Reverse Engineering for Beginners

Dennis Yurichev
<dennis(a)yurichev.com>

©2013-2015, Dennis Yurichev.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit
http://creativecommons.org/licenses/by-nc-nd/3.0/.

Text version (June 12, 2016).
The latest version (and Russian edition) of this text is accessible at beginners.re. An A4-format version is also available.

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

You can also follow me on twitter to get information about updates of this text: @yurichev¹, or subscribe to the mailing list².

The cover was made by Andy Nechaevsky: facebook.

¹twitter.com/yurichev
²yurichev.com
Call for translators!

You may want to help me with translation 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.

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

Korean, Chinese and Persian languages are reserved by publishers.

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

So do not hesitate to contact me: dennis(a)yurichev.com.
# Abridged contents

<table>
<thead>
<tr>
<th>Level</th>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Code patterns</td>
<td>1</td>
</tr>
<tr>
<td>II</td>
<td>Important fundamentals</td>
<td>615</td>
</tr>
<tr>
<td>III</td>
<td>Slightly more advanced examples</td>
<td>627</td>
</tr>
<tr>
<td>IV</td>
<td>Java</td>
<td>856</td>
</tr>
<tr>
<td>V</td>
<td>Finding important/interesting stuff in the code</td>
<td>914</td>
</tr>
<tr>
<td>VI</td>
<td>OS-specific</td>
<td>951</td>
</tr>
<tr>
<td>VII</td>
<td>Tools</td>
<td>1036</td>
</tr>
<tr>
<td>VIII</td>
<td>Examples of real-world RE tasks</td>
<td>1043</td>
</tr>
<tr>
<td>IX</td>
<td>Examples of reversing proprietary file formats</td>
<td>1218</td>
</tr>
<tr>
<td>X</td>
<td>Other things</td>
<td>1255</td>
</tr>
</tbody>
</table>
# Contents

## I Code patterns

### 1 A short introduction to the CPU

1.1 A couple of words about different ISA\(^4\)s ........................................... 5

### 2 The simplest Function

2.1 x86 ................................................................. 6
2.2 ARM .............................................................. 7
2.3 MIPS .............................................................. 7
2.3.1 A note about MIPS instruction/register names ....................... 8

### 3 Hello, world!

3.1 x86 ................................................................. 9
3.1.1 MSVC ......................................................... 9
3.1.2 GCC ......................................................... 11
3.1.3 GCC: AT&T syntax ......................................... 13
3.2 x86-64 ........................................................... 15
3.2.1 MSVC—x86-64 .............................................. 15
3.2.2 GCC—x86-64 ................................................ 16
3.3 GCC—one more thing ........................................ 17
3.4 ARM ............................................................. 18
3.4.1 Non-optimizing Keil 6/2013 (ARM mode) ......................... 19
3.4.2 Non-optimizing Keil 6/2013 (Thumb mode) ....................... 21
3.4.3 Optimizing Xcode 4.6.3 (LLVM) (ARM mode) .................. 22
3.4.4 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode) .............. 23
3.4.5 ARM64 ....................................................... 26
3.5 MIPS ............................................................. 28
3.5.1 A word about the “global pointer” ................................ 28
3.5.2 Optimizing GCC ........................................... 29

\(^4\)Instruction Set Architecture
# 16.2 Division

- 16.2.1 Division using shifts

# 16.3 Exercise

# 17 Floating-point unit

- 17.1 IEEE 754
- 17.2 x86
- 17.3 ARM, MIPS, x86/x64 SIMD
- 17.4 C/C++
- 17.5 Simple example
  - 17.5.1 x86
  - 17.5.2 ARM: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)
  - 17.5.3 ARM: Optimizing Keil 6/2013 (Thumb mode)
  - 17.5.4 ARM64: Optimizing GCC (Linaro) 4.9
  - 17.5.5 ARM64: Non-optimizing GCC (Linaro) 4.9
  - 17.5.6 MIPS
- 17.6 Passing floating point numbers via arguments
  - 17.6.1 x86
  - 17.6.2 ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)
  - 17.6.3 ARM + Non-optimizing Keil 6/2013 (ARM mode)
  - 17.6.4 ARM64 + Optimizing GCC (Linaro) 4.9
  - 17.6.5 MIPS
- 17.7 Comparison example
  - 17.7.1 x86
  - 17.7.2 ARM
  - 17.7.3 ARM64
  - 17.7.4 MIPS
- 17.8 Stack, calculators and reverse Polish notation
- 17.9 x64
- 17.10 Exercises

# 18 Arrays

- 18.1 Simple example
  - 18.1.1 x86
  - 18.1.2 ARM
  - 18.1.3 MIPS
- 18.2 Buffer overflow
  - 18.2.1 Reading outside array bounds
  - 18.2.2 Writing beyond array bounds
- 18.3 Buffer overflow protection methods
  - 18.3.1 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)
- 18.4 One more word about arrays
- 18.5 Array of pointers to strings
  - 18.5.1 x64
18.5.2 32-bit ARM .................................................. 387
18.5.3 ARM64 ...................................................... 389
18.5.4 MIPS ......................................................... 390
18.5.5 Array overflow ............................................. 391
18.6 Multidimensional arrays ..................................... 394
18.6.1 Two-dimensional array example ......................... 396
18.6.2 Access two-dimensional array as one-dimensional ... 397
18.6.3 Three-dimensional array example ....................... 400
18.6.4 More examples ............................................. 405
18.7 Pack of strings as a two-dimensional array ............... 405
18.7.1 32-bit ARM .................................................. 408
18.7.2 ARM64 ...................................................... 408
18.7.3 MIPS ......................................................... 409
18.7.4 Conclusion .................................................. 410
18.8 Conclusion ..................................................... 410
18.9 Exercises ....................................................... 410

19 Manipulating specific bit(s) .................................. 411
19.1 Specific bit checking ......................................... 411
19.1.1 x86 ......................................................... 411
19.1.2 ARM ......................................................... 415
19.2 Setting and clearing specific bits .......................... 417
19.2.1 x86 .......................................................... 418
19.2.2 ARM + Optimizing Keil 6/2013 (ARM mode) .......... 424
19.2.3 ARM + Optimizing Keil 6/2013 (Thumb mode) ....... 425
19.2.4 ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode) 425
19.2.5 ARM: more about the BIC instruction .................. 426
19.2.6 ARM64: Optimizing GCC (Linaro) 4.9 ................ 426
19.2.7 ARM64: Non-optimizing GCC (Linaro) 4.9 .......... 426
19.2.8 MIPS ......................................................... 427
19.3 Shifts ............................................................ 427
19.4 Setting and clearing specific bits: FPU\(^5\) example .... 428
19.4.1 A word about the XOR operation ....................... 429
19.4.2 x86 .......................................................... 429
19.4.3 MIPS ......................................................... 431
19.4.4 ARM .......................................................... 432
19.5 Counting bits set to 1 ........................................ 435
19.5.1 x86 .......................................................... 437
19.5.2 x64 .......................................................... 445
19.5.3 ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode) 449
19.5.4 ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode) 450
19.5.5 ARM64 + Optimizing GCC 4.9 ......................... 450

\(^5\)Floating-point unit
## Contents

### 22.1 Pseudo-random number generator example
- 22.1.1 x86 .................................................. 517
- 22.1.2 MIPS .................................................. 518
- 22.1.3 ARM (ARM mode) ............................. 520

### 22.2 Calculating machine epsilon
- 22.2.1 x86 .................................................. 523
- 22.2.2 ARM64 ............................................... 524
- 22.2.3 MIPS ................................................ 524
- 22.2.4 Conclusion ...................................... 525

### 22.3 Fast square root calculation

### 23 Pointers to functions
- 23.1 MSVC ............................................... 528
  - 23.1.1 MSVC + OllyDbg ............................ 532
  - 23.1.2 MSVC + tracer ............................. 534
  - 23.1.3 MSVC + tracer (code coverage) .... 536
- 23.2 GCC .................................................. 536
  - 23.2.1 GCC + GDB (with source code) .... 538
  - 23.2.2 GCC + GDB (no source code) ..... 539

### 24 64-bit values in 32-bit environment
- 24.1 Returning of 64-bit value ...................... 543
  - 24.1.1 x86 ............................................. 543
  - 24.1.2 ARM ............................................ 544
  - 24.1.3 MIPS .......................................... 544
- 24.2 Arguments passing, addition, subtraction
  - 24.2.1 x86 ............................................. 545
  - 24.2.2 ARM ............................................ 547
  - 24.2.3 MIPS .......................................... 548
- 24.3 Multiplication, division ......................... 550
  - 24.3.1 x86 ............................................. 550
  - 24.3.2 ARM ............................................ 552
  - 24.3.3 MIPS .......................................... 554
- 24.4 Shifting right .......................... 555
  - 24.4.1 x86 ............................................. 555
  - 24.4.2 ARM ............................................ 556
  - 24.4.3 MIPS .......................................... 556
- 24.5 Converting 32-bit value into 64-bit one ............................. 557
  - 24.5.1 x86 ............................................. 557
  - 24.5.2 ARM ............................................ 557
  - 24.5.3 MIPS .......................................... 558

### 25 SIMD
- 25.1 Vectorization ..................................... 560
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.1.1</td>
<td>Addition example</td>
</tr>
<tr>
<td>25.1.2</td>
<td>Memory copy example</td>
</tr>
<tr>
<td>25.2</td>
<td>SIMD strlen() implementation</td>
</tr>
<tr>
<td>26</td>
<td>64 bits</td>
</tr>
<tr>
<td>26.1</td>
<td>x86-64</td>
</tr>
<tr>
<td>26.2</td>
<td>ARM</td>
</tr>
<tr>
<td>26.3</td>
<td>Float point numbers</td>
</tr>
<tr>
<td>27</td>
<td>Working with floating point numbers using SIMD</td>
</tr>
<tr>
<td>27.1</td>
<td>Simple example</td>
</tr>
<tr>
<td>27.1.1</td>
<td>x64</td>
</tr>
<tr>
<td>27.1.2</td>
<td>x86</td>
</tr>
<tr>
<td>27.2</td>
<td>Passing floating point number via arguments</td>
</tr>
<tr>
<td>27.3</td>
<td>Comparison example</td>
</tr>
<tr>
<td>27.3.1</td>
<td>x64</td>
</tr>
<tr>
<td>27.3.2</td>
<td>x86</td>
</tr>
<tr>
<td>27.4</td>
<td>Calculating machine epsilon: x64 and SIMD</td>
</tr>
<tr>
<td>27.5</td>
<td>Pseudo-random number generator example revisited</td>
</tr>
<tr>
<td>27.6</td>
<td>Summary</td>
</tr>
<tr>
<td>28</td>
<td>ARM-specific details</td>
</tr>
<tr>
<td>28.1</td>
<td>Number sign (#) before number</td>
</tr>
<tr>
<td>28.2</td>
<td>Addressing modes</td>
</tr>
<tr>
<td>28.3</td>
<td>Loading a constant into a register</td>
</tr>
<tr>
<td>28.3.1</td>
<td>32-bit ARM</td>
</tr>
<tr>
<td>28.3.2</td>
<td>ARM64</td>
</tr>
<tr>
<td>28.4</td>
<td>Relocs in ARM64</td>
</tr>
<tr>
<td>29</td>
<td>MIPS-specific details</td>
</tr>
<tr>
<td>29.1</td>
<td>Loading constants into register</td>
</tr>
<tr>
<td>29.2</td>
<td>Further reading about MIPS</td>
</tr>
</tbody>
</table>

### II Important fundamentals

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>Signed number representations</td>
</tr>
<tr>
<td>31</td>
<td>Endianness</td>
</tr>
<tr>
<td>31.1</td>
<td>Big-endian</td>
</tr>
<tr>
<td>31.2</td>
<td>Little-endian</td>
</tr>
<tr>
<td>31.3</td>
<td>Example</td>
</tr>
<tr>
<td>31.4</td>
<td>Bi-endian</td>
</tr>
<tr>
<td>31.5</td>
<td>Converting data</td>
</tr>
</tbody>
</table>
CONTENTS

32 Memory 622

33 CPU 624
  33.1 Branch predictors 624
  33.2 Data dependencies 624

34 Hash functions 625
  34.1 How do one-way functions work? 625

III Slightly more advanced examples 627

35 Temperature converting 628
  35.1 Integer values 628
    35.1.1 Optimizing MSVC 2012 x86 629
    35.1.2 Optimizing MSVC 2012 x64 631
  35.2 Floating-point values 631

36 Fibonacci numbers 636
  36.1 Example #1 636
  36.2 Example #2 641
  36.3 Summary 646

37 CRC32 calculation example 647

38 Network address calculation example 653
  38.1 calc_network_address() 655
  38.2 form_IP() 656
  38.3 print_as_IP() 658
  38.4 form_netmask() and set_bit() 660
  38.5 Summary 661

39 Loops: several iterators 663
  39.1 Three iterators 663
  39.2 Two iterators 664
  39.3 Intel C++ 2011 case 667

40 Duff’s device 671

41 Division by 9 676
  41.1 x86 676
  41.2 ARM 678
    41.2.1 Optimizing Xcode 4.6.3 (LLVM) (ARM mode) 678
    41.2.2 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode) 679
    41.2.3 Non-optimizing Xcode 4.6.3 (LLVM) and Keil 6/2013 679
41.3 MIPS ............................................................. 680
41.4 How it works ...................................................... 680
  41.4.1 More theory .................................................. 682
41.5 Getting the divisor .............................................. 682
  41.5.1 Variant #1 .................................................. 682
  41.5.2 Variant #2 .................................................. 684
41.6 Exercise .......................................................... 684
42 String to number conversion (atoi()) ............................ 685
  42.1 Simple example ................................................. 685
    42.1.1 Optimizing MSVC 2013 x64 ................................. 686
    42.1.2 Optimizing GCC 4.9.1 x64 ................................ 687
    42.1.3 Optimizing Keil 6/2013 (ARM mode) ...................... 688
    42.1.4 Optimizing Keil 6/2013 (Thumb mode) .................... 688
    42.1.5 Optimizing GCC 4.9.1 ARM64 ............................. 689
  42.2 A slightly advanced example ................................ 690
    42.2.1 Optimizing GCC 4.9.1 x64 ................................. 691
    42.2.2 Optimizing Keil 6/2013 (ARM mode) ...................... 693
42.3 Exercise .......................................................... 694
43 Inline functions ................................................... 695
  43.1 Strings and memory functions ................................. 696
    43.1.1 strcmp() .................................................. 696
    43.1.2 strlen() .................................................. 699
    43.1.3 strcpy() .................................................. 700
    43.1.4 memset() .................................................. 700
    43.1.5 memcpy() .................................................. 702
    43.1.6 memcmp() .................................................. 706
    43.1.7 IDA script ................................................ 707
44 C99 restrict ........................................................ 708
45 Branchless abs() function ....................................... 712
  45.1 Optimizing GCC 4.9.1 x64 ................................... 712
  45.2 Optimizing GCC 4.9 ARM64 .................................. 713
46 Variadic functions ................................................. 714
  46.1 Computing arithmetic mean ................................... 714
    46.1.1 cdecl calling conventions ................................ 715
    46.1.2 Register-based calling conventions ....................... 716
  46.2 vprintf() function case ...................................... 719
47 Strings trimming .................................................. 722
  47.1 x64: Optimizing MSVC 2013 ................................... 723
## Contents

- 47.2 x64: Non-optimizing GCC 4.9.1 ........................................ 725
- 47.3 x64: Optimizing GCC 4.9.1 ........................................... 727
- 47.4 ARM64: Non-optimizing GCC (Linaro) 4.9 ....................... 729
- 47.5 ARM64: Optimizing GCC (Linaro) 4.9 ............................. 730
- 47.6 ARM: Optimizing Keil 6/2013 (ARM mode) ..................... 731
- 47.7 ARM: Optimizing Keil 6/2013 (Thumb mode) ................... 732
- 47.8 MIPS ................................................................. 733

## 48 toupper() function

- 48.1 x64 ................................................................. 737
  - 48.1.1 Two comparison operations .................................... 737
  - 48.1.2 One comparison operation .................................... 738
- 48.2 ARM ................................................................. 739
  - 48.2.1 GCC for ARM64 .................................................. 740
- 48.3 Summary ............................................................... 741

## 49 Incorrectly disassembled code

- 49.1 Disassembling from an incorrect start (x86) .................... 742
- 49.2 How does random noise looks disassembled? .................. 743

## 50 Obfuscation

- 50.1 Text strings .......................................................... 750
- 50.2 Executable code ........................................................ 751
  - 50.2.1 Inserting garbage ................................................. 751
  - 50.2.2 Replacing instructions with bloated equivalents ........... 752
  - 50.2.3 Always executed/never executed code ......................... 752
  - 50.2.4 Making a lot of mess ............................................ 752
  - 50.2.5 Using indirect pointers ......................................... 753
- 50.3 Virtual machine / pseudo-code ..................................... 754
- 50.4 Other things to mention ............................................. 754
- 50.5 Exercise .............................................................. 754

## 51 C++

- 51.1 Classes ............................................................... 755
  - 51.1.1 A simple example ............................................... 755
  - 51.1.2 Class inheritance ............................................... 764
  - 51.1.3 Encapsulation .................................................... 768
  - 51.1.4 Multiple inheritance ............................................ 771
  - 51.1.5 Virtual methods ................................................ 776
- 51.2 ostream ............................................................... 780
- 51.3 References ........................................................... 781
- 51.4 STL ................................................................. 782
  - 51.4.1 std::string ...................................................... 782
  - 51.4.2 std::list .......................................................... 793
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>51.4.3 std::vector</td>
<td>808</td>
</tr>
<tr>
<td>51.4.4 std::map and std::set</td>
<td>818</td>
</tr>
<tr>
<td>52 Negative array indices</td>
<td>835</td>
</tr>
<tr>
<td>53 Windows 16-bit</td>
<td>839</td>
</tr>
<tr>
<td>53.1 Example #1</td>
<td>839</td>
</tr>
<tr>
<td>53.2 Example #2</td>
<td>840</td>
</tr>
<tr>
<td>53.3 Example #3</td>
<td>841</td>
</tr>
<tr>
<td>53.4 Example #4</td>
<td>843</td>
</tr>
<tr>
<td>53.5 Example #5</td>
<td>846</td>
</tr>
<tr>
<td>53.6 Example #6</td>
<td>851</td>
</tr>
<tr>
<td>53.6.1 Global variables</td>
<td>853</td>
</tr>
<tr>
<td>IV Java</td>
<td>856</td>
</tr>
<tr>
<td>54 Java</td>
<td>857</td>
</tr>
<tr>
<td>54.1 Introduction</td>
<td>857</td>
</tr>
<tr>
<td>54.2 Returning a value</td>
<td>858</td>
</tr>
<tr>
<td>54.3 Simple calculating functions</td>
<td>864</td>
</tr>
<tr>
<td>54.4 JVM(^6) memory model</td>
<td>867</td>
</tr>
<tr>
<td>54.5 Simple function calling</td>
<td>868</td>
</tr>
<tr>
<td>54.6 Calling beep()</td>
<td>871</td>
</tr>
<tr>
<td>54.7 Linear congruential PRNG(^7)</td>
<td>871</td>
</tr>
<tr>
<td>54.8 Conditional jumps</td>
<td>873</td>
</tr>
<tr>
<td>54.9 Passing arguments</td>
<td>876</td>
</tr>
<tr>
<td>54.10 Bitfields</td>
<td>878</td>
</tr>
<tr>
<td>54.11 Loops</td>
<td>879</td>
</tr>
<tr>
<td>54.12 switch()</td>
<td>882</td>
</tr>
<tr>
<td>54.13 Arrays</td>
<td>884</td>
</tr>
<tr>
<td>54.13.1 Simple example</td>
<td>884</td>
</tr>
<tr>
<td>54.13.2 Summing elements of array</td>
<td>885</td>
</tr>
<tr>
<td>54.13.3 The only argument of the main() function is an array too</td>
<td>886</td>
</tr>
<tr>
<td>54.13.4 Pre-initialized array of strings</td>
<td>887</td>
</tr>
<tr>
<td>54.13.5 Variadic functions</td>
<td>890</td>
</tr>
<tr>
<td>54.13.6 Two-dimensional arrays</td>
<td>893</td>
</tr>
<tr>
<td>54.13.7 Three-dimensional arrays</td>
<td>894</td>
</tr>
<tr>
<td>54.13.8 Summary</td>
<td>896</td>
</tr>
<tr>
<td>54.14 Strings</td>
<td>896</td>
</tr>
<tr>
<td>54.14.1 First example</td>
<td>896</td>
</tr>
<tr>
<td>54.14.2 Second example</td>
<td>897</td>
</tr>
</tbody>
</table>

\(^6\)Java virtual machine

\(^7\)Pseudorandom number generator
## V Finding important/interesting stuff in the code

### 55 Identification of executable files
4. Microsoft Visual C++
   - 55.1 Name mangling
4. GCC
   - 55.2 Name mangling
   - 55.3 Cygwin
   - 55.4 MinGW
4. Intel FORTRAN
4. Watcom, OpenWatcom
   - 55.4.1 Name mangling
4. Borland
   - 55.5 Delphi
4. Other known DLLs

### 56 Communication with the outer world (win32)
4. Often used functions in the Windows API
4. tracer: Intercepting all functions in specific module

### 57 Strings
4. Text strings
   - 57.1 C/C++
   - 57.2 Borland Delphi
   - 57.3 Unicode
   - 57.4 Base64
4. Error/debug messages
4. Suspicious magic strings

### 58 Calls to assert()

### 59 Constants
4. Magic numbers
   - 59.1 Dates
   - 59.2 DHCP
4. Searching for constants
## Contents

### Finding the right instructions

#### XOR instructions

#### Hand-written assembly code

### Suspicious code patterns

#### Using magic numbers while tracing

### Other things

#### General idea

#### C++

#### Some binary file patterns

#### Memory "snapshots" comparing

#### Windows registry

#### Blink-comparator

### OS-specific

#### Arguments passing methods (calling conventions)

#### cdecl

#### stdcall

#### fastcall

#### GCC regparm

#### Watcom/OpenWatcom

#### thiscall

#### x86-64

#### Windows x64

#### Linux x64

#### Return values of float and double type

#### Modifying arguments

#### Taking a pointer to function argument

### Thread Local Storage

#### Linear congruential generator revisited

#### Win32

#### Linux

### System calls (syscall-s)

#### Linux

#### Position-independent code
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIII</td>
<td>Examples of real-world RE tasks</td>
<td>1043</td>
</tr>
<tr>
<td>74</td>
<td>Task manager practical joke (Windows Vista)</td>
<td>1045</td>
</tr>
<tr>
<td>74.1</td>
<td>Using LEA to load values</td>
<td>1049</td>
</tr>
<tr>
<td>75</td>
<td>Color Lines game practical joke</td>
<td>1050</td>
</tr>
<tr>
<td>76</td>
<td>Minesweeper (Windows XP)</td>
<td>1054</td>
</tr>
<tr>
<td>76.1</td>
<td>Exercises</td>
<td>1061</td>
</tr>
<tr>
<td>77</td>
<td>Hand decompiling + Z3 SMT solver</td>
<td>1062</td>
</tr>
<tr>
<td>77.1</td>
<td>Hand decompiling</td>
<td>1062</td>
</tr>
<tr>
<td>77.2</td>
<td>Now let's use the Z3 SMT solver</td>
<td>1067</td>
</tr>
<tr>
<td>78</td>
<td>Dongles</td>
<td>1074</td>
</tr>
<tr>
<td>78.1</td>
<td>Example #1: MacOS Classic and PowerPC</td>
<td>1074</td>
</tr>
<tr>
<td>78.2</td>
<td>Example #2: SCO OpenServer</td>
<td>1085</td>
</tr>
<tr>
<td>78.2.1</td>
<td>Decrypting error messages</td>
<td>1097</td>
</tr>
<tr>
<td>78.3</td>
<td>Example #3: MS-DOS</td>
<td>1101</td>
</tr>
<tr>
<td>79</td>
<td>“QR9”: Rubik's cube inspired amateur crypto-algorithm</td>
<td>1109</td>
</tr>
<tr>
<td>80</td>
<td>SAP</td>
<td>1153</td>
</tr>
<tr>
<td>80.1</td>
<td>About SAP client network traffic compression</td>
<td>1153</td>
</tr>
<tr>
<td>80.2</td>
<td>SAP 6.0 password checking functions</td>
<td>1171</td>
</tr>
<tr>
<td>81</td>
<td>Oracle RDBMS</td>
<td>1177</td>
</tr>
<tr>
<td>81.1</td>
<td>V$VERSION table in the Oracle RDBMS</td>
<td>1177</td>
</tr>
<tr>
<td>81.2</td>
<td>X$KSMLRU table in Oracle RDBMS</td>
<td>1190</td>
</tr>
<tr>
<td>81.3</td>
<td>V$TIMER table in Oracle RDBMS</td>
<td>1192</td>
</tr>
<tr>
<td>82</td>
<td>Handwritten assembly code</td>
<td>1198</td>
</tr>
<tr>
<td>82.1</td>
<td>EICAR test file</td>
<td>1198</td>
</tr>
<tr>
<td>83</td>
<td>Demos</td>
<td>1200</td>
</tr>
<tr>
<td>83.1</td>
<td>10 PRINT CHR$(205.5+RND(1)); : GOTO 10</td>
<td>1200</td>
</tr>
<tr>
<td>83.1.1</td>
<td>Trixter's 42 byte version</td>
<td>1201</td>
</tr>
<tr>
<td>83.1.2</td>
<td>My attempt to reduce Trixter's version: 27 bytes</td>
<td>1202</td>
</tr>
<tr>
<td>83.1.3</td>
<td>Taking random memory garbage as a source of randomness</td>
<td>1203</td>
</tr>
<tr>
<td>83.1.4</td>
<td>Conclusion</td>
<td>1204</td>
</tr>
<tr>
<td>83.2</td>
<td>Mandelbrot set</td>
<td>1205</td>
</tr>
<tr>
<td>83.2.1</td>
<td>Theory</td>
<td>1206</td>
</tr>
<tr>
<td>83.2.2</td>
<td>Let's get back to the demo</td>
<td>1212</td>
</tr>
<tr>
<td>83.2.3</td>
<td>My “fixed” version</td>
<td>1215</td>
</tr>
<tr>
<td>IX</td>
<td>Examples of reversing proprietary file formats</td>
<td>1218</td>
</tr>
<tr>
<td>-----</td>
<td>---------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>84</td>
<td>Primitive XOR-encryption</td>
<td>1219</td>
</tr>
<tr>
<td>84.1</td>
<td>Norton Guide: simplest possible 1-byte XOR encryption</td>
<td>1220</td>
</tr>
<tr>
<td>84.1.1</td>
<td>Entropy</td>
<td>1221</td>
</tr>
<tr>
<td>84.2</td>
<td>Simplest possible 4-byte XOR encryption</td>
<td>1223</td>
</tr>
<tr>
<td>84.2.1</td>
<td>Exercise</td>
<td>1226</td>
</tr>
<tr>
<td>85</td>
<td>Millenium game save file</td>
<td>1227</td>
</tr>
<tr>
<td>86</td>
<td>Oracle RDBMS: .SYM-files</td>
<td>1234</td>
</tr>
<tr>
<td>87</td>
<td>Oracle RDBMS: .MSB-files</td>
<td>1247</td>
</tr>
<tr>
<td>87.1</td>
<td>Summary</td>
<td>1254</td>
</tr>
<tr>
<td>X</td>
<td>Other things</td>
<td>1255</td>
</tr>
<tr>
<td>88</td>
<td>npad</td>
<td>1256</td>
</tr>
<tr>
<td>89</td>
<td>Executable files patching</td>
<td>1259</td>
</tr>
<tr>
<td>89.1</td>
<td>Text strings</td>
<td>1259</td>
</tr>
<tr>
<td>89.2</td>
<td>x86 code</td>
<td>1259</td>
</tr>
<tr>
<td>90</td>
<td>Compiler intrinsic</td>
<td>1261</td>
</tr>
<tr>
<td>91</td>
<td>Compiler’s anomalies</td>
<td>1262</td>
</tr>
<tr>
<td>92</td>
<td>OpenMP</td>
<td>1264</td>
</tr>
<tr>
<td>92.1</td>
<td>MSVC</td>
<td>1267</td>
</tr>
<tr>
<td>92.2</td>
<td>GCC</td>
<td>1270</td>
</tr>
<tr>
<td>93</td>
<td>Itanium</td>
<td>1273</td>
</tr>
<tr>
<td>94</td>
<td>8086 memory model</td>
<td>1278</td>
</tr>
<tr>
<td>95</td>
<td>Basic blocks reordering</td>
<td>1280</td>
</tr>
<tr>
<td>95.1</td>
<td>Profile-guided optimization</td>
<td>1280</td>
</tr>
<tr>
<td>XI</td>
<td>Books/blogs worth reading</td>
<td>1283</td>
</tr>
<tr>
<td>96</td>
<td>Books</td>
<td>1284</td>
</tr>
<tr>
<td>96.1</td>
<td>Windows</td>
<td>1284</td>
</tr>
<tr>
<td>96.2</td>
<td>C/C++</td>
<td>1284</td>
</tr>
<tr>
<td>96.3</td>
<td>x86 / x86-64</td>
<td>1284</td>
</tr>
</tbody>
</table>
A.4.2 SSE and AVX registers .................................................. 1299
A.5 Debugging registers ...................................................... 1299
  A.5.1 DR6 ............................................................................ 1300
  A.5.2 DR7 ............................................................................ 1300
A.6 Instructions ................................................................. 1301
  A.6.1 Prefixes ..................................................................... 1301
  A.6.2 Most frequently used instructions .............................. 1302
  A.6.3 Less frequently used instructions .............................. 1309
  A.6.4 FPU instructions ....................................................... 1316
  A.6.5 Instructions having printable ASCII opcode ............. 1318

B ARM ................................................................................. 1321
  B.1 Terminology ............................................................... 1321
  B.2 Versions ....................................................................... 1321
  B.3 32-bit ARM (AArch32) .................................................. 1322
    B.3.1 General purpose registers ...................................... 1322
    B.3.2 Current Program Status Register (CPSR) ............... 1322
    B.3.3 VFP (floating point) and NEON registers ............... 1323
  B.4 64-bit ARM (AArch64) ................................................... 1323
    B.4.1 General purpose registers ..................................... 1323
  B.5 Instructions ............................................................... 1324
    B.5.1 Conditional codes table ......................................... 1325

C MIPS ................................................................................. 1326
  C.1 Registers ..................................................................... 1326
    C.1.1 General purpose registers GPR® ................................ 1326
    C.1.2 Floating-point registers ......................................... 1327
  C.2 Instructions .................................................................. 1327
    C.2.1 Jump instructions .................................................. 1327

D Some GCC library functions ............................................. 1329

E Some MSVC library functions .......................................... 1330

F Cheatsheets ...................................................................... 1331
  F.1 IDA .............................................................................. 1331
  F.2 OllyDbg ........................................................................ 1332
  F.3 MSVC .......................................................................... 1333
  F.4 GCC ............................................................................ 1333
  F.5 GDB .......................................................................... 1333

8 General Purpose Registers
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acronyms used</td>
<td>1336</td>
</tr>
<tr>
<td>Glossary</td>
<td>1342</td>
</tr>
<tr>
<td>Index</td>
<td>1345</td>
</tr>
<tr>
<td>Bibliography</td>
<td>1354</td>
</tr>
</tbody>
</table>
Preface

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

Topics discussed in-depth

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

Topics touched upon

Oracle RDBMS (81 on page 1177), Itanium (93 on page 1273), copy-protection dongles (78 on page 1074), LD_PRELOAD (67.2 on page 979), stack overflow, ELF\(^{10}\), win32 PE file format (68.2 on page 988), x86-64 (26.1 on page 580), critical sections (68.4 on page 1033), syscalls (66 on page 973), TLS\(^{11}\), position-independent code (PIC\(^{12}\)) (67.1 on page 975), profile-guided optimization (95.1 on page 1280), C++ STL (51.4 on page 782), OpenMP (92 on page 1264), SEH (68.3 on page 998).

Exercises and tasks

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

\(^9\)Database management systems
\(^{10}\)Executable file format widely used in *NIX systems including Linux
\(^{11}\)Thread Local Storage
\(^{12}\)Position Independent Code: 67.1 on page 975
About the author

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

Praise for Reverse Engineering for Beginners

• “It’s very well done .. and for free .. amazing.”\(^{13}\) Daniel Bilar, Siege Technologies, LLC.

• “... excellent and free”\(^{14}\) Pete Finnigan, Oracle RDBMS security guru.


• “... my compliments for the very nice tutorial!” Herbert Bos, full professor at the Vrije Universiteit Amsterdam, co-author of Modern Operating Systems (4th Edition).

• “... It is amazing and unbelievable.” Luis Rocha, CISSP / ISSAP, Technical Manager, Network & Information Security at Verizon Business.

• “Thanks for the great work and your book.” Joris van de Vis, SAP Netweaver & Security specialist.

• “... reasonable intro to some of the techniques.”\(^{15}\) Mike Stay, teacher at the Federal Law Enforcement Training Center, Georgia, US.

• “I love this book! I have several students reading it at the moment, plan to use it in graduate course.”\(^{16}\) Sergey Bratus, Research Assistant Professor at the Computer Science Department at Dartmouth College

\(^{13}\) twitter.com/daniel_bilar/status/436578617221742593

\(^{14}\) twitter.com/petefinnigan/status/400551705797869568

\(^{15}\) reddit

\(^{16}\) twitter.com/sergeybratus/status/505590326560833536
• “Dennis @Yurichev has published an impressive (and free!) book on reverse engineering”¹⁷ Tanel Poder, Oracle RDBMS performance tuning expert.

• “This book is some kind of Wikipedia to beginners...” Archer, Chinese Translator, IT Security Researcher.

Thanks

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

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

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

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.

For translating the book into Portuguese: Thales Stevan de A. Gois.


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

For illustrations and cover art: Andy Nechaevsky.

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

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

Donors

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

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50),

¹⁷ twitter.com/TanelPoder/status/524668104065159169
mini-FAQ

Q: Why should one learn assembly language these days?
A: Unless you are an OS\textsuperscript{18} developer, you probably don’t need to code in assembly—modern compilers are much better at performing optimizations than humans\textsuperscript{19}. Also, modern CPU\textsuperscript{20}s are very complex devices and assembly knowledge doesn’t really help one to understand their internals.

That being said, there are at least two areas where a good understanding of assembly can be helpful: First and foremost, security/malware research. It is also a good way to gain a better understanding of your compiled code whilst debugging. This book is therefore intended for those who want to understand assembly language rather than to code in it, which is why there are many examples of compiler output contained within.

Q: I clicked on a hyperlink inside a PDF-document, how do I go back?
A: In Adobe Acrobat Reader click Alt+LeftArrow.

Q: Your book is huge! Is there anything shorter?
A: A shortened (lite) version can be found here: http://beginners.re/#lite.

Q: I’m not sure if I should try to learn reverse engineering or not.
A: The average time to become familiar with the contents of the shortened LITE-version should be about 1-2 month(s). You may also try the reverse engineering challenges.

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

\textsuperscript{18}Operating System
\textsuperscript{19}A very good text about this topic: [Fog13b]
\textsuperscript{20}Central processing unit
A: Of course! That’s why the book is licensed under the Creative Commons license. If you also want to build your own version of the book—read here to find out more.

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

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

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

A: There are hiring threads that appear from time to time on reddit, devoted to RE\(^{21}\) (2013 Q3, 2014). Try looking there.

A somewhat related hiring thread can be found in the “netsec” subreddit: 2014 Q2.

Q: I have a question...

A: Send it to me by email (dennis(a)yurichev.com).

This is the A5-format version for e-book readers. Although the content is mostly the same, the illustrations are resized and probably not readable.

You may try to change scale in your e-book reader.

Otherwise, you can always view them in the A4-format version here: 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 my book (as it was in August 2014) into Korean.

It’s now available at their website.

The translator is Byungho Min (twitter/tais9).

The cover art was done by my artistic friend, Andy Nechaevsky: facebook/andy-dinka.

They also hold the copyright to the Korean translation.

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

---

\(^{21}\)reddit.com/r/ReverseEngineering/
Part I

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

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

**Exercises**

When the author of this book studied assembly language, he also often compiled small C-functions and then rewrote them gradually to assembly, trying to make their code as short as possible. This probably is not worth doing in real-world scenarios today, because it’s hard to compete with modern compilers in terms of efficiency. It is, however, a very good way to gain a better understanding of assembly. Feel free, therefore, to take any assembly code from this book and try to make it shorter. However, don’t forget to test what you have written.

**Optimization levels and debug information**

Source code can be compiled by different compilers with various optimization levels. A typical compiler has about three such levels, where level zero means disable optimization. Optimization can also be targeted towards code size or code speed. A non-optimizing compiler is faster and produces more understandable (albeit verbose) code, whereas an optimizing compiler is slower and tries to produce code that runs faster (but is not necessarily more compact). In addition to optimization levels and direction, a compiler can include in the resulting file some debug information, thus producing code for easy debugging. One of the important features of the ‘debug’ code is that it might contain links between each line of the source code and the respective machine code addresses. Optimizing compilers, on the other hand, tend to produce output where entire lines of source code can be optimized away and thus not even be present in the resulting machine code. Reverse engineers can encounter either version, simply because some developers turn on the

---

22In fact, he still does it when he can’t understand what a particular bit of code does.
compiler's optimization flags and others do not. Because of this, we'll try to work on examples of both debug and release versions of the code featured in this book, where possible.
Chapter 1

A short introduction to the CPU

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

A short glossary:

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

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

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

- **CPU register**: Each CPU has a fixed set of general purpose registers (GPR). ≈ 8 in x86, ≈ 16 in x86-64, ≈ 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 PL¹ and could only use eight 32-bit (or 64-bit) variables. Yet a lot can be done using just these!

One might wonder why there needs to be a difference between machine code and a PL. The answer lies in the fact that humans and CPUs are not alike—it is much easier for humans to use a high-level PL like C/C++, Java, Python, etc., but it is easier for a CPU to use a much lower level of abstraction. Perhaps it would be possible to invent a CPU that can execute high-level PL code, but it would be many times more complex than the CPUs we know of today. In a similar fashion, it is very inconvenient for humans to write in assembly language, due to it being so low-level and difficult to write in without making a huge number of annoying mistakes. The program that converts the high-level PL code into assembly is called a compiler.²

---

¹Programming language
²Old-school Russian literature also use term “translator”.

4
1.1 A couple of words about different ISAs

The x86 ISA has always been one with variable-length opcodes, so when the 64-bit era came, the x64 extensions did not impact the ISA very significantly. In fact, the x86 ISA still contains a lot of instructions that first appeared in 16-bit 8086 CPU, yet are still found in the CPUs of today. ARM is a RISC\(^3\) CPU designed with constant-length opcode in mind, which had some advantages in the past. In the very beginning, all ARM instructions were encoded in 4 bytes\(^4\). This is now referred to as “ARM mode”. Then they thought it wasn’t as frugal as they first imagined. In fact, most used CPU instructions\(^5\) in real world applications can be encoded using less information. They therefore added another ISA, called Thumb, where each instruction was encoded in just 2 bytes. This is now referred as “Thumb mode”. However, not all ARM instructions can be encoded in just 2 bytes, so the Thumb instruction set is somewhat limited. It is worth noting that code compiled for ARM mode and Thumb mode may of course coexist within one single program. The ARM creators thought Thumb could be extended, giving rise to Thumb-2, which appeared in ARMv7. Thumb-2 still uses 2-byte instructions, but has some new instructions which have the size of 4 bytes. There is a common misconception that Thumb-2 is a mix of ARM and Thumb. This is incorrect. Rather, Thumb-2 was extended to fully support all processor features so it could compete with ARM mode—a goal that was clearly achieved, as the majority of applications for iPod/iPhone/iPad are compiled for the Thumb-2 instruction set (admittedly, largely due to the fact that Xcode does this by default). Later the 64-bit ARM came out. This ISA has 4-byte opcodes, and lacked the need of any additional Thumb mode. However, the 64-bit requirements affected the ISA, resulting in us now having three ARM instruction sets: ARM mode, Thumb mode (including Thumb-2) and ARM64. These ISAs intersect partially, but it can be said that they are different ISAs, rather than variations of the same one. Therefore, we would try to add fragments of code in all three ARM ISAs in this book. There are, by the way, many other RISC ISAs with fixed length 32-bit opcodes, such as MIPS, PowerPC and Alpha AXP.

---

\(^3\)Reduced instruction set computing
\(^4\)By the way, fixed-length instructions are handy because one can calculate the next (or previous) instruction address without effort. This feature will be discussed in the switch() operator (13.2.2 on page 233) section.
\(^5\)These are MOV/PUSH/CALL/Jcc
Chapter 2

The simplest Function

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

Listing 2.1: C/C++ Code

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

Let's compile it!

2.1 x86

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

Listing 2.2: Optimizing GCC/MSVC (assembly output)

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

There are just two instructions: the first places the value 123 into the EAX register, which is used by convention for storing the return value and the second one is RET, which returns execution to the caller.

The caller will take the result from the EAX register.
There are a few differences on the ARM platform:

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

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

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

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

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

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

### 2.3 MIPS

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

The GCC assembly output below lists registers by number:

Listing 2.4: Optimizing GCC 4.4.5 (assembly output)

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

...while IDA\(^1\) does it—by their pseudonames:

Listing 2.5: Optimizing GCC 4.4.5 (IDA)

```assembly
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.

---

\(^1\) Interactive Disassembler and debugger developed by Hex-Rays
The other instruction is the jump instruction (J or JR) which returns the execution flow to the caller, jumping to the address in the $31 (or $RA) register.

This is the register analogous to LR\textsuperscript{2} in ARM.

You might be wondering why 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 just need to remember that in MIPS, the instruction following a jump or branch instruction is executed before the jump/branch instruction itself.

As a consequence, branch instructions always swap places with the instruction which must be executed beforehand.

2.3.1 A note about MIPS instruction/register names

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

\textsuperscript{2}Link Register
Chapter 3

Hello, world!

Let's use the famous example from the book “The C programming Language”[Ker88]:

```c
#include <stdio.h>

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

3.1 x86

3.1.1 MSVC

Let's compile it in MSVC 2010:

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

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

<table>
<thead>
<tr>
<th>Listing 3.1: MSVC 2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST SEGMENT</td>
</tr>
<tr>
<td>$SG3830 DB 'hello, world', 0AH, 00H</td>
</tr>
<tr>
<td>CONST ENDS</td>
</tr>
<tr>
<td>PUBLIC _main</td>
</tr>
<tr>
<td>EXTRN _printf:PROC</td>
</tr>
<tr>
<td>; Function compile flags: /Odtp</td>
</tr>
</tbody>
</table>
MSVC produces assembly listings in Intel-syntax. The difference between Intel-syntx and AT&T-syntx will be discussed in 3.1.3 on page 13.

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

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

That is why the example may be rewritten as follows:

```c
#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: 57.1.1 on page 926.

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)\(^1\).

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

\(^1\)You can read more about it in the section about function prologues and epilogues (4 on page 36).
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\(^2\).

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

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

```
Listing 3.2: Oracle RDBMS 10.2 Linux (app.o file)
.text:0800029A push ebx
.text:0800029B call qksfroChild
.text:080002A0 pop ecx
```

After calling `printf()`, the original C/C++ code contains the statement `return 0` —return 0 as the result of the `main()` function.

In the generated code this is implemented by the instruction `XOR EAX, EAX`. XOR is in fact just “eXclusive OR”\(^3\) but the compilers often use it instead of MOV EAX, 0—again because it is a slightly shorter opcode (2 bytes for XOR against 5 for MOV).

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

The last instruction RET returns the control to the caller. Usually, this is C/C++ CRT\(^4\) code, which, in turn, returns control to the OS.

### 3.1.2 GCC

Now let’s try to compile the same C/C++ code in the GCC 4.4.1 compiler in Linux: `gcc 1.c -o 1`. Next, with the assistance of the IDA disassembler, let’s see how

\(^2\)CPU flags, however, are modified

\(^3\)wikipedia

\(^4\)C runtime library : 68.1 on page 983
the main() function was created. IDA, like MSVC, uses Intel-syntax\(^5\).

Listing 3.3: code in IDA

```assembly
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 it is saved onto the stack. In addition, the function prologue contains AND ESP, 0FFFFFFFH — 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)\(^6\).

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

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

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

Then the printf() function is called.

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

The last instruction, LEAVE — is the equivalent of the MOV ESP, EBP and POP EBP instruction pair — in other words, this instruction sets the stack pointer (ESP) back and restores the EBP register to its initial state. This is necessary since we

\(^5\)We could also have GCC produce assembly listings in Intel-syntax by applying the options -S -masm=intel.

\(^6\)Wikipedia: Data structure alignment
modified these register values (ESP and EBP) at the beginning of the function (by executing `MOV EBP, ESP / AND ESP, ...`).

### 3.1.3 GCC: AT&T syntax

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

**Listing 3.4: let's compile in GCC 4.7.3**

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

We get this:

**Listing 3.5: GCC 4.7.3**

```assembly
.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_register 4
  ret
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
.section .note.GNU-stack,"
```
The listing contains many macros (beginning with dot). These are not interesting for us at the moment.

For now, for the sake of simplification, we can ignore them (except the \texttt{.string} macro which encodes a null-terminated character sequence just like a C-string). Then we'll see this:\textsuperscript{7}

egin{verbatim}
Listing 3.6: GCC 4.7.3

.LC0:
 .string "hello, world\n"

main:
    pushl %ebp
    movl %esp, %ebp
    andl $-16, %esp
    subl $16, %esp
    movl $.LC0, (%esp)
    call printf
    movl $0, %eax
    leave
    ret

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

\begin{itemize}
  \item Source and destination operands are written in opposite order.
    In Intel-syntax: \texttt{<instruction> <destination operand> <source operand>}.
    In AT&T syntax: \texttt{<instruction> <source operand> <destination operand>}.
    Here is an easy way to memorise the difference: when you deal with Intel-
    syntax, you can imagine that there is an equality sign (=) between operands
    and when you deal with AT&T-syntax imagine there is a right arrow (\rightarrow) \textsuperscript{8}.
  \item AT&T: Before register names, a percent sign must be written (%) and before
    numbers a dollar sign ($). Parentheses are used instead of brackets.
  \item AT&T: A suffix is added to instructions to define the operand size:
    \begin{itemize}
      \item q \textendash quad (64 bits)
      \item l \textendash long (32 bits)
      \item w \textendash word (16 bits)
      \item b \textendash byte (8 bits)
    \end{itemize}
\end{itemize}

\textsuperscript{7}This GCC option can be used to eliminate “unnecessary” macros: \texttt{-fno-asynchronous-unwind-tables}
\textsuperscript{8}By the way, in some C standard functions (e.g., \texttt{memcpy()}, \texttt{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.
Let’s go back to the compiled result: it is identical to what we saw in IDA. With one subtle difference: 0xFFFFFFFF0h is presented as $-16$. It is the same thing: 16 in the decimal system is 0x10 in hexadecimal. -0x10 is equal to 0xFFFFFFFF0 (for a 32-bit data type).

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

### 3.2 x86-64

#### 3.2.1 MSVC—x86-64

Let’s also try 64-bit MSVC:

```assembly
Listing 3.7: MSVC 2012 x64
$SG2989 DB 'hello, world', 0AH, 00H
main PROC
    sub    rsp, 40
    lea    rcx, OFFSET FLAT:$SG2989
    call   printf
    xor    eax, eax
    add    rsp, 40
    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 exists a popular way to pass function arguments via registers (*fastcall*) 64.3 on page 954. I.e., a part of the function arguments is passed in registers, the rest—via the stack. In Win64, 4 function arguments are passed in the RCX, RDX, R8, R9 registers. That is what we see here: a pointer to the string for `printf()` is now passed not in the stack, but in the RCX register. The pointers are 64-bit now, so they are passed in the 64-bit registers (which have the R-prefix). However, for backward compatibility, it is still possible to access the 32-bit parts, using the E-prefix. This is how the RAX/EAX/AX/AL register looks like in x86-64:

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX^x64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AL</td>
</tr>
</tbody>
</table>
The main() function returns an *int*-typed value, which is, in C/C++, for better backward compatibility and portability, still 32-bit, so that is why the EAX register is cleared at the function end (i.e., the 32-bit part of the register) instead of RAX. There are also 40 bytes allocated in the local stack. This is called the “shadow space”, about which we are going to talk later: *8.2.1 on page 137*.

### 3.2.2 GCC—x86-64

Let’s also try GCC in 64-bit Linux:

#### Listing 3.8: GCC 4.4.6 x64

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

A method to pass function arguments in registers is also used in Linux, *BSD and Mac OS X is* [Mit13].

The first 6 arguments are passed in the RDI, RSI, RDX, RCX, R8, R9 registers, and the rest—via the stack.

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

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

If we open the compiled object file (.o), we can also see all the instructions’ opcodes:

#### Listing 3.9: GCC 4.4.6 x64

```
.text:00000000004004D0 main    proc near
.text:00000000004004D0 48 83 EC 08     sub    rsp, 8
.text:00000000004004D4 BF E8 05 40 00    mov    edi, offset "hello, world\n"
.text:00000000004004D9 31 C0           xor    eax, eax
```

---

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

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

### 3.3 GCC—one more thing

The fact that an anonymous C-string has `const` type (3.1.1 on page 10), and that C-strings allocated in constants segment are guaranteed to be immutable, has an interesting consequence: the compiler may use a specific part of the string.

Let's try this example:

```c
#include <stdio.h>

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

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

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

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

Listing 3.10: GCC 4.8.1 + IDA listing
Indeed: when we print the “hello world” string these two words are positioned in memory adjacently and \puts()\ called from f2() function is not aware that this string is divided. In fact, it’s not divided; it’s divided only “virtually”, in this listing.

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

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

### 3.4 ARM

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

- Apple Xcode 4.6.3 IDE (with the LLVM-GCC 4.2 compiler\(^\text{10}\)).

\(^{10}\text{It is indeed so: Apple Xcode 4.6.3 uses open-source GCC as front-end compiler and LLVM code generator}\)
32-bit ARM code is used (including Thumb and Thumb-2 modes) in all cases in this book, if not mentioned otherwise. When we talk about 64-bit ARM here, we call it ARM64.

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

Let’s start by compiling our example in Keil:

```bash
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\(^{11}\), but it is more important for us to see the instructions “as is” so let’s see the compiled result in IDA.

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

```
.text:00000000 main
.text:00000000 10 40 2D E9 STMFD SP!, {R4,LR}
.text:00000004 1E 0E 8F E2 ADR R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB BL __2printf
.text:0000000C 00 00 A0 E3 MOV R0, #0
.text:00000010 10 80 BD E8 LDMFD SP!, {R4,PC}
.text:000000EC 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}`\(^{12}\), 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\(^{14}\) 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

---

\(^{11}\) e.g. ARM mode lacks PUSH/POP instructions

\(^{12}\) `STMFD`\(^{13}\)

\(^{13}\) `stack pointer`. SP/ESP/RSP in x86/x64. SP in ARM.

\(^{14}\) `stack pointer`. SP/ESP/RSP in x86/x64. SP in ARM.
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 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”. Such code can be 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 instruction calls the printf() function. Here's how this instruction works:

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

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

That is a difference between “pure” RISC-processors like ARM and CISC-processors like x86, where the return address is usually stored on the stack.

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

Next, the MOV R0, #0 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 is an inverse instruction of STMFD. It loads values from the stack (or any other memory place) in order to save them into

---

15 Program Counter. IP/EIP/RIP in x86/64. PC in ARM.
16 Read more about it in relevant section (67.1 on page 975)
17 Branch with Link
18 Complex instruction set computing
19 Read more about this in next section (5 on page 38)
20 MOVE
21 LDMFD
CHAPTER 3. HELLO, WORLD!

R4 and PC, and increments the stack pointer SP. It works like POP here.

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

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

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

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

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

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

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

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

We are getting (in IDA):

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

```
.text:00000000 main
.text:00000000 10 B5 PUSH {R4,LR}
.text:00000002 C0 A0 ADR R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9 BL __2printf
.text:00000008 00 20 MOVS R0, #0
.text:0000000A 10 BD POP {R4,PC}
.text:00000304 68 65 6C 6C+aHelloWorld DCB "hello, world",0 ; DATA XREF: main+2
```

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

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. MOVVS writes 0 into the R0 register in order to return zero.

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

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

| __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\} |

The instructions STMFD and LDMFD are already familiar to us.

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

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

The MOVT R0, #0 (MOVe Top) instruction writes 0 into higher 16 bits of the register. The issue here is that the generic MOV instruction in ARM mode may write only the lower 16 bits of the register. Remember, all instruction opcodes in ARM mode are limited in size to 32 bits. Of course, this limitation is not related to moving data between registers. That’s why an additional instruction MOVT exists for writing into the higher bits (from 16 to 31 inclusive). Its usage here, however, is redundant because the MOV R0, #0x1686 instruction above cleared the higher part of the register. This is probably a shortcoming of the compiler.
The ADD R0, PC, R0 instruction adds the value in the PC to the value in the R0, to calculate the absolute address of the “Hello world!” string. As we already know, it is “position-independent code” so this correction is essential here.

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

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

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

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

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

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

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

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

__cstring:00003E70  48 65 6C 6C 6F 20+aHelloWorld DB "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

\textsuperscript{23}It has also to be noted the puts() does not require a \textbackslash n new line symbol at the end of a string, so we do not see it here.

\textsuperscript{24}ciselant.de/projects/gcc_printf/gcc_printf.html
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\(^{25}\) 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):

\[
\text{__symbolstub1:00003FEC _puts ; CODE XREF: } \nearrow \text{_hello_world+E} \\
\text{__symbolstub1:00003FEC 44 F0 9F E5 LDR PC, =__imp__puts}
\]

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

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

Because it is not very space-efficient.

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

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

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

\(^{25}\)Return Address
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
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)

3.4.5 ARM64

GCC

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

```
Listing 3.15: Non-optimizing GCC 4.8.1 + objdump

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
  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: B.4.1 on page 1323. 64-bit registers have X- prefixes, while its 32-bit parts—W-.

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

The exclamation mark after the operand means that 16 is to be subtracted from SP first, and only then are values from register pair to be written into the stack. This is also called pre-index. About the difference between post-index and pre-index read here: 28.2 on page 607.
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\textsuperscript{26} in ARM64, and X30 as LR, so that’s why they are saved in the function prologue and restored in the function epilogue.

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

ADRP and ADD instructions are used to fill the address of the string “Hello!” into the X0 register, because the first function argument is passed in this register. There are no instructions, whatsoever, in ARM that can store a large number into a register (because the instruction length is limited to 4 bytes, read more about it here: \texttt{28.3.1} on page 608). So several instructions must be utilised. The first instruction (ADRP) writes the address of the 4KiB page, where the string is located, into X0, and the second one (ADD) just adds the remainder to the address. More about that in: \texttt{28.4} on page 610.

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

\texttt{puts()} is called afterwards using the BL instruction. This was already discussed: \texttt{3.4.3} on page 23.

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

<table>
<thead>
<tr>
<th>High 32-bit part</th>
<th>low 32-bit part</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0</td>
<td>W0</td>
</tr>
</tbody>
</table>

The function result is returned via X0 and \texttt{main()} returns 0, so that’s how the return result is prepared. But why use the 32-bit part? Because the \texttt{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 \texttt{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 \texttt{main()} returns a 64-bit value:

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

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

uint64\_t main()
{
    printf ("Hello!\n");
    return 0;
}
```

\textsuperscript{26}Frame Pointer
The result is the same, but that's how MOV at that line looks like now:

### Listing 3.17: Non-optimizing GCC 4.8.1 + objdump

<table>
<thead>
<tr>
<th>4005a4:</th>
<th>d2800000 mov x0, #0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>// #0</td>
</tr>
</tbody>
</table>

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

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

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

## 3.5 MIPS

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

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

It’s possible, however, to load data from the address in the range of $register - 32768...register + 32767$ 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”) 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 on page 1278 or the MS-DOS memory managers like XMS/EMS where all memory was divided in 64KiB blocks.

---

27 Block Started by Symbol
This concept is not unique to MIPS. At least PowerPC uses this technique as well.

### 3.5.2 Optimizing GCC

Let’s consider the following example, which illustrates the “global pointer” concept.

#### Listing 3.18: 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
```

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

---

28 The MIPS registers table is available in appendix C.1 on page 1326.
JALR ("Jump and Link Register") jumps to the address stored in the $25 register (address of `puts()` while saving the address of the next instruction (LW) in RA. This is very similar to ARM. Oh, and one important thing is that the address saved in RA is not the address of the next instruction (because it’s in a delay slot and is executed before the jump instruction), but the address of the instruction after the next one (after the delay slot). Hence, $PC+8$ is written to RA during the execution of JALR, in our case, this is the address of the LW instruction next to ADDIU.

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

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

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

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

<table>
<thead>
<tr>
<th>.text:00000000 main:</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:00000000 var_10 = -0x10</td>
</tr>
<tr>
<td>.text:00000000 var_4 = -4</td>
</tr>
<tr>
<td>.text:00000000</td>
</tr>
<tr>
<td>; function prologue.</td>
</tr>
<tr>
<td>; set the GP:</td>
</tr>
<tr>
<td>.text:00000000 lui $gp, (__gnu_local_gp &gt;&gt; 16)</td>
</tr>
<tr>
<td>.text:00000004 addiu $sp, -0x20</td>
</tr>
<tr>
<td>.text:00000008 la $gp, (__gnu_local_gp &amp; 0xFFFF)</td>
</tr>
<tr>
<td>; save the RA to the local stack:</td>
</tr>
<tr>
<td>.text:000000C sw $ra, 0x20+var_4($sp)</td>
</tr>
<tr>
<td>; save the GP to the local stack:</td>
</tr>
<tr>
<td>; for some reason, this instruction is missing in the GCC assembly output:</td>
</tr>
</tbody>
</table>

29 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. 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.

### 3.5.3 Non-optimizing GCC

Non-optimizing GCC is more verbose.

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

```assembly
$LCO:
    .ascii "Hello, world!\012\000"
main:
    ; function prologue.
    ; save the RA ($31) and FP in the stack:
    addiu $sp,$sp,-32
    sw $31,28($sp)
    sw $fp,24($sp)
    ; set the FP (stack frame pointer):
```

---

30 Apparently, functions generating listings are not so critical to GCC users, so some unfixed errors 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 3.21: Non-optimizing GCC 4.4.5 (IDA)
Interestingly, IDA recognized the LUI/ADDIU instructions pair and coalesced them into one LA (“Load Address”) pseudoinstruction at line 15. We may also see that this pseudoinstruction has a size of 8 bytes! This is a pseudoinstruction (or macro) because it’s not a real MIPS instruction, but rather a handy name for an instruction pair.
Another thing is that IDA doesn’t recognize NOP instructions, so here they are at lines 22, 26 and 41. It is OR $AT, $ZERO. Essentially, this instruction applies the OR operation to the contents of the $AT register with zero, which is, of course, an idle instruction. MIPS, like many other ISAs, doesn’t have a separate NOP instruction.

3.5.4 Role of the stack frame in this example

The address of the text string is passed in the register. Why setup a local stack anyway? The reason for this lies in the fact that the values of registers RA and GP have to be saved somewhere (because printf() is called), and the local stack is used for this purpose. If this was a leaf function, it would have been possible to get rid of the function prologue and epilogue, for example: 2.3 on page 7.

3.5.5 Optimizing GCC: load it into GDB

Listing 3.22: sample GDB session

```
root@debian-mips:~# gcc hw.c -O3 -o hw
root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "mips-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/hw...(no debugging symbols found)... done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw
Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:  lui gp,0x42
```
3.6 Conclusion

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

3.7 Exercises

- http://challenges.re/48
- http://challenges.re/49
Chapter 4

Function prologue and epilogue

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

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

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

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

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

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

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

4.1 Recursion

Epilogues and prologues can negatively affect the recursion performance.
More about recursion in this book: 36.3 on page 646.
Chapter 5

Stack

The stack is one of the most fundamental data structures in computer science\(^1\). 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\(^2\)/LDMED\(^3\) instructions are intended to deal with a descending stack (grows downwards, starting with a high address and progressing to a lower one). The STMFA\(^4\)/LDMFA\(^5\), STMEA\(^6\)/LDMEA\(^7\) instructions

\(^1\) wikipedia.org/wiki/Call_stack
\(^2\) Store Multiple Empty Descending (ARM instruction)
\(^3\) Load Multiple Empty Descending (ARM instruction)
\(^4\) Store Multiple Full Ascending (ARM instruction)
\(^5\) Load Multiple Full Ascending (ARM instruction)
\(^6\) Store Multiple Empty Ascending (ARM instruction)
\(^7\) Load Multiple Empty Ascending (ARM instruction)
are intended to deal with an ascending stack (grows upwards, starting from a low address and progressing to a higher one).

## 5.1 Why does the stack grow backwards?

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

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

![Diagram of heap and stack growth](image)

In [RT74] we can read:

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

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

5.2.1 Save the function’s return address

x86

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

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

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

Overflowing the stack is straightforward. Just run eternal recursion:

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

MSVC 2008 reports the problem:

```
c:\tmp6>c\ 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
msvc6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, function will cause runtime stack overflow
```

...but generates the right code anyway:

```
?f@@YAXXZ PROC ; f
 ; File c:\tmp6\ss.cpp
 ; Line 2
     push    ebp
     mov     ebp, esp
 ; Line 3
     call    ?f@@YAXXZ ; f
 ; Line 4
     pop      ebp
     ret      0
?f@@YAXXZ ENDP ; f
```
...Also if we turn on the compiler optimization (/Ox option) the optimized code will not overflow the stack and will work correctly\(^8\) instead:

```c
?f@@YAXXZ PROC ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@if:
; Line 3
    jmp SHORT $LL3@if
?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!” (3.4 on page 18), 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*\(^9\). 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\(^10\). This can be also useful for situations when memory for the stack is not yet allocated or not available.

Some examples of leaf functions: 8.3.2 on page 141, 8.3.3 on page 141, 19.17 on page 425, 19.33 on page 449, 19.5.4 on page 450, 15.4 on page 285, 15.2 on page 283, 17.3 on page 308.

### 5.2.2 Passing function arguments

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

\(^8\) irony here
\(^10\) Some time ago, on PDP-11 and VAX, the CALL instruction (calling other functions) was expensive; up to 50% of execution time might be spent on it, so it was considered that having a big number of small functions is an anti-pattern [Ray03, Chapter 4, Part II].
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 \texttt{f()} function's very first instruction:

<table>
<thead>
<tr>
<th>ESP</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP+4</td>
<td>argument#1, marked in IDA as \texttt{arg_0}</td>
</tr>
<tr>
<td>ESP+8</td>
<td>argument#2, marked in IDA as \texttt{arg_4}</td>
</tr>
<tr>
<td>ESP+0xC</td>
<td>argument#3, marked in IDA as \texttt{arg_8}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

For more information on other calling conventions see also section (64 on page 952). It is worth noting that nothing obliges programmers to pass arguments through the stack. It is not a requirement. One could implement any other method without using the stack at all.

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

By the way, the callee function does not have any information about how many arguments were passed. C functions with a variable number of arguments (like \texttt{printf()}) determine their number using format string specifiers (which begin with the \% symbol).

If we write something like:

\begin{verbatim}
printf("%d %d %d", 1234);
\end{verbatim}

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

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

In fact, the CRT-code is calling \texttt{main()} roughly as:

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

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

### 5.2.3 Local variable storage

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

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

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

It is worth noting the `alloca()` function. This function works like `malloc()`, but allocates memory directly on the stack.

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

It is worth noting how `alloca()` is implemented. In simple terms, this function just shifts ESP downwards toward the stack bottom by the number of bytes you need and sets ESP as a pointer to the allocated block.

Let’s try:

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

12 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`
void f()
{
    char *buf=(char*)alloca (600);
    #ifdef __GNUC__
        snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
    #else
        _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
    #endif

        puts (buf);
} ;

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

**MSVC**

Let's compile (MSVC 2010):

```
Listing 5.1: MSVC 2010

```
```c
...
    mov    eax, 600  ; 00000258H
    call   __alloca_probe_16
    mov    esi, esp

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

    push   esi
    call   _puts
    add    esp, 28  ; 0000001cH
...
```
The sole alloca() argument is passed via EAX (instead of pushing it into the stack)\(^{13}\). After the alloca() call, ESP points to the block of 600 bytes and we can use it as memory for the buf array.

**GCC + Intel syntax**

GCC 4.4.1 does the same without calling external functions:

<table>
<thead>
<tr>
<th>Listing 5.2: GCC 4.7.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LC0:</td>
</tr>
<tr>
<td>.string &quot;hi! %d, %d, %d\n&quot;</td>
</tr>
<tr>
<td>f:</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>push ebx</td>
</tr>
<tr>
<td>sub esp, 660</td>
</tr>
<tr>
<td>lea ebx, [esp+39]</td>
</tr>
<tr>
<td>and ebx, -16          ; align</td>
</tr>
<tr>
<td>mov DWORD PTR [esp], ebx ; s</td>
</tr>
<tr>
<td>mov DWORD PTR [esp+20], 3</td>
</tr>
<tr>
<td>mov DWORD PTR [esp+16], 2</td>
</tr>
<tr>
<td>mov DWORD PTR [esp+12], 1</td>
</tr>
<tr>
<td>mov DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; &quot;hi! %d, %d, %d\n&quot;</td>
</tr>
<tr>
<td>mov DWORD PTR [esp+4], 600 ; maxlen</td>
</tr>
<tr>
<td>call _snprintf</td>
</tr>
<tr>
<td>mov DWORD PTR [esp], ebx ; s</td>
</tr>
<tr>
<td>call puts</td>
</tr>
<tr>
<td>mov ebx, DWORD PTR [ebp-4]</td>
</tr>
<tr>
<td>leave</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>

**GCC + AT&T syntax**

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

<table>
<thead>
<tr>
<th>Listing 5.3: GCC 4.7.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LC0:</td>
</tr>
</tbody>
</table>

\(^{13}\)It is because alloca() is rather a compiler intrinsic (90 on page 1261) 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\(^{14}\) alloca() implementation also has code which reads from the memory just allocated, in order to let the OS map physical memory to this VM\(^{15}\) region.
The code is the same as in the previous listing.

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

### 5.2.5 (Windows) SEH

SEH\(^{16}\) records are also stored on the stack (if they are present).

Read more about it: (68.3 on page 998).

### 5.2.6 Buffer overflow protection

More about it here (18.2 on page 370).

### 5.2.7 Automatic deallocation of data in stack

Perhaps, the reason for storing local variables and SEH records in the stack is that they are freed automatically upon function exit, using just one instruction to correct the stack pointer (it is often `ADD`). Function arguments, as we could say, are also

\(^{16}\)Structured Exception Handling: 68.3 on page 998
deallocated automatically at the end of function. In contrast, everything stored in the heap must be deallocated explicitly.

### 5.3 A typical stack layout

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP-0xC</td>
<td>local variable #2, marked in IDA as var_8</td>
</tr>
<tr>
<td>ESP-8</td>
<td>local variable #1, marked in IDA as var_4</td>
</tr>
<tr>
<td>ESP-4</td>
<td>saved value of EBP</td>
</tr>
<tr>
<td>ESP</td>
<td>return address</td>
</tr>
<tr>
<td>ESP+4</td>
<td>argument#1, marked in IDA as arg_0</td>
</tr>
<tr>
<td>ESP+8</td>
<td>argument#2, marked in IDA as arg_4</td>
</tr>
<tr>
<td>ESP+0xC</td>
<td>argument#3, marked in IDA as arg_8</td>
</tr>
</tbody>
</table>

### 5.4 Noise in stack

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

```c
#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();
};
```
Listing 5.4: Non-optimizing MSVC 2010

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

_c$ = -12 ; size = 4
_b$ = -8 ; size = 4
_a$ = -4 ; size = 4
_f1 PROC
    push ebp
    mov ebp, esp
    sub esp, 12
    mov DWORD PTR _a$[ebp], 1
    mov DWORD PTR _b$[ebp], 2
    mov DWORD PTR _c$[ebp], 3
    mov esp, ebp
    pop ebp
    ret 0
_f1 ENDP

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

_main PROC
    push ebp
    mov ebp, esp
    call _f1
    call _f2
    xor eax, eax
_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
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:

![OllyDbg](image)

Figure 5.1: OllyDbg: f1()

When f1() assigns the variables $a$, $b$ and $c$, their values are stored at the address 0x1FF860 and so on.
And when \( f_2() \) executes:

![OllyDbg: f2()](image)

... \( a, b \) and \( c \) of \( f_2() \) are located at the same addresses! No one has overwritten the values yet, so at that point they are still untouched.

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

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

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

### 5.4.1 MSVC 2013

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

```plaintext
3, 2, 1
```

Why?

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

### 5.5 Exercises

- [http://challenges.re/51](http://challenges.re/51)
- [http://challenges.re/52](http://challenges.re/52)
Now let's extend the *Hello, world!* ([3 on page 9]) example, replacing `printf()` in the `main()` function body with this:

```c
#include <stdio.h>

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

### 6.1 x86

#### 6.1.1 x86: 3 arguments

**MSVC**

When we compile it with MSVC 2010 Express we get:

```assembly
$SG3830 DB 'a=%d; b=%d; c=%d', 00H
...
push 3
```
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. $4 \times 4 = 16$ — they occupy exactly 16 bytes in the stack: a 32-bit pointer to a string and 3 numbers of type int.

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

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

See also the calling conventions section (64 on page 952).

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:

Here is a real-world example:

```
Listing 6.1: x86

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

Click on the PUSH EBP instruction, press F2 (set breakpoint) and press F9 (run). We need to perform these actions in order to skip CRT-code, because we aren't really interested in it yet.
Press F8 (step over) 6 times, i.e. skip 6 instructions:

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:

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

It is possible to right-click on the format string, click on “Follow in dump”, and the format string will appear in the debugger left-bottom window, which always displays some part of the memory. These memory values can be edited. It is possible to change the format string, in which case the result of our example would be different. It is not very useful in this particular case, but it could be good as an
exercice so you start building a feel of how everything works here.
Press F8 (step over).

We see the following output in the console:

![Figure 6.4: printf() function executed](image)

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

![Figure 6.5: OllyDbg after printf() execution](image)

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

A very important fact is that neither the ESP value, nor the stack state have been changed! We clearly see that the format string and corresponding 3 values are still there. This is indeed the cdecl calling convention behaviour: callee does not return ESP back to its previous value. The caller is responsible to do so.
CHAPTER 6. PRINTF() WITH SEVERAL ARGUMENTS

Press F8 again to execute ADD ESP, 10 instruction:

![Image of OllyDbg window showing after ADD ESP, 10 instruction execution]

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

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

GCC

Now let’s compile the same program in Linux using GCC 4.4.1 and take a look at what we have got in IDA:

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

push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 10h
mov eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
mov [esp+10h+var_4], 3
mov [esp+10h+var_8], 2
mov [esp+10h+var_C], 1
mov [esp+10h+var_10], eax
call _printf
```
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\(^1\) in Linux.

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

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

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

**Listing 6.2: let’s set breakpoint on printf()**

```bash
(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.

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

\(^1\)GNU debugger
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
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. 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,0xfffffffff0
  0x08048423 <+6>: sub esp,0x10
  0x08048426 <+9>: mov DWORD PTR [esp+0xc],0x3
```
Execute next instruction. **GDB** shows ending bracket, meaning, it ends the block.

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

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

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

### 6.1.2 x64: 8 arguments

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

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
}
```
As it was mentioned earlier, the first 4 arguments has to be passed through the 
RCX, RDX, R8, R9 registers in Win64, while all the rest—via the stack. That is
exactly what we see here. However, the MOV instruction, instead of PUSH, is used
for preparing the stack, so the values are stored to the stack in a straightforward
manner.

Listing 6.3: MSVC 2012 x64

```assembly
$SG2923 DB 'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n\n', 0aH, 00H
main PROC
  sub  rsp, 88
  mov  DWORD PTR [rsp+64], 8
  mov  DWORD PTR [rsp+56], 7
  mov  DWORD PTR [rsp+48], 6
  mov  DWORD PTR [rsp+40], 5
  mov  DWORD PTR [rsp+32], 4
  mov  r9d, 3
  mov  r8d, 2
  mov  edx, 1
  lea  rcx, OFFSET FLAT:$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 remember: 8 bytes are allocated for any data type
shorter than 64 bits. This is established for the convenience's sake: it makes it
easy to calculate the address of arbitrary argument. Besides, they are all located
at aligned memory addresses. It is the same in the 32-bit environments: 4 bytes
are reserved for all data types.
GCC

The picture is similar for x86-64 *NIX OS-es, except that the first 6 arguments are passed through the RDI, RSI, RDX, RCX, R8, R9 registers. All the rest—via the stack. GCC generates the code storing the string pointer into EDI instead of RDI—we noted that previously: 3.2.2 on page 17.

We also noted earlier that the EAX register has been cleared before a printf() call: 3.2.2 on page 17.

Listing 6.4: Optimizing GCC 4.4.6 x64

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

main:
    sub rdp, 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 [rdp+16], 8
    mov DWORD PTR [rdp+8], 7
    mov DWORD PTR [rdp], 6
    call printf

    ; return 0
    xor eax, eax
    add rdp, 40
    ret
```

GCC + GDB

Let’s try this example in GDB.

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

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

(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2
Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d;
; e=%d; f=%d; g=%d; h=%d\n") at printf.c:29
29   printf.c: No such file or directory.

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

(gdb) info registers
rax     0x0 0
rbx     0x0 0
rcx     0x3 3
rdx     0x2 2
rsi     0x1 1
rdi     0x400628 4195880
rbp     0x7fffffffdf60 0x7fffffffdf60
rsp     0x7fffffffdf38 0x7fffffffdf38
r8      0x4 4
r9      0x5 5
r10     0x7fffffffdfce0 140737488346336
r11     0x7fffffff7a65f60 140737348263776
r12     0x400440 4195880
r13     0x7fffffffde040 140737488347200
r14     0x0 0
r15     0x0 0
rip     0x7fffffff7a65f60 0x7fffffff7a65f60 __printf>

Listing 6.6: let's inspect the format string
CHAPTER 6. PRINTF() WITH SEVERAL ARGUMENTS

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

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: 0x0000000000000008. That's OK, because the values have int type, which is 32-bit. So, the high register or stack element part may contain “random garbage”.

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

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.
6.2 ARM

6.2.1 ARM: 3 arguments

ARM’s traditional scheme for passing arguments (calling convention) behaves as follows: the first 4 arguments are passed through the R0-R3 registers; the remaining arguments via the stack. This resembles the arguments passing scheme infastcall (64.3 on page 954) or win64 (64.5.1 on page 956).

32-bit ARM

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

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

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

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

There is nothing unusual so far.

Optimizing Keil 6/2013 generates the same code.

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

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

```
.text:00000000 main
.text:00000000 10 B5 PUSH {R4,LR}
.text:00000002 03 23 MOVS R3, #3
.text:00000004 02 22 MOVS R2, #2
.text:00000006 01 21 MOVS R1, #1
.text:00000008 02 A0 ADR R0, aADBDCD ; "a=%d; b=%d; c=%d"
.text:00000010 00 F0 0D F8 BL __2printf
.text:00000014 00 20 MOVS R0, #0 ; return 0
.text:00000018 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`:

```c
#include <stdio.h>

void main()
```
The result is somewhat unusual:

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

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

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

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

A similar example is presented here: 13.1.1 on page 212.

**ARM64**

**Non-optimizing GCC (Linaro) 4.9**

Listing 6.10: Non-optimizing GCC (Linaro) 4.9

```
.LC1:
```
The first instruction STP (Store Pair) saves FP (X29) and LR (X30) in the stack. The second ADD X29, SP, 0 instruction forms the stack frame. It is just writing the value of SP into X29.

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

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

Optimizing GCC (Linaro) 4.9 generates the same code.

### 6.2.2 ARM: 8 arguments

Let's use again the example with 9 arguments from the previous section: 6.1.2 on page 64.

```c
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
};
```
Optimizing Keil 6/2013: ARM mode

This code can be divided into several parts:

- **Function prologue:**

  The very first `STR LR, [SP,#var_4]` instruction saves `LR` on the stack, because we are going to use this register for the `printf()` call. Exclamation mark at the end indicates pre-index. This implies that `SP` is to be decreased by 4 first, and then `LR` will be saved at the address stored in `SP`. This is similar to `PUSH` in x86. Read more about it at: 28.2 on page 607.

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

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

- Passing 4 via the stack: 4 is stored in R0 and then this value, with the help of the STR R0, [SP,#0x18+var_18] instruction is saved on the stack. var_18 is -0x18, so the offset is to be 0, thus the value from the R0 register (4) is to be written to the address written in SP.

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

- printf() call.

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

  The LDR PC, [SP+4+var_4],#4 instruction loads the saved LR value from the stack into the PC register, thus causing the function to exit. There is no exclamation mark—indeed, PC is loaded first from the address stored in SP (4 + 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\(^2\). 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\(^3\).

---

**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
```

\(^2\) Read more about it: 28.2 on page 607.
\(^3\) It is impossible to set IP/EIP/RIP value using POP in x86, but anyway, you got the analogy right.
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 00 00 40 E3 MOVT R0, #0
__text:00002920 04 20 A0 E3 MOV R2, #4
__text:00002924 00 00 8F E0 ADD R0, PC, R0
__text:00002928 06 30 A0 E3 MOV R3, #6
__text:0000292C 05 10 A0 E3 MOV R1, #5
__text:00002930 00 20 8D E5 STR R2, [SP,#0x1C+var_1C]
__text:00002934 0A 10 8D E9 STMFA SP, {R1,R3,R12}
__text:00002938 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
```
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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R3, #3</td>
<td>__text:00002948 03 30 A0 E3</td>
<td></td>
</tr>
<tr>
<td>STR R9, [SP,#0x1C+var_C]</td>
<td>__text:0000294C 10 90 8D E5</td>
<td></td>
</tr>
<tr>
<td>BL __printf</td>
<td>__text:00002950 A4 05 00 EB</td>
<td></td>
</tr>
<tr>
<td>MOV SP, R7</td>
<td>__text:00002954 07 D0 A0 E1</td>
<td></td>
</tr>
<tr>
<td>LDMFD SP!, {R7,PC}</td>
<td>__text:00002958 80 80 BD E8</td>
<td></td>
</tr>
</tbody>
</table>

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

ARM64

Non-optimizing GCC (Linaro) 4.9

Listing 6.11: Non-optimizing GCC (Linaro) 4.9

```
.LC2:
    .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d
    \n"
f3:
; grab more space in stack:
    sub sp, sp, #32
; save FP and LR in stack frame:
    stp x29, x30, [sp,16]
; set stack frame (FP=SP):
    add x29, sp, 16
    adrp x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
    add x0, x0, :lo12:.LC2
    mov w1, 8 ; 9th argument
    str w1, [sp] ; store 9th argument in the
 ; stack
    mov w1, 1
    mov w2, 2
    mov w3, 3
    mov w4, 4
    mov w5, 5
    mov w6, 6
    mov w7, 7
    bl printf
    sub sp, x29, #16
; restore FP and LR
    ldp x29, x30, [sp,16]
```
The first 8 arguments are passed in X- or W-registers: [ARM13c]. A string pointer requires a 64-bit register, so it's passed in X0. All other values have a int 32-bit type, so they are stored in the 32-bit part of the registers (W-). The 9th argument (8) is passed via the stack. Indeed: it's not possible to pass large number of arguments through registers, because the number of registers is limited.

Optimizing GCC (Linaro) 4.9 generates the same code.

## 6.3 MIPS

### 6.3.1 3 arguments

**Optimizing GCC 4.4.5**

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

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

**Listing 6.12: Optimizing GCC 4.4.5 (assembly output)**

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

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

Listing 6.13: Optimizing GCC 4.4.5 (IDA)

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

; function prologue:
.text:00000000 lui $gp, (__gnu_local_gp >> 16)
.text:00000004 addiu $sp, -0x20
.text:00000008 la $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C sw $ra, 0x20+var_4($sp)
.text:00000010 sw $gp, 0x20+var_10($sp)
; load address of printf():
.text:00000014 lw $t9, (printf & 0xFFFF)($gp)
; load address of the text string and set 1st argument of printf():
.text:00000018 la $a0, $LC0    # "a=%d\n\t\t; b=%d; c=%d"
; set 2nd argument of printf():
.text:00000020 li $a1, 1
; set 3rd argument of printf():
.text:00000024 li $a2, 2
; call printf():
.text:00000028 jalr $t9
; set 4th argument of printf() (branch delay slot):
.text:0000002C li $a3, 3
; function epilogue:
.text:00000030 lw $ra, 0x20+var_4($sp)
; set return value to 0:
.text:00000034 move $v0, $zero
; return
.text:00000038 jr $ra
IDA has coalesced pair of LUI and ADDIU instructions into one LA pseudoinstruction. That's why there are no instruction at address 0x1C: because LA occupies 8 bytes.

**Non-optimizing GCC 4.4.5**

Non-optimizing GCC is more verbose:

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

.text:0000003C addiu $sp, 0x20 ; branch delay

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

main:
; function prologue:
 addiu $sp,$sp,-32
 sw $31,28($sp)
 sw $fp,24($sp)
 move $fp,$sp
 lui $28,%hi(__gnu_local_gp)
 addiu $28,$28,%lo(__gnu_local_gp)

; load address of the text string:
 lui $2,%hi($LC0)
 addiu $2,$2,%lo($LC0)

; set 1st argument of printf():
 move $4,$2

; set 2nd argument of printf():
 li $5,1 # 0x1

; set 3rd argument of printf():
 li $6,2 # 0x2

; set 4th argument of printf():
 li $7,3 # 0x3

; get address of printf():
 lw $2,%call16(printf)($28)
 nop

; call printf():
 move $25,$2
 jalr $25
 nop

; function epilogue:
 lw $28,16($fp)

; set return value to 0:
 move $2,$0
 move $sp,$fp
```
Listing 6.15: Non-optimizing GCC 4.4.5 (IDA)

```
.main:
.addiu $sp, -0x20
.sw $ra, 0x20+var_4($sp)
.sw $fp, 0x20+var_8($sp)
.move $fp, $sp
.la $gp, __gnu_local_gp
.sw $gp, 0x20+var_10($sp)

.text:0000001C  la $v0, aADBDCD # "a=%d\n \n b=%d; c=%d"

.set 1st argument of printf():
.move $a0, $v0

.set 2nd argument of printf():
.li $a1, 1

.set 3rd argument of printf():
.li $a2, 2

.set 4th argument of printf():
.li $a3, 3

.get address of printf():
.lw $v0, (printf & 0xFFFF)($gp)
.or $at, $zero

.call printf():
.move $t9, $v0
.jalr $t9
.or $at, $zero ; NOP

.function epilogue:
.lw $gp, 0x20+var_10($fp)

.set return value to 0:
.move $v0, $zero
.move $sp, $fp
.lw $ra, 0x20+var_4($sp)
.lw $fp, 0x20+var_8($sp)
```
6.3.2 8 arguments

Let’s use again the example with 9 arguments from the previous section: 6.1.2 on page 64.

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

Optimizing GCC 4.4.5

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

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

Listing 6.16: Optimizing GCC 4.4.5 (assembly output)

```
$LCO:
   .ascii "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n\n\012\000"
main:
; function prelogue:
   lui      $28,%hi(__gnu_local_gp)
   addiu    $sp,$sp,-56
   addiu    $28,$28,%lo(__gnu_local_gp)
   sw       $31,52($sp)
; pass 5th argument in stack:
   li       $2,4              # 0x4
   sw       $2,16($sp)
; pass 6th argument in stack:
```
li $2,5 # 0x5
sw $2,20($sp)

; pass 7th argument in stack:
li $2,6 # 0x6
sw $2,24($sp)

; pass 8th argument in stack:
li $2,7 # 0x7
lw $25,%call16(printf($28)
sw $2,28($sp)

; pass 1st argument in $a0:
 lui $4,%hi($LC0)

; pass 9th argument in stack:
li $2,8 # 0x8
sw $2,32($sp)
addiu $4,$4,%lo($LC0)

; pass 2nd argument in $a1:
li $5,1 # 0x1

; pass 3rd argument in $a2:
li $6,2 # 0x2

call printf():
jalr $25

; pass 4th argument in $a3 (branch delay slot):
li $7,3 # 0x3

; function epilogue:
lw $31,52($sp)

; set return value to 0:
mov $2,$0

; return
j $31
addiu $sp,$sp,56 ; branch delay slot

Listing 6.17: Optimizing GCC 4.4.5 (IDA)
.text:00000008  la $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C  sw $ra, 0x38+var_4($sp)
.text:00000010  sw $gp, 0x38+var_10($sp)
; pass 5th argument in stack:
.text:00000014  li $v0, 4
.text:00000018  sw $v0, 0x38+var_28($sp)
; pass 6th argument in stack:
.text:0000001C  li $v0, 5
.text:00000020  sw $v0, 0x38+var_24($sp)
; pass 7th argument in stack:
.text:00000024  li $v0, 6
.text:00000028  sw $v0, 0x38+var_20($sp)
; pass 8th argument in stack:
.text:0000002C  li $v0, 7
.text:00000030  lw $t9, (printf & 0xFFFF)($gp)
.text:00000034  sw $v0, 0x38+var_1C($sp)
; prepare 1st argument in $a0:
.text:00000038  lui $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%"
; pass 9th argument in stack:
.text:0000003C  li $v0, 8
.text:00000040  sw $v0, 0x38+var_18($sp)
; pass 1st argument in $a1:
.text:00000044  la $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%"
; pass 2nd argument in $a1:
.text:00000048  li $a1, 1
; pass 3rd argument in $a2:
.text:0000004C  li $a2, 2
; call printf():
.text:00000050  jalr $t9
; pass 4th argument in $a3 (branch delay slot):
.text:00000054  li $a3, 3
; function epilogue:
.text:00000058  lw $ra, 0x38+var_4($sp)
; set return value to 0:
.text:0000005C  move $v0, $zero
; return
.text:00000060  jr $ra
.text:00000064  addiu $sp, 0x38 ; branch delay
Non-optimizing GCC is more verbose:

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

```assembly
$LCO:
    .ascii "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n\012\000"
main:
    ; function prologue:
    addiu $sp,$sp,-56
    sw $31,52($sp)
    sw $fp,48($sp)
    move $fp,$sp
    lui $28,%hi(__gnu_local_gp)
    addiu $28,$28,%lo(__gnu_local_gp)
    lui $2,%hi($LCO)
    addiu $2,$2,%lo($LCO)
    ; pass 5th argument in stack:
    li $3,4  # 0x4
    sw $3,16($sp)
    ; pass 6th argument in stack:
    li $3,5  # 0x5
    sw $3,20($sp)
    ; pass 7th argument in stack:
    li $3,6  # 0x6
    sw $3,24($sp)
    ; pass 8th argument in stack:
    li $3,7  # 0x7
    sw $3,28($sp)
    ; pass 9th argument in stack:
    li $3,8  # 0x8
    sw $3,32($sp)
    ; pass 1st argument in $a0:
    move $4,$2
    ; pass 2nd argument in $a1:
    li $5,1  # 0x1
    ; pass 3rd argument in $a2:
    li $6,2  # 0x2
    ; pass 4th argument in $a3:
    li $7,3  # 0x3
    ; call printf():
    lw $2,%call16(printf)($28)
    nop
    move $25,$2
    jalr $25
    nop
```

; function epilogue:
    lw     $28,40($fp)
; set return value to 0:
    move   $2,$0
    move   $sp,$fp
    lw     $31,52($sp)
    lw     $fp,48($sp)
    addiu  $sp,$sp,56
; return
    j      $31
    nop

Listing 6.19: Non-optimizing GCC 4.4.5 (IDA)
6.4 Conclusion

Here is a rough skeleton of the function call:

Listing 6.20: x86

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

Listing 6.21: x64 (MSVC)

MOV RCX, 1st argument  
MOV RDX, 2nd argument
Listing 6.22: x64 (GCC)

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

Listing 6.23: ARM

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

Listing 6.24: ARM64

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

Listing 6.25: MIPS (O32 calling convention)

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

6.5 By the way

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

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

The CPU is not aware of calling conventions whatsoever. 

We may also recall how newcoming assembly language programmers passing ar-
guments into other functions: usually via registers, without any explicit order, or 
even via global variables. Of course, it works fine.
Chapter 7

scanf()

Now let's use scanf().

7.1 Simple example

```c
#include <stdio.h>

int main()
{
    int x;
    printf("Enter X:\n");
    scanf("%d", &x);
    printf("You entered %d...\n", x);
    return 0;
}
```

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

7.1.1 About pointers

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

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

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

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

Pointers are also widely used when a function needs to return more than one value (we are going to get back to this later (10 on page 152)).

`scanf()` function—is such a case.

Besides the fact that the function needs to indicate how many values were successfully read, it also needs to return all these values.

In C/C++ the pointer type is only needed for compile-time type checking.

Internally, in the compiled code there is no information about pointer types at all.

### 7.1.2 x86

**MSVC**

Here is what we get after compiling with MSVC 2010:

```
CONST SEGMENT
$SG3831 DB 'Enter X:', 0aH, 00H
$SG3832 DB '%d', 00H
$SG3833 DB 'You entered %d...', 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
```
Chapter 7. Scanf()

_x$ = -4 ; size = 4

_main PROC
    push ebp
    mov ebp, esp
    push ecx
    push OFFSET $SG3831 ; 'Enter X:'
    call _printf
    add esp, 4
    lea eax, DWORD PTR _x$[ebp]
    push eax
    push OFFSET $SG3832 ; '%d'
    call _scanf
    add esp, 8
    mov ecx, DWORD PTR _x$[ebp]
    push ecx
    push OFFSET $SG3833 ; 'You entered %d...'
    call _printf
    add esp, 8
    ; return 0
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
_TEXT ENDS

x is a local variable.

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

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

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

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

Over the span of the function’s execution, EBP is pointing to the current stack frame making it possible to access local variables and function arguments via EBP+offset.

It is also possible to use ESP for the same purpose, although that is not very convenient since it changes frequently. The value of the EBP could be perceived as a frozen state of the value in ESP at the start of the function’s execution.
Here is a typical stack frame layout in 32-bit environment:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP-8</td>
<td>local variable #2, marked in IDA as var_8</td>
</tr>
<tr>
<td>EBP-4</td>
<td>local variable #1, marked in IDA as var_4</td>
</tr>
<tr>
<td>EBP</td>
<td>saved value of EBP</td>
</tr>
<tr>
<td>EBP+4</td>
<td>return address</td>
</tr>
<tr>
<td>EBP+8</td>
<td>argument#1, marked in IDA as arg_0</td>
</tr>
<tr>
<td>EBP+0xC</td>
<td>argument#2, marked in IDA as arg_4</td>
</tr>
<tr>
<td>EBP+0x10</td>
<td>argument#3, marked in IDA as arg_8</td>
</tr>
</tbody>
</table>

The `scanf()` function in our example has two arguments. The first one is a pointer to the string containing `%d` and the second is the address of the `x` variable.

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

`LEA` stands for load effective address, and is often used for forming an address ([A.6.2 on page 1304]).

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

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

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

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

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

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

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

Right-click the EAX in the registers window and then select “Follow in stack”. This address will appear in the stack window. The red arrow has been added, pointing to the variable in the local stack. At that moment this location contains some garbage (0x6E494714). Now with the help of PUSH instruction the address of this stack element is going to be stored to the same stack on the next position. Let’s trace with F8 until the scanf() execution completes. During the scanf() execution, we input, for example, 123, in the console window:
**Figure 7.3: OllyDbg: `scanf()` executed**

`scanf()` returns 1 in EAX, which implies that it has read successfully one value. If we look again at the stack element corresponding to the local variable it now contains 0x7B (123).
Later this value is copied from the stack to the ECX register and passed to \printf():

![Figure 7.4: OllyDbg: preparing the value for passing to printf()](image)

**GCC**

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

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

push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 20h
mov [esp+20h+var_20], offset aEnterX ; "⤦ Enter X:"

\call _puts
\mov eax, offset aD ; "%d"
\lea edx, [esp+20h+var_4]
\mov \[esp+20h+var_1C], edx
\mov \[esp+20h+var_20], eax
\call ___isoc99_scanf
\mov edx, [esp+20h+var_4]
\mov eax, offset aYouEnteredD___ ; "You ▸ entered %d...\n"

\mov [esp+20h+var_1C], edx
\mov [esp+20h+var_20], eax
\call _printf
\mov eax, 0

96
```
GCC replaced the printf() call with call to puts(). The reason for this was explained in (3.4.3 on page 23).

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

By the way

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

### 7.1.4 x64

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

**MSVC**

Listing 7.1: MSVC 2012 x64

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

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

_EXIT SEGMENT
EXIT ENDS

END main

```

GCC

Listing 7.2: Optimizing GCC 4.4.6 x64

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

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

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

7.1.5 ARM

Optimizing Keil 6/2013 (Thumb mode)
In order for `scanf()` to be able to read it needs a parameter—pointer to an `int`. `int` is 32-bit, so we need 4 bytes to store it somewhere in memory, and it fits exactly in a 32-bit register. A place for the local variable `x` is allocated in the stack and IDA has named it `var_8`. It is not necessary, however, to allocate a such since `SP` (stack pointer) is already pointing to that space and it can be used directly. So, `SP`’s value is copied to the R1 register and, together with the format-string, passed to `scanf()`. Later, with the help of the `LDR` instruction, this value is moved from the stack to the R1 register in order to be passed to `printf()`.

**ARM64**

Listing 7.3: Non-optimizing GCC 4.9.1 ARM64

```
.LC0:
.string "Enter X:"
.LC1:
.string "%d"
.LC2:
.string "You entered %d...
"
scanf_main:
; subtract 32 from SP, then save FP and LR in stack frame:
  stp x29, x30, [sp, -32]!
; set stack frame (FP=SP)
  add x29, sp, 0
; load pointer to the "Enter X:" string:
```
There is 32 bytes are allocated for stack frame, which is bigger than it needed. Perhaps, some memory aligning issue? The most interesting part is finding space for the $x$ variable in the stack frame (line 22). Why 28? Somehow, compiler decided to place this variable at the end of stack frame instead of beginning. The address is passed to scanf(), which just stores the user input value in the memory at that address. This is 32-bit value of type int. The value is fetched at line 27 and then passed to printf().

7.1.6 MIPS

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

Listing 7.4: Optimizing GCC 4.4.5 (assembly output)

```assembly
$LC0:
.asci "Enter X:\000"
```
Chapter 7: scanf()

$LC1:
    .ascii "%d\000"

$LC2:
    .ascii "You entered %d...\012\000"

main:
    ; function prologue:
    lui $28,%hi(__gnu_local_gp)
    addiu $sp,$sp,-40
    addiu $28,$28,%lo(__gnu_local_gp)
    sw $31,36($sp)
    ; call puts():
    lw $25,%call16(puts)($28)
    lui $4,%hi($LC0)
    jalr $25
    addiu $4,$4,%lo($LC0) ; branch delay slot
    ; call scanf():
    lw $28,16($sp)
    lui $4,%hi($LC1)
    lw $25,%call16(__isoc99_scanf)($28)
    ; set 2nd argument of scanf(), $a1=$sp+24:
    addiu $5,$sp,24
    jalr $25
    addiu $4,$4,%lo($LC1) ; branch delay slot
    ; call printf():
    lw $28,16($sp)
    ; set 2nd argument of printf(),
    ; load word at address $sp+24:
    lw $5,24($sp)
    lw $25,%call16(printf)($28)
    lui $4,%hi($LC2)
    jalr $25
    addiu $4,$4,%lo($LC2) ; branch delay slot
    ; function epilogue:
    lw $31,36($sp)
    ; set return value to 0:
    move $2,$0
    ; return:
    j $31
    addiu $sp,$sp,40 ; branch delay slot

IDA displays the stack layout as follows:

Listing 7.5: Optimizing GCC 4.4.5 (IDA)

.text:00000000 main:
.text:00000000
CHAPTER 7. SCANF() CHAPTER 7. SCANF()

```
.text:00000000 var_18 = -0x18
.text:00000000 var_10 = -0x10
.text:00000000 var_4 = -4
.text:00000000
; function prologue:
.text:00000000 lui $gp, (__gnu_local_gp >> 16)
.text:00000004 addiu $sp, -0x28
.text:00000008 la $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C sw $ra, 0x28+var_4($sp)
.text:00000010 sw $gp, 0x28+var_18($sp)
; call puts():
.text:00000014 lw $t9, (puts & 0xFFFF)($gp)
.text:00000018 lui $a0, ($LC0 >> 16) # "Enter X:"
.text:0000001C jalr $t9
.text:00000020 la $a0, ($LC0 & 0xFFFF) # "Enter X:" ; branch delay slot
; call scanf():
.text:00000024 lw $gp, 0x28+var_18($sp)
.text:00000028 lui $a0, ($LC1 >> 16) # "%d"
.text:0000002C lw $t9, (__isoc99_scanf & 0xFFFF)($gp)
; set 2nd argument of scanf(), $a1=$sp+24:
.text:00000030 addiu $a1, $sp, 0x28+var_10
.text:00000034 jalr $t9 ; branch delay slot
.text:00000038 la $a0, ($LC1 & 0xFFFF) # "%d" ; branch delay slot
; call printf():
.text:0000003C lw $gp, 0x28+var_18($sp)
; set 2nd argument of printf(), load word at address $sp+24:
.text:00000040 lw $a1, 0x28+var_10($sp)
.text:00000044 lw $t9, (printf & 0xFFFF)($gp)
.text:00000048 lui $a0, ($LC2 >> 16) # "You entered %d...
.text:0000004C jalr $t9
.text:00000050 la $a0, ($LC2 & 0xFFFF) # "You entered %d...
; branch delay slot
; function epilogue:
.text:00000054 lw $ra, 0x28+var_4($sp)
; set return value to 0:
.text:00000058 move $v0, $zero
```

102
7.2 Global variables

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

```c
#include <stdio.h>

// now x is global variable
int x;

int main()
{
    printf("Enter X:\n");
    scanf("%d", &x);
    printf("You entered %d...\n", x);
    return 0;
}
```

7.2.1 MSVC: x86

```
_DATA SEGMENT
COMM _x:DWORD
$SG2456   DB 'Enter X:', 0aH, 00H
$SG2457   DB '%d', 00H
$SG2458   DB 'You entered %d...', 0aH, 00H
_DATA ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_main PROC
```
In this case the x variable is defined in the _DATA segment and no memory is allocated in the local stack. It is accessed directly, not through the stack. Uninitialized global variables take no space in the executable file (indeed, why one needs to allocate space for variables initially set to zero?), but when someone accesses their address, the OS will allocate a block of zeroes there\(^1\).

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

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

We got:

```c
_DATA SEGMENT
_x DD 0AH
...
```

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

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

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

\(^1\)That is how a VM behaves
_x is marked with ? with the rest of the variables that do not need to be initialized. This implies that after loading the .exe to the memory, a space for all these variables is to be allocated and filled with zeroes [ISO07, 6.7.8p10]. But in the .exe file these uninitialized variables do not occupy anything. This is convenient for large arrays, for example.
7.2.2 MSVC: x86 + OllyDbg

Things are even simpler here:

![OllyDbg screenshot](image.png)

Figure 7.5: OllyDbg: after `scanf()` execution

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

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

But why is the first byte 7B? Thinking logically, `00 00 00 7B` should be there. The cause for this is referred as endianness, and x86 uses little-endian. This implies that the lowest byte is written first, and the highest written last. Read more about it at: 31 on page 619.

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

The memory address of `x` is `0x00C53394`. 
In OllyDbg we can review the process memory map (Alt-M) and we can see that this address is inside the .data PE-segment of our program:

![OllyDbg: process memory map](image)

Figure 7.6: OllyDbg: process memory map

### 7.2.3 GCC: x86

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

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

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

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

### 7.2.4 MSVC: x64

Listing 7.6: MSVC 2012 x64

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

### 7.2.5 ARM: Optimizing Keil 6/2013 (Thumb mode)
So, the x variable is now global and for this reason located in another segment, namely the data segment (.data). One could ask, why are the text strings located in the code segment (.text) and x is located right here? Because it is a variable and by definition its value could change. Moreover it could possibly change often. While text strings has constant type, they will not be changed, so they are located in the .text segment. The code segment might sometimes be located in a ROM\(^2\) chip (remember, we now deal with embedded microelectronics, and memory scarcity is

\(^2\)Read-only memory
common here), and changeable variables—in RAM\(^3\). It is not very economical to store constant variables in RAM when you have ROM. Furthermore, constant variables in RAM must be initialized, because after powering on, the RAM, obviously, contains random information.

Moving forward, we see a pointer to the \(x\) (off\_2C) variable in the code segment, and that all operations with the variable occur via this pointer. That is because the \(x\) variable could be located somewhere far from this particular code fragment, so its address must be saved somewhere in close proximity to the code. The LDR instruction in Thumb mode can only address variables in a range of 1020 bytes from its location, and in ARM-mode—variables in range of ±4095 bytes. And so the address of the \(x\) variable must be located somewhere in close proximity, because there is no guarantee that the linker would be able to accommodate the variable somewhere nearby the code, it may well be even in an external memory chip!

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

### 7.2.6 ARM64

Listing 7.7: Non-optimizing GCC 4.9.1 ARM64

```assembly
    .comm  x,4,4

.LC0:
    .string "Enter X:"

.LC1:
    .string "%d"

.LC2:
    .string "You entered %d...\n"

f5:

; save FP and LR in stack frame:
    stp   x29, x30, [sp, -16]!

; set stack frame (FP=SP)
    add   x29, sp, 0

; load pointer to the "Enter X:" string:
    adrp  x0, .LC0
    add   x0, x0, :lo12:.LC0
    bl    puts

; load pointer to the "%d" string:
    adrp  x0, .LC1
    add   x0, x0, :lo12:.LC1

; form address of \(x\) global variable:
```

\(^3\)Random-access memory
In this case the $x$ variable is declared as global and its address is calculated using the ADRP/ADD instruction pair (lines 21 and 25).

### 7.2.7 MIPS

**Uninitialized global variable**

So now the $x$ variable is global. Let’s compile to executable file rather than object file and load it into IDA. IDA displays the $x$ variable in the .sbss ELF section (remember the “Global Pointer”? 3.5.1 on page 28), since the variable is not initialized at the start.

**Listing 7.8: Optimizing GCC 4.4.5 (IDA)**

```
.text:004006C0 main:
.text:004006C0 var_10 = -0x10
.text:004006C0 var_4 = -4
.text:004006C0
; function prologue:
.text:004006C0 lui $gp, 0x42
.text:004006C4 addiu $sp, -0x20
.text:004006C8 li $gp, 0x418940
.text:004006CC sw $ra, 0x20+var_4($sp)
.text:004006D0 sw $gp, 0x20+var_10($sp)
; call puts():
.text:004006D4 la $t9, puts
.text:004006D8 lui $a0, 0x40
```
CHAPTER 7. SCANF()  

.text:004006DC  jalr $t9 ; puts
.text:004006E0  la $a0, aEnterX # "\nEnter X:"; branch delay slot
; call scanf():
.text:004006E4  lw $gp, 0x20+var_10($sp)
.text:004006E8  lui $a0, 0x40
.text:004006EC  la $t9, __isoc99_scanf
; prepare address of x:
.text:004006F0  la $a1, x
.text:004006F4  jalr $t9 ; __isoc99_scanf
.text:004006F8  la $a0, aD # "%d"
; branch delay slot
; call printf():
.text:004006FC  lw $gp, 0x20+var_10($sp)
.text:00400700  lui $a0, 0x40
; get address of x:
.text:00400704  la $v0, x
.text:00400708  la $t9, printf
; load value from "x" variable and pass it to printf() in $a1:
.text:0040070C  lw $a1, (x - 0x41099C)($v0)
.text:00400710  jalr $t9 ; printf
.text:00400714  la $a0, aYouEnteredD___ # "You entered %d...\n"; branch delay slot
; function epilogue:
.text:00400718  lw $ra, 0x20+var_4($sp)
.text:0040071C  move $v0, $zero
.text:00400720  jr $ra
.text:00400724  addiu $sp, 0x20 ; branch

... 

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

IDA reduces the amount of information, so we'll also do a listing using objdump and comment it:

Listing 7.9: Optimizing GCC 4.4.5 (objdump)

```
004006c0 <main>:
  ; function prologue:
  4006c0:  3c1c0042  lui gp,0x42
  4006c4:  27bdffe0  addiu sp,sp,-32
  4006c8:  279c8940  addiu gp,gp,-30400
```
Now we see the $x$ variable address is read from a 64KiB data buffer using GP and adding negative offset to it (line 18). More than that, the addresses of the three external functions which are used in our example (printf(), scanf(), printf()), are also read from the 64KiB global data buffer using GP (lines 9, 16 and 26). GP points to the middle of the buffer, and such offset suggests that all three function's addresses, and also the address of the $x$ variable, are all stored somewhere at the beginning of that buffer. That make sense, because our example is tiny.
Another thing worth mentioning is that the function ends with two NOPs (MOVE $AT, $AT — an idle instruction), in order to align next function’s start on 16-byte boundary.

**Initialized global variable**

Let’s alter our example by giving the \textit{x} variable a default value:

\begin{verbatim}
int x=10; // default value
\end{verbatim}

Now IDA shows that the \textit{x} variable is residing in the .data section:

```
Listing 7.10: Optimizing GCC 4.4.5 (IDA)

.text:004006A0 main:
.text:004006A0 var_10 = -0x10
.text:004006A0 var_8 = -8
.text:004006A0 var_4 = -4
.text:004006A0
.text:004006A0 lui $gp, 0x42
.text:004006A4 addiu $sp, -0x20
.text:004006A8 li $gp, 0x418930
.text:004006AC sw $ra, 0x20+var_4($sp)
.text:004006B0 sw $s0, 0x20+var_8($sp)
.text:004006B4 sw $gp, 0x20+var_10($sp)
.text:004006B8 lui $a0, 0x40
.text:004006BC la $t9, __isoc99_scanf
.text:004006D0 addiu $a1, $s0, (x - 0x410000)
.text:004006DC jalr $t9 ; __isoc99_scanf
.text:004006E0 la $a0, aD # "%d"
.text:004006E4 lw $gp, 0x20+var_10($sp)
.text:004006EC la $t9, printf
.text:004006F0 lui $a0, 0x40
```
Why not .sdata? Perhaps that this depends on some GCC option? Nevertheless, now $x$ is in .data, which is a general memory area, and we can take a look how to work with variables there.

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

Here is also the objdump listing for close inspection:

Listing 7.11: Optimizing GCC 4.4.5 (objdump)

004006a0 <main>:
  4006a0:  3c1c0042  lui   gp,0x42
  4006a4:  27bdffe0  addiu  sp,sp,-32
  4006a8:  279c8930  addiu  gp,gp,-30416
  4006ac:  afbf001c  sw    ra,28(sp)
  4006b0:  afbc0010  sw    gp,16(sp)
  4006b4:  8f998034  lw     t9,-32716(gp)
  4006bc:  3c040040  lui   a0,0x40
  4006c0:  0320f809  jalr  t9
  4006c4:  248408d0  addiu  a0,a0,2256
  4006c8:  8fbc0010  lw    gp,16(sp)
; prepare high part of x address:
  4006cc:  3c100041  lui   s0,0x41
  4006d0:  8f998038  lw    t9,-32712(gp)
  4006d4:  3c040040  lui   a0,0x40
; add low part of x address:
  4006d8:  26050920  addiu  a1,s0,2336
; now address of x is in $a1.
  4006dc:  0320f809  jalr  t9
  4006e0:  248408dc  addiu  a0,a0,2268
  4006e4:  8fbc0010  lw    gp,16(sp)
; high part of x address is still in $s0.
We see that the address is formed using LUI and ADDIU, but the high part of address is still in the $S0 register, and it is possible to encode the offset in a LW ("Load Word") instruction, so one single LW is enough to load a value from the variable and pass it to printf().

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

### 7.3 scanf() result checking

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

```c
#include <stdio.h>

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

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

    return 0;
}
```
By standard, the `scanf()` function returns the number of fields it has successfully read.

In our case, if everything goes fine and the user enters a number `scanf()` returns 1, or in case of error (or `EOF`) — 0.

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

This works as expected:

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

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

### 7.3.1 MSVC: x86

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

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

$LN2@main:
push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
call    _printf
add     esp, 4

$LN1@main:
oxor    eax, eax
```
The caller function (main()) needs the callee function (scanf()) result, so the callee returns it in the EAX register.

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

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

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

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

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

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

7.3.2 MSVC: x86: IDA

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

6 x86 flags, see also: wikipedia.
While analysing code in IDA, it is very helpful to leave notes for oneself (and others). In instance, analysing this example, we see that JNZ is to be triggered in case of an error. So it is possible to move the cursor to the label, press "n" and rename it to "error". Create another label—into “exit”. Here is my result:

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

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

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

Let’s hide two blocks and give them names:

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

To expand previously collapsed parts of the code, use “+” on the numerical pad.
By pressing "space", we can see how IDA represents a function as a graph:

```assembly
; int cdecl main()
_main proc near

var_4 dwld ptr -4
argc dwld ptr 8
argv dwld ptr 0Ch
enop dwld ptr 0h

push ebp
mov ebp, esp
push ecx
push offset format
.call dprintf
add esp, 4
lea eax, [ebp+var_4]
push eax
push offset a0
.call dscanf
add esp, 8
cmp eax, 1
jnz short error

mov ecx, [ebp+var_4]
push ecx
push offset aYou
.call dprintf
add esp, 8
jmp short exit

exit:
xor eax, eax
mov esp, ebp
pop ebp
ret

_main endp
```

Figure 7.7: Graph mode in IDA

There are two arrows after each conditional jump: green and red. The green arrow points to the block which executes if the jump is triggered, and red if otherwise.
It is possible to fold nodes in this mode and give them names as well ("group nodes"). Let’s do it for 3 blocks:

```c
; int cdecl main()
_main proc near

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

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

Figure 7.8: Graph mode in IDA with 3 nodes folded

That is very useful. It could be said that a very important part of the reverse engineers’ job (and any other researcher as well) is to reduce the amount of information they deal with.
7.3.3 MSVC: x86 + OllyDbg

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

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

![Figure 7.9: OllyDbg: passing variable address into scanf()](image)

```
Figure 7.9: OllyDbg: passing variable address into scanf()
```
While `scanf()` executes, in the console we enter something that is definitely not a number, like “asdasd”. `scanf()` finishes with 0 in EAX, which indicates that an error has occurred:

![Figure 7.10: OllyDbg: scanf() returning error](image)

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

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

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

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

![Figure 7.11: console window](image)

Indeed, 1850296084 is a decimal representation of the number in the stack (0x6E494714)!
This can also be used as a simple example of executable file patching. We may try to patch the executable so the program would always print the input, no matter what we enter.

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

We can see this:

![Hiew: main() function](image)

Hiew finds ASCII\(^8\) strings and displays them, as it does with the imported functions' names.

---

\(^7\) that's what also called “dynamic linking”

\(^8\) ASCII Zero (null-terminated ASCII string)
Move the cursor to address `.00401027` (where the JNZ instruction, we have to bypass, is located), press F3, and then type “9090” (meaning two NOPs):

![Hiew: replacing JNZ with two NOPs](image)

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

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

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

### 7.3.5 MSVC: x64

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

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

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

```
var_8 = -8
```
The new instructions here are CMP and BEQ\(^9\).

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

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

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

**ARM64**

Listing 7.14: Non-optimizing GCC 4.9.1 ARM64

```assembly
PUSH {R3,LR}
ADR R0, aEnterX ; "Enter X:\n"
BL __2printf
MOV R1, SP
ADR R0, aD ; "%d"
BL __0scanf
CMP R0, #1
BEQ loc_1E
ADR R0, aWhatYouEntered ; "What you entered
⤦ ? Huh?\n"
BL __2printf
loc_1A ; CODE XREF: main+26
MOVS R0, #0
POP {R3,PC}
loc_1E ; CODE XREF: main+12
LDR R1, [SP,#8+var_8]
ADR R0, aYouEnteredD___ ; "You entered %d
⤦ ...\n"
BL __2printf
B loc_1A
```

\(^9\)(PowerPC, ARM) Branch if Equal
Code flow in this case forks with the use of CMP/BNE (Branch if Not Equal) instructions pair.
Listing 7.15: Optimizing GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18 = -0x18
.text:004006A0 var_10 = -0x10
.text:004006A0 var_4 = -4
.text:004006A0
.text:004006A0 lui $gp, 0x42
.text:004006A4 addiu $sp, -0x28
.text:004006A8 li $gp, 0x418960
.text:004006AC sw $ra, 0x28+var_4($sp)
.text:004006B0 sw $gp, 0x28+var_18($sp)
.text:004006B4 la $t9, puts
.text:004006B8 lui $a0, 0x40
.text:004006BC jalr $t9 ; puts
.text:004006C0 la $a0, aEnterX # " \nEnter X:"
.text:004006C4 lw $gp, 0x28+var_18($sp)
.text:004006C8 lui $a0, 0x40
.text:004006CC la $t9, __isoc99_scanf
.text:004006D0 la $a0, %d # "%d"
.text:004006D4 jalr $t9 ; __isoc99_scanf
.text:004006D8 addiu $a1, $sp, 0x28+var_10 #
.\branch delay slot
.text:004006DC li $v1, 1
.text:004006E0 lw $gp, 0x28+var_18($sp)
.text:004006E4 beq $v0, $v1, loc_40070C
.text:004006E8 or $at, $zero #
.\branch delay slot, NOP
.text:004006EC la $t9, puts
.text:004006F0 lui $a0, 0x40
.text:004006F4 jalr $t9 ; puts
.text:004006F8 la $a0, aWhatYouEntered #
.\"What you entered? Huh?"
.text:004006FC lw $ra, 0x28+var_4($sp)
.text:00400700 move $v0, $zero
.text:00400704 jr $ra
.text:00400708 addiu $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C la $t9, printf
.text:00400710 lw $a1, 0x28+var_10($sp)
.text:00400714 lui $a0, 0x40
.text:00400718 jalr $t9 ; printf
```
scanf() returns the result of its work in register $V0. It is checked at address 0x004006E4 by comparing the values in $V0 with $V1 (1 was stored in $V1 earlier, at 0x004006DC). BEQ stands for “Branch Equal”. If the two values are equal (i.e., success), the execution jumps to address 0x0040070C.

### 7.3.8 Exercise

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

### 7.4 Exercise

- [http://challenges.re/53](http://challenges.re/53)
Chapter 8

Accessing passed arguments

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

Listing 8.1: simple example

```c
#include <stdio.h>

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

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

8.1 x86

8.1.1 MSVC

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

Listing 8.2: MSVC 2010 Express

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

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

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

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

N.B.: Function arguments are not members of the function’s stack frame, they are rather members of the stack frame of the caller function. Hence, OllyDbg marked “Arg” elements as members of another stack frame.

![OllyDbg: inside of f() function](image)

Figure 8.1: OllyDbg: inside of f() function

8.1.3 GCC

Let’s compile the same in GCC 4.4.1 and see the results in IDA:

Listing 8.3: GCC 4.4.1

```
f          public f
proc near

arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h

push ebp
mov ebp, esp
```
CHAPTER 8. ACCESSING PASSED ARGUMENTS

mov eax, [ebp+arg_0] ; 1st argument
imul eax, [ebp+arg_4] ; 2nd argument
add eax, [ebp+arg_8] ; 3rd argument
pop ebp
retn

f endp

public main
main proc near
var_10 = dword ptr -10h
var_C = dword ptr -0Ch
var_8 = dword ptr -8
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 10h
mov [esp+10h+var_8], 3 ; 3rd argument
mov [esp+10h+var_C], 2 ; 2nd argument
mov [esp+10h+var_10], 1 ; 1st argument
call f
mov edx, offset aD ; "%d\n"
mov [esp+10h+var_C], eax
mov [esp+10h+var_10], edx
call _printf
mov eax, 0
leave
retn
main endp

The result is almost the same with some minor differences discussed earlier.
The stack pointer is not set back after the two function calls (f and printf), because the penultimate LEAVE (A.6.2 on page 1304) instruction takes care of this at the end.

8.2 x64

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

Optimizing MSVC:

Listing 8.4: Optimizing MSVC 2012 x64

```assembly
$SG2997 DB '%d', 0aH, 00H

main PROC
  sub    rsp, 40
  mov    edx, 2
  lea    r8d, QWORD PTR [rdx+1] ; R8D=3
  lea    ecx, QWORD PTR [rdx-1] ; ECX=1
  call   f
  lea    rcx, OFFSET FLAT:$SG2997 ; '%d'
  mov    edx, eax
  call   printf
  xor    eax, eax
  add    rsp, 40
ret 0
main ENDP

f PROC
  ; ECX - 1st argument
  ; EDX - 2nd argument
  ; R8D - 3rd argument
  imul   ecx, edx
  lea    eax, DWORD PTR [r8+rcx]
ret 0
f ENDP
```

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

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

Listing 8.5: MSVC 2012 x64

```assembly
f proc near
  ; shadow space:
  arg_0 = dword ptr 8
  arg_8 = dword ptr 10h
  arg_10 = dword ptr 18h
  ; ECX - 1st argument
```

136
It looks somewhat puzzling because all 3 arguments from the registers are saved to
the stack for some reason. This is called “shadow space”\(^1\): every Win64 may (but
is not required to) save all 4 register values there. This is done for two reasons:
1) it is too lavish to allocate a whole register (or even 4 registers) for an input
argument, so it will be accessed via stack; 2) the debugger is always aware where
to find the function arguments at a break\(^2\).

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

It is a **caller** responsibility to allocate “shadow space” in the stack.

### 8.2.2 GCC

Optimizing GCC generates more or less understandable code:

\(^1\) [MSDN](https://msdn.microsoft.com)

\(^2\) [MSDN](https://msdn.microsoft.com)
Listing 8.6: Optimizing GCC 4.4.6 x64

```assembly
f:
    ; EDI - 1st argument
    ; ESI - 2nd argument
    ; EDX - 3rd argument
    imul   esi, edi
    lea    eax, [rdx+rsi]
    ret

main:
    sub    rsp, 8
    mov    edx, 3
    mov    esi, 2
    mov    edi, 1
    call   f
    mov    edi, OFFSET FLAT:.LC0 ; "%d\n"
    mov    esi, eax
    xor    eax, eax ; number of vector registers passed
    call   printf
    xor    eax, eax
    add    rsp, 8
    ret
```

Non-optimizing GCC:

Listing 8.7: GCC 4.4.6 x64

```assembly
f:
    ; EDI - 1st argument
    ; ESI - 2nd argument
    ; EDX - 3rd argument
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp-4], edi
    mov    DWORD PTR [rbp-8], esi
    mov    DWORD PTR [rbp-12], edx
    mov    eax, DWORD PTR [rbp-4]
    imul   eax, DWORD PTR [rbp-8]
    add    eax, DWORD PTR [rbp-12]
    leave
    ret

main:
    push   rbp
    mov    rbp, rsp
    mov    edx, 3
    mov    esi, 2
    mov    edi, 1
    call   f
```
There are no “shadow space” requirements in System V *NIX [Mit13], but the callee may need to save its arguments somewhere in case of registers shortage.

### 8.2.3 GCC: uint64_t instead of int

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

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

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

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
};

int main()
{
    printf ("%lld\n", f(0x1122334455667788,
                      0x1111111122222222,
                      0x3333333344444444));
    return 0;
};
```

Listing 8.8: Optimizing GCC 4.4.6 x64

```assembly
f proc near
imul rsi, rdi
lea rax, [rdx+rsi]
retn
f endp

main proc near
sub rsp, 8
```
The code is the same, but this time the full size registers (prefixed by R-) are used.

8.3 ARM

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

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

As was noted before, in ARM the first 4 values are usually passed in the first 4 registers (R0-R3).
The `f()` function, as it seems, uses the first 3 registers (R0-R2) as arguments.

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

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

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

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

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

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

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

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

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

The MLA instruction is not available in Thumb mode, so the compiler generates the code doing these two operations (multiplication and addition) separately. First

---

\(^3\) [Wikipedia: Multiply–accumulate operation](https://en.wikipedia.org/wiki/Multiply–accumulate_operation)

\(^4\) [wikipedia](https://en.wikipedia.org/wiki/)
the MULS instruction multiplies R0 by R1, leaving the result in register R1. The second instruction (ADDS) adds the result and R2 leaving the result in register R0.

### 8.3.4 ARM64

**Optimizing GCC (Linaro) 4.9**

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

Listing 8.9: Optimizing GCC (Linaro) 4.9

```lang-assembly
f:
    madd    w0, w0, w1, w2
    ret

main:
; save FP and LR to stack frame:
    stp    x29, x30, [sp, -16]!
    mov    w2, 3
    mov    w1, 2
    add    x29, sp, 0
    mov    w0, 1
    bl    f
    mov    w1, w0
    adrp    x0, .LC7
    add    x0, x0, :lo12:.LC7
    bl    printf
; return 0
    mov    w0, 0
; restore FP and LR
    ldp    x29, x30, [sp], 16
    ret

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

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

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

uint64_t f (uint64_t a, uint64_t b, uint64_t c) {
```
The `f()` function is the same, only the whole 64-bit X-registers are now used. Long 64-bit values are loaded into the registers by parts, this is also described here: 28.3.1 on page 608.

### Non-optimizing GCC (Linaro) 4.9

The non-optimizing compiler is more redundant:
The code saves its input arguments in the local stack, in case someone (or something) in this function needs using the \( W_0 \ldots W_2 \) registers. This prevents overwriting the original function arguments, which may be needed again in the future. This is called *Register Save Area*. [ARM13c] The callee, however, is not obliged to save them. This is somewhat similar to “Shadow Space”: 8.2.1 on page 137.

Why did the optimizing GCC 4.9 drop this argument saving code? Because it did some additional optimizing work and concluded that the function arguments will not be needed in the future and also that the registers \( W_0 \ldots W_2 \) will not be used.

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

### 8.4 MIPS

#### Listing 8.10: Optimizing GCC 4.4.5

```
.text:00000000 f:
  ; $a0=a
  ; $a1=b
  ; $a2=c
.text:00000000 mult  $a1, $a0
.text:00000004 mflo  $v0
.text:00000008 jr    $ra
.text:0000000C addu $v0, $a2, $v0 ; ⤤
.branch delay slot
; result in $v0 upon return

.text:00000010 main:
.text:00000010 var_10 = -0x10
.text:00000010 var_4 = -4
.text:00000010 lui $gp, (__gnu_local_gp >> 16)
.text:00000014 addiu $sp, -0x20
.text:00000018 la    $gp, (__gnu_local_gp & 0xFFFF)
.text:0000001C sw    $ra, 0x20+var_4($sp)
.text:00000020 sw    $gp, 0x20+var_10($sp)
; set c:
```
The first four function arguments are passed in four registers prefixed by A-.

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

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

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

The 32-bit result is left in $V0.

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

\(^5\)http://go.yurichev.com/17326
relative address of \( f() \) is known and fixed.
Chapter 9

More about results returning

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

9.1 Attempt to use the result of a function returning void

So, what if the main() function return value was declared of type void and not int?

The so-called startup-code is calling main() roughly as follows:

```
push envp
push argv
push argc
call main
push eax
call exit
```

In other words:

```
exit(main(argc,argv,envp));
```

\(^1\)See also: MSDN: Return Values (C++): [MSDN](https://msdn.microsoft.com/en-us/library/3sw9y90h)
If you declare `main()` as `void`, nothing is to be returned explicitly (using the `return` statement), then something random, that was stored in the EAX register at the end of `main()` becomes the sole argument of the `exit()` function. Most likely, there will be a random value, left from your function execution, so the exit code of program is pseudorandom.

We can illustrate this fact. Please note that here the `main()` function has a `void` return type:

```c
#include <stdio.h>

void main()
{
    printf ("Hello, world!\n");
};
```

Let's compile it in Linux.

GCC 4.8.1 replaced `printf()` with `puts()` (we have seen this before: 3.4.3 on page 23), but that's OK, since `puts()` returns the number of characters printed out, just like `printf()`. Please notice that EAX is not zeroed before `main()`'s end. This implies that the value of EAX at the end of `main()` contains what `puts()` has left there.

```
Listing 9.1: GCC 4.8.1

.LC0:
    .string "Hello, world!"
main:
    push   ebp
    mov    ebp, esp
    and    esp, -16
    sub    esp, 16
    mov    DWORD PTR [esp], OFFSET FLAT:.LC0
    call   puts
    leave
    ret
```

Let's write a bash script that shows the exit status:

```
Listing 9.2: tst.sh

#!/bin/sh
./hello_world
echo $?
```

And run it:
$ tst.sh
Hello, world!
14

14 is the number of characters printed.

## 9.2 What if we do not use the function result?

`printf()` returns the count of characters successfully output, but the result of this function is rarely used in practice. It is also possible to call a function whose essence is in returning a value, and not use it:

```c
int f()
{
    // skip first 3 random values
    rand();
    rand();
    rand();
    // and use 4th
    return rand();
};
```

The result of the `rand()` function is left in EAX, in all four cases. But in the first 3 cases, the value in EAX is just not used.

## 9.3 Returning a structure

Let’s go back to the fact that the return value is left in the EAX register. That is why old C compilers cannot create functions capable of returning something that does not fit in one register (usually `int`), but if one needs it, one have to return information via pointers passed as function’s arguments. So, usually, if a function needs to return several values, it returns only one, and all the rest—via pointers. Now it has become possible to return, let’s say, an entire structure, but that is still not very popular. If a function has to return a large structure, the caller must allocate it and pass a pointer to it via the first argument, transparently for the programmer. That is almost the same as to pass a pointer in the first argument manually, but the compiler hides it.

Small example:

```c
struct s
{
```
int a;
int b;
int c;

};

struct s get_some_values (int a)
{
    struct s rt;
    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;
    return rt;
};

...what we got (MSVC 2010 /Ox):

$T3853 = 8 ; size = 4
_a$ = 12 ; size = 4
$get_some_values@@YA?AUs@@H@Z PROC ; get_some_values
    mov  ecx, DWORD PTR _a$[esp-4]
    mov  eax, DWORD PTR $T3853[esp-4]
    lea  edx, DWORD PTR [ecx+1]
    mov  DWORD PTR [eax], edx
    lea  edx, DWORD PTR [ecx+2]
    add  ecx, 3
    mov  DWORD PTR [eax+4], edx
    mov  DWORD PTR [eax+8], ecx
    ret  0
$get_some_values@@YA?AUs@@H@Z ENDP ; get_some_values

The macro name for internal passing of pointer to a structure here is $T3853.
This example can be rewritten using the C99 language extensions:

struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
As we see, the function is just filling the structure's fields allocated by the caller function, as if a pointer to the structure was passed. So there are no performance drawbacks.
Chapter 10

Pointers

Pointers are often used to return values from functions (recall scanf() case (7 on page 90)). For example, when a function needs to return two values.

10.1 Global variables example

```c
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product) {
    *sum=x+y;
    *product=x*y;
}

int sum, product;

void main() {
    f1(123, 456, &sum, &product);
    printf("sum=%d, product=%d\n", sum, product);
}
```

This compiles to:

```
Listing 10.1: Optimizing MSVC 2010 (/Ob0)

COMM _product:DWORD
COMM _sum:DWORD
$SG2803 DB 'sum=%d, product=%d', 0aH, 00H
```

152
\_x_ = 8 ; size 4
\_y_ = 12 ; size 4
\_sum_ = 16 ; size 4
\_product_ = 20 ; size 4

_f1 PROC
    mov ecx, DWORD PTR \_y_[esp-4]
    mov eax, DWORD PTR \_x_[esp-4]
    lea edx, DWORD PTR [eax+ecx]
    imul eax, ecx
    mov ecx, DWORD PTR \_product_[esp-4]
    push esi
    mov esi, DWORD PTR \_sum_[esp]
    mov DWORD PTR [esi], edx
    mov DWORD PTR [ecx], eax
    pop esi
    ret 0
_f1 ENDP

_main PROC
    push OFFSET \_product
    push OFFSET \_sum
    push 456 ; 000001c8H
    push 123 ; 0000007bH
    call _f1
    mov eax, DWORD PTR \_product
    mov ecx, DWORD PTR \_sum
    push eax
    push ecx
    push OFFSET $SG2803
    call DWORD PTR __imp__printf
    add esp, 28 ; 0000001cH
    xor eax, eax
    ret 0
_main ENDP
Let's see this in OllyDbg:

![Figure 10.1: OllyDbg: global variables addresses are passed to f1()](image)

First, global variables' addresses are passed to `f1()`. We can click “Follow in dump” on the stack element, and we can see the place in the data segment allocated for the two variables.
These variables are zeroed, because non-initialized data (from BSS) is cleared before the execution begins: [ISO07, 6.7.8p10]. They reside in the data segment, we can verify this by pressing Alt-M and reviewing the memory map:

Figure 10.2: OllyDbg: memory map
Let's trace (F7) to the start of f1():

![OllyDbg screenshot](image)

Figure 10.3: OllyDbg: f1() starts

Two values are visible in the stack 456 (0x1C8) and 123 (0x7B), and also the addresses of the two global variables.
Let’s trace until the end of \( f1() \). In the left bottom window we see how the results of the calculation appear in the global variables:

Figure 10.4: OllyDbg: \( f1() \) execution completed
Now the global variables' values are loaded into registers ready for passing to `printf()` (via the stack):

![OllyDbg: global variables' addresses are passed into printf()](image)

**Figure 10.5:** OllyDbg: global variables' addresses are passed into `printf()`

### 10.2 Local variables example

Let's rework our example slightly:

Listing 10.2: now the sum and product variables are local

```c
void main()
{
    int sum, product; // now variables are local in this function
    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
}
```

`f1()` code will not change. Only the code of `main()` will do:

Listing 10.3: Optimizing MSVC 2010 (/Ob0)

```assembly
_product$ = -8 ; size ✔
\r = 4
_sum$ = -4 ; size ✔
\r = 4
_main PROC ; Line 10
    sub esp, 8
    ; Line 13
    lea eax, DWORD PTR _product$[esp+8]
```
push   eax
lea   ecx, DWORD PTR _sum$[esp+12]
push  ecx
push  456 ;
 push 123 ;
call   _f1
; Line 14
mov   edx, DWORD PTR _product$[esp+24]
mov   eax, DWORD PTR _sum$[esp+24]
push  edx
push  eax
push  OFFSET $SG2803
call  DWORD PTR __imp__printf
; Line 15
xor   eax, eax
add   esp, 36 ;
 ret   0

Let's look again with OllyDbg. The addresses of the local variables in the stack are 0x2EF854 and 0x2EF858. We see how these are pushed into the stack:

![Figure 10.6: OllyDbg: local variables’ addresses are pushed into the stack](image)

Figure 10.6: OllyDbg: local variables’ addresses are pushed into the stack
f1() starts. So far there is only random garbage in the stack at 0x2EF854 and 0x2EF858:

![OllyDbg: f1() starting](image)

Figure 10.7: OllyDbg: f1() starting
f1() completes:

Figure 10.8: OllyDbg: f1() completes execution

We now find 0xDB18 and 0x243 at addresses 0x2EF854 and 0x2EF858. These values are the f1() results.

10.3 Conclusion

f1() could return pointers to any place in memory, located anywhere. This is in essence the usefulness of the pointers.

By the way, C++ references work exactly the same way. Read more about them: (51.3 on page 781).
Chapter 11

GOTO operator

The GOTO operator is generally considered as anti-pattern. [Dij68], Nevertheless, it can be used reasonably [Knu74], [Yur13, p. 1.3.2].

Here is a very simple example:

```c
#include <stdio.h>

int main()
{
    printf("begin\n");
    goto exit;
    printf("skip me!\n");
exit:
    printf("end\n");
};
```

Here is what we have got in MSVC 2012:

```
Listing 11.1: MSVC 2012

$SG2934 DB 'begin', 0aH, 00H
$SG2936 DB 'skip me!', 0aH, 00H
$SG2937 DB 'end', 0aH, 00H

_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2934 ; 'begin'
    call _printf
    add esp, 4
    jmp SHORT $exit

```

163
The `goto` statement has been simply replaced by a `JMP` instruction, which has the same effect: unconditional jump to another place. The second `printf()` could be executed only with human intervention, by using a debugger or by patching the code.
This could also be useful as a simple patching exercise. Let’s open the resulting executable in Hiew:

![Hiew screen capture](image)

Figure 11.1: Hiew
Place the cursor to address JMP (0x410), press F3 (edit), press zero twice, so the opcode becomes EB 00:

Figure 11.2: Hiew

The second byte of the JMP opcode denotes the relative offset for the jump, 0 means the point right after the current instruction. So now JMP not skipping the second printf() call.

Press F9 (save) and exit. Now if we run the executable we should see this:

Figure 11.3: Patched executable output

The same result could be achieved by replacing the JMP instruction with 2 NOP instructions. NOP has an opcode of 0x90 and length of 1 byte, so we need 2 instructions as JMP replacement (which is 2 bytes in size).

11.1 Dead code

The second printf() call is also called “dead code” in compiler terms. This means that the code will never be executed. So when you compile this example with optimizations, the compiler removes “dead code”, leaving no trace of it:
However, the compiler forgot to remove the “skip me!” string.

11.2 Exercise

Try to achieve the same result using your favorite compiler and debugger.
Chapter 12

Conditional jumps

12.1 Simple example

```c
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf("a>b\n");
    if (a==b)
        printf("a==b\n");
    if (a<b)
        printf("a<b\n");
}

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf("a>b\n");
    if (a==b)
        printf("a==b\n");
    if (a<b)
        printf("a<b\n");
}

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
}
```
12.1.1 x86

x86 + MSVC

Here is how the f_signed() function looks like:

Listing 12.1: Non-optimizing MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
  push ebp
  mov ebp, esp
  mov eax, DWORD PTR _a$[ebp]
  cmp eax, DWORD PTR _b$[ebp]
  jle SHORT $LN3@f_signed
  push OFFSET $SG737 ; 'a>b'
  call _printf
  add esp, 4
$LN3@f_signed:
  mov ecx, DWORD PTR _a$[ebp]
  cmp ecx, DWORD PTR _b$[ebp]
  jne SHORT $LN2@f_signed
  push OFFSET $SG739 ; 'a==b'
  call _printf
  add esp, 4
$LN2@f_signed:
  mov edx, DWORD PTR _a$[ebp]
  cmp edx, DWORD PTR _b$[ebp]
  jge SHORT $LN4@f_signed
  push OFFSET $SG741 ; 'a<b'
  call _printf
  add esp, 4
$LN4@f_signed:
  pop ebp
  ret 0
_f_signed ENDP
```

The first instruction, JLE, stands for Jump if Less or Equal. In other words, if the second operand is larger or equal to the first one, the control flow will be passed to the specified in the instruction address or label. If this condition does not trigger because the second operand is smaller than the first one, the control flow would
not be altered and the first printf() would be executed. The second check is JNE: *Jump if Not Equal*. The control flow will not change if the operands are equal.

The third check is JGE: *Jump if Greater or Equal*—jump if the first operand is larger than the second or if they are equal. So, if all three conditional jumps are triggered, none of the printf() calls would be executed whatsoever. This is impossible without special intervention.

Now let's take a look at the `f_unsigned()` function. The `f_unsigned()` function is the same as `f_signed()`, with the exception that the JBE and JAE instructions are used instead of JLE and JGE, as follows:

```
Listing 12.2: GCC

_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_f_unsigned PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    cmp eax, DWORD PTR _b$[ebp]
    jbe SHORT $LN3@f_unsigned
    push OFFSET $SG2761 ; 'a>b'
    call _printf
    add esp, 4
$LN3@f_unsigned:
    mov ecx, DWORD PTR _a$[ebp]
    cmp ecx, DWORD PTR _b$[ebp]
    jne SHORT $LN2@f_unsigned
    push OFFSET $SG2763 ; 'a==b'
    call _printf
    add esp, 4
$LN2@f_unsigned:
    mov edx, DWORD PTR _a$[ebp]
    cmp edx, DWORD PTR _b$[ebp]
    jae SHORT $LN4@f_unsigned
    push OFFSET $SG2765 ; 'a<b'
    call _printf
    add esp, 4
$LN4@f_unsigned:
    pop ebp
    ret 0
_f_unsigned ENDP
```

As already mentioned, the branch instructions are different: JBE—*Jump if Below or Equal* and JAE—*Jump if Above or Equal*. These instructions (JA/JAE/JB/JBE) differ from JG/JGE/JL/JLE in the fact that they work with unsigned numbers.
See also the section about signed number representations (30 on page 617). That is why if we see JG/JL in use instead of JA/JB or vice-versa, we can be almost sure that the variables are signed or unsigned, respectively.

Here is also the main() function, where there is nothing much new to us:

Listing 12.3: main()

```
_main PROC
    push ebp
    mov ebp, esp
    push 2
    push 1
    call _f_signed
    add esp, 8
    push 2
    push 1
    call _f_unsigned
    add esp, 8
    xor eax, eax
    pop ebp
    ret 0
_main ENDP
```
We can see how flags are set by running this example in OllyDbg. Let's begin with `f_unsigned()`, which works with unsigned numbers. `CMP` is executed thrice here, but for the same arguments, so the flags are the same each time.

Result of the first comparison:

![Figure 12.1: OllyDbg: f_unsigned(): first conditional jump](image)

So, the flags are: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. They are named with one character for brevity in OllyDbg.

OllyDbg gives a hint that the (JBE) jump is to be triggered now. Indeed, if we take a look into [Int13], we can read there that JBE is triggering if CF=1 or ZF=1. The condition is true here, so the jump is triggered.
The next conditional jump:

![Image of OllyDbg debugger interface showing a second conditional jump]

**Figure 12.2: OllyDbg: f_unsigned(): second conditional jump**

OllyDbg gives a hint that JNZ is to be triggered now. Indeed, JNZ triggering if ZF=0 (zero flag).
The third conditional jump, JNB:

![OllyDbg: f_unsigned(): third conditional jump](image)

In [Int13] we can see that JNB triggers if CF=0 (carry flag). That is not true in our case, so the third `printf()` will execute.
Now let's review the `f_signed()` function, which works with signed values, in OllyDbg.

Flags are set in the same way: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

The first conditional jump JLE is to be triggered:

In [Int13] we find that this instruction is triggered if ZF=1 or SF≠OF. SF≠OF in our case, so the jump triggers.
The second JNZ conditional jump triggering: if ZF=0 (zero flag):

Figure 12.5: OllyDbg: f_signed(): second conditional jump
The third conditional jump JGE will not trigger because it would only do so if SF=OF, and that is not true in our case:

Figure 12.6: OllyDbg: f_signed(): third conditional jump
We can try to patch the executable file in a way that the `f_unsigned()` function would always print “a==b”, no matter the input values. Here is how it looks in Hiew:

![Figure 12.7: Hiew: f_unsigned() function](image)

Essentially, we need to accomplish three tasks:

- force the first jump to always trigger;
- force the second jump to never trigger;
- force the third jump to always trigger.

Thus we can direct the code flow to always pass through the second `printf()`, and output “a==b”.

Three instructions (or bytes) has to be patched:

- The first jump becomes JMP, but the jump offset would remain the same.
- The second jump might be triggered sometimes, but in any case it will jump to the next instruction, because, we set the jump offset to 0. In these instructions the jump offset is added to the address for the next instruction. So if the offset is 0, the jump will transfer the control to the next instruction.
- The third jump we replace with JMP just as we do with the first one, so it will always trigger.
Here is the modified code:

```assembly
Listing 12.4: GCC 4.8.1 f_signed()

f_signed:
    mov    eax, DWORD PTR [esp+8]
    cmp    DWORD PTR [esp+4], eax
```

If we miss to change any of these jumps, then several `printf()` calls may execute, while we want to execute only one.

**Non-optimizing GCC**

Non-optimizing GCC 4.4.1 produces almost the same code, but with `puts()` (3.4.3 on page 23) instead of `printf()`.

**Optimizing GCC**

An observant reader may ask, why execute `CMP` several times, if the flags has the same values after each execution? Perhaps optimizing MSVC can not do this, but optimizing GCC 4.8.1 can go deeper:
We also see JMP puts here instead of CALL puts / RETN. This kind of trick will have explained later: 13.1.1 on page 211.

This type of x86 code is somewhat rare. MSVC 2012 as it seems, can't generate such code. On the other hand, assembly language programmers are fully aware of the fact that Jcc instructions can be stacked. So if you see such stacking somewhere, it is highly probable that the code was hand-written.

The f_unsigned() function is not that æsthetically short:

```
f_unsigned:
    push    esi
    push    ebx
    sub     esp, 20
    mov     esi, DWORD PTR [esp+32]
    mov     ebx, DWORD PTR [esp+36]
    cmp     esi, ebx
    ja      .L13
    cmp     esi, ebx ; this instruction could be removed
    je      .L14

.L10:
    jb      .L15
    add     esp, 20
    pop     ebx
    pop     esi
    ret

.L15:
    mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
    add     esp, 20
    pop     ebx
    pop     esi
    jmp     puts
```
Nevertheless, there are two CMP instructions instead of three. So optimization algorithms of GCC 4.8.1 are probably not perfect yet.

### 12.1.2 ARM

#### 32-bit ARM

Optimizing Keil 6/2013 (ARM mode)

Listing 12.6: Optimizing Keil 6/2013 (ARM mode)

```
.L13:
    mov    DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
call   puts
cmp    esi, ebx
jne    .L10
.L14:
    mov    DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
add    esp, 20
pop    ebx
pop    esi
jmp    puts
```

```
 Nevertheless, there are two CMP instructions instead of three. So optimization algorithms of GCC 4.8.1 are probably not perfect yet.

### 12.1.2 ARM

#### 32-bit ARM

Optimizing Keil 6/2013 (ARM mode)

Listing 12.6: Optimizing Keil 6/2013 (ARM mode)

```
.text:000000B8 EXPORT f_signed
.text:000000B8 f_signed ; CODE XREF:
   main+C
.text:000000B8 70 40 2D E9 STMFD SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1 MOV R4, R1
.text:000000C0 04 00 50 E1 CMP R0, R4
.text:000000C4 00 50 A0 E1 MOV R5, R0
.text:000000C8 1A 0E 8F C2 ADRGT R0, aAB ; "a>b"
.text:000000CC A1 18 00 CB BLGT __2printf
.text:000000D0 04 00 55 E1 CMP R5, R4
.text:000000D4 67 0F 8F 02 ADREQ R0, aAB_0 ; "a==b"
.text:000000DC 9E 18 00 EA B __2printf
.text:000000E0 70 40 BD A8 LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8 LDMFD SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2 ADR R0, aAB_1 ; "a<b"
.text:000000EC 99 18 00 EA B __2printf
.text:000000EC ; End of function f_signed
```
Many instructions in ARM mode could be executed only when specific flags are set. E.g. this is often used when comparing numbers.

For instance, the ADD instruction is in fact named ADDAL internally, where AL stands for Always, i.e., execute always. The predicates are encoded in 4 high bits of the 32-bit ARM instructions (condition field). The B instruction for unconditional jumping is in fact conditional and encoded just like any other conditional jump, but has AL in the condition field, and it implies execute Always, ignoring flags.

The ADRGT instruction works just like ADR but executes only in case the previous CMP instruction founds one of the numbers greater than the another, while comparing the two (Greater Than).

The next BLGT instruction behaves exactly as BL and is triggered only if the result of the comparison was the same (Greater Than). ADRGT writes a pointer to the string a>b\n into R0 and BLGT calls printf(). therefore, these instructions with suffix -GT are to execute only in case the value in R0 (which is a) is bigger than the value in R4 (which is b).

Moving forward we see the ADREQ and BLEQ instructions. They behave just like ADR and BL, but are to be executed only if operands were equal to each other during the last comparison. Another CMP is located before them (because the printf() execution may have tampered the flags).

Then we see LDMGEFD, this instruction works just like LDMFD\(^1\), but is triggered only when one of the values is greater or equal than the other (Greater or Equal).

The LDMGEFD SP!, {R4-R6,PC} instruction acts like a function epilogue, but it will be triggered only if \(a \geq b\), and only then the function execution will finish.

But if that condition is not satisfied, i.e., \(a < b\), then the control flow will continue to the next “LDMFD SP!, {R4-R6,LR}” instruction, which is one more function epilogue. This instruction restores not only the R4-R6 registers state, but also LR instead of PC, thus, it does not returns from the function.

The last two instructions call printf() with the string «a<b\n» as a sole argument. We already examined an unconditional jump to the printf() function instead of function return in «printf() with several arguments» section (6.2.1 on page 70).

f_unsigned is similar, only the ADRHI, BLHI, and LDMCSFD instructions are used there, these predicates (HI = Unsigned higher, CS = Carry Set (greater than or equal)) are analogous to those examined before, but for unsigned values.

There is not much new in the main() function for us:

```asm
.text:00000128 EXPORT main
```

\(^1\)LDMFD
That is how you can get rid of conditional jumps in ARM mode.

Why is this so good? Read here: 33.1 on page 624.

There is no such feature in x86, except the CMOVcc instruction, it is the same as MOV, but triggered only when specific flags are set, usually set by CMP.
Only B instructions in Thumb mode may be supplemented by condition codes, so the Thumb code looks more ordinary.

BLE is a normal conditional jump Less than or Equal, BNE — Not Equal, BGE — Greater than or Equal.

f_unsigned is similar, only other instructions are used while dealing with unsigned values: BLS (Unsigned lower or same) and BCS (Carry Set (Greater than or equal)).

**ARM64: Optimizing GCC (Linaro) 4.9**

---

**Listing 12.9: f_signed()**

```assembly
f_signed:
; W0=a, W1=b
    cmp   w0, w1
    bgt   .L19 ; Branch if Greater Than (a>b)
    beq   .L20 ; Branch if Equal (a==b)
    bge   .L15 ; Branch if Greater than or Equal (a>=b)
                (impossible here)
                ; a<b
    adrp   x0, .LC11 ; "a<b"
    add   x0, x0, :lo12:.LC11
    b     puts
.L19:
    adrp   x0, .LC9 ; "a>b"
    add   x0, x0, :lo12:.LC9
    b     puts
.L15: ; impossible here
    ret
.L20:
    adrp   x0, .LC10 ; "a==b"
    add   x0, x0, :lo12:.LC10
    b     puts
```

**Listing 12.10: f_unsigned()**

```assembly
f_unsigned:
    stp   x29, x30, [sp, -48]!
; W0=a, W1=b
    cmp   w0, w1
    add   x29, sp, 0
    str   x19, [sp,16]
    mov   w19, w0
    bhi   .L25 ; Branch if HIgher (a>b)
```

---
The comments were added by the author of this book. What is striking is that the compiler is not aware that some conditions are not possible at all, so there is dead code at some places, which can never be executed.

Exercise

Try to optimize these functions manually for size, removing redundant instructions, without adding new ones.

12.1.3 MIPS

One distinctive MIPS feature is the absence of flags. Apparently, it was done to simplify the analysis of data dependencies.
There are instructions similar to SETcc in x86: SLT ("Set on Less Than": signed version) and SLTU (unsigned version). These instructions sets destination register value to 1 if the condition is true or to 0 if otherwise.

The destination register is then checked using BEQ ("Branch on Equal") or BNE ("Branch on Not Equal") and a jump may occur.

So, this instruction pair has to be used in MIPS for comparison and branch.

Let's first start with the signed version of our function:

Listing 12.11: Non-optimizing GCC 4.4.5 (IDA)

```plaintext
.text:00000000 f_signed: # CODE
  XREF: main+18
.text:00000000 var_10 = -0x10
.text:00000000 var_8  = -8
.text:00000000 var_4  = -4
.text:00000000 arg_0  = 0
.text:00000000 arg_4  = 4
.text:00000000
.text:00000000 addiu $sp, -0x20
.text:00000004 sw $ra, 0x20+var_4($sp)
.text:00000008 sw $fp, 0x20+var_8($sp)
.text:0000000c move $fp, $sp
.text:00000010 la $gp, __gnu_local_gp
.text:00000018 sw $gp, 0x20+var_10($sp)
; store input values into local stack:
.text:0000001c sw $a0, 0x20+arg_0($fp)
.text:00000020 sw $a1, 0x20+arg_4($fp)
; reload them.
.text:00000024 lw $v1, 0x20+arg_0($fp)
.text:00000028 lw $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text:0000002c or $at, $zero ; NOP
; this is pseudoinstruction. in fact, "slt $v0,$v0,$v1" is there.
; so $v0 will be set to 1 if $v0<$v1 (b<a) or to 0 if otherwise:
.text:00000030 slt $v0, $v1
; jump to loc_5c, if condition is not true.
; this is pseudoinstruction. in fact, "beq $v0,$zero,loc_5c" is there:
.text:00000034 beqz $v0, loc_5C
; print "a>b" and finish
.text:00000038 or $at, $zero ; branch
; delay slot, NOP
.text:0000003c lui $v0, (unk_230 >> 16) # "a>b"
```

187
CHAPTER 12. CONDITIONAL JUMPS

.addiu $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.lw $v0, (puts & 0xFFFF)($gp)
or $at, $zero ; NOP
.move $t9, $v0
.jalr $t9
.or $at, $zero ; branch
\ delay slot, NOP
.lw $gp, 0x20+var_10($fp)
.loc_5C: # CODE
\ XREF: f_signed+34
.lw $v1, 0x20+arg_0($fp)
lw $v0, 0x20+arg_4($fp)
or $at, $zero ; NOP
; check if a==b, jump to loc_90 if its not true:
bne $v1, $v0, loc_90
.or $at, $zero ; branch
\ delay slot, NOP
; condition is true, so print "a==b" and finish:
.lui $v0, (aAB >> 16) # "a==b"
.addiu $a0, $v0, (aAB & 0xFFFF)
.lw $v0, (puts & 0xFFFF)($gp)
or $at, $zero ; branch
\ delay slot, NOP
; condition is true, print "a<b" and finish:
lui $v0, (aAB_0 >> 16) # "a<b"
"SLT REG0, REG0, REG1" is reduced by IDA to its shorter form “SLT REG0, REG1”. We also see there BEQZ pseudoinstruction (“Branch if Equal to Zero”), which are in fact “BEQ REG, $ZERO, LABEL”.

The unsigned version is just the same, but SLTU (unsigned version, hence “U” in name) is used instead of SLT:

Listing 12.12: Non-optimizing GCC 4.4.5 (IDA)
CHAPTER 12. CONDITIONAL JUMPS

.text:000000FC  sw  $a0, 0x20+arg_0($fp)
.text:00000100  sw  $a1, 0x20+arg_4($fp)
.text:00000104  lw  $v1, 0x20+arg_0($fp)
.text:00000108  lw  $v0, 0x20+arg_4($fp)
.text:0000010C  or  $at, $zero
.text:00000110  sltu  $v0, $v1
.text:00000114  beqz  $v0, loc_13C
.text:00000118  or  $at, $zero
.text:0000011C  lui  $v0, (unk_230 >> 16)
.text:00000120  addiu  $a0, $v0, (unk_230 & 0xFFFF)
.text:00000124  lw  $v0, (puts & 0xFFFF)($gp)
.text:00000128  or  $at, $zero
.text:0000012C  move  $t9, $v0
.text:00000130  jalr  $t9
.text:00000134  or  $at, $zero
.text:00000138  lw  $gp, 0x20+var_10($fp)
.text:0000013C  loc_13C:  # CODE
.xreff f_unsigned+34
.text:0000013C  lw  $v1, 0x20+arg_0($fp)
.text:00000140  lw  $v0, 0x20+arg_4($fp)
.text:00000144  or  $at, $zero
.text:00000148  bne  $v1, $v0, loc_170
.text:0000014C  or  $at, $zero
.text:00000150  lui  $v0, (aAB >> 16)  # "a==b"
.text:00000154  addiu  $a0, $v0, (aAB & 0xFFFF)
.text:00000158  lw  $v0, (puts & 0xFFFF)($gp)
.text:0000015C  or  $at, $zero
.text:00000160  move  $t9, $v0
.text:00000164  jalr  $t9
.text:00000168  or  $at, $zero
.text:0000016C  lw  $gp, 0x20+var_10($fp)
.text:00000170  loc_170:  # CODE
.xreff f_unsigned+68
.text:00000170  lw  $v1, 0x20+arg_0($fp)
.text:00000174  lw  $v0, 0x20+arg_4($fp)
.text:00000178  or  $at, $zero
.text:0000017C  sltu  $v0, $v1, $v0
.text:00000180  beqz  $v0, loc_1A8
.text:00000184  or  $at, $zero
.text:00000188  lui  $v0, (aAB_0 >> 16)  # "a==b"
12.2 Calculating absolute value

A simple function:

```c
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
}
```

12.2.1 Optimizing MSVC

This is how the code is usually generated:

```
i$ = 8
my_abs PROC
    ; ECX = input
    test    ecx, ecx
```

Listing 12.13: Optimizing MSVC 2012 x64
; check for sign of input value
; skip NEG instruction if sign is positive
    jns   SHORT $LN2@my_abs
; negate value
    neg  ecx
$LN2@my_abs:
; prepare result in EAX:
    mov  eax, ecx
    ret 0
my_abs  ENDP

GCC 4.9 does mostly the same.

### 12.2.2 Optimizing Keil 6/2013: Thumb mode

#### Listing 12.14: Optimizing Keil 6/2013: Thumb mode

my_abs PROC
    CMP    r0,#0
; is input value equal to zero or greater than zero?
; skip RSBS instruction then
    BGE   |L0.6|
; subtract input value from 0:
    RSBS  r0,r0,#0
|L0.6|
    BX     lr
ENDP

ARM lacks a negate instruction, so the Keil compiler uses the “Reverse Subtract” instruction, which just subtracts with reversed operands.

### 12.2.3 Optimizing Keil 6/2013: ARM mode

It is possible to add condition codes to some instructions in ARM mode, so that is what the Keil compiler does:

#### Listing 12.15: Optimizing Keil 6/2013: ARM mode

my_abs PROC
    CMP    r0,#0
; execute "Reverse Subtract" instruction only if input value is less than 0:
    RSBLT  r0,r0,#0
    BX     lr
ENDP
Now there are no conditional jumps and this is good: 33.1 on page 624.

12.2.4 Non-optimizing GCC 4.9 (ARM64)

ARM64 has instruction NEG for negating:

Listing 12.16: Optimizing GCC 4.9 (ARM64)

```
my_abs:
    sub sp, sp, #16
    str w0, [sp,12]
    ldr w0, [sp,12]
    ; compare input value with contents of WZR register
    ; (which always holds zero)
    cmp w0, wzr
    bge .L2
    ldr w0, [sp,12]
    neg w0, w0
    b .L3
.L2:
    ldr w0, [sp,12]
.L3:
    add sp, sp, 16
    ret
```

12.2.5 MIPS

Listing 12.17: Optimizing GCC 4.4.5 (IDA)

```
my_abs:
    ; jump if $a0<0:
    bltz $a0, locret_10
    ; just return input value ($a0) in $v0:
    move $v0, $a0
    jr $ra
    or $at, $zero ; branch delay slot, NOP
locret_10:
    ; negate input value and store it in $v0:
    jr $ra
    ; this is pseudoinstruction. in fact, this
    negu $v0, $a0
```

Here we see a new instruction: BLTZ (“Branch if Less Than Zero”). There is also the NEGU pseudoinstruction, which just does subtraction from zero. The “U” suffix
in both SUBU and NEGU implies that no exception to be raised in case of integer overflow.

### 12.2.6 Branchless version?

You could have also a branchless version of this code. This we will review later: on page 712.

### 12.3 Ternary conditional operator

The ternary conditional operator in C/C++ has the following syntax:

```
expression ? expression : expression
```

Here is an example:

```c
const char* f (int a)
{
    return a==10 ? "it is ten" : "it is not ten";
};
```

#### 12.3.1 x86

Old and non-optimizing compilers generate assembly code just as if an `if/else` statement was used:

Listing 12.18: Non-optimizing MSVC 2008

```assembly
$SG746 DB 'it is ten', 00H
$SG747 DB 'it is not ten', 00H

tv65 = -4 ; this will be used as a temporary variable
_a$ = 8
_f PROC
    push ebp
    mov ebp, esp
    push ecx
    ; compare input value with 10
    cmp DWORD PTR _a$[ebp], 10
    ; jump to $LN3@f if not equal
    jne SHORT $LN3@f
    ; store pointer to the string into temporary variable:
```

194
Listing 12.19: Optimizing MSVC 2008

```
SG792 DB 'it is ten', 00H
SG793 DB 'it is not ten', 00H
_a$ = 8 ; size = 4
_f PROC
 ; compare input value with 10
 cmp DWORD PTR _a$[esp-4], 10
 mov eax, OFFSET SG792 ; 'it is ten'
 ; jump to $LN4@f if equal
 je SHORT $LN4@f
 mov eax, OFFSET SG793 ; 'it is not ten'
$LN4@f:
 ret 0
_f ENDP
```

Newer compilers are more concise:

Listing 12.20: Optimizing MSVC 2012 x64

```
SG1355 DB 'it is ten', 00H
SG1356 DB 'it is not ten', 00H
a$ = 8
f PROC
 ; load pointers to the both strings
 lea rdx, OFFSET FLAT:SG1355 ; 'it is ten'
 lea rax, OFFSET FLAT:SG1356 ; 'it is not ten'
 ; compare input value with 10
 cmp ecx, 10
```

Optimizing GCC 4.8 for x86 also uses the CMOVcc instruction, while the non-optimizing GCC 4.8 uses conditional jumps.

### 12.3.2 ARM

Optimizing Keil for ARM mode also uses the conditional instructions ADRcc:

Listing 12.21: Optimizing Keil 6/2013 (ARM mode)

```assembly
f PROC
; compare input value with 10
CMP r0,#0xa
; if comparison result is EQual, copy pointer to the "it is ten" string into R0
ADRREQ r0,|L0.16| ; "it is ten"
; if comparison result is Not Equal, copy pointer to the "it is not ten" string into R0
ADRNE r0,|L0.28| ; "it is not ten"
BX lr
ENDP

|L0.16|
DCB "it is ten",0
|L0.28|
DCB "it is not ten",0
```

Without manual intervention, the two instructions ADREQ and ADRNE cannot be executed in the same run.

Optimizing Keil for Thumb mode needs to use conditional jump instructions, since there are no load instructions that support conditional flags:

Listing 12.22: Optimizing Keil 6/2013 (Thumb mode)

```assembly
f PROC
; compare input value with 10
CMP r0,#0xa
; jump to |L0.8| if EQual
BEQ |L0.8|
ADR r0,|L0.12| ; "it is not ten"
BX lr
```
12.3.3 ARM64

Optimizing GCC (Linaro) 4.9 for ARM64 also uses conditional jumps:

Listing 12.23: Optimizing GCC (Linaro) 4.9

```assembly
f:
    cmp   x0, 10
    beq   .L3 ; branch if equal
    adrp  x0, .LC1 ; "it is ten"
    add   x0, x0, :lo12:.LC1
    ret
.L3:
    adrp  x0, .LC0 ; "it is not ten"
    add   x0, x0, :lo12:.LC0
    ret
.LC0:
    .string "it is ten"
.LC1:
    .string "it is not ten"
```

That is because ARM64 does not have a simple load instruction with conditional flags, like ADRcc in 32-bit ARM mode or CMOVcc in x86 [ARM13a, p390, C5.5]. It has, however, “Conditional SElect” instruction (CSEL), but GCC 4.9 does not seem to be smart enough to use it in such piece of code.

12.3.4 MIPS

Unfortunately, GCC 4.4.5 for MIPS is not very smart, either:

Listing 12.24: Optimizing GCC 4.4.5 (assembly output)

```assembly
$LC0:
    .ascii "it is not ten\000"
$LC1:
```

197
12.3.5 Let’s rewrite it in an if/else way

```c
const char* f (int a)
{
    if (a==10)
        return "it is ten";
    else
        return "it is not ten";
};
```

Interestingly, optimizing GCC 4.8 for x86 was also able to use CMOVcc in this case:

```
Listing 12.25: Optimizing GCC 4.8

.LC0:
    .string "it is ten"
.LC1:
    .string "it is not ten"
f:
.LFB0:
    ; compare input value with 10
    cmp DWORD PTR [esp+4], 10
    mov edx, OFFSET FLAT:.LC1 ; "it is not ten"
    mov eax, OFFSET FLAT:.LC0 ; "it is ten"
    ; if comparison result is Not Equal, copy EDX value to EAX
    ; if not, do nothing
    cmovne eax, edx
    ret
```

198
CHAPTER 12. CONDITIONAL JUMPS

Optimizing Keil in ARM mode generates code identical to listing 12.21.

But the optimizing MSVC 2012 is not that good (yet).

12.3.6 Conclusion

Why optimizing compilers try to get rid of conditional jumps? Read here about it: 33.1 on page 624.

12.4 Getting minimal and maximal values

12.4.1 32-bit

```c
int my_max(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
};

int my_min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
};
```

Listing 12.26: Non-optimizing MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    ; compare A and B:
    cmp eax, DWORD PTR _b$[ebp]
    ; jump, if A is greater or equal to B:
    jge SHORT $LN2@my_min
    ; reload A to EAX if otherwise and jump to exit
    mov eax, DWORD PTR _a$[ebp]
    jmp SHORT $LN3@my_min
_my_min ENDP
```

199
These two functions differ only in the conditional jump instruction: JGE (“Jump if Greater or Equal”) is used in the first one and JLE (“Jump if Less or Equal”) in the second.

