-bit EDX register is allocated for a *char* variable and it then can be sure that the highest bits has no any noise at any point.

After that we also see a new instruction—`NOT`. This instruction inverts all bits in the operand. You can say that it is a synonym to the `XOR ECX, 0xffffffffh` instruction. `NOT` and the following `ADD` calculate the pointer difference and subtract 1, just in a different way. At the start `ECX`, where the pointer to *str* is stored, gets inverted and 1 is subtracted from it.

See also: “Signed number representations” ( [30 on page 432](#)).

In other words, at the end of the function just after loop body, these operations are executed:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

... and this is effectively equivalent to:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Why did GCC decide it would be better? Hard to guess. But perhaps the both variants are equivalent in efficiency.

## 15.1.2 ARM

### 32-bit ARM

#### Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 15.2: Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
_strlen
eos  = -8
str  = -4

    SUB    SP, SP, #8 ; allocate 8 bytes for local variables
    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 ; free allocated 8 bytes
    BX     LR
```

Non-optimizing LLVM generates too much code, however, here we can see how the function works with local variables in the stack. There are only two local variables in our function: *eos* and *str*. In this listing, generated by [IDA](#), we have manually renamed *var\_8* and *var\_4* to *eos* and *str*.

The first instructions just saves the input values into both *str* and *eos*.

The body of the loop starts at label *loc\_2CB8*.

The first three instructions in the loop body (LDR, ADD, STR) load the value of *eos* into R0. Then the value is [incremented](#) and saved back into *eos*, which is located in the stack.

The next instruction, `LDRSB R0, [R0]` (“Load Register Signed Byte”), loads a byte from memory at the address stored in R0 and sign-extends it to 32-bit<sup>2</sup>. This is similar to the `MOVSX` instruction in x86. The compiler treats this byte as signed

<sup>2</sup>The Keil compiler treats the *char* type as signed, just like MSVC and GCC.

since the *char* type is signed according to the C standard. It was already written about it ([15.1.1 on page 190](#)) in this section, in relation to x86.

It has to be noted that it is impossible to use 8- or 16-bit part of a 32-bit register in ARM separately of the whole register, as it is in x86. Apparently, it is because x86 has a huge history of backwards compatibility with its ancestors up to the 16-bit 8086 and even 8-bit 8080, but ARM was developed from scratch as a 32-bit RISC-processor. Consequently, in order to process separate bytes in ARM, one has to use 32-bit registers anyway.

So, `LDRSB` loads bytes from the string into `R0`, one by one. The following `CMP` and `BEQ` instructions check if the loaded byte is 0. If it's not 0, control passes to the start of the body of the loop. And if it's 0, the loop ends.

At the end of the function, the difference between *eos* and *str* is calculated, 1 is subtracted from it, and resulting value is returned via `R0`.

N.B. Registers were not saved in this function. That's because in the ARM calling convention registers `R0-R3` are “scratch registers”, intended for arguments passing, and we're not required to restore their value when the function exits, since the calling function will not use them anymore. Consequently, they may be used for anything we want. No other registers are used here, so that is why we have nothing to save on the stack. Thus, control may be returned back to calling function by a simple jump (`BX`), to the address in the `LR` register.

### Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

Listing 15.3: Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

```
_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
```

As optimizing LLVM concludes, *eos* and *str* do not need space on the stack, and can always be stored in registers. Before the start of the loop body, *str* is always in `R0`, and *eos*—in `R1`.

The `LDRB.W R2, [R1], #1` instruction loads a byte from the memory at the address stored in `R1`, to `R2`, sign-extending it to a 32-bit value, but not just that. `#1` at the instruction's end implies “Post-indexed addressing”, which means that 1 is to be added to `R1` after the byte is loaded.

Read more about it: [28.2 on page 425](#).

Then you can see `CMP` and `BNE`<sup>3</sup> in the body of the loop, these instructions continue looping until 0 is found in the string.

`MVNS`<sup>4</sup> (inverts all bits, like `NOT` in x86) and `ADD` instructions compute  $eos - str - 1$ . In fact, these two instructions compute  $R0 = str + eos$ , which is effectively equivalent to what was in the source code, and why it is so, was already explained here ([15.1.1 on page 195](#)).

Apparently, LLVM, just like GCC, concludes that this code can be shorter (or faster).

### Optimizing Keil 6/2013 (ARM mode)

Listing 15.4: Optimizing Keil 6/2013 (ARM mode)

```
_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
```

<sup>3</sup>(PowerPC, ARM) Branch if Not Equal

<sup>4</sup>MoVe Not

Almost the same as what we saw before, with the exception that the `str - eos - 1` expression can be computed not at the function's end, but right in the body of the loop. The `-EQ` suffix, as we may recall, implies that the instruction executes only if the operands in the `CMP` that was executed before were equal to each other. Thus, if `R0` contains 0, both `SUBEQ` instructions executes and result is left in the `R0` register.

## ARM64

### Optimizing GCC (Linaro) 4.9

```
my_strlen:
    mov     x1, x0
    ; X1 is now temporary pointer (eos), acting like cursor
.L58:
    ; load byte from X1 to W2, increment X1 (post-index)
    ldrb    w2, [x1],1
    ; Compare and Branch if NonZero: compare W2 with 0, jump to .L58 if it is not
    cbnz    w2, .L58
    ; calculate difference between initial pointer in X0 and current address in X1
    sub     x0, x1, x0
    ; decrement lowest 32-bit of result
    sub     w0, w0, #1
    ret
```

The algorithm is the same as in [15.1.1 on page 191](#): find a zero byte, calculate the difference between the pointers and decrement the result by 1. Some comments were added by the author of this book.

The only thing worth noting is that our example is somewhat wrong: `my_strlen()` returns 32-bit `int`, while it has to return `size_t` or another 64-bit type.

The reason is that, theoretically, `strlen()` can be called for a huge blocks in memory that exceeds 4GB, so it must able to return a 64-bit value on 64-bit platforms. Because of my mistake, the last `SUB` instruction operates on a 32-bit part of register, while the penultimate `SUB` instruction works on full the 64-bit register (it calculates the difference between the pointers). It's my mistake, it is better to leave it as is, as an example of how the code could look like in such case.

### Non-optimizing GCC (Linaro) 4.9

```
my_strlen:
; function prologue
    sub     sp, sp, #32
; first argument (str) will be stored in [sp,8]
    str     x0, [sp,8]
    ldr     x0, [sp,8]
; copy "str" to "eos" variable
    str     x0, [sp,24]
    nop
.L62:
; eos++
    ldr     x0, [sp,24] ; load "eos" to X0
    add     x1, x0, 1   ; increment X0
    str     x1, [sp,24] ; save X0 to "eos"
; load byte from memory at address in X0 to W0
    ldrb    w0, [x0]
; is it zero? (WZR is the 32-bit register always contain zero)
    cmp     w0, wzr
; jump if not zero (Branch Not Equal)
    bne     .L62
; zero byte found. now calculate difference.
; load "eos" to X1
    ldr     x1, [sp,24]
; load "str" to X0
    ldr     x0, [sp,8]
; calculate difference
    sub     x0, x1, x0
; decrement result
    sub     w0, w0, #1
; function epilogue
```

```
add    sp, sp, 32
ret
```

It's more verbose. The variables are often tossed here to and from memory (local stack). The same mistake here: the decrement operation happens on a 32-bit register part.

### 15.1.3 MIPS

Listing 15.5: Optimizing GCC 4.4.5 (IDA)

```
my_strlen:
; "eos" variable will always reside in $v1:
        move    $v1, $a0

loc_4:
; load byte at address in "eos" into $a1:
        lb      $a1, 0($v1)
        or      $at, $zero ; load delay slot, NOP
; if loaded byte is not zero, jump to loc_4:
        bnez    $a1, loc_4
; increment "eos" anyway:
        addiu   $v1, 1 ; branch delay slot
; loop finished. invert "str" variable:
        nor     $v0, $zero, $a0
; $v0=-str-1
        jr      $ra
; return value = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
        addu    $v0, $v1, $v0 ; branch delay slot
```

MIPS lacks a NOT instruction, but has NOR which is OR + NOT operation. This operation is widely used in digital electronics<sup>5</sup>, but isn't very popular in computer programming. So, the NOT operation is implemented here as NOR DST, \$ZERO, SRC.

From fundamentals [30 on page 432](#) we know that bitwise inverting a signed number is the same as changing its sign and subtracting 1 from the result. So what NOT does here is to take the value of *str* and transform it into  $-str - 1$ . The addition operation that follows prepares result.

<sup>5</sup>NOR is called "universal gate". For example, the Apollo Guidance Computer used in the Apollo program, was built by only using 5600 NOR gates: [\[Eic11\]](#).

## Chapter 16

# Replacing arithmetic instructions to other ones

In the pursuit of optimization, one instruction may be replaced by another, or even with a group of instructions. For example, ADD and SUB can replace each other: line 18 in listing.52.1.

For example, the LEA instruction is often used for simple arithmetic calculations: [A.6.2 on page 886](#).

## 16.1 Multiplication

### 16.1.1 Multiplication using addition

Here is a simple example:

Listing 16.1: Optimizing MSVC 2010

```
unsigned int f(unsigned int a)
{
    return a*8;
};
```

Multiplication by 8 is replaced by 3 addition instructions, which do the same. Apparently, MSVC's optimizer decided that this code can be faster.

```
_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
```

### 16.1.2 Multiplication using shifting

Multiplication and division instructions by a numbers that's a power of 2 are often replaced by shift instructions.

```
unsigned int f(unsigned int a)
{
    return a*4;
};
```

Listing 16.2: Non-optimizing MSVC 2010

```
_a$ = 8 ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
```

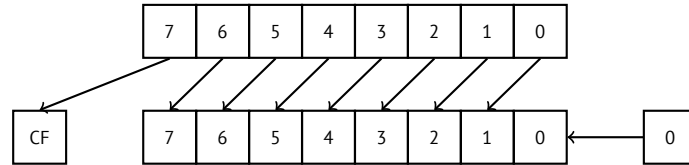
```

mov    eax, DWORD PTR _a$[ebp]
shl    eax, 2
pop    ebp
ret    0
_f     ENDP

```

Multiplication by 4 is just shifting the number to the left by 2 bits and inserting 2 zero bits at the right (as the last two bits). It is just like multiplying 3 by 100 –we need to just add two zeroes at the right.

That's how the shift left instruction works:



The added bits at right are always zeroes.

Multiplication by 4 in ARM:

Listing 16.3: Non-optimizing Keil 6/2013 (ARM mode)

```

f PROC
    LSL    r0,r0,#2
    BX     lr
ENDP

```

Multiplication by 4 in MIPS:

Listing 16.4: Optimizing GCC 4.4.5 (IDA)

```

jr      $ra
sll     $v0, $a0, 2 ; branch delay slot

```

SLL is “Shift Left Logical”.

### 16.1.3 Multiplication using shifting, subtracting, and adding

It's still possible to get rid of the multiplication operation when you multiply by numbers like 7 or 17 again by using shifting. The mathematics used here is relatively easy.

#### 32-bit

```

#include <stdint.h>

int f1(int a)
{
    return a*7;
};

int f2(int a)
{
    return a*28;
};

int f3(int a)
{
    return a*17;
};

```

#### x86

## Listing 16.5: Optimizing 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 for ARM mode takes advantage of the second operand's shift modifiers:

## Listing 16.6: Optimizing Keil 6/2013 (ARM mode)

```

; 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
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
    BX     lr
    ENDP

```



But there are no such modifiers in Thumb mode. It also can't optimize `f2()`:

Listing 16.7: Optimizing Keil 6/2013 (Thumb mode)

```
; a*7
||f1|| PROC
    LSLS    r1,r0,#3
; R1=R0<<3=a<<3=a*8
    SUBS    r0,r1,r0
; R0=R1-R0=a*8-a=a*7
    BX      lr
    ENDP

; a*28
||f2|| PROC
    MOVS    r1,#0x1c ; 28
; R1=28
    MULS    r0,r1,r0
; R0=R1*R0=28*a
    BX      lr
    ENDP

; a*17
||f3|| PROC
    LSLS    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

Listing 16.8: Optimizing 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-bit

```
#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**

Listing 16.9: Optimizing 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 for ARM64 is also terse, thanks to the shift modifiers:

Listing 16.10: Optimizing 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:
    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
```

## 16.2 Division

### 16.2.1 Division using shifts

Example of division by 4:

```
unsigned int f(unsigned int a)
{
    return a/4;
};
```

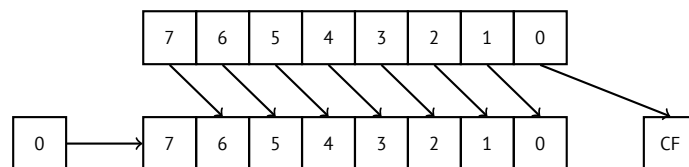
We get (MSVC 2010):

Listing 16.11: MSVC 2010

```
_a$ = 8 ; size = 4
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    shr     eax, 2
    ret     0
_f ENDP
```

The SHR (*SH*ift *R*ight) instruction in this example is shifting a number by 2 bits to the right. The two freed bits at left (e.g., two most significant bits) are set to zero. The two least significant bits are dropped. In fact, these two dropped bits are the division operation remainder.

The SHR instruction works just like SHL, but in the other direction.



It is easy to understand if you imagine the number 23 in the decimal numeral system. 23 can be easily divided by 10 just by dropping last digit (3—division remainder). 2 is left after the operation as a [quotient](#).

So the remainder is dropped, but that's OK, we work on integer values anyway, these are not a [real numbers](#)!

Division by 4 in ARM:

Listing 16.12: Non-optimizing Keil 6/2013 (ARM mode)

```
f PROC
    LSR     r0, r0, #2
    BX      lr
    ENDP
```

Division by 4 in MIPS:

Listing 16.13: Optimizing GCC 4.4.5 (IDA)

```
jr    $ra
srl   $v0, $a0, 2 ; branch delay slot
```

The SRL instruction is “Shift Right Logical”.

## 16.3 Exercise

- <http://challenges.re/59>

## Chapter 17

# Floating-point unit

The **FPU** is a device within the main **CPU**, specially designed to deal with floating point numbers. It was called “coprocessor” in the past and it stays somewhat aside of the main **CPU**.

### 17.1 IEEE 754

A number in the IEEE 754 format consists of a *sign*, a *significand* (also called *fraction*) and an *exponent*.

### 17.2 x86

It is worth looking into stack machines<sup>1</sup> or learning the basics of the Forth language<sup>2</sup>, before studying the **FPU** in x86.

It is interesting to know that in the past (before the 80486 CPU) the coprocessor was a separate chip and it was not always pre-installed on the motherboard. It was possible to buy it separately and install it<sup>3</sup>. Starting with the 80486 DX CPU, the **FPU** is integrated in the **CPU**.

The FWAIT instruction reminds us of that fact—it switches the **CPU** to a waiting state, so it can wait until the **FPU** is done with its work. Another rudiment is the fact that the **FPU** instruction opcodes start with the so called “escape”-opcodes (D8 . . DF), i.e., opcodes passed to a separate coprocessor.

The FPU has a stack capable to holding 8 80-bit registers, and each register can hold a number in the IEEE 754<sup>4</sup> format. They are ST(0)..ST(7). For brevity, IDA and OllyDbg show ST(0) as ST, which is represented in some textbooks and manuals as “Stack Top”.

### 17.3 ARM, MIPS, x86/x64 SIMD

In ARM and MIPS the FPU is not a stack, but a set of registers. The same ideology is used in the SIMD extensions of x86/x64 CPUs.

### 17.4 C/C++

The standard C/C++ languages offer at least two floating number types, *float* (*single-precision*<sup>5</sup>, 32 bits)<sup>6</sup> and *double* (*double-precision*<sup>7</sup>, 64 bits).

GCC also supports the *long double* type (*extended precision*<sup>8</sup>, 80 bit), which MSVC doesn’t.

<sup>1</sup>[wikipedia.org/wiki/Stack\\_machine](http://wikipedia.org/wiki/Stack_machine)

<sup>2</sup>[wikipedia.org/wiki/Forth\\_\(programming\\_language\)](http://wikipedia.org/wiki/Forth_(programming_language))

<sup>3</sup>For example, John Carmack used fixed-point arithmetic ([wikipedia.org/wiki/Fixed-point\\_arithmetic](http://wikipedia.org/wiki/Fixed-point_arithmetic)) values in his Doom video game, stored in 32-bit GPR registers (16 bit for integral part and another 16 bit for fractional part), so Doom could work on 32-bit computers without FPU, i.e., 80386 and 80486 SX.

<sup>4</sup>[wikipedia.org/wiki/IEEE\\_floating\\_point](http://wikipedia.org/wiki/IEEE_floating_point)

<sup>5</sup>[wikipedia.org/wiki/Single-precision\\_floating-point\\_format](http://wikipedia.org/wiki/Single-precision_floating-point_format)

<sup>6</sup>the single precision floating point number format is also addressed in the *Working with the float type as with a structure* ( 21.6.2 on page 356) section

<sup>7</sup>[wikipedia.org/wiki/Double-precision\\_floating-point\\_format](http://wikipedia.org/wiki/Double-precision_floating-point_format)

<sup>8</sup>[wikipedia.org/wiki/Extended\\_precision](http://wikipedia.org/wiki/Extended_precision)

The *float* type requires the same number of bits as the *int* type in 32-bit environments, but the number representation is completely different.

## 17.5 Simple example

Let's consider this simple example:

```
#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));
};
```

### 17.5.1 x86

#### MSVC

Compile it in MSVC 2010:

Listing 17.1: 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]

; current stack state: ST(0) = _a

    fdiv    QWORD PTR __real@40091eb851eb851f

; current stack state: ST(0) = result of _a divided by 3.14

    fld     QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b; ST(1) = result of _a divided by 3.14

    fmul    QWORD PTR __real@4010666666666666

; current stack state:
; ST(0) = result of _b * 4.1;
; ST(1) = result of _a divided by 3.14

    faddp   ST(1), ST(0)

