```
 ____                              
|  _ \ _____   _____ _ __ ___  ___ 
| |_) / _ \ \ / / _ \ '__/ __|/ _ \
|  _ <  __/\ V /  __/ |  \__ \  __/
|_| \_\___| \_/ \___|_|  |___/\___|

 _____             _                      _             
| ____|_ __   __ _(_)_ __   ___  ___ _ __(_)_ __   __ _ 
|  _| | '_ \ / _` | | '_ \ / _ \/ _ \ '__| | '_ \ / _` |
| |___| | | | (_| | | | | |  __/  __/ |  | | | | | (_| |
|_____|_| |_|\__, |_|_| |_|\___|\___|_|  |_|_| |_|\__, |
             |___/                                |___/ 
  __            
 / _| ___  _ __ 
| |_ / _ \| '__|
|  _| (_) | |   
|_|  \___/|_|   

 ____             _                           
| __ )  ___  __ _(_)_ __  _ __   ___ _ __ ___ 
|  _ \ / _ \/ _` | | '_ \| '_ \ / _ \ '__/ __|
| |_) |  __/ (_| | | | | | | | |  __/ |  \__ \
|____/ \___|\__, |_|_| |_|_| |_|\___|_|  |___/
            |___/                             
```

Reverse Engineering for Beginners

(Understanding Assembly Language)

Why two titles? Read here: on page xvii.

Dennis Yurichev
<book@beginners.re>

©2013-2020, Dennis Yurichev.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/.

Text version (December 9, 2020).

The latest version (and Russian edition) of this text is accessible at https://beginners.re/.

Call for translators!

You may want to help me with translating this work into languages other than English and Russian. Just send me any piece of translated text (no matter how short) and I'll put it into my LaTeX source code.

Do not ask, if you should translate. Just do something. I stopped responding "what should I do" emails.

Also, read here.

The language statistics is available right here: https://beginners.re/.

Speed isn't important, because this is an open-source project, after all. Your name will be mentioned as a project contributor. Korean, Chinese, and Persian languages are reserved by publishers. 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 flawed, so I'm grateful for notes about Russian text as well!

So do not hesitate to contact me: <book@beginners.re>.

Abridged contents

1 Code Patterns 1

2 Important fundamentals 564

3 Slightly more advanced examples 592

4 Java 851

5 Finding important/interesting stuff in the code 903

6 OS-specific 947

7 Tools 1020

8 Case studies 1025

9 Examples of reversing proprietary file formats 1191

10 Dynamic binary instrumentation 1265

11 Other things 1276

12 Books/blogs worth reading 1301

13 Communities 1304

Afterword 1306

Appendix 1308

Acronyms Used 1345

Glossary 1352

Index 1355

Contents

1 Code Patterns 1
1.1 The method . . . 1
1.2 Some basics . . . 2
1.2.1 A short introduction to the CPU . . . 2
1.2.2 Numeral Systems . . . 3
1.2.3 Converting From One Radix To Another . . . 4
1.3 An Empty Function . . . 7
1.3.1 x86 . . . 8
1.3.2 ARM . . . 8
1.3.3 MIPS . . . 8
1.3.4 Empty Functions in Practice . . . 9
1.4 Returning Values . . . 10
1.4.1 x86 . . . 10
1.4.2 ARM . . . 10
1.4.3 MIPS . . . 11
1.5 Hello, world! . . . 11
1.5.1 x86 . . . 12
1.5.2 x86-64 . . . 19
1.5.3 ARM . . . 24
1.5.4 MIPS . . . 33
1.5.5 Conclusion . . . 39
1.5.6 Exercises . . . 39
1.6 Function prologue and epilogue . . . 39
1.6.1 Recursion . . . 40
1.7 An Empty Function: redux . . . 40
1.8 Returning Values: redux . . . 40
1.9 Stack . . . 40

1.9.1 Why does the stack grow backwards? 41
1.9.2 What is the stack used for? 42
1.9.3 A typical stack layout 50
1.9.4 Noise in stack 50
1.9.5 Exercises 55
1.10 Almost empty function 55
1.11 printf() with several arguments 56
1.11.1 x86 56
1.11.2 ARM 70
1.11.3 MIPS 77
1.11.4 Conclusion 85
1.11.5 By the way 87
1.12 scanf() 87
1.12.1 Simple example 87
1.12.2 The classic mistake 99
1.12.3 Global variables 100
1.12.4 scanf() 111
1.12.5 Exercise 124
1.13 Worth noting: global vs. local variables 125
1.14 Accessing passed arguments 125
1.14.1 x86 125
1.14.2 x64 128
1.14.3 ARM 132
1.14.4 MIPS 136
1.15 More about results returning 137
1.15.1 Attempt to use the result of a function returning *void* 137
1.15.2 What if we do not use the function result? 139
1.15.3 Returning a structure 139
1.16 Pointers 141
1.16.1 Returning values 141
1.16.2 Swap input values 151
1.17 GOTO operator 152
1.17.1 Dead code 155
1.17.2 Exercise 156
1.18 Conditional jumps 156
1.18.1 Simple example 156
1.18.2 Calculating absolute value 177
1.18.3 Ternary conditional operator 180
1.18.4 Getting minimal and maximal values 184
1.18.5 Conclusion 190
1.18.6 Exercise 192
1.19 Software cracking 192
1.20 Impossible shutdown practical joke (Windows 7) 194
1.21 switch()/case/default 195
1.21.1 Small number of cases 195
1.21.2 A lot of cases 210
1.21.3 When there are several *case* statements in one block 224
1.21.4 Fall-through 229
1.21.5 Exercises 231

1.22 Loops . . . 231
1.22.1 Simple example . . . 231
1.22.2 Memory blocks copying routine . . . 246
1.22.3 Condition check . . . 249
1.22.4 Conclusion . . . 250
1.22.5 Exercises . . . 252
1.23 More about strings . . . 253
1.23.1 strlen() . . . 253
1.23.2 Boundaries of strings . . . 266
1.24 Replacing arithmetic instructions to other ones . . . 267
1.24.1 Multiplication . . . 267
1.24.2 Division . . . 274
1.24.3 Exercise . . . 275
1.25 Floating-point unit . . . 275
1.25.1 IEEE 754 . . . 275
1.25.2 x86 . . . 275
1.25.3 ARM, MIPS, x86/x64 SIMD . . . 276
1.25.4 C/C++ . . . 276
1.25.5 Simple example . . . 276
1.25.6 Passing floating point numbers via arguments . . . 288
1.25.7 Comparison example . . . 291
1.25.8 Some constants . . . 329
1.25.9 Copying . . . 330
1.25.10 Stack, calculators and reverse Polish notation . . . 330
1.25.11 80 bits? . . . 330
1.25.12 x64 . . . 330
1.25.13 Exercises . . . 330
1.26 Arrays . . . 330
1.26.1 Simple example . . . 331
1.26.2 Buffer overflow . . . 340
1.26.3 Buffer overflow protection methods . . . 348
1.26.4 One more word about arrays . . . 353
1.26.5 Array of pointers to strings . . . 354
1.26.6 Multidimensional arrays . . . 363
1.26.7 Pack of strings as a two-dimensional array . . . 373
1.26.8 Conclusion . . . 378
1.26.9 Exercises . . . 378
1.27 Example: a bug in Angband . . . 379
1.28 Manipulating specific bit(s) . . . 382
1.28.1 Specific bit checking . . . 382
1.28.2 Setting and clearing specific bits . . . 387
1.28.3 Shifts . . . 397
1.28.4 Setting and clearing specific bits: FPU[1] example . . . 397
1.28.5 Counting bits set to 1 . . . 403
1.28.6 Conclusion . . . 421
1.28.7 Exercises . . . 424
1.29 Linear congruential generator . . . 424

[1] Floating-Point Unit

1.29.1 x86 . . . 425
1.29.2 x64 . . . 426
1.29.3 32-bit ARM . . . 427
1.29.4 MIPS . . . 428
1.29.5 Thread-safe version of the example . . . 431
1.30 Structures . . . 431
1.30.1 MSVC: SYSTEMTIME example . . . 431
1.30.2 Let's allocate space for a structure using malloc() . . . 435
1.30.3 UNIX: struct tm . . . 438
1.30.4 Fields packing in structure . . . 451
1.30.5 Nested structures . . . 460
1.30.6 Bit fields in a structure . . . 463
1.30.7 Exercises . . . 472
1.31 The classic *struct* bug . . . 472
1.32 Unions . . . 473
1.32.1 Pseudo-random number generator example . . . 474
1.32.2 Calculating machine epsilon . . . 478
1.32.3 FSCALE instruction replacement . . . 480
1.32.4 Fast square root calculation . . . 482
1.33 Pointers to functions . . . 483
1.33.1 MSVC . . . 484
1.33.2 GCC . . . 491
1.33.3 Danger of pointers to functions . . . 496
1.34 64-bit values in 32-bit environment . . . 497
1.34.1 Returning of 64-bit value . . . 497
1.34.2 Arguments passing, addition, subtraction . . . 498
1.34.3 Multiplication, division . . . 502
1.34.4 Shifting right . . . 507
1.34.5 Converting 32-bit value into 64-bit one . . . 508
1.35 LARGE_INTEGER structure case . . . 510
1.36 SIMD . . . 513
1.36.1 Vectorization . . . 514
1.36.2 SIMD `strlen()` implementation . . . 527
1.37 64 bits . . . 531
1.37.1 x86-64 . . . 531
1.37.2 ARM . . . 540
1.37.3 Float point numbers . . . 540
1.37.4 64-bit architecture criticism . . . 540
1.38 Working with floating point numbers using SIMD . . . 541
1.38.1 Simple example . . . 541
1.38.2 Passing floating point number via arguments . . . 549
1.38.3 Comparison example . . . 550
1.38.4 Calculating machine epsilon: x64 and SIMD . . . 553
1.38.5 Pseudo-random number generator example revisited . . . 554
1.38.6 Summary . . . 554
1.39 ARM-specific details . . . 555
1.39.1 Number sign (#) before number . . . 555
1.39.2 Addressing modes . . . 555
1.39.3 Loading a constant into a register . . . 556

1.39.4 Relocs in ARM64 . . . 559
1.40 MIPS-specific details . . . 561
1.40.1 Loading a 32-bit constant into register . . . 561
1.40.2 Further reading about MIPS . . . 563

2 Important fundamentals 564
2.1 Integral datatypes . . . 564
2.1.1 Bit . . . 564
2.1.2 Nibble AKA nybble . . . 565
2.1.3 Byte . . . 566
2.1.4 Wide char . . . 566
2.1.5 Signed integer vs unsigned . . . 567
2.1.6 Word . . . 567
2.1.7 Address register . . . 569
2.1.8 Numbers . . . 569
2.2 Signed number representations . . . 572
2.2.1 Using IMUL over MUL . . . 574
2.2.2 Couple of additions about two's complement form . . . 575
2.2.3 -1 . . . 576
2.3 Integer overflow . . . 576
2.4 AND . . . 577
2.4.1 Checking if a value is on 2^n boundary . . . 577
2.4.2 KOI-8R Cyrillic encoding . . . 578
2.5 AND and OR as subtraction and addition . . . 579
2.5.1 ZX Spectrum ROM text strings . . . 579
2.6 XOR (exclusive OR) . . . 582
2.6.1 Logical difference . . . 582
2.6.2 Everyday speech . . . 582
2.6.3 Encryption . . . 582
2.6.4 RAID[2]4 . . . 582
2.6.5 XOR swap algorithm . . . 583
2.6.6 XOR linked list . . . 584
2.6.7 Switching value trick . . . 584
2.6.8 Zobrist hashing / tabulation hashing . . . 585
2.6.9 By the way . . . 586
2.6.10 AND/OR/XOR as MOV . . . 586
2.7 Population count . . . 586
2.8 Endianness . . . 587
2.8.1 Big-endian . . . 587
2.8.2 Little-endian . . . 587
2.8.3 Example . . . 587
2.8.4 Bi-endian . . . 588
2.8.5 Converting data . . . 588
2.9 Memory . . . 589
2.10 CPU . . . 589
2.10.1 Branch predictors . . . 589
2.10.2 Data dependencies . . . 590
2.11 Hash functions . . . 590