There is one unneeded JMP instruction in each function, which MSVC probably left by mistake.

**Branchless**

ARM for Thumb mode reminds us of x86 code:

Listing 12.27: Optimizing Keil 6/2013 (Thumb mode)
The functions differ in the branching instruction: BGT and BLT.

It's possible to use conditional suffixes in ARM mode, so the code is shorter. MOVcc is to be executed only if the condition is met:

Listing 12.28: Optimizing Keil 6/2013 (ARM mode)

```assembly
my_max PROC
; R0=A
; R1=B
; compare A and B:
  CMP r0,r1
; branch if A is less B:
  BLT |L0.14|
; otherwise (A>=B) return R1 (B):
  MOVS r0,r1
|L0.14|
; return
  BX lr
ENDP

my_min PROC
; R0=A
; R1=B
; compare A and B:
  CMP r0,r1
; branch if A is greater then B:
  BGT |L0.6|
; otherwise (A<=B) return R1 (B):
  MOVS r0,r1
|L0.6|
; return
  BX lr
ENDP
```
CHAPTER 12. CONDITIONAL JUMPS

12.4.2 64-bit

```
#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
    ...
}
```

Optimizing GCC 4.8.1 and optimizing MSVC 2013 can use CMOVcc instruction, which is analogous to MOVcc in ARM:

```
my_max:
    mov    edx, DWORD PTR [esp+4]
    mov    eax, DWORD PTR [esp+8]
    ; EDX=A
    ; EAX=B
    ; compare A and B:
    cmp    edx, eax
    ; if A>=B, load A value into EAX
    ; the instruction idle if otherwise (if A<B)
    cmovge eax, edx
    ret

my_min:
    mov    edx, DWORD PTR [esp+4]
    mov    eax, DWORD PTR [esp+8]
    ; EDX=A
    ; EAX=B
    ; compare A and B:
    cmp    edx, eax
    ; if A<=B, load A value into EAX
    ; the instruction idle if otherwise (if A>B)
    cmovle eax, edx
    ret
```
if (a>b)
    return a;
else
    return b;
};

int64_t my_min(int64_t a, int64_t b)
{
    if (a<b)
        return a;
    else
        return b;
};

There is some unneeded value shuffling, but the code is comprehensible:

Listing 12.30: Non-optimizing GCC 4.9.1 ARM64

my_max:
    sub sp, sp, #16
    str x0, [sp,8]
    str x1, [sp]
    ldr x1, [sp,8]
    ldr x0, [sp]
    cmp x1, x0
    ble .L2
    ldr x0, [sp,8]
    b .L3
    .L2:
    ldr x0, [sp]
    .L3:
    add sp, sp, 16
    ret

my_min:
    sub sp, sp, #16
    str x0, [sp,8]
    str x1, [sp]
    ldr x1, [sp,8]
    ldr x0, [sp]
    cmp x1, x0
    bge .L5
    ldr x0, [sp,8]
    b .L6
    .L5:
    ldr x0, [sp]
    .L6:
    add sp, sp, 16
Branchless

No need to load function arguments from the stack, as they are already in the registers:

Listing 12.31: Optimizing GCC 4.9.1 x64

```assembly
my_max:
; RDI=A
; RSI=B
; compare A and B:
   cmp    rdi, rsi
; prepare B in RAX for return:
   mov    rax, rsi
; if A>=B, put A (RDI) in RAX for return.
; this instruction is idle if otherwise (if A<B)
   cmovge rax, rdi
ret

my_min:
; RDI=A
; RSI=B
; compare A and B:
   cmp    rdi, rsi
; prepare B in RAX for return:
   mov    rax, rsi
; if A<=B, put A (RDI) in RAX for return.
; this instruction is idle if otherwise (if A>B)
   cmovle rax, rdi
ret
```

MSVC 2013 does almost the same.

ARM64 has the CSEL instruction, which works just as MOVcc in ARM or CMOVcc in x86, just the name is different: “Conditional SELect”.

Listing 12.32: Optimizing GCC 4.9.1 ARM64

```assembly
my_max:
; X0=A
; X1=B
; compare A and B:
   cmp    x0, x1
; select X0 (A) to X0 if X0>=X1 or A>=B (Greater or Equal)
; select X1 (B) to X0 if A<B
```


```
csel  x0, x0, x1, ge
ret

my_min:
; X0=A
; X1=B
; compare A and B:
  cmp  x0, x1
; select X0 (A) to X0 if X0<=X1 or A<=B (Less or Equal)
; select X1 (B) to X0 if A>B
  csel  x0, x0, x1, le
ret
```

### 12.4.3 MIPS

Unfortunately, GCC 4.4.5 for MIPS is not that good:

Listing 12.33: Optimizing GCC 4.4.5 (IDA)

```
my_max:
; set $v1 $a1<$a0:
  slt  $v1, $a1, $a0
; jump, if $a1<$a0:
  beqz $v1, locret_10
; this is branch delay slot
; prepare $a1 in $v0 in case of branch triggered:
  move $v0, $a1
; no branch triggered, prepare $a0 in $v0:
  move $v0, $a0

locret_10:
  jr  $ra
  or  $at, $zero ; branch delay slot, NOP

; the min() function is same, but input operands in SLT instruction are swapped:
my_min:
  slt  $v1, $a0, $a1
  beqz $v1, locret_28
  move $v0, $a1
  move $v0, $a0

locret_28:
  jr  $ra
  or  $at, $zero ; branch delay slot, NOP
```
Do not forget about the *branch delay slots*: the first MOVE is executed *before* BEQZ, the second MOVE is executed only if the branch wasn’t taken.

### 12.5 Conclusion

#### 12.5.1 x86

Here’s the rough skeleton of a conditional jump:

Listing 12.34: x86

```assembly
CMP register, register/value
Jcc true ; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

#### 12.5.2 ARM

Listing 12.35: ARM

```assembly
CMP register, register/value
Bcc true ; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

#### 12.5.3 MIPS

Listing 12.36: Check for zero

```assembly
BEQZ REG, label
...
```
### 12.5.4 Branchless

If the body of a condition statement is very short, the conditional move instruction can be used: MOVcc in ARM (in ARM mode), CSEL in ARM64, CMOVcc in x86.

**ARM**

It's possible to use conditional suffixes in ARM mode for some instructions:

Listing 12.42: ARM (ARM mode)

```assembly
CMP register, register/value
istr1_cc ; some instruction will be executed if condition code is true
istr2_cc ; some other instruction will be executed if other condition code is true
... etc...
```
Of course, there is no limit for the number of instructions with conditional code suffixes, as long as the CPU flags are not modified by any of them.

Thumb mode has the IT instruction, allowing to add conditional suffixes to the next four instructions. Read more about it: 17.7.2 on page 352.

Listing 12.43: ARM (Thumb mode)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP register, register/value</td>
<td>Set these suffixes: if-then-else-else-else</td>
</tr>
<tr>
<td>ITEEE EQ</td>
<td>instruction will be executed if condition is true</td>
</tr>
<tr>
<td>instr1</td>
<td>instruction will be executed if condition is false</td>
</tr>
<tr>
<td>instr2</td>
<td>instruction will be executed if condition is false</td>
</tr>
<tr>
<td>instr3</td>
<td>instruction will be executed if condition is false</td>
</tr>
<tr>
<td>instr4</td>
<td>instruction will be executed if condition is false</td>
</tr>
</tbody>
</table>

12.6 Exercise

(ARM64) Try rewriting the code in listing 12.23 by removing all conditional jump instructions and using the CSEL instruction.
Chapter 13

switch()/case/default

13.1 Small number of cases

```c
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
        default: printf ("something unknown\n"); break;
    }
}

int main()
{
    f (2); // test
}
```

13.1.1 x86

Non-optimizing MSVC

Result (MSVC 2010):
Our function with a few cases in switch() is in fact analogous to this construction:

```c
void f (int a)
{
    if (a==0)
        printf("zero\n");
```
else if (a==1)
    printf ("one\n");
else if (a==2)
    printf ("two\n");
else
    printf ("something unknown\n");
};

If we work with switch() with a few cases it is impossible to be sure if it was a real switch() in the source code, or just a pack of if() statements. This implies that switch() is like syntactic sugar for a large number of nested if().

There is nothing especially new to us in the generated code, with the exception of the compiler moving input variable \textit{a} to a temporary local variable tv64\footnote{Local variables in stack are prefixed with tv—that's how MSVC names internal variables for its needs}.

If we compile this in GCC 4.4.1, we'll get almost the same result, even with maximal optimization turned on (-O3 option).

**Optimizing MSVC**

Now let's turn on optimization in MSVC (/Ox): cl 1.c /Fa1.asm /Ox

Listing 13.2: MSVC

```
_a$ = 8 ; size = 4
_f   PROC
    mov   eax, DWORD PTR _a$[esp-4]
    sub   eax, 0
    je    SHORT $LN4@f
    sub   eax, 1
    je    SHORT $LN3@f
    sub   eax, 1
    je    SHORT $LN2@f
    mov   DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH, 00H
    jmp   _printf
$LN2@f:
    mov   DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp   _printf
$LN3@f:
    mov   DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp   _printf
$LN4@f:
```

Here we can see some dirty hacks.

First: the value of \( a \) is placed in EAX and 0 is subtracted from it. Sounds absurd, but it is done to check if the value in EAX was 0. If yes, the ZF flag is to be set (e.g. subtracting from 0 is 0) and the first conditional jump \( \text{JE} \) (\text{Jump if Equal} or synonym \( \text{JZ} \) — \text{Jump if Zero}) is to be triggered and control flow is to be passed to the \$LN4@f label, where the 'zero' message is being printed. If the first jump doesn’t get triggered, 1 is subtracted from the input value and if at some stage the result is 0, the corresponding jump is to be triggered.

And if no jump gets triggered at all, the control flow passes to \text{printf()} with string argument 'something unknown'.

Second: we see something unusual for us: a string pointer is placed into the \( a \) variable, and then \text{printf()} is called not via \text{CALL}, but via \text{JMP}. There is a simple explanation for that: the \text{caller} pushes a value to the stack and calls our function via \text{CALL}. \text{CALL} itself pushes the return address (RA) to the stack and does an unconditional jump to our function address. Our function at any point of execution (since it do not contain any instruction that moves the stack pointer) has the following stack layout:

- ESP—points to RA
- ESP+4—points to the \( a \) variable

On the other side, when we need to call \text{printf()} here we need exactly the same stack layout, except for the first \text{printf()} argument, which needs to point to the string. And that is what our code does.

It replaces the function’s first argument with the address of the string and jumps to \text{printf()}, as if we didn’t call our function \( f() \), but directly \text{printf()}. \text{printf()} prints a string to \text{stdout} and then executes the \text{RET} instruction, which \text{POPs} RA from the stack and control flow is returned not to \( f() \) but rather to \( f() \)’s \text{callee}, bypassing the end of the \( f() \) function.

All this is possible because \text{printf()} is called right at the end of the \( f() \) function in all cases. In some way, it is similar to the \text{longjmp()} function. And of course, it is all done for the sake of speed.

A similar case with the ARM compiler is described in “\text{printf()} with several arguments”section, here (\textit{6.2.1 on page 70}).

---

\(^2\) \textit{wikipedia}
Since this example is tricky, let’s trace it in OllyDbg.

OllyDbg can detect such switch() constructs, and it can add some useful comments. EAX is 2 in the beginning, that’s the function’s input value:

![Image of OllyDbg window showing the trace of the switch statement]

Figure 13.1: OllyDbg: EAX now contain the first (and only) function argument
0 is subtracted from 2 in EAX. Of course, EAX still contains 2. But the ZF flag is now 0, indicating that the resulting value is non-zero:

![Figure 13.2: OllyDbg: SUB executed](image-url)
DEC is executed and EAX now contains 1. But 1 is non-zero, so the ZF flag is still 0:

![Figure 13.3: OllyDbg: first DEC executed](image-url)
Next DEC is executed. EAX is finally 0 and the ZF flag gets set, because the result is zero:

Figure 13.4: OllyDbg: second DEC executed

OllyDbg shows that this jump is to be taken now.
A pointer to the string “two” is to be written into the stack now:

Figure 13.5: OllyDbg: pointer to the string is to be written at the place of the first argument

Please note: the current argument of the function is 2 and 2 is now in the stack at the address 0x001EF850.
MOV writes the pointer to the string at address 0x001EF850 (see the stack window). Then, jump happens. This is the first instruction of the `printf()` function in MSVCR100.DLL (This example was compiled with /MD switch):

![Figure 13.6: OllyDbg: first instruction of printf() in MSVCR100.DLL](image)

Now `printf()` treats the string at 0x00FF3010 as its only argument and prints the string.
This is the last instruction of `printf()`:

```
Figure 13.7: OllyDbg: last instruction of `printf()` in MSVCR100.DLL

The string “two” was just printed to the console window.
```
Now let's press F7 or F8 (step over) and return...not to f(), but rather to main():

![OllyDbg: return to main()](image)

Yes, the jump was direct, from the guts of printf() to main(). Because RA in the stack points not to some place in f(), but rather to main(). And CALL 0x00FF1000 was the actual instruction which called f().

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

```assembly
.text:0000014C f1:
.text:0000014C 00 00 50 E3 CMP R0, #0
.text:00000150 13 0E 8F 02 ADREQ R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A BEQ loc_170
.text:00000158 01 00 50 E3 CMP R0, #1
.text:0000015C 4B 0F 8F 02 ADREQ R0, aOne ; "one\n"
.text:00000160 02 00 00 0A BEQ loc_170
.text:00000164 02 00 50 E3 CMP R0, #2
.text:00000168 4A 0F 8F 12 ADRNE R0, aSomethingUnkn0 ; "\n\something unknown\n"
.text:0000016C 4E 0F 8F 02 ADREQ R0, aTwo ; "two\n"
.text:00000170 . loc_170: ; CODE XREF: f1+8
.text:00000170 78 18 00 EA B __2printf
```

Again, by investigating this code we cannot say if it was a switch() in the original source code, or just a pack of if() statements.
Anyway, we see here predicated instructions again (like `ADREQ (Equal)`) which is triggered only in case \( R0 = 0 \), and then loads the address of the string «zero
» into R0. The next instruction `BEQ` redirects control flow to \( \text{loc}_170 \), if \( R0 = 0 \).

An astute reader may ask, will `BEQ` trigger correctly since `ADREQ` before it has already filled the R0 register with another value? Yes, it will since `BEQ` checks the flags set by the CMP instruction, and ADREQ does not modify any flags at all.

The rest of the instructions are already familiar to us. There is only one call to `printf()`, at the end, and we have already examined this trick here (6.2.1 on page 70). In the end, there are three paths to `printf()`.

The last instruction, `CMP R0, #2`, is needed to check if \( a = 2 \). If it is not true, then ADRNE loads a pointer to the string «something unknown 
» into R0, since \( a \) was already checked to be equal to 0 or 1, and we can sure that the \( a \) variable is not equal to these numbers at this point. And if \( R0 = 2 \), a pointer to the string «two
» will be loaded by ADREQ into R0.

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

```assembly
.text:000000D4 f1:
.text:000000D4 10 B5 PUSH {R4,LR}
.text:000000D6 00 28 CMP R0, #0
.text:000000D8 05 D0 BEQ zero_case
.text:000000DA 01 28 CMP R0, #1
.text:000000DC 05 D0 BEQ one_case
.text:000000DE 02 28 CMP R0, #2
.text:000000E0 05 D0 BEQ two_case
.text:000000E2 91 A0 ADR R0, aSomethingUnkno ; "something unknown
".
.text:000000E4 04 E0 B default_case

.text:000000E6 zero_case: ; CODE XREF: f1+4
.text:000000E6 95 A0 ADR R0, aZero ; "zero
"
.text:000000E8 02 E0 B default_case

.text:000000EA one_case: ; CODE XREF: f1+8
.text:000000EA 96 A0 ADR R0, aOne ; "one
"
.text:000000EC 00 E0 B default_case

.text:000000EE two_case: ; CODE XREF: f1+C
.text:000000EE 97 A0 ADR R0, aTwo ; "two
"
.text:000000F0 default_case ; CODE XREF: f1+10 ; f1+14
.text:000000F0 06 F0 7E F8 BL __2printf
.text:000000F4 10 BD POP {R4,PC}
```
As was already mentioned, it is not possible to add conditional predicates to most instructions in Thumb mode, so the Thumb-code here is somewhat similar to the easily understandable x86 CISC-style code.

### 13.1.4 ARM64: Non-optimizing GCC (Linaro) 4.9

```assembly
.LC12: .string "zero"
.LC13: .string "one"
.LC14: .string "two"
.LC15: .string "something unknown"

f12:
  stp x29, x30, [sp, -32]!
  add x29, sp, 0
  str w0, [x29,28]
  ldr w0, [x29,28]
  cmp w0, 1
  beq .L34
  cmp w0, 2
  beq .L35
  cmp w0, wzr
  bne .L38 ; jump to default label
  adrp x0, .LC12 ; "zero"
  add x0, x0, :lo12:.LC12
  bl puts
  b .L32

.L34:
  adrp x0, .LC13 ; "one"
  add x0, x0, :lo12:.LC13
  bl puts
  b .L32

.L35:
  adrp x0, .LC14 ; "two"
  add x0, x0, :lo12:.LC14
  bl puts
  b .L32

.L38:
  adrp x0, .LC15 ; "something unknown"
  add x0, x0, :lo12:.LC15
  bl puts
  nop

.L32:
  ldp x29, x30, [sp], 32
```
The type of the input value is `int`, hence register W0 is used to hold it instead of the whole X0 register. The string pointers are passed to `puts()` using an ADRP/ADD instructions pair just like it was demonstrated in the “Hello, world!” example: \textit{3.4.5 on page 27}.

### 13.1.5 ARM64: Optimizing GCC (Linaro) 4.9

```assembly
f12:
    cmp     w0, 1
    beq     .L31
    cmp     w0, 2
    beq     .L32
    cbz     w0, .L35
    ; default case
    adrp    x0, .LC15 ; "something unknown"
    add     x0, x0, :lo12:.LC15
    b        puts
.L35:
    adrp    x0, .LC12 ; "zero"
    add     x0, x0, :lo12:.LC12
    b        puts
.L32:
    adrp    x0, .LC14 ; "two"
    add     x0, x0, :lo12:.LC14
    b        puts
.L31:
    adrp    x0, .LC13 ; "one"
    add     x0, x0, :lo12:.LC13
    b        puts
```

Better optimized piece of code. CBZ (\textit{Compare and Branch on Zero}) instruction does jump if W0 is zero. There is also a direct jump to `puts()` instead of calling it, like it was explained before: \textit{13.1.1 on page 211}.

### 13.1.6 MIPS

Listing 13.3: Optimizing GCC 4.4.5 (IDA)

```assembly
f:
    lui     $gp, (__gnu_local_gp >> 16)
    ; is it 1?
    li      $v0, 1
```

223
```assembly
beq  $a0, $v0, loc_60
la   $gp, (__gnu_local_gp & 0xFFFF); branch
    ↓ delay slot
; is it 2?
lw   $v0, 2
beq  $a0, $v0, loc_4C
or   $at, $zero; branch delay slot, NOP
; jump, if not equal to 0:
bnez $a0, loc_38
or   $at, $zero; branch delay slot, NOP
; zero case:
lui $a0, ($LC0 >> 16) # "zero"
lw   $t9, (puts & 0xFFFF)($gp)
or  $at, $zero; load delay slot, NOP
jr   $t9; branch delay slot, NOP
la   $a0, ($LC0 & 0xFFFF) # "zero"; branch
    ↓ delay slot

loc_38: # CODE XREF: f+1C
lui $a0, ($LC3 >> 16) # "something unknown"
lw   $t9, (puts & 0xFFFF)($gp)
or  $at, $zero; load delay slot, NOP
jr   $t9
la   $a0, ($LC3 & 0xFFFF) # "something unknown"; branch delay slot

loc_4C: # CODE XREF: f+14
lwi $a0, ($LC2 >> 16) # "two"
lw   $t9, (puts & 0xFFFF)($gp)
or  $at, $zero; load delay slot, NOP
jr   $t9
la   $a0, ($LC2 & 0xFFFF) # "two"; branch
    ↓ delay slot

loc_60: # CODE XREF: f+8
lui $a0, ($LC1 >> 16) # "one"
lw   $t9, (puts & 0xFFFF)($gp)
```
The function always ends with calling `puts()` , so here we see a jump to `puts()` (JR: “Jump Register”) instead of “jump and link”. We talked about this earlier: 13.1.1 on page 211.

We also often see NOP instructions after LW ones. This is “load delay slot”: another delay slot in MIPS. An instruction next to LW may execute at the moment while LW loads value from memory. However, the next instruction must not use the result of LW. Modern MIPS CPUs have a feature to wait if the next instruction uses result of LW, so this is somewhat outdated, but GCC still adds NOPs for older MIPS CPUs. In general, it can be ignored.

### 13.1.7 Conclusion

A `switch()` with few cases is indistinguishable from an `if/else` construction, for example: listing.13.1.1.

### 13.2 A lot of cases

If a `switch()` statement contains a lot of cases, it is not very convenient for the compiler to emit too large code with a lot JE/JNE instructions.

```c
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    }
}

int main()
{
```
f (2); // test
};

13.2.1  x86

Non-optimizing MSVC

We get (MSVC 2010):

Listing 13.4: MSVC 2010

tv64 = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
    push ebp
    mov ebp, esp
    push ecx
    mov eax, DWORD PTR _a$[ebp]
    mov DWORD PTR tv64[ebp], eax
    cmp DWORD PTR tv64[ebp], 4
    ja SHORT $LN1@f
    mov ecx, DWORD PTR tv64[ebp]
    jmp DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push OFFSET $SG739 ; 'zero', 0aH, 00H
    call _printf
    add esp, 4
    jmp SHORT $LN9@f
$LN5@f:
    push OFFSET $SG741 ; 'one', 0aH, 00H
    call _printf
    add esp, 4
    jmp SHORT $LN9@f
$LN4@f:
    push OFFSET $SG743 ; 'two', 0aH, 00H
    call _printf
    add esp, 4
    jmp SHORT $LN9@f
$LN3@f:
    push OFFSET $SG745 ; 'three', 0aH, 00H
    call _printf
    add esp, 4
    jmp SHORT $LN9@f
$LN2@f:
    push OFFSET $SG747 ; 'four', 0aH, 00H
    call _printf

What we see here is a set of printf() calls with various arguments. All they have not only addresses in the memory of the process, but also internal symbolic labels assigned by the compiler. All these labels are also mentioned in the $LN11@f internal table.

At the function start, if \( a \) is greater than 4, control flow is passed to label $LN1@f, where printf() with argument 'something unknown' is called.

But if the value of \( a \) is less or equals to 4, then it gets multiplied by 4 and added with the $LN11@f table address. That is how an address inside the table is constructed, pointing exactly to the element we need. For example, let’s say \( a \) is equal to 2. \( 2 \times 4 = 8 \) (all table elements are addresses in a 32-bit process and that is why all elements are 4 bytes wide). The address of the $LN11@f table + 8 is the table element where the $LN4@f label is stored. JMP fetches the $LN4@f address from the table and jumps to it.

This table is sometimes called jump table or branch table.

Then the corresponding printf() is called with argument 'two'. Literally, the jmp DWORD PTR $LN11@f[ecx*4] instruction implies jump to the DWORD that is stored at address $LN11@f + ecx * 4.

npad (88 on page 1256) is assembly language macro that aligning the next label so that it is to be stored at an address aligned on a 4 byte (or 16 byte) boundary. This is very suitable for the processor since it is able to fetch 32-bit values from memory through the memory bus, cache memory, etc, in a more effective way if it

---

3The whole method was once called computed GOTO in early versions of FORTRAN: wikipedia. Not quite relevant these days, but what a term!
is aligned.
Let's try this example in OllyDbg. The input value of the function (2) is loaded into EAX:

![Figure 13.9: OllyDbg: function's input value is loaded in EAX](image-url)
The input value is checked, is it bigger than 4? If not, the “default” jump is not taken:

Figure 13.10: OllyDbg: 2 is no bigger than 4: no jump is taken
Here we see a jumptable:

![Image of OllyDbg interface showing a jumptable]

**Figure 13.11:** OllyDbg: calculating destination address using jumptable

Here we’ve clicked “Follow in Dump” → “Address constant”, so now we see the *jumptable* in the data window. These are 5 32-bit values\(^4\). ECX is now 2, so the second element (counting from zero) of the table is to be used. It’s also possible to click “Follow in Dump” → “Memory address” and OllyDbg will show the element addressed by the _JMP_ instruction. That’s 0x010B103A.

---

\(^4\)They are underlined by OllyDbg because these are also FIXUPS: [68.2.6 on page 992](#), we are going to come back to them later
After the jump we are at 0x010B103A: the code printing “two” will now be executed:

![OllyDbg screenshot showing the case label](image)

Figure 13.12: OllyDbg: now we at the case: label

### Non-optimizing GCC

Let’s see what GCC 4.4.1 generates:

#### Listing 13.5: GCC 4.4.1

```
public f  
f proc near ; CODE XREF: main+10

var_18 = dword ptr -18h  
arg_0 = dword ptr 8

push ebp  
mov ebp, esp  
sub esp, 18h  
cmp [ebp+arg_0], 4  
ja short loc_8048444  
mov eax, [ebp+arg_0]  
shl eax, 2  
mov eax, ds:off_804855C[eax]  
jmp eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C

test [esp+18h+var_18], offset aZero ; "zero"
call _puts  
jmp short locret_8048450
```
loc_804840C: ; DATA XREF: .rodata:08048560
  mov [esp+18h+var_18], offset aOne ; "one"
call _puts
jmp short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
  mov [esp+18h+var_18], offset aTwo ; "two"
call _puts
jmp short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
  mov [esp+18h+var_18], offset aThree ; "three"
call _puts
jmp short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
  mov [esp+18h+var_18], offset aFour ; "four"
call _puts
jmp short locret_8048450

loc_8048444: ; CODE XREF: f+A
  mov [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
call _puts

locret_8048450: ; CODE XREF: f+26
  leave
  retn
f endp

off_804855C dd offset loc_80483FE ; DATA XREF: f+12
  dd offset loc_804840C
  dd offset loc_804841A
  dd offset loc_8048428
  dd offset loc_8048436

It is almost the same, with a little nuance: argument arg_0 is multiplied by 4 by shifting it to left by 2 bits (it is almost the same as multiplication by 4) (16.2.1 on page 297). Then the address of the label is taken from the off_804855C array, stored in EAX, and then JMP EAX does the actual jump.

### 13.2.2 ARM: Optimizing Keil 6/2013 (ARM mode)
Listing 13.6: Optimizing Keil 6/2013 (ARM mode)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000174</td>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>00000174</td>
<td>05 00 50 E3</td>
<td>CMP R0, #5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; switch 5</td>
</tr>
<tr>
<td>00000178</td>
<td>00 F1 8F 30</td>
<td>ADDCC PC, PC, R0,LSL#2 ; switch jump</td>
</tr>
<tr>
<td>0000017C</td>
<td>0E 00 00 EA</td>
<td>B default_case ; jumptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; 00000178 default case</td>
</tr>
<tr>
<td>00000180</td>
<td>loc_180</td>
<td>CODE XREF: f2+4</td>
</tr>
<tr>
<td>00000180</td>
<td>03 00 00 EA</td>
<td>B zero_case ; jumptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; 00000178 case 0</td>
</tr>
<tr>
<td>00000184</td>
<td>loc_184</td>
<td>CODE XREF: f2+4</td>
</tr>
<tr>
<td>00000184</td>
<td>04 00 00 EA</td>
<td>B one_case ; jumptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; 00000178 case 1</td>
</tr>
<tr>
<td>00000188</td>
<td>loc_188</td>
<td>CODE XREF: f2+4</td>
</tr>
<tr>
<td>00000188</td>
<td>05 00 00 EA</td>
<td>B two_case ; jumptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; 00000178 case 2</td>
</tr>
<tr>
<td>0000018C</td>
<td>loc_18C</td>
<td>CODE XREF: f2+4</td>
</tr>
<tr>
<td>0000018C</td>
<td>06 00 00 EA</td>
<td>B three_case ; jumptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; 00000178 case 3</td>
</tr>
<tr>
<td>00000190</td>
<td>loc_190</td>
<td>CODE XREF: f2+4</td>
</tr>
<tr>
<td>00000190</td>
<td>07 00 00 EA</td>
<td>B four_case ; jumptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; 00000178 case 4</td>
</tr>
<tr>
<td>00000194</td>
<td>zero_case</td>
<td>CODE XREF: f2+4</td>
</tr>
<tr>
<td>00000194</td>
<td>; f2:loc_180</td>
<td></td>
</tr>
<tr>
<td>00000194</td>
<td>EC 00 8F E2</td>
<td>ADR R0, aZero ; jumptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; 00000178 case 0</td>
</tr>
<tr>
<td>00000198</td>
<td>06 00 00 EA</td>
<td>B loc_1B8</td>
</tr>
<tr>
<td>0000019C</td>
<td>one_case</td>
<td>CODE XREF: f2+4</td>
</tr>
<tr>
<td>0000019C</td>
<td>; f2:loc_184</td>
<td></td>
</tr>
<tr>
<td>0000019C</td>
<td>EC 00 8F E2</td>
<td>ADR R0, aOne ; jumptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; 00000178 case 1</td>
</tr>
<tr>
<td>000001A0</td>
<td>04 00 00 EA</td>
<td>B loc_1B8</td>
</tr>
</tbody>
</table>
This code makes use of the ARM mode feature in which all instructions have a fixed size of 4 bytes.

Let’s keep in mind that the maximum value for \( a \) is 4 and any greater value will cause “something unknown\n” string to be printed.

The first \( \text{CMP R0, #5} \) instruction compares the input value of \( a \) with 5.

The next \( \text{ADDCC PC, PC, R0, LSL#2} \) instruction is being executed only if \( R0 < 5 \) (\( CC=\text{Carry clear / Less than} \)). Consequently, if ADDCC does not trigger (it is a \( R0 \geq 5 \) case), a jump to \( \text{default_case} \) label will occur.

But if \( R0 < 5 \) and ADDCC triggers, the following is to be happen:

The value in R0 is multiplied by 4. In fact, LSL#2 \( ^5 \) at the instruction’s suffix stands for “shift left by 2 bits”. But as we will see later (\( 16.2.1 \) on page 296) in section

\(^5\)ADD—addition
“Shifts”, shift left by 2 bits is equivalent to multiplying by 4.

Then we add \( R0 \times 4 \) to the current value in PC, thus jumping to one of the B (Branch) instructions located below.

At the moment of the execution of ADDCC, the value in PC is 8 bytes ahead (0x180) than the address at which the ADDCC instruction is located (0x178), or, in other words, 2 instructions ahead.

This is how the pipeline in ARM processors works: when ADDCC is executed, the processor at the moment is beginning to process the instruction after the next one, so that is why PC points there. This has to be memorized.

If \( a = 0 \), then is to be added to the value in PC, and the actual value of the PC will be written into PC (which is 8 bytes ahead) and a jump to the label loc_180 will happen, which is 8 bytes ahead of the point where the ADDCC instruction is.

If \( a = 1 \), then \( PC + 8 + a \times 4 = PC + 8 + 1 \times 4 = PC + 12 = 0x184 \) will be written to PC, which is the address of the loc_184 label.

With every 1 added to \( a \), the resulting PC is increased by 4. 4 is the instruction length in ARM mode and also, the length of each B instruction, of which there are 5 in row.

Each of these five B instructions passes control further, to what was programmed in the switch(). Pointer loading of the corresponding string occurs there, etc.

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

Listing 13.7: Optimizing Keil 6/2013 (Thumb mode)

```
000000F6 EXPORT f2
000000F6 f2
000000F6 10 B5 PUSH {R4,LR}
000000F8 03 00 MOVS R3, R0
000000FA 06 F0 69 F8 BL __ARM_common_switch8_thumb ; switch 6 cases
  \( \text{\textbackslash r curvearrowse} \)
000000FE 05 DCB 5
000000FF 04 06 08 0A 0C 10 DCB 4, 6, 8, 0xA, 0xC, 0x10 ; \( \text{\textbackslash r curvearrowse} \)
  \( \text{\textbackslash r curvearrowse} \) jump table for switch statement
00000105 00 ALIGN 2
00000106
00000106 zero_case ; CODE XREF: f2+4
00000106 8D A0 ADR R0, aZero ; jumptable \( \text{\textbackslash r curvearrowse} \)
  \( \text{\textbackslash r curvearrowse} \) 000000FA case 0
00000108 06 E0 B loc_118
0000010A
```
CHAPTER 13. SWITCH()/CASE/DEFAULT

0000010A one_case ; CODE XREF: f2+4
  0000010A 8E A0 ADR R0, aOne ; jumptable
  ↓ 000000FA case 1
0000010C 04 E0 B loc_118

0000010E two_case ; CODE XREF: f2+4
  0000010E 8F A0 ADR R0, aTwo ; jumptable
  ↓ 000000FA case 2
00000110 02 E0 B loc_118

00000112 three_case ; CODE XREF: f2+4
  00000112 90 A0 ADR R0, aThree ; jumptable
  ↓ 000000FA case 3
00000114 00 E0 B loc_118

00000116 four_case ; CODE XREF: f2+4
  00000116 91 A0 ADR R0, aFour ; jumptable
  ↓ 000000FA case 4
00000118

00000118 loc_118 ; CODE XREF: f2+12
  00000118 ; f2+16
00000118 06 F0 6A F8 BL __2printf
0000011C 10 BD POP \{R4,PC\}

0000011E default_case ; CODE XREF: f2+4
  0000011E 82 A0 ADR R0, aSomethingUnkno ;
  ↓ jumptable 000000FA default case
00000120 FA E7 B loc_118

000061D0 EXPORT __ARM_common_switch8_thumb
  000061D0 __ARM_common_switch8_thumb ; CODE
  ↓ XREF: example6_f2+4
000061D0 78 47 BX PC
000061D2 00 00 ALIGN 4
000061D2 ; End of function
  ↓ __ARM_common_switch8_thumb
000061D2
000061D4 __32__ARM_common_switch8_thumb ;
  ↓ CODE XREF: __ARM_common_switch8_thumb
000061D4 01 C0 5E E5 LDRB R12, [LR,#-1]
One cannot be sure that all instructions in Thumb and Thumb-2 modes has the same size. It can even be said that in these modes the instructions have variable lengths, just like in x86.

So there is a special table added that contains information about how much cases are there (not including default-case), and an offset for each with a label to which control must be passed in the corresponding case.

A special function is present here in order to deal with the table and pass control, named `__ARM_common_switch8_thumb`. It starts with `BX PC`, whose function is to switch the processor to ARM-mode. Then you see the function for table processing. It is too complex to describe it here now, so let's omit it.

It is interesting to note that the function uses the `LR` register as a pointer to the table. Indeed, after calling of this function, `LR` contains the address after `BL __ARM_common_switch8_thumb` instruction, where the table starts.

It is also worth noting that the code is generated as a separate function in order to reuse it, so the compiler not generates the same code for every switch() statement. IDA successfully perceived it as a service function and a table, and added comments to the labels like `jumptable 000000FA case 0`.

### 13.2.4 MIPS

Listing 13.8: Optimizing GCC 4.4.5 (IDA)

```assembly
f:
    lui   $gp, (__gnu_local_gp >> 16)
; jump to loc_24 if input value is lesser than 5:
    sltiu  $v0, $a0, 5
    bnez  $v0, loc_24
    la  $gp, (__gnu_local_gp & 0xFFFF) ; branch
    delay slot
; input value is greater or equal to 5.
; print "something unknown" and finish:
    lui   $a0, ($LC5 >> 16) # "something unknown"
```

238
lw  $t9, (puts & 0xFFFF)($gp)
or  $at, $zero ; NOP
ejr  $t9
la  $a0, ($LC5 & 0xFFFF) # "something unknown" ; branch delay slot

loc_24:                       # CODE XREF: f+8
; load address of jumptable
; LA is pseudoinstruction, LUI and ADDIU pair are there in fact:
   la  $v0, off_120
; multiply input value by 4:
   sll  $a0, 2
; sum up multiplied value and jumptable address:
   addu  $a0, $v0, $a0
; load element from jumptable:
   lw  $v0, 0($a0)
or  $at, $zero ; NOP
; jump to the address we got in jumptable:
   jr  $v0
or  $at, $zero ; branch delay slot, NOP

sub_44:                       # DATA XREF: .rodata
  ↩:0000012C
; print "three" and finish
  lui  $a0, ($LC3 >> 16) # "three"
  lw  $t9, (puts & 0xFFFF)($gp)
or  $at, $zero ; NOP
  jr  $t9
  la  $a0, ($LC3 & 0xFFFF) # "three" ; ↩ branch delay slot

sub_58:                       # DATA XREF: .rodata
  ↩:00000130
; print "four" and finish
  lui  $a0, ($LC4 >> 16) # "four"
  lw  $t9, (puts & 0xFFFF)($gp)
or  $at, $zero ; NOP
  jr  $t9
  la  $a0, ($LC4 & 0xFFFF) # "four" ; branch ↩ delay slot

sub_6C:                       # DATA XREF: .rodata:
  ↩ off_120
; print "zero" and finish
  lui  $a0, ($LC0 >> 16) # "zero"
  lw  $t9, (puts & 0xFFFF)($gp)
or  $at, $zero ; NOP
The new instruction for us is SLTIU (“Set on Less Than Immediate Unsigned”). This is the same as SLTU (“Set on Less Than Unsigned”), but “I” stands for “immediate”, i.e., a number has to be specified in the instruction itself.

BNEZ is “Branch if Not Equal to Zero”.

Code is very close to the other ISAs. SLL (“Shift Word Left Logical”) does multiplication by 4. MIPS is a 32-bit CPU after all, so all addresses in the jumptable are 32-bit ones.

### 13.2.5 Conclusion

Rough skeleton of `switch0`:

Listing 13.9: x86
MOV REG, input
CMP REG, 4 ; maximal number of cases
JA default
SHL REG, 2 ; find element in table. shift for 3 bits in x64.
MOV REG, jump_table[REG]
JMP REG

case1:
  ; do something
  JMP exit
case2:
  ; do something
  JMP exit
case3:
  ; do something
  JMP exit
case4:
  ; do something
  JMP exit
case5:
  ; do something
  JMP exit

default:
  ...

exit:
  ....

jump_table dd case1
dd case2
dd case3
dd case4
dd case5

The jump to the address in the jump table may also be implemented using this instruction: JMP jump_table[REG*4]. Or JMP jump_table[REG*8] in x64.

A jumptable is just array of pointers, like the one described later: 18.5 on page 385.
13.3 When there are several case statements in one block

Here is a very widespread construction: several case statements for a single block:

```c
#include <stdio.h>

void f(int a)
{
    switch (a)
    {
    case 1:
    case 2:
    case 7:
    case 10:
        printf ("1, 2, 7, 10\n");
        break;
    case 3:
    case 4:
    case 5:
    case 6:
        printf ("3, 4, 5\n");
        break;
    case 8:
    case 9:
    case 20:
    case 21:
        printf ("8, 9, 21\n");
        break;
    case 22:
        printf ("22\n");
        break;
    default:
        printf ("default\n");
        break;
    }
};

int main()
{
    f(4);
};
```

It’s too wasteful to generate a block for each possible case, so what is usually done is to generate each block plus some kind of dispatcher.
13.3.1 MSVC

Listing 13.10: Optimizing MSVC 2010

```assembly
.a$ = 8
_f PROC
    mov eax, DWORD PTR _a$[esp-4]
    dec eax
    cmp eax, 21
    ja SHORT $LN1@f
    movzx eax, BYTE PTR _a$[esp-4]
    jmp DWORD PTR __imp__printf
$LN5@f:
    mov DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
    jmp DWORD PTR __imp__printf
$LN4@f:
    mov DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
    jmp DWORD PTR __imp__printf
$LN3@f:
    mov DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
    jmp DWORD PTR __imp__printf
$LN2@f:
    mov DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
    jmp DWORD PTR __imp__printf
$LN1@f:
    mov DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
    jmp DWORD PTR __imp__printf
$LN10@f:
    DB 0 ; a=1
    DB 0 ; a=2
    DB 1 ; a=3
npad 2 ; align $LN11@f table on 16-byte boundary
$LN11@f:
    DD $LN5@f ; print '1, 2, 7, 10'
    DD $LN4@f ; print '3, 4, 5'
    DD $LN3@f ; print '8, 9, 21'
    DD $LN2@f ; print '22'
    DD $LN1@f ; print 'default'
```

243
We see two tables here: the first table ($LN10$) is an index table, and the second one ($LN11$) is an array of pointers to blocks.

First, the input value is used as an index in the index table (line 13).

Here is a short legend for the values in the table: 0 is the first case block (for values 1, 2, 7, 10), 1 is the second one (for values 3, 4, 5), 2 is the third one (for values 8, 9, 21), 3 is the fourth one (for value 22), 4 is for the default block.

There we get an index for the second table of code pointers and we jump to it (line 14).

What is also worth noting is that there is no case for input value 0. That's why we see the DEC instruction at line 10, and the table starts at $a = 1$, because there is no need to allocate a table element for $a = 0$.

This is a very widespread pattern.

So why is this economical? Why isn’t it possible to make it as before (13.2.1 on page 232), just with one table consisting of block pointers? The reason is that the elements in index table are 8-bit, hence it’s all more compact.

### 13.3.2 GCC

GCC does the job in the way we already discussed (13.2.1 on page 232), using just one table of pointers.
There is no code to be triggered if the input value is 0, so GCC tries to make the jump table more compact and so it starts at 1 as an input value.

GCC 4.9.1 for ARM64 uses an even cleverer trick. It’s able to encode all offsets as 8-bit bytes. Let’s recall that all ARM64 instructions have a size of 4 bytes. GCC is using the fact that all offsets in my tiny example are in close proximity to each other. So the jump table consisting of single bytes.

Listing 13.11: Optimizing GCC 4.9.1 ARM64

```assembly
f14:
; input value in W0
    sub    w0, w0, #1
    cmp    w0, 21
; branch if less or equal (unsigned):
    bls    .L9
.L2:
; print "default":
    adrp   x0, .LC4
    add    x0, x0, :lo12:.LC4
    b      puts
.L9:
; load jumptable address to X1:
    adrp   x1, .L4
    add    x1, x1, :lo12:.L4
; W0=input_value-1
; load byte from the table:
    ldrb   w0, [x1,w0,uxtw]
; load address of the Lrtx label:
    adr    x1, .Lrtx4
; multiply table element by 4 (by shifting 2 bits left) and add (or subtract) to the address of Lrtx:
    add    x0, x1, w0, sxtb #2
; jump to the calculated address:
    br     x0
; this label is pointing in code (text) segment:
.Lrtx4:
    .section   .rodata
; everything after ".section" statement is allocated in the read-only data (rodata) segment:
.L4:
    .byte   (.L3 - .Lrtx4) / 4   ; case 1
    .byte   (.L3 - .Lrtx4) / 4   ; case 2
    .byte   (.L5 - .Lrtx4) / 4   ; case 3
    .byte   (.L5 - .Lrtx4) / 4   ; case 4
    .byte   (.L5 - .Lrtx4) / 4   ; case 5
    .byte   (.L5 - .Lrtx4) / 4   ; case 6
```
.byte (.L3 - .Lrtx4) / 4 ; case 7
.byte (.L6 - .Lrtx4) / 4 ; case 8
.byte (.L6 - .Lrtx4) / 4 ; case 9
.byte (.L3 - .Lrtx4) / 4 ; case 10
.byte (.L2 - .Lrtx4) / 4 ; case 11
.byte (.L2 - .Lrtx4) / 4 ; case 12
.byte (.L2 - .Lrtx4) / 4 ; case 13
.byte (.L2 - .Lrtx4) / 4 ; case 14
.byte (.L2 - .Lrtx4) / 4 ; case 15
.byte (.L2 - .Lrtx4) / 4 ; case 16
.byte (.L2 - .Lrtx4) / 4 ; case 17
.byte (.L2 - .Lrtx4) / 4 ; case 18
.byte (.L2 - .Lrtx4) / 4 ; case 19
.byte (.L6 - .Lrtx4) / 4 ; case 20
.byte (.L6 - .Lrtx4) / 4 ; case 21
.byte (.L7 - .Lrtx4) / 4 ; case 22

.text
; everything after ".text" statement is allocated in the code (text) segment:

.L7:
; print "22"
  adrp  x0, .LC3
  add  x0, x0, :lo12:.LC3
  b  puts

.L6:
; print "8, 9, 21"
  adrp  x0, .LC2
  add  x0, x0, :lo12:.LC2
  b  puts

.L5:
; print "3, 4, 5"
  adrp  x0, .LC1
  add  x0, x0, :lo12:.LC1
  b  puts

.L3:
; print "1, 2, 7, 10"
  adrp  x0, .LC0
  add  x0, x0, :lo12:.LC0
  b  puts

.LC0:
.string "1, 2, 7, 10"

.LC1:
.string "3, 4, 5"

.LC2:
.string "8, 9, 21"

.LC3:
.string "22"

.LC4:
Let's compile this example to object file and open it in IDA. Here is the jump table:

Listing 13.12: jumptable in IDA

```
.rodata:0000000000000064 AREA .rodata, DATA, READONLY
.rodata:0000000000000064 ; ORG 0x64
.rodata:0000000000000064 $d DCB 9 ; case 1
.rodata:0000000000000065 DCB 9 ; case 2
.rodata:0000000000000066 DCB 6 ; case 3
.rodata:0000000000000067 DCB 6 ; case 4
.rodata:0000000000000068 DCB 6 ; case 5
.rodata:0000000000000069 DCB 6 ; case 6
.rodata:000000000000006A DCB 9 ; case 7
.rodata:000000000000006B DCB 3 ; case 8
.rodata:000000000000006C DCB 3 ; case 9
.rodata:000000000000006D DCB 9 ; case 10
.rodata:000000000000006E DCB 0xF7 ; case 11
.rodata:000000000000006F DCB 0xF7 ; case 12
.rodata:0000000000000070 DCB 0xF7 ; case 13
.rodata:0000000000000071 DCB 0xF7 ; case 14
.rodata:0000000000000072 DCB 0xF7 ; case 15
.rodata:0000000000000073 DCB 0xF7 ; case 16
.rodata:0000000000000074 DCB 0xF7 ; case 17
.rodata:0000000000000075 DCB 0xF7 ; case 18
.rodata:0000000000000076 DCB 0xF7 ; case 19
.rodata:0000000000000077 DCB 3 ; case 20
.rodata:0000000000000078 DCB 3 ; case 21
.rodata:0000000000000079 DCB 0 ; case 22
.rodata:000000000000007B ; .rodata ends
```

So in case of 1, 9 is to be multiplied by 4 and added to the address of Lrtx4 label. In case of 22, 0 is to be multiplied by 4, resulting in 0. Right after the Lrtx4 label is the L7 label, where you can find the code that prints “22”. There is no jump table in the code segment, it's allocated in a separate .rodata section (there is no special need to place it in the code section).

There are also negative bytes (0xF7), they are used for jumping back to the code that prints the “default” string (at .L2).

### 13.4 Fall-through

Another very popular usage of `switch()` is the fall-through. Here is a small example:
### CHAPTER 13. SWITCH() / CASE / DEFAULT

#### Listing 13.13: MSVC 2012

```c
#define R 1
#define W 2
#define RW 3

void f(int type)
{
    int read=0, write=0;

    switch (type)
    {
    case RW:
        read=1;
    case W:
        write=1;
        break;
    case R:
        read=1;
        break;
    default:
        break;
    }
    printf ("read=%d, write=%d\n", read, write);
}
```

If `type = 1 (R)`, `read` is to be set to 1, if `type = 2 (W)`, `write` is to be set to 2. In case of `type = 3 (RW)`, both `read` and `write` is to be set to 1.

The code at line 14 is executed in two cases: if `type = RW` or if `type = W`. There is no “break” for “case RW”x and that’s OK.

### 13.4.1 MSVC x86

Listing 13.13: MSVC 2012

```assembly
SG1305 DB 'read=%d, write=%d', 0AH, 00H
_write$ = -12 ; size = 4
_read$ = -8 ; size = 4
tv64 = -4 ; size = 4
_type$ = 8 ; size = 4
_f PROC
    push ebp
    mov ebp, esp
    sub esp, 12
    mov DWORD PTR _read$[ebp], 0
    mov DWORD PTR _write$[ebp], 0
```

248
The code mostly resembles what is in the source. There are no jumps between labels $LN4@f$ and $LN3@f$: so when code flow is at $LN4@f$, read is first set to 1, then write. This is why it’s called fall-through: code flow falls through one piece of code (setting read) to another (setting write). If type = W, we land at $LN3@f$, so no code setting read to 1 is executed.

### 13.4.2 ARM64

Listing 13.14: GCC (Linaro) 4.9

```
.LC0:
    .string "read=%d, write=%d\n"
f:
    stp    x29, x30, [sp, -48]!
    add    x29, sp, 0
    str    w0, [x29,28]
```

Merely the same thing. There are no jumps between labels .L4 and .L3.

### 13.5 Exercises

#### 13.5.1 Exercise #1

It's possible to rework the C example in 13.2 on page 225 in such way that the compiler can produce even smaller code, but will work just the same. Try to achieve it.
Chapter 14

Loops

14.1 Simple example

14.1.1 x86

There is a special LOOP instruction in x86 instruction set for checking the value in register ECX and if it is not 0, to decrement ECX and pass control flow to the label in the LOOP operand. Probably this instruction is not very convenient, and there are no any modern compilers which emit it automatically. So, if you see this instruction somewhere in code, it is most likely that this is a manually written piece of assembly code.

In C/C++ loops are usually constructed using for(), while() or do/while() statements.

Let’s start with for().

This statement defines loop initialization (set loop counter to initial value), loop condition (is the counter bigger than a limit?), what is done at each iteration (increment/decrement) and of course loop body.

```c
for (initialization; condition; at each iteration)
{
    loop_body;
}
```

The generated code is consisting of four parts as well.

Let’s start with a simple example:

```c
#include <stdio.h>
```
void printing_function(int i) {
    printf("f(%d)\n", i);
};

int main() {
    int i;
    for (i=2; i<10; i++)
        printing_function(i);
    return 0;
};

Result (MSVC 2010):

Listing 14.1: MSVC 2010

_i$ = -4
_main PROC
    push ebp
    mov ebp, esp
    push ecx
    mov DWORD PTR _i$[ebp], 2 ; loop initialization
    jmp SHORT $LN3@main
$LN2@main:
    mov eax, DWORD PTR _i$[ebp] ; here is what we do after each iteration:
    add eax, 1 ; add 1 to (i) value
    mov DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp DWORD PTR _i$[ebp], 10 ; this condition is checked *before* each iteration
    jge SHORT $LN1@main ; if (i) is biggest or equals to 10, lets finish loop'
    mov ecx, DWORD PTR _i$[ebp] ; loop body:
    call printing_function(i)
    push ecx
    call _printing_function
    add esp, 4
    jmp SHORT $LN2@main ; jump to loop begin
$LN1@main: ; loop end
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
As we see, nothing special.

GCC 4.4.1 emits almost the same code, with one subtle difference:

Listing 14.2: GCC 4.4.1

```assembly
main proc near
var_20 = dword ptr -20h
var_4 = dword ptr -4
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 20h
mov [esp+20h+var_4], 2 ; (i) initializing
jmp short loc_8048476
loc_8048465:
mov eax, [esp+20h+var_4]
mov [esp+20h+var_20], eax
call printing_function
add [esp+20h+var_4], 1 ; (i) increment
loc_8048476:
cmp [esp+20h+var_4], 9
jle short loc_8048465 ; if i<=9, continue
loop
mov eax, 0
leave
retn
main endp
```

Now let's see what we get with optimization turned on (/Ox):

Listing 14.3: Optimizing MSVC

```assembly
_main PROC
push esi
mov esi, 2
$LL3@main:
push esi
call _printing_function
inc esi
add esp, 4
cmp esi, 10 ; 0000000Ah
jl SHORT $LL3@main
xor eax, eax
pop esi
```

253
What happens here is that space for the $i$ variable is not allocated in the local stack anymore, but uses an individual register for it, ESI. This is possible in such small functions where there aren't many local variables.

One very important thing is that the $f()$ function must not change the value in ESI. Our compiler is sure here. And if the compiler decides to use the ESI register in $f()$ too, its value would have to be saved at the function's prologue and restored at the function's epilogue, almost like in our listing: please note PUSH ESI/POP ESI at the function start and end.

Let's try GCC 4.4.1 with maximal optimization turned on (-O3 option):

Listing 14.4: Optimizing GCC 4.4.1

```
main proc near
var_10 = dword ptr -10h

push   ebp
mov    ebp, esp
and esp, 0FFFFFFF0h
sub esp, 10h
mov [esp+10h+var_10], 2
call printing_function
mov [esp+10h+var_10], 3
call printing_function
mov [esp+10h+var_10], 4
call printing_function
mov [esp+10h+var_10], 5
call printing_function
mov [esp+10h+var_10], 6
call printing_function
mov [esp+10h+var_10], 7
call printing_function
mov [esp+10h+var_10], 8
call printing_function
mov [esp+10h+var_10], 9
call printing_function
xor eax, eax
leave
retn
main endp
```

Huh, GCC just unwound our loop.
Loop unwinding has an advantage in the cases when there aren’t much iterations and we could cut some execution time by removing all loop support instructions. On the other side, the resulting code is obviously larger.

Big unrolled loops are not recommended in modern times, because bigger functions may require bigger cache footprint\(^1\).

OK, let’s increase the maximum value of the \(i\) variable to 100 and try again. GCC does:

Listing 14.5: GCC

```assembly
public main
main proc near
var_20 = dword ptr -20h
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
push ebx
mov ebx, 2 ; i=2
sub esp, 1Ch ; aligning label loc_80484D0 (loop body begin) by 16-byte border:
  nop
loc_80484D0: ; pass (i) as first argument to printing_function():
  mov [esp+20h+var_20], ebx
  add ebx, 1 ; i++
  call printing_function
  cmp ebx, 64h ; i==100?
  jnz short loc_80484D0 ; if not, continue
  add esp, 1Ch
  xor eax, eax ; return 0
  pop ebx
  mov esp, ebp
  pop ebp
main endp
```

It is quite similar to what MSVC 2010 with optimization (/Ox) produce, with the exception that the EBX register is allocated for the \(i\) variable. GCC is sure this register will not be modified inside of the \(f()\) function, and if it will, it will be saved

\(^1\)A very good article about it: [Dre07]. Another recommendations about loop unrolling from Intel are here: [Int14, p. 3.4.1.7].
at the function prologue and restored at epilogue, just like here in the `main()` function.
14.1.2 x86: OllyDbg

Let’s compile our example in MSVC 2010 with /Ox and /Ob0 options and load it into OllyDbg.

It seems that OllyDbg is able to detect simple loops and show them in square brackets, for convenience:

By tracing (F8 — step over) we see ESI incrementing. Here, for instance, \( ESI = i = 6 \):

9 is the last loop value. That’s why JL is not triggering after the increment, and the function will finish:
As we might see, it is not very convenient to trace manually in the debugger. That's a reason we will try tracer.

We open compiled example in IDA, find the address of the instruction PUSH ESI (passing the sole argument to f()), which is 0x401026 for this case and we run the tracer:

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX just sets a breakpoint at the address and tracer will then print the state of the registers.

In the tracer.log This is what we see:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
```
EAX=0x00000005  EBX=0x00000000  ECX=0x6f0a5617  EDX=0x000ee188
ESI=0x00000008  EDI=0x00333378  EBP=0x0024fbfc  ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005  EBX=0x00000000  ECX=0x6f0a5617  EDX=0x000ee188
ESI=0x00000009  EDI=0x00333378  EBP=0x0024fbfc  ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)

We see how the value of ESI register changes from 2 to 9.

Even more than that, the tracer can collect register values for all addresses within the function. This is called trace there. Every instruction gets traced, all interesting register values are recorded. Then, an IDA.idc-script is generated, that adds comments. So, in the IDA we've learned that the main() function address is 0x00401020 and we run:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF stands for set breakpoint on function.

As a result, we get the loops_2.exe.idc and loops_2.exe_clear.idc scripts.
CHAPTER 14. LOOPS

We load `loops_2.exe.idc` into IDA and see:

![IDA with .idc-script loaded](image)

We see that ESI can be from 2 to 9 at the start of the loop body, but from 3 to 0xA (10) after the increment. We can also see that `main()` is finishing with 0 in EAX.

**tracer** also generates `loops_2.exe.txt`, that contains information about how many times each instruction was executed and register values:

```
Listing 14.6: loops_2.exe.txt

0x401020 (.text+0x20), e= 1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e= 1 [MOV ESI, 2]
0x401026 (.text+0x26), e= 8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e= 8 [CALL 8D1000h] tracing nested maximum level (1) reached, skipping this CALL 8D1000h=0
  ↘ x8d1000
0x40102c (.text+0x2c), e= 8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e= 8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e= 8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e= 8 [JL 8D1026h] SF=false,true OF=false
  ↘ =false
0x401035 (.text+0x35), e= 1 [XOR EAX, EAX]
0x401037 (.text+0x37), e= 1 [POP ESI]
0x401038 (.text+0x38), e= 1 [RETN] EAX=0
```

We can use grep here.
### 14.1.4 ARM

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

```assembly
main
    STMFD SP!, {R4,LR}
    MOV R4, #2
    B loc_368
loc_35C ; CODE XREF: main+1C
    MOV R0, R4
    BL printing_function
    ADD R4, R4, #1
loc_368 ; CODE XREF: main+8
    CMP R4, #0xA
    BLT loc_35C
    MOV R0, #0
    LDMFD SP!, {R4,PC}
```

Iteration counter \( i \) is to be stored in the R4 register.

The “MOV R4, #2” instruction just initializes \( i \).

The “MOV R0, R4” and “BL printing_function” instructions compose the body of the loop, the first instruction preparing the argument for \( f() \) function and the second calling the function.

The “ADD R4, R4, #1” instruction just adds 1 to the \( i \) variable at each iteration.

“CMP R4, #0xA” compares \( i \) with 0xA (10). The next instruction BLT (Branch Less Than) jumps if \( i \) is less than 10.

Otherwise, 0 is to be written into R0 (since our function returns 0) and function execution finishes.

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

```assembly
_main
    PUSH {R4,LR}
    MOV S R4, #2
loc_132 ; CODE XREF: _main+E
    MOVS R0, R4
    BL printing_function
    ADDS R4, R4, #1
    CMP R4, #0xA
```

Iteration counter \( i \) is to be stored in the R4 register.

The “MOV R4, #2” instruction just initializes \( i \).

The “MOV R0, R4” and “BL printing_function” instructions compose the body of the loop, the first instruction preparing the argument for \( f() \) function and the second calling the function.

The “ADD R4, R4, #1” instruction just adds 1 to the \( i \) variable at each iteration.

“CMP R4, #0xA” compares \( i \) with 0xA (10). The next instruction BLT (Branch Less Than) jumps if \( i \) is less than 10.

Otherwise, 0 is to be written into R0 (since our function returns 0) and function execution finishes.
Practically the same.

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

```assembly
main

PUSH {R4,R7,LR}
MOVW R4, #0x1124 ; "%d\n"
MOVS R1, #2
MOVT.W R4, #0
ADD R7, SP, #4
ADD R4, PC
MOV R0, R4
BLX _printf
MOV R0, R4
MOVS R1, #3
BLX _printf
MOV R0, R4
MOVS R1, #4
BLX _printf
MOV R0, R4
MOVS R1, #5
BLX _printf
MOV R0, R4
MOVS R1, #6
BLX _printf
MOV R0, R4
MOVS R1, #7
BLX _printf
MOV R0, R4
MOVS R1, #8
BLX _printf
MOV R0, R4
MOVS R1, #9
BLX _printf
MOV R0, #0
POP {R4,R7,PC}
```

In fact, this was in my `f()` function:

```c
void printing_function(int i)
{
    printf("%d\n", i);
}
```
So, LLVM not just *unrolled* the loop, but also *inlined* my very simple function \( f() \), and inserted its body 8 times instead of calling it. This is possible when the function is so simple (like mine) and when it is not called too much (like here).

### ARM64: Optimizing GCC 4.9.1

Listing 14.7: Optimizing GCC 4.9.1

```assembly
printing_function:
    ; prepare second argument of printf():
    mov    w1, w0
    ; load address of the "f(%d)\n" string
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
    ; just branch here instead of branch with link and return:
    b      printf
main:
    ; save FP and LR in the local stack:
    stp    x29, x30, [sp, -32]!
    ; set up stack frame:
    add    x29, sp, 0
    ; save contents of X19 register in the local stack:
    str    x19, [sp,16]
    ; we will use W19 register as counter.
    ; set initial value of 2 to it:
    mov    w19, 2
    .L3:
    ; prepare first argument of printing_function():
    mov    w0, w19
    ; increment counter register.
    add    w19, w19, 1
    ; W0 here still holds value of counter value before increment.
    bl     printing_function
    ; is it end?
    cmp    w19, 10
    ; no, jump to the loop body begin:
    bne    .L3
    ; return 0
    mov    w0, 0
    ; restore contents of X19 register:
    ldr    x19, [sp,16]
    ; restore FP and LR values:
    ldp    x29, x30, [sp], 32
    ret
```

263
ARM64: Non-optimizing GCC 4.9.1

Listing 14.8: Non-optimizing GCC 4.9.1 -fno-inline

```assembly
printing_function:
; prepare second argument of printf():
    mov    w1, w0
; load address of the "f(%d)\n" string
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
; just branch here instead of branch with link and return:
    b      printf
main:
; save FP and LR in the local stack:
    stp    x29, x30, [sp, -32]!
; set up stack frame:
    add    x29, sp, 0
; save contents of X19 register in the local stack:
    str    x19, [sp,16]
; we will use W19 register as counter.
; set initial value of 2 to it:
    mov    w19, 2
.L3:
; prepare first argument of printing_function():
    mov    w0, w19
; increment counter register.
    add    w19, w19, 1
; W0 here still holds value of counter value before increment.
    bl     printing_function
; is it end?
    cmp    w19, 10
; no, jump to the loop body begin:
    bne    .L3
; return 0
    mov    w0, 0
; restore contents of X19 register:
    ldr    x19, [sp,16]
; restore FP and LR values:
    ldp    x29, x30, [sp], 32
    ret
.LC0:
    .string "f(%d)\n"
```
Listing 14.9: Non-optimizing GCC 4.4.5 (IDA)

main:

; IDA is not aware of variable names in local stack
; We gave them names manually:
  i            = -0x10
  saved_FP    = -8
  saved_RA    = -4

; function prologue:
  addiu  $sp, -0x28
  sw     $ra, 0x28+saved_RA($sp)
  sw     $fp, 0x28+saved_FP($sp)
  move   $fp, $sp

; initialize counter at 2 and store this value in local stack
  li      $v0, 2
  sw      $v0, 0x28+i($fp)

; pseudoinstruction. "BEQ $ZERO, $ZERO, loc_9C" there in fact:
  b      loc_9C
  or     $at, $zero ; branch delay slot, NOP

loc_80:  # CODE XREF: main+48
  ; load counter value from local stack and
  ; call printing_function():
  lw      $a0, 0x28+i($fp)
  jal     printing_function
  or      $at, $zero ; branch delay slot, NOP

  ; load counter, increment it, store it back:
  lw      $v0, 0x28+i($fp)
  or      $at, $zero ; NOP
  addiu   $v0, 1
  sw      $v0, 0x28+i($fp)

loc_9C:  # CODE XREF: main+18
  ; check counter, is it 10?
  lw      $v0, 0x28+i($fp)
  or      $at, $zero ; NOP
  slti    $v0, 0xA

  ; if it is less than 10, jump to loc_80 (loop body begin):
  bnez    $v0, loc_80
  or      $at, $zero ; branch delay slot, NOP

  ; finishing, return 0:
The instruction that's new to us is “B”. It is actually the pseudoinstruction (BEQ).

14.1.6 One more thing

In the generated code we can see: after initializing $i$, the body of the loop is not to be executed, as the condition for $i$ is checked first, and only after that loop body can be executed. And that is correct. Because, if the loop condition is not met at the beginning, the body of the loop must not be executed. This is possible in the following case:

```assembly
move $v0, $zero
; function epilogue:
move $sp, $fp
lw $ra, 0x28+saved_RA($sp)
lw $fp, 0x28+saved_FP($sp)
addiu $sp, 0x28
jr $ra
or $at, $zero ; branch delay slot, NOP
```

If `total_entries_to_process` is 0, the body of the loop must not be executed at all. This is why the condition checked before the execution.

However, an optimizing compiler may swap the condition check and loop body, if it sure that the situation described here is not possible (like in the case of our very simple example and Keil, Xcode (LLVM), MSVC in optimization mode).

14.2 Memory blocks copying routine

Real-world memory copy routines may copy 4 or 8 bytes at each iteration, use SIMD\(^2\), vectorization, etc. But for the sake of simplicity, this example is the simplest possible.

```c
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
```

\(^2\)Single instruction, multiple data
14.2.1 Straight-forward implementation

Listing 14.10: GCC 4.9 x64 optimized for size (-Os)

my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block

; initialize counter (i) at 0
xor eax, eax
.L2:
; all bytes copied? exit then:
    cmp rax, rdx
    je .L5
; load byte at RSI+i:
    mov cl, BYTE PTR [rsi+rax]
; store byte at RDI+i:
    mov BYTE PTR [rdi+rax], cl
    inc rax ; i++
    jmp .L2
.L5:
    ret

Listing 14.11: GCC 4.9 ARM64 optimized for size (-Os)

my_memcpy:
; X0 = destination address
; X1 = source address
; X2 = size of block

; initialize counter (i) at 0
    mov x3, 0
.L2:
; all bytes copied? exit then:
    cmp x3, x2
    beq .L5
; load byte at X1+i:
    ldrb w4, [x1,x3]
; store byte at X1+i:
    strb w4, [x0,x3]
    inc rax ; i++
    jmp .L2
Listing 14.12: Optimizing Keil 6/2013 (Thumb mode)

my_memcpy PROC
    ; R0 = destination address
    ; R1 = source address
    ; R2 = size of block
    PUSH   {r4,lr}
    ; initialize counter (i) at 0
    MOVS   r3,#0
    ; condition checked at the end of function, so jump there:
    B      |L0.12|
    |L0.6|
    ; load byte at R1+i:
    LDRB   r4,[r1,r3]
    ; store byte at R1+i:
    STRB   r4,[r0,r3]
    ; i++
    ADDS   r3,r3,#1
    |L0.12|
    ; i<size?
    CMP    r3,r2
    ; jump to the loop begin if its so:'
    BCC    |L0.6|
    POP    {r4,pc}
ENDP

14.2.2 ARM in ARM mode

Keil in ARM mode takes full advantage of conditional suffixes:


my_memcpy PROC
    ; R0 = destination address
    ; R1 = source address
    ; R2 = size of block
    ; initialize counter (i) at 0
    MOV     r3,#0
    |L0.4|
    ; all bytes copied?
    CMP     r3,r2
That’s why there is only one branch instruction instead of 2.

### 14.2.3 MIPS

Listing 14.14: GCC 4.4.5 optimized for size (-Os) (IDA)

```assembly
my_memcpy:
    ; jump to loop check part:
    b loc_14

    ; initialize counter (i) at 0
    ; it will always reside in \$v0:
    move \$v0, \$zero ; branch delay slot

loc_8:
    # CODE XREF: my_memcpy

|   +1C
|; load byte as unsigned at address in \$t0 to \$v1:
|    lbu \$v1, 0(\$t0)

|; increment counter (i):
|    addiu \$v0, 1

|; store byte at \$a3
|    sb \$v1, 0(\$a3)

loc_14:
    # CODE XREF: my_memcpy

|; check if counter (i) in \$v0 is still less then 3rd function
|  argument ("cnt" in \$a2):
|    sltu \$v1, \$v0, \$a2

|; form address of byte in source block:
|    addu \$t0, \$a1, \$v0
|  \$t0 = \$a1+$\$v0 = src+i

|; jump to loop body if counter sill less then "cnt":
```

The following block is executed only if "less than" condition, i.e., if R2<R3 or i<size.

- load byte at R1+i:
  ```
  LDRBCC r12,[r1,r3]
  ```

- store byte at R1+i:
  ```
  STRBCC r12,[r0,r3]
  ```

- i++
  ```
  ADDCC r3,r3,#1
  ```

The last instruction of the "conditional block".

- jump to loop begin if i<size
- do nothing otherwise (i.e., if i>=size)
  ```
  BCC |L0.4|
  ```

- return
  ```
  BX lr
  ```

ENDP
Here we have two new instructions: LBU ("Load Byte Unsigned") and SB ("Store Byte"). Just like in ARM, all MIPS registers are 32-bit wide, there are no byte-wide parts like in x86. So when dealing with single bytes, we have to allocate whole 32-bit registers for them. LBU loads a byte and clears all other bits ("Unsigned"). On the other hand, LB ("Load Byte") instruction sign-extends the loaded byte to a 32-bit value. SB just writes a byte from lowest 8 bits of register to memory.

### 14.2.4 Vectorization

Optimizing GCC can do much more on this example: 25.1.2 on page 568.

### 14.3 Conclusion

Rough skeleton of loop from 2 to 9 inclusive:

```assembly
Listing 14.15: x86
mov [counter], 2 ; initialization
jmp check
body:
    ; loop body
    ; do something here
    ; use counter variable in local stack
    add [counter], 1 ; increment
check:
    cmp [counter], 9
    jle body
```

The increment operation may be represented as 3 instructions in non-optimized code:

```assembly
Listing 14.16: x86
MOV [counter], 2 ; initialization
JMP check
body:
```
If the body of the loop is short, a whole register can be dedicated to the counter variable:

**Listing 14.17: x86**

```
MOV EBX, 2 ; initialization
JMP check

body:
; loop body
; do something here
; use counter in EBX, but do not modify it!
INC EBX ; increment

check:
  CMP EBX, 9
  JLE body
```

Some parts of the loop may be generated by compiler in different order:

**Listing 14.18: x86**

```
MOV [counter], 2 ; initialization
JMP label_check

label_increment:
  ADD [counter], 1 ; increment

label_check:
  CMP [counter], 10
  JGE exit

; loop body
; do something here
; use counter variable in local stack
JMP label_increment

exit:
```

Usually the condition is checked *before* loop body, but the compiler may rearrange it in a way that the condition is checked *after* loop body. This is done when the compiler is sure that the condition is always *true* on the first iteration, so the body of the loop is to be executed at least once:
Using the LOOP instruction. This is rare, compilers are not using it. When you see it, it's a sign that this piece of code is hand-written:

Listing 14.20: x86

; count from 10 to 1
MOV ECX, 10
body:
; loop body
; do something here
; use counter in ECX, but do not modify it!
LOOP body

ARM. The R4 register is dedicated to counter variable in this example:

Listing 14.21: ARM

MOV R4, 2 ; initialization
B check
body:
; loop body
; do something here
; use counter in R4, but do not modify it!
ADD R4,R4, #1 ; increment
check:
CMP R4, #10
BLT body

14.4 Exercises

- http://challenges.re/54
- http://challenges.re/55
- http://challenges.re/56
- http://challenges.re/57
Chapter 15

Simple C-strings processing

15.1 strlen()

Let’s talk about loops one more time. Often, the strlen() function\(^1\) is implemented using a while() statement. Here is how it is done in the MSVC standard libraries:

```c
int my_strlen (const char * str)
{
    const char *eos = str;
    while( *eos++ ) ;
    return( eos - str - 1 );
}

int main()
{
    // test
    return my_strlen("hello");
}
```

\(^1\)counting the characters in a string in the C language
Non-optimizing MSVC

Let's compile:

```assembly
_strlen PROC
    push ebp
    mov ebp, esp
    push ecx
    mov eax, DWORD PTR _str$[ebp] ; place pointer to string from "str"
mov DWORD PTR _eos$[ebp], eax ; place it to local variable "eos"
$LN2@strlen_:  
    mov ecx, DWORD PTR _eos$[ebp] ; ECX=eos
    ; take 8-bit byte from address in ECX and place it as 32-bit value to EDX with sign extension
    movsx edx, BYTE PTR [ecx]
    mov eax, DWORD PTR _eos$[ebp] ; EAX=eos
    add eax, 1 ; increment EAX
    mov DWORD PTR _eos$[ebp], eax ; place EAX back to "eos"
test edx, edx ; EDX is zero?
    je SHORT $LN1@strlen_ ; yes, then finish loop
    jmp SHORT $LN2@strlen_ ; continue loop
$LN1@strlen_:  
    ; here we calculate the difference between two pointers
    mov eax, DWORD PTR _eos$[ebp]
    sub eax, DWORD PTR _str$[ebp]
    sub eax, 1 ; subtract 1 and return result
    mov esp, ebp
    pop ebp
    ret 0
_strlen_ ENDP
```

We get two new instructions here: MOVSX and TEST.

The first one—MOVSX—takes a byte from an address in memory and stores the value in a 32-bit register. MOVSX stands for MOV with Sign-Extend. MOVSX sets the rest of the bits, from the 8th to the 31th, to 1 if the source byte is negative or to 0 if is positive.
And here is why.

By default, the \texttt{char} type is signed in MSVC and GCC. If we have two values of which one is \texttt{char} and the other is \texttt{int}, (\texttt{int} is signed too), and if the first value contain -2 (coded as \texttt{0xFE}) and we just copy this byte into the \texttt{int} container, it makes \texttt{0x000000FE}, and this from the point of signed \texttt{int} view is 254, but not -2. In signed \texttt{int}, -2 is coded as \texttt{0xFFFFFFFF}. So if we need to transfer \texttt{0xFE} from a variable of \texttt{char} type to \texttt{int}, we need to identify its sign and extend it. That is what \texttt{MOVSX} does.

You can also read about it in “\textit{Signed number representations}” section (30 on page 617).

It’s hard to say if the compiler needs to store a \texttt{char} variable in \texttt{EDX}, it could just take a 8-bit register part (for example \texttt{DL}). Apparently, the compiler’s register allocator works like that.

Then we see TEST EDX, EDX. You can read more about the TEST instruction in the section about bit fields (19 on page 411). Here this instruction just checks if the value in \texttt{EDX} equals to 0.

\textbf{Non-optimizing GCC}

Let’s try GCC 4.4.1:

```
public strlen
strlen proc near
eos = dword ptr -4
arg_0 = dword ptr 8

    push    ebp
    mov     ebp, esp
    sub     esp, 10h
    mov     eax, [ebp+arg_0]
    mov [ebp+eos], eax

loc_80483F0:
    mov     eax, [ebp+eos]
    movzx   eax, byte ptr [eax]
    test    al, al
    setnz   al
    add     [ebp+eos], 1
    test    al, al
    jnz     short loc_80483F0
    mov     edx, [ebp+eos]
    mov     eax, [ebp+arg_0]
    mov     ecx, edx
```
The result is almost the same as in MSVC, but here we see MOVZX instead of MOVSX. MOVZX stands for MOV with Zero-Extend. This instruction copies a 8-bit or 16-bit value into a 32-bit register and sets the rest of the bits to 0. In fact, this instruction is convenient only because it enable us to replace this instruction pair: xor eax, eax / mov al, [...].

On the other hand, it is obvious that the compiler could produce this code: mov al, byte ptr [eax] / test al, al—it is almost the same, however, the highest bits of the EAX register will contain random noise. But let’s think it is compiler’s drawback—it cannot produce more understandable code. Strictly speaking, the compiler is not obliged to emit understandable (to humans) code at all.

The next new instruction for us is SETNZ. Here, if AL doesn’t contain zero, test al, al— it is almost the same, however, the highest bits of the EAX register will contain random noise. But let’s think it is compiler’s drawback—it cannot produce more understandable code. Strictly speaking, the compiler is not obliged to emit understandable (to humans) code at all.

Optimizing MSVC

Now let’s compile all this in MSVC 2012, with optimizations turned on (/Ox):

```
Listing 15.1: Optimizing MSVC 2012 /Ob0

_str$ = 8 ; size = 4
_strlen PROC
    mov edx, DWORD PTR _str$[esp-4] ; EDX -> pointer to the string
    mov eax, edx ; move to EAX

$LL2@strlen:
    mov cl, BYTE PTR [eax] ; CL = *EAX
    inc eax ; EAX++
    test cl, cl ; CL==0?
    jne SHORT $LL2@strlen ; no, continue loop
    sub eax, edx ; calculate pointers difference
    dec eax ; decrement EAX
    ret 0
_strlen ENDP
```
Now it is all simpler. Needless to say, the compiler could use registers with such efficiency only in small functions with a few local variables.

INC/DEC – are increment/decrement instructions, in other words: add or subtract 1 to/from a variable.
We can try this (optimized) example in OllyDbg. Here is the first iteration:

![OllyDbg: first iteration start](image)

We see that OllyDbg found a loop and, for convenience, wrapped its instructions in brackets. By clicking the right button on EAX, we can choose “Follow in Dump” and the memory window scrolls to the right place. Here we can see the string “hello!” in memory. There is at least one zero byte after it and then random garbage. If OllyDbg sees a register with a valid address in it, that points to some string, it is shown as a string.
Let’s press F8 (step over) a few times, to get to the start of the body of the loop:

![OllyDbg: second iteration start](image)

**Figure 15.2: OllyDbg: second iteration start**

We see that EAX contains the address of the second character in the string.
We have to press F8 enough number of times in order to escape from the loop:

![OllyDbg screenshot](image)

Figure 15.3: OllyDbg: pointers difference to be calculated now

We see that EAX now contains the address of zero byte that’s right after the string. Meanwhile, EDX hasn’t changed, so it still pointing to the start of the string. The difference between these two addresses is being calculated now.
The SUB instruction just got executed:

![OllyDbg: EAX to be decremented now](image)

The difference of pointers is in the EAX register now—7. Indeed, the length of the “hello!” string is 6, but with the zero byte included—7. But strlen() must return the number of non-zero characters in the string. So the decrement executes and then the function returns.

### Optimizing GCC

Let's check GCC 4.4.1 with optimizations turned on (-O3 key):

```assembly
public strlen
strlen proc near
arg_0 = dword ptr 8

push ebp
mov ebp, esp
mov ecx, [ebp+arg_0]
mov eax, ecx

loc_8048418:
movzx edx, byte ptr [eax]
add eax, 1
test dl, dl
jnz short loc_8048418
not ecx
```

Figure 15.4: OllyDbg: EAX to be decremented now
Here GCC is almost the same as MSVC, except for the presence of MOVZX.

However, here MOVZX could be replaced with mov dl, byte ptr [eax].

Probably it is simpler for GCC’s code generator to remember the whole 32-bit EDX register is allocated for a char variable and it then can be sure that the highest bits has no any noise at any point.

After that we also see a new instruction—NOT. This instruction inverts all bits in the operand. You can say that it is a synonym to the XOR ECX, 0xffffffffh instruction. NOT and the following ADD calculate the pointer difference and subtract 1, just in a different way. At the start ECX, where the pointer to str is stored, gets inverted and 1 is subtracted from it.

See also: “Signed number representations” (30 on page 617).

In other words, at the end of the function just after loop body, these operations are executed:

```c
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

... and this is effectively equivalent to:

```c
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Why did GCC decide it would be better? Hard to guess. But perhaps the both variants are equivalent in efficiency.

15.1.2 ARM

32-bit ARM

Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)
CHAPTER 15. SIMPLE C-STRINGS PROCESSING

Listing 15.2: Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```assembly
_strlen

eos = -8
str = -4

SUB   SP, SP, #8 ; allocate 8 bytes for local variables
STR   R0, [SP,#8+str]
LDR   R0, [SP,#8+str]
STR   R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
  LDR   R0, [SP,#8+eos]
  ADD   R1, R0, #1
  STR   R1, [SP,#8+eos]
  LDRSB R0, [R0]
  CMP   R0, #0
  BEQ   loc_2CD4
  B     loc_2CB8

loc_2CD4 ; CODE XREF: _strlen+24
  LDR   R0, [SP,#8+eos]
  LDR   R1, [SP,#8+str]
  SUB   R0, R0, R1 ; R0=eos-str
  SUB   R0, R0, #1 ; R0=R0-1
  ADD   SP, SP, #8 ; free allocated 8 bytes
  BX    LR
```

Non-optimizing LLVM generates too much code, however, here we can see how the function works with local variables in the stack. There are only two local variables in our function: `eos` and `str`. In this listing, generated by IDA, we have manually renamed `var_8` and `var_4` to `eos` and `str`.

The first instructions just saves the input values into both `str` and `eos`.

The body of the loop starts at label `loc_2CB8`.

The first three instruction in the loop body (LDR, ADD, STR) load the value of `eos` into R0. Then the value is incremented and saved back into `eos`, which is located in the stack.

The next instruction, LDRSB R0, [R0] (“Load Register Signed Byte”), loads a byte from memory at the address stored in R0 and sign-extends it to 32-bit. This is similar to the MOVVSX instruction in x86. The compiler treats this byte as signed since the `char` type is signed according to the C standard. It was already written about it (15.1.1 on page 274) in this section, in relation to x86.

2The Keil compiler treats the `char` type as signed, just like MSVC and GCC.
It has to be noted that it is impossible to use 8- or 16-bit part of a 32-bit register in ARM separately of the whole register, as it is in x86. Apparently, it is because x86 has a huge history of backwards compatibility with its ancestors up to the 16-bit 8086 and even 8-bit 8080, but ARM was developed from scratch as a 32-bit RISC-processor. Consequently, in order to process separate bytes in ARM, one has to use 32-bit registers anyway.

So, \texttt{LDRSB} loads bytes from the string into \texttt{R0}, one by one. The following \texttt{CMP} and \texttt{BEQ} instructions check if the loaded byte is 0. If it’s not 0, control passes to the start of the body of the loop. And if it’s 0, the loop ends.

At the end of the function, the difference between \texttt{eos} and \texttt{str} is calculated, 1 is subtracted from it, and resulting value is returned via \texttt{R0}.

N.B. Registers were not saved in this function. That’s because in the ARM calling convention registers R0-R3 are “scratch registers”, intended for arguments passing, and we’re not required to restore their value when the function exits, since the calling function will not use them anymore. Consequently, they may be used for anything we want. No other registers are used here, so that is why we have nothing to save on the stack. Thus, control may be returned back to calling function by a simple jump (\texttt{BX}), to the address in the \texttt{LR} register.

\textbf{Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)}

\begin{center}
\textbf{Listing 15.3: Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)}
\end{center}

\begin{verbatim}
_strlen  MOV    R1, R0
loc_2DF6 LDRB.W R2, [R1],#1
           CMP    R2, #0
           BNE    loc_2DF6
           MVNS   R0, R0
           ADD    R0, R1
           BX     LR
\end{verbatim}

As optimizing LLVM concludes, \texttt{eos} and \texttt{str} do not need space on the stack, and can always be stored in registers. Before the start of the loop body, \texttt{str} is always in \texttt{R0}, and \texttt{eos}—in \texttt{R1}.

The \texttt{LDRB.W R2, [R1],#1} instruction loads a byte from the memory at the address stored in \texttt{R1}, to \texttt{R2}, sign-extending it to a 32-bit value, but not just that. #1 at the instruction’s end implies “Post-indexed addressing”, which means that 1 is to be added to \texttt{R1} after the byte is loaded.
Read more about it: 28.2 on page 607.

Then you can see CMP and BNE\(^3\) in the body of the loop, these instructions continue looping until 0 is found in the string.

MVNS\(^4\) (inverts all bits, like NOT in x86) and ADD instructions compute \(eos - str - 1\). In fact, these two instructions compute \(R0 = str + eos\), which is effectively equivalent to what was in the source code, and why it is so, was already explained here (15.1.1 on page 282).

Apparently, LLVM, just like GCC, concludes that this code can be shorter (or faster).

**Optimizing Keil 6/2013 (ARM mode)**

<table>
<thead>
<tr>
<th>Listing 15.4: Optimizing Keil 6/2013 (ARM mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_strlen</td>
</tr>
<tr>
<td>MOV R1, R0</td>
</tr>
<tr>
<td>loc_2C8</td>
</tr>
<tr>
<td>LDRB R2, [R1],#1</td>
</tr>
<tr>
<td>CMP R2, #0</td>
</tr>
<tr>
<td>SUBEQ R0, R1, R0</td>
</tr>
<tr>
<td>SUBEQ R0, R0, #1</td>
</tr>
<tr>
<td>BNE loc_2C8</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
</tbody>
</table>

Almost the same as what we saw before, with the exception that the \(str - eos - 1\) expression can be computed not at the function’s end, but right in the body of the loop. The -EQ suffix, as we may recall, implies that the instruction executes only if the operands in the CMP that was executed before were equal to each other. Thus, if R0 contains 0, both SUBEQ instructions executes and result is left in the R0 register.

**ARM64**

**Optimizing GCC (Linaro) 4.9**

```assembly
my_strlen:
    mov x1, x0 ; X1 is now temporary pointer (eos), acting like cursor
.L58:
```

\(^3\)(PowerPC, ARM) Branch if Not Equal

\(^4\)MoVe Not
The algorithm is the same as in 15.1.1 on page 276: find a zero byte, calculate the difference between the pointers and decrement the result by 1. Some comments were added by the author of this book.

The only thing worth noting is that our example is somewhat wrong: my_strlen() returns 32-bit int, while it has to return size_t or another 64-bit type.

The reason is that, theoretically, strlen() can be called for a huge blocks in memory that exceeds 4GB, so it must able to return a 64-bit value on 64-bit platforms. Because of my mistake, the last SUB instruction operates on a 32-bit part of register, while the penultimate SUB instruction works on full the 64-bit register (it calculates the difference between the pointers). It’s my mistake, it is better to leave it as is, as an example of how the code could look like in such case.

Non-optimizing GCC (Linaro) 4.9

```assembly
my_strlen:
    ; function prologue
    sub     sp, sp, #32
    ; first argument (str) will be stored in [sp,8]
    str     x0, [sp,8]
    ldr     x0, [sp,8]
    ; copy "str" to "eos" variable
    str     x0, [sp,24]
    nop
.L62:
    ; eos++
    ldr     x0, [sp,24]  ; load "eos" to X0
    add     x1, x0, 1   ; increment X0
    str     x1, [sp,24]  ; save X0 to "eos"
    ; load byte from memory at address in X0 to W0
    ldrb    w0, [x0]
    ; is it zero? (WZR is the 32-bit register always contain zero)
    cmp     w0, wzr
```

CHAPTER 15. SIMPLE C-STRINGS PROCESSING

; jump if not zero (Branch Not Equal)
    bne  .L62

; zero byte found. now calculate difference.
; load "eos" to X1
    ldr x1, [sp,24]
; load "str" to X0
    ldr x0, [sp,8]
; calculate difference
    sub x0, x1, x0
; decrement result
    sub w0, w0, #1
; function epilogue
    add sp, sp, 32
    ret

It's more verbose. The variables are often tossed here to and from memory (local stack). The same mistake here: the decrement operation happens on a 32-bit register part.

15.1.3 MIPS

Listing 15.5: Optimizing GCC 4.4.5 (IDA)

my_strlen:
; "eos" variable will always reside in $v1:
    move $v1, $a0

loc_4:
; load byte at address in "eos" into $a1:
    lb $a1, 0($v1)
    or $at, $zero ; load delay slot, NOP
; if loaded byte is not zero, jump to loc_4:
    bnez $a1, loc_4
; increment "eos" anyway:
    addiu $v1, 1 ; branch delay slot
; loop finished. invert "str" variable:
    nor $v0, $zero, $a0
; $v0=-str-1
    jr $ra
; return value = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
    addu $v0, $v1, $v0 ; branch delay slot

MIPS lacks a NOT instruction, but has NOR which is OR + NOT operation. This operation is widely used in digital electronics\(^5\), but isn't very popular in computer pro-

\(^5\)NOR is called “universal gate”. For example, the Apollo Guidance Computer used in the Apollo program, was built by only using 5600 NOR gates: [Eic11].
gramming. So, the NOT operation is implemented here as NOR DST, $ZERO, SRC.

From fundamentals 30 on page 617 we know that bitwise inverting a signed number is the same as changing its sign and subtracting 1 from the result. So what NOT does here is to take the value of str and transform it into $-str-1$. The addition operation that follows prepares result.
Chapter 16

Replacing arithmetic instructions to other ones

In the pursuit of optimization, one instruction may be replaced by another, or even with a group of instructions.

For example, ADD and SUB can replace each other: line 18 in listing 52.1.

For example, the LEA instruction is often used for simple arithmetic calculations: A.6.2 on page 1304.

16.1 Multiplication

16.1.1 Multiplication using addition

Here is a simple example:

Listing 16.1: Optimizing MSVC 2010

```c
unsigned int f(unsigned int a)
{
    return a*8;
}
```

Multiplication by 8 is replaced by 3 addition instructions, which do the same. Apparently, MSVC’s optimizer decided that this code can be faster.
16.1.2 Multiplication using shifting

Multiplication and division instructions by a numbers that's a power of 2 are often replaced by shift instructions.

```c
unsigned int f(unsigned int a)
{
    return a*4;
}
```

Listing 16.2: Non-optimizing MSVC 2010

Multiplication by 4 is just shifting the number to the left by 2 bits and inserting 2 zero bits at the right (as the last two bits). It is just like multiplying 3 by 100 —we need to just add two zeroes at the right.

That's how the shift left instruction works:
The added bits at right are always zeroes.

Multiplication by 4 in ARM:

Listing 16.3: Non-optimizing Keil 6/2013 (ARM mode)

```
f PROC
    LSL    r0,r0,#2
    BX     lr
ENDP
```

Multiplication by 4 in MIPS:

Listing 16.4: Optimizing GCC 4.4.5 (IDA)

```
jr      $ra
sll     $v0, $a0, 2 ; branch delay slot
```

SLL is “Shift Left Logical”.

### 16.1.3 Multiplication using shifting, subtracting, and adding

It’s still possible to get rid of the multiplication operation when you multiply by numbers like 7 or 17 again by using shifting. The mathematics used here is relatively easy.

#### 32-bit

```c
#include <stdint.h>

int f1(int a)
{
    return a*7;
};

int f2(int a)
{
    return a*28;
};

int f3(int a)
{
    return a*17;
};
```
Listing 16.5: Optimizing MSVC 2012

; a*7
_a$ = 8
_f1 PROC
    mov ecx, DWORD PTR _a$[esp-4]
    ; ECX=a
    lea eax, DWORD PTR [ecx*8]
    ; EAX=ECX*8
    sub eax, ecx
    ; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    ret 0
_f1 ENDP

; a*28
_a$ = 8
_f2 PROC
    mov ecx, DWORD PTR _a$[esp-4]
    ; ECX=a
    lea eax, DWORD PTR [ecx*8]
    ; EAX=ECX*8
    sub eax, ecx
    ; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    shl eax, 2
    ; EAX=EAX<<2=(a*7)*4=a*28
    ret 0
_f2 ENDP

; a*17
_a$ = 8
_f3 PROC
    mov eax, DWORD PTR _a$[esp-4]
    ; EAX=a
    shl eax, 4
    ; EAX=EAX<<4=EAX*16=a*16
    add eax, DWORD PTR _a$[esp-4]
    ; EAX=EAX+a=a*16+a=a*17
    ret 0
_f3 ENDP

ARM

Keil for ARM mode takes advantage of the second operand's shift modifiers:
Listing 16.6: Optimizing Keil 6/2013 (ARM mode)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>PROC</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>a*7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RSB</td>
<td>r0,r0,r0,LSL #3</td>
</tr>
<tr>
<td>;</td>
<td>R0=R0&lt;&lt;3-R0=R0<em>8-R0=a</em>8-a=a*7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BX</td>
<td>lr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDP</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>a*28</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RSB</td>
<td>r0,r0,r0,LSL #3</td>
</tr>
<tr>
<td>;</td>
<td>R0=R0&lt;&lt;3-R0=R0<em>8-R0=a</em>8-a=a*7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSL</td>
<td>r0,r0,#2</td>
</tr>
<tr>
<td>;</td>
<td>R0=R0&lt;&lt;2=R0<em>4=a</em>7<em>4=a</em>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BX</td>
<td>lr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDP</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>a*17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD</td>
<td>r0,r0,r0,LSL #4</td>
</tr>
<tr>
<td>;</td>
<td>R0=R0+R0&lt;&lt;4=R0+R0<em>16=R0</em>17=a*17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BX</td>
<td>lr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDP</td>
<td></td>
</tr>
</tbody>
</table>

But there are no such modifiers in Thumb mode. It also can’t optimize f2()

Listing 16.7: Optimizing Keil 6/2013 (Thumb mode)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>PROC</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>a*7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSLS</td>
<td>r1,r0,#3</td>
</tr>
<tr>
<td>;</td>
<td>R1=R0&lt;&lt;3=a&lt;&lt;3=a*8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SUBS</td>
<td>r0,r1,r0</td>
</tr>
<tr>
<td>;</td>
<td>R0=R1-R0=a<em>8-a=a</em>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BX</td>
<td>lr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDP</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>a*28</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOVS</td>
<td>r1,#0x1c ; 28</td>
</tr>
<tr>
<td>;</td>
<td>R1=28</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MULS</td>
<td>r0,r1,r0</td>
</tr>
<tr>
<td>;</td>
<td>R0=R1<em>R0=28</em>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BX</td>
<td>lr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDP</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>a*17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROC</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 16. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

MIPS

Listing 16.8: Optimizing GCC 4.4.5 (IDA)

```mips
_f1:
  sll $v0, $a0, 3
  jr $ra
  subu $v0, $a0; branch delay slot
  $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2:
  sll $v0, $a0, 5
  sll $a0, 2
  jr $ra
  subu $v0, $a0; branch delay slot
  $v0 = $a0*32-$a0*4 = $a0*28

_f3:
  sll $v0, $a0, 4
  jr $ra
  addu $v0, $a0; branch delay slot
  $v0 = $a0*16+$a0 = $a0*17
```

64-bit

```c
#include <stdint.h>

int64_t f1(int64_t a)
{
    return a*7;
};
```
int64_t f2(int64_t a)
{
    return a*28;
};

int64_t f3(int64_t a)
{
    return a*17;
};

ARM64

Listing 16.9: Optimizing MSVC 2012

; a*7
f1:
    lea rax, [0+rdi*8]
; RAX=RDI*8=a*8
    sub rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
    ret

; a*28
f2:
    lea rax, [0+rdi*4]
; RAX=RDI*4=a*4
    sal rdi, 5
; RDI=RDI<<5=RDI*32=a*32
    sub rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
    mov rax, rdi
    ret

; a*17
f3:
    mov rax, rdi
    sal rax, 4
; RAX=RAX<<4=a*16
    add rax, rdi
; RAX=a*16+a=a*17
    ret

x64
CHAPTER 16. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

GCC 4.9 for ARM64 is also terse, thanks to the shift modifiers:

**Listing 16.10: Optimizing GCC (Linaro) 4.9 ARM64**

```assembly
; a*7
f1:
    lsl x1, x0, 3 ; X1=X0<<3=X0*8=a*8
    sub x0, x1, x0 ; X0=X1-X0=a*8-a=a*7
    ret

; a*28
f2:
    lsl x1, x0, 5 ; X1=X0<<5=a*32
    sub x0, x1, x0, lsl 2 ; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
    ret

; a*17
f3:
    add x0, x0, x0, lsl 4 ; X0=X0+X0<<4=a+a*16=a*17
    ret
```

16.2 Division

16.2.1 Division using shifts

Example of division by 4:

```c
unsigned int f(unsigned int a)
{
    return a/4;
};
```

We get (MSVC 2010):

**Listing 16.11: MSVC 2010**

```assembly
    _a$ = 8 ; size ↘
    ↓ = 4
    _f PROC
        mov   eax, DWORD PTR _a$[esp-4]
        shr   eax, 2
    _f ENDP
```

296
The SHR (Shift Right) instruction in this example is shifting a number by 2 bits to the right. The two freed bits at left (e.g., two most significant bits) are set to zero. The two least significant bits are dropped. In fact, these two dropped bits are the division operation remainder.

The SHR instruction works just like SHL, but in the other direction.

It is easy to understand if you imagine the number 23 in the decimal numeral system. 23 can be easily divided by 10 just by dropping last digit (3—division remainder). 2 is left after the operation as a quotient.

So the remainder is dropped, but that’s OK, we work on integer values anyway, these are not a real numbers!

Division by 4 in ARM:

Listing 16.12: Non-optimizing Keil 6/2013 (ARM mode)

```assembly
f PROC
  LSR r0, r0, #2
  BX lr
ENDP
```

Division by 4 in MIPS:

Listing 16.13: Optimizing GCC 4.4.5 (IDA)

```assembly
jr $ra
srl $v0, $a0, 2 ; branch delay slot
```

The SRL instruction is “Shift Right Logical”.

16.3 Exercise

- [http://challenges.re/59](http://challenges.re/59)
Chapter 17

Floating-point unit

The FPU is a device within the main CPU, specially designed to deal with floating point numbers. It was called “coprocessor” in the past and it stays somewhat aside of the main CPU.

17.1 IEEE 754

A number in the IEEE 754 format consists of a sign, a significand (also called fraction) and an exponent.

17.2 x86

It is worth looking into stack machines\(^1\) or learning the basics of the Forth language\(^2\), before studying the FPU in x86.

It is interesting to know that in the past (before the 80486 CPU) the coprocessor was a separate chip and it was not always pre-installed on the motherboard. It was possible to buy it separately and install it\(^3\). Starting with the 80486 DX CPU, the FPU is integrated in the CPU.

\(^1\)wikipedia.org/wiki/Stack_machine  
\(^2\)wikipedia.org/wiki/Forth_(programming_language)  
\(^3\)For example, John Carmack used fixed-point arithmetic (wikipedia.org/wiki/Fixed-point_arithmetic) values in his Doom video game, stored in 32-bit GPR registers (16 bit for integral part and another 16 bit for fractional part), so Doom could work on 32-bit computers without FPU, i.e., 80386 and 80486 SX.
The FWAIT instruction reminds us of that fact—it switches the CPU to a waiting state, so it can wait until the FPU is done with its work. Another rudiment is the fact that the FPU instruction opcodes start with the so called “escape”-opcodes (D8..DF), i.e., opcodes passed to a separate coprocessor.

The FPU has a stack capable to holding 8 80-bit registers, and each register can hold a number in the IEEE 754\(^4\) format. They are ST(0)..ST(7). For brevity, IDA and OllyDbg show ST(0) as ST, which is represented in some textbooks and manuals as “Stack Top”.

### 17.3 ARM, MIPS, x86/x64 SIMD

In ARM and MIPS the FPU is not a stack, but a set of registers. The same ideology is used in the SIMD extensions of x86/x64 CPUs.

### 17.4 C/C++

The standard C/C++ languages offer at least two floating number types, \textit{float} (single-precision\(^5\), 32 bits) \(^6\) and \textit{double} (double-precision\(^7\), 64 bits).

GCC also supports the \textit{long double} type (extended precision\(^8\), 80 bit), which MSVC doesn’t.

The \textit{float} type requires the same number of bits as the \textit{int} type in 32-bit environments, but the number representation is completely different.

### 17.5 Simple example

Let’s consider this simple example:

```c
#include <stdio.h>

double f (double a, double b) {
    return a/3.14 + b*4.1;
}
```

---

\(^4\)wikipedia.org/wiki/IEEE_floating_point
\(^5\)wikipedia.org/wiki/Single-precision_floating-point_format
\(^6\)the single precision floating point number format is also addressed in the Working with the float type as with a structure (21.6.2 on page 510) section
\(^7\)wikipedia.org/wiki/Double-precision_floating-point_format
\(^8\)wikipedia.org/wiki/Extended_precision
int main()
{
    printf (":f\n", f(1.2, 3.4));
};

17.5.1 x86

MSVC

Compile it in MSVC 2010:

Listing 17.1: MSVC 2010: f()

CONST SEGMENT
__real@4010666666666666 DQ 04010666666666666r ; 4.1
CONST ENDS
CONST SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14
CONST ENDS
_TEXT SEGMENT
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f PROC
    push ebp
    mov ebp, esp
    fld QWORD PTR _a$[ebp]
        ; current stack state: ST(0) = _a

    fdiv QWORD PTR __real@40091eb851eb851f
        ; current stack state: ST(0) = result of _a divided by 3.14

   fld QWORD PTR _b$[ebp]
        ; current stack state: ST(0) = _b; ST(1) = result of _a divided by 3.14

    fmul QWORD PTR __real@4010666666666666
        ; current stack state:
        ; ST(0) = result of _b * 4.1;
        ; ST(1) = result of _a divided by 3.14
faddp  ST(1), ST(0)

; current stack state: ST(0) = result of addition

    pop    ebp
    ret  0

_f  ENDP

FLD takes 8 bytes from stack and loads the number into the ST(0) register, automatically converting it into the internal 80-bit format (extended precision).

FDIV divides the value in ST(0) by the number stored at address __real@40091eb851eb851f — the value 3.14 is encoded there. The assembly syntax doesn’t support floating point numbers, so what we see here is the hexadecimal representation of 3.14 in 64-bit IEEE 754 format.

After the execution of FDIV ST(0) holds the quotient.

By the way, there is also the FDIVP instruction, which divides ST(1) by ST(0), popping both these values from stack and then pushing the result. If you know the Forth language⁹, you can quickly understand that this is a stack machine¹⁰.

The subsequent FLD instruction pushes the value of \( b \) into the stack.

After that, the quotient is placed in ST(1), and ST(0) has the value of \( b \).

The next FMUL instruction does multiplication: \( b \) from ST(0) is multiplied by the value at __real@4010666666666666 (the number 4.1 is there) and leaves the result in the ST(0) register.

The last FADDP instruction adds the two values at top of stack, storing the result in ST(1) and then popping the value of ST(0), thereby leaving the result at the top of the stack, in ST(0).

The function must return its result in the ST(0) register, so there are no any other instructions except the function epilogue after FADDP.

---


2 pairs of 32-bit words are marked by red in the stack. Each pair is a double-number in IEEE 754 format and is passed from `main()`. We see how the first FLD loads a value (1.2) from the stack and puts it into ST(0):

![Figure 17.1: OllyDbg: first FLD executed](image)

Because of unavoidable conversion errors from 64-bit IEEE 754 floating point to 80-bit (used internally in the FPU), here we see 1.999..., which is close to 1.2. EIP now points to the next instruction (FDIV), which loads a double-number (a constant) from memory. For convenience, OllyDbg shows its value: 3.14
Let's trace further. FDIV was executed, now ST(0) contains 0.382... (quotient):

Figure 17.2: OllyDbg: FDIV executed
Third step: the next FLD was executed, loading 3.4 into ST(0) (here we see the approximate value 3.39999...):

At the same time, quotient is pushed into ST(1). Right now, EIP points to the next instruction: FMUL. It loads the constant 4.1 from memory, which OllyDbg shows.
Next: FMUL was executed, so now the product is in ST(0):

Figure 17.4: OllyDbg: FMUL executed
Next: FADDP was executed, now the result of the addition is in \texttt{ST(0)}, and \texttt{ST(1)} is cleared:

![OllyDbg: FADDP executed](image)

The result is left in \texttt{ST(0)}, because the function returns its value in \texttt{ST(0)}. \texttt{main()} takes this value from the register later.

We also see something unusual: the 13.93\ldots value is now located in \texttt{ST(7)}. Why?

As we have read some time before in this book, the FPU registers are a stack: \textbf{17.2 on page 299}. But this is a simplification. Imagine if it was implemented \textit{in hardware} as it’s described, then all 7 register’s contents must be moved (or copied) to adjacent registers during pushing and popping, and that’s a lot of work. In reality, the FPU has just 8 registers and a pointer (called TOP) which contains a register number, which is the current “top of stack”. When a value is pushed to the stack, TOP is pointed to the next available register, and then a value is written to that register. The procedure is reversed if a value is popped, however, the register which was freed is not cleared (it could possibly be cleared, but this is more work which can degrade performance). So that’s what we see here. It can be said that FADDP saved the sum in the stack, and then popped one element. But in fact, this instruction saved the sum and then shifted TOP. More precisely, the registers of the FPU are a circular buffer.
GCC 4.4.1 (with -O3 option) emits the same code, just slightly different:

Listing 17.2: Optimizing GCC 4.4.1

```assembly
public f
f proc near
arg_0 = qword ptr 8
arg_8 = qword ptr 10h

push ebp
fld ds:dbl_8048608 ; 3.14
; stack state now: ST(0) = 3.14
mov ebp, esp
fdivr [ebp+arg_0]
; stack state now: ST(0) = result of division
fld ds:dbl_8048610 ; 4.1
; stack state now: ST(0) = 4.1, ST(1) = result of division
fmul [ebp+arg_8]
; stack state now: ST(0) = result of multiplication, ST(1) = result of division
pop ebp
faddp st(1), st
; stack state now: ST(0) = result of addition
ret
f endp
```

The difference is that, first of all, 3.14 is pushed to the stack (into ST(0)), and then the value in arg_0 is divided by the value in ST(0).

FDIVR stands for Reverse Divide — to divide with divisor and dividend swapped with each other. There is no likewise instruction for multiplication since it is a commutative operation, so we just have FMUL without its -R counterpart.

FADDP adds the two values but also pops one value from the stack. After that operation, ST(0) holds the sum.
17.5.2 ARM: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Until ARM got standardized floating point support, several processor manufacturers added their own instructions extensions. Then, VFP (Vector Floating Point) was standardized.

One important difference from x86 is that in ARM, there is no stack, you work just with registers.

Listing 17.3: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
VLDR D16, =3.14
VMOV D17, R0, R1 ; load "a"
VMOV D18, R2, R3 ; load "b"
VDIV.F64 D16, D17, D16 ; a/3.14
VLDR D17, =4.1
VMUL.F64 D17, D18, D17 ; b*4.1
VADD.F64 D16, D17, D16 ; +
VMOV R0, R1, D16
BX LR
```

So, we see here new some registers used, with D prefix. These are 64-bit registers, there are 32 of them, and they can be used both for floating-point numbers (double) but also for SIMD (it is called NEON here in ARM). There are also 32 32-bit S-registers, intended to be used for single precision floating pointer numbers (float). It is easy to remember: D-registers are for double precision numbers, while S-registers—for single precision numbers. More about it: B.3.3 on page 1323.

Both constants (3.14 and 4.1) are stored in memory in IEEE 754 format.

VLDR and VMOV, as it can be easily deduced, are analogous to the LDR and MOV instructions, but they work with D-registers. It has to be noted that these instructions, just like the D-registers, are intended not only for floating point numbers, but can be also used for SIMD (NEON) operations and this will also be shown soon.

The arguments are passed to the function in a common way, via the R-registers, however each number that has double precision has a size of 64 bits, so two R-registers are needed to pass each one.

VMOV D17, R0, R1 at the start, composes two 32-bit values from R0 and R1 into one 64-bit value and saves it to D17.

VMOV R0, R1, D16 is the inverse operation: what was in D16 is split in two registers, R0 and R1, because a double-precision number that needs 64 bits for storage, is returned in R0 and R1.
VDIV, VMUL and VADD, are instruction for processing floating point numbers that compute quotient, product and sum, respectively.

The code for Thumb-2 is same.

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

```assembly
f
PUSH {R3-R7,LR}
MOV S R7, R2
MOV S R4, R3
MOV S R5, R0
MOV S R6, R1
LDR R2, =0x66666666 ; 4.1
LDR R3, =0x40106666
MOV S R0, R7
MOV S R1, R4
BL __aeabi_dmul
MOV S R7, R0
MOV S R4, R1
LDR R2, =0x51EB851F ; 3.14
LDR R3, =0x40091EB8
MOV S R0, R5
MOV S R1, R6
BL __aeabi_ddiv
MOV S R2, R7
MOV S R3, R4
BL __aeabi_dadd
POP {R3-R7,PC}
```

; 4.1 in IEEE 754 form:
dword_364 DCD 0x66666666 ; DATA XREF: f+A
dword_368 DCD 0x40106666 ; DATA XREF: f+C
; 3.14 in IEEE 754 form:
dword_36C DCD 0x51EB851F ; DATA XREF: f+1A
dword_370 DCD 0x40091EB8 ; DATA XREF: f+1C

Keil generated code for a processor without FPU or NEON support. The double-precision floating-point numbers are passed via generic R-registers, and instead of FPU-instructions, service library functions are called (like __aeabi_dmul, __aeabi_ddiv, __aeabi_dadd) which emulate multiplication, division and addition for floating-point numbers. Of course, that is slower than FPU-coprocessor, but still better than nothing.

By the way, similar FPU-emulating libraries were very popular in the x86 world when coprocessors were rare and expensive, and were installed only on expensive
The FPU-coprocessor emulation is called *soft float* or *armel (emulation)* in the ARM world, while using the coprocessor's FPU-instructions is called *hard float* or *armhf*.

### 17.5.4 ARM64: Optimizing GCC (Linaro) 4.9

Very compact code:

```assembly
Listing 17.4: Optimizing GCC (Linaro) 4.9

f:
; D0 = a, D1 = b
    ldr d2, .LC25 ; 3.14
; D2 = 3.14
    fdiv d0, d0, d2
; D0 = D0/D2 = a/3.14
    ldr d2, .LC26 ; 4.1
; D2 = 4.1
    fmul d0, d1, d2, d0
; D0 = D1*D2+D0 = b*4.1+a/3.14
    ret

; constants in IEEE 754 format:
.LC25:
    .word 1374389535 ; 3.14
    .word 1074339512
.LC26:
    .word 1717986918 ; 4.1
    .word 1074816614
```

### 17.5.5 ARM64: Non-optimizing GCC (Linaro) 4.9

Listing 17.5: Non-optimizing GCC (Linaro) 4.9

```assembly
Listing 17.5: Non-optimizing GCC (Linaro) 4.9

f:
    sub sp, sp, #16
    str d0, [sp,8] ; save "a" in Register Save
    str d1, [sp] ; save "b" in Register Save
    ldr x1, [sp,8]
    ; X1 = a
    ldr x0, .LC25
    ; X0 = 3.14
```
Non-optimizing GCC is more verbose. There is a lot of unnecessary value shuffling, including some clearly redundant code (the last two FMOV instructions). Probably, GCC 4.9 is not yet good in generating ARM64 code. What is worth noting is that ARM64 has 64-bit registers, and the D-registers are 64-bit ones as well. So the compiler is free to save values of type `double` in GPRs instead of the local stack. This isn’t possible on 32-bit CPUs.
And again, as an exercise, you can try to optimize this function manually, without introducing new instructions like FMADD.

### 17.5.6 MIPS

MIPS can support several coprocessors (up to 4), the zeroth of which is a special control coprocessor, and first coprocessor is the FPU.

As in ARM, the MIPS coprocessor is not a stack machine, it has 32 32-bit registers ($F0-$F31): C.1.2 on page 1327. When one needs to work with 64-bit double values, a pair of 32-bit F-registers is used.

Listing 17.6: Optimizing GCC 4.4.5 (IDA)

```assembly
f:
 ; $f12-$f13=A
 ; $f14-$f15=B
     lui $v0, (dword_C4 >> 16) ; ?
 ; load low 32-bit part of 3.14 constant to $f0:
     lwc1 $f0, dword_BC
     or $at, $zero ; load delay slot
 \incerely, NOP
 ; load high 32-bit part of 3.14 constant to $f1:
     lwc1 $f1, $LC0
     lui $v0, ($LC1 >> 16) ; ?
 ; A in $f12-$f13, 3.14 constant in $f0-$f1, do division:
     div.d $f0, $f12, $f0
 ; $f0-$f1=A/3.14
 ; load low 32-bit part of 4.1 to $f2:
     lwc1 $f2, dword_C4
     or $at, $zero ; load delay slot
 \incerely, NOP
 ; load high 32-bit part of 4.1 to $f3:
     lwc1 $f3, $LC1
     or $at, $zero ; load delay slot
 \incerely, NOP
 ; B in $f14-$f15, 4.1 constant in $f2-$f3, do multiplication:
     mul.d $f2, $f14, $f2
 ; $f2-$f3=B*4.1
     jr $ra
 ; sum 64-bit parts and leave result in $f0-$f1:
     add.d $f0, $f2 ; branch delay
 \enerima slot, NOP

.rodata.cst8:000000B8 $LC0:    .word 0x40091EB8
 \enerima # DATA XREF: f+C
```

312
The new instructions here are:

- **LWC1** loads a 32-bit word into a register of the first coprocessor (hence “1” in instruction name). A pair of LWC1 instructions may be combined into a L.D pseudoinstruction.

- **DIV.D, MUL.D, ADD.D** do division, multiplication, and addition respectively (“D” in the suffix stands for double precision, “S” stands for single precision)

There is also a weird compiler anomaly: the LUI instructions that we’ve marked with a question mark. It’s hard for me to understand why load a part of a 64-bit constant of *double* type into the $V0 register. These instruction have no effect. If someone knows more about it, please drop an email to author\(^\text{11}\).

## 17.6 Passing floating point numbers via arguments

```c
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));
    return 0;
}
```

### 17.6.1 x86

Let’s see what we get in (MSVC 2010):

```
CONST SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r ; 1.54
```

\(^{11}\)dennis(a)yurichev.com
CONST ENDS

_main PROC
    push ebp
    mov ebp, esp
    sub esp, 8 ; allocate space for the first variable
    fld QWORD PTR __real@3ff8a3d70a3d70a4
    fstp QWORD PTR [esp]
    sub esp, 8 ; allocate space for the second variable
    fld QWORD PTR __real@40400147ae147ae1
    fstp QWORD PTR [esp]
    call _pow
    add esp, 8 ; "return back" place of one variable.
    ; in local stack here 8 bytes still reserved for us.
    ; result now in ST(0)
    fstp QWORD PTR [esp] ; move result from ST(0) to local
    ; stack for printf()
    push OFFSET $SG2651
    call _printf
    add esp, 12
    xor eax, eax
    pop ebp
    ret 0

_main ENDP

FLD and FSTP move variables between the data segment and the FPU stack. pow()\(^{12}\) takes both values from the stack of the FPU and returns its result in the ST(0) register. printf() takes 8 bytes from the local stack and interprets them as double type variable.

By the way, a pair of MOV instructions could be used here for moving values from the memory into the stack, because the values in memory are stored in IEEE 754 format, and pow() also takes them in this format, so no conversion is necessary. That's how it's done in the next example, for ARM: 17.6.2.

### 17.6.2 ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

_main

var_C = -0xC

PUSH {R7,LR}

\(^{12}\) a standard C function, raises a number to the given power (exponentiation)
As it was mentioned before, 64-bit floating pointer numbers are passed in R-registers pairs. This code is a bit redundant (certainly because optimization is turned off), since it is possible to load values into the R-registers directly without touching the D-registers.

So, as we see, the \_pow function receives its first argument in R0 and R1, and its second one in R2 and R3. The function leaves its result in R0 and R1. The result of \_pow is moved into D16, then in the R1 and R2 pair, from where printf() takes the resulting number.

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

```asm
STMFD SP!, {R4-R6,LR}
LDR R2, =0xA3D70A4 ; y
LDR R3, =0x3FF8A3D7
LDR R0, =0xAE147AE1 ; x
LDR R1, =0x40400147
BL pow
MOV R4, R0
MOV R2, R4
MOV R3, R1
ADR R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n\n```

As it was mentioned before, 64-bit floating pointer numbers are passed in R-registers pairs. This code is a bit redundant (certainly because optimization is turned off), since it is possible to load values into the R-registers directly without touching the D-registers.

So, as we see, the _pow function receives its first argument in R0 and R1, and its second one in R2 and R3. The function leaves its result in R0 and R1. The result of _pow is moved into D16, then in the R1 and R2 pair, from where printf() takes the resulting number.

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

```asm
STMFD SP!, {R4-R6,LR}
LDR R2, =0xA3D70A4 ; y
LDR R3, =0x3FF8A3D7
LDR R0, =0xAE147AE1 ; x
LDR R1, =0x40400147
BL pow
MOV R4, R0
MOV R2, R4
MOV R3, R1
ADR R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n\n```
D-registers are not used here, just R-register pairs.

### 17.6.4 ARM64 + Optimizing GCC (Linaro) 4.9

Listing 17.8: Optimizing GCC (Linaro) 4.9

```
f:
    stp    x29, x30, [sp, -16]!
    add    x29, sp, 0
    ldr    d1, .LC1 ; load 1.54 into D1
    ldr    d0, .LC0 ; load 32.01 into D0
    bl     pow
    ; result of pow() in D0
    adrp   x0, .LC2
    add    x0, x0, :lo12:.LC2
    bl     printf
    mov    w0, 0
    ldp    x29, x30, [sp], 16
    ret

.LC0:
    ; 32.01 in IEEE 754 format
    .word   -1374389535
    .word   1077936455

.LC1:
    ; 1.54 in IEEE 754 format
    .word   171798692
    .word   1073259479

.LC2:
    .string  "32.01 ^ 1.54 = %lf\n"
```

The constants are loaded into D0 and D1: pow() takes them from there. The result will be in D0 after the execution of pow(). It is to be passed to printf() without any modification and moving, because printf() takes arguments of integral types and pointers from X-registers, and floating point arguments from D-registers.
Listing 17.9: Optimizing GCC 4.4.5 (IDA)

main:

```
var_10 = -0x10
var_4  = -4

; function prologue:
    lui   $gp, (dword_9C >> 16)
    addiu $sp, -0x20
    la    $gp, (__gnu_local_gp & 0xFFFF)
    sw    $ra, 0x20+var_4($sp)
    sw    $gp, 0x20+var_10($sp)
    lui   $v0, (dword_A4 >> 16); ?

; load low 32-bit part of 32.01:
    lwc1  $f12, dword_9C

; load address of pow() function:
    lw    $t9, (pow & 0xFFFF)($gp)

; load high 32-bit part of 32.01:
    lwc1  $f13, $LC0
    lui   $v0, ($LC1 >> 16); ?

; load low 32-bit part of 1.54:
    lwc1  $f14, dword_A4
    or    $at, $zero; load delay slot, NOP

; load high 32-bit part of 1.54:
    lwc1  $f15, $LC1

; call pow():
    jalr  $t9
    or    $at, $zero; branch delay slot, NOP
    lw    $gp, 0x20+var_10($sp)

; copy result from $f0 and $f1 to $a3 and $a2:
    mfc1  $a3, $f0
    lw    $t9, (printf & 0xFFFF)($gp)
    mfc1  $a2, $f1

; call printf():
    lui   $a0, ($LC2 >> 16)  # "32.01 ^ 1.54 = %f"
    la    $a0, ($LC2 & 0xFFFF)  # "32.01 ^ 1.54 = %f"

; function epilogue:
    lw    $ra, 0x20+var_4($sp)

; return 0:
    move  $v0, $zero
    jr     $ra
    addiu  $sp, 0x20
```
And again, we see here LUI loading a 32-bit part of a double number into $V0. And again, it's hard to comprehend why.

The new instruction for us here is MFC1 ("Move From Coprocessor 1"). The FPU is coprocessor number 1, hence "1" in the instruction name. This instruction transfers values from the coprocessor's registers to the registers of the CPU (GPR). So in the end the result from pow() is moved to registers $A3 and $A2, and printf() takes a 64-bit double value from this register pair.

### 17.7 Comparison example

Let's try this:

```c
#include <stdio.h>

double d_max (double a, double b) {
    if (a>b)
        return a;
    return b;
};

int main() {
    printf ("%f\n", d_max (1.2, 3.4));
}
```
Despite the simplicity of the function, it will be harder to understand how it works.

### 17.7.1 x86

#### Non-optimizing MSVC

MSVC 2010 generates the following:

```assembly
PUBLIC _d_max
_TEXT SEGMENT
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_d_max PROC
    push ebp
    mov ebp, esp
    fld QWORD PTR _b$[ebp]
    ; current stack state: ST(0) = _b
    ; compare _b (ST(0)) and _a, and pop register
    fcomp QWORD PTR _a$[ebp]
    ; stack is empty here
    fnstsw ax
    test ah, 5
    jmp SHORT $LN1@d_max
    ; we are here only if a>b
    fld QWORD PTR _a$[ebp]
    jmp SHORT $LN2@d_max
$LN1@d_max:
    fld QWORD PTR _b$[ebp]
$LN2@d_max:
    pop ebp
    ret 0
_d_max ENDP
```

So, FLD loads _b into ST(0).
FCOMP compares the value in ST(0) with what is in _a and sets C3/C2/C0 bits in FPU status word register, accordingly. This is a 16-bit register that reflects the current state of the FPU.

After the bits are set, the FCOMP instruction also pops one variable from the stack. This is what distinguishes it from FCOM, which is just compares values, leaving the stack in the same state.

Unfortunately, CPUs before Intel P6 don’t have any conditional jumps instructions which check the C3/C2/C0 bits. Probably, it is a matter of history (remember: FPU was separate chip in past).

Modern CPU starting at Intel P6 have FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions —which do the same, but modify the ZF/PF/CF CPU flags.

The FNSTSW instruction copies FPU the status word register to AX. C3/C2/C0 bits are placed at positions 14/10/8, they are at the same positions in the AX register and all they are placed in the high part of AX —AH.

- If \( b > a \) in our example, then C3/C2/C0 bits are to be set as following: 0, 0, 0.
- If \( a > b \), then the bits are: 0, 0, 1.
- If \( a = b \), then the bits are: 1, 0, 0.
- If the result is unordered (in case of error), then the set bits are: 1, 1, 1.

This is how C3/C2/C0 bits are located in the AX register:

\[
\begin{array}{c c c c}
14 & 10 & 9 & 8 \\
\hline
C3 & C2 & C1 & C0
\end{array}
\]

This is how C3/C2/C0 bits are located in the AH register:

\[
\begin{array}{c c c c c}
6 & 2 & 1 & 0 \\
\hline
C3 & C2 & C1 & C0
\end{array}
\]

After the execution of \texttt{test ah, 5}^14, only C0 and C2 bits (on 0 and 2 position) are considered, all other bits are just ignored.

Now let’s talk about the \textit{parity flag}, another notable historical rudiment.

This flag is set to 1 if the number of ones in the result of the last calculation is even, and to 0 if it is odd.

Let’s look into Wikipedia:\textsuperscript{15}:

---

\textsuperscript{13}Intel P6 is Pentium Pro, Pentium II, etc

\textsuperscript{14}5_{10}=101_{2}

\textsuperscript{15}wikipedia.org/wiki/Parity_flag
One common reason to test the parity flag actually has nothing to do with parity. The FPU has four condition flags (C0 to C3), but they can not be tested directly, and must instead be first copied to the flags register. When this happens, C0 is placed in the carry flag, C2 in the parity flag and C3 in the zero flag. The C2 flag is set when e.g. incomparable floating point values (NaN or unsupported format) are compared with the FUCOM instructions.

As noted in Wikipedia, the parity flag used sometimes in FPU code, let’s see how. The PF flag is to be set to 1 if both C0 and C2 are set to 0 or both are 1, in which case the subsequent JP (jump if PF==1) is triggering. If we recall the values of C3/C2/C0 for various cases, we can see that the conditional jump JP is triggering in two cases: if \( b > a \) or \( a = b \) (C3 bit is not considered here, since it was cleared by the test ah, 5 instruction).

It is all simple after that. If the conditional jump was triggered, FLD loads the value of \( _b \) in ST(0), and if it was not triggered, the value of \( _a \) is loaded there.

And what about checking C2?

The C2 flag is set in case of error (NaN, etc), but our code doesn’t check it. If the programmer cares about FPU errors, he/she must add additional checks.
First OllyDbg example: $a=1.2$ and $b=3.4$

Let's load the example into OllyDbg:

![Figure 17.6: OllyDbg: first FLD is executed](image)

Current arguments of the function: $a = 1.2$ and $b = 3.4$ (We can see them in the stack: two pairs of 32-bit values). $b$ (3.4) is already loaded in $ST(0)$. Now FCOMP is being executed. OllyDbg shows the second FCOMP argument, which is in stack right now.
FCOMP is executed:

![Image of OllyDbg showing FCOMP execution](image)

**Figure 17.7**: OllyDbg: FCOMP is executed

We see the state of the FPU's condition flags: all zeroes. The popped value is reflected as ST(7), it was written earlier about reason for this: 17.5.1 on page 306.
FNSTSW is executed:

We see that the AX register contain zeroes: indeed, all condition flags are zero. (OllyDbg disassembles the FNSTSW instruction as FSTSW—they are synonyms).
The PF flag is set to 1. Indeed: the number of bits set in 0 is 0 and 0 is an even number. OllyDbg disassembles JP as JPE\textsuperscript{16}—they are synonyms. And it is about to trigger now.

\textsuperscript{16}Jump Parity Even (x86 instruction)
JPE triggered, FLD loads the value of $b$ (3.4) in ST(0):

The function finishes its work.
Second OllyDbg example: \(a=5.6\) and \(b=-4\)

Let’s load example into OllyDbg:

![OllyDbg image]

Figure 17.11: OllyDbg: first `FLD` executed

Current function arguments: \(a = 5.6\) and \(b = -4\). \(b\) (-4) is already loaded in \(ST(0)\). FCOMP about to execute now. OllyDbg shows the second FCOMP argument, which is in stack right now.
CHAPTER 17. FLOATING-POINT UNIT

FCOMP executed:

Figure 17.12: OllyDbg: FCOMP executed

We see the state of the FPU’s condition flags: all zeroes except C0.
Figure 17.13: OllyDbg: FNSTSW executed

We see that the AX register contains 0x100: the C0 flag is at the 16th bit.
The PF flag is cleared. Indeed: the count of bits set in 0x100 is 1 and 1 is an odd number. JPE is being skipped now.
JPE wasn't triggered, so FLD loads the value of \( a (5.6) \) in \( ST(0) \):

![OllyDbg: second FLD executed]

The function finishes its work.

**Optimizing MSVC 2010**

```
Listing 17.11: Optimizing MSVC 2010

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_d_max PROC
  fld QWORD PTR _b$[esp-4]
  fld QWORD PTR _a$[esp-4]
  ; current stack state: ST(0) = _a, ST(1) = _b
  fcom ST(1) ; compare _a and ST(1) = (_b)
  fnstsw ax
  test ah, 65 ; 00000041H
  jne SHORT $LN5@d_max
  ; copy ST(0) to ST(1) and pop register,
  ; leave (_a) on top
  fstp ST(1)
```

331
FCOM differs from FCOMP in the sense that it just compares the values and doesn’t change the FPU stack. Unlike the previous example, here the operands are in reverse order, which is why the result of the comparison in C3/C2/C0 is different:

- If \( a > b \) in our example, then C3/C2/C0 bits are to be set as: 0, 0, 0.
- If \( b > a \), then the bits are: 0, 0, 1.
- If \( a = b \), then the bits are: 1, 0, 0.

The test ah, 65 instruction leaves just two bits — C3 and C0. Both will be zero if \( a > b \): in that case the JNE jump will not be triggered. Then FSTP ST(1) follows — this instruction copies the value from ST(0) to the operand and pops one value from the FPU stack. In other words, the instruction copies ST(0) (where the value of \( _a \) is now) into ST(1). After that, two copies of \( _a \) are at the top of the stack. Then, one value is popped. After that, ST(0) contains \( _a \) and the function is finishes.

The conditional jump JNE is triggering in two cases: if \( b > a \) or \( a = b \). ST(0) is copied into ST(0), it is just like an idle (NOP) operation, then one value is popped from the stack and the top of the stack (ST(0)) is contain what was in ST(1) before (that is \( _b \)). Then the function finishes. The reason this instruction is used here probably is because the FPU has no other instruction to pop a value from the stack and discard it.
First OllyDbg example: $a=1.2$ and $b=3.4$

Both FLD are executed:

![Figure 17.16: OllyDbg: both FLD are executed](image)

FCOM being executed: OllyDbg shows the contents of ST(0) and ST(1) for convenience.
FCOM is done:

C0 is set, all other condition flags are cleared.

Figure 17.17: OllyDbg: FCOM is done
FNSTSW is done, AX=0x3100:

Figure 17.18: OllyDbg: FNSTSW is executed
TEST is executed:

Figure 17.19: OllyDbg: TEST is executed

ZF=0, conditional jump is about to trigger now.
FSTP ST (or FSTP ST(0)) was executed —1.2 was popped from the stack, and 3.4 was left on top:

We see that the FSTP ST instruction works just like popping one value from the FPU stack.

Figure 17.20: OllyDbg: FSTP is executed
Second OllyDbg example: \(a=5.6\) and \(b=-4\)

Both FLD are executed:

![Image showing OllyDbg output](image_url)

Figure 17.21: OllyDbg: both FLD are executed

FCOM is about to execute.
Figure 17.22: OllyDbg: FCOM is finished

All conditional flags are cleared.
FNSTSw done, AX=0x3000:

Figure 17.23: OllyDbg: FNSTSw was executed
Figure 17.24: OllyDbg: TEST was executed

ZF=1, jump will not happen now.
FSTP ST(1) was executed: a value of 5.6 is now at the top of the FPU stack.

We now see that the FSTP ST(1) instruction works as follows: it leaves what was at the top of the stack, but clears ST(1).

**GCC 4.4.1**

Listing 17.12: GCC 4.4.1

```
d_max proc near
    b     = qword ptr -10h
    a     = qword ptr -8
    a_first_half = dword ptr 8
    a_second_half = dword ptr 0Ch
    b_first_half = dword ptr 10h
    b_second_half = dword ptr 14h

    push    ebp
    mov     ebp, esp
    sub     esp, 10h

; put a and b to local stack:
```

Figure 17.25: OllyDbg: FSTP was executed
FUCOMPP is almost like FCOM, but pops both values from the stack and handles “not-a-numbers” differently.

A bit about not-a-numbers.

The FPU is able to deal with special values which are not-a-numbers or NaNs. These are infinity, result of division by 0, etc. Not-a-numbers can be “quiet” and “signaling”. It is possible to continue to work with “quiet” NaNs, but if one tries to

17 wikipedia.org/wiki/NaN
do any operation with “signaling” NaNs, an exception is to be raised.

FCOM raising an exception if any operand is NaN. FUCOM raising an exception only if any operand is a signaling NaN (SNaN).

The next instruction is SAHF (Store AH into Flags) — this is a rare instruction in code not related to the FPU. 8 bits from AH are moved into the lower 8 bits of the CPU flags in the following order:

```
7 6 4 2 0
SFZF AF PF CF
```

Let’s recall that FNSTSW moves the bits that interest us (C3/C2/C0) into AH and they are in positions 6, 2, 0 of the AH register:

```
6 2 1 0
C3 C2 C1 C0
```

In other words, the `fnstsw ax / sahf` instruction pair moves C3/C2/C0 into ZF, PF and CF.

Now let’s also recall the values of C3/C2/C0 in different conditions:

- If \( a \) is greater than \( b \) in our example, then C3/C2/C0 are to be set to: 0, 0, 0.
- If \( a \) is less than \( b \), then the bits are to be set to: 0, 0, 1.
- If \( a = b \), then: 1, 0, 0.

In other words, these states of the CPU flags are possible after three FUCOMPP/FNSTSW/SAHF instructions:

- If \( a > b \), the CPU flags are to be set as: ZF=0, PF=0, CF=0.
- If \( a < b \), then the flags are to be set as: ZF=0, PF=0, CF=1.
- And if \( a = b \), then: ZF=1, PF=0, CF=0.

Depending on the CPU flags and conditions, SETNBE stores 1 or 0 to AL. It is almost the counterpart of JNBE, with the exception that SETcc stores 1 or 0 in AL, but Jcc does actually jump or not. SETNBE stores 1 only if CF=0 and ZF=0. If it is not true, 0 is to be stored into AL.

Only in one case both CF and ZF are 0: if \( a > b \).

Then 1 is to be stored to AL, the subsequent JZ is not to be triggered and the function will return _a. In all other cases, _b is to be returned.

**Optimizing GCC 4.4.1**

\(^{18}cc\) is condition code
Listing 17.13: Optimizing GCC 4.4.1

public d_max

d_max proc near

arg_0 = qword ptr 8
arg_8 = qword ptr 10h

push ebp
mov ebp, esp
fld [ebp+arg_0] ; _a
fld [ebp+arg_8] ; _b

; stack state now: ST(0) = _b, ST(1) = _a
fxch st(1)

; stack state now: ST(0) = _a, ST(1) = _b
fucom st(1); compare _a and _b
fnstsw ax
sahf
ja short loc_8048448

; store ST(0) to ST(0) (idle operation), pop value at top of stack,
; leave _b at top
fstp st
jmp short loc_804844A

loc_8048448:
; store _a to ST(0), pop value at top of stack, leave _a at top
fstp st(1)

loc_804844A:
pop ebp
retn

d_max endp

It is almost the same except that JA is used after SAHF. Actually, conditional jump instructions that check “larger”, “lesser” or “equal” for unsigned number comparison (these are JA, JAE, JB, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) check only flags CF and ZF.

Let’s recall where bits C3/C2/C0 are located in the AH register after the execution of FSTSW/FNSTSW:

\[
\begin{array}{c|c|c|c}
| 6 | 5 | 4 | 3 | 2 | 1 | 0 |
\end{array}
\]

\[
\begin{array}{c|c|c|c}
| C3 | C2 | C1 | C0 |
\end{array}
\]
Let's also recall, how the bits from AH are stored into the CPU flags the execution of SAHF:

```
SFZF AF PF CF
```

After the comparison, the C3 and C0 bits are moved into ZF and CF, so the conditional jumps are able work after. JA is triggering if both CF are ZF zero.

Thereby, the conditional jumps instructions listed here can be used after a FN-STSW/SAHF instruction pair.

Apparently, the FPU C3/C2/C0 status bits were placed there intentionally, to easily map them to base CPU flags without additional permutations?

**GCC 4.8.1 with -O3 optimization turned on**

Some new FPU instructions were added in the P6 Intel family\(^\text{19}\). These are FU-COMI (compare operands and set flags of the main CPU) and FCMOVcc (works like CMOVcc, but on FPU registers). Apparently, the maintainers of GCC decided to drop support of pre-P6 Intel CPUs (early Pentiums, 80486, etc).

And also, the FPU is no longer separate unit in P6 Intel family, so now it is possible to modify/check flags of the main CPU from the FPU.

So what we get is:

```
Listing 17.14: Optimizing GCC 4.8.1
fld QWORD PTR [esp+4] ; load "a"
fld QWORD PTR [esp+12] ; load "b"
; ST0=b, ST1=a
fxch st(1)
; ST0=a, ST1=b
; compare "a" and "b"
fucomi st, st(1)
; copy ST1 ("b" here) to ST0 if a<=b
; leave "a" in ST0 otherwise
fcmovbe st, st(1)
; discard value in ST1
fstp st(1)
ret
```

Hard to guess why FXCH (swap operands) is here. It's possible to get rid of it easily by swapping the first two FLD instructions or by replacing FCMOVBE (below or equal) by FCMOVA (above). Probably it's a compiler inaccuracy.

\(^{19}\)Starting at Pentium Pro, Pentium-II, etc.
So FUCOMI compares ST(0) \((a)\) and ST(1) \((b)\) and then sets some flags in the main CPU. FCMOVBE checks the flags and copies ST(1) \((b\) here at the moment) to ST(0) \((a\) here) if \(ST0(a) \leq ST1(b)\). Otherwise \((a > b)\), it leaves \(a\) in ST(0).

The last FSTP leaves ST(0) on top of the stack, discarding the contents of ST(1).

Let’s trace this function in GDB:

Listing 17.15: Optimizing GCC 4.8.1 and GDB

```
1 dennis@ubuntuvm:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
2 dennis@ubuntuvm:~/polygon$ gdb d_max
3 GNU gdb (GDB) 7.6.1-ubuntu
4 Copyright (C) 2013 Free Software Foundation, Inc.
5 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
6 This is free software: you are free to change and redistribute it.
7 There is NO WARRANTY, to the extent permitted by law. Type "show copying"
   and "show warranty" for details.
8 This GDB was configured as "i686-linux-gnu".
9 For bug reporting instructions, please see: <http://www.gnu.org/software/gdb/bugs/>...
10 Reading symbols from /home/dennis/polygon/d_max...(no debugging symbols found)...done.
11 (gdb) b d_max
12 Breakpoint 1 at 0x80484a0
13 (gdb) run
14 Starting program: /home/dennis/polygon/d_max
15 Breakpoint 1, 0x080484a0 in d_max ()
16 (gdb) ni
17 0x080484a4 in d_max ()
18 (gdb) disas $eip
19 Dump of assembler code for function d_max:
20   0x080484a0 <+0>:  fldl 0x4(%esp)
21 => 0x080484a4 <+4>:  fldl 0xc(%esp)
22   0x080484a8 <+8>:  fxch %st(1)
23   0x080484aa <+10>:  fucomi %st(1),%st
24   0x080484ac <+12>:  fcmovbe %st(1),%st
25   0x080484ae <+14>:  fstp %st(1)
26   0x080484b0 <+16>:  ret
27 End of assembler dump.
28 (gdb) ni
29 0x080484a8 in d_max ()
30 (gdb) info float
31   R7: Valid 0x3fff999999999999800 +1.199999999999999956
32 =>R6: Valid 0x4000d999999999999999999999999911
```
36 R5: Empty 0x00000000000000000000000000000000
37 R4: Empty 0x00000000000000000000000000000000
38 R3: Empty 0x00000000000000000000000000000000
39 R2: Empty 0x00000000000000000000000000000000
40 R1: Empty 0x00000000000000000000000000000000
41 R0: Empty 0x00000000000000000000000000000000
42
43 Status Word: 0x3000
44 TOP: 6
45 Control Word: 0x037f IM DM ZM OM UM PM
46 PC: Extended Precision (64-bits)
47 RC: Round to nearest
48 Tag Word: 0x0fff
49 Instruction Pointer: 0x73:0x080484a4
50 Operand Pointer: 0x7b:0xbffff118
51 Opcode: 0x0000
52
53 (gdb) ni
54 0x080484aa in d_max ()
55 (gdb) info float
56 R7: Valid 0x4000d99999999999999 +3.399999999999999911
57 => R6: Valid 0x3fff9999999999999999 +1.199999999999999956
58 R5: Empty 0x00000000000000000000000000000000
59 R4: Empty 0x00000000000000000000000000000000
60 R3: Empty 0x00000000000000000000000000000000
61 R2: Empty 0x00000000000000000000000000000000
62 R1: Empty 0x00000000000000000000000000000000
63 R0: Empty 0x00000000000000000000000000000000
64
65 Status Word: 0x3000
66 TOP: 6
67 Control Word: 0x037f IM DM ZM OM UM PM
68 PC: Extended Precision (64-bits)
69 RC: Round to nearest
70 Tag Word: 0x0fff
71 Instruction Pointer: 0x73:0x080484a8
72 Operand Pointer: 0x7b:0xbffff118
73 Opcode: 0x0000
74 (gdb) disas $eip
75 Dump of assembler code for function d_max:
76 0x080484a0 <+0>: fldl 0x4(%esp)
77 0x080484a4 <+4>: fldl 0xc(%esp)
78 0x080484a8 <+8>: fxch %st(1)
79 => 0x080484aa <+10>: fucomi %st(1),%st
80 0x080484ac <+12>: fcmovbe %st(1),%st
81 0x080484ae <+14>: fstp %st(1)
82 0x080484b0 <+16>: ret
83
84 End of assembler dump.
(gdb) ni
0x080484ac in d_max()
(gdb) info registers
eax 0x1 1
ecx 0xbfffe4c -1073745468
edx 0x8048340 134513472
ebx 0xb7fbc000 -1208225792
esp 0xbffff10c 0xbffff10c
ebp 0xbffff128 0xbffff128
esi 0x0 0
edi 0x0 0
eip 0x80484ac 0x80484ac <d_max+12>
eflags 0x203 [ CF IF ]

cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x0 0

(gdb) ni
0x080484ae in d_max()
(gdb) info float
R7: Valid 0x4000d999999999999800 +3.399999999999999911
=>R6: Valid 0x4000d999999999999800 +3.399999999999999911
R5: Empty 0x00000000000000000000000000000000
R4: Empty 0x00000000000000000000000000000000
R3: Empty 0x00000000000000000000000000000000
R2: Empty 0x00000000000000000000000000000000
R1: Empty 0x00000000000000000000000000000000
R0: Empty 0x00000000000000000000000000000000

Status Word: 0x3000
TOP: 6
Control Word: 0x037f IM DM ZM OM UM PM
PC: Extended Precision (64-bits)
RC: Round to nearest
Tag Word: 0x0fff
Instruction Pointer: 0x73:0x080484ac
Operand Pointer: 0x7b:0xbffff118
Opcode: 0x0000
(gdb) disas $eip
Dump of assembler code for function d_max:
0x080484a0 <+0>: fldl 0x4(%esp)
0x080484a4 <+4>: fldl 0xc(%esp)
0x080484a8 <+8>: fxch %st(1)
0x080484aa <+10>: fucomi %st(1),%st
0x080484ac <+12>: fcmovbe %st(1),%st

(gdb)
Using “ni”, let’s execute the first two FLD instructions.

Let’s examine the FPU registers (line 33).

As it was mentioned before, the FPU registers set is a circular buffer rather than a stack (17.5.1 on page 306). And GDB doesn’t show STx registers, but internal the FPU registers (Rx). The arrow (at line 35) points to the current top of the stack. You can also see the TOP register contents in Status Word (line 44)—it is 6 now, so the stack top is now pointing to internal register 6.

The values of a and b are swapped after FXCH is executed (line 54).

FUCOMI is executed (line 83). Let’s see the flags: CF is set (line 95).

FCMOVBE has copied the value of b (see line 104).

FSTP leaves one value at the top of stack (line 136). The value of TOP is now 7, so
the FPU stack top is pointing to internal register 7.

17.7.2 ARM

Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 17.16: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV D16, R2, R3 ; b</td>
<td></td>
</tr>
<tr>
<td>VMOV D17, R0, R1 ; a</td>
<td></td>
</tr>
<tr>
<td>VCMPE.F64 D17, D16</td>
<td></td>
</tr>
<tr>
<td>VMRS APSR_nzcv, FPSCR</td>
<td></td>
</tr>
<tr>
<td>VMOVGT.F64 D16, D17 ; copy &quot;b&quot; to D16</td>
<td></td>
</tr>
<tr>
<td>VMOV R0, R1, D16</td>
<td></td>
</tr>
<tr>
<td>BX LR</td>
<td></td>
</tr>
</tbody>
</table>

A very simple case. The input values are placed into the D17 and D16 registers and then compared using the VCMPE instruction. Just like in the x86 coprocessor, the ARM coprocessor has its own status and flags register (FPSCR\(^{20}\)), since there is a need to store coprocessor-specific flags. And just like in x86, there are no conditional jump instruction in ARM, that can check bits in the status register of the coprocessor. So there is VMRS, which copies 4 bits (N, Z, C, V) from the coprocessor status word into bits of the general status register (APSR\(^{21}\)).

VMOVGT is the analog of the MOVGT, instruction for D-registers, it executes if one operand is greater than the other while comparing (GT—Greater Than).

If it gets executed, the value of \( b \) is to be written into D16 (that is currently stored in D17).

Otherwise the value of \( a \) stays in the D16 register.

The penultimate instruction VMOV prepares the value in the D16 register for returning it via the R0 and R1 register pair.

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

Listing 17.17: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV D16, R2, R3 ; b</td>
<td></td>
</tr>
<tr>
<td>VMOV D17, R0, R1 ; a</td>
<td></td>
</tr>
<tr>
<td>VCMPE.F64 D17, D16</td>
<td></td>
</tr>
<tr>
<td>VMRS APSR_nzcv, FPSCR</td>
<td></td>
</tr>
</tbody>
</table>

\(^{20}\)(ARM) Floating-Point Status and Control Register

\(^{21}\)(ARM) Application Program Status Register
Almost the same as in the previous example, however slightly different. As we already know, many instructions in ARM mode can be supplemented by condition predicate.

But there is no such thing in Thumb mode. There is no space in the 16-bit instructions for 4 more bits in which conditions can be encoded.

However, Thumb-2 was extended to make it possible to specify predicates to old Thumb instructions.

Here, in the IDA-generated listing, we see the VMOVGT instruction, as in previous example.

In fact, the usual VMOV is encoded there, but IDA adds the -GT suffix to it, since there is a “IT GT” instruction placed right before it.

The IT instruction defines a so-called *if-then block*. After the instruction it is possible to place up to 4 instructions, each of them has a predicate suffix. In our example, IT GT implies that the next instruction is to be executed, if the GT (*Greater Than*) condition is true.

Here is a more complex code fragment, by the way, from Angry Birds (for iOS):

```assembly
ITE NE
VMOVNE R2, R3, D16
VMOVEQ R2, R3, D17
BLX _objc_msgSend ; not prefixed
```

ITE stands for *if-then-else* and it encodes suffixes for the next two instructions. The first instruction executes if the condition encoded in ITE (NE, *not equal*) is true at, and the second—if the condition is not true. (The inverse condition of NE is EQ (*equal*).)

The instruction followed after the second VMOV (or VMOVEQ) is a normal one, not prefixed (BLX).

One more that’s slightly harder, which is also from Angry Birds:

```assembly
... ITTTT EQ
```
Four “T” symbols in the instruction mnemonic mean that the four subsequent instructions are to be executed if the condition is true. That’s why IDA adds the -EQ suffix to each one of them.

And if there was be, for example, ITEE EQ (if-then-else-else-else), then the suffixes would have been set as follows:

- -EQ
- -NE
- -NE
- -NE

Another fragment from Angry Birds:

Listing 17.20: Angry Birds Classic

```assembly
... 
CMP.W R0, #0xFFFFFFFF
ITTE LE 
SUBL.E.W R10, R0, #1
NEG.E.LE R0, R0
MOVG.T R10, R0
MOVS R6, #0 ; not prefixed
CBZ R0, loc_1E7E32 ; not prefixed
...
```

ITTE (if-then-then-else) implies that the 1st and 2nd instructions are to be executed if the LE (Less or Equal) condition is true, and the 3rd—if the inverse condition (GT—Greater Than) is true.

Compilers usually don’t generate all possible combinations. For example, in the mentioned Angry Birds game (classic version for iOS) only these variants of the IT instruction are used: IT, ITE, ITT, ITTE, ITTT, ITTTT. How to learn this? In IDA it is possible to produce listing files, so it was created with an option to show 4 bytes for each opcode. Then, knowing the high part of the 16-bit opcode (IT is 0xBF), we do the following using grep:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

By the way, if you program in ARM assembly language manually for Thumb-2 mode, and you add conditional suffixes, the assembler will add the IT instructions automatically with the required flags where it is necessary.

**Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)**

Listing 17.21: Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
b = -0x20
a = -0x18
val_to_return = -0x10
saved_R7 = -4

STR R7, [SP,#saved_R7]!
MOV R7, SP
SUB SP, SP, #0x1C
BIC SP, SP, #7
VMOV D16, R2, R3
VMOV D17, R0, R1
VSTR D16, [SP,#0x20+a]
VSTR D16, [SP,#0x20+b]
VLDR D16, [SP,#0x20+a]
VLDR D17, [SP,#0x20+b]
VCMPE.F64 D16, D17
VMRS APSR_nzcv, FPSCR
BLE loc_2E08
VLDR D16, [SP,#0x20+a]
VSTR D16, [SP,#0x20+val_to_return]
B loc_2E10

loc_2E08
VLDR D16, [SP,#0x20+b]
VSTR D16, [SP,#0x20+val_to_return]

loc_2E10
VLDR D16, [SP,#0x20+val_to_return]
VMOV R0, R1, D16
MOV SP, R7
LDR R7, [SP+0x20+b],#4
BX LR
```

Almost the same as we already saw, but there is too much redundant code because the \(a\) and \(b\) variables are stored in the local stack, as well as the return value.

**Optimizing Keil 6/2013 (Thumb mode)**
Listing 17.22: Optimizing Keil 6/2013 (Thumb mode)

```assembly
PUSH {R3-R7,LR}
MOVS R4, R2
MOVS R5, R3
MOVS R6, R0
MOVS R7, R1
BL __aeabi_cdrcmple
BCS loc_1C0
MOVS R0, R6
MOVS R1, R7
POP {R3-R7,PC}
```

```assembly
loc_1C0

MOVS R0, R4
MOVS R1, R5
POP {R3-R7,PC}
```

Keil doesn’t generate FPU-instructions since it cannot rely on them being supported on the target CPU, and it cannot be done by straightforward bitwise comparing. So it calls an external library function to do the comparison: __aeabi_cdrcmple.

N.B. The result of the comparison is to be left in the flags by this function, so the following BCS (Carry set—Greater than or equal) instruction can work without any additional code.

### 17.7.3 ARM64

**Optimizing GCC (Linaro) 4.9**

```assembly
d_max:
; D0 - a, D1 - b
    fcmpe d0, d1
    fcsel d0, d0, d1, gt
; now result in D0
    ret
```

The ARM64 ISA has FPU-instructions which set APSR the CPU flags instead of FPSCR for convenience. The FPU is not a separate device here anymore (at least, logically). Here we see FCMPE. It compares the two values passed in D0 and D1 (which are the first and second arguments of the function) and sets APSR flags (N, Z, C, V).

FCSEL (Floating Conditional Select) copies the value of D0 or D1 into D0 depending on the condition (GT—Greater Than), and again, it uses flags in APSR register.
instead of FPSCR. This is much more convenient, compared to the instruction set in older CPUs.

If the condition is true (GT), then the value of D0 is copied into D0 (i.e., nothing happens). If the condition is not true, the value of D1 is copied into D0.