; current stack state: ST(0) = result of addition

    pop     ebp
    ret     0
_f ENDP
```

FLD takes 8 bytes from stack and loads the number into the ST(0) register, automatically converting it into the internal 80-bit format (*extended precision*).

FDIV divides the value in ST(0) by the number stored at address `__real@40091eb851eb851f` —the value 3.14 is encoded there. The assembly syntax doesn't support floating point numbers, so what we see here is the hexadecimal representation of 3.14 in 64-bit IEEE 754 format.

After the execution of FDIV ST(0) holds the [quotient](#).

By the way, there is also the FDIVP instruction, which divides ST(1) by ST(0), popping both these values from stack and then pushing the result. If you know the Forth language<sup>9</sup>, you can quickly understand that this is a stack machine<sup>10</sup>.

The subsequent FLD instruction pushes the value of *b* into the stack.

After that, the quotient is placed in ST(1), and ST(0) has the value of *b*.

The next FMUL instruction does multiplication: *b* from ST(0) is multiplied by by value at `__real@4010666666666666` (the numer 4.1 is there) and leaves the result in the ST(0) register.

The last FADDP instruction adds the two values at top of stack, storing the result in ST(1) and then popping the value of ST(0), thereby leaving the result at the top of the stack, in ST(0).

The function must return its result in the ST(0) register, so there are no any other instructions except the function epilogue after FADDP.

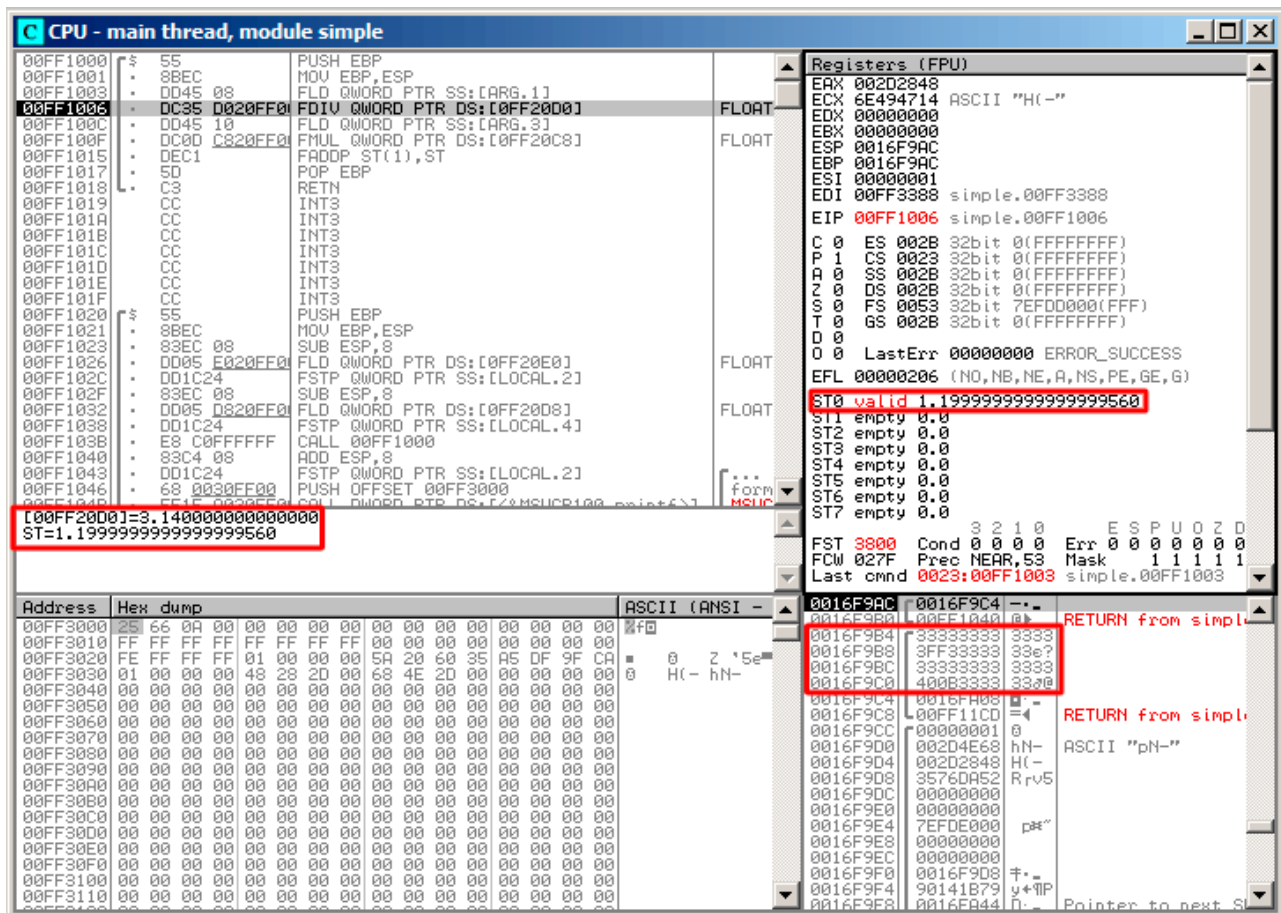
---

<sup>9</sup>[wikipedia.org/wiki/Forth\\_\(programming\\_language\)](http://wikipedia.org/wiki/Forth_(programming_language))

<sup>10</sup>[wikipedia.org/wiki/Stack\\_machine](http://wikipedia.org/wiki/Stack_machine)

## MSVC + OllyDbg

2 pairs of 32-bit words are marked by red in the stack. Each pair is a double-number in IEEE 754 format and is passed from `main()`. We see how the first `FLD` loads a value (1.2) from the stack and puts it into `ST(0)`:

Figure 17.1: OllyDbg: first `FLD` executed

Because of unavoidable conversion errors from 64-bit IEEE 754 floating point to 80-bit (used internally in the FPU), here we see 1.999..., which is close to 1.2. EIP now points to the next instruction (`FDIV`), which loads a double-number (a constant) from memory. For convenience, OllyDbg shows its value: 3.14

Let's trace further. FDIV was executed, now ST(0) contains 0.382... (quotient):

The screenshot shows the OllyDbg interface with the CPU window displaying assembly code for the 'simple' module. The instruction at address 00FF100C is `FDIV QWORD PTR DS:[0FF20C8]`, which has been executed. The floating-point stack (FPU registers) is visible on the right, showing `ST0 valid 0.3821656050955413719` highlighted in red. The stack window at the bottom shows the memory dump and ASCII representation of the stack data.

Address	Hex dump	ASCII (ANSI -)
00FF3000	25 66 0A 00 00 00 00 00 00 00 00 00 00 00 00 00	%f0
00FF3010	FF FF FF FF FF FF FF FF 5A 20 60 35 A5 DF 9F CA	0 0 2 '5e
00FF3020	FE FF FF FF 01 00 00 00 68 4E 2D 00 00 00 00 00	0 H(- hN-
00FF3030	01 00 00 00 48 28 2D 00 68 4E 2D 00 00 00 00 00	
00FF3040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3060	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	
00FF3080	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	
00FF30A0	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	
00FF30C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Figure 17.2: OllyDbg: FDIV executed



Third step: the next FLD was executed, loading 3.4 into ST(0) (here we see the approximate value 3.39999...):

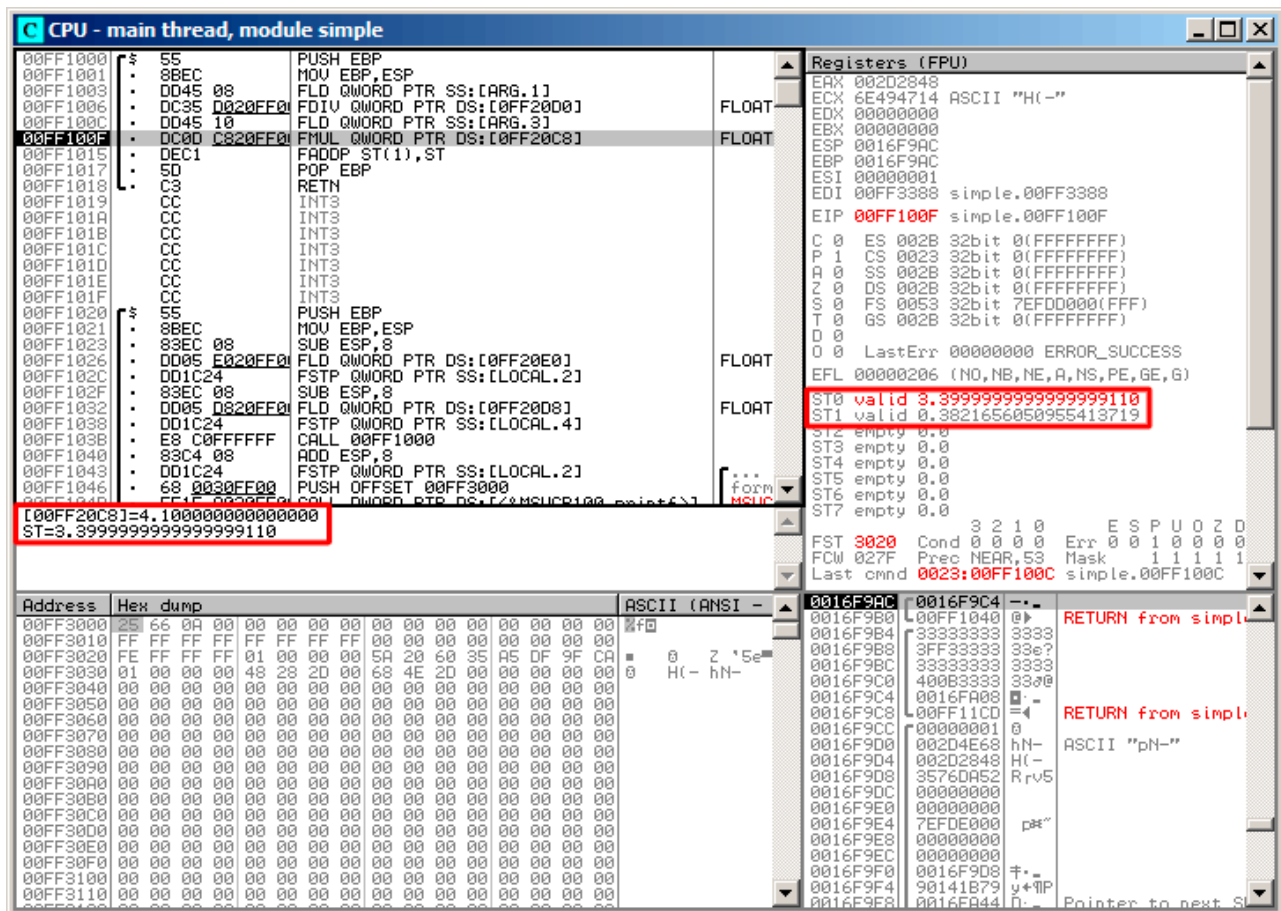


Figure 17.3: OllyDbg: second FLD executed

At the same time, *quotient* is pushed into ST(1). Right now, EIP points to the next instruction: FMUL. It loads the constant 4.1 from memory, which OllyDbg shows.

Next: FMUL was executed, so now the **product** is in ST(0):

The screenshot shows the OllyDbg interface with the CPU window displaying assembly instructions for the 'main' thread in the 'simple' module. The instruction at address 00FF1015 is 'FMUL QWORD PTR DS:[0FF20C8]', which is highlighted. The Registers (FPU) window on the right shows the state of the floating-point registers. ST(0) and ST(1) are highlighted with a red box, showing the result of the FMUL operation: ST(0) valid 13.9399999999999997730 and ST(1) valid 0.3821656050955413719. The CPU window also shows the state of the registers, with ST(0) and ST(1) highlighted.

Address	Hex dump	ASCII (ANSI -)
00FF3000	25 66 0A 00 00 00 00 00 00 00 00 00 00 00 00 00	%f0
00FF3010	FF FF FF FF FF FF FF FF 0A 20 60 35 A5 DF 9F CA	0 2 '5e
00FF3020	FE FF FF FF 01 00 00 00 5A 20 60 35 A5 DF 9F CA	0 H(- hN-
00FF3030	01 00 00 00 48 28 2D 00 68 4E 2D 00 00 00 00 00	
00FF3040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3050	00 00 00 00 00 00 00 00 00 00 00 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	
00FF30D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF30F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00FF3110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Figure 17.4: OllyDbg: FMUL executed

Next: FADDP was executed, now the result of the addition is in ST(0), and ST(1) is cleared:

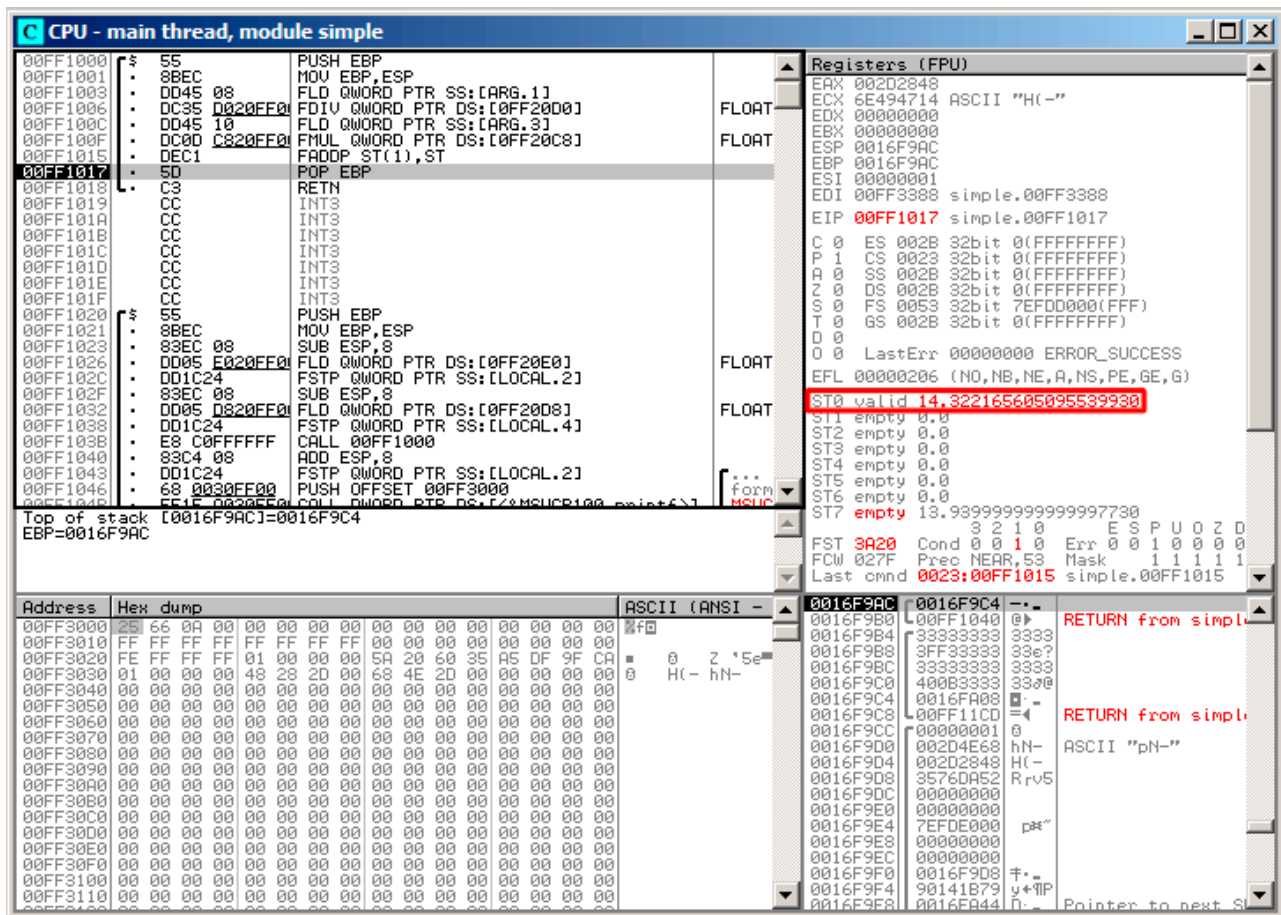


Figure 17.5: OllyDbg: FADDP executed

The result is left in ST(0), because the function returns its value in ST(0). `main()` takes this value from the register later.

We also see something unusual: the 13.93...value is now located in ST(7). Why?

As we have read some time before in this book, the **FPU** registers are a stack: [17.2 on page 206](#). But this is a simplification. Imagine if it was implemented *in hardware* as it's described, then all 7 register's contents must be moved (or copied) to adjacent registers during pushing and popping, and that's a lot of work. In reality, the **FPU** has just 8 registers and a pointer (called TOP) which contains a register number, which is the current "top of stack". When a value is pushed to the stack, TOP is pointed to the next available register, and then a value is written to that register. The procedure is reversed if a value is popped, however, the register which was freed is not cleared (it could possibly be cleared, but this is more work which can degrade performance). So that's what we see here. It can be said that FADDP saved the sum in the stack, and then popped one element. But in fact, this instruction saved the sum and then shifted TOP. More precisely, the registers of the **FPU** are a circular buffer.

## GCC

GCC 4.4.1 (with `-O3` option) emits the same code, just slightly different:

Listing 17.2: Optimizing GCC 4.4.1

```

f
  public f
  proc near

arg_0      = qword ptr 8
arg_8      = qword ptr 10h

          push    ebp
          fld     ds:dbl_8048608 ; 3.14

; stack state now: ST(0) = 3.14

```

```

        mov     ebp, esp
        fdivr   [ebp+arg_0]

; stack state now: ST(0) = result of division

        fld     ds:dbl_8048610 ; 4.1

; stack state now: ST(0) = 4.1, ST(1) = result of division

        fmul    [ebp+arg_8]

; stack state now: ST(0) = result of multiplication, ST(1) = result of division

        pop     ebp
        faddp   st(1), st

; stack state now: ST(0) = result of addition

        retn
f      endp

```

The difference is that, first of all, 3.14 is pushed to the stack (into ST(0)), and then the value in `arg_0` is divided by the value in ST(0).

FDIVR stands for *Reverse Divide*—to divide with divisor and dividend swapped with each other. There is no likewise instruction for multiplication since it is a commutative operation, so we just have FMUL without its -R counterpart.

FADDP adds the two values but also pops one value from the stack. After that operation, ST(0) holds the sum.

## 17.5.2 ARM: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Until ARM got standardized floating point support, several processor manufacturers added their own instructions extensions. Then, VFP (*Vector Floating Point*) was standardized.

One important difference from x86 is that in ARM, there is no stack, you work just with registers.

Listing 17.3: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```

f
    VLDR        D16, =3.14
    VMOV        D17, R0, R1 ; load "a"
    VMOV        D18, R2, R3 ; load "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

```

So, we see here new some registers used, with D prefix. These are 64-bit registers, there are 32 of them, and they can be used both for floating-point numbers (double) but also for SIMD (it is called NEON here in ARM). There are also 32 32-bit S-registers, intended to be used for single precision floating pointer numbers (float). It is easy to remember: D-registers are for double precision numbers, while S-registers—for single precision numbers. More about it: [B.3.3 on page 897](#).

Both constants (3.14 and 4.1) are stored in memory in IEEE 754 format.

VLDR and VMOV, as it can be easily deduced, are analogous to the LDR and MOV instructions, but they work with D-registers. It has to be noted that these instructions, just like the D-registers, are intended not only for floating point numbers, but can be also used for SIMD (NEON) operations and this will also be shown soon.

The arguments are passed to the function in a common way, via the R-registers, however each number that has double precision has a size of 64 bits, so two R-registers are needed to pass each one.

VMOV D17, R0, R1 at the start, composes two 32-bit values from R0 and R1 into one 64-bit value and saves it to D17.

VMOV R0, R1, D16 is the inverse operation: what was in D16 is split in two registers, R0 and R1, because a double-precision number that needs 64 bits for storage, is returned in R0 and R1.

VDIV, VMUL and VADD, are instruction for processing floating point numbers that compute [quotient](#), [product](#) and sum, respectively.

The code for Thumb-2 is same.

### 17.5.3 ARM: Optimizing Keil 6/2013 (Thumb mode)

```
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 in IEEE 754 form:
dword_364      DCD 0x66666666          ; DATA XREF: f+A
dword_368      DCD 0x40106666          ; DATA XREF: f+C
; 3.14 in IEEE 754 form:
dword_36C      DCD 0x51EB851F          ; DATA XREF: f+1A
dword_370      DCD 0x40091EB8          ; DATA XREF: f+1C
```

Keil generated code for a processor without FPU or NEON support. The double-precision floating-point numbers are passed via generic R-registers, and instead of FPU-instructions, service library functions are called (like `__aeabi_dmul`, `__aeabi_ddiv`, `__aeabi_dadd`) which emulate multiplication, division and addition for floating-point numbers. Of course, that is slower than FPU-coprocessor, but still better than nothing.

By the way, similar FPU-emulating libraries were very popular in the x86 world when coprocessors were rare and expensive, and were installed only on expensive computers.

The FPU-coprocessor emulation is called *soft float* or *armel (emulation)* in the ARM world, while using the coprocessor's FPU-instructions is called *hard float* or *armhf*.

### 17.5.4 ARM64: Optimizing GCC (Linaro) 4.9

Very compact code:

Listing 17.4: Optimizing 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

; constants in IEEE 754 format:
.LC25:
```

```

        .word    1374389535      ; 3.14
        .word    1074339512
.LC26:
        .word    1717986918      ; 4.1
        .word    1074816614

```

## 17.5.5 ARM64: Non-optimizing GCC (Linaro) 4.9

Listing 17.5: Non-optimizing GCC (Linaro) 4.9

```

f:
    sub    sp, sp, #16
    str    d0, [sp,8]      ; save "a" in Register Save Area
    str    d1, [sp]        ; save "b" in 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
; D0 = D0+D1 = a/3.14 + b*4.1

    fmov   x0, d0 ; \ redundant code
    fmov   d0, x0 ; /
    add    sp, sp, 16
    ret

.LC25:
    .word    1374389535      ; 3.14
    .word    1074339512
.LC26:
    .word    1717986918      ; 4.1
    .word    1074816614

```

Non-optimizing GCC is more verbose. There is a lot of unnecessary value shuffling, including some clearly redundant code (the last two FMOV instructions). Probably, GCC 4.9 is not yet good in generating ARM64 code. What is worth noting is that ARM64 has 64-bit registers, and the D-registers are 64-bit ones as well. So the compiler is free to save values of type *double* in *GPRs* instead of the local stack. This isn't possible on 32-bit CPUs.

And again, as an exercise, you can try to optimize this function manually, without introducing new instructions like FMADD.

## 17.5.6 MIPS

MIPS can support several coprocessors (up to 4), the zeroth of which is a special control coprocessor, and first coprocessor is the FPU.

As in ARM, the MIPS coprocessor is not a stack machine, it has 32 32-bit registers (\$F0-\$F31): [C.1.2 on page 899](#). When one needs to work with 64-bit *double* values, a pair of 32-bit F-registers is used.

Listing 17.6: Optimizing GCC 4.4.5 (IDA)

```
f:
; $f12-$f13=A
; $f14-$f15=B
        lui    $v0, (dword_C4 >> 16) ; ?
; load low 32-bit part of 3.14 constant to $f0:
        lwc1   $f0, dword_BC
        or     $at, $zero           ; load delay slot, NOP
; load high 32-bit part of 3.14 constant to $f1:
        lwc1   $f1, $LC0
        lui    $v0, ($LC1 >> 16)   ; ?
; A in $f12-$f13, 3.14 constant in $f0-$f1, do division:
        div.d  $f0, $f12, $f0
; $f0-$f1=A/3.14
; load low 32-bit part of 4.1 to $f2:
        lwc1   $f2, dword_C4
        or     $at, $zero           ; load delay slot, NOP
; load high 32-bit part of 4.1 to $f3:
        lwc1   $f3, $LC1
        or     $at, $zero           ; load delay slot, NOP
; B in $f14-$f15, 4.1 constant in $f2-$f3, do multiplication:
        mul.d  $f2, $f14, $f2
; $f2-$f3=B*4.1
        jr     $ra
; sum 64-bit parts and leave result in $f0-$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
```

The new instructions here are:

- LWC1 loads a 32-bit word into a register of the first coprocessor (hence “1” in instruction name). A pair of LWC1 instructions may be combined into a L.D pseudoinstruction.
- DIV.D, MUL.D, ADD.D do division, multiplication, and addition respectively (“D” in the suffix stands for double precision, “S” stands for single precision)

There is also a weird compiler anomaly: the LUI instructions that we’ve marked with a question mark. It’s hard for me to understand why load a part of a 64-bit constant of *double* type into the \$V0 register. These instruction have no effect. If someone knows more about it, please drop an email to author<sup>11</sup>.

## 17.6 Passing floating point numbers via arguments

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

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

    return 0;
}
```

<sup>11</sup>dennis(a)yurichev.com



**17.6.1 x86**

Let's see what we get in (MSVC 2010):

Listing 17.7: 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 ; allocate space for the first variable
    fld     QWORD PTR __real@3ff8a3d70a3d70a4
    fstp    QWORD PTR [esp]
    sub     esp, 8 ; allocate space for the second variable
    fld     QWORD PTR __real@40400147ae147ae1
    fstp    QWORD PTR [esp]
    call    _pow
    add     esp, 8 ; "return back" place of one variable.

; in local stack here 8 bytes still reserved for us.
; result now in ST(0)

    fstp    QWORD PTR [esp] ; move result from ST(0) to local stack for printf()
    push    OFFSET $SG2651
    call    _printf
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0
_main     ENDP

```

FLD and FSTP move variables between the data segment and the FPU stack. `pow()`<sup>12</sup> takes both values from the stack of the FPU and returns its result in the ST(0) register. `printf()` takes 8 bytes from the local stack and interprets them as *double* type variable.

By the way, a pair of MOV instructions could be used here for moving values from the memory into the stack, because the values in memory are stored in IEEE 754 format, and `pow()` also takes them in this format, so no conversion is necessary. That's how it's done in the next example, for ARM: [17.6.2](#).

**17.6.2 ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)**

```

_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}

```

<sup>12</sup>a standard C function, raises a number to the given power (exponentiation)



dbl_2F90	DCFD 32.01	; DATA XREF: _main+6
dbl_2F98	DCFD 1.54	; DATA XREF: _main+E

As it was mentioned before, 64-bit floating pointer numbers are passed in R-registers pairs. This code is a bit redundant (certainly because optimization is turned off), since it is possible to load values into the R-registers directly without touching the D-registers.

So, as we see, the `_pow` function receives its first argument in R0 and R1, and its second one in R2 and R3. The function leaves its result in R0 and R1. The result of `_pow` is moved into D16, then in the R1 and R2 pair, from where `printf()` takes the resulting number.

### 17.6.3 ARM + Non-optimizing Keil 6/2013 (ARM mode)

```

_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
; double x
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

```

D-registers are not used here, just R-register pairs.

### 17.6.4 ARM64 + Optimizing GCC (Linaro) 4.9

Listing 17.8: Optimizing GCC (Linaro) 4.9

```

f:
        stp      x29, x30, [sp, -16]!
        add      x29, sp, 0
        ldr      d1, .LC1 ; load 1.54 into D1
        ldr      d0, .LC0 ; load 32.01 into D0
        bl       pow
; result of pow() in D0
        adrp     x0, .LC2
        add      x0, x0, :lo12:LC2
        bl       printf
        mov      w0, 0
        ldp      x29, x30, [sp], 16
        ret

.LC0:
; 32.01 in IEEE 754 format
        .word    -1374389535
        .word    1077936455

.LC1:
; 1.54 in IEEE 754 format
        .word    171798692
        .word    1073259479

.LC2:
        .string  "32.01 ^ 1.54 = %lf\n"

```

The constants are loaded into D0 and D1: `pow()` takes them from there. The result will be in D0 after the execution of `pow()`. It is to be passed to `printf()` without any modification and moving, because `printf()` takes arguments of [integral types](#) and pointers from X-registers, and floating point arguments from D-registers.

## 17.6.5 MIPS

Listing 17.9: Optimizing GCC 4.4.5 (IDA)

```
main:
var_10      = -0x10
var_4       = -4

; function prologue:
    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) ; ?
; load low 32-bit part of 32.01:
    lwc1    $f12, dword_9C
; load address of pow() function:
    lw      $t9, (pow & 0xFFFF)($gp)
; load high 32-bit part of 32.01:
    lwc1    $f13, $LC0
    lui     $v0, ($LC1 >> 16) ; ?
; load low 32-bit part of 1.54:
    lwc1    $f14, dword_A4
    or      $at, $zero ; load delay slot, NOP
; load high 32-bit part of 1.54:
    lwc1    $f15, $LC1
; call pow():
    jalr    $t9
    or      $at, $zero ; branch delay slot, NOP
    lw      $gp, 0x20+var_10($sp)
; copy result from $f0 and $f1 to $a3 and $a2:
    mfc1    $a3, $f0
    lw      $t9, (printf & 0xFFFF)($gp)
    mfc1    $a2, $f1
; call printf():
    lui     $a0, ($LC2 >> 16) # "32.01 ^ 1.54 = %lf\n"
    jalr    $t9
    la      $a0, ($LC2 & 0xFFFF) # "32.01 ^ 1.54 = %lf\n"
; function epilogue:
    lw      $ra, 0x20+var_4($sp)
; return 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
```

And again, we see here LUI loading a 32-bit part of a *double* number into \$V0. And again, it's hard to comprehend why.

The new instruction for us here is MFC1 ("Move From Coprocessor 1"). The FPU is coprocessor number 1, hence "1" in the instruction name. This instruction transfers values from the coprocessor's registers to the registers of the CPU ([GPR](#)). So in the end the result from `pow()` is moved to registers \$A3 and \$A2, and `printf()` takes a 64-bit double value from this register pair.

## 17.7 Comparison example

Let's try this:

```
#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));
};
```

Despite the simplicity of the function, it will be harder to understand how it works.

### 17.7.1 x86

#### Non-optimizing MSVC

MSVC 2010 generates the following:

Listing 17.10: Non-optimizing 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]

; current stack state: ST(0) = _b
; compare _b (ST(0)) and _a, and pop register

    fcomp   QWORD PTR _a$[ebp]

; stack is empty here

    fnstsw  ax
    test    ah, 5
    jp      SHORT $LN1@d_max

; we are here only if 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
```

So, FLD loads `_b` into `ST(0)`.

FCOMP compares the value in `ST(0)` with what is in `_a` and sets `C3/C2/C0` bits in FPU status word register, accordingly. This is a 16-bit register that reflects the current state of the FPU.

After the bits are set, the FCOMP instruction also pops one variable from the stack. This is what distinguishes it from FCOM, which is just compares values, leaving the stack in the same state.

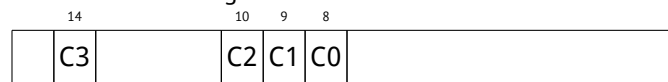
Unfortunately, CPUs before Intel P6<sup>13</sup> don't have any conditional jumps instructions which check the C3/C2/C0 bits. Probably, it is a matter of history (remember: FPU was separate chip in past).

Modern CPU starting at Intel P6 have FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions –which do the same, but modify the ZF/PF/CF CPU flags.

The FNSTSW instruction copies FPU the status word register to AX. C3/C2/C0 bits are placed at positions 14/10/8, they are at the same positions in the AX register and all they are placed in the high part of AX –AH.

- If  $b > a$  in our example, then C3/C2/C0 bits are to be set as following: 0, 0, 0.
- If  $a > b$ , then the bits are: 0, 0, 1.
- If  $a = b$ , then the bits are: 1, 0, 0.
- If the result is unordered (in case of error), then the set bits are: 1, 1, 1.

This is how C3/C2/C0 bits are located in the AX register:



This is how C3/C2/C0 bits are located in the AH register:



After the execution of `test ah, 5`<sup>14</sup>, only C0 and C2 bits (on 0 and 2 position) are considered, all other bits are just ignored.