[2]Redundant Array of Independent Disks

2.11.1 How do one-way functions work? 590

3 Slightly more advanced examples **592**

3.1 Zero register 592
3.2 Double negation 596
3.3 const correctness 597
3.3.1 Overlapping const strings 599
3.4 strstr() example 600
3.5 qsort() revisited 601
3.6 Temperature converting 601
3.6.1 Integer values 602
3.6.2 Floating-point values 604
3.7 Fibonacci numbers 607
3.7.1 Example #1 607
3.7.2 Example #2 612
3.7.3 Summary 615
3.8 CRC32 calculation example 616
3.9 Network address calculation example 620
3.9.1 calc_network_address() 622
3.9.2 form_IP() 622
3.9.3 print_as_IP() 624
3.9.4 form_netmask() and set_bit() 626
3.9.5 Summary 627
3.10 Loops: several iterators 627
3.10.1 Three iterators 628
3.10.2 Two iterators 628
3.10.3 Intel C++ 2011 case 631
3.11 Duff's device 632
3.11.1 Should one use unrolled loops? 636
3.12 Division using multiplication 636
3.12.1 x86 636
3.12.2 How it works 638
3.12.3 ARM 638
3.12.4 MIPS 640
3.12.5 Exercise 640
3.13 String to number conversion (atoi()) 641
3.13.1 Simple example 641
3.13.2 A slightly advanced example 645
3.13.3 Exercise 649
3.14 Inline functions 649
3.14.1 Strings and memory functions 650
3.15 C99 restrict 660
3.16 Branchless *abs()* function 663
3.16.1 Optimizing GCC 4.9.1 x64 664
3.16.2 Optimizing GCC 4.9 ARM64 664
3.17 Variadic functions 665
3.17.1 Computing arithmetic mean 665
3.17.2 *vprintf()* function case 670
3.17.3 Pin case 672

3.17.4 Format string exploit 672
3.18 Strings trimming 673
3.18.1 x64: Optimizing MSVC 2013 675
3.18.2 x64: Non-optimizing GCC 4.9.1 677
3.18.3 x64: Optimizing GCC 4.9.1 678
3.18.4 ARM64: Non-optimizing GCC (Linaro) 4.9 679
3.18.5 ARM64: Optimizing GCC (Linaro) 4.9 681
3.18.6 ARM: Optimizing Keil 6/2013 (ARM mode) 682
3.18.7 ARM: Optimizing Keil 6/2013 (Thumb mode) 682
3.18.8 MIPS 683
3.19 toupper() function 685
3.19.1 x64 686
3.19.2 ARM 688
3.19.3 Using bit operations 689
3.19.4 Summary 691
3.20 Obfuscation 691
3.20.1 Text strings 691
3.20.2 Executable code 692
3.20.3 Virtual machine / pseudo-code 696
3.20.4 Other things to mention 696
3.20.5 Exercise 696
3.21 C++ 697
3.21.1 Classes 697
3.21.2 ostream 718
3.21.3 References 720
3.21.4 STL 721
3.21.5 Memory 766
3.22 Negative array indices 767
3.22.1 Addressing string from the end 767
3.22.2 Addressing some kind of block from the end 768
3.22.3 Arrays started at 1 768
3.23 More about pointers 771
3.23.1 Working with addresses instead of pointers 772
3.23.2 Passing values as pointers; tagged unions 775
3.23.3 Pointers abuse in Windows kernel 776
3.23.4 Null pointers 782
3.23.5 Array as function argument 788
3.23.6 Pointer to a function 789
3.23.7 Pointer to a function: copy protection 790
3.23.8 Pointer to a function: a common bug (or typo) 791
3.23.9 Pointer as object identificator 791
3.23.10 Oracle RDBMS and a simple garbage collector for C/C++ 793
3.24 Loop optimizations 794
3.24.1 Weird loop optimization 794
3.24.2 Another loop optimization 796
3.25 More about structures 798
3.25.1 Sometimes a C structure can be used instead of array 798
3.25.2 Unsized array in C structure 800
3.25.3 Version of C structure 801

3.25.4 High-score file in “Block out” game and primitive serialization . 804
3.26 memmove() and memcpy() . . . 809
3.26.1 Anti-debugging trick . . . 811
3.27 setjmp/longjmp . . . 811
3.28 Other weird stack hacks . . . 814
3.28.1 Accessing arguments/local variables of caller . . . 814
3.28.2 Returning string . . . 816
3.29 OpenMP . . . 818
3.29.1 MSVC . . . 821
3.29.2 GCC . . . 824
3.30 Signed division using shifts . . . 826
3.31 Another heisenbug . . . 828
3.32 The case of forgotten return . . . 829
3.33 Homework: more about function pointers and unions . . . 834
3.34 Windows 16-bit . . . 836
3.34.1 Example#1 . . . 836
3.34.2 Example #2 . . . 837
3.34.3 Example #3 . . . 837
3.34.4 Example #4 . . . 839
3.34.5 Example #5 . . . 842
3.34.6 Example #6 . . . 846

4 Java **851**
4.1 Java . . . 851
4.1.1 Introduction . . . 851
4.1.2 Returning a value . . . 852
4.1.3 Simple calculating functions . . . 858
4.1.4 JVM[3] memory model . . . 861
4.1.5 Simple function calling . . . 861
4.1.6 Calling beep() . . . 864
4.1.7 Linear congruential PRNG[4] . . . 864
4.1.8 Conditional jumps . . . 866
4.1.9 Passing arguments . . . 869
4.1.10 Bitfields . . . 870
4.1.11 Loops . . . 871
4.1.12 switch() . . . 874
4.1.13 Arrays . . . 875
4.1.14 Strings . . . 886
4.1.15 Exceptions . . . 889
4.1.16 Classes . . . 893
4.1.17 Simple patching . . . 896
4.1.18 Summary . . . 902

5 Finding important/interesting stuff in the code **903**
5.1 Identification of executable files . . . 904
5.1.1 Microsoft Visual C++ . . . 904
5.1.2 GCC . . . 904

[3]Java Virtual Machine
[4]Pseudorandom Number Generator

5.1.3 Intel Fortran . . . 905
5.1.4 Watcom, OpenWatcom . . . 905
5.1.5 Borland . . . 905
5.1.6 Other known DLLs . . . 907
5.2 Communication with outer world (function level) . . . 907
5.3 Communication with the outer world (win32) . . . 907
5.3.1 Often used functions in the Windows API . . . 908
5.3.2 Extending trial period . . . 909
5.3.3 Removing nag dialog box . . . 909
5.3.4 tracer: Intercepting all functions in specific module . . . 909
5.4 Strings . . . 910
5.4.1 Text strings . . . 910
5.4.2 Finding strings in binary . . . 916
5.4.3 Error/debug messages . . . 918
5.4.4 Suspicious magic strings . . . 918
5.5 Calls to assert() . . . 919
5.6 Constants . . . 920
5.6.1 Magic numbers . . . 921
5.6.2 Specific constants . . . 923
5.6.3 Searching for constants . . . 923
5.7 Finding the right instructions . . . 923
5.8 Suspicious code patterns . . . 925
5.8.1 XOR instructions . . . 925
5.8.2 Hand-written assembly code . . . 926
5.9 Using magic numbers while tracing . . . 927
5.10 Loops . . . 927
5.10.1 Some binary file patterns . . . 929
5.10.2 Memory “snapshots” comparing . . . 937
5.11 ISA[5] detection . . . 939
5.11.1 Incorrectly disassembled code . . . 939
5.11.2 Correctly disassembled code . . . 945
5.12 Other things . . . 945
5.12.1 General idea . . . 945
5.12.2 Order of functions in binary code . . . 945
5.12.3 Tiny functions . . . 946
5.12.4 C++ . . . 946
5.12.5 Crash on purpose . . . 946

6 OS-specific **947**
6.1 Arguments passing methods (calling conventions) . . . 947
6.1.1 cdecl . . . 947
6.1.2 stdcall . . . 947
6.1.3 fastcall . . . 949
6.1.4 thiscall . . . 950
6.1.5 x86-64 . . . 951
6.1.6 Return values of *float* and *double* type . . . 954
6.1.7 Modifying arguments . . . 954
6.1.8 Taking a pointer to function argument . . . 956

[5] Instruction Set Architecture

6.1.9 Python ctypes problem (x86 assembly homework) 958
6.1.10 Cdecl example: a DLL 958
6.2 Thread Local Storage 959
6.2.1 Linear congruential generator revisited 959
6.3 System calls (syscall-s) 965
6.3.1 Linux 966
6.3.2 Windows 966
6.4 Linux 967
6.4.1 Position-independent code 967
6.4.2 *LD_PRELOAD* hack in Linux 970
6.5 Windows NT 973
6.5.1 CRT (win32) 973
6.5.2 Win32 PE 978
6.5.3 Windows SEH 988
6.5.4 Windows NT: Critical section 1017

7 Tools **1020**
7.1 Binary analysis 1020
7.1.1 Disassemblers 1021
7.1.2 Decompilers 1021
7.1.3 Patch comparison/diffing 1021
7.2 Live analysis 1021
7.2.1 Debuggers 1022
7.2.2 Library calls tracing 1022
7.2.3 System calls tracing 1022
7.2.4 Network sniffing 1023
7.2.5 Sysinternals 1023
7.2.6 Valgrind 1023
7.2.7 Emulators 1023
7.3 Other tools 1024
7.3.1 SMT solvers 1024
7.3.2 Calculators 1024
7.4 Do You Think Something Is Missing Here? 1024

8 Case studies **1025**
8.1 Mahjong solitaire prank (Windows 7) 1026
8.2 Task manager practical joke (Windows Vista) 1028
8.2.1 Using LEA to load values 1032
8.3 Color Lines game practical joke 1033
8.4 Minesweeper (Windows XP) 1036
8.4.1 Finding grid automatically 1043
8.4.2 Exercises 1044
8.5 Hacking Windows clock 1044
8.6 (Windows 7) Solitaire: practical jokes 1054
8.6.1 51 cards 1054
8.6.2 53 cards 1062
8.7 FreeCell prank (Windows 7) 1063
8.7.1 Part I 1063
8.7.2 Part II: breaking the *Select Game* submenu 1068

8.8 Dongles . . . 1070
8.8.1 Example #1: MacOS Classic and PowerPC . . . 1070
8.8.2 Example #2: SCO OpenServer . . . 1080
8.8.3 Example #3: MS-DOS . . . 1093
8.9 Encrypted database case #1 . . . 1100
8.9.1 Base64 and entropy . . . 1100
8.9.2 Is data compressed? . . . 1102
8.9.3 Is data encrypted? . . . 1103
8.9.4 CryptoPP . . . 1104
8.9.5 Cipher Feedback mode . . . 1107
8.9.6 Initializing Vector . . . 1109
8.9.7 Structure of the buffer . . . 1110
8.9.8 Noise at the end . . . 1113
8.9.9 Conclusion . . . 1114
8.9.10 Post Scriptum: brute-forcing IV[6] . . . 1114
8.10 Overclocking Cointerra Bitcoin miner . . . 1115
8.11 Breaking simple executable code encryptor . . . 1121
8.11.1 Other ideas to consider . . . 1126
8.12 SAP . . . 1127
8.12.1 About SAP client network traffic compression . . . 1127
8.12.2 SAP 6.0 password checking functions . . . 1142
8.13 Oracle RDBMS . . . 1147
8.13.1 V$VERSION table in the Oracle RDBMS . . . 1147
8.13.2 X$KSMLRU table in Oracle RDBMS . . . 1157
8.13.3 V$TIMER table in Oracle RDBMS . . . 1159
8.14 Handwritten assembly code . . . 1164
8.14.1 EICAR test file . . . 1164
8.15 Demos . . . 1165
8.15.1 10 PRINT CHR$(205.5+RND(1)); : GOTO 10 . . . 1166
8.15.2 Mandelbrot set . . . 1170
8.16 A nasty bug in MSVCRT.DLL . . . 1182
8.17 Other examples . . . 1190