Non-optimizing GCC (Linaro) 4.9

```
d_max:
; save input arguments in "Register Save Area"
    sub    sp, sp, #16
    str    d0, [sp,8]
    str    d1, [sp]
; reload values
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    fmov   d0, x1
    fmov   d1, x0
; D0 - a, D1 - b
    fcmpe  d0, d1
    ble    .L76
; a>b; load D0 (a) into X0
    ldr    x0, [sp,8]
    b      .L74
.L76:
; a<=b; load D1 (b) into X0
    ldr    x0, [sp]
.L74:
; result in X0
    fmov   d0, x0
; result in D0
    add    sp, sp, 16
    ret
```

Non-optimizing GCC is more verbose. First, the function saves its input argument values in the local stack (Register Save Area). Then the code reloads these values into registers X0/X1 and finally copies them to D0/D1 to be compared using FCMPE. A lot of redundant code, but that is how non-optimizing compilers work. FCMPE compares the values and sets the APSR flags. At this moment, the compiler is not thinking yet about the more convenient FCSEL instruction, so it proceed using old methods: using the BLE instruction (Branch if Less than or Equal). In the first case ($a > b$), the value of $a$ gets loaded into X0. In the other case ($a \leq b$), the value of $b$ gets loaded into X0. Finally, the value from X0 gets copied into D0, because the return value needs to be in this register.
Exercise

As an exercise, you can try optimizing this piece of code manually by removing redundant instructions and not introducing new ones (including FCSEL).

Optimizing GCC (Linaro) 4.9—float

Let’s also rewrite this example to use float instead of double.

```c
float f_max (float a, float b)
{
    if (a>b)
        return a;
    return b;
}
```

It is the same code, but the S-registers are used instead of D- ones. It's because numbers of type float are passed in 32-bit S-registers (which are in fact the lower parts of the 64-bit D-registers).

17.7.4 MIPS

The co-processor of the MIPS processor has a condition bit which can be set in the FPU and checked in the CPU. Earlier MIPS-es have only one condition bit (called FCC0), later ones have 8 (called FCC7-FCC0). This bit (or bits) are located in the register called FCCR.

Listing 17.23: Optimizing GCC 4.4.5 (IDA)

```
d_max:
    ; set FPU condition bit if $f14<$f12 (b<a):
    c.lt.d $f14, $f12
    or $at, $zero ; NOP
    ; jump to locret_14 if condition bit is set
    bc1t locret_14
```
C.LT.D compares two values. LT is the condition “Less Than”. D implies values of type *double*. Depending on the result of the comparison, the FCC0 condition bit is either set or cleared.

BC1T checks the FCC0 bit and jumps if the bit is set. T mean that the jump is to be taken if the bit is set (“True”). There is also the instruction “BC1F” which jumps if the bit is cleared (“False”).

Depending on the jump, one of function arguments is placed into $F0$.

### 17.8 Stack, calculators and reverse Polish notation

Now we understand why some old calculators used reverse Polish notation. For example, for addition of 12 and 34 one has to enter 12, then 34, then press “plus” sign. It’s because old calculators were just stack machine implementations, and this was much simpler than to handle complex parenthesized expressions.

### 17.9 x64

On how floating point numbers are processed in x86-64, read more here: [27 on page 591](#).

### 17.10 Exercises

- [http://challenges.re/60](http://challenges.re/60)
- [http://challenges.re/61](http://challenges.re/61)

---

Chapter 18

Arrays

An array is just a set of variables in memory that lie next to each other and that have the same type\(^1\).

18.1 Simple example

```c
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf("a[%d]=%d\n", i, a[i]);

    return 0;
}
```

\(^1\)AKA \(^2\) “homogeneous container”
MSVC

Let’s compile:

Listing 18.1: MSVC 2008

```assembly
_TEXT SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 84 ; 0000054H
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN6@main
$LN5@main:
    mov eax, DWORD PTR _i$[ebp]
    add eax, 1
    mov DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp DWORD PTR _i$[ebp], 20 ; 0000014H
    jge SHORT $LN4@main
    mov ecx, DWORD PTR _i$[ebp]
    shl ecx, 1
    mov edx, DWORD PTR _i$[ebp]
    mov DWORD PTR _a$[ebp+edx*4], ecx
    jmp SHORT $LN5@main
$LN4@main:
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN3@main
$LN2@main:
    mov eax, DWORD PTR _i$[ebp]
    add eax, 1
    mov DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp DWORD PTR _i$[ebp], 20 ; 0000014H
    jge SHORT $LN1@main
    mov ecx, DWORD PTR _i$[ebp]
    mov edx, DWORD PTR _a$[ebp+ecx*4]
    push edx
    mov eax, DWORD PTR _i$[ebp]
    push eax
    push OFFSET $SG2463
    call _printf
    add esp, 12 ; 0000000cH
    jmp SHORT $LN2@main
```
$LN1@main:
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret 0
_main   ENDP

Nothing very special, just two loops: the first is a filling loop and second is a printing loop. The `shl ecx, 1` instruction is used for value multiplication by 2 in ECX, more about below 16.2.1 on page 297.

80 bytes are allocated on the stack for the array, 20 elements of 4 bytes.
Let's try this example in OllyDbg.

We see how the array gets filled: each element is 32-bit word of *int* type and its value is the index multiplied by 2:

![OllyDbg: after array filling](image)

Since this array is located in the stack, we can see all its 20 elements there.

**GCC**

Here is what GCC 4.4.1 does:

```
Listing 18.2: GCC 4.4.1

```
```c
main proc near ; DATA XREF: _start+17

  var_70 = dword ptr -70h
  var_6C = dword ptr -6Ch
  var_68 = dword ptr -68h
  i_2 = dword ptr -54h
  i = dword ptr -4

  push ebp
```
By the way, variable $a$ is of type $int^*$ (the pointer to $int$)—you can pass a pointer to an array to another function, but it’s more correct to say that a pointer to the first element of the array is passed (the addresses of rest of the elements are calculated in an obvious way). If you index this pointer as $a[idx]$, $idx$ is just to be added to the pointer and the element placed there (to which calculated pointer is pointing) is to be returned.

An interesting example: a string of characters like “string” is an array of characters and it has a type of $const char[]$. An index can also be applied to this pointer. And
that is why it is possible to write things like “string”[i]—this is a correct C/C++ expression!

18.1.2 ARM

Non-optimizing Keil 6/2013 (ARM mode)

```
EXPORT _main

_main
STMFD SP!, {R4,LR}
SUB SP, SP, #0x50 ; allocate place for 20 int variables

; first loop

    MOV R4, #0 ; i
    B  loc_4A0
loc_494
    MOV R0, R4,LSL#1 ; R0=R4*2
    STR R0, [SP,R4,LSL#2] ; store R0 to SP+R4<<2
(same as SP+R4*4)
    ADD R4, R4, #1 ; i=i+1
loc_4A0
    CMP R4, #20 ; i<20?
    BLT loc_494 ; yes, run loop body again

; second loop

    MOV R4, #0 ; i
    B  loc_4C4
loc_4B0
    LDR R2, [SP,R4,LSL#2] ; (second printf argument) R2=*(SP+R4<<4) (same as *(SP+R4*4))
    MOV R1, R4 ; (first printf argument) R1=i
    ADR R0, aADD ; "a[%d]=%d\n"
    BL __2printf
    ADD R4, R4, #1 ; i=i+1
loc_4C4
    CMP R4, #20 ; i<20?
    BLT loc_4B0 ; yes, run loop body again
    MOV R0, #0 ; value to return
```
int type requires 32 bits for storage (or 4 bytes), so to store 20 int variables 80 (0x50) bytes are needed. So that is why the SUB SP, SP, #0x50 instruction in the function’s prologue allocates exactly this amount of space in the stack.

In both the first and second loops, the loop iterator i is placed in the R4 register.

The number that is to be written into the array is calculated as \( i \times 2 \), which is effectively equivalent to shifting it left by one bit, so MOV R0, R4, LSL#1 instruction does this.

STR R0, [SP, R4, LSL#2] writes the contents of R0 into the array. Here is how a pointer to array element is calculated: SP points to the start of the array, R4 is \( i \). So shifting \( i \) left by 2 bits is effectively equivalent to multiplication by 4 (since each array element has a size of 4 bytes) and then it's added to the address of the start of the array.

The second loop has an inverse LDR R2, [SP, R4, LSL#2] instruction. It loads the value we need from the array, and the pointer to it is calculated likewise.

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

```
_MAIN
PUSH {R4,R5,LR}
; allocate place for 20 int variables + one more variable
SUB SP, SP, #0x54

; first loop
MOVS R0, #0 ; i
MOV R5, SP ; pointer to first array element

loc_1CE
LSLS R1, R0, #1 ; R1=i<<1 (same as i*2)
LSLS R2, R0, #2 ; R2=i<<2 (same as i*4)
ADDS R0, R0, #1 ; i=i+1
CMP R0, #20 ; i<20?
STR R1, [R5,R2] ; store R1 to *(R5+R2) (same R5+i*4)
BLT loc_1CE ; yes, i<20, run loop body again

; second loop
```
Thumb code is very similar. Thumb mode has special instructions for bit shifting (like LSLS), which calculates the value to be written into the array and the address of each element in the array as well.

The compiler allocates slightly more space in the local stack, however, the last 4 bytes are not used.

Non-optimizing GCC 4.9.1 (ARM64)

Listing 18.3: Non-optimizing GCC 4.9.1 (ARM64)

```
.LC0:
.string "a[%d]=%d\n"
main:
; save FP and LR in stack frame:
    stp    x29, x30, [sp, -112]!
; set stack frame (FP=SP)
    add    x29, sp, 0
; setting initial counter variable at 0 (WZR is the register always holding zero):
    str    wzr, [x29,108]
; jump to loop condition checking code:
    b      .L2
.L3:
; load value of "i" variable:
    ldr    w0, [x29,108]
; multiplicate it by 2:
    lsl    w2, w0, 1
; find a place of an array in local stack:
```
add x0, x29, 24
; load 32-bit integer from local stack and sign extend it to 64-bit one:
    ldrsw x1, [x29,108]
; calculate address of element (X0+X1<<2=array address+i*4) and store W2 (i*2) there:
    str w2, [x0,x1,lsl 2]
; increment counter (i):
    ldr w0, [x29,108]
    add w0, w0, 1
    str w0, [x29,108]
.L2:
; check if we finished:
    ldr w0, [x29,108]
    cmp w0, 19
    ble .L3
; second part of the function begins here.
; setting initial counter variable at 0.
; by the way, the same place in the local stack was used for counter,
; because the same local variable (i) is being used as counter.
    str wzr, [x29,108]
    b .L4
.L5:
; calculate array address:
    add x0, x29, 24
; load "i" value:
    ldrsw x1, [x29,108]
; load value from the array at the address (X0+X1<<2 = address of array + i*4)
    ldr w2, [x0,x1,lsl 2]
; load address of the "a[%d]=%d\n" string:
    adrp x0, .LC0
    add x0, x0, :lo12:.LC0
; load "i" variable to W1 and pass it to printf() as second argument:
    ldr w1, [x29,108]
; W2 still contains the value of array element which was just loaded.
; call printf():
    bl printf
; increment "i" variable:
    ldr w0, [x29,108]
    add w0, w0, 1
    str w0, [x29,108]
.L4:
; are we finished?
    ldr w0, [x29,108]
18.1.3 MIPS

The function uses a lot of S-registers which must be preserved, so that's why its values are saved in the function prologue and restored in the epilogue.

Listing 18.4: Optimizing GCC 4.4.5 (IDA)

```
main:

var_70     = -0x70
var_68     = -0x68
var_14     = -0x14
var_10     = -0x10
var_C      = -0xC
var_8      = -8
var_4      = -4

; function prologue:
    lui   $gp, (__gnu_local_gp >> 16)
    addiu $sp, -0x80
    la    $gp, (__gnu_local_gp & 0xFFFF)
    sw    $ra, 0x80+var_4($sp)
    sw    $s3, 0x80+var_8($sp)
    sw    $s2, 0x80+var_C($sp)
    sw    $s1, 0x80+var_10($sp)
    sw    $s0, 0x80+var_14($sp)
    sw    $gp, 0x80+var_70($sp)
    addiu $s1, $sp, 0x80+var_68
    move  $v1, $s1
    move  $v0, $zero

; that value will be used as a loop terminator.
; it was precalculated by GCC compiler at compile stage:
    li    $a0, 0x28  # '('

loc_34:        # CODE XREF: main+3C
; store value into memory:
    sw    $v0, 0($v1)
; increase value to be stored by 2 at each iteration:
```
addiu $v0, 2  ; loop terminator reached?
bne $v0, $a0, loc_34  ; add 4 to address anyway:
addiu $v1, 4  ; array filling loop is ended
; second loop begin
la $s3, $LC0 # "a[%d]=%d\n"
; "i" variable will reside in $s0:
move $s0, $zero
li $s2, 0x14
loc_54: # CODE XREF: main+70
; call printf():
lw $t9, (printf & 0xFFFF)($gp)
lw $a2, 0($s1)
move $a1, $s0
move $a0, $s3
jalr $t9
; increment "i":
addiu $s0, 1
lw $gp, 0x80+var_70($sp)
; jump to loop body if end is not reached:
bne $s0, $s2, loc_54
; move memory pointer to the next 32-bit word:
addiu $s1, 4
; function epilogue
lw $ra, 0x80+var_4($sp)
move $v0, $zero
lw $s3, 0x80+var_8($sp)
lw $s2, 0x80+var_C($sp)
lw $s1, 0x80+var_10($sp)
lw $s0, 0x80+var_14($sp)
jr $ra
addiu $sp, 0x80
$LC0: .ascii "a[%d]=%d\n"<0> # DATA XREF: main+44

Something interesting: there are two loops and the first one doesn't needs $i$, it
needs only $i \times 2$ (increased by 2 at each iteration) and also the address in memory
(increased by 4 at each iteration). So here we see two variables, one (in $V0)
increasing by 2 each time, and another (in $V1) — by 4.

The second loop is where printf() is called and it reports the value of $i$ to the
user, so there is a variable which is increased by 1 each time (in $S0) and also a
memory address (in $S1) increased by 4 each time.
That reminds us of loop optimizations we considered earlier: on page 663. Their goal is to get rid of multiplications.

### 18.2 Buffer overflow

#### 18.2.1 Reading outside array bounds

So, array indexing is just `array[index]`. If you study the generated code closely, you'll probably note the missing index bounds checking, which could check if it is less than 20. What if the index is 20 or greater? That's the one C/C++ feature it is often blamed for.

Here is a code that successfully compiles and works:

```c
#include <stdio.h>

int main()
{
    int a[20];
    int i;
    
    for (i=0; i<20; i++)
        a[i]=i*2;

    printf ("a[20]=%d\n", a[20]);

    return 0;
}
```

Compilation results (MSVC 2008):

```
Listing 18.5: Non-optimizing MSVC 2008

SG2474 DB 'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 84
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN3@main
$LN2@main:
    mov eax, DWORD PTR _i$[ebp]
    add eax, 1
```

The code produced this result:

```
Figure 18.2: OllyDbg: console output
```

It is just *something* that was lying in the stack near to the array, 80 bytes away from its first element.
Let’s try to find out where did this value come from, using OllyDbg. Let’s load and find the value located right after the last array element:

![OllyDbg screenshot](image)

Figure 18.3: OllyDbg: reading of the 20th element and execution of `printf()`

What is this? Judging by the stack layout, this is the saved value of the EBP register.
Let's trace further and see how it gets restored:

![Figure 18.4: OllyDbg: restoring value of EBP](image)

Indeed, how it could be different? The compiler may generate some additional code to check the index value to be always in the array's bounds (like in higher-level programming languages\(^3\)) but this makes the code slower.

### 18.2.2 Writing beyond array bounds

OK, we read some values from the stack *illegally*, but what if we could write something to it?

Here is what we have got:

```c
#include <stdio.h>

int main()
{
    int a[20];
    int i;
}
```

\(^3\)Java, Python, etc
for (i=0; i<30; i++)
    a[i]=i;

return 0;

MSVC

And what we get:

Listing 18.6: Non-optimizing MSVC 2008

_TEXT SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main PROC
push ebp
mov ebp, esp
sub esp, 84
mov DWORD PTR _i$[ebp], 0
jmp SHORT $LN3@main
$LN2@main:
mov eax, DWORD PTR _i$[ebp]
add eax, 1
mov DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp DWORD PTR _i$[ebp], 30 ; 0000001eH
    jge SHORT $LN1@main
    mov ecx, DWORD PTR _i$[ebp]
    mov edx, DWORD PTR _i$[ebp] ; that instruction is obviously redundant
    mov DWORD PTR _a$[ebp+ecx*4], edx ; ECX could be used as second operand here instead
    jmp SHORT $LN2@main
$LN1@main:
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP

The compiled program crashes after running. No wonder. Let’s see where exactly does it is crash.
Let's load it into OllyDbg, and trace until all 30 elements are written:

![Figure 18.5: OllyDbg: after restoring the value of EBP](image)
Trace until the function end:

Now please keep your eyes on the registers.

EIP is 0x15 now. It is not a legal address for code—at least for win32 code! We got there somehow against our will. It is also interesting that the EBP register contain 0x14, ECX and EDX—0x1D.

Let’s study stack layout a bit more.

After the control flow was passed to main(), the value in the EBP register was saved on the stack. Then, 84 bytes were allocated for the array and the \( i \) variable. That’s \((20 + 1) \times \text{sizeof(int)}\). ESP now points to the \_i variable in the local stack and after the execution of the next \text{PUSH something}, \text{something} is appearing next to \_i.

That’s the stack layout while the control is in main():

<table>
<thead>
<tr>
<th>ESP</th>
<th>4 bytes allocated for ( i ) variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP+4</td>
<td>80 bytes allocated for a[20] array</td>
</tr>
<tr>
<td>ESP+84</td>
<td>saved EBP value</td>
</tr>
<tr>
<td>ESP+88</td>
<td>return address</td>
</tr>
</tbody>
</table>
a[19]=something statement writes the last int in the bounds of the array (in bounds so far!)

a[20]=something statement writes something to the place where the value of EBP is saved.

Please take a look at the register state at the moment of the crash. In our case, 20 was written in the 20th element. At the function end, the function epilogue restores the original EBP value. (20 in decimal is 0x14 in hexadecimal). Then RET gets executed, which is effectively equivalent to POP EIP instruction.

The RET instruction takes the return address from the stack (that is the address in CRT), which was called main(), and 21 is stored there (0x15 in hexadecimal). The CPU traps at address 0x15, but there is no executable code there, so exception gets raised.

Welcome! It is called a buffer overflow.

Replace the int array with a string (char array), create a long string deliberately and pass it to the program, to the function, which doesn’t check the length of the string and copies it in a short buffer, and you’ll able to point the program to an address to which it must jump. It’s not that simple in reality, but that is how it emerged.

GCC

Let’s try the same code in GCC 4.4.1. We get:

<table>
<thead>
<tr>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>public main</td>
</tr>
<tr>
<td>main proc near</td>
</tr>
<tr>
<td>a = dword ptr -54h</td>
</tr>
<tr>
<td>i = dword ptr -4</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>sub esp, 60h ; 96</td>
</tr>
<tr>
<td>mov [ebp+i], 0</td>
</tr>
<tr>
<td>jmp short loc_80483D1</td>
</tr>
<tr>
<td>loc_80483C3:</td>
</tr>
<tr>
<td>mov eax, [ebp+i]</td>
</tr>
<tr>
<td>mov edx, [ebp+i]</td>
</tr>
<tr>
<td>mov [ebp+eax*4+a], edx</td>
</tr>
<tr>
<td>add [ebp+i], 1</td>
</tr>
<tr>
<td>loc_80483D1:</td>
</tr>
<tr>
<td>cmp [ebp+i], 1Dh</td>
</tr>
</tbody>
</table>

---

4 wikipedia
5 Classic article about it: [One96].
Running this in Linux will produce: Segmentation fault.

If we run this in the GDB debugger, we get this:

```
(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x000000016 in ?? ()
(gdb) info registers
eax 0x0 0
ecx 0xd2f96388 -755407992
edx 0x1d 29
ebx 0x26eff4 2551796
esp 0xbffff4b0 0xbffff4b0
ebp 0x15 0x15
esi 0x0 0
edi 0x0 0
eip 0x16 0x16
eflags 0x10202 [ IF RF ]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51
```

The register values are slightly different than in win32 example, since the stack layout is slightly different too.

### 18.3 Buffer overflow protection methods

There are several methods to protect against this scourge, regardless of the C/C++ programmers’ negligence. MSVC has options like:

- `/RTCs` Stack Frame runtime checking
- `/GZ` Enable stack checks (/RTCs)

---

6 compiler-side buffer overflow protection methods: [wikipedia.org/wiki/Buffer_overflow_protection](http://wikipedia.org/wiki/Buffer_overflow_protection)
One of the methods is to write a random value between the local variables in stack at function prologue and to check it in function epilogue before the function exits. If value is not the same, do not execute the last instruction RET, but stop (or hang). The process will halt, but that is much better than a remote attack to your host.

This random value is called a “canary” sometimes, it is related to the miners’ canary\(^7\), they were used by miners in the past days in order to detect poisonous gases quickly. Canaries are very sensitive to mine gases, they become very agitated in case of danger, or even die.

If we compile our very simple array example (18.1 on page 359) in MSVC with RTC1 and RTCs option, you can see a call to @RTC_CheckStackVars@8 a function at the end of the function that checks if the “canary” is correct.

Let's see how GCC handles this. Let's take an alloca() (5.2.4 on page 43) example:

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

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

By default, without any additional options, GCC 4.7.3 inserts a “canary” check into the code:

```
Listing 18.7: GCC 4.7.3
.LC0:
    .string "hi! %d, %d, %d\n"
f:
    push   ebp
    mov    ebp, esp
    push   ebx
```

\(^7\) [wikipedia.org/wiki/Domestic_canary#Miner.27s_canary](wikipedia.org/wiki/Domestic_canary#Miner.27s_canary)
The random value is located in gs:20. It gets written on the stack and then at the end of the function the value in the stack is compared with the correct “canary” in gs:20. If the values are not equal, the \texttt{__stack_chk_fail} function is called and we can see in the console something like that (Ubuntu 13.04 x86):

\texttt{buffer overflow detected}: ./2_1 terminated

\texttt{Memory map:}

08048000-08049000 r-xp 00000000 08:01 2097586 /home/dennis/2

\_1
gs is the so-called segment register. These registers were used widely in MS-DOS and DOS-extenders times. Today, its function is different. To say it briefly, the gs register in Linux always points to the TLS (65 on page 965)—some information specific to thread is stored there. By the way, in win32 the fs register plays the same role, pointing to TIB\(^8\)^ \(^9\).

More information can be found in the Linux kernel source code (at least in 3.11 version), in arch/x86/include/asm/stackprotector.h this variable is described in the comments.

\(^8\)Thread Information Block
\(^9\)wikipedia.org/wiki/Win32_Thread_Information_Block
18.3.1 Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

Let's get back to our simple array example (18.1 on page 359), again, now we can see how LLVM checks the correctness of the "canary":

```assembly
_Pmain_

var_64  = -0x64
var_60  = -0x60
var_5C  = -0x5C
var_58  = -0x58
var_54  = -0x54
var_50  = -0x50
var_4C  = -0x4C
var_48  = -0x48
var_44  = -0x44
var_40  = -0x40
var_3C  = -0x3C
var_38  = -0x38
var_34  = -0x34
var_30  = -0x30
var_2C  = -0x2C
var_28  = -0x28
var_24  = -0x24
var_20  = -0x20
var_1C  = -0x1C
var_18  = -0x18
canary  = -0x14
var_10  = -0x10

PUSH   {R4-R7,LR}
ADD    R7, SP, #0xC
STR.W  R8, [SP,#0xC+var_10]!
SUB    SP, SP, #0x54
MOVW   R0, #aobjc_methtype ; "objc_methtype"
MOVS   R2, #0
MOVT.W R0, #0
MOVS   R5, #0
ADD    R0, PC
LDR.W  R8, [R0]
LDR.W  R0, [R8]
STR    R0, [SP,#0x64+canary]
MOVS   R0, #2
STR    R2, [SP,#0x64+var_64]
STR    R0, [SP,#0x64+var_60]
MOVS   R0, #4
STR    R0, [SP,#0x64+var_5C]
MOVS   R0, #6
```
```assembly
STR  R0, [SP,#0x64+var_58]
MOV  R0, #8
STR  R0, [SP,#0x64+var_54]
MOV  R0, #0xA
STR  R0, [SP,#0x64+var_50]
MOV  R0, #0xC
STR  R0, [SP,#0x64+var_4C]
MOV  R0, #0xE
STR  R0, [SP,#0x64+var_48]
MOV  R0, #0x10
STR  R0, [SP,#0x64+var_44]
MOV  R0, #0x12
STR  R0, [SP,#0x64+var_40]
MOV  R0, #0x14
STR  R0, [SP,#0x64+var_3C]
MOV  R0, #0x16
STR  R0, [SP,#0x64+var_38]
MOV  R0, #0x18
STR  R0, [SP,#0x64+var_34]
MOV  R0, #0x1A
STR  R0, [SP,#0x64+var_30]
MOV  R0, #0x1C
STR  R0, [SP,#0x64+var_2C]
MOV  R0, #0x1E
STR  R0, [SP,#0x64+var_28]
MOV  R0, #0x20
STR  R0, [SP,#0x64+var_24]
MOV  R0, #0x22
STR  R0, [SP,#0x64+var_20]
MOV  R0, #0x24
STR  R0, [SP,#0x64+var_1C]
MOV  R0, #0x26
STR  R0, [SP,#0x64+var_18]
MOV  R4, 0xFD ; "a[%d]=%d\n"
MOV  R0, SP
ADDS R6, R0, #4
ADD  R4, PC
B    loc_2F1C

; second loop begin

loc_2F14
    ADDS R0, R5, #1
    LDR.W R2, [R6,R5,LSL#2]
    MOV  R5, R0

loc_2F1C
```
First of all, as we see, LLVM “unrolled” the loop and all values were written into an array one-by-one, pre-calculated, as LLVM concluded it can work faster. By the way, instructions in ARM mode may help to do this even faster, and finding this could be your homework.

At the function end we see the comparison of the “canaries”—the one in the local stack and the correct one, to which \texttt{R8} points. If they are equal to each other, a 4-instruction block is triggered by \texttt{ITTTT EQ}, which contains writing 0 in \texttt{R0}, the function epilogue and exit. If the “canaries” are not equal, the block being skipped, and the jump to \texttt{___stack_chk_fail} function will occur, which, perhaps, will halt execution.

### 18.4 One more word about arrays

Now we understand why it is impossible to write something like this in C/C++ code:

```c
void f(int size)
{
    int a[size];
    ...
}
```

That’s just because the compiler must know the exact array size to allocate space for it in the local stack layout on at the compiling stage.

If you need an array of arbitrary size, allocate it by using \texttt{malloc()} , then access the allocated memory block as an array of variables of the type you need.

Or use the C99 standard feature[ISO07, pp. 6.7.5/2], and it works like \texttt{alloca()} (5.2.4 on page 43) internally.
It's also possible to use garbage collecting libraries for C. And there are also libraries supporting smart pointers for C++.

### 18.5 Array of pointers to strings

Here is an example for an array of pointers.

**Listing 18.8: Get month name**

```c
#include <stdio.h>

const char* month1[] = {
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
};

// in 0..11 range
const char* get_month1 (int month)
{
    return month1[month];
}
```

#### 18.5.1 x64

**Listing 18.9: Optimizing MSVC 2013 x64**

<table>
<thead>
<tr>
<th>_DATA SEGMENT</th>
<th>FLAT:$SG3122</th>
<th>FLAT:$SG3123</th>
<th>FLAT:$SG3124</th>
<th>FLAT:$SG3125</th>
<th>FLAT:$SG3126</th>
<th>FLAT:$SG3127</th>
</tr>
</thead>
<tbody>
<tr>
<td>month1 DQ</td>
<td>DQ</td>
<td>DQ</td>
<td>DQ</td>
<td>DQ</td>
<td>DQ</td>
<td>DQ</td>
</tr>
</tbody>
</table>
The code is very simple:

- The first MOVSD instruction copies a 32-bit value from ECX (where month argument is passed) to RAX with sign-extension (because the month argument is of type int). The reason for the sign extension is that this 32-bit value is to be used in calculations with other 64-bit values. Hence, it has to be promoted to 64-bit.\(^\text{10}\)

- Then the address of the pointer table is loaded into RCX.

- Finally, the input value (month) is multiplied by 8 and added to the address. Indeed: we are in a 64-bit environment and all address (or pointers) require exactly 64 bits (or 8 bytes) for storage. Hence, each table element is 8 bytes wide. And that’s why to pick a specific element, month * 8 bytes has to be

\(^{10}\)It is somewhat weird, but negative array index could be passed here as month (negative array indices will have been explained later: 52 on page 835). And if this happens, the negative input value of int type is sign-extended correctly and the corresponding element before table is picked. It is not going to work correctly without sign-extension.
skipped from the start. That’s what MOV does. In addition, this instruction also loads the element at this address. For 1, an element would be a pointer to a string that contains “February”, etc.

Optimizing GCC 4.9 can do the job even better\textsuperscript{11}: 

\begin{verbatim}
Listing 18.10: Optimizing GCC 4.9 x64
movsx rdi, edi
mov rax, QWORD PTR month1[0+rdi*8]
ret
\end{verbatim}

32-bit MSVC

Let’s also compile it in the 32-bit MSVC compiler:

\begin{verbatim}
Listing 18.11: Optimizing MSVC 2013 x86
_month$ = 8
_get_month1 PROC
    mov eax, DWORD PTR _month$[esp-4]
    mov eax, DWORD PTR _month1[eax*4]
    ret 0
_get_month1 ENDP
\end{verbatim}

The input value does not need to be extended to 64-bit value, so it is used as is. And it’s multiplied by 4, because the table elements are 32-bit (or 4 bytes) wide.

18.5.2 32-bit ARM

ARM in ARM mode

\begin{verbatim}
Listing 18.12: Optimizing Keil 6/2013 (ARM mode)
get_month1 PROC
    LDR r1,|L0.100|
    LDR r0,[r1,r0,LSL #2]
    BX lr
ENDP
\end{verbatim}

\textsuperscript{11}“0+” was left in the listing because GCC assembler output is not tidy enough to eliminate it. It’s \textit{displacement}, and it’s zero here.
The address of the table is loaded in R1. All the rest is done using just one LDR instruction. Then input value `month` is shifted left by 2 (which is the same as multiplying by 4), then added to R1 (where the address of the table is) and then a table element is loaded from this address. The 32-bit table element is loaded into R0 from the table.

**ARM in Thumb mode**

The code is mostly the same, but less dense, because the LSL suffix cannot be specified in the LDR instruction here:

```
get_month1 PROC
    LSLS    r0,r0,#2
    LDR     r1,|L0.64|
    LDR     r0,[r1,r0]
    BX      lr
ENDP
```
Listing 18.13: Optimizing GCC 4.9 ARM64

def get_month1:
    adrp  x1, .LANCHOR0
    add  x1, x1, :lo12:.LANCHOR0
    ldr  x0, [x1,w0,sxtw 3]
    ret

.LANCHOR0 = . + 0
    .type month1, %object
    .size  month1, 96

month1:
    .xword  .LC2
    .xword  .LC3
    .xword  .LC4
    .xword  .LC5
    .xword  .LC6
    .xword  .LC7
    .xword  .LC8
    .xword  .LC9
    .xword  .LC10
    .xword  .LC11
    .xword  .LC12
    .xword  .LC13

.LC2:
    .string "January"

.LC3:
    .string "February"

.LC4:
    .string "March"

.LC5:
    .string "April"

.LC6:
    .string "May"

.LC7:
    .string "June"

.LC8:
    .string "July"

.LC9:
    .string "August"

.LC10:
    .string "September"

.LC11:
    .string "October"

.LC12:
    .string "November"
The address of the table is loaded in X1 using ADRP/ADD pair. Then corresponding element is picked using just one LDR, which takes W0 (the register where input argument month is), shifts it 3 bits to the left (which is the same as multiplying by 8), sign-extends it (this is what “sxtw” suffix implies) and adds to X0. Then the 64-bit value is loaded from the table into X0.

18.5.4 MIPS

Listing 18.14: Optimizing GCC 4.4.5 (IDA)

```
get_month1:
  ; load address of table into $v0:
  la       $v0, month1
  ; take input value and multiply it by 4:
  sll      $a0, 2
  ; sum up address of table and multiplied value:
  addu     $a0, $v0
  ; load table element at this address into $v0:
  lw       $v0, 0($a0)
  ; return
  jr        $ra
or        $at, $zero ; branch delay slot, NOP
```

```
.data # .data.rel.local
.globl month1
month1: .word aJanuary    # "January"
        .word aFebruary   # "February"
        .word aMarch      # "March"
        .word aApril      # "April"
        .word aMay        # "May"
        .word aJune       # "June"
        .word aJuly       # "July"
        .word aAugust     # "August"
        .word aSeptember  # "September"
        .word aOctober    # "October"
        .word aNovember   # "November"
        .word aDecember   # "December"

.data # .rodata.str1.4
aJanuary: .ascii "January"<0>
aFebruary: .ascii "February"<0>
aMarch: .ascii "March"<0>
aApril: .ascii "April"<0>
```
18.5.5 Array overflow

Our function accepts values in the range of 0..11, but what if 12 is passed? There is no element in table at this place. So the function will load some value which happens to be there, and return it. Soon after, some other function can try to get a text string from this address and may crash.

Let's compile the example in MSVC for win64 and open it in IDA to see what the linker has placed after the table:

Listing 18.15: Executable file in IDA

<table>
<thead>
<tr>
<th>Address</th>
<th>Data Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000140001003</td>
<td>dq offset aJanuary_1</td>
<td>January</td>
</tr>
<tr>
<td>000000000140001010</td>
<td>dq offset aFebruary_1</td>
<td>February</td>
</tr>
<tr>
<td>000000000140001018</td>
<td>dq offset aMarch_1</td>
<td>March</td>
</tr>
<tr>
<td>000000000140001020</td>
<td>dq offset aApril_1</td>
<td>April</td>
</tr>
<tr>
<td>000000000140001028</td>
<td>dq offset aMay_1</td>
<td>May</td>
</tr>
<tr>
<td>000000000140001030</td>
<td>dq offset aJune_1</td>
<td>June</td>
</tr>
<tr>
<td>000000000140001032</td>
<td>dq offset aJuly_1</td>
<td>July</td>
</tr>
<tr>
<td>000000000140001034</td>
<td>dq offset aAugust_1</td>
<td>August</td>
</tr>
<tr>
<td>000000000140001036</td>
<td>dq offset aSeptember_1</td>
<td>September</td>
</tr>
<tr>
<td>000000000140001038</td>
<td>dq offset aOctober_1</td>
<td>October</td>
</tr>
<tr>
<td>000000000140001040</td>
<td>dq offset aNovember_1</td>
<td>November</td>
</tr>
<tr>
<td>000000000140001042</td>
<td>dq offset aDecember_1</td>
<td>December</td>
</tr>
<tr>
<td>000000000140001044</td>
<td>aJanuary_1</td>
<td>'January',0</td>
</tr>
<tr>
<td>000000000140001046</td>
<td>aFebruary_1</td>
<td>'February',0</td>
</tr>
<tr>
<td>000000000140001048</td>
<td>aMarch_1</td>
<td>'March',0</td>
</tr>
<tr>
<td>000000000140001050</td>
<td>aApril_1</td>
<td>'April',0</td>
</tr>
<tr>
<td>000000000140001052</td>
<td>aMay_1</td>
<td>'May',0</td>
</tr>
<tr>
<td>000000000140001054</td>
<td>aJune_1</td>
<td>'June',0</td>
</tr>
<tr>
<td>000000000140001056</td>
<td>aJuly_1</td>
<td>'July',0</td>
</tr>
<tr>
<td>000000000140001058</td>
<td>aAugust_1</td>
<td>'August',0</td>
</tr>
<tr>
<td>000000000140001060</td>
<td>aSeptember_1</td>
<td>'September',0</td>
</tr>
<tr>
<td>000000000140001062</td>
<td>aOctober_1</td>
<td>'October',0</td>
</tr>
<tr>
<td>000000000140001064</td>
<td>aNovember_1</td>
<td>'November',0</td>
</tr>
<tr>
<td>000000000140001066</td>
<td>aDecember_1</td>
<td>'December',0</td>
</tr>
</tbody>
</table>

391
Month names are came right after. Our program is tiny, so there isn’t much data to pack in the data segment, so it just the month names. But it should be noted that there might be really anything that linker has decided to put by chance.

So what if 12 is passed to the function? The 13th element will be returned. Let’s see how the CPU treats the bytes there as a 64-bit value:

Listing 18.16: Executable file in IDA

```asm
off_140011000 dq offset qword_140011060 ; DATA XREF: .text
    dq offset aFebruary_1 ; "February"
    dq offset aMarch_1 ; "March"
    dq offset aApril_1 ; "April"
    dq offset aMay_1 ; "May"
    dq offset aJune_1 ; "June"
    dq offset aJuly_1 ; "July"
    dq offset aAugust_1 ; "August"
    dq offset aSeptember_1 ; "September"
    dq offset aOctober_1 ; "October"
    dq offset aNovember_1 ; "November"
    dq offset aDecember_1 ; "December"
qword_140011060 dq 797261756E614Ah ; DATA XREF: 
    sub_140001020+4
aFebruary_1 db 'February',0 ; DATA XREF: .data
    dq offset aFebruary_1 ; "February"
    dq offset aMarch_1 ; "March"
    dq offset aApril_1 ; "April"
    dq offset aMay_1 ; "May"
    dq offset aJune_1 ; "June"
    dq offset aJuly_1 ; "July"
    dq offset aAugust_1 ; "August"
    dq offset aSeptember_1 ; "September"
    dq offset aOctober_1 ; "October"
    dq offset aNovember_1 ; "November"
    dq offset aDecember_1 ; "December"
```

And this is 0x797261756E614A. Soon after, some other function (presumably, one that processes strings) may try to read bytes at this address, expecting a C-string there. Most likely it is about to crash, because this value doesn’t look like a valid address.

**Array overflow protection**

If something can go wrong, it will

Murphy’s Law

It’s a bit naïve to expect that every programmer who use your function or library will never pass an argument larger than 11. There exists the philosophy that says “fail early and fail loudly” or “fail-fast”, which teaches to report problems as early
as possible and stop. One such method in C/C++ is assertions. We can modify our program to fail if an incorrect value is passed:

Listing 18.17: assert() added

```c
const char* get_month1_checked (int month)
{
    assert (month<12);
    return month1[month];
}
```

The assertion macro checks for valid values at every function start and fails if the expression is false.

Listing 18.18: Optimizing MSVC 2013 x64

```c
$SG3143 DB 'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
    DB 'c', 00H, 00H, 00H
$SG3144 DB 'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H,
    DB '1', 00H, '2', 00H, 00H, 00H
month$ = 48
get_month1_checked PROC
$LN5:
    push rbx
    sub rsp, 32
    movsxd rbx, ecx
    cmp ebx, 12
    jl SHORT $LN3@get_month1
    lea rdx, OFFSET FLAT:$SG3143
    lea rcx, OFFSET FLAT:$SG3144
    mov r8d, 29
    call _wassert
$LN3@get_month1:
    lea rcx, OFFSET FLAT:month1
    mov rax, QWORD PTR [rcx+rbx*8]
    add rsp, 32
    pop rbx
    ret 0
get_month1_checked ENDP
```

In fact, assert() is not a function, but macro. It checks for a condition, then passes also the line number and file name to another function which reports this information to the user. Here we see that both file name and condition are encoded in UTF-16. The line number is also passed (it's 29).

This mechanism is probably the same in all compilers. Here is what GCC does:
So the macro in GCC also passes the function name for convenience.

Nothing is really free, and this is true for the sanitizing checks as well. They make your program slower, especially if the assert() macros used in small time-critical functions. So MSVC, for example, leaves the checks in debug builds, but in release builds they all disappear.

Microsoft Windows NT kernels come in “checked” and “free” builds\(^\text{12}\). The first has validation checks (hence, “checked”), the second one doesn’t (hence, “free” of checks).

### 18.6 Multidimensional arrays

Internally, a multidimensional array is essentially the same thing as a linear array. Since the computer memory is linear, it is an one-dimensional array. For convenience, this multi-dimensional array can be easily represented as one-dimensional.

For example, this is how the elements of the 3x4 array are placed in one-dimensional array of 12 cells:

CHAPTER 18. ARRAYS

<table>
<thead>
<tr>
<th>Offset in memory</th>
<th>array element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0][0]</td>
</tr>
<tr>
<td>1</td>
<td>[0][1]</td>
</tr>
<tr>
<td>2</td>
<td>[0][2]</td>
</tr>
<tr>
<td>3</td>
<td>[0][3]</td>
</tr>
<tr>
<td>4</td>
<td>[1][0]</td>
</tr>
<tr>
<td>5</td>
<td>[1][1]</td>
</tr>
<tr>
<td>6</td>
<td>[1][2]</td>
</tr>
<tr>
<td>7</td>
<td>[1][3]</td>
</tr>
<tr>
<td>8</td>
<td>[2][0]</td>
</tr>
<tr>
<td>9</td>
<td>[2][1]</td>
</tr>
<tr>
<td>10</td>
<td>[2][2]</td>
</tr>
<tr>
<td>11</td>
<td>[2][3]</td>
</tr>
</tbody>
</table>

Table 18.1: Two-dimensional array represented in memory as one-dimensional

Here is how each cell of 3*4 array are placed in memory:

```
  0  1  2  3
 4  5  6  7
 8  9 10 11
```

Table 18.2: Memory addresses of each cell of two-dimensional array

So, in order to calculate the address of the element we need, we first multiply the first index by 4 (array width) and then add the second index. That’s called *row-major order*, and this method of array and matrix representation is used in at least C/C++ and Python. The term *row-major order* in plain English language means: “first, write the elements of the first row, then the second row …and finally the elements of the last row”.

Another method for representation is called *column-major order* (the array indices are used in reverse order) and it is used at least in FORTRAN, MATLAB and R. *column-major order* term in plain English language means: “first, write the elements of the first column, then the second column …and finally the elements of the last column”.

Which method is better? In general, in terms of performance and cache memory, the best scheme for data organization is the one, in which the elements are accessed sequentially. So if your function accesses data per row, *row-major order* is better, and vice versa.
18.6.1 Two-dimensional array example

We are going to work with an array of type `char`, which implies that each element requires only one byte in memory.

Row filling example

Let’s fill the second row with these values 0..3:

```c
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // clear array
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // fill second row by 0..3:
    for (y=0; y<4; y++)
        a[1][y]=y;
}
```

All three rows are marked with red. We see that second row now has values 0, 1, 2 and 3:

![Figure 18.7: OllyDbg: array is filled](image)

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex_dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>00C33370</td>
<td>00 00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00C33380</td>
<td>02 00 00 00 00 C3 66 47 4E C3 66 47 4E 00</td>
</tr>
<tr>
<td>00C33390</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00C333A0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00C333B0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
Column filling example

Let's fill the third column with values: 0..2:

Listing 18.21: Column filling example

```c
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // clear array
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // fill third column by 0..2:
    for (x=0; x<3; x++)
        a[x][2]=x;
}
```

The three rows are also marked in red here. We see that in each row, at third position these values are written: 0, 1 and 2.

Figure 18.8: OllyDbg: array is filled

18.6.2 Access two-dimensional array as one-dimensional

We can be easily assured that it's possible to access a two-dimensional array as one-dimensional array in at least two ways:

```c
#include <stdio.h>

char a[3][4];
```
char get_by_coordinates1 (char array[3][4], int a, int b)
{
    return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
    // treat input array as one-dimensional
    // 4 is array width here
    return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
    // treat input array as pointer,
    // calculate address, get value at it
    // 4 is array width here
    return *(array+a*4+b);
};

int main()
{
    a[2][3]=123;
    printf ("%d\n", get_by_coordinates1(a, 2, 3));
    printf ("%d\n", get_by_coordinates2(a, 2, 3));
    printf ("%d\n", get_by_coordinates3(a, 2, 3));
};

Compile and run it: it shows correct values.
What MSVC 2013 did is fascinating, all three routines are just the same!

Listing 18.22: Optimizing MSVC 2013 x64

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=address of array
; RDX=a
; R8=b
    movsxd rax, r8d
; EAX=b
    movsxd r9, edx
; R9=a
    add rax, rcx
; RAX=b+address of array
    movzx eax, BYTE PTR [rax+r9*4]
CHAPTER 18. ARRAYS

get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24

get_by_coordinates2 PROC
  movsxd rax, r8d
  movsxd r9, edx
  add rax, rcx
  movzx eax, BYTE PTR [rax+r9*4]
  ret 0
get_by_coordinates2 ENDP

get_by_coordinates1 PROC
  movsxd rax, r8d
  movsxd r9, edx
  add rax, rcx
  movzx eax, BYTE PTR [rax+r9*4]
  ret 0
get_by_coordinates1 ENDP

GCC also generates equivalent routines, but slightly different:

Listing 18.23: Optimizing GCC 4.9 x64

get_by_coordinates1:
  ; sign-extend input 32-bit int values "a" and "b" to 64-bit ones
  movsx  rsi, esi
  movsx  rdx, edx
  lea    rax, [rdi+rsi*4]
  ; RAX=RDI+RSI*4=address of array+a*4
  movzx  eax, BYTE PTR [rax+rdx]
  ; AL=load byte at address RAX+RDX=address of array+a*4+b
  ret

get_by_coordinates2:
  lea    eax, [rdx+rsi*4]
  ; RAX=RDX+RSI*4=b+a*4

399
18.6.3 Three-dimensional array example

It's thing in multidimensional arrays. Now we are going to work with an array of type `int`: each element requires 4 bytes in memory.

Let's see:

Listing 18.24: simple example

```c
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value) {
    a[x][y][z]=value;
};
```

x86

We get (MSVC 2010):

Listing 18.25: MSVC 2010

```
_DATA SEGMENT
COMM    _a:DWORD:01770H
_DATA ENDS
PUBLIC    _insert
```
CHAPTER 18. ARRAYS

Nothing special. For index calculation, three input arguments are used in the formula $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$, to represent the array as multidimensional. Do not forget that the int type is 32-bit (4 bytes), so all coefficients must be multiplied by 4.

Listing 18.26: GCC 4.4.1

| public insert |
| proc near |

| x  | = dword ptr 8 |
| y  | = dword ptr 0Ch |
| z  | = dword ptr 10h |
| value | = dword ptr 14h |

| push ebp |
| mov ebp, esp |
| push ebx |
| mov ebx, [ebp+x] |
| mov eax, [ebp+y] |
| mov ecx, [ebp+z] |
| lea edx, [eax+eax] |
| mov eax, edx |
| shl eax, 4 |

$\downarrow <<4 = y^2*16 = y*32$
The GCC compiler does it differently. For one of the operations in the calculation (30y), GCC produces code without multiplication instructions. This is how it done:  

\[(y + y) \ll 4 - (y + y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y.\]  

Thus, for the 30y calculation, only one addition operation, one bitwise shift operation and one subtraction operation are used. This works faster.

**ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb mode)**

Listing 18.27: Non-optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, [ebp+value]</td>
<td>mov eax, [ebp+value]</td>
</tr>
<tr>
<td>mov dword ptr ds:a[edx*4], eax</td>
<td>*(a+edx)=value</td>
</tr>
</tbody>
</table>

; allocate place in local stack for 4 values of int type
SUB SP, SP, #0x10
MOV R9, 0xFC2 ; a
ADD R9, PC
LDR.W R9, [R9]
STR R0, [SP,#0x10+x]
STR R1, [SP,#0x10+y]
STR R2, [SP,#0x10+z]
STR R3, [SP,#0x10+value]
LDR R0, [SP,#0x10+value]
LDR R1, [SP,#0x10+z]
LDR R2, [SP,#0x10+y]
LDR R3, [SP,#0x10+x]
MOV R12, 2400
MUL.W R3, R3, R12  
ADD R3, R9  
MOV R9, 120  
MUL.W R2, R2, R9  
ADD R2, R3  
LSLS R1, R1, #2 ; R1=R1<<2  
ADD R1, R2  
STR R0, [R1] ; R1 - address of array element  
; deallocate chunk in local stack, allocated for 4 values of int type  
ADD SP, SP, #0x10  
BX LR  

Non-optimizing LLVM saves all variables in local stack, which is redundant. The address of the array element is calculated by the formula we already saw.

**ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)**

Listing 18.28: Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

```assembly
_insert
MOVW R9, #0x10FC
MOV.W R12, #2400
MOVT.W R9, #0
RSB.W R1, R1, R1,LSL#4 ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD R9, PC ; R9 = pointer to a array
LDR.W R9, [R9]
MLA.W R0, R0, R12, R9 ; R0 - x, R12 - 2400, R9 - pointer to a. R0=x*2400 + ptr to a
ADD.W R0, R0, R1,LSL#3 ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + y*30*4 + x*600*4
STR.W R3, [R0,R2,LSL#2] ; R2 - z, R3 - value. address=R0+z*4  
BX LR
```

The tricks for replacing multiplication by shift, addition and subtraction which we already saw are also present here.

Here we also see a new instruction for us: **RSB (Reverse Subtract)**. It works just as **SUB**, but it swaps its operands with each other before execution. Why? **SUB** and RSB are instructions, to the second operand of which shift coefficient may be applied: (LSL#4). But this coefficient can be applied only to second operand.
That’s fine for commutative operations like addition or multiplication (operands may be swapped there without changing the result). But subtraction is a non-commutative operation, so RSB exist for these cases.

The LDR.W R9, [R9] instruction works like LEA (A.6.2 on page 1304) in x86, but it does nothing here, it is redundant. Apparently, the compiler did not optimize it out.

**MIPS**

My example is tiny, so the GCC compiler decided to put the $a$ array into the 64KiB area addressable by the Global Pointer.

Listing 18.29: Optimizing GCC 4.4.5 (IDA)

```assembly
insert:
; $a0=x
; $a1=y
; $a2=z
; $a3=value
    sll  $v0, $a0, 5
    sll  $a0, 3
    addu $a0, $v0
    sll  $v1, $a1, 5
    sll  $v0, $a0, 4
    sll  $a1, 1
    subu $a1, $v1, $a1
    subu $a0, $v0, $a0
    la   $gp, __gnu_local_gp
    addu $a0, $a1, $a0
    addu $a0, $a2
    lw   $v0, (a & 0xFFFF)($gp)
    sll  $a0, 2
    addu $a0, $v0, $a0
```

18.6.4 More examples

The computer screen is represented as a 2D array, but the video-buffer is a linear 1D array. We talk about it here: 83.2 on page 1205.

18.7 Pack of strings as a two-dimensional array

Let's revisit the function that returns the name of a month: listing 18.8. As you see, at least one memory load operation is needed to prepare a pointer to the string that's the month's name. Is it possible to get rid of this memory load operation? In fact yes, if you represent the list of strings as a two-dimensional array:

```c
#include <stdio.h>
#include <assert.h>

const char month2[12][10]=
{
    { 'J','a','n','u','a','r','y', 0, 0, 0 },
    { 'F','e','b','r','u','a','r','y', 0, 0, 0 },
    { 'M','a','r','c','h', 0, 0, 0, 0, 0 },
    { 'A','p','r','i','l', 0, 0, 0, 0, 0 },
    { 'M','a','y', 0, 0, 0, 0, 0, 0, 0 },
    { 'J','u','n','e', 0, 0, 0, 0, 0, 0 },
    { 'J','u','l','y', 0, 0, 0, 0, 0, 0 },
    { 'A','u','g','u','s','t', 0, 0, 0 },
    { 'S','e','p','t','e','m','b','e','r', 0 },
    { 'O','c','t','o','b','e','r', 0, 0, 0 },
    { 'N','o','v','e','m','b','e','r', 0, 0 },
    { 'D','e','c','e','m','b','e','r', 0, 0 }
};
// in 0..11 range
const char* get_month2 (int month)
{
    return &month2[month][0];
}
```
Here is what we’ve get:

Listing 18.30: Optimizing MSVC 2013 x64

```assembly
get_month2 PROC
    ; sign-extend input argument and promote to 64-bit value
    movsxd rax, ecx
    lea rcx, QWORD PTR [rax+rax*4]
    ; RCX=month+month*4=month*5
    lea rax, OFFSET FLAT:month2
    ; RAX=pointer to table
    lea rax, QWORD PTR [rax+rcx*2]
    ; RAX=pointer to table + RCX*2=pointer to table + month*10
    ret 0
get_month2 ENDP
```

There are no memory accesses at all. All this function does is to calculate a point at which the first character of the name of the month is: `pointer_to_the_table + month * 10`. There are also two LEA instructions, which effectively work as several MUL and MOV instructions.

The width of the array is 10 bytes. Indeed, the longest string here—“September”—is 9 bytes, and plus the terminating zero is 10 bytes. The rest of the month names are padded by zero bytes, so they all occupy the same space (10 bytes). Thus, our function works even faster, because all string start at an address which can be calculated easily.

Optimizing GCC 4.9 can do it even shorter:

Listing 18.31: Optimizing GCC 4.9 x64

```assembly
movsx rdi, edi
lea rax, [rdi+rdi*4]
lea rax, month2[rax+rax]
ret
```
LEA is also used here for multiplication by 10.

Non-optimizing compilers do multiplication differently.

### Listing 18.32: Non-optimizing GCC 4.9 x64

```assembly
get_month2:
push  rbp
mov   rbp, rsp
mov   DWORD PTR [rbp-4], edi
mov   eax, DWORD PTR [rbp-4]
movsx rdx, eax
; RDX = sign-extended input value
mov   rax, rdx
; RAX = month
sal   rax, 2
; RAX = month<<2 = month*4
add   rax, rdx
; RAX = RAX+RDX = month*4+month = month*5
add   rax, rax
; RAX = RAX*2 = month*5*2 = month*10
add   rax, OFFSET FLAT:month2
; RAX = month*10 + pointer to the table
pop   rbp
ret
```

Non-optimizing MSVC just uses IMUL instruction:

### Listing 18.33: Non-optimizing MSVC 2013 x64

```assembly
month$ = 8
get_month2 PROC
    mov    DWORD PTR [rsp+8], ecx
movsxrd xax, DWORD PTR month$[rsp]
    ; RAX = sign-extended input value into 64-bit one
imul   rax, rax, 10
    ; RAX = RAX*10
lea    rcx, OFFSET FLAT:month2
    ; RCX = pointer to the table
add    rcx, rax
    ; RCX = RCX+RAX = pointer to the table+month*10
mov    rax, rcx
    ; RAX = pointer to the table+month*10
mov    ecx, 1
    ; RCX = 1
imul   rcx, rcx, 0
    ; RCX = 1*0 = 0
add    rax, rcx
    ; RAX = pointer to the table+month*10 + 0 = pointer to the table+month*10
POP RCX
```

407
But one thing is weird here: why add multiplication by zero and adding zero to the final result? This looks like a compiler code generator quirk, which wasn’t caught by the compiler’s tests (the resulting code works correctly, after all). We intentionally consider such pieces of code so the reader would understand, that sometimes one shouldn’t puzzle over such compiler artifacts.

### 18.7.1 32-bit ARM

Optimizing Keil for Thumb mode uses the multiplication instruction MULS:

**Listing 18.34: Optimizing Keil 6/2013 (Thumb mode)**

```assembly
; R0 = month
  MOV S r1,#0xa
; R1 = 10
  MUL S r0,r1,r0
; R0 = R1*R0 = 10*month
  LDR r1,|L0.68|
; R1 = pointer to the table
  ADD S r0,r0,r1
; R0 = R0+R1 = 10*month + pointer to the table
  BX lr
```

Optimizing Keil for ARM mode uses add and shift operations:

**Listing 18.35: Optimizing Keil 6/2013 (ARM mode)**

```assembly
; R0 = month
  LDR r1,|L0.104|
; R1 = pointer to the table
  ADD r0,r0,r0,LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = month*5
  ADD r0,r1,r0,LSL #1
; R0 = R1+R0<<2 = pointer to the table + month*5*2 = pointer to the table + month*10
  BX lr
```

### 18.7.2 ARM64

**Listing 18.36: Optimizing GCC 4.9 ARM64**

```assembly
; W0 = month
```
SXTW is used for sign-extension and promoting input 32-bit value into a 64-bit one and storing it in X0. ADRP/ADD pair is used for loading the address of the table. The ADD instructions also has a LSL suffix, which helps with multiplications.

18.7.3 MIPS

Listing 18.37: Optimizing GCC 4.4.5 (IDA)

```
globl get_month2
get_month2:
    ; $a0=month
        sll $v0, $a0, 3
    ; $v0 = $a0<<3 = month*8
        sll $a0, 1
    ; $a0 = $a0<<1 = month*2
        addu $a0, $v0
    ; $a0 = month*2+month*8 = month*10
    ; load address of the table:
        la $v0, month2
    ; sum up table address and index we calculated and return:
        jr $ra
        addu $v0, $a0

month2:   .ascii "January"<0>
          .byte 0, 0
aFebruary: .ascii "February"<0>
           .byte 0
aMarch:   .ascii "March"<0>
          .byte 0, 0, 0, 0
aApril:   .ascii "April"<0>
          .byte 0, 0, 0, 0
aMay:     .ascii "May"<0>
          .byte 0, 0, 0, 0, 0
aJune:    .ascii "June"<0>
          .byte 0, 0, 0, 0, 0
```
18.7.4 Conclusion

This is a bit old-school technique to store text strings. You may find a lot of it in Oracle RDBMS, for example. It’s hard to say if it’s worth doing on modern computers. Nevertheless, it was a good example of arrays, so it was added to this book.

18.8 Conclusion

An array is a pack of values in memory located adjacently. It’s true for any element type, including structures. Access to a specific array element is just a calculation of its address.

18.9 Exercises

- [http://challenges.re/62](http://challenges.re/62)
- [http://challenges.re/63](http://challenges.re/63)
- [http://challenges.re/64](http://challenges.re/64)
- [http://challenges.re/65](http://challenges.re/65)
- [http://challenges.re/66](http://challenges.re/66)
Chapter 19

Manipulating specific bit(s)

A lot of functions define their input arguments as flags in bit fields. Of course, they could be substituted by a set of bool-typed variables, but it is not frugally.

19.1 Specific bit checking

19.1.1 x86

Win32 API example:

```c
HANDLE fh;

fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, \\
FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, \\
, NULL);
```

We get (MSVC 2010):

```
Listing 19.1: MSVC 2010

<table>
<thead>
<tr>
<th>Push</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>0</td>
</tr>
<tr>
<td>push</td>
<td>128</td>
</tr>
<tr>
<td>push</td>
<td>00000080H</td>
</tr>
<tr>
<td>push</td>
<td>4</td>
</tr>
<tr>
<td>push</td>
<td>0</td>
</tr>
<tr>
<td>push</td>
<td>1</td>
</tr>
<tr>
<td>push</td>
<td>-1073741824</td>
</tr>
<tr>
<td>push</td>
<td>c00000000H</td>
</tr>
<tr>
<td>push</td>
<td>OFFSET $SG78813</td>
</tr>
</tbody>
</table>
```


call DWORD PTR __imp__CreateFileA@28
mov DWORD PTR _fh$[ebp], eax

Let's take a look in WinNT.h:

Listing 19.2: WinNT.h

```
#define GENERIC_READ   (0x80000000L)
#define GENERIC_WRITE  (0x40000000L)
#define GENERIC_EXECUTE (0x20000000L)
#define GENERIC_ALL    (0x10000000L)
```

Everything is clear, GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000, and that value is used as the second argument for the CreateFile() function.

How would CreateFile() check these flags? If we look in KERNEL32.DLL in Windows XP SP3 x86, we'll find this fragment of code in CreateFileW:

Listing 19.3: KERNEL32.DLL (Windows XP SP3 x86)

```
.test:7C83D429          test    byte ptr [ebp+⤦dwDesiredAccess+3], 40h
    .text:7C83D42D        mov     [ebp+var_8], 1
    .text:7C83D434        jz     short loc_7C83D417
    .text:7C83D436        jmp     loc_7C810817
```

Here we see the TEST instruction, however it doesn't take the whole second argument, but only the most significant byte (ebp+dwDesiredAccess+3) and checks it for flag 0x40 (which implies the GENERIC_WRITE flag here) TEST is basically the same instruction as AND, but without saving the result (recall the fact CMP is merely the same as SUB, but without saving the result (7.3.1 on page 118)).

The logic of this code fragment is as follows:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

If AND instruction leaves this bit, the ZF flag is to be cleared and the JZ conditional jump is not to be triggered. The conditional jump is triggered only if the 0x40000000 bit is absent in dwDesiredAccess variable —then the result of AND is 0, ZF is to be set and the conditional jump is to be triggered.

Let's try GCC 4.4.1 and Linux:

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

---

void main()
{
    int handle;
    handle=open("file", O_RDWR | O_CREAT);
};

We get:

Listing 19.4: GCC 4.4.1

    public main
    main proc near
    var_20 = dword ptr -20h
    var_1C = dword ptr -1Ch
    var_4 = dword ptr -4
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    sub esp, 20h
    mov [esp+20h+var_1C], 42h
    mov [esp+20h+var_20], offset aFile ; "file"
    call _open
    mov [esp+20h+var_4], eax
    leave
    retn
    main endp

If we take a look in the open() function in the libc.so.6 library, it is only a syscall:

Listing 19.5: open() (libc.so.6)

    .text:000BE69B mov edx, [esp+4+mode] ; mode
    .text:000BE69F mov ecx, [esp+4+flags] ; flags
    .text:000BE6A3 mov ebx, [esp+4+filename] ; filename
    .text:000BE6A7 mov eax, 5
    .text:000BE6AC int 80h ; LINUX

So, the bit fields for open() are apparently checked somewhere in the Linux kernel.

Of course, it is easy to download both Glibc and the Linux kernel source code, but we are interested in understanding the matter without it.
So, as of Linux 2.6, when the `sys_open` syscall is called, control eventually passes to `do_sys_open`, and from there—to the `do_filp_open()` function (it's located in the kernel source tree in `fs/namei.c`).

N.B. Aside from passing arguments via the stack, there is also a method of passing some of them via registers. This is also called fastcall (64.3 on page 954). This works faster since CPU does not need to access the stack in memory to read argument values. GCC has the option `regparm`\(^2\), through which it's possible to set the number of arguments that can be passed via registers.

The Linux 2.6 kernel is compiled with `-mregparm=3` option \(^3\) \(^4\).

What this means to us is that the first 3 arguments are to be passed via registers EAX, EDX and ECX, and the rest via the stack. Of course, if the number of arguments is less than 3, only part of registers set is to be used.

So, let's download Linux Kernel 2.6.31, compile it in Ubuntu: `make vmlinux`, open it in IDA, and find the `do_filp_open()` function. At the beginning, we see (the comments are mine):

```
Listing 19.6: do_filp_open() (linux kernel 2.6.31)
do_filp_open proc near
...
push    ebp
mov     ebp, esp
push    edi
push    esi
push    ebx
mov     ebx, ecx
add     ebx, 1
sub     esp, 98h
mov     esi, [ebp+arg_4] ; acc_mode (5th arg)
test    bl, 3
mov     [ebp+var_80], eax ; dfd (1th arg)
mov     [ebp+var_7C], edx ; pathname (2th arg)
mov     [ebp+var_78], ecx ; open_flag (3th arg)
jnz short loc_C01EF684
mov     ebx, ecx       ; ebx <- open_flag
```

GCC saves the values of the first 3 arguments in the local stack. If that wasn't done, the compiler would not touch these registers, and that would be too tight environment for the compiler's register allocator.

Let's find this fragment of code:

\(^2\) [ohse.de/uwe/articles/gcc-attributes.html#func-regparm](https://ohse.de/uwe/articles/gcc-attributes.html#func-regparm)

\(^3\) [kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f](https://kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f)

\(^4\) See also `arch/x86/include/asm/calling.h` file in kernel tree
0x40—is what the O_CREAT macro equals to. open_flag gets checked for the presence of the 0x40 bit, and if this bit is 1, the next JNZ instruction is triggered.

### 19.1.2 ARM

The O_CREAT bit is checked differently in Linux kernel 3.8.0.

Listing 19.8: linux kernel 3.8.0

```c
struct file *do_filp_open(int dfd, struct filename *pathname,
const struct open_flags *op)
{
...
    filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
...
}

static struct file *path_openat(int dfd, struct filename *
    pathname,
    struct nameidata *nd, const struct open_flags *
    op, int flags)
{
...
    error = do_last(nd, &path, file, op, &opened, pathname);
    ...
}

static int do_last(struct nameidata *nd, struct path *
    path,
    struct file *file, const struct open_flags *
    op,
```
int *opened, struct filename *name)
{
...
    if (!((open_flag & O_CREAT))) {
        ...
        error = lookup_fast(nd, path, &inode);
        ...
    } else {
        ...
        error = complete_walk(nd);
    }
...}

Here is how the kernel compiled for ARM mode looks in IDA:

Listing 19.9: do_last() (vmlinux)

... .text:C0169EA8 MOV R9, R3 ; R3 - (4th argument) open_flag
     ↓ .text:C0169ED4 LDR R6, [R9] ; R6 - open_flag
     ↓ .text:C0169F68 TST R6, #0x40 ; jumptable C0169F00 default case
          ↓ .text:C0169F6C BNE 0x0
          ↓ .text:C0169F70 LDR R2, [R4,#0x10]  
          ↓ .text:C0169F74 ADD R12, R4, #8  
          ↓ .text:C0169F78 LDR R3, [R4,#0xC]  
          ↓ .text:C0169F7C MOV R0, R4  
          ↓ .text:C0169F80 STR R12, [R11,#var_50
          ↓ .text:C0169F84 LDRB R3, [R2,R3]  
          ↓ .text:C0169F88 MOV R2, R8  
          ↓ .text:C0169F8C CMP R3, #0  
          ↓ .text:C0169F90 ORRNNE R1, R1, #3  
          ↓ .text:C0169F94 STRNE R1, [R4,#0x24]  
          ↓ .text:C0169F98 ANDS R3, R6, #0
          ↓ x200000
          ↓ .text:C0169F9C MOV R1, R12  
          ↓ .text:C0169FA0 LDRNE R3, [R4,#0x24]  
          ↓ .text:C0169FA4 ANDNNE R3, R3, #1  
          ↓ .text:C0169FA8 EORNNE R3, R3, #1  
          ↓ .text:C0169FAC STR R3, [R11,#var_54
          ↓ ]
TST is analogous to the TEST instruction in x86.

We can “spot” visually this code fragment by the fact the `lookup_fast()` is to be executed in one case and `complete_walk()` in the other. This corresponds to the source code of the `do_last()` function.

The `O_CREAT` macro equals to 0x40 here too.

### 19.2 Setting and clearing specific bits

For example:

```c
#include <stdio.h>

#define IS_SET(flag, bit)  ((flag) & (bit))
#define SET_BIT(var, bit)  ((var) |= (bit))
#define REMOVE_BIT(var, bit)  ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};

int main()
{
    f(0x12340678);
};
```
19.2.1  x86

Non-optimizing MSVC

We get (MSVC 2010):

```
Listing 19.10: MSVC 2010

_rt$ = -4    ; size = 4
_a$ = 8      ; size = 4
_f PROC
  push ebp
  mov ebp, esp
  push ecx
  mov eax, DWORD PTR _a$[ebp]
  mov DWORD PTR _rt$[ebp], eax
  mov ecx, DWORD PTR _rt$[ebp]
  or ecx, 16384 ; 00004000H
  mov DWORD PTR _rt$[ebp], ecx
  mov edx, DWORD PTR _rt$[ebp]
  and edx, -513 ; fffffdffH
  mov DWORD PTR _rt$[ebp], edx
  mov eax, DWORD PTR _rt$[ebp]
  mov esp, ebp
  pop ebp
  ret 0
_f ENDP
```

The OR instruction sets one bit to value while ignoring the rest.

AND resets one bit. It can be said that AND just copies all bits except one. Indeed, in the second AND operand only the bits that need to be saved are set, just the one do not want to copy is not (which is 0 in the bitmask). It is the easier way to memorize the logic.
Let’s try this example in OllyDbg. First, let’s see the binary form of the constants we are going to use:

0x200 (00000000000000000000000000100000000) (i.e., the 10th bit (counting from 1st)).

Inverted 0x200 is 0xFFFFFDFF (11111111111111111111111111111111).

0x4000 (0000000000000000000000000001000000000000) (i.e., the 15th bit).

The input value is: 0x12340678 (10010001101000000011001111000). We see how it’s loaded:

![OllyDbg: value is loaded into ECX](image)

Figure 19.1: OllyDbg: value is loaded into ECX
OR got executed:

![OllyDbg: OR executed screenshot](image)

15th bit is set: 0x12344678 (1001000110100010001100111100).

Figure 19.2: OllyDbg: OR executed
The value is reloaded again (because the compiler is not in optimizing mode):

Figure 19.3: OllyDbg: value was reloaded into EDX
AND got executed:

Figure 19.4: OllyDbg: AND executed

The 10th bit was cleared (or, in other words, all bits were left except the 10th) and the final value now is 0x1234478 (100100011010001000100111100).  

Optimizing MSVC

If we compile it in MSVC with optimization turned on (/Ox), the code is even shorter:

Listing 19.11: Optimizing MSVC

```assembly
_a$ = 8 ; size = 4
_f PROC
    mov eax, DWORD PTR _a$[esp-4]
    and eax, -513 ; fffffdffH
    or eax, 16384 ; 00004000H
    ret 0
_f ENDP
```

Non-optimizing GCC

Let’s try GCC 4.4.1 without optimization:

Listing 19.12: Non-optimizing GCC

```assembly
public f
f proc near
```


```assembly
var_4 = dword ptr -4
arg_0 = dword ptr 8

push ebp
mov ebp, esp
sub esp, 10h
mov eax, [ebp+arg_0]
mov [ebp+var_4], eax
or [ebp+var_4], 4000h
and [ebp+var_4], 0FFFFFFFDFFh
mov eax, [ebp+var_4]
leave
retn
```

That's shorter. It is worth noting the compiler works with the EAX register part via the AH register—that is the EAX register part from the 8th to the 15th bits included.

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX^x64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
N.B. The 16-bit CPU 8086 accumulator was named AX and consisted of two 8-bit halves—AL (lower byte) and AH (higher byte). In 80386 almost all registers were extended to 32-bit, the accumulator was named EAX, but for the sake of compatibility, its older parts may be still accessed as AX/AH/AL.

Since all x86 CPUs are successors of the 16-bit 8086 CPU, these older 16-bit op-codes are shorter than the newer 32-bit ones. That's why the or ah, 40h instruction occupies only 3 bytes. It would be more logical way to emit here or eax, 04000h but that is 5 bytes, or even 6 (in case the register in the first operand is not EAX).

### Optimizing GCC and regparm

It would be even shorter if to turn on the -O3 optimization flag and also set regparm=3.

#### Listing 19.14: Optimizing GCC

```assembly
public  f
proc  near
push   ebp
or     ah, 40h
mov    ebp, esp
and    ah, 0FDh
pop    ebp
retn
endp

f
```

Indeed, the first argument is already loaded in EAX, so it is possible to work with it in-place. It is worth noting that both the function prologue (push ebp / mov ebp, esp) and epilogue (pop ebp) can easily be omitted here, but GCC probably is not good enough to do such code size optimizations. However, such short functions are better to be inlined functions (43 on page 695).

### 19.2.2 ARM + Optimizing Keil 6/2013 (ARM mode)

#### Listing 19.15: Optimizing Keil 6/2013 (ARM mode)

<table>
<thead>
<tr>
<th>Code</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>02 0C C0 E3</td>
<td>BIC</td>
<td>R0, R0, #0x200</td>
<td></td>
</tr>
<tr>
<td>01 09 80 E3</td>
<td>ORR</td>
<td>R0, R0, #0x4000</td>
<td></td>
</tr>
<tr>
<td>1E FF 2F E1</td>
<td>BX</td>
<td>LR</td>
<td></td>
</tr>
</tbody>
</table>

BIC (Bitwise bit Clear) is an instruction for clearing specific bits. This is just like the AND instruction, but with inverted operand. I.e., it’s analogous to a NOT +AND instruction pair.
ORR is “logical or”, analogous to OR in x86.
So far it’s easy.

### 19.2.3 ARM + Optimizing Keil 6/2013 (Thumb mode)

Listing 19.16: Optimizing Keil 6/2013 (Thumb mode)

```
01 21 89 03     MOVS  R1, 0x4000
08 43 ORRS R0, R1
49 11 ASRS R1, R1, #5 ; generate 0x200 and place to R1
88 43 BICS R0, R1
70 47 BX LR
```

Seems like Keil decided that the code in Thumb mode, making 0x200 from 0x4000, is more compact than the code for writing 0x200 to an arbitrary register.
So that is why, with the help of ASRS (arithmetic shift right), this value is calculated as $0x4000 \gg 5$.

### 19.2.4 ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 19.17: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
42 0C C0 E3 BIC R0, R0, #0x4200
01 09 80 E3 ORR R0, R0, #0x4000
1E FF 2F E1 BX LR
```

The code that was generated by LLVM, in source code form could be something like this:

```c
REMOVE_BIT (rt, 0x4200);
SET_BIT (rt, 0x4000);
```

And it does exactly what we need. But why 0x4200? Perhaps, that an artifact from LLVM’s optimizer\(^5\). Probably a compiler’s optimizer error, but the generated code works correctly anyway.
You can read more about compiler anomalies here (91 on page 1262).

Optimizing Xcode 4.6.3 (LLVM) for Thumb mode generates the same code.

---

\(^5\)It was LLVM build 2410.2.00 bundled with Apple Xcode 4.6.3
19.2.5 ARM: more about the BIC instruction

Let’s rework the example slightly:

```c
int f(int a)
{
    int rt=a;
    REMOVE_BIT (rt, 0x1234);
    return rt;
}
```

Then the optimizing Keil 5.03 in ARM mode does:

```assembly
f PROC
    BIC r0,r0,#0x1000
    BIC r0,r0,#0x234
    BX lr
ENDP
```

There are two BIC instructions, i.e., bits 0x1234 are cleared in two passes. This is because it’s not possible to encode 0x1234 in a BIC instruction, but it’s possible to encode 0x1000 and 0x234.

19.2.6 ARM64: Optimizing GCC (Linaro) 4.9

Optimizing GCC compiling for ARM64 can use the AND instruction instead of BIC:

```assembly
Listing 19.18: Optimizing GCC (Linaro) 4.9
f:
    and w0, w0, -513 ; 0xFFFFFFFFFFFFFDFF
    orr w0, w0, 16384 ; 0x4000
    ret
```

19.2.7 ARM64: Non-optimizing GCC (Linaro) 4.9

Non-optimizing GCC generates more redundant code, but works just like optimized:

```assembly
Listing 19.19: Non-optimizing GCC (Linaro) 4.9
f:
    sub sp, sp, #32
    str w0, [sp,12]
```
19.2.8 MIPS

Listing 19.20: Optimizing GCC 4.4.5 (IDA)

f:
; $a0=a
    ori   $a0, 0x4000
; $a0=a|0x4000
    li    $v0, 0xFFFFFFFFDFF
    jr    $ra
    and   $v0, $a0, $v0
; at finish: $v0 = $a0&$v0 = a|0x4000 & 0xFFFFFFFFDFF

ORI is, of course, the OR operation. “I” in the instruction name mean that the value is embedded in the machine code.

But after that we have AND. There was no way to use ANDI because it’s not possible to embed the 0xFFFFFFFFDFF number in a single instruction, so the compiler has to load 0xFFFFFFFFDFF into register $V0 first and then generates AND which takes all its values from registers.

19.3 Shifts

Bit shifts in C/C++ are implemented using << and >> operators.

The x86 ISA has the SHL (SHift Left) and SHR (SHift Right) instructions for this.

Shift instructions are often used in division and multiplications by powers of two: $2^n$ (e.g., 1, 2, 4, 8, etc): 16.1.2 on page 290, 16.2.1 on page 296.

Shifting operations are also so important because they are often used for specific bit isolation or for constructing a value of several scattered bits.
19.4 Setting and clearing specific bits: FPU example

Here is how bits are located in the float type in IEEE 754 form:

```
0 2 2 2 3 3 0 3 1
S exponent mantissa or fraction
```

(S—sign)

The sign of number is in the MSB\(^6\). Will it be possible to change the sign of a floating point number without any FPU instructions?

```c
#include <stdio.h>

float my_abs (float i)
{
    unsigned int tmp=(*(unsigned int*)&i) & 0x7FFFFFFF;
    return *(float*)&tmp;
}

float set_sign (float i)
{
    unsigned int tmp=(*(unsigned int*)&i) | 0x80000000;
    return *(float*)&tmp;
}

float negate (float i)
{
    unsigned int tmp=(*(unsigned int*)&i) ^ 0x80000000;
    return *(float*)&tmp;
}

int main()
{
    printf ("my_abs():\n");
    printf ("%f\n", my_abs (123.456));
    printf ("%f\n", my_abs (-456.123));
    printf ("set_sign():\n");
    printf ("%f\n", set_sign (123.456));
    printf ("%f\n", set_sign (-456.123));
    printf ("negate():\n");
    printf ("%f\n", negate (123.456));
    printf ("%f\n", negate (-456.123));
}
```

\(^6\)Most significant bit/byte
We need this trickery in C/C++ to copy to/from float value without actual conversion. So there are three functions: my_abs() resets MSB; set_sign() sets MSB and negate() flips it.

### 19.4.1 A word about the XOR operation

XOR is widely used when one needs just to flip specific bit(s).

Indeed, the XOR operation applied with 1 effectively inverts a bit:

<table>
<thead>
<tr>
<th>input A</th>
<th>input B</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

And vice-versa, the XOR operation applied with 0 does nothing, i.e., it’s an idle operation.

This is a very important property of the XOR operation and it’s highly recommended to memorize it.

### 19.4.2 x86

The code is pretty straightforward:

```assembly
Listing 19.21: Optimizing MSVC 2012

_ttmp$ = 8
_i$ = 8
_my_abs PROC
    and DWORD PTR _i$[esp-4], 2147483647 ; 7
    ffffH
    fld DWORD PTR _tmp$[esp-4]
    ret 0
_my_abs ENDP

_ttmp$ = 8
_i$ = 8
_set_sign PROC
    or DWORD PTR _i$[esp-4], -2147483648 ; 8
    80000000H
    fld DWORD PTR _tmp$[esp-4]
    ret 0
_set_sign ENDP
```
An input value of type float is taken from the stack, but treated as an integer value. AND and OR reset and set the desired bit. XOR flips it.

Finally, the modified value is loaded into ST0, because floating-point numbers are returned in this register.

Now let's try optimizing MSVC 2012 for x64:

Listing 19.22: Optimizing MSVC 2012 x64

```assembly
_my_abs PROC
    movss DWORD PTR [rsp+8], xmm0
    mov eax, DWORD PTR i$[rsp]
    btr eax, 31
    mov DWORD PTR tmp$[rsp], eax
    movss xmm0, DWORD PTR tmp$[rsp]
    ret 0
_my_abs ENDP

_set_sign PROC
    movss DWORD PTR [rsp+8], xmm0
    mov eax, DWORD PTR i$[rsp]
    bts eax, 31
    mov DWORD PTR tmp$[rsp], eax
    movss xmm0, DWORD PTR tmp$[rsp]
    ret 0
_set_sign ENDP

_negate PROC
    movss DWORD PTR [rsp+8], xmm0
    mov eax, DWORD PTR i$[rsp]
    btc eax, 31
    ret 0
_negate ENDP
```
The input value is passed in XMM0, then it is copied into the local stack and then we see some instructions that are new to us: BTR, BTS, BTC.

These instructions are used for resetting (BTR), setting (BTS) and inverting (or complementing: BTC) specific bits. The 31st bit is MSB, counting from 0.

Finally, the result is copied into XMM0, because floating point values are returned through XMM0 in Win64 environment.

### 19.4.3 MIPS

GCC 4.4.5 for MIPS does mostly the same:

```
my_abs:
; move from coprocessor 1:
mfc1 $v1, $f12
li $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; do AND:
and $v0, $v1
; move to coprocessor 1:
mtc1 $v0, $f0
; return
jr $ra
or $at, $zero ; branch delay slot

set_sign:
; move from coprocessor 1:
mfc1 $v0, $f12
lui $v1, 0x8000
; $v1=0x80000000
; do OR:
or $v0, $v1, $v0
; move to coprocessor 1:
mtc1 $v0, $f0
; return
jr $ra
or $at, $zero ; branch delay slot
```

```
; move from coprocessor 1:
mfc1 $v0, $f12
lui $v1, 0x8000
; $v1=0x80000000
; do XOR:
xor $v0, $v1, $v0
; move to coprocessor 1:
mtc1 $v0, $f0
; return
jr $ra
or $at, $zero ; branch delay slot

One single LUI instruction is used to load 0x80000000 into a register, because LUI is clearing the low 16 bits and these are zeroes in the constant, so one LUI without subsequent ORI is enough.

### 19.4.4 ARM

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

Listing 19.24: Optimizing Keil 6/2013 (ARM mode)

```
my_abs PROC
; clear bit:
    BIC    r0, r0, #0x80000000
    BX     lr
ENDP

set_sign PROC
; do OR:
    ORR    r0, r0, #0x80000000
    BX     lr
ENDP

negate PROC
; do XOR:
    EOR    r0, r0, #0x80000000
    BX     lr
ENDP
```

So far so good. ARM has the BIC instruction, which explicitly clears specific bit(s). EOR is the ARM instruction name for XOR (“Exclusive OR”).

#### Optimizing Keil 6/2013 (Thumb mode)
CHAPTER 19. MANIPULATING SPECIFIC BIT(S)

Listing 19.25: Optimizing Keil 6/2013 (Thumb mode)

```assembly
my_abs PROC
    LSLS     r0,r0,#1
    ; r0=i<<1
    LSRS     r0,r0,#1
    ; r0=(i<<1)>>1
    BX       lr
    ENDP

set_sign PROC
    MOVS     r1,#1
    ; r1=1
    LSLS     r1,r1,#31
    ; r1=1<<31=0x80000000
    ORRS     r0,r0,r1
    ; r0=r0 | 0x80000000
    BX       lr
    ENDP

negate PROC
    MOVS     r1,#1
    ; r1=1
    LSLS     r1,r1,#31
    ; r1=1<<31=0x80000000
    EORS     r0,r0,r1
    ; r0=r0 ^ 0x80000000
    BX       lr
    ENDP
```

Thumb mode in ARM offers 16-bit instructions and not much data can be encoded in them, so here a MOVS/LSLS instruction pair is used for forming the 0x80000000 constant. It works like this: \(1 \ll 31 = 0\times80000000\).

The code of my_abs is weird and it effectively works like this expression: \((i \ll 1) \gg 1\). This statement looks meaningless. But nevertheless, when input \(\ll 1\) is executed, the MSB (sign bit) is just dropped. When the subsequent result \(\gg 1\) statement is executed, all bits are now in their own places, but MSB is zero, because all “new” bits appearing from the shift operations are always zeroes. That is how the LSLS/LSRS instruction pair clears MSB.

Optimizing GCC 4.6.3 (Raspberry Pi, ARM mode)

Listing 19.26: Optimizing GCC 4.6.3 for Raspberry Pi (ARM mode)

```assembly
my_abs
    ; copy from S0 to R2:
```
Let's run Raspberry Pi Linux in QEMU and it emulates an ARM FPU, so S-registers are used here for floating point numbers instead of R-registers.

The 

<table>
<thead>
<tr>
<th>FMRS R2, S0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIC R3, R2, #0x80000000</td>
</tr>
<tr>
<td>FMSR S0, R3</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
</tbody>
</table>

set_sign
<table>
<thead>
<tr>
<th>copy from S0 to R2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMRS R2, S0</td>
</tr>
<tr>
<td>ORR R3, R2, #0x80000000</td>
</tr>
<tr>
<td>FMSR S0, R3</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
</tbody>
</table>

negate
<table>
<thead>
<tr>
<th>copy from S0 to R2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMRS R2, S0</td>
</tr>
<tr>
<td>ADD R3, R2, #0x80000000</td>
</tr>
<tr>
<td>FMSR S0, R3</td>
</tr>
<tr>
<td>BX LR</td>
</tr>
</tbody>
</table>

The FMRS instruction copies data from GPR to the FPU and back. 

my_abs() and set_sign() looks as expected, but negate()? Why is there ADD instead of XOR?

It's hard to believe, but the instruction ADD register, 0x80000000 works just like XOR register, 0x80000000. First of all, what's our goal? The goal is to flip the MSB, so let's forget about the XOR operation. From school-level mathematics we may remember that adding values like 1000 to other values never affects the last 3 digits. For example: 1234567 + 10000 = 1244567 (last 4 digits are never affected). But here we operate in binary base and 0x80000000 is 10000000000000000000000000000000, i.e., only the highest bit is set. Adding 0x80000000 to any value never affects the lowest 31 bits, but affects only the MSB. Adding 1 to 0 is resulting in 1. Adding 1 to 1 is resulting in 10 in binary form, but the 32th bit (counting from zero) gets dropped, because our registers are 32 bit wide, so the result is 0. That's why XOR can be replaced by ADD here. It's hard to say why GCC decided to do this, but it works correctly.
CHAPTER 19. MANIPULATING SPECIFIC BIT(S)

19.5 Counting bits set to 1

Here is a simple example of a function that calculates the number of bits set in the input value.

This operation is also called “population count”\(^7\).

```c
#include <stdio.h>

#define IS_SET(flag, bit) ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
}

int main()
{
    f(0x12345678); // test
}
```

In this loop, the iteration count value \(i\) is counting from 0 to 31, so the \(1 \ll i\) statement is counting from 1 to 0x80000000. Describing this operation in natural language, we would say \textit{shift 1 by n bits left}. In other words, \(1 \ll i\) statement consequently produces all possible bit positions in a 32-bit number. The freed bit at right is always cleared.

Here is a table of all possible \(1 \ll i\) for \(i = 0 \ldots 31:\)

\[\begin{array}{r|r}
\hline
i & \text{result} \\
\hline
0 & 1 \\
1 & 2 \\
2 & 4 \\
3 & 8 \\
4 & 16 \\
5 & 32 \\
6 & 64 \\
7 & 128 \\
8 & 256 \\
9 & 512 \\
10 & 1024 \\
11 & 2048 \\
12 & 4096 \\
13 & 8192 \\
14 & 16384 \\
15 & 32768 \\
16 & 65536 \\
17 & 131072 \\
18 & 262144 \\
19 & 524288 \\
20 & 1048576 \\
21 & 2097152 \\
22 & 4194304 \\
23 & 8388608 \\
24 & 16777216 \\
25 & 33554432 \\
26 & 67108864 \\
27 & 134217728 \\
28 & 268435456 \\
29 & 536870912 \\
30 & 1073741824 \\
31 & 2147483648 \\
\hline
\end{array}\]

\(^7\)modern x86 CPUs (supporting SSE4) even have a POPCNT instruction for it.
### C/C++ expression | Power of two | Decimal form | Hexadecimal form
--- | --- | --- | ---
$1 \ll 0$ | $1$ | $1$ | $1$
$1 \ll 1$ | $2^1$ | $2$ | $2$
$1 \ll 2$ | $2^2$ | $4$ | $4$
$1 \ll 3$ | $2^3$ | $8$ | $8$
$1 \ll 4$ | $2^4$ | $16$ | $0x10$
$1 \ll 5$ | $2^5$ | $32$ | $0x20$
$1 \ll 6$ | $2^6$ | $64$ | $0x40$
$1 \ll 7$ | $2^7$ | $128$ | $0x80$
$1 \ll 8$ | $2^8$ | $256$ | $0x100$
$1 \ll 9$ | $2^9$ | $512$ | $0x200$
$1 \ll 10$ | $2^{10}$ | $1024$ | $0x400$
$1 \ll 11$ | $2^{11}$ | $2048$ | $0x800$
$1 \ll 12$ | $2^{12}$ | $4096$ | $0x1000$
$1 \ll 13$ | $2^{13}$ | $8192$ | $0x2000$
$1 \ll 14$ | $2^{14}$ | $16384$ | $0x4000$
$1 \ll 15$ | $2^{15}$ | $32768$ | $0x8000$
$1 \ll 16$ | $2^{16}$ | $65536$ | $0x10000$
$1 \ll 17$ | $2^{17}$ | $131072$ | $0x20000$
$1 \ll 18$ | $2^{18}$ | $262144$ | $0x40000$
$1 \ll 19$ | $2^{19}$ | $524288$ | $0x80000$
$1 \ll 20$ | $2^{20}$ | $1048576$ | $0x100000$
$1 \ll 21$ | $2^{21}$ | $2097152$ | $0x200000$
$1 \ll 22$ | $2^{22}$ | $4194304$ | $0x400000$
$1 \ll 23$ | $2^{23}$ | $8388608$ | $0x800000$
$1 \ll 24$ | $2^{24}$ | $16777216$ | $0x1000000$
$1 \ll 25$ | $2^{25}$ | $33554432$ | $0x2000000$
$1 \ll 26$ | $2^{26}$ | $67108864$ | $0x4000000$
$1 \ll 27$ | $2^{27}$ | $134217728$ | $0x8000000$
$1 \ll 28$ | $2^{28}$ | $268435456$ | $0x10000000$
$1 \ll 29$ | $2^{29}$ | $536870912$ | $0x20000000$
$1 \ll 30$ | $2^{30}$ | $1073741824$ | $0x40000000$
$1 \ll 31$ | $2^{31}$ | $2147483648$ | $0x80000000$

These constant numbers (bit masks) very often appear in code and a practicing reverse engineer must be able to spot them quickly. You probably haven’t to memorize the decimal numbers, but the hexadecimal ones are very easy to remember.

These constants are very often used for mapping flags to specific bits. For example, here is excerpt from `ssl_private.h` from Apache 2.4.6 source code:

```c
/**
 * Define the SSL options
```
Let's get back to our example.

The IS_SET macro checks bit presence in a. The IS_SET macro is in fact the logical AND operation (AND) and it returns 0 if the specific bit is absent there, or the bit mask, if the bit is present. The if() operator in C/C++ triggers if the expression in it is not zero, it might be even 123456, that is why it always works correctly.

19.5.1 x86

MSVC

Let's compile (MSVC 2010):

```Assembly
Listing 19.27: MSVC 2010

_r$ = -8 ; size = 4
_i$ = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
push ebp
mov ebp, esp
sub esp, 8
mov DWORD PTR _rt$[ebp], 0
mov DWORD PTR _i$[ebp], 0
jmp SHORT $LN4@f
$LN3@f:
mov eax, DWORD PTR _i$[ebp] ; increment of i
add eax, 1
mov DWORD PTR _i$[ebp], eax
$LN4@f:
cmp DWORD PTR _i$[ebp], 32 ; 00000020H
jge SHORT $LN2@f ; loop finished?
mov edx, 1
mov ecx, DWORD PTR _i$[ebp]
shl edx, cl ; EDX=EDX<<CL
and edx, DWORD PTR _a$[ebp]
```

je SHORT $LN1@f ; result of AND instruction
was 0?

; then skip next instructions
mov eax, DWORD PTR _rt$[ebp] ; no, not zero
add eax, 1 ; increment rt
mov DWORD PTR _rt$[ebp], eax

$LN1@f:
jmp SHORT $LN3@f

$LN2@f:
mov eax, DWORD PTR _rt$[ebp]
mov esp, ebp
pop ebp
ret 0

_f ENDP
Let’s load this example into OllyDbg. Let the input value be 0x12345678.

For $i = 1$, we see how $i$ is loaded into ECX:

![OllyDbg screenshot](image)

Figure 19.5: OllyDbg: $i = 1$, $i$ is loaded into ECX

EDX is 1. SHL is to be executed now.
SHL was executed:

\[
\text{EDX} = 1 \ll 1 = 2
\]

Figure 19.6: OllyDbg: \( i = 1, \text{EDX} = 1 \ll 1 = 2 \)

EDX contain \( 1 \ll 1 \) (or 2). This is a bit mask.
AND sets ZF to 1, which implies that the input value (0x12345678) ANDed with 2 results in 0:

![CPU - main thread, module shifts](image)

Figure 19.7: OllyDbg: $i = 1$, is there that bit in the input value? No. (ZF =1)

So, there is no corresponding bit in the input value. The piece of code, which increments the counter is not to be executed: the JZ instruction bypassing it.
Let’s trace a bit further and $i$ is now 4. SHL is to be executed now:

![OllyDbg: $i = 4, i$ is loaded into ECX](image)

**Figure 19.8:** OllyDbg: $i = 4, i$ is loaded into ECX
CHAPTER 19. MANIPULATING SPECIFIC BIT(S)

EDX = 1 << 4 (or 0x10 or 16):

![Figure 19.9: OllyDbg: i = 4, EDX = 1 << 4 = 0x10](image)

This is another bit mask.
AND is executed:

Figure 19.10: OllyDbg: $i = 4$, is there that bit in the input value? Yes. ($ZF = 0$)

$ZF$ is 0 because this bit is present in the input value. Indeed, $0x12345678 \& 0x10 = 0x10$. This bit counts: the jump is not triggering and the bit counter incrementing.

The function returns 13. This is total number of bits set in $0x12345678$.

GCC

Let’s compile it in GCC 4.4.1:

```
Listing 19.28: GCC 4.4.1

public f
f proc near

rt = dword ptr -0Ch
i = dword ptr -8
arg_0 = dword ptr 8

push ebp
mov ebp, esp
push ebx
sub esp, 10h
mov [ebp+rt], 0
mov [ebp+i], 0
jmp short loc_80483EF

loc_80483D0:
    mov eax, [ebp+i]
    mov edx, 1
```

```
```
### 19.5.2 x64

Let's modify the example slightly to extend it to 64-bit:

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

#define IS_SET(flag, bit) (((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
}
```

**Non-optimizing GCC 4.8.2**

So far so easy.
Listing 19.29: Non-optimizing GCC 4.8.2

```assembly
f:
    push rbp
    mov rbp, rsp
    mov QWORD PTR [rbp-24], rdi ; a
    mov DWORD PTR [rbp-12], 0 ; rt=0
    mov QWORD PTR [rbp-8], 0 ; i=0
    jmp .L2
.L4:
    mov rax, QWORD PTR [rbp-8]
    mov rdx, QWORD PTR [rbp-24]
    ; RAX = i, RDX = a
    mov ecx, eax
    ; ECX = i
    shr rdx, cl
    ; RDX = RDX>>CL = a>>i
    mov rax, rdx
    ; RAX = RDX = a>>i
    and eax, 1
    ; EAX = EAX&1 = (a>>i)&1
    test rax, rax
    ; the last bit is zero?
    ; skip the next ADD instruction, if it was so.
    je .L3
    add DWORD PTR [rbp-12], 1 ; rt++
.L3:
    add QWORD PTR [rbp-8], 1 ; i++
.L2:
    cmp QWORD PTR [rbp-8], 63 ; i<63?
    jbe .L4 ; jump to the loop body
    mov eax, DWORD PTR [rbp-12] ; return rt
    pop rbp
    ret
```

Optimizing GCC 4.8.2

Listing 19.30: Optimizing GCC 4.8.2

```assembly
1 f:
2    xor eax, eax ; rt variable will be in EAX register
3    xor ecx, ecx ; i variable will be in ECX register
4 .L3:
5    mov rsi, rdi ; load input value
```
This code is terser, but has a quirk. In all examples that we see so far, we were incrementing the "rt" value after comparing a specific bit, but the code here increments "rt" before (line 6), writing the new value into register EDX. Thus, if the last bit is 1, the CMOVNE\(^8\) instruction (which is a synonym for CMOVNZ\(^9\)) commits the new value of “rt” by moving EDX (“proposed rt value”) into EAX (“current rt” to be returned at the end). Hence, the incrementing is done at each step of loop, i.e., 64 times, without any relation to the input value.

The advantage of this code is that it contain only one conditional jump (at the end of the loop) instead of two jumps (skipping the “rt” value increment and at the end of loop). And that might work faster on the modern CPUs with branch predictors: \(33.1\) on page 624.

The last instruction is REP RET (opcode F3 C3) which is also called FATRET by MSVC. This is somewhat optimized version of RET, which is recommended by AMD to be placed at the end of function, if RET goes right after conditional jump: [AMD13b, p.15]\(^10\).

### Optimizing MSVC 2010

**Listing 19.31: MSVC 2010**

```assembly
a$ = 8
f PROC
    ; RCX = input value
    xor eax, eax
    mov edx, 1
    lea r8d, QWORD PTR [rax+64]
    ; R8D=64
    npad 5
$LL4@f:
```

---

\(^8\) Conditional MOVe if Not Equal

\(^9\) Conditional MOVe if Not Zero

\(^10\) More information on it: [http://go.yurichev.com/17328](http://go.yurichev.com/17328)
test rdx, rcx  ; there are no such bit in input value?
; skip the next INC instruction then.
    je SHORT $LN3@f
    inc eax    ; rt++
$LN3@f:
    rol rdx, 1    ; RDX=RDX<<1
    dec r8    ; R8--
    jne SHORT $LL4@f
fatret 0
f ENDP

Here the ROL instruction is used instead of SHL, which is in fact “rotate left” instead of “shift left”, but in this example it works just as SHL.

You can read more about the rotate instruction here: A.6.3 on page 1315.

R8 here is counting from 64 to 0. It’s just like an inverted i.

Here is a table of some registers during the execution:

<table>
<thead>
<tr>
<th>RDX</th>
<th>R8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000001</td>
<td>64</td>
</tr>
<tr>
<td>0x0000000000000002</td>
<td>63</td>
</tr>
<tr>
<td>0x0000000000000004</td>
<td>62</td>
</tr>
<tr>
<td>0x0000000000000008</td>
<td>61</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x4000000000000000</td>
<td>2</td>
</tr>
<tr>
<td>0x8000000000000000</td>
<td>1</td>
</tr>
</tbody>
</table>

At the end we see the FATRET instruction, which was explained here: 19.5.2 on the preceding page.

Optimizing MSVC 2012

Listing 19.32: MSVC 2012

a$ = 8
f PROC
    ; RCX = input value
    xor eax, eax
    mov edx, 1
    lea r8d, QWORD PTR [rax+32]
    ; EDX = 1, R8D = 32
    npad 5
$LL4@f:
    ; pass 1 ------------------------------------
Optimizing MSVC 2012 does almost the same job as optimizing MSVC 2010, but somehow, it generates two identical loop bodies and the loop count is now 32 instead of 64. To be honest, it’s not possible to say why. Some optimization trick? Maybe it’s better for the loop body to be slightly longer? Anyway, such code is relevant here to show that sometimes the compiler output may be really weird and illogical, but perfectly working.

19.5.3 ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 19.33: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R1, R0</td>
<td>R1 = R0</td>
</tr>
<tr>
<td>MOV R0, #0</td>
<td>R0 = 0</td>
</tr>
<tr>
<td>MOV R2, #1</td>
<td>R2 = 1</td>
</tr>
<tr>
<td>MOV R3, R0</td>
<td>R3 = R0</td>
</tr>
<tr>
<td>TST R1, R2,LSL R3</td>
<td>set flags according to R1 &amp; (R2&lt;&lt;R3)</td>
</tr>
<tr>
<td>ADD R3, R3, #1</td>
<td>R3++</td>
</tr>
<tr>
<td>ADDNE R0, R0, #1</td>
<td>if ZF flag is cleared by TST, then R0++</td>
</tr>
<tr>
<td>CMP R3, #32</td>
<td>R3 = 32</td>
</tr>
<tr>
<td>BNE loc_2E54</td>
<td></td>
</tr>
<tr>
<td>BX LR</td>
<td></td>
</tr>
</tbody>
</table>

TST is the same things as TEST in x86.
As was noted before (41.2.1 on page 679), there are no separate shifting instructions in ARM mode. However, there are modifiers LSL (Logical Shift Left), LSR (Logical Shift Right), ASR (Arithmetic Shift Right), ROR (Rotate Right) and RRX (Rotate Right with Extend), which may be added to such instructions as MOV, TST, CMP, ADD, SUB, RSB\textsuperscript{11}.

These modifiers define how to shift the second operand and by how many bits. Thus the “TST R1, R2, LSL R3” instruction works here as $R1 \land (R2 \ll R3)$.

### 19.5.4 ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

Almost the same, but here are two LSL.W/TST instructions are used instead of a single TST, because in Thumb mode it is not possible to define LSL modifier directly in TST.

<table>
<thead>
<tr>
<th>loc_2F7A</th>
<th>MOV</th>
<th>R1, R0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MOVS</td>
<td>R0, #0</td>
</tr>
<tr>
<td></td>
<td>MOV.W</td>
<td>R9, #1</td>
</tr>
<tr>
<td></td>
<td>MOVS</td>
<td>R3, #0</td>
</tr>
<tr>
<td>LSL.W</td>
<td>R2, R9, R3</td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td>R2, R1</td>
<td></td>
</tr>
<tr>
<td>ADD.W</td>
<td>R3, R3, #1</td>
<td></td>
</tr>
<tr>
<td>IT NE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDNE</td>
<td>R0, #1</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>R3, #32</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>loc_2F7A</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>LR</td>
<td></td>
</tr>
</tbody>
</table>

### 19.5.5 ARM64 + Optimizing GCC 4.9

Let’s take the 64-bit example which has been already used: 19.5.2 on page 445.

**Listing 19.34: Optimizing GCC (Linaro) 4.8**

```assembly
f:
mov  w2, 0 ; rt=0
mov  x5, 1
mov  w1, w2
.L2:
  lsl  x4, x5, x1 ; w4 = w5<<w1 = 1<<i
  add  w3, w2, 1 ; new_rt=rt+1
  tst  x4, x0 ; (1<<i) & a
```

\textsuperscript{11}These instructions are also called “data processing instructions”
add    w1, w1, 1 ; i++
; result of TST was non-zero?
; then w2=w3 or rt=new_rt.
; otherwise: w2=w2 or rt=rt (idle operation)
csel   w2, w3, w2, ne
cmp    w1, 64 ; i<64?
bne  .L2 ; yes
mov    w0, w2 ; return rt
ret

The result is very similar to what GCC generates for x64: 19.30 on page 446.
The CSEL instruction is “Conditional SELect”. It just chooses one variable of two depending on the flags set by TST and copies the value into W2, which holds the “rt” variable.

19.5.6  ARM64 + Non-optimizing GCC 4.9

And again, we’ll work on the 64-bit example which was already used: 19.5.2 on page 445.
The code is more verbose, as usual.

Listing 19.35: Non-optimizing GCC (Linaro) 4.8

f:
  sub    sp, sp, #32
  str    x0, [sp,8] ; store "a" value to Register Save Area
  str    wzr, [sp,24] ; rt=0
  str    wzr, [sp,28] ; i=0
  b      .L2
.L4:
  ldr    w0, [sp,28]
  mov    x1, 1
  lsl    x0, x1, x0 ; X0 = X1<<X0 = 1<<i
  mov    x1, x0 ; X1 = 1<<1
  ldr    x0, [sp,8] ; X0 = a
  and    x0, x1, x0 ; X0 = X1&X0 = (1<<i) & a
  cmp    x0, xzr ; X0 contain zero? then jump to .L3, skipping "rt" increment
            beq    .L3
  ; rt++
  ldr    w0, [sp,24]
CHAPTER 19. MANIPULATING SPECIFIC BIT(S)

19.5.7 MIPS

Non-optimizing GCC

Listing 19.36: Non-optimizing GCC 4.4.5 (IDA)

f:
; IDA is not aware of variable names, we gave them manually:
rt = -0x10
i = -0xC
var_4 = -4
a = 0

addiu $sp, -0x18
sw $fp, 0x18+var_4($sp)
move $fp, $sp
sw $a0, 0x18+a($fp)
; initialize rt and i variables to zero:
sw $zero, 0x18+rt($fp)
sw $zero, 0x18+i($fp)
; jump to loop check instructions:
b loc_68
or $at, $zero ; branch delay slot, NOP

loc_20:
li $v1, 1
lw $v0, 0x18+i($fp)
or $at, $zero ; load delay slot, NOP
That is verbose: all local variables are located in the local stack and reloaded each time they’re needed. The SLLV instruction is “Shift Word Left Logical Variable”, it differs from SLL only in that the shift amount is encoded in the SLL instruction (and is fixed, as a consequence), but SLLV takes shift amount from a register.
That is terser. There are two shift instructions instead of one. Why? It’s possible to replace the first SLLV instruction with an unconditional branch instruction that jumps right to the second SLLV. But this is another branching instruction in the function, and it’s always favorable to get rid of them: 33.1 on page 624.

Listing 19.37: Optimizing GCC 4.4.5 (IDA)

```assembly
f:
; $a0=a
; rt variable will reside in $v0:
    move $v0, $zero
; i variable will reside in $v1:
    move $v1, $zero
    li $t0, 1
    li $a3, 32
    sllv $a1, $t0, $v1
; $a1 = $t0<<$v1 = 1<<i
loc_14:
    and $a1, $a0
; $a1 = a&(1<<i)
; increment i:
    addiu $v1, 1
; jump to loc\_28 if a&(1<<i)==0 and increment rt:
    beqz $a1, loc_28
    addiu $a2, $v0, 1
; if BEQZ was not triggered, save updated rt into $v0:
    move $v0, $a2
loc_28:
; if i!=32, jump to loc_14 and also prepare next shifted value:
    bne $v1, $a3, loc_14
    sllv $a1, $t0, $v1
; return
    jr $ra
    or $at, $zero ; branch delay slot, NOP
```

19.6 Conclusion

Analogous to the C/C++ shifting operators ≪ and ≫, the shift instructions in x86 are SHR/SHL (for unsigned values) and SAR/SHL (for signed values).
The shift instructions in ARM are LSR/LSL (for unsigned values) and ASR/LSL (for signed values). It’s also possible to add shift suffix to some instructions (which are called “data processing instructions”).

### 19.6.1 Check for specific bit (known at compile stage)

Test if the 1000000 bit (0x40) is present in the register’s value:

**Listing 19.38: C/C++**

```c
if (input&0x40)
...
```

**Listing 19.39: x86**

```assembly
TEST REG, 40h
JNZ is_set
; bit is not set
```

**Listing 19.40: x86**

```assembly
TEST REG, 40h
JZ is_cleared
; bit is set
```

**Listing 19.41: ARM (ARM mode)**

```assembly
TST REG, #0x40
BNE is_set
; bit is not set
```

Sometimes, AND is used instead of TEST, but the flags that are set are the same.

### 19.6.2 Check for specific bit (specified at runtime)

This is usually done by this C/C++ code snippet (shift value by \( n \) bits right, then cut off lowest bit):

**Listing 19.42: C/C++**

```c
if ((value>>n)&1)
... 
```

This is usually implemented in x86 code as:
Or (shift 1 bit \( n \) times left, isolate this bit in input value and check if it's not zero):

Listing 19.44: C/C++

```c
if (value & (1<<n))
    ....
```

This is usually implemented in x86 code as:

Listing 19.45: x86

```assembly
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

### 19.6.3 Set specific bit (known at compile stage)

Listing 19.46: C/C++

```c
value=value|0x40;
```

Listing 19.47: x86

```assembly
OR REG, 40h
```

Listing 19.48: ARM (ARM mode) and ARM64

```assembly
ORR R0, R0, #0x40
```

### 19.6.4 Set specific bit (specified at runtime)

Listing 19.49: C/C++

```c
value=value|(1<<n);
```

This is usually implemented in x86 code as:
CHAPTER 19. MANIPULATING SPECIFIC BIT(S)

19.6.5 Clear specific bit (known at compile stage)

Just apply AND operation with the inverted value:

<table>
<thead>
<tr>
<th>Listing 19.51: C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>value=value&amp;(~0x40);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 19.52: x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND REG, 0FFFFFFFBh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 19.53: x64</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND REG, 0FFFFFFFFFFFFFBFh</td>
</tr>
</tbody>
</table>

This is actually leaving all bits set except one.

ARM in ARM mode has BIC instruction, which works like the NOT +AND instruction pair:

<table>
<thead>
<tr>
<th>Listing 19.54: ARM (ARM mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIC R0, R0, #0x40</td>
</tr>
</tbody>
</table>

19.6.6 Clear specific bit (specified at runtime)

<table>
<thead>
<tr>
<th>Listing 19.55: C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>value=value&amp;(~(1&lt;&lt;n));</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 19.56: x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>; CL=n</td>
</tr>
<tr>
<td>MOV REG, 1</td>
</tr>
<tr>
<td>SHL REG, CL</td>
</tr>
<tr>
<td>NOT REG</td>
</tr>
<tr>
<td>AND input_value, REG</td>
</tr>
</tbody>
</table>
19.7 Exercises

- http://challenges.re/67
- http://challenges.re/68
- http://challenges.re/69
- http://challenges.re/70
Chapter 20

Linear congruential generator as pseudorandom number generator

The linear congruential generator is probably the simplest possible way to generate random numbers. It’s not in favour in modern times\(^1\), but it’s so simple (just one multiplication, one addition and one AND operation), we can use it as an example.

```c
#include <stdint.h>

// constants from the Numerical Recipes book
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}
```

\(^1\)Mersenne twister is better
There are two functions: the first one is used to initialize the internal state, and the second one is called to generate pseudorandom numbers.

We see that two constants are used in the algorithm. They are taken from [Pre+07]. Let’s define them using a \#define C/C++ statement. It’s a macro. The difference between a C/C++ macro and a constant is that all macros are replaced with their value by C/C++ preprocessor, and they don’t take any memory, unlike variables. In contrast, a constant is a read-only variable. It’s possible to take a pointer (or address) of a constant variable, but impossible to do so with a macro.

The last AND operation is needed because by C-standard \texttt{my\_rand()} has to return a value in the 0..32767 range. If you want to get 32-bit pseudorandom values, just omit the last AND operation.

\section{x86}

Listing 20.1: Optimizing MSVC 2013

\begin{Verbatim}
_BSS SEGMENT
_rand_state DD 01H DUP (?)
_BSS ENDS

_init$ = 8
_srand PROC
  mov eax, DWORD PTR _init$[esp-4]
  mov DWORD PTR _rand_state, eax
  ret 0
_srand ENDP

_TEXT SEGMENT
_rand PROC
  imul eax, DWORD PTR _rand_state, 1664525
  add eax, 1013904223 ; 3c6ef35fH
  mov DWORD PTR _rand_state, eax
  and eax, 32767 ; 00007fffH
  ret 0
_rand ENDP

_TEXT ENDS
\end{Verbatim}

Here we see it: both constants are embedded into the code. There is no memory allocated for them. The \texttt{my\_srand()} function just copies its input value into the internal \texttt{rand\_state} variable.

\texttt{my\_rand()} takes it, calculates the next \texttt{rand\_state}, cuts it and leaves it in the EAX register.
The non-optimized version is more verbose:

**Listing 20.2: Non-optimizing MSVC 2013**

```
_BSS SEGMENT
_rand_state DD 01H DUP (?)
_BSS ENDS

_init$ = 8
_srand PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _init$[ebp]
    mov DWORD PTR _rand_state, eax
    pop ebp
    ret 0
_srand ENDP

_TEXT SEGMENT
_rand PROC
    push ebp
    mov ebp, esp
    imul eax, DWORD PTR _rand_state, 1664525
    mov DWORD PTR _rand_state, eax
    mov ecx, DWORD PTR _rand_state
    add ecx, 1013904223 ; 3c6ef35fH
    mov DWORD PTR _rand_state, ecx
    mov eax, DWORD PTR _rand_state
    and eax, 32767 ; 00007fffH
    pop ebp
    ret 0
_rand ENDP

_TEXT ENDS
```

**20.2 x64**

The x64 version is mostly the same and uses 32-bit registers instead of 64-bit ones (because we are working with `int` values here). But `my_srand()` takes its input argument from the ECX register rather than from stack:

**Listing 20.3: Optimizing MSVC 2013 x64**

```
_BSS SEGMENT
rand_state DD 01H DUP (?)
_BSS ENDS
```
/init$ = 8
my_srand PROC
; ECX = input argument
  mov DWORD PTR rand_state, ecx
  ret 0
my_srand ENDP

_TEXT SEGMENT
my_rand PROC
  imul eax, DWORD PTR rand_state, 1664525 ; 0019660dH
  add eax, 1013904223 ; 3
  mov DWORD PTR rand_state, eax
  and eax, 32767 ; 00007
  ret 0
my_rand ENDP

_TEXT ENDS

GCC compiler generates mostly the same code.

20.3 32-bit ARM

Listing 20.4: Optimizing Keil 6/2013 (ARM mode)

my_srand PROC
  LDR r1,[L0.52] ; load pointer to rand_state
  STR r0,[r1,#0] ; save rand_state
  BX lr
ENDP

my_rand PROC
  LDR r0,[L0.52] ; load pointer to rand_state
  LDR r2,[L0.56] ; load RNG_a
  LDR r1,[r0,#0] ; load rand_state
  MUL r1,r2,r1
  LDR r2,[L0.60] ; load RNG_c
  ADD r1,r1,r2
  STR r1,[r0,#0] ; save rand_state
; AND with 0x7FFF:
  LSL r0,r1,#17
  LSR r0,r0,#17

ENDP
It's not possible to embed 32-bit constants into ARM instructions, so Keil has to place them externally and load them additionally. One interesting thing is that it's not possible to embed the 0x7FFF constant as well. So what Keil does is shifting \( \text{rand\_state} \) left by 17 bits and then shifting it right by 17 bits. This is analogous to the \((\text{rand\_state} \ll 17) \gg 17\) statement in C/C++. It seems to be useless operation, but what it does is clearing the high 17 bits, leaving the low 15 bits intact, and that's our goal after all.

Optimizing Keil for Thumb mode generates mostly the same code.

### 20.4 MIPS

Listing 20.5: Optimizing GCC 4.4.5 (IDA)

```
my_srand:
; store $a0 to \text{rand\_state}:
    lui   $v0, (\text{rand\_state} >> 16)
    jr    $ra
    sw    $a0, \text{rand\_state}
my_rand:
; load \text{rand\_state} to $v0:
    lui   $v1, (\text{rand\_state} >> 16)
    lw    $v0, \text{rand\_state}
    or    $at, $zero ; load delay slot
; multiplicate \text{rand\_state} in $v0 by 1664525 (RNG_a):
    sll   $a1, $v0, 2
    sll   $a0, $v0, 4
    addu  $a0, $a1, $a0
    sll   $a1, $a0, 6
    subu  $a0, $a1, $a0
```
addu \$a0, \$v0
sll \$a1, \$a0, 5
addu \$a0, \$a1
sll \$a0, 3
addu \$v0, \$a0, \$v0
sll \$a0, \$v0, 2
addu \$v0, \$a0

; add 1013904223 (RNG_c)
; the LI instruction is coalesced by IDA from LUI and ORI
li \$a0, 0x3C6EF35F
addu \$v0, \$a0

; store to rand_state:
sw \$v0, (rand_state & 0xFFFF)(\$v1)
jr \$ra
andi \$v0, 0x7FFF ; branch delay slot

Wow, here we see only one constant (0x3C6EF35F or 1013904223). Where is the other one (1664525)?

It seems that multiplication by 1664525 is done by just using shifts and additions! Let’s check this assumption:

#define RNG_a 1664525

int f (int a)
{
    return a*RNG_a;
}

Listing 20.6: Optimizing GCC 4.4.5 (IDA)

f:
sll \$v1, \$a0, 2
sll \$v0, \$a0, 4
addu \$v0, \$v1, \$v0
sll \$v1, \$v0, 6
subu \$v0, \$v1, \$v0
addu \$v0, \$a0
sll \$v1, \$v0, 5
addu \$v0, \$v1
sll \$v0, 3
addu \$a0, \$v0, \$a0
sll \$v0, \$a0, 2
jr \$ra
addu \$v0, \$a0, \$v0 ; branch delay slot

Indeed!
20.4.1 MIPS relocations

We will also focus on how such operations as load from memory and store to memory actually work. The listings here are produced by IDA, which hides some details.

We’ll run objdump twice: to get a disassembled listing and also relocations list:

Listing 20.7: Optimizing GCC 4.4.5 (objdump)

```bash
# objdump -D rand_O3.o
...

00000000 <my_srand>:
  0:  3c020000 lui v0,0x0
  4:  03e00008 jr ra
  8:  ac440000 sw a0,0(v0)

0000000c <my_rand>:
   c:  3c030000 lui v1,0x0
   10:  8c620000 lw v0,0(v1)
   14:  00200825 move at,at
   18:  00022880 sll a1,v0,0x2
   1c:  00022100 sll a0,v0,0x4
   20:  00a42021 addu a0,a1,a0
   24:  00042980 sll a1,a0,0x6
   28:  00a42023 subu a0,a1,a0
   2c:  00822021 addu a0,a0,v0
   30:  00042940 sll a1,a0,0x5
   34:  00852021 addu a0,a0,a1
   38:  000420c0 sll a0,a0,0x3
   3c:  00821021 addu v0,a0,v0
   40:  00022080 sll a0,v0,0x2
   44:  00441021 addu v0,v0,a0
   48:  3c043c6e lui a0,0x3c6e
   4c:  3484f35f ori a0,a0,0xf35f
   50:  00441021 addu v0,v0,a0
   54:  ac620000 sw v0,0(v1)
   58:  03e00008 jr ra
   5c:  30427fff andi v0,v0,0x7fff

# objdump -r rand_O3.o
...

RELOCATION RECORDS FOR [.text]:
OFFSET  TYPE    VALUE
```
Let's consider the two relocations for the my_srand() function. The first one, for address 0 has a type of R_MIPS_HI16 and the second one for address 8 has a type of R_MIPS_LO16. That implies that address of the beginning of the .bss segment is to be written into the instructions at address of 0 (high part of address) and 8 (low part of address). The rand_state variable is at the very start of the .bss segment. So we see zeroes in the operands of instructions LUI and SW, because nothing is there yet—the compiler don't know what to write there. The linker will fix this, and the high part of the address will be written into the operand of LUI and the low part of the address—to the operand of SW. SW will sum up the low part of the address and what is in register $V0 (the high part is there).

It’s the same story with the my_rand() function: R_MIPS_HI16 relocation instructs the linker to write the high part of the .bss segment address into instruction LUI. So the high part of the rand_state variable address is residing in register $V1. The LW instruction at address 0x10 sums up the high and low parts and loads the value of the rand_state variable into $V1. The SW instruction at address 0x54 do the summing again and then stores the new value to the rand_state global variable.

IDA processes relocations while loading, thus hiding these details, but we ought to remember them.

### 20.5 Thread-safe version of the example

The thread-safe version of the example is to be demonstrated later: 65.1 on page 966.
Chapter 21

Structures

A C/C++ structure, with some assumptions, is just a set of variables, always stored in memory together, not necessary of the same type

21.1 MSVC: SYSTEMTIME example

Let’s take the SYSTEMTIME win32 structure that describes time.
This is how it’s defined:

Listing 21.1: WinBase.h

typedef struct __SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;

Let’s write a C function to get the current time:

#include <windows.h>
#include <stdio.h>

1AKA “heterogeneous container”
2MSDN: SYSTEMTIME structure
void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            t.wYear, t.wMonth, t.wDay,
            t.wHour, t.wMinute, t.wSecond);

    return;
};

We get (MSVC 2010):

Listing 21.2: MSVC 2010 /GS-

_t$ = -16 ; size = 16
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 16
    lea eax, DWORD PTR _t$[ebp]
    push eax
    call DWORD PTR __imp__GetSystemTime@4
    movzx ecx, WORD PTR _t$[ebp+12] ; wSecond
    push ecx
    movzx edx, WORD PTR _t$[ebp+10] ; wMinute
    push edx
    movzx eax, WORD PTR _t$[ebp+8] ; wHour
    push eax
    movzx ecx, WORD PTR _t$[ebp+6] ; wDay
    push ecx
    movzx edx, WORD PTR _t$[ebp+2] ; wMonth
    push edx
    movzx eax, WORD PTR _t$[ebp] ; wYear
    push eax
    push OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0
    call _printf
    add esp, 28
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
16 bytes are allocated for this structure in the local stack — that is exactly sizeof(WORD)*8 (there are 8 WORD variables in the structure).

Pay attention to the fact that the structure begins with the wYear field. It can be said that a pointer to the SYSTEMTIME structure is passed to the GetSystemTime() ³, but it is also can be said that a pointer to the wYear field is passed, and that is the same! GetSystemTime() writes the current year to the WORD pointer pointing to, then shifts 2 bytes ahead, writes current month, etc, etc.

³MSDN: SYSTEMTIME structure
21.1.1 OllyDbg

Let’s compile this example in MSVC 2010 with /GS- /MD keys and run it in OllyDbg. Let’s open windows for data and stack at the address which is passed as the first argument of the `GetSystemTime()` function, and let’s wait until it’s executed. We see this:

![Figure 21.1: OllyDbg: GetSystemTime() just executed](image)

The system time of the function execution on my computer is 9 december 2014, 22:29:52:

![Figure 21.2: OllyDbg: printf() output](image)

So we see these 16 bytes in the data window:

```
DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 D4 03
```

Each two bytes represent one field of the structure. Since the endianness is little endian, we see the low byte first and then the high one. Hence, these are the values currently stored in memory:
<table>
<thead>
<tr>
<th>Hexadecimal number</th>
<th>Decimal number</th>
<th>Field name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07DE</td>
<td>2014</td>
<td>wYear</td>
</tr>
<tr>
<td>0x000C</td>
<td>12</td>
<td>wMonth</td>
</tr>
<tr>
<td>0x0002</td>
<td>2</td>
<td>wDayOfWeek</td>
</tr>
<tr>
<td>0x0009</td>
<td>9</td>
<td>wDay</td>
</tr>
<tr>
<td>0x0016</td>
<td>22</td>
<td>wHour</td>
</tr>
<tr>
<td>0x001D</td>
<td>29</td>
<td>wMinute</td>
</tr>
<tr>
<td>0x0034</td>
<td>52</td>
<td>wSecond</td>
</tr>
<tr>
<td>0x03D4</td>
<td>980</td>
<td>wMilliseconds</td>
</tr>
</tbody>
</table>

The same values are seen in the stack window, but they are grouped as 32-bit values.

And then `printf()` just takes the values it needs and outputs them to the console. Some values aren’t output by `printf()` (wDayOfWeek and wMilliseconds), but they are in memory right now, available for use.

### 21.1.2 Replacing the structure with array

The fact that the structure fields are just variables located side-by-side, can be easily demonstrated by doing the following. Keeping in mind the SYSTEMTIME structure description, it’s possible to rewrite this simple example like this:

```c
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf("%04d-%02d-%02d %02d:%02d:%02d
",

    return;
}
```

The compiler grumbles a bit:

```
systemtime2.c(7): warning C4133: 'function': incompatible types - from 'WORD [8]' to 'LPSYSTEMTIME'
```
But nevertheless, it produces this code:

```
Listing 21.3: Non-optimizing MSVC 2010

SG78573 DB '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
_array$ = -16 ; size = 16
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 16
    lea eax, DWORD PTR _array$[ebp]
    push eax
    call DWORD PTR __imp__GetSystemTime@4
    movzx ecx, WORD PTR _array$[ebp+12] ; wSecond
    push ecx
    movzx edx, WORD PTR _array$[ebp+10] ; wMinute
    push edx
    movzx eax, WORD PTR _array$[ebp+8] ; wHour
    push eax
    movzx ecx, WORD PTR _array$[ebp+6] ; wDay
    push ecx
    movzx edx, WORD PTR _array$[ebp+2] ; wMonth
    push edx
    movzx eax, WORD PTR _array$[ebp] ; wYear
    push eax
    push OFFSET $SG78573
    call __printf
    add esp, 28
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
```

And it works just as the same!

It is very interesting that the result in assembly form cannot be distinguished from the result of the previous compilation. So by looking at this code, one cannot say for sure if there was a structure declared, or an array.

Nevertheless, no sane person would do it, as it is not convenient. Also the structure fields may be changed by developers, swapped, etc.

We will not study this example in OllyDbg, because it will be just the same as in the case with the structure.
21.2 Let’s allocate space for a structure using malloc()

Sometimes it is simpler to place structures not the in local stack, but in the heap:

```c
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;
    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));
    GetSystemTime (t);
    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            t->wYear, t->wMonth, t->wDay,
            t->wHour, t->wMinute, t->wSecond);
    free (t);
    return;
}
```

Let’s compile it now with optimization (/Ox) so it would be easy see what we need.

List 21.4: Optimizing MSVC

```
_main PROC
    push esi
    push 16
    call _malloc
    add esp, 4
    mov esi, eax
    push esi
    call DWORD PTR __imp__GetSystemTime@4
    movzx eax, WORD PTR [esi+12] ; wSecond
    movzx ecx, WORD PTR [esi+10] ; wMinute
    movzx edx, WORD PTR [esi+8] ; wHour
    push eax
    movzx eax, WORD PTR [esi+6] ; wDay
    push ecx
    movzx ecx, WORD PTR [esi+2] ; wMonth
    push edx
    movzx edx, WORD PTR [esi] ; wYear
    push eax
    push ecx
    push edx
473
```
push OFFSET $SG78833
call _printf
push esi
call _free
add esp, 32
xor eax, eax
pop esi
ret 0

_main ENDP

So, sizeof(SYSTEMTIME) = 16 and that is exact number of bytes to be allocated by malloc(). It returns a pointer to a freshly allocated memory block in the EAX register, which is then moved into the ESI register. GetSystemTime() win32 function takes care of saving value in ESI, and that is why it is not saved here and continues to be used after the GetSystemTime() call.

New instruction — MOVZX (Move with Zero eXtend). It may be used in most cases as MOVSX, but it sets the remaining bits to 0. That's because printf() requires a 32-bit int, but we got a WORD in the structure — that is 16-bit unsigned type. That's why by copying the value from a WORD into int, bits from 16 to 31 must be cleared, because a random noise may be there, which is left from the previous operations on the register(s).

In this example, it's possible to represent the structure as an array of 8 WORDs:

```c
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,

    free (t);

    return;
}
```

We get:
Listing 21.5: Optimizing MSVC

```assembly
SG78594 DB '%04d-%02d-%02d %02d:%02d:%02d', 0AH, 00H

_main PROC
    push esi
    push 16
    call _malloc
    add esp, 4
    mov esi, eax
    push esi
    call DWORD PTR __imp__GetSystemTime@4
    movzx eax, WORD PTR [esi+12]
    movzx ecx, WORD PTR [esi+10]
    movzx edx, WORD PTR [esi+8]
    push eax
    movzx eax, WORD PTR [esi+6]
    push ecx
    movzx ecx, WORD PTR [esi+2]
    push edx
    movzx edx, WORD PTR [esi]
    push eax
    push ecx
    push edx
    push OFFSET $SG78594
    call __printf
    push esi
    call _free
    add esp, 32
    xor eax, eax
    pop esi
    ret 0

__main ENDP
```

Again, we got the code cannot be distinguished from the previous one. And again it should be noted, you haven’t to do this in practice, unless you really know what you are doing.

### 21.3 UNIX: struct tm

#### 21.3.1 Linux

Let’s take the `tm` structure from `time.h` in Linux for example:

```c
#include <stdio.h>
```
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
}

Let's compile it in GCC 4.4.1:

Listing 21.6: GCC 4.4.1

main proc near
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    sub     esp, 40h
    mov     dword ptr [esp], 0 ; first argument for time()
    call    time
    mov     [esp+3Ch], eax
    lea     eax, [esp+3Ch] ; take pointer to what time() returned
    lea     edx, [esp+10h] ; at ESP+10h struct tm will begin
    mov     [esp+4], edx ; pass pointer to the structure
    begin
    mov     [esp], eax ; pass pointer to result of time()
    call    localtime_r
    mov     eax, [esp+24h] ; tm_year
    lea     edx, [eax+76Ch] ; edx=eax+1900
    mov     eax, offset format ; "Year: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+20h] ; tm_mon
    mov     eax, offset aMonthD ; "Month: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
Somehow, IDA did not write the local variables’ names in the local stack. But since we already are experienced reverse engineers :-) we may do it without this information in this simple example.

Please also pay attention to the `lea edx, [eax+76Ch]` —this instruction just adds 0x76C (1900) to value in EAX, but doesn’t modify any flags. See also the relevant section about LEA (A.6.2 on page 1304).

**GDB**

Let’s try to load the example into GDB 4:

Listing 21.7: GDB

dennis@ubuntuvm:~:/polygon$ date
Mon Jun  2 18:10:37 EEST 2014
dennis@ubuntuvm:~:/polygon$ gcc GCC_tm.c -o GCC_tm
dennis@ubuntuvm:~:/polygon$ gdb GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.

---

4The `date` result is slightly corrected for demonstration purposes. Of course, it’s not possible to run GDB that quickly, in the same second.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>... Reading symbols from /home/dennis/polygon/GCC_tm...(no debugging symbols found)...done.
(gdb) b printf
Breakpoint 1 at 0x8048330
(gdb) run
Starting program: /home/dennis/polygon/GCC_tm
Breakpoint 1, __printf (format=0x80485c0 "Year: %d\n") at printf.c:29
29 printf.c: No such file or directory.
(gdb) x/20x $esp
0xbffff0dc: 0x080484c3 0x080485c0 0x000007de
0xbffff0ec: 0x08048301 0x538c93ed 0x00000025
0xbffff0fc: 0x00000012 0x00000002 0x0000005
0xbffff10c: 0x00000001 0x00000098 0x00000001
0xbffff11c: 0x0804b090 0x08048530 0x00000000
(gdb)

We can easily find our structure in the stack. First, let's see how it's defined in time.h:

Listing 21.8: time.h

```c
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
```
Pay attention that 32-bit \textit{int} is used here instead of \textit{WORD} in \texttt{SYSTEMTIME}. So, each field occupies 32-bit.

Here are the fields of our structure in the stack:

\begin{verbatim}
0xbffff0dc: 0x080484c3 0x080485c0 0x000007de  \\
  0x00000000
0xbffff0ec: 0x08048301 0x538c93ed 0x00000025 sec  \\
  0x0000000a min
0xbffff0fc: 0x00000012 hour 0x00000002 mday 0x00000005 mon  \\
  0x00000072 year
0xbffff10c: 0x00000001 wday 0x00000098 yday 0x00000001 isdst  \\
  0x00002a30
0xbffff11c: 0x0804b090 0x08048530 0x00000000  \\
  0x00000000
\end{verbatim}

Or as a table:

<table>
<thead>
<tr>
<th>Hexadecimal number</th>
<th>decimal number</th>
<th>field name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000025</td>
<td>37</td>
<td>tm_sec</td>
</tr>
<tr>
<td>0x0000000a</td>
<td>10</td>
<td>tm_min</td>
</tr>
<tr>
<td>0x00000012</td>
<td>18</td>
<td>tm_hour</td>
</tr>
<tr>
<td>0x00000002</td>
<td>2</td>
<td>tm_mday</td>
</tr>
<tr>
<td>0x00000005</td>
<td>5</td>
<td>tm_mon</td>
</tr>
<tr>
<td>0x00000072</td>
<td>114</td>
<td>tm_year</td>
</tr>
<tr>
<td>0x00000001</td>
<td>1</td>
<td>tm_wday</td>
</tr>
<tr>
<td>0x00000098</td>
<td>152</td>
<td>tm_yday</td>
</tr>
<tr>
<td>0x00000001</td>
<td>1</td>
<td>tm_isdst</td>
</tr>
</tbody>
</table>

Just like \texttt{SYSTEMTIME} (\ref{21.1 on page 467}), there are also other fields available that are not used, like \texttt{tm_wday}, \texttt{tm_yday}, \texttt{tm_isdst}.

\section*{21.3.2 ARM}

\subsection*{Optimizing Keil 6/2013 (Thumb mode)}

Same example:

\begin{verbatim}
Listing 21.9: Optimizing Keil 6/2013 (Thumb mode)
var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
\end{verbatim}
var_28 = -0x28
var_24 = -0x24
timer = -0xC

```assembly
PUSH {LR}
MOVS R0, #0 ; timer
SUB SP, SP, #0x34
BL time
STR R0, [SP,#0x38+timer]
MOV R1, SP ; tp
ADD R0, SP, #0x38+timer ; timer
BL localtime_r
LDR R1, =0x76C
LDR R0, [SP,#0x38+var_24]
ADDS R1, R0, R1
ADR R0, aYearD ; "Year: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_28]
ADR R0, aMonthD ; "Month: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_2C]
ADR R0, aDayD ; "Day: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_30]
ADR R0, aHourD ; "Hour: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_34]
ADR R0, aMinutesD ; "Minutes: %d\n"
BL __2printf
LDR R1, [SP,#0x38+var_38]
ADR R0, aSecondsD ; "Seconds: %d\n"
BL __2printf
ADD SP, SP, #0x34
POP {PC}
```

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

IDA “knows” the tm structure (because IDA “knows” the types of the arguments of library functions like localtime_r()), so it shows here structure elements accesses and their names.

Listing 21.10: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```assembly
var_38 = -0x38
var_34 = -0x34
```
PUSH {R7,LR}
MOV R7, SP
SUB SP, SP, #0x30
MOVS R0, #0 ; time_t *
BLX _time
ADD R1, SP, #0x38+var_34 ; struct tm *
STR R0, [SP,#0x38+var_38]
MOV R0, SP ; time_t *
BLX _localtime_r
LDR R1, [SP,#0x38+var_34.tm_year]
MOV R0, 0xF44 ; "Year: %d\n"
ADD R0, PC ; char *
ADDW R1, R1, #0x76C
BLX _printf
LDR R1, [SP,#0x38+var_34.tm_mon]
MOV R0, 0xF3A ; "Month: %d\n"
ADD R0, PC ; char *
BLX _printf
LDR R1, [SP,#0x38+var_34.tm_mday]
MOV R0, 0xF35 ; "Day: %d\n"
ADD R0, PC ; char *
BLX _printf
LDR R1, [SP,#0x38+var_34.tm_hour]
MOV R0, 0xF2E ; "Hour: %d\n"
ADD R0, PC ; char *
BLX _printf
LDR R1, [SP,#0x38+var_34.tm_min]
MOV R0, 0xF28 ; "Minutes: %d\n"
ADD R0, PC ; char *
BLX _printf
LDR R1, [SP,#0x38+var_34]
MOV R0, 0xF25 ; "Seconds: %d\n"
ADD R0, PC ; char *
BLX _printf
ADD SP, SP, #0x30
POP {R7,PC}

...
21.3.3 MIPS

Listing 21.11: Optimizing GCC 4.4.5 (IDA)

```assembly
main:

; IDA is not aware of structure field names, we named them manually:

var_40 = -0x40
var_38 = -0x38
seconds = -0x34
minutes = -0x30
hour = -0x2C
day = -0x28
month = -0x24
year = -0x20
var_4 = -4

lui $gp, (__gnu_local_gp >> 16)
addiu $sp, -0x50
la $gp, (__gnu_local_gp & 0xFFFF)
sw $ra, 0x50+var_4($sp)
sw $gp, 0x50+var_40($sp)
lw $t9, (time & 0xFFFF)($gp)
or $at, $zero ; load delay slot, NOP
jalr $t9
move $a0, $zero ; branch delay slot, NOP
lw $gp, 0x50+var_40($sp)
addiu $a0, $sp, 0x50+var_38
lw $t9, (localtime_r & 0xFFFF)($gp)
addiu $a1, $sp, 0x50+seconds
jalr $t9
sw $v0, 0x50+var_38($sp) ; branch delay slot
lw $gp, 0x50+var_40($sp)
lw $a1, 0x50+year($sp)
lw $t9, (printf & 0xFFFF)($gp)
la $a0, $LC0 # "Year: \%d\n"
jalr $t9
```

482
addiu $a1, 1900 ; branch delay slot
lw $gp, 0x50+var_40($sp)
lw $a1, 0x50+month($sp)
lw $t9, (printf & 0xFFFF)($gp)
lui $a0, ($LC1 >> 16) # "Month: %d\n"
jalr $t9
la $a0, ($LC1 & 0xFFFF) # "Month: %d\n" ;

branch delay slot
lw $gp, 0x50+var_40($sp)
lw $a1, 0x50+day($sp)
lw $t9, (printf & 0xFFFF)($gp)
lui $a0, ($LC2 >> 16) # "Day: %d\n"
jalr $t9
la $a0, ($LC2 & 0xFFFF) # "Day: %d\n" ;

branch delay slot
lw $gp, 0x50+var_40($sp)
lw $a1, 0x50+hour($sp)
lw $t9, (printf & 0xFFFF)($gp)
lui $a0, ($LC3 >> 16) # "Hour: %d\n"
jalr $t9
la $a0, ($LC3 & 0xFFFF) # "Hour: %d\n" ;

branch delay slot
lw $gp, 0x50+var_40($sp)
lw $a1, 0x50+minutes($sp)
lw $t9, (printf & 0xFFFF)($gp)
lui $a0, ($LC4 >> 16) # "Minutes: %d\n"
jalr $t9
la $a0, ($LC4 & 0xFFFF) # "Minutes: %d\n" ;

branch delay slot
lw $gp, 0x50+var_40($sp)
lw $a1, 0x50+seconds($sp)
lw $t9, (printf & 0xFFFF)($gp)
lui $a0, ($LC5 >> 16) # "Seconds: %d\n"
jalr $t9
la $a0, ($LC5 & 0xFFFF) # "Seconds: %d\n" ;

branch delay slot
lw $ra, 0x50+var_4($sp)
or $at, $zero ; load delay slot, NOP
jr $ra
addiu $sp, 0x50

$LC0: .ascii "Year: %d\n"<0>
$LC1: .ascii "Month: %d\n"<0>
$LC2: .ascii "Day: %d\n"<0>
$LC3: .ascii "Hour: %d\n"<0>
$LC4: .ascii "Minutes: %d\n"<0>
$LC5: .ascii "Seconds: %d\n"<0>
This is an example where the branch delay slots can confuse us. For example, there is the instruction “addiu $a1, 1900” at line 35 which adds 1900 to the year number. It’s executed before the corresponding JALR at line 34, do not forget about it.

### 21.3.4 Structure as a set of values

In order to illustrate that the structure is just variables laying side-by-side in one place, let’s rework our example while looking at the `tm` structure definition again: listing 21.8.

```c
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year,
    \< tm_wday, tm_yday, tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);
    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
}
```

N.B. The pointer to the `tm_sec` field is passed into `localtime_r`, i.e., to the first element of the “structure”.

The compiler warns us:

```
Listing 21.12: GCC 4.7.3

GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from incompatible pointer type [enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of type 'int *'
```

But nevertheless, it generates this:
Listing 21.13: GCC 4.7.3

main proc near

var_30 = dword ptr -30h
var_2C = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec = dword ptr -18h
tm_min = dword ptr -14h
tm_hour = dword ptr -10h
tm_mday = dword ptr -0Ch
tm_mon = dword ptr -8

push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 30h
call __main
mov [esp+30h+var_30], 0 ; arg 0
call time
mov [esp+30h+unix_time], eax
lea eax, [esp+30h+tm_sec]
mov [esp+30h+var_2C], eax
lea eax, [esp+30h+unix_time]
mov [esp+30h+var_30], eax
call localtime_r
mov eax, [esp+30h+tm_year]
add eax, 1900
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aYearD ; "Year: %d\n"
call printf
mov eax, [esp+30h+tm_mon]
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
call printf
mov eax, [esp+30h+tm_mday]
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aDayD ; "Day: %d\n"
call printf
mov eax, [esp+30h+tm_hour]
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
call printf
mov eax, [esp+30h+tm_min]
This code is identical to what we saw previously and it is not possible to say, was it a structure in original source code or just a pack of variables.

And this works. However, it is not recommended to do this in practice. Usually, non-optimizing compilers allocates variables in the local stack in the same order as they were declared in the function. Nevertheless, there is no guarantee.

By the way, some other compiler may warn about the `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min` variables, but not `tm_sec` are used without being initialized. Indeed, the compiler is not aware that these are to be filled by `localtime_r()` function.

We chose this example, since all structure fields are of type `int`. This would not work if structure fields are 16-bit (WORD), like in the case of the `SYSTEMTIME` structure—`GetSystemTime()` will fill them incorrectly (because the local variables are aligned on a 32-bit boundary). Read more about it in next section: “Fields packing in structure” (21.4 on page 490).

So, a structure is just a pack of variables laying on one place, side-by-side. We could say that the structure is the instruction to the compiler, directing it to hold variables in one place. By the way, in some very early C versions (before 1972), there were no structures at all [Rit93].

There is no debugger example here: it is just the same as you already saw.

### 21.3.5 Structure as an array of 32-bit words

```c
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
}```
```c
int i;

unix_time=time(NULL);

localtime_r (&unix_time, &t);

for (i=0; i<9; i++)
{
    int tmp=((int*)&t)[i];
    printf("0x%08X (%d)\n", tmp, tmp);
}

We just cast a pointer to structure to an array of int's. And that works! We run the example at 23:51:45 26-July-2014.

0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
0x00000001 (1)

The variables here are in the same order as they are enumerated in the definition of the structure: 21.8 on page 478.

Here is how it gets compiled:

Listing 21.14: Optimizing GCC 4.8.1
```

```
main proc near
    push ebp
    mov ebp, esp
    push esi
    push ebx
    and esp, 0FFFFFFF0h
    sub esp, 40h
    mov dword ptr [esp], 0 ; timer
    lea ebx, [esp+14h]
    call _time
    lea esi, [esp+38h]
    mov [esp+4], ebx ; tp
    mov [esp+10h], eax
    lea eax, [esp+10h]
    mov [esp], eax ; timer
```
Indeed: the space in the local stack is first treated as a structure, and then it's treated as an array.

It’s even possible to modify the fields of the structure through this pointer.

And again, it’s dubiously hackish way to do things, not recommended for use in production code.

**Exercise**

As an exercise, try to modify (increase by 1) the current month number, treating the structure as an array.

### 21.3.6 Structure as an array of bytes

We can go even further. Let’s cast the pointer to an array of bytes and dump it:

```c
#include <stdio.h>
#include <time.h>
```
void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);
    localtime_r (&unix_time, &t);
    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf("\n");
    }
};

0x2D 0x00 0x00 0x00
0x33 0x00 0x00 0x00
0x17 0x00 0x00 0x00
0x1A 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0x72 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0xCE 0x00 0x00 0x00
0x01 0x00 0x00 0x00

We also run this example also at 23:51:45 26-July-2014. The values are just the same as in the previous dump (21.3.5 on page 487), and of course, the lowest byte goes first, because this is a little-endian architecture (31 on page 619).

Listing 21.15: Optimizing GCC 4.8.1

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>proc near</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push</td>
<td>ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>ebp, esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push</td>
<td>edi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push</td>
<td>esi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push</td>
<td>ebx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>esp, 0FFFFFFFF0h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>esp, 40h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>dword ptr [esp], 0 ; timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lea</td>
<td>esi, [esp+14h]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>_time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^5\)The time and date are the same for demonstration purposes. Byte values are fixed up.
CHAPTER 21. STRUCTURES

21.4 Fields packing in structure

One important thing is fields packing in structures\(^6\).

Let's take a simple example:

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

---

\(^6\)See also: [Wikipedia: Data structure alignment](https://en.wikipedia.org/wiki/Data_structure_alignment)
As we see, we have two `char` fields (each is exactly one byte) and two more — `int` (each — 4 bytes).

### 21.4.1 x86

This compiles to:

```
Listing 21.16: MSVC 2012 /GS- /Ob0

1  _tmp$ = -16
2  _main PROC
3    push ebp
4    mov ebp, esp
5    sub esp, 16
6    mov BYTE PTR _tmp$[ebp], 1    ; set field a
7    mov DWORD PTR _tmp$[ebp+4], 2    ; set field b
8    mov BYTE PTR _tmp$[ebp+8], 3    ; set field c
9    mov DWORD PTR _tmp$[ebp+12], 4    ; set field d
10   sub esp, 16                      ; allocate place for
temporary structure
11   mov eax, esp
12   mov ecx, DWORD PTR _tmp$[ebp]    ; copy our structure to
the temporary one
```
We pass the structure as a whole, but in fact, as we can see, the structure is being copied to a temporary one (a place in stack is allocated in line 10 for it, and then all 4 fields, one by one, are copied in lines 12 ... 19), and then its pointer (address) is to be passed. The structure is copied because it’s not known whether the \( f() \) function going to modify the structure or not. If it gets changed, then the structure in main() has to remain as it was. We could use C/C++ pointers, and the resulting code will be almost the same, but without the copying.

As we can see, each field’s address is aligned on a 4-byte boundary. That’s why each char occupies 4 bytes here (like int). Why? Because it is easier for the CPU to access memory at aligned addresses and to cache data from it.
However, it is not very economical.

Let’s try to compile it with option (/Zp1) (\Zp[n] pack structures on n-byte boundary).

Listing 21.17: MSVC 2012 /GS- /Zp1

```
  _main PROC
  push ebp
  mov ebp, esp
  sub esp, 12
  mov BYTE PTR _tmp$[ebp], 1 ; set field a
  mov DWORD PTR _tmp$[ebp+1], 2 ; set field b
  mov BYTE PTR _tmp$[ebp+5], 3 ; set field c
  mov DWORD PTR _tmp$[ebp+6], 4 ; set field d
  sub esp, 12 ; allocate place for temporary structure
  mov eax, esp
  mov ecx, DWORD PTR _tmp$[ebp] ; copy 10 bytes
  mov DWORD PTR [eax], ecx
  mov edx, DWORD PTR _tmp$[ebp+4]
  mov DWORD PTR [eax+4], edx
  mov cx, WORD PTR _tmp$[ebp+8]
  mov WORD PTR [eax+8], cx
  call _f
  add esp, 12
  xor eax, eax
  mov esp, ebp
  pop ebp
  ret 0
  _main ENDP

__TEXT SEGMENT
_s$ = 8 ; size = 10
?f@@YAXUs@@@Z PROC ; f
  push ebp
  mov ebp, esp
  mov eax, DWORD PTR _s$[ebp+6]
  push eax
  movsx ecx, BYTE PTR _s$[ebp+5]
  push ecx
  mov edx, DWORD PTR _s$[ebp+1]
  push edx
  movsx eax, BYTE PTR _s$[ebp]
  push eax
  push OFFSET $SG3842
  call _printf
  add esp, 20
  pop ebp
  ret 0
```
Now the structure takes only 10 bytes and each char value takes 1 byte. What does it give to us? Size economy. And as drawback—the CPU accessing these fields slower than it could.

The structure is also copied in main(). Not field-by-field, but directly 10 bytes, using three pairs of MOV. Why not 4? The compiler decided that it’s better to copy 10 bytes using 3 MOV pairs than to copy two 32-bit words and two bytes using 4 MOV pairs. By the way, such copy implementation using MOV instead of calling the memcpy() function is widely used, because it’s faster than a call to memcpy()—for short blocks, of course: 43.1.5 on page 702.

As it can be easily guessed, if the structure is used in many source and object files, all these must be compiled with the same convention about structures packing.

Aside from MSVC /Zp option which sets how to align each structure field, there is also the #pragma pack compiler option, which can be defined right in the source code. It is available in both MSVC\textsuperscript{7} and GCC\textsuperscript{8}.

Let’s get back to the SYSTEMTIME structure that consists of 16-bit fields. How does our compiler know to pack them on 1-byte alignment boundary?

WinNT.h file has this:

```
Listing 21.18: WinNT.h
#include "pshpack1.h"
```

And this:

```
Listing 21.19: WinNT.h
#include "pshpack4.h" // 4 byte packing is \textsuperscript{\leftarrow} the default
```

The file PshPack1.h looks like:

```
Listing 21.20: PshPack1.h
#if ! (defined(lint) \|\| defined(RC_INVOKED))
#if ( _MSC_VER >= 800 \&\& !defined(_M_I86)) \|\| defined(_\PUSHPOP\_SUPPORTED)
#pragma warning(disable:4103)
#else
#endif
#pragma pack(push,1)
#else
#endif
```

\textsuperscript{7} MSDN: Working with Packing Structures
\textsuperscript{8} Structure-Packing Pragmas
This tells the compiler how to pack the structures defined after `#pragma pack`. 
OllyDbg + fields are packed by default

Let's try our example (where the fields are aligned by default (4 bytes)) in OllyDbg:

![Figure 21.3: OllyDbg: Before printf() execution](image)

We see our 4 fields in the data window. But where do the random bytes (0x30, 0x37, 0x01) come from, that are next to the first (a) and third (c) fields? By looking at our listing 21.16 on page 491, we can see that the first and third fields are char, therefore only one byte is written, 1 and 3 respectively (lines 6 and 8). The remaining 3 bytes of the 32-bit words are not being modified in memory! Hence, random garbage is left there. This garbage doesn't influence the printf() output in any way, because the values for it are prepared using the MOVSX instruction, which takes bytes, not words: listing.21.16 (lines 34 and 38).

By the way, the MOVSX (sign-extending) instruction is used here, because char is signed by default in MSVC and GCC. If the type unsigned char or uint8_t was used here, MOVZX instruction would have been used instead.
OllyDbg + fields aligning on 1 byte boundary

Things are much clearer here: 4 fields occupy 10 bytes and the values are stored side-by-side

![CPU - main thread, module packing](image)

**Figure 21.4:** OllyDbg: Before `printf()` execution

### 21.4.2 ARM

Optimizing Keil 6/2013 (Thumb mode)

**Listing 21.21:** Optimizing Keil 6/2013 (Thumb mode)

```assembly
.text:0000003E  exit ; CODE XREF: f+16
.text:0000003E 05 B0     ADD     SP, SP, #0x14
.text:00000040 00 BD     POP {PC}

.text:00000280  f
.text:00000280
.text:00000280  var_18 = -0x18
.text:00000280  a = -0x14
.text:00000280  b = -0x10
.text:00000280  c = -0xC
.text:00000280  d = -8
.text:00000280
.text:00000280  0F B5     PUSH {R0-R3, LR}
.text:00000282  81 B0     SUB     SP, SP, #4
.text:00000284  04 98     LDR     R0, [SP,#14] ; d
.text:00000286  02 9A     LDR     R2, [SP,#8] ; b
.text:00000288  00 90     STR     R0, [SP]
.text:0000028A  68 46     MOV     R0, SP
.text:0000028C  03 7B     LDRB    R3, [R0,#12] ; c
```
As we may recall, here a structure is passed instead of pointer to one, and since the first 4 function arguments in ARM are passed via registers, the structure’s fields are passed via R0-R3.

LDRB loads one byte from memory and extends it to 32-bit, taking its sign into account. This is similar to MOVVSX in x86. Here it is used to load fields \( a \) and \( c \) from the structure.

One more thing we spot easily is that instead of function epilogue, there is jump to another function’s epilogue! Indeed, that was quite different function, not related in any way to ours, however, it has exactly the same epilogue (probably because, it hold 5 local variables too (\( 5 \times 4 = 0x14 \)). Also it is located nearby (take a look at the addresses). Indeed, it doesn’t matter which epilogue gets executed, if it works just as we need. Apparently, Keil decides to reuse a part of another function to economize. The epilogue takes 4 bytes while jump — only 2.

**ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)**

```
Listing 21.22: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

```

```assembly
var_C = -0xC

PUSH {R7,LR}
MOV R7, SP
SUB SP, SP, #4
MOV R9, R1 ; b
MOV R1, R0 ; a
MOV R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
SXTB R1, R1 ; prepare a
MOVT.W R0, #0
STR R3, [SP,#0xC+var_C] ; place d to stack for printf
)
ADD R0, PC ; format-string
SXTB R3, R2 ; prepare c
MOV R2, R9 ; b
BLX __2printf
ADD SP, SP, #4
POP {R7,PC}
```
SXTB (Signed Extend Byte) is analogous to MOVsx in x86. All the rest — just the same.

### 21.4.3 MIPS

Listing 21.23: Optimizing GCC 4.4.5 (IDA)

```plaintext
f:
var_18 = -0x18
var_10 = -0x10
var_4  = -4
arg_0  =  0
arg_4  =  4
arg_8  =  8
arg_C  =  0xC

; $a0 = s.a
; $a1 = s.b
; $a2 = s.c
; $a3 = s.d
lui $gp, (__gnu_local_gp >> 16)
addiu $sp, -0x28
la $gp, (__gnu_local_gp & 0xFFFF)
sw $ra, 0x28+var_4($sp)
sw $gp, 0x28+var_10($sp)
; prepare byte from 32-bit big-endian integer:
sra $t0, $a0, 24
move $v1, $a1
; prepare byte from 32-bit big-endian integer:
sra $v0, $a2, 24
lw $t9, (printf & 0xFFFF)($gp)
sw $a0, 0x28+arg_0($sp)
lui $a0, ($LC0 >> 16)  # "a=%d; b=%d; c=%d; d=%d
sw $a3, 0x28+var_18($sp)
sw $a1, 0x28+arg_4($sp)
sw $a2, 0x28+arg_8($sp)
sw $a3, 0x28+arg_C($sp)
la $a0, ($LC0 & 0xFFFF)  # "a=%d; b=%d; c=%d; d=%d
move $a1, $t0
move $a2, $v1
jalr $t9
move $a3, $v0 ; branch delay slot
lw $ra, 0x28+var_4($sp)
```
Structure fields come in registers $A0..$A3 and then get reshuffled into $A1..$A4 for printf(). But there are two SRA (“Shift Word Right Arithmetic”) instructions, which prepare char fields. Why? MIPS is a big-endian architecture by default on page 619, and the Debian Linux we work in is big-endian as well. So when byte variables are stored in 32-bit structure slots, they occupy the high 31..24 bits. And when a char variable needs to be extended into a 32-bit value, it must be shifted right by 24 bits. char is a signed type, so an arithmetical shift is used here instead of logical.

### 21.4.4 One more word

Passing a structure as a function argument (instead of a passing pointer to structure) is the same as passing all structure fields one by one. If the structure fields are packed by default, the f() function can be rewritten as:

```c
void f(char a, int b, char c, int d)
{
    printf("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

And that leads to the same code.

### 21.5 Nested structures

Now what about situations when one structure is defined inside of another?

```c
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
```
int b;
struct inner_struct c;
char d;
int e;
}

void f(struct outer_struct s)
{
  printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
          s.a, s.b, s.c.a, s.c.b, s.d, s.e);
}

int main()
{
  struct outer_struct s;
  s.a=1;
  s.b=2;
  s.c.a=100;
  s.c.b=101;
  s.d=3;
  s.e=4;
  f(s);
}

... in this case, both inner_struct fields are to be placed between the a,b and d,e fields of the outer_struct.

Let's compile (MSVC 2010):

Listing 21.24: Optimizing MSVC 2010 /Ob0

$SG2802 DB 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0AH, 00

_TEXT SEGMENT
_s$ = 8
_f PROC
  mov   eax, DWORD PTR _s$[esp+16]
  movsx ecx, BYTE PTR _s$[esp+12]
  mov   edx, DWORD PTR _s$[esp+8]
  push  eax
  mov   eax, DWORD PTR _s$[esp+8]
  push  ecx
  mov   ecx, DWORD PTR _s$[esp+8]
  push  edx
  movsx edx, BYTE PTR _s$[esp+8]
  push  eax
  push  ecx

501
One curious thing here is that by looking onto this assembly code, we do not even see that another structure was used inside of it! Thus, we would say, nested structures are unfolded into linear or one-dimensional structure.

Of course, if we replace the struct inner_struct c; declaration with struct inner_struct *c; (thus making a pointer here) the situation will be quite different.
21.5.1 OllyDbg

Let’s load the example into OllyDbg and take a look at outer_struct in memory:

![OllyDbg: Before printf() execution](image)

Figure 21.5: OllyDbg: Before printf() execution

That’s how the values are located in memory:

- *(outer_struct.a)* (byte) 1 + 3 bytes of random garbage;
- *(outer_struct.b)* (32-bit word) 2;
- *(inner_struct.a)* (32-bit word) 0x64 (100);
- *(inner_struct.b)* (32-bit word) 0x65 (101);
- *(outer_struct.d)* (byte) 3 + 3 bytes of random garbage;
- *(outer_struct.e)* (32-bit word) 4.

21.6 Bit fields in a structure

21.6.1 CPUID example

The C/C++ language allows to define the exact number of bits for each structure field. It is very useful if one needs to save memory space. For example, one bit is enough for a *bool* variable. But of course, it is not rational if speed is important.

Let’s consider the CPUID\(^9\) instruction example. This instruction returns information about the current CPU and its features.

If the EAX is set to 1 before the instruction’s execution, CPUID returning this information packed into the EAX register:

\(^9\)wikipedia
MSVC 2010 has CPUID macro, but GCC 4.4.1 does not. So let’s make this function by ourselves for GCC with the help of its built-in assembler\(^\text{10}\).