Now let's talk about the *parity flag*, another notable historical rudiment.

This flag is set to 1 if the number of ones in the result of the last calculation is even, and to 0 if it is odd.

Let's look into Wikipedia<sup>15</sup>:

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.

As noted in Wikipedia, the parity flag used sometimes in FPU code, let's see how.

The PF flag is to be set to 1 if both C0 and C2 are set to 0 or both are 1, in which case the subsequent JP (*jump if PF==1*) is triggering. If we recall the values of C3/C2/C0 for various cases, we can see that the conditional jump JP is triggering in two cases: if  $b > a$  or  $a = b$  (C3 bit is not considered here, since it was cleared by the `test ah, 5` instruction).

It is all simple after that. If the conditional jump was triggered, FLD loads the value of `_b` in ST(0), and if it was not triggered, the value of `_a` is loaded there.

### And what about checking C2?

The C2 flag is set in case of error (NaN, etc), but our code doesn't check it. If the programmer cares about FPU errors, he/she must add additional checks.

<sup>13</sup>Intel P6 is Pentium Pro, Pentium II, etc

<sup>14</sup>5=101b

<sup>15</sup>[wikipedia.org/wiki/Parity\\_flag](https://wikipedia.org/wiki/Parity_flag)

First OllyDbg example:  $a=1.2$  and  $b=3.4$ 

Let's load the example into OllyDbg:

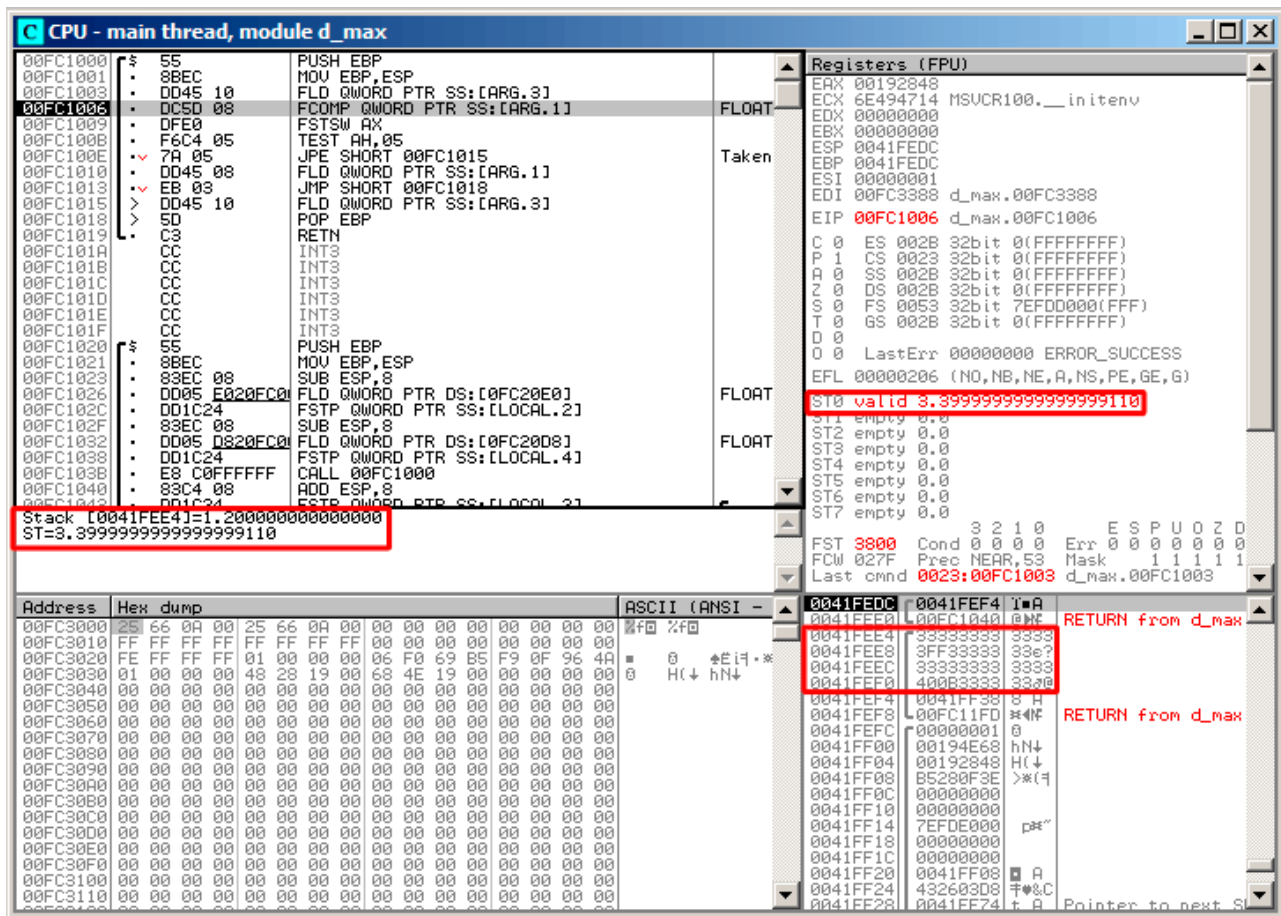


Figure 17.6: OllyDbg: first FLD is executed

Current arguments of the function:  $a = 1.2$  and  $b = 3.4$  (We can see them in the stack: two pairs of 32-bit values).  $b$  (3.4) is already loaded in ST(0). Now FCOMP is being executed. OllyDbg shows the second FCOMP argument, which is in stack right now.

1 COM1 IS executed.

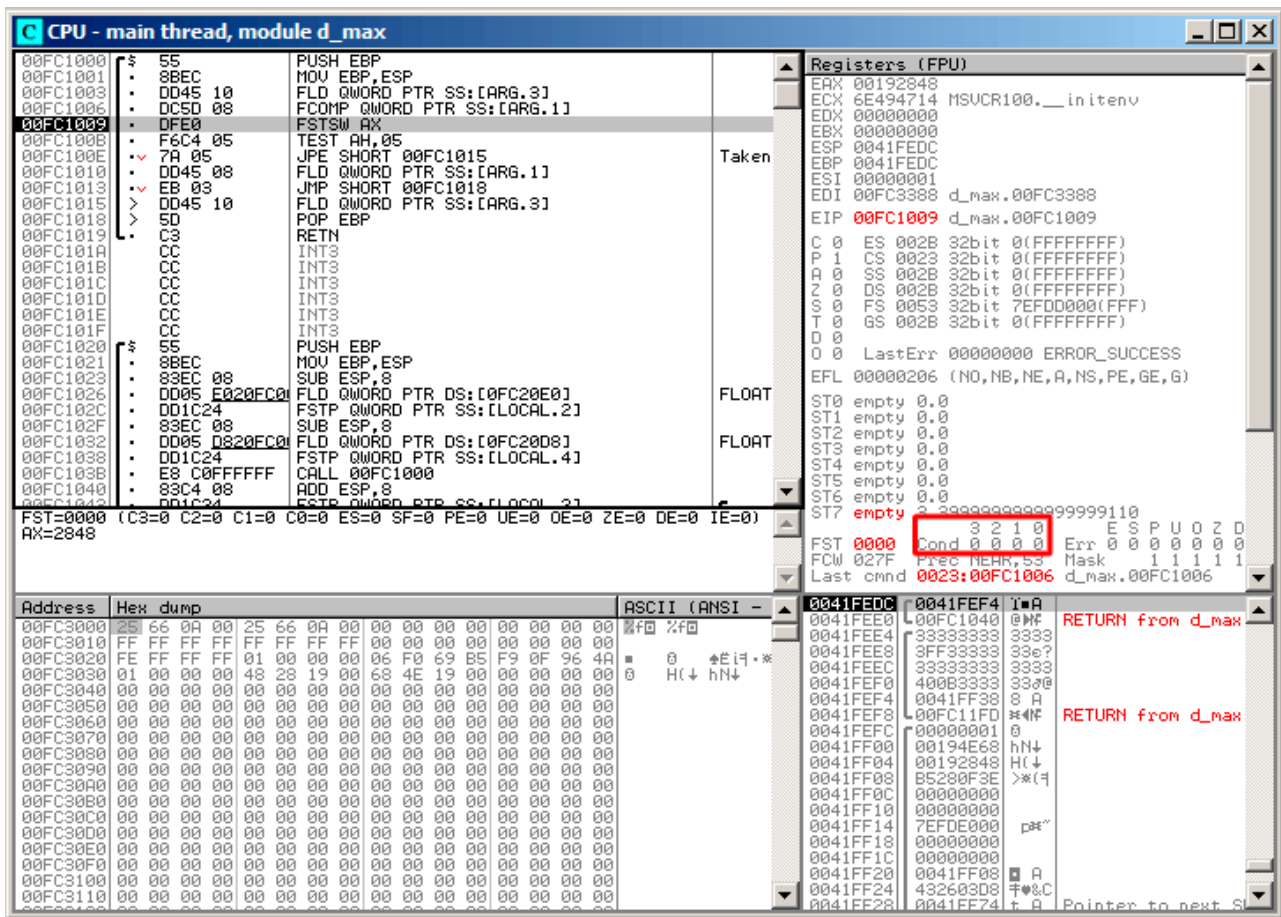


Figure 17.7: OllyDbg: FCOMP is executed

We see the state of the FPU's condition flags: all zeroes. The popped value is reflected as ST(7), it was written earlier about reason for this: [17.5.1 on page 213](#).

FNSTSW is executed:

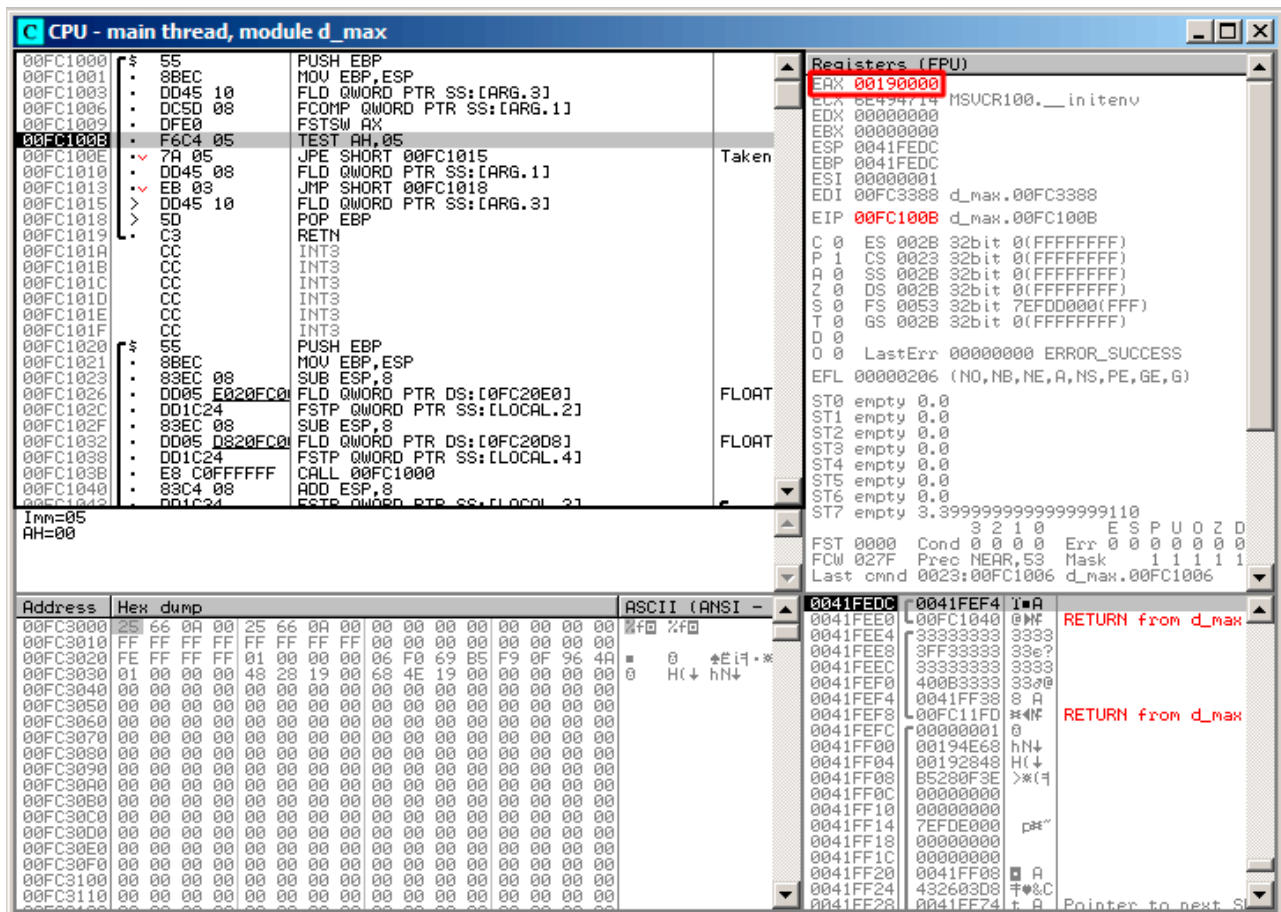


Figure 17.8: OllyDbg: FNSTSW is executed

We see that the AX register contain zeroes: indeed, all condition flags are zero. (OllyDbg disassembles the FNSTSW instruction as FSTSW—they are synonyms).

TEST is executed:

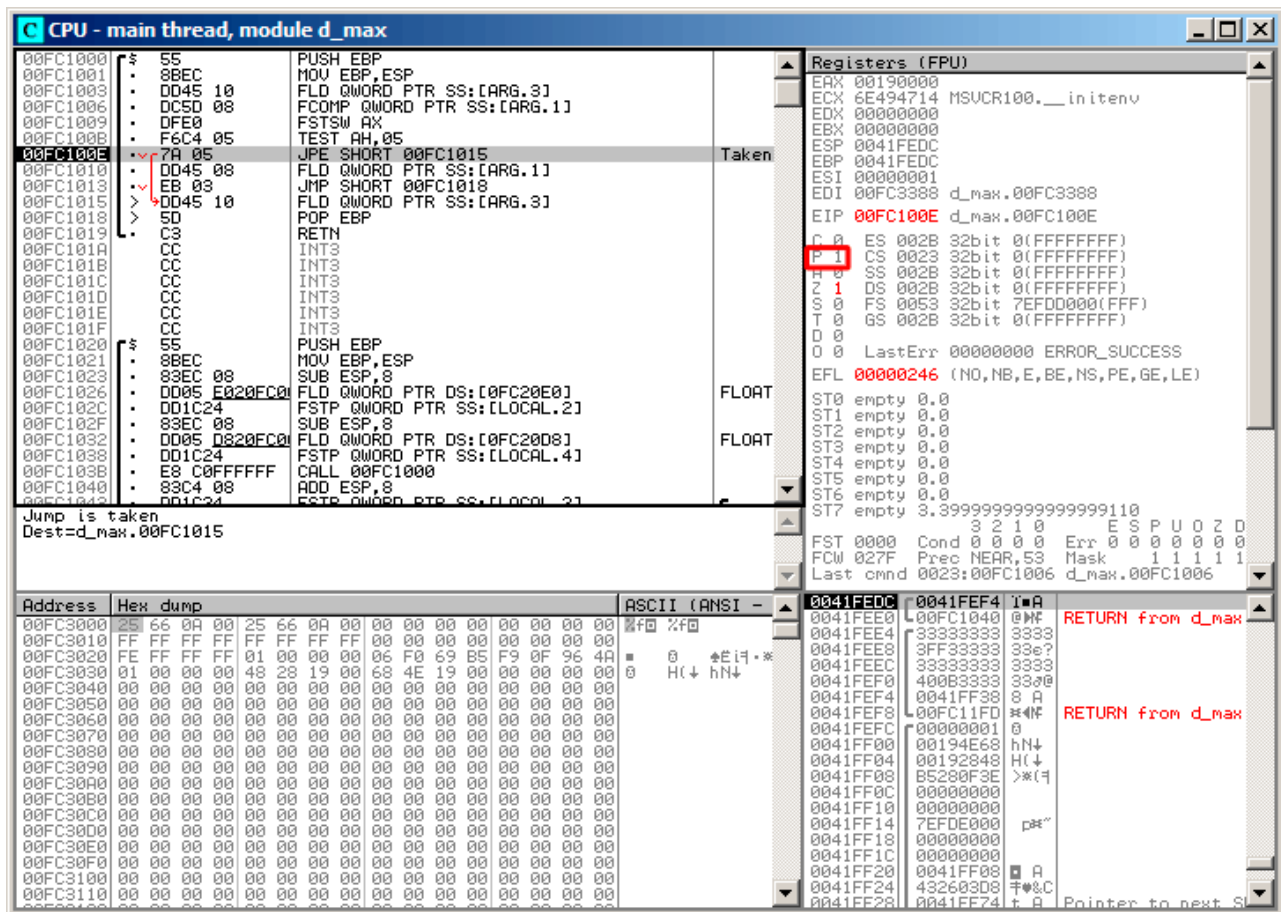


Figure 17.9: OllyDbg: TEST is executed

The PF flag is set to 1. Indeed: the number of bits set in 0 is 0 and 0 is an even number. OllyDbg disassembles JP as [JPE<sup>16</sup>](#)—they are synonyms. And it is about to trigger now.

<sup>16</sup>Jump Parity Even (x86 instruction)



JPE triggered, FLD loads the value of  $b$  (3.4) in ST(0):

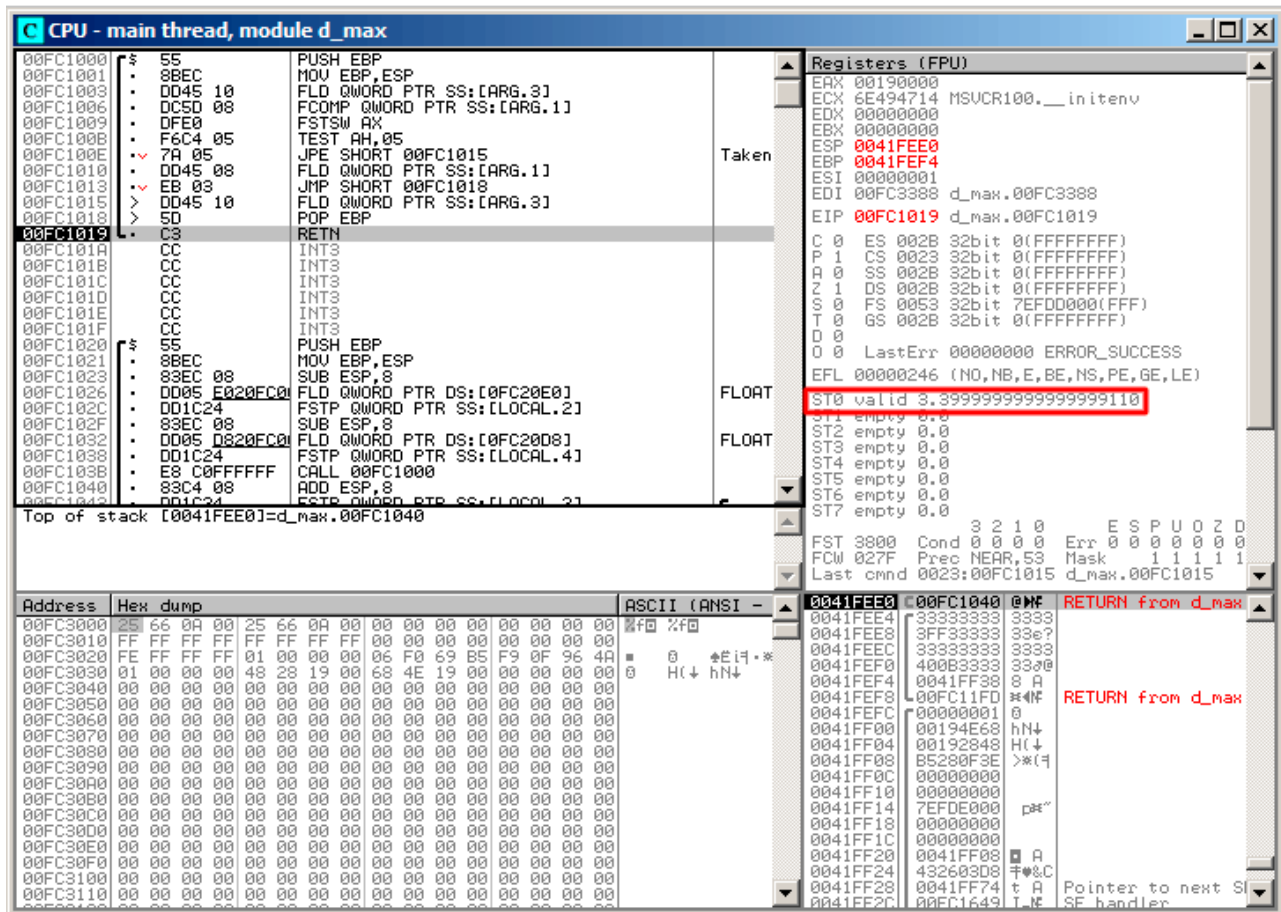


Figure 17.10: OllyDbg: second FLD is executed

The function finishes its work.

Second OllyDbg example:  $a=5.6$  and  $b=-4$ 

Let's load example into OllyDbg:

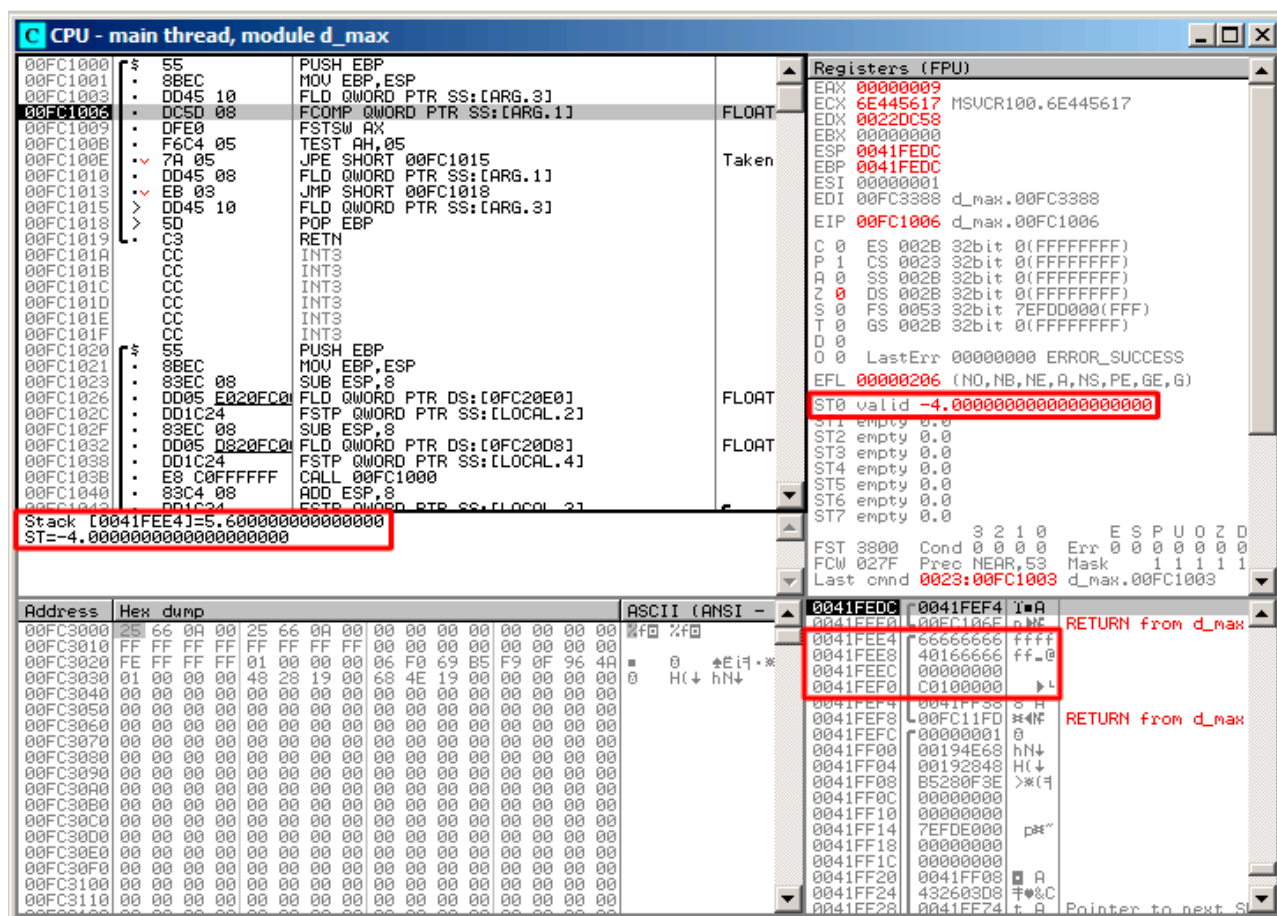


Figure 17.11: OllyDbg: first FLD executed

Current function arguments:  $a = 5.6$  and  $b = -4$ .  $b$  (-4) is already loaded in ST(0). FCOMP about to execute now. OllyDbg shows the second FCOMP argument, which is in stack right now.