9 Examples of reversing proprietary file formats 1191
9.1 Primitive XOR-encryption . . . 1191
9.1.1 Simplest ever XOR encryption . . . 1191
9.1.2 Norton Guide: simplest possible 1-byte XOR encryption . . . 1193
9.1.3 Simplest possible 4-byte XOR encryption . . . 1196
9.1.4 Simple encryption using XOR mask . . . 1200
9.1.5 Simple encryption using XOR mask, case II . . . 1209
9.1.6 Homework . . . 1216
9.2 Information entropy . . . 1216
9.2.1 Analyzing entropy in Mathematica . . . 1217
9.2.2 Conclusion . . . 1227
9.2.3 Tools . . . 1227
9.2.4 A word about primitive encryption like XORing . . . 1227
9.2.5 More about entropy of executable code . . . 1227
9.2.6 PRNG . . . 1228

[6] Initialization Vector

9.2.7 More examples . . . 1228
9.2.8 Entropy of various files . . . 1228
9.2.9 Making lower level of entropy . . . 1230
9.3 Millenium game save file . . . 1231
9.4 *fortune* program indexing file . . . 1238
9.4.1 Hacking . . . 1244
9.4.2 The files . . . 1245
9.5 Oracle RDBMS: .SYM-files . . . 1245
9.6 Oracle RDBMS: .MSB-files . . . 1258
9.6.1 Summary . . . 1264
9.7 Exercises . . . 1264
9.8 Further reading . . . 1264

10 Dynamic binary instrumentation 1265
10.1 Using PIN DBI for XOR interception . . . 1265
10.2 Cracking Minesweeper with PIN . . . 1269
10.2.1 Intercepting all rand() calls . . . 1269
10.2.2 Replacing rand() calls with our function . . . 1270
10.2.3 Peeking into placement of mines . . . 1271
10.2.4 Exercise . . . 1274
10.3 Building Pin . . . 1274
10.4 Why “instrumentation”? . . . 1275

11 Other things 1276
11.1 Executable files patching . . . 1276
11.1.1 x86 code . . . 1276
11.2 Function arguments number statistics . . . 1277
11.3 Compiler intrinsic . . . 1278
11.4 Compiler’s anomalies . . . 1278
11.4.1 Oracle RDBMS 11.2 and Intel C++ 10.1 . . . 1278
11.4.2 MSVC 6.0 . . . 1279
11.4.3 ftol2() in MSVC 2012 . . . 1279
11.4.4 Summary . . . 1281
11.5 Itanium . . . 1281
11.6 8086 memory model . . . 1284
11.7 Basic blocks reordering . . . 1286
11.7.1 Profile-guided optimization . . . 1286
11.8 My experience with Hex-Rays 2.2.0 . . . 1288
11.8.1 Bugs . . . 1288
11.8.2 Odd peculiarities . . . 1290
11.8.3 Silence . . . 1291
11.8.4 Comma . . . 1293
11.8.5 Data types . . . 1294
11.8.6 Long and messed expressions . . . 1294
11.8.7 De Morgan’s laws and decompilation . . . 1295
11.8.8 My plan . . . 1297
11.8.9 Summary . . . 1297
11.9 Cyclomatic complexity . . . 1297

12 Books/blogs worth reading 1301
12.1 Books and other materials . . . 1301
12.1.1 Reverse Engineering . . . 1301
12.1.2 Windows . . . 1301
12.1.3 C/C++ . . . 1302
12.1.4 x86 / x86-64 . . . 1302
12.1.5 ARM . . . 1302
12.1.6 Assembly language . . . 1303
12.1.7 Java . . . 1303
12.1.8 UNIX . . . 1303
12.1.9 Programming in general . . . 1303
12.1.10 Cryptography . . . 1303
12.1.11 Something even easier . . . 1303

13 Communities 1304

Afterword 1306
13.1 Questions? . . . 1306

Appendix 1308
.1 x86 . . . 1308
.1.1 Terminology . . . 1308
.1.2 General purpose registers . . . 1308
.1.3 FPU registers . . . 1313
.1.4 SIMD registers . . . 1315
.1.5 Debugging registers . . . 1315
.1.6 Instructions . . . 1317
.1.7 npad . . . 1333
.2 ARM . . . 1335
.2.1 Terminology . . . 1335
.2.2 Versions . . . 1335
.2.3 32-bit ARM (AArch32) . . . 1336
.2.4 64-bit ARM (AArch64) . . . 1337
.2.5 Instructions . . . 1338
.3 MIPS . . . 1338
.3.1 Registers . . . 1338
.3.2 Instructions . . . 1339
.4 Some GCC library functions . . . 1340
.5 Some MSVC library functions . . . 1340
.6 Cheatsheets . . . 1341
.6.1 IDA . . . 1341
.6.2 OllyDbg . . . 1341
.6.3 MSVC . . . 1342
.6.4 GCC . . . 1342
.6.5 GDB . . . 1342

Acronyms Used **1345**

Glossary **1352**

Index **1355**

Preface

What is with two titles?

The book was named “Reverse Engineering for Beginners” in 2014-2018, but I always suspected this makes readership too narrow.

Infosec people know about “reverse engineering”, but I’ve rarely hear the “assembler” word from them.

Likewise, the “reverse engineering” term is somewhat cryptic to a general audience of programmers, but they know about “assembler”.

In July 2018, as an experiment, I’ve changed the title to “Assembly Language for Beginners” and posted the link to Hacker News website[7], and the book was received generally well.

So let it be, the book now has two titles.

However, I’ve changed the second title to “Understanding Assembly Language”, because someone had already written “Assembly Language for Beginners” book. Also, people say “for Beginners” sounds a bit sarcastic for a book of ~1000 pages.

The two books differ only by title, filename (UAL-XX.pdf versus RE4B-XX.pdf), URL and a couple of the first pages.

About reverse engineering

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[8] structure

This book is about the first meaning.

Prerequisites

Basic knowledge of the C PL[9]. Recommended reading: 12.1.3 on page 1302.

Exercises and tasks

…can be found at: `http://challenges.re`.

Praise for this book

`https://beginners.re/#praise`.

[7] `https://news.ycombinator.com/item?id=17549050`
[8] Database Management Systems
[9] Programming Language

Universities

The book is recommended at least at these universities: https://beginners.re/#uni.

Thanks

For patiently answering all my questions: SkullC0DEr.

For sending me notes about mistakes and inaccuracies: Alexander Lysenko, Federico Ramondino, Mark Wilson, Razikhova Meiramgul Kayratovna, Anatoly Prokofiev, Kostya Begunets, Valentin “netch” Nechayev, Aleksandr Plakhov, Artem Metla, Alexander Yastrebov, Vlad Golovkin[10], Evgeny Proshin, Alexander Myasnikov, Alexey Tretiakov, Oleg Peskov, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon[11], Ben L., Etienne Khan, Norbert Szetei[12], Marc Remy, Michael Hansen, Derk Barten, The Renaissance[13], Hugo Chan, Emil Mursalimov, Tanner Hoke, Tan90909090@GitHub, Ole Petter Orhagen, Sourav Punoriyar, Vitor Oliveira, Alexis Ehret, Maxim Shlochiski, Greg Paton, Pierrick Lebourgeois, Abdullah Alomair.

For helping me in other ways: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar, Peter Sovietov, Misha “tiphareth” Verbitsky.

For translating the book into Simplified Chinese: Antiy Labs (antiy.cn), Archer.

For translating the book into Korean: Byungho Min.

For translating the book into Dutch: Cedric Sambre (AKA Midas).

For translating the book into Spanish: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames, Emiliano Estevarena.

For translating the book into Portuguese: Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe, Primo David Santini.

For translating the book into Italian: Federico Ramondino[14], Paolo Stivanin[15], twyK, Fabrizio Bertone, Matteo Sticco, Marco Negro[16], bluepulsar.

For translating the book into French: Florent Besnard[17], Marc Remy[18], Baudouin Landais, Téo Dacquet[19], BlueSkeye@GitHub[20].

For translating the book into German: Dennis Siekmeier[21], Julius Angres[22], Dirk

[10] goto-vlad@github
[11] https://github.com/pixjuan
[12] https://github.com/73696e65
[13] https://github.com/TheRenaissance
[14] https://github.com/pinkrab
[15] https://github.com/paolostivanin
[16] https://github.com/Internaut401
[17] https://github.com/besnardf
[18] https://github.com/mremy
[19] https://github.com/T30rix
[20] https://github.com/BlueSkeye
[21] https://github.com/DSiekmeier
[22] https://github.com/JAngres

Loser[23], Clemens Tamme, Philipp Schweinzer.

For translating the book into Polish: Kateryna Rozanova, Aleksander Mistewicz, Wiktoria Lewicka, Marcin Sokołowski.

For translating the book into Japanese: shmz@github[24],4ryuJP@github[25].

For proofreading: Vladimir Botov, Andrei Brazhuk, Mark "Logxen" Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev[26] did a great amount of work in proofreading and correcting many mistakes.

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), Anonymous ($15), Vladislav Chelnokov ($25), Nenad Noveljic ($50), Ryan Smith ($25), Andreas Schommer (€5), Nikolay Gavrilov ($300), Ernesto Bonev Reynoso ($30).

Thanks a lot to every donor!

mini-FAQ

Q: Is this book simpler/easier than others?

A: No, it is at about the same level as other books of this subject.

Q: I'm too frightened to start reading this book, there are more than 1000 pages. "...for Beginners" in the name sounds a bit sarcastic.

[23] https://github.com/PolymathMonkey
[24] https://github.com/shmz
[25] https://github.com/4ryuJP
[26] https://vasil.ludost.net/

A: All sorts of listings are the bulk of the book. The book is indeed for beginners, there is a lot missing (yet).

Q: What are the prerequisites for reading this book?

A: A basic understanding of C/C++ is desirable.

Q: Should I really learn x86/x64/ARM and MIPS at once? Isn't it too much?

A: Starters can read about just x86/x64, while skipping or skimming the ARM and MIPS parts.

Q: Can I buy a Russian or English hard copy/paper book?

A: Unfortunately, no. No publisher got interested in publishing a Russian or English version so far. Meanwhile, you can ask your favorite copy shop to print and bind it. https://yurichev.com/news/20200222_printed_RE4B/.

Q: Is there an epub or mobi version?

A: No. The book is highly dependent on TeX/LaTeX-specific hacks, so converting to HTML (epub/mobi are a set of HTMLs) would not be easy.

Q: Why should one learn assembly language these days?

A: Unless you are an OS[27] developer, you probably don't need to code in assembly—the latest compilers (2010s) are much better at performing optimizations than humans [28].

Also, the latest CPU[29]s are very complex devices, and assembly knowledge doesn't really help towards understand their internals.

That being said, there are at least two areas where a good understanding of assembly can be helpful: First and foremost, for security/malware research. It is also a good way to gain a better understanding of your compiled code while 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. In Evince click "<" button.

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 (CC BY-SA 4.0).

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

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

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

[27] Operating System
[28] A very good text on this topic: [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]
[29] Central Processing Unit

A: There are hiring threads that appear from time to time on reddit, devoted to RE[30]. Try looking there.

A somewhat related hiring thread can be found in the “netsec” subreddit.

Q: Compilers’ versions in the book are outdated already...

A: No need to follow all steps precisely. Use the compilers you already have installed on your OS. Also, there is: Compiler Explorer.

Q: I have a question...

A: Send it to me by email (<book@beginners.re>).

About the Korean translation

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

It’s available now at their website.

The translator is Byungho Min (twitter/tais9). The cover art was done by the artistic Andy Nechaevsky, a friend of the author: facebook/andydinka. Acorn also holds the copyright to the Korean translation.

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

About the Persian/Farsi translation

In 2016 the book was translated by Mohsen Mostafa Jokar (who is also known to Iranian community for his translation of Radare manual[31]). It is available on the publisher’s website[32] (Pendare Pars).

Here is a link to a 40-page excerpt: `https://beginners.re/farsi.pdf`.

National Library of Iran registration information: `http://opac.nlai.ir/opac-prod/bibliographic/4473995`.

About the Chinese translation

In April 2017, translation to Chinese was completed by Chinese PTPress. They are also the Chinese translation copyright holders.

The Chinese version is available for order here: `http://www.epubit.com.cn/book/details/4174`. A partial review and history behind the translation can be found here: `http://www.cptoday.cn/news/detail/3155`.