```c
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
    asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"("code"));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX {
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main() {
    struct CPUID_1_EAX *tmp;
    int b[4];
    #ifdef _MSC_VER
    __cpuid(b,1);
    #endif
    #ifdef __GNUC__

\(^{10}\text{More about internal GCC assembler}\)
After CPUID fills EAX/EBX/ECX/EDX, these registers are to be written in the b[] array. Then, we have a pointer to the CPUID_1_EAX structure and we point it to the value in EAX from the b[] array.

In other words, we treat a 32-bit int value as a structure. Then we read specific bits from the structure.

**MSVC**

Let's compile it in MSVC 2008 with /Ox option:

```c
_b$ = -16 ; size = 16
_main PROC
    sub    esp, 16
    push   ebx
    xor    ecx, ecx
    mov    eax, 1
    cpuid
    push   esi
    lea    esi, DWORD PTR _b$[esp+24]
    mov    DWORD PTR [esi], eax
    mov    DWORD PTR [esi+4], ebx
    mov    DWORD PTR [esi+8], ecx
    mov    DWORD PTR [esi+12], edx
    mov    esi, DWORD PTR _b$[esp+24]
    mov    eax, esi
    and    eax, 15
return 0;
};
```
push eax
push OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
call _printf

mov ecx, esi
shr ecx, 4
and ecx, 15
push ecx
push OFFSET $SG15436 ; 'model=%d', 0aH, 00H
call _printf

mov edx, esi
shr edx, 8
and edx, 15
push edx
push OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
call _printf

mov eax, esi
shr eax, 12
and eax, 3
push eax
push OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
call _printf

mov ecx, esi
shr ecx, 16
and ecx, 15
push ecx
push OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
call _printf

shr esi, 20
and esi, 255
push esi
push OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
call _printf
add esp, 48
pop esi

xor eax, eax
pop ebx

add esp, 16
ret 0
_main ENDP
The SHR instruction shifting the value in EAX by the number of bits that must be skipped, e.g., we ignore some bits at the right side.

The AND instruction clears the unneeded bits on the left, or, in other words, leaves only those bits in the EAX register we need.
Let’s load our example into OllyDbg and see, what values are set in EAX/EBX/ECX/EDX after the execution of CPUID:

![Figure 21.6: OllyDbg: After CPUID execution](image)

EAX has 0x000206A7 (my CPU is Intel Xeon E3-1220). This is 00000000000000100000011010100111 in binary form.

Here is how the bits are distributed by fields:

<table>
<thead>
<tr>
<th>field</th>
<th>in binary form</th>
<th>in decimal form</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved2</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>extended_family_id</td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>extended_model_id</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>reserved1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>processor_id</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>family_id</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>model</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>stepping</td>
<td>0111</td>
<td>7</td>
</tr>
</tbody>
</table>
GCC

Let's try GCC 4.4.1 with -O3 option.

Listing 21.26: Optimizing GCC 4.4.1

```assembly
main proc near ; DATA XREF: _start+17
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    push esi
    mov esi, 1
    push ebx
    mov eax, esi
    sub esp, 18h
    cpuid
    mov esi, eax
    and eax, 0Fh
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aSteppingD ; "stepping=%d
\
\n"
    mov dword ptr [esp], 1
    call ___printf_chk
    mov eax, esi
    shr eax, 4
    and eax, 0Fh
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aModelD ; "model=%d
\n"
    mov dword ptr [esp], 1
    call ___printf_chk
    mov eax, esi
    shr eax, 8
```

Figure 21.7: OllyDbg: Result
Almost the same. The only thing worth noting is that GCC somehow combines the calculation of extended_model_id and extended_family_id into one block, instead of calculating them separately before each printf() call.

### 21.6.2 Working with the float type as with a structure

As we already noted in the section about FPU (17 on page 298), both float and double types consist of a sign, a significand (or fraction) and an exponent. But will
we be able to work with these fields directly? Let’s try this with float.

![Diagram showing the structure of a float](image)

(S—sign)

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // fractional part
    unsigned int exponent : 8; // exponent + 0x3FF
    unsigned int sign : 1; // sign bit
};

float f(float in)
{
    float f = in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));
    memcpy (&t, &f, sizeof (float));
    t.sign = 1; // set negative sign
    t.exponent = t.exponent + 2; // multiply d by $2^n$ (n here is 2)
    memcpy (&f, &t, sizeof (float));
    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};
```

The float_as_struct structure occupies the same amount of memory as float, i.e., 4 bytes or 32 bits.

Now we are setting the negative sign in the input value and also, by adding 2 to the exponent, we thereby multiply the whole number by $2^2$, i.e., by 4.

Let’s compile in MSVC 2008 without optimization turned on:
Listing 21.27: Non-optimizing MSVC 2008

```
_t$ = -8 ; size = 4
_f$ = -4 ; size = 4
__in$ = 8 ; size = 4
?f@@YAMM@Z PROC ; f
    push ebp
    mov ebp, esp
    sub esp, 8

    fld DWORD PTR __in$[ebp]
    fstp DWORD PTR _f$[ebp]

    push 4
    lea eax, DWORD PTR _f$[ebp]
    push eax
    lea ecx, DWORD PTR _t$[ebp]
    push ecx
    call __memcpy
    add esp, 12

    mov edx, DWORD PTR _t$[ebp]
    or edx, -2147483648 ; 80000000H - set minus sign
    mov DWORD PTR _t$[ebp], edx

    mov eax, DWORD PTR _t$[ebp]
    shr eax, 23 ; 00000017H - drop significand
    and eax, 255 ; 000000ffH - leave here only exponent
    add eax, 2 ; add 2 to it
    and eax, 255 ; 000000ffH
    shl eax, 23 ; 00000017H - shift result to place of bits 30:23
    mov ecx, DWORD PTR _t$[ebp]
    and ecx, -2139095041 ; 807ffffffH - drop exponent

    or ecx, eax
    mov DWORD PTR _t$[ebp], ecx

    push 4
    lea edx, DWORD PTR _t$[ebp]
    push edx
    lea eax, DWORD PTR _f$[ebp]
    push eax
    call __memcpy
    add esp, 12
```
A bit redundant. If it was compiled with /Ox flag there would be no memcpy() call, the f variable is used directly. But it is easier to understand by looking at the unoptimized version.

What would GCC 4.4.1 with -O3 do?

Listing 21.28: Optimizing GCC 4.4.1

; f(float)
public _Z1ff

_Z1ff proc near

var_4 = dword ptr -4
arg_0 = dword ptr 8

push ebp
mov ebp, esp
sub esp, 4
mov eax, [ebp+arg_0]
or eax, 80000000h ; set minus sign
mov edx, eax
and eax, 807FFFFFh ; leave only significand and exponent in EAX
shr edx, 23 ; prepare exponent
add edx, 2 ; add 2
movzx edx, dl ; clear all bits except 7:0 in EAX
shl edx, 23 ; shift new calculated exponent to its place
or eax, edx ; add new exponent and original value without exponent
mov [ebp+var_4], eax
fld [ebp+var_4]
leave
retn

_Z1ff endp

main proc near
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
The function is almost understandable. However, what is interesting is that GCC was able to calculate the result of `f(1.234)` during compilation despite all this hodge-podge with the structure fields and prepared this argument to `printf()` as precalculated at compile time!

### 21.7 Exercises

- [http://challenges.re/71](http://challenges.re/71)
- [http://challenges.re/72](http://challenges.re/72)
Chapter 22

Unions

C/C++ union is mostly used for interpreting a variable (or memory block) of one data type as a variable of another data type.

22.1 Pseudo-random number generator example

If we need float random numbers between 0 and 1, the simplest thing is to use a PRNG like the Mersenne twister. It produces random 32-bit values in DWORD form. Then we can transform this value to float and then divide it by RAND_MAX (0xFFFFFFFF in our case)—we getting a value in the 0..1 interval.

But as we know, division is slow. Also, we would like to issue as few FPU operations as possible. Can we get rid of the division?

Let’s recall what a floating point number consists of: sign bit, significand bits and exponent bits. We just need to store random bits in all significand bits to get a random float number!

The exponent cannot be zero (the floating number is denormalized in this case), so we are storing 01111111 to exponent—this means that the exponent is 1. Then we filling the significand with random bits, set the sign bit to 0 (which means a positive number) and voilà. The generated numbers is to be between 1 and 2, so we must also subtract 1.

A very simple linear congruential random numbers generator is used in my example\(^1\), it produces 32-bit numbers. The PRNG is initialized with the current time in UNIX timestamp format.

\(^1\)the idea was taken from: http://go.yurichev.com/17308
Here we represent the float type as an union—it is the C/C++ construction that enables us to interpret a piece of memory as different types. In our case, we are able to create a variable of type union and then access to it as it is float or as it is uint32_t. It can be said, it is just a hack. A dirty one.

The integer PRNG code is the same as we already considered: 20 on page 459. So this code in compiled form is omitted.

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

// integer PRNG definitions, data and routines:

// constants from the Numerical Recipes book
const uint32_t RNG_a=1664525;
cost uint32_t RNG_c=1013904223;
uint32_t RNG_state; // global variable

void my_srand(uint32_t i)
{
    RNG_state=i;
};

uint32_t my_rand()
{
    RNG_state=RNG_state*RNG_a+RNG_c;
    return RNG_state;
};

// FPU PRNG definitions and routines:

union uint32_t_float
{
    uint32_t i;
    float f;
};

float float_rand()
{
    union uint32_t_float tmp;
    tmp.i=my_rand() & 0x007fffffff | 0x3F800000;
    return tmp.f-1;
};

// test

int main()
```
CHAPTER 22. UNIONS

{  
    my_srand(time(NULL)); // PRNG initialization

    for (int i=0; i<100; i++)
        printf ("%f\n", float_rand());

    return 0;
};

22.1.1 x86

Listing 22.1: Optimizing MSVC 2010

$SG4238 DB '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff0000000000000r ; 1

tv130 = -4
_tmp$ = -4

?float_rand@@YAMXZ PROC
    push ecx
    call ?my_rand@@YAIXZ
    ; EAX=pseudorandom value
    and eax, 8388607 ; 007
    ↓ fffffH
    or eax, 1065353216 ; 3
    ↓ f800000H
    ; EAX=pseudorandom value & 0x007fffffff | 0x3f800000
    ; store it into local stack:
    mov DWORD PTR _tmp$[esp+4], eax
    ; reload it as float point number:
    fld DWORD PTR _tmp$[esp+4]
    ; subtract 1.0:
    fsub QWORD PTR __real@3ff000000000000000000
    ; store value we got into local stack and reload it:
    fstp DWORD PTR tv130[esp+4] ; \ these instructions are redundant
    fld DWORD PTR tv130[esp+4] ; /
    pop ecx
    ret 0
?float_rand@@YAMXZ ENDP

_main PROC
    push esi
    xor eax, eax
    call _time

517
CHAPTER 22. UNIONS

Function names are so strange here because this example was compiled as C++ and this is name mangling in C++, we will talk about it later: 51.1.1 on page 757.

If we compile this in MSVC 2012, it uses the SIMD instructions for the FPU, read more about it here: 27.5 on page 604.

### 22.1.2 MIPS

Listing 22.2: Optimizing GCC 4.4.5

```
float_rand:

var_10 = -0x10
var_4 = -4

lui $gp, (__gnu_local_gp >> 16)
addiu $sp, -0x20
la $gp, (__gnu_local_gp & 0xFFFF)
sw $ra, 0x20+var_4($sp)
sw $gp, 0x20+var_10($sp)

; call my_rand():
jal my_rand
or $at, $zero; branch delay slot, NOP

; $v0=32-bit pseudorandom value
li $v1, 0x7FFFFF

; $v1=0x7FFFFF
and $v1, $v0, $v1

; $v1=pseudorandom value & 0x7FFFFF
```
lui $a0, 0x3F80  
; $a0=0x3F800000
or $v1, $a0  
; $v1=pseudorandom value & 0x7FFFFF | 0x3F800000
; matter of the following instruction is still hard to get:
lui $v0, ($LC0 >> 16)  
; load 1.0 into $f0:
lwc1 $f0, $LC0  
; move value from $v1 to coprocessor 1 (into register $f2)
; it behaves like bitwise copy, no conversion done:
mtc1 $v1, $f2  
lw $ra, 0x20+var_4($sp)  
; subtract 1.0. leave result in $f0:
sub.s $f0, $f2, $f0  
jr $ra
addiu $sp, 0x20 ; branch delay slot

main:

var_18 = -0x18
var_10 = -0x10
var_C = -0xC
var_8 = -8
var_4 = -4

lui $gp, (__gnu_local_gp >> 16)
addiu $sp, -0x28
la $gp, (__gnu_local_gp & 0xFFFF)
sw $ra, 0x28+var_4($sp)
sw $s2, 0x28+var_8($sp)
sw $s1, 0x28+var_C($sp)
sw $s0, 0x28+var_10($sp)
sw $gp, 0x28+var_18($sp)
lw $t9, (time & 0xFFFF)($gp)
or $at, $zero ; load delay slot, NOP
jalr $t9
move $a0, $zero ; branch delay slot
lui $s2, ($LC1 >> 16) # "%f
move $a0, $v0
la $s2, ($LC1 & 0xFFFF) # "%f
move $s0, $zero
jal my_srand
li $s1, 0x64 # 'd' ; branch delay slot
loc_104:
jal float_rand
addiu $s0, 1
CHAPTER 22. UNIONS

Listing 22.3: Optimizing GCC 4.6.3 (IDA)

float_rand
  STMFD SP!, {R3,LR}
  ; R0=pseudorandom value
  FLDS  S0, =1.0
  ; S0=1.0
  BIC  R3, R0, #0xFF000000
  BIC  R3, R3, #0x800000
  ORR  R3, R3, #0x3F800000
  ; R3=pseudorandom value & 0x007ffffff | 0x3f800000
  ; copy from R3 to FPU (register S15).
  ; it behaves like bitwise copy, no conversion done:
  FMSR  S15, R3
  ; subtract 1.0 and leave result in S0:
  FSUBS  S0, S15, S0
  LDMFD  SP!, {R3,PC}

flt_5C  DCFS 1.0

There is also an useless LUI instruction added for some weird reason. We considered this artifact earlier: 17.5.6 on page 313.

22.1.3 ARM (ARM mode)

Listing 22.3: Optimizing GCC 4.6.3 (IDA)
main

STMFD SP!, {R4,LR}
MOV R0, #0
BL  time
BL  my_srand
MOV R4, #0x64 ; 'd'

loc_78
BL  float_rand

; S0=pseudorandom value
LDR R0, =aF  ; "%f"
; convert float type value into double type value (printf() will need it):
FCVTDS D7, S0
; bitwise copy from D7 into R2/R3 pair of registers (for printf()):
FMRRD  R2, R3, D7
BL  printf
SUBS  R4, R4, #1
BNE  loc_78
MOV  R0, R4
LDMFD  SP!, {R4,PC}

aF  DCB "%f",0xA,0

We'll also make a dump in objdump and we'll see that the FPU instructions have different names than in IDA. Apparently, IDA and binutils developers used different manuals? Perhaps, it would be good to know both instruction name variants.

Listing 22.4: Optimizing GCC 4.6.3 (objdump)

```
00000038  <float_rand>:
  38:  e92d4008   push {r3, lr}
  3c:  ebfffffe  bl  10 <my_rand>
  40:  ed9f0a05  vldr  s0, [pc, #20]  ; 5c <
       xff000000
  44:  e3c034ff  bic  r3, r0, #-16777216  ; 0
       x800000
  48:  e3c33502  bic  r3, r3, #8388608  ; 0
       x800000
  4c:  e38335fe  orr  r3, r3, #1065353216  ; 0
       x3f800000
  50:  ee073a90  vmov  s15, r3
  54:  ee370ac0  vsub.f32  s0, s15, s0
  58:  e8bd8008   pop  {r3, pc}
  5c:  3f800000  svccc  0x00800000
```
The instructions at 0x5c in float_rand() and at 0x38 in main() are random noise.

### 22.2 Calculating machine epsilon

The machine epsilon is the smallest possible value the FPU can work with. The more bits allocated for floating point number, the smaller the machine epsilon. It is $2^{-23} = 1.19e-07$ for float and $2^{-52} = 2.22e-16$ for double.

It's interesting, how easy it's to calculate the machine epsilon:

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

union uint_float
{
    uint32_t i;
    float f;
};

float calculate_machine_epsilon(float start)
{
    union uint_float v;
    v.f=start;
    v.i++;
    return v.f-start;
}

void main()
```
What we do here is just treat the fraction part of the IEEE 754 number as integer and add 1 to it. The resulting floating number is equal to \textit{starting\_value + machine\_epsilon}, so we just need to subtract the starting value (using floating point arithmetic) to measure, what difference one bit reflects in the single precision (\textit{float}).

The \textit{union} serves here as a way to access IEEE 754 number as a regular integer. Adding 1 to it in fact adds 1 to the \textit{fraction} part of the number, however, needless to say, overflow is possible, which will add another 1 to the exponent part.

\subsection{\textsc{x86}}

\begin{lstlisting}[language=C++]
#define tv130 8
#define _v$ 8
#define _start$ 8

_calculate_machine_epsilon PROC
    fld DWORD PTR _start$[esp-4]
    fst DWORD PTR _v$[esp-4] ; this instruction is redundant
    inc DWORD PTR _v$[esp-4]
    fsubr DWORD PTR _v$[esp-4]
    fstp DWORD PTR tv130[esp-4] ; \ this instruction pair is also redundant
    fld DWORD PTR tv130[esp-4] ; /
    ret 0
_calculate_machine_epsilon ENDP
\end{lstlisting}

The second FST instruction is redundant: there is no need to store the input value in the same place (the compiler decided to allocate the \textit{v} variable at the same point in the local stack as the input argument).

Then it is incremented with INC, as it is a normal integer variable. Then it is loaded into the FPU as a 32-bit IEEE 754 number, FSUBR does the rest of job and the resulting value is stored in ST0.

The last FSTP/FLD instruction pair is redundant, but the compiler didn’t optimize it out.
22.2.2 ARM64

Let's extend our example to 64-bit:

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

typedef union
{   uint64_t i;
    double d;
} uint_double;

double calculate_machine_epsilon(double start)
{
    uint_double v;
    v.d=start;
    v.i++;
    return v.d-start;
}

void main()
{
    printf("%g\n", calculate_machine_epsilon(1.0));
}
```

ARM64 has no instruction that can add a number to a FPU D-register, so the input value (that came in D0) is first copied into GPR, incremented, copied to FPU register D1, and then subtraction occurs.

**Listing 22.6: Optimizing GCC 4.9 ARM64**

<table>
<thead>
<tr>
<th>calculate_machine_epsilon:</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmov x0, d0                            ; load input value of double type</td>
</tr>
<tr>
<td>add x0, x0, 1                         ; X0++</td>
</tr>
<tr>
<td>fmov d1, x0                            ; move it to FPU register</td>
</tr>
<tr>
<td>fsub d0, d1, d0                        ; subtract</td>
</tr>
</tbody>
</table>

See also this example compiled for x64 with SIMD instructions: 27.4 on page 604.

22.2.3 MIPS

The new instruction here is MTC1 ("Move To Coprocessor 1"), it just transfers data from GPR to the FPU's registers.
Listing 22.7: Optimizing GCC 4.4.5 (IDA)

calculate_machine_epsilon:
  mfc1 $v0, $f12
  or $at, $zero ; NOP
  addiu $v1, $v0, 1
  mtc1 $v1, $f2
  jr $ra
  sub.s $f0, $f2, $f12 ; branch delay slot

22.2.4 Conclusion

It’s hard to say whether someone may need this trickery in real-world code, but as
was mentioned many times in this book, this example serves well for explaining
the IEEE 754 format and unions in C/C++.

22.3 Fast square root calculation

Another well-known algorithm where float is interpreted as integer is fast calcula-
tion of square root.

Listing 22.8: The source code is taken from Wikipedia: http://go.yurichev.com/17364

/* Assumes that float is in the IEEE 754 single precision floating point format
   */
/* and that int is 32 bits. */
float sqrt_approx(float z)
{
  int val_int = *(int*)&z; /* Same bits, but as an int */
  /*
   * To justify the following code, prove that
   * ((val_int / 2^m) - b) / 2) + b) * 2^m = ((val_int - 2^m) / 2) + ((b + 1) / 2) * 2^m)
   * where
   * b = exponent bias
   * m = number of mantissa bits
   */
As an exercise, you can try to compile this function and to understand, how it works.

There is also well-known algorithm of fast calculation of $\frac{1}{\sqrt{x}}$. Algorithm became popular, supposedly, because it was used in Quake III Arena.

A pointer to a function, as any other pointer, is just the address of the function’s start in its code segment.

They are often used for calling callback functions. Well-known examples are:

- `qsort()`\(^2\), `atexit()`\(^3\) from the standard C library;
- *NIX OS signals;\(^4\)
- thread starting: `CreateThread()` (win32), `pthread_create()` (POSIX);
- lots of win32 functions, like `EnumChildWindows()`\(^5\).
- lots of places in the Linux kernel, for example the filesystem driver functions are called via callbacks: [http://go.yurichev.com/17076](http://go.yurichev.com/17076)
- The GCC plugin functions are also called via callbacks: [http://go.yurichev.com/17077](http://go.yurichev.com/17077)
- Another example of function pointers is a table in the “dwm” Linux window manager that defines shortcuts. Each shortcut has a corresponding function to call if a specific key is pressed: [GitHub](https://github.com). As we can see, such table is easier to handle than a large switch() statement.

So, the `qsort()` function is an implementation of quicksort in the C/C++ standard library. The functions is able to sort anything, any type of data, as long as you have

\(^1\) [wikipedia](https://en.wikipedia.org)
\(^2\) [wikipedia](https://en.wikipedia.org)
\(^3\) [http://go.yurichev.com/17073](http://go.yurichev.com/17073)
\(^4\) [wikipedia](https://en.wikipedia.org)
\(^5\) [MSDN](https://msdn.microsoft.com)
a function to compare these two elements and `qsort()` is able to call it.

The comparison function can be defined as:

```c
int (*compare)(const void *, const void *)
```

Let’s use a slightly modified example which was found here:

```c
/* ex3 Sorting ints with qsort */
#include <stdio.h>
#include <stdlib.h>
int comp(const void * _a, const void * _b)
{
    const int *a=(const int *)_a;
    const int *b=(const int *)_b;
    if (*a==*b)
        return 0;
    else
        if (*a < *b)
            return -1;
        else
            return 1;
}
int main(int argc, char* argv[])
{
    int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
    int i;
    /* Sort the array */
    qsort(numbers,10,sizeof(int),comp); 
    for (i=0;i<9;i++)
        printf("Number = %d\n",numbers[ i ]);
    return 0;
}
```

### 23.1 MSVC

Let’s compile it in MSVC 2010 (some parts were omitted for the sake of brevity) with `/Ox` option:
Listing 23.1: Optimizing MSVC 2010: /GS- /MD

```assembly
__a$ = 8 ; size
 ViewBag = 4
__b$ = 12 ; size
 ViewBag = 4

__comp PROC
  mov eax, DWORD PTR __a$[esp-4]
  mov ecx, DWORD PTR __b$[esp-4]
  mov eax, DWORD PTR [eax]
  mov ecx, DWORD PTR [ecx]
  cmp eax, ecx
  jne SHORT $LN4@comp
  xor eax, eax
  ret 0
$LN4@comp:
  xor edx, edx
  cmp eax, ecx
  setge dl
  lea eax, DWORD PTR [edx+edx-1]
  ret 0
__comp ENDP

__numbers$ = -40 ; size
 ViewBag = 40
__argc$ = 8 ; size
 ViewBag = 4
__argv$ = 12 ; size
 ViewBag = 4

__main PROC
  sub esp, 40 ;
    00000028H
  push esi
  push OFFSET __comp
  push 4
  lea eax, DWORD PTR __numbers$[esp+52]
  push 10
    0000000aH
  push eax
  mov DWORD PTR __numbers$[esp+60], 1892 ;
    000000764H
  mov DWORD PTR __numbers$[esp+64], 45 ;
    0000002dH
  mov DWORD PTR __numbers$[esp+68], 200 ;
    000000c8H
  mov DWORD PTR __numbers$[esp+72], -98 ;
    ffffffff9eH
```

529
Nothing surprising so far. As a fourth argument, the address of label _comp is passed, which is just a place where comp() is located, or, in other words, the address of the very first instruction of that function.

How does qsort() call it?

Let’s take a look at this function, located in MSVCR80.DLL (a MSVC DLL module with C standard library functions):
comp— is the fourth function argument. Here the control gets passed to the address in the comp argument. Before it, two arguments are prepared for comp(). Its result is checked after its execution.

That’s why it is dangerous to use pointers to functions. First of all, if you call qsort() with an incorrect function pointer, qsort() may pass control flow to an incorrect point, the process may crash and this bug will be hard to find.

The second reason is that the callback function types must comply strictly, calling the wrong function with wrong arguments of wrong types may lead to serious problems, however, the crashing of the process is not a problem here — the problem is how to determine the reason for the crash — because the compiler may be silent about the potential problems while compiling.
23.1.1 MSVC + OllyDbg

Let’s load our example into OllyDbg and set a breakpoint on `comp()`.

We can see how the values are compared at the first `comp()` call:

![OllyDbg: first call of comp()](image)

OllyDbg shows the compared values in the window under the code window, for convenience. We can also see that the SP points to RA, where the `qsort()` function is (located in MSVCR100.DLL).
By tracing (F8) until the RETN instruction and pressing F8 one more time, we return to the `qsort()` function:

![Image](image.jpg)

Figure 23.2: OllyDbg: the code in `qsort()` right after `comp()` call

That was a call to the comparison function.
Here is also a screenshot of the moment of the second call of `comp()` — now values that have to be compared are different:

![OllyDbg: second call of comp()](image)

**Figure 23.3: OllyDbg: second call of comp()**

### 23.1.2 MSVC + tracer

Let's also see which pairs are compared. These 10 numbers are being sorted: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

We got the address of the first `CMP` instruction in `comp()`, it is 0x0040100C and we’ve set a breakpoint on it:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

Now we get some information about the registers at the breakpoint:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100C
EAX=0x000000764 EBX=0x0051f7c8 ECX=0x000000005 EDX=0x000000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xffffe7960 EDX=0x000000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x000000764 EBX=0x0051f7c8 ECX=0x000000005 EDX=0x000000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```
Let's filter out EAX and ECX and we got:

| EAX=0x00000764 | ECX=0x00000005 |
| EAX=0x00000005 | ECX=0xffffe7960 |
| EAX=0x00000764 | ECX=0x00000005 |
| EAX=0x00000002 | ECX=0x00000005 |
| EAX=0x00000058 | ECX=0x00000005 |
| EAX=0x0000043f | ECX=0x00000005 |
| EAX=0xffffcfc7 | ECX=0x00000005 |
| EAX=0x000000c8 | ECX=0x00000005 |
| EAX=0xffffff9e | ECX=0x00000005 |
| EAX=0x00000ff7 | ECX=0x00000005 |
| EAX=0xffffff9e | ECX=0x00000005 |
| EAX=0xffffcfc7 | ECX=0xfffe7960 |
| EAX=0x00000005 | ECX=0xffffcfc7 |
| EAX=0xffffff9e | ECX=0xffffcfc7 |
| EAX=0xffffcfc7 | ECX=0xfffe7960 |
| EAX=0x000000c8 | ECX=0x00000ff7 |
| EAX=0x00000ff7 | ECX=0x00000ff7 |
| EAX=0x0000058 | ECX=0x00000ff7 |
| EAX=0x00000764 | ECX=0x00000ff7 |
| EAX=0x000000c8 | ECX=0x00000058 |
| EAX=0x0000058 | ECX=0x00000058 |
| EAX=0x0000043f | ECX=0x00000058 |
| EAX=0x000000c8 | ECX=0x00000058 |
| EAX=0x0000002d | ECX=0x000000c8 |
| EAX=0x0000002d | ECX=0x00000058 |
| EAX=0x0000002d | ECX=0x000000c8 |
| EAX=0x0000002d | ECX=0x00000058 |

That's 34 pairs. Therefore, the quick sort algorithm needs 34 comparison operations to sort these 10 numbers.
23.1.3 MSVC + tracer (code coverage)

We can also use the tracer's feature to collect all possible register values and show them in IDA.

Let's trace all instructions in `comp()`:

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

We get an .idc-script for loading into IDA and load it:

![Figure 23.4: tracer and IDA. N.B.: some values are cut at right](image)

IDA gave the function a name (PtFuncCompare) — because IDA sees that the pointer to this function is passed to `qsort()`.

We see that the `a` and `b` pointers are pointing to various places in the array, but the step between them is 4, as 32-bit values are stored in the array.

We see that the instructions at 0x401010 and 0x401012 were never executed (so they left as white): indeed, `comp()` has never returned 0, because there are no equal elements in the array.

23.2 GCC

Not a big difference:

```
Listing 23.3: GCC
lea eax, [esp+40h+var_28]
```
The implementation of \texttt{qsort()} is located in \texttt{libc.so.6} and it is in fact just a wrapper\textsuperscript{6} for \texttt{qsort_r()}.  

\textsuperscript{6}a concept like \texttt{thunk function}
In turn, it is calling quicksort(), where our defined function is called via a passed pointer:

Listing 23.4: (file libc.so.6, glibc version—2.10.1)

```
.text:0002DDF6     mov    edx, [ebp+arg_10]
.text:0002DDF9     mov    [esp+4], esi
.text:0002DDFD     mov    [esp], edi
.text:0002DE00     mov    [esp+8], edx
.text:0002DE04     call   [ebp+arg_C]
...
```

23.2.1 GCC + GDB (with source code)

Obviously, we have the C-source code of our example (23 on page 528), so we can set a breakpoint (b) on line number (11—the line where the first comparison occurs). We also need to compile the example with debugging information included (-g), so the table with addresses and corresponding line numbers is present. We can also print values using variable names (p): the debugging information also has tells us which register and/or local stack element contains which variable.

We can also see the stack (bt) and find out that there is some intermediate function msort_with_tmp() used in Glibc.

Listing 23.5: GDB session

dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see: <http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out
```
23.2.2 GCC + GDB (no source code)

But often there is no source code at all, so we can disassemble the \texttt{comp()} function (\texttt{disas}), find the very first CMP instruction and set a breakpoint (\texttt{b}) at that address. At each breakpoint, we are going to dump all register contents (\texttt{info})
registers). The stack information is also available (bt), but partially: there is no line number information for comp().

Listing 23.6: GDB session

dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
  0x0804844d <+0>: push ebp
  0x0804844e <+1>: mov ebp,esp
  0x08048450 <+3>: sub esp,0x10
  0x08048453 <+6>: mov eax,DWORD PTR [ebp+0x8]
  0x08048456 <+9>: mov DWORD PTR [ebp-0x8],eax
  0x08048459 <+12>: mov eax,DWORD PTR [ebp+0xc]
  0x0804845c <+15>: mov DWORD PTR [ebp-0x4],eax
  0x0804845f <+18>: mov eax,DWORD PTR [ebp-0x8]
  0x08048462 <+21>: mov edx,DWORD PTR [eax]
  0x08048464 <+23>: mov eax,DWORD PTR [ebp-0x4]
  0x08048467 <+26>: mov eax,DWORD PTR [eax]
  0x08048469 <+28>: cmp edx,eax
  0x0804846b <+30>: jne 0x8048474 <comp+39>
  0x0804846d <+32>: mov eax,0x0
  0x08048472 <+37>: jmp 0x804848e <comp+65>
  0x08048474 <+39>: mov eax,DWORD PTR [ebp-0x8]
  0x08048477 <+42>: mov edx,DWORD PTR [eax]
  0x08048479 <+44>: mov eax,DWORD PTR [ebp-0x4]
  0x0804847c <+47>: mov eax,DWORD PTR [eax]
  0x0804847e <+49>: cmp edx,eax
  0x08048480 <+51>: jge 0x8048489 <comp+60>
  0x08048482 <+53>: mov eax,0xffffffff
  0x08048487 <+58>: jmp 0x804848e <comp+65>
  0x08048489 <+60>: mov eax,0x1
0x0804848e <+65>: .leave
0x0804848f <+66>:  ret

End of assembler dump.

(gdb) b *0x08048469
Breakpoint 1 at 0x08048469
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax 0x2d 45
ecx 0xbffff0f8 -1073745672
edx 0x764 1892
ebx 0xb7fc0000 -1208221696
esp 0xbfffeeb8 0xbfffeeb8
ebp 0xbfffeec8 0xbfffeec8
esi 0xbffff0fc -1073745668
edi 0xbffff010 -1073745904
eip 0x8048469 0x8048469 <comp+28>
eflags 0x286 [ PF SF IF ]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51

(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax 0xff7 4087
ecx 0xbffff104 -1073745660
edx 0xffffffff9e -98
ebx 0xb7fc0000 -1208221696
esp 0xbfffeeb58 0xbfffeeb58
ebp 0xbfffeeb68 0xbfffeeb68
esi 0xbffff010 -1073745904
eip 0x8048469 0x8048469 <comp+28>
eflags 0x282 [ SF IF ]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax 0xffffffff9e  -98
ecx 0xbfffff100  -1073745664
edx 0xc8          200
ebx 0xb7fc0000   -1208221696
esp 0xbfffeeb8    0xbfffeeb8
ebp 0xbfffeec8    0xbfffeec8
esi 0xbffff104    -1073745660
edi 0xbffff010    -1073745904
eip 0x8048469     0x8048469 <comp+28>
eflags 0x286 [ PF SF IF ]
cs 0x73           115
ss 0x7b           123
ds 0x7b           123
es 0x7b           123
fs 0x0            0
gs 0x33           51
(gdb) bt
#0 0x08048469 in comp ()
    1 0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=  
        b@entry=0xbffff0f8, n=n@entry=2) at msort.c:65
#1 0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0  
        xbffff07c) at msort.c:45
#2 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8,  
        n=n@entry=5) at msort.c:53
#3 0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0  
        xbffff07c) at msort.c:45
#4 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8,  
        n=n@entry=10) at msort.c:53
#5 0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0  
        xbffff07c) at msort.c:45
#6 __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry  
        =4, cmp=cmp@entry=0x804844d <comp>,  
        arg=arg@entry=0x0) at msort.c:297
#7 0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0  
        x804844d <comp>) at msort.c:307
#8 0x0804850d in main ()
Chapter 24

64-bit values in 32-bit environment

In a 32-bit environment, GPR’s are 32-bit, so 64-bit values are stored and passed as 32-bit value pairs \(^1\).

24.1 Returning of 64-bit value

```c
#include <stdint.h>

uint64_t f ()
{
    return 0x1234567890ABCDEF;
}
```

24.1.1 x86

In a 32-bit environment, 64-bit values are returned from functions in the EDX:EAX register pair.

Listing 24.1: Optimizing MSVC 2010

```assembly
_f PROC
    mov eax, -1867788817 ; 90abcdefH
_f ENDPROCE
```

\(^1\)By the way, 32-bit values are passed as pairs in 16-bit environment in the same way: 53.4 on page 843
24.1.2 ARM

A 64-bit value is returned in the R0-R1 register pair (R1 is for the high part and R0 for the low part):

Listing 24.2: Optimizing Keil 6/2013 (ARM mode)

```
| f PROC
  LDR r0, L0.12
  LDR r1, L0.16
  BX lr
ENDP

| L0.12 |
DCD 0x90abcdef

| L0.16 |
DCD 0x12345678
```

24.1.3 MIPS

A 64-bit value is returned in the V0-V1 ($2-$3) register pair (V0 ($2) is for the high part and V1 ($3) for the low part):

Listing 24.3: Optimizing GCC 4.4.5 (assembly listing)

```
li $3,-1867841536 # 0
li $2,305397760 # 0x12340000
ori $3,$3,0xcdef
j $31
ori $2,$2,0x5678
```

Listing 24.4: Optimizing GCC 4.4.5 (IDA)

```
lui $v1, 0x90AB
lui $v0, 0x1234
li $v1, 0x90ABCDEF
jr $ra
li $v0, 0x12345678
```
CHAPTER 24. 64-BIT VALUES IN 32-BIT ENVIRONMENT

24.2

CHAPTER 24. 64-BIT VALUES IN 32-BIT ENVIRONMENT

Arguments passing, addition, subtraction

#include <stdint.h>
uint64_t f_add (uint64_t a, uint64_t b)
{
return a+b;
};
void f_add_test ()
{
#ifdef __GNUC__
printf ("%lld\n", f_add(12345678901234, 23456789012345)⤦
Ç );
#else
printf ("%I64d\n", f_add(12345678901234, ⤦
Ç 23456789012345));
#endif
};
uint64_t f_sub (uint64_t a, uint64_t b)
{
return a-b;
};

24.2.1

x86
Listing 24.5: Optimizing MSVC 2012 /Ob1

_a$ = 8
_b$ = 16
_f_add PROC
mov
add
mov
adc
ret
_f_add ENDP

; size = 8
; size = 8
eax,
eax,
edx,
edx,
0

DWORD
DWORD
DWORD
DWORD

_f_add_test PROC
push
5461
push
1972608889
push
2874
push
1942892530
call
_f_add

PTR
PTR
PTR
PTR

_a$[esp-4]
_b$[esp-4]
_a$[esp]
_b$[esp]

;
;
;
;

545

00001555H
75939f79H
00000b3aH
73ce2ff_subH


We can see in the \texttt{f_add_test()} function that each 64-bit value is passed using two 32-bit values, high part first, then low part.

Addition and subtraction occur in pairs as well.

In addition, the low 32-bit part are added first. If carry was occurred while adding, the \texttt{CF} flag is set. The following \texttt{ADC} instruction adds the high parts of the values, and also adds 1 if \texttt{CF} = 1.

Subtraction also occurs in pairs. The first \texttt{SUB} may also turn on the \texttt{CF} flag, which is to be checked in the subsequent \texttt{SBB} instruction: if the carry flag is on, then 1 is also to be subtracted from the result.

It is easy to see how the \texttt{f_add()} function result is then passed to \texttt{printf()}.
CHAPTER 24. 64-BIT VALUES IN 32-BIT ENVIRONMENT

mov DWORD PTR [esp+4], eax
mov DWORD PTR [esp+8], edx
mov DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\12\0"
call _printf
add esp, 28
ret

_f_sub:
  mov eax, DWORD PTR [esp+4]
  mov edx, DWORD PTR [esp+8]
  sub eax, DWORD PTR [esp+12]
  sbb edx, DWORD PTR [esp+16]
  ret

GCC code is the same.

24.2.2 ARM

Listing 24.7: Optimizing Keil 6/2013 (ARM mode)

f_add PROC
  ADDS r0,r0,r2
  ADC r1,r1,r3
  BX lr
ENDP

f_sub PROC
  SUBS r0,r0,r2
  SBC r1,r1,r3
  BX lr
ENDP

f_add_test PROC
  PUSH {r4,lr}
  LDR r2,|L0.68| ; 0x75939f79
  LDR r3,|L0.72| ; 0x00001555
  LDR r0,|L0.76| ; 0x73ce2ff2
  LDR r1,|L0.80| ; 0x00000b3a
  BL f_add
  POP {r4,lr}
  MOV r2,r0
  MOV r3,r1
  ADR r0,|L0.84| ; "%I64d\n"
  B __2printf
ENDP
The first 64-bit value is passed in R0 and R1 register pair, the second in R2 and R3 register pair. ARM has the ADC instruction as well (which counts carry flag) and SBC ("subtract with carry").

Important thing: when the low parts are added/subtracted, ADDS and SUBS instructions with -S suffix are used. The -S suffix stands for “set flags”, and flags (esp. carry flag) is what consequent ADC/SBC instructions definitely need. Otherwise, instructions without the -S suffix would do the job (ADD and SUB).

### 24.2.3 MIPS

Listing 24.8: Optimizing GCC 4.4.5 (IDA)

```assembly
f_add:
; $a0 - high part of a
; $a1 - low part of a
; $a2 - high part of b
; $a3 - low part of b
    addu $v1, $a3, $a1 ; sum up low parts
    addu $a0, $a2, $a0 ; sum up high parts
; will carry generated while summing up low parts?
; if yes, set $v0 to 1
    sltu $v0, $v1, $a3
    jr $ra
; add 1 to high part of result if carry should be generated:
    addu $v0, $a0 ; branch delay slot
; $v0 - high part of result
; $v1 - low part of result
f_sub:
; $a0 - high part of a
; $a1 - low part of a
; $a2 - high part of b
; $a3 - low part of b
    subu $v1, $a1, $a3 ; subtract low parts
```
MIPS has no flags register, so there is no such information present after the execution of arithmetic operations. So there are no instructions like x86’s ADC and SBB. To know if the carry flag would be set, a comparison (using “SLTU” instruction) also occurs, which sets the destination register to 1 or 0. This 1 or 0 is then added or subtracted to/from the final result.
24.3 Multiplication, division

```c
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b) {
    return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b) {
    return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b) {
    return a % b;
};
```

24.3.1 x86

Listing 24.9: Optimizing MSVC 2013 /Ob1

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     edx, DWORD PTR _a$[ebp]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __allmul ; long long multiplication
    pop     ebp
    ret 0
_f_mul ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_div PROC
    push    ebp
```
Multiplication and division are more complex operations, so usually the compiler embeds calls to a library functions doing that.

These functions are described here: E on page 1330.

Listing 24.10: Optimizing GCC 4.8.1 -fno-inline
add    ecx, ebx
add    edx, ecx
pop    ebx
ret

__f_div:
sub    esp, 28
mov    eax, DWORD PTR [esp+40]
mov    edx, DWORD PTR [esp+44]
mov    DWORD PTR [esp+8], eax
mov    eax, DWORD PTR [esp+32], edx
mov    edx, DWORD PTR [esp+36]
mov    DWORD PTR [esp], eax
mov    DWORD PTR [esp+4], edx
call   ___udivdi3 ; unsigned division
add    esp, 28
ret

__f_rem:
sub    esp, 28
mov    eax, DWORD PTR [esp+40]
mov    edx, DWORD PTR [esp+44]
mov    DWORD PTR [esp+8], eax
mov    eax, DWORD PTR [esp+32], edx
mov    edx, DWORD PTR [esp+36]
mov    DWORD PTR [esp], eax
mov    DWORD PTR [esp+4], edx
call   ___umoddi3 ; unsigned modulo
add    esp, 28
ret

GCC does the expected, but the multiplication code is inlined right in the function, thinking it could be more efficient. GCC has different library function names: D on page 1329.

24.3.2 ARM

Keil for Thumb mode inserts library subroutine calls:

Listing 24.11: Optimizing Keil 6/2013 (Thumb mode)

||f_mul|| PROC
PUSH    {r4,lr}
BL      ___aeabi_lmul
Keil for ARM mode, on the other hand, is able to produce 64-bit multiplication code:

Listing 24.12: Optimizing Keil 6/2013 (ARM mode)

```assembly
||f_mul|| PROC
PUSH {r4,lr}
UMULL r12,r4,r0,r2
MLA r1,r2,r1,r4
MLA r1,r0,r3,r1
MOV r0,r12
POP {r4,pc}
ENDP

||f_div|| PROC
PUSH {r4,lr}
BL __aeabi_uldivmod
POP {r4,pc}
ENDP

||f_rem|| PROC
PUSH {r4,lr}
BL __aeabi_uldivmod
MOV r0,r2
MOV r1,r3
POP {r4,pc}
ENDP
```
Optimizing GCC for MIPS can generate 64-bit multiplication code, but has to call a library routine for 64-bit division:

Listing 24.13: Optimizing GCC 4.4.5 (IDA)

```assembly
f_mul:
mult    $a2, $a1
mflo    $v0
or      $at, $zero ; NOP
or      $at, $zero ; NOP
mult    $a0, $a3
mflo    $a0
addu    $v0, $a0
or      $at, $zero ; NOP
multu   $a3, $a1
mfhi    $a2
mflo    $v1
jr       $ra
addu    $v0, $a2

f_div:
var_10  = -0x10
var_4   = -4

lui     $gp, (___gnu_local_gp >> 16)
addiu   $sp, -0x20
la      $gp, (___gnu_local_gp & 0xFFFF)
sw      $ra, 0x20+var_4($sp)
sw      $gp, 0x20+var_10($sp)
lw      $t9, (___udivdi3 & 0xFFFF)($gp)
or      $at, $zero
jalr    $t9
or      $at, $zero
lw      $ra, 0x20+var_4($sp)
or      $at, $zero
jr       $ra
addiu   $sp, 0x20

f_rem:
var_10  = -0x10
var_4   = -4

lui     $gp, (___gnu_local_gp >> 16)
addiu   $sp, -0x20
```

554
There are a lot of NOPs, probably delay slots filled after the multiplication instruction (it's slower than other instructions, after all).

### 24.4 Shifting right

```c
#include <stdint.h>

uint64_t f (uint64_t a) {
    return a>>7;
};
```

#### 24.4.1 x86

Listing 24.14: Optimizing MSVC 2012 /Ob1

```assembly
_a$ = 8 ; size = 8
_f PROC
    mov eax, DWORD PTR _a$[esp-4]
    mov edx, DWORD PTR _a$[esp]
    shrd eax, edx, 7
    shr edx, 7
    ret 0
_f ENDP
```

Listing 24.15: Optimizing GCC 4.8.1 -fno-inline

```assembly
_f:
    mov edx, DWORD PTR [esp+8]
    mov eax, DWORD PTR [esp+4]
    shrd eax, edx, 7
```
Shifting also occurs in two passes: first the lower part is shifted, then the higher part. But the lower part is shifted with the help of the SHRD instruction, it shifts the value of EDX by 7 bits, but pulls new bits from EAX, i.e., from the higher part. The higher part is shifted using the more popular SHR instruction: indeed, the freed bits in the higher part must be filled with zeroes.

### 24.4.2 ARM

ARM doesn’t have such instruction as SHRD in x86, so the Keil compiler ought to do this using simple shifts and OR operations:

**Listing 24.16: Optimizing Keil 6/2013 (ARM mode)**

<table>
<thead>
<tr>
<th>PROC</th>
<th>R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSR</td>
<td>R0</td>
</tr>
<tr>
<td>ORR</td>
<td>R0</td>
</tr>
<tr>
<td>LSR</td>
<td>R1</td>
</tr>
<tr>
<td>BX</td>
<td>R1</td>
</tr>
</tbody>
</table>

**Listing 24.17: Optimizing Keil 6/2013 (Thumb mode)**

<table>
<thead>
<tr>
<th>PROC</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSLS</td>
<td>R2</td>
</tr>
<tr>
<td>LSRS</td>
<td>R0</td>
</tr>
<tr>
<td>ORRS</td>
<td>R0</td>
</tr>
<tr>
<td>LSRS</td>
<td>R1</td>
</tr>
</tbody>
</table>

### 24.4.3 MIPS

GCC for MIPS follows the same algorithm as Keil does for Thumb mode:

**Listing 24.18: Optimizing GCC 4.4.5 (IDA)**

```
f:
  sll $v0, $a0, 25
  srl $v1, $a1,  7
  or  $v1, $v0, $v1
  jr   $ra
  srl $v0, $a0,  7
```
24.5 Converting 32-bit value into 64-bit one

```c
#include <stdint.h>

int64_t f (int32_t a)
{
    return a;
}
```

### 24.5.1 x86

Listing 24.19: Optimizing MSVC 2012

```asm
 faker PROC
    mov      eax, DWORD PTR _a@[esp-4]
    cdq
    ret 0
 faker ENDP
```

Here we also run into necessity to extend a 32-bit signed value into a 64-bit signed one. Unsigned values are converted straightforwardly: all bits in the higher part must be set to 0. But this is not appropriate for signed data types: the sign has to be copied into the higher part of the resulting number. The CDQ instruction does that here, it takes its input value in EAX, extends it to 64-bit and leaves it in the EDX:EAX register pair. In other words, CDQ gets the number sign from EAX (by getting the most significant bit in EAX), and depending of it, sets all 32 bits in EDX to 0 or 1. Its operation is somewhat similar to the MOVsx instruction.

### 24.5.2 ARM

Listing 24.20: Optimizing Keil 6/2013 (ARM mode)

```asm
<table>
<thead>
<tr>
<th>f</th>
</tr>
</thead>
</table>
| PROC
    ASR r1,r0,#31
    BX lr
 ENDP
```

Keil for ARM is different: it just arithmetically shifts right the input value by 31 bits. As we know, the sign bit is MSB, and the arithmetical shift copies the sign bit into the “emerged” bits. So after “ASR r1,r0,#31”, R1 containing 0xFFFFFFFF if
the input value was negative and 0 otherwise. R1 contains the high part of the resulting 64-bit value.

In other words, this code just copies the MSB (sign bit) from the input value in R0 to all bits of the high 32-bit part of the resulting 64-bit value.

### 24.5.3 MIPS

GCC for MIPS does the same as Keil did for ARM mode:

Listing 24.21: Optimizing GCC 4.4.5 (IDA)

```
f:
sra $v0, $a0, 31
jr $ra
move $v1, $a0
```
Chapter 25

SIMD

SIMD is an acronym: *Single Instruction, Multiple Data*.

As its name implies, it processes multiple data using only one instruction.

Like the FPU, that CPU subsystem looks like a separate processor inside x86.

SIMD began as MMX in x86. 8 new 64-bit registers appeared: MM0-MM7.

Each MMX register can hold 2 32-bit values, 4 16-bit values or 8 bytes. For example, it is possible to add 8 8-bit values (bytes) simultaneously by adding two values in MMX registers.

One simple example is a graphics editor that represents an image as a two dimensional array. When the user changes the brightness of the image, the editor must add or subtract a coefficient to/from each pixel value. For the sake of brevity if we say that the image is grayscale and each pixel is defined by one 8-bit byte, then it is possible to change the brightness of 8 pixels simultaneously. By the way, this is the reason why the *saturation* instructions are present in SIMD. When the user changes the brightness in the graphics editor, overflow and underflow are not desirable, so there are addition instructions in SIMD which are not adding anything if the maximum value is reached, etc.

When MMX appeared, these registers were actually located in the FPU’s registers. It was possible to use either FPU or MMX at the same time. One might think that Intel saved on transistors, but in fact the reason of such symbiosis was simpler — older OSes that are not aware of the additional CPU registers would not save them at the context switch, but saving the FPU registers. Thus, MMX-enabled CPU + old OS + process utilizing MMX features will still work.

SSE—is extension of the SIMD registers to 128 bits, now separate from the FPU.

AVX—another extension, to 256 bits.
Now about practical usage.

Of course, this is memory copy routines (memcpy), memory comparing (memcmp) and so on.

One more example: the DES encryption algorithm takes a 64-bit block and a 56-bit key, encrypt the block and produces a 64-bit result. The DES algorithm may be considered as a very large electronic circuit, with wires and AND/OR/NOT gates.

Bitslice DES\(^1\) — is the idea of processing groups of blocks and keys simultaneously. Let’s say, variable of type \texttt{unsigned int} on x86 can hold up to 32 bits, so it is possible to store there intermediate results for 32 block-key pairs simultaneously, using 64+56 variables of type \texttt{unsigned int}.

There is an utility to brute-force Oracle RDBMS passwords/ hashes (ones based on DES), using slightly modified bitslice DES algorithm for SSE2 and AVX—now it is possible to encrypt 128 or 256 block-keys pairs simultaneously.

http://go.yurichev.com/17313

\section{25.1 Vectorization}

Vectorization\(^2\) is when, for example, you have a loop taking couple of arrays for input and producing one array. The loop body takes values from the input arrays, does something and puts the result into the output array. Vectorization is to process several elements simultaneously.

Vectorization is not very fresh technology: the author of this textbook saw it at least on the Cray Y-MP supercomputer line from 1988 when he played with its “lite” version Cray Y-MP EL\(^3\).

For example:

```c
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

This fragment of code takes elements from A and B, multiplies them and saves the result into C.

If each array element we have is 32-bit \texttt{int}, then it is possible to load 4 elements from A into a 128-bit XMM-register, from B to another XMM-registers, and by executing \texttt{PMULLD} (Multiply Packed Signed Dword Integers and Store Low Result) and

---

\(^1\)http://go.yurichev.com/17329
\(^2\)Wikipedia: vectorization
\(^3\)Remotely. It is installed in the museum of supercomputers: http://go.yurichev.com/17081
PMULHW (Multiply Packed Signed Integers and Store High Result), it is possible to get 4 64-bit products at once.

Thus, loop body execution count is $1024/4$ instead of 1024, that is 4 times less and, of course, faster.

### 25.1.1 Addition example

Some compilers can do vectorization automatically in simple cases, e.g., Intel C++\(^4\).

Here is tiny function:

```c
int f (int sz, int *ar1, int *ar2, int *ar3) {
    for (int i=0; i<sz; i++)
        ar3[i]=ar1[i]+ar2[i];
    return 0;
}
```

**Intel C++**

Let's compile it with Intel C++ 11.1.051 win32:
```
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

We got (in IDA):

```assembly
; int __cdecl f(int, int *, int *, int *)
    public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz     = dword ptr  4
ar1    = dword ptr  8
ar2    = dword ptr 0Ch
ar3    = dword ptr 10h

    push    edi
    push    esi
    push    ebx
    push    esi
    mov     edx, [esp+10h+sz]
    test    edx, edx
    jle     loc_15B
```

\(^4\)More about Intel C++ automatic vectorization: Excerpt: Effective Automatic Vectorization
mov  eax, [esp+10h+ar3]
cmp  edx, 6
jle  loc_143
cmp  eax, [esp+10h+ar2]
  jbe  short loc_36
mov  esi, [esp+10h+ar2]
sub  esi, eax
lea  ecx, ds:0[edx*4]
eg  esi
cmp  ecx, esi
loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
  cmp  eax, [esp+10h+ar2]
  jnb  loc_143
  mov  esi, [esp+10h+ar2]
  sub  esi, eax
  lea  ecx, ds:0[edx*4]
  cmp  esi, ecx
  jb  loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
  cmp  eax, [esp+10h+ar1]
     jbe  short loc_67
mov  esi, [esp+10h+ar1]
sub  esi, eax
neg  esi
cmp  ecx, esi
loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
  cmp  eax, [esp+10h+ar1]
  jnb  loc_143
  mov  esi, [esp+10h+ar1]
  sub  esi, eax
  cmp  ecx, esi
  jb  loc_143

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
mov  edi, eax ; edi = ar1
and  edi, 0Fh ; is ar1 16-byte aligned?
jz  short loc_9A ; yes
test  edi, 3
jnz  loc_162
neg  edi
add  edi, 10h
shr  edi, 2
loc_9A: ; CODE XREF: f(int, int *, int *, int *)+84
  lea ecx, [edi+4]
  cmp edx, ecx
  jl loc_162
  mov ecx, edx
  sub ecx, edi
  and ecx, 3
  neg ecx
  add ecx, edx
  test edi, edi
  jbe short loc_D6
  mov ebx, [esp+10h+ar2]
  mov [esp+10h+var_10], ecx
  mov ecx, [esp+10h+ar1]
  xor esi, esi

loc_C1: ; CODE XREF: f(int, int *, int *, int *)+CD
  mov edx, [ecx+esi*4]
  add edx, [ebx+esi*4]
  mov [eax+esi*4], edx
  inc esi
  cmp esi, edi
  jb short loc_C1
  mov ecx, [esp+10h+var_10]
  mov edx, [esp+10h+sz]
  mov esi, [esp+10h+ar2]
  lea esi, [esi+edi*4] ; is ar2+i*4 16-byte aligned?
  test esi, 0Fh
  jz short loc_109 ; yes!
  mov ebx, [esp+10h+ar1]
  mov esi, [esp+10h+ar2]

loc_D6: ; CODE XREF: f(int, int *, int *, int *)+B2
  mov esi, [esp+10h+ar2]
  lea esi, [esi+edi*4] ; is ar2+i*4 16-byte aligned?
  test esi, 0Fh
  jz short loc_109 ; yes!
  mov ebx, [esp+10h+ar1]
  mov esi, [esp+10h+ar2]
  movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
  movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not
  16-byte aligned, so load it to XMM0
  padd xmm1, xmm0
  movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
  add edi, 4
  cmp edi, ecx
  jb short loc_ED
  jmp short loc_127

loc_109: ; CODE XREF: f(int, int *, int *, int *)+E3
  mov ebx, [esp+10h+ar1]
mov esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
    movdqu xmm0, xmmword ptr [ebx+edi*4]
    padd xmm0, xmmword ptr [esi+edi*4]
    movdqa xmmword ptr [eax+edi*4], xmm0
    add edi, 4
    cmp edi, ecx
    jb short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
    cmp ecx, edx
    jnb short loc_15B
    mov esi, [esp+10h+ar1]
    mov edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
    mov ebx, [esi+ecx*4]
    add ebx, [edi+ecx*4]
    mov [eax+ecx*4], ebx
    inc ecx
    cmp ecx, edx
    jb short loc_133
    jmp short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
    mov esi, [esp+10h+ar1]
    mov edi, [esp+10h+ar2]
    xor ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
    mov ebx, [esi+ecx*4]
    add ebx, [edi+ecx*4]
    mov [eax+ecx*4], ebx
    inc ecx
    cmp ecx, edx
    jb short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+1A
    xor eax, eax
    pop ecx
    pop ebx
    pop esi
    pop edi
The SSE2-related instructions are:

- **MOVDQU (Move Unaligned Double Quadword)**— just loads 16 bytes from memory into a XMM-register.

- **PADDD (Add Packed Integers)**— adds 4 pairs of 32-bit numbers and leaves the result in the first operand. By the way, no exception is raised in case of overflow and no flags are to be set, just the low 32 bits of the result are to be stored. If one of PADDD's operands is the address of a value in memory, then the address must be aligned on a 16-byte boundary. If it is not aligned, an exception will be triggered.

- **MOVDQA (Move Aligned Double Quadword)** is the same as MOVDQU, but requires the address of the value in memory to be aligned on a 16-bit boundary. If it is not aligned, exception will be raised. MOVDQA works faster than MOVDQU, but requires aforesaid.

So, these SSE2-instructions are to be executed only in case there are more than 4 pairs to work on and the pointer ar3 is aligned on a 16-byte boundary.

Also, if ar2 is aligned on a 16-byte boundary as well, this fragment of code is to be executed:

```
movdqu xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddd xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

Otherwise, the value from ar2 is to be loaded into XMM0 using MOVDQU, which does not require aligned pointer, but may work slower:

```
movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte aligned, so load it to XMM0
paddd xmm1, xmm0
movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

In all other cases, non-SSE2 code is to be executed.

---

5More about data alignment: [Wikipedia: Data structure alignment](https://en.wikipedia.org/wiki/Data_structure_alignment)
GCC may also vectorize in simple cases, if the -O3 option is used and SSE2 support is turned on: -msse2.

What we get (GCC 4.4.1):

```assembly
; f(int, int *, int *, int *)
public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18 = dword ptr -18h
var_14 = dword ptr -14h
var_10 = dword ptr -10h
arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h
arg_C = dword ptr 14h

push ebp
mov ebp, esp
push edi
push esi
push ebx
sub esp, 0Ch
mov ecx, [ebp+arg_0]
mov esi, [ebp+arg_4]
mov edi, [ebp+arg_8]
mov ebx, [ebp+arg_C]
test ecx, ecx
jle short loc_80484D8
cmp ecx, 6
lea eax, [ebx+10h]
ja short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
; f(int,int *,int *,int *)+61 ...
xor eax, eax
nop
lea esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
mov edx, [edi+eax*4]
add edx, [esi+eax*4]
mov [ebx+eax*4], edx
add eax, 1
```

6 More about GCC vectorization support: [http://go.yurichev.com/17083](http://go.yurichev.com/17083)
cmp    eax, ecx
jnz    short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
            ; f(int,int *,int *,int *)+A5
    add    esp, 0Ch
    xor    eax, eax
    pop    ebx
    pop    esi
    pop    edi
    pop    ebp
    retn

align 8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
    test   bl, 0Fh
    jnz    short loc_80484C1
    lea    edx, [esi+10h]
    cmp    ebx, edx
    jbe    loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
    lea    edx, [edi+10h]
    cmp    ebx, edx
    ja     short loc_8048503
    cmp    edi, eax
    jbe    short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
    mov    eax, ecx
    shr    eax, 2
    mov    [ebp+var_14], eax
    shl    eax, 2
    test   eax, eax
    mov    [ebp+var_10], eax
    jz     short loc_8048547
    mov    [ebp+var_18], ecx
    mov    ecx, [ebp+var_14]
    xor    eax, eax
    xor    edx, edx
    nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
    movdqu  xmm1, xmmword ptr [edi+eax]
    movdqu  xmm0, xmmword ptr [esi+eax]
    add     edx, 1
Almost the same, however, not as meticulously as Intel C++.

25.1.2 Memory copy example

Let's revisit the simple memcpy() example (14.2 on page 266):
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
};

And that's what optimizations GCC 4.9.1 did:

Listing 25.1: Optimizing GCC 4.9.1 x64

my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block
    test    rdx, rdx
    je      .L41
    lea     rax, [rdi+16]
    cmp     rsi, rax
    lea     rax, [rsi+16]
    setae   cl
    cmp     rdi, rax
    setae   al
    or      cl, al
    je      .L13
    cmp     rdx, 22
    jbe     .L13
    mov     rcx, rsi
    push    rbp
    push    rbx
    neg     rcx
    and     ecx, 15
    cmp     rcx, rdx
    cmova   rcx, rdx
    xor     eax, eax
    test    rcx, rcx
    je      .L4
    movzx   eax, BYTE PTR [rsi]
    cmp     rcx, 1
    mov     BYTE PTR [rdi], al
    je      .L15
    movzx   eax, BYTE PTR [rsi+1]
    cmp     rcx, 2
    mov     BYTE PTR [rdi+1], al
je .L16
movzx eax, BYTE PTR [rsi+2]
cmp rcx, 3
mov BYTE PTR [rdi+2], al
je .L17
movzx eax, BYTE PTR [rsi+3]
cmp rcx, 4
mov BYTE PTR [rdi+3], al
je .L18
movzx eax, BYTE PTR [rsi+4]
cmp rcx, 5
mov BYTE PTR [rdi+4], al
je .L19
movzx eax, BYTE PTR [rsi+5]
cmp rcx, 6
mov BYTE PTR [rdi+5], al
je .L20
movzx eax, BYTE PTR [rsi+6]
cmp rcx, 7
mov BYTE PTR [rdi+6], al
je .L21
movzx eax, BYTE PTR [rsi+7]
cmp rcx, 8
mov BYTE PTR [rdi+7], al
je .L22
movzx eax, BYTE PTR [rsi+8]
cmp rcx, 9
mov BYTE PTR [rdi+8], al
je .L23
movzx eax, BYTE PTR [rsi+9]
cmp rcx, 10
mov BYTE PTR [rdi+9], al
je .L24
movzx eax, BYTE PTR [rsi+10]
cmp rcx, 11
mov BYTE PTR [rdi+10], al
je .L25
movzx eax, BYTE PTR [rsi+11]
cmp rcx, 12
mov BYTE PTR [rdi+11], al
je .L26
movzx eax, BYTE PTR [rsi+12]
cmp rcx, 13
mov BYTE PTR [rdi+12], al
je .L27
movzx eax, BYTE PTR [rsi+13]
cmp rcx, 15
```assembly
mov  BYTE PTR [rdi+13], al
jne  .L28
movzx eax, BYTE PTR [rsi+14]
mov  BYTE PTR [rdi+14], al
mov  eax, 15
.L4:
    mov  r10, rdx
    lea  r9, [rdx-1]
    sub  r10, rcx
    lea  r8, [r10-16]
    sub  r9, rcx
    shr  r8, 4
    add  r8, 1
    mov  r11, r8
    sal  r11, 4
    cmp  r9, 14
    jbe  .L6
    lea  rbp, [rsi+rcx]
    xor  r9d, r9d
    add  rcx, rdi
    xor  ebx, ebx
.L7:
    movdqa xmm0, XMMWORD PTR [rbp+0+r9]
    add  rbx, 1
    movups XMMWORD PTR [rcx+r9], xmm0
    add  r9, 16
    cmp  rbx, r8
    jb   .L7
    add  rax, r11
    cmp  r10, r11
    je   .L1
.L6:
    movzx ecx, BYTE PTR [rsi+rax]
    mov  BYTE PTR [rdi+rax], cl
    lea  rcx, [rax+1]
    cmp  rdx, rcx
    jbe  .L1
    movzx ecx, BYTE PTR [rsi+1+rax]
    mov  BYTE PTR [rdi+1+rax], cl
    lea  rcx, [rax+2]
    cmp  rdx, rcx
    jbe  .L1
    movzx ecx, BYTE PTR [rsi+2+rax]
    mov  BYTE PTR [rdi+2+rax], cl
    lea  rcx, [rax+3]
    cmp  rdx, rcx
    jbe  .L1
```

571
movzx  ecx, BYTE PTR [rsi+3+rax]
omv     BYTE PTR [rdi+3+rax], cl
lea     rcx, [rax+4]
cmp     rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+4+rax]
omv     BYTE PTR [rdi+4+rax], cl
lea     rcx, [rax+5]
cmp     rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+5+rax]
omv     BYTE PTR [rdi+5+rax], cl
lea     rcx, [rax+6]
cmp     rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+6+rax]
omv     BYTE PTR [rdi+6+rax], cl
lea     rcx, [rax+7]
cmp     rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+7+rax]
omv     BYTE PTR [rdi+7+rax], cl
lea     rcx, [rax+8]
cmp     rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+8+rax]
omv     BYTE PTR [rdi+8+rax], cl
lea     rcx, [rax+9]
cmp     rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+9+rax]
omv     BYTE PTR [rdi+9+rax], cl
lea     rcx, [rax+10]
cmp     rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+10+rax]
omv     BYTE PTR [rdi+10+rax], cl
lea     rcx, [rax+11]
cmp     rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+11+rax]
omv     BYTE PTR [rdi+11+rax], cl
lea     rcx, [rax+12]
cmp     rdx, rcx
jbe     .L1
movzx  ecx, BYTE PTR [rsi+12+rax]
omv     BYTE PTR [rdi+12+rax], cl
lea  rcx, [rax+13]
cmp  rdx, rcx
jbe  .L1
movzx ecx, BYTE PTR [rsi+13+rax]
mov  BYTE PTR [rdi+13+rax], cl
lea  rcx, [rax+14]
cmp  rdx, rcx
jbe  .L1
movzx edx, BYTE PTR [rsi+14+rax]
mov  BYTE PTR [rdi+14+rax], dl
.L1:
    pop  rbx
    pop  rbp
.L41:
    rep ret
.L13:
    xor  eax, eax
.L3:
movzx ecx, BYTE PTR [rsi+rax]
mov  BYTE PTR [rdi+rax], cl
add  rax, 1
cmp  rax, rdx
jne  .L3
    rep ret
.L28:
mov  eax, 14
    jmp  .L4
.L15:
mov  eax, 1
    jmp  .L4
.L16:
mov  eax, 2
    jmp  .L4
.L17:
mov  eax, 3
    jmp  .L4
.L18:
mov  eax, 4
    jmp  .L4
.L19:
mov  eax, 5
    jmp  .L4
.L20:
mov  eax, 6
    jmp  .L4
.L21:
mov  eax, 7
25.2 SIMD strlen() implementation

It has to be noted that the SIMD instructions can be inserted in C/C++ code via special macros\(^7\). For MSVC, some of them are located in the intrin.h file.

It is possible to implement the strlen() function\(^8\) using SIMD instructions that works 2-2.5 times faster than the common implementation. This function loads 16 characters into a XMM-register and check each against zero\(^9\).

```c
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=((unsigned int)str)&0xFFFFFFF0) == (unsigned int)str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
```

\(^7\) MSDN: MMX, SSE, and SSE2 Intrinsics
\(^8\) strlen() -- standard C library function for calculating string length
\(^9\) The example is based on source code from: [http://go.yurichev.com/17330](http://go.yurichev.com/17330).
int mask = 0;

for (;;) {
    xmm1 = _mm_load_si128((__m128i *)s);
    xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
    if ((mask = _mm_movemask_epi8(xmm1)) != 0) {
        unsigned long pos;
        _BitScanForward(&pos, mask);
        len += (size_t)pos;
        break;
    }
    s += sizeof(__m128i);
    len += sizeof(__m128i);
}

return len;

Let’s compile it in MSVC 2010 with /Ox option:

Listing 25.2: Optimizing MSVC 2010

    _pos$75552 = -4 ; size = 4
    _str$ = 8 ; size = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

        push  ebp
        mov   ebp, esp
        and   esp, -16 ; fffffff0H
        mov   eax, DWORD PTR _str$[ebp]
        sub   esp, 12 ; 0000000cH
        push  esi
        mov   esi, eax
        and   esi, -16 ; fffffff0H
        xor   edx, edx
        mov   ecx, eax
cmp   esi, eax
je    SHORT $LN4@strlen_sse
lea   edx, DWORD PTR [eax+1]
npad  3 ; align next label
$LL11@strlen_sse:
    mov   cl, BYTE PTR [eax]
    inc   eax
test  cl, cl
jne   SHORT $LL11@strlen_sse
First, we check if the `str` pointer is aligned on a 16-byte boundary. If not, we call the generic `strlen()` implementation.

Then, we load the next 16 bytes into the `XMM1` register using `MOVDQA`.

An observant reader might ask, why can’t `MOVDQU` be used here since it can load data from the memory regardless pointer alignment?

Yes, it might be done in this way: if the pointer is aligned, load data using `MOVDQA`, if not — use the slower `MOVDQU`.

But here we are may hit another caveat:

In the Windows NT line of OS (but not limited to it), memory is allocated by pages of 4 KiB (4096 bytes). Each win32-process has 4 GiB available, but in fact, only some parts of the address space are connected to real physical memory. If the process is accessing an absent memory block, an exception is to be raised. That’s how VM
So, a function loading 16 bytes at once may step over the border of an allocated memory block. Let's say that the **OS** has allocated 8192 (0x2000) bytes at address 0x008c0000. Thus, the block is the bytes starting from address 0x008c0000 to 0x008c1fff inclusive.

After the block, that is, starting from address 0x008c2000 there is nothing at all, e.g. the **OS** not allocated any memory there. Any attempt to access memory starting from that address will raise an exception.

And let's consider the example in which the program is holding a string that contains 5 characters almost at the end of a block, and that is not a crime.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x008c1ff8</td>
<td>'h'</td>
</tr>
<tr>
<td>0x008c1ff9</td>
<td>'e'</td>
</tr>
<tr>
<td>0x008c1ffa</td>
<td>'l'</td>
</tr>
<tr>
<td>0x008c1ffb</td>
<td>'l'</td>
</tr>
<tr>
<td>0x008c1ffc</td>
<td>'o'</td>
</tr>
<tr>
<td>0x008c1ffd</td>
<td>'\x00'</td>
</tr>
<tr>
<td>0x008c1ffe</td>
<td>random noise</td>
</tr>
<tr>
<td>0x008c1fff</td>
<td>random noise</td>
</tr>
</tbody>
</table>

So, in normal conditions the program calls `strlen()`, passing it a pointer to the string 'hello' placed in memory at address 0x008c1ff8. `strlen()` reads one byte at a time until 0x008c1ffd, where there's a zero byte, and then it stops.

Now if we implement our own `strlen()` reading 16 byte at once, starting at any address, aligned or not, MOVQDQU may attempt to load 16 bytes at once at address 0x008c1ff8 up to 0x008c2008, and then an exception will be raised. That situation is to be avoided, of course.

So then we'll work only with the addresses aligned on a 16 byte boundary, which in combination with the knowledge that the **OS**' page size is usually aligned on a 16-byte boundary gives us some warranty that our function will not read from unallocated memory.

Let's get back to our function.

`_mm_setzero_si128()`— is a macro generating `pxor xmm0, xmm0`—it just clears the XMM0 register.

`_mm_load_si128()`— is a macro for MOVQDQA, it just loads 16 bytes from the address into the XMM1 register.

`_mm_cmpeq_epi8()`— is a macro for PCMPEQB, an instruction that compares two XMM-registers bytewise.

---

<sup>10</sup> wikipedia
And if some byte was equals to the one in the other register, there will be \texttt{0xff} at this point in the result or \texttt{0} if otherwise.

For example.

XMM1: 11223344556677880000000000000000
XMM0: 11ab3440078778811111111111111111

After the execution of \texttt{pcmpeqb xmm1, xmm0}, the XMM1 register contains:

XMM1: ff0000ff0000ffff0000000000000000

In our case, this instruction compares each 16-byte block with a block of 16 zero-bytes, which was set in the XMM0 register by \texttt{pxor xmm0, xmm0}.

The next macro is \_\_\_mm\_movemask\_\_epi8() —that is the \texttt{PMOVMSKB} instruction.

It is very useful with \texttt{PCMPEQB}.

\texttt{pmovmskb eax, xmm1}

This instruction sets first EAX bit to 1 if the most significant bit of the first byte in XMM1 is 1. In other words, if the first byte of the XMM1 register is \texttt{0xff}, then the first bit of EAX is to be 1, too.

If the second byte in the XMM1 register is \texttt{0xff}, then the second bit in EAX is to be set to 1. In other words, the instruction is answering the question “which bytes in XMM1 are \texttt{0xff}?" and returns 16 bits in the EAX register. The other bits in the EAX register are to be cleared.

By the way, do not forget about this quirk of our algorithm. There might be 16 bytes in the input like:

```
 15 14 13 12 11 10 9  3  2  1  0
```

| “h” | “e” | “l” | “l” | “o” | 0 | garbage | 0 | garbage |

It is the \texttt{‘hello’} string, terminating zero, and some random noise in memory. If we load these 16 bytes into XMM1 and compare them with the zeroed XMM0, we are getting something like \textsuperscript{11}:

XMM1: 0000ff00000000000000ff0000000000

This means that the instruction found two zero bytes, and it is not surprising.

\texttt{PMOVMSKB} in our case will set EAX to (in binary representation): \texttt{001000000100000b}.

Obviously, our function must take only the first zero bit and ignore the rest.

The next instruction is \texttt{BSF} (\textit{Bit Scan Forward}). This instruction finds the first bit set to 1 and stores its position into the first operand.

\texttt{EAX=0010000000100000b}

\textsuperscript{11}An order from \texttt{MSB} to \texttt{LSB}\textsuperscript{12} is used here.
After the execution of `bsf eax, eax`, EAX contains 5, meaning 1 was found at the 5th bit position (starting from zero).

MSVC has a macro for this instruction: `_BitScanForward`.

Now it is simple. If a zero byte was found, its position is added to what we have already counted and now we have the return result.

Almost all.

By the way, it is also has to be noted that the MSVC compiler emitted two loop bodies side by side, for optimization.

By the way, SSE 4.2 (that appeared in Intel Core i7) offers more instructions where these string manipulations might be even easier: [http://go.yurichev.com/17331](http://go.yurichev.com/17331)
Chapter 26

64 bits

26.1 x86-64

It is a 64-bit extension to the x86 architecture.

From the reverse engineer's perspective, the most important changes are:

- Almost all registers (except FPU and SIMD) were extended to 64 bits and got a R- prefix. 8 additional registers were added. Now GPR’s are: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

It is still possible to access the older register parts as usual. For example, it is possible to access the lower 32-bit part of the RAX register using EAX:

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX^64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The new R8-R15 registers also have their lower parts: R8D-R15D (lower 32-bit parts), R8W-R15W (lower 16-bit parts), R8L-R15L (lower 8-bit parts).

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The number of SIMD registers was doubled from 8 to 16: XMM0-XMM15.
In Win64, the function calling convention is slightly different, somewhat resemblingfastcall (64.3 on page 954). The first 4 arguments are stored in the RCX, RDX, R8, R9 registers, the rest—in the stack. The caller function must also allocate 32 bytes so the callee may save there 4 first arguments and use these registers for its own needs. Short functions may use arguments just from registers, but larger ones may save their values on the stack.

System V AMD64 ABI (Linux, *BSD, Mac OS X)[Mit13] also somewhat resembles fastcall, it uses 6 registers RDI, RSI, RDX, RCX, R8, R9 for the first 6 arguments. All the rest are passed via the stack.

See also the section on calling conventions (64 on page 952).

• The C/C++ int type is still 32-bit for compatibility.

• All pointers are 64-bit now.

This provokes irritation sometimes: now one needs twice as much memory for storing pointers, including cache memory, despite the fact that x64 CPUs can address only 48 bits of external RAM.

Since now the number of registers is doubled, the compilers have more space for maneuvering called register allocation. For us this implies that the emitted code containing less number of local variables.

For example, the function that calculates the first S-box of the DES encryption algorithm processes 32/64/128/256 values at once (depending on DES_type type (uint32, uint64, SSE2 or AVX)) using the bitslice DES method (read more about this technique here (25 on page 560)):

```c
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#define DES_type unsigned __int64

void s1 (DES_type a1,
        DES_type a2,
        ...)
```
DES_type a3,
DES_type a4,
DES_type a5,
DES_type a6,
DES_type *out1,
DES_type *out2,
DES_type *out3,
DES_type *out4

) {
    DES_type x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x16;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
    *out2 ^= x22;
    x23 = x1 | x5;
    x24 = x23 ^ x8;
    x25 = x18 & ~x2;
    x26 = a2 & ~x25;
    x27 = x24 ^ x26;
    x28 = x6 | x7;
    x29 = x28 ^ x25;
There are a lot of local variables. Of course, not all those going into the local stack.
Let's compile it with MSVC 2008 with /Ox option:

Listing 26.1: Optimizing MSVC 2008

PUBLIC   s1
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_x6$ = -20 ; size = 4
_x3$ = -16 ; size = 4
_x1$ = -12 ; size = 4
_x8$ = -8 ; size = 4
_x4$ = -4 ; size = 4
_a1$ = 8 ; size = 4
_a2$ = 12 ; size = 4
_a3$ = 16 ; size = 4
_x33$ = 20 ; size = 4
_x7$ = 20 ; size = 4
_a4$ = 20 ; size = 4
_a5$ = 24 ; size = 4
tv326 = 28 ; size = 4
_x36$ = 28 ; size = 4
_x28$ = 28 ; size = 4
_a6$ = 28 ; size = 4
_out1$ = 32 ; size = 4
_x24$ = 36 ; size = 4
_out2$ = 36 ; size = 4
_out3$ = 40 ; size = 4
_out4$ = 44 ; size = 4

_s1 PROC
sub esp, 20 ; 00000014H
mov edx, DWORD PTR _a5$[esp+16]
push ebx
mov ebx, DWORD PTR _a4$[esp+20]
push ebp
push esi
mov esi, DWORD PTR _a3$[esp+28]
push edi
mov edi, ebx
not edi
mov ebp, edi
and edi, DWORD PTR _a5$[esp+32]
mov ecx, edx
not ecx
and ebp, esi
mov eax, ecx
and eax, esi
and ecx, ebx
mov DWORD PTR _x1$[esp+36], eax
xor eax, ebx
mov esi, ebp
or esi, edx
mov DWORD PTR _x4$[esp+36], esi
and esi, DWORD PTR _a6$[esp+32]
mov DWORD PTR _x7$[esp+32], ecx
mov edx, esi
xor edx, eax
mov DWORD PTR _x6$[esp+36], edx
mov edx, DWORD PTR _a3$[esp+32]
xor edx, ebx
mov ebx, esi
xor ebx, DWORD PTR _a5$[esp+32]
mov DWORD PTR _x8$[esp+36], edx
and ebx, edx
mov ecx, edx
mov edx, ebx
xor edx, ebp
or edx, DWORD PTR _a6$[esp+32]
not ecx
and ecx, DWORD PTR _a6$[esp+32]
xor edx, edi
mov edi, edx
or edi, DWORD PTR _a2$[esp+32]
mov DWORD PTR _x3$[esp+36], ebp
mov ebp, DWORD PTR _a2$[esp+32]
xor edi, ebx
and edi, DWORD PTR _a1$[esp+32]
mov ebx, ecx
xor ebx, DWORD PTR _x7$[esp+32]
not edi
or ebx, ebp
xor edi, ebx
mov ebx, edi
mov eax, DWORD PTR _out2$[esp+32]
xor ebx, DWORD PTR [edi]
not eax
xor ebx, DWORD PTR _x6$[esp+36]
and eax, edx
mov DWORD PTR [edi], ebx
mov ebx, DWORD PTR _x7$[esp+32]
or ebx, DWORD PTR _x6$[esp+36]
mov edi, esi
or edi, DWORD PTR _x1$[esp+36]
mov DWORD PTR _x28$[esp+32], ebx
xor edi, DWORD PTR _x8$[esp+36]
mov DWORD PTR _x24$[esp+32], edi
xor edi, ecx
not edi
and edi, edx
mov ebx, edi
and ebx, ebp
xor ebx, DWORD PTR _x28$[esp+32]
xor ebx, eax
not eax
mov DWORD PTR _x33$[esp+32], ebx
and ebx, DWORD PTR _a1$[esp+32]
and eax, ebp
xor eax, ebx
mov ebx, DWORD PTR _out4$[esp+32]
xor eax, DWORD PTR [ebx]
xor eax, DWORD PTR _x24$[esp+32]
mov DWORD PTR [ebx], eax
mov eax, DWORD PTR _x28$[esp+32]
and eax, DWORD PTR _a3$[esp+32]
mov ebx, DWORD PTR _x3$[esp+36]
or edi, DWORD PTR _a3$[esp+32]
mov DWORD PTR _x36$[esp+32], eax
not eax
and eax, edx
or ebx, ebp
xor ebx, eax
not eax
and eax, DWORD PTR _x24$[esp+32]
not ebp
or eax, DWORD PTR _x3$[esp+36]
not esi
and ebp, eax
or eax, edx
xor eax, DWORD PTR _a5$[esp+32]
mov edx, DWORD PTR _x36$[esp+32]
xor edx, DWORD PTR _x4$[esp+36]
xor ebp, edi
mov edi, DWORD PTR _out1$[esp+32]
not eax
and eax, DWORD PTR _a2$[esp+32]
not ebp
and ebp, DWORD PTR _a1$[esp+32]
and edx, esi
xor eax, edx
or eax, DWORD PTR _a1$[esp+32]
not ebp
xor ebp, DWORD PTR [edi]
not ecx
and ecx, DWORD PTR _x33$[esp+32]
xor ebp, ebx
not eax
mov DWORD PTR [edi], ebp
xor eax, ecx
mov ecx, DWORD PTR _out3$[esp+32]
xor eax, DWORD PTR [ecx]
pop edi
pop esi
xor eax, ebx
pop ebp
mov DWORD PTR [ecx], eax
pop ebx
add esp, 20 ; 00000014H
5 variables were allocated in the local stack by the compiler.

Now let's try the same thing in the 64-bit version of MSVC 2008:

Listing 26.2: Optimizing MSVC 2008

```assembly
a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1 PROC
$LN3:
    mov QWORD PTR [rsp+24], rbx
    mov QWORD PTR [rsp+32], rbp
    mov QWORD PTR [rsp+16], rdx
    mov QWORD PTR [rsp+8], rcx
    push rsi
    push rdi
    push r12
    push r13
    push r14
    push r15
    mov r15, QWORD PTR a5$[rsp]
    mov rcx, QWORD PTR a6$[rsp]
    mov rbp, r8
    mov r10, r9
    mov rax, r15
    mov rdx, rbp
    not rax
    xor rdx, r9
    not r10
    mov r11, rax
    and rax, r9
    mov rsi, r10
    mov QWORD PTR x36$1$[rsp], rax
    and r11, r8
    and rsi, r8
    and r10, r15
    ret 0
_s1 ENDP
```
mov    r13, rdx
mov    rbx, r11
xor    rbx, r9
mov    r9, QWORD PTR a2$[rsp]
mov    r12, rsi
or     r12, r15
not    r13
and    r13, rcx
mov    r14, r12
and    r14, rcx
mov    rax, r14
mov    r8, r14
xor    r8, rbx
xor    rax, r15
not    rbx
and    rax, rdx
mov    rdi, rax
xor    rdi, rsi
or     rdi, rcx
xor    rdi, r10
and    rbx, rdi
mov    rcx, rdi
or     rcx, r9
xor    rcx, rax
mov    rax, r13
xor    rax, QWORD PTR x36$1$[rsp]
and    rcx, QWORD PTR a1$[rsp]
or     rax, r9
not    rcx
xor    rcx, rax
mov    rax, QWORD PTR out2$[rsp]
xor    rcx, QWORD PTR [rax]
xor    rcx, r8
mov    QWORD PTR [rax], rcx
mov    rax, QWORD PTR x36$1$[rsp]
mov    rcx, r14
or     rax, r8
or     rcx, r11
mov    r11, r9
xor    rcx, rdx
mov    QWORD PTR x36$1$[rsp], rax
mov    r8, rsi
mov    rdx, rcx
xor    rdx, r13
not    rdx
and    rdx, rdi
mov    r10, rdx
and r10, r9
xor r10, rax
xor r10, rbx
not rbx
and rbx, r9
mov rax, r10
and rax, QWORD PTR a1$[rsp]
xor rbx, rax
mov rax, QWORD PTR out4$[rsp]
xor rbx, QWORD PTR [rax]
xor rbx, rcx
mov QWORD PTR [rax], rbx
mov rbx, QWORD PTR x36$1$[rsp]
and rbx, rbp
mov r9, rbx
not r9
and r9, rdi
or r8, r11
mov rax, QWORD PTR out1$[rsp]
xor r8, r9
not r9
and r9, rcx
or rdx, rbp
mov rbp, QWORD PTR [rsp+80]
or r9, rsi
xor rbx, r12
mov rcx, r11
not rcx
not r14
not r13
and rcx, r9
or r9, rdi
and rbx, r14
xor r9, r15
xor rcx, rdx
mov rdx, QWORD PTR a1$[rsp]
not r9
not rcx
and r13, r10
and r9, r11
and rcx, rdx
xor r9, rbx
mov rbx, QWORD PTR [rsp+72]
not rcx
xor rcx, QWORD PTR [rax]
or r9, rdx
not r9
xor  rcx,  r8
mov  QWORD  PTR [rax], rcx
mov  rax,  QWORD  PTR  out3$[rsp]
xor  r9,  r13
xor  r9,  QWORD  PTR [rax]
xor  r9,  r8
mov  QWORD  PTR [rax],  r9
pop  r15
pop  r14
pop  r13
pop  r12
pop  rdi
pop  rsi
ret  0
s1   ENDP

Nothing was allocated in the local stack by the compiler, x36 is synonym for a5. By the way, there are CPUs with much more GPR's, e.g. Itanium (128 registers).

### 26.2 ARM

64-bit instructions appeared in ARMv8.

### 26.3 Float point numbers

How floating point numbers are processed in x86-64 is explained here: 27 on the following page.
Chapter 27

Working with floating point numbers using SIMD

Of course, the FPU has remained in x86-compatible processors when the SIMD extensions were added.

The SIMD extensions (SSE2) offer an easier way to work with floating-point numbers.

The number format remains the same (IEEE 754).

So, modern compilers (including those generating for x86-64) usually use SIMD instructions instead of FPU ones.

It can be said that it’s good news, because it’s easier to work with them.

We are going to reuse the examples from the FPU section here: 17 on page 298.

27.1 Simple example

```c
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
}
```
27.1.1 x64

The input floating point values are passed in the XMM0-XMM3 registers, all the rest—via the stack\(^1\).

\(a\) is passed in XMM0, \(b\)—via XMM1. The XMM-registers are 128-bit (as we know from the section about SIMD: 25 on page 559), but the double values are 64 bit, so only lower register half is used.

DIVSD is an SSE-instruction that stands for “Divide Scalar Double-Precision Floating-Point Values”, it just divides one value of type double by another, stored in the lower halves of operands.

The constants are encoded by compiler in IEEE 754 format.

MULSD and ADDSD work just as the same, but do multiplication and addition.

The result of the function’s execution in type double is left in the in XMM0 register.

That is how non-optimizing MSVC works:

\footnotesize
\begin{verbatim}
Listing 27.2: MSVC 2012 x64
__real@4010666666666666 DQ 04010666666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14
a$ = 8
b$ = 16
f PROC
  movsdx QWORD PTR [rsp+16], xmm1
f ENDP
\end{verbatim}
\end{footnotesize}

\(^1\) MSDN: Parameter Passing
Slightly redundant. The input arguments are saved in the “shadow space” (8.2.1 on page 137), but only their lower register halves, i.e., only 64-bit values of type double. GCC produces the same code.

### 27.1.2 x86

Let’s also compile this example for x86. Despite the fact it’s generating for x86, MSVC 2012 uses SSE2 instructions:

#### Listing 27.3: Non-optimizing MSVC 2012 x86

```assembly
.tv70 = -8 ; size = 8
._a$ = 8 ; size = 8
._b$ = 16 ; size = 8
_f PROC
    push ebp
    mov ebp, esp
    sub esp, 8
    movsd xmm0, QWORD PTR _a$[ebp]
    divsd xmm0, QWORD PTR __real@40091eb851eb851f
    movsd xmm1, QWORD PTR _b$[ebp]
    mulsd xmm1, QWORD PTR __real@4010666666666666
    addsd xmm0, xmm1
    movsd QWORD PTR tv70[ebp], xmm0
    fld QWORD PTR tv70[ebp]
    mov esp, ebp
    pop ebp
    ret 0
_f ENDP
```

#### Listing 27.4: Optimizing MSVC 2012 x86

```assembly
.tv67 = 8 ; size = 8
._a$ = 8 ; size = 8
._b$ = 16 ; size = 8
_f PROC
    movsd xmm1, QWORD PTR _a$[esp-4]
```
It's almost the same code, however, there are some differences related to calling conventions: 1) the arguments are passed not in XMM registers, but in the stack, like in the FPU examples (17 on page 298); 2) the result of the function is returned in ST(0) — in order to do so, it's copied (through local variable tv) from one of the XMM registers to ST(0).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>divsd xmm1, QWORD PTR __real@40091eb851eb851f</td>
<td>Divide double-precision floating-point numbers</td>
</tr>
<tr>
<td>movsd xmm0, QWORD PTR _b$[esp-4]</td>
<td>Move double-precision floating-point numbers</td>
</tr>
<tr>
<td>mulsd xmm0, QWORD PTR __real@4010666666666666</td>
<td>Multiply double-precision floating-point numbers</td>
</tr>
<tr>
<td>addsd xmm1, xmm0</td>
<td>Add double-precision floating-point numbers</td>
</tr>
<tr>
<td>movsd QWORD PTR tv67[esp-4], xmm1</td>
<td>Move double-precision floating-point numbers</td>
</tr>
<tr>
<td>fld QWORD PTR tv67[esp-4]</td>
<td>Load double-precision floating-point number</td>
</tr>
<tr>
<td>ret 0</td>
<td>Return from function</td>
</tr>
</tbody>
</table>

_endp
Let's try the optimized example in OllyDbg:

Figure 27.1: OllyDbg: MOVSD loads the value of $a$ into XMM1
Figure 27.2: OllyDbg: DIVSD calculated quotient and stored it in XMM1
Figure 27.3: OllyDbg: MULSD calculated product and stored it in XMM0
Figure 27.4: OllyDbg: ADDSD adds value in XMM0 to XMM1
We see that OllyDbg shows the XMM registers as pairs of *double* numbers, but only the *lower* part is used. Apparently, OllyDbg shows them in that format because the SSE2 instructions (suffixed with `-SD`) are executed right now. But of course, it’s possible to switch the register format and to see their contents as 4 *float*-numbers or just as 16 bytes.
#include <math.h>  
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

    return 0;
}

They are passed in the lower halves of the XMM0-XMM3 registers.

Listing 27.5: Optimizing MSVC 2012 x64

$SG1354 DB '32.01 ^ 1.54 = %lf', 0aH, 00H
__real@40400147ae147ae1 DQ 040400147ae147ae1r ; 32.01  
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r ; 1.54

main PROC
    sub     rsp, 40 ; ↯ 00000028H
    movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
    movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
    call    pow
    lea     rcx, OFFSET FLAT:$SG1354
    movaps  xmm1, xmm0
    movd    rdx, xmm1
    call    printf
    xor     eax, eax
    add     rsp, 40 ; ↯ 00000028H
    ret     0
main ENDP

There is no MOVSDX instruction in Intel [Int13] and AMD [AMD13a] manuals, there it is called just MOVSD. So there are two instructions sharing the same name in x86 (about the other see: A.6.2 on page 1305). Apparently, Microsoft developers wanted to get rid of the mess, so they renamed it to MOVSDX. It just loads a value into the lower half of a XMM register.

pow() takes arguments from XMM0 and XMM1, and returns result in XMM0. It is then moved to RDX for printf(). Why? Maybe because printf()—is a variable arguments function?
Listing 27.6: Optimizing GCC 4.4.6 x64

```
.LC2:  .string "32.01 ^ 1.54 = %lf\n"
main:
  sub    rsp, 8
  movsd  xmm1, QWORD PTR .LC0[rip]
  movsd  xmm0, QWORD PTR .LC1[rip]
  call   pow
    ; result is now in XMM0
  mov    edi, OFFSET FLAT:.LC2
  mov    eax, 1 ; number of vector registers passed
  call   printf
  xor    eax, eax
  add    rsp, 8
  ret
.LC0:
  .long  171798692
  .long  1073259479
.LC1:
  .long  2920577761
  .long  1077936455
```

GCC generates clearer output. The value for `printf()` is passed in XMM0. By the way, here is a case when 1 is written into EAX for `printf()`—this implies that one argument will be passed in vector registers, just as the standard requires [Mit13].

### 27.3 Comparison example

```
#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;
    return b;
};

int main()
{
    printf ("%f\n", d_max (1.2, 3.4));
    printf ("%f\n", d_max (5.6, -4));
};
```
27.3.1 x64

Listing 27.7: Optimizing MSVC 2012 x64

```assembly
a$ = 8
b$ = 16
d_max PROC
    comisd xmm0, xmm1
    ja SHORT $LN2@d_max
    movaps xmm0, xmm1
$LN2@d_max:
    fatret 0
d_max ENDP
```

Optimizing MSVC generates a code very easy to understand.

COMISD is “Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS”. Essentially, that is what it does.

Non-optimizing MSVC generates more redundant code, but it is still not hard to understand:

Listing 27.8: MSVC 2012 x64

```assembly
a$ = 8
b$ = 16
d_max PROC
    movsdx QWORD PTR [rsp+16], xmm1
    movsdx QWORD PTR [rsp+8], xmm0
    movsdx xmm0, QWORD PTR a$[rsp]
    comisd xmm0, QWORD PTR b$[rsp]
    jbe SHORT $LN1@d_max
    movsdx xmm0, QWORD PTR a$[rsp]
    jmp SHORT $LN2@d_max
$LN1@d_max:
    movsdx xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
    fatret 0
d_max ENDP
```

However, GCC 4.4.6 did more optimizations and used the MAXSD (“Return Maximum Scalar Double-Precision Floating-Point Value”) instruction, which just choose the maximum value!

Listing 27.9: Optimizing GCC 4.4.6 x64

```assembly
d_max:
    maxsd xmm0, xmm1
    ret
```

602
Let's compile this example in MSVC 2012 with optimization turned on:

**Listing 27.10: Optimizing MSVC 2012 x86**

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_d_max PROC
    movsd xmm0, QWORD PTR _a$[esp-4]
    comisd xmm0, QWORD PTR _b$[esp-4]
    jbe SHORT $LN1@d_max
    fld QWORD PTR _a$[esp-4]
    ret 0
$LN1@d_max:
    fld QWORD PTR _b$[esp-4]
    ret 0
_d_max ENDP
```

Almost the same, but the values of \( a \) and \( b \) are taken from the stack and the function result is left in ST(0).

If we load this example in OllyDbg, we can see how the `COMISD` instruction compares values and sets/clears the CF and PF flags:

![Figure 27.6: OllyDbg: COMISD changed CF and PF flags](image-url)
CHAPTER 27. WORKING WITH FLOATING POINT NUMBERS USING SIMD

27.4 Calculating machine epsilon: x64 and SIMD

Let’s revisit the “calculating machine epsilon” example for double listing.22.2.2. Now we compile it for x64:

```
Listing 27.11: Optimizing MSVC 2012 x64

v$ = 8
calculate_machine_epsilon PROC
    movsdx QWORD PTR v$[rsp], xmm0
    movaps xmm1, xmm0
    inc QWORD PTR v$[rsp]
    movsdx xmm0, QWORD PTR v$[rsp]
    subsd xmm0, xmm1
    ret 0
calculate_machine_epsilon ENDP
```

There is no way to add 1 to a value in 128-bit XMM register, so it must be placed into memory.

There is, however, the ADDSD instruction (Add Scalar Double-Precision Floating-Point Values) which can add a value to the lowest 64-bit half of a XMM register while ignoring the higher one, but MSVC 2012 probably is not that good yet ².

Nevertheless, the value is then reloaded to a XMM register and subtraction occurs. SUBSD is “Subtract Scalar Double-Precision Floating-Point Values”, i.e., it operates on the lower 64-bit part of 128-bit XMM register. The result is returned in the XMM0 register.

27.5 Pseudo-random number generator example revisited

Let’s revisit “pseudo-random number generator example” example listing.22.1. If we compile this in MSVC 2012, it will use the SIMD instructions for the FPU.

```
Listing 27.12: Optimizing MSVC 2012

__real@3f800000 DD 03f800000r ; 1

?float_rand@@YAMXZ PROC
    push ecx
```

²As an exercise, you may try to rework this code to eliminate the usage of the local stack.
All instructions have the -SS suffix, which stands for “Scalar Single”. “Scalar” implies that only one value is stored in the register. “Single” stands for float data type.

### 27.6 Summary

Only the lower half of XMM registers is used in all examples here, to store number in IEEE 754 format.

Essentially, all instructions prefixed by -SD (“Scalar Double-Precision”)—are instructions working with floating point numbers in IEEE 754 format, stored in the lower 64-bit half of a XMM register.

And it is easier than in the FPU, probably because the SIMD extensions were evolved in a less chaotic way than the FPU ones in the past. The stack register model is not used.

If you would try to replace `double` with `float` in these examples, the same instructions will be used, but prefixed with -SS (“Scalar Single-Precision”), for example, MOVSS, COMISS, ADDSS, etc.

“Scalar” implies that the SIMD register containing only one value instead of several. Instructions working with several values in a register simultaneously have “Packed” in their name.
Needless to say, the SSE2 instructions work with 64-bit IEEE 754 numbers (double), while the internal representation of the floating-point numbers in FPU is 80-bit numbers. Hence, the FPU may produce less round-off errors and as a consequence, FPU may give more precise calculation results.
Chapter 28

ARM-specific details

28.1 Number sign (#) before number

The Keil compiler, IDA and objdump precede all numbers with the “#” number sign, for example: listing.14.1.4. But when GCC 4.9 generates assembly language output, it doesn’t, for example: listing.39.3.

The ARM listings in this book are somewhat mixed.

It’s hard to say, which method is right. Supposedly, one has to obey the rules accepted in environment he/she works in.

28.2 Addressing modes

This instruction is possible in ARM64:

```
ldr x0, [x29, 24]
```

This means add 24 to the value in X29 and load the value from this address. Please note that 24 is inside the brackets. The meaning is different if the number is outside the brackets:

```
ldr w4, [x1], 28
```

This means load the value at the address in X1, then add 28 to X1.

ARM allows you to add or subtract a constant to/from the address used for loading. And it’s possible to do that both before and after loading.
There is no such addressing mode in x86, but it is present in some other processors, even on PDP-11. There is a legend that the pre-increment, post-increment, pre-decrement and post-decrement modes in PDP-11, were “guilty” for the appearance of such C language (which developed on PDP-11) constructs as *ptr++, *++ptr, *ptr--, *--ptr. By the way, this is one of the hard to memorize C features. This is how it is:

<table>
<thead>
<tr>
<th>C term</th>
<th>ARM term</th>
<th>C statement</th>
<th>how it works</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post-increment</td>
<td>post-indexed addressing</td>
<td>*ptr++</td>
<td>use *ptr value, then increment ptr pointer</td>
</tr>
<tr>
<td>Post-decrement</td>
<td>post-indexed addressing</td>
<td>*ptr--</td>
<td>use *ptr value, then decrement ptr pointer</td>
</tr>
<tr>
<td>Pre-increment</td>
<td>pre-indexed addressing</td>
<td>*++ptr</td>
<td>increment ptr pointer, then use *ptr value</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>pre-indexed addressing</td>
<td>*--ptr</td>
<td>decrement ptr pointer, then use *ptr value</td>
</tr>
</tbody>
</table>

Pre-indexing is marked with an exclamation mark in the ARM assembly language. For example, see line 2 in listing 3.15.

Dennis Ritchie (one of the creators of the C language) mentioned that it probably was invented by Ken Thompson (another C creator) because this processor feature was present in PDP-7 [Rit86][Rit93]. Thus, C language compilers may use it, if it is present on the target processor.

That’s very convenient for array processing.

### 28.3 Loading a constant into a register

#### 28.3.1 32-bit ARM

Aa we already know, all instructions have a length of 4 bytes in ARM mode and 2 bytes in Thumb mode. Then how can we load a 32-bit value into a register, if it’s not possible to encode it in one instruction?

Let’s try:

```c
unsigned int f()
{
    return 0x12345678;
};
```

Listing 28.1: GCC 4.6.3 -O3 ARM mode

```
f: 1dr r0, .L2
```
So, the 0x12345678 value is just stored aside in memory and loaded if needed. But it's possible to get rid of the additional memory access.

Listing 28.2: GCC 4.6.3 -O3 -march=armv7-a (ARM mode)

```
movw r0, #22136 ; 0x5678
movt r0, #4660 ; 0x1234
bx lr
```

We see that the value is loaded into the register by parts, the lower part first (using MOVW), then the higher (using MOVT).

This implies that 2 instructions are necessary in ARM mode for loading a 32-bit value into a register. It's not a real problem, because in fact there are not many constants in real code (except of 0 and 1). Does it mean that the two-instruction version is slower than one-instruction version? Doubtfully. Most likely, modern ARM processors are able to detect such sequences and execute them fast.

On the other hand, IDA is able to detect such patterns in the code and disassembles this function as:

```
MOV R0, 0x12345678
BX LR
```

### 28.3.2 ARM64

```
uint64_t f()
{
    return 0x12345678ABCDEF01;
}
```

Listing 28.3: GCC 4.9.1 -O3

```
mov  x0, 61185 ; 0xef01
movk x0, 0xabcd, lsl 16
movk x0, 0x5678, lsl 32
movk x0, 0x1234, lsl 48
ret
```

MOVK stands for “MOV Keep”, i.e., it writes a 16-bit value into the register, not touching the rest of the bits. The LSL suffix shifts left the value by 16, 32 and 48 bits at each step. The shifting is done before loading. This implies that 4 instructions are necessary to load a 64-bit value into a register.
Storing floating-point number into register

It’s possible to store a floating-point number into a D-register using only one instruction.

For example:

```c
double a()
{
    return 1.5;
}
```

Listing 28.4: GCC 4.9.1 -O3 + objdump

```assembly
0000000000000000 <a>:
  0: 1e6f1000 fmov d0, #1.500000000000000000e+000
  4: d65f03c0 ret
```

The number 1.5 was indeed encoded in a 32-bit instruction. But how? In ARM64, there are 8 bits in the FMOV instruction for encoding some floating-point numbers. The algorithm is called VFPExpandImm() in [ARM13a]. This is also called mini-float\(^1\). We can try different values: the compiler is able to encode 30.0 and 31.0, but it couldn’t encode 32.0, as 8 bytes have to be allocated for this number in the IEEE 754 format:

```c
double a()
{
    return 32;
}
```

Listing 28.5: GCC 4.9.1 -O3

```assembly
a:
  1dr d0, .LC0
  ret

.LC0:
  .word 0
  .word 1077936128
```

### 28.4 Relocs in ARM64

As we know, there are 4-byte instructions in ARM64, so it is impossible to write a large number into a register using a single instruction. Nevertheless, an executable image can be loaded at any random address in memory, so that’s why

\(^1\)wikipedia
relocs exists. Read more about them (in relation to Win32 PE): 68.2.6 on page 992.

The address is formed using the ADRP and ADD instruction pair in ARM64. The first loads a 4KiB-page address and the second one adds the remainder. Let's compile the example from “Hello, world!” (listing.6) in GCC (Linaro) 4.9 under win32:

Listing 28.6: GCC (Linaro) 4.9 and objdump of object file

```bash
...>aarch64-linux-gnu-gcc.exe hw.c -c
...>aarch64-linux-gnu-objdump.exe -d hw.o
...
```

```bash
0000000000000000 <main>:
  0: a9bf7bfd stp x29, x30, [sp,#-16]!
  4: 910003fd mov x29, sp
  8: 90000000 adrp x0, 0 <main>
  c: 91000000 add x0, x0, #0x0
  10: 94000000 bl 0 <printf>
  14: 52800000 mov w0, #0x0
   ← // #0
  18: a8c17bfd ldp x29, x30, [sp],#16
  1c: d65f03c0 ret
```

```bash
...>aarch64-linux-gnu-objdump.exe -r hw.o
...
```

RELOCATION RECORDS FOR [.text]:

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000008</td>
<td>R_AARCH64_ADR_PREL_PG_HI21</td>
<td>.rodata</td>
</tr>
<tr>
<td>000000000000000c</td>
<td>R_AARCH64_ADD_ABS_LO12_NC</td>
<td>.rodata</td>
</tr>
<tr>
<td>0000000000000010</td>
<td>R_AARCH64_CALL26</td>
<td>printf</td>
</tr>
</tbody>
</table>

So there are 3 relocs in this object file.

- The first one takes the page address, cuts the lowest 12 bits and writes the remaining high 21 bits to the ADRP instruction’s bit fields. This is because we don’t need to encode the low 12 bits, and the ADRP instruction has space only for 21 bits.

- The second one puts the 12 bits of the address relative to the page start into the ADD instruction’s bit fields.

- The last, 26-bit one, is applied to the instruction at address 0x10 where the jump to the printf() function is. All ARM64 (and in ARM in ARM mode) instruction addresses have zeroes in the two lowest bits (because all instructions have a size of 4 bytes), so one need to encode only the highest
26 bits of 28-bit address space ($\pm 128$MB).

There are no such relocs in the executable file: because it’s known where the “Hello!” string is located, in which page, and the address of `puts()` is also known. So there are values set already in the ADRP, ADD and BL instructions (the linker has written them while linking):

```
Listing 28.7: objdump of executable file
```

```
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 48656c6c 6f210000 ........Hello!
```

As an example, let’s try to disassemble the BL instruction manually. 0x97ffffffa0 is 101011111111111111111110100000b. According to [ARM13a, p. C5.6.26], `imm26` is the last 26 bits: `imm26 = 11111111111111111110100000`. It is 0x3FFFFFFA0, but the MSB is 1, so the number is negative, and we can convert it manually to convenient form for us. By the rules of negation (30 on page 618), just invert all bits: (it is 1011111=0x5F), and add 1 (0x5F+1=0x60). So the number in signed form is -0x60. Let’s multiplyc -0x60 by 4 (because address stored in opcode is divided by 4): it is -0x180. Now let’s calculate destination address: 0x4005a0 + (-0x180) = 0x400420 (please note: we consider the address of the BL instruction, not the current value of PC, which may be different!). So the destination address is 0x400420.

More about ARM64-related relocs: [ARM13b].
Chapter 29

MIPS-specific details

29.1 Loading constants into register

```c
unsigned int f()
{
    return 0x12345678;
};
```

All instructions in MIPS, just like ARM, have a of 32-bit, so it’s not possible to embed a 32-bit constant into one instruction. So this translates to at least two instructions: the first loads the high part of the 32-bit number and the second one applies an OR operation, which effectively sets the low 16-bit part of the target register:

```
Listing 29.1: GCC 4.4.5 -O3 (assembly output)
li $2,305397760 # 0x12340000
j $31
ori $2,$2,0x5678 ; branch delay slot
```

IDA is fully aware of such frequently encountered code patterns, so, for convenience it shows the last ORI instruction as the LI pseudoinstruction, which allegedly loads a full 32-bit number into the $V0 register.

```
Listing 29.2: GCC 4.4.5 -O3 (IDA)
lui $v0, 0x1234
jr $ra
li $v0, 0x12345678 ; branch delay slot
```
The GCC assembly output has the LI pseudoinstruction, but in fact, LUI (“Load Upper Immediate”) is there, which stores a 16-bit value into the high part of the register.

29.2 Further reading about MIPS

[Swe10].
Part II

Important fundamentals
Chapter 30

Signed number representations

There are several methods for representing signed numbers\(^1\), but “two’s complement” is the most popular one in computers.

Here is a table for some byte values:

<table>
<thead>
<tr>
<th>binary</th>
<th>hexadecimal</th>
<th>unsigned</th>
<th>signed (2’s complement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111111</td>
<td>0x7f</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>01111110</td>
<td>0x7e</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000010</td>
<td>0x6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>00000101</td>
<td>0x5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>00000100</td>
<td>0x4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>00000011</td>
<td>0x3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>00000010</td>
<td>0x2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>00000001</td>
<td>0x1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00000000</td>
<td>0x0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11111111</td>
<td>0xff</td>
<td>255</td>
<td>-1</td>
</tr>
<tr>
<td>11111110</td>
<td>0xfe</td>
<td>254</td>
<td>-2</td>
</tr>
<tr>
<td>11111101</td>
<td>0xfd</td>
<td>253</td>
<td>-3</td>
</tr>
<tr>
<td>11111100</td>
<td>0xfc</td>
<td>252</td>
<td>-4</td>
</tr>
<tr>
<td>11111011</td>
<td>0xfb</td>
<td>251</td>
<td>-5</td>
</tr>
<tr>
<td>11111010</td>
<td>0xfa</td>
<td>250</td>
<td>-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000010</td>
<td>0x82</td>
<td>130</td>
<td>-126</td>
</tr>
<tr>
<td>10000001</td>
<td>0x81</td>
<td>129</td>
<td>-127</td>
</tr>
<tr>
<td>10000000</td>
<td>0x80</td>
<td>128</td>
<td>-128</td>
</tr>
</tbody>
</table>

\(^1\)Wikipedia
The difference between signed and unsigned numbers is that if we represent 0xFFFFFFFFFE and 0x00000002 as unsigned, then the first number (4294967294) is bigger than the second one (2). If we represent them both as signed, the first one becomes −2, and it is smaller than the second (2). That is the reason why conditional jumps (12 on page 168) are present both for signed (e.g. JG, JL) and unsigned (JA, JB) operations.

For the sake of simplicity, this is what one needs to know:

• Numbers can be signed or unsigned.

• C/C++ signed types:
  - `int64_t` (-9,223,372,036,854,775,808..9,223,372,036,854,775,807) (-9.2.. 9.2 quintillions) or 0x8000000000000000..0xFFFFFFFFFFFFFFFF,
  - `int` (-2,147,483,648..2,147,483,647 (-2.15..2.15Gb) or 0x80000000..0x7FFFFFFF,
  - `char` (-128..127 or 0x80..0x7F,
  - `ssize_t`.

• Unsigned:
  - `uint64_t` (0..18,446,744,073,709,551,615 (18 quintillions) or 0..0xFFFFFFFFFFFFFFFF,
  - unsigned `int` (0..4,294,967,295 (4.3Gb) or 0..0xFFFFFFFF),
  - unsigned `char` (0..255 or 0..0xFF),
  - `size_t`.

• Signed types have the sign in the most significant bit: 1 mean “minus”, 0 mean “plus”.

• Promoting to a larger data types is simple: 24.5 on page 557.

• Negation is simple: just invert all bits and add 1. We can remember that a number of inverse sign is located on the opposite side at the same proximity from zero. The addition of one is needed because zero is present in the middle.

• The addition and subtraction operations work well for both signed and unsigned values. But for multiplication and division operations, x86 has different instructions: IDIV/IMUL for signed and DIV/MUL for unsigned.

• Here are some more instructions that work with signed numbers: CBW/CWD/CWDE/CDQ/CDQE (A.6.3 on page 1309), MOVsx (15.1.1 on page 274), SAR (A.6.3 on page 1315).
Chapter 31

Endianness

The endianness is a way of representing values in memory.

31.1 Big-endian

The 0x12345678 value is represented in memory as:

<table>
<thead>
<tr>
<th>address in memory</th>
<th>byte value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>0x12</td>
</tr>
<tr>
<td>+1</td>
<td>0x34</td>
</tr>
<tr>
<td>+2</td>
<td>0x56</td>
</tr>
<tr>
<td>+3</td>
<td>0x78</td>
</tr>
</tbody>
</table>

Big-endian CPUs include Motorola 68k, IBM POWER.

31.2 Little-endian

The 0x12345678 value is represented in memory as:

<table>
<thead>
<tr>
<th>address in memory</th>
<th>byte value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>0x78</td>
</tr>
<tr>
<td>+1</td>
<td>0x56</td>
</tr>
<tr>
<td>+2</td>
<td>0x34</td>
</tr>
<tr>
<td>+3</td>
<td>0x12</td>
</tr>
</tbody>
</table>

Little-endian CPUs include Intel x86.
31.3 Example

Let's take big-endian MIPS Linux installed and ready in QEMU. And let's compile this simple example:

```c
#include <stdio.h>

int main()
{
    int v, i;

    v=123;

    printf("%02X %02X %02X %02X\n",
           *(char*)&v,
           **(((char*)&v)+1),
           **(((char*)&v)+2),
           **(((char*)&v)+3));
}
```

After running it we get:

```
root@debian-mips:# ./a.out
00 00 00 7B
```

That is it. 0x7B is 123 in decimal. In little-endian architectures, 7B is the first byte (you can check on x86 or x86-64), but here it is the last one, because the highest byte goes first.

That's why there are separate Linux distributions for MIPS ("mips" (big-endian) and "mipsel" (little-endian)). It is impossible for a binary compiled for one endianness to work on an OS with different endianness.

There is another example of MIPS big-endianness in this book: 21.4.3 on page 499.

31.4 Bi-endian

CPUs that may switch between endianness are ARM, PowerPC, SPARC, MIPS, IA64, etc.

---

1 Available for download here: [http://go.yurichev.com/17008](http://go.yurichev.com/17008)
2 Intel Architecture 64 (Itanium): 93 on page 1273
31.5 Converting data

The BSWAP instruction can be used for conversion.

TCP/IP network data packets use the big-endian conventions, so that is why a program working on a little-endian architecture has to convert the values.

The htonl() and htons() functions are usually used.

In TCP/IP, big-endian is also called “network byte order”, while byte order on the computer “host byte order”. “host byte order” is little-endian on Intel x86 and other little-endian architectures, but it is big-endian on IBM POWER, so htonl() and htons() don’t shuffle any bytes on the latter.
Chapter 32

Memory

There are 3 main types of memory:

- Global memory AKA “static memory allocation”. No need to allocate explicitly, the allocation is done just by declaring variables/arrays globally. These are global variables, residing in the data or constant segments. They are available globally (hence, considered as an anti-pattern). Not convenient for buffers/arrays, because they must have a fixed size. Buffer overflows that occur here usually overwrite variables or buffers residing next to them in memory. There's an example in this book: 7.2 on page 103.

- Stack AKA “allocate on stack”. The allocation is done just by declaring variables/arrays locally in the function. These are usually local variables for the function. Sometimes these local variable are also available to descending functions (to callee functions, if caller passes a pointer to a variable to the callee to be executed). Allocation and deallocation are very fast, it just SP needs to be shifted. But they're also not convenient for buffers/arrays, because the buffer size has to be fixed, unless alloca() (5.2.4 on page 43) (or a variable-length array) is used. Buffer overflows usually overwrite important stack structures: 18.2 on page 370.

- Heap AKA “dynamic memory allocation”. Allocation/deallocation is done by calling malloc()/free() or new/delete in C++. This is the most convenient method: the block size may be set at runtime. Resizing is possible (using realloc()), but can be slow. This is the slowest way to allocate memory: the memory allocator must support and update all control structures while allocating and deallocating. Buffer overflows usually overwrite these structures. Heap allocations are also source of memory leak problems: each memory block has to be deallocated explicitly, but one may forget about it, or do it incorrectly. Another problem is the “use after free”—using a mem-
ory block after \texttt{free()} was called on it, which is very dangerous. Example in this book: \texttt{21.2 on page 473}. 
Chapter 33

CPU

33.1 Branch predictors

Some modern compilers try to get rid of conditional jump instructions. Examples in this book are: 12.1.2 on page 182, 12.3 on page 194, 19.5.2 on page 445.

This is because the branch predictor is not always perfect, so the compilers try to do without conditional jumps, if possible.

Conditional instructions in ARM (like ADRcc) are one way, another one is the CMOVcc x86 instruction.

33.2 Data dependencies

Modern CPUs are able to execute instructions simultaneously (OOE\(^1\)), but in order to do so, the results of one instruction in a group must not influence the execution of others. Hence, the compiler endeavors to use instructions with minimal influence on the CPU state.

That's why the LEA instruction is so popular, because it does not modify CPU flags, while other arithmetic instructions does.

\(^1\) Out-of-order execution
Chapter 34

Hash functions

A very simple example is CRC32, an algorithm that provides “stronger” checksum for integrity checking purposes. It is impossible to restore the original text from the hash value, it has much less information: But CRC32 is not cryptographically secure: it is known how to alter a text in a way that the resulting CRC32 hash value will be the one we need. Cryptographic hash functions are protected from this.

MD5, SHA1, etc are such functions and they are widely used to hash user passwords in order to store them in a database. Indeed: an internet forum database may not contain user passwords (a stolen database can compromise all users’ passwords) but only hashes (so a cracker can’t reveal the passwords). Besides, an internet forum engine does not need to be aware of your password, it needs only to check if its hash is the same as the one in the database, and give you access if they match. One of the simplest password cracking methods is just to try hashing all possible passwords in order to see which matches the resulting value that we need. Other methods are much more complex.

34.1 How do one-way functions work?

A one-way function is a function which is able to transform one value into another, while it is impossible (or very hard) to reverse it. Some people have difficulties while understanding how this is possible at all. Here is a simple demonstration.

We have a vector of 10 numbers in range 0..9, each is present only once, for example:

4 6 0 1 3 5 7 8 9 2
The algorithm for the simplest possible one-way function is:

- take the number at zeroth position (4 in our case);
- take the number at first position (6 in our case);
- swap numbers at positions of 4 and 6.

Let’s mark the numbers at positions 4 and 6:

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let’s swap them and we get this result:

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>0</th>
<th>1</th>
<th>7</th>
<th>5</th>
<th>3</th>
<th>8</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

While looking at the result, and even if we know the algorithm, we can’t know unambiguously the initial state, because the first two numbers could be 0 and/or 1, and then they could participate in the swapping procedure.

This is an utterly simplified example for demonstration. Real one-way functions are much more complex.
Part III

Slightly more advanced examples
Chapter 35

Temperature converting

Another very popular example in programming books for beginners is a small program that converts Fahrenheit temperature to Celsius or back.

\[ C = \frac{5 \cdot (F - 32)}{9} \]

We can also add simple error handling: 1) we must check if the user has entered a correct number; 2) we must check if the Celsius temperature is not below \(-273\) (which is below absolute zero, as we may remember from school physics lessons).

The `exit()` function terminates the program instantly, without returning to the caller function.

### 35.1 Integer values

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int celsius, fahr;
    printf("Enter temperature in Fahrenheit:\n");
    if (scanf("%d", &fahr)!=1)
    {
        printf("Error while parsing your input\n");
        exit(0);
    }
}
```

celsius = 5 * (fahr-32) / 9;

if (celsius<-273)
{
    printf ("Error: incorrect temperature!\n");
    exit(0);
};
printf ("Celsius: %d\n", celsius);

35.1.1 Optimizing MSVC 2012 x86

Listing 35.1: Optimizing MSVC 2012 x86

SG4228 DB 'Enter temperature in Fahrenheit:', 0AH, 00H
SG4230 DB '%d', 00H
SG4231 DB 'Error while parsing your input', 0AH, 00H
SG4233 DB 'Error: incorrect temperature!', 0AH, 00H
SG4234 DB 'Celsius: %d', 0AH, 00H

_fahr$ = -4 ; size
_gas = 4

_main PROC
    push ecx
    push esi
    mov esi, DWORD PTR __imp__printf
    push OFFSET $SG4228 ; 'Enter temperature in
    call esi ; call printf()
    lea eax, DWORD PTR _fahr$[esp+12]
    push eax
    push OFFSET $SG4230 ; '%d'
    call DWORD PTR __imp__scanf
    add esp, 12 ;

_0000000cH
    cmp eax, 1
    je SHORT $LN2@main
    push OFFSET $SG4231 ; 'Error while parsing
    call esi ; call printf()
    add esp, 4
    push 0
    call DWORD PTR __imp__exit

$LN9@main:
$LN2@main:
What we can say about it:

- The address of printf() is first loaded in the ESI register, so the subsequent printf() calls are done just by the CALL ESI instruction. It’s a very popular compiler technique, possible if several consequent calls to the same function are present in the code, and/or if there is a free register which can be used for this.

- We see the ADD EAX, -32 instruction at the place where 32 has to be subtracted from the value. \( EAX = EAX + (-32) \) is equivalent to \( EAX = EAX - 32 \) and somehow, the compiler decided to use ADD instead of SUB. Maybe it’s worth it, it’s hard to be sure.
• The LEA instruction is used when the value is to be multiplied by 5: `lea ecx, DWORD PTR [eax+eax*4]`. Yes, \( i + i \times 4 \) is equivalent to \( i \times 5 \) and LEA works faster then IMUL. By the way, the SHL EAX, 2 / ADD EAX, EAX instruction pair could be also used here instead— some compilers do it like.

• The division by multiplication trick (41 on page 676) is also used here.

• `main()` returns 0 if we don’t have `return 0` at its end. The C99 standard tells us [ISO07, p. 5.1.2.2.3] that `main()` will return 0 in case the `return` statement is missing. This rule works only for the `main()` function. Though, MSVC doesn’t officially support C99, but maybe it support it partially?

### 35.1.2 Optimizing MSVC 2012 x64

The code is almost the same, but we can find `INT 3` instructions after each `exit()` call.

```
xor ecx, ecx
    call QWORD PTR __imp_exit
    int 3
```

`INT 3` is a debugger breakpoint.

It is known that `exit()` is one of the functions which can never return 1, so if it does, something really odd has happened and it’s time to load the debugger.

### 35.2 Floating-point values

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

int main()
{
    double celsius, fahr;
    printf("Enter temperature in Fahrenheit:\n");
    if (scanf("%lf", &fahr)!=1)
    {
        printf("Error while parsing your input\n");
        exit(0);
    }
    printf("Enter temperature in Fahrenheit:\n");
    exit(0);

    celsius = 5 * (fahr-32) / 9;

    another popular one is `longjmp()`
```
if (celsius<-273)
{
    printf("Error: incorrect temperature!\n");
    exit(0);
};
printf("Celsius: %lf\n", celsius);

MSVC 2010 x86 uses FPU instructions...

Listing 35.2: Optimizing MSVC 2010 x86

$SG4038 DB 'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4040 DB '%lf', 00H
$SG4041 DB 'Error while parsing your input', 0aH, 00H
$SG4043 DB 'Error: incorrect temperature!', 0aH, 00H
$SG4044 DB 'Celsius: %lf', 0aH, 00H

__real@c0711000000000000 DQ 0c0711000000000000r ; -273
__real@4022000000000000 DQ 040220000000000000r ; 9
__real@4014000000000000 DQ 040140000000000000r ; 5
__real@4040000000000000 DQ 040400000000000000r ; 32

_fahr$ = -8 ; size ▽
\_ = 8

_main PROC
    sub esp, 8
    push esi
    mov esi, DWORD PTR __imp__printf
    push OFFSET $SG4038 ; 'Enter temperature in\n        Fahrenheit:'
    call esi ; call printf()
    lea eax, DWORD PTR _fahr$[esp+16]
    push eax
    push OFFSET $SG4040 ; '%lf'
    call DWORD PTR __imp__scanf
    add esp, 12 ; ▽
    cmp eax, 1
    je SHORT $LN2@main
    push OFFSET $SG4041 ; 'Error while parsing\n        your input'
    call esi ; call printf()
    add esp, 4
    push 0
    call DWORD PTR __imp__exit
$LN2@main:
…but MSVC 2012 uses SIMD instructions instead:

Listing 35.3: Optimizing MSVC 2010 x86

```
 fld  QWORD PTR _fahr$[esp+12]
 fsub QWORD PTR __real@4040000000000000 ; 32
 fmul QWORD PTR __real@4014000000000000 ; 5
 fdiv QWORD PTR __real@4022000000000000 ; 9
 fld QWORD PTR __real@c0711000000000000 ; -273
 fcomp ST(1)
 fnstsw ax
 test ah, 65
 ; 00000041H
 jne SHORT $LN1@main
 push OFFSET $SG4043 ; 'Error: incorrect 
 temperature!'
 fstp ST(0)
 call esi ; call printf()
 add esp, 4
 push 0
 call DWORD PTR __imp__exit

$LN1@main:
 sub esp, 8
 fstp QWORD PTR [esp]
 push OFFSET $SG4044 ; 'Celsius: %lf'
 call esi
 add esp, 12
 ; 0000000cH
 ; return 0 - by C99 standard
 xor eax, eax
 pop esi
 add esp, 8
 ret 0

$LN10@main:
 _main ENDP
```

Listing 35.3: Optimizing MSVC 2010 x86

```
$SG4228 DB 'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB '%lf', 00H
$SG4231 DB 'Error while parsing your input', 0aH, 00H
$SG4233 DB 'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB 'Celsius: %lf', 0aH, 00H
__real@c0711000000000000 DQ 0c0711000000000000r ; -273
__real@4040000000000000 DQ 04040000000000000000r ; 32
__real@4022000000000000 DQ 04022000000000000000r ; 9
__real@4014000000000000 DQ 04014000000000000000r ; 5
_fahr$ = -8
size
```
_main PROC
sub esp, 8
push esi
mov esi, DWORD PTR __imp__printf
push OFFSET $SG4228 ; 'Enter temperature in
\ Fahrenheit:'
call esi ; call printf()
lea eax, DWORD PTR _fahr$[esp+16]
push eax
push OFFSET $SG4230 ; '%lf'
call DWORD PTR __imp__scanf
add esp, 12 ;
\ 0000000cH
cmp eax, 1
je SHORT $LN2@main
push OFFSET $SG4231 ; 'Error while parsing
\ your input'
call esi ; call printf()
add esp, 4
push 0
call DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
movsd xmm1, QWORD PTR _fahr$[esp+12]
subsd xmm1, QWORD PTR __real@4040000000000000 ; 32
movsd xmm0, QWORD PTR __real@c0711000000000000 ; -273
mulsd xmm1, QWORD PTR __real@4014000000000000 ; 5
divsd xmm1, QWORD PTR __real@4022000000000000 ; 9
comisd xmm0, xmm1
jbe SHORT $LN1@main
push OFFSET $SG4233 ; 'Error: incorrect
\ temperature!'
call esi ; call printf()
add esp, 4
push 0
call DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
sub esp, 8
movsd QWORD PTR [esp], xmm1
push OFFSET $SG4234 ; 'Celsius: %lf'
call esi ; call printf()
add esp, 12 ;
\ 0000000cH
; return 0 - by C99 standard
xor eax, eax
pop esi
Of course, SIMD instructions are available in x86 mode, including those working with floating point numbers. It's somewhat easier to use them for calculations, so the new Microsoft compiler uses them.

We can also see that the $-273$ value is loaded into XMM0 register too early. And that's OK, because the compiler may emit instructions not in the order they are in the source code.
Chapter 36

Fibonacci numbers

Another widespread example used in programming textbooks is a recursive function that generates the Fibonacci numbers\(^1\). The sequence is very simple: each consecutive number is the sum of the previous two. The first two numbers are 1’s or 0, 1 and 1.

The sequence starts like this:

\[
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, \ldots
\]

### 36.1 Example #1

The implementation is simple. This program generates the sequence until 21.

```c
#include <stdio.h>

void fib (int a, int b, int limit)
{
    printf ("%d\n", a+b);
    if (a+b > limit)
        return;
    fib (b, a+b, limit);
};

int main()
{
    printf ("0\n1\n1\n" );
    fib (1, 1, 20);
}
```

\(^1\)http://go.yurichev.com/17332
Listing 36.1: MSVC 2010 x86

```assembly
;a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_limit$ = 16 ; size = 4
_fib PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    add eax, DWORD PTR _b$[ebp]
    push eax
    push OFFSET $SG2643
    call DWORD PTR __imp__printf
    add esp, 8
    mov ecx, DWORD PTR _a$[ebp]
    add ecx, DWORD PTR _b$[ebp]
    cmp ecx, DWORD PTR _limit$[ebp]
    jle SHORT $LN1@fib
    jmp SHORT $LN2@fib
$LN1@fib:
    mov edx, DWORD PTR _limit$[ebp]
    push edx
    mov eax, DWORD PTR _a$[ebp]
    add eax, DWORD PTR _b$[ebp]
    push eax
    mov ecx, DWORD PTR _b$[ebp]
    push ecx
    call _fib
    add esp, 12
$LN2@fib:
    pop ebp
    ret 0
_fib ENDP

_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2647
    call DWORD PTR __imp__printf
    add esp, 4
    push 20
    push 1
    push 1
    call _fib
    add esp, 12
    xor eax, eax
```

};
We will illustrate the stack frames with this.
Let’s load the example in OllyDbg and trace to the last call of \( f() \):

![OllyDbg screenshot](image-url)

Figure 36.1: OllyDbg: last call of \( f() \)
Let's investigate the stack more closely. Comments were added by the author of this book:

```
0035F940 00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F944 00000008 1st argument: a
0035F948 0000000D 2nd argument: b
0035F94C 00000014 3rd argument: limit
0035F950 /0035F964 saved EBP register
0035F954 |00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F958 |00000005 1st argument: a
0035F95C |00000008 2nd argument: b
0035F960 |00000014 3rd argument: limit
0035F964 }0035F978 saved EBP register
0035F968 }00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F96C }00000003 1st argument: a
0035F970 }00000005 2nd argument: b
0035F974 }00000014 3rd argument: limit
0035F978 }0035F98C saved EBP register
0035F97C }00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F980 }00000002 1st argument: a
0035F984 }00000003 2nd argument: b
0035F988 }00000014 3rd argument: limit
0035F98C }0035F9A0 saved EBP register
0035F990 }00FD1039 RETURN to fib.00FD1039 from fib.00FD1000
0035F994 }00000001 1st argument: a
0035F998 }00000002 2nd argument: b
0035F99C }00000014 3rd argument: limit
0035F9A0 }0035F9B4 saved EBP register
0035F9A4 }00FD105C RETURN to fib.00FD105C from fib.00FD1000
0035F9A8 }00000001 1st argument: a \n0035F9AC }00000001 2nd argument: b | prepared in main() for f1()
0035F9B0 |00000014 3rd argument: limit /
0035F9B4 |0035F9F8 saved EBP register
0035F9B8 |00FD11D0 RETURN to fib.00FD11D0 from fib.00FD1040
0035F9BC |00000001 main() 1st argument: argc \n0035F9C0 |006812C8 main() 2nd argument: argv | prepared in CRT for main()
0035F9C4 |00682940 main() 3rd argument: envp /
```

The function is recursive, hence stack looks like a “sandwich”. We see that the `limit` argument is always the same (0x14 or 20), but the `a` and `b` arguments are different for each call. There are also the RA-s and the saved EBP values. OllyDbg is able to determine the EBP-based frames, so it draws these brackets. The values

---

2 By the way, it’s possible to select several entries in OllyDbg and copy them to the clipboard (Ctrl-C). That’s what was done for this example.

3 i.e., it calls itself
inside each bracket make the stack frame, in other words, the stack area which each function incarnation uses as scratch space. We can also say that each function incarnation must not access stack elements beyond the boundaries of its frame (excluding function arguments), although it's technically possible. It's usually true, unless the function has bugs. Each saved EBP value is the address of the previous stack frame: this is the reason why some debuggers can easily divide the stack in frames and dump each function's arguments.

As we see here, each function incarnation prepares the arguments for the next function call.

At the end we see the 3 arguments for main(). argc is 1 (yes, indeed, we have ran the program without command-line arguments).

This easily to lead to a stack overflow: just remove (or comment out) the limit check and it will crash with exception 0xC00000FD (stack overflow.)

### 36.2 Example #2

My function has some redundancy, so let's add a new local variable next and replace all “a+b” with it:

```c
#include <stdio.h>

void fib (int a, int b, int limit)
{
    int next = a + b;
    printf ("%d\n", next);
    if (next > limit)
        return;
    fib (b, next, limit);
}

int main()
{
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
}
```

This is the output of non-optimizing MSVC, so the next variable is actually allocated in the local stack:

Listing 36.2: MSVC 2010 x86

```
_next$ = -4 ; size = 4
_a$ = 8    ; size = 4
```
_b$ = 12  ; size = 4
_limit$ = 16 ; size = 4
_fib PROC
    push ebp
    mov ebp, esp
    push ecx
    mov eax, DWORD PTR _a$[ebp]
    add eax, DWORD PTR _b$[ebp]
    mov DWORD PTR _next$[ebp], eax
    mov ecx, DWORD PTR _next$[ebp]
    push ecx
    push OFFSET $SG2751 ; '%d'
    call DWORD PTR __imp__printf
    add esp, 8
    mov edx, DWORD PTR _next$[ebp]
    cmp edx, DWORD PTR _limit$[ebp]
    jle SHORT $LN1@fib
    jmp SHORT $LN2@fib
$LN1@fib:
    mov eax, DWORD PTR _limit$[ebp]
    push eax
    mov ecx, DWORD PTR _next$[ebp]
    push ecx
    mov edx, DWORD PTR _b$[ebp]
    push edx
    call _fib
    add esp, 12
$LN2@fib:
    mov esp, ebp
    pop ebp
    ret 0
_fib ENDP

_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2753 ; "0\n1\n1\n"
    call DWORD PTR __imp__printf
    add esp, 4
    push 20
    push 1
    push 1
    call _fib
    add esp, 12
    xor eax, eax
    pop ebp
    ret 0
Let’s load it in OllyDbg once again:

Figure 36.2: OllyDbg: last call of f( )

Now the next variable is present in each frame.
Let's investigate the stack more closely. The author has again added his comments:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0029FC14</td>
<td>00E0103A</td>
<td>RETURN to fib2.00E0103A from fib2.00E01000</td>
</tr>
<tr>
<td>0029FC18</td>
<td>000000008</td>
<td>1st argument: a</td>
</tr>
<tr>
<td>0029FC1C</td>
<td>00000000D</td>
<td>2nd argument: b</td>
</tr>
<tr>
<td>0029FC20</td>
<td>000000014</td>
<td>3rd argument: limit</td>
</tr>
<tr>
<td>0029FC24</td>
<td>0000000D</td>
<td>&quot;next&quot; variable</td>
</tr>
<tr>
<td>0029FC28</td>
<td>/0029FC40</td>
<td>saved EBP register</td>
</tr>
<tr>
<td>0029FC2C</td>
<td>00E0103A</td>
<td>RETURN to fib2.00E0103A from fib2.00E01000</td>
</tr>
<tr>
<td>0029FC30</td>
<td>000000005</td>
<td>1st argument: a</td>
</tr>
<tr>
<td>0029FC34</td>
<td>000000008</td>
<td>2nd argument: b</td>
</tr>
<tr>
<td>0029FC38</td>
<td>000000014</td>
<td>3rd argument: limit</td>
</tr>
<tr>
<td>0029FC3C</td>
<td>00000008</td>
<td>&quot;next&quot; variable</td>
</tr>
<tr>
<td>0029FC40</td>
<td>]0029FC58</td>
<td>saved EBP register</td>
</tr>
<tr>
<td>0029FC44</td>
<td>00E0103A</td>
<td>RETURN to fib2.00E0103A from fib2.00E01000</td>
</tr>
<tr>
<td>0029FC48</td>
<td>000000003</td>
<td>1st argument: a</td>
</tr>
<tr>
<td>0029FC4C</td>
<td>000000005</td>
<td>2nd argument: b</td>
</tr>
<tr>
<td>0029FC50</td>
<td>000000014</td>
<td>3rd argument: limit</td>
</tr>
<tr>
<td>0029FC54</td>
<td>00000005</td>
<td>&quot;next&quot; variable</td>
</tr>
<tr>
<td>0029FC58</td>
<td>]0029FC70</td>
<td>saved EBP register</td>
</tr>
<tr>
<td>0029FC5C</td>
<td>00E0103A</td>
<td>RETURN to fib2.00E0103A from fib2.00E01000</td>
</tr>
<tr>
<td>0029FC60</td>
<td>000000002</td>
<td>1st argument: a</td>
</tr>
<tr>
<td>0029FC64</td>
<td>000000003</td>
<td>2nd argument: b</td>
</tr>
<tr>
<td>0029FC68</td>
<td>000000014</td>
<td>3rd argument: limit</td>
</tr>
<tr>
<td>0029FC6C</td>
<td>00000003</td>
<td>&quot;next&quot; variable</td>
</tr>
<tr>
<td>0029FC70</td>
<td>]0029FC88</td>
<td>saved EBP register</td>
</tr>
<tr>
<td>0029FC74</td>
<td>00E0103A</td>
<td>RETURN to fib2.00E0103A from fib2.00E01000</td>
</tr>
<tr>
<td>0029FC78</td>
<td>00000001</td>
<td>1st argument: a</td>
</tr>
<tr>
<td>0029FC7C</td>
<td>00000002</td>
<td>2nd argument: b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0029FC80</td>
<td>000000014</td>
<td>3rd argument: limit</td>
</tr>
<tr>
<td>0029FC84</td>
<td>00000002</td>
<td>&quot;next&quot; variable</td>
</tr>
<tr>
<td>0029FC88</td>
<td>]0029FC9C</td>
<td>saved EBP register</td>
</tr>
<tr>
<td>0029FC8C</td>
<td>00E0106C</td>
<td>RETURN to fib2.00E0106C from fib2.00E01000</td>
</tr>
<tr>
<td>0029FC90</td>
<td>000000001</td>
<td>1st argument: a</td>
</tr>
<tr>
<td>0029FC94</td>
<td>00000001</td>
<td>2nd argument: b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0029FC98</td>
<td>000000014</td>
<td>3rd argument: limit</td>
</tr>
<tr>
<td>0029FC9C</td>
<td>]0029FCE0</td>
<td>saved EBP register</td>
</tr>
<tr>
<td>0029FCA0</td>
<td>00E011E0</td>
<td>RETURN to fib2.00E011E0 from fib2.00E01050</td>
</tr>
<tr>
<td>0029FCA4</td>
<td>00000001</td>
<td>main() 1st argument: argc</td>
</tr>
<tr>
<td>0029FCA8</td>
<td>000812C8</td>
<td>main() 2nd argument: argv</td>
</tr>
</tbody>
</table>

Here we see it: the next value is calculated in each function incarnation, then passed as argument b to the next incarnation.
36.3 Summary

Recursive functions are aesthetically nice, but technically may degrade performance because of their heavy stack usage. Everyone who writes performance critical code probably should avoid recursion.

For example, the author of this book once wrote a function to seek a particular node in a binary tree. As a recursive function it looked quite stylish but since additional time was spent at each function call for the prologue/epilogue, it was working a couple of times slower than an iterative (recursion-free) implementation.

By the way, that is the reason that some functional PL$^4$ compilers (where recursion is used heavily) use tail call.

---

$^4$LISP, Python, Lua, etc.
Chapter 37

CRC32 calculation example

This is a very popular table-based CRC32 hash calculation technique¹.

```c
/* By Bob Jenkins, (c) 2006, Public Domain */
#include <stdio.h>
#include <stddef.h>
#include <string.h>
typedef unsigned long ub4;
typedef unsigned char ub1;

static const ub4 crctab[256] = {
  0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0,
  0xc76772f5, 0x9009820b, 0xe30eb317, 0x8e0ef34d, 0x0966c459, 0,
  0xf663f50f, 0x7807c625, 0xe101f7b1, 0x9b0bec47, 0x0f6ffed3, 0,
  0x7f63d219, 0xe808033f, 0x9101044b, 0x880835fd, 0x01676480, 0,
  0xe0609516, 0x770766b2, 0xe906b768, 0x9601c81e, 0x0668f9a4, 0,
  0xe765aeb0, 0x7005d97c, 0xe202ebec, 0x9505bba2, 0x0f6a8c38, 0,
  0xe6671d84, 0x73034e1a, 0xe5047f00, 0x9203aeb6, 0x0161fde2, 0,
  0x7868dc78, 0xe363aefc, 0x94049f5a, 0x8303cf26, 0x0c64d80c, 0,
  0xe1618912, 0x7606b880, 0xe8018976, 0x9f0618f2, 0x0765eae8, 0,
  0xe067ba74, 0x7f008ddc, 0xe9091e6a, 0x960e4ef0, 0x066bfbf6, 0,
  0xe76e5ebc, 0x7001aeb2, 0xe20f3d78, 0x9b020dc4, 0x0f6d9c60, 0,
  0xe6646d5c, 0x7908fdd8, 0xe1036c44, 0x98073e70, 0x01668f06, 0,
  0xe861d69c, 0x70059518, 0xe5086f84, 0x920f0f7e, 0x0663d04a, 0,
  0xe06480da, 0x7701e13c, 0xe3067188, 0x940ffcc4, 0x036ac250, 0,
  0xe26d23f6, 0x750d3272, 0xe50cd368, 0x970deeb4, 0x026b825a, 0,
  0xe76052c0, 0x720903f6, 0xe000f462, 0x910b04f8, 0x0764aa84, 0,
  0xe66d3414, 0x7b026550, 0xe90356c6, 0x90003652, 0x00679738, 0,
  0xe3632710, 0x7d0a1836, 0xe40ff922, 0x930cf978, 0x0969b00e, 0,
  0xe16e8b00, 0x7a0ddaf4, 0xe6041b6a, 0x9f034b50, 0x00669aad, 0,
  0xe565aeb6, 0x7c05dab2, 0xe2026b38, 0x97015b24, 0x0a6002e0, 0,
  0xe869337c, 0x7f0af428, 0xe0052494, 0x93001488, 0x0f64a3f4, 0,
  0xe767d4a0, 0x700f853c, 0xe906b6c8, 0x96014774, 0x066816e0, 0,
  0xe167877a, 0x7505d830, 0xe302ae2c, 0x9405d8af, 0x0f6a8155, 0,
  0xe6635241, 0x790083d7, 0xe807b433, 0x9f0044d9, 0x006b8435, 0,
  0xe26d250b, 0x7e0af69f, 0xe5011705, 0x9206679b, 0x07643671, 0,
  0xe06806ed, 0x730d9777, 0xe70388f3, 0x90089869, 0x0161f9b5, 0,
};
```

¹The source code was taken from here: [http://go.yurichev.com/17327](http://go.yurichev.com/17327)
0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0
\x56b3c423,
0xcfba9599, 0xb8bda50f, 0xe88bd433,
0xfcb9887c, 0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0
\x56b3c423,
0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x76dc4190, 0
\x01db7106,
0x91646c97, 0xe635c01, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0
\x806d3d2d,
0x56b3c423,
/* how to derive the values in crctab[] from polynomial 0
\texttt{xeddb88320} */

void build_table()
{
ub4 i, j;
for (i=0; i<256; ++i) {
    j = i;
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    printf("0x%.8lx, ", j);
    if (i%6 == 5) printf("\n") ;
}

/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
We are interesting in the `crc()` function only. By the way, pay attention to the two loop initializers in the `for()` statement: `hash=len, i=0`. The C/C++ standard allows this, of course. The emitted code will contain two operations in the loop initialization part instead of one.

Let's compile it in MSVC with optimization `/Ox`. For the sake of brevity, only the `crc()` function is listed here, with my comments.

```
    _key$ = 8       ; size = 4
    _len$ = 12      ; size = 4
    _hash$ = 16     ; size = 4
    _crc PROC
    mov edx, DWORD PTR _len$[esp-4]
    xor ecx, ecx ; i will be stored in ECX
    mov eax, edx
    test edx, edx
    jbe SHORT $LN1@crc
    push ebx
    push esi
    mov esi, DWORD PTR _key$[esp+4] ; ESI = key
    push edi

    $LL3@crc:
    ; work with bytes using only 32-bit registers. byte from address key+i we store into EDI
    movzx edi, BYTE PTR [ecx+esi]
    mov ebx, eax ; EBX = (hash = len)
    and ebx, 255 ; EBX = hash & 0xff
    ; XOR EDI, EBX (EDI=EDI^EBX) - this operation uses all 32 bits of each register

    _crc ENDP
```
; but other bits (8-31) are cleared all time, so it's OK'
; these are cleared because, as for EDI, it was done by MOVZX
; high bits of EBX was cleared by AND EBX, 255 instruction above
(255 = 0xff)

xor edi, ebx

; EAX=EAX>>8; bits 24-31 taken "from nowhere" will be cleared
shr eax, 8

; EAX=EAX^crctab[EDI*4] - choose EDI-th element from crctab[]
table
xor eax, DWORD PTR _crctab[edi*4]
inc ecx ; i++
cmp ecx, edx ; i<len ?
jb SHORT $LL3@crc ; yes
pop edi
pop esi
pop ebx

$LN1@crc:
    ret 0
_crc ENDP

Let's try the same in GCC 4.4.1 with -O3 option:

public crc
crc proc near

key = dword ptr 8
hash = dword ptr 0Ch

push ebp
xor edx, edx
mov ebp, esp
push esi
mov esi, [ebp+key]
push esi
mov esi, [ebp+hash]
push ebp
mov ebx, [ebp+hash]
test ebx, ebx
mov eax, ebx
jz short loc_80484D3
nop ; padding
lea esi, [esi+0] ; padding; works as NOP
(ESI does not changing here)

loc_80484B8:
    mov ecx, eax ; save previous state of hash to ECX
GCC has aligned the loop start on a 8-byte boundary by adding NOP and lea esi, [esi+0] (that is an idle operation too). Read more about it in npad section (88 on page 1256).
As we know, a TCP/IP address (IPv4) consists of four numbers in the 0...255 range, i.e., four bytes. Four bytes can be fit in a 32-bit variable easily, so an IPv4 host address, network mask or network address can all be 32-bit integers.

From the user’s point of view, the network mask is defined as four numbers and is formatted like 255.255.255.0 or so, but network engineers (sysadmins) use a more compact notation (CIDR\(^1\), like /8, /16 or similar. This notation just defines the number of bits the mask has, starting at the MSB.

\(^{1}\text{Classless Inter-Domain Routing}\)
CHAPTER 38. NETWORK ADDRESS CALCULATION EXAMPLE

Mask
/30
/29
/28
/27
/26
/24
/23
/22
/21
/20
/19
/18
/17
/16
/8

Hosts
4
8
16
32
64
256
512
1024
2048
4096
8192
16384
32768
65536
16777216

Usable
2
6
14
30
62
254
510
1022
2046
4094
8190
16382
32766
65534
16777214

CHAPTER 38. NETWORK ADDRESS CALCULATION EXAMPLE

Netmask
255.255.255.252
255.255.255.248
255.255.255.240
255.255.255.224
255.255.255.192
255.255.255.0
255.255.254.0
255.255.252.0
255.255.248.0
255.255.240.0
255.255.224.0
255.255.192.0
255.255.128.0
255.255.0.0
255.0.0.0

Hex mask
fffffffc
fffffff8
fffffff0
ffffffe0
ffffffc0
ffffff00
fffffe00
fffffc00
fffff800
fffff000
ffffe000
ffffc000
ffff8000
ffff0000
ff000000

class C network

class B network
class A network

Here is a small example, which calculates the network address by applying the
network mask to the host address.
#include <stdio.h>
#include <stdint.h>
uint32_t form_IP (uint8_t ip1, uint8_t ip2, uint8_t ip3, ⤦
Ç uint8_t ip4)
{
return (ip1<<24) | (ip2<<16) | (ip3<<8) | ip4;
};
void print_as_IP (uint32_t a)
{
printf ("%d.%d.%d.%d\n",
(a>>24)&0xFF,
(a>>16)&0xFF,
(a>>8)&0xFF,
(a)&0xFF);
};
// bit=31..0
uint32_t set_bit (uint32_t input, int bit)
{
return input=input|(1<<bit);
};
uint32_t form_netmask (uint8_t netmask_bits)
{

654


CHAPTER 38. NETWORK ADDRESS CALCULATION EXAMPLE

```c
uint32_t netmask=0;
uint8_t i;

for (i=0; i<netmask_bits; i++)
    netmask=set_bit(netmask, 31-i);

return netmask;
};

void calc_network_address (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4, uint8_t netmask_bits)
{
    uint32_t netmask=form_netmask(netmask_bits);
    uint32_t ip=form_IP(ip1, ip2, ip3, ip4);
    uint32_t netw_adr;

    printf("netmask=");
    print_as_IP (netmask);

    netw_adr=ip&netmask;

    printf("network address=");
    print_as_IP (netw_adr);
};

int main()
{
    calc_network_address (10, 1, 2, 4, 24); // 10.1.2.4, /24
    calc_network_address (10, 1, 2, 4, 8); // 10.1.2.4, /8
    calc_network_address (10, 1, 2, 4, 25); // 10.1.2.4, /25
    calc_network_address (10, 1, 2, 64, 26); // 10.1.2.4, /26
};
```

### 38.1 calc_network_address()

calc_network_address() function is simplest one: it just ANDs the host address with the network mask, resulting in the network address.

Listing 38.1: Optimizing MSVC 2012 /Ob0

```c
_ip1$ = 8 ; size = 1
```
At line 22 we see the most important AND—here the network address is calculated.

### 38.2 form_IP()

The `form_IP()` function just puts all 4 bytes into a 32-bit value.

Here is how it is usually done:

- Allocate a variable for the return value. Set it to 0.
- Take the fourth (lowest) byte, apply OR operation to this byte and return the value. The return value contain the 4th byte now.
- Take the third byte, shift it left by 8 bits. You’ll get a value like 0x0000bb00 where bb is your third byte. Apply the OR operation to the resulting value and it. The return value has contained 0x000000aa so far, so ORing the values will produce a value like 0x0000bbaa.
• Take the second byte, shift it left by 16 bits. You’ll get a value like 0x00cc0000, where cc is your second byte. Apply the OR operation to the resulting value and return it. The return value has contained 0x0000bbbaa so far, so ORing the values will produce a value like 0x00ccbbbaa.

• Take the first byte, shift it left by 24 bits. You’ll get a value like 0xdd000000, where dd is your first byte. Apply the OR operation to the resulting value and return it. The return value contain 0x00ccbbbaa so far, so ORing the values will produce a value like 0xddccbbbaa.

And this is how it’s done by non-optimizing MSVC 2012:

Listing 38.2: Non-optimizing MSVC 2012

```plaintext
; denote ip1 as "dd", ip2 as "cc", ip3 as "bb", ip4 as "aa".
_ip1$ = 8 ; size = 1
_ip2$ = 12 ; size = 1
_ip3$ = 16 ; size = 1
_ip4$ = 20 ; size = 1
_form_IP PROC
    push ebp
    mov ebp, esp
    movzx eax, BYTE PTR _ip1$[ebp]
    ; EAX=000000dd
    shl eax, 24
    ; EAX=dd000000
    movzx ecx, BYTE PTR _ip2$[ebp]
    ; ECX=000000cc
    shl ecx, 16
    ; ECX=00cc0000
    or eax, ecx
    ; EAX=ddcc0000
    movzx edx, BYTE PTR _ip3$[ebp]
    ; EDX=000000bb
    shl edx, 8
    ; EDX=0000bb00
    or eax, edx
    ; EAX=ddccbb00
    movzx ecx, BYTE PTR _ip4$[ebp]
    ; ECX=000000aa
    or eax, ecx
    ; EAX=ddccbbaa
    pop ebp
    ret 0
_form_IP ENDP
```

Well, the order is different, but, of course, the order of the operations doesn’t matters.
Optimizing MSVC 2012 does essentially the same, but in a different way:

Listing 38.3: Optimizing MSVC 2012 /Ob0

```assembly
; denote ip1 as "dd", ip2 as "cc", ip3 as "bb", ip4 as "aa".
_ip1$ = 8 ; size = 1
_ip2$ = 12 ; size = 1
_ip3$ = 16 ; size = 1
_ip4$ = 20 ; size = 1
_form_IP PROC
    movzx eax, BYTE PTR _ip1$[esp-4]
    ; EAX=000000dd
    movzx ecx, BYTE PTR _ip2$[esp-4]
    ; ECX=000000cc
    shl eax, 8
    ; EAX=0000dd00
    or eax, ecx
    ; EAX=0000ddcc
    movzx ecx, BYTE PTR _ip3$[esp-4]
    ; ECX=000000bb
    shl eax, 8
    ; EAX=0000ddcc00
    or eax, ecx
    ; EAX=0000ddccb00
    movzx ecx, BYTE PTR _ip4$[esp-4]
    ; ECX=000000aa
    shl eax, 8
    ; EAX=0000ddccb00
    or eax, ecx
    ; EAX=0000ddccb00
    ret 0
_form_IP ENDP
```

We could say that each byte is written to the lowest 8 bits of the return value, and then the return value is shifted left by one byte at each step. Repeat 4 times for each input byte.

That's it! Unfortunately, there are probably no other ways to do it. There are no popular CPUs or ISAs which has instruction for composing a value from bits or bytes. It's all usually done by bit shifting and ORing.

### 38.3 print_as_IP()

print_as_IP() does the inverse: splitting a 32-bit value into 4 bytes.
Slicing works somewhat simpler: just shift input value by 24, 16, 8 or 0 bits, take the bits from zeroth to seventh (lowest byte), and that’s it:

Listing 38.4: Non-optimizing MSVC 2012

```assembly
.a$ = 8 ; size = 4
._print_as_IP PROC
  push ebp
  mov ebp, esp
  mov eax, DWORD PTR _a$[ebp]
  ; EAX=ddccbbaa
  and eax, 255
  ; EAX=000000aa
  push eax
  mov ecx, DWORD PTR _a$[ebp]
  ; ECX=ddccbbaa
  shr ecx, 8
  ; ECX=00ddccbb
  and ecx, 255
  ; ECX=000000bb
  push ecx
  mov edx, DWORD PTR _a$[ebp]
  ; EDX=ddccbbaa
  shr edx, 16
  ; EDX=0000ddcc
  and edx, 255
  ; EDX=000000cc
  push edx
  mov eax, DWORD PTR _a$[ebp]
  ; EAX=ddccbbaa
  shr eax, 24
  ; EAX=000000dd
  and eax, 255 ; probably redundant instruction
  ; EAX=000000dd
  push eax
  push OFFSET $SG2973 ; '%d.%d.%d.%d'
call DWORD PTR __imp__printf
  add esp, 20
  pop ebp
  ret 0
._print_as_IP ENDP
```

Optimizing MSVC 2012 does almost the same, but without unnecessary reloading of the input value:

Listing 38.5: Optimizing MSVC 2012 /Ob0

```assembly
.a$ = 8 ; size = 4
._print_as_IP PROC
```
38.4 form_netmask() and set_bit()

form_netmask() makes a network mask value from CIDR notation. Of course, it would be much effective to use here some kind of a precalculated table, but we consider it in this way intentionally, to demonstrate bit shifts. We will also write a separate function set_bit(). It’s a not very good idea to create a function for such primitive operation, but it would be easy to understand how it all works.

Listing 38.6: Optimizing MSVC 2012 /Ob0

```
_input$ = 8 ; size = 4
_bit$ = 12 ; size = 4
_set_bit PROC
  mov  ecx, DWORD PTR _bit$[esp-4]
  movzx eax, cl
  mov eax, ecx
  shr eax, 8
  and eax, 255
  push eax
  mov eax, ecx
  shr eax, 16
  and eax, 255
  push eax
  shr ecx, 24
  push ecx
  push OFFSET $SG3020 ; '%d.%d.%d.%d'
call DWORD PTR __imp__printf
  add esp, 20
  ret 0
_set_bit ENDP
```
set_bit() is primitive: it just shift left 1 to number of bits we need and then ORs it with the “input” value. form_netmask() has a loop: it will set as many bits (starting from the MSB) as passed in the netmask_bits argument

38.5 Summary

That’s it! We run it and getting:

```
netmask=255.255.255.0
network address=10.1.2.0
netmask=255.0.0.0
network address=10.0.0.0
```
<table>
<thead>
<tr>
<th>netmask</th>
<th>network address</th>
</tr>
</thead>
<tbody>
<tr>
<td>255.255.255.128</td>
<td>10.1.2.0</td>
</tr>
<tr>
<td>255.255.255.192</td>
<td>10.1.2.64</td>
</tr>
</tbody>
</table>
Chapter 39

Loops: several iterators

In most cases loops have only one iterator, but there could be several in the resulting code.

Here is a very simple example:

```c
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;

    // copy from one array to another in some weird scheme
    for (i=0; i<cnt; i++)
        a1[i*3]=a2[i*7];
}
```

There are two multiplications at each iteration and they are costly operations. Can we optimize it somehow? Yes, if we notice that both array indices are jumping on values that we can easily calculate without multiplication.

### 39.1 Three iterators

Listing 39.1: Optimizing MSVC 2013 x64

```
f PROC
    ; RDX=a1
    ; RCX=a2
    ; R8=cnt
```
CHAPTER 39. LOOPS: SEVERAL ITERATORS

Now there are 3 iterators: the cnt variable and two indices, which are increased by 12 and 28 at each iteration. We can rewrite this code in C/C++:

```c
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;
    size_t idx1=0; idx2=0;

    // copy from one array to another in some weird scheme
    for (i=0; i<cnt; i++)
    {
        a1[idx1]=a2[idx2];
        idx1+=3;
        idx2+=7;
    }
}
```

So, at the cost of updating 3 iterators at each iteration instead of one, we can remove two multiplication operations.

### 39.2 Two iterators

GCC 4.9 does even more, leaving only 2 iterators:

```c
Listing 39.2: Optimizing GCC 4.9 x64

; RDI=a1
; RSI=a2
; RDX=cnt
f:
```
test  rdx, rdx  ; cnt==0? exit then
je   .L1
; calculate last element address in "a2" and leave it in RDX
lea  rax, [0+rdx*4]
;  RAX=RDX*4=cnt*4
sal  rdx, 5
;  RDX=RDX<<5=cnt*32
sub  rdx, rax
;  RDX=RDX-RAX=cnt*32-cnt*4=cnt*28
add  rdx, rsi
;  RDX=RDX+RSI=a2+cnt*28
.L3:
  mov  eax, DWORD PTR [rsi]
  add  rsi, 28
  add  rdi, 12
  mov  DWORD PTR [rdi-12], eax
  cmp  rsi, rdx
  jne  .L3
.L1:
  rep ret

There is no counter variable any more: GCC concluded that it is not needed. The last element of the a2 array is calculated before the loop begins (which is easy: \(cnt \times 7\)) and that's how the loop is to be stopped: just iterate until the second index has not reached this precalculated value.

You can read more about multiplication using shifts/additions/subtractions here: 16.1.3 on page 291.

This code can be rewritten into C/C++ like that:

```c
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;
    size_t idx1=0, idx2=0;
    size_t last_idx2=cnt*7;

    // copy from one array to another in some weird scheme
    for (; ;)
    {
        a1[idx1]=a2[idx2];
        idx1+=3;
        idx2+=7;
        if (idx2==last_idx2)
            break;
    }
}
```
GCC (Linaro) 4.9 for ARM64 does the same, but it precalculates the last index of \texttt{a1} instead of \texttt{a2}, which, of course has the same effect:

Listing 39.3: Optimizing GCC (Linaro) 4.9 ARM64

```assembly
; X0=a1
; X1=a2
; X2=cnt
f:
    cbz x2, .L1 ; cnt==0? exit then
    ; calculate last element of "a1" array
    add x2, x2, x2, lsl 1
    ; X2=X2+X2<<1=X2+X2*2=X2*3
    mov x3, 0
    lsl x2, x2, 2
    ; X2=X2<<2=X2*4=X2*3*4=X2*12
    .L3:
    ldr w4, [x1],28 ; load at X1, add 28 to X1
    (post-increment)
    str w4, [x0,x3] ; store at X0+X3=a1+X3
    add x3, x3, 12 ; shift X3
    cmp x3, x2 ; end?
    bne .L3
    .L1:
    ret
```

GCC 4.4.5 for MIPS does the same:

Listing 39.4: Optimizing GCC 4.4.5 for MIPS (IDA)

```assembly
; $a0=a1
; $a1=a2
; $a2=cnt
f:
    ; jump to loop check code:
    beqz $a2, locret_24
    ; initialize counter (i) at 0:
    move $v0, $zero ; branch delay slot, NOP
loc_8:
    ; load 32-bit word at $a1
    lw $a3, 0($a1)
    ; increment counter (i):
    addiu $v0, 1
    ; check for finish (compare "i" in $v0 and "cnt" in $a2):
    sltu $v1, $v0, $a2
    ; store 32-bit word at $a0:
```

...
39.3 Intel C++ 2011 case

Compiler optimizations can also be weird, but nevertheless, still correct. Here is what the Intel C++ compiler 2011 does:

**Listing 39.5: Optimizing Intel C++ 2011 x64**

```
f PROC
; parameter 1: rcx = a1
; parameter 2: rdx = a2
; parameter 3: r8 = cnt
.B1.1:: ; Preds .B1.0
    test r8, r8
    jbe exit ; Prob 50%
.B1.2:: ; Preds .B1.1
    cmp r8, 6
    jbe just_copy ; Prob 50%
.B1.3:: ; Preds .B1.2
    cmp rcx, rdx
    jbe .B1.5 ; Prob 50%
```

```
CHAPTER 39. LOOPS: SEVERAL ITERATORS

```
.B1.4:: ; Preds .B1.3
   mov   r10, r8
   mov   r9, rcx
   shl   r10, 5
   lea   rax, QWORD PTR [r8*4]
   mov   r9, rcx
   sub   r10, rax
   cmp   r9, r10
   jge   just_copy2 ; Prob 50%

.B1.5:: ; Preds .B1.3 .B1.4
   cmp   rdx, rcx
   jbe   just_copy ; Prob 50%

.B1.6:: ; Preds .B1.5
   mov   r9, rdx
   lea   rax, QWORD PTR [r8*8]
   mov   r9, rcx
   lea   r10, QWORD PTR [rax+r8*4]
   cmp   r9, r10
   jl    just_copy ; Prob 50%
```

; LOE rdx rcx rbx rbp rsi rdi
just_copy2:: ; Preds .B1.4 .B1.6
; R8 = cnt
; RDX = a2
; RCX = a1
    xor r10d, r10d
    ;8.2
    xor r9d, r9d
    ;
    xor eax, eax
    ;
    ; LOE rax rdx rcx rbx rbp rsi
    rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10
    xmm11 xmm12 xmm13 xmm14 xmm15
.B1.8:: ; Preds .B1.8 just_copy2
    mov r11d, DWORD PTR [rax+rdx]
    ;3.6
    inc r10
    ;8.2
    mov DWORD PTR [r9+rcx], r11d
    ;3.6
    add r9, 12
    ;8.2
    add rax, 28
    ;8.2
    cmp r10, r8
    ;8.2
    jb .B1.8 ; Prob 82%
    ;8.2
    jmp exit ; Prob 100%
    ;8.2
    ; LOE rax rdx rcx rbx rbp rsi
    rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10
    xmm11 xmm12 xmm13 xmm14 xmm15
just_copy:: ; Preds .B1.2 .B1.5 .B1.6
; R8 = cnt
; RDX = a2
; RCX = a1
    xor r10d, r10d
    ;8.2
    xor r9d, r9d
    ;
    xor eax, eax
    ;
    ; LOE rax rdx rcx rbx rbp rsi
    rdi r8 r9 r10 r12 r13 r14 r15 xmm6 xmm7 xmm8 xmm9 xmm10
    xmm11 xmm12 xmm13 xmm14 xmm15
.B1.11:: ; Preds .B1.11 just_copy
First, there are some decisions taken, then one of the routines is executed. Looks like it is a check if arrays intersect. This is very well known way of optimizing memory block copy routines. But copy routines are the same! This is probably an error of the Intel C++ optimizer, which still produces workable code, though.