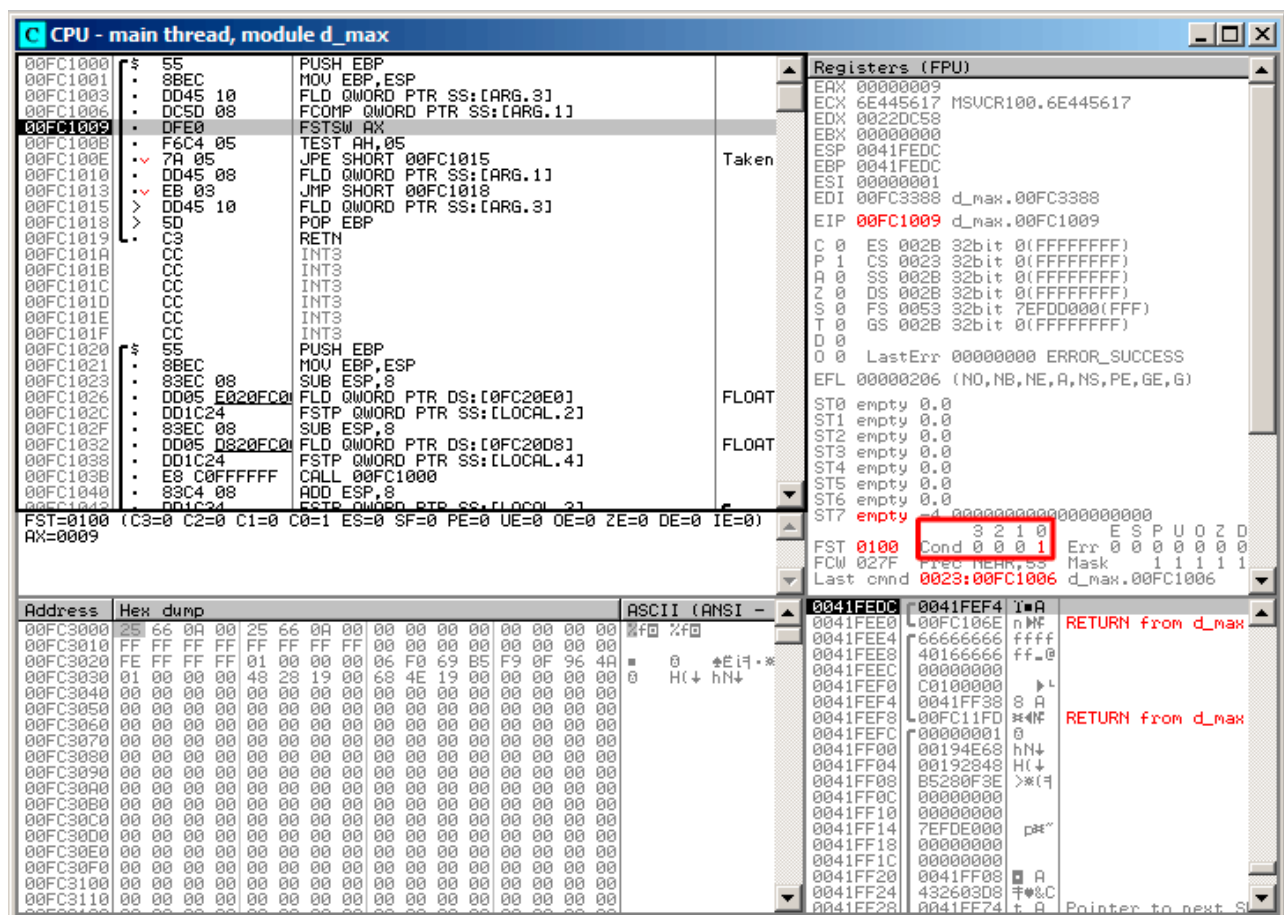


Figure 17.12: OllyDbg: FCOMP executed

We see the state of the FPU's condition flags: all zeroes except C0.

FNSTSW executed:

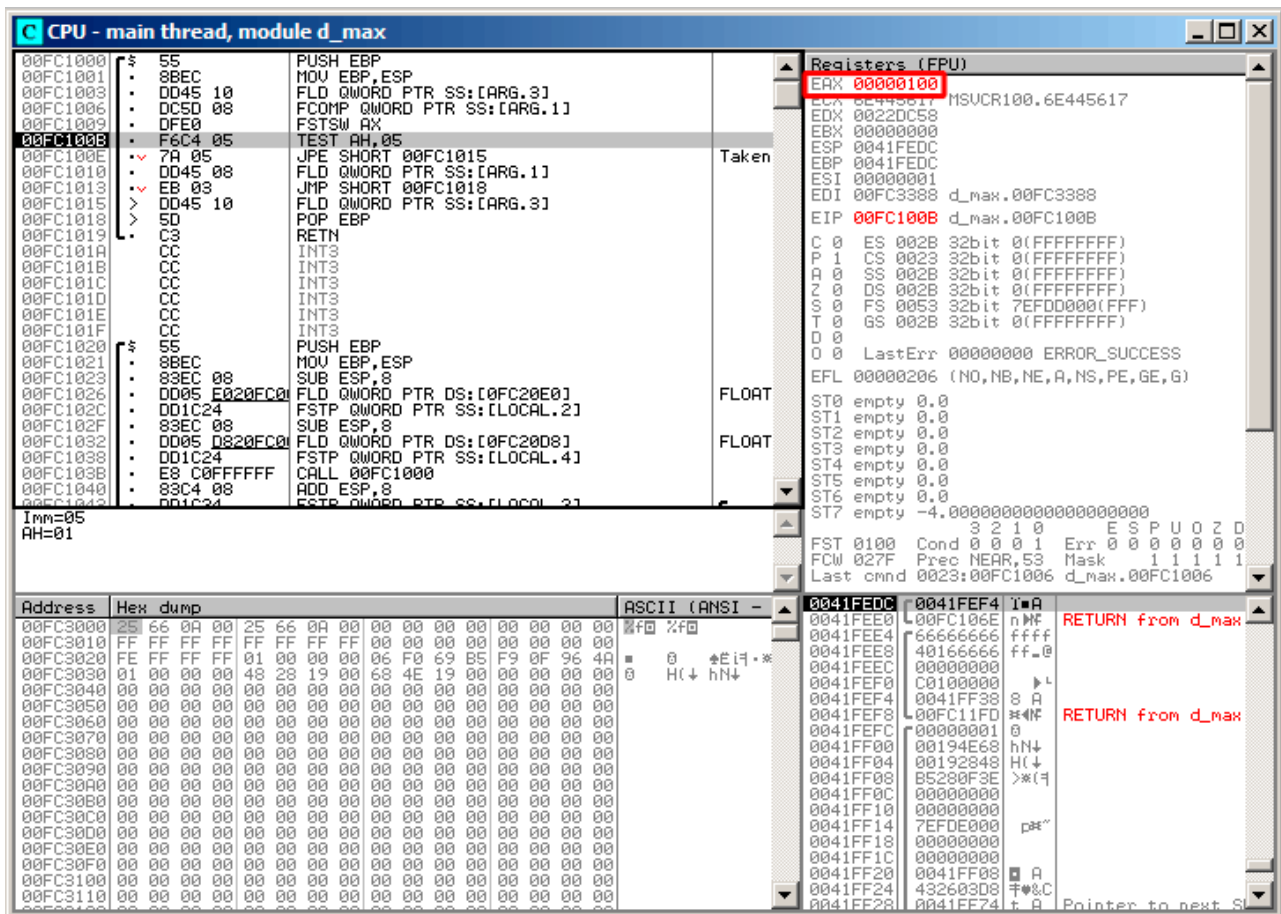


Figure 17.13: OllyDbg: FNSTSW executed

We see that the AX register contains 0x100: the C0 flag is at the 16th bit.

TEST executed:

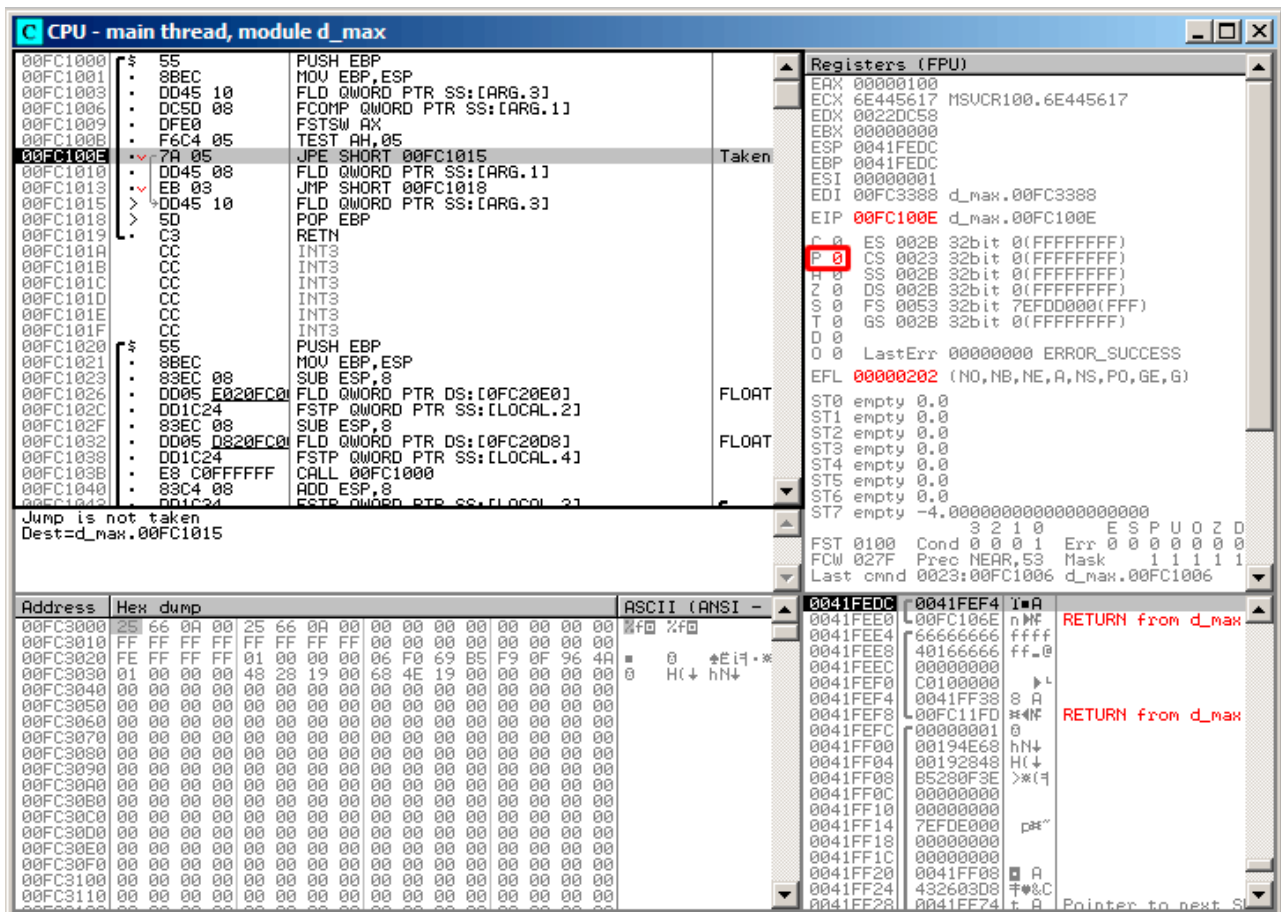


Figure 17.14: OllyDbg: TEST executed

The PF flag is cleared. Indeed: the count of bits set in 0x100 is 1 and 1 is an odd number. `JPE` is being skipped now.

JPE wasn't triggered, so FLD loads the value of  $a$  (5.6) in ST(0):

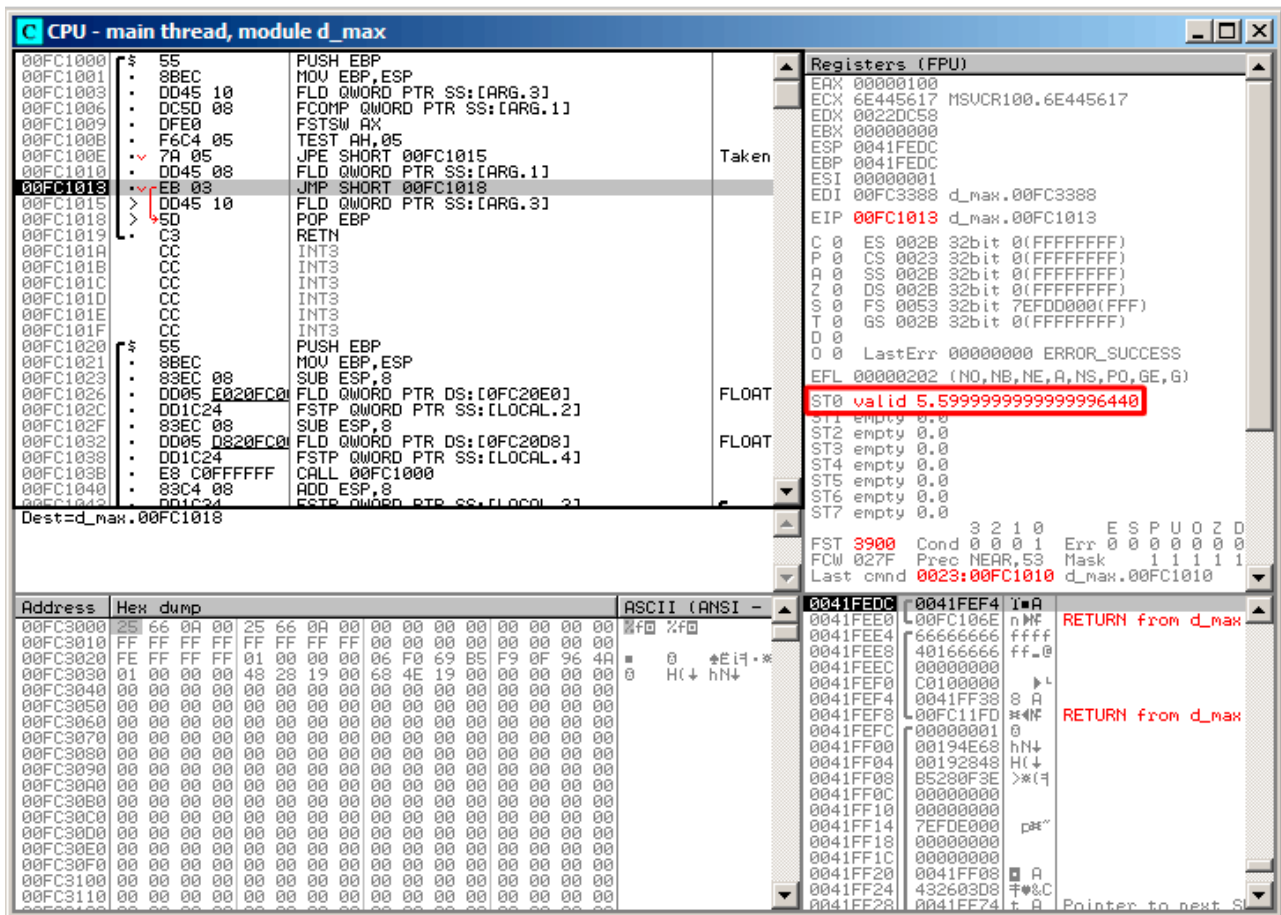


Figure 17.15: OllyDbg: second FLD executed

The function finishes its work.

## Optimizing MSVC 2010

Listing 17.11: Optimizing 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]

; current stack state: ST(0) = _a, ST(1) = _b

    fcom ST(1) ; compare _a and ST(1) = (_b)
    fnstsw ax
    test ah, 65 ; 00000041H
    jne SHORT $LN5@d_max
; copy ST(0) to ST(1) and pop register,
; leave (_a) on top
    fstp ST(1)

; current stack state: ST(0) = _a

    ret 0
$LN5@d_max:
; copy ST(0) to ST(0) and pop register,
; leave (_b) on top
    fstp ST(0)