The principal translator is Archer, to whom the author owes very much. He was extremely meticulous (in a good sense) and reported most of the known mistakes

[30] reddit.com/r/ReverseEngineering/
[31] `http://rada.re/get/radare2book-persian.pdf`
[32] `http://goo.gl/2Tzx0H`

and bugs, which is very important in literature such as this book. The author would recommend his services to any other author!

The guys from Antiy Labs has also helped with translation. Here is preface written by them.

Chapter 1

Code Patterns

1.1 The method

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. [1]. He did this so many times that the relationship between the C/C++ code and what the compiler produced was imprinted deeply in his mind. It's now easy for him to imagine instantly a rough outline of a C code's appearance and function. Perhaps this technique could be helpful for others.

By the way, there is a great website where you can do the same, with various compilers, instead of installing them on your box. You can use it as well: https://godbolt.org/.

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 the latest 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 that opti-

[1] In fact, he still does this when he can't understand what a particular bit of code does. A recent example from the year 2019: `p += p+(i&1)+2;` from the "SAT0W" SAT-solver by D.Knuth.

mization is completely disabled. 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, a compiler can include some debug information in the resulting file, producing code that is easy to debug. One of the important features of the ´debug' code is that it might contain links between each line of the source code and its respective machine code address. 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, wherever possible.

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

1.2 Some basics

1.2.1 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[2]). ≈ 8 in x86, ≈ 16 in x86-64, and also ≈ 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 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, or Python, 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

[2]General Purpose Registers

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*. [3].

A couple of words about different ISAs

The x86 ISA has always had variable-length instructions, 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[4] CPU designed with constant-length instructions in mind, which had some advantages in the past. In the very beginning, all ARM instructions were encoded in 4 bytes[5]. This is now referred to as "ARM mode". Then they realized it wasn't as frugal as they first imagined. In fact, the most common CPU instructions[6] in real world applications can be encoded using less information. They therefore added another ISA, called Thumb, in which each instruction was encoded in just 2 bytes. This is now referred to 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 can 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. (Though, admittedly, this is largely due to the fact that Xcode does this by default). Later the 64-bit ARM came out. This ISA has 4-byte instructions, and lacked the need of any additional Thumb mode. However, 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 will 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 instructions, such as MIPS, PowerPC and Alpha AXP.

1.2.2 Numeral Systems

> Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.
>
> Alan A. A. Donovan, Brian W. Kernighan — The Go Programming Language

[3]Old-school Russian literature also uses the term "translator".

[4]Reduced Instruction Set Computing

[5]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 (1.21.2 on page 217) section.

[6]e.g. MOV/PUSH/CALL/Jcc

Humans have become accustomed to a decimal numeral system, probably because almost everyone has 10 fingers. Nevertheless, the number "10" has no significant meaning in science and mathematics. The natural numeral system in digital electronics is binary: 0 is for an absence of current in the wire, and 1 for presence. 10 in binary is 2 in decimal, 100 in binary is 4 in decimal, and so on.

If the numeral system has 10 digits, it has a *radix* (or *base*) of 10. The binary numeral system has a *radix* of 2.

Important things to recall:

1) A *number* is a number, while a *digit* is a term from writing systems, and is usually one character

2) The value of a number does not change when converted to another radix; only the writing notation for that value has changed (and therefore the way of representing it in RAM[7]).

1.2.3 Converting From One Radix To Another

Positional notation is used in almost every numerical system. This means that a digit has weight relative to where it is placed inside of the larger number. If 2 is placed at the rightmost place, it's 2, but if it's placed one digit before rightmost, it's 20.

What does 1234 stand for?

$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4 = 1234$ or $1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$

It's the same story for binary numbers, but the base is 2 instead of 10. What does 0b101011 stand for?

$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43$ or $32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$

There is such a thing as non-positional notation, such as the Roman numeral system. [8]. Perhaps, humankind switched to positional notation because it's easier to do basic operations (addition, multiplication, etc.) on paper by hand.

Binary numbers can be added, subtracted and so on in the very same as taught in schools, but only 2 digits are available.

Binary numbers are bulky when represented in source code and dumps, so that is where the hexadecimal numeral system can be useful. A hexadecimal radix uses the digits 0..9, and also 6 Latin characters: A..F. Each hexadecimal digit takes 4 bits or 4 binary digits, so it's very easy to convert from binary number to hexadecimal and back, even manually, in one's mind.

hexadecimal	binary	decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3

[7] Random-Access Memory

[8] About numeric system evolution, see [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195–213.]

4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

How can one tell which radix is being used in a specific instance?

Decimal numbers are usually written as is, i.e., 1234. Some assemblers allow an identifier on decimal radix numbers, in which the number would be written with a ”d” suffix: 1234d.

Binary numbers are sometimes prepended with the ”0b” prefix: 0b100110111 (GCC[9] has a non-standard language extension for this[10]). There is also another way: using a ”b” suffix, for example: 100110111b. This book tries to use the ”0b” prefix consistently throughout the book for binary numbers.

Hexadecimal numbers are prepended with ”0x” prefix in C/C++ and other PLs: 0x1234ABCD. Alternatively, they are given a ”h” suffix: 1234ABCDh. This is common way of representing them in assemblers and debuggers. In this convention, if the number is started with a Latin (A..F) digit, a 0 is added at the beginning: 0ABCDEFh. There was also convention that was popular in 8-bit home computers era, using $ prefix, like $ABCD. The book will try to stick to ”0x” prefix throughout the book for hexadecimal numbers.

Should one learn to convert numbers mentally? A table of 1-digit hexadecimal numbers can easily be memorized. As for larger numbers, it’s probably not worth tormenting yourself.

Perhaps the most visible hexadecimal numbers are in URL[11]s. This is the way that non-Latin characters are encoded. For example: `https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9` is the URL of Wiktionary article about “naïveté” word.

Octal Radix

Another numeral system heavily used in the past of computer programming is octal. In octal there are 8 digits (0..7), and each is mapped to 3 bits, so it’s easy to convert numbers back and forth. It has been superseded by the hexadecimal system almost

[9] GNU Compiler Collection
[10] `https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html`
[11] Uniform Resource Locator

everywhere, but, surprisingly, there is a *NIX utility, used often by many people, which takes octal numbers as argument: chmod.

As many *NIX users know, chmod argument can be a number of 3 digits. The first digit represents the rights of the owner of the file (read, write and/or execute), the second is the rights for the group to which the file belongs, and the third is for everyone else. Each digit that chmod takes can be represented in binary form:

decimal	binary	meaning
7	111	**rwx**
6	110	**rw-**
5	101	**r-x**
4	100	**r--**
3	011	**-wx**
2	010	**-w-**
1	001	**--x**
0	000	**---**

So each bit is mapped to a flag: read/write/execute.

The importance of chmod here is that the whole number in argument can be represented as octal number. Let's take, for example, 644. When you run `chmod 644 file`, you set read/write permissions for owner, read permissions for group and again, read permissions for everyone else. If we convert the octal number 644 to binary, it would be `110100100`, or, in groups of 3 bits, `110 100 100`.

Now we see that each triplet describe permissions for owner/group/others: first is `rw-`, second is `r--` and third is `r--`.

The octal numeral system was also popular on old computers like PDP-8, because word there could be 12, 24 or 36 bits, and these numbers are all divisible by 3, so the octal system was natural in that environment. Nowadays, all popular computers employ word/address sizes of 16, 32 or 64 bits, and these numbers are all divisible by 4, so the hexadecimal system is more natural there.

The octal numeral system is supported by all standard C/C++ compilers. This is a source of confusion sometimes, because octal numbers are encoded with a zero prepended, for example, 0377 is 255. Sometimes, you might make a typo and write ”09” instead of 9, and the compiler would report an error. GCC might report something like this:
`error: invalid digit "9" in octal constant`.

Also, the octal system is somewhat popular in Java. When the IDA shows Java strings with non-printable characters, they are encoded in the octal system instead of hexadecimal. The JAD Java decompiler behaves the same way.

Divisibility

When you see a decimal number like 120, you can quickly deduce that it's divisible by 10, because the last digit is zero. In the same way, 123400 is divisible by 100,

because the two last digits are zeros.

Likewise, the hexadecimal number 0x1230 is divisible by 0x10 (or 16), 0x123000 is divisible by 0x1000 (or 4096), etc.

The binary number 0b1000101000 is divisible by 0b1000 (8), etc.

This property can often be used to quickly realize if an address or a size of some block in memory is padded to some boundary. For example, sections in PE[12] files are almost always started at addresses ending with 3 hexadecimal zeros: 0x41000, 0x10001000, etc. The reason behind this is the fact that almost all PE sections are padded to a boundary of 0x1000 (4096) bytes.

Multi-Precision Arithmetic and Radix

Multi-precision arithmetic can use huge numbers, and each one may be stored in several bytes. For example, RSA keys, both public and private, span up to 4096 bits, and maybe even more.

In [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] we find the following idea: when you store a multi-precision number in several bytes, the whole number can be represented as having a radix of $2^8 = 256$, and each digit goes to the corresponding byte. Likewise, if you store a multi-precision number in several 32-bit integer values, each digit goes to each 32-bit slot, and you may think about this number as stored in radix of 2^{32}.

How to Pronounce Non-Decimal Numbers

Numbers in a non-decimal base are usually pronounced by digit by digit: “one-zero-zero-one-one-...”. Words like “ten” and “thousand” are usually not pronounced, to prevent confusion with the decimal base system.

Floating point numbers

To distinguish floating point numbers from integers, they are usually written with “.0” at the end, like 0.0, 123.0, etc.

1.3 An Empty Function

The simplest possible function is arguably one that does nothing:

Listing 1.1: C/C++ Code

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

Let’s compile it!

[12] Portable Executable

1.3.1 x86

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

Listing 1.2: Optimizing GCC/MSVC (assembly output)

```
f:
        ret
```

There is just one instruction: RET, which returns execution to the caller.

1.3.2 ARM

Listing 1.3: Optimizing Keil 6/2013 (ARM mode) assembly output

```
f       PROC
        BX       lr
        ENDP
```

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.

1.3.3 MIPS

There are two naming conventions used in the world of MIPS when naming registers: by number (from $0 to $31) or by pseudo name ($V0, $A0, etc.).

The GCC assembly output below lists registers by number:

Listing 1.4: Optimizing GCC 4.4.5 (assembly output)

```
        j       $31
        nop
```

…while IDA[13] does it by pseudo name:

Listing 1.5: Optimizing GCC 4.4.5 (IDA)

```
        j       $ra
        nop
```

The first 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[14] in ARM.

The second instruction is NOP[15], which does nothing. We can ignore it for now.

[13] Interactive Disassembler and Debugger developed by Hex-Rays
[14] Link Register
[15] No Operation

A Note About MIPS Instructions and Register Names

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

1.3.4 Empty Functions in Practice

Despite the fact empty functions seem useless, they are quite frequent in low-level code.

First of all, they are quite popular in debugging functions, like this one:

Listing 1.6: C/C++ code

```
void dbg_print (const char *fmt, ...)
{
#ifdef _DEBUG
        // open log file
        // write to log file
        // close log file
#endif
};

void some_function()
{
        ...

        dbg_print ("we did something\n");

        ...
};
```

In a non-debug build (as in a “release”), `_DEBUG` is not defined, so the `dbg_print()` function, despite still being called during execution, will be empty.

Similarly, a popular method of software protection is to make one build for legal customers, and another demo build. A demo build can lack of some important functions, as with this example:

Listing 1.7: C/C++ code

```
void save_file ()
{
#ifndef DEMO
        // a real saving code
#endif
};
```

The `save_file()` function could be called when the user clicks `File->Save` on the menu. The demo version might be delivered with this menu item disabled, but even if a software cracker would enable it, only an empty function with no useful code will be called.

IDA marks such functions with names like `nullsub_00`, `nullsub_01`, etc.

1.4 Returning Values

Another simple function is the one that simply returns a constant value:

Listing 1.8: C/C++ Code

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

Let's compile it.

1.4.1 x86

Here's what both the GCC and MSVC compilers produce (with optimization) on the x86 platform:

Listing 1.9: 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.

1.4.2 ARM

There are a few differences on the ARM platform:

Listing 1.10: 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`.