We intentionally considering such example code in this book so the reader would understand that compiler output is weird at times, but still correct, because when the compiler was tested, is passed the tests.
Chapter 40

Duff’s device

Duff’s device\(^1\) is an unrolled loop with the possibility to jump inside it. The unrolled loop is implemented using a fallthrough `switch()` statement.

We would use here a slightly simplified version of Tom Duff’s original code.

Let’s say, we need to write a function that clears a region in memory. One can come with a simple loop, clearing byte by byte. It’s obviously slow, since all modern computers have much wider memory bus. So the better way is to clear the memory region using 4 or 8 byte blocks. Since we are going to work with a 64-bit example here, we are going to clear the memory in 8 byte blocks. So far so good. But what about the tail? Memory clearing routine can also be called for regions of size that’s not a multiple of 8.

So here is the algorithm:

- calculate the number of 8-byte blocks, clear them using 8-byte (64-bit) memory accesses;
- calculate the size of the tail, clear it using 1-byte memory accesses.

The second step can be implemented using a simple loop. But let’s implement it as an unrolled loop:

\[
\begin{align*}
\text{#include <stdint.h>} \\
\text{#include <stdio.h>} \\
\text{void bzero(uint8_t* dst, size_t count)} \\
\{ \\
\text{int i;} \\
\text{if (count&(~7))} \\
\end{align*}
\]

\(^1\text{wikipedia}\)
// work out 8-byte blocks
for (i=0; i<count>>3; i++)
{
    *(uint64_t*)dst=0;
    dst=dst+8;
}

// work out the tail
switch(count & 7)
{
    case 7: *dst++ = 0;
    case 6: *dst++ = 0;
    case 5: *dst++ = 0;
    case 4: *dst++ = 0;
    case 3: *dst++ = 0;
    case 2: *dst++ = 0;
    case 1: *dst++ = 0;
    case 0: // do nothing
        break;
}

Let's first understand how the calculation is done. The memory region size comes as a 64-bit value. And this value can be divided in two parts:

```
  7 6 5 4 3 2 1 0
  ... B B B B S S S
```

(“B” is number of 8-byte blocks and “S” is length of the tail in bytes).

When we divide the input memory region size by 8, the value is just shifted right by 3 bits. But to calculate the remainder, we just need to isolate the lowest 3 bits! So the number of 8-byte blocks is calculated as \( \text{count} >> 3 \) and remainder as \( \text{count} \& 7 \).

We also need to find out if we are going to execute the 8-byte procedure at all, so we need to check if the value of \( \text{count} \) is greater than 7. We do this by clearing the 3 lowest bits and comparing the resulting number with zero, because all we need here is to answer the question, is the high part of \( \text{count} \) non-zero.

Of course, this works because 8 is \( 2^3 \) and division by numbers that are \( 2^n \) is easy. It's not possible for other numbers.

It's actually hard to say if these hacks are worth using, because they lead to hard-to-read code. However, these tricks are very popular and a practicing programmer, even if he/she is not using them, nevertheless has to understand them.

So the first part is simple: get the number of 8-byte blocks and write 64-bit zero values to memory.
The second part is an unrolled loop implemented as fallthrough switch() statement. First, let’s express in plain English what we have to do here. We have to “write as many zero bytes in memory, as $count \& 7$ value tells us”. If it’s 0, jump to the end, there is no work to do. If it’s 1, jump to the place inside switch() statement where only one storage operation is to be executed. If it’s 2, jump to another place, where two storage operation are to be executed, etc. 7 as input value leads to the execution of all 7 operations. There is no 8, because a memory region of 8 bytes is to be processed by the first part of our function.

So we wrote an unrolled loop. It was definitely faster on older computers than normal loops (and conversely, modern CPUs works better for short loops than for unrolled ones). Maybe this is still meaningful on modern low-cost embedded MCU\(^2\)s.

Let’s see what the optimizing MSVC 2012 does:

```plaintext
dst$ = 8
count$ = 16
bzero PROC
    test rdx, -8
    je SHORT $LN11@bzero
    ; work out 8-byte blocks
    xor r10d, r10d
    mov r9, rdx
    shr r9, 3
    mov r8d, r10d
    test r9, r9
    je SHORT $LN11@bzero
    npad 5
$LL19@bzero:
    inc r8d
    mov QWORD PTR [rcx], r10
    add rcx, 8
    movsxd rax, r8d
    cmp rax, r9
    jb SHORT $LL19@bzero
$LN11@bzero:
    ; work out the tail
    and edx, 7
    dec rdx
    cmp rdx, 6
    ja SHORT $LN9@bzero
    lea r8, OFFSET FLAT:__ImageBase
    mov eax, DWORD PTR $LN22@bzero[r8+rdx*4]
    add rax, r8
    jmp rax
```

\(^2\) Microcontroller unit
The first part of the function is predictable. The second part is just an unrolled loop and a jump passing control flow to the correct instruction inside it. There is no other code between the MOV/INC instruction pairs, so the execution is to fall until the very end, executing as many pairs as needed.

By the way, we can observe that the MOV/INC pair consumes a fixed number of bytes (3+3). So the pair consumes 6 bytes. Knowing that, we can get rid of the switch() jumptable, we can just multiple the input value by 6 and jump to \( \text{current} \_ \text{RIP} + \text{input} \_ \text{value} \times 6 \). This can also be faster because we are not in need to fetch a value from the jumptable. It’s possible that 6 probably is not a very good constant for fast multiplication and maybe it’s not worth it, but you get the idea\(^3\). That is what old-school demomakers did in the past with unrolled

\(^3\)As an exercise, you can try to rework the code to get rid of the jumptable. The instruction pair can
be rewritten in a way that it will consume 4 bytes or maybe 8. 1 byte is also possible (using STOSB instruction).
Chapter 41

Division by 9

A very simple function:

```c
int f(int a)
{
    return a/9;
};
```

41.1 x86

...is compiled in a very predictable way:

Listing 41.1: MSVC

```assembly
_a$ = 8 ; size = 4
_f PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    cdq ; sign extend EAX to EDX:EAX
    mov ecx, 9
    idiv ecx
    pop ebp
    ret 0
_f ENDP
```

IDIV divides the 64-bit number stored in the EDX:EAX register pair by the value in the ECX. As a result, EAX will contain the quotient, and EDX—the remainder.
The result is returned from the \( f() \) function in the EAX register, so the value is not moved after the division operation, it is in right place already. Since IDIV uses the value in the EDX: EAX register pair, the CDQ instruction (before IDIV) extends the value in EAX to a 64-bit value taking its sign into account, just as MOVsx does. If we turn optimization on (/Ox), we get:

Listing 41.2: Optimizing MSVC

```
_a$ = 8  ; size = 4
_f PROC
    mov ecx, DWORD PTR _a$[esp-4]
    mov eax, 954437177 ; 38e38e39H
    imul ecx
    sar edx, 1
    mov eax, edx
    shr eax, 31 ; 0000001fH
    add eax, edx
    ret 0
_f ENDP
```

This is division by multiplication. Multiplication operations work much faster. And it is possible to use this trick \(^1\) to produce code which is effectively equivalent and faster.

This is also called “strength reduction” in compiler optimizations.

GCC 4.4.1 generates almost the same code even without additional optimization flags, just like MSVC with optimization turned on:

Listing 41.3: Non-optimizing GCC 4.4.1

```
public f
f proc near
arg_0 = dword ptr 8
    push ebp
    mov ebp, esp
    mov ecx, [ebp+arg_0]
    mov edx, 954437177 ; 38E38E39h
    mov eax, ecx
    imul edx
    sar edx, 1
    mov eax, edx
    shr eax, 1Fh
    add eax, edx
    ret 0
_f ENDP
```

\(^1\) Read more about division by multiplication in [War02, pp. 10-3]
41.2 ARM

The ARM processor, just like in any other “pure” RISC processor lacks an instruction for division. It also lacks a single instruction for multiplication by a 32-bit constant (recall that a 32-bit constant cannot fit into a 32-bit opcode). By taking advantage of this clever trick (or hack), it is possible to do division using only three instructions: addition, subtraction and bit shifts (19 on page 411).

Here is an example that divides a 32-bit number by 10, from [Ltd94, 3.3 Division by a Constant]. The output consists of the quotient and the remainder.

; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
    SUB    a2, a1, #10       ; keep (x-10) for later
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1   ; calc (x-10) - (x/10)*10
    ADDPL  a1, a1, #1         ; fix-up quotient
    ADDMI  a2, a2, #10        ; fix-up remainder
    MOV    pc, lr

41.2.1 Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

__text:00002C58 39 1E 08 E3 E3 18 43 E3  MOV    R1, 0x38E38E39
__text:00002C60 10 F1 50 E7  SMMUL  R0, R0, R1
__text:00002C64 C0 10 A0 E1  MOV    R1, R0,ASR#1
__text:00002C68 A0 0F 81 E0  ADD    R0, R1, R0,LSR#31
__text:00002C6C 1E FF 2F E1  BX     LR
This code is almost the same as the one generated by the optimizing MSVC and GCC. Apparently, LLVM uses the same algorithm for generating constants.

The observant reader may ask, how does MOV writes a 32-bit value in a register, when this is not possible in ARM mode. It is impossible indeed, but, as we see, there are 8 bytes per instruction instead of the standard 4, in fact, there are two instructions. The first instruction loads 0x8E39 into the low 16 bits of register and the second instruction is MOV, it loads 0x383E into the high 16 bits of the register. IDA is fully aware of such sequences, and for the sake of compactness reduces them to one single “pseudo-instruction”.

The SMMUL (Signed Most Significant Word Multiply) instruction two multiplies numbers, treating them as signed numbers and leaving the high 32-bit part of result in the R0 register, dropping the low 32-bit part of the result.

The “MOV R1, R0, ASR#1” instruction is an arithmetic shift right by one bit.

“ADD R0, R1, R0, LSR#31” is $R0 = R1 + R0 >> 31$

There is no separate shifting instruction in ARM mode. Instead, an instructions like (MOV, ADD, SUB, RSB)\(^2\) can have a suffix added, that says if the second operand must be shifted, and if yes, by what value and how. ASR stands for Arithmetic Shift Right, LSR—Logical Shift Right.

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

<table>
<thead>
<tr>
<th>instruction</th>
<th>R1</th>
<th>R0, 0x38E38E39</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMMUL.W</td>
<td>R0, R0, R1</td>
<td></td>
</tr>
<tr>
<td>ASRS</td>
<td>R1, R0, #1</td>
<td></td>
</tr>
<tr>
<td>ADD.W</td>
<td>R0, R1, R0, LSR#31</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>LR</td>
<td></td>
</tr>
</tbody>
</table>

There are separate instructions for shifting in Thumb mode, and one of them is used here—ASRS (arithmetic shift right).

### 41.2.3 Non-optimizing Xcode 4.6.3 (LLVM) and Keil 6/2013

Non-optimizing LLVM does not generate the code we saw before in this section, but instead inserts a call to the library function \(\text{\_\_divsi3}\).

What about Keil: it inserts a call to the library function \(\text{\_\_aeabi_idivmod}\) in all cases.

---

\(^2\) These instructions are also called “data processing instructions”
For some reason, optimizing GCC 4.4.5 generate just a division instruction:

**Listing 41.4: Optimizing GCC 4.4.5 (IDA)**

```assembly
f:
    li   $v0, 9
    bnez $v0, loc_10
    div  $a0, $v0 ; branch delay slot
    break 0x1C00 ; "break 7" in assembly output

and objdump

loc_10:
    mflo  $v0
    jr   $ra
    or   $at, $zero ; branch delay slot, NOP
```

Here we see here a new instruction: BREAK. It just raises an exception. In this case, an exception is raised if the divisor is zero (it’s not possible to divide by zero in conventional math). But GCC probably did not do very well the optimization job and did not see that $V0 is never zero. So the check is left here. So if $V0 is zero somehow, BREAK is to be executed, signaling to the OS about the exception. Otherwise, MFLO executes, which takes the result of the division from the LO register and copies it in $V0.

By the way, as we may know, the MUL instruction leaves the high 32 bits of the result in register HI and the low 32 bits in register LO. DIV leaves the result in the LO register, and remainder in the HI register.

If we alter the statement to “a % 9”, the MFHI instruction is to be used here instead of MFLO.

### 41.4 How it works

That’s how division can be replaced by multiplication and division with $2^n$ numbers:

\[
\text{result} = \frac{\text{input}}{\text{divisor}} = \text{input} \cdot \frac{2^n}{\text{divisor}} = \text{input} \cdot M
\]

Where $M$ is a *magic* coefficient.

This is how $M$ can be computed:

\[
M = \frac{2^n}{\text{divisor}}
\]
So these code snippets usually have this form:

\[ \text{result} = \frac{\text{input} \cdot M}{2^n} \]

Division by \(2^n\) is usually done by simply shifting to the right. If \(n < 32\), then the low part of the product is shifted (in EAX or RAX). If \(n \geq 32\), then the high part of the product is shifted (in EDX or RDX).

\(n\) is chosen in order to minimize the error.

When doing signed division, the sign of the multiplication result is also added to the output result.

Take a look at the difference:

```c
int f3_32_signed(int a)
{
    return a/3;
};

unsigned int f3_32_unsigned(unsigned int a)
{
    return a/3;
};
```

In the unsigned version of the function, the magic coefficient is \(0xAAAAAAAB\) and the multiplication result is divided by \(2^{33}\).

In the signed version of the function, the magic coefficient is \(0x55555556\) and the multiplication result is divided by \(2^{32}\). There are no division instruction though: the result is just taken from EDX.

The sign is also taken from the multiplication result: the high 32 bits of the result are shifted by 31 (leaving the sign in the least significant bit of EAX). 1 is added to the final result if the sign is negative, for result correction.

Listing 41.5: Optimizing MSVC 2012

```assembly
_f3_32_unsigned PROC
    mov eax, -1431655765 ; aaaaaaabH
    mul DWORD PTR _a$[esp-4] ; unsigned multiply
    ; EDX=(input*0xaaaaaaab)/2^32
    shr edx, 1
    ; EDX=(input*0xaaaaaaab)/2^33
    mov eax, edx
    ret 0
_f3_32_unsigned ENDP

_f3_32_signed PROC
```
41.4.1 More theory

It works, because that is how it's possible to replace division by multiplication:

\[
\frac{x}{c} = \frac{1}{c}
\]

\(\frac{1}{c}\) is called multiplicative inverse and can be precomputed by compiler.

But this is for real numbers What about integers? It's possible to find multiplicative inverse for integer in the environment of modulo arithmetics. CPU registers fits nicely: each is limited by 32 or 64 bits, so almost any arithmetic operation on registers are in fact opeartions on modulo \(2^{32}\) or \(2^{64}\).

Read more about it in [War02, pp. 10-3].

41.5 Getting the divisor

41.5.1 Variant #1

Often, the code looks like this:

```asm
mov  eax, 1431655766 ; 55555556H
imul  DWORD PTR _a$[esp-4] ; signed multiply
; take high part of product
; it is just the same as if to shift product by 32 bits right or
to divide it by \(2^{32}\)
shr  eax, 31 ; 0000001fH
add  eax, edx ; add 1 if sign is negative
ret 0
_f3_32_signed ENDP
```

\[^3\text{Wikipedia}\]
Let's denote the 32-bit magic coefficient as $M$, the shifting coefficient as $C$ and the divisor as $D$.

The divisor we need to get is:

$$D = \frac{2^{32+C}}{M}$$

For example:

**Listing 41.6: Optimizing MSVC 2012**

```assembly
    mov eax, 2021161081 ;
    imul DWORD PTR _a$[esp-4]
    sar edx, 3
    mov eax, edx
    shr eax, 31 ;
    add eax, edx
```

This is:

$$D = \frac{2^{32+3}}{2021161081}$$

The numbers are larger than 32-bit, so we can use Wolfram Mathematica for convenience:

**Listing 41.7: Wolfram Mathematica**

```wolfram
In[1]:=N[2^(32+3)/2021161081]
Out[1]:=17.
```

So the divisor from the code we used as example is 17.

For x64 division, things are the same, but $2^{64}$ has to be used instead of $2^{32}$:

```c
uint64_t f1234(uint64_t a)
{
    return a/1234;
};
```

**Listing 41.8: Optimizing MSVC 2012 x64**

```assembly
f1234 PROC
    mov rax, 7653754429286296943 ; 6
    mul rcx
```

683
41.5.2 Variant #2

A variant with omitted arithmetic shift also exists:

```
mov eax, 55555556h ; 1431655766
imul ecx
mov eax, edx
shr eax, 1Fh
```

The method of getting divisor is simplified:

\[
D = \frac{2^{32}}{M}
\]

As of my example, this is:

\[
D = \frac{2^{32}}{1431655766}
\]

And again we use Wolfram Mathematica:

```
Listing 41.10: Wolfram Mathematica
In[1]:=N[2^32/16^^55555556]
Out[1]:=3.
```

The divisor is 3.

41.6 Exercise

- [http://challenges.re/27](http://challenges.re/27)
Chapter 42

String to number conversion (atoi())

Let's try to reimplement the standard atoi() C function.

42.1 Simple example

Here is the simplest possible way to read a number represented in ASCII\(^1\) encoding. It's not error-prone: a character other than a digit leads to incorrect result.

```c
#include <stdio.h>

int my_atoi (char *s)
{
    int rt=0;

    while (*s)
    {
        rt=rt*10 + (*s-'0');
        s++;
    }

    return rt;
}

int main()
```

\(^1\) American Standard Code for Information Interchange
So what the algorithm does is just reading digits from left to right. The zero ASCII character is subtracted from each digit. The digits from “0” to “9” are consecutive in the ASCII table, so we do not even need to know the exact value of the “0” character. All we need to know is that “0” minus “0” is 0, “9” minus “0” is 9 and so on. Subtracting “0” from each character results in a number from 0 to 9 inclusive. Any other character leads to an incorrect result, of course! Each digit has to be added to the final result (in variable “rt”), but the final result is also multiplied by 10 at each digit. In other words, the result is shifted left by one position in decimal form on each iteration. The last digit is added, but there is no no shift.

42.1.1 Optimizing MSVC 2013 x64

Listing 42.1: Optimizing MSVC 2013 x64

```assembly
s$ = 8
my_atoi PROC
; load first character
    movzx r8d, BYTE PTR [rcx]
; EAX is allocated for "rt" variable
; its 0 at start'
    xor eax, eax
; first character is zero-byte, i.e., string terminator?
    test r8b, r8b
    je SHORT $LN9@my_atoi
$LL2@my_atoi:
    lea edx, DWORD PTR [rax+rax*4]
; EDX=RAX+RAX*4=rt+rt*4=rt*5
    movsx eax, r8b
; EAX=input character
    movzx r8d, BYTE PTR [rcx+1]
; shift pointer in RCX to the next character:
    lea rcx, QWORD PTR [rcx+1]
    lea eax, DWORD PTR [rax+rdx*2]
; EAX=RAX+RDX*2=input character + rt*5*2=input character + rt*10
; correct digit by subtracting 48 (0x30 or '0')
    add eax, -48
; ffffffffffffffffd0H
; was last character zero?
```

```c
{    printf ("%d\n", my_atoi ("1234"));
    printf ("%d\n", my_atoi ("1234567890"));
};
```
A character can be loaded in two places: the first character and all subsequent characters. This is done for loop regrouping. There is no instruction for multiplication by 10; two LEA instructions do this instead. MSVC sometimes uses the ADD instruction with a negative constant instead of SUB. This is the case. It’s very hard to say why this is better than SUB. But MSVC does this often.

### 42.1.2 Optimizing GCC 4.9.1 x64

Optimizing GCC 4.9.1 is more concise, but there is one redundant RET instruction at the end. One would be enough.

Listing 42.2: Optimizing GCC 4.9.1 x64

```assembly
my_atoi:
; load input character into EDX
  movsx edx, BYTE PTR [rdi]
; EAX is allocated for "rt" variable
  xor eax, eax
; exit, if loaded character is null byte
  test dl, dl
  je .L4
.L3:
  lea eax, [rax+rax*4]
; EAX=RAX*5=rt*5
; shift pointer to the next character:
  add rdi, 1
  lea eax, [rdx-48+rax*2]
; EAX=input character - 48 + RAX*2 = input character - '0' + rt*10
; load next character:
  movsx edx, BYTE PTR [rdi]
; goto loop begin, if loaded character is not null byte
  test dl, dl
  jne .L3
  rep ret
.L4:
  rep ret
```
### 42.1.3 Optimizing Keil 6/2013 (ARM mode)

Listing 42.3: Optimizing Keil 6/2013 (ARM mode)

```
my_atoi PROC
; R1 will contain pointer to character
    MOV r1,r0
; R0 will contain "rt" variable
    MOV r0,#0
    B |L0.28|
|L0.12|
    ADD r0,r0,r0,LSL #2
; R0=R0+R0<<2=rt*5
    ADD r0,r2,r0,LSL #1
; R0=input character + rt*5<<1 = input character + rt*10
; correct whole thing by subtracting '0' from rt:
    SUB r0,r0,#0x30
; shift pointer to the next character:
    ADD r1,r1,#1
|L0.28|
; load input character to R2
    LDRB r2,[r1,#0]
; is it null byte? if no, jump to loop body.
    CMP r2,#0
    BNE |L0.12|
; exit if null byte.
; "rt" variable is still in R0 register, ready to be used in
caller function
    BX lr
ENDP
```

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

Listing 42.4: Optimizing Keil 6/2013 (Thumb mode)

```
my_atoi PROC
; R1 will be pointer to the input character
    MOVS r1,r0
; R0 is allocated to "rt" variable
    MOVS r0,#0
    B |L0.16|
|L0.6|
    MOVS r3,#0xa
; R3=10
    MULS r0,r3,r0
; R0=R3*R0=rt*10
```

---

688
Interestingly, from school mathematics we may remember that the order of addition and subtraction operations doesn’t matter. That’s our case: first, the $rt \times 10 - '0'$ expression is computed, then the input character value is added to it. Indeed, the result is the same, but the compiler did some regrouping.

### 42.1.5 Optimizing GCC 4.9.1 ARM64

The ARM64 compiler can use the pre-increment instruction suffix:
42.2 A slightly advanced example

My new code snippet is more advanced, now it checks for the “minus” sign at the first character and reports an error if a non-digit was found in the input string:

```c
#include <stdio.h>

int my_atoi (char *s)
{
    int negative=0;
    int rt=0;
    if (*s=='-')
    {
        negative=1;
        s++;
    }
    while (*s)
    {
        if (*s<'0' || *s>'9')
        {
            printf("Error! Unexpected char: '%c'
\n", *s);
            exit(0);
        }
        rt=rt*10 + (*s-'0');
        s++;
    }
    return rt * negative;
}
```
if (negative)
    return -rt;
return rt;
};

int main()
{
    printf("%d\n", my_atoi("1234"));
    printf("%d\n", my_atoi("1234567890"));
    printf("%d\n", my_atoi("-1234"));
    printf("%d\n", my_atoi("-1234567890"));
    printf("%d\n", my_atoi("-a1234567890")); // error
};

42.2.1 Optimizing GCC 4.9.1 x64

Listing 42.6: Optimizing GCC 4.9.1 x64

.LC0:
    .string "Error! Unexpected char: \'%c\'\n"

my_atoi:
    sub    rsp, 8
    movsx  edx, BYTE PTR [rdi]
    ; check for minus sign
    cmp    dl, 45 ; '-'
    je     .L22
    xor    esi, esi
    test   dl, dl
    je     .L20
    L10:
    ; ESI=0 here if there was no minus sign and 1 if it was
    lea    eax, [rdx-48]
    ; any character other than digit will result unsigned number
    ; greater than 9 after subtraction
    ; so if it is not digit, jump to L4, where error will be
    reported:
    cmp    al, 9
    ja     .L4
    xor    eax, eax
    jmp    .L6
L7:
    lea    ecx, [rdx-48]
    cmp    cl, 9
    ja     .L4
If the “minus” sign was encountered at the string start, the NEG instruction is to be executed at the end. It just negates the number.

There is one more thing that needs mentioning. How would a common programmer check if the character is not a digit? Just how we have it in the source code:

```c
if (*s<'0' || *s>'9')
...
```

There are two comparison operations. What is interesting is that we can replace both operations by single one: just subtract “0” from character value, treat result
as unsigned value (this is important) and check if it’s greater than 9.

For example, let’s say that the user input contains the dot character ("."), which has ASCII code 46. $46 - 48 = -2$ if we treat the result as a signed number. Indeed, the dot character is located two places earlier than the “0” character in the ASCII table. But it is $0xFFFFFFFF$ ($4294967294$) if we treat the result as an unsigned value, and that’s definitely bigger than 9!

The compilers do this often, so it’s important to recognize these tricks.

Another example of it in this book: 48.1.2 on page 738.

Optimizing MSVC 2013 x64 does the same tricks.

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

Listing 42.7: Optimizing Keil 6/2013 (ARM mode)

```assembly
my_atoi PROC
PUSH {r4-r6,lr}
MOV r4,r0
LDRB r0,[r0,#0]
MOV r6,#0
MOV r5,r6
CMP r0,#0x2D '-'
; R6 will contain 1 if minus was encountered, 0 if otherwise
MOVEQ r6,#1
ADDEQ r4,r4,#1
B |L0.36|
SUB r0,r1,#0x30
CMP r0,#0xa
BCC |L0.64|
ADR r0,|L0.220|
BL __2printf
MOV r0,#0
BL exit

|L0.36|
LDRB r0,[r4],#1
ADD r1,r5,r5,LSL #2
ADD r0,r0,r1,LSL #1
SUB r5,r0,#0x30
|L0.80|
LDRB r1,[r4,#0]
CMP r1,#0
BNE |L0.36|
CMP r6,#0
; negate result
```

693
There is no NEG instruction in 32-bit ARM, so the “Reverse Subtraction” operation (line 31) is used here. It is triggered if the result of the CMP instruction (at line 29) was “Not Equal” (hence -NE suffix). So what RSBNE does is to subtract the resulting value from 0. It works just like the regular subtraction operation, but swaps operands. Subtracting any number from 0 results in negation: \[0 - x = -x.\]

Thumb mode code is mostly the same.

GCC 4.9 for ARM64 can use the NEG instruction, which is available in ARM64.

### 42.3 Exercise

Oh, by the way, security researchers deals often with unpredictable behaviour of program while handling of incorrect data. For example, while fuzzing. As an exercise, you may try to enter non-digit characters and see what happens. Try to explain, what happened and why.
Chapter 43

Inline functions

Inlined code is when the compiler, instead of placing a call instruction to a small or tiny function, just places its body right in-place.

Listing 43.1: A simple example

```c
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
    return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
    int celsius=atol(argv[1]);
    printf ("%d\n", celsius_to_fahrenheit (celsius));
};
```

... is compiled in very predictable way, however, if we turn on GCC optimizations (-O3), we'll see:

Listing 43.2: Optimizing GCC 4.8.1

```assembly
    push   ebp
    mov    ebp, esp
    and    esp, -16
    sub    esp, 16
    call   ___main
    mov    eax, DWORD PTR [ebp+12]
    mov    eax, DWORD PTR [eax+4]
```

CHAPTER 43. INLINE FUNCTIONS

```
mov    DWORD PTR [esp], eax
call   _atol
mov    edx, 1717986919
mov    DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d\12\0"
lea    ecx, [eax+eax*8]
mov    eax, ecx
imul   edx
sar    ecx, 31
sar    edx
sub    edx, ecx
add    edx, 32
mov    DWORD PTR [esp+4], edx
call   _printf
leave
ret
```

(Here the division is done by multiplication( 41 on page 676).)

Yes, our small function celsius_to_fahrenheit() was just placed before the printf() call. Why? It can be faster than executing this function’s code plus the overhead of calling/returning.

Modern optimizing compilers are choosing small functions for inlining automatically. But it’s possible to force compiler additionally to inline some function, if to mark it with the “inline” keyword in its declaration.

### 43.1 Strings and memory functions

Another very common automatic optimization tactic is the inlining of string functions like strcpy(), strcmp(), strlen(), memset(), memcmp(), memcpy(), etc.

Sometimes it’s faster than to call a separate function.

These are very frequent patterns and it is highly advisable for reverse engineers to learn to detect automatically.

#### 43.1.1 strcmp()

```
bool is_bool (char *s)
{
    if (strcmp (s, "true")==0)
        return true;
    if (strcmp (s, "false")==0)
```
Listing 43.4: Optimizing GCC 4.8.1

```
return false;

assert(0);
```

```
.LC0:
    .string "true"
.LC1:
    .string "false"

is_bool:
.LFB0:
push    edi
mov     ecx, 5
push    esi
mov     edi, OFFSET FLAT:.LC0
sub     esp, 20
mov     esi, DWORD PTR [esp+32]
repz cmpsb
je      .L3
mov     esi, DWORD PTR [esp+32]
mov     ecx, 6
mov     edi, OFFSET FLAT:.LC1
repz cmpsb
seta    cl
setb    dl
xor     eax, eax
cmp     cl, dl
jne     .L8
add     esp, 20
pop     esi
pop     edi
ret

.L8:
    mov     DWORD PTR [esp], 0
call    assert
add     esp, 20
pop     esi
pop     edi
ret

.L3:
    add     esp, 20
mov      eax, 1
pop      esi
pop      edi
ret
```
Listing 43.5: Optimizing MSVC 2010

```asm
$SG3454 DB 'true', 00H
$SG3456 DB 'false', 00H

_s$ = 8 ; size = 4
?is_bool@YA_NPAD@Z PROC ; is_bool
    push esi
    mov esi, DWORD PTR _s$_[esp]
    mov ecx, OFFSET $SG3454 ; 'true'
    mov eax, esi
    npad 4 ; align next label
$LL6@is_bool:
    mov dl, BYTE PTR [eax]
    cmp dl, BYTE PTR [ecx]
    jne SHORT $LN7@is_bool
    test dl, dl
    je SHORT $LN8@is_bool
    mov dl, BYTE PTR [eax+1]
    cmp dl, BYTE PTR [ecx+1]
    jne SHORT $LN7@is_bool
    add eax, 2
    add ecx, 2
    test dl, dl
    jne SHORT $LL6@is_bool
$LN8@is_bool:
    xor eax, eax
    jmp SHORT $LN9@is_bool
$LN7@is_bool:
    sbb eax, eax
    sbb eax, -1
$LN9@is_bool:
    test eax, eax
    jne SHORT $LN2@is_bool

    mov al, 1
    pop esi

    ret 0
$LN2@is_bool:
    mov ecx, OFFSET $SG3456 ; 'false'
    mov eax, esi
$LL10@is_bool:
    mov dl, BYTE PTR [eax]
    cmp dl, BYTE PTR [ecx]
    jne SHORT $LN11@is_bool
    test dl, dl
    jne SHORT $LL10@is_bool
```

43.1.2 strlen()

Listing 43.6: strlen() example

```c
int strlen_test(char *s1)
{
    return strlen(s1);
}
```

Listing 43.7: Optimizing MSVC 2010
43.1.3 strcpy()

Listing 43.8: strcpy() example

```c
void strcpy_test(char *s1, char *outbuf)
{
    strcpy(outbuf, s1);
}
```

Listing 43.9: Optimizing MSVC 2010

```c
void strcpy_test(char *s1, char *outbuf)
{
    strcpy(outbuf, s1);
}
```

43.1.4 memset()

Example#1
Many compilers don’t generate a call to `memset()` for short blocks, but rather insert a pack of MOVs:

Listing 43.11: Optimizing GCC 4.9.1 x64

```assembly
f:
    mov    QWORD PTR [rdi], 0
    mov    QWORD PTR [rdi+8], 0
    mov    QWORD PTR [rdi+16], 0
    mov    QWORD PTR [rdi+24], 0
    ret
```

By the way, that remind us of unrolled loops: [14.1.4 on page 262](#).

### Example #2

Listing 43.12: 67 bytes

```c
#include <stdio.h>

void f(char *out)
{
    memset(out, 0, 67);
}
```

When the block size is not a multiple of 4 or 8, the compilers can behave differently. For instance, MSVC 2012 continues to insert MOVs:

Listing 43.13: Optimizing MSVC 2012 x64

```assembly
out$ = 8
f      PROC
    xor    eax, eax
    mov    QWORD PTR [rcx], rax
    mov    QWORD PTR [rcx+8], rax
    mov    QWORD PTR [rcx+16], rax
    mov    QWORD PTR [rcx+24], rax
    mov    QWORD PTR [rcx+32], rax
```

701
...while GCC uses REP STOSQ, concluding that this would be shorter than a pack of MOVs:

**Listing 43.14: Optimizing GCC 4.9.1 x64**

```assembly
f:
    mov    QWORD PTR [rdi], 0
    mov    QWORD PTR [rdi+59], 0
    mov    rcx, rdi
    lea    rdi, [rdi+8]
    xor    eax, eax
    and    rdi, -8
    sub    rcx, rdi
    add    ecx, 67
    shr    ecx, 3
    rep stosq
    ret
```

### 43.1.5 memcpy()

**Short blocks**

The routine to copy short blocks is often implemented as a sequence of MOV instructions.

**Listing 43.15: memcpy() example**

```c
void memcpy_7(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 7);
}
```

**Listing 43.16: Optimizing MSVC 2010**

```assembly
_inbuf$ = 8 ; size = 4
_outbuf$ = 12 ; size = 4
_memcpy_7 PROC
    mov    ecx, DWORD PTR _inbuf$[esp-4]
```
Listing 43.17: Optimizing GCC 4.8.1

```assembly
memcpy_7:
push    ebx
mov    eax, DWORD PTR [esp+8]
mov    ecx, DWORD PTR [esp+12]
mov    ebx, DWORD PTR [eax]
lea    edx, [ecx+10]
mov    DWORD PTR [ecx+10], ebx
movzx   ecx, WORD PTR [eax+4]
mov    WORD PTR [edx+4], cx
movzx   eax, BYTE PTR [eax+6]
mov    BYTE PTR [edx+6], al
pop    ebx
ret
```

That's usually done as follows: 4-byte blocks are copied first, then a 16-bit word (if needed), then the last byte (if needed).

Structures are also copied using MOV: 21.4.1 on page 494.

**Long blocks**

The compilers behave differently in this case.

Listing 43.18: memcpy() example

```c
void memcpy_128(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 128);
};

void memcpy_123(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 123);
};
```
For copying 128 bytes, MSVC uses a single MOVSD instruction (because 128 divides evenly by 4):

Listing 43.19: Optimizing MSVC 2010

```assembly
_inbuf$ = 8 ; size = 4
_outbuf$ = 12 ; size = 4
_memcpy_128 PROC
    push esi
    mov esi, DWORD PTR _inbuf$[esp]
    push edi
    mov edi, DWORD PTR _outbuf$[esp+4]
    add edi, 10
    mov ecx, 32
    rep movsd
    pop edi
    pop esi
    ret 0
_memcpy_128 ENDP
```

When copying 123 bytes, 30 32-byte words are copied first using MOVSD (that’s 120 bytes), then 2 bytes are copied using MOVSW, then one more byte using MOVSB.

Listing 43.20: Optimizing MSVC 2010

```assembly
_inbuf$ = 8 ; size = 4
_outbuf$ = 12 ; size = 4
_memcpy_123 PROC
    push esi
    mov esi, DWORD PTR _inbuf$[esp]
    push edi
    mov edi, DWORD PTR _outbuf$[esp+4]
    add edi, 10
    mov ecx, 30
    rep movsd
    movsw
    movsb
    pop edi
    pop esi
    ret 0
_memcpy_123 ENDP
```

GCC uses one big universal functions, that works for any block size:

Listing 43.21: Optimizing GCC 4.8.1

```assembly
memcpy_123:
.LFB3:
    push edi
```
CHAPTER 43. INLINE FUNCTIONS

Universal memory copy functions usually work as follows: calculate how many 32-bit words can be copied, then copy them using MOVSD, then copy the remaining
More complex copy functions use SIMD instructions and also take the memory alignment in consideration. As an example of SIMD strlen() function: 25.2 on page 574.

### 43.1.6 memcmp()

Listing 43.22: memcmp() example

```c
void memcpy_1235(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 1235);
}
```

For any block size, MSVC 2010 inserts the same universal function:

Listing 43.23: Optimizing MSVC 2010

```assembly
_buf1$ = 8 ; size = 4
_buf2$ = 12 ; size = 4
_memcmp_1235 PROC
    mov edx, DWORD PTR _buf2$[esp-4]
    mov ecx, DWORD PTR _buf1$[esp-4]
    push esi
    push edi
    mov esi, 1235
    add edx, 10
$LL4@memcmp_123:
    mov eax, DWORD PTR [edx]
    cmp eax, DWORD PTR [ecx]
    jne SHORT $LN10@memcmp_123
    sub esi, 4
    add ecx, 4
    add edx, 4
    cmp esi, 4
    jae SHORT $LL4@memcmp_123
$LN10@memcmp_123:
    movzx edi, BYTE PTR [ecx]
    movzx eax, BYTE PTR [edx]
    sub eax, edi
    jne SHORT $LN7@memcmp_123
    movzx eax, BYTE PTR [edx+1]
    movzx edi, BYTE PTR [ecx+1]
    sub eax, edi
    jne SHORT $LN7@memcmp_123
    movzx eax, BYTE PTR [edx+2]
```

706
43.1.7 IDA script

There is also a small IDA script for searching and folding such very frequently seen pieces of inline code:

GitHub.
Chapter 44

C99 restrict

Here is a reason why FORTRAN programs, in some cases, work faster than C/C++ ones.

```c
void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    }
}
```

That’s very simple example with one specific thing in it: the pointer to the `update_me` array could be a pointer to the `sum` array, `product` array, or even the `sum_product` array—nothing forbids that, right?

The compiler is fully aware of this, so it generates code with four stages in the loop body:

- calculate next `sum[i]`
- calculate next `product[i]`
- calculate next `update_me[i]`
- calculate next `sum_product[i]`—on this stage, we need to load from memory the already calculated `sum[i]` and `product[i]`
Is it possible to optimize the last stage? Since we have already calculated \(\text{sum}[i]\) and \(\text{product}[i]\) it is not necessary to load them again from memory. Yes, but compiler is not sure that nothing was overwritten in the 3rd stage! This is called “pointer aliasing”, a situation when the compiler cannot be sure that a memory to which a pointer is pointing was not changed.

`restrict` in the C99 standard[ISO07, pp. 6.7.3/1] is a promise, given by programmer to the compiler that the function arguments marked by this keyword always points to different memory locations and never intersects.

To be more precise and describe this formally, `restrict` shows that only this pointer is to be used to access an object, and no other pointer will be used for it. It can be even said the object will be accessed only via one single pointer, if it is marked as `restrict`.

Let’s add this keyword to each pointer argument:

```c
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict product, int* restrict sum_product, int* restrict update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    }
};
```

Let’s see results:

Listing 44.1: GCC x64: f1()

```
f1:
    push r15 r14 r13 r12 rbp rdi rsi rbx
    mov r13, QWORD PTR 120[rsp]
    mov rbp, QWORD PTR 104[rsp]
    mov r12, QWORD PTR 112[rsp]
    test r13, r13
    je .L1
    add r13, 1
    xor ebx, ebx
    mov edi, 1
    xor r11d, r11d
    jmp .L4
.L6:
    mov r11, rdi
    mov rdi, rax
```

709
Listing 44.2: GCC x64: f2()

f2:
push r13 r12 rbp rdi rsi rbx
mov r13, QWORD PTR 104[rsp]
mov rbp, QWORD PTR 88[rsp]
mov r12, QWORD PTR 96[rsp]
test r13, r13
je .L7
add r13, 1
xor r10d, r10d
mov edi, 1
xor eax, eax
jmp .L10

.L11:
mov rax, rdi
mov rdi, r11

.L10:
mov esi, DWORD PTR [rcx+rax*4]
mov r11d, DWORD PTR [rdx+rax*4]
The difference between the compiled \( f_1() \) and \( f_2() \) functions is as follows: in \( f_1() \), \( \text{sum}[i] \) and \( \text{product}[i] \) are reloaded in the middle of the loop, and in \( f_2() \) there is no such thing, the already calculated values are used, since we “promised” the compiler that no one and nothing will change the values in \( \text{sum}[i] \) and \( \text{product}[i] \) during the execution of the loop’s body, so it is “sure” that there is no need to load the value from memory again. Obviously, the second example works faster.

But what if the pointers in the function’s arguments intersect somehow? This is on the programmer’s conscience, and the results will be incorrect.

Let’s go back to FORTRAN. Compilers of this programming language treats all pointers as such, so when it was not possible to set \textit{restrict} in C, FORTRAN can generate faster code in these cases.

How practical is it? In the cases when the function works with several big blocks in memory. There are a lot of such in linear algebra, for instance. A lot of linear algebra is done on supercomputers/HPC\(^1\), so probably that is why, traditionally, FORTRAN is still used there [Loh10].

But when the number of iterations is not very big, certainly, the speed boost may not to be significant.

\(^{1}\)High-Performance Computing
Chapter 45

Branchless \texttt{abs()} function

Let’s revisit an example we considered earlier 12.2 on page 191 and ask ourselves, is it possible to make a branchless version of the function in x86 code?

```c
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
}
```

And the answer is yes.

45.1 Optimizing GCC 4.9.1 x64

We could see it if we compile it using optimizing GCC 4.9:

Listing 45.1: Optimizing GCC 4.9 x64

```
my_abs:
    mov    edx, edi
    mov    eax, edi
    sar    edx, 31
; EDX is 0xFFFFFFFF here if sign of input value is minus
; EDX is 0 if sign of input value is plus (including 0)
; the following two instructions have effect only if EDX is
; 0xFFFFFFFF
; or idle if EDX is 0
    xor    eax, edx
```
This is how it works:

Arithmetically shift the input value left by 31. Arithmetical shift implies sign extension, so if the MSB is 1, all 32 bits are to be filled with 1, or with 0 if otherwise. In other words, the SAR REG, 31 instruction makes 0xFFFFFFFF if the sign was negative or 0 if positive. After the execution of SAR, we have this value in EDX. Then, if the value is 0xFFFFFFFF (i.e., the sign is negative), the input value is inverted (because XOR REG, 0xFFFFFFFF is effectively an inverse all bits operation). Then, again, if the value is 0xFFFFFFFF (i.e., the sign is negative), 1 is added to the final result (because subtracting −1 from some value resulting in incrementing it). Inversion of all bits and incrementing is exactly how two's complement value is negated: 30 on page 617.

We may observe that the last two instruction do something if the sign of the input value is negative. Otherwise (if the sign is positive) they do nothing at all, leaving the input value untouched.

The algorithm is explained in [War02, pp. 2-4]. It’s hard to say, how GCC did it, deduced it by itself or found a suitable pattern among known ones?

### 45.2 Optimizing GCC 4.9 ARM64

GCC 4.9 for ARM64 generates mostly the same, just decides to use the full 64-bit registers. There are less instructions, because the input value can be shifted using a suffixed instruction (“asr”) instead of using a separate instruction.

```assembly
my_abs:
; sign-extend input 32-bit value to X0 64-bit register:
  sxtw x0, w0
  eor x1, x0, x0, asr 63
; X1=X0^(X0>>63) (shift is arithmetical)
  sub x0, x1, x0, asr 63
; X0=X1-(X0>>63)=X0^(X0>>63)-(X0>>63) (all shifts are arithmetical)
  ret
```
Chapter 46

Variadic functions

Functions like printf() and scanf() can have a variable number of arguments. How are these arguments accessed?

46.1 Computing arithmetic mean

Let's imagine that we need to calculate arithmetic mean, and for some weird reason we need to specify all the values as function arguments.

But it's impossible to get the number of arguments in a variadic function in C/C++, so let's denote the value of −1 as a terminator.

There is the standard stdarg.h header file which define macros for dealing with such arguments. The printf() and scanf() functions use them as well.

```c
#include <stdio.h>
#include <stdarg.h>

int arith_mean(int v, ...) {
    va_list args;
    int sum=v, count=1, i;
    va_start(args, v);

    while(1) {
        i=va_arg(args, int);
        if (i==-1) // terminator
            break;
        sum=sum+i;
    }
}
```


The first argument has to be treated just like a normal argument. All other arguments are loaded using the `va_arg` macro and then summed.

So what is inside?

### 46.1.1 cdecl calling conventions

Listing 46.1: Optimizing MSVC 6.0

```assembly
_v$ = 8
_arith_mean PROC NEAR
    mov    eax, DWORD PTR _v$[esp-4] ; load 1st argument into sum
    push   esi
    mov    esi, 1 ; count=1
    lea    edx, DWORD PTR _v$[esp] ; address of the 1st argument
$L838:
    mov    ecx, DWORD PTR [edx+4] ; load next argument
    add    edx, 4 ; shift pointer to the next argument
    cmp    ecx, -1 ; is it -1?
    je     SHORT $L856 ; exit if so
    add    eax, ecx ; sum = sum + loaded argument
    inc    esi ; count++
$L856:
    ; calculate quotient
    cdq
    idiv   esi
    pop    esi
    ret    0
_arith_mean ENDP
```
The arguments, as we may see, are passed to \texttt{main()} one-by-one. The first argument is pushed into the local stack as first. The terminating value ($-1$) is pushed last.

The \texttt{arith\_mean()} function takes the value of the first argument and stores it in the \texttt{sum} variable. Then, it sets the \texttt{EDX} register to the address of the second argument, takes the value from it, adds it to \texttt{sum}, and does this in an infinite loop, until $-1$ is found.

When it’s found, the sum is divided by the number of all values (excluding $-1$) and the \texttt{quotient} is returned.

So, in other words, the function treats the stack fragment as an array of integer values of infinite length. Now we can understand why the cdecl calling convention forces us to push the first argument into the stack as last. Because otherwise, it would not be possible to find the first argument, or, for printf-like functions, it would not be possible to find the address of the format-string.

### 46.1.2 Register-based calling conventions

The observant reader may ask, what about calling conventions where the first few arguments are passed in registers? Let’s see:

<table>
<thead>
<tr>
<th>Listing 46.2: Optimizing MSVC 2012 x64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{SG3013} \ DB \ 'd', 0AH, 00H</td>
</tr>
<tr>
<td>$v$ = 8 \ arith_mean PROC</td>
</tr>
</tbody>
</table>
We see that the first 4 arguments are passed in the registers and two more—in the stack. The arith_mean() function first places these 4 arguments into the
Shadow Space and then treats the Shadow Space and stack behind it as a single continuous array!

What about GCC? Things are slightly clumsier here, because now the function is divided in two parts: the first part saves the registers into the “red zone”, processes that space, and the second part of the function processes the stack:

Listing 46.3: Optimizing GCC 4.9.1 x64

```assembly
arith_mean:
    lea rax, [rsp+8]
    ; save 6 input registers in "red zone" in the local stack
    mov QWORD PTR [rsp-40], rsi
    mov QWORD PTR [rsp-32], rdx
    mov QWORD PTR [rsp-16], r8
    mov QWORD PTR [rsp-24], rcx
    mov esi, 8
    mov QWORD PTR [rsp-64], rax
    lea rax, [rsp-48]
    mov QWORD PTR [rsp-8], r9
    mov DWORD PTR [rcx], 8
    lea rdx, [rsp+8]
    mov r8d, 1
    mov QWORD PTR [rsp-56], rax
    jmp .L5
.L7:
    ; work out saved arguments
    lea rax, [rsp-48]
    mov ecx, esi
    add esi, 8
    add rcx, rax
    mov ecx, DWORD PTR [rcx-4]
    cmp ecx, -1
    je .L4
.L8:
    add edi, ecx
    add r8d, 1
.L5:
    ; decide, which part we will work out now.
    ; is current argument number less or equal 6?
    cmp esi, 47
    jbe .L7 ; no, process saved arguments
    then
    ; work out arguments from stack
    mov rcx, rdx
    add rdx, 8
    mov ecx, DWORD PTR [rcx-4]
    cmp ecx, -1
```

718
By the way, a similar usage of the *Shadow Space* is also considered here: 64.8 on page 962.

### 46.2 vprintf() function case

Many programmers define their own logging functions which take a printf-like format string + a variable number of arguments.

Another popular example is the die() function, which prints some message and exits. We need some way to pack input arguments of unknown number and pass them to the printf() function. But how? That's why there are functions with “v” in name. One of them is vprintf(): it takes a format-string and a pointer to a variable of type va_list:

```c
#include <stdlib.h>
#include <stdarg.h>

void die (const char * fmt, ...) {
```
By closer examination, we can see that `va_list` is a pointer to an array. Let's compile:

**Listing 46.4: Optimizing MSVC 2010**

```
_fmt$ = 8
_die PROC
    ; load 1st argument (format-string)
    mov    ecx, DWORD PTR _fmt$[esp-4]
    ; get pointer to the 2nd argument
    lea    eax, DWORD PTR _fmt$[esp]
    push   eax
    push   ecx
    call   _vprintf
    push   0
    call   _exit
$LN3@die:
int     3
_die ENDP
```

We see that all our function does is just taking a pointer to the arguments and passing it to `vprintf()`, and that function is treating it like an infinite array of arguments!

**Listing 46.5: Optimizing MSVC 2012 x64**

```
_fmt$ = 48
die     PROC
    ; save first 4 arguments in Shadow Space
    mov    QWORD PTR [rsp+8], rcx
    mov    QWORD PTR [rsp+16], rdx
    mov    QWORD PTR [rsp+24], r8
    mov    QWORD PTR [rsp+32], r9
    sub    rsp, 40
    lea    rdx, QWORD PTR fmt$[rsp+8] ; pass pointer to the 1st argument
    ; RCX here is still points to the 1st argument (format-string) of die()
    ; so vprintf() will take it right from RCX
    call   vprintf
    xor    ecx, ecx
    call   exit
_die ENDP
```
<table>
<thead>
<tr>
<th>int</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>die</td>
<td>ENDP</td>
</tr>
</tbody>
</table>
Chapter 47

Strings trimming

A very common string processing task is to remove some characters at the start and/or at the end.

In this example, we are going to work with a function which removes all newline characters (CR\(^1\)/LF\(^2\)) from the end of the input string:

```c
#include <stdio.h>
#include <string.h>

char* str_trim (char *s)
{
    char c;
    size_t str_len;

    // work as long as \r or \n is at the end of string
    // stop if some other character there or its an empty string
    // (at start or due to our operation)
    for (str_len=strlen(s); str_len>0 && (c=s[str_len-1]));
    {
        if (c=='\r' || c=='\n')
            s[str_len-1]=0;
        else
            break;
    }

    return s;
}
```

\(^1\)Carriage return (13 or \'\r\' in C/C++)

\(^2\)Line feed (10 or \'\n\' in C/C++)
int main()
{
    // test
    // strdup() is used to copy text string into data
    // segment, because it will crash on Linux otherwise,
    // where text strings are allocated in constant data
    // segment, and not modifiable.
    printf ("[%s]\n", str_trim (strdup("")));
    printf ("[%s]\n", str_trim (strdup("\n")));
    printf ("[%s]\n", str_trim (strdup("\r")));
    printf ("[%s]\n", str_trim (strdup("\n\r")));
    printf ("[%s]\n", str_trim (strdup("\r\n")));
    printf ("[%s]\n", str_trim (strdup("test1\r\n")));
    printf ("[%s]\n", str_trim (strdup("test2\n\r")));
    printf ("[%s]\n", str_trim (strdup("test3\n\r\n\r")));
    printf ("[%s]\n", str_trim (strdup("test4\n")));
    printf ("[%s]\n", str_trim (strdup("test5\r")));
    printf ("[%s]\n", str_trim (strdup("test6\r\r\r")));
};

The input argument is always returned on exit, this is convenient when you need
to chain string processing functions, like it was done here in the main() function.

The second part of for() (str_len>0 && (c=s[str_len-1])) is the so called
"short-circuit" in C/C++ and is very convenient [Yur13, p. 1.3.8]. The C/C++ compil-
ers guarantee an evaluation sequence from left to right. So if the first clause is
false after evaluation, the second one is never to be evaluated.

47.1  x64: Optimizing MSVC 2013

Listing 47.1: Optimizing MSVC 2013 x64

s$ = 8
str_trim PROC

; RCX is the first function argument and it always holds pointer
to the string
    mov rdx, rcx
; this is strlen() function inlined right here:
; set RAX to 0xFFFFFFFFFFFFFFFF (-1)
    or rax, -1

723
First, MSVC inlined the strlen() function code, because it concluded this is to be faster than the usual strlen() work + the cost of calling it and returning from it. This is called inlining: 43 on page 695.

The first instruction of the inlined strlen() is OR RAX, 0xFFFFFFFFFFFFFFFF.
It's hard to say why MSVC uses OR instead of MOV RAX, 0xFFFFFFFFFFFFFFFF, but it does this often. And of course, it is equivalent: all bits are set, and a number with all bits set is \(-1\) in two's complement arithmetic: \(30\) on page 617.

Why would the \(-1\) number be used in \(\text{strlen}()\), one might ask. Due to optimizations, of course. Here is the code that MSVC generated:

```
Listing 47.2: Inlined strlen() by MSVC 2013 x64

; RCX = pointer to the input string
; RAX = current string length
or rax, -1
label:
  inc rax
  cmp BYTE PTR [rcx+rax], 0
  jne SHORT label
; RAX = string length
```

Try to write shorter if you want to initialize the counter at 0! OK, let’s try:

```
Listing 47.3: Our version of strlen()

; RCX = pointer to the input string
; RAX = current string length
xor rax, rax
label:
  cmp byte ptr [rcx+rax], 0
  jz exit
  inc rax
  jmp label
exit:
; RAX = string length
```

We failed. We need to use additional JMP instruction!

So what the MSVC 2013 compiler did is to move the INC instruction to the place before the actual character loading. If the first character is 0, that’s OK, RAX is 0 at this moment, so the resulting string length is 0.

The rest in this function seems easy to understand.

47.2 x64: Non-optimizing GCC 4.9.1

```
str_trim:
  push rbp
  mov rbp, rsp
  sub rsp, 32
```
mov QWORD PTR [rbp-24], rdi
; for() first part begins here
mov rax, QWORD PTR [rbp-24]
move rdi, rax
call strlen
mov QWORD PTR [rbp-8], rax ; str_len
; for() first part ends here
jmp .L2
; for() body begins here
.L5:
cmp BYTE PTR [rbp-9], 13 ; c=='\r'? 
je .L3

cmp BYTE PTR [rbp-9], 10 ; c=='\n'? 
jne .L4
.L3:

mov rax, QWORD PTR [rbp-8] ; str_len
lea rdx, [rax-1] ; EDX=str_len-1
mov rax, QWORD PTR [rbp-24] ; s
add rax, rdx ; RAX=s+str_len-1
mov BYTE PTR [rax], 0 ; s[str_len-1]=0
; for() body ends here
; for() third part begins here
sub QWORD PTR [rbp-8], 1 ; str_len--
; for() third part ends here
.L2:
; for() second part begins here

cmp QWORD PTR [rbp-8], 0 ; str_len==0?
je .L4 ; exit then
; check second clause, and load "c"
mov rax, QWORD PTR [rbp-8] ; RAX=str_len
lea rdx, [rax-1] ; RDX=str_len-1
mov rax, QWORD PTR [rbp-24] ; RAX=s
add rax, rdx ; RAX=s+str_len-1
movzx eax, BYTE PTR [rax] ; AL=s[str_len-1]
mov BYTE PTR [rbp-9], al ; store loaded char into "c"
hcmp BYTE PTR [rbp-9], 0 ; is it zero?
jne .L5 ; yes? exit then
; for() second part ends here
.L4:
; return "s"
mov rax, QWORD PTR [rbp-24]
leave
ret

Comments are added by the author of the book. After the execution of strlen(), the control is passed to the L2 label, and there two clauses are checked, one after
another. The second will never be checked, if the first one \((\text{str}_\text{len}==0)\) is false (this is “short-circuit”).

Now let’s see this function in short form:

- First for() part (call to \text{strlen()}\))
- goto L2
- L5: for() body. goto exit, if needed
- for() third part (decrement of \text{str}_\text{len})
- L2: for() second part: check first clause, then second. goto loop body begin or exit.
- L4: // exit
- return s

### 47.3 x64: Optimizing GCC 4.9.1

```c
str_trim:
    push rbx
    mov rbx, rdi
    ; RBX will always be "s"
    call strlen
    ; check for \text{str}_\text{len}==0 and exit if its so'
    test rax, rax
    je .L9
    lea rdx, [rax-1]
    ; RDX will always contain \text{str}_\text{len}-1 value, not \text{str}_\text{len}
    ; so RDX is more like buffer index variable
    lea rsi, [rbx+rdx] ; RSI=s+\text{str}_\text{len}-1
    movzx ecx, BYTE PTR [rsi] ; load character
    test cl, cl
    je .L9 ; exit if its zero'
    cmp cl, 10
    je .L4
    cmp cl, 13 ; exit if its not' '
                 ; and not '

.L4:
    ; this is weird instruction. we need RSI=s-1 here.
    ; its possible to get it by' MOV RSI, EBX / DEC RSI
    ; but this is two instructions instead of one
    sub rsi, rax
    ; RSI = s+\text{str}_\text{len}-1-\text{str}_\text{len} = s-1
    ; main loop begin
```
Now this is more complex. The code before the loop's body start is executed only once, but it has the CR/LF characters check too! What is this code duplication for?

The common way to implement the main loop is probably this:

- (loop start) check for CR/LF characters, make decisions
- store zero character

But GCC has decided to reverse these two steps. Of course, store zero character cannot be first step, so another check is needed:

- workout first character. match it to CR/LF, exit if character is not CR/LF
- (loop begin) store zero character
- check for CR/LF characters, make decisions

Now the main loop is very short, which is good for modern CPUs.

The code doesn't use the str_len variable, but str_len-1. So this is more like an index in a buffer. Apparently, GCC notices that the str_len-1 statement is used twice. So it's better to allocate a variable which always holds a value that's smaller than the current string length by one, and decrement it (this is the same effect as decrementing the str_len variable).
47.4 ARM64: Non-optimizing GCC (Linaro) 4.9

This implementation is straightforward:

Listing 47.4: Non-optimizing GCC (Linaro) 4.9

```
str_trim:
    stp   x29, x30, [sp, -48]!
    add  x29, sp, 0
    str  x0, [x29,24] ; copy input argument into local
        stack
    ldr   x0, [x29,24] ; s
    bl    strlen
    str   x0, [x29,40] ; str_len variable in local stack
    b .L2
; main loop begin
.L5:
    ldrb  w0, [x29,39]
    ; W0=c
    cmp   w0, 13 ; is it '\r'?
    beq   .L3
    ldrb  w0, [x29,39]
    ; W0=c
    cmp   w0, 10 ; is it '\n'?
    bne   .L4 ; goto exit if it is not
.L3:
    ldr   x0, [x29,40]
    ; X0=str_len
    sub   x0, x0, #1
    ; X0=str_len-1
    ldr   x1, [x29,24]
    ; X1=s
    add   x0, x1, x0
    ; X0=s+str_len-1
    strb  wzr, [x0] ; write byte at s+str_len-1
; decrement str_len:
    ldr   x0, [x29,40]
    ; X0=str_len
    sub   x0, x0, #1
    ; X0=str_len-1
    str   x0, [x29,40]
; save X0 (or str_len-1) to local stack
.L2:
    ldr   x0, [x29,40]
    ; str_len==0?
    cmp   x0, xzr
    ; goto exit then
    beq   .L4
```
47.5 ARM64: Optimizing GCC (Linaro) 4.9

This is a more advanced optimization. The first character is loaded at the beginning, and compared against 10 (the LF character). Characters are also loaded in the main loop, for the characters after first one. This is somewhat similar to the 47.3 on page 727 example.

Listing 47.5: Optimizing GCC (Linaro) 4.9

```
str Trim:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0
    str    x19, [sp,16]
    mov    x19, x0
    ; X19 will always hold value of "s"
    bl     strlen
    ; X0=str_len
    cbz    x0, .L9     ; goto L9 (exit) if str_len==0
    sub    x1, x0, #1
    ; X1=X0-1=str_len-1
    add    x3, x19, x1
    ; X3=X19+X1=s+str_len-1
```
CHAPTER 47. STRINGS TRIMMING

ldrb w2, [x19,x1] ; load byte at address X19+X1=s+str_len-1
w2=loaded character

; is it zero? jump to exit then
cbz w2, .L9

; is it '\n'?
omp w2, 10
bne .L15

.L12:
; main loop body. loaded character is always 10 or 13 at this moment!

sub x2, x1, x0
; X2=X1-X0=str_len-1-str_len=-1
add x2, x3, x2
; X2=X3+X2=s+str_len-1+(-1)=s+str_len-2
strb wzr, [x2,1] ; store zero byte at address s+str_len-2+1=s+str_len-1

cbz x1, .L9 ; str_len-1==0? goto exit, if so
sub x1, x1, #1 ; str_len--
ldrb w2, [x19,x1] ; load next character at address X19+X1=s+str_len-1

; is it '\n'?
omp w2, 10
beq .L12 ; jump to begin loop, if its' '\n'

.L15:
; is it '\r'?
omp w2, 13
beq .L12 ; yes, jump to the loop body

.begin

.L9:
; return "s"

mov x0, x19
ldr x19, [sp,16]
ldp x29, x30, [sp], 32
ret

47.6 ARM: Optimizing Keil 6/2013 (ARM mode)

And again, the compiler took advantage of ARM mode's conditional instructions, so the code is much more compact.

Listing 47.6: Optimizing Keil 6/2013 (ARM mode)

str_trim PROC
    PUSH {r4,lr}
    ; R0=s
    MOV r4,r0
    ; R4=s
    BL strlen ; strlen() takes "s" value from R0
    ret
47.7 ARM: Optimizing Keil 6/2013 (Thumb mode)

There are less conditional instructions in Thumb mode, so the code is simpler. But there are is really weird thing with the 0x20 and 0x19 offsets. Why did the Keil compiler do so? Honestly, it's hard to say. Probably, this is a quirk of Keil's optimization process. Nevertheless, the code works correctly.

Listing 47.7: Optimizing Keil 6/2013 (Thumb mode)
CHAPTER 47. STRINGS TRIMMING

47.8 MIPS

Listing 47.8: Optimizing GCC 4.4.5 (IDA)

```
str_trim:
    ; IDA is not aware of local variable names, we gave them
    ; manually:
    saved_GP     =    -0x10
    saved_S0     =    -8
    saved_RA     =    -4

    lui $gp, (__gnu_local_gp >> 16)
    addiu $sp, -0x20
    la  $gp, (__gnu_local_gp & 0xFFFF)
    sw  $ra, 0x20+saved_RA($sp)
    sw  $s0, 0x20+saved_S0($sp)
    sw  $gp, 0x20+saved_GP($sp)

    ; call strlen(). input string address is still
    ; in $a0, strlen() will take it from there:
    lw  $t9, (strlen & 0xFFFF)($gp)
```
or   $at, $zero ; load delay slot, NOP
jalr $t9

; input string address is still in $a0, put it to $s0:
move $s0, $a0  ; branch delay slot

; result of strlen() (i.e., length of string) is in $v0 now;
jump to exit if $v0==0 (i.e., if length of string is 0):
    beqz $v0, exit
    or $at, $zero ; branch delay slot, NOP
    addiu $a1, $v0, -1

; $a1 = $v0-1 = str_len-1
    addu $a1, $s0, $a1

; $a1 = input string address + $a1 = s+str_len-1
; load byte at address $a1:
lb   $a0, 0($a1)
or   $at, $zero ; load delay slot, NOP

; loaded byte is zero? jump to exit if its so':
    beqz $a0, exit
    or $at, $zero ; branch delay slot, NOP
    addiu $v1, $v0, -2

; $v1 = str_len-2
    addu $v1, $s0, $v1

; $v1 = $s0+$v1 = s+str_len-2
    li   $a2, 0xD

; skip loop body:
    b   loc_6C
    li   $a3, 0xA  ; branch delay slot

loc_5C:
; load next byte from memory to $a0:
lb   $a0, 0($v1)
or   $at, $zero ; load delay slot, NOP

; $a1=s+str_len-2
; jump to exit if loaded byte is zero:
    beqz $a0, exit

; decrement str_len:
    addiu $v0, -1  ; branch delay slot
loc_6C:
; at this moment, $a0=loaded byte, $a2=0xD (CR symbol) and $a3=0xA (LF symbol)
; loaded byte is CR? jump to loc_7C then:
    beq   $a0, $a2, loc_7C
addiu $v0, -1  ; branch delay slot

; loaded byte is LF? jump to exit if its not LF':
bne   $a0, $a3, exit
or    $at, $zero ; branch delay slot, NOP
loc_7C:
; loaded byte is CR at this moment
; jump to loc_5c (loop body begin) if str_len (in $v0) is not zero:
bnez $v0, loc_5C
; simultaneously, store zero at that place in memory:
    sb $zero, 0($a1); branch delay slot
; "exit" label was named by me manually:
exit:
    lw $ra, 0x20+saved_RA($sp)
    move $v0, $s0
    lw $s0, 0x20+saved_S0($sp)
    jr $ra
    addiu $sp, 0x20; branch delay slot

Registers prefixed with S- are also called “saved temporaries”, so $S0 value is saved in the local stack and restored upon finish.
Chapter 48

toupper() function

Another very popular function transforms a symbol from lower case to upper case, if needed:

```c
char toupper (char c)
{
    if(c>='a' && c<='z')
        return c-'a'+'A';
    else
        return c;
}
```

The 'a'+'A' expression is left in the source code for better readability, it will be optimized by compiler, of course.

The ASCII code of “a” is 97 (or 0x61), and 65 (or 0x41) for “A”. The difference (or distance) between them in the ASCII table is 32 (or 0x20).

For better understanding, the reader may take a look at the 7-bit standard ASCII table:

1However, to be meticulous, there still could be compilers which can't optimize such expressions and will leave them right in the code.
48.1.x64

48.1.1 Two comparison operations

Non-optimizing MSVC is straightforward: the code checks if the input symbol is in [97..122] range (or in ['a'..'z'] range) and subtracts 32 if it's true. There are also some minor compiler artefact:

```assembly
1  c$ = 8
2  toupper PROC
3  mov     BYTE PTR [rsp+8], cl
4  movsx   eax, BYTE PTR c$[rsp]
5  cmp     eax, 97
6  jl      SHORT $LN2@toupper
7  movsx   eax, BYTE PTR c$[rsp]
8  cmp     eax, 122
9  jg      SHORT $LN2@toupper
10 movsx   eax, BYTE PTR c$[rsp]
11 sub     eax, 32
12 jmp     SHORT $LN3@toupper
13 jmp     SHORT $LN1@toupper ; compiler artefact
14 $LN2@toupper:
15 movzx   eax, BYTE PTR c$[rsp] ; unnecessary casting
16 $LN1@toupper:
17 $LN3@toupper: ; compiler artefact
18 ret     0
19 toupper ENDP
```

It's important to notice that the input byte is loaded into a 64-bit local stack slot at line 3. All the remaining bits ([8..63]) are untouched, i.e., contain some random noise (you'll see it in debugger). All instructions operate only on byte-level, so it's fine. The last MOVZX instruction at line 15 takes the byte from the local stack slot.
and zero-extends it to a \textit{int} 32-bit data type.

Non-optimizing GCC does mostly the same:

\begin{verbatim}
Listing 48.2: Non-optimizing GCC 4.9 (x64)
toupper:
push rbp
mov rbp, rsp
mov eax, edi
mov BYTE PTR [rbp-4], al
cmp BYTE PTR [rbp-4], 96
    jle .L2
cmp BYTE PTR [rbp-4], 122
    jg .L2
movzx eax, BYTE PTR [rbp-4]
sub eax, 32
    jmp .L3
.L2:
    movzx eax, BYTE PTR [rbp-4]
.L3:
pop rbp
ret
\end{verbatim}

48.1.2 One comparison operation

Optimizing MSVC does a better job, it generates only one comparison operation:

\begin{verbatim}
Listing 48.3: Optimizing MSVC 2013 (x64)
toupper PROC
    lea eax, DWORD PTR [rcx-97]
cmp al, 25
    ja SHORT $LN2@toupper
movsx eax, cl
sub eax, 32
    jmp .L3
$LN2@toupper:
    movzx eax, cl
ret 0
toupper ENDP
\end{verbatim}

It was explained earlier how to replace the two comparison operations with a single one: \textit{42.2.1 on page 692}.

We will now rewrite this in C/C++:
int tmp = c - 97;
if (tmp > 25)
    return c;
else
    return c - 32;

The tmp variable should be signed. This makes two subtraction operations in case of a transformation plus one comparison. In contrast the original algorithm uses two comparison operations plus one subtracting.

Optimizing GCC is even better, it gets rid of the jumps (which is good: 33.1 on page 624) by using the CMOVcc instruction:

Listing 48.4: Optimizing GCC 4.9 (x64)

1 toupper:
2     lea    edx, [rdi-97] ; 0x61
3     lea    eax, [rdi-32] ; 0x20
4     cmp    dl, 25
5     cmova  eax, edi
6     ret

At line 3 the code prepares the subtracted value in advance, as if the conversion will always happen. At line 5 the subtracted value in EAX is replaced by the untouched input value if a conversion is not needed. And then this value (of course incorrect) is dropped. Advance subtracting is a price the compiler pays for the absence of conditional jumps.

48.2 ARM

Optimizing Keil for ARM mode also generates only one comparison:

Listing 48.5: Optimizing Keil 6/2013 (ARM mode)

toupper PROC
    SUB     r1, r0, #0x61
    CMP     r1, #0x19
    SUBLS   r0, r0, #0x20
    ANDLS   r0, r0, #0xff
    BX      lr
ENDP

The SUBLS and ANDLS instructions are executed only if the value in R1 is less than 0x19 (or equal). They also do the actual conversion.
Optimizing Keil for Thumb mode generates only one comparison operation as well:

**Listing 48.6: Optimizing Keil 6/2013 (Thumb mode)**

```
toupper PROC
    MOVS r1, r0
    SUBS r1, r1, #0x61
    CMP r1, #0x19
    BHI |L0.14|
    SUBS r0, r0, #0x20
    LSLS r0, r0, #24
    LSRS r0, r0, #24
|L0.14|
    BX lr
ENDP
```

The last two LSLS and LSRS instructions work like `AND reg, 0xFF`: they are equivalent to the C/C++-expression `(i << 24) >> 24`. Seems like that Keil for Thumb mode deduced that two 2-byte instructions are shorter than the code that loads the 0xFF constant into a register plus an AND instruction.

### 48.2.1 GCC for ARM64

**Listing 48.7: Non-optimizing GCC 4.9 (ARM64)**

```
toupper:
    sub sp, sp, #16
    strb w0, [sp,15]
    ldrb w0, [sp,15]
    cmp w0, 96
    bls .L2
    ldrb w0, [sp,15]
    cmp w0, 122
    bhi .L2
    ldrb w0, [sp,15]
    sub w0, w0, #32
    uxtb w0, w0
    b .L3
.L2:
    ldrb w0, [sp,15]
.L3:
    add sp, sp, 16
    ret
```

**Listing 48.8: Optimizing GCC 4.9 (ARM64)**
48.3 Summary

All these compiler optimizations are very popular nowadays and a practicing reverse engineer usually sees such code patterns often.
Chapter 49

Incorrectly disassembled code

Practicing reverse engineers often have to deal with incorrectly disassembled code.

49.1 Disassembling from an incorrect start (x86)

Unlike ARM and MIPS (where any instruction has a length of 2 or 4 bytes), x86 instructions have variable size, so any disassembler that starts in the middle of a x86 instruction may produce incorrect results.

As an example:

```
add    [ebp-31F7Bh], cl
dec    dword ptr [ecx-3277Bh]
dec    dword ptr [ebp-2CF7Bh]
ing    dword ptr [ebx-7A76F33Ch]
fdm    st(4), st
db 0FFh
dec    dword ptr [ecx-21F7Bh]
dec    dword ptr [ecx-22373h]
dec    dword ptr [ecx-2276Bh]
dec    dword ptr [ecx-22B63h]
dec    dword ptr [ecx-22F4Bh]
dec    dword ptr [ecx-23343h]
jmp    dword ptr [esi-74h]
xchg   eax, ebp
cl
std
```

```

cl
```

```

std
```

```
db 0FFh
db 0FFh
```
There are incorrectly disassembled instructions at the beginning, but eventually the disassembler gets on the right track.

### 49.2 How does random noise looks disassembled?

Common properties that can be spotted easily are:

- Unusually big instruction dispersion. The most frequent x86 instructions are PUSH, MOV, CALL, but here we see instructions from all instruction groups: FPU instructions, IN/OUT instructions, rare and system instructions, everything mixed up in one single place.

- Big and random values, offsets and immediates.

- Jumps having incorrect offsets, often jumping in the middle of another instructions.
Listing 49.1: random noise (x86)

```assembly
mov   bl, 0Ch
mov   ecx, 0D38558Dh
mov   eax, ds:2C869A86h
db   67h
mov   dl, 0CCh
insb
movsb
push  eax
xor [edx-53h], ah
fcom qword ptr [edi-45A0EF72h]
op pop esp
pop ss
in eax, dx
dec ebx
push esp
lds esp, [esi-41h]
reft
rcl  dword ptr [eax], cl
mov  cl, 9Ch
mov  ch, 0DFh
push cs
insb
mov  esi, 0D9C65E4Dh
imul ebp, [ecx], 66h
pushf
sal  dword ptr [ebp-64h], cl
sub  eax, 0AC433D64h
out  8Ch, eax
pop ss
sbb  [eax], ebx
aas
xchg cl, [ebx+ebx*4+14B31Eh]
jecxz short near ptr loc_58+1
xor  al, 0C6h
inc edx
db   36h
pusha
stosb
test  [ebx], ebx
sub  al, 0D3h ; 'L'
pop  eax
stosb

loc_58: ; CODE XREF: seg000:0000004A
test  [esi], eax
inc   ebp
```
das
db 64h
pop ecx
hlt

das
pop edx
out 0B0h, al
lodsb
push ebx
cdq
out dx, al
sub al, 0Ah
sti
outsd
add dword ptr [edx], 96FCBE4Bh
and eax, 0E537EE4Fh
inc esp
stosd
cdq
push ecx
in al, 0CBh
mov ds:0D114C45Ch, al
mov esi, 659D1985h

Listing 49.2: random noise (x86-64)
lea esi, [rax+rdx*4+43558D29h]
loc_AF3: ; CODE XREF: seg000:0000000000000B46
rcl byte ptr [rsi+rax*8+29BB423Ah], 1
lea ecx, cs:0FFFFFFFFB2A6780Fh
mov al, 96h
mov ah, 0CEh
push rsp
lodsb byte ptr [esi]

db 2Fh ; /

pop rsp
db 64h
retf 0E993h

cmp ah, [rax+4Ah]
movzx rsi, dword ptr [rbp-25h]
push 4Ah
movzx rdi, dword ptr [rdi+rdx*8]
Listing 49.3: random noise (ARM (ARM mode))

BLNE 0xFE16A9D8
BGE 0x1634D0C
SVCCS 0x450685
STRNVT R5, [PC],#-0x964
LDCGE p6, c14, [R0],#0x168
STCCSL p9, c9, [LR],#0x14C
CMNHIP PC, R10,LSL#22
FLDMIADNV LR!, {D4}
MCR p5, 2, R2,c15,c6, 4
BLGE 0x1139558
BLGT 0xFF9146E4
STRNEB R5, [R4],#0xCA2
STMNEIB R5, {R0,R4,R6,R7,R9-SP,PC}
STMIA R8, {R0,R2-R4,R7,R8,R10,SP,LR}^
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRB SP, [R8], PC, ROR#18</td>
<td>0x9B0362</td>
</tr>
<tr>
<td>LDCCS p9, c13, [R6, #0x1BC]</td>
<td>0xA73173</td>
</tr>
<tr>
<td>LDRGE R8, [R9, #0x66E]</td>
<td></td>
</tr>
<tr>
<td>STRNEB R5, [R8], #-0x8C3</td>
<td></td>
</tr>
<tr>
<td>STCCSL p15, c9, [R7, #-0x84]</td>
<td></td>
</tr>
<tr>
<td>RSBLs LR, R2, R11, ASR LR</td>
<td></td>
</tr>
<tr>
<td>SVCGE 0x9B0362</td>
<td></td>
</tr>
<tr>
<td>SVCGET 0x9B0362</td>
<td></td>
</tr>
<tr>
<td>STMNEDB R11!, {R0, R1, R4-R6, R8, R10, R11, SP}</td>
<td></td>
</tr>
<tr>
<td>STR R0, [R3], #-0xCE4</td>
<td></td>
</tr>
<tr>
<td>LDCGT p15, c8, [R1, #0x2CC]</td>
<td></td>
</tr>
<tr>
<td>LDRCCB R1, [R11], -R7, ROR#30</td>
<td></td>
</tr>
<tr>
<td>BLLT 0xFED9D58C</td>
<td></td>
</tr>
<tr>
<td>BL 0x13E60F4</td>
<td></td>
</tr>
<tr>
<td>LDMVsIB R3!, {R1, R4-R7}^</td>
<td></td>
</tr>
<tr>
<td>USATNE R10, #7, SP, LSL#11</td>
<td></td>
</tr>
<tr>
<td>LDRGEB LR, [R1], #0xE56</td>
<td></td>
</tr>
<tr>
<td>STRPLT R9, [LR], #0x567</td>
<td></td>
</tr>
<tr>
<td>LDRLT R11, [R1], #-0x29B</td>
<td></td>
</tr>
<tr>
<td>Svcnv 0x12DB29</td>
<td></td>
</tr>
<tr>
<td>MVNNVS R5, SP, LSL#25</td>
<td></td>
</tr>
<tr>
<td>LDCL p8, c14, [R12, #-0x288]</td>
<td></td>
</tr>
<tr>
<td>STCNEl p2, c6, [R6, #-0xBC]!</td>
<td></td>
</tr>
<tr>
<td>Svcnv 0x2E5A2F</td>
<td></td>
</tr>
<tr>
<td>BLX 0x1A8C97E</td>
<td></td>
</tr>
<tr>
<td>TEQGE R3, #0x1100000</td>
<td></td>
</tr>
<tr>
<td>STMLSIA R6, {R3, R6, R10, R11, SP}</td>
<td></td>
</tr>
<tr>
<td>BICPLS R12, R2, #0x5800</td>
<td></td>
</tr>
<tr>
<td>BNE 0x7CC408</td>
<td></td>
</tr>
<tr>
<td>TEQGE R2, R4, LSL#20</td>
<td></td>
</tr>
<tr>
<td>SUBS R1, R11, #0x28C</td>
<td></td>
</tr>
<tr>
<td>BICVS R3, R12, R7, ASR R0</td>
<td></td>
</tr>
<tr>
<td>LDRMI R7, [LR], R3, LSL#21</td>
<td></td>
</tr>
<tr>
<td>BLMi 0x1A79234</td>
<td></td>
</tr>
<tr>
<td>STMVCDB R6, {R0-R3, R6, R7, R10, R11}</td>
<td></td>
</tr>
<tr>
<td>EORMi R12, R6, #0xC5</td>
<td></td>
</tr>
<tr>
<td>MCRRCS p1, 0xF, R1, R3, c2</td>
<td></td>
</tr>
</tbody>
</table>

Listing 49.4: random noise (ARM (Thumb mode))

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSRS R3, R6, #0x12</td>
<td></td>
</tr>
<tr>
<td>LDRH R1, [R7, #0x2C]</td>
<td></td>
</tr>
<tr>
<td>SUBS R0, #0x55 ; 'U'</td>
<td></td>
</tr>
<tr>
<td>ADR R1, loc_3C</td>
<td></td>
</tr>
<tr>
<td>LDR R2, [SP, #0x218]</td>
<td></td>
</tr>
<tr>
<td>CMP R4, #0x86</td>
<td></td>
</tr>
<tr>
<td>SXTB R7, R4</td>
<td></td>
</tr>
<tr>
<td>LDR R4, [R1, #0x4C]</td>
<td></td>
</tr>
</tbody>
</table>
Listing 49.5: random noise (MIPS little endian)

```
lw   $t9, 0xCB3($t5)
sb   $t5, 0x3855($t0)
sltiu $a2, $a0, -0x657A
ldr   $t4, -0x4D99($a2)
daddi $s0, $s1, 0x50A4
lw   $s7, -0x2353($s4)
bgtzl  $a1, 0x17C5C

.byte 0x17
.byte 0xED
.byte 0x4B  # K
.byte 0x54  # T

lwc2   $31, 0x66C5($sp)
luw   $s1, 0x10D3($a1)
ldr   $t6, -0x204B($zero)
lwc1   $f30, 0x4DBE($s2)
```
daddiu $t1, $s1, 0x6BD9
lwu $s5, -0x2C64($v1)
cop0 0x13D642D
bne $gp, $t4, 0xFFFF9EF0
lh $ra, 0x1819($s1)
sdl $fp, -0x6474($t8)
jal 0x78C0050
ori $v0, $s2, 0xC634
blez $gp, 0xFFFEA9D4
swl $t8, -0x2CD4($s2)
sltiu $a1, $k0, 0x685
sdc1 $f15, 0x5964($at)
sw $s0, -0x19A6($a1)
sltiu $t6, $a3, -0x66AD
lb $t7, -0x4F6($t3)
sd $fp, 0x4B02($a1)

It is also important to keep in mind that cleverly constructed unpacking and decryption code (including self-modifying) may looks like noise as well, but still execute correctly.
Chapter 50

Obfuscation

The obfuscation is an attempt to hide the code (or its meaning) from reverse engineers.

50.1 Text strings

As we saw in (57 on page 926), text strings may be really helpful. Programmers who are aware of this try to hide them, making it impossible to find the string in IDA or any hex editor.

Here is the simplest method.

This is how the string can be constructed:

```
mov byte ptr [ebx], 'h'
mov byte ptr [ebx+1], 'e'
mov byte ptr [ebx+2], 'l'
mov byte ptr [ebx+3], 'l'
mov byte ptr [ebx+4], 'o'
mov byte ptr [ebx+5], '
'mov byte ptr [ebx+6], 'w'
mov byte ptr [ebx+7], 'o'
mov byte ptr [ebx+8], 'r'
mov byte ptr [ebx+9], 'l'
mov byte ptr [ebx+10], 'd'
```

The string is also can be compared with another one like this:

```
mov ebx, offset username
cmp byte ptr [ebx], 'j'
```
In both cases, it is impossible to find these strings straightforwardly in a hex editor.

By the way, this is a way to work with the strings when it is impossible to allocate space for them in the data segment, for example in a PIC or in shellcode.

Another method is to use sprintf() for the construction:

```c
sprintf(buf, "%s%c%s%c%s", "hel","l","o w","o","rld");
```

The code looks weird, but as a simple anti-reversing measure, it may be helpful.

Text strings may also be present in encrypted form, then every string usage is to be preceded by a string decrypting routine. For example: 78.2 on page 1085.

## 50.2 Executable code

### 50.2.1 Inserting garbage

Executable code obfuscation implies inserting random garbage code between real one, which executes but does nothing useful.

A simple example:

### Listing 50.1: original code

```assembly
add eax, ebx
mul ecx
```

### Listing 50.2: obfuscated code

```assembly
xor esi, 011223344h ; garbage
add esi, eax ; garbage
add eax, ebx ; garbage
mov edx, eax ; garbage
shl edx, 4 ; garbage
mul ecx ; garbage
xor esi, ecx ; garbage
```
Here the garbage code uses registers which are not used in the real code (ESI and EDX). However, the intermediate results produced by the real code may be used by the garbage instructions for some extra mess—why not?

50.2.2 Replacing instructions with bloated equivalents

- MOV op1, op2 can be replaced by the PUSH op2 / POP op1 pair.
- JMP label can be replaced by the PUSH label / RET pair. IDA will not show the references to the label.
- CALL label can be replaced by the following instructions triplet: PUSH label_after_CALL_instruction / PUSH label / RET.
- PUSH op can also be replaced with the following instructions pair: SUB ESP, 4 (or 8) / MOV [ESP], op.

50.2.3 Always executed/never executed code

If the developer is sure that ESI at always 0 at that point:

```
mov   esi, 1
...   ; some code not touching ESI
dec   esi
...   ; some code not touching ESI
cmp   esi, 0
jz    real_code
; fake luggage
real_code:
```

The reverse engineer needs some time to get into it.

This is also called an opaque predicate.

Another example (and again, the developer is sure that ESI is always zero):

```
add   eax, ebx   ; real code
mul   ecx        ; real code
add   eax, esi   ; opaque predicate. XOR, AND or SHL, etc, can be here instead of ADD.
```

50.2.4 Making a lot of mess
instruction 1
instruction 2
instruction 3

Can be replaced with:

begin:       jmp   ins1_label

ins2_label:  instruction 2
             jmp   ins3_label

ins3_label:  instruction 3
             jmp   exit:

ins1_label:  instruction 1
             jmp   ins2_label

exit:

50.2.5 Using indirect pointers

dummy_data1   db  100h dup (0)
message1      db  'hello world',0
dummy_data2   db  200h dup (0)
message2      db  'another message',0

func           proc
...            
             mov   eax, offset dummy_data1 ; PE or ELF
             \  reloc here
             add   eax, 100h
             push  eax
             call  dump_string
             ...
             mov   eax, offset dummy_data2 ; PE or ELF
             \  reloc here
             add   eax, 200h
             push  eax
             call  dump_string
             ...
func           endp

IDA will show references only to dummy_data1 and dummy_data2, but not to the text strings.
Global variables and even functions may be accessed like that.

## 50.3 Virtual machine / pseudo-code

A programmer can construct his/her own PL or ISA and interpreter for it. (Like the pre-5.0 Visual Basic, .NET or Java machines). The reverse engineer will have to spend some time to understand the meaning and details of all of the ISA's instructions. Probably, he/she will also have to write a disassembler/decompiler of some sort.

## 50.4 Other things to mention

My own (yet weak) attempt to patch the Tiny C compiler to produce obfuscated code: [http://go.yurichev.com/17220](http://go.yurichev.com/17220).

Using the MOV instruction for really complicated things: [Dol13].

## 50.5 Exercise

- [http://challenges.re/29](http://challenges.re/29)
Chapter 51

C++

51.1 Classes

51.1.1 A simple example

Internally, the representation of C++ classes is almost the same as the structures. Let’s try an example with two variables, two constructors and one method:

```c
#include <stdio.h>

class c
{
  private:
    int v1;
    int v2;
  public:
    c() // default ctor
    {
      v1=667;
      v2=999;
    }

    c(int a, int b) // ctor
    {
      v1=a;
      v2=b;
    }

    void dump()
};
```
Here's what's going on. For each object (instance of class c) 8 bytes are allocated, exactly the size needed to store the 2 variables.
For \( c1 \) a default argumentless constructor \( ??0c@@QAE@XZ \) is called. For \( c2 \) another constructor \( ??0c@@QAE@HH@Z \) is called and two numbers are passed as arguments.

A pointer to the object (\textit{this} in C++ terminology) is passed in the ECX register. This is called thiscall (\textit{51.1.1})—the method for passing a pointer to the object.

MSVC does it using the ECX register. Needless to say, it is not a standardized method, other compilers can do it differently, e.g., via the first function argument (like GCC).

Why do these functions have such odd names? That's \textit{name mangling}.

A C++ class may contain several methods sharing the same name but having different arguments—that is polymorphism. And of course, different classes may have their own methods with the same name.

\textit{Name mangling} enable us to encode the class name + method name + all method argument types in one ASCII string, which is then used as an internal function name. That's all because neither the linker, nor the DLL OS loader (mangled names may be among the DLL exports as well) knows anything about C++ or \textit{OOP}\(^1\).

The \textit{dump()} function is called two times.

Now let's see the constructors’ code:

\begin{verbatim}
 Listing 51.2: MSVC

_this$ = -4 ; size = 4
??0c@@QAE@XZ PROC ; c::c, COMDAT
 ; _this$ = ecx
 push ebp
 mov ebp, esp
 push ecx
 mov DWORD PTR _this$[ebp], ecx
 mov eax, DWORD PTR _this$[ebp]
 mov DWORD PTR [eax], 667
 mov ecx, DWORD PTR _this$[ebp]
 mov DWORD PTR [ecx+4], 999
 mov eax, DWORD PTR _this$[ebp]
 mov esp, ebp
 pop ebp
 ret 0
??0c@@QAE@XZ ENDP ; c::c

_this$ = -4 ; size = 4
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
\end{verbatim}

\(^1\)Object-Oriented Programming
The constructors are just functions, they use a pointer to the structure in ECX, copying the pointer into their own local variable, however, it is not necessary.

From the C++ standard [ISO13, p. 12.1] we know that constructors are not required to return any values. In fact, internally, the constructors return a pointer to the newly created object, i.e., this.

Now the dump() method:

Listing 51.3: MSVC

```assembly
; _this$ = ecx
    push ebp
    mov ebp, esp
    push ecx
    mov DWORD PTR _this$[ebp], ecx
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR _a$[ebp]
    mov DWORD PTR [eax], ecx
    mov edx, DWORD PTR _this$[ebp]
    mov eax, DWORD PTR _b$[ebp]
    mov DWORD PTR [edx+4], eax
    mov eax, DWORD PTR _this$[ebp]
    mov esp, ebp
    pop ebp
    ret 8
??0c@@QAE@HH@Z ENDP ; c::c
```

```assembly
??0c@@QAE@HH@Z PROC ; c::dump, COMDAT
    ; _this$ = ecx
        push ebp
    mov ebp, esp
        push ecx
        mov DWORD PTR _this$[ebp], ecx
        mov eax, DWORD PTR _this$[ebp]
        mov ecx, DWORD PTR [eax+4]
        push ecx
        mov edx, DWORD PTR _this$[ebp]
        mov eax, DWORD PTR [edx]  
        push eax
        push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
        call _printf
        add esp, 12
        mov esp, ebp
        pop ebp
        ret 0
??dump@c@@QAEXXZ ENDP ; c::dump
```

758
Simple enough: dump() takes a pointer to the structure that contains the two int's from ECX, takes both values from it and passes them to printf().

The code is much shorter if compiled with optimizations (/Ox):

Listing 51.4: MSVC

??0c@@QAE@XZ PROC ; c::c, COMDAT
 ; _this$ = ecx
   mov eax, ecx
   mov DWORD PTR [eax], 667
   mov DWORD PTR [eax+4], 999
   ret 0
??0c@@QAE@XZ ENDP ; c::c

_a$ = 8 ; size = 4
_b$ = 12 ; size = 4

??0c@@QAE@HH@Z PROC ; c::c, COMDAT
 ; _this$ = ecx
   mov edx, DWORD PTR _b$[esp-4]
   mov eax, ecx
   mov ecx, DWORD PTR _a$[esp-4]
   mov DWORD PTR [eax], ecx
   mov DWORD PTR [eax+4], edx
   ret 8
??0c@@QAE@HH@Z ENDP ; c::c

?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
 ; _this$ = ecx
   mov eax, DWORD PTR [ecx+4]
   mov ecx, DWORD PTR [ecx]
   push eax
   push ecx
   push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
   call _printf
   add esp, 12
   ret 0
?dump@c@@QAEXXZ ENDP ; c::dump

That's all. The other thing we need to note is that the stack pointer was not corrected with add esp, X after the constructor was called. At the same time, the constructor has ret 8 instead of RET at the end.

This is all because the thiscall (51.1.1 on page 757) calling convention is used here, which together with the stdcall (64.2 on page 952) method offers the callee to correct the stack instead of the caller. The ret x instruction adds X to the value in ESP, then passes the control to the caller function.

See also the section about calling conventions (64 on page 952).
It also has to be noted that the compiler decides when to call the constructor and destructor—but we already know that from the C++ language basics.

**MSVC—x86-64**

As we already know, the first 4 function arguments in x86-64 are passed in RCX, RDX, R8 and R9 registers, all the rest—via the stack. Nevertheless, the this pointer to the object is passed in RCX, the first argument of the method in RDX, etc. We can see this in the c(int a, int b) method internals:

Listing 51.5: Optimizing MSVC 2012 x64

```assembly
; void dump()

?dump@QEAAXXXZ PROC ; c::dump
  mov   r8d, DWORD PTR [rcx+4]
  mov   edx, DWORD PTR [rcx]
  lea   rcx, OFFSET FLAT:?_C@_07NJBDCIEC?CFd?DL?5?CFd
  jmp   ?6?$AA@ ; '%d; %d'
  jmp   printf
?dump@QEAAXXXZ ENDP ; c::dump

; c(int a, int b)

??0c@QEAAXH@Z PROC ; c::c
  mov   DWORD PTR [rcx], edx    ; 1st argument: a
  mov   DWORD PTR [rcx+4], r8d  ; 2nd argument: b
  mov   rax, rcx
  ret   0
??0c@QEAAXH@Z ENDP ; c::c

; default ctor

??0c@QEAAXZ PROC ; c::c
  mov   DWORD PTR [rcx], 667
  mov   DWORD PTR [rcx+4], 999
  mov   rax, rcx
  ret   0
??0c@QEAAXZ ENDP ; c::c
```

The int data type is still 32-bit in x64, so that is why 32-bit register parts are used here.

We also see JMP printf instead of RET in the dump() method, that hack we already saw earlier: 13.1.1 on page 211.

---

2 Apparently, for easier porting of 32-bit C/C++ code to x64
It is almost the same story in GCC 4.4.1, with a few exceptions.

Listing 51.6: GCC 4.4.1

```assembly
public main
main proc near

var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_18 = dword ptr -18h
var_10 = dword ptr -10h
var_8 = dword ptr -8

push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 20h
lea eax, [esp+20h+var_8]
mov [esp+20h+var_20], eax
call _ZN1cC1Ev
mov [esp+20h+var_18], 6
mov [esp+20h+var_1C], 5
lea eax, [esp+20h+var_10]
mov [esp+20h+var_20], eax
call _ZN1cC1Eii
lea eax, [esp+20h+var_8]
mov [esp+20h+var_20], eax
call _ZN1c4dumpEv
lea eax, [esp+20h+var_10]
mov [esp+20h+var_20], eax
call _ZN1c4dumpEv
mov eax, 0
leave
retn
main endp
```

Here we see another name mangling style, specific to GNU. It can also be noted that the pointer to the object is passed as the first function argument—invisible to programmer, of course.

First constructor:

```assembly
public _ZN1cC1Ev ; weak
_ZN1cC1Ev proc near ; CODE XREF: main+10
```

---

3There is a good document about the various name mangling conventions in different compilers: [Fog14].
arg_0 = dword ptr 8

push ebp
mov ebp, esp
mov eax, [ebp+arg_0]
mov dword ptr [eax], 667
mov eax, [ebp+arg_0]
mov dword ptr [eax+4], 999
pop ebp
retn

_ITN1cC1Ev endp

It just writes two numbers using the pointer passed in the first (and only) argument.

Second constructor:

public _ZN1cC1Eii
_ZN1cC1Eii proc near

arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h

push ebp
mov ebp, esp
mov eax, [ebp+arg_0]
mov edx, [ebp+arg_4]
mov [eax], edx
mov eax, [ebp+arg_0]
mov edx, [ebp+arg_8]
mov [eax+4], edx
pop ebp
retn

_ZN1cC1Eii endp

This is a function, the analog of which can look like this:

void ZN1cC1Eii (int *obj, int a, int b)
{
    *obj=a;
    *(obj+1)=b;
};

...and that is completely predictable.

Now the dump() function:
This function in its *internal representation* has only one argument, used as pointer to the object (*this*).

This function could be rewritten in C like this:

```c
void ZN1c4dumpEv (int *obj)
{
    printf ("%d; %d\n", *obj, *(obj+1));
};
```

Thus, if we base our judgment on these simple examples, the difference between MSVC and GCC is the style of the encoding of function names (*name mangling*) and the method for passing a pointer to the object (via the ECX register or via the first argument).

**GCC—x86-64**

The first 6 arguments, as we already know, are passed in the RDI, RSI, RDX, RCX, R8 and R9 [Mit13] registers, and the pointer to *this* via the first one (RDI) and that is what we see here. The int data type is also 32-bit here. The JMP instead of RET *hack* is also used here.
Inherited classes are similar to the simple structures we already discussed, but extended in inheritable classes.

Let’s take this simple example:

```cpp
#include <stdio.h>

class object
{
public:
    int color;
    object() { };
    object (int color) { this->color=color; };
    void print_color() { printf ("color=%d\n", color); };
};

class box : public object
{
private:
    int width, height, depth;
};
```

```c
; default ctor

_FUNCTION_NAME(cC2E)
    mov DWORD PTR [rdi], 667
    mov DWORD PTR [rdi+4], 999
    ret

; c(int a, int b)

_FUNCTION_NAME(cC2Ei)
    mov DWORD PTR [rdi], esi
    mov DWORD PTR [rdi+4], edx
    ret

; dump()

_FUNCTION_NAME(c4dumpE)
    mov edx, DWORD PTR [rdi+4]
    mov esi, DWORD PTR [rdi]
    xor eax, eax
    mov edi, OFFSET FLAT:.LC0 ; "%d; %d\n"
    jmp printf
```
public:
    box(int color, int width, int height, int depth)
    {
        this->color=color;
        this->width=width;
        this->height=height;
        this->depth=depth;
    }
    void dump()
    {
        printf("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, depth);
    }
};

class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    }
    void dump()
    {
        printf("this is sphere. color=%d, radius=%d\n", color, radius);
    }
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    b.print_color();
    s.print_color();

    b.dump();
    s.dump();

    return 0;
};

Let's investigate the generated code of the dump() functions/methods and also
object::print_color(), and see the memory layout for the structures-objects (for 32-bit code).

So, here are the dump() methods for several classes, generated by MSVC 2008 with /Ox and /Ob0 options 4

Listing 51.8: Optimizing MSVC 2008 /Ob0

??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ⤦
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx]
push eax

; 'color=%d', 0aH, 00H
  push OFFSET ??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@
call _printf
  add esp, 8
ret 0
?print_color@object@@QAEXXZ ENDP ; object::print_color

Listing 51.9: Optimizing MSVC 2008 /Ob0

?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx+12]
    mov edx, DWORD PTR [ecx+8]
push eax
    mov eax, DWORD PTR [ecx+4]
    push edx
    push eax
    push ecx

; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, ⤦
  00H ; `string'
  push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
call _printf
  add esp, 20
ret 0
?dump@box@@QAEXXZ ENDP ; box::dump

Listing 51.10: Optimizing MSVC 2008 /Ob0

4The /Ob0 option stands for disabling inline expansion since function inlining can make our experiment harder
So, here is the memory layout:

(base class object)

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>int color</td>
</tr>
</tbody>
</table>

(inherited classes)

**box:**

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>int color</td>
</tr>
<tr>
<td>+0x4</td>
<td>int width</td>
</tr>
<tr>
<td>+0x8</td>
<td>int height</td>
</tr>
<tr>
<td>+0xC</td>
<td>int depth</td>
</tr>
</tbody>
</table>

**sphere:**

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>int color</td>
</tr>
<tr>
<td>+0x4</td>
<td>int radius</td>
</tr>
</tbody>
</table>

Let's see main() function body:

Listing 51.11: Optimizing MSVC 2008 /Ob0
The inherited classes must always add their fields after the base classes' fields, to make it possible for the base class methods to work with their own fields.

When the `object::print_color()` method is called, a pointers to both the `box` and `sphere` objects are passed as this, and it can work with these objects easily since the `color` field in these objects is always at the pinned address (at offset +0x0).

It can be said that the `object::print_color()` method is agnostic in relation to the input object type as long as the fields are pinned at the same addresses, and this condition is always true.

And if you create inherited class of the `box` class, the compiler will add the new fields after the `depth` field, leaving the `box` class fields at the pinned addresses.

Thus, the `box::dump()` method will work fine for accessing the `color/width/height/depths` fields, which are always pinned at known addresses.

The code generated by GCC is almost the same, with the sole exception of passing the `this` pointer (as it was explained above, it is passed as the first argument instead of using the ECX register).

### 51.1.3 Encapsulation

Encapsulation is hiding the data in the `private` sections of the class, e.g. to allow access to them only from this class methods.

However, are there any marks in code the about the fact that some field is private and some other—not?
No, there are no such marks.

Let's try this simple example:

```c
#include <stdio.h>

class box
{
    private:
        int color, width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        }
        void dump()
        {
            printf("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, depth);
        }
};
```

Let's compile it again in MSVC 2008 with /Ox and /Ob0 options and see the `box::dump()` method code:

```assembly
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
 ; _this$ = ecx
    mov eax, DWORD PTR [ecx+12]
    mov edx, DWORD PTR [ecx+8]
    push eax
    mov eax, DWORD PTR [ecx+4]
    mov ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
 ; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, \n    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?\n    $CFd?0?5width?$DN?$CFd?0@
    call _printf
    add esp, 20
    ret 0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Here is a memory layout of the class:
All fields are private and not allowed to be accessed from any other function, but knowing this layout, can we create code that modifies these fields?

To do this we'll add the `hack_oop_encapsulation()` function, which is not going to compile if it looked like this:

```cpp
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // that code cant be compiled:
    // "error C2248: 'box::width' : cannot access private member declared in class 'box'
};
```

Nevertheless, if we cast the `box` type to a *pointer to an int array*, and we modify the array of *int*-s that we have, we can succeed.

```cpp
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(o);
    ptr_to_object[1]=123;
};
```

This function's code is very simple—it can be said that the function takes a pointer to an array of *int*-s for input and writes *123* to the second *int*:

```assembly
?hack_oop_encapsulation@@YAXPAVbox@@Z PROC 
	; 
	; hack_oop_encapsulation
    mov eax, DWORD PTR _o$[esp-4]
    mov DWORD PTR [eax+4], 123
    ret 0
?hack_oop_encapsulation@@YAXPAVbox@@Z ENDP 
	; hack_oop_encapsulation
```

Let's check how it works:

```cpp
int main()
{
    box b(1, 10, 20, 30);
    b.dump();
}
```
Let’s run:

```
this is box. color=1, width=10, height=20, depth=30
this is box. color=1, width=123, height=20, depth=30
```

We see that the encapsulation is just protection of class fields only in the compilation stage. The C++ compiler is not allowing the generation of code that modifies protected fields straightforwardly, nevertheless, it is possible with the help of dirty hacks.

### 51.1.4 Multiple inheritance

Multiple inheritance is creating a class which inherits fields and methods from two or more classes.

Let’s write a simple example again:

```c++
#include <stdio.h>

class box {
    public:
        int width, height, depth;
        box() { };
        box(int width, int height, int depth)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. width=%d, height=%d, depth=%d\n", width, height, depth);
        };
        int get_volume()
        {
            return width * height * depth;
        };
};
```


```cpp
class solid_object
{
  public:
    int density;
  solid_object() {}
  solid_object(int density)
  {
    this->density=density;
  }
  int get_density()
  {
    return density;
  }
  void dump()
  {
    printf("this is solid_object. density=%d\n", density);
  }
};

class solid_box: box, solid_object
{
  public:
    solid_box (int width, int height, int depth, int density)
    {
      this->width=width;
      this->height=height;
      this->depth=depth;
      this->density=density;
    }
    void dump()
    {
      printf("this is solid_box. width=%d, height=%d, depth=%d, density=%d\n", width, height, depth, density);
    }
    int get_weight() { return get_volume() * get_density(); }
};

int main()
{
  box b(10, 20, 30);
  solid_object so(100);
  solid_box sb(10, 20, 30, 3);
};
```
Let's compile it in MSVC 2008 with the \_/0x and /Ob0 options and see the code of box::dump(), solid_object::dump() and solid_box::dump():

Listing 51.12: Optimizing MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
 ;_this$ = ecx
 mov eax, DWORD PTR [ecx+8]
 mov edx, DWORD PTR [ecx+4]
 push eax
 mov eax, DWORD PTR [ecx]
 push edx
 push eax

; 'this is box. width=%d, height=%d, depth=%d', 0aH, 00H
 push OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?width?$DN?$CFd
 push OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?depth?$DN?$CFd
 call _printf
 add esp, 16
 ret 0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Listing 51.13: Optimizing MSVC 2008 /Ob0

```
?dump@solid_object@@QAEXXZ PROC ; solid_object::dump, COMDAT
 ;_this$ = ecx
 mov eax, DWORD PTR [ecx]
 push eax

; 'this is solid_object. density=%d', 0aH
 push OFFSET ??_C@_0CC@KICFJINL@this?5is?5solid_object?4?density?$DN?$CFd
 call _printf
 add esp, 8
 ret 0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump
```

Listing 51.14: Optimizing MSVC 2008 /Ob0

```
?dump@solid_box@@QAEXXZ PROC ; solid_box::dump, COMDAT
 ;_this$ = ecx
```
So, the memory layout for all three classes is:

**box class:**

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>width</td>
</tr>
<tr>
<td>+0x4</td>
<td>height</td>
</tr>
<tr>
<td>+0x8</td>
<td>depth</td>
</tr>
</tbody>
</table>

**solid_object class:**

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>density</td>
</tr>
</tbody>
</table>

It can be said that the **solid_box** class memory layout is **united**:

**solid_box class:**

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>width</td>
</tr>
<tr>
<td>+0x4</td>
<td>height</td>
</tr>
<tr>
<td>+0x8</td>
<td>depth</td>
</tr>
<tr>
<td>+0xC</td>
<td>density</td>
</tr>
</tbody>
</table>

The code of the box::get_volume() and solid_object::get_density() methods is trivial:

Listing 51.15: Optimizing MSVC 2008 /Ob0

```
;get_volume@box@@QAEHXZ PROC ; box::get_volume, COMDAT
;_this$ = ecx
    mov eax, DWORD PTR [ecx+8]
    imul eax, DWORD PTR [ecx+4]
    imul eax, DWORD PTR [ecx]
```

774
But the code of the `solid_box::get_weight()` method is much more interesting:

```assembly
Listing 51.17: Optimizing MSVC 2008 /Ob0

?get_weight@solid_box@@QAEHXZ PROC ; solid_box::get_weight, COMDAT;
    _this$ = ecx
    push esi
    mov esi, ecx
    push edi
    lea ecx, DWORD PTR [esi+12]
    call ?get_density@solid_object@@QAEHXZ ; solid_object::get_density
    mov ecx, esi
    mov edi, eax
    call ?get_volume@box@@QAEHXZ ; box::get_volume
    imul eax, edi
    pop edi
    pop esi
    ret 0
?get_weight@solid_box@@QAEHXZ ENDP ; solid_box::get_weight
```

`get_weight()` just calls two methods, but for `get_volume()` it just passes pointer to this, and for `get_density()` it passes a pointer to this incremented by 12 (or 0xC) bytes, and there, in the `solid_box` class memory layout, the fields of the `solid_object` class start.

Thus, the `solid_object::get_density()` method will believe like it is dealing with the usual `solid_object` class, and the `box::get_volume()` method will work with its three fields, believing this is just the usual object of class `box`.

Thus, we can say, an object of a class, that inherits from several other classes, is representing in memory as a *united* class, that contains all inherited fields. And each inherited method is called with a pointer to the corresponding structure’s part.
51.1.5 Virtual methods

Yet another simple example:

```cpp
#include <stdio.h>

class object
{
    public:
        int color;
    object() { }
    object (int color) { this->color=color; }
    virtual void dump()
    {
        printf ("color=%d\n", color);
    }
};

class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        }
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d
ایکی چئی، depth=%d\n", color, width, height, depth);
        }
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
            this->radius=radius;
        }
        void dump()
        {
```
Class object has a virtual method dump() that is being replaced in the inheriting box and sphere classes.

If we are in an environment where it is not known the type of an object, as in the main() function in example, where the virtual method dump() is called, the information about its type must be stored somewhere, to be able to call the relevant virtual method.

Let’s compile it in MSVC 2008 with the /Ox and /Ob0 options and see the code of main():

```assembly
_s$ = -32 ; size = 12
_b$ = -20 ; size = 20
_main PROC
    sub esp, 32
    push 30
    push 20
    push 10
    push 1
    lea ecx, DWORD PTR _b$[esp+48]
    call ??0box@@QAE@HH@Z ; box::box
    push 40
    push 2
    lea ecx, DWORD PTR _s$[esp+40]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    mov eax, DWORD PTR _b$[esp+32]
    mov edx, DWORD PTR [eax]
    lea ecx, DWORD PTR _b$[esp+32]
    call edx
```
A pointer to the `dump()` function is taken somewhere from the object. Where could we store the address of the new method? Only somewhere in the constructors: there is no other place since nothing else is called in the `main()` function.  

Let’s see the code of the constructor of the `box` class:

```plaintext
??_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@ ; box `RTTI TypeDescriptor'
    DD 00H
    DB `.?AVbox@@', 00H

??_R1A@?0A@EA@box@@8 DD FLAT:??_R0?AVbox@@@8 ; box::`RTTI BaseClass Descriptor at (0,-1,0,64)'
    DD 01H
    DD 00H
    DD 0fffffffH
    DD 00H
    DD 040H
    DD FLAT:??_R3box@@8

??_R2box@@8 DD FLAT:??_R1A@?0A@EA@box@@8 ; box::`RTTI BaseClass Array'
    DD FLAT:??_R1A@?0A@EA@object@@8

??_R3box@@8 DD 00H ; box::`RTTI Class Hierarchy Descriptor'
    DD 00H
    DD 02H
    DD FLAT:??_R2box@@8

??_R4box@@6B@ DD 00H ; box::`RTTI Complete Object Locator'
    DD 00H
    DD 00H
    DD FLAT:??_R0?AVbox@@@8
    DD FLAT:??_R3box@@8

??_7box@@6B@ DD FLAT:??_R4box@@6B@ ; box::`vftable'
    DD FLAT:??dump@box@@UAEXXZ
```

---

5 You can read more about pointers to functions in the relevant section: (23 on page 527)
Here we see a slightly different memory layout: the first field is a pointer to some table box::`vftable' (the name was set by the MSVC compiler).

In this table we see a link to a table named box::`RTTI Complete Object Locator' and also a link to the box::dump() method. These are called virtual methods table and RTTI\(^6\). The table of virtual methods contains the addresses of methods and the RTTI table contains information about types. By the way, the RTTI tables are used while calling dynamic_cast and typeid in C++. You can also see here the class name as a plain text string. Thus, a method of the base object class may call the virtual method object::dump(), which in turn will call a method of an inherited class, since that information is present right in the object's structure.

Some additional CPU time is needed for doing look-ups in these tables and finding the right virtual method address, thus virtual methods are widely considered as slightly slower than common methods.

In GCC-generated code the RTTI tables are constructed slightly differently.

\(^6\)Run-time type information
51.2 ostream

Let's start again with a “hello world” example, but now we are going to use ostream:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Almost any C++ textbook tells us that the `<<` operation can be replaced (overloaded) for other types. That is what is done in ostream. We see that `operator<<` is called for ostream:

Listing 51.18: MSVC 2012 (reduced listing)

```
$SG37112 DB 'Hello, world!', 0aH, 00H

_main PROC
    push OFFSET $SG37112
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<\n        std::char_traits<char> >
    add esp, 8
    xor eax, eax
    ret 0
_main ENDP
```

Let's modify the example:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, " << "world!\n";
}
```

And again, from many C++ textbooks we know that the result of each `operator<<` in ostream is forwarded to the next one. Indeed:

Listing 51.19: MSVC 2012

```
$SG37112 DB 'world!', 0aH, 00H
$SG37113 DB 'Hello, ', 00H
```
If we would rename `operator<<` method name to `f()`, that code will looks like:

```cpp
f(f(std::cout, "Hello, "), "world!");
```

GCC generates almost the same code as MSVC.

## 51.3 References

In C++, references are pointers (10 on page 152) as well, but they are called *safe*, because it is harder to make a mistake while dealing with them [ISO13, p. 8.3.2]. For example, reference must always be pointing to an object of the corresponding type and cannot be NULL [Cli, p. 8.6]. Even more than that, references cannot be changed, it is impossible to point them to another object (reseat) [Cli, p. 8.5].

If we are going to try to change the example with pointers (10 on page 152) to use references instead ...

```cpp
void f2 (int x, int y, int & sum, int & product)
{
    sum=x+y;
    product=x*y;
};
```

...then we can see that the compiled code is just the same as in the pointers example (10 on page 152):
Listing 51.20: Optimizing MSVC 2010

```c
_x$ = 8 ; size
_y$ = 12 ; size
_sum$ = 16 ; size
_product$ = 20 ; size

?f2@@YAXHHAAH0@Z PROC ; f2
    mov ecx, DWORD PTR _y$[esp-4]
    mov eax, DWORD PTR _x$[esp-4]
    lea edx, DWORD PTR [eax+ecx]
    imul eax, ecx
    mov ecx, DWORD PTR _product$[esp-4]
    push esi
    mov esi, DWORD PTR _sum$[esp]
    mov DWORD PTR [esi], edx
    mov DWORD PTR [ecx], eax
    pop esi
    ret 0
?f2@@YAXHHAAH0@Z ENDP ; f2
```

(The reason why C++ functions has such strange names is explained here: 51.1.1 on page 757.)

Hence, C++ references are as much efficient as usual pointers.

### 51.4 STL

N.B.: all examples here were checked only in 32-bit environment. x64 wasn’t checked.

#### 51.4.1 std::string

**Internals**

Many string libraries [Yur13, p. 2.2] implement a structure that contains a pointer to a string buffer, a variable that always contains the current string length (which is very convenient for many functions: [Yur13, p. 2.2.1]) and a variable containing the current buffer size. The string in the buffer is usually terminated with zero, in order to be able to pass a pointer to the buffer into the functions that take usual C `ASCIIZ` strings.
It is not specified in the C++ standard [ISO13] how std::string has to be implemented, however, it is usually implemented as explained above.

The C++ string is not a class (as QString in Qt, for instance) but a template (basic_string), this is done in order to support various character types: at least char and wchar_t.

So, std::string is a class with char as its base type. And std::wstring is a class with wchar_t as its base type.

**MSVC**

The MSVC implementation may store the buffer in place instead of using a pointer to a buffer (if the string is shorter than 16 symbols).

This implies that a short string is to occupy at least $16 + 4 + 4 = 24$ bytes in 32-bit environment or at least $16 + 8 + 8 = 32$ bytes in 64-bit one, and if the string is longer than 16 characters, we also have to add the length of the string itself.

Listing 51.21: example for MSVC

```cpp
#include <string>
#include <stdio.h>

struct std_string
{
  union
  {
    char buf[16];
    char* ptr;
  } u;
  size_t size; // AKA 'Mysize' in MSVC
  size_t capacity; // AKA 'Myres' in MSVC
};

void dump_std_string(std::string s)
{
  struct std_string *p=(struct std_string*)&s;
  printf ("[%s] size:%d capacity:%d\n", p->size>16 ? p->u.ptr : p->u.buf, p->size, p->capacity);
}

int main()
{
  std::string s1="short string";
  std::string s2="string longer that 16 bytes";
```
Almost everything is clear from the source code.

A couple of notes:

If the string is shorter than 16 symbols, a buffer for the string is not to be allocated in the heap. This is convenient because in practice, a lot of strings are short indeed. Looks like that Microsoft’s developers chose 16 characters as a good balance.

One very important thing here can be seen at the end of main(): we’re not using the c_str() method, nevertheless, if we compile and run this code, both strings will appear in the console!

This is why it works.

In the first case the string is shorter than 16 characters and the buffer with the string is located in the beginning of the std::string object (it can be treated as a structure). printf() treats the pointer as a pointer to the null-terminated array of characters, hence it works.

Printing the second string (longer than 16 characters) is even more dangerous: it is a typical programmer’s mistake (or typo) to forget to write c_str(). This works because at the moment a pointer to buffer is located at the start of structure. This may stay unnoticed for a long time, until a longer string appears there at some time, then the process will crash.

GCC

GCC’s implementation of this structure has one more variable—reference count.

One interesting fact is that in GCC a pointer an instance of std::string instance points not to the beginning of the structure, but to the buffer pointer. In libstdc++-v3\include\bits\basic_string.h we can read that it was done for more convenient debugging:

```cpp
* The reason you want _M_data pointing to the character %
< array and
* not the _Rep is so that the debugger can see the string
* contents. (Probably we should add a non-inline member to
< get
```
* the _Rep for the debugger to use, so users can check the actual string length.

**basic_string.h source code**

We consider this in our example:

```cpp
#include <string>
#include <stdio.h>

struct std_string
{
  size_t length;
  size_t capacity;
  size_t refcount;
};

void dump_std_string(std::string s)
{
  char *p1=*(char**)&s; // GCC type checking workaround
  struct std_string *p2=(struct std_string*)(p1-sizeof(struct
  std_string));
  printf ("[%s] size:%d capacity:%d\n", p1, p2->length, p2->capacity);
}

int main()
{
  std::string s1="short string";
  std::string s2="string longer that 16 bytes";

  dump_std_string(s1);
  dump_std_string(s2);

  // GCC type checking workaround:
  printf ("%s\n", *(char**)&s1);
  printf ("%s\n", *(char**)&s2);
}
```

A trickery has to be used to imitate the mistake we already have seen above because GCC has stronger type checking, nevertheless, printf() works here without c_str() as well.
A more complex example

```c++
#include <string>
#include <stdio.h>

int main()
{
    std::string s1="Hello, ";
    std::string s2="world!\n";
    std::string s3=s1+s2;
    printf ("%s\n", s3.c_str());
}
```

Listing 51.23: MSVC 2012

```
$SG39512 DB 'Hello, ', 00H
$SG39514 DB 'world!', 0aH, 00H
$SG39581 DB '%s', 0aH, 00H

_s2$ = -72 ; size = 24
_s3$ = -48 ; size = 24
_s1$ = -24 ; size = 24

_main PROC
    sub esp, 72
    push 7
    push OFFSET $SG39512
    lea ecx, DWORD PTR _s1$[esp+80]
    mov DWORD PTR _s1$[esp+100], 15
    mov DWORD PTR _s1$[esp+96], 0
    mov BYTE PTR _s1$[esp+80], 0
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?
    ↦ $allocator@D@2@@std@@QAEAAV12@PBDI@Z ; std::basic_string
    ↦ char,std::char_traits<char>,std::allocator<char> >::
    ↦ assign

    push 7
    push OFFSET $SG39514
    lea ecx, DWORD PTR _s2$[esp+80]
    mov DWORD PTR _s2$[esp+100], 15
    mov DWORD PTR _s2$[esp+96], 0
    mov BYTE PTR _s2$[esp+80], 0
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?
    ↦ $allocator@D@2@@std@@QAEAAV12@PBDI@Z ; std::basic_string
    ↦ char,std::char_traits<char>,std::allocator<char> >::
    ↦ assign
```
lea eax, DWORD PTR _s2$[esp+72]
push eax
lea eax, DWORD PTR _s1$[esp+76]
push eax
lea eax, DWORD PTR _s3$[esp+80]
push eax
call ??$?HDU?$char_traits@D@std@@V?$allocator@D1@std@@YA?⤦AV?$basic_string@DU?$char_traits@D@std@@V?⤦$allocator@D2@0@ABV10@0@Z ; std::operator+<char,std::⤦char_traits<char>,std::allocator<char> >

; inlined c_str() method:
cmp DWORD PTR _s3$[esp+104], 16
lea eax, DWORD PTR _s3$[esp+84]
cmovae eax, DWORD PTR _s3$[esp+84]
push eax
push OFFSET $SG39581
call _printf
add esp, 20

cmp DWORD PTR _s3$[esp+92], 16
jb SHORT $LN119@main
push DWORD PTR _s3$[esp+72]
call ??3@YAXPAX@Z ; operator delete
add esp, 4

$LN119@main:
cmp DWORD PTR _s2$[esp+92], 16
mov DWORD PTR _s3$[esp+92], 15
mov DWORD PTR _s3$[esp+88], 0
mov BYTE PTR _s3$[esp+72], 0
jb SHORT $LN151@main
push DWORD PTR _s2$[esp+72]
call ??3@YAXPAX@Z ; operator delete
add esp, 4

$LN151@main:
cmp DWORD PTR _s1$[esp+92], 16
mov DWORD PTR _s2$[esp+92], 15
mov DWORD PTR _s2$[esp+88], 0
mov BYTE PTR _s2$[esp+72], 0
jb SHORT $LN195@main
push DWORD PTR _s1$[esp+72]
call ??3@YAXPAX@Z ; operator delete
add esp, 4

$LN195@main:
xor eax, eax
The compiler does not construct strings statically: it would not be possible anyway if the buffer needs to be located in the heap. Instead, the ASCIIZ strings are stored in the data segment, and later, at runtime, with the help of the “assign” method, the s1 and s2 strings are constructed. And with the help of operator+, the s3 string is constructed.

Please note that there is no call to the c_str() method, because its code is tiny enough so the compiler inlined it right there: if the string is shorter than 16 characters, a pointer to buffer is left in EAX, otherwise the address of the string buffer located in the heap is fetched.

Next, we see calls to the 3 destructors, they are called if the string is longer than 16 characters: then the buffers in the heap have to be freed. Otherwise, since all three std::string objects are stored in the stack, they are freed automatically, when the function ends.

As a consequence, processing short strings is faster, because of less heap accesses.

GCC code is even simpler (because the GCC way, as we saw above, is to not store shorter strings right in the structure):

```
.LC0:  
    .string "Hello, "
.LC1:  
    .string "world!\n"
main:
push ebp
mov ebp, esp
push edi
push esi
push ebx
and esp, -16
sub esp, 32
lea ebx, [esp+28]
lea edi, [esp+20]
mov DWORD PTR [esp+8], ebx
lea esi, [esp+24]
mov DWORD PTR [esp+4], OFFSET FLAT:.LC0
mov DWORD PTR [esp], edi

call __ZNSsC1EPKcRKSaIcE
mov DWORD PTR [esp+8], ebx
```
It can be seen that it’s not a pointer to the object that is passed to destructors, but rather an address 12 bytes (or 3 words) before, i.e., a pointer to the real start of the
std::string as a global variable

Experienced C++ programmers knows that global variables of STL\(^7\) types can be defined without problems.

Yes, indeed:

```cpp
#include <stdio.h>
#include <string>

std::string s="a string";

int main()
{
    printf ("%s\n", s.c_str());
};
```

But how and where std::string constructor will be called?

In fact, this variable is to be initialized even before `main()` start.

Listing 51.25: MSVC 2012: here is how a global variable is constructed and also its destructor is registered

```assembly
??__Es@@YAXXZ PROC
    push 8
    push OFFSET $SG39512 ; 'a string'
    mov ecx, OFFSET ?s@@3V?$basic_string@DU?
        /rcurvearrowse $char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?
        /rcurvearrowse $allocator@D@2@@std@@QAEAAV12@PBDI@Z ; std::basic_string<
        /rcurvearrowse char,std::char_traits<char>,std::allocator<char> >::assign
    push OFFSET ??__Fs@@YAXXZ ; 'dynamic atexit destructor for 's'
    call _atexit
    pop ecx
    ret 0
??__Es@@YAXXZ ENDP
```

Listing 51.26: MSVC 2012: here a global variable is used in `main()`

```assembly
$SG39512 DB 'a string', 00H
$SG39519 DB '%s', 0aH, 00H
```

\(^7\)(C++) Standard Template Library: 51.4 on page 782
Listing 51.27: MSVC 2012: this destructor function is called before exit

??_Fs@@YAXXXZ PROC
  push ecx
  cmp DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 16
  jmp SHORT $LN23@dynamic
  push esi
  mov esi, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+16, 0
  mov DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A, 0
  pop ecx
  ret 0
$LN23@dynamic:
  ret 0
In fact, a special function with all constructors of global variables is called from CRT, before main(). More than that: with the help of atexit() another function is registered, which contain calls to all destructors of such global variables.

GCC works likewise:

```assembly
Listing 51.28: GCC 4.8.1

main:
    push ebp
    mov ebp, esp
    and esp, -16
    sub esp, 16
    mov eax, DWORD PTR s
    mov DWORD PTR [esp], eax
    call puts
    xor eax, eax
    leave
    ret

.LC0:
    .string "a string"

_GLOBAL__sub_I_s:
    sub esp, 44
    lea eax, [esp+31]
    mov DWORD PTR [esp+8], eax
    mov DWORD PTR [esp+4], OFFSET FLAT:.LCO
    mov DWORD PTR [esp], OFFSET FLAT:_ZNSsC1EPKcRKSaIcE
    call __ZNSsD1Ev
    mov DWORD PTR [esp+4], OFFSET FLAT:__dso_handle
    mov DWORD PTR [esp+8], OFFSET FLAT:_ZNSsD1Ev
    call __cxa_atexit
    add esp, 44
    ret

.LFE645:
    .size _GLOBAL__sub_I_s, .-_GLOBAL__sub_I_s
    .section .init_array,"aw"
    .align 4
    .long _GLOBAL__sub_I_s
    .globl s
    .bss
    .align 4
    .type  s, @object
    .size  s, 4

s:
    .zero 4
```
But it does not create a separate function for this, each destructor is passed to `atexit()`, one by one.

### 51.4.2 std::list

This is the well-known doubly-linked list: each element has two pointers, to the previous and next elements.

This implies that the memory footprint is enlarged by 2 words for each element (8 bytes in 32-bit environment or 16 bytes in 64-bit).

C++ STL just adds the "next" and "previous" pointers to the existing structure of the type that you want to unite in a list.

Let's work out an example with a simple 2-variable structure that we want to store in a list.

Although the C++ standard [ISO13] does not say how to implement it, both MSVC's and GCC's implementations are straightforward and similar, so here is only one source code for both:

```c
#include <stdio.h>
#include <list>
#include <iostream>

struct a
{
    int x;
    int y;
};

struct List_node
{
    struct List_node* _Next;
    struct List_node* _Prev;
    int x;
    int y;
};

void dump_List_node (struct List_node *n)
{
    printf ("ptr=0x%p _Next=0x%p _Prev=0x%p x=%d y=%d\n", 
            n, n->_Next, n->_Prev, n->x, n->y);
}
```
void dump_list_vals (struct List_node* n)
{
    struct List_node* current=n;

    for (;;)
    {
        dump_list_node (current);
        current=current->_Next;
        if (current==n) // end
            break;
    }
}

void dump_list_val (unsigned int *a)
{
    #ifdef _MSC_VER
        // GCC implementation does not have "size" field
        printf ("_Myhead=0x%p, _Mysize=%d\n", a[0], a[1]);
    #endif
    dump_list_vals ((struct List_node*)a[0]);
}

int main()
{
    std::list<struct a> l;

    printf ("* empty list:\n");
    dump_list_val((unsigned int*)(void*)&l);

    struct a t1;
    t1.x=1;
    t1.y=2;
    l.push_front (t1);
    t1.x=3;
    t1.y=4;
    l.push_front (t1);
    t1.x=5;
    t1.y=6;
    l.push_back (t1);

    printf ("* 3-elements list:\n");
    dump_list_val((unsigned int*)(void*)(&l));

    std::list<struct a>::iterator tmp;
    printf ("node at .begin:\n");
    tmp=l.begin();
    dump_list_node ((struct List_node *)(void**)&tmp);
}
GCC

Let's start with GCC.

When we run the example, we'll see a long dump, let's work with it in pieces.

* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0

Here we see an empty list. Despite the fact it is empty, it has one element with garbage (AKA dummy node) in \( x \) and \( y \). Both the “next” and “prev” pointers are pointing to the self node:
At this moment, the .begin and .end iterators are equal to each other.