; current stack state: ST(0) = _b

```



ret	0
_d_max	ENDP

FCOM differs from FCOMP in the sense that it just compares the values and doesn't change the FPU stack. Unlike the previous example, here the operands are in reverse order, which is why the result of the comparison in C3/C2/C0 is different:

- If  $a > b$  in our example, then C3/C2/C0 bits are to be set as: 0, 0, 0.
- If  $b > a$ , then the bits are: 0, 0, 1.
- If  $a = b$ , then the bits are: 1, 0, 0.

The test `ah, 65` instruction leaves just two bits –C3 and C0. Both will be zero if  $a > b$ : in that case the JNE jump will not be triggered. Then `FSTP ST(1)` follows –this instruction copies the value from ST(0) to the operand and pops one value from the FPU stack. In other words, the instruction copies ST(0) (where the value of `_a` is now) into ST(1). After that, two copies of `_a` are at the top of the stack. Then, one value is popped. After that, ST(0) contains `_a` and the function is finishes.

The conditional jump JNE is triggering in two cases: if  $b > a$  or  $a = b$ . ST(0) is copied into ST(0), it is just like an idle (NOP) operation, then one value is popped from the stack and the top of the stack (ST(0)) is contain what was in ST(1) before (that is `_b`). Then the function finishes. The reason this instruction is used here probably is because the FPU has no other instruction to pop a value from the stack and discard it.

## First OllyDbg example: a=1.2 and b=3.4

Both FLD are executed:

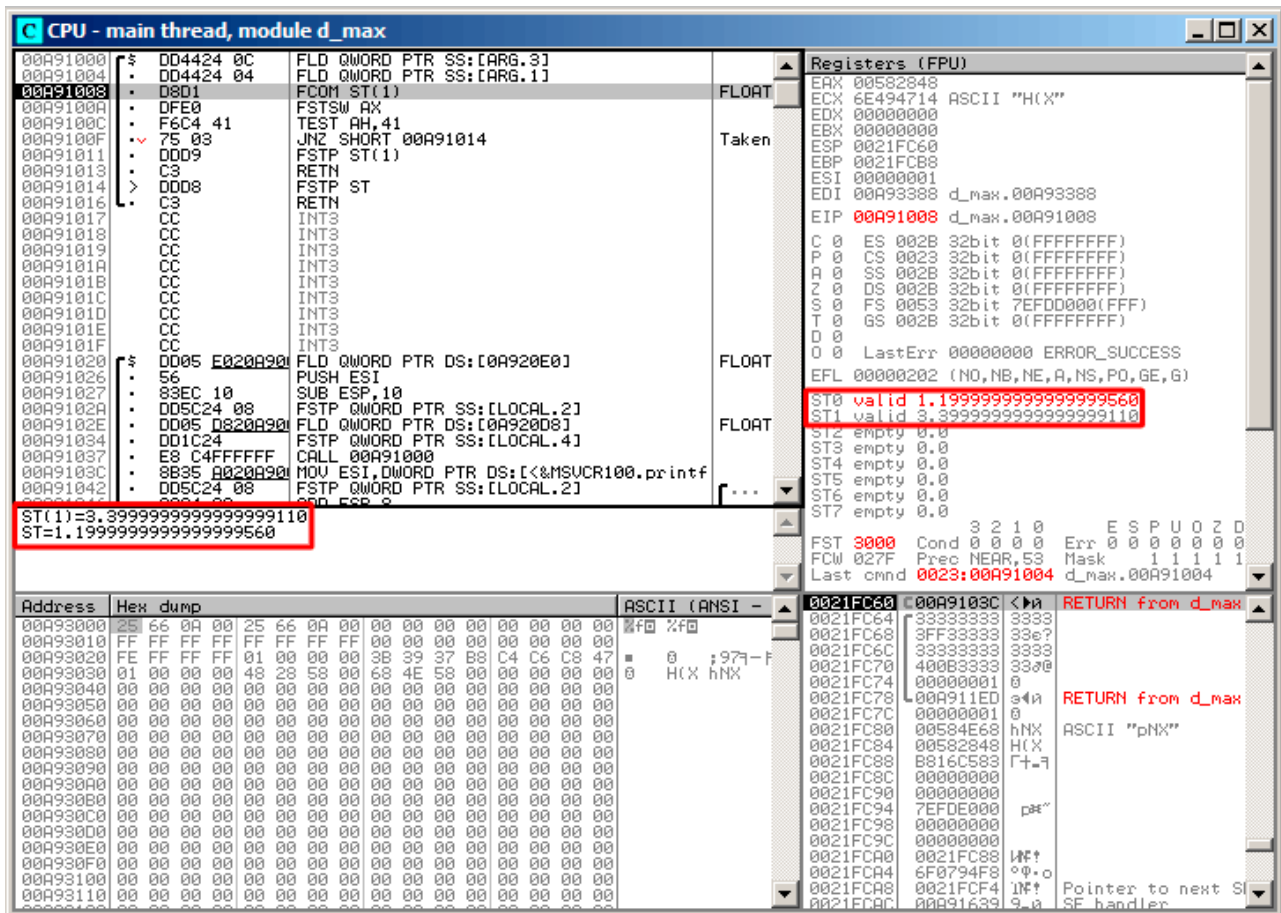


Figure 17.16: OllyDbg: both FLD are executed

FCOM being executed: OllyDbg shows the contents of ST(0) and ST(1) for convenience.



FCOM is done:

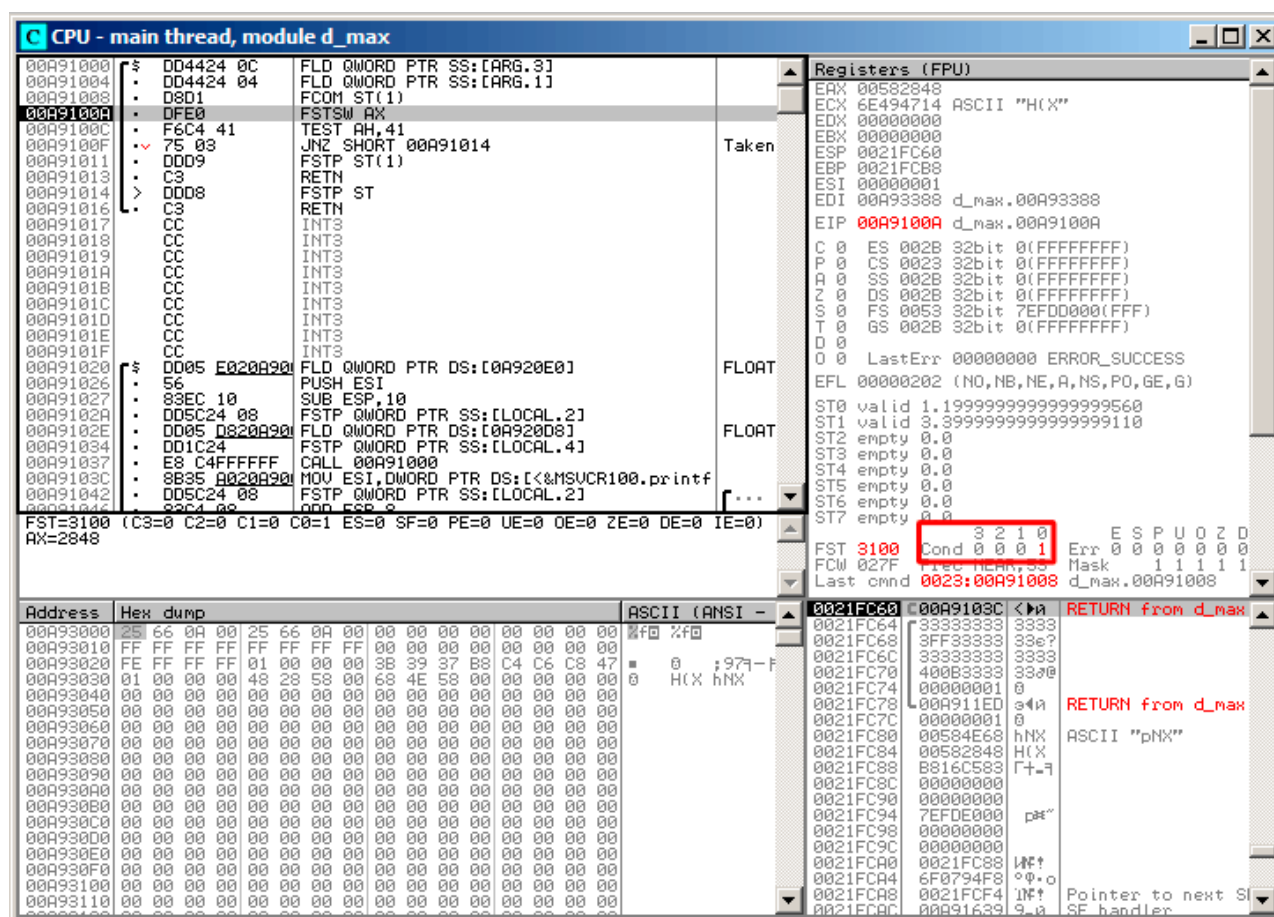


Figure 17.17: OllyDbg: FCOM is done

C0 is set, all other condition flags are cleared.

FNSTSW is done, AX=0x3100:

The screenshot shows the OllyDbg interface with the following components:

- CPU - main thread, module d\_max:**
  - Address 00A91000: DD4424 0C FLD QWORD PTR SS:[ARG.3]
  - Address 00A91004: DD4424 04 FLD QWORD PTR SS:[ARG.1]
  - Address 00A91008: D8D1 FCOM ST(1)
  - Address 00A9100A: DFE0 FSTSW AX
  - Address 00A9100C: F6C4 41 TEST AH,41
  - Address 00A9100F: 75 03 JNZ SHORT 00A91014
  - Address 00A91011: DDD9 FSTP ST(1)
  - Address 00A91013: C3 RETN
  - Address 00A91014: DDD8 FSTP ST
  - Address 00A91016: C3 RETN
  - Address 00A91017: CC INT3
  - Address 00A91018: CC INT3
  - Address 00A91019: CC INT3
  - Address 00A9101A: CC INT3
  - Address 00A9101B: CC INT3
  - Address 00A9101C: CC INT3
  - Address 00A9101D: CC INT3
  - Address 00A9101E: CC INT3
  - Address 00A9101F: CC INT3
  - Address 00A91020: DD05 F020A90 FLD QWORD PTR DS:[0A920E0]
  - Address 00A91022: 56 PUSH ESI
  - Address 00A91027: 83EC 10 SUB ESP,10
  - Address 00A9102A: DD5C24 08 FSTP QWORD PTR SS:[LOCAL.2]
  - Address 00A9102E: DD05 D820A90 FLD QWORD PTR DS:[0A920D8]
  - Address 00A91034: DD1C24 FSTP QWORD PTR SS:[LOCAL.4]
  - Address 00A91037: E8 C4FFFFFF CALL 00A91000
  - Address 00A9103C: 8B35 A020A90 MOV ESI,DWORD PTR DS:[<&MSUCR100.printf
  - Address 00A91042: DD5C24 08 FSTP QWORD PTR SS:[LOCAL.2]
  - Address 00A91046: 56 POP ESP
- Registers (FPU):**
  - EAX 00583100 ASCII "Administrator"
  - ECX 6E497719 ASCII "H(X"
  - EDX 00000000
  - EBX 00000000
  - ESP 0021FC60
  - EBP 0021FCB8
  - ESI 00000001
  - EDI 00A93388 d\_max.00A93388
  - EIP 00A9100C d\_max.00A9100C
  - 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 valid 1.1999999999999999560
  - ST1 valid 3.3999999999999999110
  - ST2 empty 0.0
  - ST3 empty 0.0
  - ST4 empty 0.0
  - ST5 empty 0.0
  - ST6 empty 0.0
  - ST7 empty 0.0
- Disassembly:**
  - Address 00A9103C: 8B35 A020A90 MOV ESI,DWORD PTR DS:[<&MSUCR100.printf
- Registers (GPR):**
  - EAX 00583100
  - ECX 6E497719
  - EDX 00000000
  - EBX 00000000
  - ESP 0021FC60
  - EBP 0021FCB8
  - ESI 00000001
  - EDI 00A93388
  - EIP 00A9100C
- Registers (FPU):**
  - EAX 00583100
  - ECX 6E497719
  - EDX 00000000
  - EBX 00000000
  - ESP 0021FC60
  - EBP 0021FCB8
  - ESI 00000001
  - EDI 00A93388
  - EIP 00A9100C
- Registers (GPR):**
  - EAX 00583100
  - ECX 6E497719
  - EDX 00000000
  - EBX 00000000
  - ESP 0021FC60
  - EBP 0021FCB8
  - ESI 00000001
  - EDI 00A93388
  - EIP 00A9100C

Figure 17.18: OllyDbg: FNSTSW is executed

TEST is executed:

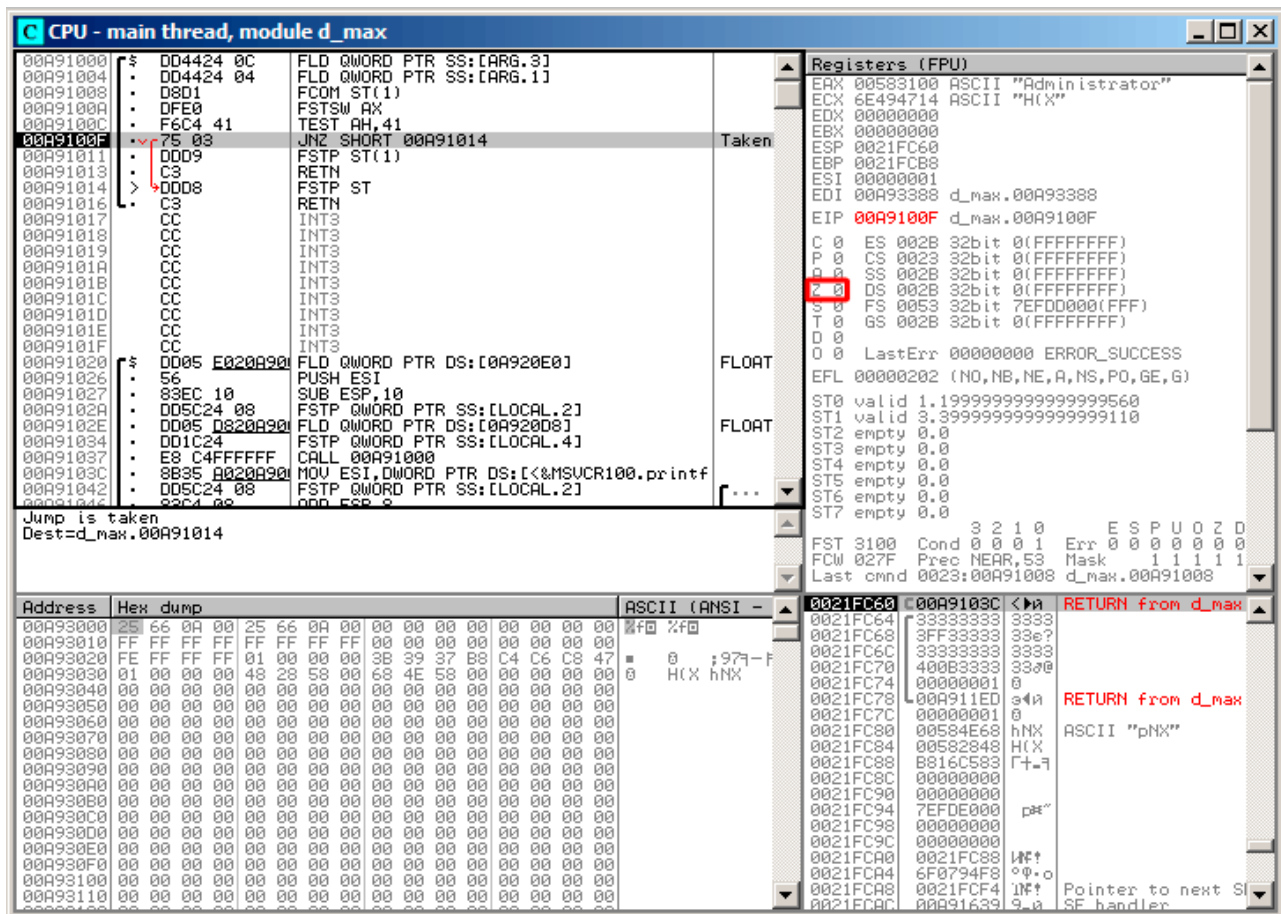


Figure 17.19: OllyDbg: TEST is executed

ZF=0, conditional jump is about to trigger now.

FSTP ST (or FSTP ST(0)) was executed –1.2 was popped from the stack, and 3.4 was left on top:

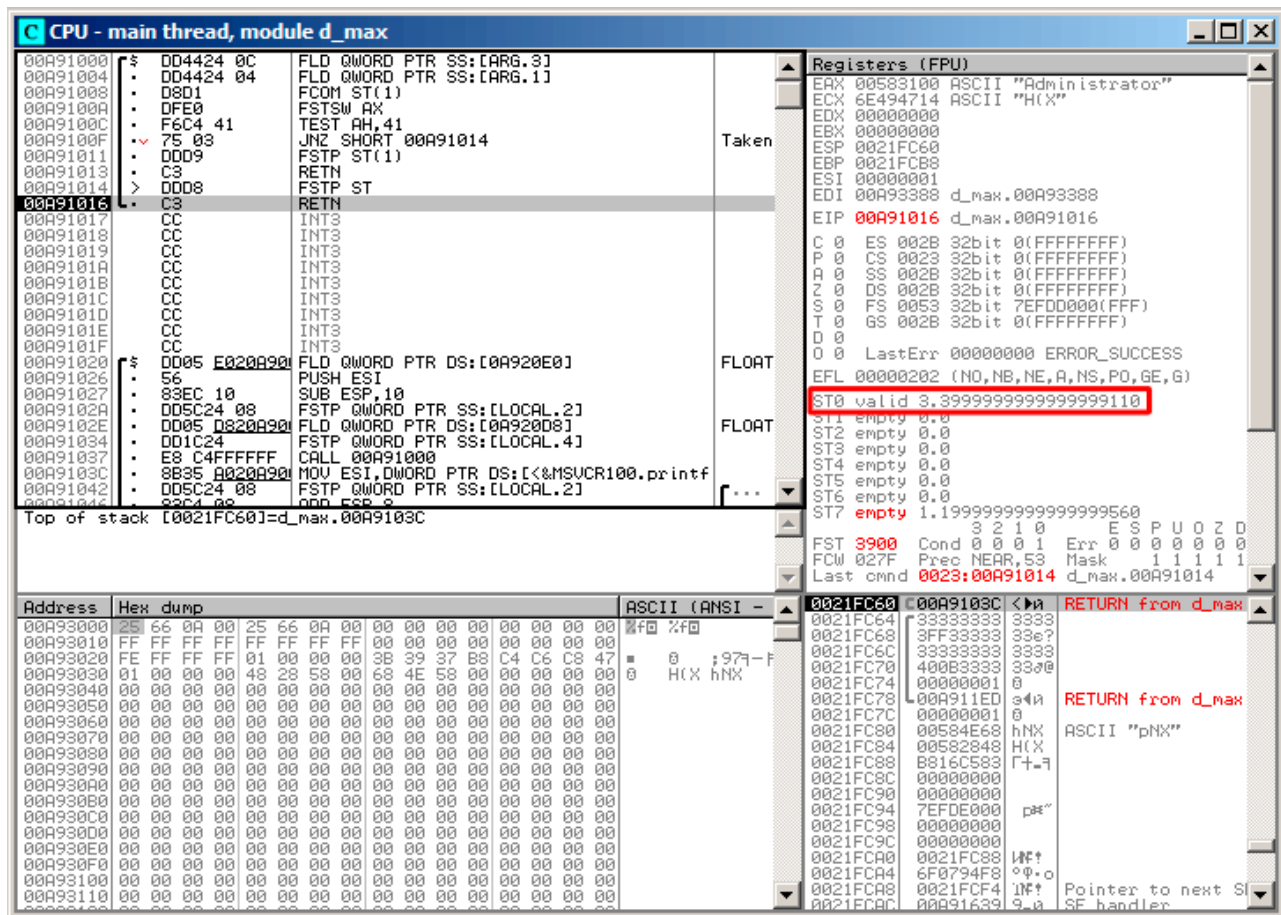


Figure 17.20: OllyDbg: FSTP is executed

We see that the FSTP ST instruction works just like popping one value from the FPU stack.

Second OllyDbg example:  $a=5.6$  and  $b=-4$ 

Both FLD are executed:

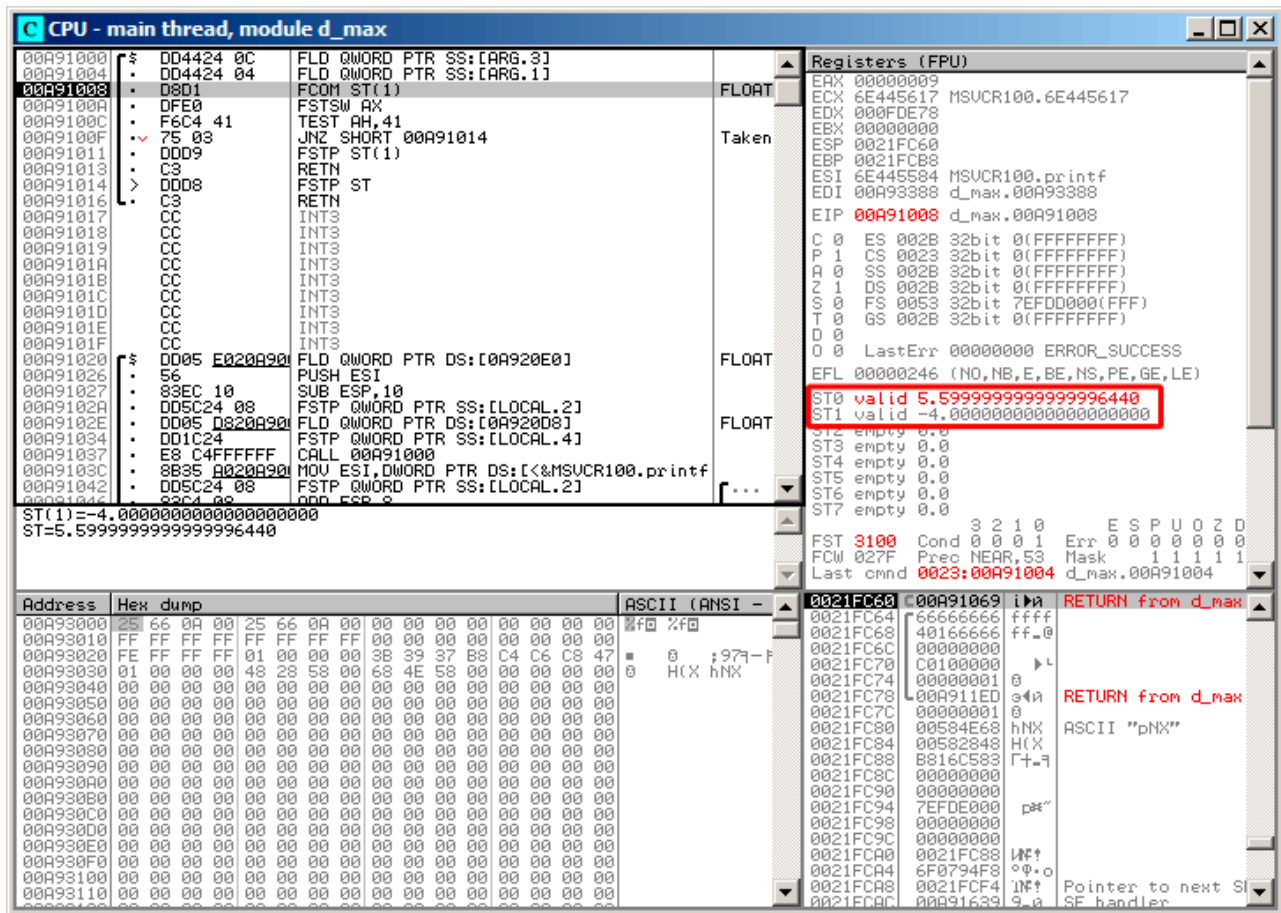


Figure 17.21: OllyDbg: both FLD are executed

FCOM is about to execute.

FCOM is done:

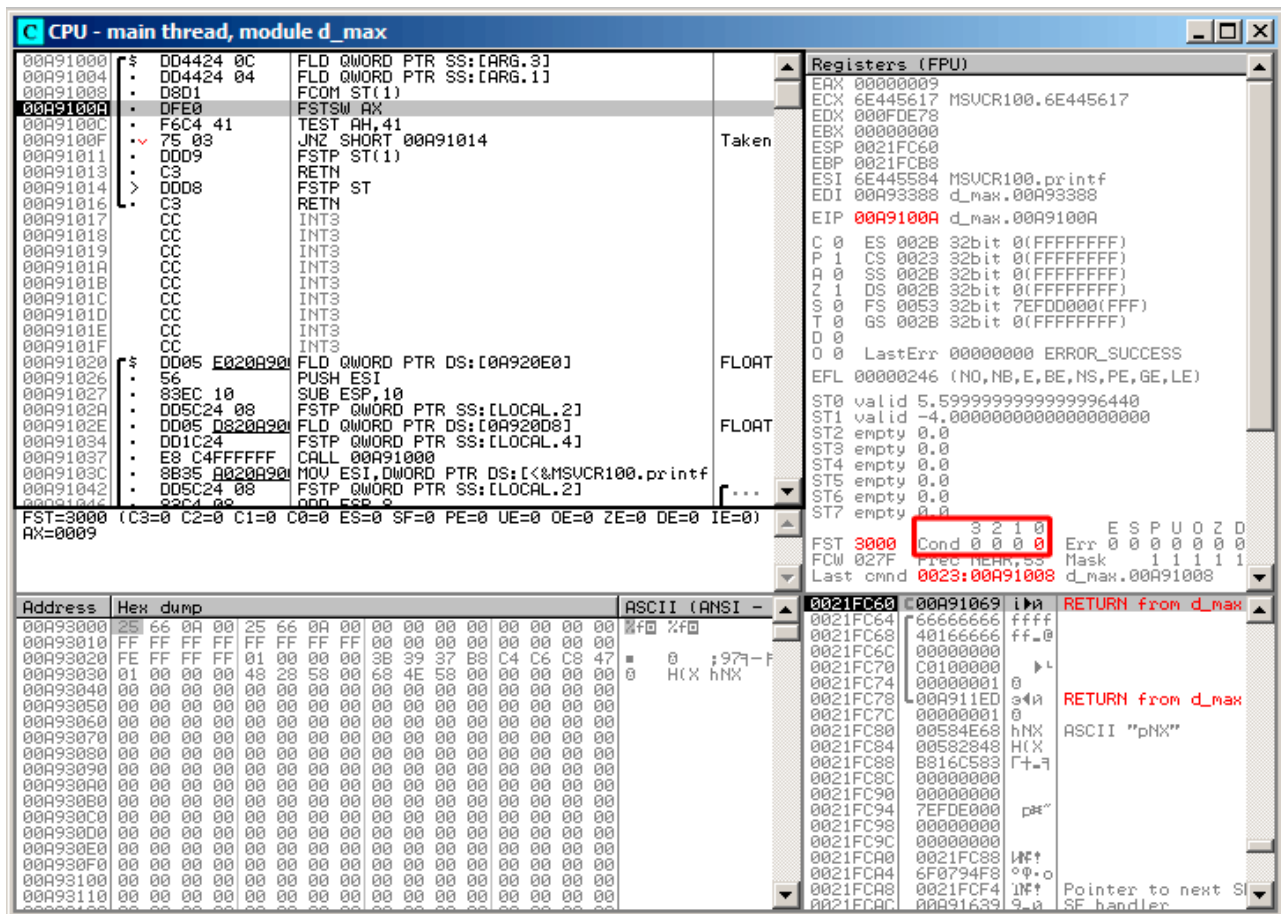


Figure 17.22: OllyDbg: FCOM is finished

All conditional flags are cleared.



FNSTSW done, AX=0x3000:

**CPU - main thread, module d\_max**

Address	Hex dump	ASCII (ANSI -)
00A93000	25 66 0A 00 25 66 0A 00 00 00 00 00 00 00 00 00	%f0 %f0
00A93010	FF FF FF FF FF FF FF FF 3B 39 37 B8 C4 C6 C8 47	0 0 ;979-F
00A93020	01 00 00 00 48 28 58 00 68 4E 58 00 00 00 00 00	0 H(X hNX
00A93030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A930F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00A93110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

**Registers (FPU)**

Register	Value
EAX	00003000
ECX	6E445617 MSUCR100.6E445617
EDX	000FDE78
EBX	00000000
ESP	0021FC60
EBP	0021FCB8
ESI	6E445684 MSUCR100.printf
EDI	00A93388 d_max.00A93388
EIP	00A9100C d_max.00A9100C
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)
ST0	valid 5.5999999999999996440
ST1	valid -4.00000000000000000000
ST2	empty 0.0
ST3	empty 0.0
ST4	empty 0.0
ST5	empty 0.0
ST6	empty 0.0
ST7	empty 0.0

**Instruction List**

Address	Disassembly	Comment
00A91000	DD4424 0C FLD QWORD PTR SS:[ARG.3]	
00A91004	DD4424 04 FLD QWORD PTR SS:[ARG.1]	
00A91008	D8D1 FCOM ST(1)	
00A9100A	DFF0 FSTSW AX	
00A9100C	F6C4 41 TEST AH,41	
00A9100F	75 03 JNZ SHORT 00A91014	Taken
00A91011	DD09 FSTP ST(1)	
00A91013	C3 RETN	
00A91014	DD08 FSTP ST	
00A91016	C3 RETN	
00A91017	CC INT3	
00A91018	CC INT3	
00A91019	CC INT3	
00A9101A	CC INT3	
00A9101B	CC INT3	
00A9101C	CC INT3	
00A9101D	CC INT3	
00A9101E	CC INT3	
00A9101F	CC INT3	
00A91020	DD05 F020A90 FLD QWORD PTR DS:[0A920E0]	FLOAT
00A91026	56 PUSH ESI	
00A91027	83EC 10 SUB ESP,10	
00A9102A	DD5C24 08 FSTP QWORD PTR SS:[LOCAL.2]	FLOAT
00A9102E	DD05 D820A90 FLD QWORD PTR DS:[0A920D8]	FLOAT
00A91034	DD1C24 FSTP QWORD PTR SS:[LOCAL.4]	
00A91037	E8 C4FFFFFF CALL 00A91000	
00A9103C	8B35 0020A90 MOV ESI,DWORD PTR DS:[<MSUCR100.printf	
00A91042	DD5C24 08 FSTP QWORD PTR SS:[LOCAL.2]	
00A91046	83C4 08 ADD ESP,8	

**Return from d\_max**

Address	Hex dump	ASCII (ANSI -)
00A91069	66 66 66 66 ffff	
00A9106A	40 16 66 66 ff-0	
00A9106B	00 00 00 00	
00A9106C	C0 10 00 00	
00A9106D	00 00 00 01	
00A9106E	00 A9 11 ED	
00A9106F	00 00 00 01	
00A91070	00 53 4E 68 hNX	
00A91071	00 58 28 48 hX	
00A91072	89 1C 58 3	
00A91073	00 00 00 00	
00A91074	00 00 00 00	
00A91075	00 00 00 00	
00A91076	00 00 00 00	
00A91077	00 00 00 00	
00A91078	00 00 00 00	
00A91079	00 00 00 00	
00A9107A	00 00 00 00	
00A9107B	00 00 00 00	
00A9107C	00 00 00 00	
00A9107D	00 00 00 00	
00A9107E	00 00 00 00	
00A9107F	00 00 00 00	
00A91080	00 00 00 00	
00A91081	00 00 00 00	
00A91082	00 00 00 00	
00A91083	00 00 00 00	
00A91084	00 00 00 00	
00A91085	00 00 00 00	
00A91086	00 00 00 00	
00A91087	00 00 00 00	
00A91088	00 00 00 00	
00A91089	00 00 00 00	
00A9108A	00 00 00 00	
00A9108B	00 00 00 00	
00A9108C	00 00 00 00	
00A9108D	00 00 00 00	
00A9108E	00 00 00 00	
00A9108F	00 00 00 00	
00A91090	00 00 00 00	
00A91091	00 00 00 00	
00A91092	00 00 00 00	
00A91093	00 00 00 00	
00A91094	00 00 00 00	
00A91095	00 00 00 00	
00A91096	00 00 00 00	
00A91097	00 00 00 00	
00A91098	00 00 00 00	
00A91099	00 00 00 00	
00A9109A	00 00 00 00	
00A9109B	00 00 00 00	
00A9109C	00 00 00 00	
00A9109D	00 00 00 00	
00A9109E	00 00 00 00	
00A9109F	00 00 00 00	
00A910A0	00 00 00 00	
00A910A1	00 00 00 00	
00A910A2	00 00 00 00	
00A910A3	00 00 00 00	
00A910A4	00 00 00 00	
00A910A5	00 00 00 00	
00A910A6	00 00 00 00	
00A910A7	00 00 00 00	
00A910A8	00 00 00 00	
00A910A9	00 00 00 00	
00A910AA	00 00 00 00	
00A910AB	00 00 00 00	
00A910AC	00 00 00 00	
00A910AD	00 00 00 00	
00A910AE	00 00 00 00	
00A910AF	00 00 00 00	
00A910B0	00 00 00 00	
00A910B1	00 00 00 00	
00A910B2	00 00 00 00	
00A910B3	00 00 00 00	
00A910B4	00 00 00 00	
00A910B5	00 00 00 00	
00A910B6	00 00 00 00	
00A910B7	00 00 00 00	
00A910B8	00 00 00 00	
00A910B9	00 00 00 00	
00A910BA	00 00 00 00	
00A910BB	00 00 00 00	
00A910BC	00 00 00 00	
00A910BD	00 00 00 00	
00A910BE	00 00 00 00	
00A910BF	00 00 00 00	
00A910C0	00 00 00 00	
00A910C1	00 00 00 00	
00A910C2	00 00 00 00	
00A910C3	00 00 00 00	
00A910C4	00 00 00 00	
00A910C5	00 00 00 00	
00A910C6	00 00 00 00	
00A910C7	00 00 00 00	
00A910C8	00 00 00 00	
00A910C9	00 00 00 00	
00A910CA	00 00 00 00	
00A910CB	00 00 00 00	
00A910CC	00 00 00 00	
00A910CD	00 00 00 00	
00A910CE	00 00 00 00	
00A910CF	00 00 00 00	
00A910D0	00 00 00 00	
00A910D1	00 00 00 00	
00A910D2	00 00 00 00	
00A910D3	00 00 00 00	
00A910D4	00 00 00 00	
00A910D5	00 00 00 00	
00A910D6	00 00 00 00	
00A910D7	00 00 00 00	
00A910D8	00 00 00 00	
00A910D9	00 00 00 00	
00A910DA	00 00 00 00	
00A910DB	00 00 00 00	
00A910DC	00 00 00 00	
00A910DD	00 00 00 00	
00A910DE	00 00 00 00	
00A910DF	00 00 00 00	
00A910E0	00 00 00 00	
00A910E1	00 00 00 00	
00A910E2	00 00 00 00	
00A910E3	00 00 00 00	
00A910E4	00 00 00 00	
00A910E5	00 00 00 00	
00A910E6	00 00 00 00	
00A910E7	00 00 00 00	
00A910E8	00 00 00 00	
00A910E9	00 00 00 00	
00A910EA	00 00 00 00	
00A910EB	00 00 00 00	
00A910EC	00 00 00 00	
00A910ED	00 00 00 00	
00A910EE	00 00 00 00	
00A910EF	00 00 00 00	
00A910F0	00 00 00 00	
00A910F1	00 00 00 00	
00A910F2	00 00 00 00	
00A910F3	00 00 00 00	
00A910F4	00 00 00 00	
00A910F5	00 00 00 00	
00A910F6	00 00 00 00	
00A910F7	00 00 00 00	
00A910F8	00 00 00 00	
00A910F9	00 00 00 00	
00A910FA	00 00 00 00	
00A910FB	00 00 00 00	
00A910FC	00 00 00 00	
00A910FD	00 00 00 00	
00A910FE	00 00 00 00	
00A910FF	00 00 00 00	

Figure 17.23: OllyDbg: FNSTSW was executed

TEST is done:

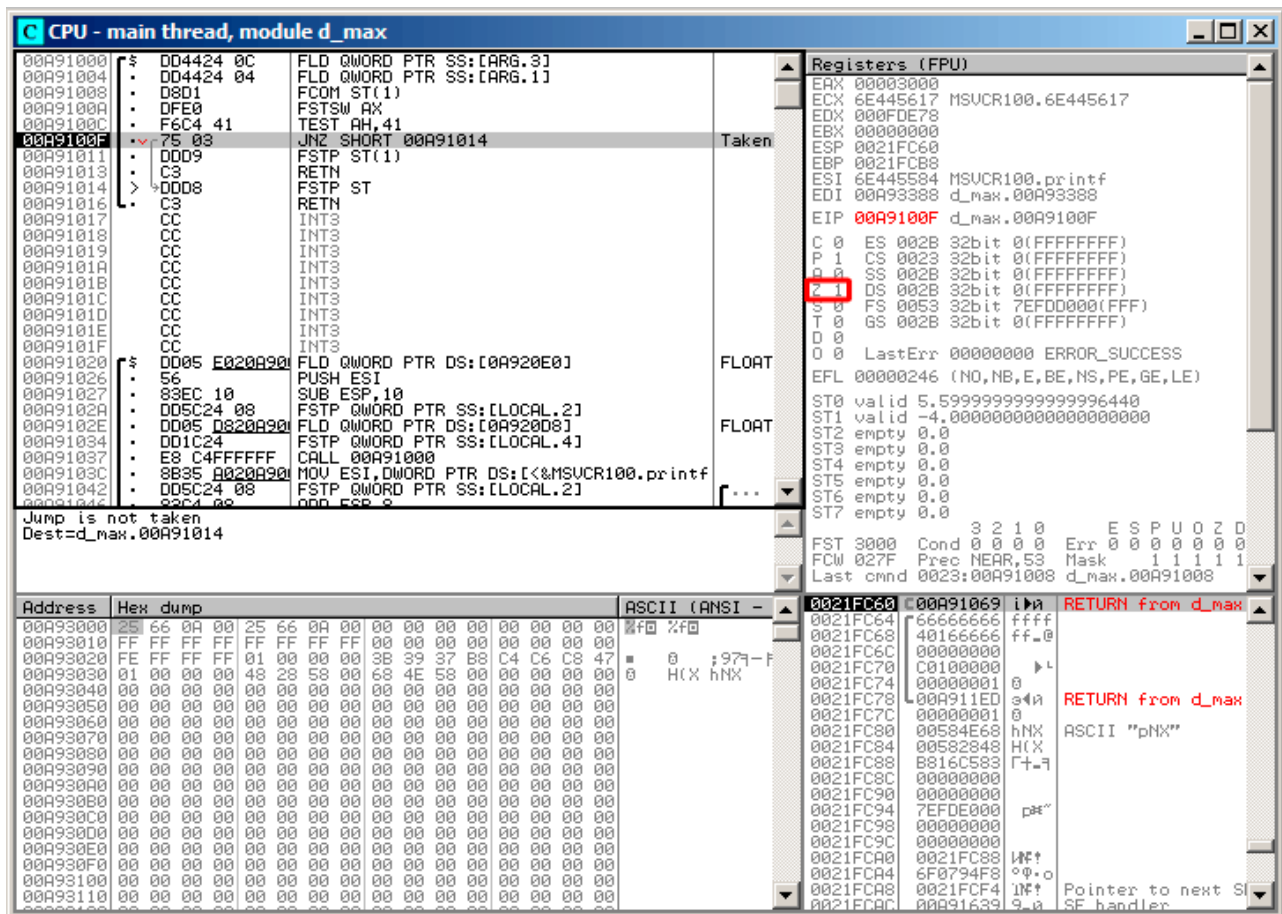


Figure 17.24: OllyDbg: TEST was executed

ZF=1, jump will not happen now.



FSTP ST(1) was executed: a value of 5.6 is now at the top of the FPU stack.

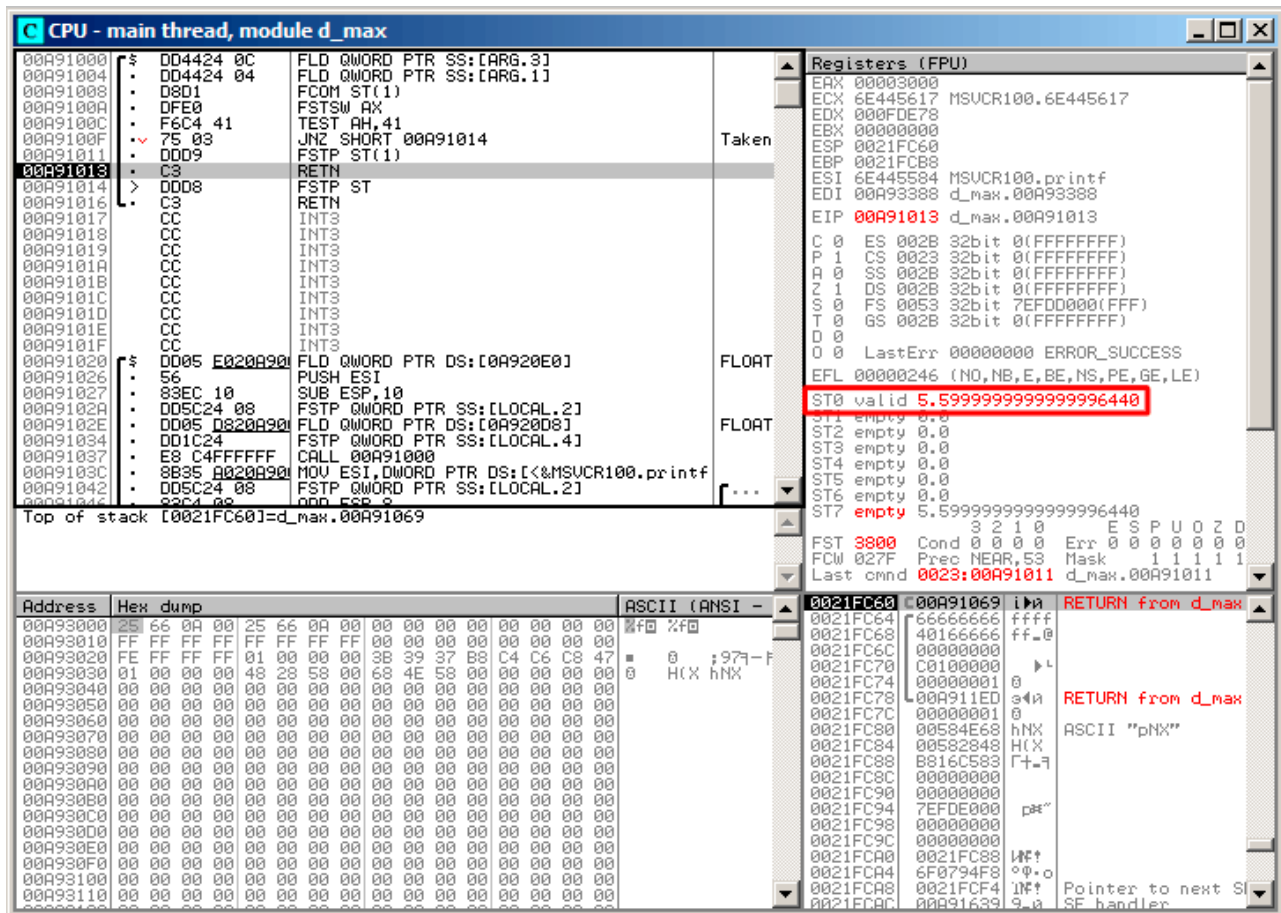


Figure 17.25: OllyDbg: FSTP was executed

We now see that the FSTP ST(1) instruction works as follows: it leaves what was at the top of the stack, but clears ST(1).

#### GCC 4.4.1

Listing 17.12: 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

; put a and b to local stack:

    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

; load a and b to FPU stack:
```

```

fld    [ebp+a]
fld    [ebp+b]

; current stack state: ST(0) - b; ST(1) - a

    fxch    st(1) ; this instruction swapping ST(1) and ST(0)

; current stack state: ST(0) - a; ST(1) - b

    fucompp    ; compare a and b and pop two values from stack, i.e., a and b
    fnstsw    ax ; store FPU status to AX
    sahf      ; load SF, ZF, AF, PF, and CF flags state from AH
    setnbe    al ; store 1 to AL, if CF=0 and ZF=0
    test     al, al      ; AL==0 ?
    jz       short loc_8048453 ; yes
    fld      [ebp+a]
    jmp      short locret_8048456

loc_8048453:
    fld      [ebp+b]

locret_8048456:
    leave
    retn
d_max endp