It is worth noting that `MOV` is a misleading name for the instruction in both the x86 and ARM ISAs.

The data is not in fact *moved*, but *copied*.

1.4.3 MIPS

The GCC assembly output below lists registers by number:

Listing 1.11: Optimizing GCC 4.4.5 (assembly output)

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

...while IDA does it by their pseudo names:

Listing 1.12: 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.

You might be wondering why the positions of 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 must simply keep in mind 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 executed immediately beforehand.

In practice, functions which merely return 1 (*true*) or 0 (*false*) are very frequent.

The smallest ever of the standard UNIX utilities, */bin/true* and */bin/false* return 0 and 1 respectively, as an exit code. (Zero as an exit code usually means success, non-zero means error.)

1.5 Hello, world!

Let's use the famous example from the book [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)]:

Listing 1.13: C/C++ Code

```
#include <stdio.h>

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

1.5.1 x86

MSVC

Let's compile it in MSVC 2010:

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

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

Listing 1.14: 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 differences between Intel-syntax and AT&T-syntax will be discussed in 1.5.1 on page 15.

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[]`[Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)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/C++ strings: 5.4.1 on page 910.

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) [16].

After the function prologue we see the call to the `printf()` function:
`CALL _printf`. Before the call, a 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[17].

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 supposedly 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 1.15: Oracle RDBMS 10.2 Linux (app.o file)

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

However, MSVC can do the same.

Listing 1.16: MineSweeper from Windows 7 32-bit

```
.text:0102106F                 push    0
.text:01021071                 call    ds:time
.text:01021077                 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"[18] 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`).

[16] You can read more about it in the section about function prologues and epilogues (1.6 on page 39).
[17] CPU flags, however, are modified
[18] Wikipedia

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

The last instruction RET returns the control to the caller. Usually, this is C/C++ CRT[19] code which in turn returns control to the OS.

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[20].

Listing 1.17: code in IDA

```
main            proc near

var_10          = dword ptr -10h

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

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 saved onto the stack. In addition, the function prologue has `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).

`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.

[19]C Runtime library

[20]We could also have GCC produce assembly listings in Intel-syntax by applying the options `-S -masm=intel`.

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, …`).

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 1.18: let's compile in GCC 4.7.3

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

We get this:

Listing 1.19: 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"
        .section        .note.GNU-stack,"",@progbits
```

The listing contains many macros (the parts that begin with a dot). These are not interesting for us at the moment.

For now, for the sake of simplicity, 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 [21]:

Listing 1.20: 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 memorize 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 (→) [22].

- 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)

To go back to the compiled result: it is almost identical to what was displayed by IDA. There is one subtle difference: `0FFFFFFF0h` is presented as `$-16`. It's 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 set to 0 by using the usual `MOV`, not `XOR`. `MOV` just loads a value to a register. Its name is a misnomer (as the data is not moved but rather copied). In other architectures, this instruction is named "LOAD" or "STORE" or something similar.

[21] This GCC option can be used to eliminate "unnecessary" macros: *-fno-asynchronous-unwind-tables*

[22] 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.

String patching (Win32)

We can easily find the “hello, world” string in the executable file using Hiew:

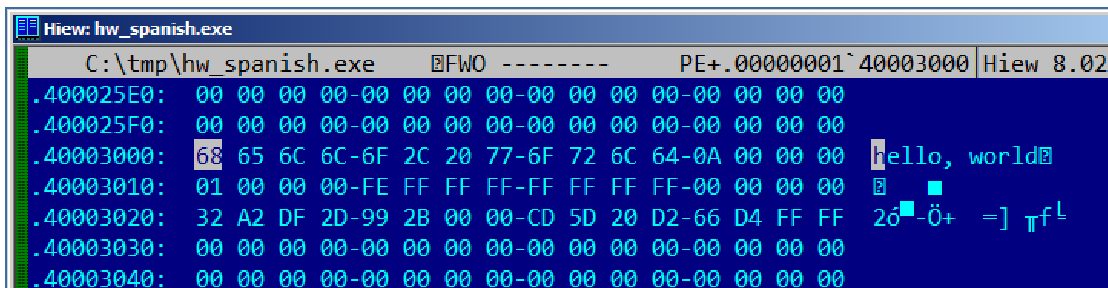


Figure 1.1: Hiew

And we can try to translate our message into Spanish:

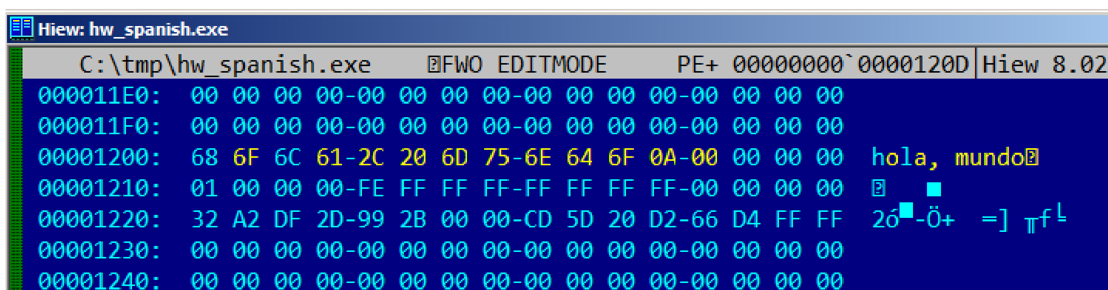


Figure 1.2: Hiew

The Spanish text is one byte shorter than English, so we also added the 0x0A byte at the end (\n) with a zero byte.

It works.

What if we want to insert a longer message? There are some zero bytes after original English text. It’s hard to say if they can be overwritten: they may be used somewhere in CRT code, or maybe not. Anyway, only overwrite them if you really know what you’re doing.

String patching (Linux x64)

Let’s try to patch a Linux x64 executable using rada.re:

Listing 1.21: rada.re session

```
dennis@bigbox ~/tmp % gcc hw.c
```

```
dennis@bigbox ~/tmp % radare2 a.out
 -- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHhello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x004005c4  6865 6c6c 6f2c 2077 6f72 6c64 0000 0000  hello, world....
0x004005d4  011b 033b 3000 0000 0500 0000 1cfe ffff  ...;0...........
0x004005e4  7c00 0000 5cfe ffff 4c00 0000 52ff ffff  |...\...L...R...
0x004005f4  a400 0000 6cff ffff c400 0000 dcff ffff  ....l...........
0x00400604  0c01 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400614  0178 1001 1b0c 0708 9001 0710 1400 0000  .x..............
0x00400624  1c00 0000 08fe ffff 2a00 0000 0000 0000  ........*.......
0x00400634  0000 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400644  0178 1001 1b0c 0708 9001 0000 2400 0000  .x..........$...
0x00400654  1c00 0000 98fd ffff 3000 0000 000e 1046  ........0......F
0x00400664  0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422  ..J..w...?.;*3$"
0x00400674  0000 0000 1c00 0000 4400 0000 a6fe ffff  ........D.......
0x00400684  1500 0000 0041 0e10 8602 430d 0650 0c07  .....A....C..P..
0x00400694  0800 0000 4400 0000 6400 0000 a0fe ffff  ....D...d.......
0x004006a4  6500 0000 0042 0e10 8f02 420e 188e 0345  e....B....B....E
0x004006b4  0e20 8d04 420e 288c 0548 0e30 8606 480e  . ..B.(..H.0..H.

[0x004005c4]> oo+
File a.out reopened in read-write mode

[0x004005c4]> w hola, mundo\x00

[0x004005c4]> q

dennis@bigbox ~/tmp % ./a.out
hola, mundo
```

Here's what's going on: I searched for the "hello" string using the / command, then I set the *cursor* (*seek*, in rada.re terms) to that address. Then I want to be sure that this is really that place: px dumps bytes there. oo+ switches rada.re to *read-write* mode. w writes an ASCII string at the current *seek*. Note the \00 at the end—this is a zero byte. q quits.

This is a real story of software cracking

An image processing software, when not registered, added watermarks, like "This image was processed by evaluation version of [software name]", across a picture. We tried at random: we found that string in the executable file and put spaces instead of it. Watermarks disappeared. Technically speaking, they continued to appear. With the help of Qt functions, the watermark was still added to the resulting image. But

adding spaces didn't alter the image itself...

Software *localization* of MS-DOS era

This method was a common way to translate MS-DOS software to Russian language back to 1980's and 1990's. This technique is available even for those who are not aware of machine code and executable file formats. The new string shouldn't be bigger than the old one, because there's a risk of overwriting another value or code there. Russian words and sentences are usually slightly longer than its English counterparts, so that is why *localized* software has a lot of weird acronyms and hardly readable abbreviations.

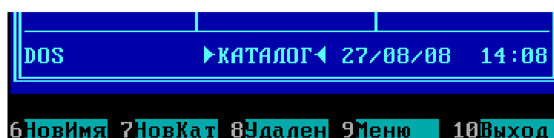


Figure 1.3: *Localized* Norton Commander 5.51

Perhaps this also happened to other languages during that era, in other countries.

As for Delphi strings, the string's size must also be corrected, if needed.

1.5.2 x86-64

MSVC: x86-64

Let's also try 64-bit MSVC:

Listing 1.22: MSVC 2012 x64

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

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

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 is a popular way to pass function arguments via registers (*fastcall*) 6.1.3 on page 949. I.e., a part of the function's arguments are passed in registers, and the rest—via the stack. In Win64, 4 function arguments are passed in the RCX, RDX, R8, and R9 registers. That is what we see here: a pointer to the string for `printf()` is now passed not in the stack, but rather 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:

<table>
<tr><td colspan="8">Byte number:</td></tr>
<tr><td>7th</td><td>6th</td><td>5th</td><td>4th</td><td>3rd</td><td>2nd</td><td>1st</td><td>0th</td></tr>
<tr><td colspan="8">RAX[x64]</td></tr>
<tr><td colspan="4"></td><td colspan="4">EAX</td></tr>
<tr><td colspan="6"></td><td colspan="2">AX</td></tr>
<tr><td colspan="6"></td><td>AH</td><td>AL</td></tr>
</table>

The `main()` function returns an *int*-typed value, which in C/C++ is still 32-bit, for better backward compatibility and portability, so that is why the EAX register is cleared at the function end (i.e., the 32-bit part of the register) instead of with RAX. There are also 40 bytes allocated in the local stack. This is called the “shadow space”, which we’ll talk about later: 1.14.2 on page 129.