If we push 3 elements, the list internally will be:

```
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

The last element is still at 0x0028fe90, it not to be moved until the list’s disposal. It still contain random garbage in \( x \) and \( y \) (5 and 6). By coincidence, these values are the same as in the last element, but it doesn’t mean that they are meaningful.

Here is how these 3 elements are stored in memory:
The \( l \) variable always points to the first node.

The .begin() and .end() iterators are not variables, but functions, which when called return pointers to the corresponding nodes.

Having a dummy element (AKA sentinel node) is a very popular practice in implementing doubly-linked lists. Without it, a lot of operations may become slightly more complex and, hence, slower.

The iterator is in fact just a pointer to a node. list.begin() and list.end() just return pointers.

<table>
<thead>
<tr>
<th>node at .begin:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node at .end:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6</td>
</tr>
</tbody>
</table>

The fact that the last element has a pointer to the first and the first element has a pointer to the last one remind us circular list.

This is very helpful here: having a pointer to the first list element, i.e., that is in the \( l \) variable, it is easy to get a pointer to the last one quickly, without the need to traverse the whole list. Inserting an element at the list end is also quick, thanks to this feature.

\( \text{operator-- and operator++ just set the current iterator's value to the current_node->prev or current_node->next values. The reverse iterators (.rbegin, .rend) work just as the same, but in reverse.} \)
operator* just returns a pointer to the point in the node structure, where the user's structure starts, i.e., a pointer to the first element of the structure ($x$).

The list insertion and deletion are trivial: just allocate a new node (or deallocate) and update all pointers to be valid.

That's why an iterator may become invalid after element deletion: it may still point to the node that was already deallocated. This is also called a dangling pointer. And of course, the information from the freed node (to which iterator still points) cannot be used anymore.

The GCC implementation (as of 4.8.1) doesn't store the current size of the list: this implies a slow .size() method: it has to traverse the whole list to count the elements, because it doesn't have any other way to get the information. This mean that this operation is $O(n)$, i.e., it gets slower steadily as the list grows.

Listing 51.29: Optimizing GCC 4.8.1 -fno-inline-small-functions

```assembly
main proc near
  push ebp
  mov ebp, esp
  push esi
  push ebx
  and esp, 0FFFFFFF0h
  sub esp, 20h
  lea ebx, [esp+10h]
  mov dword ptr [esp], offset s ; "* empty list:"
  mov [esp+10h], ebx
  mov [esp+14h], ebx
  call puts
  mov [esp], ebx
  call _Z13dump_List_valPj ; dump_List_val(uint *)
  lea esi, [esp+18h]
  mov [esp+4], esi
  mov [esp], ebx
  mov dword ptr [esp+18h], 1 ; X for new element
  mov dword ptr [esp+1Ch], 2 ; Y for new element
  call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,⤦
  std::allocator<a>>::push_front(a const&)
  mov [esp+4], esi
  mov [esp], ebx
  mov dword ptr [esp+18h], 3 ; X for new element
  mov dword ptr [esp+1Ch], 4 ; Y for new element
  call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,⤦
  std::allocator<a>>::push_front(a const&)
  mov dword ptr [esp], 10h
  mov dword ptr [esp+18h], 5 ; X for new element
  mov dword ptr [esp+1Ch], 6 ; Y for new element
  call _Znwj ; operator new(uint)```

798
cmp    eax, 0FFFFFFFF8h
jz     short loc_80002A6
mov    ecx, [esp+1Ch]
mov    edx, [esp+18h]
mov    [eax+0Ch], ecx
mov    [eax+8], edx

loc_80002A6: ; CODE XREF: main+86
mov    [esp+4], ebx
mov    [esp], eax
call   _ZNSt8__detail15_List_node_base7_M_hookEPS0_ ; std::
跛__detail::_List_node_base::_M_hook(std::__detail::
跛_List_node_base*)
mov    dword ptr [esp], offset a3ElementsList ; "* 3-elements
跛 list:
跛 call puts
mov    [esp], ebx
call   _Z13dump_List_valPj ; dump_List_val(uint *)
mov    dword ptr [esp], offset aNodeAt_begin ; "node at .
跛 begin:
跛 call puts
mov    eax, [esp+10h]
mov    [esp], eax
call   _Z14dump_List_nodeP9List_node ; dump_List_node(
跛 List_node *)
mov    dword ptr [esp], offset aNodeAt_end ; "node at .end:
跛 call puts
mov    [esp], ebx
call   _Z14dump_List_nodeP9List_node ; dump_List_node(
跛 List_node *)
mov    dword ptr [esp], offset aLetSCountFromT ; "* let's 
跛 count from the begin:
跛 call puts
mov    esi, [esp+10h]
mov    eax, [esi+0Ch]
mov    [esp+0Ch], eax
mov    eax, [esi+8]
mov    dword ptr [esp+4], offset a1stElementDD ; "1st element 
跛 : %d %d\n"
mov    dword ptr [esp], 1
mov    [esp+8], eax
call   __printf_chk
mov    esi, [esi]; operator++: get ->next pointer
mov    eax, [esi+0Ch]
mov    [esp+0Ch], eax
mov    eax, [esi+8]
mov    dword ptr [esp+4], offset a2ndElementDD ; "2nd element 
跛
```assembly
\texttt{\textbackslash rcurvearrowse \textbackslash rcurvearrowse} %d %d
mov dword ptr [esp], 1
mov [esp+8], eax
call \_printf\_chk
mov esi, [esi]; operator++: get ->next pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a3rdElementDD ; "3rd element\textbackslash rcurvearrowse %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call \_printf\_chk
mov eax, [esi]; operator++: get ->next pointer
mov edx, [eax+0Ch]
mov [esp+0Ch], edx
mov eax, [eax+8]
mov dword ptr [esp+4], offset aElementAt\_endD ; "element \textbackslash rcurvearrowse at .end(): %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call \_printf\_chk
mov dword ptr [esp], offset aLet\_S\_Count\_Fro\_0 ; "* let's \textbackslash rcurvearrowse count from the end:"
call puts
mov eax, [esp+1Ch]
mov dword ptr [esp+4], offset aElementAt\_endD ; "element \textbackslash rcurvearrowse at .end(): %d %d\n"
mov dword ptr [esp], 1
mov [esp+0Ch], eax
mov eax, [esp+18h]
mov [esp+8], eax
call \_printf\_chk
mov esi, [esp+14h]
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a3rdElementDD ; "3rd element\textbackslash rcurvearrowse %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call \_printf\_chk
mov esi, [esi+4]; operator--: get ->prev pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a2ndElementDD ; "2nd element\textbackslash rcurvearrowse %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call \_printf\_chk
mov esi, [esi+4]; operator--: get ->prev pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a2ndElementDD ; "2nd element\textbackslash rcurvearrowse %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call \_printf\_chk
mov esi, [esi+4]; operator--: get ->prev pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a2ndElementDD ; "2nd element\textbackslash rcurvearrowse %d %d\n"
```
```assembly
; %d %d
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov eax, [esi+4] ; operator--: get ->prev pointer
mov edx, [eax+0Ch]
mov [esp+0Ch], edx
mov eax, [eax+8]
mov dword ptr [esp+4], offset a1stElementDD ; "1st element
; %d %d"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov dword ptr [esp], offset aRemovingLastEl ; "removing
; last element..."
call puts
mov esi, [esp+14h]
mov [esp], esi
call _ZNSt8__detail15_List_node_base9_M_unhookEv ; std::
; __detail::List_node_base::_M_unhook(void)
mov [esp], esi ; void *
call _ZdlPv ; operator delete(void *)
mov [esp], ebx
call _Z13dump_List_valPj ; dump_List_val(uint *)
mov [esp], ebx
call _ZNSt10_List_baseI1aSaIS0_EE8_M_clearEv ; std::
; _List_base<a,std::allocator<a>>::_M_clear(void)
lea esp, [ebp-8]
xor eax, eax
pop ebx
pop esi
pop ebp
retn
main endp

Listing 51.30: The whole output

* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

801
MSVC

MSVC's implementation (2012) is just the same, but it also stores the current size of the list. This implies that the \texttt{.size()} method is very fast ($O(1)$): it just reads one value from memory. On the other hand, the size variable must be updated at each insertion/deletion.

MSVC's implementation is also slightly different in the way it arranges the nodes:

GCC has its dummy element at the end of the list, while MSVC's is at the beginning.
Listing 51.31: Optimizing MSVC 2012 /Fa2.asm /GS- /Ob1

_l$ = -16 ; size = 8
_t1$ = -8 ; size = 8

_main PROC
        sub    esp, 16
        push   ebx
        push   esi
        push   edi
        push   0
        push   0
        lea    ecx, DWORD PTR _l$[esp+36]
        mov    DWORD PTR _l$[esp+40], 0
                ; allocate first "garbage" element
        call   ?_Buynode0@?$List_alloc@$OA@U?$_List_base_types@Ua@V@
                ↘ _$allocator@Ua@@std@@std@@std@@QAEPau@
                ↘ $_List_node@Ua@@PAX@2@PAU32@0@Z ; std::List_alloc<0,std
                ↘ ::_List_base_types<a,std::allocator<a>> > >::_Buynode0
        mov    edi, DWORD PTR __imp__printf
        mov    ebx, eax
        push   OFFSET $SG40685 ; '* empty list:'
        mov    DWORD PTR _l$[esp+32], ebx
        call   edi  ; printf
        lea    eax, DWORD PTR _l$[esp+32]
        push   eax
        call   ?dump_List_val@YAXPAI@Z ; dump_List_val
        mov    esi, DWORD PTR [ebx]
        add    esp, 8
        lea    eax, DWORD PTR _t1$[esp+28]
        push   eax
        push   DWORD PTR [esi+4]
        lea    ecx, DWORD PTR _l$[esp+36]
        push   esi
        mov    DWORD PTR _t1$[esp+40], 1 ; data for a new node
        mov    DWORD PTR _t1$[esp+44], 2 ; data for a new node
                ; allocate new node
        call   ??$_Buynode@ABUa@@@?$List_buy@Ua@V@
                ↘ _$allocator@Ua@@std@@std@@std@@QAEPau@
                ↘ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@Z ; std::List_buy<a,
                ↘ std::allocator<a>> >::_Buynode<a const &>
        mov    DWORD PTR [esi+4], eax
        mov    ecx, DWORD PTR [eax+4]
        mov    DWORD PTR _t1$[esp+28], 3 ; data for a new node
        mov    DWORD PTR [ecx], eax
        mov    esi, DWORD PTR [ebx]
        lea    eax, DWORD PTR _t1$[esp+28]
        push   eax
        push   DWORD PTR [esi+4]
lea ecx, DWORD PTR _l$[esp+36]
push esi
mov DWORD PTR _t1$[esp+44], 4 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V? ↵
\$allocator@Ua@@@std@@@std@@QAEPAU?
\$:_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a, ↵
\$std::allocator<a> >::_Buynode<a const &>
mov DWORD PTR [esi+4], eax
mov ecx, DWORD PTR [eax+4]
mov DWORD PTR _t1$[esp+28], 5 ; data for a new node
mov DWORD PTR [ecx], eax
lea eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [ebx+4]
lea ecx, DWORD PTR _l$[esp+36]
push ebx
mov DWORD PTR _t1$[esp+44], 6 ; data for a new node
; allocate new node
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V? ↵
\$allocator@Ua@@@std@@@std@@QAEPAU?
\$:_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a, ↵
\$std::allocator<a> >::_Buynode<a const &>
mov DWORD PTR [ebx+4], eax
mov ecx, DWORD PTR [eax+4]
push OFFSET $SG40689 ; '* 3-elements list:'
mov DWORD PTR _l$[esp+36], 3
mov DWORD PTR [ecx], eax
call edi ; printf
lea eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@YAXPAI@Z ; dump_List_val
push OFFSET $SG40831 ; 'node at .begin:'
call edi ; printf
push DWORD PTR [ebx]; get next field of node / variable points to
call ?dump_List_node@YAXPAUList_node@@@@Z ; dump_List_node
push OFFSET $SG40835; 'node at .end:'
call edi ; printf
push ebx; pointer to the node 1$ variable points to!
call ?dump_List_node@YAXPAUList_node@@@@Z ; dump_List_node
push OFFSET $SG40839; '* let's count from the begin:'
call edi ; printf
mov esi, DWORD PTR [ebx]; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40846; '1st element: %d %d'
call edi ; printf
mov esi, DWORD PTR [esi]; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40848 ; '2nd element: %d %d'
call edi; printf
mov esi, DWORD PTR [esi]; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40850 ; '3rd element: %d %d'
call edi; printf
mov eax, DWORD PTR [esi]; operator++: get ->next pointer
add esp, 64
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
push OFFSET $SG40852 ; 'element at .end(): %d %d'
call edi; printf
push OFFSET $SG40853 ; '* let''s count from the end:'
call edi; printf
push DWORD PTR [ebx+12]; use x and y fields from the node l
variable points to
push DWORD PTR [ebx+8]
push OFFSET $SG40860 ; 'element at .end(): %d %d'
call edi; printf
mov esi, DWORD PTR [ebx+4]; operator--: get ->prev ↩
pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40862 ; '3rd element: %d %d'
call edi; printf
mov esi, DWORD PTR [esi+4]; operator--: get ->prev ↩
pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40864 ; '2nd element: %d %d'
call edi; printf
mov eax, DWORD PTR [esi+4]; operator--: get ->prev ↩
pointer
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
push OFFSET $SG40866 ; '1st element: %d %d'
call edi; printf
add esp, 64
push OFFSET $SG40867 ; 'removing last element...'
call edi; printf
mov edx, DWORD PTR [ebx+4]
add esp, 4

; prev=next?
Unlike GCC, MSVC’s code allocates the dummy element at the start of the function with the help of the “Buynode” function, it is also used to allocate the rest of the nodes (GCC’s code allocates the first element in the local stack).
Listing 51.32: The whole output

```cpp
* empty list:
_Myhead=0x003CC258, _Mysize=0
ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y=4522072

* 3-elements list:
_Myhead=0x003CC258, _Mysize=3
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC2A0 _Prev=0x003CC288 x=1 y=2
ptr=0x003CC2A0 _Next=0x003CC258 _Prev=0x003CC270 x=5 y=6
node at .begin:
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
node at .end:
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072

* let's count from the begin:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 6226002 4522072

* let's count from the end:
element at .end(): 6226002 4522072
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
_Myhead=0x003CC258, _Mysize=2
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC270 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC258 _Prev=0x003CC288 x=1 y=2
```

C++11 std::forward_list

The same thing as std::list, but singly-linked one, i.e., having only the “next” field at each node. It has a smaller memory footprint, but also don’t offer the ability to traverse list backwards.
51.4.3 `std::vector`

We would call `std::vector` a safe wrapper of the PODT\(^8\) C array. Internally it is somewhat similar to `std::string` (51.4.1 on page 782): it has a pointer to the allocated buffer, a pointer to the end of the array, and a pointer to the end of the allocated buffer.

The array's elements lie in memory adjacently to each other, just like in a normal array (18 on page 359). In C++11 there is a new method called `.data()`, that returns a pointer to the buffer, like `.c_str()` in `std::string`.

The buffer allocated in the heap can be larger than the array itself.

Both MSVC's and GCC's implementations are similar, just the names of the structure's fields are slightly different\(^9\), so here is one source code that works for both compilers. Here is again the C-like code for dumping the structure of `std::vector`:

```c
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
    // MSVC names:
    int *Myfirst;
    int *Mylast;
    int *Myend;

    // GCC structure is the same, but names are: _M_start, _M_finish, _M_end_of_storage
};

void dump(struct vector_of_ints *in)
{
    printf (“_Myfirst=%p, _Mylast=%p, _Myend=%p\n", in->Myfirst
⤦, in->Mylast, in->Myend);
    size_t size=(in->Mylast-in->Myfirst);
    size_t capacity=(in->Myend-in->Myfirst);
    printf (“size=%d, capacity=%d\n", size, capacity);
    for (size_t i=0; i<size; i++)
        printf (“element %d: %d\n”, i, in->Myfirst[i]);
};

int main()
{
```

\(^8\)(C++) Plain Old Data Type
\(^9\)GCC internals: [http://go.yurichev.com/17086](http://go.yurichev.com/17086)
CHAPTER 51. C++

CHAPTER 51. C++

std::vector<int> c;
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(1);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(2);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(3);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(4);
dump ((struct vector_of_ints*)(void*)&c);
c.reserve (6);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(5);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(6);
dump ((struct vector_of_ints*)(void*)&c);
printf ("%d\n", c.at(5)); // with bounds checking
printf ("%d\n", c[8]); // operator[], without bounds
checking
};

Here is the output of this program when compiled in MSVC:
_Myfirst=00000000,
size=0, capacity=0
_Myfirst=0051CF48,
size=1, capacity=1
element 0: 1
_Myfirst=0051CF58,
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0051C278,
size=3, capacity=3
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0051C290,
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180,
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3

_Mylast=00000000, _Myend=00000000
_Mylast=0051CF4C, _Myend=0051CF4C

_Mylast=0051CF60, _Myend=0051CF60

_Mylast=0051C284, _Myend=0051C284

_Mylast=0051C2A0, _Myend=0051C2A0

_Mylast=0051B190, _Myend=0051B198

809


As it can be seen, there is no allocated buffer when `main()` starts. After the first `push_back()` call, a buffer is allocated. And then, after each `push_back()` call, both array size and buffer size (capacity) are increased. But the buffer address changes as well, because `push_back()` reallocates the buffer in the heap each time. It is costly operation, that's why it is very important to predict the size of the array in the future and reserve enough space for it with the `.reserve()` method. The last number is garbage: there are no array elements at this point, so a random number is printed. This illustrates the fact that operator[] of `std::vector` does not check of the index is in the array's bounds. The slower `.at()` method, however, does this checking and throws an `std::out_of_range` exception in case of error.

Let's see the code:

Listing 51.33: MSVC 2012 /GS- /Ob1

```
$SG52650 DB '%d', 0aH, 00H
$SG52651 DB '%d', 0aH, 00H

_this$ = -4 ; size = 4
_Pos$ = 8 ; size = 4

?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z PROC ; std::vector<int,std::allocator<int> >::at, COMDAT
; _this$ = ecx
  push ebp
  mov ebp, esp
  push ecx
  mov DWORD PTR _this$[ebp], ecx
  mov eax, DWORD PTR _this$[ebp]
```
CHAPTER 51. C++

```
mov ecx, DWORD PTR _this$[ebp]
mov edx, DWORD PTR [eax+4]
sub edx, DWORD PTR [ecx]
sar edx, 2
cmp edx, DWORD PTR __Pos$[ebp]
ja SHORT $LN1@at
push OFFSET ??_C@_0BM@NMJKDPO@invalid?5vector?$DMT?$DO?5 subscriipt?$AA@
call DWORD PTR __imp_?_Xout_of_range@std@@YAXPBD@Z

$LN1@at:
mov eax, DWORD PTR _this$[ebp]
mov ecx, DWORD PTR [eax]
mov edx, DWORD PTR __Pos$[ebp]
lea eax, DWORD PTR [ecx+edx*4]

$LN3@at:
mov esp, ebp
pop ebp
ret 4

atat@$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ENDP ; std::vector<int,std::allocator<int> >::at

_c$ = -36 ; size = 12
$T1 = -24 ; size = 4
$T2 = -20 ; size = 4
$T3 = -16 ; size = 4
$T4 = -12 ; size = 4
$T5 = -8 ; size = 4
$T6 = -4 ; size = 4

_main PROC
push ebp
mov ebp, esp
sub esp, 36
mov DWORD PTR _c$[ebp], 0 ; Myfirst
mov DWORD PTR _c$[ebp+4], 0 ; Mylast
mov DWORD PTR _c$[ebp+8], 0 ; Myend
lea eax, DWORD PTR _c$[ebp]
push eax
call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
add esp, 4
mov DWORD PTR $T6[ebp], 1
lea ecx, DWORD PTR $T6[ebp]
push ecx
lea ecx, DWORD PTR _c$[ebp]
call ?push_back@$vector@HV@$allocator@H@std@@@std@@QAEAAHI@Z ; std::vector<int,std::allocator<int> >::push_back
lea edx, DWORD PTR _c$[ebp]
```

811
push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add esp, 4
    mov DWORD PTR $T5[ebp], 2
    lea eax, DWORD PTR $T5[ebp]
    push eax
    lea ecx, DWORD PTR _c$[ebp]
    call ?push_back@$vector@HV? ↵
      $allocator@H@std@@std@@QAEX$$QAH@Z ; std::vector<int,std↵
      ::allocator<int> >::push_back
    lea ecx, DWORD PTR _c$[ebp]
    push ecx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add esp, 4
    mov DWORD PTR $T4[ebp], 3
    lea edx, DWORD PTR $T4[ebp]
    push edx
    lea ecx, DWORD PTR _c$[ebp]
    call ?push_back@$vector@HV? ↵
      $allocator@H@std@@std@@QAEX$$QAH@Z ; std::vector<int,std↵
      ::allocator<int> >::push_back
    lea eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add esp, 4
    mov DWORD PTR $T3[ebp], 4
    lea ecx, DWORD PTR $T3[ebp]
    push ecx
    lea ecx, DWORD PTR _c$[ebp]
    call ?push_back@$vector@HV? ↵
      $allocator@H@std@@std@@QAEX$$QAH@Z ; std::vector<int,std↵
      ::allocator<int> >::push_back
    lea edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add esp, 4
    push 6
    lea ecx, DWORD PTR _c$[ebp]
    call ?reserve@$vector@HV?$allocator@H@std@@std@@QAEXI@Z ; ↵
      std::vector<int,std::allocator<int> >::reserve
    lea eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add esp, 4
    mov DWORD PTR $T2[ebp], 5
    lea ecx, DWORD PTR $T2[ebp]
    push ecx
We see how the `.at()` method checks the bounds and throws an exception in case of error. The number that the last `printf()` call prints is just taken from the
memory, without any checks.

One may ask, why not use the variables like “size” and “capacity”, like it was done in \texttt{std::string}. Supposedly, this was done for faster bounds checking.

The code GCC generates is in general almost the same, but the \texttt{.at()} method is inlined:

\begin{verbatim}
Listing 51.34: GCC 4.8.1 -fno-inline-small-functions -O1

main proc near
    push ebp
    mov ebp, esp
    push edi
    push esi
    push ebx
    and esp, 0xFFFFFF0h
    sub esp, 20h
    mov dword ptr [esp+14h], 0
    mov dword ptr [esp+18h], 0
    mov dword ptr [esp+1Ch], 0
    lea eax, [esp+14h]
    mov [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov dword ptr [esp+10h], 1
    lea eax, [esp+10h]
    mov [esp+4], eax
    lea eax, [esp+14h]
    mov [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, \shortdownarrow std::allocator<int>>::push_back(int const&)
    lea eax, [esp+14h]
    mov [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov dword ptr [esp+10h], 2
    lea eax, [esp+10h]
    mov [esp+4], eax
    lea eax, [esp+14h]
    mov [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, \shortdownarrow std::allocator<int>>::push_back(int const&)
    lea eax, [esp+14h]
    mov [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov dword ptr [esp+10h], 3
    lea eax, [esp+10h]
    mov [esp+4], eax
    lea eax, [esp+14h]
    mov [esp], eax
\end{verbatim}
call __ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, std::allocator<int>>::push_back(int const&)
le a eax, [esp+14h]
mov [esp], eax
call __Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 4
le a eax, [esp+10h]
mov [esp+4], eax
le a eax, [esp+14h]
mov [esp], eax
call __ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int, std::allocator<int>>::push_back(int const&)
le a eax, [esp+14h]
mov [esp], eax
call __Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov ebx, [esp+1Ch]
sub eax, ebx
cmp eax, 17h
ja short loc_80001CF
mov edi, [esp+18h]
sar edi, 2
mov dword ptr [esp+10h], 4
le a eax, ds:0[edi*4]
mov [esp+8], eax ; n
mov [esp+4], ebx ; src
mov [esp], esi ; dest
call memmove
loc_80001AD: ; CODE XREF: main+F8
mov eax, [esp+14h]
test eax, eax
jz short loc_80001BD
mov [esp], eax ; void *
call __ZdlPv ; operator delete(void *)
loc_80001BD: ; CODE XREF: main+117
mov [esp+14h], esi
le a eax, [esi+edi*4]
mov [esp+18h], eax
add esi, 18h
mov [esp+1Ch], esi
loc_80001CF: ; CODE XREF: main+DD
    lea eax, [esp+14h]
    mov [esp], eax
    call _Z4dumpP14vector_of_ints; dump(vector_of_ints *)
    mov dword ptr [esp+10h], 5
    lea eax, [esp+10h]
    mov [esp+4], eax
    lea eax, [esp+14h]
    mov [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi; std::vector<int, std::allocator<int>>::push_back(int const&)
    lea eax, [esp+14h]
    mov [esp], eax
    call _Z4dumpP14vector_of_ints; dump(vector_of_ints *)
    mov dword ptr [esp+10h], 6
    lea eax, [esp+10h]
    mov [esp+4], eax
    lea eax, [esp+14h]
    mov [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi; std::vector<int, std::allocator<int>>::push_back(int const&)
    lea eax, [esp+14h]
    mov [esp], eax
    call _Z4dumpP14vector_of_ints; dump(vector_of_ints *)
    mov eax, [esp+14h]
    mov edx, [esp+18h]
    sub edx, eax
    cmp edx, 17h
    ja short loc_8000246
    mov dword ptr [esp], offset aVector_m_range; "vector:::_M_range_check"
    call _ZSt20__throw_out_of_rangePKc; std::__throw_out_of_range(char const*)
loc_8000246: ; CODE XREF: main+19C
    mov eax, [eax+14h]
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aD; "%d\n"
    mov dword ptr [esp], 1
    call __printf_chk
    mov eax, [esp+14h]
    mov eax, [eax+20h]
    mov [esp+8], eax
    mov dword ptr [esp+4], offset aD; "%d\n"
    mov dword ptr [esp], 1
    call __printf_chk
.reserve() is inlined as well. It calls \texttt{new()} if the buffer is too small for the new size, calls \texttt{memmove()} to copy the contents of the buffer, and calls \texttt{delete()} to free the old buffer.

Let's also see what the compiled program outputs if compiled with GCC:

\begin{verbatim}
_Myfirst=0x(nil), _Mylast=0x(nil), _Myend=0x(nil)
size=0, capacity=0
_Myfirst=0x8257008, _Mylast=0x825700c, _Myend=0x825700c
size=1, capacity=1
element 0: 1
_Myfirst=0x8257018, _Mylast=0x8257020, _Myend=0x8257020
size=2, capacity=2
element 0: 1
element 1: 2
\end{verbatim}
We can spot that the buffer size grows in a different way that in MSVC.
Simple experimentation shows that in MSVC's implementation the buffer grows by ~50% each time it needs to be enlarged, while GCC's code enlarges it by 100% each time, i.e., doubles it.

51.4.4 std::map and std::set

The binary tree is another fundamental data structure. As its name states, this is a tree where each node has at most 2 links to other nodes. Each node has key and/or value.
Binary trees are usually the structure used in the implementation of “dictionaries” of key-values (AKA “associative arrays”).

There are at least three important properties that a binary trees has:

- All keys are always stored in sorted form.
- Keys of any types can be stored easily. Binary tree algorithms are unaware of the key’s type, only a key comparison function is required.
- Finding a specific key is relatively fast in comparison with lists and arrays.

Here is a very simple example: let’s store these numbers in a binary tree: 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.

All keys that are smaller than the node key’s value are stored on the left side. All keys that are bigger than the node key’s value are stored on the right side.

Hence, the lookup algorithm is straightforward: if the value that you are looking for is smaller than the current node’s key value: move left, if it is bigger: move right, stop if the value required is equal to the node key’s value. That is why the searching algorithm may search for numbers, text strings, etc, as long as a key comparison function is provided.

All keys have unique values.

Having that, one needs $\approx \log_2 n$ steps in order to find a key in a balanced binary tree with $n$ keys. This implies that $\approx 10$ steps are needed $\approx 1000$ keys, or $\approx 13$ steps for $\approx 10000$ keys. Not bad, but the tree has always to be balanced for this: i.e., the keys has to be distributed evenly on all levels. The insertion and removal operations do some maintenance to keep the tree in a balanced state.
There are several popular balancing algorithms available, including the AVL tree and the red-black tree. The latter extends each node with a “color” value to simplify the balancing process, hence, each node may be “red” or “black”.

Both GCC’s and MSVC’s `std::map` and `std::set` template implementations use red-black trees.

`std::set` contains only keys. `std::map` is the “extended” version of `std::set`: it also has a value at each node.

**MSVC**

```cpp
#include <map>
#include <set>
#include <string>
#include <iostream>

// structure is not packed!
struct tree_node
{
    struct tree_node *Left;
    struct tree_node *Parent;
    struct tree_node *Right;
    char Color; // 0 - Red, 1 - Black
    char Isnil;
    //std::pair Myval;
    unsigned int first; // called Myval in std::set
    const char *second; // not present in std::set
};

struct tree_struct
{
    struct tree_node *Myhead;
    size_t Mysize;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool /*traverse*/)
{
    printf ("ptr=0x%p Left=0x%p Parent=0x%p Right=0x%p Color=%d %d\n",
            n, n->Left, n->Parent, n->Right, n->Color, n->Isnil);
    if (n->Isnil==0)
    {
        if (is_set)
            printf ("first=%d\n", n->first);
```
```c
else
    printf ("first=%d second=[%s]\n", n->first, n->second);
}

if (traverse)
{
    if (n->Isnil==1)
        dump_tree_node (n->Parent, is_set, true);
    else
    {
        if (n->Left->Isnil==0)
            dump_tree_node (n->Left, is_set, true);
        if (n->Right->Isnil==0)
            dump_tree_node (n->Right, is_set, true);
    }
}

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    if (is_set)
        printf ("%d\n", n->first);
    else
        printf ("%d [%s]\n", n->first, n->second);
    if (n->Left->Isnil==0)
    {
        printf ("%.*sL------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Left, is_set);
    }
    if (n->Right->Isnil==0)
    {
        printf ("%.*sR------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Right, is_set);
    }
}

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, Myhead=0x%p, Mysize=%d\n", m, m->Myhead, m->Mysize);
    dump_tree_node (m->Myhead, is_set, true);
    printf ("As a tree:\n");
    dump_as_tree (1, m->Myhead->Parent, is_set);
```
int main()
{
    // map

    std::map<int, const char*> m;
    m[10] = "ten";
    m[20] = "twenty";
    m[3] = "three";
    m[101] = "one hundred one";
    m[100] = "one hundred";
    m[12] = "twelve";
    m[107] = "one hundred seven";
    m[0] = "zero";
    m[1] = "one";
    m[6] = "six";
    m[99] = "ninety-nine";
    m[5] = "five";
    m[11] = "eleven";
    m[1001] = "one thousand one";
    m[1010] = "one thousand ten";
    m[2] = "two";
    m[9] = "nine";
    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);

    std::map<int, const char*>::iterator it1 = m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *)(void**)&it1, false,
                    false);
    it1 = m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *)(void**)&it1, false,
                    false);

    // set

    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
    s.insert(1001);
    printf ("dumping s as set:\n");
```c
dump_map_and_set ((struct tree_struct *)(void*)&s, true);
std::set<int>::iterator it2=s.begin();
printf("s.begin():\n");
dump_tree_node ((struct tree_node *)*(void**)&it2, true,
ï ➤ false);

it2=s.end();
printf("s.end():\n");
dump_tree_node ((struct tree_node *)*(void**)&it2, true,
ï ➤ false);
};
```

**Listing 51.35: MSVC 2012**

dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0
  ➤ x005BB580 Color=1 Isnil=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0
  ➤ x005BB440 Color=1 Isnil=0
first=10 second=[ten]
ptr=0x005BB4C0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0
  ➤ x005BB520 Color=1 Isnil=0
first=1 second=[one]
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0
  ➤ x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
ptr=0x005BB520 Left=0x005BB400 Parent=0x005BB4C0 Right=0
  ➤ x005BB4E0 Color=0 Isnil=0
first=5 second=[five]
ptr=0x005BB400 Left=0x005BB5A0 Parent=0x005BB520 Right=0
  ➤ x005BB3A0 Color=1 Isnil=0
first=3 second=[three]
ptr=0x005BB5A0 Left=0x005BB3A0 Parent=0x005BB400 Right=0
  ➤ x005BB3A0 Color=0 Isnil=0
first=2 second=[two]
ptr=0x005BB4E0 Left=0x005BB3A0 Parent=0x005BB520 Right=0
  ➤ x005BB5C0 Color=1 Isnil=0
first=6 second=[six]
ptr=0x005BB5C0 Left=0x005BB3A0 Parent=0x005BB4E0 Right=0
  ➤ x005BB3A0 Color=0 Isnil=0
first=9 second=[nine]
ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB3C0 Right=0
  ➤ x005BB480 Color=1 Isnil=0
first=100 second=[one hundred]
ptr=0x005BB3E0 Left=0x005BB460 Parent=0x005BB440 Right=0
  ➤ x005BB500 Color=0 Isnil=0
first=20 second=[twenty]
As a tree:

```
root----10 [ten]
   L-------1 [one]
   L-------0 [zero]
   R-------5 [five]
      L-------3 [three]
      L-------2 [two]
      R-------6 [six]
      R-------9 [nine]
   R-------100 [one hundred]
      L-------20 [twenty]
      L-------12 [twelve]
      L-------11 [eleven]
      R-------99 [ninety-nine]
      R-------107 [one hundred seven]
      L-------101 [one hundred one]
      R-------1001 [one thousand one]
      R-------1010 [one thousand ten]
```

m.begin():

```
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0
   ↓ x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
```

m.end():

```
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0
   ↓ x005BB580 Color=1 Isnil=1
```
As a tree:
root----123
 L-------12
  L-------11
   R-------100
    R-------456
     R-------1001
s.begin():
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0
     x005BB5E0 Color=0 Isnil=0
first=11
s.end():
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0
     x005BB620 Color=1 Isnil=0

The structure is not packed, so both char values occupy 4 bytes each.

As for std::map, first and second can be viewed as a single value of type
std::pair. std::set has only one value at this address in the structure in-
stead.

The current size of the tree is always present, as in the case of the implementation
of std::list in MSVC (51.4.2 on page 802).

As in the case of std::list, the iterators are just pointers to nodes. The .begin() iterator points to the minimal key. That pointer is not stored anywhere (as in
lists), the minimal key of the tree is looked up every time. `operator--` and `operator++` move the current node pointer to the predecessor or successor respectively, i.e., the nodes which have the previous or next key. The algorithms for all these operations are explained in [Cor+09].

The `.end()` iterator points to the dummy node, it has 1 in `Isnil`, which implies that the node has no key and/or value. It can be viewed as a “landing zone” in HDD\(^{10}\). The “parent” field of the dummy node points to the root node, which serves as a vertex of the tree and contains information.

**GCC**

```c
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool /
\textarrowse
traverse, bool dump_keys_and_values)
{
    printf ("ptr=0x%p M_left=0x%p M_parent=0x%p M_right=0x%p \textarrowse M_color=%d\n",
\textarrowse
```


```c
n, n->M_left, n->M_parent, n->M_right, n->M_color);

void *point_after_struct=((char*)n)+sizeof(struct tree_node);

if (dump_keys_and_values)
{
    if (is_set)
        printf ("key=%d\n", *(int*)point_after_struct);
    else
    {
        struct map_pair *p=(struct map_pair *)
        \ point_after_struct;
        printf ("key=%d value=[%s]\n", p->key, p->value);
    }
}

if (traverse==false)
    return;

if (n->M_left)
    dump_tree_node (n->M_left, is_set, traverse, \\
\ dump_keys_and_values);
if (n->M_right)
    dump_tree_node (n->M_right, is_set, traverse, \\
\ dump_keys_and_values);
}

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (is_set)
        printf ("%d\n", *(int*)point_after_struct);
    else
    {
        struct map_pair *p=(struct map_pair *)
        \ point_after_struct;
        printf ("%d [%s]\n", p->key, p->value);
    }

    if (n->M_left)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
```
```c++
void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, M_key_compare=0x%x, M_header=0x%p, M_node_count=%d\n",
            m, m->M_key_compare, &m->M_header, m->M_node_count);
    dump_tree_node (m->M_header.M_parent, is_set, true, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent, is_set);
}

int main()
{
    // map
    std::map<int, const char*> m;
    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";

    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);
}
```
std::map<int, const char*>::iterator it1=m.begin();
printf("m.begin():\n");
dump_tree_node((struct tree_node *)(void**)&it1, false, false, true);
it1=m.end();
printf("m.end():\n");
dump_tree_node((struct tree_node *)(void**)&it1, false, false, false);

// set

std::set<int> s;
s.insert(123);
s.insert(456);
s.insert(11);
s.insert(12);
s.insert(100);
s.insert(1001);
printf("dumping s as set:\n");
dump_map_and_set((struct tree_struct *)(void*)&s, true);
std::set<int>::iterator it2=s.begin();
printf("s.begin():\n");
dump_tree_node((struct tree_node *)(void**)&it2, true, false, true);
it2=s.end();
printf("s.end():\n");
dump_tree_node((struct tree_node *)(void**)&it2, true, false, false);

Listing 51.36: GCC 4.8.1

dumping m as map:
ptr=0x0028FE3C, M_key_compare=0x402b70, M_header=0x0028FE40, M_node_count=17

ptr=0x007A4988 M_left=0x007A4C00 M_parent=0x0028FE40 M_right=0
  x007A4B80 M_color=1
key=10 value=[ten]
ptr=0x007A4C00 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0
  x007A4C60 M_color=1
key=1 value=[one]
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0
  x007A4C20 M_color=0
key=0 value=[zero]
ptr=0x007A4C60 M_left=0x007A4B40 M_parent=0x007A4C00 M_right=0
  x007A4B80 M_color=1
key=5 value=[five]
As a tree:

```
root----10 [ten]
    L-------1 [one]
      L-------0 [zero]
      R-------5 [five]
            L-------3 [three]
            L-------2 [two]
            R-------6 [six]
```
R-------9 [nine]  
R-------100 [one hundred]
L-------20 [twenty]  
L-------12 [twelve]  
L-------11 [eleven]  
R-------99 [ninety-nine]  
R-------107 [one hundred seven]  
L-------101 [one hundred one]  
R-------1001 [one thousand one]  
R-------1010 [one thousand ten]

m.begin():
ptr=0x007A4BE0  M_left=0x00000000  M_parent=0x007A4C00  M_right=0
   \  \  M_color=1
key=0  value=[zero]

m.end():
ptr=0x0028FE40  M_left=0x007A4BE0  M_parent=0x007A4988  M_right=0
   \  \  M_color=0

dumping s as set:
ptr=0x0028FE20,  M_key_compare=0x8,  M_header=0x0028FE24,  
   \  \  M_node_count=6
ptr=0x007A1E80  M_left=0x01D5D890  M_parent=0x0028FE24  M_right=0
   \  \  M_color=1
key=123
ptr=0x01D5D890  M_left=0x01D5D870  M_parent=0x007A1E80  M_right=0
   \  \  M_color=1
key=12
ptr=0x01D5D870  M_left=0x00000000  M_parent=0x01D5D890  M_right=0
   \  \  M_color=0
key=11
ptr=0x01D5D870  M_left=0x00000000  M_parent=0x01D5D890  M_right=0
   \  \  M_color=0
key=100
ptr=0x01D5D850  M_left=0x00000000  M_parent=0x007A1E80  M_right=0
   \  \  M_color=1
key=456
ptr=0x01D5D8D0  M_left=0x00000000  M_parent=0x01D5D850  M_right=0
   \  \  M_color=0
key=1001

As a tree:
root----123
   L-------12
   L-------11
   R-------100
R-------456
R-------1001

s.begin():
GCC’s implementation is very similar. The only difference is the absence of the `Isnil` field, so the structure occupies slightly less space in memory than its implementation in MSVC. The dummy node is also used as a place to point the `.end()` iterator also has no key and/or value.

**Rebalancing demo (GCC)**

Here is also a demo showing us how a tree is rebalanced after some insertions.

Listing 51.37: GCC

```c++
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair {
    int key;
    const char *value;
};

struct tree_node {
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct {
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};
```

11 [http://go.yurichev.com/17084](http://go.yurichev.com/17084)
const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);
    printf ("%d
", *(int*)point_after_struct);
    if (n->M_left)
    {
        printf("%.sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left);
    };
    if (n->M_right)
    {
        printf("%.sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right);
    };
}

void dump_map_and_set(struct tree_struct *m)
{
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent);
};

int main()
{
    std::set<int> s;
    s.insert(123);
    s.insert(456);
    printf ("123, 456 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(11);
    s.insert(12);
    printf ("\n");
    printf ("11, 12 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(100);
    s.insert(1001);
    printf ("\n");
    printf ("100, 1001 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(667);
    s.insert(1);
    s.insert(4);
s.insert(7);
printf ("\n");
printf ("667, 1, 4, 7 are inserted\n");
dump_map_and_set ((struct tree_struct *)(void*)&s);
printf ("\n");
};

Listing 51.38: GCC 4.8.1

123, 456 are inserted
root----123
   R-------456

11, 12 are inserted
root----123
   L-------11
      R-------12
   R-------456

100, 1001 are inserted
root----123
   L-------12
      L-------11
         R-------100
      R-------456
      R-------1001

667, 1, 4, 7 are inserted
root----12
   L-------4
      L-------1
         R-------11
            L-------7
      R-------123
         L-------100
            R-------667
               L-------456
                  R-------1001
Chapter 52

Negative array indices

It's possible to address the space before an array by supplying a negative index, e.g., array[-1].

It's very hard to say why one should use it, there is probably only one known practical application of this technique. C/C++ array elements indices start at 0, but some PLs have a first index at 1 (at least FORTRAN). Programmers may still have this habit, so using this little trick, it's possible to address the first element in C/C++ using index 1:

```
#include <stdio.h>

int main()
{
    int random_value=0x11223344;
    unsigned char array[10];
    int i;
    unsigned char *fakearray=&array[-1];

    for (i=0; i<10; i++)
        array[i]=i;

    printf("first element \%d\n", fakearray[1]);
    printf("second element \%d\n", fakearray[2]);
    printf("last element \%d\n", fakearray[10]);

    printf("array[-1]=%02X, array[-2]=%02X, array[-3]=%02X
\", array[-1], array[-2], array[-3], array[-4]=\%02X\n", array[-4], array[-3], array[-2], array[-1], array[-2], array[-3], array[-4], array[-3], array[-2], array[-1]);
}
```
array[-4]);

};

Listing 52.1: Non-optimizing MSVC 2010

```assembly
$SG2751 DB 'first element %d', 0aH, 00H
$SG2752 DB 'second element %d', 0aH, 00H
$SG2753 DB 'last element %d', 0aH, 00H
$SG2754 DB 'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X', 0aH, 00H

_fakearray$ = -24 ; size = 4
_random_value$ = -20 ; size = 4
_array$ = -16 ; size = 10
_i$ = -4 ; size = 4

_main PROC
    push ebp
    mov ebp, esp
    sub esp, 24
    mov DWORD PTR _random_value$[ebp], 287454020 ; 11223344H
    ; set fakearray[] one byte earlier before array[]
    lea eax, DWORD PTR _array$[ebp]
    add eax, -1 ; eax=eax-1
    mov DWORD PTR _fakearray$[ebp], eax
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN3@main
    ; fill array[] with 0..9

$LN2@main:
    mov ecx, DWORD PTR _i$[ebp]
    add ecx, 1
    mov DWORD PTR _i$[ebp], ecx

$LN3@main:
    cmp DWORD PTR _i$[ebp], 10
    jge SHORT $LN1@main
    mov edx, DWORD PTR _i$[ebp]
    mov al, BYTE PTR _i$[ebp]
    mov BYTE PTR _array$[ebp+edx], al
    jmp SHORT $LN2@main

$LN1@main:
    mov ecx, DWORD PTR _fakearray$[ebp]
    ; ecx=address of fakearray[0], ecx+1 is fakearray[1] or array[0]
    movzx edx, BYTE PTR [ecx+1]
    mov edx, BYTE PTR [ecx+1]
    push edx
    push OFFSET $SG2751 ; 'first element %d'
call _printf
```

836
add esp, 8
mov eax, DWORD PTR _fakearray$[ebp]
; eax=address of fakearray[0], eax+2 is fakearray[2] or array[1]
movzx ecx, BYTE PTR [eax+2]
push ecx
push OFFSET $SG2752 ; 'second element %d'
call _printf
add esp, 8
mov edx, DWORD PTR _fakearray$[ebp]
; edx=address of fakearray[0], edx+10 is fakearray[10] or array[9]
movzx eax, BYTE PTR [edx+10]
push eax
push OFFSET $SG2753 ; 'last element %d'
call _printf
add esp, 8
lea ecx, DWORD PTR _array$[ebp]
movzx edx, BYTE PTR [ecx-4]
push edx
lea eax, DWORD PTR _array$[ebp]
movzx ecx, BYTE PTR [eax-3]
push ecx
lea edx, DWORD PTR _array$[ebp]
movzx eax, BYTE PTR [edx-2]
push eax
lea ecx, DWORD PTR _array$[ebp]
movzx edx, BYTE PTR [ecx-1]
push edx
push OFFSET $SG2754 ; 'array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X'
call _printf
add esp, 20
xor eax, eax
mov esp, ebp
pop ebp
ret 0
_main ENDP

So we have array[] of ten elements, filled with 0...9 bytes. Then we have the fakearray[] pointer, which points one byte before array[]. fakearray[1] points exactly to array[0]. But we are still curious, what is there before array[]? We have added random_value before array[] and set it to 0x11223344. The non-optimizing compiler allocated the variables in the order they were declared, so yes, the 32-bit random_value is right before the array.
We ran it, and:

| first element 0 |
| second element 1 |
| last element 9 |

Here is the stack fragment we will copypaste from OllyDbg’s stack window (with comments added by the author):

Listing 52.2: Non-optimizing MSVC 2010

<table>
<thead>
<tr>
<th>CPU Stack</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>001DFBCC</td>
<td>001DFBD3</td>
<td>; fakearray pointer</td>
</tr>
<tr>
<td>001DFBD0</td>
<td>11223344</td>
<td>; random_value</td>
</tr>
<tr>
<td>001DFBD4</td>
<td>03020100</td>
<td>; 4 bytes of array[]</td>
</tr>
<tr>
<td>001DFBD8</td>
<td>07060504</td>
<td>; 4 bytes of array[]</td>
</tr>
<tr>
<td>001DFBDC</td>
<td>00CB0908</td>
<td>; random garbage + 2 last bytes of array[]</td>
</tr>
<tr>
<td>001DFBE0</td>
<td>0000000A</td>
<td>; last i value after loop was finished</td>
</tr>
<tr>
<td>001DFBE4</td>
<td>001DFC2C</td>
<td>; saved EBP value</td>
</tr>
<tr>
<td>001DFBE8</td>
<td>00CB129D</td>
<td>; Return Address</td>
</tr>
</tbody>
</table>

The pointer to the fakearray[] (0x001DFBD3) is indeed the address of array[] in the stack (0x001DFBD4), but minus 1 byte.

It’s still very hackish and dubious trick. Doubtfully anyone should use it in production code, but as a demonstration, it fits perfectly here.
Chapter 53

Windows 16-bit

16-bit Windows programs are rare nowadays, but can be used in the cases of retro-computing or dongle hacking (78 on page 1074).

16-bit Windows versions were up to 3.11. 96/98/ME also support 16-bit code, as well as the 32-bit versions of the Windows NT line. The 64-bit versions of Windows NT line do not support 16-bit executable code at all.

The code resembles MS-DOS's one.

Executable files are of type NE-type (so-called “new executable”).

All examples considered here were compiled by the OpenWatcom 1.9 compiler, using these switches:
wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows example.c

53.1 Example#1

```c
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    MessageBeep(MB_ICONEXCLAMATION);
    return 0;
};
```
WinMain proc near
push bp
mov bp, sp
mov ax, 30h ; '0' ; MB_ICONEXCLAMATION
push ax
call MESSAGEBEEP
xor ax, ax ; return 0
pop bp
retn 0Ah
WinMain endp

Seems to be easy, so far.

### 53.2 Example #2

```c
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);
    return 0;
}
```

WinMain proc near
push bp
mov bp, sp
xor ax, ax ; NULL
push ax
push ds
mov ax, offset aHelloWorld ; 0x18. "hello, world"
push ax
push ds
mov ax, offset aCaption ; 0x10. "caption"
push ax
mov ax, 3 ; MB_YESNOCANCEL
push ax
call MESSAGEBOX
xor ax, ax ; return 0
Couple important things here: the PASCAL calling convention dictates passing the first argument first (MB_YESNOCANCEL), and the last argument—last (NULL). This convention also tells the callee to restore the stack pointer: hence the RETN instruction has 0Ah as argument, which implies that the pointer has to be increased by 10 bytes when the function exits. It is like stdcall (64.2 on page 952), but the arguments are passed in “natural” order.

The pointers are passed in pairs: first the data segment is passed, then the pointer inside the segment. There is only one segment in this example, so DS always points to the data segment of the executable.

### 53.3 Example #3

```c
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow )
{
    int result=MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);
    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK);
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)
        MessageBox (NULL, "you pressed no", "caption", MB_OK);
    return 0;
}
```

WinMain proc near
    push   bp
    mov    bp, sp
```
xor ax, ax ; NULL
push ax
push ds
mov ax, offset aHelloWorld ; "hello, world"
push ax
push ds
mov ax, offset aCaption ; "caption"
push ax
mov ax, 3 ; MB_YESNOCANCEL
push ax
call MESSAGEBOX
cmp ax, 2 ; IDCANCEL
jnz short loc_2F
xor ax, ax
push ax
push ds
mov ax, offset aYouPressedCanc ; "you pressed cancel"
jmp short loc_49
loc_2F:
cmp ax, 6 ; IDYES
jnz short loc_3D
xor ax, ax
push ax
push ds
mov ax, offset aYouPressedYes ; "you pressed yes"
jmp short loc_49
loc_3D:
cmp ax, 7 ; IDNO
jnz short loc_57
xor ax, ax
push ax
push ds
mov ax, offset aYouPressedNo ; "you pressed no"
loc_49:
push ax
push ds
mov ax, offset aCaption ; "caption"
push ax
xor ax, ax
push ax
call MESSAGEBOX
loc_57:
xor ax, ax
pop bp
Somewhat extended example from the previous section.

### 53.4 Example #4

```c
#include <windows.h>

int PASCAL func1 (int a, int b, int c)
{
    return a*b+c;
};

long PASCAL func2 (long a, long b, long c)
{
    return a*b+c;
};

long PASCAL func3 (long a, long b, long c, int d)
{
    return a*b+c-d;
};

int PASCAL WinMain( WINAPI hInstance,
                      WINAPI hPrevInstance,
                      LPSTR lpCmdLine,
                      int nCmdShow )
{
    func1 (123, 456, 789);
    func2 (600000, 700000, 800000);
    func3 (600000, 700000, 800000, 123);
    return 0;
};
```

```
c1 = word ptr 4
b = word ptr 6
a = word ptr 8

push bp
mov bp, sp
mov ax, [bp+a]
```
imul [bp+b]
add ax, [bp+c]
pop bp
retn 6

func1
endp

func2 proc near

arg_0 = word ptr 4
arg_2 = word ptr 6
arg_4 = word ptr 8
arg_6 = word ptr 0Ah
arg_8 = word ptr 0Ch
arg_A = word ptr 0Eh

push bp
mov bp, sp
mov ax, [bp+arg_8]
mov dx, [bp+arg_A]
mov bx, [bp+arg_4]
mov cx, [bp+arg_6]
call sub_B2 ; long 32-bit multiplication
add ax, [bp+arg_0]
adc dx, [bp+arg_2]
pop bp
retn 12

func2 endp

func3 proc near

arg_0 = word ptr 4
arg_2 = word ptr 6
arg_4 = word ptr 8
arg_6 = word ptr 0Ah
arg_8 = word ptr 0Ch
arg_A = word ptr 0Eh
arg_C = word ptr 10h

push bp
mov bp, sp
mov ax, [bp+arg_A]
mov dx, [bp+arg_C]
mov bx, [bp+arg_6]
mov cx, [bp+arg_8]
call sub_B2 ; long 32-bit multiplication
mov cx, [bp+arg_2]
add cx, ax
mov bx, [bp+arg_4]
adc bx, dx ; BX=high part, CX=low

mov ax, [bp+arg_0]
cwd ; AX=low part d, DX=

sub cx, ax
mov ax, cx
sbb bx, dx
mov dx, bx
pop bp
retn 14

func3 endp

WinMain proc near
push bp
mov bp, sp
mov ax, 123
push ax
mov ax, 456
push ax
mov ax, 789
push ax
mov ax, 9 ; high part of 600000
push ax
mov ax, 27C0h ; low part of 600000
push ax
mov ax, 0Ah ; high part of 700000
push ax
mov ax, 0AE60h ; low part of 700000
push ax
mov ax, 0Ch ; high part of 800000
push ax
mov ax, 3500h ; low part of 800000
push ax
call func2
mov ax, 9 ; high part of 600000
push ax
mov ax, 27C0h ; low part of 600000
push ax
mov ax, 0Ah ; high part of 700000
push ax
mov ax, 0AE60h ; low part of 700000
push ax
mov ax, 0Ch ; high part of 800000
push ax
32-bit values (the long data type implies 32 bits, while int is 16-bit) in 16-bit code (both MS-DOS and Win16) are passed in pairs. It is just like when 64-bit values are used in a 32-bit environment (24 on page 543).

`sub_B2` here is a library function written by the compiler's developers that does “long multiplication”, i.e., multiplies two 32-bit values. Other compiler functions that do the same are listed here: E on page 1330, D on page 1329.

The ADD/ADC instruction pair is used for addition of compound values: ADD may set/clear the CF flag, and ADC uses it after.

The SUB/SBB instruction pair is used for subtraction: SUB may set/clear the CF flag, SBB uses it after.

32-bit values are returned from functions in the DX:AX register pair.

Constants are also passed in pairs in WinMain() here.

The int-typed 123 constant is first converted according to its sign into a 32-bit value using the CWD instruction.

### 53.5 Example #5

```c
#include <windows.h>

int PASCAL string_compare (char *s1, char *s2)
{
    while (1)
    {
        if (*s1!=*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    }
}
```
int PASCAL string_compare_far (char far *s1, char far *s2) {
    while (1) {
        if (*s1!=*s2) return 0;
        if (*s1==0 || *s2==0) return 1; // end of string
        s1++;
        s2++;
    }
}

void PASCAL remove_digits (char *s) {
    while (*s) {
        if (*s>='0' && *s<='9') {
            *s='-';
            s++;
        }
    }
}

char str[]="hello 1234 world";

int PASCAL WinMain( HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow ) {
    string_compare ("asd", "def");
    string_compare_far ("asd", "def");
    remove_digits (str);
    MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
    return 0;
}

string_compare proc near
    arg_0 = word ptr 4
    arg_2 = word ptr 6
    push   bp
    mov    bp, sp
push    si
mov     si, [bp+arg_0]
mov     bx, [bp+arg_2]

loc_12: ; CODE XREF: string_compare+21j
    mov     al, [bx]
    cmp     al, [si]
    jz      short loc_1C
    xor     ax, ax
    jmp     short loc_2B

loc_1C: ; CODE XREF: string_compare+Ej
    test    al, al
    jz      short loc_22
    jnz     short loc_27

loc_22: ; CODE XREF: string_compare+16j
    mov     ax, 1
    jmp     short loc_2B

loc_27: ; CODE XREF: string_compare+18j
    inc     bx
    inc     si
    jmp     short loc_12

loc_2B: ; CODE XREF: string_compare+12j
    ; string_compare+1Dj
    pop     si
    pop     bp
    retn    4

string_compare endp

string_compare_far proc near ; CODE XREF: WinMain+18p

arg_0 = word ptr  4
arg_2 = word ptr  6
arg_4 = word ptr  8
arg_6 = word ptr  0Ah

    push    bp
    mov     bp, sp
    push    si
    mov     si, [bp+arg_0]
    mov     bx, [bp+arg_4]
loc_3A: ; CODE XREF: string_compare_far+35j
    mov     es, [bp+arg_6]
    mov     al, es:[bx]
    mov     es, [bp+arg_2]
    cmp     al, es:[si]
    jz      short loc_4C
    xor     ax, ax
    jmp     short loc_67

loc_4C: ; CODE XREF: string_compare_far+16j
    mov     es, [bp+arg_6]
    cmp     byte ptr es:[bx], 0
    jz      short loc_5E
    mov     es, [bp+arg_2]
    cmp     byte ptr es:[si], 0
    jnz     short loc_63

loc_5E: ; CODE XREF: string_compare_far+23j
    mov     ax, 1
    jmp     short loc_67

loc_63: ; CODE XREF: string_compare_far+2Cj
    inc     bx
    inc     si
    jmp     short loc_3A

loc_67: ; CODE XREF: string_compare_far+1Aj
    pop     si
    pop     bp
    retn    8
string_compare_far endp

remove_digits proc near ; CODE XREF: WinMain+1Fp

arg_0 = word ptr 4

    push    bp
    mov     bp, sp
    mov     bx, [bp+arg_0]

loc_72: ; CODE XREF: remove_digits+18j
    mov     al, [bx]
test    al, al
jz     short loc_86
cmp    al, 30h ; '0'
jb     short loc_83
cmp    al, 39h ; '9'
ja     short loc_83
mov    byte ptr [bx], 2Dh ; '-'

loc_83: ; CODE XREF: remove_digits+Ej
        ; remove_digits+12j
        inc bx
        jmp short loc_72

loc_86: ; CODE XREF: remove_digits+Aj
        pop bp
        retn 2
remove_digits   endp

WinMain proc near ; CODE XREF: start+EDp
    push    bp
    mov     bp, sp
    mov     ax, offset aAsd ; "asd"
push    ax
    mov     ax, offset aDef ; "def"
push    ax
call    string_compare
push    ds
    mov     ax, offset aAsd ; "asd"
push    ax
    push    ds
    mov     ax, offset aDef ; "def"
push    ax
call    string_compare_far
    mov     ax, offset aHello1234World ; "hello 1234 world"
push    ax
call    remove_digits
xor     ax, ax
push    ax
    push    ds
    mov     ax, offset aHello1234World ; "hello 1234 world"
push    ax
    push    ds
    mov     ax, offset aCaption ; "caption"
push    ax
    mov     ax, 3 ; MB_YESNOCANCEL
    push    ax
Here we see a difference between the so-called “near” pointers and the “far” pointers: another weird artefact of segmented memory in 16-bit 8086.

You can read more about it here: 94 on page 1278.

“near” pointers are those which point within the current data segment. Hence, the string_compare() function takes only two 16-bit pointers, and accesses the data from the segment that DS points to (The mov al, [bx] instruction actually works like mov al, ds:[bx]—DS is implicit here).

“far” pointers are those which may point to data in another memory segment. Hence string_compare_far() takes the 16-bit pair as a pointer, loads the high part of it in the ES segment register and accesses the data through it (mov al, es:[bx]). “far” pointers are also used in my MessageBox() win16 example: 53.2 on page 840. Indeed, the Windows kernel is not aware which data segment to use when accessing text strings, so it need the complete information.

The reason for this distinction is that a compact program may use just one 64kb data segment, so it doesn’t need to pass the high part of the address, which is always the same. A bigger program may use several 64kb data segments, so it needs to specify the segment of the data each time.

It’s the same story for code segments. A compact program may have all executable code within one 64kb-segment, then all functions in it will be called using the CALL NEAR instruction, and the code flow will be returned using RETN. But if there are several code segments, then the address of the function is to be specified by a pair, it is to be called using the CALL FAR instruction, and the code flow is to be returned using RETF.

This is what is set in the compiler by specifying “memory model”.

The compilers targeting MS-DOS and Win16 have specific libraries for each memory model: they differ by pointer types for code and data.

### 53.6 Example #6

```c
#include <windows.h>
#include <time.h>
#include <stdio.h>
```
char strbuf[256];

int PASCAL WinMain( HINSTANCE hInstance,
     HINSTANCE hPrevInstance,
     LPSTR lpCmdLine,
     int nCmdShow )
{
    struct tm *t;
    time_t unix_time;

    unix_time=time(NULL);

    t=localtime (&unix_time);

    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, t->tm_mon, t->tm_mday,
             t->tm_hour, t->tm_min, t->tm_sec);

    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};

WinMain proc near
var_4 = word ptr -4
var_2 = word ptr -2

push bp
mov bp, sp
push ax
push ax
xor ax, ax
call time_
mov [bp+var_4], ax ; low part of UNIX
mov [bp+var_2], dx ; high part of UNIX
lea ax, [bp+var_4] ; take a pointer of

call localtime_
mov bx, ax ; t
push word ptr [bx] ; second
push word ptr [bx+2] ; minute
push word ptr [bx+4] ; hour
push word ptr [bx+6] ; day
push word ptr [bx+8] ; month
UNIX time is a 32-bit value, so it is returned in the DX:AX register pair and stored in two local 16-bit variables. Then a pointer to the pair is passed to the localtime() function. The localtime() function has a struct tm allocated somewhere in the guts of the C library, so only a pointer to it is returned. By the way, this also implies that the function cannot be called again until its results are used.

For the time() and localtime() functions, a Watcom calling convention is used here: the first four arguments are passed in the AX, DX, BX and CX, registers, and the rest arguments are via the stack. The functions using this convention are also marked by underscore at the end of their name.

sprintf() does not use the PASCAL calling convention, nor the Watcom one, so the arguments are passed in the normal cdecl way (64.1 on page 952).

### 53.6.1 Global variables

This is the same example, but now these variables are global:

```c
#include <windows.h>
```


```c
#include <time.h>
#include <stdio.h>

char strbuf[256];
struct tm *t;
time_t unix_time;

int PASCAL WinMain( HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPSTR lpCmdLine,
                      int nCmdShow )
{

    unix_time=time(NULL);
    t=localtime (&unix_time);

    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d",
            t->tm_year+1900, t->tm_mon, t->tm_mday,
            t->tm_hour, t->tm_min, t->tm_sec);

    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
};
```

```asm
unix_time_low dw 0
unix_time_high dw 0
t dw 0

WinMain proc near
push bp
mov bp, sp
xor ax, ax
call time_
mov unix_time_low, ax
mov unix_time_high, dx
mov ax, offset unix_time_low
call localtime_
mov bx, ax
mov t, ax ; will not be used 

; in future...
push word ptr [bx] ; seconds
push word ptr [bx+2] ; minutes
push word ptr [bx+4] ; hour
push word ptr [bx+6] ; day
push word ptr [bx+8] ; month
mov ax, [bx+0Ah] ; year
```

854
add    ax, 1900
push   ax
mov    ax, offset a04d02d02d02d02 ; "%04d-%02d
⤦
push   ax
mov    ax, offset strbuf
push   ax
call   sprintf_
add    sp, 10h
xor    ax, ax ; NULL
push   ax
push   ds
mov    ax, offset strbuf
push   ax
push   ds
mov    ax, offset aCaption ; "caption"
push   ax
xor    ax, ax ; MB_OK
push   ax
call   MESSAGEBOX
xor    ax, ax ; return 0
pop    bp
retn   0Ah
WinMain endp

is not to be used, but the compiler emitted the code which stores the value. Because it is not sure, maybe that value will eventually be used in some other module.
Part IV

Java
Chapter 54

Java

54.1 Introduction

There are some well-known decompilers for Java (or JVM bytecode in general) \(^1\). The reason is the decompilation of JVM-bytecode is somewhat easier than for lower level x86 code:

- There is much more information about the data types.
- The JVM memory model is much more rigorous and outlined.
- The Java compiler don’t do any optimizations (the JVM JIT\(^2\) does them at runtime), so the bytecode in the class files is usually pretty readable.

When can the knowledge of JVM be useful?

- Quick-and-dirty patching tasks of class files without the need to recompile the decompiler’s results.
- Analysing obfuscated code.
- Building your own obfuscator.
- Building a compiler codegenerator (back-end) targeting JVM (like Scala, Clojure, etc \(^3\)).

Let’s start with some simple pieces of code. JDK 1.7 is used everywhere, unless mentioned otherwise.

This is the command used to decompile class files everywhere: javap -c -verbose

---

\(^1\)For example, JAD: [http://varaneckas.com/jad/](http://varaneckas.com/jad/)

\(^2\)Just-in-time compilation

This is the book I used while preparing all examples: [Jav13].

## 54.2 Returning a value

Probably the simplest Java function is the one which returns some value. Oh, and we must keep in mind that there are no “free” functions in Java in common sense, they are “methods”. Each method is related to some class, so it’s not possible to define a method outside of a class. But we’ll call them “functions” anyway, for simplicity.

```java
public class ret {
    public static int main(String[] args) {
        return 0;
    }
}
```

Let’s compile it:

```
javac ret.java
```

...and decompile it using the standard Java utility:

```
javap -c -verbose ret.class
```

And we get:

```
Listing 54.1: JDK 1.7 (excerpt)

public static int main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iconst_0
1: ireturn
```

The Java developers decided that 0 is one of the busiest constants in programming, so there is a separate short one-byte `iconst_0` instruction which pushes 0 \(^4\). There are also `iconst_1` (which pushes 1), `iconst_2`, etc, up to `iconst_5`. There is also `iconst_m1` which pushes -1.

\(^4\)Just like in MIPS, where a separate register for zero constant exists: 3.5.2 on page 30.
The stack is used in JVM for passing data to called functions and also for returning values. So `iconst_0` pushes 0 into the stack. `ireturn` returns an integer value (i in name mean integer) from the TOS\(^5\).

Let's rewrite our example slightly, now we return 1234:

```java
public class ret {
    public static int main(String[] args) {
        return 1234;
    }
}
```

...we get:

Listing 54.2: JDK 1.7 (excerpt)

```java
public static int main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
    0: sipush 1234
    3: ireturn
```

`sipush (short integer)` pushes 1234 into the stack. `short` in name implies a 16-bit value is to be pushed. The number 1234 indeed fits well in a 16-bit value.

What about larger values?

```java
public class ret {
    public static int main(String[] args) {
        return 12345678;
    }
}
```

Listing 54.3: Constant pool

```plaintext
...  
#2 = Integer 12345678
...
```

```
public static int main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
```

\(^5\)Top Of Stack
It’s not possible to encode a 32-bit number in a JVM instruction opcode, the developers didn’t leave such possibility. So the 32-bit number 12345678 is stored in so called “constant pool” which is, let’s say, the library of most used constants (including strings, objects, etc).

This way of passing constants is not unique to JVM. MIPS, ARM and other RISC CPUs also can’t encode a 32-bit number in a 32-bit opcode, so the RISC CPU code (including MIPS and ARM) has to construct the value in several steps, or to keep it in the data segment: 28.3 on page 608, 29.1 on page 613.

MIPS code also traditionally has a constant pool, named “literal pool”, the segments are called “.lit4” (for 32-bit single precision floating point number constants) and “.lit8” (for 64-bit double precision floating point number constants).

Let’s try some other data types!

Boolean:

```java
public class ret {
    public static boolean main(String[] args) {
        return true;
    }
}
```

```java
public static boolean main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
    stack=1, locals=1, args_size=1  
    0: iconst_1  
    1: ireturn
```

This JVM bytecode is no different from one returning integer 1. 32-bit data slots in the stack are also used here for boolean values, like in C/C++. But one could not use returned boolean value as integer or vice versa—type information is stored in the class file and checked at runtime.

It’s the same story with a 16-bit short:

```java
public class ret {
    public static short main(String[] args) {
```
```
public static short main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
        stack=1, locals=1, args_size=1
        0: sipush 1234
        3: ireturn

...and char!
```

```
public class ret {
    public static char main(String[] args) {
        return 'A';
    }
}
```

```
public static char main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
        stack=1, locals=1, args_size=1
        0: bipush 65
        2: ireturn
```

bipush means “push byte”. Needless to say that a char in Java is 16-bit UTF-16 character, and it’s equivalent to short, but the ASCII code of the “A” character is 65, and it’s possible to use the instruction for pushing a byte in the stack.

Let’s also try a byte:

```
public class retc {
    public static byte main(String[] args) {
        return 123;
    }
}
```

```
public static byte main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
        stack=1, locals=1, args_size=1
        0: bipush 123
```
One may ask, why bother with a 16-bit short data type which internally works as a 32-bit integer? Why use a char data type if it is the same as a short data type?

The answer is simple: for data type control and source code readability. A char may essentially be the same as a short, but we quickly grasp that it’s a placeholder for an UTF-16 character, and not for some other integer value. When using short, we show everyone that the variable’s range is limited by 16 bits. It’s a very good idea to use the boolean type where needed to, instead of the C-style int.

There is also a 64-bit integer data type in Java:

```java
class ret3
{
    public static long main(String[] args)
    {
        return 1234567890123456789L;
    }
}
```

Listing 54.4: Constant pool

```java
... #2 = Long 1234567890123456789l ...
```

```
class ret
{
    public static long main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
        stack=2, locals=1, args_size=1
        0: ldc2_w #2 // long
        3: lreturn
```

The 64-bit number is also stored in a constant pool, ldc2_w loads it and lreturn (long return) returns it.

The ldc2_w instruction is also used to load double precision floating point numbers (which also occupy 64 bits) from a constant pool:

```java
class ret
{
    public static double main(String[] args)
    {
        return 123.456d;
    }
}
```
Listing 54.5: Constant pool

... #2 = Double 123.456d ...

```java
class ret {
    public static double main(java.lang.String[])
    {
        return 123.456d;
    }
}
```

dreturn stands for “return double”.

And finally, a single precision floating point number:

```java
class ret {
    public static float main(String[] args)
    {
        return 123.456f;
    }
}
```

Listing 54.6: Constant pool

... #2 = Float 123.456f ...

```java
class ret {
    public static float main(java.lang.String[])
    {
        return 123.456f;
    }
}
```

The ldc instruction used here is the same one as for loading 32-bit integer numbers from a constant pool. freturn stands for “return float”.

Now what about function that return nothing?

```java
class ret {
    public static void main(String[] args)
    {
    }
```
This means that the `return` instruction is used to return control without returning an actual value. Knowing all this, it’s very easy to deduce the function’s (or method’s) returning type from the last instruction.

### 54.3 Simple calculating functions

Let’s continue with a simple calculating functions.

```java
public class calc {
    public static int half(int a) {
        return a/2;
    }
}
```

Here’s the output when the `iconst_2` instruction is used:

```java
public static int half(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
  0: iload_0
  1: iconst_2
  2: idiv
  3: ireturn
```

`iload_0` takes the zeroth function argument and pushes it to the stack. `iconst_2` pushes 2 in the stack. After the execution of these two instructions, this is how stack looks like:

```
+-----+
TOS  ->| 2  |
       +-----+
```
idiv just takes the two values at the TOS, divides one by the other and leaves the result at TOS:

\[ \begin{array}{c|c} \text{TOS} & \text{result} \\ \hline \text{--------} \end{array} \]

ireturn takes it and returns.

Let’s proceed with double precision floating point numbers:

```java
class calc {
    public static double half_double(double a) {
        return a/2.0;
    }
}
```

Listing 54.7: Constant pool

```java
... #2 = Double 2.0d ...
```

```java
public static double half_double(double); flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=2, args_size=1
0: dload_0
1: ldc2_w #2 // double 2.0d
4: ddiv
5: dreturn
```

It’s the same, but the `ldc2_w` instruction is used to load the constant 2.0 from the constant pool. Also, the other three instructions have the `d` prefix, meaning they work with `double` data type values.

Let’s now use a function with two arguments:

```java
class calc {
    public static int sum(int a, int b) {
        return a+b;
    }
}
```
iload_0 loads the first function argument \(a\), iload_2—second \(b\). Here is the stack after the execution of both instructions:

\[
\begin{align*}
\text{TOS} & \rightarrow | b | \\
& \quad | a | \\
& \quad +---+
\end{align*}
\]

iadd adds the two values and leaves the result at TOS:

\[
\begin{align*}
\text{TOS} & \rightarrow | \text{result} | \\
& \quad +---------+
\end{align*}
\]

Let’s extend this example to the \textit{long} data type:

```java
public static long lsum(long a, long b) {
    return a+b;
}
```

...we got:

```java
public static long lsum(long, long);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=4, args_size=2
0: lload_0
1: lload_2
2: ladd
3: lreturn
```

The second \texttt{i}load instruction takes the second argument from the 2nd slot. That’s because a 64-bit \textit{long} value occupies exactly two 32-bit slots.
Slightly more complex example:

```java
public class calc {
    public static int mult_add(int a, int b, int c) {
        return a*b+c;
    }
}
```

```java
public static int mult_add(int, int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=3, args_size=3
0: iload_0
1: iload_1
2: imul
3: iload_2
4: iadd
5: ireturn
```

The first step is multiplication. The product is left at the TOS:

```
+--------+
TOS -> | product |
+--------+
```

`iload_2` loads the third argument (c) in the stack:

```
+--------+
TOS -> | c |
+--------+
    | product |
+--------+
```

Now the `iadd` instruction can add the two values.

### 54.4 JVM memory model

x86 and other low-level environments use the stack for argument passing and as a local variables storage. JVM is slightly different.

It has:
• Local variable array (LVA\textsuperscript{6}). Used as storage for incoming function arguments and local variables. Instructions like iload\_0 load values from it. istore stores values in it. In the beginning the function arguments are stored: starting at 0 or at 1 (if the zeroth argument is occupied by this pointer). Then the local variables are allocated.

Each slot has size of 32-bit. Hence, values of long and double data types occupy two slots.

• Operand stack (or just “stack”). It’s used for computations and passing arguments while calling other functions. Unlike low-level environments like x86, it’s not possible to access the stack without using instructions which explicitly pushes or pops values to/from it.

• Heap. It is used as storage for objects and arrays.

These 3 areas are isolated from each other.

54.5 Simple function calling

Math.random() returns a pseudorandom number in range of [0.0 \ldots 1.0), but let’s say that for some reason we need to devise a function that returns a number in range of [0.0 \ldots 0.5):

```java
public class HalfRandom
{
    public static double f()
    {
        return Math.random()/2;
    }
}
```

Listing 54.8: Constant pool

```java
...#2 = Methodref     #18.#19      // java/lang/Math.\$random:
  \( )D
#3 = Double       2.0d
...#12 = Utf8      ()D
...#18 = Class     #22          // java/lang/Math
#19 = NameAndType #23:#12      // random:\( )D
#22 = Utf8      java/lang/Math
#23 = Utf8      random
```

\textsuperscript{6}(Java) Local Variable Array
invokestatic calls the Math.random() function and leaves the result at the TOS. Then the result is divided by 2.0 and returned. But how is the function name encoded? It’s encoded in the constant pool using a Methodref expression. It defines the class and method names. The first field of Methodref points to a Class expression which, in turn, points to the usual text string (“java/lang/Math”). The second Methodref expression points to a NameAndType expression which also has two links to the strings. The first string is “random”, which is the name of the method. The second string is “()D”, which encodes the function’s type. It means that it returns a double value (hence the D in the string). This is the way 1) JVM can check data for type correctness; 2) Java decompilers can restore data types from a compiled class file.

Now let’s try the “Hello, world!” example:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Listing 54.9: Constant pool

```java
... #2 = Fieldref #16.#17 // java/lang/System.eyJout:Ljava/io/PrintStream; #3 = String #18 // Hello, World #4 = Methodref #19.#20 // java/io/PrintStream.println:(Ljava/lang/String;)V ...
...
... #16 = Class #23 // java/lang/System #17 = NameAndType #24:#25 // out:Ljava/io/PrintStream; #18 = Utf8 Hello, World #19 = Class #26 // java/io/PrintStream ...
```
```
#20 = NameAndType    #27:#28 // println:(Ljava/lang/String;)V
...
#23 = Utf8     java/lang/System
#24 = Utf8     out
#25 = Utf8     Ljava/io/PrintStream;
#26 = Utf8     java/io/PrintStream
#27 = Utf8     println
#28 = Utf8     (Ljava/lang/String;)V
...
```

```
public static void main(java.lang.String[]);
   flags: ACC_PUBLIC, ACC_STATIC
   Code:
      stack=2, locals=1, args_size=1
      0: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
      3: ldc #3 // String Hello, World
      5: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      8: return
```

`ldc` at offset 3 takes a pointer to the “Hello, World” string in the constant pool and pushes it in the stack. It’s called a `reference` in the Java world, but it’s rather a pointer, or an address 7.

The familiar `invokevirtual` instruction takes the information about the `println` function (or method) from the constant pool and calls it. As we may know, there are several `println` methods, one for each data type. Our case is the version of `println` intended for the `String` data type.

But what about the first `getstatic` instruction? This instruction takes a `reference` (or address of) a field of the object `System.out` and pushes it in the stack. This value is acts like the `this` pointer for the `println` method. Thus, internally, the `println` method takes two arguments for input: 1) `this`, i.e., a pointer to an object; 2) the address of the “Hello, World” string.

Indeed, `println()` is called as a method within an initialized `System.out` object.

For convenience, the `javap` utility writes all this information in the comments.

---

7 About difference in pointers and `reference's in C++ see: 51.3 on page 781.
54.6 Calling beep()

This is a simple calling of two functions without arguments:

```java
public static void main(String[] args)
{
    java.awt.Toolkit.getDefaultToolkit().beep();
}
```

```java
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
0: invokestatic #2 // Method java/⤦.rcurvearrowse.awt/Toolkit.getDefaultToolkit:()Ljava/awt/Toolkit;
3: invokevirtual #3 // Method java/⤦.rcurvearrowse.awt/Toolkit.beep:()V
6: return
```

First `invokestatic` at offset 0 calls `java.awt.Toolkit.getDefaultToolkit()`, which returns a reference to an object of class `Toolkit`. The `invokevirtual` instruction at offset 3 calls the `beep()` method of this class.

54.7 Linear congruential PRNG

Let's try a simple pseudorandom numbers generator, which we already considered once in the book (20 on page 459):

```java
public class LCG
{
    public static int rand_state;

    public void my_srand (int init)
    {
        rand_state=init;
    }

    public static int RNG_a=1664525;
    public static int RNG_c=1013904223;

    public int my_rand ()
    {
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
    }
}
```
There are couple of class fields which are initialized at start. But how? In javap output we can find the class constructor:

```java
static {};
  flags: ACC_STATIC
  Code:
    stack=1, locals=0, args_size=0
    0: ldc #5 // int 1664525
    2: putstatic #3 // Field RNG_a:I
    5: ldc #6 // int 1013904223
    7: putstatic #4 // Field RNG_c:I
   10: return
```

That's the way variables are initialized. RNG_a occupies the 3rd slot in the class and RNG_c—4th, and putstatic puts the constants there.

The `my_srand()` function just stores the input value in `rand_state`:

```java
public void my_srand(int);
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=2, args_size=2
    0: iload_1
    1: putstatic #2 // Field rand_state:I
   14: return
```

`iload_1` takes the input value and pushes it into stack. But why not `iload_0`? It's because this function may use fields of the class, and so `this` is also passed to the function as a zeroth argument. The field `rand_state` occupies the 2nd slot in the class, so `putstatic` copies the value from the TOS into the 2nd slot.

Now `my_rand()`:

```java
public int my_rand();
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=1, args_size=1
    0: getstatic #2 // Field rand_state:I
    3: getstatic #3 // Field RNG_a:I
    6: imul
    7: putstatic #2 // Field rand_state:I
```

return `rand_state & 0x7fff;`
It just loads all the values from the object's fields, does the operations and updates `rand_state`'s value using the `putstatic` instruction. At offset 20, `rand_state` is reloaded again (because it was dropped from the stack before, by `putstatic`). This looks like non-efficient code, but be sure, the JVM is usually good enough to optimize such things really well.

### 54.8 Conditional jumps

Now let's proceed to conditional jumps.
ifge jumps to offset 7 if the value at TOS is greater or equal to 0. Don’t forget, any ifXX instruction pops the value (to be compared) from the stack.

inég just negates value at TOS.

Another example:

```java
public static int min (int a, int b) {
    if (a>b)
        return b;
    return a;
}
```

We get:

```java
public static int min(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=2, args_size=2
  0: iload_0
  1: iload_1
  2: if_icmple    7
  5: iload_1
  6: ireturn
  7: iload_0
  8: ireturn
```

if_icmple pops two values and compares them. If the second one is lesser than (or equal to) the first, a jump to offset 7 is performed.

When we define max() function ...

```java
public static int max (int a, int b) {
    if (a>b)
        return a;
    return b;
}
```

...the resulting code is the same, but the last two iload instructions (at offsets 5 and 7) are swapped:

```java
public static int max(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=2, args_size=2
  0: iload_0
  1: iload_1
```
A more complex example:

```java
public class cond {
    public static void f(int i) {
        if (i<100)
            System.out.print("<100");
        if (i==100)
            System.out.print("==100");
        if (i>100)
            System.out.print(">100");
        if (i==0)
            System.out.print("==0");
    }
}
```

```java
public static void f(int); flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
    0: iload_0
    1: bipush 100
    3: if_icmpge 14
    6: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
    9: ldc #3 // String <100
   11: invokevirtual #4 // Method java/io/PrintStream.print:(Ljava/lang/String;)V
   14: iload_0
   15: bipush 100
   17: if_icmpne 28
   20: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
   23: ldc #5 // String ==100
   25: invokevirtual #4 // Method java/io/PrintStream.print:(Ljava/lang/String;)V
   28: iload_0
   29: bipush 100
   31: if_icmple 42
   34: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
```

875
ificmpge pops two values and compares them. If the second one is larger than the first, a jump to offset 14 is performed. ificmpne and ificmple work just the same, but implement different conditions.

There is also a ifne instruction at offset 43. Its name is misnomer, it would’ve be better to name it ifnz (jump if the value at TOS is not zero). And that is what it does: it jumps to offset 54 if the input value is not zero. If zero, the execution flow proceeds to offset 46, where the “==0″ string is printed.

N.B.: JVM has no unsigned data types, so the comparison instructions operate only on signed integer values.

### 54.9 Passing arguments

Let’s extend our min() / max() example:

```java
public class minmax {
    public static int min(int a, int b) {
        if (a>b)
            return b;
        return a;
    }

    public static int max(int a, int b) {
        if (a>b)
            return a;
        return b;
    }

    public static void main(String[] args) {
```
int a=123, b=456;
int max_value=max(a, b);
int min_value=min(a, b);
System.out.println(min_value);
System.out.println(max_value);
}

Here is main() function code:

```java
public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
        stack=2, locals=5, args_size=1
        0: bipush 123
        2: istore_1
        3: sipush 456
        6: istore_2
        7: iload_1
        8: iload_2
        9: invokestatic #2 // Method max:(II)V
       12: istore_3
       13: iload_1
       14: iload_2
       15: invokestatic #3 // Method min:(II)V
       18: iload_3
       19: iload_1
       20: invokevirtual #5 // Method java/io/PrintStream.println:(I)V
       23: iload_2
       24: getstatic #4 // Field java/lang/System.out:Ljava/io/PrintStream;
       27: iload_4
       28: invokevirtual #5 // Method java/io/PrintStream.println:(I)V
       31: iload_3
       32: invokevirtual #5 // Method java/io/PrintStream.println:(I)V
       35: return
```

Arguments are passed to the other function in the stack, and the return value is left on TOS.
54.10 Bitfields

All bit-wise operations work just like in any other ISA:

```java
public static int set (int a, int b)
{
    return a | 1<<b;
}

public static int clear (int a, int b)
{
    return a & (~(1<<b));
}
```

```java
public static int set(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=2, args_size=2
0: iload_0
1: iconst_1
2: iload_1
3: ishl
4: ior
5: ireturn

public static int clear(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=2, args_size=2
0: iload_0
1: iconst_1
2: iload_1
3: ishl
4: iconst_m1
5: ixor
6: iand
7: ireturn
```

`iconst_m1` loads −1 in the stack, it's the same as the 0xFFFFFFFF number. XORing with 0xFFFFFFFF has the same effect of inverting all bits (A.6.2 on page 1308).

Let's extend all data types to 64-bit `long`:

```java
public static long lset (long a, int b)
{
    return a | 1<<b;
}
```
The code is the same, but instructions with \( l \) prefix are used, which operate on 64-bit values. Also, the second argument of the function still is of type \( \text{int} \), and when the 32-bit value in it needs to be promoted to 64-bit value the \( \text{i2l} \) instruction is used, which essentially extend the value of an \( \text{integer} \) type to a \( \text{long} \) one.

## 54.11 Loops

```java
public class Loop {

    public static void main(String[] args) {
        
```
for (int i = 1; i <= 10; i++)
{
    System.out.println(i);
}

public static void main(java.lang.String[])
{flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2, args_size=1
0: iconst_1
1: istore_1
2: iload_1
3: bipush 10
5: if_icmpgt 21
8: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
11: iload_1
12: invokevirtual #3 // Method java/io/PrintStream.println:(I)V
15: iinc 1, 1
18: goto 2
21: return

iconst_1 loads 1 into TOS, istore_1 stores it in the LVA at slot 1. Why not the zeroth slot? Because the main() function has one argument (array of String) and a pointer to it (or reference) is now in the zeroth slot.

So, the i local variable will always be in 1st slot.

Instructions at offsets 3 and 5 compare i with 10. If i is larger, execution flow passes to offset 21, where the function ends. If it’s not, println is called. i is then reloaded at offset 11, for println. By the way, we call the println method for an integer, and we see this in the comments: “(I)V” (I mean integer and V mean the return type is void).

When println finishes, i is incremented at offset 15. The first operand of the instruction is the number of a slot (1), the second is the number (1) to add to the variable.

goto is just GOTO, it jumps to the beginning of the loop’s body offset 2.

Let’s proceed with a more complex example:

public class Fibonacci
{
    public static void main(String[] args)
    {

int limit = 20, f = 0, g = 1;

for (int i = 1; i <= limit; i++)
{
    f = f + g;
    g = f - g;
    System.out.println(f);
}

public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:

stack=2, locals=5, args_size=1
0: bipush 20
2: istore_1
3: iconst_0
4: istore_2
5: iconst_1
6: istore_3
7: iconst_1
8: istore 4
10: iload 4
12: iload_1
13: if_icmpgt 37
16: iload_2
17: iload_3
18: iadd
19: istore_2
20: iload_2
21: iload_3
22: isub
23: istore_3
24: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
27: iload_2
28: invokevirtual #3 // Method java/io/PrintStream.println:(I)V
31: iinc 4, 1
34: goto 10
37: return

Here is a map of the LVA slots:

- 0 — the sole argument of main()
- 1 — limit, always contains 20
We can see that the Java compiler allocates variables in LVA slots in the same order they were declared in the source code.

There are separate `istore` instructions for accessing slots 0, 1, 2 and 3, but not for 4 and larger, so there is `istore` with an additional operand at offset 8 which takes the slot number as an operand. It’s the same with `iload` at offset 10.

But isn’t it dubious to allocate another slot for the `limit` variable, which always contains 20 (so it’s a constant in essence), and reload its value so often? JVM JIT compiler is usually good enough to optimize such things. Manual intervention in the code is probably not worth it.

### 54.12 switch()

The `switch()` statement is implemented with the `tableswitch` instruction:

```java
public static void f(int a)
{
    switch (a)
    {
    case 0: System.out.println("zero"); break;
    case 1: System.out.println("one\n"); break;
    case 2: System.out.println("two\n"); break;
    case 3: System.out.println("three\n"); break;
    case 4: System.out.println("four\n"); break;
    default: System.out.println("something unknown\n"); break;
    }
}
```

As simple, as possible:

```java
public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
0: iload_0
1: tableswitch  { // 0 to 4
    0: 36
    1: 47
    2: 58
}
```
3: 69
4: 80
default: 91
}
36: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
 39: ldc #3 // String zero
 41: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
44: goto 99
47: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
 50: ldc #5 // String one
 52: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
55: goto 99
58: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
 61: ldc #6 // String two
 63: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
66: goto 99
69: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
 72: ldc #7 // String three
 74: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
77: goto 99
80: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
 83: ldc #8 // String four
 85: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
88: goto 99
91: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
 94: ldc #9 // String
 96: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
99: return
CHAPTER 54. JAVA

54.13 Arrays

54.13.1 Simple example

Let's first create an array of 10 integers and fill it:

```java
public static void main(String[] args)
{
    int a[]=new int[10];
    for (int i=0; i<10; i++)
        a[i]=i;
    dump (a);
}
```

The `newarray` instruction creates an array object of 10 `int` elements. The array's size is set with `bipush` and left at TOS. The array's type is set in `newarray` instruction's operand. After `newarray`'s execution, a `reference` (or pointer) to the newly created array in the heap is left at the TOS. `astore_1` stores the `reference` to the 1st slot in LVA. The second part of the `main()` function is the loop which stores `i` into the corresponding array element. `aload_1` gets a `reference` of the array and places it in the stack. `iastore` then stores the integer value from the stack in the array, `reference` of which is currently in TOS. The third part of the
main() function calls the dump() function. An argument for it is prepared by load_1 (offset 23).

Now let's proceed to the dump() function:

```java
public static void dump(int a[]) {
    for (int i=0; i<a.length; i++)
        System.out.println(a[i]);
}
```

The incoming reference to the array is in the zeroth slot. The a.length expression in the source code is converted to an arraylength instruction: it takes a reference to the array and leaves the array size at TOS. iaload at offset 13 is used to load array elements, it requires to array reference be present in the stack (prepared by load_0 at 11), and also an index (prepared by iload_1 at offset 12).

Needless to say, instructions prefixed with a may be mistakenly comprehended as array instructions. It's not correct. These instructions works with references to objects. And arrays and strings are objects too.

### 54.13.2 Summing elements of array

Another example:
public class ArraySum {
    public static int f (int[] a) {
        int sum = 0;
        for (int i = 0; i < a.length; i++)
            sum = sum + a[i];
        return sum;
    }
}

public static int f(int[]); flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=3, args_size=1
    0: iconst_0
    1: istore_1
    2: iconst_0
    3: istore_2
    4: iload_2
    5: aload_0
    6: arraylength
    7: if_icmpge 22
    10: iload_1
    11: aload_0
    12: iload_2
    13: iaload
    14: iadd
    15: istore_1
    16: iinc 2, 1
    19: goto 4
    22: iload_1
    23: ireturn

**LVA** slot 0 contains a *reference* to the input array. **LVA** slot 1 contains the local variable *sum*.

### 54.13.3 The only argument of the `main()` function is an array too

We'll be using the only argument of the `main()` function, which is an array of strings:

```
public class UseArgument {
    public static void main(String[] args) {
        // code...
    }
```
The zeroth argument is the program's name (like in C/C++, etc), so the 1st argument supplied by the user is 1st.

aload_0 at 11 loads a \textit{reference} of the zeroth \texttt{LVA} slot (1st and only \texttt{main()} argument). \texttt{iconst} \texttt{1} and \texttt{aaload} at 12 and 13 take a \textit{reference} to the first (counting at 0) element of array. The \textit{reference} to the string object is at \texttt{TOS} at offset 14, and it is taken from there by \texttt{println} method.

### 54.13.4 Pre-initialized array of strings

```java
class Month {
    public static String[] months = {
```
public String get_month (int i) {
    return months[i];
}

The get_month() function is simple: Функция get_month() проста:

public java.lang.String get_month(int);
    flags: ACC_PUBLIC
    Code:
        stack=2, locals=2, args_size=2
        0: getstatic #2 // Field months:[Ljava/lang/String;
         ↓ Ljava/lang/String;
        3: iload_1
        4: aaload
        5: areturn

aload operates on an array of references. Java String are objects, so the a-
instructions are used to operate on them. areturn returns a reference to a
String object.

How is the months[] array initialized?

static {};
    flags: ACC_STATIC
    Code:
        stack=4, locals=0, args_size=0
        0: bipush 12
        1: anewarray #3 // class java/lang/String;
                          ↓ lang/String
        5: dup
        6: iconst_0
        7: ldc #4 // String January
9: aastore
10: dup
11: iconst_1
12: ldc #5 // String February
14: aastore
15: dup
16: iconst_2
17: ldc #6 // String March
19: aastore
20: dup
21: iconst_3
22: ldc #7 // String April
24: aastore
25: dup
26: iconst_4
27: ldc #8 // String May
29: aastore
30: dup
31: iconst_5
32: ldc #9 // String June
34: aastore
35: dup
36: bipush 6
38: ldc #10 // String July
40: aastore
41: dup
42: bipush 7
44: ldc #11 // String August
46: aastore
47: dup
48: bipush 8
50: ldc #12 // String September
52: aastore
53: dup
54: bipush 9
56: ldc #13 // String October
58: aastore
59: dup
60: bipush 10
62: ldc #14 // String October
64: aastore
65: dup
66: bipush 11
### 54.13.5 Variadic functions

Variadic functions actually use arrays:

```java
public static void f(int... values)
{
    for (int i=0; i<values.length; i++)
        System.out.println(values[i]);
}

public static void main(String[] args)
{
    f (1,2,3,4,5);
}
```

```java
public static void f(int... values);
flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
Code:
    stack=3, locals=2, args_size=1
  0: iconst_0
  1: istore_1
  2: iload_1
  3: aaload_0
```

`anewarray` creates a new array of references (hence a prefix). The object's type is defined in the `anewarray`'s operand, it is the “java/lang/String” string. The `bipush` 12 before `anewarray` sets the array's size. We see here a new instruction for us: `dup`. It's very well-known instruction in stack computers (including the Forth programming language) which just duplicates the value at TOS. It is used here to duplicate a reference to an array, because the `aastore` instruction pops the reference to array from the stack, but subsequent `aastore` will need it again. The Java compiler concluded that it's better to generate a `dup` instead of generating a `getstatic` instruction before each array store operation (i.e., 11 times).

`aastore` puts a reference (to string) into the array at an index which is taken from TOS.

Finally, `putstatic` puts reference to the newly created array into the second field of our object, i.e., `months` field.
f() just takes an array of integers using `aload_0` at offset 3. Then it gets the array’s size, etc.

```java
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=4, locals=1, args_size=1
    0: iconst_5
    1: newarray int
    3: dup
    4: iconst_0
    5: iconst_1
    6: iastore
    7: dup
    8: iconst_1
    9: iconst_2
   10: iastore
   11: dup
   12: iconst_2
   13: iconst_3
   14: iastore
   15: dup
   16: iconst_3
   17: iconst_4
   18: iastore
   19: dup
   20: iconst_4
   21: iconst_5
   22: iastore
   23: invokestatic #4 // Method f:([I)V
   26: return
```

The array is constructed in `main()` using the `newarray` instruction, then it’s filled, and `f()` is called.
Oh, by the way, array object is not destroyed at the end of main(). There are no destructors in Java at all, because the JVM has a garbage collector which does this automatically, when it feels it needs to.

What about the format() method? It takes two arguments at input: a string and an array of objects:

```java
public PrintStream format(String format, Object... args)
```

(http://docs.oracle.com/javase/tutorial/java/data/numberformat.html)

Let's see:

```java
public static void main(String[] args)
{
    int i=123;
    double d=123.456;
    System.out.format("int: %d double: %f.%n", i, d);
}
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=7, locals=4, args_size=1
      0: bipush 123
      2: istore_1
      3: ldc2_w #2 // double 123.456
      6: dstore_2
      7: getstatic #4 // Field java/lang/System.out:Ljava/io/PrintStream;
     10: ldc #5 // String int: %d
     12: iconst_2
     13: anewarray #6 // class java/lang/Object
     16: dup
     17: iconst_0
     18: iload_1
     19: invokestatic #7 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
    22: aastore
    23: dup
    24: iconst_1
    25: dload_2
```
So values of the \textit{int} and \textit{double} types are first promoted to \texttt{Integer} and \texttt{Double} objects using the \texttt{valueOf} methods. The \texttt{format()} method needs objects of type \texttt{Object} at input, and since the \texttt{Integer} and \texttt{Double} classes are derived from the root \texttt{Object} class, they suitable for elements in the input array. On the other hand, an array is always homogeneous, i.e., it can’t contain elements of different types, which makes it impossible to push \textit{int} and \textit{double} values in it.

An array of \texttt{Object} objects is created at offset 13, an \texttt{Integer} object is added to the array at offset 22, and a \texttt{Double} object is added to the array at offset 29.

The penultimate \texttt{pop} instruction discards the element at TOS, so when \texttt{return} is executed, the stack becomes empty (or balanced).

### 54.13.6 Two-dimensional arrays

Two-dimensional arrays in Java are just one-dimensional arrays of \textit{references} to another one-dimensional arrays.

Let’s create a two-dimensional array:

```java
public static void main(String[] args) {
    int[][] a = new int[5][10];
    a[1][2]=3;
}
```

```java
public static void main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
    stack=3, locals=2, args_size=1  
    0: iconst_5  
    1: bipush 10  
    3: multianewarray #2, 2  // class "[[I"  
    7: astore_1  
    8: aload_1  
    9: iconst_1  
   10: aaload
```
11: iconst_2
12: iconst_3
13: iastore
14: return

It’s created using the multianewarray instruction: the object’s type and dimensionality are passed as operands. The array’s size (10*5) is left in stack (using the instructions iconst_5 and bipush).

A reference to row #1 is loaded at offset 10 (iconst_1 and aaload). The column is chosen using iconst_2 at offset 11. The value to be written is set at offset 12. iastore at 13 writes the array's element.

How it is an element accessed?

```java
public static int get12 (int[][] in)
{
    return in[1][2];
}
```

```java
public static int get12(int[][]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
    0: aload_0
    1: iconst_1
    2: aaload
    3: iconst_2
    4: iaload
    5: ireturn
```

A Reference to the array’s row is loaded at offset 2, the column is set at offset 3, then iaload loads the array’s element.

54.13.7 Three-dimensional arrays

Three-dimensional arrays are just one-dimensional arrays of references to one-dimensional arrays of references.

```java
public static void main(String[] args)
{
    int[][][] a = new int[5][10][15];
    a[1][2][3]=4;
    get_elem(a);
}
```
public static void main(java.lang.String[]):  
  flags: ACC_PUBLIC, ACC_STATIC  
  Code:  
  stack=3, locals=2, args_size=1  
      0: iconst_5  
      1: bipush 10  
      3: bipush 15  
      5: multianewarray #2, 3 // class "[[[I"  
      9: astore_1  
     10: aload_1  
     11: iconst_1  
     12: aaload  
     13: iconst_2  
     14: aaload  
     15: iconst_3  
     16: iconst_4  
     17: iastore  
     18: aload_1  
     19: invokestatic #3 // Method  
            get_elem:([[[I)I  
     22: pop  
     23: return

Now it takes two aaload instructions to find right reference:

public static int get_elem (int[][][] a)  
{  
    return a[1][2][3];  
}

public static int get_elem (int[][][][]);  
  flags: ACC_PUBLIC, ACC_STATIC  
  Code:  
  stack=2, locals=1, args_size=1  
      0: aload_0  
      1: iconst_1  
      2: aaload  
      3: iconst_2  
      4: aaload  
      5: iconst_3  
      6: iaload  
      7: ireturn
54.13.8 Summary

Is it possible to do a buffer overflow in Java? No, because the array's length is always present in an array object, array bounds are controlled, and an exception is to be raised in case of out-of-bounds access.

There are no multi-dimensional arrays in Java in the C/C++ sense, so Java is not very suited for fast scientific computations.

54.14 Strings

54.14.1 First example

Strings are objects and are constructed in the same way as other objects (and arrays).

```java
public static void main(String[] args)
{
    System.out.println("What is your name?");
    String input = System.console().readLine();
    System.out.println("Hello, "+input);
}
```

```java
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=2, args_size=1
  0: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
  3: ldc #3 // String What is your name?
  5: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8: invokestatic #5 // Method java/lang/System.console:()Ljava/io/Console;
 11: invokevirtual #6 // Method java/io/Console.readLine:(Ljava/lang/String;)Ljava/lang/String;
 14: astore_1
 15: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
 18: new #7 // class java/lang/StringBuilder
 21: dup
 22: invokespecial #8 // Method java/lang/StringBuilder."<init>":()V
```

896
The `readLine()` method is called at offset 11, a reference to string (which is supplied by the user) is then stored at TOS. At offset 14 the reference to string is stored in slot 1 of LVA. The string the user entered is reloaded at offset 30 and concatenated with the “Hello,” string using the `StringBuilder` class. The constructed string is then printed using `println` at offset 37.

### 54.14.2 Second example

Another example:

```java
class strings {
    public static char test (String a) {
        return a.charAt(3);
    }

    public static String concat (String a, String b) {
        return a+b;
    }
}
```

```java
public static char test(java.lang.String); flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: aload_0
1: icast_3
2: invokevirtual #2 // Method java/lang/String.charAt:(I)C
```

---

897
The string concatenation is done using StringBuilder:

```java
public static java.lang.String concat(java.lang.String, java.lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=2, args_size=2
  0: new #3 // class java/lang/StringBuilder
  3: dup
  4: invokespecial #4 // Method java/lang/StringBuilder.<init>():V
  7: aload_0
  8: invokevirtual #5 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
 11: aload_1
 12: invokevirtual #5 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
 15: invokevirtual #6 // Method java/lang/StringBuilder.toString():Ljava/lang/String;
 18: areturn
```

Another example:

```java
public static void main(String[] args)
{
  String s="Hello!";
  int n=123;
  System.out.println("s=" + s + " n=" + n);
}
```

And again, the strings are constructed using the StringBuilder class and its append method, then the constructed string is passed to println:

```java
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=3, args_size=1
  0: ldc #2 // String Hello!
  2: astore_1
  3: bipush 123
  5: istore_2
  6: getstatic #3 // Field java/lang/System.out:Ljava/io/PrintStream;
```
54.15 Exceptions

Let's rework our Month example (54.13.4 on page 887) a bit:

Listing 54.10: IncorrectMonthException.java

```java
public class IncorrectMonthException extends Exception {

    private int index;

    public IncorrectMonthException(int index) {
        this.index = index;
    }

    public int getIndex() {
        return index;
    }
}
```
Essentially, IncorrectMonthException.class has just an object constructor and one getter method.

The IncorrectMonthException class is derived from Exception, so the IncorrectMonthException constructor first calls the constructor of the Exception class, then it puts incom-
ing integer value into the sole IncorrectMonthException class field:

```java
public IncorrectMonthException(int);
flags: ACC_PUBLIC
Code:
    stack=2, locals=2, args_size=2
    0: aload_0
1: invokespecial #1 // Method java/lang/Exception."<init>":()V
  
4: aload_0
5: iload_1
6: putfield     #2 // Field index:I
9: return
```

getIndex() is just a getter. A **reference** to IncorrectMonthException is passed in the zeroth LVA slot (this), aload_0 takes it, getfield loads an integer value from the object, ireturn returns it.

```java
public int getIndex();
flags: ACC_PUBLIC
Code:
    stack=1, locals=1, args_size=1
    0: aload_0
1: getfield     #2 // Field index:I
4: ireturn
```

Now let's take a look at get_month() in Month2.class:

```java
public static java.lang.String get_month(int) throws IncorrectMonthException;
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=1, args_size=1
    0: iload_0
1: iflt 10
4: iload_0
5: bipush 11
7: if_icmple 19
10: new #2 // class IncorrectMonthException
13: dup
14: iload_0
15: invokespecial #3 // Method IncorrectMonthException."<init>":(I)V
18: athrow
19: getstatic  #4 // Field months:Ljava/lang/String;
```
iflt at offset 1 is *if less than*.

In case of invalid index, a new object is created using the new instruction at offset 10. The object's type is passed as an operand to the instruction (which is IncorrectMonthException). Then its constructor is called, and index is passed via TOS (offset 15). When the control flow is offset 18, the object is already constructed, so now the athrow instruction takes a *reference* to the newly constructed object and signals to JVM to find the appropriate exception handler.

The athrow instruction doesn't return the control flow here, so at offset 19 there is another basic block, not related to exceptions business, where we can get from offset 7.

How do handlers work?

```java
public static void main(java.lang.String[]);
   flags: ACC_PUBLIC, ACC_STATIC
   Code:
       stack=3, locals=2, args_size=1
         0: getstatic #5 // Field java/lang/System.out:Ljava/io/PrintStream;
         3: bipush 100
         5: invokestatic #6 // Method /get_month:(I)Ljava/lang/String;
         8: invokevirtual #7 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        11: goto 47
        14: astore_1
        15: getstatic #5 // Field java/lang/System.out:Ljava/io/PrintStream;
        18: new #8 // class java/lang/StringBuilder
        21: dup
        22: invokespecial #9 // Method java/lang/StringBuilder."<init>":()V
        25: ldc #10 // String incorrect month index:
        String;
        30: aload_1
        31: invokevirtual #12 // Method IncorrectMonthException:index:()I
```
34: invokevirtual #13 // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
37: invokevirtual #14 // Method java/lang/StringBuilder.toString():Ljava/lang/String;
40: invokevirtual #7 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
43: aload_1
44: invokevirtual #15 // Method IncorrectMonthException.printStackTrace():V
47: return

Exception table:
  from  to  target  type
 0     11   14   Class IncorrectMonthException

Here is the Exception table, which defines that from offsets 0 to 11 (inclusive) an exception IncorrectMonthException may happen, and if it does, the control flow is to be passed to offset 14. Indeed, the main program ends at offset 11. At offset 14 the handler starts. It’s not possible to get here, there are no conditional/unconditional jumps to this area. But JVM will transfer the execution flow here in case of an exception. The very first astore_1 (at 14) takes the incoming reference to the exception object and stores it in LVA slot 1. Later, the getIndex() method (of this exception object) will be called at offset 31. The reference to the current exception object is passed right before that (offset 30). The rest of the code is does just string manipulation: first the integer value returned by getIndex() is converted to string by the toString() method, then it’s concatenated with the “incorrect month index: ” text string (like we saw before), then println() and printStackTrace() are called. After printStackTrace() finishes, the exception is handled and we can continue with the normal execution. At offset 47 there is a return which finishes the main() function, but there could be any other code which would execute as if no exceptions were raised.

Here is an example on how IDA shows exception ranges:

Listing 54.14: from some random .class file found on the author’s computer

```java
  .catch java/io/FileNotFoundException from met001_335 to \n    met001_360
    using met001_360
  .catch java/io/FileNotFoundException from met001_185 to \n    met001_214
    using met001_214
  .catch java/io/FileNotFoundException from met001_181 to \n    met001_192
    using met001_195
  .catch java/io/FileNotFoundException from met001_155 to \n    met001_176
    using met001_176
```
54.16 Classes

Simple class:

Listing 54.15: test.java

```java
public class test {
    public static int a;
    private static int b;

    public test() {
        a=0;
        b=0;
    }
    public static void set_a (int input) {
        a=input;
    }
    public static int get_a () {
        return a;
    }
    public static void set_b (int input) {
        b=input;
    }
    public static int get_b () {
        return b;
    }
}
```

The constructor just sets both fields to zero:
public test();
  flags: ACC_PUBLIC
Code:
  stack=1, locals=1, args_size=1
  0: aload_0
  1: invokespecial #1 // Method java/lang/Object."<init>":()V
  4: iconst_0
  5: putstatic #2 // Field a:I
  8: iconst_0
  9: putstatic #3 // Field b:I
 12: return

Setter of a:

public static void set_a(int);
  flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=1, locals=1, args_size=1
  0: iload_0
  1: putstatic #2 // Field a:I
  4: return

Getter of a:

public static int get_a();
  flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=1, locals=0, args_size=0
  0: getstatic #2 // Field a:I
  3: ireturn

Setter of b:

public static void set_b(int);
  flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=1, locals=1, args_size=1
  0: iload_0
  1: putstatic #3 // Field b:I
  4: return

Getter of b:

public static int get_b();
  flags: ACC_PUBLIC, ACC_STATIC
Code:
There is no difference in the code which works with public and private fields. But this type information is present in the .class file, and it's not possible to access private fields from everywhere.

Let's create an object and call its method:

Listing 54.16: ex1.java

```java
public class ex1 {
    public static void main(String[] args) {
        test obj=new test();
        obj.set_a (1234);
        System.out.println(obj.a);
    }
}
```

The new instruction creates an object, but doesn’t call the constructor (it is called at offset 4). The `set_a()` method is called at offset 16. The `a` field is accessed using the `getstatic` instruction at offset 21.
54.17 Simple patching

54.17.1 First example

Let's proceed with a simple code patching task.

```java
public class nag {
    public static void nag_screen() {
        System.out.println("This program is not registered");
    }
    public static void main(String[] args) {
        System.out.println("Greetings from the mega-software");
        nag_screen();
    }
}
```

How would we remove the printing of “This program is not registered” string?

Let's load the .class file into IDA:

Figure 54.1: IDA

Let's patch the first byte of the function to 177 (which is the return instruction's opcode):
But that doesn't work (JRE 1.7):

```
Exception in thread "main" java.lang.VerifyError: Expecting a \  
\ stack map frame
Exception Details:
   Location:
       nag.nag_screen()V @1: nop
   Reason:
       Error exists in the bytecode
   Bytecode:
       0000000: b100 0212 03b6 0004 b1
                       at java.lang.Class.getDeclaredMethods0(Native Method)
                       at java.lang.Class.privateGetDeclaredMethods(Class.java\    
\:2615)
                       at java.lang.Class.getMethod0(Class.java:2856)
                       at java.lang.Class.getMethod(Class.java:1668)
                       at sun.launcher.LauncherHelper.getMainMethod(\    
\ LauncherHelper.java:494)
                       at sun.launcher.LauncherHelper.checkAndLoadMain(\    
\ LauncherHelper.java:486)
```

Perhaps, JVM has some other checks related to the stack maps.
OK, let's patch it differently by removing the call to \nag():
0 is the opcode for \texttt{NOP}.

Now that works!

\subsection{Second example}

Another simple crackme example:

```java
public class password {

    public static void main(String[] args) {
        System.out.println("Please enter the password");
        String input = System.console().readLine();
        if (input.equals("secret")) {
            System.out.println("password is correct");
        } else {
            System.out.println("password is not correct");
        }
    }
}
```

Let's load it in IDA:
Figure 54.4: IDA

We see here the `ifeq` instruction which does the job. Its name stands for *if equal*, and this is a misnomer, a better name would be `ifz` (*if zero*), i.e., if value at TOS is zero, then do the jump. In our example, it jumps if the password is not correct (the `equals` method returns `false`, which is 0). The very first idea is to patch this instruction. There are two bytes in `ifeq` opcode, which encode the jump offset. To make this instruction a NOP, we must set the 3rd byte to the value of 3 (because by adding 3 to the current address we will always jump to the next instruction, since the `ifeq` instruction's length is 3 bytes):

```
: Segment type: Pure code
.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 2
    .line 3
    170 000 002  getstatic java/lang/System.out [Ljava/io/PrintStream;
    018 003  ldc "Please enter the password"
    182 000 004  invokevirtual java/io/PrintStream.println([Ljava/lang/String;)V
    .line 4
    184 000 005  invokevirtual java/lang/System.console()Ljava/io/Console;
    182 000 006  invokevirtual java/io/Console.readLine()Ljava/lang/String;
    076 000 000  astore_1 ; met002_slot001
    .line 5
    043 000 000  aload_1 ; met002_slot001
    018 007  ldc "Secret"
    182 000 008  invokevirtual java/lang/String.equals([Ljava/lang/Object;)Z
    153 000 014  ifeq met002_46
    .line 6
    170 000 002  getstatic java/lang/System.out [Ljava/io/PrintStream;
    018 009  ldc "password is correct"
    182 000 004  invokevirtual java/io/PrintStream.println([Ljava/lang/String;)V
    167 000 011  goto met002_43
    .line 8

met002_35:

    .stack use locals
    .locals 1 Object java/lang/String
    .end stack
    getstatic java/lang/System.out [Ljava/io/PrintStream;
    018 010  ldc "password is not correct"
    182 000 004  invokevirtual java/io/PrintStream.println([Ljava/lang/String;)V
    .line 9
```
That doesn’t work (JRE 1.7):

Exception in thread "main" java.lang.VerifyError: Expecting a \downarrow stackmap frame at branch target 24

Exception Details:
Location: password.main([Ljava/lang/String;)V @21: ifeq
Reason: Expected stackmap frame at this location.

Bytecode:
0000000: b200 0212 03b6 0004 b800 05b6 0006 4c2b
0000010: 1207 b600 0899 0003 b200 0212 09b6 0004
0000020: a700 0bb2 0002 120a b600 04b1

Stackmap Table:
append_frame(@35,Object[#20])
same_frame(@43)

at java.lang.Class.getDeclaredMethods0(Native Method)
at java.lang.Class.privateGetDeclaredMethods(Class.java:2615)
at java.lang.Class.getMethod0(Class.java:2856)
at java.lang.Class.getMethod(Class.java:1668)
at sun.launcher.LauncherHelper.getMainMethod(LauncherHelper.java:494)
But it should be mentioned that it worked in JRE 1.6.

We can also try to replace to all 3 ifeq opcode bytes with zero bytes (NOP), and it still won’t work. Seems like there are more stack map checks in JRE 1.7.

OK, we’ll replace the whole call to the equals method with the icnst_1 instruction plus a pack of NOPs:

```
: Segment type: Pure code
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
   .line 3
   178 000 002 getstatic java/lang/System.out Ljava/io/PrintStream;
   018 003 ldc "Please enter the password"
   192 000 004 invokevirtual java/io/PrintStream.println([Ljava/lang/String;)]V
   .line 4
   194 000 005 invokestatic java/lang/System.console()Ljava/io/Console;
   192 000 006 invokevirtual java/io/Console.readLine([Ljava/lang/String;)
   876 astore_1 ; met002_slot001
   .line 5
   004 icnst_1
   000 nop
   000 nop
   000 nop
   000 nop
   000 nop
   153 000 014 ifeq met002_05
   .line 6
   178 000 002 getstatic java/lang/System.out Ljava/io/PrintStream;
   018 009 ldc "Password is correct"
   192 000 004 invokevirtual java/io/PrintStream.println([Ljava/lang/String;)]V
   167 000 011 goto met002_43
   .line 8
   met002_05: ; CODE XREF: main+21fj
178 000 002 .stack use locals
   locals Object java/lang/String
   .end stack
```

![Figure 54.6: IDA](image)

1 needs always to be in the TOS when the ifeq instruction is executed, so ifeq would never jump.

This works.

### 54.18 Summary

What is missing in Java in comparison to C/C++?

- Structures: use classes.
- Unions: use class hierarchies.
• Unsigned data types. By the way, this makes cryptographic algorithms somewhat harder to implement in Java.

• Function pointers.
Part V

Finding important/interesting stuff in the code
Minimalism is not a prominent feature of modern software.

But not because the programmers are writing a lot, but because a lot of libraries are commonly linked statically to executable files. If all external libraries were shifted into an external DLL files, the world would be different. (Another reason for C++ are the STL and other template libraries.)

Thus, it is very important to determine the origin of a function, if it is from standard library or well-known library (like Boost\(^8\), libpng\(^9\)), or if it is related to what we are trying to find in the code.

It is just absurd to rewrite all code in C/C++ to find what we’re looking for.

One of the primary tasks of a reverse engineer is to find quickly the code he/she needs.

The IDA disassembler allow us to search among text strings, byte sequences and constants. It is even possible to export the code to .lst or .asm text files and then use grep, awk, etc.

When you try to understand what some code is doing, this easily could be some open-source library like libpng. So when you see some constants or text strings which look familiar, it is always worth to google them. And if you find the open-source project where they are used, then it’s enough just to compare the functions. It may solve some part of the problem.

For example, if a program uses XML files, the first step may be determining which XML library is used for processing, since the standard (or well-known) libraries are usually used instead of self-made one.

For example, the author of these lines once tried to understand how the compression/decompression of network packets works in SAP 6.0. It is a huge software, but a detailed .PDB with debugging information is present, and that is convenient. He finally came to the idea that one of the functions, that was called CsDecomprLZC, was doing the decompression of network packets. Immediately he tried to google its name and he quickly found the function was used in MaxDB (it is an open-source SAP project)\(^10\).

http://www.google.com/search?q=CsDecomprLZC

Astoundingly, MaxDB and SAP 6.0 software shared likewise code for the compression/decompression of network packets.

---

\(^{8}\) http://go.yurichev.com/17036

\(^{9}\) http://go.yurichev.com/17037

\(^{10}\) More about it in relevant section (80.1 on page 1153)
Chapter 55

Identification of executable files

55.1 Microsoft Visual C++

MSVC versions and DLLs that can be imported:

<table>
<thead>
<tr>
<th>Marketing version</th>
<th>Internal version</th>
<th>CL.EXE version</th>
<th>DLLs that can be imported</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6.0</td>
<td>12.00</td>
<td>msvcr.dll, msvcp60.dll</td>
</tr>
<tr>
<td>.NET (2002)</td>
<td>7.0</td>
<td>13.00</td>
<td>msvcr70.dll, msvcp70.dll</td>
</tr>
<tr>
<td>.NET 2003</td>
<td>7.1</td>
<td>13.10</td>
<td>msvcr71.dll, msvcp71.dll</td>
</tr>
<tr>
<td>2005</td>
<td>8.0</td>
<td>14.00</td>
<td>msvcr80.dll, msvcp80.dll</td>
</tr>
<tr>
<td>2008</td>
<td>9.0</td>
<td>15.00</td>
<td>msvcr90.dll, msvcp90.dll</td>
</tr>
<tr>
<td>2010</td>
<td>10.0</td>
<td>16.00</td>
<td>msvcr100.dll, msvcp100.dll</td>
</tr>
<tr>
<td>2012</td>
<td>11.0</td>
<td>17.00</td>
<td>msvcr110.dll, msvcp110.dll</td>
</tr>
<tr>
<td>2013</td>
<td>12.0</td>
<td>18.00</td>
<td>msvcr120.dll, msvcp120.dll</td>
</tr>
</tbody>
</table>

msvcp*.dll contain C++-related functions, so if it is imported, this is probably a C++ program.

55.1.1 Name mangling

The names usually start with the ? symbol.

You can read more about MSVC’s name mangling here: 51.1.1 on page 757.
55.2 GCC

Aside from *NIX targets, GCC is also present in the win32 environment, in the form of Cygwin and MinGW.

55.2.1 Name mangling

Names usually start with the _Z symbols.
You can read more about GCC's name mangling here: 51.1.1 on page 757.

55.2.2 Cygwin

cygwin1.dll is often imported.

55.2.3 MinGW

msvcrt.dll may be imported.

55.3 Intel FORTRAN

libifcoremd.dll, libifportmd.dll and libiomp5md.dll (OpenMP support) may be imported.
libifcoremd.dll has a lot of functions prefixed with for_, which means FORTRAN.

55.4 Watcom, OpenWatcom

55.4.1 Name mangling

Names usually start with the W symbol.

For example, that is how the method named “method” of the class “class” that does not have any arguments and returns void is encoded:

```
W?method$_class$n__v
```
CHAPTER 55. IDENTIFICATION OF EXECUTABLE FILES

55.5 Borland

Here is an example of Borland Delphi’s and C++Builder’s name mangling:

```plaintext
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiiit1iiiiii
@TrueColorTo16BitN$qpviiiiiiit1iiiiii
@DIB24BitTo8BitBitmap$qpviiiiiit1iiiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpv1
@TrueBitmap@$bctr$qiilll
```

The names always start with the @ symbol, then we have the class name came, method name, and encoded the types of the arguments of the method.

These names can be in the .exe imports, .dll exports, debug data, etc.

Borland Visual Component Libraries (VCL) are stored in .bpl files instead of .dll ones, for example, vcl50.dll, rtl60.dll.

Another DLL that might be imported: BORLNDMM.DLL.

55.5.1 Delphi

Almost all Delphi executables has the “Boolean” text string at the beginning of the code segment, along with other type names.

This is a very typical beginning of the CODE segment of a Delphi program, this block came right after the win32 PE file header:

```plaintext
00000400 04 10 40 00 03 07 42 6f 6f 6c 65 61 6e 01 00 00 |.. Boolean...|
  ↓ @...Boolean...|
00000410 00 00 01 00 00 00 00 10 40 00 05 46 61 6c 73 65 |........@..False|
  ↓ ........@..False|
00000420 04 54 72 75 65 8d 40 00 2c 10 40 00 09 08 57 69 |True.@.,@...Wi|
  ↓ True.@.,@...Wi|
00000430 64 65 43 68 61 72 03 00 00 00 00 ff ff 00 00 90 |deChar...........|
  ↓ deChar...........|
00000440 44 10 40 00 02 04 43 68 61 72 01 00 00 00 00 ff |D.| |
  ↓ @...Char.......|
00000450 00 00 00 90 58 10 40 00 01 08 53 6d 61 6c 6c 69 |....X.@...Smalli|
  ↓ |....X.@...Smalli|
```
| @...........@. |
| @.L..M.@. |
| @......L.@.M.@. |
| @.~@.,M@.M@. |
| @.@.J@..TIn |
| terfacedObject.. |
| @...TInterface |
| @...TInterface |
| System......@. |
| TBoundArray... |
| ..................L.@..Sy |
| stem(.@...TDateT |
| ime..%H......%D. |

The first 4 bytes of the data segment (DATA) can be 00 00 00, 32 13 8B C0
or FF FF FF FF. This information can be useful when dealing with packed/encrypted Delphi executables.

55.6 Other known DLLs

- vcomp*.dll—Microsoft's implementation of OpenMP.
Chapter 56

Communication with the outer world (win32)

Sometimes it’s enough to observe some function’s inputs and outputs in order to understand what it does. That way you can save time.

Files and registry access: for the very basic analysis, Process Monitor\(^1\) utility from SysInternals can help.

For the basic analysis of network accesses, Wireshark\(^2\) can be useful.

But then you will have to to look inside anyway.

The first thing to look for is which functions from the OS’s APIs and standard libraries are used.

If the program is divided into a main executable file and a group of DLL files, sometimes the names of the functions in these DLLs can help.

If we are interested in exactly what can lead to a call to \texttt{MessageBox()} with specific text, we can try to find this text in the data segment, find the references to it and find the points from which the control may be passed to the \texttt{MessageBox()} call we’re interested in.

If we are talking about a video game and we’re interested in which events are more or less random in it, we may try to find the \texttt{rand()} function or its replacements (like the Mersenne twister algorithm) and find the places from which those functions are called, and more importantly, how are the results used. One example:

\(^1\)http://go.yurichev.com/17301
\(^2\)http://go.yurichev.com/17303
\(^3\)Application programming interface
But if it is not a game, and \texttt{rand()} is still used, it is also interesting to know why. There are cases of unexpected \texttt{rand()} usage in data compression algorithms (for encryption imitation): blog.yurichev.com.

### 56.1 Often used functions in the Windows API

These functions may be among the imported. It is worth to note that not every function might be used in the code that was written by the programmer. A lot of functions might be called from library functions and CRT code.

- Registry access (advapi32.dll): `RegEnumKeyEx`, `Reg EnumValue`, `RegGet-Value`, `RegOpenKeyEx`, `RegQueryValueEx`.
- Dialog boxes (user32.dll): `MessageBox`, `MessageBoxEx`, `SetDlgItemText`, `GetDlgItemText`.
- Resources access (68.2.8 on page 997) : (user32.dll): `LoadMenu`.
- TCP/IP networking (ws2_32.dll): `WSARecv`, `WSASend`.
- File access (kernel32.dll): `CreateFile`, `ReadFile`, `ReadFileEx`, `WriteFile`, `WriteFileEx`.
- High-level access to the Internet (wininet.dll): `WinHttpOpen`.

---

4 MSDN
5 May have the -A suffix for the ASCII version and -W for the Unicode version
6 MSDN
7 MSDN
8 MSDN
9 MSDN
10 MSDN
11 MSDN
12 MSDN
13 MSDN
14 MSDN
15 MSDN
16 MSDN
17 MSDN
18 MSDN
19 MSDN
20 MSDN
21 MSDN
22 MSDN
23 MSDN
• Checking the digital signature of an executable file (wintrust.dll): WinVerifyTrust.

• The standard MSVC library (if it’s linked dynamically) (msvcr*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

### 56.2 tracer: Intercepting all functions in specific module

There are INT3 breakpoints in the tracer, that are triggered only once, however, they can be set for all functions in a specific DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Or, let’s set INT3 breakpoints on all functions with the xml prefix in their name:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

On the other side of the coin, such breakpoints are triggered only once.

Tracer will show the call of a function, if it happens, but only once. Another drawback—it is impossible to see the function’s arguments.

Nevertheless, this feature is very useful when you know that the program uses a DLL, but you do not know which functions are actually used. And there are a lot of functions.

For example, let’s see, what does the uptime utility from cygwin use:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Thus we may see all that cygwin1.dll library functions that were called at least once, and where from:

<table>
<thead>
<tr>
<th>One-time INT3 breakpoint:</th>
<th>cygwin1.dll!__main (called from uptime.exe!OEP+0x6d (0x40106d))</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-time INT3 breakpoint:</td>
<td>cygwin1.dll!__geteuid32 (called from uptime.exe!OEP+0xba3 (0x401ba3))</td>
</tr>
<tr>
<td>One-time INT3 breakpoint:</td>
<td>cygwin1.dll!__getuid32 (called from uptime.exe!OEP+0xbaa (0x401baa))</td>
</tr>
<tr>
<td>One-time INT3 breakpoint:</td>
<td>cygwin1.dll!__getegid32 (called from uptime.exe!OEP+0xcb7 (0x401cb7))</td>
</tr>
<tr>
<td>One-time INT3 breakpoint:</td>
<td>cygwin1.dll!__getgid32 (called from uptime.exe!OEP+0xcbe (0x401cbe))</td>
</tr>
</tbody>
</table>

24MSDN
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!OEP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP+0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!OEP+0x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!OEP+0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!OEP+0x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!OEP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!OEP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!OEP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!OEP+0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!strftime (called from uptime.exe!OEP+0x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!OEP+0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!OEP+0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!OEP+0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!OEP+0x4c3 (0x4014c3))
Chapter 57

Strings

57.1 Text strings

57.1.1 C/C++

The normal C strings are zero-terminated (ASCIIZ-strings).

The reason why the C string format is as it is (zero-terminated) is apparently historical. In [Rit79] we read:

A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

In Hiew or FAR Manager these strings looks like this:

```c
int main()
{
    printf("Hello, world!\n");
}
```

57.1.2 Borland Delphi

The string in Pascal and Borland Delphi is preceded by an 8-bit or 32-bit string length.

For example:

Listing 57.1: Delphi

```
CODE:00518AC8 dd 19h
CODE:00518ACC aLoading___Plea db 'Loading... , please wait.',0
...
CODE:00518AFC dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run... ',0
```

57.1.3 Unicode

Often, what is called Unicode is a methods for encoding strings where each character occupies 2 bytes or 16 bits. This is a common terminological mistake. Unicode is a standard for assigning a number to each character in the many writing systems of the world, but does not describe the encoding method.

The most popular encoding methods are: UTF-8 (is widespread in Internet and *NIX systems) and UTF-16LE (is used in Windows).
UTF-8 is one of the most successful methods for encoding characters. All Latin symbols are encoded just like in ASCII, and the symbols beyond the ASCII table are encoded using several bytes. 0 is encoded as before, so all standard C string functions work with UTF-8 strings just like any other string.

Let’s see how the symbols in various languages are encoded in UTF-8 and how it looks like in FAR, using the 437 codepage 1:

![Figure 57.2: FAR: UTF-8](image)

As you can see, the English language string looks the same as it is in ASCII. The Hungarian language uses some Latin symbols plus symbols with diacritic marks. These symbols are encoded using several bytes, these are underscored with red. It’s the same story with the Icelandic and Polish languages. There is also the “Euro”

---

1The example and translations was taken from here: [http://go.yurichev.com/17304](http://go.yurichev.com/17304)
currency symbol at the start, which is encoded with 3 bytes. The rest of the writing systems here have no connection with Latin. At least in Russian, Arabic, Hebrew and Hindi we can see some recurring bytes, and that is not surprise: all symbols from a writing system are usually located in the same Unicode table, so their code begins with the same numbers.

At the beginning, before the “How much?” string we see 3 bytes, which are in fact the BOM\(^2\). The BOM defines the encoding system to be used.

**UTF-16LE**

Many win32 functions in Windows have the suffixes -A and -W. The first type of functions works with normal strings, the other with UTF-16LE strings (wide). In the second case, each symbol is usually stored in a 16-bit value of type short.

The Latin symbols in UTF-16 strings look in Hiew or FAR like they are interleaved with zero byte:

```c
int wmain()
{
    wprintf (L"Hello, world!
";
};
```

![Figure 57.3: Hiew](image)

We can see this often in Windows NT system files:

\(^2\)Byte order mark
Strings with characters that occupy exactly 2 bytes are called “Unicode” in IDA:

```assembly
.data:0040E000 aHelloWorld:
.data:0040E000 unicode 0, <Hello, world!>
.data:0040E000 dw 0Ah, 0
```

Here is how the Russian language string is encoded in UTF-16LE:

What we can easily spot is that the symbols are interleaved by the diamond character (which has the ASCII code of 4). Indeed, the Cyrillic symbols are located in the fourth Unicode plane \(^3\). Hence, all Cyrillic symbols in UTF-16LE are located in the 0x400-0x4FF range.

\(^3\)wikipedia
Let's go back to the example with the string written in multiple languages. Here is how it looks like in UTF-16LE.

![Figure 57.6: FAR: UTF-16LE](image)

Here we can also see the **BOM** in the beginning. All Latin characters are interleaved with a zero byte. Some characters with diacritic marks (Hungarian and Icelandic languages) are also underscored in red.

### 57.1.4 Base64

The base64 encoding is highly popular for the cases when you need to transfer binary data as a text string. In essence, this algorithm encodes 3 binary bytes into 4 printable characters: all 26 Latin letters (both lower and upper case), digits, plus sign (“+”) and slash sign (“/”), 64 characters in total.

One distinctive feature of base64 strings is that they often (but not always) ends with 1 or 2 padding equality symbol(s) (“=”), for example:

AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+
qEJAp91A0uWs=
The equality sign (\(=\)) is never encountered in the middle of base64-encoded strings.

### 57.2 Error/debug messages

Debugging messages are very helpful if present. In some sense, the debugging messages are reporting what’s going on in the program right now. Often these are printf()-like functions, which write to log-files, or sometimes do not writing anything but the calls are still present since the build is not a debug one but release one. If local or global variables are dumped in debug messages, it might be helpful as well since it is possible to get at least the variable names. For example, one of such function in Oracle RDBMS is ksdwrt().

Meaningful text strings are often helpful. The IDA disassembler may show from which function and from which point this specific string is used. Funny cases sometimes happen\(^4\).

The error messages may help us as well. In Oracle RDBMS, errors are reported using a group of functions. You can read more about them here: [blog.yurichev.com](http://blog.yurichev.com).

It is possible to find quickly which functions report errors and in which conditions. By the way, this is often the reason for copy-protection systems to inarticulate cryptic error messages or just error numbers. No one is happy when the software cracker quickly understand why the copy-protection is triggered just by the error message.

One example of encrypted error messages is here: [78.2 on page 1085](http://sekurak.pl/tp-link-httptftp-backdoor/).

### 57.3 Suspicious magic strings

Some magic strings which are usually used in backdoors looks pretty suspicious. For example, there was a backdoor in the TP-Link WR740 home router\(^5\). The backdoor was activated using the following URL:

http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html

Indeed, the “userRpmNatDebugRpm26525557” string is present in the firmware. This string was not googleable until the wide disclosure of information about the

---

\(^4\)blog.yurichev.com

\(^5\)http://sekurak.pl/tp-link-httptftp-backdoor/
backdoor. You would not find this in any RFC\(^6\). You would not find any computer science algorithm which uses such strange byte sequences. And it doesn’t look like an error or debugging message. So it’s a good idea to inspect the usage of such weird strings.

Sometimes, such strings are encoded using base64. So it’s a good idea to decode them all and to scan them visually, even a glance should be enough.

More precise, this method of hiding backdoors is called “security through obscurity”.

\(^6\)Request for Comments
Chapter 58

Calls to assert()

Sometimes the presence of the `assert()` macro is useful too: commonly this macro leaves source file name, line number and condition in the code.

The most useful information is contained in the assert’s condition, we can deduce variable names or structure field names from it. Another useful piece of information are the file names—we can try to deduce what type of code is there. Also it is possible to recognize well-known open-source libraries by the file names.

Listing 58.1: Example of informative assert() calls

```assembly
.text:107D4B29 mov dx, [ecx+42h]
.text:107D4B2D cmp edx, 1
.text:107D4B30 jz short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ; "td->rcurvearrowse
.text:107D4B41 call ds:_assert
...

.text:107D52CA mov edx, [ebp-4]
.text:107D52CD and edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30 ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert
...
```
It is advisable to “google” both the conditions and file names, which can lead us to an open-source library. For example, if we “google” “sp->lzw_nbits <= BITS_MAX”, this predictably gives us some open-source code that’s related to the LZW compression.
Chapter 59

Constants

Humans, including programmers, often use round numbers like 10, 100, 1000, in real life as well as in the code.

The practicing reverse engineer usually know them well in hexadecimal representation: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

The constants 0xAAAAAAAA (10101010101010101010101010101010) and 0x55555555 (01010101010101010101010101010101) are also popular—those are composed of alternating bits. That may help to distinguish some signal from a signal where all bits are turned on (1111 ...) or off (0000 ...). For example, the 0x55AA constant is used at least in the boot sector, MBR\(^1\), and in the ROM of IBM-compatible extension cards.

Some algorithms, especially cryptographical ones use distinct constants, which are easy to find in code using IDA.

For example, the MD5\(^2\) algorithm initializes its own internal variables like this:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

If you find these four constants used in the code in a row, it is highly probable that this function is related to MD5.

Another example are the CRC16/CRC32 algorithms, whose calculation algorithms often use precomputed tables like this one:

---

\(^1\) Master Boot Record

\(^2\) wikipedia
/** CRC table for the CRC-16. The poly is \(0x8005 \cdot x^{16} + x^{15} + x^2 + 1\) */

```c
u16 const crc16_table[256] = {
    0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
    0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
    0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
    ...  // continuation
};
```

See also the precomputed table for CRC32: [37 on page 647](#).

## 59.1 Magic numbers

A lot of file formats define a standard file header where a *magic number(s)*[^3] is used, single one or even several.

For example, all Win32 and MS-DOS executables start with the two characters “MZ”[^4].

At the beginning of a MIDI file the “MThd” signature must be present. If we have a program which uses MIDI files for something, it’s very likely that it must check the file for validity by checking at least the first 4 bytes.

This could be done like this:

```c
(buf points to the beginning of the loaded file in memory)

cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...or by calling a function for comparing memory blocks like `memcmp()`, or any other equivalent code up to a CMPSB ([A.6.3 on page 1310](#)) instruction.

When you find such point you already can say where the loading of the MIDI file starts, also, we could see the location of the buffer with the contents of the MIDI file, what is used from the buffer, and how.

[^3]: [wikipedia](#)
[^4]: [wikipedia](#)
59.1.1 Dates

Often, one may encounter number like 0x19861115, which is clearly looks like a date (year 1986, 11th month (November), 15th day). This may be someone’s birthday (a programmer, his/her relative, child), or some other important date. The date may also be written in a reverse order, like 0x15111986.

Well-known example is 0x19540119 (magic number used in UFS2 superblock structure), which is a birthday of Marshall Kirk McKusick, prominent FreeBSD contributor.

Also, numbers like those are very popular in amateur-grade cryptography, for example, excerpt from the secret function internals from HASP3 dongle ⁵:

```c
void xor_pwd(void)
{
    int i;

    pwd ^= 0x09071966;
    for(i=0;i<8;i++)
    {
        al_buf[i] = pwd & 7; pwd = pwd >> 3;
    }
}

void emulate_func2(unsigned short seed)
{
    int i, j;
    for(i=0;i<8;i++)
    {
        ch[i] = 0;

        for(j=0;j<8;j++)
        {
            seed *= 0x1989;
            seed += 5;
            ch[i] |= (tab[(seed>>9)&0x3f]) << (7-j);
        }
    }
}
```

59.1.2 DHCP

This applies to network protocols as well. For example, the DHCP protocol’s network packets contain the so-called *magic cookie*: 0x63538263. Any code that generates DHCP packets somewhere must embed this constant into the packet. If we find it in the code we may find where this happens and, not only that. Any program which can receive DHCP packet must verify the *magic cookie*, comparing it with the constant.

For example, let’s take the dhcpcore.dll file from Windows 7 x64 and search for the constant. And we can find it, twice: it seems that the constant is used in two functions with descriptive names like DhcpExtractOptionsForValidation() and DhcpExtractFullOptions():

Listing 59.2: dhcpcore.dll (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h
⤦
 рискет; DATA XREF: DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h
⤦
 рискет; DATA XREF: DhcpExtractFullOptions+97
```

And here are the places where these constants are accessed:

Listing 59.3: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF6480875F mov eax, [rsi]
.text:000007FF64808761 cmp eax, cs:dword_7FF6483CBE8
.text:000007FF64808767 jnz loc_7FF64817179
```

And:

Listing 59.4: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF648082C7 mov eax, [r12]
.text:000007FF648082C8 cmp eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1 jnz loc_7FF648173AF
```

59.2 Searching for constants

It is easy in IDA: Alt-B or Alt-I. And for searching for a constant in a big pile of files, or for searching in non-executable files, there is a small utility called *binary grep*.

---

6 GitHub
Chapter 60

Finding the right instructions

If the program is utilizing FPU instructions and there are very few of them in the code, one can try to check each one manually with a debugger.

For example, we may be interested how Microsoft Excel calculates the formulae entered by user. For example, the division operation.

If we load excel.exe (from Office 2010) version 14.0.4756.1000 into IDA, make a full listing and to find every FDIV instruction (except the ones which use constants as a second operand—obviously, they do not suit us):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...then we see that there are 144 of them.

We can enter a string like \((1/3)\) in Excel and check each instruction.

By checking each instruction in a debugger or tracer (one may check 4 instruction at a time), we get lucky and the sought-for instruction is just the 14th:

```
.text:3011E919 DC 33  fdiv  ✓
  ↘ qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=
FPU ST(0): 1.000000
```
ST(0) holds the first argument (1) and second one is in [EBX].

The instruction after FDIV (FSTP) writes the result in memory:

```
.text:3011E91B DD 1E
text:3011E91B qword ptr [esi]
```

If we set a breakpoint on it, we can see the result:

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000EDI=0x00294804EBP=0x026CF93CESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

Also as a practical joke, we can modify it on the fly:

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000EDI=0x00395404EBP=0x0290FD9CESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel shows 666 in the cell, finally convincing us that we have found the right point.
If we try the same Excel version, but in x64, we will find only 12 FDIV instructions there, and the one we looking for is the third one.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC, set(st0\n,666)
```

It seems that a lot of division operations of float and double types, were replaced by the compiler with SSE instructions like DIVSD (DIVSD is present 268 times in total).
Chapter 61

Suspicious code patterns

61.1 XOR instructions

Instructions like XOR op, op (for example, XOR EAX, EAX) are usually used for setting the register value to zero, but if the operands are different, the “exclusive or” operation is executed. This operation is rare in common programming, but widespread in cryptography, including amateur one. It’s especially suspicious if the second operand is a big number. This may point to encrypting/decrypting, checksum computing, etc.

One exception to this observation worth noting is the “canary” (18.3 on page 378). Its generation and checking are often done using the XOR instruction.

This AWK script can be used for processing IDA listing (.lst) files:

```bash
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if($4!="esp") if ($4!="ebp") { print $1, $2, tmp, ",", $4 } }' filename.lst
```

It is also worth noting that this kind of script can also match incorrectly disassembled code (49 on page 742).

61.2 Hand-written assembly code

Modern compilers do not emit the LOOP and RCL instructions. On the other hand, these instructions are well-known to coders who like to code directly in assembly
language. If you spot these, it can be said that there is a high probability that this fragment of code was hand-written. Such instructions are marked as (M) in the instructions list in this appendix: A.6 on page 1301.

Also the function prologue/epilogue are not commonly present in hand-written assembly.

Commonly there is no fixed system for passing arguments to functions in the hand-written code.

Example from the Windows 2003 kernel (ntoskrnl.exe file):

```
MultiplyTest proc near ; CODE XREF: 
    xor cx, cx
loc_620555: ; CODE XREF: MultiplyTest+ 
    push cx
    call Multiply
    pop cx
    jb short locret_620563
    loop loc_620555
    clc
locret_620563: ; CODE XREF: MultiplyTest+ 
    retn

Multiply proc near ; CODE XREF: MultiplyTest 
    mov ecx, 81h
    mov eax, 417A000h
    mul ecx
    cmp edx, 2
    stc
    jnz short locret_62057F
    cmp eax, 0FE7A000h
    stc
    jnz short locret_62057F
    clc
locret_62057F: ; CODE XREF: Multiply+10
    ret
    ; Multiply+18

Multiply endp
```

Indeed, if we look in the WRK\textsuperscript{1} v1.2 source code, this code can be found easily in file WRK-v1.2\base\ntos\ke\i386\cpu.asm.

\textsuperscript{1}Windows Research Kernel
Chapter 62

Using magic numbers while tracing

Often, our main goal is to understand how the program uses a value that was either read from file or received via network. The manual tracing of a value is often a very labour-intensive task. One of the simplest techniques for this (although not 100% reliable) is to use your own magic number.

This resembles X-ray computed tomography in some sense: a radiocontrast agent is injected into the patient’s blood, which is then used to improve the visibility of the patient’s internal structure in to the X-rays. It is well known how the blood of healthy humans percolates in the kidneys and if the agent is in the blood, it can be easily seen on tomography, how blood is percolating, and are there any stones or tumors.

We can take a 32-bit number like 0x0badf00d, or someone’s birth date like 0x11101979 and write this 4-byte number to some point in a file used by the program we investigate.

Then, while tracing this program with tracer in code coverage mode, with the help of grep or just by searching in the text file (of tracing results), we can easily see where the value was used and how.

Example of grepable tracer results in cc mode:

0x150bf66 (_kziaia+0x14), e= 1 [MOV EBX, [EBP+8]] [EBP +8]=0xf59c934
0x150bf69 (_kziaia+0x17), e= 1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e= 1 [FS: MOV EAX, [2Ch]]
This can be used for network packets as well. It is important for the *magic number* to be unique and not to be present in the program’s code.

Aside of the tracer, DosBox (MS-DOS emulator) in heavydebug mode is able to write information about all registers’ states for each executed instruction of the program to a plain text file\(^1\), so this technique may be useful for DOS programs as well.

\(^1\)See also my blog post about this DosBox feature: [blog.yurichev.com](http://blog.yurichev.com)
Chapter 63

Other things

63.1  General idea

A reverse engineer should try to be in programmer’s shoes as often as possible. To take his/her viewpoint and ask himself, how would one solve some task the specific case.

63.2  C++

RTTI (51.1.5 on page 779)-data may be also useful for C++ class identification.

63.3  Some binary file patterns

Sometimes, we can clearly spot an array of 16/32/64-bit values visually, in hex editor. Here is an example of very typical MIPS code. As we may remember, every MIPS (and also ARM in ARM mode or ARM64) instruction has size of 32 bits (or 4 bytes), so such code is array of 32-bit values. By looking at this screenshot, we may see some kind of pattern. Vertical red lines are added for clarity:
Figure 63.1: Hiew: very typical MIPS code

Another example of such pattern here is book: **86 on page 1234**.

### 63.4 Memory “snapshots” comparing

The technique of the straightforward comparison of two memory snapshots in order to see changes was often used to hack 8-bit computer games and for hacking “high
score” files.

For example, if you had a loaded game on an 8-bit computer (there isn’t much memory on these, but the game usually consumes even less memory) and you know that you have now, let’s say, 100 bullets, you can do a “snapshot” of all memory and back it up to some place. Then shoot once, the bullet count goes to 99, do a second “snapshot” and then compare both: the must be must be a byte somewhere which was 100 in the beginning, and now it is 99. Considering the fact that these 8-bit games were often written in assembly language and such variables were global, it can be said for sure which address in memory was holding the bullet count. If you searched for all references to the address in the disassembled game code, it was not very hard to find a piece of code decrementing the bullet count, then to write a NOP instruction there, or a couple of NOP-s, and then have a game with 100 bullets forever. Games on these 8-bit computers were commonly loaded at the constant address, also, there were not much different versions of each game (commonly just one version was popular for a long span of time), so enthusiastic gamers knew which bytes must be overwritten (using the BASIC’s instruction POKE) at which address in order to hack it. This led to “cheat” lists that contained POKE instructions, published in magazines related to 8-bit games. See also: wikipedia.

Likewise, it is easy to modify “high score” files, this does not work with just 8-bit games. Notice your score count and back up the file somewhere. When the “high score” count gets different, just compare the two files, it can even be done with the DOS utility FC1 (“high score” files are often in binary form). There will be a point where a couple of bytes are different and it is easy to see which ones are holding the score number. However, game developers are fully aware of such tricks and may defend the program against it.

Somewhat similar example in this book is: 85 on page 1227.

### 63.4.1 Windows registry

It is also possible to compare the Windows registry before and after a program installation. It is a very popular method of finding which registry elements are used by the program. Probably, this is the reason why the “windows registry cleaner” shareware is so popular.

### 63.4.2 Blink-comparator

Comparison of files or memory snapshots remind us blink-comparator 2: a device used by astronomers in past, intended to find moving celestial objects. Blink-

---

1 MS-DOS utility for comparing binary files
2 [http://go.yurichev.com/17348](http://go.yurichev.com/17348)
comparator allows to switch quickly between two photographies shot in different
time, so astronomer would spot the difference visually. By the way, Pluto was
discovered by blink-comparator in 1930.
Part VI

OS-specific
Chapter 64

Arguments passing methods (calling conventions)

64.1 cdecl

This is the most popular method for passing arguments to functions in the C/C++ languages.

The glscaller also must return the value of the stack pointer (ESP) to its initial state after the callee function exits.

Listing 64.1: cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

64.2 stdcall

It’s almost the same as cdecl, with the exception that the callee must set ESP to the initial state by executing the RET x instruction instead of RET, where x = arguments number * sizeof(int). The caller is not adjusting the stack pointer, there are no add esp, x instruction.

1 The size of an int type variable is 4 in x86 systems and 8 in x64 systems
The method is ubiquitous in win32 standard libraries, but not in win64 (see below about win64).

For example, we can take the function from 8.1 on page 132 and change it slightly by adding the __stdcall modifier:

```c
int __stdcall f2 (int a, int b, int c)
{
    return a*b+c;
};
```

It is to be compiled in almost the same way as 8.2 on page 132, but you will see RET 12 instead of RET. SP is not update in the caller.

As a consequence, the number of function arguments can be easily deduced from the RETN n instruction: just divide n by 4.
64.2.1 Functions with variable number of arguments

printf()-like functions are, probably, the only case of functions with a variable number of arguments in C/C++, but it is easy to illustrate an important difference between cdecl and stdcall with their help. Let's start with the idea that the compiler knows the argument count of each printf() function call. However, the called printf(), which is already compiled and located in MSVCRT.DLL (if we talk about Windows), does not have any information about how much arguments were passed, however it can determine it from the format string. Thus, if printf() would be a stdcall function and restored stack pointer to its initial state by counting the number of arguments in the format string, this could be a dangerous situation, when one programmer's typo can provoke a sudden program crash. Thus it is not suitable for such functions to use stdcall, cdecl is better.

64.3 fastcall

That's the general naming for the method of passing some arguments via registers and the rest via the stack. It worked faster than cdecl/stdcall on older CPUs (because of smaller stack pressure). It may not help to gain any significant performance on modern (much more complex) CPUs, however.

It is not standardized, so the various compilers can do it differently. It's a well known caveat: if you have two DLLs and the one uses another one, and they are built by different compilers with differentfastcall calling conventions, you can expect problems.

Both MSVC and GCC pass the first and second arguments via ECX and EDX and the rest of the arguments via the stack.

The stack pointer must be restored to its initial state by the callee (like in stdcall).

Listing 64.4: fastcall

```
push arg3
mov edx, arg2
mov ecx, arg1
```
call function

function:  
.. do something ..
ret 4

For example, we may take the function from 8.1 on page 132 and change it slightly by adding a __fastcall modifier:

```c
int __fastcall f3 (int a, int b, int c) {
    return a*b+c;
};
```

Here is how it is to be compiled:

```
Listing 64.5: Optimizing MSVC 2010 /Ob0

_c$ = 8 ; size  
creta $ = 4  
@f3@12  PROC  
_a$ = ecx  
_b$ = edx  
  mov  eax, ecx  
  imul  eax, edx  
  add  eax, DWORD PTR _c$[esp-4]  
  ret  4  
@f3@12  ENDP  

; ...
  mov  edx, 2  
  push  3  
  lea  ecx, DWORD PTR [edx-1]  
  call  @f3@12  
  push  eax  
  push  OFFSET $SG81390  
  call  _printf  
  add  esp, 8
```

We see that the callee returns SP by using the RETN instruction with an operand. Which implies that the number of arguments can be deduced easily here as well.
64.3.1 GCC regparm

It is the evolution of fastcall\(^2\) in some sense. With the -mregparm option it is possible to set how many arguments are to be passed via registers (3 is the maximum). Thus, the EAX, EDX and ECX registers are to be used.

Of course, if the number the of arguments is less than 3, not all 3 registers are to be used.

The caller restores the stack pointer to its initial state.

For example, see (19.1.1 on page 414).

64.3.2 Watcom/OpenWatcom

Here it is called “register calling convention”. The first 4 arguments are passed via the EAX, EDX, EBX and ECX registers. All the rest—via the stack. These functions has an underscore appended to the function name in order to distinguish them from those having a different calling convention.

64.4 thiscall

This is passing the object’s this pointer to the function-method, in C++.

In MSVC, this is usually passed in the ECX register.

In GCC, the this pointer is passed as the first function-method argument. Thus it will be very visible that internally: all function-methods have an extra argument.

For an example, see (51.1.1 on page 757).

64.5 x86-64

64.5.1 Windows x64

The method of for passing arguments in Win64 somewhat resembles fastcall. The first 4 arguments are passed via RCX, RDX, R8 and R9, the rest—via the stack. The caller also must prepare space for 32 bytes or 4 64-bit values, so then the callee can save there the first 4 arguments. Short functions may use the arguments' values just from the registers, but larger ones may save their values for further use.

\(^2\)http://go.yurichev.com/17040
The caller also must return the stack pointer into its initial state.

This calling convention is also used in Windows x86-64 system DLLs (instead of stdcall in win32).

Example:

```c
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
    printf(\"%d %d %d %d %d %d %d\n\", a, b, c, d, e, f, g);
};

int main()
{
    f1(1,2,3,4,5,6,7);
};
```

Listing 64.6: MSVC 2012 /0b

```assembly
$SG2937 DB '%d %d %d %d %d %d %d', 0aH, 00H
main PROC
sub rsp, 72 ;
00000048H
    mov DWORD PTR [rsp+48], 7
    mov DWORD PTR [rsp+40], 6
    mov DWORD PTR [rsp+32], 5
    mov r9d, 4
    mov r8d, 3
    mov edx, 2
    mov ecx, 1
    call f1
    xor eax, eax
    add rsp, 72 ;
00000048H
ret 0
main ENDP
```

```
a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
```
Here we clearly see how 7 arguments are passed: 4 via registers and the remaining 3 via the stack. The code of the f1() function's prologue saves the arguments in the “scratch space”—a space in the stack intended exactly for this purpose. This is done because the compiler cannot be sure that there will be enough registers to use without these 4, which will otherwise be occupied by the arguments until the function’s execution end. The “scratch space” allocation in the stack is the caller’s duty.

Listing 64.7: Optimizing MSVC 2012 /O$b

```assembly
f1 PROC
$LN3:
    mov    DWORD PTR [rsp+32], r9d
    mov    DWORD PTR [rsp+24], r8d
    mov    DWORD PTR [rsp+16], edx
    mov    DWORD PTR [rsp+8], ecx
    sub    rsp, 72 ; ⤦ 00000048H

    mov    eax, DWORD PTR g$[rsp]
    mov    DWORD PTR [rsp+56], eax
    mov    eax, DWORD PTR f$[rsp]
    mov    DWORD PTR [rsp+48], eax
    mov    eax, DWORD PTR e$[rsp]
    mov    DWORD PTR [rsp+40], eax
    mov    eax, DWORD PTR d$[rsp]
    mov    DWORD PTR [rsp+32], eax
    mov    r9d, DWORD PTR c$[rsp]
    mov    r8d, DWORD PTR b$[rsp]
    mov    edx, DWORD PTR a$[rsp]
    lea    rcx, OFFSET FLAT:$SG2937
    call   printf

    add    rsp, 72 ; ⤦ 00000048H
    ret    0

f1 ENDP
```

$SG2777 DB '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1 PROC
If we compile the example with optimizations, it is to be almost the same, but the "scratch space" will not be used, because it won't be needed.

Also take a look on how MSVC 2012 optimizes the loading of primitive values into registers by using LEA (A.6.2 on page 1304). It's hard to say if it worth doing so, but maybe.
Another example of such thing is: 74.1 on page 1049.

**Windows x64: Passing this (C/C++)**

The *this* pointer is passed in RCX, the first argument of the method is in RDX, etc. For an example see: 51.1.1 on page 760.

**64.5.2 Linux x64**

The way arguments are passed in Linux for x86-64 is almost the same as in Windows, but 6 registers are used instead of 4 (RDI, RSI, RDX, RCX, R8, R9) and there is no “scratch space”, although the callee may save the register values in the stack, if it needs/wants to.

**Listing 64.8: Optimizing GCC 4.7.3**

```assembly
.LC0:    .string "%d %d %d %d %d %d %d\n"
f1:      sub    rsp, 40
         mov    eax, DWORD PTR [rsp+48]
         mov    DWORD PTR [rsp+8], r9d
         mov    r9d, ecx
         mov    DWORD PTR [rsp], r8d
         mov    ecx, esi
         mov    r8d, edx
         mov    esi, OFFSET FLAT:.LC0
         mov    edx, edi
         mov    edi, 1
         mov    DWORD PTR [rsp+16], eax
         xor    eax, eax
         call   __printf_chk
         add    rsp, 40
         ret
main:    sub    rsp, 24
         mov    r9d, 6
         mov    r8d, 5
         mov    DWORD PTR [rsp], 7
         mov    ecx, 4
         mov    edx, 3
         mov    esi, 2
         mov    edi, 1
         call   f1
         add    rsp, 24
```

960
N.B.: here the values are written into the 32-bit parts of the registers (e.g., EAX) but not in the whole 64-bit register (RAX). This is because each write to the low 32-bit part of a register automatically clears the high 32 bits. Supposedly, it was decided in AMD to do so to simplify porting code to x86-64.

### 64.6 Return values of float and double type

In all conventions except in Win64, the values of type float or double are returned via the FPU register ST(0).

In Win64, the values of float and double types are returned in the low 32 or 64 bits of the XMM0 register.

### 64.7 Modifying arguments

Sometimes, C/C++ programmers (not limited to these PLs, though), may ask, what can happen if they modify the arguments? The answer is simple: the arguments are stored in the stack, that is where the modification takes place. The calling functions is not using them after the callee’s exit (author of these lines have never seen any such case in his practice).

```c
#include <stdio.h>

void f(int a, int b)
{
    a=a+b;
    printf ("%d\n", a);
}
```

Listing 64.9: MSVC 2012

```assembly
_f PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    add eax, DWORD PTR _b$[ebp]
    mov WORD PTR _a$[ebp], eax
    mov DWORD PTR _a$[ebp], eax
    ret

    _a$ = 8 ; size ↗
    _b$ = 12 ; size ↗

    _f PROC
```

So yes, one can modify the arguments easily. Of course, if it is not references in C++ (51.3 on page 781), and if you not modify data to which a pointer points to, then the effect will not propagate outside the current function.

Theoretically, after the callee's return, the caller could get the modified argument and use it somehow. Maybe if it is written directly in assembly language. But the C/C++ languages standards don’t offer any way to access them.

### 64.8 Taking a pointer to function argument

... even more than that, it's possible to take a pointer to the function's argument and pass it to another function:

```c
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
    modify_a (&a);
    printf ("%d\n", a);
}
```

It's hard to understand how it works until we can see the code:

Listing 64.10: Optimizing MSVC 2010

```assembly
mov    ecx, DWORD PTR _a$[ebp]
push   ecx
push   OFFSET $SG2938 ; '%d', 0aH
call   _printf
add    esp, 8
pop    ebp
ret    0

_f    ENDP
```

$SG2796 DB '%d', 0aH, 00H

_a$ = 8

_f PROC
    lea    eax, DWORD PTR _a$[esp-4] ; just get the ↘
    ↘ address of value in local stack
    push   eax  ; and pass it to ↘
    ↘ modify_a()
    call   _modify_a
```
The address of the place in the stack where \( a \) was passed is just passed to another function. It modifies the value addressed by the pointer and then `printf()` prints the modified value.

The observant reader might ask, what about calling conventions where the function’s arguments are passed in registers?

That’s a situation where the *Shadow Space* is used. The input value is copied from the register to the *Shadow Space* in the local stack, and then this address is passed to the other function:

```
Listing 64.11: Optimizing MSVC 2012 x64

$SG2994 DB '%d', 0AH, 00H

a$ = 48
f PROC
  mov DWORD PTR [rsp+8], ecx ; save input value in *Shadow Space*
  sub rsp, 40
  lea rcx, QWORD PTR a$[rsp] ; get address of value and pass it to modify_a()
  call modify_a
  mov edx, DWORD PTR a$[rsp] ; reload value from *Shadow Space* and pass it to printf()
  lea rcx, OFFSET FLAT:$SG2994 ; '%d'
  call printf
  add rsp, 40
  ret 0
f ENDP
```

GCC also stores the input value in the local stack:

```
Listing 64.12: Optimizing GCC 4.9.1 x64

.LC0:
  .string "%d\n"
f:
  sub rsp, 24
```
GCC for ARM64 does the same, but this space is called *Register Save Area* here:

**Listing 64.13: Optimizing GCC 4.9.1 ARM64**

```assembly
f:
    stp  x29, x30, [sp, -32]!  ; setup FP
    add  x29, sp, 0          ; calculate address of variable in Register Save Area
    add  x1, x29, 32         ; calculate address of variable
    str  w0, [x1,-4]!        ; store input value there
    mov  x0, x1              ; pass address of variable to the modify_a()
    bl   modify_a
    ldr  w1, [x29,28]        ; load value from the variable and pass it to printf()
    adrp x0, .LC0 ; '%d'
add  x0, x0, :lo12:.LC0
bl   printf                ; call printf()
ldp  x29, x30, [sp], 32
ret

.LC0:
.string "%d\n"
```

By the way, a similar usage of the *Shadow Space* is also considered here: **46.1.2 on page 716.**
Chapter 65

Thread Local Storage

TLS is a data area, specific to each thread. Every thread can store what it needs there. One well-known example is the C standard global variable `errno`. Multiple threads may simultaneously call functions which return an error code in `errno`, so a global variable will not work correctly here for multi-threaded programs, so `errno` must be stored in the TLS.

In the C++11 standard, a new `thread_local` modifier was added, showing that each thread has its own version of the variable, it can be initialized, and it is located in the TLS¹:

```
Listing 65.1: C++11
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
    std::cout << tmp << std::endl;
}
```

Compiled in MinGW GCC 4.8.1, but not in MSVC 2012.

If we talk about PE files, in the resulting executable file, the `tmp` variable is to be allocated in the section devoted to the TLS.

¹ C11 also has thread support, optional though
65.1 Linear congruential generator revisited

The pseudorandom number generator we considered earlier 20 on page 459 has a flaw: it's not thread-safe, because it has an internal state variable which can be read and/or modified in different threads simultaneously.

65.1.1 Win32

Uninitialized TLS data

One solution is to add __declspec( thread ) modifier to the global variable, then it will be allocated in the TLS (line 9):

```c
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    my_srand(0x12345678);
    printf ("%d\n", my_rand());
}
```

Hiew shows us that there is a new PE section in the executable file: .tls.

Listing 65.2: Optimizing MSVC 2013 x86

| _TLS   | SEGMENT |
rand_state is now in the TLS segment, and each thread has its own version of this variable. Here is how it's accessed: load the address of the TIB from FS:2Ch, then add an additional index (if needed), then calculate the address of the TLS segment.
Then it's possible to access the `rand_state` variable through the ECX register, which points to an unique area in each thread.

The FS: selector is familiar to every reverse engineer, it is specially used to always point to TIB, so it would be fast to load the thread-specific data.

The GS: selector is used in Win64 and the address of the TLS is 0x58:

Listing 65.3: Optimizing MSVC 2013 x64

```assembly
_TLS SEGMENT
rand_state DD 01H DUP (?)
_TLS ENDS
_DATA SEGMENT
$SG85451 DB '%d', 0aH, 00H
_DATA ENDS
_TEXT SEGMENT

init$ = 8
my_srand PROC
  mov edx, DWORD PTR _tls_index
  mov rax, QWORD PTR gs:88 ; 58h
  mov r8d, OFFSET FLAT:rand_state
  mov rax, QWORD PTR [rax+rdx*8]
  mov DWORD PTR [r8+rax], ecx
  ret 0
my_srand ENDP

my_rand PROC
  mov rax, QWORD PTR gs:88 ; 58h
  mov ecx, DWORD PTR _tls_index
  mov edx, OFFSET FLAT:rand_state
  mov rcx, QWORD PTR [rax+rcx*8]
  imul eax, DWORD PTR [rcx+rdx], 1664525 ; ⤦ 0019660dH
  add eax, 1013904223 ; 3⤦ c6ef35fH
  mov DWORD PTR [rcx+rdx], eax
  and eax, 32767 ; 00007⤦ fffH
  ret 0
my_rand ENDP
_TEXT ENDS
```

968
Chapter 65. Thread Local Storage

**Initialized TLS data**

Let's say, we want to set some fixed value to `rand_state`, so in case the programmer forgets to, the `rand_state` variable would be initialized to some constant anyway (line 9):

```c
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state=1234;

void my_srand (uint32_t init)
{
    rand_state=init;
}

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    printf("%d\n", my_rand());
};
```

The code is no differ from what we already saw, but in IDA we see:

```
.tls:00404000 ; Segment type: Pure data
.tls:00404000 ; Segment permissions: Read/Write
.tls:00404000 _tls segment para public 'DATA' use32
.tls:00404000 assume cs:_tls
.tls:00404000 ;org 404000h
.tls:00404000 TlsStart db 0 ; DATA
   XREF: .rdata:TlsDirectory
.tls:00404000 TlsEnd db 0 ; DATA
   XREF: .rdata:TlsEnd_ptr
```
1234 is there and every time a new thread starts, a new TLS is allocated for it, and all this data, including 1234, will be copied there.

This is a typical scenario:

- Thread A is started. A TLS is created for it, 1234 is copied to rand_state.
- The my_rand() function is called several times in thread A. rand_state is different from 1234.
- Thread B is started. A TLS is created for it, 1234 is copied to rand_state, while thread A has a different value in the same variable.

**TLS callbacks**

But what if the variables in the TLS have to be filled with some data that must be prepared in some unusual way? Let's say, we've got the following task: the programmer can forget to call the my_srand() function to initialize the PRNG, but the generator has to be initialized at start with something truly random, instead of 1234. This is a case in which TLS callbacks can be used.

The following code is not very portable due to the hack, but nevertheless, you get the idea. What we do here is define a function (tls_callback()) which is to be called before the process and/or thread start. The function initializes the PRNG with the value returned by GetTickCount() function.

```c
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec(thread) uint32_t rand_state;

void my_srand(uint32_t init)
{
    rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
    my_srand(GetTickCount());
}
```
#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    // rand_state is already initialized at the moment (✓
    // using GetTickCount())
    printf("%d\n", my_rand());
};

Let's see it in IDA:

Listing 65.4: Optimizing MSVC 2013

```
.text:00401020 TlsCallback_0 proc near ; DATA ✓
   XREF: .rdata:TlsCallbacks
.text:00401020    call    ds:GetTickCount
.text:00401026    push    eax
.text:00401027    call    my_srand
.text:0040102C    pop     ecx
.text:0040102D    retn    0Ch
.text:0040102D TlsCallback_0 endp

...

.rdata:004020C0 TlsCallbacks dd offset TlsCallback_0 ; DATA ✓
   XREF: .rdata:TlsCallbacks_ptr
...

.rdata:00402118 TlsDirectory dd offset TlsStart
.rdata:0040211C TlsEnd_ptr dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h
```

TLS callback functions are sometimes used in unpacking routines to obscure their processing. Some people may be confused and be in the dark that some code
executed right before the OEP\(^2\).

## 65.1.2 Linux

Here is how a thread-local global variable is declared in GCC:

```c
__thread uint32_t rand_state=1234;
```

This is not the standard C/C++ modifier, but a rather GCC-specific one\(^3\).

The GS: selector is also used to access the TLS, but in a somewhat different way:

<table>
<thead>
<tr>
<th>.text:08048460</th>
<th>my_srand</th>
<th>proc near</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:08048460</td>
<td>arg_0</td>
<td>= dword ptr 4</td>
</tr>
<tr>
<td>.text:08048460</td>
<td></td>
<td>mov eax, [esp+arg_0]</td>
</tr>
<tr>
<td>.text:08048464</td>
<td></td>
<td>mov gs:0FFFFFFFCh, eax</td>
</tr>
<tr>
<td>.text:0804846A</td>
<td></td>
<td>retn</td>
</tr>
<tr>
<td>.text:0804846A</td>
<td>my_srand</td>
<td>endp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>.text:08048470</th>
<th>my_rand</th>
<th>proc near</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:08048470</td>
<td></td>
<td>imul eax, gs:0FFFFFFFCh, 19660Dh</td>
</tr>
<tr>
<td>.text:0804847B</td>
<td></td>
<td>add eax, 3C6EF35Fh</td>
</tr>
<tr>
<td>.text:08048480</td>
<td></td>
<td>mov gs:0FFFFFFFCh, eax</td>
</tr>
<tr>
<td>.text:08048486</td>
<td></td>
<td>and eax, 7FFFh</td>
</tr>
<tr>
<td>.text:0804848B</td>
<td></td>
<td>retn</td>
</tr>
<tr>
<td>.text:0804848B</td>
<td>my_rand</td>
<td>endp</td>
</tr>
</tbody>
</table>

More about it: [Dre13].

---

\(^2\)Original Entry Point

\(^3\)[http://go.yurichev.com/17062](http://go.yurichev.com/17062)
Chapter 66

System calls (syscall-s)

As we know, all running processes inside an OS are divided into two categories: those having full access to the hardware (“kernel space”) and those that do not (“user space”).

The OS kernel and usually the drivers are in the first category.

All applications are usually in the second category.

For example, Linux kernel is in kernel space, but Glibc in user space.

This separation is crucial for the safety of the OS: it is very important not to give any process the possibility to screw up something in other processes or even in the OS kernel. On the other hand, a failing driver or error inside the OS’s kernel usually leads to a kernel panic or BSOD\(^1\).

The protection in the x86 processors allows to separate everything into 4 levels of protection (rings), but both in Linux and in Windows only two are used: ring0 (“kernel space”) and ring3 (“user space”).

System calls (syscall-s) are a point where these two areas are connected. It can be said that this is the main API provided to applications.

As in Windows NT, the syscalls table resides in the SSDT\(^2\).

The usage of syscalls is very popular among shellcode and computer viruses authors, because it is hard to determine the addresses of needed functions in the system libraries, but it is easier to use syscalls. However, much more code has to be written due to the lower level of abstraction of the API. It is also worth noting that the syscall numbers may be different in various OS versions.

---

\(^1\) Blue Screen of Death  
\(^2\) System Service Dispatch Table
66.1 Linux

In Linux, a syscall is usually called via int 0x80. The call's number is passed in the EAX register, and any other parameters—in the other registers.

Listing 66.1: A simple example of the usage of two syscalls

```assembly
section .text
global _start

_start:
    mov edx, len ; buffer len
    mov ecx, msg ; buffer
    mov ebx, 1 ; file descriptor. 1 is for stdout
    mov eax, 4 ; syscall number. 4 is for sys_write
    int 0x80

    mov eax, 1 ; syscall number. 1 is for sys_exit
    int 0x80

section .data

msg db 'Hello, world!', 0xa
len equ $ - msg
```

Compilation:

```
nasm -f elf32 1.s
ld 1.o
```

The full list of syscalls in Linux: [http://go.yurichev.com/17319](http://go.yurichev.com/17319).

For system calls interception and tracing in Linux, strace(71 on page 1040) can be used.

66.2 Windows

Here they are called via int 0x2e or using the special x86 instruction SYSENTER.


Further reading:

Chapter 67

Linux

67.1 Position-independent code

While analyzing Linux shared (.so) libraries, one may frequently spot this code pattern:

Listing 67.1: libc-2.17.so x86

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near ; CODE ⬇
   XREF: sub_17350+3
.text:0012D5E3 ; ⬇
   XREF: sub_173CC+4 ...
.text:0012D5E3    mov    ebx, [esp+0]
.text:0012D5E6    retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp
...

.text:000576C0 sub_576C0 proc near ; CODE ⬇
   XREF: tmpfile+73
...

.text:000576C0 push ebp
.text:000576C1 mov    ecx, large gs:0
.text:000576C8 push    edi
.text:000576C9 push    esi
.text:000576CA push    ebx
.text:000576CB call    __x86_get_pc_thunk_bx
.text:000576D0 add    ebx, 157930h
.text:000576D6 sub    esp, 9Ch
```
All pointers to strings are corrected by some constants and the value in EBX, which is calculated at the beginning of each function. This is the so-called PIC, it is intended to be executable if placed at any random point of memory, that is why it cannot contain any absolute memory addresses.

PIC was crucial in early computer systems and is crucial now in embedded systems without virtual memory support (where all processes are placed in a single continuous memory block). It is also still used in *NIX systems for shared libraries, since they are shared across many processes while loaded in memory only once. But all these processes can map the same shared library at different addresses, so that is why a shared library has to work correctly without using any absolute addresses.

Let's do a simple experiment:

```c
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
}
```

Let's compile it in GCC 4.7.3 and see the resulting .so file in IDA:

gcc -fPIC -shared -O3 -o 1.so 1.c

```
Listing 67.2: GCC 4.7.3

.text:00000440 public __x86_get_pc_thunk_bx
```

976
That's it: the pointers to «returning %d\n» and global_variable are to be corrected at each function execution.

The __x86_get_pc_thunk_bx() function returns in EBX the address of the point after a call to itself (0x57C here). That's a simple way to get the value of the program counter (EIP) at some point. The 0x1A84 constant is related to the difference between this function's start and the so-called Global Offset Table.
Procedure Linkage Table (GOT PLT), the section right after the Global Offset Table (GOT), where the pointer to global_variable is. IDA shows these offsets in their processed form to make them easier to understand, but in fact the code is:

```
=text:00000577     call __x86_get_pc_thunk_bx
.text:0000057C     add ebx, 1A84h
.text:00000582     mov [esp+1Ch+var_4], esi
.text:00000586     mov eax, [ebx-0Ch]
.text:0000058C     mov esi, [eax]
.text:0000058E     lea eax, [ebx-1A30h]
```

Here EBX points to the GOT PLT section and to calculate a pointer to global_variable (which is stored in the GOT), 0xC must be subtracted. To calculate pointer to the «returning %d\n» string, 0x1A30 must be subtracted.

By the way, that is the reason why the AMD64 instruction set supports RIP\(^1\)-relative addressing—to simplify PIC-code.

Let’s compile the same C code using the same GCC version, but for x64.

IDA would simplify the resulting code but would suppress the RIP-relative addressing details, so we are going to use objdump instead of IDA to see the everything:

```
0000000000000720 <f1>:
  720: 48 8b 05 b9 08 20 00 mov rax, QWORD PTR [rip+0⤦/rcurvearrowse \x2008b9] # 200fe0 <_DYNAMIC+0x1d0>
  727: 53 push rbx
  728: 89 fb mov ebx, edi
  72a: 48 8d 35 20 00 00 00 lea rsi, [rip+0x20] # ⤵/rcurvearrowse 751 <_fini+0x9>
  731: bf 01 00 00 00 mov edi, 0x1
  736: 03 18 add ebx, DWORD PTR [rax]
  738: 31 c0 xor eax, eax
  73a: 89 da mov edx, ebx
  73c: e8 df fe ff ff call 620 <__printf_chk@plt>
  741: 89 d8 mov eax, ebx
  743: 5b pop rbx
  744: c3 ret
```

0x2008b9 is the difference between the address of the instruction at 0x720 and global_variable, and 0x20 is the difference between the address of the instruction at 0x72A and the «returning %d\n» string.

As you might see, the need to recalculate addresses frequently makes execution slower (it is better in x64, though). So it is probably better to link statically if you care about performance [Fog13a].

\(^1\)program counter in AMD64
67.1.1 Windows

The PIC mechanism is not used in Windows DLLs. If the Windows loader needs to load DLL on another base address, it “patches” the DLL in memory (at the FIXUP places) in order to correct all addresses. This implies that several Windows processes cannot share an once loaded DLL at different addresses in different process’ memory blocks—since each instance that’s loaded in memory is fixed to work only at these addresses.

67.2 LD_PRELOAD hack in Linux

This allows us to load our own dynamic libraries before others, even before system ones, like libc.so.6.

This, in turn, allows us to “substitute” our written functions before the original ones in the system libraries. For example, it is easy to intercept all calls to time(), read(), write(), etc.

Let’s see if we can fool the uptime utility. As we know, it tells how long the computer has been working. With the help of strace( 71 on page 1040), it is possible to see that the utility takes this information the /proc/uptime file:

```
$ strace uptime
... 
open("/proc/uptime", O_RDONLY)       = 3
lseek(3, 0, SEEK_SET)                = 0
read(3, "416166.86 414629.38\n", 2047) = 20
... 
```

It is not a real file on disk, it is a virtual one and its contents are generated on fly in the Linux kernel. There are just two numbers:

```
$ cat /proc/uptime
416690.91 415152.03
```

What we can learn from Wikipedia ²:

```
The first number is the total number of seconds the system has been up. The second number is how much of that time the machine has spent idle, in seconds.
```

²wikipedia
Let’s try to write our own dynamic library with the open(), read(), close() functions working as we need.

At first, our open() will compare the name of the file to be opened with what we need and if it is so, it will write down the descriptor of the file opened. Second, read(), if called for this file descriptor, will substitute the output, and in the rest of the cases will call the original read() from libc.so.6. And also close(), will note if the file we are currently following is to be closed.

We are going to use the dlopen() and dlsym() functions to determine the original function addresses in libc.so.6.

We need them because we must pass control to the “real” functions.

On the other hand, if we intercepted strcmp() and monitored each string comparisons in the program, then we would have to implement a strcmp(), and not use the original function.

---

```c
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool inited = false;

_Noreturn void die (const char * fmt, ...) {
    va_list va;
    va_start (va, fmt);
    vprintf (fmt, va);
    exit(0);
}

static void find_original_functions () {
    if (inited)
        return;
```

---

3 For example, here is how simple strcmp() interception works in this article written by Yong Huang.
libc_handle = dlopen("libc.so.6", RTLD_LAZY);
if (libc_handle==NULL)
    die("can't open libc.so.6\n");

open_ptr = dlsym(libc_handle, "open");
if (open_ptr==NULL)
    die("can't find open()\n");

close_ptr = dlsym(libc_handle, "close");
if (close_ptr==NULL)
    die("can't find close()\n");

read_ptr = dlsym(libc_handle, "read");
if (read_ptr==NULL)
    die("can't find read()\n");
inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
    find_original_functions();

    int fd=(*open_ptr)(pathname, flags);
    if (strcmp(pathname, "/proc/uptime")==0)
        opened_fd=fd; // that's our file! record its file descriptor
    else
        opened_fd=0;
    return fd;
};

int close(int fd)
{
    find_original_functions();

    if (fd==opened_fd)
        opened_fd=0; // the file is not opened anymore
    return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
    find_original_functions();

if (opened_fd!=0 && fd==opened_fd)
{
    // that's our file!
    return snprintf (buf, count, "%d %d", 0
    \x7fffffff, 0x7fffffff)+1;
};
// not our file, go to real read() function
return (*read_ptr)(fd, buf, count);

(Source code at GitHub)

Let's compile it as common dynamic library:

gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl

Let's run uptime while loading our library before the others:

LD_PRELOAD=`pwd`/fool_uptime.so uptime

And we see:

01:23:02 up 24855 days, 3:14, 3 users, load average: 0.00, 0.01, 0.05

If the LD_PRELOAD environment variable always points to the filename and path of our library, it is to be loaded for all starting programs.

More examples:

- Very simple interception of the strcmp() (Yong Huang) http://go.yurichev.com/17143
- Kevin Pulo—Fun with LD_PRELOAD. A lot of examples and ideas. yurichev.com
Chapter 68

Windows NT

68.1 CRT (win32)

Does the program execution start right at the main() function? No, it does not. If we would open any executable file in IDA or HIEW, we can see OEP pointing to some another code block. This code is doing some maintenance and preparations before passing control flow to our code. It is called startup-code or CRT code (C RunTime).

The main() function takes an array of the arguments passed on the command line, and also one with environment variables. But in fact a generic string is passed to the program, the CRT code finds the spaces in it and cuts it in parts. The CRT code also prepares the environment variables array envp. As for GUI\(^1\) win32 applications, WinMain is used instead of main(), having its own arguments:

```c
int CALLBACK WinMain(
    _In_  HINSTANCE hInstance,
    _In_  HINSTANCE hPrevInstance,
    _In_  LPSTR lpCmdLine,
    _In_  int nCmdShow
);
```

The CRT code prepares them as well.

Also, the number returned by the main() function is the exit code. It may be passed in CRT to the ExitProcess() function, which takes the exit code as an argument.

---

\(^1\)Graphical user interface
Usually, each compiler has its own CRT code.

Here is a typical CRT code for MSVC 2008.