```

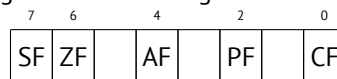
FUCOMPP is almost like FCOM, but pops both values from the stack and handles “not-a-numbers” differently.

A bit about *not-a-numbers*.

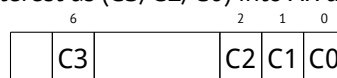
The FPU is able to deal with special values which are *not-a-numbers* or NaNs<sup>17</sup>. These are infinity, result of division by 0, etc. Not-a-numbers can be “quiet” and “signaling”. It is possible to continue to work with “quiet” NaNs, but if one tries to do any operation with “signaling” NaNs, an exception is to be raised.

FCOM raising an exception if any operand is NaN. FUCOM raising an exception only if any operand is a signaling NaN (SNaN).

The next instruction is SAHF (*Store AH into Flags*) –this is a rare instruction in code not related to the FPU. 8 bits from AH are moved into the lower 8 bits of the CPU flags in the following order:



Let’s recall that FNSTSW moves the bits that interest us (C3/C2/C0) into AH and they are in positions 6, 2, 0 of the AH register:



In other words, the `fnstsw ax / sahf` instruction pair moves C3/C2/C0 into ZF, PF and CF.

Now let’s also recall the values of C3/C2/C0 in different conditions:

- If  $a$  is greater than  $b$  in our example, then C3/C2/C0 are to be set to: 0, 0, 0.
- if  $a$  is less than  $b$ , then the bits are to be set to: 0, 0, 1.
- If  $a = b$ , then: 1, 0, 0.

In other words, these states of the CPU flags are possible after three FUCOMPP/FNSTSW/SAHF instructions:

- If  $a > b$ , the CPU flags are to be set as: ZF=0, PF=0, CF=0.
- If  $a < b$ , then the flags are to be set as: ZF=0, PF=0, CF=1.
- And if  $a = b$ , then: ZF=1, PF=0, CF=0.

Depending on the CPU flags and conditions, SETNBE stores 1 or 0 to AL. It is almost the counterpart of JNBE, with the exception that SETcc<sup>18</sup> stores 1 or 0 in AL, but Jcc does actually jump or not. SETNBE stores 1 only if CF=0 and ZF=0. If it is not true, 0 is to be stored into AL.

Only in one case both CF and ZF are 0: if  $a > b$ .

Then 1 is to be stored to AL, the subsequent JZ is not to be triggered and the function will return `_a`. In all other cases, `_b` is to be returned.

<sup>17</sup>[wikipedia.org/wiki/NaN](https://wikipedia.org/wiki/NaN)

<sup>18</sup>cc is condition code

## Optimizing GCC 4.4.1

Listing 17.13: Optimizing 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

; stack state now: ST(0) = _b, ST(1) = _a
           fxch    st(1)

; stack state now: ST(0) = _a, ST(1) = _b
           fucom   st(1) ; compare _a and _b
           fnstsw  ax
           sahf
           ja      short loc_8048448

; store ST(0) to ST(0) (idle operation), pop value at top of stack,
; leave _b at top
           fstp    st
           jmp     short loc_804844A

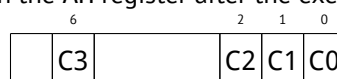
loc_8048448:
; store _a to ST(0), pop value at top of stack, leave _a at top
           fstp    st(1)

loc_804844A:
           pop     ebp
           retn
d_max      endp

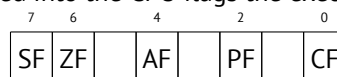
```

It is almost the same except that **JA** is used after **SAHF**. Actually, conditional jump instructions that check “larger”, “lesser” or “equal” for unsigned number comparison (these are **JA**, **JAE**, **JB**, **JBE**, **JE/JZ**, **JNA**, **JNAE**, **JNB**, **JNBE**, **JNE/JNZ**) check only flags **CF** and **ZF**.

Let’s recall where bits **C3/C2/C0** are located in the **AH** register after the execution of **FSTSW/FNSTSW**:



Let’s also recall, how the bits from **AH** are stored into the CPU flags the execution of **SAHF**:



After the comparison, the **C3** and **C0** bits are moved into **ZF** and **CF**, so the conditional jumps are able work after. **JA** is triggering if both **CF** are **ZF** zero.

Thereby, the conditional jumps instructions listed here can be used after a **FNSTSW/SAHF** instruction pair.

Apparently, the FPU **C3/C2/C0** status bits were placed there intentionally, to easily map them to base CPU flags without additional permutations?

### GCC 4.8.1 with -O3 optimization turned on

Some new FPU instructions were added in the P6 Intel family<sup>19</sup>. These are **FUCOMI** (compare operands and set flags of the main CPU) and **FCMOVcc** (works like **CMOVcc**, but on FPU registers). Apparently, the maintainers of GCC decided to drop support of pre-P6 Intel CPUs (early Pentiums, 80486, etc).

And also, the FPU is no longer separate unit in P6 Intel family, so now it is possible to modify/check flags of the main CPU from the FPU.

<sup>19</sup>Starting at Pentium Pro, Pentium-II, etc.

So what we get is:

Listing 17.14: Optimizing GCC 4.8.1

```

fld     QWORD PTR [esp+4]      ; load "a"
fld     QWORD PTR [esp+12]     ; load "b"
; ST0=b, ST1=a
fxch    st(1)
; ST0=a, ST1=b
; compare "a" and "b"
fucomi  st, st(1)
; copy ST1 ("b" here) to ST0 if a<=b
; leave "a" in ST0 otherwise
fcmovbe st, st(1)
; discard value in ST1
fstp    st(1)
ret

```

Hard to guess why FXCH (swap operands) is here. It's possible to get rid of it easily by swapping the first two FLD instructions or by replacing FCMOVBE (*below or equal*) by FCMOVA (*above*). Probably it's a compiler inaccuracy.

So FUCOMI compares ST(0) (*a*) and ST(1) (*b*) and then sets some flags in the main CPU. FCMOVBE checks the flags and copies ST(1) (*b* here at the moment) to ST(0) (*a* here) if  $ST0(a) \leq ST1(b)$ . Otherwise ( $a > b$ ), it leaves *a* in ST(0).

The last FSTP leaves ST(0) on top of the stack, discarding the contents of ST(1).

Let's trace this function in GDB:

Listing 17.15: Optimizing GCC 4.8.1 and GDB

```

1 dennis@ubuntuvms:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
2 dennis@ubuntuvms:~/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
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  0x3ffff99999999999800 +1.19999999999999956
35 =>R6: Valid  0x4000d99999999999800 +3.39999999999999911
36   R5: Empty  0x0000000000000000000
37   R4: Empty  0x0000000000000000000
38   R3: Empty  0x0000000000000000000
39   R2: Empty  0x0000000000000000000
40   R1: Empty  0x0000000000000000000
41   R0: Empty  0x0000000000000000000
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      0x000000000000000000000
58   R4: Empty      0x000000000000000000000
59   R3: Empty      0x000000000000000000000
60   R2: Empty      0x000000000000000000000
61   R1: Empty      0x000000000000000000000
62   R0: Empty      0x000000000000000000000
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
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 $eip
124 Dump of assembler code for function d_max:
125     0x080484a0 <+0>:    fldl    0x4(%esp)
126     0x080484a4 <+4>:    fldl    0xc(%esp)
127     0x080484a8 <+8>:    fxch    %st(1)
128     0x080484aa <+10>:   fucomi  %st(1),%st
129     0x080484ac <+12>:   fcmovbe %st(1),%st
130 => 0x080484ae <+14>:   fstp    %st(1)
131     0x080484b0 <+16>:   ret
132 End of assembler dump.
133 (gdb) ni
134 0x080484b0 in d_max ()
135 (gdb) info float
136 =>R7: Valid    0x4000d99999999999800 +3.39999999999999911
137   R6: Empty    0x4000d99999999999800
138   R5: Empty    0x00000000000000000000
139   R4: Empty    0x00000000000000000000
140   R3: Empty    0x00000000000000000000
141   R2: Empty    0x00000000000000000000
142   R1: Empty    0x00000000000000000000
143   R0: Empty    0x00000000000000000000
144
145 Status Word:          0x3800
146                      TOP: 7
147 Control Word:         0x037f   IM DM ZM OM UM PM
148                      PC: Extended Precision (64-bits)
149                      RC: Round to nearest
150 Tag Word:             0x3fff
151 Instruction Pointer:   0x73:0x080484ae
152 Operand Pointer:      0x7b:0xbffff118
153 Opcode:              0x0000
154 (gdb) quit
155 A debugging session is active.
156
157     Inferior 1 [process 30194] will be killed.
158
159 Quit anyway? (y or n) y
160 dennis@ubuntuv:~/polygon$

```

Using “ni”, let’s execute the first two FLD instructions.

Let’s examine the FPU registers (line 33).

As it was mentioned before, the FPU registers set is a circular buffer rather than a stack ([17.5.1 on page 213](#)). And GDB doesn’t show STx registers, but internal the FPU registers (Rx). The arrow (at line 35) points to the current top of the stack. You can also see the TOP register contents in *Status Word* (line 44)—it is 6 now, so the stack top is now pointing to internal register 6.

The values of *a* and *b* are swapped after FXCH is executed (line 54).

FUCOMI is executed (line 83). Let’s see the flags: CF is set (line 95).

FCMOVBE has copied the value of *b* (see line 104).

FSTP leaves one value at the top of stack (line 136). The value of TOP is now 7, so the FPU stack top is pointing to internal register 7.

## 17.7.2 ARM

### Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 17.16: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```

VMOV      D16, R2, R3 ; b
VMOV      D17, R0, R1 ; a
VCMPE.F64 D17, D16
VMRS      APSR_nzcv, FPSCR
VMOVGT.F64 D16, D17 ; copy "b" to D16
VMOV      R0, R1, D16
BX        LR

```

A very simple case. The input values are placed into the D17 and D16 registers and then compared using the VCMPE instruction. Just like in the x86 coprocessor, the ARM coprocessor has its own status and flags register ([FPSCR<sup>20</sup>](#)), since there is a need to store coprocessor-specific flags. And just like in x86, there are no conditional jump instruction in ARM, that can check bits in the status register of the coprocessor. So there is VMRS, which copies 4 bits (N, Z, C, V) from the coprocessor status word into bits of the *general* status register ([APSR<sup>21</sup>](#)).

VMOVGT is the analog of the MOVGT, instruction for D-registers, it executes if one operand is greater than the other while comparing (*GT—Greater Than*).

If it gets executed, the value of *b* is to be written into D16(that is currently stored in in D17).

Otherwise the value of *a* stays in the D16 register.

The penultimate instruction VMOV prepares the value in the D16 register for returning it via the R0 and R1 register pair.

### Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

Listing 17.17: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```

VMOV      D16, R2, R3 ; b
VMOV      D17, R0, R1 ; a
VCMPE.F64 D17, D16
VMRS      APSR_nzcv, FPSCR
IT GT
VMOVGT.F64 D16, D17
VMOV      R0, R1, D16
BX        LR

```

Almost the same as in the previous example, however slightly different. As we already know, many instructions in ARM mode can be supplemented by condition predicate.

But there is no such thing in Thumb mode. There is no space in the 16-bit instructions for 4 more bits in which conditions can be encoded.

However, Thumb-2 was extended to make it possible to specify predicates to old Thumb instructions.

Here, in the [IDA](#)-generated listing, we see the VMOVGT instruction, as in previous example.

In fact, the usual VMOV is encoded there, but [IDA](#) adds the -GT suffix to it, since there is a “IT GT” instruction placed right before it.

The IT instruction defines a so-called *if-then block*. After the instruction it is possible to place up to 4 instructions, each of them has a predicate suffix. In our example, IT GT implies that the next instruction is to be executed, if the *GT (Greater Than)* condition is true.

Here is a more complex code fragment, by the way, from Angry Birds (for iOS):

Listing 17.18: Angry Birds Classic

```

...
ITE NE
VMOVNE    R2, R3, D16
VMOVEQ    R2, R3, D17
BLX       _objc_msgSend ; not prefixed
...

```

ITE stands for *if-then-else* and it encodes suffixes for the next two instructions. The first instruction executes if the condition encoded in ITE (*NE, not equal*) is true at, and the second—if the condition is not true. (The inverse condition of NE is EQ (*equal*)).

The instruction followed after the second VMOV (or VMOVEQ) is a normal one, not prefixed (BLX).

<sup>20</sup>(ARM) Floating-Point Status and Control Register

<sup>21</sup>(ARM) Application Program Status Register

One more that's slightly harder, which is also from Angry Birds:

Listing 17.19: Angry Birds Classic

```
...
ITTTT EQ
MOVEQ      R0, R4
ADDEQ      SP, SP, #0x20
POPEQ.W    {R8,R10}
POPEQ      {R4-R7,PC}
BLX        __stack_chk_fail ; not prefixed
...
```

Four “T” symbols in the instruction mnemonic mean that the four subsequent instructions are to be executed if the condition is true. That's why [IDA](#) adds the -EQ suffix to each one of them.

And if there was be, for example, ITEEE EQ (*if-then-else-else-else*), then the suffixes would have been set as follows:

```
-EQ
-NE
-NE
-NE
```

Another fragment from Angry Birds:

Listing 17.20: Angry Birds Classic

```
...
CMP.W      R0, #0xFFFFFFFF
ITTE LE
SUBLE.W    R10, R0, #1
NEGLE      R0, R0
MOVGT      R10, R0
MOVS       R6, #0 ; not prefixed
CBZ        R0, loc_1E7E32 ; not prefixed
...
```

ITTE (*if-then-then-else*) implies that the 1st and 2nd instructions are to be executed if the LE (*Less or Equal*) condition is true, and the 3rd—if the inverse condition (GT—*Greater Than*) is true.

Compilers usually don't generate all possible combinations. For example, in the mentioned Angry Birds game (*classic* version for iOS) only these variants of the IT instruction are used: IT, ITE, ITT, ITTE, ITTT, ITTTT. How to learn this? In [IDA](#) It is possible to produce listing files, so it was created with an option to show 4 bytes for each opcode. Then, knowing the high part of the 16-bit opcode (IT is 0xBF), we do the following using `grep`:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

By the way, if you program in ARM assembly language manually for Thumb-2 mode, and you add conditional suffixes, the assembler will add the IT instructions automatically with the required flags where it is necessary.

### Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 17.21: Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
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

Almost the same as we already saw, but there is too much redundant code because the *a* and *b* variables are stored in the local stack, as well as the return value.

### Optimizing Keil 6/2013 (Thumb mode)

Listing 17.22: Optimizing Keil 6/2013 (Thumb mode)

	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
	POP	{R3-R7, PC}

Keil doesn't generate FPU-instructions since it cannot rely on them being supported on the target CPU, and it cannot be done by straightforward bitwise comparing. So it calls an external library function to do the comparison: `__aeabi_cdrcmple`.

N.B. The result of the comparison is to be left in the flags by this function, so the following `BCS` (*Carry set—Greater than or equal*) instruction can work without any additional code.

## 17.7.3 ARM64

### Optimizing GCC (Linaro) 4.9

```
d_max:
; D0 - a, D1 - b
    fcmpe    d0, d1
    fcsel    d0, d0, d1, gt
; now result in D0
    ret
```

The ARM64 ISA has FPU-instructions which set `APSR` the CPU flags instead of `FPSCR` for convenience. The `FPU` is not a separate device here anymore (at least, logically). Here we see `FCMPE`. It compares the two values passed in `D0` and `D1` (which are the first and second arguments of the function) and sets `APSR` flags (N, Z, C, V).

`FCSEL` (*Floating Conditional Select*) copies the value of `D0` or `D1` into `D0` depending on the condition (GT—Greater Than), and again, it uses flags in `APSR` register instead of `FPSCR`. This is much more convenient, compared to the instruction set in older CPUs.

If the condition is true (GT), then the value of `D0` is copied into `D0` (i.e., nothing happens). If the condition is not true, the value of `D1` is copied into `D0`.

**Non-optimizing GCC (Linaro) 4.9**

```

d_max:
; save input arguments in "Register Save Area"
    sub    sp, sp, #16
    str    d0, [sp,8]
    str    d1, [sp]
; reload values
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    fmov   d0, x1
    fmov   d1, x0
; D0 - a, D1 - b
    fcmpe  d0, d1
    ble    .L76
; a>b; load D0 (a) into X0
    ldr    x0, [sp,8]
    b      .L74
.L76:
; a<=b; load D1 (b) into X0
    ldr    x0, [sp]
.L74:
; result in X0
    fmov   d0, x0
; result in D0
    add    sp, sp, 16
    ret

```

Non-optimizing GCC is more verbose. First, the function saves its input argument values in the local stack (*Register Save Area*). Then the code reloads these values into registers X0/X1 and finally copies them to D0/D1 to be compared using FCMPE. A lot of redundant code, but that is how non-optimizing compilers work. FCMPE compares the values and sets the [APSR](#) flags. At this moment, the compiler is not thinking yet about the more convenient FCSEL instruction, so it proceed using old methods: using the BLE instruction (*Branch if Less than or Equal*). In the first case ( $a > b$ ), the value of  $a$  gets loaded into X0. In the other case ( $a \leq b$ ), the value of  $b$  gets loaded into X0. Finally, the value from X0 gets copied into D0, because the return value needs to be in this register.

**Exercise**

As an exercise, you can try optimizing this piece of code manually by removing redundant instructions and not introducing new ones (including FCSEL).

**Optimizing GCC (Linaro) 4.9–float**

Let's also rewrite this example to use *float* instead of *double*.

```

float f_max (float a, float b)
{
    if (a>b)
        return a;

    return b;
};

```

```

f_max:
; S0 - a, S1 - b
    fcmpe  s0, s1
    fcsl   s0, s0, s1, gt
; now result in S0
    ret

```

It is the same code, but the S-registers are used instead of D- ones. It's because numbers of type *float* are passed in 32-bit S-registers (which are in fact the lower parts of the 64-bit D-registers).

## 17.7.4 MIPS

The co-processor of the MIPS processor has a condition bit which can be set in the FPU and checked in the CPU. Earlier MIPS-es have only one condition bit (called FCC0), later ones have 8 (called FCC7-FCC0). This bit (or bits) are located in the register called FCCR.

Listing 17.23: Optimizing GCC 4.4.5 (IDA)

```
d_max:
; set FPU condition bit if $f14<$f12 (b<a):
    c.lt.d  $f14, $f12
    or      $at, $zero ; NOP
; jump to locret_14 if condition bit is set
    bc1t    locret_14
; this instruction is always executed (set return value to "a"):
    mov.d   $f0, $f12 ; branch delay slot
; this instruction is executed only if branch was not taken (i.e., if b>=a)
; set return value to "b":
    mov.d   $f0, $f14

locret_14:
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP
```

C.LT.D compares two values. LT is the condition “Less Than”. D implies values of type *double*. Depending on the result of the comparison, the FCC0 condition bit is either set or cleared.

BC1T checks the FCC0 bit and jumps if the bit is set. T mean that the jump is to be taken if the bit is set (“True”). There is also the instruction “BC1F” which jumps if the bit is cleared (“False”).

Depending on the jump, one of function arguments is placed into \$F0.

## 17.8 Stack, calculators and reverse Polish notation

Now we understand why some old calculators used reverse Polish notation <sup>22</sup>. For example, for addition of 12 and 34 one has to enter 12, then 34, then press “plus” sign. It’s because old calculators were just stack machine implementations, and this was much simpler than to handle complex parenthesized expressions.

## 17.9 x64

On how floating point numbers are processed in x86-64, read more here: [27 on page 412](#).

## 17.10 Exercises

- <http://challenges.re/60>
- <http://challenges.re/61>

<sup>22</sup>[wikipedia.org/wiki/Reverse\\_Polish\\_notation](http://wikipedia.org/wiki/Reverse_Polish_notation)

# Chapter 18

## Arrays

An array is just a set of variables in memory that lie next to each other and that have the same type<sup>1</sup>.

### 18.1 Simple example

```
#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;
};
```

#### 18.1.1 x86

##### MSVC

Let's compile:

Listing 18.1: 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
```

<sup>1</sup>AKA<sup>2</sup> "homogeneous container"

```

    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

```

Nothing very special, just two loops: the first is a filling loop and second is a printing loop. The `shl ecx, 1` instruction is used for value multiplication by 2 in ECX, more about below [16.2.1 on page 205](#).

80 bytes are allocated on the stack for the array, 20 elements of 4 bytes.

Let's try this example in OllyDbg.

We see how the array gets filled: each element is 32-bit word of *int* type and its value is the index multiplied by 2:

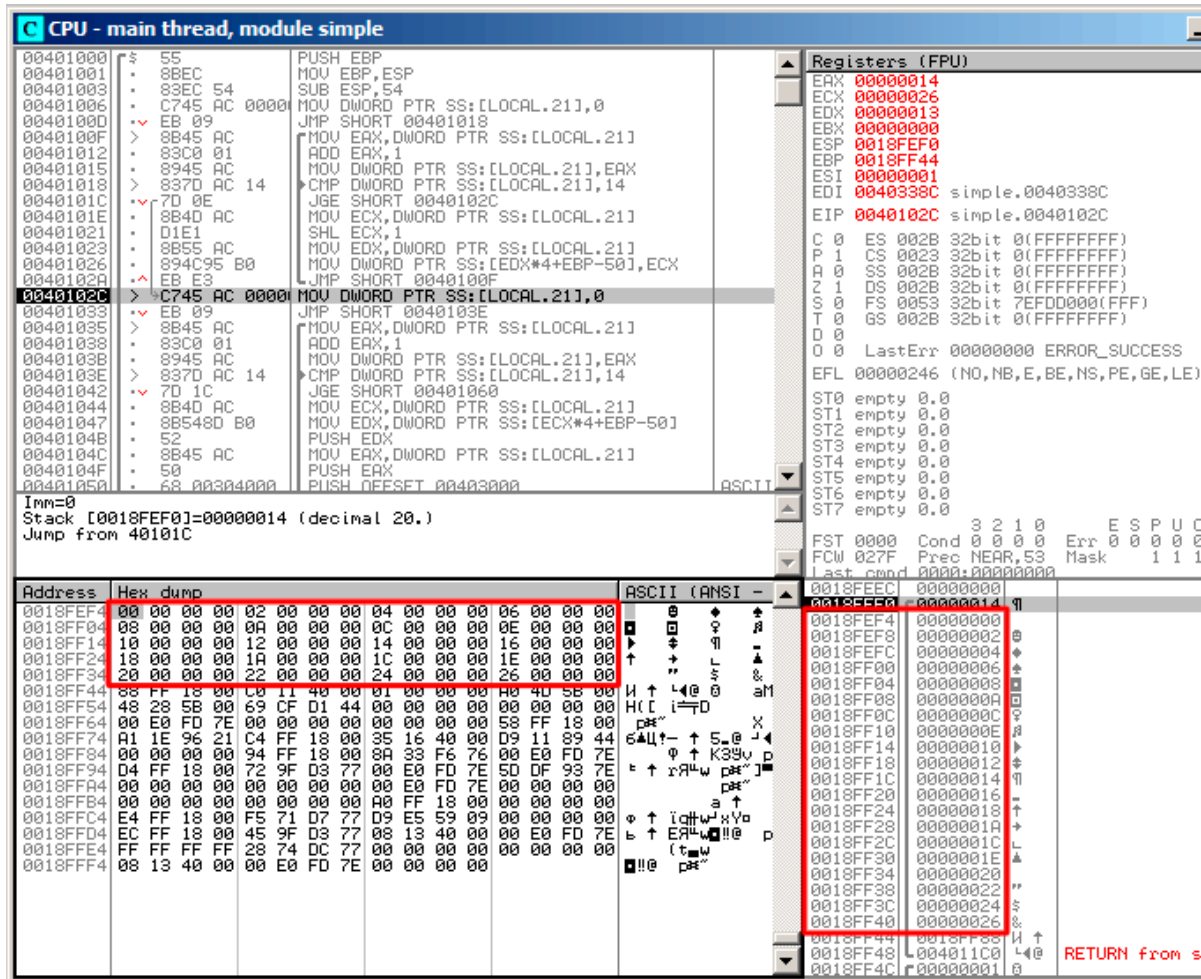


Figure 18.1: OllyDbg: after array filling

Since this array is located in the stack, we can see all its 20 elements there.

## GCC

Here is what GCC 4.4.1 does:

Listing 18.2: 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
```

```

        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

```

By the way, variable *a* is of type *int\** (the pointer to *int*)—you can pass a pointer to an array to another function, but it's more correct to say that a pointer to the first element of the array is passed (the addresses of rest of the elements are calculated in an obvious way). If you index this pointer as *a[idx]*, *idx* is just to be added to the pointer and the element placed there (to which calculated pointer is pointing) is to be returned.

An interesting example: a string of characters like “*string*” is an array of characters and it has a type of *const char[]*. An index can also be applied to this pointer. And that is why it is possible to write things like “*string*”[*i*]—this is a correct C/C++ expression!

## 18.1.2 ARM

### Non-optimizing Keil 6/2013 (ARM mode)

```

EXPORT _main
_main
    STMFD      SP!, {R4,LR}
    SUB        SP, SP, #0x50           ; allocate place for 20 int variables

; first loop

    MOV        R4, #0                  ; i
    B          loc_4A0

loc_494
    MOV        R0, R4,LSL#1             ; R0=R4*2
    STR        R0, [SP,R4,LSL#2]        ; store R0 to SP+R4<<2 (same as SP+R4*4)
    ADD        R4, R4, #1              ; i=i+1

loc_4A0
    CMP        R4, #20                 ; i<20?
    BLT        loc_494                 ; yes, run loop body again

; second loop

    MOV        R4, #0                  ; i
    B          loc_4B0

loc_4B0
    LDR        R2, [SP,R4,LSL#2]        ; (second printf argument) R2=*(SP+R4<<4) (same as
    ; *(SP+R4*4))
    MOV        R1, R4                  ; (first printf argument) 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             ; yes, run loop body again
        MOV     R0, #0              ; value to return
        ADD     SP, SP, #0x50       ; deallocate chunk, allocated for 20 int variables
        LDMFD   SP!, {R4,PC}

```

*int* type requires 32 bits for storage (or 4 bytes), so to store 20 *int* variables 80 (0x50) bytes are needed. So that is why the `SUB SP, SP, #0x50` instruction in the function's prologue allocates exactly this amount of space in the stack.

In both the first and second loops, the loop iterator *i* is placed in the R4 register.

The number that is to be written into the array is calculated as  $i * 2$ , which is effectively equivalent to shifting it left by one bit, so `MOV R0, R4, LSL#1` instruction does this.

`STR R0, [SP, R4, LSL#2]` writes the contents of R0 into the array. Here is how a pointer to array element is calculated: `SP` points to the start of the array, R4 is *i*. So shifting *i* left by 2 bits is effectively equivalent to multiplication by 4 (since each array element has a size of 4 bytes) and then it's added to the address of the start of the array.

The second loop has an inverse `LDR R2, [SP, R4, LSL#2]` instruction. It loads the value we need from the array, and the pointer to it is calculated likewise.

### Optimizing Keil 6/2013 (Thumb mode)

```

_main
        PUSH    {R4,R5,LR}
; allocate place for 20 int variables + one more variable
        SUB     SP, SP, #0x54

; first loop

        MOVS    R0, #0              ; i
        MOV     R5, SP              ; pointer to first array element

loc_1CE
        LSLS    R1, R0, #1          ; R1=i<<1 (same as i*2)
        LSLS    R2, R0, #2          ; R2=i<<2 (same as i*4)
        ADDS    R0, R0, #1          ; i=i+1
        CMP     R0, #20             ; i<20?
        STR     R1, [R5,R2]         ; store R1 to *(R5+R2) (same R5+i*4)
        BLT     loc_1CE             ; yes, i<20, run loop body again

; second loop

loc_1DC
        MOVS    R4, #0              ; i=0

        LSLS    R0, R4, #2          ; R0=i<<2 (same as i*4)
        LDR     R2, [R5,R0]         ; load from *(R5+R0) (same as 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             ; yes, i<20, run loop body again
        MOVS    R0, #0              ; value to return
; deallocate chunk, allocated for 20 int variables + one more variable
        ADD     SP, SP, #0x54
        POP     {R4,R5,PC}

```

Thumb code is very similar. Thumb mode has special instructions for bit shifting (like `LSLS`), which calculates the value to be written into the array and the address of each element in the array as well.

The compiler allocates slightly more space in the local stack, however, the last 4 bytes are not used.

### Non-optimizing GCC 4.9.1 (ARM64)



Listing 18.3: Non-optimizing GCC 4.9.1 (ARM64)

```

.LC0:
    .string "a[%d]=%d\n"
main:
; save FP and LR in stack frame:
    stp    x29, x30, [sp, -112]!
; set stack frame (FP=SP)
    add    x29, sp, 0
; setting initial counter variable at 0 (WZR is the register always holding zero):
    str    wzr, [x29,108]
; jump to loop condition checking code:
    b      .L2
.L3:
; load value of "i" variable:
    ldr    w0, [x29,108]
; multiply it by 2:
    lsl    w2, w0, 1
; find a place of an array in local stack:
    add    x0, x29, 24
; load 32-bit integer from local stack and sign extend it to 64-bit one:
    ldrsw  x1, [x29,108]
; calculate address of element (X0+X1<<2=array address+i*4) and store W2 (i*2) there:
    str    w2, [x0,x1,lsl 2]
; increment counter (i):
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L2:
; check if we finished:
    ldr    w0, [x29,108]
    cmp    w0, 19
; jump to L3 (loop body begin) if not:
    ble    .L3
; second part of the function begins here.
; setting initial counter variable at 0.
; by the way, the same place in the local stack was used for counter,
; because the same local variable (i) is being used as counter.
    str    wzr, [x29,108]
    b      .L4
.L5:
; calculate array address:
    add    x0, x29, 24
; load "i" value:
    ldrsw  x1, [x29,108]
; load value from the array at the address (X0+X1<<2 = address of array + i*4)
    ldr    w2, [x0,x1,lsl 2]
; load address of the "a[%d]=%d\n" string:
    adrp   x0, .LC0
    add    x0, x0, :lo12:LC0
; load "i" variable to W1 and pass it to printf() as second argument:
    ldr    w1, [x29,108]
; W2 still contains the value of array element which was just loaded.
; call printf():
    bl     printf
; increment "i" variable:
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L4:
; are we finished?
    ldr    w0, [x29,108]
    cmp    w0, 19
; jump to the loop body begin if not:
    ble    .L5
; return 0
    mov    w0, 0
; restore FP and LR:
    ldp    x29, x30, [sp], 112
    ret

```

**18.1.3 MIPS**

The function uses a lot of S- registers which must be preserved, so that's why its values are saved in the function prologue and restored in the epilogue.

Listing 18.4: Optimizing 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
; function prologue:
    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
; that value will be used as a loop terminator.
; it was precalculated by GCC compiler at compile stage:
    li      $a0, 0x28 # '('

loc_34:                                # CODE XREF: main+3C
; store value into memory:
    sw      $v0, 0($v1)
; increase value to be stored by 2 at each iteration:
    addiu   $v0, 2
; loop terminator reached?
    bne     $v0, $a0, loc_34
; add 4 to address anyway:
    addiu   $v1, 4
; array filling loop is ended
; second loop begin
    la      $s3, $LC0                # "a[%d]=%d\n"
; "i" variable will reside in $s0:
    move     $s0, $zero
    li      $s2, 0x14

loc_54:                                # CODE XREF: main+70
; call printf():
    lw      $t9, (printf & 0xFFFF)($gp)
    lw      $a2, 0($s1)
    move     $a1, $s0
    move     $a0, $s3
    jalr     $t9
; increment "i":
    addiu   $s0, 1
    lw      $gp, 0x80+var_70($sp)
; jump to loop body if end is not reached:
    bne     $s0, $s2, loc_54
; move memory pointer to the next 32-bit word:
    addiu   $s1, 4
; function epilogue
    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
$LC0:    .ascii  "a[%d]=%d\n"<0>    # DATA XREF: main+44

```

Something interesting: there are two loops and the first one doesn't need  $i$ , it needs only  $i * 2$  (increased by 2 at each iteration) and also the address in memory (increased by 4 at each iteration). So here we see two variables, one (in  $\$V0$ ) increasing by 2 each time, and another (in  $\$V1$ ) – by 4.

The second loop is where `printf()` is called and it reports the value of  $i$  to the user, so there is a variable which is increased by 1 each time (in  $\$S0$ ) and also a memory address (in  $\$S1$ ) increased by 4 each time.

That reminds us of loop optimizations we considered earlier: [39 on page 461](#). Their goal is to get rid of multiplications.

## 18.2 Buffer overflow

### 18.2.1 Reading outside array bounds

So, array indexing is just `array[index]`. If you study the generated code closely, you'll probably note the missing index bounds checking, which could check *if it is less than 20*. What if the index is 20 or greater? That's the one C/C++ feature it is often blamed for.

Here is a code that successfully compiles and works:

```

#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;
};

```

Compilation results (MSVC 2008):

Listing 18.5: Non-optimizing 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 $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
    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]=%d'
    call    DWORD PTR __imp__printf

```

```
    add     esp, 8
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main      ENDP
_TEXT      ENDS
END
```

The code produced this result:

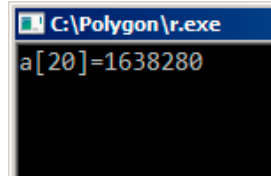


Figure 18.2: OllyDbg: console output

It is just *something* that was lying in the stack near to the array, 80 bytes away from its first element.

Let's try to find out where did this value come from, using OllyDbg. Let's load and find the value located right after the last array element:

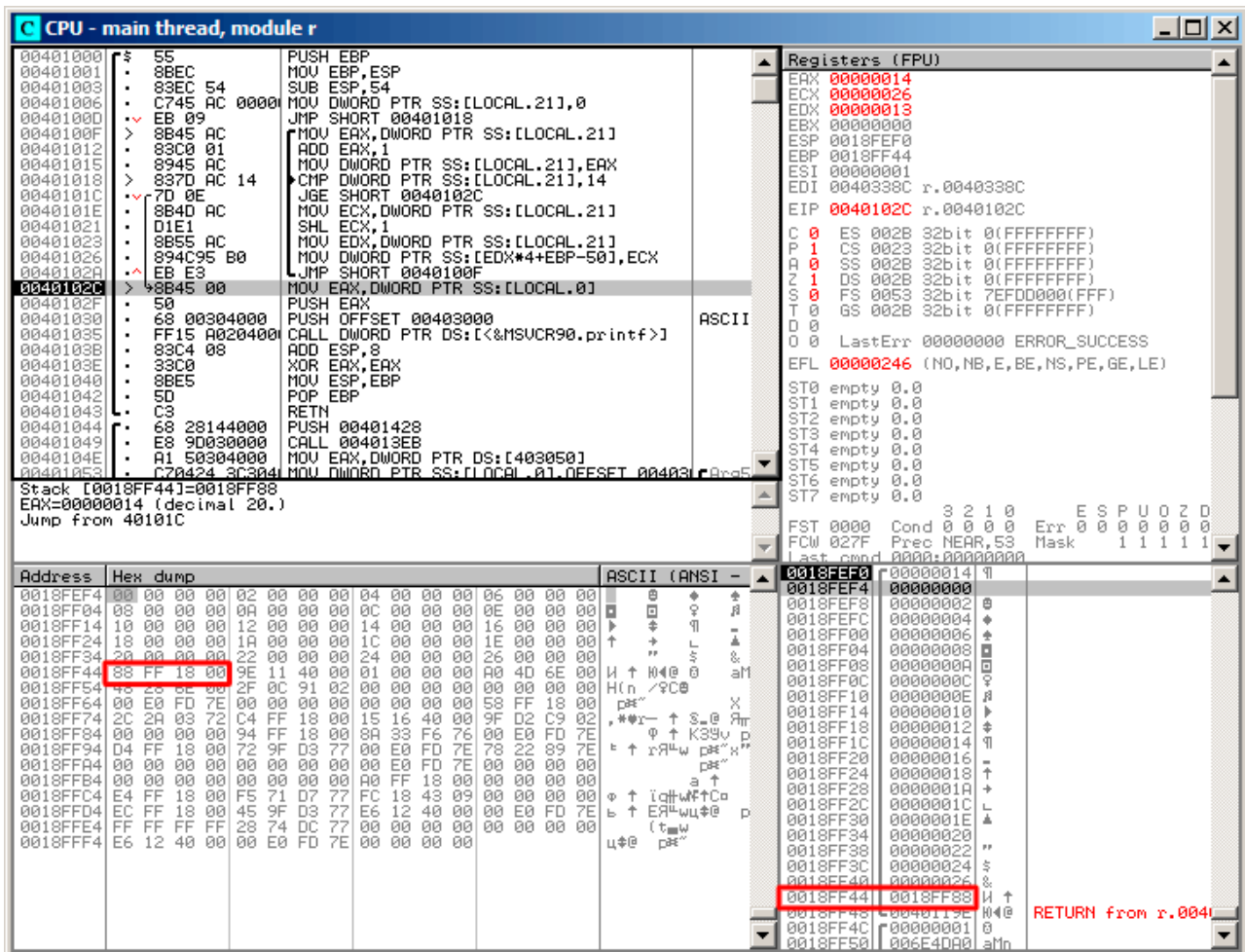


Figure 18.3: OllyDbg: reading of the 20th element and execution of printf()

What is this? Judging by the stack layout, this is the saved value of the EBP register.

Let's trace further and see how it gets restored:

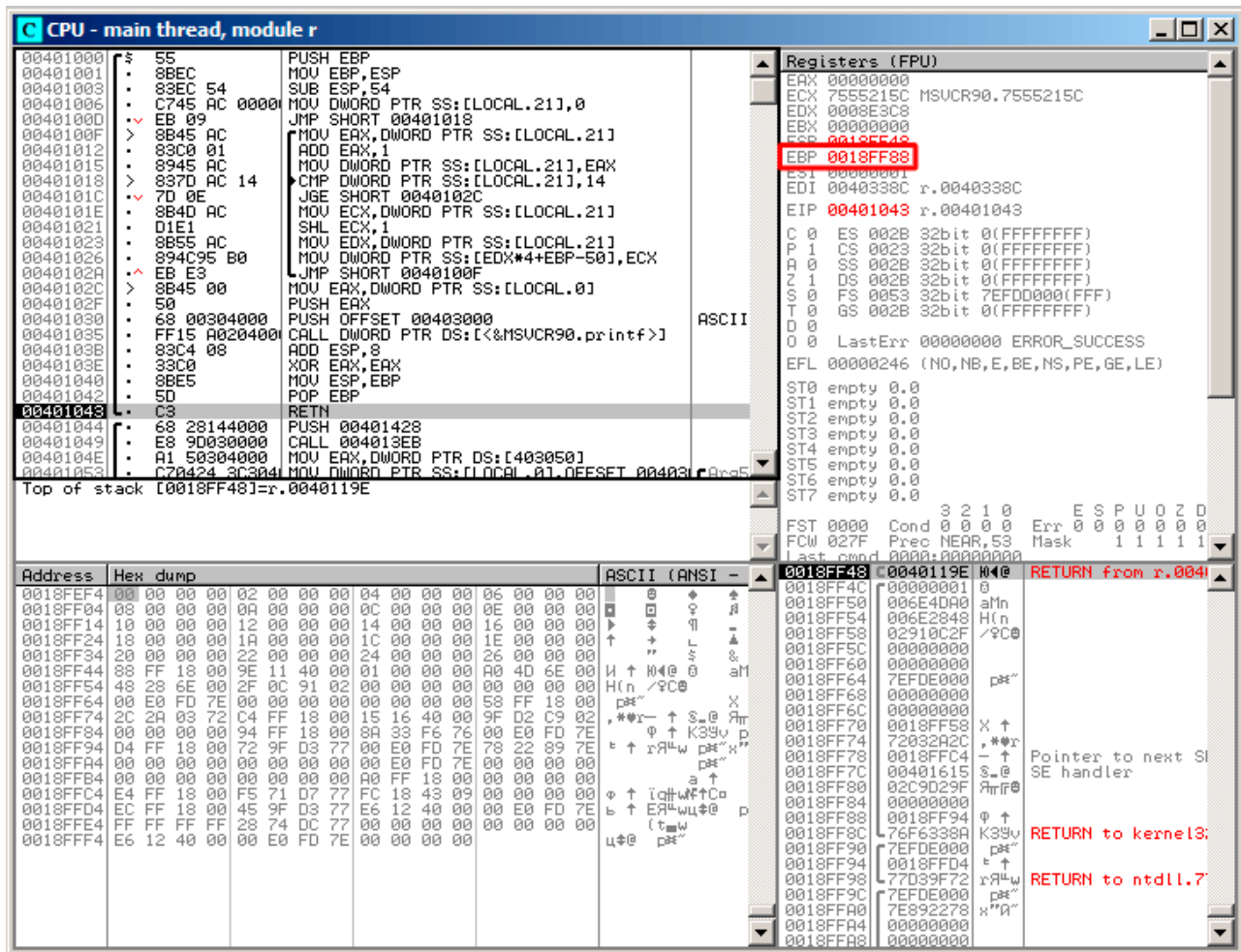


Figure 18.4: OllyDbg: restoring value of EBP

Indeed, how it could be different? The compiler may generate some additional code to check the index value to be always in the array's bounds (like in higher-level programming languages<sup>3</sup>) but this makes the code slower.

## 18.2.2 Writing beyond array bounds

OK, we read some values from the stack *illegally*, but what if we could write something to it?

Here is what we have got:

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<30; i++)
        a[i]=i;

    return 0;
};
```