GCC: x86-64

Let’s also try GCC in 64-bit Linux:

Listing 1.23: 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
```

Linux, *BSD and Mac OS X also use a method to pass function arguments in registers. [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [23].

The first 6 arguments are passed in the `RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `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). Why doesn’t it 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 (as stated in Intel manuals: 12.1.4 on page 1302).
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 [24]:

[23] Also available as `https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf`
[24] This must be enabled in **Options → Disassembly → Number of opcode bytes**

Listing 1.24: 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 has been cleared before the printf() function call. This is done because according to ABI[25] standard mentioned above, the number of used vector registers is to be passed in EAX in *NIX systems on x86-64.

Address patching (Win64)

If our example was compiled in MSVC 2013 using /MD switch (meaning a smaller executable due to MSVCR*.DLL file linkage), the main() function comes first, and can be easily found:

[25] Application Binary Interface

```
Hiew: hw2.exe
    C:\tmp\hw2.exe            ⊡FWO EDITMODE a64 PE+ 00000000`00000404 Hiew 8.02 (c)SEN
00000400: 4883EC28                       sub         rsp,028 ;'('
00000404: 488D0DF51F0000                 lea         rcx,[000002400]
0000040B: FF15D7100000                   call        q,[0000014E8]
00000411: 33C0                           xor         eax,eax
00000413: 4883C428                       add         rsp,028 ;'('
00000417: C3                             retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
00000418: 4883EC28                       sub         rsp,028 ;'('
0000041C: B84D5A0000                     mov         eax,000005A4D ;'  ZM'
00000421: 663905D8EFFFFF                 cmp         [-000000C00],ax
00000428: 7404                           jz          00000042E
000004
000004     lea         rcx,[00000000000002401]
000004               CommandSelect: Off
000004
000004
0000043F: 813850450000                   cmp         d,[rax],000004550 ;'  EP'
00000445: 75E3                           jnz         00000042A
00000447: B90B020000                     mov         ecx,00000020B
0000044C: 66394818                       cmp         [rax][018],cx
00000450: 75D8                           jnz         00000042A
00000452: 33C9                           xor         ecx,ecx
00000454: 83B8840000000E                 cmp         d,[rax][000000084],00E
0000045B: 7609                           jbe         000000466
0000045D: 3988F8000000                   cmp         [rax][0000000F8],ecx
1Help  2      3      4Select 5      6      7      8      9      10     11
```

Figure 1.4: Hiew

As an experiment, we can increment address by 1:

```
Hiew: hw2.exe
   C:\tmp\hw2.exe            FUO -------- a64 PE+.00000001`4000100B Hiew 8.02 (c)SEN
.40001000: 4883EC28                    sub         rsp,028 ;'('
.40001004: 488D0DF61F0000              lea         rcx,[00000001`40003001] ;'ello, w
.4000100B: FF15D7100000                call        printf
.40001011: 33C0                        xor         eax,eax
.40001013: 4883C428                    add         rsp,028 ;'('
.40001017: C3                          retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.40001018: 4883EC28                    sub         rsp,028 ;'('
.4000101C: B84D5A0000                  mov         eax,000005A4D ;' ZM'
.40001021: 663905D8EFFFFF              cmp         [00000001`40000000],ax
.40001028: 7404                        jz          .00000001`4000102E --2
.4000102A: 33C9                       5xor         ecx,ecx
.4000102C: EB38                        jmps        .00000001`40001066 --3
.4000102E: 48630507F0FFFF             2movsxd      rax,d,[00000001`4000003C] --4
.40001035: 488D0DC4EFFFFF              lea         rcx,[00000001`40000000]
.4000103C: 4803C1                      add         rax,rcx
.4000103F: 813850450000                cmp         d,[rax],000004550 ;' EP'
.40001045: 75E3                        jnz         .00000001`4000102A --5
.40001047: B90B020000                  mov         ecx,00000020B
.4000104C: 66394818                    cmp         [rax][018],cx
.40001050: 75D8                        jnz         .00000001`4000102A --5
.40001052: 33C9                        xor         ecx,ecx
.40001054: 83B8840000000E              cmp         d,[rax][000000084],00E
.4000105B: 7609                        jbe         .00000001`40001066 --3
.4000105D: 3988F8000000                cmp         [rax][0000000F8],ecx
1Help  2PutBlk 3Edit  4Mode  5Goto  6Refer  7Search 8Header 9Files 10Quit  11Hem
```

Figure 1.5: Hiew

Hiew shows “ello, world”. And when we run the patched executable, this very string is printed.

Pick another string from binary image (Linux x64)

The binary file I’ve got when I compile our example using GCC 5.4.0 on Linux x64 box has many other text strings. They are mostly imported function names and library names.

Run objdump to get the contents of all sections of the compiled file:

```
$ objdump -s a.out

a.out:     file format elf64-x86-64

Contents of section .interp:
 400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 400248 7838362d 36342e73 6f2e3200           x86-64.so.2.
Contents of section .note.ABI-tag:
 400254 04000000 10000000 01000000 474e5500  ............GNU.
 400264 00000000 02000000 06000000 20000000  ............ ...
Contents of section .note.gnu.build-id:
```

```
 400274 04000000 14000000 03000000 474e5500  ............GNU.
 400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10  .F.x[.......^...
 400294 cf3f7ae4                             .?z.

...
```

It's not a problem to pass address of the text string "/lib64/ld-linux-x86-64.so.2" to `printf()`:

```
#include <stdio.h>

int main()
{
    printf(0x400238);
    return 0;
}
```

It's hard to believe, but this code prints the aforementioned string.

If you would change the address to `0x400260`, the "GNU" string would be printed. This address is true for my specific GCC version, GNU toolset, etc. On your system, the executable may be slightly different, and all addresses will also be different. Also, adding/removing code to/from this source code will probably shift all addresses back or forward.

1.5.3 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 [26].
- GCC 4.9 (Linaro) (for ARM64), available as win32-executables at `http://www.linaro.org/projects/armv8/`.

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.

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 [27], but it is more important for us to see the instructions "as is" so let's see the compiled result in IDA.

[26] It is indeed so: Apple Xcode 4.6.3 uses open-source GCC as front-end compiler and LLVM code generator
[27] e.g. ARM mode lacks PUSH/POP instructions

Listing 1.25: 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}`[28], 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[30] 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[31] 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"[32].

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`[33] instruction calls the `printf()` function. Here's how this instruction works:

- store the address following the BL instruction (`0xC`) into the LR;

[28] STMFD[29]
[30] stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.
[31] Program Counter. IP/EIP/RIP in x86/64. PC in ARM.
[32] Read more about it in relevant section (6.4.1 on page 967)
[33] Branch with Link

- 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[34]-processors like x86, where the return address is usually stored on the stack. Read more about this in next section (1.9 on page 40).

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 recall, all ARM-mode instructions have a size of 4 bytes (32 bits). Hence, they can only be located 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, #0`[35] 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,PC`[36]. 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 has been 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.

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 1.26: Non-optimizing Keil 6/2013 (Thumb mode) + IDA

```
.text:00000000             main
```

[34]Complex Instruction Set Computing
[35]Meaning MOVe
[36]LDMFD[37] is an inverse instruction of STMFD

```
.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.

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 1.27: 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
__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 [*iOS ABI Function Call Guide*, (2010)][38]) is a

[38] Also available as http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf

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.

Keep in mind, 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 supposedly 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()`.

LLVM has 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 [39].

Why did the compiler replace the `printf()` with `puts()`? Presumably because `puts()` is faster [40].

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.

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

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

Listing 1.28: 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}

...
```

[39] 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.

[40] ciselant.de/projects/gcc_printf/gcc_printf.html

```
__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 `0xEx`.

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.

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[41] 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()`.

[41] Return Address

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)

Another example we can find in LAPACK library—a “Linear Algebra PACKage” written in FORTRAN. C/C++ developers also want to use LAPACK, but it’s insane to rewrite it to C/C++ and then maintain several versions. So there are short C functions callable from C/C++ environment, which are, in turn, call FORTRAN functions, and do almost anything else:

```
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();

    return F77NAME(ddot)(&n, &dx(0), &incx, &dy(0), &incy);
}
```

Also, functions like that are called “wrappers”.

ARM64

GCC

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

Listing 1.29: Non-optimizing GCC 4.8.1 + objdump

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

...

Contents of section .rodata:
 400640 01000200 00000000 48656c6c 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: .2.4 on page 1337. 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 and 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: 1.39.2 on page 555.

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[42] 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 made so 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: 1.39.3 on page 556). So several instructions must be utilized. 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: 1.39.4 on page 559.

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: 1.5.3 on page 28.

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:

[42] Frame Pointer

Listing 1.30: main() returning a value of uint64_t type

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

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

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

Listing 1.31: 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 values are 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.

1.5.4 MIPS

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 $register - 32768 ... register + 32767$ 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[43]") 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 11.6 on page 1284 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.

[43] Block Started by Symbol

Optimizing GCC

Let's consider the following example, which illustrates the "global pointer" concept.

Listing 1.32: Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
; \000 is zero byte in octal base:
        .ascii  "Hello, world!\012\000"
main:
; function prologue.
; set the GP:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
; save the RA to the local stack:
        sw      $31,28($sp)
; load the address of the puts() function from the GP to $25:
        lw      $25,%call16(puts)($28)
; load the address of the text string to $4 ($a0):
        lui     $4,%hi($LC0)
; jump to puts(), saving the return address in the link register:
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; branch delay slot
; restore the RA:
        lw      $31,28($sp)
; copy 0 from $zero to $v0:
        move    $2,$0
; return by jumping to the RA:
        j       $31
; function epilogue:
        addiu   $sp,$sp,32 ; branch delay slot + free local stack
```

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` (haven't you forgot *branch delay slots* yet?). The register $4 is also called $A0, which is used for passing the first function argument [44].

`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`.

[44]The MIPS registers table is available in appendix .3.1 on page 1338

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 pseudo instructions and the ALU[45] 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 pseudo name:

Listing 1.33: 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
```

[45] Arithmetic Logic Unit

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 [46]. The GP value has to be saved indeed, because each function can use its own 64KiB data window. The register containing the puts() address is called $T9, because registers prefixed with T- are called "temporaries" and their contents may not be preserved.

Non-optimizing GCC

Non-optimizing GCC is more verbose.

Listing 1.34: 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
```

[46]Apparently, functions generating listings are not so critical to GCC users, so some unfixed cosmetic bugs may still exist.

We see here that register FP is used as a pointer to the stack frame. We also see 3 NOPs. The second and third of which follow the branch instructions. Perhaps the GCC compiler always adds NOPs (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 1.35: 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.
; save the RA and FP in the stack:
.text:00000000                 addiu   $sp, -0x20
.text:00000004                 sw      $ra, 0x20+var_4($sp)
.text:00000008                 sw      $fp, 0x20+var_8($sp)
; set the FP (stack frame pointer):
.text:0000000C                 move    $fp, $sp
; set the GP:
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x20+var_10($sp)
; load the address of the text string:
.text:0000001C                 lui     $v0, (aHelloWorld >> 16)  # "Hello,
    world!"
.text:00000020                 addiu   $a0, $v0, (aHelloWorld & 0xFFFF)  #
    "Hello, world!"
; load the address of puts() using the GP:
.text:00000024                 lw      $v0, (puts & 0xFFFF)($gp)
.text:00000028                 or      $at, $zero ; NOP
; call puts():
.text:0000002C                 move    $t9, $v0
.text:00000030                 jalr    $t9
.text:00000034                 or      $at, $zero ; NOP
; restore the GP from local stack:
.text:00000038                 lw      $gp, 0x20+var_10($fp)
; set register $2 ($V0) to zero:
.text:0000003C                 move    $v0, $zero
; function epilogue.
; restore the SP:
.text:00000040                 move    $sp, $fp
; restore the RA:
.text:00000044                 lw      $ra, 0x20+var_4($sp)
; restore the FP:
.text:00000048                 lw      $fp, 0x20+var_8($sp)
.text:0000004C                 addiu   $sp, 0x20
; jump to the RA:
.text:00000050                 jr      $ra
.text:00000054                 or      $at, $zero ; NOP
```

Interestingly, IDA recognized the LUI/ADDIU instructions pair and coalesced them

into one LA (“Load Address”) pseudo instruction at line 15. We may also see that this pseudo instruction has a size of 8 bytes! This is a pseudo instruction (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.

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: 1.4.3 on page 11.

Optimizing GCC: load it into GDB

Listing 1.36: sample GDB session

```
root@debian-mips:~# gcc hw.c -O3 -o hw

root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
...
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)
```

1.5.5 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.

1.5.6 Exercises

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

1.6 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 instruction do: save the value of the EBP register on the stack, 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 caller:

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

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

1.6.1 Recursion

Epilogues and prologues can negatively affect the recursion performance.

More about recursion in this book: 3.7.3 on page 615.

1.7 An Empty Function: redux

Let's back to the empty function example 1.3 on page 7. Now that we know about function prologue and epilogue, this is an empty function 1.1 on page 7 compiled by non-optimizing GCC:

Listing 1.37: Non-optimizing GCC 8.2 x64 (assembly output)

```
f:
        push    rbp
        mov     rbp, rsp
        nop
        pop     rbp
        ret
```

It's RET, but function prologue and epilogue, probably, wasn't optimized and left as is. NOP is seems another compiler artifact. Anyway, the only effective instruction here is RET. All other instructions can be removed (or optimized).

1.8 Returning Values: redux

Again, when we know about function prologue and epilogue, let's recompile an example returning a value (1.4 on page 10, 1.8 on page 10) using non-optimizing GCC:

Listing 1.38: Non-optimizing GCC 8.2 x64 (assembly output)

```
f:
        push    rbp
        mov     rbp, rsp
        mov     eax, 123
        pop     rbp
        ret
```

Effective instructions here are MOV and RET, others are – prologue and epilogue.

1.9 Stack

The stack is one of the most fundamental data structures in computer science [47]. AKA[48] LIFO[49].