```assembly
___tmainCRTStartup proc near

var_24 = dword ptr -24h
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
ms_exc = CPPEH_RECORD ptr -18h

push 14h
push offset stru_4092D0
call __SEH_prolog4
mov eax, 5A4Dh
cmp ds:400000h, ax
jnz short loc_401096
mov eax, ds:40003Ch
cmp dword ptr [eax+400000h], 4550h
jnz short loc_401096
mov ecx, 10Bh
cmp [eax+400018h], cx
jnz short loc_401096
cmp dword ptr [eax+400074h], 0Eh
jbe short loc_401096
xor ecx, ecx
cmp [eax+4000E8h], ecx
setnz cl
mov [ebp+var_1C], ecx
jmp short loc_40109A

loc_40109A: ; CODE XREF: ___tmainCRTStartup+50
push 1
call __heap_init
pop ecx
test eax, eax
jnz short loc_4010AE
push 1Ch
call _fast_error_exit
pop ecx

loc_4010AE: ; CODE XREF: ___tmainCRTStartup+60
```

984
call __mtinit

test eax, eax
jnz short loc_4010BF

push 10h

call _fast_error_exit

pop ecx

loc_4010BF: ; CODE XREF: ___tmainCRTStartup+71
call sub_401F2B

and [ebp+ms_exc.disabled], 0
call __ioinit
test eax, eax
jge short loc_4010D9

push 1Bh
call __amsg_exit

pop ecx

loc_4010D9: ; CODE XREF: ___tmainCRTStartup+8B
call ds:GetCommandLineA
mov dword_40B7F8, eax
call ___crtGetEnvironmentStringsA
mov dword_40AC60, eax
call __setargv
test eax, eax
jge short loc_4010FF

push 8
call __amsg_exit

pop ecx

loc_4010FF: ; CODE XREF: ___tmainCRTStartup+B1
call __setenvp
test eax, eax
jge short loc_401110

push 9
call __amsg_exit

pop ecx

loc_401110: ; CODE XREF: ___tmainCRTStartup+C2

push 1
call __cinit

pop ecx
test eax, eax
jz short loc_401123
push eax
call __amsg_exit

pop ecx
loc_401123: ; CODE XREF: ___tmainCRTStartup+D6
  mov    eax, envp
  mov    dword_40AC80, eax
  push   eax          ; envp
  push   argv         ; argv
  push   argc         ; argc
  call   __main
  add    esp, 0Ch
  mov    [ebp+var_20], eax
  cmp    [ebp+var_1C], 0
  jnz    short $LN28
  push   eax           ; uExitCode
  call   $LN32

$LN28: ; CODE XREF: ___tmainCRTStartup+105
  call   __cexit
  jmp    short loc_401186

$LN27: ; DATA XREF: .rdata:stru_4092D0
  mov    eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0  ↳
     for function 401044
  mov    ecx, [eax]
  mov    ecx, [ecx]
  mov    [ebp+var_24], ecx
  push   eax
  push   ecx
  call   __XcptFilter
  pop    ecx
  pop    ecx

$LN24:
  ret

$LN14: ; DATA XREF: .rdata:stru_4092D0
  mov    esp, [ebp+ms_exc.old esp] ; Exception handler 0  ↳
     for function 401044
  mov    eax, [ebp+var_24]
  mov    [ebp+var_20], eax
  cmp    [ebp+var_1C], 0
  jnz    short $LN29
  push   eax           ; int
  call   __exit

$LN29: ; CODE XREF: ___tmainCRTStartup+135
Here we can see calls to GetCommandLineA() (line 62), then to setargv() (line 66) and setenvp() (line 74), which apparently fill the global variables argc, argv, envp.

Finally, main() is called with these arguments (line 97).

There are also calls to functions with self-describing names like heap_init() (line 35), ioinit() (line 54).

The heap is indeed initialized in the CRT. If you try to use malloc() in a program without CRT, it will exit abnormally with the following error:

```
runtime error R6030
- CRT not initialized
```

Global object initializations in C++ is also occur in the CRT before the execution of main(). 51.4.1 on page 790.

The value that main() returns is passed to cexit(), or in $LN32, which in turn calls doexit().

Is it possible to get rid of the CRT? Yes, if you know what you are doing.

The MSVC's linker has the /ENTRY option for setting an entry point.

```
#include <windows.h>

int main()
{
    MessageBox (NULL, "hello, world", "caption", MB_OK);
}
```

Let's compile it in MSVC 2008.

```
c1 no_crt.c user32.lib /link /entry:main
```

We are getting a runnable .exe with size 2560 bytes, that has a PE header in it, instructions calling MessageBox, two strings in the data segment, the MessageBox function imported from user32.dll and nothing else.
This works, but you cannot write \texttt{WinMain} with its 4 arguments instead of \texttt{main()}. To be precise, you can, but the arguments are not prepared at the moment of execution.

By the way, it is possible to make the .exe even shorter by aligning the \texttt{PE} sections at less than the default 4096 bytes.

\begin{verbatim}
cl no_crt.c user32.lib /link /entry:main /align:16
\end{verbatim}

Linker says:

\begin{verbatim}
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image ↵ may not run
\end{verbatim}

We get an .exe that's 720 bytes. It can be executed in Windows 7 x86, but not in x64 (an error message will be shown when you try to execute it). With even more efforts, it is possible to make the executable even shorter, but as you can see, compatibility problems arise quickly.

### 68.2 Win32 PE

\texttt{PE} is an executable file format used in Windows.

The difference between .exe, .dll and .sys is that .exe and .sys usually do not have exports, only imports.

A \texttt{DLL}, just like any other PE-file, has an entry point (\texttt{OEP}) (the function \texttt{DllMain()} is located there) but this function usually does nothing.

.sys is usually a device driver.

As of drivers, Windows requires the checksum to be present in the PE file and for it to be correct \footnote{For example, Hiew (73 on page 1042) can calculate it}. Starting at Windows Vista, a driver's files must also be signed with a digital signature. It will fail to load otherwise.

Every PE file begins with tiny DOS program that prints a message like “This program cannot be run in DOS mode.”— if you run this program in DOS or Windows 3.1 (\texttt{OS-es} which are not aware of the PE format), this message will be printed.

\footnote{Portable Executable: 68.2 \footnote{Dynamic-link library} \footnote{For example, Hiew (73 on page 1042) can calculate it}}
### 68.2.1 Terminology

- **Module**—a separate file, .exe or .dll.
- **Process**—a program loaded into memory and currently running. Commonly consists of one .exe file and bunch of .dll files.
- **Process memory**—the memory a process works with. Each process has its own. There usually are loaded modules, memory of the stack, heap(s), etc.
- **VA**\(^5\) — an address which is to be used in program while runtime.
- **Base address (of module)**—the address within the process memory at which the module is to be loaded. OS loader may change it, if the base address is already occupied by another module just loaded before.
- **RVA**\(^6\) — the VA-address minus the base address. Many addresses in PE-file tables use RVA-addresses.
- **IAT**\(^7\) — an array of addresses of imported symbols. Sometimes, the IMAGE_DIRECTORY_ENTRY_IAT data directory points at the IAT. It is worth noting that IDA (as of 6.1) may allocate a pseudo-section named .idata for IAT, even if the IAT is a part of another section!
- **INT**\(^9\) — an array of names of symbols to be imported\(^10\).

### 68.2.2 Base address

The problem is that several module authors can prepare DLL files for others to use and it is not possible to reach an agreement which addresses is to be assigned to whose modules.

So that is why if two necessary DLLs for a process have the same base address, one of them will be loaded at this base address, and the other—at some other free space in process memory, and each virtual addresses in the second DLL will be corrected.

Often, **MSVC** the linker generates the .exe files with a base address of 0x400000\(^11\), and with the code section starting at 0x401000. This mean that the RVA of the start of the code section is 0x1000. DLLs are often generated by MSVC’s linker

---

\(^5\)Virtual Address  
\(^6\)Relative Virtual Address  
\(^7\)Import Address Table  
\(^8\)[Pie02]  
\(^9\)Import Name Table  
\(^10\)[Pie02]  
\(^11\)The origin of this address choice is described here: MSDN
with a base address of 0x10000000. There is also another reason to load modules at various base addresses, in this case random ones. It is ASLR.

A shellcode trying to get executed on a compromised system must call system functions, hence, know their addresses.

In older OS (in Windows NT line: before Windows Vista), system DLL (like kernel32.dll, user32.dll) were always loaded at known addresses, and if we also recall that their versions rarely changed, the addresses of functions were fixed and shellcode could call them directly.

In order to avoid this, the ASLR method loads your program and all modules it needs at random base addresses, different every time. ASLR support is denoted in a PE file by setting the flag IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE [RA09].

### 68.2.3 Subsystem

There is also a subsystem field, usually it is:

- native (.sys-driver),
- console (console application) or
- GUI (non-console).

### 68.2.4 OS version

A PE file also specifies the minimal Windows version it needs in order to be loadable. The table of version numbers stored in the PE file and corresponding Windows codenames is here.

For example, MSVC 2005 compiles .exe files for running on Windows NT4 (version 4.00), but MSVC 2008 does not (the generated files have a version of 5.00, at least Windows 2000 is needed to run them).

---

12 This can be changed by the /BASE linker option
13 Address Space Layout Randomization
14 Wikipedia
15 Meaning, the module use Native API instead of Win32
16 Wikipedia
MSVC 2012 generates .exe files of version 6.00 by default, targeting at least Windows Vista. However, by changing the compiler’s options \(^\text{17}\), it is possible to force it to compile for Windows XP.

### 68.2.5 Sections

Division in sections, as it seems, is present in all executable file formats.

It is devised in order to separate code from data, and data—from constant data.

- Either the \texttt{IMAGE\_SCN\_CNT\_CODE} or \texttt{IMAGE\_SCN\_MEM\_EXECUTE} flags will be set on the code section—this is executable code.
- On data section—\texttt{IMAGE\_SCN\_CNT\_INITIALIZED\_DATA}, \texttt{IMAGE\_SCN\_MEM\_READ} and \texttt{IMAGE\_SCN\_MEM\_WRITE} flags.
- On an empty section with uninitialized data—\texttt{IMAGE\_SCN\_CNT\_UNINITIALIZED\_DATA}, \texttt{IMAGE\_SCN\_MEM\_READ} and \texttt{IMAGE\_SCN\_MEM\_WRITE}.
- On a constant data section (one that’s protected from writing), the flags \texttt{IMAGE\_SCN\_CNT\_INITIALIZED\_DATA} and \texttt{IMAGE\_SCN\_MEM\_READ} can be set, but not \texttt{IMAGE\_SCN\_MEM\_WRITE}. A process going to crash if it tries to write to this section.

Each section in PE-file may have a name, however, it is not very important. Often (but not always) the code section is named \texttt{.text}, the data section—\texttt{.data}, the constant data section — \texttt{.rdata} (readable data). Other popular section names are:

- \texttt{.idata}—imports section. IDA may create a pseudo-section named like this: \texttt{68.2.1 on page 989}.
- \texttt{.edata}—exports section (rare)
- \texttt{.pdata}—section containing all information about exceptions in Windows NT for MIPS, IA64 and x64: \texttt{68.3.3 on page 1027}
- \texttt{.reloc}—relocs section
- \texttt{.bss}—uninitialized data (BSS)
- \texttt{.tls}—thread local storage (TLS)
- \texttt{.rsrc}—resources
- \texttt{.CRT}—may present in binary files compiled by ancient MSVC versions

PE file packers/encryptors often garble section names or replace the names with their own.

MSVC allows you to declare data in arbitrarily named section \(^\text{18}\).

\(^{17}\text{MSDN}\)

\(^{18}\text{MSDN}\)
Some compilers and linkers can add a section with debugging symbols and other debugging information (MinGW for instance). However it is not so in modern versions of MSVC (separate PDB files are used there for this purpose).

That is how a PE section is described in the file:

```c
typedef struct __IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

A word about terminology: `PointerToRawData` it called “Offset” in Hiew and `VirtualAddress` is called “RVA” there.

### 68.2.6 Relocations (relocs)

AKA FIXUP-s (at least in Hiew).

They are also present in almost all executable file formats\(^\text{20}\). Exceptions are shared dynamic libraries compiled with PIC, or any other PIC-code.

What are they for? Obviously, modules can be loaded on various base addresses, but how to deal with global variables, for example? They must be accessed by address. One solution is position-independent code (67.1 on page 975). But it is not always convenient.

That is why a relocations table is present. There the addresses of points that need to be corrected are enumerated, in case of loading at a different base address.

For example, there is a global variable at address 0x410000 and this is how it is accessed:

---

\(^{19}\) MSDN

\(^{20}\) Even in .exe files for MS-DOS
A1 00 00 41 00    mov    eax,[000410000]

The base address of the module is 0x400000, the RVA of the global variable is 0x10000.

If the module is loaded at base address 0x500000, the real address of the global variable must be 0x510000.

As we can see, the address of variable is encoded in the instruction MOV, after the byte 0xA1.

That is why the address of the 4 bytes after 0xA1, is written in the relocs table.

If the module is loaded at a different base address, the OS loader enumerates all addresses in the table, finds each 32-bit word the address points to, subtracts the original base address from it (we get the RVA here), and adds the new base address to it.

If a module is loaded at its original base address, nothing happens.

All global variables can be treated like that.

Relocs may have various types, however, in Windows for x86 processors, the type is usually `IMAGE_REL_BASED_HIGHLOW`.

By the way, relocs are darkened in Hiew, for example: fig.7.12.

OllyDbg underlines the places in memory to which relocs are to be applied, for example: fig.13.11.

### 68.2.7 Exports and imports

As we all know, any executable program must use the OS’s services and other DLL-libraries somehow.

It can be said that functions from one module (usually DLL) must be connected somehow to the points of their calls in other modules (.exe-file or another DLL).

For this, each DLL has an “exports” table, which consists of functions plus their addresses in a module.

And every .exe file or DLL has “imports”, a table of functions it needs for execution including list of DLL filenames.

After loading the main .exe-file, the OS loader processes imports table: it loads the additional DLL-files, finds function names among the DLL exports and writes their addresses down in the IAT of the main .exe-module.
As we can see, during loading the loader must compare a lot of function names, but string comparison is not a very fast procedure, so there is a support for “ordinals” or “hints”, which are function numbers stored in the table, instead of their names. That is how they can be located faster when loading a DLL. Ordinals are always present in the “export” table.

For example, a program using the MFC\(^{21}\) library usually loads mfc*.dll by ordinals, and in such programs there are no MFC function names in INT.

When loading such programs in IDA, it will ask for a path to the mfc*.dll files in order to determine the function names. If you don’t tell IDA the path to these DLLs, there will be \textit{mfc80_123} instead of function names.

**Imports section**

Often a separate section is allocated for the imports table and everything related to it (with name like \texttt{.idata\_}), however, this is not a strict rule.

Imports are also a confusing subject because of the terminological mess. Let’s try to collect all information in one place.

\(^{21}\)Microsoft Foundation Classes
Figure 68.1: A scheme that unites all PE-file structures related to imports

The main structure is the array `IMAGE_IMPORT_DESCRIPTOR`. Each element for each DLL being imported.

Each element holds the RVA address of the text string (DLL name) (`Name`).

`OriginalFirstThunk` is the RVA address of the INT table. This is an array of RVA addresses, each of which points to a text string with a function name. Each string is prefixed by a 16-bit integer (“hint”)—“ordinal” of function.

While loading, if it is possible to find a function by ordinal, then the strings comparison will not occur. The array is terminated by zero. There is also a pointer to the IAT table named `FirstThunk`, it is just the RVA address of the place where the loader writes the addresses of the resolved functions.

The points where the loader writes addresses are marked by IDA like this: `__imp_CreateFileA`, etc.
There are at least two ways to use the addresses written by the loader.

- The code will have instructions like `call __imp_CreateFileA`, and since the field with the address of the imported function is a global variable in some sense, the address of the `call` instruction (plus 1 or 2) is to be added to the relocations table, for the case when the module is loaded at a different base address.

  But, obviously, this may enlarge relocations table significantly. Because there are might be a lot of calls to imported functions in the module. Furthermore, large relocations table slows down the process of loading modules.

- For each imported function, there is only one jump allocated, using the `JMP` instruction plus a reloc to it. Such points are also called “thunks”. All calls to the imported functions are just `CALL` instructions to the corresponding “thunk”. In this case, additional relocations are not necessary because these `CALL`s have relative addresses and do not need to be corrected.

These two methods can be combined. Possible, the linker creates individual “thunk”s if there are too many calls to the function, but not done by default.

By the way, the array of function addresses to which `FirstThunk` is pointing is not necessary to be located in the IAT section. For example, author of these lines once wrote the `PE_add_import` utility for adding imports to an existing .exe-file. Some time earlier, in the previous versions of the utility, at the place of the function you want to substitute with a call to another DLL, my utility wrote the following code:

```assembly
MOV EAX, [yourdll.dll!function]
JMP EAX
```

`FirstThunk` points to the first instruction. In other words, when loading `yourdll.dll`, the loader writes the address of the `function` function right in the code.

It also worth noting that a code section is usually write-protected, so my utility adds the `IMAGE_SCN_MEM_WRITE` flag for code section. Otherwise, the program to crash while loading with error code 5 (access denied).

One might ask: what if I supply a program with a set of DLL files which is not supposed to change (including addresses of all DLL functions), is it possible to speed up the loading process?

Yes, it is possible to write the addresses of the functions to be imported into the `FirstThunk` arrays in advance. The `Timestamp` field is present in the `IMAGE_IMPORT_DESCRIPTOR` structure. If a value is present there, then the loader compares this value with the date-time of the DLL file. If the values are equal, then the loader does not do anything, and the loading of the process can be faster. This

---

22 [yurichev.com](http://yurichev.com)
is called “old-style binding”. The BIND.EXE utility in Windows SDK is for for this. For speeding up the loading of your program, Matt Pietrek in [Pie02], suggests to do the binding shortly after your program installation on the computer of the end user.

PE-files packers/encryptors may also compress/encrypt imports table. In this case, the Windows loader, of course, will not load all necessary DLLs. Therefore, the packer/encryptor does this on its own, with the help of LoadLibrary() and the GetProcAddress() functions. That is why these two functions are often present in IAT in packed files.

In the standard DLLs from the Windows installation, IAT often is located right in the beginning of the PE file. Supposedly, it is done for optimization. While loading, the .exe file is not loaded into memory as a whole (recall huge install programs which are started suspiciously fast), it is “mapped”, and loaded into memory in parts as they are accessed. Probably, Microsoft developers decided it will be faster.

### 68.2.8 Resources

Resources in a PE file are just a set of icons, pictures, text strings, dialog descriptions. Perhaps, they were separated from the main code, so all these things could be multilingual, and it would be simpler to pick text or picture for the language that is currently set in the OS.

As a side effect, they can be edited easily and saved back to the executable file, even if one does not have special knowledge, by using the ResHack editor, for example (68.2.11 on the following page).

### 68.2.9 .NET

.NET programs are not compiled into machine code but into a special bytecode. Strictly speaking, there is bytecode instead of the usual x86 code in the .exe file; however, the entry point (OEP) points to this tiny fragment of x86 code:

```
jmp mscoree.dll!_CorExeMain
```

The .NET loader is located in mscoree.dll, which processes the PE file. It was so in all pre-Windows XP OSes. Starting from XP, the OS loader is able to detect the .NET file and run it without executing that JMP instruction.
68.2.10 TLS

This section holds initialized data for the TLS (65 on page 965) (if needed). When a new thread start, its TLS data is initialized using the data from this section.

Aside from that, the PE file specification also provides initialization of the TLS section, the so-called TLS callbacks. If they are present, they are to be called before the control is passed to the main entry point (OEP). This is used widely in the PE file packers/encryptors.

68.2.11 Tools

- objdump (present in cygwin) for dumping all PE-file structures.
- Hiew( 73 on page 1042) as editor.
- pefile—Python-library for PE-file processing 25.
- PE_add_import27— simple tool for adding symbol(s) to PE executable import table.
- PE_patcher28— simple tool for patching PE executables.
- PE_search_str.refs29— simple tool for searching for a function in PE executables which use some text string.

68.2.12 Further reading

- Daniel Pistelli—The .NET File Format 30

68.3 Windows SEH

68.3.1 Let’s forget about MSVC

In Windows, the SEH is intended for exceptions handling, nevertheless, it is language-agnostic, not related to C++ or OOP in any way. Here we are going to take a look

25http://go.yurichev.com/17052
26http://go.yurichev.com/17052
27http://go.yurichev.com/17049
28yurichev.com
29yurichev.com
30http://go.yurichev.com/17056
at SEH in its isolated (from C++ and MSVC extensions) form.

Each running process has a chain of SEH handlers, TIB has the address of the last handler. When an exception occurs (division by zero, incorrect address access, user exception triggered by calling the RaiseException() function), the OS finds the last handler in the TIB and calls it, passing all information about the CPU state (register values, etc) at the moment of the exception. The exception handler considering the exception, was it made for it? If so, it handles the exception. If not, it signals to the OS that it cannot handle it and the OS calls the next handler in the chain, until a handler which is able to handle the exception is be found.

At the very end of the chain there a standard handler that shows the well-known dialog box, informing the user about a process crash, some technical information about the CPU state at the time of the crash, and offering to collect all information and send it to developers in Microsoft.

![Figure 68.2: Windows XP](image)

**Figure 68.2: Windows XP**
The following information about your process will be reported:

**Exception Information**
Code: 0xc0000005   Flags: 0x00000000
Record: 0x0000000000000000   Address: 0x0000000000040100e

**System Information**
Windows NT 5.1 Build: 2600
CPU Vendor Code: 756E6547 - 49656E69 - 6C65746E
CPU Version: 000206A7   CPU Feature Code: 0FABFBFF
CPU AMD Feature Code: 00D1E824

**Module**
crash.exe
Image Base: 0x00400000   Image Size: 0x00000000
Checksum: 0x00000000   Time Stamp: 0x52d1973c

The following files will be included in this error report:

C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\b15_appcompat.txt

---

**Figure 68.3: Windows XP**
Earlier, this handler was called Dr. Watson \(^{31}\).

\(^{31}\) wikipedia
By the way, some developers make their own handler that sends information about the program crash to themselves. It is registered with the help of SetUnhandledExceptionFilter() and to be called if the OS does not have any other way to handle the exception. An example is Oracle RDBMS—it saves huge dumps containing all possible information about the CPU and memory state.

Let's write our own primitive exception handler:

```c
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
    unsigned i;

    printf("%s\n", __FUNCTION__);
    printf("ExceptionRecord->ExceptionCode=0x%p\n",
        ExceptionRecord->ExceptionCode);
    printf("ExceptionRecord->ExceptionFlags=0x%p\n",
        ExceptionRecord->ExceptionFlags);
    printf("ExceptionRecord->ExceptionAddress=0x%p\n",
        ExceptionRecord->ExceptionAddress);

    if (ExceptionRecord->ExceptionCode==0xE1223344)
    {
        printf("That's for us\n");
        // yes, we "handled" the exception
        return ExceptionContinueExecution;
    }
    else if (ExceptionRecord->ExceptionCode==
        EXCEPTION_ACCESS_VIOLATION)
    {
        printf("ContextRecord->Eax=0x%08X\n",
            ContextRecord->Eax);
        // will it be possible to 'fix' it?
        printf("trying to fix wrong pointer address\n");
        ContextRecord->Eax=(DWORD)&new_value;
    }
}
```

This example is based on the example from [Pie]

It must be compiled with the SAFESEH option:
```
c1 seh1.cpp /link /safeseh:no
```

More about SAFESEH here:

[MSDN]
/ yes, we "handled" the exception
return ExceptionContinueExecution;
}
else
{
    printf("We do not handle this\n");
    // someone else's problem
    return ExceptionContinueSearch;
}

int main()
{
    DWORD handler = (DWORD)except_handler; // take a pointer to our handler

    // install exception handler
    __asm
    {
        // make EXCEPTION_REGISTRATION record:
        push handler // address of handler
        push FS:[0] // address of previous handler
        mov FS:[0],ESP // add new EXCEPTION_REGISTRATION
    }

    RaiseException (0xE1223344, 0, 0, NULL);

    // now do something very bad
    int* ptr=NULL;
    int val=0;
    val=*ptr;
    printf("val=%d\n", val);

    // deinstall exception handler
    __asm
    {
        // remove our EXCEPTION_REGISTRATION record
        mov eax,[ESP] // get pointer to previous record
        mov FS:[0], EAX // install previous record
        add esp, 8 // clean our EXCEPTION_REGISTRATION off stack
    }
The FS: segment register is pointing to the TIB in win32. The very first element in the TIB is a pointer to the last handler in the chain. We save it in the stack and store the address of our handler there. The structure is named _EXCEPTION_REGISTRATION, it is a simple singly-linked list and its elements are stored right in the stack.

Listing 68.1: MSVC/VC/crt/src/exsup.inc

```c
_EXCEPTION_REGISTRATION struc
    prev     dd   ?
    handler  dd   ?
_EXCEPTION_REGISTRATION ends
```

So each “handler” field points to a handler and an each “prev” field points to the previous record in the stack. The last record has 0xFFFFFFFF (-1) in the “prev” field.

After our handler is installed, we call `RaiseException()`\(^{33}\). This is an user exception. The handler checks the code. If the code is 0xE1223344, it returns ExceptionContinueExecution, which means that handler corrected the CPU state (it is usually a correction of the EIP/ESP registers) and the OS can resume the execution of the. If you alter slightly the code so the handler returns ExceptionContinueSearch, then the OS will call the other handlers, and it's unlikely that one who can handle it will be found, since no one will have any information about it (rather about its code). You will see the standard Windows dialog about a process crash.

What is the difference between a system exceptions and a user one? Here are the system ones:

\(^{33}\text{MSDN}\)
as defined in WinBase.h | as defined in ntstatus.h
---|---
EXCEPTION_ACCESS_VIOLATION | STATUS_ACCESS_VIOLATION
EXCEPTION_DATATYPE_MISALIGNMENT | STATUS_DATATYPE_MISALIGNMENT
EXCEPTION_BREAKPOINT | STATUS_BREAKPOINT
EXCEPTION_SINGLE_STEP | STATUS_SINGLE_STEP
EXCEPTION_ARRAY_BOUNDS_EXCEEDED | STATUS_ARRAY_BOUNDS_EXCEEDED
EXCEPTION_FLT_DENORMAL_OPERAND | STATUS_FLOAT_DENORMAL_OPERAND
EXCEPTION_FLT_DIVIDE_BY_ZERO | STATUS_FLOAT_DIVIDE_BY_ZERO
EXCEPTION_FLT_INEXACT_RESULT | STATUS_FLOAT_INEXACT_RESULT
EXCEPTION_FLT_INVALID_OPERATION | STATUS_FLOAT_INVALID_OPERATION
EXCEPTION_FLT_OVERFLOW | STATUS_FLOAT_OVERFLOW
EXCEPTION_FLT_STACK_CHECK | STATUS_FLOAT_STACK_CHECK
EXCEPTION_FLT_UNDERFLOW | STATUS_FLOAT_UNDERFLOW
EXCEPTION_INT_DIVIDE_BY_ZERO | STATUS_INTEGER_DIVIDE_BY_ZERO
EXCEPTION_INT_OVERFLOW | STATUS_INTEGER_OVERFLOW
EXCEPTION_PRIV_INSTRUCTION | STATUS_PRIVILEGED_INSTRUCTION
EXCEPTION_ILLEGAL_INSTRUCTION | STATUS_ILLEGAL_INSTRUCTION
EXCEPTION_NONCONTINUABLE_EXCEPTION | STATUS_NONCONTINUABLE_EXCEPTION
EXCEPTION_STACK_OVERFLOW | STATUS.Stack_Overflow
EXCEPTION_INVALID_DISPOSITION | STATUS.INVALID_DISPOSITION
EXCEPTION_GUARD_PAGE | STATUS.GUARD_PAGE
EXCEPTION_INVALID_HANDLE | STATUS.INVALID_HANDLE
EXCEPTION_POSSIBLE_DEADLOCK | STATUS.POSSIBLE_DEADLOCK
CONTROL_C_EXIT | STATUS.CONTROL_C_EXIT

That is how the code is defined:

<table>
<thead>
<tr>
<th>S</th>
<th>U</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facility code</td>
<td>Error code</td>
<td></td>
</tr>
</tbody>
</table>

S is a basic status code: 11—error; 10—warning; 01—informational; 00—success. U—whether the code is user code.

That is why we chose 0xE1223344—0xE (1110b) mean this it is 1) user exception; 2) error. But to be honest, this example works fine without these high bits.

Then we try to read a value from memory at address 0. Of course, there is nothing at this address in win32, so an exception is raised. The very first handler is to be called—yours, and it will know about it first, by checking the code if it’s equal to the EXCEPTION_ACCESS_VIOLATION constant.

The code that’s reading from memory at address 0 is looks like this:

Listing 68.2: MSVC 2010
Will it be possible to fix this error “on the fly” and to continue with program execution? Yes, our exception handler can fix the EAX value and let the OS execute this instruction once again. So that is what we do. printf() prints 1234, because after the execution of our handler EAX is not 0, but contains the address of the global variable new_value. The execution will resume.

That is what is going on: the memory manager in the CPU signals about an error, the CPU suspends the thread, finds the exception handler in the Windows kernel, which, in turn, starts to call all handlers in the SEH chain, one by one.

We use MSVC 2010 here, but of course, there is no any guarantee that EAX will be used for this pointer.

This address replacement trick is showy, and we considering it here as an illustration of SEH’s internals. Nevertheless, it’s hard to recall any case where it is used for “on-the-fly” error fixing.

Why SEH-related records are stored right in the stack instead of some other place? Supposedly because the OS is not needing to care about freeing this information, these records are simply disposed when the function finishes its execution. This is somewhat like alloca(): (5.2.4 on page 43).

### 68.3.2 Now let’s get back to MSVC

Supposedly, Microsoft programmers needed exceptions in C, but not in C++, so they added a non-standard C extension to MSVC\(^\text{34}\). It is not related to C++ PL exceptions.

```plaintext
__try {
    ...
}
_except(filter code) {
    handler code
}
```

\(^{34}\text{MSDN}\)
“Finally” block may be instead of handler code:

```
__try
{
    ...
}
__finally
{
    ...
}
```

The filter code is an expression, telling whether this handler code corresponds to
the exception raised. If your code is too big and cannot fit into one expression, a
separate filter function can be defined.

There are a lot of such constructs in the Windows kernel. Here are a couple of
examples from there (WRK):

Listing 68.3: WRK-v1.2/base/ntos/ob/obwait.c

```c
try {
    KeReleaseMutant( (PKMUTANT)SignalObject,
                        MUTANT_INCREMENT,
                        FALSE,
                        TRUE );
}
extcept((GetExceptionCode () == STATUS_ABANDONED ||
            GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
            EXCEPTION_EXECUTE_HANDLER :
            EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();
    goto WaitExit;
}
```

Listing 68.4: WRK-v1.2/base/ntos/cache/cachesub.c

```c
try {
    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                 UserBuffer,
                 MorePages ?
                 (PAGE_SIZE - PageOffset) :
                 (ReceivedLength - PageOffset) );
}
extcept( CcCopyReadExceptionFilter( GetExceptionInformation(),
Here is also a filter code example:

Listing 68.5: WRK-v1.2/base/ntos/cache/copysup.c

LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
)

/*@++

Routine Description:

This routine serves as an exception filter and has the special job of extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR beneath us.

Arguments:

    ExceptionPointer - A pointer to the exception record that contains the real Io Status.

    ExceptionCode - A pointer to an NTSTATUS that is to receive the real status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

-*/

{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
        (ExceptionPointer->ExceptionRecord->NumberParameters >= 3) ) {
        
        
    
}
*ExceptionCode = (NTSTATUS) ExceptionPointer->\^\ ExceptionRecord->ExceptionInformation[2];
}

ASSERT( !NT_SUCCESS(*ExceptionCode) );
return EXCEPTION_EXECUTE_HANDLER;
}

Internally, SEH is an extension of the OS-supported exceptions. But the handler function is _except_handler3 (for SEH3) or _except_handler4 (for SEH4). The code of this handler is MSVC-related, it is located in its libraries, or in msvcr*.dll. It is very important to know that SEH is a MSVC thing. Other win32-compilers may offer something completely different.

**SEH3**

SEH3 has _except_handler3 as a handler function, and extends the _EXCEPTION_REGISTRATION table, adding a pointer to the *scope table* and previous try level variable. SEH4 extends the *scope table* by 4 values for buffer overflow protection.

The *scope table* is a table that consists of pointers to the filter and handler code blocks, for each nested level of *try/except*. 
Again, it is very important to understand that the OS takes care only of the prev/handle fields, and nothing more. It is the job of the _except_handler3 function to read the other fields and scope table, and decide which handler to execute and when.

The source code of the _except_handler3 function is closed. However, Sanos OS, which has a win32 compatibility layer, has the same functions reimplemented, which are somewhat equivalent to those in Windows 35. Another reimplementation is present in Wine36 and ReactOS37.

If the filter pointer is NULL, the handler pointer is the pointer to the finally code block.

During execution, the previous try level value in the stack changes, so _except_handler3 can get information about the current level of nestedness, in order to know which scope table entry to use.

---

35 [http://go.yurichev.com/17058](http://go.yurichev.com/17058)
36 GitHub
37 [http://go.yurichev.com/17060](http://go.yurichev.com/17060)
SEH3: one try/except block example

```c
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
        *p = 13; // causes an access violation exception;
        printf("hello #2!\n");
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
             EXCEPTION_EXECUTE_HANDLER :
             EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
```

Listing 68.6: MSVC 2003

```assembly
$SG74605 DB 'hello #1!', 0aH, 00H
$SG74606 DB 'hello #2!', 0aH, 00H
$SG74608 DB 'access violation, can't recover', 0aH, 00H
_DATA ENDS

; scope table:
CONST SEGMENT
$T74622 DD 0xffffffffH ; previous try level
    DD FLAT:$L74617 ; filter
    DD FLAT:$L74618 ; handler
CONST ENDS
_TEXT SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24
_main PROC NEAR
    push ebp
    mov ebp, esp
    push -1 ; previous try level
    push OFFSET FLAT:$T74622 ; scope table
    push OFFSET FLAT:__except_handler3 ; handler
```
mov eax, DWORD PTR fs:__except_list
push eax ; prev
mov DWORD PTR fs:__except_list, esp
add esp, -16
; 3 registers to be saved:
push ebx
push esi
push edi
mov DWORD PTR __$SEHRec$[ebp], esp
mov DWORD PTR _p$[ebp], 0
mov DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try
level
push OFFSET FLAT:$SG74605 ; 'hello #1!'
call _printf
add esp, 4
mov eax, DWORD PTR _p$[ebp]
mov DWORD PTR [eax], 13
push OFFSET FLAT:$SG74606 ; 'hello #2!'
call _printf
add esp, 4
mov DWORD PTR __$SEHRec$[ebp+20], -1 ; previous try
level
jmp SHORT $L74616

; filter code:
$L74617: $L74627:
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov edx, DWORD PTR [ecx]
    mov eax, DWORD PTR [edx]
    mov DWORD PTR $T74621[ebp], eax
    mov eax, DWORD PTR $T74621[ebp]
    sub eax, -1073741819; c0000005H
    neg eax
    sbb eax, eax
    inc eax
$L74619: $L74626:
    ret 0

; handler code:
$L74618: $L74619:
    mov esp, DWORD PTR __$SEHRec$[ebp]
push OFFSET FLAT:$SG74608 ; 'access violation, can't recover'
call _printf
add esp, 4
Here we see how the SEH frame is constructed in the stack. The *scope table* is located in the CONST segment—indeed, these fields are not to be changed. An interesting thing is how the *previous try level* variable has changed. The initial value is 0xFFFFFFFF (−1). The moment when the body of the *try* statement is opened is marked with an instruction that writes 0 to the variable. The moment when the body of the *try* statement is closed, −1 is written back to it. We also see the addresses of filter and handler code. Thus we can easily see the structure of the *try/except* constructs in the function.

Since the SEH setup code in the function prologue may be shared between many functions, sometimes the compiler inserts a call to the `SEH_prolog()` function in the prologue, which does just that. The SEH cleanup code is in the `SEH_epilog()` function.

Let’s try to run this example in *tracer*:

```
tracer.exe -l:2.exe --dump-seh
```

**Listing 68.7: tracer.exe output**

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054)
ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x00008e3c
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!main+0x60)
  _except_handler3)
SEH3 frame. previous trylevel=0
 scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0x60)
 handler=0x401088 (2.exe!main+0x78)
```
We see that the SEH chain consists of 4 handlers.

The first two are located in our example. Two? But we made only one? Yes, another one was set up in the CRT function _mainCRTStartup(), and as it seems that it handles at least FPU exceptions. Its source code can be found in the MSVC installation: `crt/src/winxfltr.c`.

The third is the SEH4 one in ntdll.dll, and the fourth handler is not MSVC-related and is located in ntdll.dll, and has a self-describing function name.

As you can see, there are 3 types of handlers in one chain: one is not related to MSVC at all (the last one) and two MSVC-related: SEH3 and SEH4.

**SEH3: two try/except blocks example**

```c
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
}
```
else
{
    printf("not our exception\n");
    return EXCEPTION_CONTINUE_SEARCH;
};

int main()
{
    int* p = NULL;
    __try
    {
        __try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
            printf ("0x112233 raised. now let's crash\n");
            *p = 13; // causes an access violation exception
;
        }
        __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION
;
        EXCEPTION_EXECUTE_HANDLER :
        EXCEPTION_CONTINUE_SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(),
GetExceptionInformation()))
    {
        // the filter_user_exceptions() function answering to
        // "is this exception belongs to this block?"
        // if yes, do the follow:
        printf("user exception caught\n");
    }
}

Now there are two try blocks. So the scope table now has two entries, one for each block. Previous try level changes as execution flow enters or exits the try block.

Listing 68.8: MSVC 2003

$SG74606 DB 'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB 'yes, that is our exception', 0aH, 00H
$SG74610 DB 'not our exception', 0aH, 00H
$SG74617 DB 'hello!', 0aH, 00H
$SG74619 DB '0x112233 raised. now let''s crash', 0AH, 00H
$SG74621 DB 'access violation, can''t recover', 0AH, 00H
$SG74623 DB 'user exception caught', 0AH, 00H

_code$ = 8 ; size = 4
_ep$ = 12 ; size = 4
_filter_user_exceptions PROC NEAR
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _code$[ebp]
    push eax
    push OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call _printf
    add esp, 8
    cmp DWORD PTR _code$[ebp], 1122867; 00112233H
    jne SHORT $L74607
    push OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call _printf
    add esp, 4
    mov eax, 1
    jmp SHORT $L74605
$L74607:
    push OFFSET FLAT:$SG74610 ; 'not our exception'
    call _printf
    add esp, 4
    xor eax, eax
$L74605:
    pop ebp
    ret 0
_filter_user_exceptions ENDP

; scope table:
CONST SEGMENT
$T74644 DD 0fffffffH ; previous try level for outer ▼
    DD FLAT:$L74634 ; outer block filter
    DD FLAT:$L74635 ; outer block handler
    DD 00H ; previous try level for inner ▼
        DD FLAT:$L74638 ; inner block filter
        DD FLAT:$L74639 ; inner block handler
CONST ENDS

$T74643 = -36 ; size = 4
$T74642 = -32 ; size = 4
_p$ = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24
_main PROC NEAR
    push ebp
    mov ebp, esp
    push -1 ; previous try level
    push OFFSET FLAT:$T74644
    push OFFSET FLAT:__except_handler3
    mov eax, DWORD PTR fs:__except_list
    push eax
    mov DWORD PTR fs:__except_list, esp
    add esp, -20
    push ebx
    push esi
    push edi
    mov DWORD PTR __$SEHRec$[ebp], esp
    mov DWORD PTR _p$[ebp], 0
    mov DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block
    mov DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block
    push OFFSET FLAT:$SG74617 ; 'hello!'
    call _printf
    add esp, 4
    push 0
    push 0
    push 0
    push 1122867 ; 00112233H
    call DWORD PTR __imp__RaiseException@16
    push OFFSET FLAT:$SG74619 ; '0x112233 raised. now let''s crash'
    call _printf
    add esp, 4
    mov eax, DWORD PTR _p$[ebp]
    mov DWORD PTR [eax], 13
    mov DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block
    jmp SHORT $L74615

; inner block filter:
$L74638:
$L74650:
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov edx, DWORD PTR [ecx]
    mov eax, DWORD PTR [edx]
    mov DWORD PTR $T74643[ebp], eax
    mov eax, DWORD PTR $T74643[ebp]
    sub eax, -1073741819; c0000005H
    neg eax
sbb eax, eax  
inc eax  
$L74640$:  
$L74648$:  
  ret 0  
}; inner block handler:  
$L74639$:  
  mov esp, DWORD PTR __SEHRec$[ebp]  
  push OFFSET FLAT:$SG74621 ; 'access violation, can't ↵
      ↗ recover'  
  call _printf  
  add esp, 4  
  mov DWORD PTR __SEHRec$[ebp+20], 0 ; inner try block ↵
      ↗ exited. set previous try level back to 0  
$L74615$:  
  mov DWORD PTR __SEHRec$[ebp+20], -1 ; outer try block ↵
      ↗ exited, set previous try level back to -1  
  jmp SHORT $L74633  
}; outer block filter:  
$L74634$:  
$L74651$:  
  mov ecx, DWORD PTR __SEHRec$[ebp+4]  
  mov edx, DWORD PTR [ecx]  
  mov eax, DWORD PTR [edx]  
  mov DWORD PTR $T74642[ebp], eax  
  mov ecx, DWORD PTR __SEHRec$[ebp+4]  
  push ecx  
  mov edx, DWORD PTR $T74642[ebp]  
  push edx  
  call _filter_user_exceptions  
  add esp, 8  
$L74636$:  
$L74649$:  
  ret 0  
}; outer block handler:  
$L74635$:  
  mov esp, DWORD PTR __SEHRec$[ebp]  
  push OFFSET FLAT:$SG74623 ; 'user exception caught'  
  call _printf  
  add esp, 4  
  mov DWORD PTR __SEHRec$[ebp+20], -1 ; both try blocks ↵
      ↗ exited. set previous try level back to -1  
$L74633$:  

1018
CHAPTER 68. WINDOWS NT

xor eax, eax
mov ecx, DWORD PTR __$SEHRec$[ebp+8]
mov DWORD PTR fs:__except_list, ecx
pop edi
pop esi
pop ebx
mov esp, ebp
pop ebp
ret 0
_main ENDP

If we set a breakpoint on the printf() function, which is called from the handler, we can also see how yet another SEH handler is added. Perhaps it’s another machinery inside the SEH handling process. Here we also see our scope table consisting of 2 entries.

tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh

Listing 68.9: tracer.exe output

(0) 3.exe!printf
EAX=0x0000000b EBX=0x00000000 ECX=0x00040cc58 EDX=0x0008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll!.
data!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!
data!_except_handler3)

SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x4010e8 (3.exe!
data!main+0x78) handler=0x401100 (3.exe!
data!main+0x90)

* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x771f71f5 (ntdll!.
data!__except_handler4)

SEH4 frame. previous trylevel=0
scopetable entry[0]. previous try level=-2, filter=0x771f90eb (ntdll!.
data!__safe_se_handler_table+0x20) handler=0
1019
SEH4

During a buffer overflow (18.2 on page 370) attack, the address of the scope table can be rewritten, so starting from MSVC 2005, SEH3 was upgraded to SEH4 in order to have buffer overflow protection. The pointer to the scope table is now xorred with a security cookie. The scope table was extended to have a header consisting of two pointers to security cookies. Each element has an offset inside the stack of another value: the address of the stack frame (EBP) xorred with the security_cookie, placed in the stack. This value will be read during exception handling and checked for correctness. The security cookie in the stack is random each time, so hopefully a remote attacker can’t predict it.

The initial previous try level is −2 in SEH4 instead of −1.
Here are both examples compiled in MSVC 2012 with SEH4:

**Listing 68.10: MSVC 2012: one try block example**

```assembly
$SG85485 DB 'hello #1!', 0aH, 00H
$SG85486 DB 'hello #2!', 0aH, 00H
$SG85488 DB 'access violation, can''t recover', 0aH, 00H

; scope table:
_xdata$x SEGMENT
__sehtable$main DD 0xfffffffffH ; GS Cookie Offset
   DD 00H ; GS Cookie XOR Offset
   DD 0xffffffffcH ; EH Cookie Offset
   DD 00H ; EH Cookie XOR Offset
   DD 0xfffffffffH ; previous try level
   DD FLAT:$LN12@main ; filter
   DD FLAT:$LN8@main ; handler
_xdata$x ENDS
```
$T2 = -36 ; size = 4
_p68 = -32 ; size = 4
tv68 = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24

_main PROC
push ebp
mov ebp, esp
push -2
push OFFSET __sehtable$_main
push OFFSET __except_handler4
mov eax, DWORD PTR fs:0
push eax
add esp, -20
push ebx
push esi
push edi
mov eax, DWORD PTR ___security_cookie
xor DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to
\ scope table
xor eax, ebp
push eax ; ebp ^
\ security_cookie
lea eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to \ VC_EXCEPTION_REGISTRATION_RECORD
mov DWORD PTR fs:0, eax
mov DWORD PTR __$SEHRec$[ebp], esp
mov DWORD PTR _p$[ebp], 0
mov DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
push OFFSET $SG85485 ; 'hello #1!'
call _printf
add esp, 4
mov eax, DWORD PTR _p$[ebp]
mov DWORD PTR [eax], 13
push OFFSET $SG85486 ; 'hello #2!'
call _printf
add esp, 4
mov DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try \ \ level
jmp SHORT $LN6@main

; filter:
$LN7@main:
$LN12@main:
mov ecx, DWORD PTR __$SEHRec$[ebp+4]
mov edx, DWORD PTR [ecx]
mov eax, DWORD PTR [edx]
mov DWORD PTR $T2[ebp], eax
Listing 68.11: MSVC 2012: two try blocks example

$SG85486 DB 'in filter. code=0x%08X', 0AH, 00H
$SG85488 DB 'yes, that is our exception', 0AH, 00H
$SG85490 DB 'not our exception', 0AH, 00H
$SG85497 DB 'hello!', 0AH, 00H
$SG85499 DB '0x112233 raised. now let''s crash', 0AH, 00H
$SG85501 DB 'access violation, can''t recover', 0AH, 00H
$SG85503 DB 'user exception caught', 0AH, 00H

xdata$x SEGMENT
__sehtable$_main DD 0fffffffeH ; GS Cookie Offset
                  DD 00H         ; GS Cookie XOR Offset
                  DD 0ffffffc8H ; EH Cookie Offset
                  DD 00H         ; EH Cookie Offset
_main ENDP
DD 0xfffffffeH ; previous try level for
\outer block
DD FLAT:$LN19@main ; outer block filter
DD FLAT:$LN9@main ; outer block handler
DD 00H ; previous try level for
\inner block
DD FLAT:$LN18@main ; inner block filter
DD FLAT:$LN13@main ; inner block handler

\xdata$x ENDS

$T2 = -40 ; size = 4
$T3 = -36 ; size = 4
_p$ = -32 ; size = 4
tv72 = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24

\_main PROC
push ebp
mov ebp, esp
push -2 ; initial previous try level
push OFFSET __sehtable$_main
push OFFSET __except_handler4
mov eax, DWORD PTR fs:0
push eax ; prev
add esp, -24
push ebx
push esi
push edi
mov eax, DWORD PTR ___security_cookie
xor DWORD PTR __$SEHRec$[ebp+16], eax ; xored \pointer to scope table
xor eax, ebp ; ebp ^ \security_cookie
push eax
lea eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to \VC_EXCEPTION_REGISTRATION_RECORD
mov DWORD PTR fs:0, eax
mov DWORD PTR __$SEHRec$[ebp], esp
mov DWORD PTR _p$[ebp], 0
mov DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try
\block, setting previous try level=0
mov DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try
\block, setting previous try level=1
push OFFSET $SG85497 ; 'hello!'
call _printf
add esp, 4
push 0
push 0
CHAPTER 68. WINDOWS NT

push 0
push 1122867 ; 00112233H
call DWORD PTR __imp__RaiseException@16
push OFFSET $SG85499 ; '0x112233 raised. now let''s crash
\n' call _printf
add esp, 4
mov eax, DWORD PTR _p$[ebp]
mov DWORD PTR [eax], 13
mov DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try
\nblock, set previous try level back to 0
jmp SHORT $LN2@main

; inner block filter:
$LN12@main:
$LN18@main:
mov ecx, DWORD PTR __$SEHRec$[ebp+4]
mov edx, DWORD PTR [ecx]
mov eax, DWORD PTR [edx]
mov DWORD PTR $T3[ebp], eax
cmp DWORD PTR $T3[ebp], -1073741819 ; c0000005H
jne SHORT $LN5@main
mov DWORD PTR tv72[ebp], 1
jmp SHORT $LN6@main
$LN5@main:
mov DWORD PTR tv72[ebp], 0
$LN6@main:
mov eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
ret 0

; inner block handler:
$LN13@main:
mov esp, DWORD PTR __$SEHRec$[ebp]
push OFFSET $SG85501 ; 'access violation, can''t recover'
call _printf
add esp, 4
mov DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try
\nblock, setting previous try level back to 0
$LN2@main:
mov DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both
\nblocks, setting previous try level back to -2
jmp SHORT $LN7@main

; outer block filter:
$LN8@main:
mov ecx, DWORD PTR __$SEHRec$[ebp+4]
mov edx, DWORD PTR [ecx]
mov eax, DWORD PTR [edx]
mov DWORD PTR $T2[ebp], eax
push ecx
mov edx, DWORD PTR $T2[ebp]
push edx
call _filter_user_exceptions
add esp, 8

mov esp, DWORD PTR __$SEHRec$[ebp]
push OFFSET $SG85503 ; 'user exception caught'
call _printf
add esp, 4
mov DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level back to -2

xor eax, eax
mov ecx, DWORD PTR __$SEHRec$[ebp+8]
mov DWORD PTR fs:0, ecx
pop ecx
pop edi
pop esi
pop ebx
mov esp, ebp
pop ebp
ret 0

_filter_user_exceptions ENDP

_code$ = 8 ; size = 4
_ep$ = 12 ; size = 4
_filter_user_exceptions PROC
push ebp
mov ebp, esp
mov eax, DWORD PTR _code$[ebp]
push eax
push OFFSET $SG85486 ; 'in filter. code=0x%08X'
call _printf
add esp, 8
cmp DWORD PTR _code$[ebp], 1122867 ; 00112233H
Here is the meaning of the cookies: Cookie Offset is the difference between the address of the saved EBP value in the stack and the $EBP \oplus \text{security_cookie}$ value in the stack. Cookie XOR Offset is an additional difference between the $EBP \oplus \text{security_cookie}$ value and what is stored in the stack. If this equation is not true, the process is to halt due to stack corruption:

$$\text{security_cookie} \oplus (\text{Cookie XOR Offset} + \text{address of saved EBP}) == \text{stack}[\text{address of saved EBP} + \text{Cookie Offset}]$$

If Cookie Offset is −2, this implies that it is not present.

Cookies checking is also implemented in my tracer, see GitHub for details.

It is still possible to fall back to SEH3 in the compilers after (and including) MSVC 2005 by setting the /GS- option, however, the CRT code use SEH4 anyway.

### 68.3.3 Windows x64

As you might think, it is not very fast to set up the SEH frame at each function prologue. Another performance problem is changing the previous try level value many times during the function’s execution. So things are changed completely in x64: now all pointers to try blocks, filter and handler functions are stored in another PE segment .pdata, and from there the OS’s exception handler takes all the information.

Here are the two examples from the previous section compiled for x64:

#### Listing 68.12: MSVC 2012

```assembly
$SG86276 DB 'hello #1!', 0aH, 00H
```
$SG86277 DB 'hello #2!', 0AH, 00H
$SG86279 DB 'access violation, can't recover', 0AH, 00H

pdata SEGMENT
$pdata$main DD imagerel $LN9
      DD imagerel $LN9+61
      DD imagerel $unwind$main
pdata ENDS
pdata SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
      DD imagerel main$filt$0+32
      DD imagerel $unwind$main$filt$0
pdata ENDS
xdata SEGMENT
$unwind$main DD 020609H
      DD 030023206H
      DD imagerel __C_specific_handler
      DD 01H
      DD imagerel $LN9+8
      DD imagerel $LN9+40
      DD imagerel main$filt$0
      DD imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
      DD 050023206H
xdata ENDS

_TEXT SEGMENT
main PROC
$LN9:
push rbx
sub  rsp, 32
xor  ebx, ebx
lea  rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
call printf
mov  DWORD PTR [rbx], 13
lea  rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
call printf
jmp  SHORT $LN8@main
$LN6@main:
  lea  rcx, OFFSET FLAT:$SG86279 ; 'access violation, can't recover'
call  printf
npad 1 ; align next label
$LN8@main:
  xor  eax, eax
  add  rsp, 32
  pop  rbx

Listing 68.13: MSVC 2012

| $SG86277 DB | 'in filter. code=0x%08X', 0aH, 00H |
| $SG86279 DB | 'yes, that is our exception', 0aH, 00H |
| $SG86281 DB | 'not our exception', 0aH, 00H |
| $SG86288 DB | 'hello!', 0aH, 00H |
| $SG86290 DB | '0x112233 raised. now let''s crash', 0aH, 00H |
| $SG86292 DB | 'access violation, can''t recover', 0aH, 00H |
| $SG86294 DB | 'user exception caught', 0aH, 00H |

pdata SEGMENT
pdata$filter_user_exceptions DD imagerel $LN6
DD imagerel $LN6+73
DD imagerel $unwind$filter_user_exceptions
pdata$main DD imagerel $LN14
DD imagerel $LN14+95
DD imagerel $unwind$main
pdata ENDS
pdata SEGMENT
pdata$main$filt$0 DD imagerel main$filt$0
DD imagerel main$filt$0+32
DD imagerel $unwind$main$filt$0
pdata$main$filt$1 DD imagerel main$filt$1
DD imagerel main$filt$1+30
DD imagerel $unwind$main$filt$1
pdata ENDS

xdata SEGMENT
$unwind$filter_user_exceptions DD 020601H
    DD 030023206H
$unwind$main DD 020609H
    DD 030023206H
    DD imagerel __C_specific_handler
    DD 02H
    DD imagerel $LN14+8
    DD imagerel $LN14+59
    DD imagerel main$filt$0
    DD imagerel $LN14+59
    DD imagerel $LN14+8
    DD imagerel $LN14+74
    DD imagerel main$filt$1
    DD imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
    DD 050023206H
$unwind$main$filt$1 DD 020601H
    DD 050023206H
xdata ENDS

_TEXT SEGMENT
main PROC
$LN14:
    push rbx
    sub rsp, 32
    xor ebx, ebx
    lea rcx, OFFSET FLAT:$SG86288 ; 'hello!'
    call printf
    xor r9d, r9d
    xor r8d, r8d
    xor edx, edx
    mov ecx, 1122867 ; 00112233H
    call QWORD PTR __imp_RaiseException
    lea rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised.'
    now let's crash'
    call printf
    mov DWORD PTR [rbx], 13
    jmp SHORT $LN13@main
$LN11@main:
    lea rcx, OFFSET FLAT:$SG86292 ; 'access violation, can't recover'
    call printf
    npad 1 ; align next label
$LN13@main:
jmp SHORT $LN9@main

$LN7@main:
lea rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
call printf
npad 1 ; align next label

$LN9@main:
xor eax, eax
add rsp, 32
pop rbx
ret 0
main ENDP

text$x SEGMENT
main$filt$0 PROC
push rbp
sub rsp, 32
mov rbp, rdx
$LN10@main$filt$:
mov rax, QWORD PTR [rcx]
xor ecx, ecx
cmp DWORD PTR [rax], -1073741819; c0000005H
sete cl
mov eax, ecx
$LN12@main$filt$:
add rsp, 32
pop rbp
ret 0
int 3
main$filt$0 ENDP

main$filt$1 PROC
push rbp
sub rsp, 32
mov rbp, rdx
$LN6@main$filt$:
mov rax, QWORD PTR [rcx]
mov rdx, rcx
mov ecx, DWORD PTR [rax], -1073741819; c0000005H
mov eax, ecx
$LN8@main$filt$:
add rsp, 32
pop rbp
ret 0
int 3
main$filt$1 ENDP
FILTER_USER_EXCEPTIONS PROC
$LN6:
    push rbx
    sub rsp, 32
    mov ebx, ecx
    mov edx, ecx
    lea rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x
\text{\%08X}'
\text{\%08X}'
call printf
    cmp ebx, 1122867; 00112233H
    jne SHORT $LN2@filter_use
    lea rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our 
\text{\%08X}'
    call printf
    xor eax, eax
    add rsp, 32
    pop rbx
    ret 0
$LN2@filter_use:
    lea rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
    call printf
    xor eax, eax
    add rsp, 32
    pop rbx
    ret 0
FILTER_USER_EXCEPTIONS ENDP
\text{\%08X}'

Read [Sko12] for more detailed information about this.

Aside from exception information, .pdata is a section that contains the addresses of almost all function starts and ends, hence it may be useful for a tools targeted at automated analysis.

68.3.4 Read more about SEH

[Pie], [Sko12].
Critical sections in any **OS** are very important in multithreaded environment, mostly for giving a guarantee that only one thread can access some data in a single moment of time, while blocking other threads and interrupts.

That is how a CRITICAL_SECTION structure is declared in *Windows NT* line OS:

```
Listing 68.14: (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;
    
    // The following three fields control entering and exiting the critical section for the resource
    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread; // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount; // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

That's is how EnterCriticalSection() function works:

```
Listing 68.15: Windows 2008/ntdll.dll/x86 (begin)
_RtlEnterCriticalSection@4
```

```
var_C = dword ptr -0Ch
var_8 = dword ptr -8
var_4 = dword ptr -4
arg_0 = dword ptr 8

mov edi, edi
push ebp
mov ebp, esp
sub esp, 0Ch
push esi
push edi
mov edi, [ebp+arg_0]
lea esi, [edi+4] ; LockCount
mov eax, esi
```
The most important instruction in this code fragment is BTR (prefixed with LOCK): the zeroth bit is stored in the CF flag and cleared in memory. This is an atomic operation, blocking all other CPUs’ access to this piece of memory (see the LOCK prefix before the BTR instruction). If the bit at LockCount is 1, fine, reset it and return from the function: we are in a critical section. If not—the critical section is already occupied by other thread, so wait. The wait is done there using WaitForSingleObject().

And here is how the LeaveCriticalSection() function works:

Listing 68.16: Windows 2008/ntdll.dll/x86 (begin)

```
 lock btr dword ptr [eax], 0
 jnb    wait ; jump if CF=0
loc_7DE922DD:
   mov      eax, large fs:18h
   mov      ecx, [eax+24h]
   mov      [edi+0Ch], ecx
   mov      dword ptr [edi+8], 1
   pop      edi
   xor      eax, eax
   pop      esi
   mov      esp, ebp
   pop      ebp
   retn 4
... skipped
```

```
Listing 68.16: Windows 2008/ntdll.dll/x86 (begin)

_RtlLeaveCriticalSection@4 proc near
arg_0 = dword ptr 8

   mov      edi, edi
   push     ebp
   mov      ebp, esp
   push     esi
   mov      esi, [ebp+arg_0]
   add      dword ptr [esi+8], 0FFFFFFFFh ;⤦
   ↓ RecursionCount
   jnz      short loc_7DE922B2
   push     ebx
   push     edi
   lea      edi, [esi+4]   ; LockCount
   mov      dword ptr [esi+0Ch], 0
   mov      ebx, 1
   mov      eax, edi


```
lock xadd [eax], ebx  
inc ebx  
cmp ebx, 0FFFFFFFFh  
jnz loc_7DEA8EB7

loc_7DE922B0:  
pop edi  
pop ebx

loc_7DE922B2:  
xor eax, eax  
pop esi  
pop ebp  
retn 4

... skipped

XADD is “exchange and add”. In this case, it adds 1 to LockCount and stores the result in the EBX register, and at the same time 1 goes to LockCount. This operation is atomic since it is prefixed by LOCK as well, meaning that all other CPUs or CPU cores in system are blocked from accessing this point in memory.

The LOCK prefix is very important: without it two threads, each of which works on separate CPU or CPU core can try to enter a critical section and to modify the value in memory, which will result in non-deterministic behaviour.
Part VII

Tools
Chapter 69

Disassembler

69.1 IDA

An older freeware version is available for download \(^1\).

Hot-keys cheatsheet: F.1 on page 1331

\(^1\) hex-rays.com/products/ida/support/download_freeware.shtml
Chapter 70

Debugger

70.1 OllyDbg

Very popular user-mode win32 debugger:
ollydbg.de.
Hot-keys cheatsheet: F.2 on page 1332

70.2 GDB

Not very popular debugger among reverse engineers, but very comfortable nevertheless.
Some commands: F.5 on page 1333.

70.3 tracer

The author often uses tracer\(^1\) instead of a debugger.
The author of these lines stopped using a debugger eventually, since all he needs from it is to spot function arguments while executing, or registers state at some point. Loading a debugger each time is too much, so a small utility called tracer was born. It works from command line, allows intercepting function execution, setting breakpoints at arbitrary places, reading and changing registers state, etc.

\(^1\)yurichev.com
However, for learning purposes it is highly advisable to trace code in a debugger manually, watch how the registers state changes (e.g. classic SoftICE, OllyDbg, WinDbg highlight changed registers), flags, data, change them manually, watch the reaction, etc.
Chapter 71

System calls tracing

71.0.1 strace / dtruss

It shows which system calls (syscalls( 66 on page 973)) are called by a process right now.

For example: Por ejemplo:

```
# strace df -h
...
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0..."
      512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb75b3000
```

Mac OS X has dtruss for doing the same.

Cygwin also has strace, but as far as it’s known, it works only for .exe-files compiled for the cygwin environment itself.
Chapter 72

Decompilers

There is only one known, publicly available, high-quality decompiler to C code: Hex-Rays:

hex-rays.com/products/decompiler/
Chapter 73

Other tools

• Microsoft Visual Studio Express\(^1\): Stripped-down free version of Visual Studio, convenient for simple experiments. Some useful options: F.3 on page 1333.

• Hiew\(^2\): for small modifications of code in binary files.

• binary grep: a small utility for searching any byte sequence in a big pile of files, including non-executable ones: GitHub.

\(^1\)visualstudio.com/en-US/products/visual-studio-express-vs
\(^2\)hiew.ru
Part VIII

Examples of real-world RE tasks
Let’s see if it’s possible to hack Task Manager slightly so it would detect more CPU cores.

Let us first think, how does the Task Manager know the number of cores? There is the GetSystemInfo() win32 function present in win32 userspace which can tell us this. But it’s not imported in taskmgr.exe. There is, however, another one in NTAPI, NtQuerySystemInformation(), which is used in taskmgr.exe in several places. To get the number of cores, one has to call this function with the SystemBasicInformation constant as a first argument (which is zero

The second argument has to point to the buffer which is getting all the information.

So we need to find all calls to the NtQuerySystemInformation(0, ?, ?, ?, ?) function. Let’s open taskmgr.exe in IDA. What is always good about Microsoft executables is that IDA can download the corresponding PDB file for this executable and show all function names. It is visible that Task Manager is written in C++ and some of the function names and classes are really speaking for themselves. There are classes CAdapter, CNetPage, CPerfPage, CProcInfo, CProcPage, CSvcPage, CTaskPage, CUserPage. Apparently, each class corresponds to each tab in Task Manager.

Let’s visit each call and add comment with the value which is passed as the first function argument. We will write “not zero” at some places, because the value there was clearly not zero, but something really different (more about this in the second part of this chapter). And we are looking for zero passed as argument, after all.

\[1\] MSDN
Yes, the names are really speaking for themselves.

When we closely investigate each place where NtQuerySystemInformation(0, ?, ?, ?) is called, we quickly find what we need in the InitPerfInfo() function:

```
Listing 74.1: taskmgr.exe (Windows Vista)

.text:10000B4B3 xor r9d, r9d
.text:10000B4B6 lea rdx, [rsp+0C78h+var_C58] ; buffer
.text:10000B4B8 xor ecx, ecx
.text:10000B4BD lea ebp, [r9+40h]
.text:10000B4C1 mov r8d, ebp
.text:10000B4C4 call cs:__imp_NtQuerySystemInformation ; 0
.text:10000B4CA xor ebx, ebx
.text:10000B4CC cmp eax, ebx
.text:10000B4CE jge short loc_10000B5D7
.text:10000B4D0 loc_10000B4D0: ; CODE
     ; XREF: InitPerfInfo(void)+97
.text:10000B4D0 ;
     ; InitPerfInfo(void)+AF
.text:10000B4D0 xor al, al
.text:10000B4D2 jmp loc_10000B5EA
.text:10000B4D7 ;
.text:10000B4D7 loc_10000B4D7: ; CODE
     ; XREF: InitPerfInfo(void)+36
.text:10000B4D7 mov eax, [rsp+0C78h+var_C50]
.text:10000B4DB mov esi, ebx
.text:10000B4DD mov r12d, 3E80h
```
g_cProcessors is a global variable, and this name was assigned by IDA according to the PDB loaded from Microsoft’s symbol server.

The byte is taken from var_C20. And var_C58 is passed to NtQuerySystemInformation as a pointer to the receiving buffer. The difference between 0xC20 and 0xC58 is 0x38 (56). Let’s take a look at format of the return structure, which we can find in MSDN:

```plaintext
typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    UCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION;
```

This is a x64 system, so each PVOID takes 8 byte. All reserved fields in the structure take 24 + 4 * 8 = 56 bytes. Oh yes, this implies that var_C20 is the local stack is exactly the NumberOfProcessors field of the SYSTEM_BASIC_INFORMATION structure.

Let’s check our guess. Copy taskmgr.exe from C:\Windows\System32 to some other folder (so the Windows Resource Protection will not try to restore the patched taskmgr.exe).

Let’s open it in Hiew and find the place:

```plaintext
Figure 74.2: Hiew: find the place to be patched
```
Let's replace the MOVZX instruction with ours. Let's pretend we've got 64 CPU cores. Add one additional NOP (because our instruction is shorter than the original one):

```
00 0000A8F8: 040386C7458
00 0000A8FD: 4889056A00100
00 0000A904: 66684000
00 0000A905: 90
00 0000A906: 0f47c5
00 0000A907: 3ac3
00 0000A908: 880574950100
00 0000A914: 7645
00 0000A915: 4888fb
00 0000A916: 498804
00 0000A91c: 88cd
```

Figure 74.3: Hiew: patch it

And it works! Of course, the data in the graphs is not correct. At times, Task Manager even shows an overall CPU load of more than 100%.

The biggest number Task Manager is not crashes with is 64. Apparently, Task Manager in Windows Vista was not tested on computers with a large number of cores. So there are probably some static data structures inside it limited to 64 cores.
CHAPTER 74. TASK MANAGER PRACTICAL JOKE (WINDOWS VISTA)

74.1 Using LEA to load values

Sometimes, LEA is used in taskmgr.exe instead of MOV to set the first argument of NtQuerySystemInformation():

### Listing 74.2: taskmgr.exe (Windows Vista)

```assembly
xor r9d, r9d
div dword ptr [rsp+4C8h+WndClass.
lea rdx, [rsp+4C8h+VersionInformation]
lea ecx, [r9+2] ; put 2 to ECX
mov r8d, 138h
mov ebx, eax
; ECX=SystemPerformanceInformation
call cs:__imp_NtQuerySystemInformation ; 2
...

mov r8d, 30h
lea r9, [rsp+298h+var_268]
lea rdx, [rsp+298h+var_258]
lea ecx, [r8-2Dh] ; put 3 to ECX
; ECX=SystemTimeOfDayInformation
call cs:__imp_NtQuerySystemInformation ;
\not zero
...

mov rbp, [rsi+8]
mov r8d, 20h
lea r9, [rsp+98h+arg_0]
lea rdx, [rsp+98h+var_78]
lea ecx, [r8+2Fh] ; put 0x4F to ECX
mov [rsp+98h+var_60], ebx
mov [rsp+98h+var_68], rbp
; ECX=SystemSuperfetchInformation
call cs:__imp_NtQuerySystemInformation ;
\not zero
```

It's hard to say why, but it is what MSVC often does. Maybe this is some kind of optimization and LEA works faster or better than loading values using MOV?

Another example of such thing is: 64.5.1 on page 959.
Chapter 75

Color Lines game practical joke

This is a very popular game with several implementations in existence. We can take one of them, called BallTriX, from 1997, available freely at http://go.yurichev.com/17311. Here is how it looks:

![Figure 75.1: How this game looks usually](image)

---

\(^1\)Or at http://go.yurichev.com/17365 or http://go.yurichev.com/17366.
So let's see, is it be possible to find the random generator and do some trick with it. IDA quickly recognize the standard _rand function in balltrix.exe at 0x00403DA0. IDA also shows that it is called only from one place:

```
.text:00402C9C sub_402C9C proc near ; CODE ⬇
 \ XREF: sub_402ACA+52
.text:00402C9C ⬇ ; ⬆
 \ sub_402ACA+64 ...
.text:00402C9C
.text:00402C9C arg_0 = dword ptr 8
.text:00402C9C
.text:00402C9C push ebp
.text:00402C9C mov ebp, esp
.text:00402C9F push ebx
.text:00402CA0 push esi
.text:00402CA1 push edi
.text:00402CA2 mov eax, dword_40D430
.text:00402CA7 imul eax, dword_40D440
.text:00402CAE add eax, dword_40D5C8
.text:00402CB4 mov ecx, 32000
.text:00402CB9 cdq
.text:00402CBA idiv ecx
.text:00402CBC mov dword_40D440, edx
.text:00402CC2 call _rand
.text:00402CC7 cdq
.text:00402CC8 idiv [ebp+arg_0]
.text:00402CBB mov dword_40D430, edx
.text:00402CD1 mov eax, dword_40D430
.text:00402CD6 jmp $+5
.text:00402CDB pop edi
.text:00402CDC pop esi
.text:00402CDD pop ebx
.text:00402CDE leave
.text:00402CDF retn
.text:00402CDF sub_402C9C endp
```

We'll call it “random”. Let's not to dive into this function's code yet.

This function is referred from 3 places.

Here are the first two:

```
.text:00402B16 mov eax, dword_40C03C ; 10 ⬇
 \ here
.text:00402B1B push eax
.text:00402B1C call random
.text:00402B21 add esp, 4
.text:00402B24 inc eax
.text:00402B25 mov [ebp+var_C], eax
```
Here is the third one:

```
.text:00402BBB mov eax, dword_40C058 ; 5
↓ here
.text:00402BC0 push eax
.text:00402BC1 call random
.text:00402BC6 add esp, 4
.text:00402BC9 inc eax
```

So the function has only one argument. 10 is passed in first two cases and 5 in third. We can also notice that the board has a size of 10*10 and there are 5 possible colors. This is it! The standard \texttt{rand()} function returns a number in the 0..0x7FFF range and this is often inconvenient, so many programmers implement their own random functions which returns a random number in a specified range. In our case, the range is 0..n – 1 and n is passed as the sole argument of the function. We can quickly check this in any debugger.

So let’s fix the third function call to always return zero. First, we will replace three instructions (PUSH/CALL/ADD) by NOPs. Then we’ll add XOR EAX, EAX instruction, to clear the EAX register.

```
00402BB8: 83C410 add esp,010
00402BBB: A158C04000 mov eax,[00040C058]
00402BC0: 31C0 xor eax,eax
00402BC2: 90 nop
00402BC3: 90 nop
00402BC4: 90 nop
00402BC5: 90 nop
00402BC6: 90 nop
00402BC7: 90 nop
00402BC8: 90 nop
00402BC9: 40 inc eax
00402BCA: 8B4DF8 mov ecx,[ebp][-8]
00402BCD: 8D0C49 lea ecx,[ecx][ecx]*2
00402BDE: 8B15F4D54000 mov edx,[00040D5F4]
```

So what we did is we replaced a call to the \texttt{rand()} function by a code which always returns zero.
Let’s run it now:

![Game Image]

Figure 75.2: Practical joke works

Oh yes, it works

But why are the arguments to the `random()` functions global variables? That’s just because it’s possible to change the board size in the game’s settings, so these values are not hardcoded. The 10 and 5 values are just defaults.

---

²Author of this book once did this as a joke for his coworkers with the hope that they would stop playing. They didn’t.
Chapter 76

Minesweeper (Windows XP)

For those who is not very good at playing Minesweeper, we could try to reveal the hidden mines in the debugger.

As we know, Minesweeper places mines randomly, so there has to be some kind of random number generator or a call to the standard rand() C-function. What is really cool about reversing Microsoft products is that there are PDB file with symbols (function names, etc). When we load winmine.exe into IDA, it downloads the PDB file exactly for this executable and shows all names.

So here it is, the only call to rand() is this function:

```
.text:01003940 ; __stdcall Rnd(x)  
.text:01003940 _Rnd@4 proc near  
        ↓ XREF: StartGame()+53  
.text:01003940  
        ↓ XREF: StartGame()+61  
.text:01003940  
.text:01003940 arg_0 = dword ptr 4  
.text:01003940  
.text:01003940  
.text:01003940 call ds:__imp__rand  
.text:01003940 cdq  
.text:01003947 idiv [esp+arg_0]  
.text:0100394B mov eax, edx  
.text:0100394D retn 4  
.text:0100394D _Rnd@4 endp
```

IDA named it so, and it was the name given to it by Minesweeper's developers.

The function is very simple:

```
int Rnd(int limit)
```
(There was no “limit” name in the PDB file; we manually named this argument like this.)

So it returns a random value from 0 to a specified limit.

`Rnd()` is called only from one place, a function called `StartGame()`, and as it seems, this is exactly the code which place the mines:

```
.text:010036C7     push    _xBBoxMac
.text:010036CD    call    _Rnd@4      ; Rnd(x)
.text:010036D2    push    _yBoxMac
.text:010036D8    mov     esi, eax
    inc     esi
.text:010036DA    call    _Rnd@4      ; Rnd(x)
.text:010036DB    inc     eax
.text:010036E0    mov     ecx, eax
.text:010036E1    shl     ecx, 5     ; ECX=⤦ \rcurvearrowse ECX*32
.text:010036E6    test    _rgBlk[ecx+esi], 80h
    jnz    short loc_10036C7
.text:010036EE    shl     eax, 5     ; EAX=⤦ \rcurvearrowse EAX*32
.text:010036F3    lea     eax, _rgBlk[eax+esi]
.text:010036FA    or      byte ptr [eax], 80h
.text:010036FD    dec     _cBombStart
.text:01003703    jnz    short loc_10036C7
```

Minesweeper allows you to set the board size, so the X (xBoxMac) and Y (yBoxMac) of the board are global variables. They are passed to `Rnd()` and random coordinates are generated. A mine is placed by the OR instruction at 0x010036FA. And if it was placed before (it’s possible if the pair of Rnd() generates a coordinates pair which was already was generated), then TEST and JNZ at 0x010036E6 jumps to the generation routine again.

cBombStart is the global variable containing total number of mines. So this is loop.

The width of the array is 32 (we can conclude this by looking at the SHL instruction, which multiplies one of the coordinates by 32).

The size of the rgBlk global array can be easily determined by the difference between the rgBlk label in the data segment and the next known one. It is 0x360 (864):
\[
864/32 = 27.
\]

So the array size is \(27 \times 32\)? It is close to what we know: when we try to set board size to \(100 \times 100\) in Minesweeper settings, it fallbacks to a board of size \(24 \times 30\). So this is the maximal board size here. And the array has a fixed size for any board size.

So let’s see all this in OllyDbg. We will ran Minesweeper, attaching OllyDbg to it and now we can see the memory dump at the address of the \(\text{rgBlk}\) array (0x01005340).

So we got this memory dump of the array:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>01005340</td>
<td>10 10 10 10</td>
</tr>
<tr>
<td>01005350</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005360</td>
<td>10 0F 0F 0F</td>
</tr>
<tr>
<td>01005370</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005380</td>
<td>10 0F 0F 0F</td>
</tr>
<tr>
<td>01005390</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>010053A0</td>
<td>10 0F 0F 0F</td>
</tr>
<tr>
<td>010053B0</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>010053C0</td>
<td>10 0F 0F 0F</td>
</tr>
<tr>
<td>010053D0</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>010053E0</td>
<td>10 0F 0F 0F</td>
</tr>
<tr>
<td>010053F0</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005400</td>
<td>10 0F 0F 8F</td>
</tr>
<tr>
<td>01005410</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005420</td>
<td>10 0F 0F 0F</td>
</tr>
<tr>
<td>01005430</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005440</td>
<td>10 0F 0F 8F</td>
</tr>
<tr>
<td>01005450</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005460</td>
<td>10 0F 0F 0F</td>
</tr>
<tr>
<td>01005470</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005480</td>
<td>10 10 10 10</td>
</tr>
<tr>
<td>01005490</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>010054A0</td>
<td>0F 0F 0F 0F</td>
</tr>
<tr>
<td>010054B0</td>
<td>0F 0F 0F 0F</td>
</tr>
</tbody>
</table>

\(^1\) All addresses here are for Minesweeper for Windows XP SP3 English. They may differ for other service packs.
OllyDbg, like any other hexadecimal editor, shows 16 bytes per line. So each 32-byte array row occupies exactly 2 lines here.

This is beginner level (9*9 board).

There is some square structure can be seen visually (0x10 bytes).

We will click “Run” in OllyDbg to unfreeze the Minesweeper process, then we’ll clicked randomly at the Minesweeper window and trapped into mine, but now all mines are visible:

![Minesweeper Game](image)

Figure 76.1: Mines

By comparing the mine places and the dump, we can conclude that 0x10 stands for border, 0x0F—empty block, 0x8F—mine.

Now we’ll add comments and also enclose all 0x8F bytes into square brackets:

<table>
<thead>
<tr>
<th>border:</th>
</tr>
</thead>
<tbody>
<tr>
<td>01005340 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005350 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>line #1:</td>
</tr>
<tr>
<td>01005360 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005370 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>line #2:</td>
</tr>
<tr>
<td>01005380 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>01005390 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>line #3:</td>
</tr>
<tr>
<td>010053A0 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>010053B0 0F 0F 0F 0F 0F 0F[8F]0F 10 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>line #4:</td>
</tr>
<tr>
<td>010053C0 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>010053D0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>line #5:</td>
</tr>
<tr>
<td>010053E0 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>010053F0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
</tbody>
</table>
Now we'll remove all *border bytes* (0x10) and what's beyond those:

<table>
<thead>
<tr>
<th>0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>0F 0F 0F 0F 0F 0F 0F[8F]0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>0F 0F[8F]0F[8F]0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>[8F]0F 0F 0F[8F]0F 0F 0F 0F 0F 0F</td>
</tr>
<tr>
<td>[8F]0F 0F 0F 0F[8F]0F 0F[8F]0F</td>
</tr>
</tbody>
</table>

Yes, these are mines, now it can be clearly seen and compared with the screenshot.
What is interesting is that we can modify the array right in OllyDbg. We can remove all mines by changing all 0x8F bytes by 0x0F, and here is what we’ll get in Minesweeper:

![Figure 76.2: All mines are removed in debugger](image)

We can also move all of them to the first line:

![Figure 76.3: Mines set in debugger](image)

Well, the debugger is not very convenient for eavesdropping (which was our goal anyway), so we’ll write a small utility to dump the contents of the board:

```c
// Windows XP MineSweeper cheater
// written by dennis(a)yurichev.com for http://beginners.re/  
⤦
#include <windows.h>
#include <assert.h>
#include <stdio.h>

int main (int argc, char * argv[]) {
    int i, j;
    HANDLE h;
    DWORD PID, address, rd;
```
BYTE board[27][32];

if (argc!=3)
{
    printf ("Usage: %s <PID> <address>\n", argv[0]);
    return 0;
}

assert (argv[1]!=NULL);
assert (argv[2]!=NULL);
assert (sscanf (argv[1], "%d", &PID)==1);
assert (sscanf (argv[2], "%x", &address)==1);

h=OpenProcess (PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE, FALSE, PID);

if (h==NULL)
{
    DWORD e=GetLastError();
    printf ("OpenProcess error: %08X\n", e);
    return 0;
}

if (ReadProcessMemory (h, (LPVOID)address, board, sizeof(board), &rd)!=TRUE)
{
    printf("ReadProcessMemory() failed\n");
    return 0;
}

for (i=1; i<26; i++)
{
    if (board[i][0]==0x10 && board[i][1]==0x10)
        break; // end of board
    for (j=1; j<31; j++)
    {
        if (board[i][j]==0x10)
            break; // board border
        if (board[i][j]==0x8F)
            printf ("*");
        else
            printf (" ");
    }
    printf ("\n");
Just set the PID\(^2\) and the address of the array (0x01005340 for Windows XP SP3 English) and it will dump it\(^4\).

It attaches itself to a win32 process by PID and just reads process memory an the address.

### 76.1 Exercises

- Why do the border bytes (0x10) exist in the array? What they are for if they are not visible in Minesweeper’s interface? How could it work without them?
- As it turns out, there are more values possible (for open blocks, for flagged by user, etc). Try to find the meaning of each one.
- Modify my utility so it can remove all mines or set them in a fixed pattern that you want in the Minesweeper process currently running.
- Modify my utility so it can work without the array address specified and without a PDB file. Yes, it’s possible to find board information in the data segment of Minesweeper’s running process automatically.

---

\(^2\)Program/process ID

\(^3\)PID it can be seen in Task Manager (enable it in “View → Select Columns”)

\(^4\)The compiled executable is here: beginners.re
Chapter 77

Hand decompiling + Z3 SMT solver

Amateur cryptography is usually (unintentionally) very weak and can be broken easily—for cryptographers, of course.

But let’s pretend we are not among these crypto-professionals.

Here is one-way hash function (read more about them: 34 on page 625), that converted a 64-bit value to another and we need to try to reverse its flow back.

77.1 Hand decompiling

Here its assembly language listing in IDA:

```
sub_401510 proc near
    ; ECX = input
    mov    rdx, 5D7E0D1F2E0F1F84h
    mov    rax, rcx          ; input
    imul   rax, rdx
    mov    rdx, 388D76AEE8CB1500h
    mov    ecx, eax
    and    ecx, 0Fh
    ror    rax, cl
    xor    rax, rdx
    mov    rdx, 0D2E9EE7E83C4285Bh
    mov    ecx, eax
    and    ecx, 0Fh
    rol    rax, cl
```
The example was compiled by GCC, so the first argument is passed in ECX.

If you don’t have Hex-Rays or if you distrust to it, you can try to reverse this code manually. One method is to represent the CPU registers as local C variables and replace each instruction by a one-line equivalent expression, like:

```c
uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;

    ecx=input;

    rdx=0x5D7E0D1F2E0F1F84;
    rax=rcx;
    rax*=rdx;
    rdx=0x388D76AEE8CB1500;
    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=rdx;
    rdx=0xD2E9EE7E83C4285B;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+rdx;
    rdx=0x8888888888888889;
    rax=r8;
    rax*=rdx;
    rdx=rdx>>5;
    rax=rdx;
    rcx=r8+rdx*4;
    rax=rcx-rax;
    rax=r8
    rax=_lrotl (rax, rcx&0xFF); // rotate left

    return rax;
}
```
If you are careful enough, this code can be compiled and will even work in the same way as the original.

Then, we are going to rewrite it gradually, keeping in mind all registers usage. Attention and focus is very important here—any tiny typo may ruin all your work!

Here is the first step:

```c
uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;
    ecx=input;

    rdx=0x5D7E0D1F2E0F1F84;
    rax=rcx;
    rax*=rdx;
    rdx=0x388D76AEE8CB1500;
    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=rdx;
    rdx=0xD2E9EE7E83C4285B;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+rdx;

    rdx=0x8888888888888889;
    rax=r8;
    rax*=rdx;
    // RDX here is a high part of multiplication result
    rdx=rdx>>5;
    // RDX here is division result!
    rax=rdx;

    rcx=r8+rdx*4;
    rax=rcx<<6;
    rcx=rcx-rax;
    rax=r8
    rax=_lrotl (rax, rcx&0xFF); // rotate left
    return rax;
};
```

Next step:

```c
uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;
    ecx=input;

    rdx=0x5D7E0D1F2E0F1F84;
```
We can spot the division using multiplication ([41 on page 676](#)). Indeed, let's calculate the divider in Wolfram Mathematica:

Listing 77.1: Wolfram Mathematica

```mathematica
In[1]:=N[2^(64 + 5)/16^^8888888888888889]
Out[1]:=60.
```

We get this:

```plaintext
uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;

    ecx=input;

    rdx=0x5D7E0D1F2E0F1F84;
    rax=rcx;
    rax*=rdx;
    rdx=0x388D76AEE8CB1500;
    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=rdx;
    rdx=0xD2E9EE7E83C4285B;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+rdx;

    rdx=0x8888888888888889;
    rax=r8;
    rax*=rdx;
    // RDX here is a high part of multiplication result
    rdx=rdx>>5;
    // RDX here is division result!
    rax=rdx;

    rcx=(r8+rdx*4)-(rax<<6);
    rax=r8
    rax=_lrotl (rax, rcx&0xFF); // rotate left
    return rax;
}
```
rax=rdx=r8/60;
rcx=(r8+rax*4)-(rax*64);
rax=r8
rax=_lrotl (rax, rcx&0xFF); // rotate left
return rax;
}

One more step:

```c
uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;

    rax=input;
    rax*=0x5D7E0D1F2E0F1F84;
    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=0x388D76AEE8CB1500;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+0xD2E9EE7E83C4285B;

    rcx=r8-(r8/60)*60;
rax=r8
    rax=_lrotl (rax, rcx&0xFF); // rotate left
    return rax;
}
```

By simple reducing, we finally see that it’s calculating the remainder, not the quotient:

```c
uint64_t f(uint64_t input)
{
    uint64_t rax, rbx, rcx, rdx, r8;

    rax=input;
    rax*=0x5D7E0D1F2E0F1F84;
    rax=_lrotr(rax, rax&0xF); // rotate right
    rax^=0x388D76AEE8CB1500;
    rax=_lrotl(rax, rax&0xF); // rotate left
    r8=rax+0xD2E9EE7E83C4285B;

    return _lrotl (r8, r8 % 60); // rotate left
}
```

We end up with this fancy formatted source-code:
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <intrin.h>

#define C1 0x5D7E0D1F2E0F1F84
#define C2 0x388D76AEE8CB1500
#define C3 0xD2E9EE7E83C4285B

uint64_t hash(uint64_t v)
{
    v*=C1;
    v=_lrotr(v, v&0xF); // rotate right
    v^=C2;
    v=_lrotl(v, v&0xF); // rotate left
    v+=C3;
    v=_lrotl(v, v % 60); // rotate left
    return v;
}

int main()
{
    printf("%llu\n", hash(...));
}

Since we are not cryptoanalysts we can’t find an easy way to generate the input value for some specific output value. The rotate instruction’s coefficients look frightening—it’s a warranty that the function is not bijective, it has collisions, or, speaking more simply, many inputs may be possible for one output.

Brute-force is not solution because values are 64-bit ones, that’s beyond reality.

77.2 Now let’s use the Z3 SMT solver

Still, without any special cryptographic knowledge, we may try to break this algorithm using the excellent SMT solver from Microsoft Research named Z3\(^1\). It is in fact theorem prover, but we are going to use it as SMT solver. Simply said, we can think about it as a system capable of solving huge equation systems.

Here is the Python source code:

```python
from z3 import *
```

\(^1\)http://go.yurichev.com/17314
This is going to be our first solver.

We see the variable definitions on line 7. These are just 64-bit variables. \texttt{i1}..\texttt{i6} are intermediate variables, representing the values in the registers between instruction executions.

Then we add the so-called constraints on lines 10..15. The last constraint at 17 is the most important one: we are going to try to find an input value for which our algorithm will produce 10816636949158156260.

Essentially, the SMT-solver searches for (any) values that satisfies all constraints. RotateRight, RotateLeft, URem— are functions from the Z3 Python API, not related to Python PL.

Then we run it:

```
...>python.exe 1.py
sat
[i1 = 3959740824832824396,
i3 = 8957124831728646493,
i5 = 10816636949158156260,
inp = 1364123924608584563,
outp = 10816636949158156260,
i4 = 14065440378185297801,
i2 = 4954926323707358301]
```
“sat” mean “satisfiable”, i.e., the solver was able to found at least one solution. The solution is printed in the square brackets. The last two lines are the input/output pair in hexadecimal form. Yes, indeed, if we run our function with 0x12EE577B63E80B73 as input, the algorithm will produce the value we were looking for.

But, as we noticed before, the function we work with is not bijective, so there may be other correct input values. The Z3 SMT solver is not capable of producing more than one result, but let’s hack our example slightly, by adding line 19, which implies “look for any other results than this”:

```
from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0x82E9E7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))
s.add(outp==10816636949158156260)
s.add(inp!=0x12EE577B63E80B73)

print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

Indeed, it finds another correct result:

```
../python.exe 2.py
sat
[i1 = 3959740824832824396,
i3 = 8957124831728646493,
```
This can be automated. Each found result can be added as a constraint and then the next result will be searched for. Here is a slightly more sophisticated example:

```python
from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))
s.add(outp==10816636949158156260)

result=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m[inp]
        result.append(m)
        # Create a new constraint the blocks the current model
        block = []
        for d in m:
            # d is a declaration
            if d.arity() > 0:
                raise Z3Exception("uninterpreted functions are not supported")
            # create a constant from declaration
            c=d()
```

```
if is_array(c) or c.sort().kind() == Z3_UNINTERPRETED_SORT:
    raise Z3Exception("arrays and uninterpreted sorts are not supported")
    block.append(c != m[d])
    s.add(Or(block))
else:
    print "results total=",len(result)
    break

We got:

1364123924608584563
1234567890
9223372038089343698
4611686019661955794
13835058056516731602
3096040143925676201
12319412180780452009
7707726162353064105
16931098199207839913
1906652839273745429
11130024876128521237
1574171089455909141
651833885770113333
5975809943035972467
15199181979890748275
1058749596146330371
results total= 16

So there are 16 correct input values for 0x92EE577B63E80B73 as a result.
The second is 1234567890—it is indeed the value which was used originally while preparing this example.

Let's also try to research our algorithm a bit more. Acting on a sadistic whim, let's find if the there are any possible input/output pairs in which the lower 32-bit parts are equal to each other?

Let's remove the `outp` constraint and add another, at line 17:

```python
from z3 import *
C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B
inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
```
It is indeed so:

```
sat
[ i1 = 14869545517796235860, 
  i3 = 83881713358288825253, 
  i5 = 6918262285561543945, 
  inp = 1370377541658871093, 
  outp = 1454318035175420856, 
  i4 = 5541032613289652645] 
inp=0x13048F1D12C00535
outp=0xC9D3C17A12C00535
```

Let's be more sadistic and add another constraint: last the 16 bits must be 0x1234:

```
from z3 import * 
C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B
inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))
```
s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
s.add(outp & 0xFFFF == 0x1234)

print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())

Oh yes, this possible as well:

```
sat
[i1 = 2834222860503985872,
i3 = 2294680776671411152,
i5 = 17492621421353821227,
inp = 461881484695179828,
outp = 419247225543463476,
i4 = 2294680776671411152,
i2 = 2834222860503985872]
inp=0x668EEC35F961234
outp=0x5D177215F961234
```

Z3 works very fast and it implies that the algorithm is weak, it is not cryptographic at all (like the most of the amateur cryptography).

Is it possible to tackle real cryptography by these methods? Real algorithms like AES, RSA, etc, can also be represented as huge system of equations, but these are so huge that they are impossible to work with on computers, now or in the near future. Of course, cryptographers are aware of this.

Summarizing, when dealing with amateur crypto, it’s a very good idea to try a SMT/SAT solver (like Z3).

Another article about Z3 is [Yur12].
Chapter 78

Dongles

Author of these lines, occasionally did software copy-protection dongle replacements, or “dongle emulators” and here are couple examples of how it’s happening.
About one of the cases that is not present here, you can read here: [Yur12].

78.1 Example #1: MacOS Classic and PowerPC

Here is an example of a program for MacOS Classic ¹, for PowerPC. The company who developed the software product has disappeared a long time ago, so the (legal) customer was afraid of physical dongle damage.
While running without a dongle connected, a message box with the text "Invalid Security Device" appeared. Luckily, this text string could easily be found in the executable binary file.
Let’s pretend we are not very familiar both with Mac OS Classic and PowerPC, but will try anyway.
IDA opened the executable file smoothly, reported its type as "PEF (Mac OS or Be OS executable)” (indeed, it is a standard Mac OS Classic file format).
By searching for the text string with the error message, we’ve got into this code fragment:

```
...  
seg000:000C87FC 38 60 00 01 li %r3, 1  
seg000:000C8800 48 03 93 41 bl check1
```

¹pre-UNIX MacOS
Yes, this is PowerPC code. The CPU is a very typical 32-bit RISC of 1990s era. Each instruction occupies 4 bytes (just as in MIPS and ARM) and the names somewhat resemble MIPS instruction names.

check1() is a function name we’ll give to it later. BL is Branch Link instruction, e.g., intended for calling subroutines. The crucial point is the BNE instruction which jumps if the dongle protection check passes or not if an error occurs: then the address of the text string gets loaded into the r3 register for the subsequent passing into a message box routine.

From the [SK95] we will found out that the r3 register is used for return values (and r4, in case of 64-bit values).

Another yet unknown instruction is CLRLWI. From [IBM00] we’ll learn that this instruction does both clearing and loading. In our case, it clears the 24 high bits from the value in r3 and puts them in r0, so it is analogical to MOVZ in x86 (15.1.1 on page 276), but it also sets the flags, so BNE can check them afterwards.

Let’s take a look into the check1() function:

```
seg000:00101B40 check1: # CODE XREF: seg000:00063
    E7Cp
seg000:00101B40 # sub_64070+160p ...
seg000:00101B40 .set arg_8, 8
seg000:00101B40 7C 08 02 A6 mflr %r0
seg000:00101B44 90 01 00 08 stw %r0, arg_8(%sp)
seg000:00101B48 94 21 FF C0 stwu %sp, -0x40(%sp)
seg000:00101B4C 48 01 6B 39 bl check2
seg000:00101B50 60 00 00 00 nop
seg000:00101B54 38 21 00 40 addi %sp, %sp, 0x40
seg000:00101B58 7C 08 03 A6 mtlr %r0
seg000:00101B5C 4E 80 00 20 blr
seg000:00101B60 # End of function check1
```
As you can see in IDA, that function is called from many places in the program, but only the r3 register’s value is checked after each call. All this function does is to call the other function, so it is a thunk function: there are function prologue and epilogue, but the r3 register is not touched, so check1() returns what check2() returns.

BLR² looks like the return from the function, but since IDA does the function layout, we probably do not need to care about this. Since it is a typical RISC, it seems that subroutines are called using a link register, just like in ARM.

The check2() function is more complex:

```
seg000:00118684  check2: # CODE XREF: check1+Cp
seg000:00118684seg000:00118684
seg000:00118684 .set var_18, -0x18
seg000:00118684 .set var_C, -0xC
seg000:00118684 .set var_8, -8
seg000:00118684 .set var_4, -4
seg000:00118684 .set arg_8, 8
seg000:00118684 93 E1 FF FC stw %r31, var_4(%sp)
seg000:00118688 7C 08 02 A6 mflr %r0
seg000:0011868C 83 E2 95 A8 lwz %r31, off_1485E8 # ▬
dword_24B704
seg000:00118690 .using dword_24B704, %r31
seg000:00118690seg000:00118690
seg000:00118690 93 C1 FF F8 stw %r30, var_8(%sp)
seg000:00118694 93 A1 FF F4 stw %r29, var_C(%sp)
seg000:00118698 7C 7D 1B 78 mr %r29, %r3
seg000:0011869C 90 01 00 08 stw %r0, arg_8(%sp)
seg000:001186A0 54 60 06 3E clrlwi %r0, %r3, 24
seg000:001186A4 28 00 00 01 cmplwi %r0, 1
seg000:001186A8 94 21 FF B0 stwu %sp, -0x50(%sp)
seg000:001186AC 40 82 00 0C bne loc_1186B8
seg000:001186B0 3B C0 06 3F clrlwi. %r0, %r30, 24
seg000:001186B4 48 00 00 6C b exit
seg000:001186B8
seg000:001186B8 48 00 03 D5 bl sub_118A8C
seg000:001186BC 60 00 00 00 nop
seg000:001186C0 3B C0 00 00 li %r30, 0
seg000:001186C4
seg000:001186C4 57 C0 06 3F clrlwi. %r0, %r30, 24
seg000:001186C8 41 82 00 18 beq loc_1186E0
seg000:001186CC 38 61 00 38 addi %r3, %sp, 0x50+var_18
seg000:001186D0 80 9F 00 00 lwz %r4, dword_24B704
seg000:001186D4 48 00 C0 55 bl .RBEFINDNEXT
```

²(PowerPC) Branch to Link Register
We are lucky again: some function names are left in the executable (debug symbols section? Hard to say while we are not very familiar with the file format, maybe it is some kind of PE exports? ([68.2.7 on page 993]), like .RBEFINDNEXT() and .RBEFINDFIRST(). Eventually these functions call other functions with names like .GetNextDeviceViaUSB(), .USBSendPKT(), so these are clearly dealing with an USB device.

There is even a function named .GetNextEve3Device()—sounds familiar, there was a Sentinel Eve3 dongle for ADB port (present on Macs) in 1990s.

Let's first take a look on how the r3 register is set before return, while ignoring
everything else. We know that a “good” r3 value has to be non-zero, zero r3 leads the execution flow to the message box with an error message.

There are two li %r3, 1 instructions present in the function and one li %r3, 0 (Load Immediate, i.e., loading a value into a register). The first instruction is at 0x001186B0—and frankly speaking, it’s hard to say what it means.

What we see next is, however, easier to understand: .RBEGINFIRST() is called: if it fails, 0 is written into r3 and we jump to exit, otherwise another function is called (check3())—if it fails too, .RBEGINNEXT() is called, probably in order to look for another USB device.

N.B.: clrlwi. %r0, %r3, 16 it is analogical to what we already saw, but it clears 16 bits, i.e., .RBEGINFIRST() probably returns a 16-bit value.

B (stands for branch) unconditional jump.

BEQ is the inverse instruction of BNE.

Let’s see check3():

```
seg000:0011873C       check3: # CODE XREF: check2+88p
seg000:0011873C .set var_18, -0x18
seg000:0011873C .set var_C, -0xC
seg000:0011873C .set var_8, -8
seg000:0011873C .set var_4, -4
seg000:0011873C .set arg_8, 8
seg000:0011873C 93 E1 FF FC stw %r31, var_4(%sp)
seg000:00118740 7C 08 02 A6 mflr %r0
seg000:00118744 38 A0 00 00 li %r5, 0
seg000:00118748 93 C1 FF F8 stw %r30, var_8(%sp)
seg000:0011874C 83 C2 95 A8 lwz %r30, off_1485E8 # dword_24B704
seg000:00118750 93 A1 FF FC stw %r31, var_4(%sp)
seg000:00118754 3B A3 00 00 addi %r29, %r3, 0
seg000:00118758 38 60 00 00 li %r3, 0
seg000:0011875C 90 01 00 08 stw %r0, arg_8(%sp)
seg000:00118760 94 21 FF B0 stwu %sp, -0x50(%sp)
seg000:00118764 80 DE 00 00 lwz %r6, dword_24B704
seg000:00118768 38 81 00 38 addi %r4, %sp, 0x50+var_18
seg000:0011876C 48 00 C0 5D bl .RBERead
seg000:00118770 60 00 00 00 nop
seg000:00118774 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118778 41 82 00 0C beq loc_118784
seg000:0011877C 38 60 00 00 li %r3, 0
seg000:00118780 48 00 02 F0 b exit
seg000:00118784
```
CHAPTER 78. DONGLES

```
BEGIN
seg000:00118784   loc_118784: # CODE XREF: check3+3Cj
seg000:00118784   A0 01 00 38  lhz  %r0, 0x50+var_18(%sp)
seg000:00118788   28 00 04 B2  cmplwi %r0, 0x1100
seg000:0011878C   41 82 00 0C  beq  loc_118798
seg000:00118790   38 60 00 00  li   %r3, 0
seg000:00118794   48 00 02 DC  b   exit

seg000:00118798
seg000:00118798   80 DE 00 00  lwz   %r6, dword_24B704
seg000:0011879C   38 81 00 38  addi  %r4, %sp, 0x50+var_18
seg000:001187A0   38 60 00 01  li   %r3, 1
seg000:001187A4   38 A0 00 00  li   %r5, 0
seg000:001187A8   48 00 C0 21  bl   .RBEREAD
seg000:001187AC   60 00 00 00  nop
seg000:001187B0   54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:001187B4   41 82 00 0C  beq  loc_1187C0
seg000:001187B8   38 60 00 00  li   %r3, 0
seg000:001187BC   48 00 02 B4  b   exit

seg000:001187C0
seg000:001187C0   A0 01 00 38  lhz  %r0, 0x50+var_18(%sp)
seg000:001187C4   28 00 06 4B  cmplwi %r0, %r3, 16
seg000:001187C8   41 82 00 0C  beq  loc_1187D4
seg000:001187CC   38 60 00 00  li   %r3, 0
seg000:001187D0   48 00 02 A0  b   exit

seg000:001187D4
seg000:001187D4   4B F9 F3 D9  bl   sub_B7BAC
seg000:001187D8   60 00 00 00  nop
seg000:001187DC   54 60 06 3E  clrlwi  %r0, %r3, 24
seg000:001187E0   2C 00 00 05  cmpwi  %r0, 5
seg000:001187E4   41 82 01 00  beq  loc_1188E4
seg000:001187E8   40 80 00 10  bge  loc_1187F8
seg000:001187EC   2C 00 00 04  cmpwi  %r0, 4
seg000:001187F0   40 80 00 58  bge  loc_118848
seg000:001187F4   48 00 01 8C  b   loc_118980

seg000:001187F8
seg000:001187F8   40 80 00 10  bge  loc_1187F8
seg000:001187F8   2C 00 00 0B  cmpwi  %r0, 0xB
seg000:001187FC   41 82 00 08  beq  loc_118804
seg000:00118800   48 00 01 80  b   loc_118980

seg000:00118804
seg000:00118804   80 DE 00 00  lwz   %r6, dword_24B704
seg000:00118808   38 81 00 38  addi  %r4, %sp, 0x50+var_18
seg000:0011880C   38 60 00 08  li   %r3, 8
seg000:00118810   38 A0 00 00  li   %r5, 0

```

1079
CHAPTER 78. DONGLES

seg000:00118814 48 00 BF B5 bl .RBEREAD
seg000:00118818 60 00 00 00 nop
seg000:0011881C 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118820 41 82 00 0C beq loc_11882C
seg000:00118824 38 60 00 00 li %r3, 0
seg000:00118828 48 00 02 48 b exit
seg000:0011882C
cmpng
seg000:0011882C loc_11882C: # CODE XREF: check3+E4j
seg000:0011882C A0 01 00 38 lhz %r0, 0x50+var_18(%sp)
seg000:00118830 28 00 11 30 cmplwi %r0, 0xFEAO
seg000:00118834 41 82 00 0C beq loc_118840
seg000:00118838 38 60 00 00 li %r3, 0
seg000:0011883C 48 00 02 34 b exit
seg000:00118840
seg000:00118840 loc_118840: # CODE XREF: check3+F8j
seg000:00118840 38 60 00 01 li %r3, 1
seg000:00118844 48 00 02 2C b exit
seg000:00118848
seg000:00118848 loc_118848: # CODE XREF: check3+B4j
seg000:00118848 80 DE 00 00 lwz %r6, dword_24B704
seg000:0011884C 38 81 00 38 addi %r4, %sp, 0x50+var_18
seg000:00118850 38 60 00 0A li %r3, 0xA
seg000:00118854 38 A0 00 00 li %r5, 0
seg000:00118858 48 00 BF 71 bl .RBEREAD
seg000:0011885C 60 00 00 00 nop
seg000:00118860 54 60 04 3F clrlwi. %r0, %r3, 16
seg000:00118864 41 82 00 0C beq loc_118870
seg000:00118868 38 60 00 00 li %r3, 0
seg000:0011886C 48 00 02 04 b exit
seg000:00118870
seg000:00118870 loc_118870: # CODE XREF: check3+128j
seg000:00118870 A0 01 00 38 lhz %r0, 0x50+var_18(%sp)
seg000:00118874 28 00 03 F3 cmplwi %r0, 0xA6E1
seg000:00118878 41 82 00 0C beq loc_118884
seg000:0011887C 38 60 00 00 li %r3, 0
seg000:00118880 48 00 01 F0 b exit
seg000:00118884
seg000:00118884 loc_118884: # CODE XREF: check3+13j
seg000:00118884 57 BF 06 3E clrlwi %r31, %r29, 24
seg000:00118888 28 1F 00 02 cmplwi %r31, 2
seg000:0011888C 40 82 00 0C bne loc_118898
seg000:00118890 38 60 00 01 li %r3, 1
seg000:00118894 48 00 01 DC b exit
seg000:00118898
seg000:00118898 loc_118898: # CODE XREF: check3+150j

1080
CHAPTER 78. DONGLES

seg000:00118920  loc_118920: # CODE XREF: check3+1
  ↓ D8j
seg000:00118920  57  BF  06  3E  clrlwi  %r31, %r29, 24
seg000:00118924  28  1F  00  02  cmplwi  %r31, 2
seg000:00118928  40  82  00  0C  bne  loc_118934
seg000:0011892C  38  60  00  01  li  %r3, 1
seg000:00118930  48  00  01  40  b  exit
seg000:00118934
seg000:00118934  loc_118934: # CODE XREF: check3+1
  ↓ ECj
seg000:00118934  80  DE  00  00  lwz  %r6, dword_24B704
seg000:00118938  38  81  00  38  addi  %r4, %sp, 0x50+var_18
seg000:0011893C  38  60  00  0D  li  %r3, 0xD
seg000:00118940  38  A0  00  00  li  %r5, 0
seg000:00118944  48  00  BE  85  bl  .RBEREAD
seg000:00118948  60  00  00  00  nop
seg000:0011894C  54  60  04  3F  clrlwi  %r0, %r3, 16
seg000:00118950  38  66  00  00  bne  loc_11895C
seg000:00118954  38  60  00  00  li  %r3, 0
seg000:00118958  48  00  01  18  b  exit
seg000:0011895C
seg000:0011895C  loc_11895C: # CODE XREF: check3+214
  ↓ j
seg000:0011895C  A0  01  00  38  lhz  %r0, 0x50+var_18(%sp)
seg000:00118960  28  00  07  CF  cmplwi  %r0, 0xFC7
seg000:00118964  41  82  00  0C  beq  loc_118970
seg000:00118968  38  60  00  00  li  %r3, 0
seg000:0011896C  48  00  01  04  b  exit
seg000:00118970
seg000:00118970  loc_118970: # CODE XREF: check3+228
  ↓ j
seg000:00118970  28  1F  00  03  cmplwi  %r31, 3
seg000:00118974  40  82  00  F8  bne  error
seg000:00118978  38  60  00  01  li  %r3, 1
seg000:0011897C  48  00  00  F4  b  exit
seg000:00118980
seg000:00118980  loc_118980: # CODE XREF: check3+B8j
seg000:00118980  # check3+C4j
seg000:00118980  80  DE  00  00  lwz  %r6, dword_24B704
seg000:00118984  38  81  00  38  addi  %r4, %sp, 0x50+var_18
seg000:00118988  3F  E0  00  00  li  %r31, 0
seg000:0011898C  38  60  00  04  li  %r3, 4
seg000:00118990  38  A0  00  00  li  %r5, 0
seg000:00118994  48  00  BE  35  bl  .RBEREAD
seg000:00118998  60  00  00  00  nop
seg000:0011899C  54  60  04  3F  clrlwi  %r0, %r3, 16
seg000:001189A0  41  82  00  0C  beq  loc_1189AC
seg000:00118A20
seg000:00118A20  loc_118A20: # CODE XREF: check3+2
\ D8j
seg000:00118A20 A0 01 00 38  lhz  %r0, 0x50+var_18(%sp)
seg000:00118A24 28 00 11 D3  cmplwi  %r0, 0xD300
seg000:00118A28 40 82 00 0C  bne  loc_118A34
seg000:00118A2C 3B E0 00 01  li  %r31, 1
seg000:00118A30 48 00 00 14  b   good1
seg000:00118A34
seg000:00118A34  loc_118A34: # CODE XREF: check3+2
\ ECj
seg000:00118A34 28 00 1A EB  cmplwi  %r0, 0xEBA1
seg000:00118A38 41 82 00 0C  beq  good1
seg000:00118A3C 38 60 00 00  li  %r3, 0
seg000:00118A40 48 00 00 30  b   exit
seg000:00118A44
seg000:00118A44  good1:    # CODE XREF: check3+2
\ F4j
seg000:00118A44 57 A0 06 3E  clrlwi  %r0, %r29, 24
seg000:00118A48 28 00 00 03  cmplwi  %r0, 3
seg000:00118A4C 40 82 00 20  bne  error
seg000:00118A50 57 E0 06 3F  clrlwi. %r0, %r31, 24
seg000:00118A54 41 82 00 10  beq  good
seg000:00118A58 48 00 4B F5  bl   sub_11D64C
seg000:00118A5C 60 00 00 00  nop
seg000:00118A60 48 00 00 10  b   exit
seg000:00118A64
seg000:00118A64  good:     # CODE XREF: check3+318
\ j
seg000:00118A64 38 60 00 01  li  %r3, 1
seg000:00118A68 48 00 00 08  b   exit
seg000:00118A6C
seg000:00118A6C  error:    # CODE XREF: check3+19
\ Cj
seg000:00118A6C 38 60 00 00  li  %r3, 0
seg000:00118A70
seg000:00118A70  exit:     # CODE XREF: check3+44j
seg000:00118A70 80 01 00 58  lwz  %r0, 0x50+arg_8(%sp)
seg000:00118A74 38 21 00 50  addi  %sp, %sp, 0x50
seg000:00118A78 83 E1 FF FC  lwz  %r31, var_4(%sp)
seg000:00118A7C 7C 08 03 A6  mtlr  %r0
seg000:00118A80 83 C1 FF F8  lwz  %r30, var_8(%sp)
seg000:00118A84 83 A1 FF F4  lwz  %r29, var_C(%sp)
seg000:00118A88 4E 80 00 20  blr
There are a lot of calls to `.RBEREAD()`. The function probably returns some values from the dongle, so they are compared here with some hard-coded variables using CMPLWI.

We also see that the r3 register is also filled before each call to `.RBEREAD()` with one of these values: 0, 1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Probably a memory address or something like that?

Yes, indeed, by googling these function names it is easy to find the Sentinel Eve3 dongle manual!

Perhaps, we don’t even need to learn any other PowerPC instructions: all this function does is just call `.RBEREAD()`, compare its results with the constants and returns 1 if the comparisons are fine or 0 otherwise.

OK, all we’ve got is that check1() has always to return 1 or any other non-zero value. But since we are not very confident in our knowledge of PowerPC instructions, we are going to be careful: we will patch the jumps in check2() at 0x001186FC and 0x00118718.

At 0x001186FC we’ll write bytes 0x48 and 0 thus converting the `BEQ` instruction in an `B` (unconditional jump): We can spot its opcode in the code without even referring to [IBM00].

At 0x00118718 we’ll write 0x60 and 3 zero bytes, thus converting it to a `NOP` instruction: Its opcode we could spot in the code too.

And now it all works without a dongle connected.

In summary, such small modifications can be done with IDA and minimal assembly language knowledge.

### 78.2 Example #2: SCO OpenServer

An ancient software for SCO OpenServer from 1997 developed by a company that disappeared a long time ago.

There is a special dongle driver to be installed in the system, that contains the following text strings: “Copyright 1989, Rainbow Technologies, Inc., Irvine, CA” and “Sentinel Integrated Driver Ver. 3.0 

After the installation of the driver in SCO OpenServer, these device files appear in the /dev filesystem:

```plaintext
/dev/rbsl8
/dev/rbsl9
```
The program reports an error without dongle connected, but the error string cannot be found in the executables.

Thanks to IDA, it is easy to load the COFF executable used in SCO OpenServer.

Let's also try to find “rbsl” string and indeed, found it in this code fragment:

```
.text:00022AB8 public SSQC
.text:00022AB8 SSQC proc near ; CODE XREF: SSQ+7p
.text:00022AB8 var_44 = byte ptr -44h
.text:00022AB8 var_29 = byte ptr -29h
.text:00022AB8 arg_0 = dword ptr 8
.text:00022AB8
.text:00022AB8 push ebp
.text:00022AB9 mov ebp, esp
.text:00022ABB sub esp, 44h
.text:00022ABE push edi
.text:00022ABF mov edi, offset unk_4035D0
.text:00022AC4 push esi
.text:00022AC5 mov esi, [ebp+arg_0]
.text:00022AC8 push ebx
.text:00022AC9 push esi
.text:00022ACA call strlen
.text:00022ACF add esp, 4
.text:00022AD2 cmp eax, 2
.text:00022AD7 jnz loc_22BA4
.text:00022ADD inc esi
.text:00022ADE mov al, [esi-1]
.text:00022AE1 movsx eax, al
.text:00022AE4 cmp eax, '3'
.text:00022AE9 jz loc_22B84
.text:00022AEF cmp eax, '4'
.text:00022AF4 jz loc_22B94
.text:00022AFA cmp eax, '5'
.text:00022AFF jnz short loc_22B6B
.text:00022B01 movsx ebx, byte ptr [esi]
.text:00022B04 sub ebx, '0'
.text:00022B07 mov eax, 7
.text:00022B0C add eax, ebx
.text:00022B0E push eax
.text:00022B0F lea eax, [ebp+var_44]
.text:00022B12 push offset aDevSlD ; "/dev/sl%d"
.text:00022B17 push eax
.text:00022B18 call nl_sprintf
.text:00022B1D push 0  ; int
```
push offset aDevRbsl8 ; char *
call _access
add esp, 14h
cmp eax, 0FFFFFFFFh
jz short loc_22B48
lea eax, [ebx+7]
push eax
lea eax, [ebp+var_44]
push eax
push offset aDevRbslD ; "/dev/rbsl%d"
push eax
call nl_sprintf
add esp, 0Ch

loc_22B48: ; CODE XREF: SSQC+79j
mov edx, [edi]
test edx, edx
jle short loc_22B57
push edx ; int
call _close
add esp, 4
loc_22B57: ; CODE XREF: SSQC+94j
push 2 ; int
lea eax, [ebp+var_44]
push eax ; char *
call _open
add esp, 8
test eax, eax
mov [edi], eax
jge short loc_22B78
loc_22B6B: ; CODE XREF: SSQC+47j
mov eax, 0FFFFFFFFh
pop ebx
pop esi
pop edi
mov esp, ebp
pop ebp
retn
loc_22B78: ; CODE XREF: SSQC+B1j
pop ebx
pop esi
pop edi
xor eax, eax
mov esp, ebp
pop ebp
Yes, indeed, the program needs to communicate with the driver somehow.

The only place where the SSQC() function is called is the thunk function:

```
.text:0000DBE8    public SSQ
.text:0000DBE8    SSQ proc near ; CODE XREF: sys_info+A9p
.text:0000DBE8    ; sys_info+CBp ...
.text:0000DBE8
.text:0000DBE8    arg_0 = dword ptr 8
.text:0000DBE8
.text:0000DBE8    push ebp
.text:0000DBE9    mov ebp, esp
.text:0000DBEB    mov edx, [ebp+arg_0]
.text:0000DBEB    push edx
.text:0000DBEF    call SSQC
.text:0000DBF4    add esp, 4
.text:0000DBF7    mov esp, ebp
.text:0000DBF9    pop ebp
.text:0000DBFA    retn
```

SSQC() is called from at least 2 functions.

One of these is:

```
data:0040169C  _51_52_53 dd offset aPressAnyKeyT_0 ; DATA
  ↵  XREF: init_sys+392r
.data:0040169C  
  ↵  sys_info+A1r
.data:0040169C  ; ↵ ANY KEY TO CONTINUE: "
.data:004016A0  dd offset a51 ; "51"
.data:004016A4  dd offset a52 ; "52"
.data:004016A8  dd offset a53 ; "53"
```
... 
.data:004016B8 _3C_or_3E dd offset a3c ; DATA ⤪ XREF: sys_info:loc_D67Br 
.data:004016B8 ; "3C" 
.data:004016BC dd offset a3e ; "3E" 

; these names we gave to the labels: 
.data:004016C0 answers1 dd 6B05h ; DATA ⤪ XREF: sys_info+E7r 
.data:004016C4 dd 3D87h 
.data:004016C8 answers2 dd 3Ch ; DATA ⤪ XREF: sys_info+F2r 
.data:004016CC dd 832h 
.data:004016D0 _C_and_B db 0Ch ; DATA ⤪ sys_info:OKr 
.data:004016D1 byte_4016D1 db 0Bh ; DATA ⤪ XREF: sys_info+FDr 
.data:004016D2 db 0 

...

.text:0000D652 xor eax, eax 
.text:0000D654 mov al, ds:ctl_port 
.text:0000D659 mov ecx, _51_52_53[eax*4] 
.text:0000D660 push ecx 
.text:0000D661 call SSQ 
.text:0000D666 add esp, 4 
.text:0000D669 cmp eax, 0FFFFFFFh 
.text:0000D66E jz short loc_D6D1 
.text:0000D670 xor ebx, ebx 
.text:0000D672 mov al, _C_and_B 
.text:0000D677 test al, al 
.text:0000D679 jz short loc_D6C0 
.text:0000D67B loc_D67B: ; CODE XREF: sys_info+106j 
.text:0000D67B mov eax, _3C_or_3E[ebx*4] 
.text:0000D682 push eax 
.text:0000D683 call SSQ 
.text:0000D688 push offset a4g ; "4G" 
.text:0000D68D call SSQ 
.text:0000D692 push offset a0123456789 ; ⤪ "0123456789" 
.text:0000D697 call SSQ
“3C” and “3E” sound familiar: there was a Sentinel Pro dongle by Rainbow with no memory, providing only one crypto-hashing secret function.
You can read a short description of what hash function is here: 34 on page 625.

But let’s get back to the program. So the program can only check the presence or absence of a connected dongle. No other information can be written to such dongle, as it has no memory. The two-character codes are commands (we can see how the commands are handled in the SSQC() function) and all other strings are hashed inside the dongle, being transformed into a 16-bit number. The algorithm was secret, so it was not possible to write a driver replacement or to remake the dongle hardware that would emulate it perfectly. However, it is always possible to intercept all accesses to it and to find what constants the hash function results are compared to. But we need to say that it is possible to build a robust software copy protection scheme based on secret cryptographic hash-function: let it encrypt/decrypt the data files your software uses.

But let’s get back to the code.

Codes 51/52/53 are used for LPT printer port selection. 3x/4x are used for “family” selection (that’s how Sentinel Pro dongles are differentiated from each other: more than one dongle can be connected to a LPT port).

The only non-2-character string passed to the hashing function is "0123456789". Then, the result is compared against the set of valid results. If it is correct, 0xC or 0xB is to be written into the global variable ctl_model.

Another text string that gets passed is "PRESS ANY KEY TO CONTINUE: ", but the result is not checked. Hard to say why, probably by mistake 3.

Let’s see where the value from the global variable ctrl_mode is used.

One such place is:

```
.text:0000D708 prep_sys proc near ; CODE XREF: init_sys+46Ap
.text:0000D708
.text:0000D708 var_14 = dword ptr -14h
.text:0000D708 var_10 = byte ptr -10h
.text:0000D708 var_8 = dword ptr -8
.text:0000D708 var_2 = word ptr -2
.text:0000D708
.text:0000D708 push ebp
.text:0000D709 mov eax, ds:net_env
.text:0000D70E mov ebp, esp
.text:0000D710 sub esp, 1Ch
.text:0000D713 test eax, eax
.text:0000D715 jnz short loc_D734
.text:0000D717 mov al, ds:ctl_model
.text:0000D71C test al, al
.text:0000D71E jnz short loc_D77E
```

3What a strange feeling: to find bugs in such ancient software.
If it is 0, an encrypted error message is passed to a decryption routine and printed.

The error string decryption routine seems a simple xorring:

```assembly
.err_warn proc near ; CODE
    var_55 = byte ptr -55h
    var_54 = byte ptr -54h
    arg_0 = dword ptr 8
    arg_4 = dword ptr 0Ch
    arg_8 = dword ptr 10h
    arg_C = dword ptr 14h

    push ebp
    mov ebp, esp
    sub esp, 54h
    push edi
    mov ecx, [ebp+arg_8] ; key
    xor edi, edi
    test ecx, ecx
    push esi
    jle short loc_A466
    mov esi, [ebp+arg_C] ; key
    mov edx, [ebp+arg_4] ; string
```

```assembly
    ClosSem
    startup_err
```
CHAPTER 78. DONGLES

.loc_A453: ; CODE ↗
 xor     eax, eax
 mov     al, [edx+edi]
 xor     eax, esi
 add     esi, 3
 inc     edi
 cmp     edi, ecx
 mov     [ebp+edi+var_55], al
 jl      short loc_A453

.loc_A466: ; CODE ↗
 mov     [ebp+edi+var_54], 0
 mov     eax, [ebp+arg_0]
 cmp     eax, 18h
 jnz     short loc_A49C
 lea     eax, [ebp+var_54]
 push    eax
 call    status_line
 add     esp, 4

.loc_A481: ; CODE ↗
 push    50h
 push    0
 lea     eax, [ebp+var_54]
 push    eax
 call    memset
 call    pcv_refresh
 add esp, 0Ch
 push    esi
 pop     edi
 mov     esp, ebp
 pop     ebp
 retn

.loc_A49C: ; CODE ↗
 push    0
 lea     edx, [ebp+var_54]
 mov     edx, [ebp+arg_0]
 push    edx
 push    eax
 call    pcv_lputs
 add esp, 0Ch
 jmp     short loc_A481
That’s why we were unable to find the error messages in the executable files, because they are encrypted (which is is popular practice).

Another call to the SSQ() hashing function passes the “offln” string to it and compares the result with 0xFE81 and 0x12A9. If they don’t match, it works with some timer() function (maybe waiting for a poorly connected dongle to be re-connected and check again?) and then decrypts another error message to dump.
.text:0000DAB4  jle  short loc_DA55
.text:0000DAB6  mov  eax, ds:net_env
.text:0000DABB  test  eax, eax
.text:0000DABD  jz  short error
...
.text:0000DAF7  error:  ; CODE  ↘
  ↗  XREF:  sync_sys+255j
.text:0000DAF7  ;  ↗
  ↗  sync_sys+274j  ...
.text:0000DAF7  mov  [ebp+var_8], offset  ↗
  ↗  encrypted_error_message2
.text:0000DAFE  mov  [ebp+var_C], 17h  ;  ↗
  ↗  decrypting key
.text:0000DB05  jmp  decrypt_end_print_message  ↗
  ↗  decrypt_end_print_message
...
;  this name we gave to label:
.text:0000D9B6  decrypt_end_print_message:  ; CODE  ↗
  ↗  XREF:  sync_sys+29Dj
.text:0000D9B6  ;  ↗
  ↗  sync_sys+2ABj
.text:0000D9B6  mov  eax, [ebp+var_18]
.text:0000D9B9  test  eax, eax
.text:0000D9BB  jnz  short loc_D9FB
.text:0000D9BD  mov  edx, [ebp+var_C];  key
.text:0000D9C0  mov  ecx, [ebp+var_8];  ↗
  ↗  string
.text:0000D9C3  push  edx
.text:0000D9C4  push  20h
.text:0000D9C6  push  ecx
.text:0000D9C7  push  18h
.text:0000D9C9  call  err_warn
.text:0000D9CE  push  0Fh
.text:0000D9D0  push  190h
.text:0000D9D5  call  sound
.text:0000D9DA  mov  [ebp+var_18], 1
.text:0000D9E1  add  esp, 18h
.text:0000D9E4  call  pcv_kbhit
.text:0000D9E9  test  eax, eax
.text:0000D9EB  jz  short loc_D9FB
...

1096
Bypassing the dongle is pretty straightforward: just patch all jumps after the relevant CMP instructions.

Another option is to write our own SCO OpenServer driver, containing a table of questions and answers, all of those which present in the program.

### 78.2.1 Decrypting error messages

By the way, we can also try to decrypt all error messages. The algorithm that is located in the `err_warn()` function is very simple, indeed:

```assembly
Listing 78.1: Decryption function
```

```assembly
.text:0000A44D mov esi, [ebp+arg_C] ; key
.text:0000A450 mov edx, [ebp+arg_4] ; string
.text:0000A453 loc_A453:
.text:0000A453 xor eax, eax
.text:0000A458 mov al, [edx+edi] ; load encrypted byte
.text:0000A458 xor eax, esi ; decrypt it
.text:0000A45A add esi, 3 ; change key for the next byte
.text:0000A45D inc edi
.text:0000A45E cmp edi, ecx
.text:0000A460 mov [ebp+edi+var_55], al
.text:0000A464 jl short loc_A453
```

As we can see, not just string is supplied to the decryption function, but also the key:

```assembly
.text:0000DAF7 error: ; CODE
\ XREF: sync_sys+255j
.text:0000DAF7 ;
\ sync_sys+274j ...
.text:0000DAF7 mov [ebp+var_8], offset
\ encrypted_error_message2
.text:0000DAFE mov [ebp+var_C], 17h ; \ decrypting key
```

1097
The algorithm is a simple xoring: each byte is xored with a key, but the key is increased by 3 after the processing of each byte.

We can write a simple Python script to check our hypothesis:

```
Listing 78.2: Python 3.x

#!/usr/bin/python
import sys

msg=[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47,
    0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
    0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A]

key=0x17
tmp=key
for i in msg:
    sys.stdout.write ("%c" % (i^tmp))
    tmp=tmp+3
sys.stdout.flush()
```

And it prints: “check security device connection”. So yes, this is the decrypted message.
 CHAPTER 78. DONGLES

There are also other encrypted messages with their corresponding keys. But needless to say, it is possible to decrypt them without their keys. First, we can see that the key is in fact a byte. It is because the core decryption instruction (XOR) works on byte level. The key is located in the ESI register, but only one byte part of ESI is used. Hence, a key may be greater than 255, but its value is always to be rounded.

As a consequence, we can just try brute-force, trying all possible keys in the 0..255 range. We are also going to skip the messages that contain unprintable characters.

Listing 78.3: Python 3.x

```python
#!/usr/bin/python
import sys, curses.ascii

msgs=[
    [0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47, 0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76, 0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A],
    [0x49, 0x65, 0x2D, 0x63, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0x5C, 8, 0x4F, 0x4B, 0x47, 0x5D, 0x54, 0x5F, 0x1D, 0x26, 0x2C, 0x33, 0x27, 0x28, 0x6F, 0x72, 0x75, 0x78, 0x7B, 0x7E, 0x41, 0x44],
    [0x45, 0x61, 0x31, 0x67, 0x72, 0x79, 0x68, 0x52, 0x4A, 0x52, 0x50, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x2B, 0x39, 0x39, 0x3C, 0x38, 0x79, 0x3A, 0x30, 0x17, 0x0B, 0x0C],
    [0x40, 0x64, 0x75, 0x7F, 0x6F, 0, 0x4C, 0x40, 9, 0x4D, 0x5A, 0x46, 0x5D, 0x57, 0x49, 0x57, 0x3B, 0x21, 0x23, 0x6A, 0x38, 0x23, 0x36, 0x24, 0x2A, 0x7C, 0x3A, 0x1A, 0x6, 0x0D, 0x0E, 0x0A, 0x14],
    [0x72, 0x7C, 0x72, 0x79, 0x76, 0x50, 0x43, 0x4A, 0x59, 0x5D, 0x5B, 0x41, 0x41, 0x1B, 0x5A, 0x24, 0x32, 0x2E, 0x29, 0x28, 0x70, 0x20, 0x22, 0x33, 0x36, 0x0D, 0x0B, 0x48, 0x44, 0x4E]]
```
def is_string_printable(s):
    return all(list(map(lambda x: curses.ascii.isprint(x), s)))

cnt=1
for msg in msgs:
    print ("message #%d" % cnt)
    for key in range(0,256):
        result=[]
tmp=key
        for i in msg:
            result.append (i^tmp)
tmp=tmp+3
        if is_string_printable (result):
            print ("key=", key, "value=", ''.join(list(map(chr, result))))
    cnt=cnt+1

And we get:

Listing 78.4: Results

message #1
key= 20 value=`eb^h%|`hudw|_af{n~f%ljmSbnwlpk
key= 21 value=ajcji"}cawtgv{^bgto}g"millcmvkqh
key= 22 value=bkd\j#rbbvsfuz!cduh|d#bhomdlujni
key= 23 value=check security device connection
key= 24 value=lifbl!pd|tqhsx#ejwjjbb!'nQofbshlo
message #2
key= 7 value=No security device found
key= 8 value=An#rbbvsVuz!cduhld#ghtme?!#!"!!#!
message #3
key= 7 value=Bk<waoqNUpu$`yreo\wpmpusj,bkIjh
key= 8 value=Mj?vfnrOjqv%gxqd`_vwlstlk/c1Hii
key= 9 value=Lm>ugasLkvw&fgpgag^uvcrwml.'mwhj
key= 10 value=Ol!td`tMhwx`efwbf!tubuvnm!anvok
key= 11 value=No security device station found
key= 12 value=In#rjbvsnuz!{duhdd#r`\whho#gPtme
message #4
key= 14 value=Number of authorized users exceeded
key= 15 value=Ovlmdq!hg#`juknuhydk!vrbsp!Zy`dbefe
message #5
key= 17 value=check security device station
key= 18 value=`ijbh!td`tMhwx`efwbf!tubuVnm!'!

There is some garbage, but we can quickly find the English-language messages!

By the way, since the algorithm is a simple xoring encryption, the very same function can be used to encrypt messages. If needed, we can encrypt our own messages,
and patch the program by inserting them.

### 78.3 Example #3: MS-DOS

Another very old software for MS-DOS from 1995 also developed by a company that disappeared a long time ago.

In the pre-DOS extenders era, all the software for MS-DOS mostly relied on 16-bit 8086 or 80286 CPUs, so en masse the code was 16-bit. The 16-bit code is mostly same as you already saw in this book, but all registers are 16-bit and there are less instructions available.

The MS-DOS environment has no system drivers, and any program can deal with the bare hardware via ports, so here you can see the OUT/IN instructions, which are present in mostly in drivers in our times (it is impossible to access ports directly in user mode on all modern OSes).

Given that, the MS-DOS program which works with a dongle has to access the LPT printer port directly. So we can just search for such instructions. And yes, here they are:

```assembly
seg030:0034    out_port proc far ; CODE XREF: sent_pro
\ +22p
seg030:0034    ; sent_pro+2Ap ...
seg030:0034    arg_0 = byte ptr 6
seg030:0034    push bp
seg030:0035    mov bp, sp
seg030:0037    mov dx, _out_port ; 0x378
seg030:003B    mov al, [bp+arg_0]
seg030:003E    out dx, al
seg030:003F    pop bp
seg030:0040    retf
seg030:0040    out_port endp
```

(All label names in this example were given by me).

**out_port()** is referenced only in one function:

```assembly
seg030:0041    sent_pro proc far ; CODE XREF: *
\ +34p
seg030:0041    var_3 = byte ptr -3
seg030:0041    var_2 = word ptr -2
seg030:0041    arg_0 = dword ptr 6
seg030:0041
```

1101
Segment 030: enter 4, 0
push si
push di
mov dx, _in_port_1 ; 0x37A
in al, dx
mov bl, al
and bl, 0FEh
or bl, 4
mov al, bl
out dx, al
push 0FFh
push cs
call near ptr out_port
pop cx
push 0D3h
push cs
call near ptr out_port
pop cx
xor si, si
jmp short loc_359D4
inc si
loc_359D3: ; CODE XREF: sent_pro+37j
loc_359D4: ; CODE XREF: sent_pro+30j
cmp si, 96h
jl short loc_359D3
push 0C3h
push cs
call near ptr out_port
pop cx
push 0C7h
push cs
call near ptr out_port
pop cx
push 0D3h
push cs
call near ptr out_port
pop cx
push 0C3h
push cs
call near ptr out_port
pop cx
push 0C7h
CHAPTER 78. DONGLES

seg030:009D 0E  push cs
seg030:009E E8 93 FF  call near ptr out_port
seg030:00A1 59  pop  cx
seg030:00A2 68 D3 00  push  0D3h
seg030:00A5 0E  push  cs
seg030:00A6 E8 8B FF  call near ptr out_port
seg030:00A9 59  pop  cx
seg030:00AA BF FF FF  mov  di, 0FFFFh
seg030:00AD EB 40  jmp short loc_35A4F
seg030:00AF
seg030:00AF loc_35A0F:  ; CODE XREF: sent_pro+BDj
seg030:00AF BE 04 00  mov  si, 4
seg030:00B2
seg030:00B2 loc_35A12:  ; CODE XREF: sent_pro+ACj
seg030:00B2 D1 E7  shl  di, 1
seg030:00B4 8B 16 80 E7  mov  dx, _in_port_2 ; 0x379
seg030:00B8 EC  in  al, dx
seg030:00B9 A8 80  test  al, 80h
seg030:00BB 75 03  jnz short loc_35A20
seg030:00BD 83 CF 01  or  di, 1
seg030:00C0
seg030:00C0 loc_35A20:  ; CODE XREF: sent_pro+7Aj
seg030:00C0 F7 46 FE 08+  test  [bp+var_2], 8
seg030:00C5 74 05  jz  short loc_35A2C
seg030:00C7 68 D7 00  push  0D7h ; '+'
seg030:00CA EB 0B  jmp short loc_35A37
seg030:00CC
seg030:00CC loc_35A2C:  ; CODE XREF: sent_pro+84j
seg030:00CC 68 C3 00  push  0C3h
seg030:00CF 0E  push  cs
seg030:00D0 E8 61 FF  call near ptr out_port
seg030:00D3 59  pop  cx
seg030:00D4 68 C7 00  push  0C7h
seg030:00D7
seg030:00D7 loc_35A37:  ; CODE XREF: sent_pro+89j
seg030:00D7 0E  push  cs
seg030:00D8 E8 59 FF  call near ptr out_port
seg030:00DB 59  pop  cx
seg030:00DC 68 D3 00  push  0D3h
seg030:00DF 0E  push  cs
seg030:00E0 E8 51 FF  call near ptr out_port
seg030:00E3 59  pop  cx
seg030:00E4 8B 46 FE  mov  ax, [bp+var_2]
seg030:00E7 D1 E0  shl  ax, 1
seg030:00E9 89 46 FE  mov  [bp+var_2], ax
seg030:00EC 4E  dec  si
seg030:00ED 75 C3  jnz short loc_35A12
This is again a Sentinel Pro “hashing” dongle as in the previous example. It is
noticeably because text strings are passed here, too, and 16 bit values are returned and compared with others.

So that is how Sentinel Pro is accessed via ports. The output port address is usually 0x378, i.e., the printer port, where the data to the old printers in pre-USB era was passed to. The port is uni-directional, because when it was developed, no one imagined that someone will need to transfer information from the printer. The only way to get information from the printer is a status register on port 0x379, which contains such bits as “paper out”, “ack”, “busy”—thus the printer may signal to the host computer if it is ready or not and if paper is present in it. So the dongle returns information from one of these bits, one bit at each iteration.

_in_port_2 contains the address of the status word (0x379) and _in_port_1 contains the control register address (0x37A).

It seems that the dongle returns information via the “busy” flag at seg030:00B9: each bit is stored in the DI register, which is returned at the end of the function.

What do all these bytes sent to output port mean? Hard to say. Probably commands to the dongle. But generally speaking, it is not necessary to know: it is easy to solve our task without that knowledge.

Here is the dongle checking routine:

```
00000000 struct_0 struc ; (sizeof=0x1B)
00000000 field_0 db 25 dup(?) ; string(C)
00000019 _A dw ?
0000001B struct_0 ends

dseg:3CBC 61 63 72 75+_Q struct_0 <'hello', 01122h>
dseg:3CBC 6E 00 00 00+ ; DATA XREF: check_dongle+2Eo
... skipped ...

dseg:3E00 63 6F 66 66+ struct_0 <'coffee', 7EB7h>
dseg:3E1B 64 6F 67 00+ struct_0 <'dog', 0FFADh>
dseg:3E36 63 61 74 00+ struct_0 <'cat', 0FF5Fh>
dseg:3E51 70 61 70 65+ struct_0 <'paper', 0FFDFh>
dseg:3E6C 63 6F 6B 65+ struct_0 <'coke', 0F568h>
dseg:3E87 63 6C 6F 63+ struct_0 <'clock', 55EAh>
dseg:3EA2 64 69 72 00+ struct_0 <'dir', 0FFAEh>
dseg:3EBD 63 6F 70 79+ struct_0 <'copy', 0F557h>

seg030:0145 check_dongle proc far ; CODE XREF: 
" sub_3771D+3EP
```

---

4 If we consider Centronics only. The following IEEE 1284 standard allows the transfer of information from the printer.
CHASED DONGLES

seg030:0145  var_6 = dword ptr -6
seg030:0145  var_2 = word ptr -2
seg030:0145
seg030:0145  C8 06 00 00 enter 6, 0
seg030:0149  56 push si
seg030:014A  66 6A 00 push large 0 ; newtime
seg030:014D  6A 00 push 0 ; cmd
seg030:014F  9A C1 18 00+ call _biostime
seg030:0154  52 push dx
seg030:0155  50 push ax
seg030:0156  66 58 pop eax
seg030:0158  83 C4 06 add sp, 6
seg030:015B  66 89 46 FA mov [bp+var_6], eax
seg030:015F  6A 00 push 0 ; cmd
seg030:0161  9A C1 18 00+ call _biostime
seg030:0164  52 push dx
seg030:0165  50 push ax
seg030:0166  66 58 pop eax
seg030:0168  83 C4 06 add sp, 6
seg030:016B  66 89 46 FA mov [bp+var_6], eax
seg030:016D  6A 00 push 0 ; cmd
seg030:016F  9A C1 18 00+ call _biostime
seg030:0172  52 push dx
seg030:0173  50 push ax
seg030:0174  66 58 pop eax
seg030:0176  83 C4 04 add sp, 4
seg030:0179  66 89 46 FA mov [bp+var_6], eax
seg030:017B  6A 00 push 0 ; cmd
seg030:017D  9A C1 18 00+ call _biostime
seg030:0180  52 push dx
seg030:0181  50 push ax
seg030:0182  66 58 pop eax
seg030:0184  83 C4 04 add sp, 4
seg030:0187  66 89 46 FA mov [bp+var_6], eax
seg030:0189  6A 00 push 0 ; cmd
seg030:018A  9A C1 18 00+ call _biostime
seg030:018C  52 push dx
seg030:018D  50 push ax
seg030:018E  66 58 pop eax
seg030:0190  83 C4 04 add sp, 4
seg030:0193  66 89 46 FA mov [bp+var_6], eax
seg030:0195  6A 00 push 0 ; cmd
seg030:0196  9A C1 18 00+ call _biostime
seg030:0198  52 push dx
seg030:0199  50 push ax
seg030:019A  66 58 pop eax
seg030:019C  8B 87 D5 3C mov ax, _Q.A[bx]
seg030:019E  6A 00 push 0 ; cmd
seg030:019F  9A C1 18 00+ call _biostime
seg030:01A2  52 push dx
seg030:01A3  74 05 jz short loc_35B0A
seg030:01A5  B8 01 00 mov ax, 1
seg030:01A8  EB 02 jmp short loc_35B0C
seg030:01AA
seg030:01AA  loc_35B0A: ; CODE XREF: check_dongle+1
seg030:01AA  FJ
seg030:01AA  ; check_dongle+5EJ
Since the routine can be called very frequently, e.g., before the execution of each important software feature, and accessing the dongle is generally slow (because of the slow printer port and also slow MCU in the dongle), they probably added a way to skip some dongle checks, by checking the current time in the biostime() function.

The get_rand() function uses the standard C function:

```
seg030:01BF  get_rand proc far ; CODE XREF: 
  check_dongle+25p
seg030:01BF
seg030:01BF  arg_0 = word ptr 6
seg030:01BF
seg030:01BF  55  push  bp
seg030:01C0  8B  EC  mov  bp, sp
seg030:01C2  9A  3D 21 00+  call  _rand
seg030:01C7  66  0F  BF C0  movsx  eax, ax
seg030:01CB  66  0F  56+  movsx  edx, [bp+arg_0]
seg030:01D0  66  0F  BF 56+  movsx  edx, [bp+arg_0]
seg030:01D4  66  BB 00 80+  mov  ebx, 8000h
seg030:01DA  66  99  cdq
seg030:01DC  66  F7 FB  idiv  ebx
seg030:01DF  5D  pop  bp
seg030:01E0  CB  retf
seg030:01E0  get_rand endp
```

So the text string is selected randomly, passed into the dongle, and then the result of the hashing is compared with the correct value.

The text strings seem to be constructed randomly as well, during software development.

And this is how the main dongle checking function is called:

```
seg033:087B  9A 45 01 96+  call  check_dongle
seg033:0880  0B C0  or  ax, ax
seg033:0882  74 62  jz  short OK
seg033:0884  83 3E 60 42+  cmp  word_620E0, 0
seg033:0889  75 5B  jnz  short OK
```
Bypassing the dongle is easy, just force the check_dongle() function to always return 0.

For example, by inserting this code at its beginning:

```assembly
mov ax, 0
retf
```

The observant reader might recall that the strcpy() C function usually requires two pointers in its arguments, but we see that 4 values are passed:

```assembly
seg033:088F 1E push ds
seg033:0890 68 22 44 push offset aTrupcRequiresA ; "⤦ This Software Requires a Software Lock
seg033:0893 1E push ds
seg033:0894 68 60 E9 push offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+ call _strcpy
seg033:089C 83 C4 08 add sp, 8
seg033:089F 1E push ds
seg033:08A0 68 42 44 push offset aPleaseContactA ; "⤦ Please Contact ...
seg033:08A3 1E push ds
seg033:08A4 68 60 E9 push offset byte_6C7E0 ; dest
seg033:08A7 9A CD 64 00+ call _strcat
```

This is related to MS-DOS' memory model. You can read more about it here: 94 on page 1278.

So as you may see, strcpy() and any other function that take pointer(s) in arguments work with 16-bit pairs.

Let's get back to our example. DS is currently set to the data segment located in the executable, that is where the text string is stored.

In the sent_pro() function, each byte of the string is loaded at seg030:00EF: the LES instruction loads the ES:BX pair simultaneously from the passed argument. The MOV at seg030:00F5 loads the byte from the memory at which the ES:BX pair points.
Chapter 79

“QR9”: Rubik’s cube inspired amateur crypto-algorithm

Sometimes amateur cryptosystems appear to be pretty bizarre.

The author of this book was once asked to reverse engineer an amateur cryptoalgorithm of some data encryption utility, the source code for which was lost.\(^1\)

Here is the listing exported from IDA for the original encryption utility:

```
.text:00541000 set_bit proc near ; CODE 7
    XREF: rotate1+42
.text:00541000
    rotate2+42 ...
.text:00541000
.text:00541000 arg_0 = dword ptr 4
.text:00541000 arg_4 = dword ptr 8
.text:00541000 arg_8 = dword ptr 0Ch
.text:00541000 arg_C = byte ptr 10h
.text:00541000
.text:00541000 = dword ptr 4
.text:00541000 = dword ptr 8
.text:00541000 = dword ptr 0Ch
.text:00541000 = byte ptr 10h
.text:00541000
.text:00541000 mov al, [esp+arg_C]
.text:00541004 mov ecx, [esp+arg_8]
.text:00541008 push esi
.text:00541009 mov esi, [esp+4+arg_0]
.text:0054100D test al, al
.text:0054100F mov eax, [esp+4+arg_4]
.text:00541013 mov dl, 1
.text:00541015 jz short loc_54102B
.text:00541017 shl dl, cl
.text:00541019 mov cl, cube64[eax+esi*8]
```

\(^1\)He also got permission from the customer to publish the algorithm’s details
CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

.text:00541020 or cl, dl
.text:00541022 mov cube64[eax+esi*8], cl
.text:00541029 pop esi
.text:0054102A retn
.text:0054102B
.text:0054102B loc_54102B: ; CODE ⤤
.text:0054102B shl dl, cl
.text:0054102D mov cl, cube64[eax+esi*8]
.text:00541034 not dl
.text:00541036 and cl, dl
.text:00541038 mov cube64[eax+esi*8], cl
.text:0054103F pop esi
.text:00541040 retn
.text:00541040 set_bit endp
.text:00541040
.text:00541041 align 10h
.text:00541050 ; =============== SUBROUTINE ⤤
.text:00541050 get_bit proc near ; CODE ⤤
.text:00541050 arg_0 = dword ptr 4
.text:00541050 arg_4 = dword ptr 8
.text:00541050 arg_8 = byte ptr 0Ch
.text:00541050
.text:00541050 mov eax, [esp+arg_4]
.text:00541054 mov ecx, [esp+arg_0]
.text:00541058 mov al, cube64[eax+ecx*8]
.text:0054105C mov cl, [esp+arg_8]
.text:00541063 shr al, cl
.text:00541065 and al, 1
.text:00541067 retn
.text:00541067 get_bit endp
.text:00541067
.text:00541068 align 10h
.text:00541070 ; =============== SUBROUTINE ⤤
.text:00541070 rotate1 proc near ; CODE ⤤
CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```assembly
.text:00541070  sub    esp, 40h
.text:00541073  push   ebx
.text:00541074  push   ebp
.text:00541075  mov    ebp, [esp+48h+arg_0]
.text:00541079  push   esi
.text:0054107A  push   edi
.text:0054107B  xor    edi, edi    ; EDI is loop1 counter
.text:0054107D  lea    ebx, [esp+50h+internal_array_64]
.text:00541081  xor    esi, esi    ; ESI is loop2 counter
.text:00541084  push   ebp    ; arg_0
.text:00541085  push   esi
.text:00541086  call   get_bit
.text:0054108B  add   esp, 0Ch
.text:0054108E  mov    [ebx+esi], al    ; store to internal array
.text:00541091  inc    esi
.text:00541092  cmp    esi, 8
.text:00541095  jl     short first_loop2_begin
.text:00541097  inc    edi
.text:00541098  add   ebx, 8
.text:0054109B  cmp   edi, 8
.text:0054109E  jl     short first_loop1_begin
.text:005410A0  lea    ebx, [esp+50h+internal_array_64]
.text:005410A4  mov    edi, 7    ; EDI is loop1 counter, initial state is 7
.text:005410A9  xor    esi, esi    ; ESI is loop2 counter
```

1111
.text:005410AB_second_loop2_begin: ; CODE  
   ⊲ XREF: rotate1+4E
   .text:005410AB      mov    al, [ebx+esi] ; value  
   ⊲ from internal array
   .text:005410AE      push   eax
   .text:005410AF      push   ebp ; arg_0
   .text:005410B0      push   edi
   .text:005410B1      push   esi
   .text:005410B2      call   set_bit
   .text:005410B7      add    esp, 10h
   .text:005410BA      inc    esi ; ⊲ increment loop2 counter
   .text:005410BB      cmp    esi, 8
   .text:005410BE      jl     short second_loop2_begin
   .text:005410C0      dec    edi ; ⊲ decrement loop2 counter
   .text:005410C1      add    ebx, 8
   .text:005410C4      cmp    edi, 0FFFFFFFFh
   .text:005410C7      jg     short second_loop1_begin
   .text:005410C9      pop    edi
   .text:005410CA      pop    esi
   .text:005410CB      pop    ebp
   .text:005410CC      pop    ebx
   .text:005410CD      add    esp, 40h
   .text:005410D0      retn
   .text:005410E0      align 10h
   .text:005410E0      rotate2 proc near ; CODE  
   ⊲ XREF: rotate_all_with_password+7A
   .text:005410E0      internal_array_64= byte ptr -40h
   .text:005410E0      arg_0 = dword ptr 4
   .text:005410E0      align 10h
   .text:005410E0      rotate2 endp
   .text:005410E0      rotate1
   .text:005410E0      align 10h
   .text:005410E0      rotate2  ; ================ SUBROUTINE  
   ⊲ ================
   .text:005410E0      rotate1
   .text:005410E0      rotate2
CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

.text:005410ED  lea  ebx, [esp+50h+
          \internal_array_64]
.text:005410F1
.text:005410F1  loc_5410F1:  ; CODE 
          \XREF: rotate2+2E
.text:005410F1  xor  esi, esi  ; loop2 
          \counter
.text:005410F3
.text:005410F3  loc_5410F3:  ; CODE 
          \XREF: rotate2+25
.text:005410F3  push  esi  ; loop2
.text:005410F4  push  edi  ; loop1
.text:005410F5  push  ebp  ; arg_0
.text:005410F6  call  get_bit
.text:005410FB  add  esp, 0Ch
.text:005410FE  mov  [ebx+esi], al  ; store 
          \to internal array
.text:00541101  inc  esi  ;
          \increment loop1 counter
.text:00541102  cmp  esi, 8
.text:00541105  jl  short  loc_5410F3
.text:00541107  inc  edi  ;
          \increment loop2 counter
.text:00541108  add  ebx, 8
.text:0054110B  cmp  edi, 8
.text:0054110E  jl  short  loc_5410F1
.text:00541110  lea  ebx, [esp+50h+
          \internal_array_64]
.text:00541114  mov  edi, 7  ; loop1 
          \counter is initial state 7
.text:00541119
.text:00541119  loc_541119:  ; CODE 
          \XREF: rotate2+57
.text:00541119  xor  esi, esi  ; loop2 
          \counter
.text:0054111B
.text:0054111B  loc_54111B:  ; CODE 
          \XREF: rotate2+4E
.text:0054111B  mov  al, [ebx+esi]  ; get 
          \byte from internal array
.text:0054111E  push  eax
.text:0054111F  push  edi  ; loop1 
          \counter
.text:00541120  push  esi  ; loop2 
          \counter
.text:00541121  push  ebp  ; arg_0
.text:00541122  call  set_bit
CHAPTER 79. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

.demo:00541127  add esp, 10h
.demo:0054112A  inc esi ;
.demo:0054112B  cmp esi, 8
.demo:0054112E  jl short loc_54111B
.demo:00541130  dec edi ;
.demo:00541131  add ebx, 8
.demo:00541134  cmp edi, 0FFFFFFFFh
.demo:00541137  jg short loc_541119
.demo:00541139  pop edi
.demo:0054113A  pop esi
.demo:0054113B  pop ebp
.demo:0054113C  pop ebx
.demo:0054113D  add esp, 40h
.demo:00541140  retn
.demo:00541140  rotate2 endp
.demo:00541141  align 10h
.demo:00541150  ; =============== S U B R O U T I N E
.demo:00541150  ========================================
.demo:00541150  rotate3 proc near
.demo:00541150  ; CODE
.demo:00541150  XREF: rotate_all_with_password+66
.demo:00541150  var_40 = byte ptr -40h
.demo:00541150  arg_0 = dword ptr 4
.demo:00541150  sub esp, 40h
.demo:00541153  push ebx
.demo:00541154  push ebp
.demo:00541155  mov ebp, [esp+48h+arg_0]
.demo:00541159  push esi
.demo:0054115A  push edi
.demo:0054115B  xor edi, edi
.demo:0054115D  lea ebx, [esp+50h+var_40]
.demo:00541161  loc_541161: ; CODE
.demo:00541161  XREF: rotate3+2E
.demo:00541161  xor esi, esi
.demo:00541163  push esi
.demo:00541163  xchg esp, ebp
.demo:00541163  xchg ebp, esp
.demo:00541163  push ebp
.demo:00541163  push esi
.demo:00541164  push ebx
.demo:00541164  push ebp
.text:00541165    push   edi
.text:00541166    call    get_bit
.text:0054116B    add    esp, 0Ch
.text:0054116E    mov    [ebx+esi], al
.text:00541171    inc    esi
.text:00541172    cmp    esi, 8
.text:00541175    jl     short loc_541163
.text:00541177    inc    edi
.text:00541178    add    ebx, 8
.text:0054117B    cmp    edi, 8
.text:0054117E    jl     short loc_541161
.text:00541180    xor    ebx, ebx
.text:00541182    lea    edi, [esp+50h+var_40]
.text:00541186    loc_541186:    ; CODE ⊥
    XREF: rotate3+54
.text:00541186    mov    esi, 7
.text:0054118B    loc_54118B:    ; CODE ⊥
    XREF: rotate3+4E
.text:0054118B    mov    al, [edi]
.text:0054118D    push   eax
.text:0054118E    push   ebx
.text:0054118F    push   ebp
.text:00541190    push   esi
.text:00541191    call    set_bit
.text:00541196    add    esp, 10h
.text:00541199    inc    edi
.text:0054119A    dec    esi
.text:0054119B    cmp    esi, 0FFFFFFFh
.text:0054119E    jg     short loc_54118B
.text:005411A0    inc    ebx
.text:005411A1    cmp    ebx, 8
.text:005411A4    jl     short loc_541186
.text:005411A6    pop    edi
.text:005411A7    pop    esi
.text:005411A8    pop    ebp
.text:005411A9    pop    ebx
.text:005411AA    add    esp, 40h
.text:005411AD    retn
.text:005411AD    rotate3 endp
.text:005411AD
.text:005411AE    align 10h
.text:005411B0    ; =============== SUBROUTINE ⊥
    =============== SUBROUTINE ⊥
.text:005411B0

CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

.text:005411B0 rotate_all_with_password proc near ; CODE

  XREF: crypt+1F
text:005411B0 ;

  XREF: decrypt+36
text:005411B0

.text:005411B0 arg_0 = dword ptr 4
.text:005411B0 arg_4 = dword ptr 8
.text:005411B0

.text:005411B0 mov eax, [esp+arg_0]
.text:005411B4 push ebp
.text:005411B5 mov ebp, eax
.text:005411B7 cmp byte ptr [ebp], 0
.text:005411BA jz exit
.text:005411C0 push ebx
.text:005411C1 mov ebx, [esp+8+arg_4]
.text:005411C5 push esi
.text:005411C6 push edi
.text:005411C7

.text:005411C7 loop_begin: ; CODE

  XREF: rotate_all_with_password+9F
text:005411C7 movsx eax, byte ptr [ebp+0]
.text:005411CB push eax ; C
.text:005411CC call _tolower
.text:005411D1 add esp, 4
.text:005411D4 cmp al, 'a'
.text:005411D6 jl short skip_subtracting
.text:005411D8 cmp al, 'z'
.text:005411DA jg short skip_subtracting

  XREF: next_character_in_password
text:005411DC movsx ecx, al
.text:005411DF sub ecx, 'a'
.text:005411E2 cmp ecx, 24
.text:005411E5 jle short skip_subtracting
.text:005411E7 sub ecx, 24
.text:005411EA

.text:005411EA skip_subtracting: ; CODE

  XREF: rotate_all_with_password+35
text:005411EA mov eax, 55555556h
.text:005411EF imul ecx
.text:005411F1 mov eax, edx
.text:005411F3 shr eax, 1Fh
.text:005411F6 add edx, eax
.text:005411F8 mov eax, ecx
.text:005411FA mov esi, edx
.text:005411FC mov ecx, 3

1116
```plaintext
.text:00541201 cdq
.text:00541202 idiv ecx
.text:00541204 sub edx, 0
.text:00541207 jz short call_rotate1
.text:00541209 dec edx
.text:0054120A jz short call_rotate2
.text:0054120C dec edx
.text:0054120D jnz short next_character_in_password
.text:0054120F test ebx, ebx
.text:00541211 jle short next_character_in_password
.text:00541213 mov edi, ebx
.text:00541215 push esi
.text:00541216 call rotate3
.text:0054121B add esp, 4
.text:0054121E dec edi
.text:0054121F jnz short call_rotate3
.text:00541221 jmp short next_character_in_password
.text:00541223 push esi
.text:00541226 call rotate3
.text:0054122F add esp, 4
.text:00541232 dec edi
.text:00541233 jnz short call_rotate2
.text:00541235 jmp short next_character_in_password
.text:00541237 push esi
.text:00541240 call rotate2
.text:00541243 add esp, 4
.text:00541246 dec edi
.text:00541247 jnz short loc_541229
.text:00541250 jmp short next_character_in_password
.text:00541252 push esi
.text:00541255 call rotate2
.text:0054125E add esp, 4
.text:00541261 dec edi
.text:00541262 jnz short loc_541229
.text:00541265 jmp short next_character_in_password
```

---

CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```plaintext
.text:00541201 cdq
.text:00541202 idiv ecx
.text:00541204 sub edx, 0
.text:00541207 jz short call_rotate1
.text:00541209 dec edx
.text:0054120A jz short call_rotate2
.text:0054120C dec edx
.text:0054120D jnz short next_character_in_password
.text:0054120F test ebx, ebx
.text:00541211 jle short next_character_in_password
.text:00541213 mov edi, ebx
.text:00541215 call_rotate3: ; CODE
.text:00541215 push esi
.text:00541216 call rotate3
.text:0054121B add esp, 4
.text:0054121E dec edi
.text:0054121F jnz short call_rotate3
.text:00541221 jmp short next_character_in_password
.text:00541223 push esi
.text:00541226 call rotate3
.text:0054122F add esp, 4
.text:00541232 dec edi
.text:00541233 jnz short call_rotate2
.text:00541235 jmp short next_character_in_password
.text:00541237 push esi
.text:00541240 call rotate2
.text:00541243 add esp, 4
.text:00541246 dec edi
.text:00541247 jnz short loc_541229
.text:00541250 jmp short next_character_in_password
.text:00541252 push esi
.text:00541255 call rotate2
.text:0054125E add esp, 4
.text:00541261 dec edi
.text:00541262 jnz short loc_541229
.text:00541265 jmp short next_character_in_password
```
CHAPTER 79. "QR9": RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

.text:0054123D
.text:0054123D loc_54123D: ; CODE ↗
    XREF: rotate_all_with_password+97
.text:0054123D push esi
.text:0054123E call rotate1
.text:00541243 add esp, 4
.text:00541246 dec edi
.text:00541247 jnz short loc_54123D
.text:00541249
.text:00541249 next_character_in_password: ; CODE ↗
    XREF: rotate_all_with_password+26
.text:00541249 ; rotate_all_with_password+2A ...
    .text:00541249 mov al, [ebp+1]
.text:0054124C inc ebp
.text:0054124D test al, al
.text:0054124F jnz loop_begin
.text:00541255 pop edi
.text:00541256 pop esi
.text:00541257 pop ebx
.text:00541258 exit: ; CODE ↗
    XREF: rotate_all_with_password+A
    .text:00541258 pop ebp
.text:00541259 retn
.text:00541259 rotate_all_with_password endp
.text:00541259
.text:00541260 ; =============== S U B R O U T I N E ↗
    ===============
.text:00541260 crypt proc near ; CODE ↗
    XREF: crypt_file+8A
.text:00541260 push ebx
.text:00541260 crypt proc near ; CODE ↗
    XREF: crypt_file+8A
.text:00541260 arg_0 = dword ptr 4
.text:00541260 arg_4 = dword ptr 8
.text:00541260 arg_8 = dword ptr 0Ch
.text:00541260 push ebx
.text:00541260 mov ebx, [esp+4+arg_0]
.text:00541265 push ebp
.text:00541266 push esi
.text:00541267 push edi
.text:00541268 xor ebp, ebp
.text:0054126A

1118
CHAPTER 79. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

.text:0054126A loc_54126A: ; CODE
        XREF: crypt+41
.text:0054126A  mov    eax, [esp+10h+arg_8]
.text:0054126E  mov    ecx, 10h
.text:00541273  mov    esi, ebx
.text:00541275  mov    edi, offset cube64
.text:0054127A  push   1
.text:0054127C  push   eax
.text:0054127D  rep movsd call rotate_all_with_password
.text:0054127F  mov    eax, [esp+18h+arg_4]
.text:00541284  mov    edi, ebx
.text:00541288  add    ebp, 40h
.text:0054128A  add    esp, 8
.text:0054128D  add    ebx, 40h
.text:00541290  cmp    ebp, eax
.text:00541295  mov    esi, offset cube64
.text:0054129A  add    ebx, 40h
.text:0054129D  cmp    ebp, eax
.text:005412A1  jl     short loc_54126A
.text:005412A3  pop    edi
.text:005412A4  pop    esi
.text:005412A5  pop    ebp
.text:005412A6  pop    ebx
.text:005412A7  retn
.text:005412A7  crypt endp
.text:005412A8  align 10h
.text:005412B0  ; =============== SUBROUTINE ===============
.text:005412B0 ; int __cdecl decrypt(int, int, void *Src)
.text:005412B0 decrypt proc near ; CODE
        XREF: decrypt_file+99
.text:005412B0 .arg_0 = dword ptr 4
.text:005412B0 .arg_4 = dword ptr 8
.text:005412B0 Src = dword ptr 0Ch
.text:005412B0 mov    eax, [esp+Src]
.text:005412B4 push   ebx
.text:005412B5 push   ebp
.text:005412B6 push   esi
.text:005412B7 push   edi
.text:005412B8 push   eax ; Src
CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

.text:005412B9    call __strdup
.text:005412BE    push eax ; Str
.text:005412BF    mov [esp+18h+Src], eax
.text:005412C3    call __strrev
.text:005412C8    mov ebx, [esp+18h+arg_0]
.text:005412CC    add esp, 8
.text:005412CF    xor ebp, ebp
.text:005412D1    loc_5412D1: ; CODE
XREF: decrypt+58
.text:005412D1    mov ecx, 10h
.text:005412D6    mov esi, ebx
.text:005412D8    mov edi, offset cube64
.text:005412DD    push 3
.text:005412DF    rep movsd
.text:005412E1    mov ecx, [esp+14h+Src]
.text:005412E5    push ecx
.text:005412E6    call rotate_all_with_password
.text:005412EB    mov eax, [esp+18h+arg_4]
.text:005412EF    mov edi, ebx
.text:005412F1    add ebp, 40h
.text:005412F4    add esp, 8
.text:005412F7    mov ecx, 10h
.text:005412FC    mov esi, offset cube64
.text:00541301    add ebx, 40h
.text:00541304    cmp ebp, eax
.text:00541306    rep movsd
.text:00541308    jl short loc_5412D1
.text:0054130A    mov edx, [esp+10h+Src] ; Memory
.text:0054130E    push edx
.text:0054130F    call _free
.text:00541314    add esp, 4
.text:00541317    pop edi
.text:00541318    pop esi
.text:00541319    pop ebp
.text:0054131A    pop ebx
.text:0054131B    retn
.text:0054131B    decrypt endp
.text:0054131B    ; SUBROUTINE
.align 10h
.text:00541320    ==crypt_file==
.text:00541320    int __cdecl crypt_file(int Str, char *Filename, int password)
CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

`.text:00541320` crypt_file proc near ; CODE ↗
_XREF: _main+42_
`.text:00541320`
`.text:00541320` Str = dword ptr 4
 `.text:00541320` Filename = dword ptr 8
 `.text:00541320` password = dword ptr 0Ch
 `.text:00541320`
`.text:00541320` mov eax, [esp+Str]
`.text:00541324` push ebp
`.text:00541325` push offset Mode ; "rb"
`.text:0054132A` push eax ; ↗
_XREF: _main+43_
`.text:0054132B` call _fopen ; open ↗
`.text:0054132B` Filename ↗
`.text:0054132B` call _fopen ; open ↗
`.text:0054132B` file ↗
`.text:00541330` mov ebp, eax
`.text:00541332` add esp, 8
`.text:00541335` test ebp, ebp
`.text:00541337` jnz loc_541348
`.text:0054133D` push offset Format ; " ↗
`.text:0054133E` call _printf ↗
`.text:0054133E` Cannot open input file!
`.
`.text:00541343` add esp, 4
`.text:00541346` pop ebp
`.text:00541347` retn
`.text:00541348 loc_541348:` ; CODE ↗
_XREF: crypt_file+17_
`.text:00541348` push ebx
`.text:00541349` push esi
`.text:0054134A` push edi
`.text:0054134B` push 2 ; Origin
`.text:0054134D` push 0 ; Offset
`.text:0054134F` push ebp ; File
“.text:00541350` call _fseek ; get ↗
`.text:00541355` push ebp ; File
`.text:00541356` push ebp ; File
`.text:0054135F` push 0 ; Origin
`.text:0054136D` push 0 ; Offset
`.text:0054135F` push ebp ; File
`.text:00541360` move [esp+2Ch+Str], eax
`.text:00541364` call _fseek ; rewind ↗
`.text:00541369` mov esi, [esp+2Ch+Str]
`.text:0054136D` and esi, 0FFFFFFFC0h ; reset ↗
`.text:0054136F` all lowest 6 bits
`.text:00541370` add esi, 40h ; align ↗
1121
size to 64-byte border
push esi ; Size
call _malloc
mov ecx, esi
mov ebx, eax ; allocated buffer pointer -> to EBX
mov edx, ecx
xor eax, eax
mov edi, ebx
push ebp ; File
shr ecx, 2
rep stosd
mov ecx, edx
push 1 ; Count
and ecx, 3
rep stosb ; memset (buffer, 0, aligned_size)
mov eax, [esp+38h+Str]
push ebx ; DstBuf
call _fread ; read
push ebp ; File
call _fclose
mov ecx, [esp+44h+password]
push ecx ; password
push esi ; aligned size
push ebx ; buffer
call crypt ; do crypt
mov edx, [esp+50h+Filename]
add esp, 40h
push offset aWb ; "wb"
push edx ;
call _fopen
mov edi, eax
push edi ; File
push 1 ; Count
push 3 ; Size
push offset aQr9 ; "QR9"
call _fwrite ; write
push edi ; File
.text:005413D3  push  1 ; Count
.text:005413D5  lea   eax, [esp+30h+Str]
.text:005413D9  push  4 ; Size
.text:005413DB  push  eax ; Str
.text:005413DC  call  _fwrite ; write
  \ original file size
.text:005413E1  push  edi ; File
.text:005413E2  push  1 ; Count
.text:005413E4  push  esi ; Size
.text:005413E5  push  ebx ; Str
.text:005413E6  call  _fwrite ; write
  \ encrypted file
.text:005413EB  push  edi ; File
.text:005413EC  call  _fclose
.text:005413F1  push  ebx ; Memory
.text:005413F2  call  _free
.text:005413F7  add   esp, 40h
.text:005413FA  pop   edi
.text:005413FC  pop   esi
.text:005413FD  pop   ebx
.text:005413FE  retn
.text:005413FE  encrypt_file endp
.text:005413FF  align 10h
.text:00541400  ; =============== S U B R O U T I N E  \\
.text:00541400  \=======================================
.text:00541400  ; int __cdecl decrypt_file(char *Filename, int, \\
.text:00541400  \ void *Src)
.text:00541400  decrypt_file   proc near ; CODE
  \ XREF: _main+6E
.text:00541400  decrypt_file  proc near ; CODE
  \ XREF: _main+6E
.text:00541400  Filename   = dword ptr  4
.text:00541400  arg_4      = dword ptr  8
.text:00541400  Src        = dword ptr  0Ch
.text:00541400  
.text:00541400  mov    eax, [esp+Filename]
.text:00541404  push   ebx
.text:00541405  push   ebp
.text:00541406  push   esi
.text:00541407  push   edi
.text:00541408  push   offset aRb ; "rb"
.text:0054140D  push    eax
  \Filename
CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

.text:0054140E  call _fopen
.text:00541413  mov esi, eax
.text:00541415  add esp, 8
.text:00541418  test esi, esi
.text:0054141A  jnz short loc_54142E
.text:0054141C  push offset aCannotOpenIn_0 ;
                "Cannot open input file!\n"
.text:00541421  call _printf
.text:00541426  add esp, 4
.text:00541429  pop edi
.text:0054142A  pop esi
.text:0054142B  pop ebp
.text:0054142C  pop ebx
.text:0054142D  retn
.text:0054142E  push 2 ; Origin
.text:00541430  push 0 ; Offset
.text:00541432  push esi ; File
.text:00541433  call _fseek
.text:00541438  push esi ; File
.text:00541439  call _ftell
.text:0054143E  push 0 ; Origin
.text:00541440  push 0 ; Offset
.text:00541442  push esi ; File
.text:00541443  mov ebp, eax
.text:00541445  call _fseek
.text:0054144A  push ebp ; Size
.text:0054144B  call _malloc
.text:00541450  push esi ; File
.text:00541451  mov ebx, eax
.text:00541453  push 1 ; Count
.text:00541455  push ebp ;
                "ElementSize"
.text:00541456  push ebx ; DstBuf
.text:00541457  call _fread
.text:0054145C  push esi ; File
.text:0054145D  call _fclose
.text:00541462  add esp, 34h
.text:00541465  mov ecx, 3
.text:0054146A  mov edi, offset aQr9_0 ; "QR9"
.text:0054146F  mov esi, ebx
.text:00541471  xor edx, edx
.text:00541473  repe cmpsb
.text:00541475  jz short loc_541489
Let's start from the top. Here is a function that takes two file names and password.
Open the file and report if an error occurs:

```assembly
mov eax, [esp+Str]
push ebp
push offset Mode ; "rb"
push eax ;
push Filename
call _fopen ; open file
mov ebp, eax
add esp, 8
test ebp, ebp
jnz short loc_541348
push offset Format ; "
 Cannot open input file!
"
call _printf
add esp, 4
pop ebp
retn
```

Get the file size via fseek()/ftell():

```assembly
push ebx
push esi
push edi
push 2 ; Origin
push 0 ; Offset
push ebp ; File
; move current file position to the end
call _fseek
push ebp ; File
call _ftell ; get current file position
```
This fragment of code calculates the file size aligned on a 64-byte boundary. This is because this cryptographic algorithm works with only 64-byte blocks. The operation is pretty straightforward: divide the file size by 64, forget about the remainder and add 1, then multiply by 64. The following code removes the remainder as if the value was already divided by 64 and adds 64. It is almost the same.

Allocate buffer with aligned size:

```
.text:00541373 push esi ; Size
.text:00541374 call _malloc
```

Call memset(), e.g., clear the allocated buffer\(^2\).