MSVC

And what we get:

<sup>3</sup>Java, Python, etc

## Listing 18.6: Non-optimizing MSVC 2008

```
_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$[ebp] ; that instruction is obviously redundant
mov      DWORD PTR _a$[ebp+ecx*4], edx ; ECX could be used as second operand here instead
jmp      SHORT $LN2@main
$LN1@main:
xor      eax, eax
mov      esp, ebp
pop      ebp
ret      0
_main    ENDP
```

The compiled program crashes after running. No wonder. Let's see where exactly does it is crash.

Let's load it into OllyDbg, and trace until all 30 elements are written:

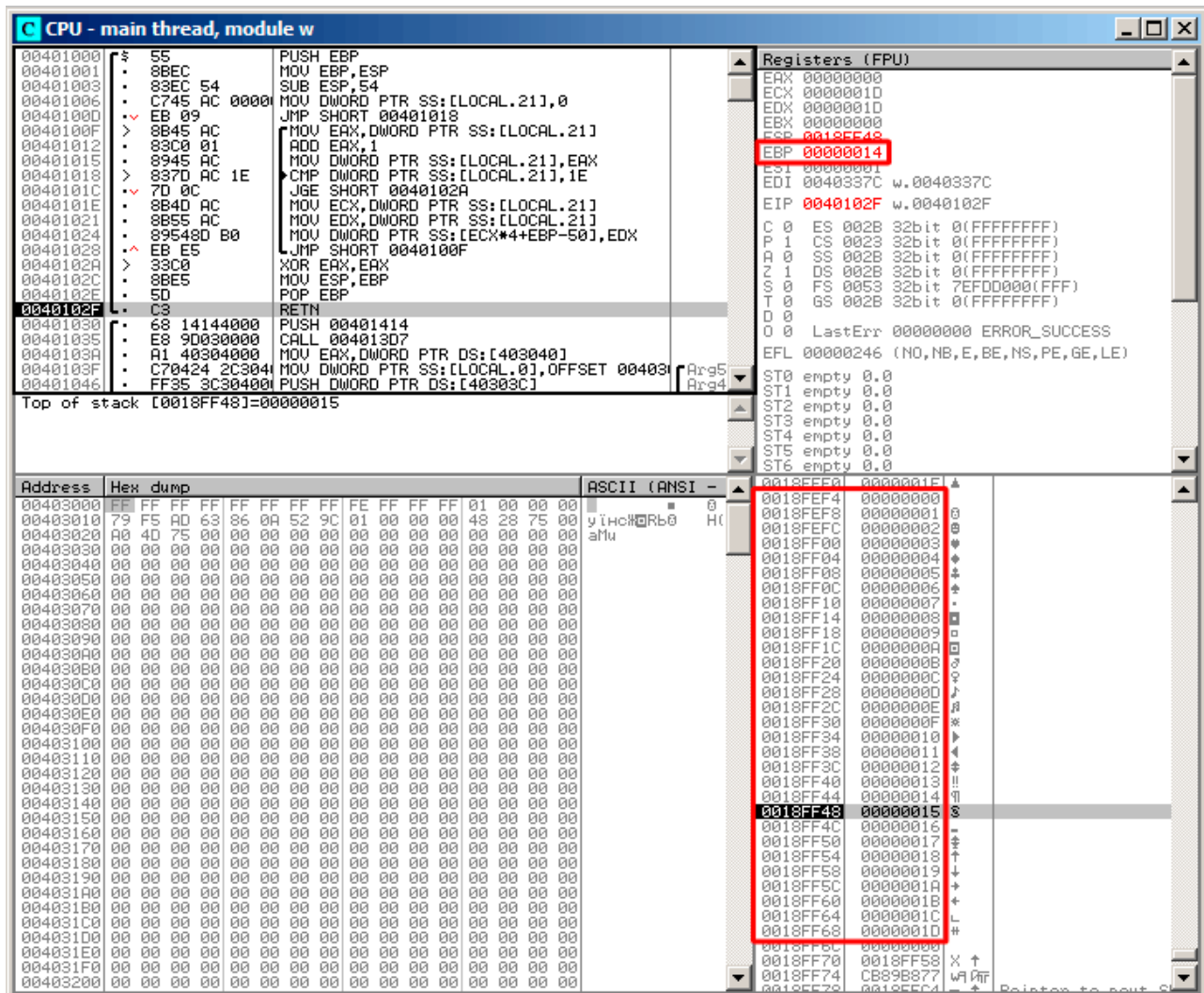


Figure 18.5: OllyDbg: after restoring the value of EBP



Trace until the function end:

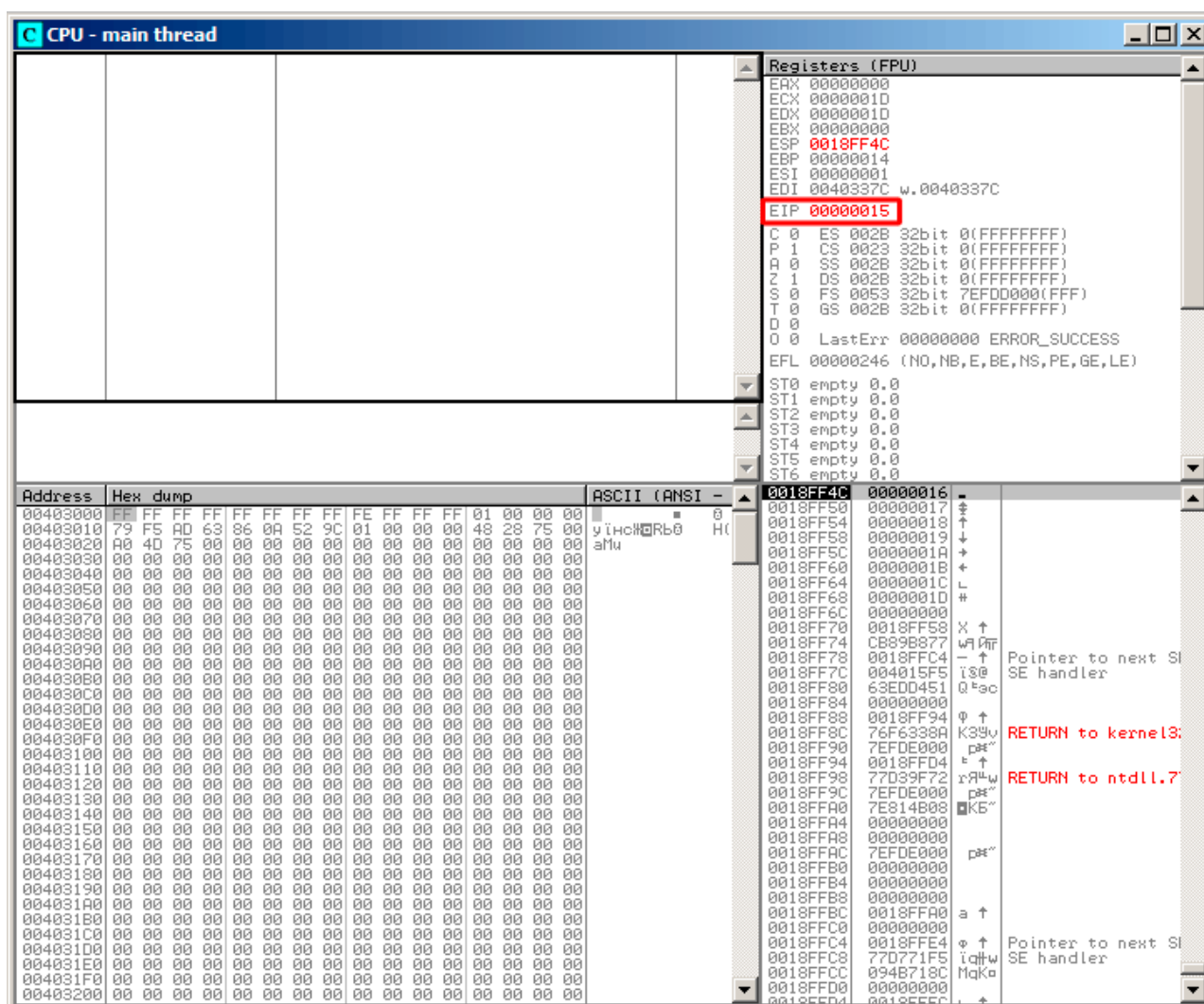


Figure 18.6: OllyDbg: EIP was restored, but OllyDbg can't disassemble at 0x15

Now please keep your eyes on the registers.

EIP is 0x15 now. It is not a legal address for code—at least for win32 code! We got there somehow against our will. It is also interesting that the EBP register contain 0x14, ECX and EDX—0x1D.

Let's study stack layout a bit more.

After the control flow was passed to `main()`, the value in the EBP register was saved on the stack. Then, 84 bytes were allocated for the array and the `i` variable. That's  $(20+1)*\text{sizeof}(\text{int})$ . ESP now points to the `_i` variable in the local stack and after the execution of the next PUSH something, *something* is appearing next to `_i`.

That's the stack layout while the control is in `main()`:

ESP	4 bytes allocated for <i>i</i> variable
ESP+4	80 bytes allocated for <code>a[20]</code> array
ESP+84	saved EBP value
ESP+88	return address

`a[19]=something` statement writes the last *int* in the bounds of the array (in bounds so far!)

`a[20]=something` statement writes *something* to the place where the value of EBP is saved.

Please take a look at the register state at the moment of the crash. In our case, 20 was written in the 20th element. At the function end, the function epilogue restores the original EBP value. (20 in decimal is 0x14 in hexadecimal). Then RET gets executed, which is effectively equivalent to POP EIP instruction.

The RET instruction takes the return address from the stack (that is the address in CRT), which was called `main()`, and 21 is stored there (0x15 in hexadecimal). The CPU traps at address 0x15, but there is no executable code there, so exception gets raised.

Welcome! It is called a *buffer overflow*<sup>4</sup>.

Replace the *int* array with a string (*char* array), create a long string deliberately and pass it to the program, to the function, which doesn't check the length of the string and copies it in a short buffer, and you'll be able to point the program to an address to which it must jump. It's not that simple in reality, but that is how it emerged<sup>5</sup>

## GCC

Let's try the same code in GCC 4.4.1. We get:

```

main          public main
              proc near
a
i             = dword ptr -54h
              = 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

```

Running this in Linux will produce: Segmentation fault.

If we run this in the GDB debugger, we get this:

```

(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)

```

The register values are slightly different than in win32 example, since the stack layout is slightly different too.

<sup>4</sup>[wikipedia](#)

<sup>5</sup>Classic article about it: [\[One96\]](#).

## 18.3 Buffer overflow protection methods

There are several methods to protect against this scourge, regardless of the C/C++ programmers' negligence. MSVC has options like<sup>6</sup>:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

One of the methods is to write a random value between the local variables in stack at function prologue and to check it in function epilogue before the function exits. If value is not the same, do not execute the last instruction RET, but stop (or hang). The process will halt, but that is much better than a remote attack to your host.

This random value is called a “canary” sometimes, it is related to the miners' canary<sup>7</sup>, they were used by miners in the past days in order to detect poisonous gases quickly. Canaries are very sensitive to mine gases, they become very agitated in case of danger, or even die.

If we compile our very simple array example ( 18.1 on page 254) in MSVC with RTC1 and RTCs option, you can see a call to `@_RTC_CheckStackVars@8` a function at the end of the function that checks if the “canary” is correct.

Let's see how GCC handles this. Let's take an `alloca()` ( 5.2.4 on page 27) example:

```
#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);
};
```

By default, without any additional options, GCC 4.7.3 inserts a “canary” check into the code:

Listing 18.7: 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 ; canary
    mov     DWORD PTR [ebp-12], eax
    xor     eax, eax
    call    _snprintf
    mov     DWORD PTR [esp], ebx
    call    puts
    mov     eax, DWORD PTR [ebp-12]
    xor     eax, DWORD PTR gs:20 ; check canary
    jne     .L5
```

<sup>6</sup>compiler-side buffer overflow protection methods: [wikipedia.org/wiki/Buffer\\_overflow\\_protection](http://wikipedia.org/wiki/Buffer_overflow_protection)

<sup>7</sup>[wikipedia.org/wiki/Domestic\\_canary#Miner.27s\\_canary](http://wikipedia.org/wiki/Domestic_canary#Miner.27s_canary)

```

    mov     ebx, DWORD PTR [ebp-4]
    leave
    ret
.L5:
    call    __stack_chk_fail

```

The random value is located in `gs : 20`. It gets written on the stack and then at the end of the function the value in the stack is compared with the correct “canary” in `gs : 20`. If the values are not equal, the `__stack_chk_fail` function is called and we can see in the console something like that (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` is the so-called segment register. These registers were used widely in MS-DOS and DOS-extenders times. Today, its function is different. To say it briefly, the `gs` register in Linux always points to the [TLS](#) ([65 on page 656](#))—some information specific to thread is stored there. By the way, in win32 the `fs` register plays the same role, pointing to [TIB](#)<sup>8 9</sup>.

More information can be found in the Linux kernel source code (at least in 3.11 version), in `arch/x86/include/asm/stackprotector.h` this variable is described in the comments.

### 18.3.1 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

Let’s get back to our simple array example ([18.1 on page 254](#)), again, now we can see how LLVM checks the correctness of the “canary”:

```

_main
var_64      = -0x64
var_60      = -0x60
var_5C      = -0x5C
var_58      = -0x58
var_54      = -0x54
var_50      = -0x50
var_4C      = -0x4C
var_48      = -0x48
var_44      = -0x44
var_40      = -0x40

```

<sup>8</sup>Thread Information Block

<sup>9</sup>[wikipedia.org/wiki/Win32\\_Thread\\_Information\\_Block](http://wikipedia.org/wiki/Win32_Thread_Information_Block)

```

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
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

```

```

; second loop begin

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          ; canary still correct?
    MOVEQ    R0, #0
    ADDEQ    SP, SP, #0x54
    LDREQ.W  R8, [SP+0x64+var_64],#4
    POPEQ    {R4-R7,PC}
    BLX      __stack_chk_fail

```

First of all, as we see, LLVM “unrolled” the loop and all values were written into an array one-by-one, pre-calculated, as LLVM concluded it can work faster. By the way, instructions in ARM mode may help to do this even faster, and finding this could be your homework.

At the function end we see the comparison of the “canaries”—the one in the local stack and the correct one, to which R8 points. If they are equal to each other, a 4-instruction block is triggered by ITTTT EQ, which contains writing 0 in R0, the function epilogue and exit. If the “canaries” are not equal, the block being skipped, and the jump to `__stack_chk_fail` function will occur, which, perhaps, will halt execution.

## 18.4 One more word about arrays

Now we understand why it is impossible to write something like this in C/C++ code:

```

void f(int size)
{
    int a[size];
    ...
};

```

That’s just because the compiler must know the exact array size to allocate space for it in the local stack layout on at the compiling stage.

If you need an array of arbitrary size, allocate it by using `malloc()`, then access the allocated memory block as an array of variables of the type you need.

Or use the C99 standard feature [ISO07, pp. 6.7.5/2], and it works like `alloca()` ( 5.2.4 on page 27) internally.

It’s also possible to use garbage collecting libraries for C. And there are also libraries supporting smart pointers for C++.

## 18.5 Array of pointers to strings

Here is an example for an array of pointers.

Listing 18.8: Get month name

```

#include <stdio.h>

const char* month1[]=
{
    "January",
    "February",
    "March",
    "April",
    "May",

```

```

    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
};

// in 0..11 range
const char* get_month1 (int month)
{
    return month1[month];
};

```

### 18.5.1 x64

Listing 18.9: Optimizing 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

```

The code is very simple:

- The first `MOVSXD` instruction copies a 32-bit value from `ECX` (where *month* argument is passed) to `RAX` with sign-extension (because the *month* argument is of type *int*). The reason for the sign extension is that this 32-bit value is to be used in calculations with other 64-bit values. Hence, it has to be promoted to 64-bit<sup>10</sup>.
- Then the address of the pointer table is loaded into `RCX`.
- Finally, the input value (*month*) is multiplied by 8 and added to the address. Indeed: we are in a 64-bit environment and all address (or pointers) require exactly 64 bits (or 8 bytes) for storage. Hence, each table element is 8 bytes wide.

<sup>10</sup>It is somewhat weird, but negative array index could be passed here as *month* (negative array indices will have been explained later: [52 on page 572](#)). And if this happens, the negative input value of *int* type is sign-extended correctly and the corresponding element before table is picked. It is not going to work correctly without sign-extension.

And that's why to pick a specific element,  $month * 8$  bytes has to be skipped from the start. That's what MOV does. In addition, this instruction also loads the element at this address. For 1, an element would be a pointer to a string that contains "February", etc.

Optimizing GCC 4.9 can do the job even better<sup>11</sup>:

Listing 18.10: Optimizing GCC 4.9 x64

```
movsx    rdi, edi
mov      rax, QWORD PTR month1[0+rdi*8]
ret
```

### 32-bit MSVC

Let's also compile it in the 32-bit MSVC compiler:

Listing 18.11: Optimizing 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
```

The input value does not need to be extended to 64-bit value, so it is used as is. And it's multiplied by 4, because the table elements are 32-bit (or 4 bytes) wide.

## 18.5.2 32-bit ARM

### ARM in ARM mode

Listing 18.12: Optimizing Keil 6/2013 (ARM mode)

```
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
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
```

<sup>11</sup>"0+" was left in the listing because GCC assembler output is not tidy enough to eliminate it. It's *displacement*, and it's zero here.



```

DCD    ||.conststring||+0x32
DCD    ||.conststring||+0x3c
DCD    ||.conststring||+0x44
DCD    ||.conststring||+0x4d

```

The address of the table is loaded in R1. All the rest is done using just one LDR instruction. Then input value *month* is shifted left by 2 (which is the same as multiplying by 4), then added to R1 (where the address of the table is) and then a table element is loaded from this address. The 32-bit table element is loaded into R0 from the table.

### ARM in Thumb mode

The code is mostly the same, but less dense, because the LSL suffix cannot be specified in the LDR instruction here:

```

get_month1 PROC
    LSLS    r0,r0,#2
    LDR     r1,|L0.64|
    LDR     r0,[r1,r0]
    BX     lr
ENDP

```

## 18.5.3 ARM64

Listing 18.13: Optimizing 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:
    .string "April"
.LC6:
    .string "May"
.LC7:
    .string "June"
.LC8:
    .string "July"
.LC9:
    .string "August"
.LC10:
    .string "September"
.LC11:
    .string "October"
.LC12:

```

```
.string "November"
.LC13:
.string "December"
```

The address of the table is loaded in X1 using ADRP/ADD pair. Then corresponding element is picked using just one LDR, which takes W0 (the register where input argument *month* is), shifts it 3 bits to the left (which is the same as multiplying by 8), sign-extends it (this is what “sxtw” suffix implies) and adds to X0. Then the 64-bit value is loaded from the table into X0.

## 18.5.4 MIPS

Listing 18.14: Optimizing GCC 4.4.5 (IDA)

```
get_month1:
; load address of table into $v0:
    la      $v0, month1
; take input value and multiply it by 4:
    sll     $a0, 2
; sum up address of table and multiplied value:
    addu    $a0, $v0
; load table element at this address into $v0:
    lw      $v0, 0($a0)
; return
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP

.data # .data.rel.local
.globl month1
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>
```

## 18.5.5 Array overflow

Our function accepts values in the range of 0..11, but what if 12 is passed? There is no element in table at this place. So the function will load some value which happens to be there, and return it. Soon after, some other function can try to get a text string from this address and may crash.

Let’s compile the example in MSVC for win64 and open it in [IDA](#) to see what the linker has placed after the table:

Listing 18.15: Executable file in 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

```

Month names are came right after. Our program is tiny, so there isn't much data to pack in the data segment, so it just the month names. But it should be noted that there might be really anything that linker has decided to put by chance.

So what if 12 is passed to the function? The 13th element will be returned. Let's see how the CPU treats the bytes there as a 64-bit value:

Listing 18.16: Executable file in 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

```

And this is 0x797261756E614A. Soon after, some other function (presumably, one that processes strings) may try to read bytes at this address, expecting a C-string there. Most likely it is about to crash, because this value doesn't look like a valid address.

## Array overflow protection

If something can go wrong, it will

Murphy's Law

It's a bit naïve to expect that every programmer who use your function or library will never pass an argument larger than 11. There exists the philosophy that says “fail early and fail loudly” or “fail-fast”, which teaches to report problems as early as possible and stop. One such method in C/C++ is assertions. We can modify our program to fail if an incorrect value is passed:

Listing 18.17: assert() added

```

const char* get_month1_checked (int month)
{
    assert (month<12);
    return month1[month];
};

```

The assertion macro checks for valid values at every function start and fails if the expression is false.

Listing 18.18: Optimizing 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

```

In fact, `assert()` is not a function, but macro. It checks for a condition, then passes also the line number and file name to another function which reports this information to the user. Here we see that both file name and condition are encoded in UTF-16. The line number is also passed (it's 29).

This mechanism is probably the same in all compilers. Here is what GCC does:

Listing 18.19: Optimizing 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"

```

So the macro in GCC also passes the function name for convenience.

Nothing is really free, and this is true for the sanitizing checks as well. They make your program slower, especially if the `assert()` macros used in small time-critical functions. So MSVC, for example, leaves the checks in debug builds, but in release builds they all disappear.

Microsoft [Windows NT](https://msdn.microsoft.com/en-us/library/windows/hardware/ff543450(v=vs.85).aspx) kernels come in “checked” and “free” builds<sup>12</sup>. The first has validation checks (hence, “checked”), the second one doesn't (hence, “free” of checks).

<sup>12</sup>[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)