[47] wikipedia.org/wiki/Call_stack
[48] Also Known As
[49] Last In First Out

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.

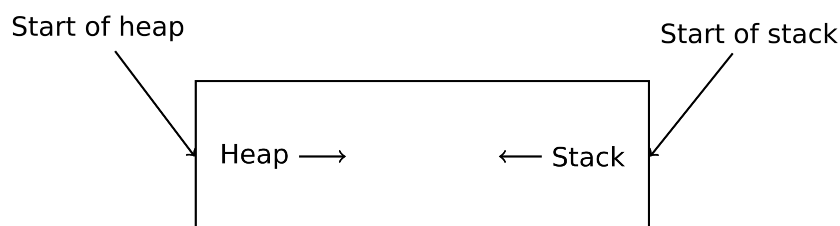
ARM supports both descending and ascending stacks.

For example the STMFD/LDMFD, STMED[50]/LDMED[51] instructions are intended to deal with a descending stack (grows downwards, starting with a high address and progressing to a lower one). The STMFA[52]/LDMFA[53], STMEA[54]/LDMEA[55] instructions are intended to deal with an ascending stack (grows upwards, starting from a low address and progressing to a higher one).

1.9.1 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 [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)][56]we can read:

[50]Store Multiple Empty Descending (ARM instruction)
[51]Load Multiple Empty Descending (ARM instruction)
[52]Store Multiple Full Ascending (ARM instruction)
[53]Load Multiple Full Ascending (ARM instruction)
[54]Store Multiple Empty Ascending (ARM instruction)
[55]Load Multiple Empty Ascending (ARM instruction)
[56]Also available as URL

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

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.

1.9.2 What is the stack used for?

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
; 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*[57] instead:

```
?f@@YAXXZ PROC                      ; f
; 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.

ARM

ARM programs also use the stack for saving return addresses, but differently. As mentioned in "Hello, world!" (1.5.3 on page 24), 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*[58]. 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 [59]. This can be also useful for situations when memory for the stack is not yet allocated or not available.

[57] irony here

[58] infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

[59] 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 [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

Some examples of leaf functions: 1.14.3 on page 133, 1.14.3 on page 133, 1.282 on page 395, 1.298 on page 417, 1.28.5 on page 417, 1.192 on page 264, 1.190 on page 262, 1.209 on page 285.

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 IDA as arg_0
ESP+8	argument#2, marked in IDA as arg_4
ESP+0xC	argument#3, marked in IDA as arg_8
…	…

For more information on other calling conventions see also section (6.1 on page 947).

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()) can 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[60], 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
...
```

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,

[60] Not random in strict sense, but rather unpredictable: 1.9.4 on page 50

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.

Another related example: 6.1.10.

Alternative ways of passing arguments

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.

A somewhat popular way among assembly language newbies is to pass arguments via global variables, like:

Listing 1.39: Assembly code

```
        ...

        mov     X, 123
        mov     Y, 456
        call    do_something

        ...

X       dd      ?
Y       dd      ?

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

But this method has obvious drawback: *do_something()* function cannot call itself recursively (or via another function), because it has to zap its own arguments. The same story with local variables: if you hold them in global variables, the function couldn't call itself. And this is also not thread-safe [61]. A method to store such information in stack makes this easier—it can hold as many function arguments and/or values, as much space it has.

[Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] mentions even weirder schemes particularly convenient on IBM System/360.

MS-DOS had a way of passing all function arguments via registers, for example, this is piece of code for ancient 16-bit MS-DOS prints "Hello, world!":

```
mov  dx, msg      ; address of message
mov  ah, 9        ; 9 means "print string" function
int  21h          ; DOS "syscall"
```

[61] Correctly implemented, each thread would have its own stack with its own arguments/variables.

```
mov  ah, 4ch        ; "terminate program" function
int  21h            ; DOS "syscall"

msg  db 'Hello, World!\$'
```

This is quite similar to 6.1.3 on page 949 method. And also it's very similar to calling syscalls in Linux (6.3.1 on page 966) and Windows.

If a MS-DOS function is going to return a boolean value (i.e., single bit, usually indicating error state), CF flag was often used.

For example:

```
mov ah, 3ch         ; create file
lea dx, filename
mov cl, 1
int 21h
jc  error
mov file_handle, ax
...
error:
...
```

In case of error, CF flag is raised. Otherwise, handle of newly created file is returned via AX.

This method is still used by assembly language programmers. In Windows Research Kernel source code (which is quite similar to Windows 2003) we can find something like this (file *base/ntos/ke/i386/cpu.asm*):

```
        public  Get386Stepping
Get386Stepping  proc

        call    MultiplyTest            ; Perform multiplication test
        jnc     short G3s00             ; if nc, muttest is ok
        mov     ax, 0
        ret
G3s00:
        call    Check386B0              ; Check for B0 stepping
        jnc     short G3s05             ; if nc, it's B1/later
        mov     ax, 100h                ; It is B0/earlier stepping
        ret

G3s05:
        call    Check386D1              ; Check for D1 stepping
        jc      short G3s10             ; if c, it is NOT D1
        mov     ax, 301h                ; It is D1/later stepping
        ret

G3s10:
        mov     ax, 101h                ; assume it is B1 stepping
        ret
```

```
        ...

MultiplyTest    proc

        xor     cx,cx                   ; 64K times is a nice round number
mlt00:  push    cx
        call    Multiply                ; does this chip's multiply work?
        pop     cx
        jc      short mltx              ; if c, No, exit
        loop    mlt00                   ; if nc, YEs, loop to try again
        clc
mltx:
        ret

MultiplyTest    endp
```

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.

x86: alloca() function

It is worth noting the `alloca()` function [62]. This function works like `malloc()`, but allocates memory directly on the stack. The allocated memory chunk does not have to be freed via a `free()` function call,
since the function epilogue (1.6 on page 39) 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, making ESP pointing 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
```

[62] 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\crt\src\intel`

```
#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 1.40: MSVC 2010

```
...

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

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

    push    esi
    call    _puts
    add     esp, 28

...
```

The sole `alloca()` argument is passed via EAX (instead of pushing it into the stack) [63].

GCC + Intel syntax

GCC 4.4.1 does the same without calling external functions:

[63] It is because alloca() is rather a compiler intrinsic (11.3 on page 1278) 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[64] alloca() implementation also has code which reads from the memory just allocated, in order to let the OS map physical memory to this VM[65] region. After the `alloca()` call, ESP points to the block of 600 bytes and we can use it as memory for the `buf` array.

Listing 1.41: 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-byte 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 1.42: GCC 4.7.3

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

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)`.

(Windows) SEH

SEH[66] records are also stored on the stack (if they are present). Read more about it: (6.5.3 on page 988).

Buffer overflow protection

More about it here (1.26.2 on page 340).

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.

1.9.3 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 IDA as `var_8`
ESP-8	local variable#1, marked in IDA as `var_4`
ESP-4	saved value ofEBP
ESP	Return Address
ESP+4	argument#1, marked in IDA as `arg_0`
ESP+8	argument#2, marked in IDA as `arg_4`
ESP+0xC	argument#3, marked in IDA as `arg_8`
…	…

1.9.4 Noise in stack

> When one says that something seems random, what one usually means in practice is that one cannot see any regularities in it.
>
> Stephen Wolfram, A New Kind of Science.

[66] Structured Exception Handling

Often in this book “noise” or “garbage” values in the stack or memory are mentioned. Where do they come from? These are what has been left 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();
        f2();
};
```

Compiling ...

Listing 1.43: 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:

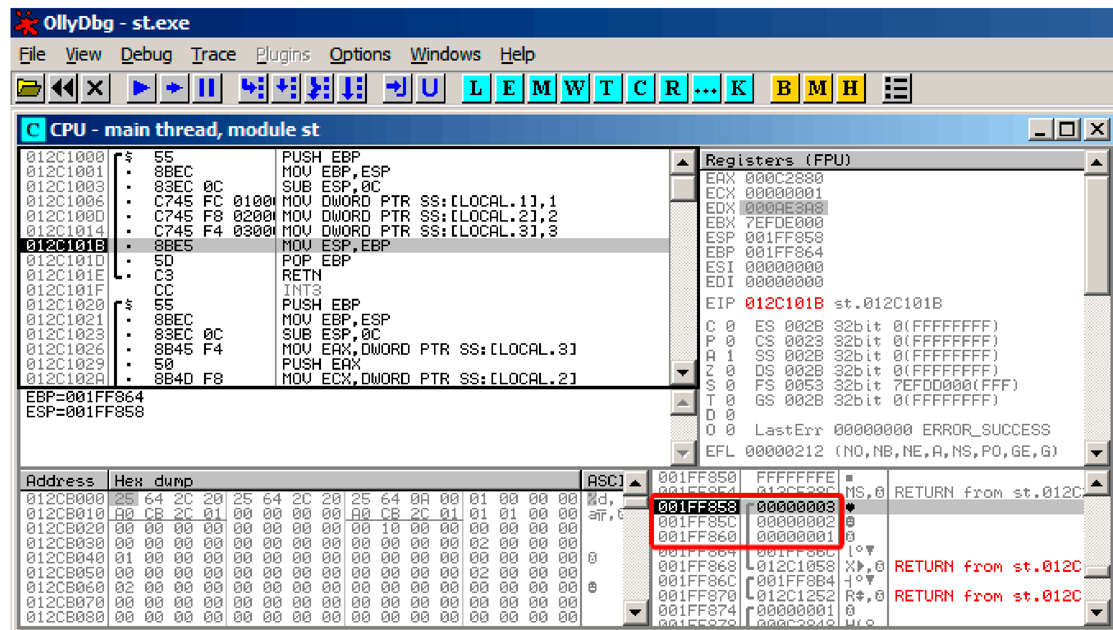


Figure 1.6: OllyDbg: f1()

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

And when f2() executes:

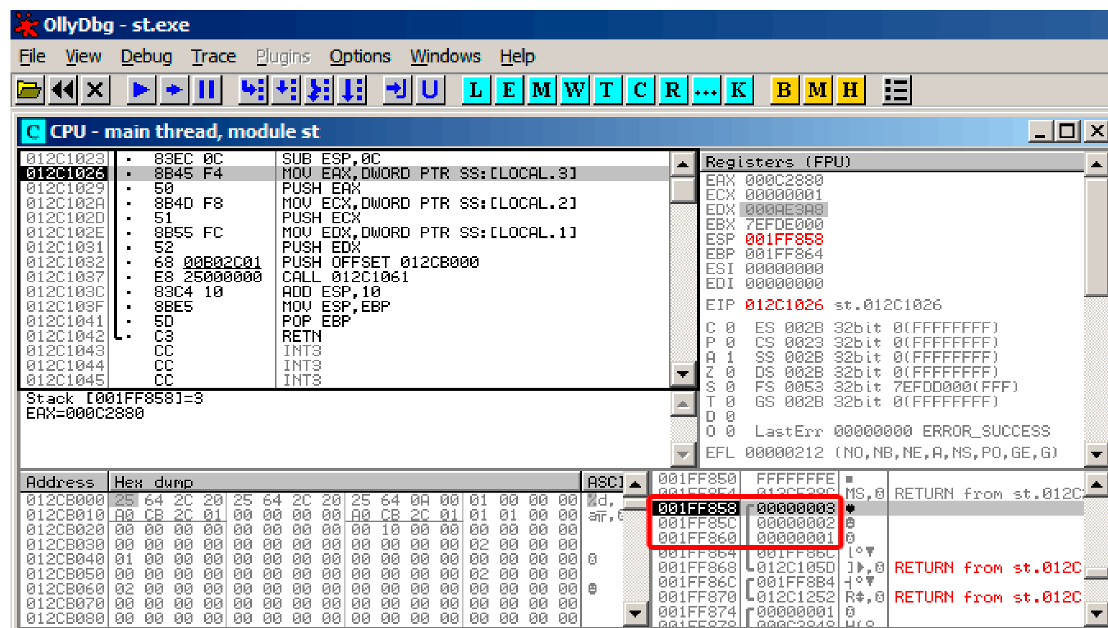


Figure 1.7: OllyDbg: f2()

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

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 1.44: 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 supposedly something changed inside of compiler guts, so it behaves slightly different.

1.9.5 Exercises

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

1.10 Almost empty function

This is a real piece of code I found in Boolector[67]:

```
// forward declaration. the function is residing in some other module:
int boolector_main (int argc, char **argv);

// executable
int main (int argc, char **argv)
{
        return boolector_main (argc, argv);
}
```

Why would anyone do so? It's unclear, but we can guess that `boolector_main()` may be compiled in some kind of DLL or dynamic library, and be called from a test suite. Surely, a test suite can prepare `argc/argv` variables as CRT would do it.

Interestingly enough, how this compiles:

[67] https://boolector.github.io/

Listing 1.45: Non-optimizing GCC 8.2 x64 (assembly output)

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR -4[rbp], edi
        mov     QWORD PTR -16[rbp], rsi
        mov     rdx, QWORD PTR -16[rbp]
        mov     eax, DWORD PTR -4[rbp]
        mov     rsi, rdx
        mov     edi, eax
        call    boolector_main
        leave
        ret
```

We've got here: prologue, unnecessary (not optimized) shuffling of two arguments, CALL, epilogue, RET. But let's see optimizing version:

Listing 1.46: Optimizing GCC 8.2 x64 (assembly output)

```
main:
        jmp     boolector_main
```

As simple as that: stack/registers are untouched and `boolector_main()` has the same arguments set. So all we need to do is pass execution to another address.

This is close to thunk function.

We will see something more advanced later: 1.11.2 on page 71, 1.21.1 on page 197.

1.11 printf() with several arguments

Now let's extend the *Hello, world!* (1.5 on page 11) 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;
};
```

1.11.1 x86

x86: 3 integer 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 (6.1 on page 947).

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 1.47: 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 1.8: OllyDbg: the very start of the main() function

Click on the PUSH EBP instruction, press F2 (set breakpoint) and press F9 (run). We have 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:

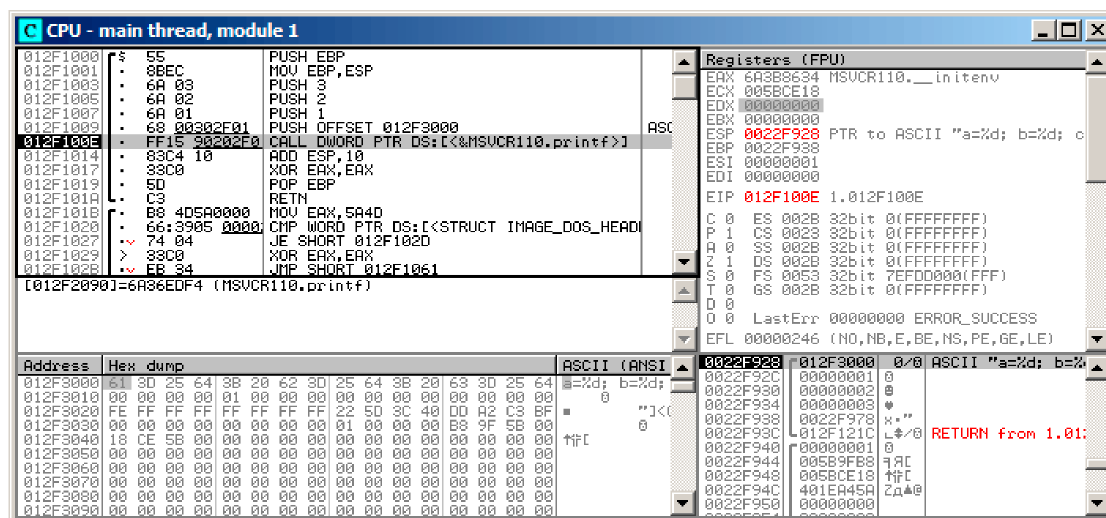


Figure 1.9: OllyDbg: before `printf()` execution

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:

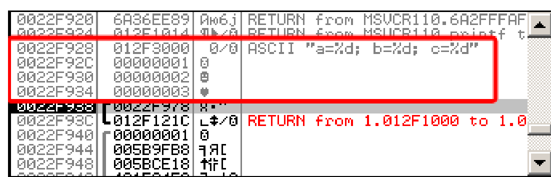


Figure 1.10: OllyDbg: stack after the argument values have been pushed (The red rectangular border was added by the author in a graphics editor)

We can see 3 columns there: address in the stack, value in the stack and some additional OllyDbg comments. OllyDbg can detect pointers to ASCII strings in stack, so it reports the `printf()`-string here.

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:

```
a=1; b=2; c=3
```

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

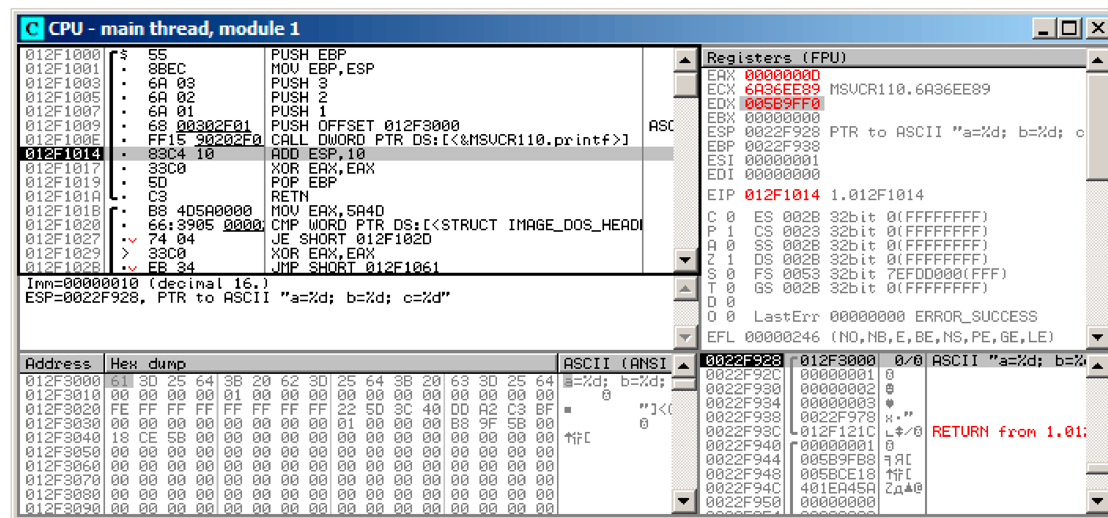


Figure 1.11: OllyDbg after `printf()` execution

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 behavior: 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:

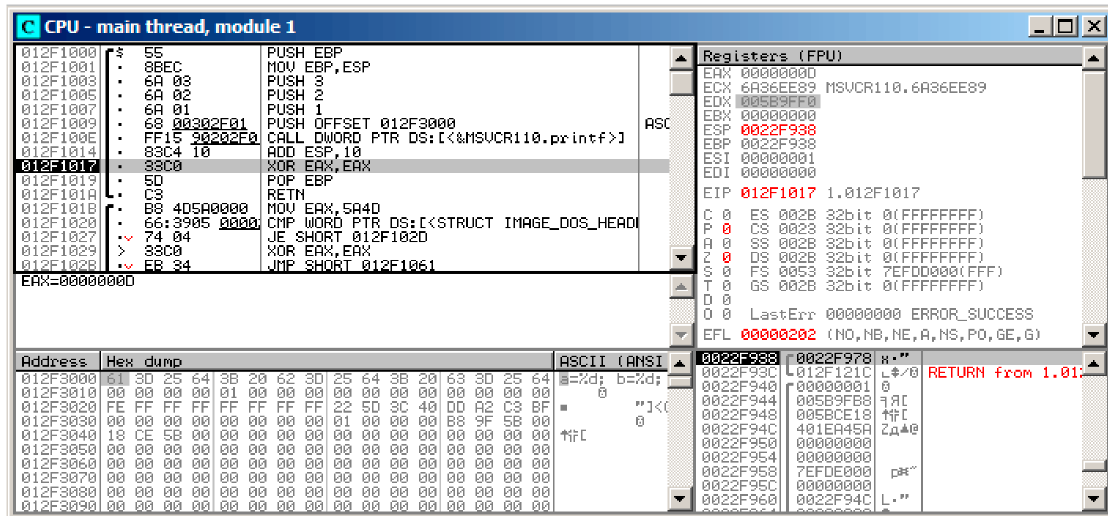


Figure 1.12: OllyDbg: after ADD ESP, 10 instruction execution

ESP has changed, but the values are still in the stack! Yes, of course; no one needs to set these values to zeros 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[68] in Linux.

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

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

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.48: 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
   0x804844a <main+45>: mov    $0x0,%eax
   0x804844f <main+50>: leave
   0x8048450 <main+51>: ret
```

[68] GNU Debugger

```
   0x8048451:   xchg   %ax,%ax
   0x8048453:   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
0x80484f0:      "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    $0xfffffff0,%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. But 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,0xfffffff0
   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 line of C/C++ code. 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>
...
```

x64: 8 integer 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 1.49: MSVC 2012 x64

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

main    PROC
        sub     rsp, 88

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

        ; 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 recall: 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: 1.5.2 on page 21.

We also noted earlier that the EAX register has been cleared before a printf() call: 1.5.2 on page 21.

Listing 1.50: 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:.LC0
        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
...
Reading symbols from /home/dennis/polygon/2...done.
```

Listing 1.51: 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            0x7fffffffdce0   140737488346336
r11            0x7ffff7a65f60   140737348263776
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x7ffff7a65f60   0x7ffff7a65f60 <__printf>
...
```

Listing 1.52: 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: 0x00007fffffffe048      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 are of *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
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            0x7fffffffd5b0   140737488344496
r11            0x7ffff7a95900   140737348458752
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
```

```
r15            0x0      0
rip            0x40057b 0x40057b <main+78>
...
```

1.11.2 ARM

ARM: 3 integer 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 (6.1.3 on page 949) or win64 (6.1.5 on page 951).

32-bit ARM

Non-optimizing Keil 6/2013 (ARM mode)

Listing 1.53: 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 1.54: 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 1.55: 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 has been called then the control from printf() will be returned to that point. Therefore we do not have to save LR because we do not have necessity to modify LR. And we do not have necessity 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: 1.21.1 on page 198.

A somewhat simpler case was described above: 1.10 on page 55.

ARM64

Non-optimizing GCC (Linaro) 4.9

Listing 1.56: 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. 1, 2 and 3 are 32-bit *int* values, so they are loaded into 32-bit register parts [69]

Optimizing GCC (Linaro) 4.9 generates the same code.

ARM: 8 integer arguments

Let's use again the example with 9 arguments from the previous section: 1.11.1 on page 66.

```
#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;
};
```

[69]Changing 1 to 1L will make it a 64-bit value that would be loaded into 64-bit register. See more about integer constants/literals: 1, 2.

Optimizing Keil 6/2013: ARM mode

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

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: 1.39.2 on page 555.

  The second `SUB SP, SP, #0x14` instruction decreases SP (the stack pointer) in order to allocate `0x14` (20) bytes on the stack. Indeed, we have 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.
- `printf()` call.
- Function epilogue:

  The `ADD SP, SP, #0x14` instruction restores the SP pointer back to its former value, thus annulling everything what has been stored on the stack. Of course, what has been 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 ($4 + var_4 = 4 + (-4) = 0$), so this instruction is analogous to `LDR PC, [SP],#4`), and then SP is increased by 4. This is referred as *post-index*[70]. 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[71].

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
```

[70] Read more about it: 1.39.2 on page 555.
[71] It is impossible to set `IP/EIP/RIP` value using `POP` in x86, but anyway, you got the analogy right.

```
.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
__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/Thumb 2-instructions are used instead.

ARM64

Non-optimizing GCC (Linaro) 4.9

Listing 1.57: 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 frame pointer (FP=SP+16):
        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: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][72]. 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.

1.11.3 MIPS

3 integer 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

[72] Also available as http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

The rest of the book (>1000 pages) is available to members of Yurichev's private club: https://yurichev.com/club/.

Join a club by becoming my patron at patreon. Or, by sending any amount of bitcoins to the following address: bc1qh9fem2j37lm5gqyesuefprqpmx2sj75ucvfnzs, and also sending the Transaction ID by email: *club@yurichev.com*.