```
.text:00541379 mov ecx, esi ; allocated buffer pointer -> to EBX
.text:0054137B mov ebx, eax ; allocated buffer pointer -> to EBX
.text:0054137D mov edx, ecx
.text:0054137F xor eax, eax
.text:00541381 mov edi, ebx
.text:00541383 push ebp ; File
.text:00541384 shr ecx, 2
.text:00541387 rep stosd
.text:00541389 mov ecx, edx
.text:0054138B push 1 ; Count
.text:0054138D and ecx, 3
.text:00541390 rep stosb ; memset (buffer, 0, \textbackslashaligned\textunderscore size)
```

Read file via the standard C function fread().

```
.text:00541392 mov eax, [esp+38h+Str]
```

\(^2\) malloc() + memset() could be replaced by calloc()
Call crypt(). This function takes a buffer, buffer size (aligned) and a password string.

Create the output file. By the way, the developer forgot to check if it is was created correctly! The file opening result is being checked, though.

The newly created file handle is in the EDI register now. Write signature “QR9”.

Write the actual file size (not aligned):
Write the encrypted buffer:

```
push    edi ; File
push    1  ; Count
push    esi ; Size
push    ebx ; Str
call    _fwrite ; write
```

Close the file and free the allocated buffer:

```
push    edi ; File
call    _fclose  ; Memory
push    ebx ; Memory
pop     edi
pop     esi
pop     ebx
pop     ebp
retn
```

Here is the reconstructed C code:

```c
void crypt_file(char *fin, char* fout, char *pw) {
    FILE *f;
    int flen, flen_aligned;
    BYTE *buf;

    f=fopen(fin, "rb");
    if (f==NULL) {
        printf ("Cannot open input file!\n");
        return;
    }

    fseek (f, 0, SEEK_END);
    flen=ftell (f);
    fseek (f, 0, SEEK_SET);
    flen_aligned=(flen&0xFFFFFFC0)+0x40;
    ```
The decryption procedure is almost the same:

```
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, 
  void *Src)
.text:00541400 decrypt_file proc near
.text:00541400 Filename = dword ptr 4
.text:00541400 arg_4 = dword ptr 8
.text:00541400 Src = dword ptr 0Ch
.text:00541400
.text:00541400 mov eax, [esp+Filename]
.text:00541404 push ebx
.text:00541405 push ebp
.text:00541406 push esi
.text:00541407 push edi
.text:00541408 push offset aRb ; "rb"
.text:0054140D push eax ;
.text:0054140E call _fopen
.text:00541413 mov esi, eax
.text:00541415 add esp, 8
.text:00541418 test esi, esi
.text:0054141A jnz short loc_54142E
.text:0054141C push offset aCannotOpenIn_0 ;
.text:00541421 call _printf
.text:00541426 add esp, 4
.text:00541429 pop edi
```
Check signature (first 3 bytes):

```
.text:00541462   add    esp, 34h
.text:00541465   mov    ecx, 3
.text:0054146A   mov    edi, offset aQr9_0 ; "QR9"
.text:0054146F   mov    esi, ebx
.text:00541471   xor    edx, edx
.text:00541473   repe cmovb
.text:00541475   jz     short loc_541489
```

Report an error if the signature is absent:

```
.text:00541477   push    offset aFileIsNotCrypt ; "File is not encrypted!\n"
.text:0054147C   call    _printf
.text:00541481   add    esp, 4
.text:00541484   pop     edi
.text:00541485   pop     esi
```
Here is the reconstructed C code:

```c
void decrypt_file(char *fin, char* fout, char *pw) {
    FILE *f;
    int real_flen, flen;
    BYTE *buf;

    f=fopen(fin, "rb");
```
if (f==NULL)
{
    printf ("Cannot open input file!\n");
    return;
};

fseek (f, 0, SEEK_END);
flen=ftell (f);
fseek (f, 0, SEEK_SET);

buf=(BYTE*)malloc (flen);

fread (buf, flen, 1, f);
fclose (f);

if (memcmp (buf, "QR9", 3)!=0)
{
    printf ("File is not encrypted!\n");
    return;
};

memcpy (&real_flen, buf+3, 4);
decrypt (buf+(3+4), flen-(3+4), pw);

f=fopen(fout, "wb");
fwrite (buf+(3+4), real_flen, 1, f);
fclose (f);

free (buf);

OK, now let's go deeper.

Function crypt():

.text:00541260 crypt proc near
.text:00541260 arg_0 = dword ptr 4
.text:00541260 arg_4 = dword ptr 8
.text:00541260 arg_8 = dword ptr 0Ch
.text:00541260 push ebx
.text:00541261 mov ebx, [esp+4+arg_0]
CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

This fragment of code copies a part of the input buffer to an internal array we later name "cube64". The size is in the ECX register. MOVSD stands for move 32-bit dword, so, 16 32-bit dwords are exactly 64 bytes.

```
.text:0054126A  mov  eax, [esp+10h+arg_8]
.text:0054126A  mov  ecx, 10h
.text:00541273  mov  esi, ebx ; EBX is pointer within input buffer
.text:00541275  mov  edi, offset cube64
.text:0054127A  push 1
.text:0054127C  push eax
.text:0054127D  rep movsd
```

Call `rotate_all_with_password()`:

```
.text:0054127F  call  rotate_all_with_password
```

Copy encrypted contents back from "cube64" to buffer:

```
.text:00541284  mov  eax, [esp+18h+arg_4]
.text:00541288  mov  edi, ebx
.text:0054128A  add  ebx, 40h
.text:0054128D  add  esp, 8
.text:00541290  mov  ecx, 10h
.text:00541295  mov  esi, offset cube64
.text:0054129A  add  ebx, 40h ; add 64 to input buffer pointer
.text:0054129D  cmp  ebx, eax ; EBP contain amount of encrypted data.
.text:0054129F  rep movsd
```

If EBP is not bigger that the size input argument, then continue to the next block.

```
.text:005412A1  jl  short loc_54126A
.text:005412A3  pop  edi
.text:005412A4  pop  esi
.text:005412A5  pop  ebp
.text:005412A6  pop  ebx
.text:005412A7  retn
.text:005412A7  crypt   endp
```
Reconstructed crypt() function:

```c
void crypt (BYTE *buf, int sz, char *pw)
{
    int i=0;

    do
    {
        memcpy (cube, buf+i, 8*8);
        rotate_all (pw, 1);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);
}
```

OK, now let's go deeper in function `rotate_all_with_password()`. It takes two arguments: password string and a number. In `crypt()`, the number 1 is used, and in the `decrypt()` function (where `rotate_all_with_password()` function is called too), the number is 3.

```
.text:005411B0  rotate_all_with_password proc near
.text:005411B0
.text:005411B0 arg_0 = dword ptr 4
.text:005411B0 arg_4 = dword ptr 8
.text:005411B0
.text:005411B0 mov eax, [esp+arg_0]
.text:005411B4 push ebp
.text:005411B5 mov ebp, eax

Check the current character in the password. If it is zero, exit:

```
.text:005411B7 cmp byte ptr [eax], 0
.text:005411BA jz exit
.text:005411C0 push ebx
.text:005411C1 mov ebx, [esp+8+arg_4]
.text:005411C5 push esi
.text:005411C6 push edi
.text:005411C7 loop_begin:

Call tolower(), a standard C function.
```
```c
.movsx eax, byte ptr [ebp+0]
.push eax
.call _tolower
.add esp, 4
```
Hmm, if the password contains non-Latin character, it is skipped! Indeed, when we run the encryption utility and try non-Latin characters in the password, they seem to be ignored.

```
.text:005411D4 cmp al, 'a'
.text:005411D6 jl short ⤇
.text:005411D8 cmp al, 'z'
.text:005411DA jg short ⤇
.text:005411DC movsx ecx, al
```

Subtract the value of “a” (97) from the character.

```
.text:005411DF sub ecx, 'a' ; 97
```

After subtracting, we’ll get 0 for “a” here, 1 for “b”, etc. And 25 for “z”.

```
.text:005411E2 cmp ecx, 24
.text:005411E5 jle short skip_subtracting
.text:005411E7 sub ecx, 24
```

It seems, “y” and “z” are exceptional characters too. After that fragment of code, “y” becomes 0 and “z” –1. This implies that the 26 Latin alphabet symbols become values in the range of 0..23, (24 in total).

```
.text:005411EA
.text:005411EA skip_subtracting:
      ; CODE ⤇
      ⇦ XREF: rotate_all_with_password+35
```

This is actually division via multiplication. You can read more about it in the “Division by 9” section (41 on page 676).

The code actually divides the password character’s value by 3.

```
.text:005411EA mov eax, 55555556h
.text:005411EF imul ecx
.text:005411F1 mov eax, edx
.text:005411F3 shr eax, 1Fh
.text:005411F6 add edx, eax
.text:005411F8 mov eax, ecx
.text:005411FA mov esi, edx
.text:005411FC mov ecx, 3
.text:00541201 cdq
.text:00541202 idiv ecx
```

EDX is the remainder of the division.
CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

If the remainder is 2, call rotate3(). EDI is the second argument of the rotate_all_with_password() function. As we already noted, 1 is for the encryption operations and 3 is for the decryption. So, here is a loop. When encrypting, rotate1/2/3 are to be called the same number of times as given in the first argument.

```
.text:00541215 call_rotate3:  
.text:00541215 push esi 
.text:00541216 call rotate3 
.text:0054121B add esp, 4 
.text:0054121E dec edi 
.text:0054121F jnz short call_rotate3 
.text:00541221 jmp short loc_541229 

.text:00541223 call_rotate2:  
.text:00541223 test ebx, ebx 
.text:00541225 jle short loc_541229 

.text:00541227 mov edi, ebx 

.loc_541229:  
.push esi 
.call rotate2 
.add esp, 4 
.dec edi 
.jnz short loc_541229 
jmp short loc_541229 

.text:00541237 call_rotate1:  
.text:00541237 test ebx, ebx 
.text:00541239 jle short loc_541229 

.loc_541229:  
.mov edi, ebx 
```
Fetch the next character from the password string.

```
.mov    al, [ebp+1]
```

Increment the character pointer in the password string:

```
.inc    ebp
.test   al, al
.jnz    loop_begin
.pop    edi
.pop    esi
.pop    ebx
.exit:
.pop    ebp
.retn
.rotate_all_with_password endp
```

Here is the reconstructed C code:

```c
void rotate_all (char *pwd, int v)
{
    char *p=pwd;

    while (*p)
    {
        char c=*p;
        int q;

        c=tolower (c);

        if (c>='a' && c<='z')
        {
            q=c-'a';
            if (q>24)
                q-=24;
            int quotient=q/3;
```
Now let's go deeper and investigate the rotate1/2/3 functions. Each function calls another two functions. We eventually will name them set_bit() and get_bit().

Let's start with get_bit():

```
.text:00541050 get_bit proc near
.text:00541050 .text:00541050 arg_0 = dword ptr 4
.text:00541050 .text:00541050 arg_4 = dword ptr 8
.text:00541050 .text:00541050 arg_8 = byte ptr 0Ch
.text:00541050 .text:00541050 arg_C = byte ptr 10h
.text:00541050 mov eax, [esp+arg_4]
.text:00541054 mov ecx, [esp+arg_0]
.text:00541058 mov al, cube64[eax+ecx*8]
.text:0054105F mov cl, [esp+arg_8]
.text:00541063 shr al, cl
.text:00541065 and al, 1
.text:00541067 retn
.text:00541067 get_bit endp
```

...in other words: calculate an index in the cube64 array: arg_4 + arg_0 * 8. Then shift a byte from the array by arg_8 bits right. Isolate the lowest bit and return it.

Let's see another function, set_bit():

```
.text:00541000 set_bit proc near
.text:00541000 .text:00541000 arg_0 = dword ptr 4
.text:00541000 .text:00541000 arg_4 = dword ptr 8
.text:00541000 .text:00541000 arg_8 = dword ptr 0Ch
.text:00541000 .text:00541000 arg_C = byte ptr 10h
```
The value in the DL is 1 here. It gets shifted left by arg_8. For example, if arg_8 is 4, the value in the DL register is to be 0x10 or 1000b in binary form.

Get a bit from array and explicitly set it.

Store it back:

If arg_C is not zero...

...invert DL. For example, if DL's state after the shift was 0x10 or 1000b in binary form, there is 0xEF to be after the NOT instruction (or 11101111b in binary form).

This instruction clears the bit, in other words, it saves all bits in CL which are also set in DL except those in DL which are cleared. This implies that if DL is 11101111b in binary form, all bits are to be saved except the 5th (counting from lowest bit).
It is almost the same as `get_bit()`, except, if `arg_C` is zero, the function clears the specific bit in the array, or sets it otherwise.

We also know that the array's size is 64. The first two arguments both in the `set_bit()` and `get_bit()` functions could be seen as 2D coordinates. Then the array is to be an 8*8 matrix.

Here is a C representation of what we know up to now:

```c
#define IS_SET(flag, bit) ((flag) & (bit))
#define SET_BIT(var, bit) ((var) |= (bit))
#define REMOVE_BIT(var, bit) ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int shift, int bit)
{
    if (bit)
        SET_BIT (cube[x][y], 1<<shift);
    else
        REMOVE_BIT (cube[x][y], 1<<shift);
};

bool get_bit (int x, int y, int shift)
{
    if ((cube[x][y]>>shift)&1==1)
        return 1;
    return 0;
};
```

Now let's get back to the rotate1/2/3 functions.

```assembly
.text:00541070 rotate1 proc near
.text:00541070
```

Internal array allocation in the local stack, with size of 64 bytes:

```assembly
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0 = dword ptr 4
.text:00541070
.text:00541070 sub esp, 40h
.text:00541073 push ebx
.text:00541074 push ebp
```

1141
EBX is a pointer to the internal array:

```
.text:0054107D   lea   ebx, [esp+50h+internal_array_64]
.text:00541081
```

Here we have two nested loops:

```
.text:00541081   first_loop1_begin:
.text:00541081   xor   esi, esi    ; ESI is loop 2 counter
.text:00541083
.text:00541083   first_loop2_begin:
.text:00541083   push  ebp   ; arg_0
.text:00541084   push  esi   ; loop 1 counter
.text:00541085   push  edi   ; loop 2 counter
.text:00541086   call  get_bit
.text:0054108B   add   esp, 0Ch
.text:0054108E   mov  [ebx+esi], al  ; store to internal array
.text:00541091   inc   esi    ; increment loop 1 counter
.text:00541092   cmp   esi, 8
.text:00541095   jl    short first_loop2_begin
.text:00541097   inc   edi    ; increment loop 2 counter
.text:00541098   add   ebx, 8
.text:0054109B   cmp   edi, 8
.text:0054109E   jl    short first_loop1_begin
```

...we see that both loops’ counters are in the range of 0..7. Also they are used as the first and second argument for the get_bit() function. The third argument to get_bit() is the only argument of rotate1(). The return value from get_bit() is placed in the internal array.

Prepare a pointer to the internal array again:

```
.text:005410A0   lea   ebx, [esp+50h+internal_array_64]
.text:005410A4   mov  edi, 7    ; EDI is loop 1 counter, initial state is 7
```

1142
...this code is placing the contents of the internal array to the cube global array via the `set_bit()` function, but in a different order! Now the counter of the first loop is in the range of 7 to 0, decrementing at each iteration!

The C code representation looks like:

```c
void rotate1 (int v)
{
    bool tmp[8][8]; // internal array
    int i, j;

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            tmp[i][j]=get_bit (i, j, v);

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            tmp[i][j]=get_bit (i, j, v);
}
```
set_bit (j, 7-i, v, tmp[x][y]);

Not very understandable, but if we take a look at \texttt{rotate2()} function:

\begin{verbatim}
.text:005410E0 rotate2 proc near
.text:005410E0
.text:005410E0 internal_array_64 = byte ptr -40h
.text:005410E0 arg_0 = dword ptr 4
.text:005410E0
.text:005410E0 sub esp, 40h
.text:005410E3 push ebx
.text:005410E4 push ebp
.text:005410E5 mov ebp, [esp+48h+arg_0]
.text:005410E9 push esi
.text:005410EA push edi
.text:005410EB xor edi, edi ; loop 1 counter
.text:005410ED lea ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:
.text:005410F1 xor esi, esi ; loop 2 counter
.text:005410F3
.text:005410F3 loc_5410F3:
.text:005410F3 push esi ; loop 2 counter
.text:005410F4 push edi ; loop 1 counter
.text:005410F5 push ebp ; arg_0
.text:005410F6 call get_bit
.text:005410FB add esp, 0Ch
.text:005410FE mov [ebx+esi], al ; store to internal
array
.text:00541101 inc esi ; increment loop 1 counter
.text:00541102 cmp esi, 8
.text:00541105 jl short loc_5410F3
.text:00541107 inc edi ; increment loop 2 counter
.text:00541108 add ebx, 8
.text:0054110B cmp edi, 8
.text:0054110E jl short loc_5410F1
.text:00541110 lea ebx, [esp+50h+internal_array_64]
.text:00541114 mov edi, 7 ; loop 1 counter is initial state 7
.text:00541119
.text:00541119 loc_541119:
.text:00541119 xor esi, esi ; loop 2 counter
.text:0054111B
.text:0054111B loc_54111B:
.text:0054111B mov al, [ebx+esi] ; get byte from
\end{verbatim}
It is almost the same, except the order of the arguments of the `get_bit()` and `set_bit()` is different. Let's rewrite it in C-like code:

```c
void rotate2 (int v)
{
    bool tmp[8][8]; // internal array
    int i, j;

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            tmp[i][j]=get_bit (v, i, j);

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            set_bit (v, j, 7-i, tmp[i][j]);
}
```

Let's also rewrite the `rotate3()` function:

```c
void rotate3 (int v)
{
    bool tmp[8][8];
    int i, j;

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
            set_bit (v, j, 7-i, tmp[i][j]);
}
```
Well, now things are simpler. If we consider cube64 as a 3D cube of size 8*8*8, where each element is a bit, get_bit() and set_bit() take just the coordinates of a bit as input.

The rotate1/2/3 functions are in fact rotating all bits in a specific plane. These three functions are one for each cube side and the v argument sets the plane in the range of 0..7.

Maybe, the algorithm's author was thinking of an 8*8*8 Rubik's cube 3?! Yes, indeed.

Let's look closer into the decrypt() function, here is its rewritten version:

```c
void decrypt (BYTE *buf, int sz, char *pw)
{
    char *p=strdup (pw);
    strrev (p);
    int i=0;

    do
    {
        memcpy (cube, buf+i, 8*8);
        rotate_all (p, 3);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);

    free (p);
}
```

It is almost the same as for crypt(), but the password string is reversed by the strrev() 4 standard C function and rotate_all() is called with argument 3.

This implies that in case of decryption, each corresponding rotate1/2/3 call is to be performed thrice.

3 wikipedia
4 MSDN
This is almost as in Rubik’s cube! If you want to get back, do the same in reverse order and direction! If you need to undo the effect of rotating one place in clockwise direction, rotate it once in counter-clockwise direction, or thrice in clockwise direction.

rotate1() is apparently for rotating the “front” plane. rotate2() is apparently for rotating the “top” plane. rotate3() is apparently for rotating the “left” plane.

Let’s get back to the core of the rotate_all() function:

```c
q=c-'a';
if (q>24)
  q-=24;

int quotient=q/3; // in range 0..7
int remainder=q % 3;

switch (remainder)
{
  case 0: for (int i=0; i<v; i++) rotate1 (quotient); break; // front
  case 1: for (int i=0; i<v; i++) rotate2 (quotient); break; // top
  case 2: for (int i=0; i<v; i++) rotate3 (quotient); break; // left
}
```

Now it is much simpler to understand: each password character defines a side (one of three) and a plane (one of 8). 3*8 = 24, that is why two the last two characters of the Latin alphabet are remapped to fit an alphabet of exactly 24 elements.

The algorithm is clearly weak: in case of short passwords you can see that in the encrypted file there are the original bytes of the original file in a binary file editor.

Here is the whole source code reconstructed:

```c
#include <windows.h>
#include <stdio.h>
#include <assert.h>

#define IS_SET(flag, bit) ((flag) & (bit))
#define SET_BIT(var, bit) ((var) |= (bit))
#define REMOVE_BIT(var, bit) ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int z, bool bit) {
```
CHAPTER 79. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

if (bit)
    SET_BIT (cube[x][y], 1<<z);
else
    REMOVE_BIT (cube[x][y], 1<<z);
};

bool get_bit (int x, int y, int z)
{
    if ((cube[x][y]>>z)&1==1)
        return true;
    return false;
};

void rotate_f (int row)
{
    bool tmp[8][8];
    int x, y;

    for (x=0; x<8; x++)
        for (y=0; y<8; y++)
            tmp[x][y]=get_bit (x, y, row);

    for (x=0; x<8; x++)
        for (y=0; y<8; y++)
            set_bit (y, 7-x, row, tmp[x][y]);
};

void rotate_t (int row)
{
    bool tmp[8][8];
    int y, z;

    for (y=0; y<8; y++)
        for (z=0; z<8; z++)
            tmp[y][z]=get_bit (row, y, z);

    for (y=0; y<8; y++)
        for (z=0; z<8; z++)
            set_bit (row, z, 7-y, tmp[y][z]);
};

void rotate_l (int row)
{
    bool tmp[8][8];
    int x, z;

    for (x=0; x<8; x++)
        for (z=0; z<8; z++)
            tmp[x][z]=get_bit (row, x, z);

    for (x=0; x<8; x++)
        for (z=0; z<8; z++)
            set_bit (row, 7-x, z, tmp[x][z]);
};
for (z=0; z<8; z++)
    tmp[x][z]=get_bit (x, row, z);

for (x=0; x<8; x++)
    for (z=0; z<8; z++)
        set_bit (7-z, row, x, tmp[x][z]);
}

void rotate_all (char *pwd, int v)
{
    char *p=pwd;

    while (*p)
    {
        char c=*p;
        int q;

        c=tolower (c);

        if (c>='a' && c<='z')
        {
            q=c-'a';
            if (q>24)
            {
                q-=24;
            }

            int quotient=q/3;
            int remainder=q % 3;

            switch (remainder)
            {
            case 0: for (int i=0; i<v; i++)
                    
                    rotate_f (quotient); break;
            case 1: for (int i=0; i<v; i++)
                    
                    rotate_t (quotient); break;
            case 2: for (int i=0; i<v; i++)
                    
                    rotate_l (quotient); break;
            }
            
            p++;
        }

    }
}

void crypt (BYTE *buf, int sz, char *pw)
{
    int i=0;
 CHAPTER 79. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

void decrypt (BYTE *buf, int sz, char *pw)
{
    char *p=strdup (pw);
    strrev (p);
    int i=0;

    do
    {
        memcpy (cube, buf+i, 8*8);
        rotate_all (p, 3);
        memcpy (buf+i, cube, 8*8);
        i+=64;
    }
    while (i<sz);

    free (p);
}

void crypt_file(char *fin, char* fout, char *pw)
{
    FILE *f;
    int flen, flen_aligned;
    BYTE *buf;

    f=fopen(fin, "rb");
    if (f==NULL)
    {
        printf ("Cannot open input file!\n");
        return;
    }

    fseek (f, 0, SEEK_END);
    flen=ftell (f);
    fseek (f, 0, SEEK_SET);

    flen_aligned=(flen&0xFFFFFFFC0)+0x40;

buf=(BYTE*)malloc (flen_aligned);
memset (buf, 0, flen_aligned);

fread (buf, flen, 1, f);
fclose (f);
crypt (buf, flen_aligned, pw);
f=fopen(fout, "wb");
fwrite ("QR9", 3, 1, f);
fwrite (&flen, 4, 1, f);
fwrite (buf, flen_aligned, 1, f);
fclose (f);
free (buf);

void decrypt_file(char *fin, char* fout, char *pw) {
    FILE *f;
    int real_flen, flen;
    BYTE *buf;

    f=fopen(fin, "rb");
    if (f==NULL) {
    printf ("Cannot open input file!\n");
    return;
    }

    fseek (f, 0, SEEK_END);
    flen=ftell (f);
    fseek (f, 0, SEEK_SET);
    buf=(BYTE*)malloc (flen);
    fread (buf, flen, 1, f);
    fclose (f);
    if (memcmp (buf, "QR9", 3)!=0)
```c
{   printf("File is not encrypted!\n");
    return;
};

memcpy (&real_flen, buf+3, 4);
decrypt (buf+(3+4), flen-(3+4), pw);
f=fopen(fout, "wb");
fwrite (buf+(3+4), real_flen, 1, f);
close (f);
free (buf);
};

// run: input output 0/1 password
// 0 for encrypt, 1 for decrypt

int main(int argc, char *argv[])
{
    if (argc!=5)
    {
        printf("Incorrect parameters!\n");
        return 1;
    };
    if (strcmp (argv[3], "0") == 0)
        crypt_file (argv[1], argv[2], argv[4]);
    else
        if (strcmp (argv[3], "1") == 0)
            decrypt_file (argv[1], argv[2], argv[4]);
        else
            printf("Wrong param %s\n", argv[3]);
    return 0;
};
```
Chapter 80

SAP

80.1 About SAP client network traffic compression

(Tracing the connection between the TDW_NOCOMPRESS SAPGUI\(^1\) environment variable and the pesky annoying pop-up window and the actual data compression routine.)

It is known that the network traffic between SAPGUI and SAP is not encrypted by default, but compressed (see here\(^2\) and here\(^3\)).

It is also known that by setting the environment variable *TDW_NOCOMPRESS* to 1, it is possible to turn the network packet compression off.

But you will see a annoying pop-up window that cannot be closed:

\(^1\)SAP GUI client  
\(^2\)http://go.yurichev.com/17221  
\(^3\)blog.yurichev.com
Let's see if we can remove the window somehow.

But before this, let's see what we already know. First: we know that the environment variable `TDW_NOCOMPRESS` is checked somewhere inside the SAPGUI client. Second: a string like “data compression switched off” must be present somewhere in it. With the help of the FAR file manager we can find that both of these strings are stored in the SAPguilib.dll file.

So let's open SAPguilib.dll in IDA and search for the “TDW_NOCOMPRESS” string. Yes, it is present and there is only one reference to it.

We see the following fragment of code (all file offsets are valid for SAPGUI 720 win32, SAPguilib.dll file version 7200,1,0,9009):

```assembly
.text:6440D51B lea eax, [ebp+2108h+var_211C]
.text:6440D51E push eax ; int
.text:6440D51F push offset aTdw_nocompress ;
=text:6440D524 mov byte ptr [edi+15h], 0
=text:6440D528 call chk_env
=text:6440D52D pop ecx
=text:6440D52E pop ecx
=text:6440D52F push offset byte_64443AF8
=text:6440D534 lea ecx, [ebp+2108h+var_211C]
=text:6440D537 ; demangled name: int ATL::CStringT::Compare(char const*)const
=text:6440D537 call ds:mfc90_1603
```

*http://go.yurichev.com/17347*
The string returned by \texttt{chk\_env()} via its second argument is then handled by the MFC string functions and then \texttt{atoi()}\textsuperscript{5} is called. After that, the numerical value is stored in \texttt{edi+15h}.

Also take a look at the \texttt{chk\_env()} function (we gave this name to it manually):

\texttt{\textbackslash .text:64413F20 ; int \_cdecl chk\_env(char *VarName, int)}
\texttt{\textbackslash .text:64413F20 chk\_env proc near}
\texttt{\textbackslash .text:64413F20 DstSize = dword ptr -0Ch}
\texttt{\textbackslash .text:64413F20 var\_8 = dword ptr -8}
\texttt{\textbackslash .text:64413F20 DstBuf = dword ptr -4}
\texttt{\textbackslash .text:64413F20 VarName = dword ptr 8}
\texttt{\textbackslash .text:64413F20 arg\_4 = dword ptr 0Ch}
\texttt{\textbackslash .text:64413F20 \textbackslash .text:64413F20 push ebp}
\texttt{\textbackslash .text:64413F21 mov ebp, esp}
\texttt{\textbackslash .text:64413F23 sub esp, 0Ch}
\texttt{\textbackslash .text:64413F26 mov [ebp+DstSize], 0}
\texttt{\textbackslash .text:64413F2D mov [ebp+DstBuf], 0}
\texttt{\textbackslash .text:64413F34 push offset unk_6444C88C}
\texttt{\textbackslash .text:64413F39 mov ecx, [ebp+arg\_4]}

\texttt{\textbackslash ; (demangled name) ATL::CStringT::operator=(char const *)}
\texttt{\textbackslash .text:64413F3C call ds:mfc90_820}
\texttt{\textbackslash .text:64413F42 mov eax, [ebp+VarName]}
\texttt{\textbackslash ; VarName}
\texttt{\textbackslash .text:64413F46 mov ecx, [ebp+DstSize]}
\texttt{\textbackslash .text:64413F49 push ecx}

\textsuperscript{5}standard C library function that converts the digits in a string to a number
mov edx, [ebp+DstBuf]
push edx ; DstBuf
lea eax, [ebp+DstSize]
push eax
push DstSize
lea DstBuf
push DstSize
call ds:getenv_s
add esp, 10h
mov [ebp+var_8], eax
cmp [ebp+var_8], 0
jz short loc_64413F68
xor eax, eax
jmp short loc_64413FBC
loc_64413F68:
cmp [ebp+DstSize], 0
jnz short loc_64413F72
xor eax, eax
jmp short loc_64413FBC
loc_64413F72:
mov ecx, [ebp+DstSize]
push ecx
mov ecx, [ebp+arg_4]
call ds:mfc90_2691
mov [ebp+DstBuf], eax
mov edx, [ebp+VarName]
push edx
mov eax, [ebp+DstSize]
push eax
mov ecx, [ebp+DstBuf]
push ecx ; DstBuf
lea edx, [ebp+DstSize]
push edx
call ds:getenv_s
add esp, 10h
mov [ebp+var_8], eax
push 0FFFFFFFFh
mov ecx, [ebp+arg_4]
call ds:mfc90_5835

; demangled name: ATL::CSimpleStringT<char, 1>::Preallocate(int)

; demangled name: ATL::CSimpleStringT::ReleaseBuffer(int)
Yes. The `getenv_s()`\(^6\) function is a Microsoft security-enhanced version of `getenv()`\(^7\). There are also some MFC string manipulations.

Lots of other environment variables are checked as well. Here is a list of all variables that are being checked and what SAPGUI would write to its trace log when logging is turned on:

\(^6\)MSDN
\(^7\)Standard C library returning environment variable
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPTTRACE</td>
<td>“GUI-OPTION: Trace set to %d”</td>
</tr>
<tr>
<td>TDW_HEXDUMP</td>
<td>“GUI-OPTION: Hexdump enabled”</td>
</tr>
<tr>
<td>TDW_WORKDIR</td>
<td>“GUI-OPTION: working directory ‘%s’”</td>
</tr>
<tr>
<td>TDW_SPLASHSCREENOFF</td>
<td>“GUI-OPTION: Splash Screen Off” / “GUI-OPTION: Splash Screen On”</td>
</tr>
<tr>
<td>TDW_REPLYTIMEOUT</td>
<td>“GUI-OPTION: reply timeout %d milliseconds”</td>
</tr>
<tr>
<td>TDW_PLAYBACKTIMEOUT</td>
<td>“GUI-OPTION: PlaybackTimeout set to %d milliseconds”</td>
</tr>
<tr>
<td>TDW_NOCOMPRESS</td>
<td>“GUI-OPTION: no compression read”</td>
</tr>
<tr>
<td>TDW_EXPERT</td>
<td>“GUI-OPTION: expert mode”</td>
</tr>
<tr>
<td>TDW_PLAYBACKPROGRESS</td>
<td>“GUI-OPTION: PlaybackProgress”</td>
</tr>
<tr>
<td>TDW_PLAYBACKNETTRAFFIC</td>
<td>“GUI-OPTION: PlaybackNetTraffic”</td>
</tr>
<tr>
<td>TDW_PLAYLOG</td>
<td>“GUI-OPTION: /PlayLog is YES, file %s”</td>
</tr>
<tr>
<td>TDW_PLAYTIME</td>
<td>“GUI-OPTION: /PlayTime set to %d milliseconds”</td>
</tr>
<tr>
<td>TDW_LOGFILE</td>
<td>“GUI-OPTION: TDW_LOGFILE ‘%s’”</td>
</tr>
<tr>
<td>TDW_WAN</td>
<td>“GUI-OPTION: WAN - low speed connection enabled”</td>
</tr>
<tr>
<td>TDW_FULLMENU</td>
<td>“GUI-OPTION: FullMenu enabled”</td>
</tr>
<tr>
<td>SAP_CP / SAP_CODEPAGE</td>
<td>“GUI-OPTION: SAP_CODEPAGE ‘%d’”</td>
</tr>
<tr>
<td>UPDOWNLOAD_CP</td>
<td>“GUI-OPTION: UPDOWNLOAD_CP ‘%d’”</td>
</tr>
<tr>
<td>SNC_PARTNERNAME</td>
<td>“GUI-OPTION: SNC name ‘%s’”</td>
</tr>
<tr>
<td>SNC_QOP</td>
<td>“GUI-OPTION: SNC_QOP ‘%s’”</td>
</tr>
<tr>
<td>SNC_LIB</td>
<td>“GUI-OPTION: SNC is set to: %s”</td>
</tr>
<tr>
<td>SAPGUI_INPLACE</td>
<td>“GUI-OPTION: environment variable SAPGUI_INPLACE is on”</td>
</tr>
</tbody>
</table>

The settings for each variable are written in the array via a pointer in the EDI register. EDI is set before the function call:

```
.text:6440EE00 lea edi, [ebp+2884h+var_2884]
.text:6440EE03 lea ecx, [esi+24h]
.text:6440EE06 call load_command_line
.text:6440EE0B mov edi, eax
.text:6440EE0D xor ebx, ebx
.text:6440EE0F cmp edi, ebx
.text:6440EE11 jz short loc_6440EE42
.text:6440EE13 push edi
.text:6440EE14 push offset aSapguiStoppedA
.text:6440EE19 push dword_644F93E8
.text:6440EE1F call FEWTraceError
```

Now, can we find the “data record mode switched on” string? Yes, and the only reference is in CDwsGui::PrepareInfoWindow(). How do we get know the class/method names? There are a lot of special debugging calls that write to the log files, like:

```
.text:64405160 push dword ptr [esi+2854h]
```
It is very useful.

So let's see the contents of this pesky annoying pop-up window's function:

```
拙自：CDwsGui::PrepareInfoWindow

拙自：pvParam = byte ptr -3Ch
拙自：var_38 = dword ptr -38h
拙自：var_34 = dword ptr -34h
拙自：rc = tagRECT ptr -2Ch
拙自：cy = dword ptr -1Ch
拙自：h = dword ptr -18h
拙自：var_14 = dword ptr -14h
拙自：var_10 = dword ptr -10h
拙自：var_4 = dword ptr -4
拙自：push 30h
拙自：mov eax, offset loc_64438E00
拙自：call __EH_prolog3
拙自：mov [ebp+var_4], ecx
拙自：lea edi, [esi+2854h]
拙自：push offset aEnvironmentInf
拙自：mov ecx, edi
```

...or:

```
拙自：CClient::Start: set shortcut user to '\%'...

拙自：push eax
拙自：push offset aClientStart_6
拙自：push dword ptr [esi+2848h]
拙自：call dbg
拙自：add esp, 0Ch
```

It is very useful.

So let's see the contents of this pesky annoying pop-up window's function:
; demangled name: ATL::CStringT::operator=(char const *)
.text:64404F7B call ds:mfc90_820
.text:64404F81 cmp [esi+38h], ebx
.text:64404F84 mov ebx, ds:mfc90_2539
.text:64404F8A jbe short loc_64404FA9
.text:64404F8C push dword ptr [esi+34h]
.text:64404F8F lea eax, [ebp+var_14]
.text:64404F92 push offset aWorkingDirecto ;
 ↯ "working directory: '%s'
"
.text:64404F97 push eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404F98 call ebx ; mfc90_2539
.text:64404F9A add esp, 0Ch
.text:64404F9D lea eax, [ebp+var_14]
.text:64404FA0 push eax
.text:64404FA1 mov ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::
 ↯ CSimpleStringT<char, 1> const &)
.text:64404FA3 call ds:mfc90_941
.text:64404FA9 loc_64404FA9:
.text:64404FA9 mov eax, [esi+38h]
.text:64404FAC test eax, eax
.text:64404FAC jbe short loc_64404FD3
.text:64404FBD push eax
.text:64404FBE lea eax, [ebp+var_14]
.text:64404FBB push offset aTraceLevelDAct ;
 ↯ "trace level \%d activated\n"
.text:64404FBD push eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404FBA call ebx ; mfc90_2539
.text:64404FBC add esp, 0Ch
.text:64404FBE lea eax, [ebp+var_14]
.text:64404FBD push eax
.text:64404FCC mov ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::
 ↯ CSimpleStringT<char, 1> const &)
.text:64404FCD call ds:mfc90_941
.text:64404FCE xor ebx, ebx
.text:64404FCF inc ebx
.text:64404FD0 mov [ebp+var_10], ebx
.text:64404FD1 jmp short loc_64404FD6
.text:64404FD3
xor    ebx, ebx
inc    ebx

cmp    [esi+38h], ebx
jbe    short loc_64404FF1

cmp    dword ptr [esi+2978h], 0
jz     short loc_64404FF1

push   offset aHexdumpInTrace
"hexdump in trace activated\n"

mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
call   ds:mfc90_945

loc_64404FF1:

cmp    byte ptr [esi+78h], 0
jz     short loc_64405007

push   offset aLoggingActivat
"logging activated\n"

mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
call   ds:mfc90_945

loc_64405007:

cmp    byte ptr [esi+3Dh], 0
jz     short bypass

push   offset aDataCompressio
"data compression switched off\n"

mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
call   ds:mfc90_945

loc_6440501D:
bypass:

mov    eax, [esi+20h]
test   eax, eax

jz     short loc_6440503A

cmp    dword ptr [eax+28h], 0
jz     short loc_6440503A

push   offset aDataRecordMode
"data record mode switched on\n"
For maximum data security delete the s"...
CHAPTER 80. SAP

.text:6440509F  push  [ebp+var_10] ; hdc
.text:644050A2  call  edi ; SelectObject
.text:644050A4  and  [ebp+rc.left], 0
.text:644050A8  and  [ebp+rc.top], 0
.text:644050AC  mov  [ebp+h], eax
.text:644050AF  push  401h ; format
.text:644050B4  lea  eax, [ebp+rc]
.text:644050B7  push  eax ; lprc
.text:644050B8  lea  ecx, [esi+2854h]
.text:644050BE  mov  [ebp+rc.right], ebx
.text:644050C1  mov  [ebp+rc.bottom], 0B4h

; demangled name: ATL::CSimpleStringT::GetLength(void)
.text:644050C8  call  ds:mfc90_3178
.text:644050CE  push  eax ; cchText
.text:644050CF  lea  ecx, [esi+2854h]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:644050D5  call  ds:mfc90_910
.text:644050DB  push  eax ; lpchText
.text:644050DC  push  [ebp+var_10] ; hdc
.text:644050DF  call  ds:DrawTextA
.text:644050E5  push  4 ; nIndex
.text:644050ED  mov  ecx, [ebp+rc.bottom]
.text:644050E7  call  ds:GetSystemMetrics
.text:644050ED  mov  ecx, [ebp+rc.top]
.text:644050F3  cmp  [ebp+h], 0
.text:644050F7  lea  eax, [eax+ecx+28h]
.text:644050FB  mov  [ebp+cy], eax
.text:644050FE  jz  short loc_64405108
.text:64405100  push  [ebp+h] ; h
.text:64405103  push  [ebp+var_10] ; hdc
.text:64405106  call  edi ; SelectObject
.text:64405108  loc_64405108:
.text:64405108  push  [ebp+var_10] ; hDC
.text:6440510B  push  0 ; hWnd
.text:6440510D  call  ds:ReleaseDC
.text:64405113  loc_64405113:
.text:64405113  mov  eax, [ebp+var_38]
.text:64405116  push  80h ; uFlags
.text:6440511B  push  [ebp+cy] ; cy
.text:6440511E  inc  eax
push ebx ; cx
push eax ; Y
mov eax, [ebp+var_34]
add eax, 0FFFFFED4h
cdq
sub eax, edx
sar eax, 1
push eax ; X
push 0
push dword ptr [esi+285Ch] ; hWndInsertAfter
push dword ptr [esi+2854h]
call ds:SetWindowPos
xor ebx, ebx
inc ebx
jmp short loc_6440514D
push offset byte_64443AF8 ; demangled name: ATL::CStringT::operator=(char const *)
call ds:mfc90_820
push dword ptr [esi+2848h] ; aCdwsguiPrepare
push dword ptr [esi+2854h]
call dbg
add esp, 0Ch
mov dword_644F858C, ebx
push dword_644F858C, 2
push offset aCdwsguiPrepare ; "\nCDwsGui::PrepareInfoWindow: sapgui env"
push dword ptr [esi+2848h]
call dbg
add esp, 0Ch
call sub_6441C910
mov dword_644F858C, ebx
call sub_6441C920
or [ebp+var_4], 0FFFFFFFFh
lea ecx, [ebp+var_14]
call ds:mfc90_601
call __EH_epilog3
retn
CDwsGui__PrepareInfoWindow endp
At the start of the function ECX contains a pointer to the object (since it is a this-call (51.1.1 on page 757)-type of function). In our case, the object obviously has class type of CDwsGui. Depending on the option turned on in the object, a specific message part is to be concatenated with the resulting message.

If the value at address this+0x3D is not zero, the compression is off:

```
.text:64405007  loc_64405007:
.text:64405007  cmp     byte ptr [esi+3Dh], 0
.text:6440500B  jz      short bypass
.text:6440500D  push    offset aDataCompression;
  "data compression switched off"
.text:64405012  mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014  call    ds:mfc90_945
.text:6440501A  mov     [ebp+var_10], ebx
.text:6440501D  bypass:
```

It is interesting that finally the var_10 variable state defines whether the message is to be shown at all:

```
.text:6440503C  cmp     [ebp+var_10], ebx
.text:6440503F  jnz     exit; bypass drawing

; add strings "For maximum data security delete" / "the setting(s) as soon as possible!":

.text:64405045  push    offset aForMaximumData;\
  "\nFor maximum data security delete\nthe s"
.text:6440504A  call    ds:mfc90_945; ATL::\
  CStringT::operator+=(char const *)
.text:64405050  xor     edi, edi
.text:64405052  push    edi
  fWinIni
.text:64405053  lea     eax, [ebp+pvParam]
.text:64405056  push    eax
  pvParam
.text:64405057  push    edi
  uiParam
.text:64405058  push    30h
  uiAction
.text:6440505A  call    ds:SystemParametersInfoA
.text:64405060  mov     eax, [ebp+var_34]
.text:64405063  cmp     eax, 1600
.text:64405068  jle     short loc_64405072
```
Let's check our theory on practice.

JNZ at this line ...

... replace it with just JMP, and we get SAPGUI working without the pesky annoying pop-up window appearing!

Now let's dig deeper and find a connection between the 0x15 offset in the load_command_line() function (we gave it this name) and the this+0x3D variable in CDwsGui::PrepareInfoWindow. Are we sure the value is the same?

We are starting to search for all occurrences of the 0x15 value in code. For a small programs like SAPGUI, it sometimes works. Here is the first occurrence we've got:

```assembly
.sub_64404C19 proc near
    arg_0 = dword ptr 4
    push ebx
    push ebp
    push esi
    push edi
    mov edi, [esp+10h+arg_0]
    mov eax, [edi]
    mov esi, ecx ; ESI/ECX are pointers to some unknown object.
    mov [esi], eax
    mov eax, [edi+4]
    mov [esi+4], eax
    mov eax, [edi+8]
    mov [esi+8], eax
    lea eax, [edi+0Ch]
    push eax
```

Next, we will continue with the explanation and analysis of the code.
The function was called from the function named CDwsGui::CopyOptions! And thanks again for debugging information.

But the real answer is in CDwsGui::Init():

Finally, we understand: the array filled in the load_command_line() function is actually placed in the CDwsGui class, but at address this+0x28. 0x15 + 0x28 is exactly 0x3D. OK, we found the point where the value is copied to.

Let’s also find the rest of the places where the 0x3D offset is used. Here is one of them in the CDwsGui::SapguiRun function (again, thanks to the debugging calls):

; demangled name: const char* ATL::CSimpleStringT::operator = (class ATL::CSimpleStringT & ...)
Let's check our findings. Replace the `setz al` here with the `xor eax, eax` / `nop` instructions, clear the TDW_NOCOMPRESS environment variable and run SAPGUI. Wow! There pesky annoying window is no more (just as expected, because the variable is not set) but in Wireshark we can see that the network packets are not compressed anymore! Obviously, this is the point where the compression flag is to be set in the `CConnectionContext` object.

So, the compression flag is passed in the 5th argument of `CConnectionContext::CreateNetwork`. Inside the function, another one is called:

```
... push [ebp+compression]
.push [ebp+arg_C]
.push [ebp+arg_8]
.push [ebp+arg_4]
.push [ebp+arg_0]
call CNetwork__CNetwork
```

The compression flag is passed here in the 5th argument to the `CNetwork::CNetwork` constructor.

And here is how the `CNetwork` constructor sets the flag in the `CNetwork` object according to its 5th argument and another variable which probably could also affect network packets compression.

```
cmp [ebp+compression], esi
jz short set_EAX_to_0
mov al, [ebx+78h] ; another value may affect compression?
```

```
cmp al, '3'
jz short set_EAX_to_1
```

```
cmp al, '4'
jnz short set_EAX_to_0
```
At this point we know the compression flag is stored in the CNetwork class at address this+0x3A4.

Now let's dig through SAPguilib.dll for the 0x3A4 value. And here is the second occurrence in CDwsGui::OnClientMessageWrite (endless thanks for the debugging information):

```
.loc_64406F76:    mov    ecx, [ebp+7728h+var_7794]  
.loc_64406F79:    cmp    dword ptr [ecx+3A4h], 1  
.loc_64406F80:    jnz    compression_flag_is_zero  
.loc_64406F82:    mov    eax, [esi+20h]  
.loc_64406F83:    test   eax, eax  
.loc_64406F84:    ja     short loc_64406FFF  
.loc_64406F85:    mov    ecx, [esi+18h]  
.loc_64406F87:    lea    edx, [ebp+7728h+Dst]  
.loc_64406F88:    push   ecx, eax  
.loc_64406F89:    push   dword ptr [edi+28C0h]  
.loc_64406F90:    push   edax  
.loc_64406FAF:    push   ecx, eax  
.loc_64406FAD:    push   dword ptr [edi+28C0h]  
.loc_64406FAE:    lea    edx, [ebp+7728h+var_77A4]  
.loc_64406FBB:    push   dword ptr [edi+2868h]  
.loc_64406FBC:    push   edx  
.loc_64406FBD:    push   30000  
.loc_64406FBE:    lea    edx, [ebp+7728h+Dst]  
.loc_64406FBB:    push   edx  
.loc_64406FBF:    push   ecx  
.loc_64406FCA:    push   eax  
.loc_64406FDB:    push   dword ptr [edi+28C0h]  
```
Let's take a look in `sub_644055C5`. In it we can only see the call to `memcpy()` and another function named (by IDA) `sub_64417440`.

And, let's take a look inside `sub_64417440`. What we see is:

Voilà! We've found the function that actually compresses the data. As it was shown in past\(^8\) this function is used in SAP and also the open-source MaxDB project. So it is available in source form.

Doing the last check here:

Replace JNZ here for an unconditional JMP. Remove the environment variable `TDW_NOCOMPRESS`. Voilà! In Wireshark we see that the client messages are not compressed. The server responses, however, are compressed.

So we found exact connection between the environment variable and the point where data compression routine can be called or bypassed.

---

\(^8\) [http://go.yurichev.com/17312](http://go.yurichev.com/17312)
80.2 SAP 6.0 password checking functions

One time when the author of this book have returned again to his SAP 6.0 IDES installed in a VMware box, he figured out that he forgot the password for the SAP* account, then he have remembered it, but then we got this error message «Password logon no longer possible - too many failed attempts», since he’ve made all these attempts in trying to recall it.

The first extremely good news was that the full disp+work.pdb PDB file is supplied with SAP, and it contain almost everything: function names, structures, types, local variable and argument names, etc. What a lavish gift!

There is TYPEINFODUMP⁹ utility for converting PDB files into something readable and grepable.

Here is an example of a function information + its arguments + its local variables:

```
FUNCTION ThVmcSysEvent
  Address:  10143190 Size:  675 bytes Index:  60483
  Type: int NEAR_C ThVmcSysEvent (unsigned int, unsigned char, unsigned short*)
Flags: 0
PARAMETER events
  Address: Reg335+288 Size:  4 bytes Index:  60488
  Type: unsigned int
Flags: d0
PARAMETER opcode
  Address: Reg335+296 Size:  1 bytes Index:  60490
  Type: unsigned char
Flags: d0
PARAMETER serverName
  Address: Reg335+304 Size:  8 bytes Index:  60492
  Type: unsigned short*
Flags: d0
STATIC_LOCAL_VAR func
  Address:  12274af0 Size:  8 bytes Index:  60495
  Type: wchar_t*
Flags: 80
LOCAL_VAR admhead
  Address: Reg335+304 Size:  8 bytes Index:  60498
```

⁹http://go.yurichev.com/17038
And here is an example of some structure:

```c
STRUCT DBSL_STMTID
Size: 120  Variables: 4  Functions: 0  Base classes: 0
MEMBER moduletype
   Type: DBSL_MODULETYPE
   Offset: 0  Index: 3  TypeIndex: 38653
MEMBER module
   Type: wchar_t module[40]
   Offset: 4  Index: 3  TypeIndex: 831
MEMBER stmtnum
   Type: long
   Offset: 84  Index: 3  TypeIndex: 440
MEMBER timestamp
   Type: wchar_t timestamp[15]
   Offset: 88  Index: 3  TypeIndex: 6612
```

Wow!

Another good news: debugging calls (there are plenty of them) are very useful.

Here you can also notice the ct_level global variable\textsuperscript{10}, that reflects the current trace level.

There are a lot of debugging inserts in the disp+work.exe file:

```
cmp    cs:ct_level, 1
jl     short loc_1400375DA
call   DpLock
lea    rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov    edx, 4Eh ; line
call   CTrcSaveLocation
mov    r8, cs:func_48
mov    rcx, cs:hdl ; hdl
```

\textsuperscript{10}More about trace level: \url{http://go.yurichev.com/17039}
lea   rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov   r9d, ebx
call  DpTrcErr
call  DpUnlock

If the current trace level is bigger or equal to threshold defined in the code here, a debugging message is to be written to the log files like dev_w0, dev_disp, and other dev* files.

Let's try grepping in the file that we have got with the help of the TYPEINFODUMP utility:

cat "disp+work.pdb.d" | grep FUNCTION | grep -i password

We have got:

FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
FUNCTION password_is_initial
FUNCTION rcui::AgiPassword::IsVisible
FUNCTION password_distance_ok
FUNCTION get_password_downwards_compatibility
FUNCTION dySignUnSkipUserPassword
FUNCTION rcui::AgiPassword::GetType
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$2
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$0
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$1
FUNCTION usm_set_password
FUNCTION rcui::AgiPassword::TraceTo
FUNCTION days_since_last_password_change
FUNCTION rsecgrp_generate_random_password
FUNCTION rcui::AgiPassword::`scalar deleting destructor'
FUNCTION password_attempt_limit_exceeded
FUNCTION handle_incorrect_password
FUNCTION `rcui::AgiPassword::`scalar deleting destructor
FUNCTION calculate_new_password_hash
FUNCTION shift_password_to_history
FUNCTION rcui::AgiPassword::GetType
FUNCTION found_password_in_history
FUNCTION `rcui::AgiPassword::`scalar deleting destructor
FUNCTION rcui::AgiObj::IsaPassword
FUNCTION password_idle_check
FUNCTION SlicHwPasswordForDay
FUNCTION rcui::AgiPassword::IsaPassword
FUNCTION rcui::AgiPassword::AgiPassword
FUNCTION delete_user_password
FUNCTION usm_set_user_password
FUNCTION Password_API
FUNCTION get_password_change_for_SSO
FUNCTION password_in_USR40
FUNCTION rsec_agrp_abap_generate_random_password

Let’s also try to search for debug messages which contain the words «password» and «locked». One of them is the string «user was locked by subsequently failed password logon attempts», referenced in function password_attempt_limit_exceeded().

Other strings that this function can write to a log file are: «password logon attempt will be rejected immediately (preventing dictionary attacks)», «failed-logon lock: expired (but not removed due to 'read-only' operation)», «failed-logon lock: expired => removed».

After playing for a little with this function, we noticed that the problem is exactly in it. It is called from the chckpass() function — one of the password checking functions.

First, we would like to make sure that we are at the correct point:
Run tracer:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args ↵
  :3,unicode
```

```
PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L" ↵
  ↵ Brewered1 ", 0x41) (called ↵
  ↵ from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2236|TID=2248|(0) disp+work.exe!chckpass -> 0x35
```

The call path is: syssigni() -> DyISigni() -> dychusr() -> usrexist() -> chckpass().

The number 0x35 is an error returned in chckpass() at that point:

```
.text:00000001402ED567 loc_1402ED567:
  ↵ ; CODE XREF: chckpass+B4
.text:00000001402ED567 mov rcx, rbx ↵
  ↵ ; usr02
.text:00000001402ED56A call ↵
  ↵ password_idle_check
.text:00000001402ED56F cmp eax, 33h
.text:00000001402ED572 jz loc_1402EDB4E
.text:00000001402ED578 cmp eax, 36h
```
Fine, let's check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!
  \ password_attempt_limit_exceeded,args:4,unicode,rt:0
```

| PID=2744|TID=360|(0) disp+work.exe!  |
|-----------------------------------------------|
|,password_attempt_limit_exceeded (0x202c770, 0, 0x257758, 0) (called from 0x1402ed58b (disp+work.exe!chckpass+0xe6)) |

| PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0 |

Excellent! We can successfully login now.

By the way, we can pretend we forgot the password, fixing the `chckpass()` function to return a value of 0 is enough to bypass the check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode,rt:0
```
What also can be said while analyzing the `password_attempt_limit_exceeded()` function is that at the very beginning of it, this call can be seen:

```
lea    rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call   sapgpparam
test   rax, rax
jz     short loc_1402E19DE
movzx  eax, word ptr [rax]
cmp    ax, 'N'
jz     short loc_1402E19D4
cmp    ax, 'n'
jz     short loc_1402E19D4
cmp    ax, '0'
jnz    short loc_1402E19DE
```

Obviously, function `sapgpparam()` is used to query the value of some configuration parameter. This function can be called from 1768 different places. It seems that with the help of this information, we can easily find the places in code, the control flow of which can be affected by specific configuration parameters.

It is really sweet. The function names are very clear, much clearer than in the Oracle RDBMS. It seems that the `disp+work` process is written in C++. It was apparently rewritten some time ago?
Chapter 81

Oracle RDBMS

81.1 V$VERSION table in the Oracle RDBMS

Oracle RDBMS 11.2 is a huge program, its main module oracle.exe contain approx. 124,000 functions. For comparison, the Windows 7 x86 kernel (ntoskrnl.exe) contains approx. 11,000 functions and the Linux 3.9.8 kernel (with default drivers compiled) – 31,000 functions.

Let’s start with an easy question. Where does Oracle RDBMS get all this information, when we execute this simple statement in SQL*Plus:

```
SQL> select * from V$VERSION;
```

And we get:

```
BANNER

Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 
Production
PL/SQL Release 11.2.0.1.0 - Production
CORE 11.2.0.1.0 Production
TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production
```

Let’s start. Where in the Oracle RDBMS can we find the string V$VERSION?

In the win32-version, oracle.exe file contains the string, it’s easy to see. But we can also use the object (.o) files from the Linux version of Oracle RDBMS since,
unlike the win32 version oracle.exe, the function names (and global variables as well) are preserved there.

So, the kqf.o file contains the V$VERSION string. The object file is in the main Oracle-library libserver11.a.

A reference to this text string can find in the kqfviw table stored in the same file, kqf.o:

```
Listing 81.1: kqf.o

<table>
<thead>
<tr>
<th>rodata:0800C4A0 kqfviw dd 0Bh</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XREF: kqfchk:loc_8003A6D</td>
</tr>
<tr>
<td>rodata:0800C4A0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4A4 dd offset _2__STRING_10102_0</td>
<td></td>
</tr>
<tr>
<td>&quot;GV$WAITSTAT&quot;</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4A8 dd 4</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4AC dd offset _2__STRING_10103_0</td>
<td></td>
</tr>
<tr>
<td>&quot;NULL&quot;</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4B0 dd 3</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4B4 dd 0</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4B8 dd 195h</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4BC dd 4</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4C0 dd 0</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4C4 dd 0FFFFC1CBh</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4C8 dd 3</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4CC dd 0</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4D0 dd 0Ah</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4D4 dd offset _2__STRING_10104_0</td>
<td></td>
</tr>
<tr>
<td>&quot;V$WAITSTAT&quot;</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4D8 dd 4</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4DC dd offset _2__STRING_10103_0</td>
<td></td>
</tr>
<tr>
<td>&quot;NULL&quot;</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4E0 dd 3</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4E4 dd 0</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4E8 dd 4Eh</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4EC dd 3</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4F0 dd 0</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4F4 dd 0FFFFC003h</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4F8 dd 4</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C4FC dd 0</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C500 dd 5</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C504 dd offset _2__STRING_10105_0</td>
<td></td>
</tr>
<tr>
<td>&quot;GV$BH&quot;</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C508 dd 4</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C50C dd offset _2__STRING_10103_0</td>
<td></td>
</tr>
<tr>
<td>&quot;NULL&quot;</td>
<td></td>
</tr>
<tr>
<td>rodata:0800C510 dd 3</td>
<td></td>
</tr>
</tbody>
</table>
```
By the way, often, while analysing Oracle RDBMS’s internals, you may ask yourself, why are the names of the functions and global variables so weird. Probably, because Oracle RDBMS is a very old product and was developed in C in the 1980s. And that was a time when the C standard guaranteed that the function names/variables can support only up to 6 characters inclusive: «6 significant initial characters in an external identifier»\(^1\)

Probably, the table \texttt{kqfviw} contains most (maybe even all) views prefixed with V$, these are \textit{fixed views}, present all the time. Superficially, by noticing the cyclic recurrence of data, we can easily see that each \texttt{kqfviw} table element has 12 32-bit fields. It is very simple to create a 12-elements structure in \texttt{IDA} and apply it to all table elements. As of Oracle RDBMS version 11.2, there are 1023 table elements, i.e., in it are described 1023 of all possible \textit{fixed views}. We are going to return to this number later.

As we can see, there is not much information in these numbers in the fields. The first number is always equal to the name of the view (without the terminating zero. This is correct for each element. But this information is not very useful.

We also know that the information about all fixed views can be retrieved from a \textit{fixed view} named \texttt{V$FIXED_VIEW_DEFINITION} (by the way, the information for this view is also taken from the \texttt{kqfviw} and \texttt{kfqvip} tables.) By the way, there are 1023 elements in those too. Coincidence? No.

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';

VIEW_NAME
-------------------------------

VIEW_DEFINITION
-------------------------------

V$VERSION
select BANNER from GV$VERSION where inst_id = USERENV('Instance')

So, V$VERSION is some kind of a thunk view for another view, named GV$VERSION, which is, in turn:

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$VERSION';

VIEW_NAME
-------------------------------

VIEW_DEFINITION
-------------------------------

GV$VERSION
select inst_id, banner from x$version

The tables prefixed with X$ in the Oracle RDBMS are service tables too, undocumented, cannot be changed by the user and are refreshed dynamically.

If we search for the text `select BANNER from GV$VERSION where inst_id = USERENV('Instance')` in the kqf.o file, we find it in the kqfvip table:

Listing 81.2: kqf.o

```
rodata:080185A0 kqfvip dd offset _2__STRING_11126_0 ;
  DATA XREF: kqfgvcn+18
.rodata:080185A0
  kqfgvt+F
.rodata:080185A0
  " select inst_id,decode(indx,1,'data bloc"
.rodata:080185A4 dd offset kqfv459_c_0
.rodata:080185A8 dd 0
.rodata:080185AC dd 0
...
```
The table appear to have 4 fields in each element. By the way, there are 1023 elements in it, again, the number we already know. The second field points to another table that contains the table fields for this fixed view. As for V$VERSION, this table contains only two elements, the first is 6 and the second is the BANNER string (the number 6 is this string's length) and after, a terminating element that contains 0 and a null C string:

Listing 81.3: kqf.o

By joining data from both kqfviw and kqfvip tables, we can get the SQL statements which are executed when the user wants to query information from a specific fixed view.

So we can write an oracle tables\(^2\) program, to gather all this information from Oracle RDBMS for Linux's object files. For V$VERSION, we find this:

Listing 81.4: Result of oracle tables

\(^2\)yurichev.com
and:

**Listing 81.5: Result of oracle tables**

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0
  ↘ xffffffc192 0x1
kqfvip_element.statement: [select inst_id, banner from 
  ↘ x$version]
kqfvip_element.params:
[INST_ID] [BANNER]
```

The **GV$VERSION fixed view** is different from **V$VERSION** only in that it contains one more field with the identifier *instance*. Anyway, we are going to stick with the **X$VERSION** table. Just like any other X$-table, it is undocumented, however, we can query it:

```
SQL> select * from x$version;
ADDR INDX INST_ID
-------- ---------- ----------
BANNER

0DBAF574 0 1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 
  ↘ Production
...
```

This table has some additional fields, like **ADDR** and **INDX**.

While scrolling `kqf.o` in **IDA** we can spot another table that contains a pointer to the **X$VERSION** string, this is `kqftab`:

**Listing 81.6: kqf.o**

```
.rodata:0803CAC0 dd 9 ; ↘
  ↘ element number 0x1f6
.rodata:0803CAC4 dd offset _2__STRING_13113_0 ; ↘
  ↘ "X$VERSION"
.rodata:0803CAC8 dd 4
.rodata:0803CACC dd offset _2__STRING_13114_0 ; ↘
  ↘ "kqvt"
.rodata:0803CAD0 dd 4
.rodata:0803CAD4 dd 4
.rodata:0803CAD8 dd 0
.rodata:0803CADC dd 4
.rodata:0803CAE0 dd 0Ch
```
There are a lot of references to the X$-table names, apparently, to all Oracle RDBMS 11.2 X$-tables. But again, we don’t have enough information. It’s not clear what does the kqvt string stands for. The kq prefix may mean kernel or query. v apparently stands for version and t—type? Hard to say.

A table with a similar name can be found in kqf.o:

```
Listing 81.7: kqf.o

.krodata:0808C360 kqvt_c_0 kqftap_param <4, offset _2__STRING_19_0, 917h, 0, 0, 4, 0, 0>
.krodata:0808C360 ; DATA
   XREF: .krodata:08042680
.krodata:0808C360 ; "ADDR"
.krodata:0808C384 kqftap_param <4, offset _2__STRING_20_0, 0B02h, 0, 0, 4, 0, 0> ; "INDX"
.krodata:0808C3A8 kqftap_param <7, offset _2__STRING_21_0, 0B02h, 0, 0, 4, 0, 0> ; "INST_ID"
.krodata:0808C3CC kqftap_param <6, offset _2__STRING_5017_0, 601h, 0, 0, 50h, 0, 0> ; "BANNER"
.krodata:0808C3F0 kqftap_param <0, offset _2__STRING_0_0, 0, 0, 0, 0, 0, 0>
```

It contains information about all fields in the X$VERSION table. The only reference to this table is in the kqftap table:

```
Listing 81.8: kqf.o

.krodata:08042680 kqftap_element <0, offset _2__STRING_0, offset kqvrow, 0> ; element 0x1f6
```
It is interesting that this element here is \texttt{0x1f6th} (502nd), just like the pointer to the \texttt{X$\text{VERSION}} string in the \texttt{kqftab} table. Probably, the \texttt{kqftap} and \texttt{kqftab} tables complement each other, just like \texttt{kqfvip} and \texttt{kqfviw}. We also see a pointer to the \texttt{kqvrow()} function. Finally, we got something useful!

So we will add these tables to our oracle tables\textsuperscript{3} utility too. For \texttt{X$\text{VERSION}} we get:

\begin{verbatim}
Listing 81.9: Result of oracle tables

kqftab_element.name: [X$\text{VERSION}] ?: [kqvt] 0x4 0x4 0xc 0
   \x00 xfffe075 0x3
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[BANNER] ?: 0x601 0x0 0x0 0x0 0x50 0x0 0x0
kqftap_element.fn1=kqvrow
kqftap_element.fn2=NULL
\end{verbatim}

With the help of \texttt{tracer}, it is easy to check that this function is called 6 times in row (from the \texttt{qerfxFetch()} function) while querying the \texttt{X$\text{VERSION}} table.

Let’s run \texttt{tracer} in cc mode (it comments each executed instruction):

\begin{verbatim}
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc
\end{verbatim}

\begin{verbatim}
_kqvrow_ proc near
var_7C    = byte ptr -7Ch
var_18    = dword ptr -18h
var_14    = dword ptr -14h
Dest      = dword ptr -10h
var_C     = dword ptr -0Ch
var_8     = dword ptr -8
var_4     = dword ptr -4
arg_8     = dword ptr 10h
arg_C     = dword ptr 14h
arg_14    = dword ptr 1Ch
arg_18    = dword ptr 20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES
push      ebp
mov       ebp, esp
sub       esp, 7Ch
mov       eax, [ebp+arg_14] ; [EBP+1Ch]=1
mov       ecx, TlsIndex ; [69AEB08h]=0
\end{verbatim}

\textsuperscript{3}yurichev.com

\footnotesize

1184

\end{verbatim}
CHAPTER 81. ORACLE RDBMS

mov edx, large fs:2Ch
mov edx, [edx+ecx*4] ; [EDX+ECX*4]=0xc98c938
cmp eax, 2 ; EAX=1
mov eax, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
jz loc_2CE1288
mov ecx, [eax] ; [EAX]=0..5
mov [ebp+var_4], edi ; EDI=0xc98c938

loc_2CE10F6: ; CODE XREF: _kqvrow_+10A
               ; _kqvrow_+1A9
    cmp ecx, 5 ; ECX=0..5
               ; loc_56C11C7
    ja loc_56C11C7
    mov edi, [ebp+arg_18] ; [EBP+20h]=0
    mov [ebp+var_14], edx ; EDX=0xc98c938
    mov [ebp+var_8], ebx ; EBX=0
    mov ebx, eax ; EAX=0xcdfe554
    mov [ebp+var_C], esi ; ESI=0xcdfe248

loc_2CE110D: ; CODE XREF: _kqvrow_+29E00E6
    mov edx, ds:off_628B09C[ecx*4] ; [ECX*4+628B09Ch]=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a
    jmp edx ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
    push offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
    mov ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
    xor edx, edx
    mov esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
    push edx ; EDX=0
    push edx ; EDX=0
    push 50h
    push ecx ; ECX=0x8a172b4
    push dword ptr [esi+10494h] ; [ESI+10494h]=0
    jmp edx ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
    push offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
    mov ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
    xor edx, edx
    mov esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
    push edx ; EDX=0
    push edx ; EDX=0
    push 50h
    push ecx ; ECX=0x8a172b4
    push dword ptr [esi+10494h] ; [ESI+10494h]=0

    call _kghalf ; tracing nested maximum

    level (1) reached, skipping this CALL
    mov esi, ds:_imp__vsnnum ; [59771A8h]=0x61bc49e0
    mov [ebp+Dest], eax ; EAX=0xce22fbb0
    mov [ebx+8], eax ; EAX=0xce22fbb0
    mov [ebx+4], eax ; EAX=0xce22fbb0
    mov edi, [esi] ; [ESI]=0xb200100
    mov esi, ds:_imp__vsnstr ; [597D6D4h]=0x65852148

    , "- Production"
    push esi ; ESI=0x65852148, "- "

Production"
mov ebx, edi  ; EDI=0xb200100
shr ebx, 18h ; EBX=0xb200100
mov ecx, edi  ; EDI=0xb200100
shr ecx, 14h ; ECX=0xb200100
and ecx, 0Fh ; ECX=0xb2
mov edx, edi  ; EDI=0xb200100
shr edx, 0Ch ; EDX=0xb200100
movzx edx, dl ; DL=0
mov eax, edi  ; EDI=0xb200100
shr eax, 8  ; EAX=0xb200100
and eax, 0Fh ; EAX=0xb2001
and edi, 0FFh ; EDI=0xb200100
push edi    ; EDI=0
mov edi, [ebp+arg_18] ; [EBP+20h]=0
push eax    ; EAX=1
mov eax, ds:__imp__vsnban ; [597D6D8h]=0x65852100
push edx    ; EDX=0
push ecx    ; ECX=2
push ebx    ; EBX=0xb
mov ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
push eax    ; EAX=0x65852100, "Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d.%d %s"
push edx    ; EDX=0
push ecx    ; ECX=2
push ebx    ; EBX=0xb
mov ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
push eax    ; EAX=0x65852100, "Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d.%d %s"
add esp, 38h
mov dword ptr [ebx], 1

loc_2CE1192: ; CODE XREF: _kqvrow_+FB
          ; _kqvrow_+128 ...
test edi, edi    ; EDI=0
jnz __VInfreq__kqvrow
mov esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
mov edi, [ebp+var_4] ; [EBP-4]=0xc98c938
mov eax, ebx     ; EBX=0xcdfe554
mov ebx, [ebp+var_8] ; [EBP-8]=0
loc_2CE11A8: ; CODE XREF: _kqvrow_+29E00F6
    mov esp, ebp
    pop ebp
    retn ; EAX=0xcdfe558

loc_2CE11AC: ; DATA XREF: .rdata:0628B0A0
    mov edx, [ebx+8] ; [EBX+8]=0xce2ffb0, "Oracle
\ Database 11g Enterprise Edition Release 11.2.0.1.0 -
\ Production"
    mov dword ptr [ebx], 2
    mov [ebx+4], edx ; EDX=0xce2ffb0, "Oracle
\ Database 11g Enterprise Edition Release 11.2.0.1.0 -
\ Production"
    push edx ; EDX=0xce2ffb0, "Oracle
\ Database 11g Enterprise Edition Release 11.2.0.1.0 -
\ Production"
    call _kkxvsn ; tracing nested maximum
\ level (1) reached, skipping this CALL
    pop ecx
    mov edx, [ebx+4] ; [EBX+4]=0xce2ffb0, "PL/SQL
\ Release 11.2.0.1.0 - Production"
    movzx ecx, byte ptr [edx] ; [EDX]=0x50
    test ecx, ecx ; ECX=0x50
    jnz short loc_2CE1192
    mov edx, [ebp+var_14]
    mov esi, [ebp+var_C]
    mov eax, ebx
    mov ebx, [ebp+var_8]
    mov ecx, [eax]
    jmp loc_2CE10F6

loc_2CE11DB: ; DATA XREF: .rdata:0628B0A4
    push 0
    push 50h
    mov edx, [ebx+8] ; [EBX+8]=0xce2ffb0, "PL/SQL
\ Release 11.2.0.1.0 - Production"
    mov [ebx+4], edx ; EDX=0xce2ffb0, "PL/SQL
\ Release 11.2.0.1.0 - Production"
    push edx ; EDX=0xce2ffb0, "PL/SQL
\ Release 11.2.0.1.0 - Production"
    call _lmxver ; tracing nested maximum
\ level (1) reached, skipping this CALL
    add esp, 0Ch
    mov dword ptr [ebx], 3
    jmp short loc_2CE1192

loc_2CE11F6: ; DATA XREF: .rdata:0628B0A8
CHAPTER 81. ORACLE RDBMS

mov edx, [ebx+8] ; [EBX+8]=0xce2ffb0
mov [ebp+var_18], 50h
mov [ebx+4], edx ; EDX=0xce2ffb0
push 0
call _npinli ; tracing nested maximum ▽
level (1) reached, skipping this CALL
pop ecx
test eax, eax ; EAX=0
jnz loc_56C11DA
mov ecx, [ebp+var_14] ; [EBP-14h]=0xc98c938
lea edx, [ebp+var_18] ; [EBP-18h]=0x50
push edx ; EDX=0xd76c93c
push dword ptr [ebx+8] ; [EBX+8]=0xce2ffb0
push dword ptr [ecx+13278h] ; [ECX+13278h]=0
xacce190
call _nrtnsvrs ; tracing nested maximum ▽
level (1) reached, skipping this CALL
add esp, 0Ch
loc_2CE122B: ; CODE XREF: _kqvrow_+29E0118
mov dword ptr [ebx], 4
jmp loc_2CE1192
loc_2CE1236: ; DATA XREF: .rdata:0628B0AC
lea edx, [ebp+var_7C] ; [EBP-7Ch]=1
push edx ; EDX=0xd76c8d8
push 0
mov esi, [ebx+8] ; [EBX+8]=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 - Production"
mov [ebx+4], esi ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 - Production"
mov ecx, 50h
mov [ebp+var_18], ecx ; ECX=0x50
push ecx ; ECX=0x50
push esi ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 - Production"
call _lxvers ; tracing nested maximum ▽
level (1) reached, skipping this CALL
add esp, 10h
mov edx, [ebp+var_18] ; [EBP-18h]=0x50
mov dword ptr [ebx], 5
test edx, edx ; EDX=0x50
jnz loc_2CE1192
mov edx, [ebp+var_14]
mov esi, [ebp+var_C]
mov eax, ebx
mov ebx, [ebp+var_8]
Now it is easy to see that the row number is passed from outside. The function returns the string, constructing it as follows:

| String 1 | Using vsnstr, vsnnum, vsnban global variables. Calling sprintf(). |
| String 2 | Calling kkxvsn(). |
| String 3 | Calling lmxver(). |
| String 4 | Calling npinli(), nrtnsvrs(). |
| String 5 | Calling lxvers(). |

That's how the corresponding functions are called for determining each module's version.
X$KSMLRU table in Oracle RDBMS

There is a mention of a special table in the *Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]* note:

There is a fixed table called X$KSMLRU that tracks allocations in the shared pool that cause other objects in the shared pool to be aged out. This fixed table can be used to identify what is causing the large allocation.

If many objects are being periodically flushed from the shared pool then this will cause response time problems and will likely cause library cache latch contention problems when the objects are reloaded into the shared pool.

One unusual thing about the X$KSMLRU fixed table is that the contents of the fixed table are erased whenever someone selects from the fixed table. This is done since the fixed table stores only the largest allocations that have occurred. The values are reset after being selected so that subsequent large allocations can be noted even if they were not quite as large as others that occurred previously. Because of this resetting, the output of selecting from this table should be carefully kept since it cannot be retrieved back after the query is issued.

However, as it can be easily checked, the contents of this table are cleared each time it’s queried. Are we able to find why? Let’s get back to tables we already know: kqftab and kqftap which were generated with oracle tables’ help, that contain all information about the X$-tables. We can see here that the ksmlrs() function is called to prepare this table’s elements:

Listing 81.10: Result of oracle tables

```
kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0
  xffffff0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRDUR] ?: 0xb02 0x0 0x0 0x0 0x4 0x4 0x0
kqftap_param.name=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0
  x0
kqftap_param.name=[KSMLRCOM] ?: 0x501 0x0 0x0 0x0 0x14 0xc 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
kqftap_param.name=[KSMLRNUM] ?: 0x2 0x0 0x0 0x0 0x4 0x24 0x0
```

---

4yurichev.com
Indeed, with tracer's help it is easy to see that this function is called each time we query the X$KSMLRU table.

Here we see a references to the ksmsplu_sp() and ksmsplu_jp() functions, each of them calls the ksmsplu() in the end. At the end of the ksmsplu() function we see a call to memset():

Listing 81.11: ksm.o

```
.text:00434C50 loc_434C50: ; DATA ↳
\ XREF: .rdata:off_5E50EA8
.text:00434C50 mov edx, [ebp-4]
.text:00434C53 mov [eax], esi
.text:00434C55 mov esi, [edi]
.text:00434C57 mov [eax+4], esi
.text:00434C5A mov [edi], eax
.text:00434C5C add edx, 1
.text:00434C5F mov [ebp-4], edx
.text:00434C62 jnz loc_434B7D
.text:00434C68 mov ecx, [ebp+14h]
.text:00434C6B mov ebx, [ebp-10h]
.text:00434C6E mov esi, [ebp-0Ch]
.text:00434C71 mov edi, [ebp-8]
.text:00434C74 lea eax, [ecx+8Ch]
.text:00434C7A push 370h ; Size
.text:00434C7F push 0 ; Val
.text:00434C81 push eax ; Dst
.text:00434C82 call __intel_fast_memset
.text:00434C87 add esp, 0Ch
.text:00434C8A mov esp, ebp
.text:00434C8C pop ebp
.text:00434C8D ret
.text:00434C8D _ksmsplu endp
```
Constructions like `memset(block, 0, size)` are often used just to zero memory block. What if we take a risk, block the `memset()` call and see what happens?

Let’s run `tracer` with the following options: set breakpoint at 0x434C7A (the point where the arguments to `memset()` are to be passed), so that `tracer` will set program counter EIP to the point where the arguments passed to `memset()` are to be cleared (at 0x434C8A). It can be said that we just simulate an unconditional jump from address 0x434C7A to 0x434C8A.

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0 szczególn
ceptarow 0x00434C8A)
```

(Important: all these addresses are valid only for the win32 version of Oracle RDBMS 11.2)

Indeed, now we can query the `X$KSMLRU` table as many times as we want and it is not being cleared anymore!

Do not try this at home (“MythBusters!”). Do not try this on your production servers.

It is probably not a very useful or desired system behaviour, but as an experiment for locating a piece of code that we need, it perfectly suits our needs!

### 81.3 V$TIMER table in Oracle RDBMS

V$TIMER is another fixed view that reflects a rapidly changing value:

```
V$TIMER displays the elapsed time in hundredths of a second.
Time is measured since the beginning of the epoch, which is operating system specific, and wraps around to 0 again whenever the value overflows four bytes (roughly 497 days).
```

(From Oracle RDBMS documentation)

It is interesting that the periods are different for Oracle for win32 and for Linux. Will we be able to find the function that generates this value?

As we can see, this information is finally taken from the `X$KSUTM` table.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$TIMER';
```

```
VIEW_NAME
-------------------------------
```

[^5]: [http://go.yurichev.com/17088](http://go.yurichev.com/17088)
VIEW_DEFINITION

\[\text{V$TIMER} \]
select HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$TIMER';

Now we are stuck in a small problem, there are no references to value generating function(s) in the tables kqftab/kqftap:

Listing 81.12: Result of oracle tables

```
kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0
\[xxxxf09b 0x3
kqftap_param.name=[ADDR] ?: 0x10917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0x20b02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSUTMTIM] ?: 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e
kqftap_element.fn1=NULL
kqftap_element.fn2=NULL
```

When we try to find the string KSUTMTIM, we see it in this function:

```
kqfd_DRN_ksutm_c proc near ; DATA XREF: .rodata\[ \[0805B4E8
\[ arg_0 = dword ptr 8
arg_8 = dword ptr 10h
arg_C = dword ptr 14h
push ebp
mov ebp, esp
push [ebp+arg_C]
push offset ksugtm
push offset _2__STRING_1263_0 ; "KSUTMTIM"
push [ebp+arg_8]
```
The kqfd_DRN_ksutm_c() function is mentioned in the kqfd_tab_registry_0 table:

```
dd offset _2__STRING_62_0 ; "X$KSUTM"
dd offset kqfd_OPN_ksutm_c
dd offset kqfd_tabl_fetch
dd 0
dd 0
dd offset kqfd_DRN_ksutm_c
```

There is a function ksugtm() referenced here. Let's see what's in it (Linux x86):

```
Listing 81.13: ksu.o

ksugtm proc near

var_1C = byte ptr -1Ch
arg_4 = dword ptr 0Ch

push    ebp
mov     ebp, esp
sub     esp, 1Ch
lea     eax, [ebp+var_1C]
push    eax
call    slgcs
pop     ecx
mov     edx, [ebp+arg_4]
mov     [edx], eax
mov     eax, 4
mov     esp, ebp
pop     ebp
retn
ksugtm endp
```

The code in the win32 version is almost the same.

Is this the function we are looking for? Let's see:
Let's try again:

SQL> select * from V$TIMER;
HSECS
----------
27294929

SQL> select * from V$TIMER;
HSECS
----------
27295006

SQL> select * from V$TIMER;
HSECS
----------
27295167

Listing 81.14: tracer output

TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9 "8. "

TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: D1 7C A0 01 ".|..

TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9 "8. "

TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: 1E 7D A0 01 ".}..

TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9 "8. "

TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: BF 7D A0 01 ".}..
Indeed—the value is the same we see in SQL*Plus and it is returned via the second argument.

Let’s see what is in slgcs() (Linux x86):

```assembly
slgcs proc near

var_4 = dword ptr -4
arg_0 = dword ptr 8

push    ebp
mov     ebp, esp
push    esi
mov     [ebp+var_4], ebx
mov     eax, [ebp+arg_0]
call    $+5
pop     ebx

nop ; PIC mode
mov     ebx, offset _GLOBAL_OFFSET_TABLE_
mov     dword ptr [eax], 0
call    sltrgatime64 ; PIC mode
push    0
push    0Ah
push    edx
push    eax

call    __udivdi3 ; PIC mode
mov     ebx, [ebp+var_4]
add     esp, 10h
mov     esp, ebp
pop     ebp
retn

slgcs endp
```

(it is just a call to sltrgatime64() and division of its result by 10 (41 on page 676))

And win32-version:

```assembly
_slgcs proc near ; CODE XREF:
   \_dbgefgHtElResetCount+15 ; _dbgerRunActions+1528

   db       66h
   nop
push    ebp
mov     ebp, esp
mov     ebp, esp
mov     eax, [ebp+8]
```

1196
It is just the result of `GetTickCount()` divided by 10 (41 on page 676).

Voilà! That’s why the win32 version and the Linux x86 version show different results, because they are generated by different OS functions.

`Drain` apparently implies connecting a specific table column to a specific function.

We will add support of the table `kqfd_tab_registry_0` to oracle tables, now we can see how the table column’s variables are connected to a specific function:

```
[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksusg_c]
```

`OPN`, apparently stands for, `open`, and `DRN`, apparently, for `drain`.

---

6 MSDN
7 yurichev.com
Chapter 82

Handwritten assembly code

82.1 EICAR test file

This .COM-file is intended for testing antivirus software, it is possible to run in in MS-DOS and it prints this string: “EICAR-STANDARD-ANTIVIRUS-TEST-FILE!”

Its important property is that it’s consists entirely of printable ASCII-symbols, which, in turn, makes it possible to create it in any text editor:

\[
X5O!P%@AP[4\PZ54(P^)7CC)7}\$EICAR-STANDARD-ANTIVIRUS-TEST-FILE! \swarrow \$H+H*
\]

Let’s decompile it:

; initial conditions: SP=0FFFEh, SS:[SP]=0
0100 58 pop ax
; AX=0, SP=0
0101 35 4F 21 xor ax, 214Fh
; AX = 214Fh and SP = 0
0104 50 push ax
; AX = 214Fh, SP = FFFEh and SS:[FFFE] = 214Fh
0105 25 40 41 and ax, 4140h
; AX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
0108 50 push ax
; AX = 140h, SP = FFFCh, SS:[FFFF] = 140h and SS:[FFFE] = 214Fh
0109 5B pop bx
; AX = 140h, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010A 34 5C xor al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh

\(^{1}\)wikipedia
010C 50  push  ax
010D 5A  pop  dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFEh and SS: [FFE] = 214Fh

010E 58  pop  ax
; AX = 214Fh, BX = 140h, DX = 11Ch and SP = 0
010F 35 34 28  xor  ax, 2834h
; AX = 97Bh, BX = 140h, DX = 11Ch and SP = 0

0112 50  push  ax
0113 5E  pop  si
; AX = 97Bh, BX = 140h, DX = 11Ch, SI = 97Bh and SP = 0
0114 29 37  sub  [bx], si
0116 43  inc  bx
0117 43  inc  bx
0118 29 37  sub  [bx], si
011A 7D 24  jge short near ptr word_10140
011C 45 49 43  ... db 'EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$'
0140 48 2B  word_10140  dw 2B48h ; CD 21 (INT 21) will be here
0142 48 2A  dw 2A48h ; CD 20 (INT 20) will be here
0144 0D  db 0Dh
0145 0A  db 0Ah

We will add comments about the registers and stack after each instruction.

Essentially, all these instructions are here only to execute this code:

| B4 09 | MOV AH, 9 |
| BA 1C 01 | MOV DX, 11Ch |
| CD 21 | INT 21h |
| CD 20 | INT 20h |

INT 21h with 9th function (passed in AH) just prints a string, the address of which is passed in DS:DX. By the way, the string has to be terminated with the '$' sign. Apparently, it’s inherited from CP/M and this function was left in DOS for compatibility. INT 20h exits to DOS.

But as we can see, these instruction’s opcodes are not strictly printable. So the main part of EICAR file is:

- preparing the register (AH and DX) values that we need;
- preparing INT 21 and INT 20 opcodes in memory;
- executing INT 21 and INT 20.

By the way, this technique is widely used in shellcode construction, when one need to pass x86 code in string form.

Here is also a list of all x86 instructions which have printable opcodes: A.6.5 on page 1318.
Chapter 83

Demos

Demos (or demomaking) were an excellent exercise in mathematics, computer graphics programming and very tight x86 hand coding.

83.1 10 PRINT CHR$(205.5+RND(1)); : GOTO 10

All examples here are MS-DOS .COM files.

In [al12] we can read about one of the most simple possible random maze generators. It just prints a slash or backslash characters randomly and endlessly, resulting in something like this:
There are a few known implementations for 16-bit x86.

### 83.1.1 Trixter’s 42 byte version

The listing was taken from his website[^1], but the comments are mine.

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>mov al, 1</td>
<td></td>
<td></td>
<td>set 40x25 video mode</td>
</tr>
<tr>
<td>00000002</td>
<td>int 010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000004</td>
<td>xor bh, bh</td>
<td></td>
<td></td>
<td>set video page for int 10h call</td>
</tr>
<tr>
<td>00000006</td>
<td>mov cx, 007D0</td>
<td></td>
<td></td>
<td>2000 characters to output</td>
</tr>
<tr>
<td>00000009</td>
<td>xor ax, ax</td>
<td></td>
<td></td>
<td>get random value from timer chip</td>
</tr>
<tr>
<td>0000000B</td>
<td>pushf</td>
<td></td>
<td></td>
<td>push flags</td>
</tr>
<tr>
<td>0000000C</td>
<td>cli</td>
<td></td>
<td></td>
<td>disable</td>
</tr>
<tr>
<td>0000000D</td>
<td>out 043, al</td>
<td></td>
<td></td>
<td>write 0 to port 43h</td>
</tr>
<tr>
<td>0000000F</td>
<td>in al, 040</td>
<td></td>
<td></td>
<td>read 16-bit value from port 40h</td>
</tr>
<tr>
<td>00000011</td>
<td>mov ah, al</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

[^1]: [http://go.yurichev.com/17305](http://go.yurichev.com/17305)
The pseudo-random value here is in fact the time that has passed from the system's boot, taken from the 8253 time chip, the value increases by one 18.2 times per second.

By writing zero to port 43h, we send the command “select counter 0”, “counter latch”, “binary counter” (not a BCD value).

The interrupts are enabled back with the POPF instruction, which restores the IF flag as well.

It is not possible to use the IN instruction with registers other than AL, hence the shuffling.

### 83.1.2 My attempt to reduce Trixter's version: 27 bytes

We can say that since we use the timer not to get a precise time value, but a pseudo-random one, we do not need to spend time (and code) to disable the interrupts. Another thing we can say is that we need only one bit from the low 8-bit part, so let's read only it.

We can reduced the code slightly and we've got 27 bytes:

```assembly
00000000: B9D007 mov cx,007D0 ; limit output to 2000 characters
00000003: 31C0 xor ax,ax ; command to timer chip
00000005: E643 out 043,al
00000007: E440 in al,040 ; read 8-bit of timer
```

---

2Binary-coded decimal
83.1.3 Taking random memory garbage as a source of randomness

Since it is MS-DOS, there is no memory protection at all, we can read from whatever address we want. Even more than that: a simple LODSB instruction reads a byte from the DS:SI address, but it's not a problem if the registers' values are not set up, let it read 1) random bytes; 2) from a random place in memory!

It is suggested in Trixter's webpage\(^3\) to use LODSB without any setup.

It is also suggested that the SCASB instruction can be used instead, because it sets a flag according to the byte it reads.

Another idea to minimize the code is to use the INT 29h DOS syscall, which just prints the character stored in the AL register.

That is what Peter Ferrie and Andrey “herm1t” Baranovich did (11 and 10 bytes) \(^4\):

### Listing 83.1: Andrey “herm1t” Baranovich: 11 bytes

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>B05C</td>
<td>mov al,05C</td>
<td>; read AL byte from random place of memory</td>
</tr>
<tr>
<td>00000002</td>
<td>AE</td>
<td>scasb</td>
<td>; PF = parity(AL - random_memory_byte) = parity(5Ch - random_memory_byte)</td>
</tr>
<tr>
<td>00000003</td>
<td>7A02</td>
<td>jp</td>
<td>00000007</td>
</tr>
<tr>
<td>00000005</td>
<td>B02F</td>
<td>mov al,02F</td>
<td>;'/'</td>
</tr>
<tr>
<td>00000007</td>
<td>CD29</td>
<td>int 029</td>
<td>; output AL to screen</td>
</tr>
<tr>
<td>00000009</td>
<td>EBF5</td>
<td>jmp</td>
<td>000000000 ; loop endlessly</td>
</tr>
</tbody>
</table>

SCASB also uses the value in the AL register, it subtract a random memory byte's value from the 5Ch value in AL. JP is a rare instruction, here it used for checking the parity flag (PF), which is generated by the formulae in the listing. As a consequence,

---

\(^3\) [http://go.yurichev.com/17305](http://go.yurichev.com/17305)

\(^4\) [http://go.yurichev.com/17087](http://go.yurichev.com/17087)
the output character is determined not by some bit in a random memory byte, but by a sum of bits, this (hopefully) makes the result more distributed.

It is possible to make this even shorter by using the undocumented x86 instruction SALC (AKA SETALC) (“Set AL CF”). It was introduced in the NEC V20 CPU and sets AL to 0xFF if CF is 1 or to 0 if otherwise.

Listing 83.2: Peter Ferrie: 10 bytes

```
; AL is random at this point
00000000: AE scasb
; CF is set according subtracting random memory byte from AL.
; so it is somewhat random at this point
00000001: D6 setalc
; AL is set to 0xFF if CF=1 or to 0 if otherwise
00000002: 242D and al,02D ;'-'
; AL here is 0x2D or 0
00000004: 042F add al,02F ;'/'
; AL here is 0x5C or 0x2F
00000006: CD29 int 029 ; output AL to screen
00000008: EBF6 jmps 000000000 ; loop endlessly
```

So it is possible to get rid of conditional jumps at all. The ASCII code of backslash (“\”) is 0x5C and 0x2F for slash (“/”). So we need to convert one (pseudo-random) bit in the CF flag to a value of 0x5C or 0x2F.

This is done easily: by AND-ing all bits in AL (where all 8 bits are set or cleared) with 0x2D we have just 0 or 0x2D. By adding 0x2F to this value, we get 0x5C or 0x2F. Then we just output it to the screen.

83.1.4 Conclusion

It is also worth mentioning that the result may be different in DOSBox, Windows NT and even MS-DOS, due to different conditions: the timer chip can be emulated differently and the initial register contents may be different as well.
83.2 Mandelbrot set

Mandelbrot set is a fractal, which exhibits self-similarity. When you increase scale, you see that this characteristic pattern repeating infinitely.

Here is a demo⁵ written by “Sir_Lagsalot” in 2009, that draws the Mandelbrot set, which is just a x86 program with executable file size of only 64 bytes. There are only 30 16-bit x86 instructions.

Here it is what it draws:

Let’s try to understand how it works.

⁵Download it here,
83.2.1 Theory

A word about complex numbers

A complex number is a number that consists of two parts—real (Re) and imaginary (Im).

The complex plane is a two-dimensional plane where any complex number can be placed: the real part is one coordinate and the imaginary part is the other.

Some basic rules we need to know:

- **Addition:** \((a + bi) + (c + di) = (a + c) + (b + d)i\)
  
  In other words:
  
  \[
  \text{Re}(\text{sum}) = \text{Re}(a) + \text{Re}(b) \\
  \text{Im}(\text{sum}) = \text{Im}(a) + \text{Im}(b)
  \]

- **Multiplication:** \((a + bi)(c + di) = (ac - bd) + (bc + ad)i\)
  
  In other words:
  
  \[
  \text{Re}(\text{product}) = \text{Re}(a) \cdot \text{Re}(c) - \text{Re}(b) \cdot \text{Re}(d) \\
  \text{Im}(\text{product}) = \text{Im}(b) \cdot \text{Im}(c) + \text{Im}(a) \cdot \text{Im}(d)
  \]

- **Square:** \((a + bi)^2 = (a + bi)(a + bi) = (a^2 - b^2) + (2ab)i\)
  
  In other words:
  
  \[
  \text{Re}(\text{square}) = \text{Re}(a)^2 - \text{Im}(a)^2 \\
  \text{Im}(\text{square}) = 2 \cdot \text{Re}(a) \cdot \text{Im}(a)
  \]

How to draw the Mandelbrot set

The Mandelbrot set is a set of points for which the \(z_{n+1} = z_n^2 + c\) recursive sequence (where \(z\) and \(c\) are complex numbers and \(c\) is the starting value) does not approach infinity.

In plain English language:

- Enumerate all points on screen.
- Check if the specific point is in the Mandelbrot set.
- Here is how to check it:
  - Represent the point as a complex number.
  - Calculate the square of it.
- Add the starting value of the point to it.
- Does it go off limits? If yes, break.
- Move the point to the new place at the coordinates we just calculated.
- Repeat all this for some reasonable number of iterations.

- The point is still in limits? Then draw the point.
- The point has eventually gone off limits?
  - (For a black-white image) do not draw anything.
  - (For a colored image) transform the number of iterations to some color. So the color shows the speed with which point has gone off limits.

Here is Pythonesque algorithm for both complex and integer number representations:

```python
Listing 83.3: For complex numbers

def check_if_is_in_set(P):
    P_start=P
    iterations=0

    while True:
        if (P>bounds):
            break
        P=P^2+P_start
        if iterations > max_iterations:
            break
        iterations++

    return iterations

# black-white
for each point on screen P:
    if check_if_is_in_set (P) < max_iterations:
        draw point

# colored
for each point on screen P:
    iterations = if check_if_is_in_set (P)
    map iterations to color
    draw color point
```

The integer version is where the operations on complex numbers are replaced with integer operations according to the rules which were explained above.

```python
Listing 83.4: For integer numbers
```

1207
def check_if_is_in_set(X, Y):
    X_start=X
    Y_start=Y
    iterations=0

    while True:
        if (X^2 + Y^2 > bounds):
            break
        new_X=X^2 - Y^2 + X_start
        new_Y=2*X*Y + Y_start
        if iterations > max_iterations:
            break
        iterations++
    return iterations

# black-white
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        if check_if_is_in_set (X,Y) < max_iterations:
            draw point at X, Y

# colored
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        iterations = if check_if_is_in_set (X,Y)
        map iterations to color
        draw color point at X,Y

Here is also a C# source which is present in the Wikipedia article\textsuperscript{6}, but we'll modify it so it will print the iteration numbers instead of some symbol \textsuperscript{7}:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Mnoj
{
    class Program
    {
        static void Main(string[] args)
        {
            double realCoord, imagCoord;
            double realTemp, imagTemp, realTemp2, arg;
```
int iterations;
for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
{
    for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
    {
        iterations = 0;
        realTemp = realCoord;
        imagTemp = imagCoord;
        arg = (realCoord * realCoord) + (imagCoord * imagCoord);
        while ((arg < 2*2) && (iterations < 40))
        {
            realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp) - realCoord;
            imagTemp = (2 * realTemp * imagTemp) - imagCoord;
            realTemp = realTemp2;
            arg = (realTemp * realTemp) + (imagTemp * imagTemp);
            iterations += 1;
        }
        Console.Write("{0,2:D} ", iterations);
    }
    Console.Write("\n");
}
Console.ReadKey();

Here is the resulting file, which is too wide to be included here:
beginners.re.

The maximal number of iterations is 40, so when you see 40 in this dump, it mean
that this point was wandering for 40 iterations but never got off limits. A number
less then 40 mean that point remained inside the bounds only for \(n\) iterations, then it went outside them.
There is a cool demo available at http://go.yurichev.com/17309, which shows visually how the point moves on the plane at each iteration for some specific point. Here are two screenshots.

First, we’ve clicked inside the yellow area and saw that the trajectory (green line) eventually swirls at some point inside:

![Figure 83.1: Click inside yellow area](image)

This implies that the point we've clicked belongs to the Mandelbrot set.
Then we’ve clicked outside the yellow area and saw a much more chaotic point movement, which quickly went off bounds:

![Figure 83.2: Click outside yellow area](image)

This means the point does not belong to the Mandelbrot set.

Another good demo is available here: [http://go.yurichev.com/17310](http://go.yurichev.com/17310).
83.2.2 Let's get back to the demo

The demo, although very tiny (just 64 bytes or 30 instructions), implements the common algorithm described here, but using some coding tricks.

The source code is easily downloadable, so here is it, but let's also add comments:

Listing 83.5: Commented source code

```assembly
; X is column on screen
; Y is row on screen

; X=0, Y=0           X=319, Y=0
; +------------------------------->
; |
; |
; |
; v
; X=0, Y=199         X=319, Y=199

; switch to VGA 320*200*256 graphics mode
mov al,13h
int 10h
; initial BX is 0
; initial DI is 0xFFFFE
; DS:BX (or DS:0) is pointing to Program Segment Prefix at this moment
les ax,[bx]
; ES:AX=9FFF:20CD

FillLoop:
; set DX to 0. CWD works as: DX:AX = sign_extend(AX).
; AX here 0x20CD (at startup) or less then 320 (when getting back after loop),
    ; so DX will always be 0.
cwd
mov ax,di
; AX is current pointer within VGA buffer
div cx,
; divide current pointer by 320
mov cx,320
div cx
; DX (start_X) - remainder (column: 0..319); AX - result (row: 0..199)
```
sub ax,100
; AX=AX-100, so AX (start_Y) now is in range -100..99
DX is in range 0..319 or 0x0000..0x013F
dec dh
; DX now is in range 0xFF00..0x0003F (-256..63)
xor bx,bx
xor si,si
; BX (temp_X)=0; SI (temp_Y)=0
; get maximal number of iterations
; CX is still 320 here, so this is also maximal number of iteration

MandelLoop:
mov bp,si
; BP = temp_Y
imul si,bx
; SI = temp_X*temp_Y
add si,si
; SI = SI*2 = (temp_X*temp_Y)*2
imul bx,bx
; BX = BX^2 = temp_X^2
jo MandelBreak
; overflow?
imul bp,bp
; BP = BP^2 = temp_Y^2
jo MandelBreak
; overflow?
add bx,bp
; BX = BX+BP = temp_X^2 + temp_Y^2
jo MandelBreak
; overflow?
sub bx,bp
; BX = BX-BP = temp_X^2 - temp_Y^2
\frac{\text{correct scale:}}{\text{}}
sar bx,6
; BX=BX/64
add bx,dx
; BX=BX+start_X
; now temp_X = temp_X^2 - temp_Y^2 + start_X
sar si,6
; SI=SI/64
add si,ax
; SI=SI+start_Y
; now temp_Y = (temp_X*temp_Y)*2 + start_Y
loop MandelLoop

MandelBreak:
; CX=iterations
xchg ax,cx
; AX=iterations. store AL to VGA buffer at ES:[DI]
stosb
; stosb also increments DI, so DI now points to the next point in VGA buffer
; jump always, so this is eternal loop here
jmp FillLoop

Algorithm:
• Switch to 320*200 VGA video mode, 256 colors. \( 320 \times 200 = 64000 \) (0xFA00). Each pixel is encoded by one byte, so the buffer size is 0xFA00 bytes. It is addressed using the ES:DI registers pair.

ES must be 0xA000 here, because this is the segment address of the VGA video buffer, but storing 0xA000 to ES requires at least 4 bytes (PUSH 0A000h / POP ES). You can read more about the 16-bit MS-DOS memory model here: 94 on page 1278.

Assuming that BX is zero here, and the Program Segment Prefix is at the zeroth address, the 2-byte LES AX, [BX] instruction stores 0x20CD to AX and 0x9FFF to ES. So the program starts to draw 16 pixels (or bytes) before the actual video buffer. But this is MS-DOS, there is no memory protection, so a write happens into the very end of conventional memory, and usually, there is nothing important. That’s why you see a red strip 16 pixels wide at the right side. The whole picture is shifted left by 16 pixels. This is the price of saving 2 bytes.

• A infinite loop processes each pixel. Probably, the most common way to enumerate all pixels on the screen is with two loops: one for the X coordinate, another for the Y coordinate. But then you’ll need to multiply the coordinates to address a byte in the VGA video buffer. The author of this demo decided to do it otherwise: enumerate all bytes in the video buffer by using one single loop instead of two, and get the coordinates of the current point using division. The resulting coordinates are: X in the range of \(-256..63\) and Y in the range of \(-100..99\). You can see on the screenshot that the picture is somewhat shifted to the right part of screen. That’s because the biggest heart-shaped black hole usually appears on coordinates 0,0 and these are shifted here to right. Could the author just subtract 160 from the value to get X in the range of \(-160..159\)? Yes, but the instruction SUB DX, 160 takes 4 bytes, while DEC DH—2 bytes (which subtracts 0x100 (256) from DX). So the whole picture is shifted for the cost of another 2 bytes of saved space.

  - Check, if the current point is inside the Mandelbrot set. The algorithm is the one that has been described here.
  - The loop is organized using the LOOP instruction, which uses the CX register as counter. The author could set the number of iterations to some specific number, but he didn’t: 320 is already present in CX (was set at line 35), and this is good maximal iteration number anyway. We save here some space by not the reloading CX register with another value.
  - IMUL is used here instead of MUL, because we work with signed values: remember that the 0,0 coordinates has to be somewhere near the center of the screen. It’s the same with SAR (arithmetic shift for signed values):
it’s used instead of SHR.

- Another idea is to simplify the bounds check. We need to check a coordinate pair, i.e., two variables. What the author does is to checks thrice for overflow: two squaring operations and one addition. Indeed, we use 16-bit registers, which hold signed values in the range of -32768..32767, so if any of the coordinates is greater than 32767 during the signed multiplication, this point is definitely out of bounds: we jump to the MandelBreak label.

- There is also a division by 64 (SAR instruction). 64 sets scale. Try to increase the value and you can get a closer look, or to decrease if for a more distant look.

- We are at the MandelBreak label, there are two ways of getting here: the loop ended with CX=0 (the point is inside the Mandelbrot set); or because an overflow has happened (CX still holds some value). Now we write the low 8-bit part of CX (CL) to the video buffer. The default palette is rough, nevertheless, 0 is black: hence we see black holes in the places where the points are in the Mandelbrot set. The palette can be initialized at the program’s start, but remember, this is only a 64 bytes program!

- The program runs in an infinite loop, because an additional check where to stop, or any user interface will result in additional instructions.

Some other optimization tricks:

- The 1-byte CWD is used here for clearing DX instead of the 2-byte XOR DX, DX or even the 3-byte MOV DX, 0.

- The 1-byte XCHG AX, CX is used instead of the 2-byte MOV AX, CX. The current value of AX is not needed here anyway.

- DI (position in video buffer) is not initialized, and it is 0xFFFE at the start. That’s OK, because the program works for all DI in the range of 0..0xFFFF eternally, and the user can’t notice that it is started off the screen (the last pixel of a 320*200 video buffer is at address 0xF9FF). So some work is actually done off the limits of the screen. Otherwise, you’ll need an additional instructions to set DI to 0 and check for the video buffer’s end.

### 83.2.3 My “fixed” version

Listing 83.6: My “fixed” version

```
1  org 100h
2  mov al,13h
```

---

8 More information about initial register values: [http://go.yurichev.com/17004](http://go.yurichev.com/17004)
int 10h

; set palette
mov dx, 3c8h
mov al, 0
out dx, al
mov cx, 100h
inc dx
100:
mov al, cl
shl ax, 2
out dx, al ; red
out dx, al ; green
out dx, al ; blue
loop 100

push 0a000h
pop es
xor di, di
FillLoop:
cwd
mov ax,di
mov cx,320
div cx
sub ax,100
sub dx,160
xor bx,bx
xor si,si
MandelLoop:
mov bp,si
imul si,bx
add si,si
imul bx,bx
jo MandelBreak
imul bp,bp
jo MandelBreak
add bx,bp
jo MandelBreak
sub bx,bp
sub bx,bp
sar bx,6
add bx,dx
The author of these lines made an attempt to fix all these oddities: now the palette is smooth grayscale, the video buffer is at the correct place (lines 19..20), the picture is drawn on center of the screen (line 30), the program eventually ends and waits for the user’s keypress (lines 58..68). But now it’s much bigger: 105 bytes (or 54 instructions) \(^9\).

---

\(^9\)You can experiment by yourself: get DosBox and NASM and compile it as: nasm fiole.asm -fbin -o file.com

---

Figure 83.3: My “fixed” version
Part IX

Examples of reversing proprietary file formats
Chapter 84

Primitive XOR-encryption
Norton Guide\(^1\) was popular in the epoch of MS-DOS, it was a resident program that worked as a hypertext reference manual.

Norton Guide’s databases are files with the extension .ng, the contents of which look encrypted:

![View X68.JG - Far 2.0.1807 x64 Administrator](image)

Figure 84.1: Very typical look

Why did we think that it’s encrypted but not compressed? We see that the 0x1A byte (looking like “→”) occurs often, it would not be possible in a compressed file. We also see long parts that consist only of latin letters, and they look like strings in an unknown language.

\(^1\)wikipedia
Since the 0x1A byte occurs so often, we can try to decrypt the file, assuming that it's encrypted by the simplest XOR-encryption. If we apply XOR with the 0x1A constant to each byte in Hiew, we can see familiar English text strings:

XOR encryption with one single constant byte is the simplest possible encryption method, which is, nevertheless, encountered sometimes.

Now we understand why the 0x1A byte was occurring so often: because there are so many zero bytes and they were replaced by 0x1A in encrypted form.

But the constant might be different. In this case, we could try every constant in the 0..255 range and look for something familiar in the decrypted file. 256 is not so much.


84.1.1 Entropy

A very important property of such primitive encryption systems is that the information entropy of the encrypted/decrypted block is the same. Here is my analysis in Wolfram Mathematica 10.
What we do here is load the file, get its entropy, decrypt it, save it and get the entropy again (the same!). Mathematica also offers some well-known English language texts for analysis. So we also get the entropy of Shakespeare's sonnets, and it is close to the entropy of the file we just analyzed. The file we analyzed consists of English language sentences, which are close to the language of Shakespeare. And the XOR-ed bitwise English language text has the same entropy.

However, this is not true when the file is XOR-ed with a pattern larger than one byte.

The file we analyzed can be downloaded here: [http://go.yurichev.com/17350](http://go.yurichev.com/17350).

**One more word about base of entropy**

Wolfram Mathematica calculates entropy with base of $e$ (base of the natural logarithm), and the UNIX `ent` utility\(^2\) uses base 2. So we set base 2 explicitly in `Entropy` command, so Mathematica will give us the same results as the `ent` utility.

\(^2\)[http://www.fourmilab.ch/random/]
84.2 Simplest possible 4-byte XOR encryption

If a longer pattern was used for XOR-encryption, for example a 4 byte pattern, it’s easy to spot as well. For example, here is the beginning of the kernel32.dll file (32-bit version from Windows Server 2008):

![Figure 84.3: Original file](image)

The image shows the beginning of the kernel32.dll file, which includes machine code and metadata relevant to the file's structure and contents.
Here it is “encrypted” with a 4-byte key:

![Image of encrypted file](image.png)

**Figure 84.4: “Encrypted” file**

It’s very easy to spot the recurring 4 symbols. Indeed, the header of a PE-file has a lot of long zero areas, which are the reason for the key to become visible.
Here is the beginning of a PE-header in hexadecimal form:

![Image of PE-header](image-url)

Figure 84.5: PE-header
Here it is “encrypted”:

![Figure 84.6: “Encrypted” PE-header](image)

It’s easy to spot that the key is the following 4 bytes: 8C 61 D2 63. With this information, it’s easy to decrypt the whole file.

So it is important to remember these properties of PE-files: 1) PE-header has many zero-filled areas; 2) all PE-sections are padded with zeroes at a page boundary (4096 bytes), so long zero areas are usually present after each section.

Some other file formats may contain long zero areas. It’s typical for files used by scientific and engineering software.

For those who want to inspect these files on their own, they are downloadable here: [http://go.yurichev.com/17352](http://go.yurichev.com/17352).

### 84.2.1 Exercise

- [http://challenges.re/50](http://challenges.re/50)
Chapter 85

Millenium game save file

The “Millenium Return to Earth” is an ancient DOS game (1991), that allows you to mine resources, build ships, equip them on other planets, and so on\(^1\). Like many other games, it allows you to save all game state into a file. Let’s see if we can find something in it.

\(^{1}\) It can be downloaded for free [here](#)
So there is a mine in the game. Mines at some planets work faster, or slower on others. The set of resources is also different.

Here we can see what resources are mined at the time:

![Figure 85.1: Mine: state 1](image1.png)

Let's save a game state. This is a file of size 9538 bytes.

Let's wait some “days” here in the game, and now we’ve got more resources from the mine:

![Figure 85.2: Mine: state 2](image2.png)
Let's save game state again.

Now let's try to just do binary comparison of the save files using the simple DOS/Windows FC utility:

```
...> FC /b 2200save.i.v1 2200SAVE.I.V2

Comparing files 2200save.i.v1 and 2200SAVE.I.V2
00000016: 0D 04
00000017: 03 04
0000001C: 1F 1E
00000146: 27 3B
00000BDA: 0E 16
00000BDC: 66 9B
00000BDE: 0E 16
00000BE0: 0E 16
00000BE6: DB 4C
00000BE7: 00 01
00000BE8: 99 E8
00000BEC: A1 F3
00000BEE: 83 C7
00000BFB: A8 28
00000BFD: 98 18
00000BFF: A8 28
00000C01: A8 28
00000C07: D8 58
00000C09: E4 A4
00000C0D: 38 B8
00000C0F: E8 68
...
```

The output is incomplete here, there are more differences, but we will cut result to show the most interesting.

In the first state, we have 14 “units” of hydrogen and 102 “units” of oxygen. We have 22 and 155 “units” respectively in the second state. If these values are saved into the save file, we would see this in the difference. And indeed we do. There is 0x0E (14) at position 0xBDA and this value is 0x16 (22) in the new version of the file. This is probably hydrogen. There is 0x66 (102) at position 0xBDC in the old version and 0x9B (155) in the new version of the file. This seems to be the oxygen.

Both files are available on the website for those who wants to inspect them (or experiment) more: beginners.re.
Here is the new version of file opened in Hiew, we marked the values related to the resources mined in the game:

![Figure 85.3: Hiew: state 1](image)

Let’s check each, and these are. These are clearly 16-bit values: not a strange thing for 16-bit DOS software where the \textit{int} type has 16-bit width.
Let’s check our assumptions. We will write the 1234 (0x4D2) value at the first position (this must be hydrogen):

![Figure 85.4: Hiew: let's write 1234 (0x4D2) there](image)

Then we will load the changed file in the game and took a look at mine statistics:

![Figure 85.5: Let's check for hydrogen value](image)

So yes, this is it.
Now let's try to finish the game as soon as possible, set the maximal values everywhere:

Figure 85.6: Hiew: let's set maximal values

0xFFFF is 65535, so yes, we now have a lot of resources:

Figure 85.7: All resources are 65535 (0xFFFF) indeed
Let’s skip some “days” in the game and oops! We have a lower amount of some resources:

![Image of resource variables overflow]

Figure 85.8: Resource variables overflow

That’s just overflow. The game’s developer probably didn’t think about such high amounts of resources, so there are probably no overflow checks, but the mine is “working” in the game, resources are added, hence the overflows. Apparently, it was a bad idea to be that greedy.

There are probably a lot of more values saved in this file.

So this is very simple method of cheating in games. High score files often can be easily patched like that.

More about files and memory snapshots comparing: 63.4 on page 948.
Chapter 86

Oracle RDBMS: .SYM-files

When an Oracle RDBMS process experiences some kind of crash, it writes a lot of information into log files, including stack trace, like this:

```
----- Call Stack Trace -----
calling      call  entry      argument
           | | |     |     |     |     |
values in hex
location    type  point  (? means dubious value)
------------------------ ------------------------

_ksrrow() 00000000
_opifch2()+2729 CALLptr 00000000 23D4B914
          | | | |     |     |     |
          | | | | E47F264 1F19AE2

_kpoal8()+2832 CALLrel _opifch2() 89 5 EB1CC74
          | | | |     |     |     |
          | | | | EB1F0A0 0
_opiodr()+1248 CALLreg 00000000 5E 1C
          | | | | EB1F0A0 0
_ttcpip()+1051 CALLreg 00000000 5E 1C
          | | | | EB1F0A0 0
_opitsk()+1404 CALL??? 00000000
          | | | | | | EB1ED30
          | | | | | E52E 0 EB1F1F8
_opiino()+980 CALLrel _opitsk() 0 0
          | | | |     |     |     |
          | | | | EB1F0A0 0
_opiodr()+1248 CALLreg 00000000 3C 4 EB1FBF4
          | | | |     |     |     |
          | | | | EB1F0A0 0
_opidrv()+1201 CALLrel _opiodr() 3C 4 EB1FBF4
          | | | |     |     |     |
          | | | | EB1F0A0 0
_sou2o()+55 CALLrel _opidrv() 3C 4 EB1FBF4
          | | | |     |     |     |
          | | | | EB1F0A0 0
_opimai_real()+124 CALLrel _sou2o() EB1FC04 3C 4
          | | | |     |     |     |
          | | | | EB1FBF4
```
But of course, Oracle RDBMS’s executables must have some kind of debug information or map files with symbol information included or something like that.

Windows NT Oracle RDBMS has symbol information in files with .SYM extension, but the format is proprietary. (Plain text files are good, but needs additional parsing, hence offer slower access.)

Let’s see if we can understand its format. We will pick the shortest orawtc8.sym file that comes with the orawtc8.dll file in Oracle 8.1.7.

---

1 We can chose an ancient Oracle RDBMS version intentionally due to the smaller size of its modules
Here is the file opened in Hiew:

![The whole file in Hiew](image)

Figure 86.1: The whole file in Hiew

By comparing the file with other .SYM files, we can quickly see that OSYM is always header (and footer), so this is maybe the file’s signature.

We also see that basically, the file format is: OSYM + some binary data + zero delimited text strings + OSYM. The strings are, obviously, function and global variable names.
We will mark the OSYM signatures and strings here:

Well, let’s see. In Hiew, we will mark the whole strings block (except the trailing OSYM signatures) and put it into a separate file. Then we run UNIX `strings` and `wc` utilities to count the text strings:

```bash
strings strings_block | wc -l
```

66

So there are 66 text strings. Please note that number.

We can say, in general, as a rule, the number of *anything* is often stored separately in binary files. It’s indeed so, we can find the 66 value (0x42) at the file’s start, right after the OSYM signature:

```bash
$ hexdump -C orawtc8.sym
00000000 4f 53 59 4d 00 00 00 10 00 10 80 10 00 10 66
```

1237
Of course, 0x42 here is not a byte, but most likely a 32-bit value packed as little-endian, hence we see 0x42 and then at least 3 zero bytes.

Why do we believe it’s 32-bit? Because, Oracle RDBMS’s symbol files may be pretty big. The oracle.sym file for the main oracle.exe (version 10.2.0.4) executable contains 0x3A38E (238478) symbols. A 16-bit value isn’t enough here.

We can check other .SYM files like this and it proves our guess: the value after the 32-bit OSYM signature always reflects the number of text strings in the file.

It’s a general feature of almost all binary files: a header with a signature plus some other information about the file.

Now let’s investigate closer what this binary block is. Using Hiew again, we put the block starting at address 8 (i.e., after the 32-bit count value) ending at the strings block, into a separate binary file.
Let's see the binary block in Hiew:

![Figure 86.3: Binary block](image)

There is a clear pattern in it.
We will add red lines to divide the block:

Hiew, like almost any other hexadecimal editor, shows 16 bytes per line. So the pattern is clearly visible: there are 4 32-bit values per line.

The pattern is visually visible because some values here (till address 0x104) are always in 0x1000xxxx form, started with 0x10 and zero bytes. Other values (starting at 0x108) are in 0x0000xxxx form, so always started with two zero bytes.

Let’s dump the block as an array of 32-bit values:

Listing 86.1: first column is address

```
$ od -v -t x4 binary_block
0000000 10001000 10001080 100010f0 10001150
0000020 10001160 100011c0 100011d0 10001370
0000040 10001540 10001550 10001560 10001580
0000060 100015a0 100015a6 100015ac 100015b2
0000100 100015b8 100015be 100015c4 100015ca
0000120 100015d0 100015e0 100016b0 10001760
0000140 10001766 1000176c 10001780 100017b0
0000160 100017d0 100017e0 10001810 10001816
0000200 10002000 10002004 10002008 1000200c
```
There are 132 values, that's $66 \times 2$. Probably, there are two 32-bit values for each symbol, but maybe there are two arrays? Let's see.

Values starting with 0x1000 may be addresses. This is a .SYM file for a DLL after all, and the default base address of win32 DLLs is 0x10000000, and the code usually starts at 0x10001000.

When we open the orawtc8.dll file in IDA, the base address is different, but nevertheless, the first function is:

```
.text:60351000  sub_60351000  proc near
.text:60351000
.text:60351000  arg_0        = dword ptr 8
.text:60351000  arg_4        = dword ptr 0Ch
.text:60351000  arg_8        = dword ptr 10h
.text:60351000
.text:60351000  push  ebp
.text:60351001  mov  ebp, esp
.text:60351003  mov  eax, dword_60353014
.text:60351008  cmp  eax, 0FFFFFFFFh
.text:6035100B  jnz short loc_6035104F
.text:6035100D  mov  ecx, hModule
```
Wow, “ax_reg” string sounds familiar. It’s indeed the first string in the strings block! So the name of this function seems to be “ax_reg”.

The second function is:

```
.text:60351080 sub_60351080 proc near
.text:60351080 arg_0 = dword ptr 8
.text:60351080 arg_4 = dword ptr 0Ch
.text:60351080 push ebp
.text:60351081 mov ebp, esp
.text:60351083 mov eax, dword_60353018
.text:60351088 cmp eax, 0FFFFFFFFh
.text:6035108B jnz short loc_603510CF
.text:6035108D xor eax, eax
.text:60351093 mov ecx, hModule
.text:60351095 cmp ecx, 0FFFFFFFFh
.text:60351098 mov dword_60353018, eax
.text:6035109D jnz short loc_603510B1
.text:6035109F call sub_603510F0
.text:603510A4 mov ecx, eax
.text:603510A6 mov eax, dword_60353018
.text:603510AB mov hModule, ecx
.text:603510B1 loc_603510B1: ; CODE ↵
.text:603510B1 test ecx, ecx
```
The “ax_unreg” string is also the second string in the strings block! The starting address of the second function is 0x60351080, and the second value in the binary block is 10001080. So this is the address, but for a DLL with the default base address.

We can quickly check and be sure that the first 66 values in the array (i.e., the first half of the array) are just function addresses in the DLL, including some labels, etc. Well, what's the other part of array then? The other 66 values that start with 0x0000? These seem to be in range [0...0x3F8]. And they do not look like bitfields: the series of numbers is increasing. The last hexadecimal digit seems to be random, so, it's unlikely the address of something (it would be divisible by 4 or maybe 8 or 0x10 otherwise).

Let's ask ourselves: what else Oracle RDBMS's developers would save here, in this file? Quick wild guess: it could be the address of the text string (function name). It can be quickly checked, and yes, each number is just the position of the first character in the strings block.

This is it! All done.

We will write an utility to convert these .SYM files into IDA script, so we can load the .idc script and it sets the function names:

```c
#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include <assert.h>
#include <malloc.h>
#include <fcntl.h>
#include <string.h>

int main (int argc, char *argv[]) {
    uint32_t sig, cnt, offset;
    uint32_t *d1, *d2;
    int h, i, remain, file_len;
    char *d3;
    uint32_t array_size_in_bytes;

    assert (argv[1]); // file name
```
assert (argv[2]); // additional offset (if needed)

// additional offset
assert (sscanf (argv[2], "%X", &offset)==1);

// get file length
assert ((h=open (argv[1], _O_RDONLY | _O_BINARY, 0))!=-1);
assert ((file_len=lseek (h, 0, SEEK_END))!=-1);
assert (lseek (h, 0, SEEK_SET)!=-1);

// read signature
assert (read (h, &sig, 4)==4);
// read count
assert (read (h, &cnt, 4)==4);

assert (sig==0x4D59534F); // OSYM

// skip timedatestamp (for 11g)
//_lseek (h, 4, 1);

array_size_in_bytes=cnt*sizeof(uint32_t);

// load symbol addresses array
d1=(uint32_t*)malloc (array_size_in_bytes);
assert (d1);
assert (read (h, d1, array_size_in_bytes)==array_size_in_bytes);

// load string offsets array
d2=(uint32_t*)malloc (array_size_in_bytes);
assert (d2);
assert (read (h, d2, array_size_in_bytes)==array_size_in_bytes);

// calculate strings block size
remain=file_len-(8+4)-(cnt*8);

// load strings block
assert (d3=(char*)malloc (remain));
assert (read (h, d3, remain)==remain);

printf ("#include <idc.idc>\n\n");
printf ("static main() {\n\n");
for (i=0; i<cnt; i++)
Here is an example of its work:

```c
#include <idc.idc>

static main() {
    MakeName(0x60351000, "_ax_reg");
    MakeName(0x60351080, "_ax_unreg");
    MakeName(0x603510F0, "_loaddll");
    MakeName(0x60351150, "_wtcsrin0");
    MakeName(0x60351160, "_wtcsrin");
    MakeName(0x603511C0, "_wtcsrfre");
    MakeName(0x603511D0, "_wtclkm");
    MakeName(0x60351370, "_wtcstu");
    ...
}
```

The example files were used in this example are here: beginners.re.
Oh, let’s also try Oracle RDBMS for win64. There has to be 64-bit addresses instead, right?

The 8-byte pattern is visible even easier here:

![Figure 86.5: .SYM-file example from Oracle RDBMS for win64](image)

So yes, all tables now have 64-bit elements, even string offsets! The signature is now OSYMAM64, to distinguish the target platform, apparently.

This is it! Here is also library which has functions to access Oracle RDBMS.SYM-files: GitHub.
Chapter 87

Oracle RDBMS: .MSB-files

When working toward the solution of a problem, it always helps if you know the answer.

Murphy's Laws, Rule of Accuracy

This is a binary file that contains error messages with their corresponding numbers. Let’s try to understand its format and find a way to unpack it.

There are Oracle RDBMS error message files in text form, so we can compare the text and packed binary files.

This is the beginning of the ORAUS.MSG text file with some irrelevant comments stripped:

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>&quot;normal, successful completion&quot;</td>
</tr>
<tr>
<td>00001</td>
<td>&quot;unique constraint (%s.%s) violated&quot;</td>
</tr>
<tr>
<td>00017</td>
<td>&quot;session requested to set trace event&quot;</td>
</tr>
<tr>
<td>00018</td>
<td>&quot;maximum number of sessions exceeded&quot;</td>
</tr>
<tr>
<td>00019</td>
<td>&quot;maximum number of session licenses exceeded&quot;</td>
</tr>
<tr>
<td>00020</td>
<td>&quot;maximum number of processes (%s) exceeded&quot;</td>
</tr>
<tr>
<td>00021</td>
<td>&quot;session attached to some other process; cannot switch session&quot;</td>
</tr>
<tr>
<td>00022</td>
<td>&quot;invalid session ID; access denied&quot;</td>
</tr>
<tr>
<td>00023</td>
<td>&quot;session references process private memory; cannot detach session&quot;</td>
</tr>
<tr>
<td>00024</td>
<td>&quot;logins from more than one process not allowed in single-process mode&quot;</td>
</tr>
</tbody>
</table>

Open-source text files don’t exist in Oracle RDBMS for every .MSB file, so that’s why we will work on their file format.
00025, 00000, "failed to allocate %s"
00026, 00000, "missing or invalid session ID"
00027, 00000, "cannot kill current session"
00028, 00000, "your session has been killed"
00029, 00000, "session is not a user session"
00030, 00000, "User session ID does not exist."
00031, 00000, "session marked for kill"
...

The first number is the error code. The second is perhaps maybe some additional flags.
Now let’s open the ORAUS.MSB binary file and find these text strings. And there are:

![Hiew: first block](image)

Figure 87.1: Hiew: first block

We see the text strings (including those from the beginning of the ORAUS.MSG file) interleaved with some binary values. By quick investigation, we can see that main part of the binary file is divided by blocks of size 0x200 (512) bytes.
Let's see the contents of the first block:

![Figure 87.2: Hiew: first block](image)

Here we see the texts of the first messages errors. What we also see is that there are no zero bytes between the error messages. This implies that these are not null-terminated C strings. As a consequence, the length of each error message must be encoded somehow. Let's also try to find the error numbers. The ORAUS.MSG files starts with these: 0, 1, 17 (0x11), 18 (0x12), 19 (0x13), 20 (0x14), 21 (0x15), 22 (0x16), 23 (0x17), 24 (0x18)... We will find these numbers in the beginning of the block and mark them with red lines. The period between error codes is 6 bytes. This implies that there are probably 6 bytes of information allocated for each error message.

The first 16-bit value (0xA here or 10) mean thenumber of messages in each block: this can be checked by investigating other blocks. Indeed: the error messages have arbitrary size. Some are longer, some are shorter. But block size is always fixed, hence, you never know how many text messages can be packed in each block.

As we already noted, since these are not null-terminating C strings, their size must be encoded somewhere. The size of the first string “normal, successful completion” is 29 (0x1D) bytes. The size of the second string “unique constraint (%s.%s) violated” is 34 (0x22) bytes. We can’t find these values (0x1D or/and 0x22) in the block.

There is also another thing. Oracle RDBMS has to determine the position of the
string it needs to load in the block, right? The first string “normal, successful completion” starts at position 0x1444 (if we count starting at the beginning of the file) or at 0x44 (from the block’s start). The second string “unique constraint (%s.%s) violated” starts at position 0x1461 (from the file’s start) or at 0x61 (from the at the block’s start). These numbers (0x44 and 0x61) are familiar somehow! We can clearly see them at the start of the block.

So, each 6-byte block is:

- 16-bit error number;
- 16-bit zero (maybe additional flags);
- 16-bit starting position of the text string within the current block.

We can quickly check the other values and be sure our guess is correct. And there is also the last “dummy” 6-byte block with an error number of zero and starting position beyond the last error message’s last character. Probably that’s how text message length is determined? We just enumerate 6-byte blocks to find the error number we need, then we get the text string’s position, then we get the position of the text string by looking at the next 6-byte block! This way we determine the string’s boundaries! This method allows to save some space by not saving the text string's size in the file! It’s not possible to say it saves a lot of space, but it’s a clever trick.
Let's back to the header of .MSB-file:

Figure 87.3: Hiew: file header

Now we can quickly find the number of blocks in the file (marked by red). We can checked other .MSB-files and we see that it's true for all of them. There are a lot of other values, but we will not investigate them, since our job (an unpacking utility) was done. If we have to write a .MSB file packer, we would probably need to understand the meaning of the other values.
There is also a table that came after the header which probably contains 16-bit values:

![Figure 87.4: Hiew: last_errno table](image)

Their size can be determined visually (red lines are drawn here). While dumping these values, we have found that each 16-bit number is the last error code for each block.

So that's how Oracle RDBMS quickly finds the error message:

- load a table we will call last_errno (that contains the last error number for each block);
- find a block that contains the error code we need, assuming all error codes increase across each block and across the file as well;
- load the specific block;
- enumerate the 6-byte structures until the specific error number is found;
- get the position of the first character from the current 6-byte block;
- get the position of the last character from the next 6-byte block;
- load all characters of the message in this range.

This is C program that we wrote which unpacks .MSB-files: `beginners.re`.
There are also the two files which were used in the example (Oracle RDBMS 11.1.0.6): beginners.re, beginners.re.

87.1 Summary

The method is probably too old-school for modern computers. Supposedly, this file format was developed in the mid-80’s by someone who also coded for big iron with memory/disk space economy in mind. Nevertheless, it was an interesting and yet easy task to understand a proprietary file format without looking into Oracle RDBMS’s code.
Part X

Other things
Chapter 88

npad

It is an assembly language macro for aligning labels on a specific boundary. That's often needed for the busy labels to where the control flow is often passed, e.g., loop body starts. So the CPU can load the data or code from the memory effectively, through the memory bus, cache lines, etc.

Taken from listing.inc (MSVC):

By the way, it is a curious example of the different NOP variations. All these instructions have no effects whatsoever, but have a different size.

Having a single idle instruction instead of couple of NOP-s, is accepted to be better for CPU performance.

```plaintext
;; LISTING.INC
;;
;; This file contains assembler macros and is included by the files created
;; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

;; non destructive nops
npad macro size
if size eq 1
  nop
else
  if size eq 2
    mov edi, edi
```
else
    if size eq 3
        ; lea ecx, [ecx+00]
        DB 8DH, 49H, 00H
    else
        if size eq 4
            ; lea esp, [esp+00]
            DB 8DH, 64H, 24H, 00H
        else
            if size eq 5
                add eax, DWORD PTR 0
            else
                if size eq 6
                    ; lea ebx, [ebx+00000000]
                    DB 8DH, 9BH, 00H, 00H, 00H, 00H
                else
                    if size eq 7
                        ; lea esp, [esp+00000000]
                        DB 8DH, 0A4H, 24H, 00H, 00H, 00H
                    else
                        if size eq 8
                            ; jmp .+8; .npad 6
                            DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H
                        else
                            if size eq 9
                                ; jmp .+9; .npad 7
                                DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H
                            else
                                if size eq 10
                                    ; jmp .+A; .npad 7; .npad 1
                                    DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H
                                else
                                    if size eq 11
                                        ; jmp .+B; .npad 7; .npad 2
                                        DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H
                                    else
                                        if size eq 12
                                            ; jmp .+C; .npad 7; .npad 3
                                            DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H
                                        else
                                            if size eq 13
                                                ; jmp .+D; .npad 7; .npad 4
                                                DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H
                                            else
                                                ; 8DH, 64H, 24H, 00H
                                            else
                                                ; 8DH, 64H, 24H, 00H
                                            else
                                                ; 8DH, 64H, 24H, 00H
                                        else
                                            ; 8DH, 64H, 24H, 00H
                                    else
                                        ; 8DH, 64H, 24H, 00H
                                else
                                    ; 8DH, 64H, 24H, 00H
                            else
                                ; 8DH, 64H, 24H, 00H
                        else
                            ; 8DH, 64H, 24H, 00H
                    else
                        ; 8DH, 64H, 24H, 00H
                else
                    ; 8DH, 64H, 24H, 00H
            else
                ; 8DH, 64H, 24H, 00H
        else
            ; 8DH, 64H, 24H, 00H
    else
        ; 8DH, 64H, 24H, 00H
if size eq 14
  ; jmp .+E; .npad 7; .npad 5
  DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 05H, 00H, 00H, 00H, 00H
else
  if size eq 15
    ; jmp .+F; .npad 7; .npad 6
    DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 9BH, 00H, 00H, 00H
  else
    %out error: unsupported npad size
    .err
  endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endi
Chapter 89

Executable files patching

89.1 Text strings

The C strings are the thing that is the easiest to patch (unless they are encrypted) in any hex editor. This technique is available even for those who are not aware of machine code and executable file formats. The new string has not to be bigger than the old one, because there's a risk of overwriting another value or code there. Using this method, a lot of software was localized in the MS-DOS era, at least in the ex-USSR countries in 80’s and 90’s. It was the reason why some weird abbreviations were present in the localized software: there was no room for longer strings.

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

89.2 x86 code

Frequent patching tasks are:

- One of the most frequent jobs is to disable some instruction. It is often done by filling it using byte 0x90 (NOP).

- Conditional jumps, which have an opcode like 74 xx (JZ), can be filled with two NOPs. It is also possible to disable a conditional jump by writing 0 at the second byte (jump offset).

- Another frequent job is to make a conditional jump to always trigger: this can be done by writing 0xEB instead of the opcode, which stands for JMP.

- A function’s execution can be disabled by writing RETN (0xC3) at its beginning. This is true for all functions excluding stdcall (64.2 on page 952).
While patching \texttt{stdcall} functions, one has to determine the number of arguments (for example, by finding RETN in this function), and use RETN with a 16-bit argument (0xC2).

- Sometimes, a disabled functions has to return 0 or 1. This can be done by \texttt{MOV EAX, 0} or \texttt{MOV EAX, 1}, but it’s slightly verbose. A better way is \texttt{XOR EAX, EAX} (2 bytes 0x31 0xC0) or \texttt{XOR EAX, EAX / INC EAX} (3 bytes 0x31 0xC0 0x40).

A software may be protected against modifications. This protection is often done by reading the executable code and calculating a checksum. Therefore, the code must be read before protection is triggered. This can be determined by setting a breakpoint on reading memory.

\texttt{tracer} has the BPM option for this.

PE executable file relocations (\textit{68.2.6 on page 992}) must not to be touched while patching, because the Windows loader may overwrite your new code. (They are grayed in Hiew, for example: \textit{fig.7.12}). As a last resort, it is possible to write jumps that circumvent the relocations, or you will need to edit the relocations table.
Chapter 90

Compiler intrinsic

A function specific to a compiler which is not an usual library function. The compiler generates a specific machine code instead of a call to it. It is often a pseudofunction for specific CPU instruction.

For example, there are no cyclic shift operations in C/C++ languages, but they are present in most CPUs. For programmer's convenience, at least MSVC has pseudofunctions _rotl0 and _rotr0\(^1\) which are translated by the compiler directly to the ROL/ROR x86 instructions.

Another example are functions to generate SSE-instructions right in the code. Full list of MSVC intrinsics: MSDN.

\(^1\)MSDN
Chapter 91

Compiler’s anomalies

Intel C++ 10.1, which was used for Oracle RDBMS 11.2 Linux86 compilation, may emit two JZ in row, and there are no references to the second JZ. The second JZ is thus meaningless.

Listing 91.1: kdli.o from libserver11.a

| .text:08114CF1 | loc_8114CF1: ; CODE XREF: ⤲ |
| └── PGOSF539_kdlimemSer+89A
| .text:08114CF1 | ; ⤲ |
| └── PGOSF539_kdlimemSer+3994
| .text:08114CF1 8B 45 08 | mov eax, [ebp+arg_0] |
| .text:08114CF4 0F B6 50 14 | movzx edx, byte ptr [eax+14h] |
| .text:08114CF8 F6 C2 01 | test dl, 1 |
| .text:08114CFB 0F 85 17 08 00 00 | jnz loc_8115518 |
| .text:08114D01 85 C9 | test ecx, ecx |
| .text:08114D09 0F 84 09 08 00 00 | jz loc_8115518 |
| .text:08114D0F 8B 53 08 | mov edx, [ebx+8] |
| .text:08114D12 85 55 FC | mov [ebp+var_4], edx |
| .text:08114D15 31 C0 | xor eax, eax |
| .text:08114D17 89 45 F4 | mov [ebp+var_C], eax |
| .text:08114D1A 50 | push eax |
| .text:08114D1B 52 | push edx |
| .text:08114D1C E8 03 54 00 00 | call len2nbytes |
| .text:08114D21 83 C4 08 | add esp, 8 |

Listing 91.2: from the same code

| .text:0811A2A5 | loc_811A2A5: ; CODE XREF: ⤲ |
| └── kdliSerLengths+11C |
It is probably a code generator bug that was not found by tests, because resulting code works correctly anyway.

Other compiler anomalies here in this book: 19.2.4 on page 425, 39.3 on page 667, 47.7 on page 732, 18.7 on page 408, 12.4.1 on page 200, 19.5.2 on page 449.

Such cases are demonstrated here in this book, to show that such compilers errors are possible and sometimes one should not to rack one’s brain while thinking why did the compiler generate such strange code.
Chapter 92

OpenMP

OpenMP is one of the simplest ways to parallelize simple algorithms.

As an example, let’s try to build a program to compute a cryptographic nonce. In my simplistic example, the nonce is a number added to the plain unencrypted text in order to produce a hash with some specific features. For example, at some step, the Bitcoin protocol requires to find such nonce so the resulting hash contains a specific number of consecutive zeroes. This is also called “proof of work” ¹ (i.e., the system proves that it did some intensive calculations and spent some time for it).

My example is not related to Bitcoin in any way, it will try to add numbers to the “hello, world!_” string in order to find such number that when “hello, world!_<number>” is hashed with the SHA512 algorithm, it will contain at least 3 zero bytes.

Let’s limit our brute-force to the interval in 0..INT32_MAX-1 (i.e., 0x7FFFFFFE or 2147483646).

The algorithm is pretty straightforward:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "sha512.h"

int found=0;
int32_t checked=0;

int32_t* __min;
int32_t* __max;
```

¹wikipedia
time_t start;

#ifdef __GNUC__
#define min(X,Y) (((X) < (Y) ? (X) : (Y))
#define max(X,Y) (((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
{
    uint8_t buf[32];
    struct sha512_ctx ctx;
    uint8_t res[64];

    // update statistics
    int t=omp_get_thread_num();
    if (__min[t]==-1)
        __min[t]=nonce;
    if (__max[t]==-1)
        __max[t]=nonce;
    __min[t]=min(__min[t], nonce);
    __max[t]=max(__max[t], nonce);

    // idle if valid nonce found
    if (found)
        return;
    memset (buf, 0, sizeof(buf));
    sprintf (buf, "hello, world!_%d", nonce);
    sha512_init_ctx (&ctx);
    sha512_process_bytes (buf, strlen(buf), &ctx);
    sha512_finish_ctx (&ctx, &res);
    if (res[0]==0 && res[1]==0 && res[2]==0)
    {
        printf ("found (thread %d): [%s]. seconds spent
⤦
=\%d\n", t, buf, time(NULL)-start);
        found=1;
    }
    #pragma omp atomic
    checked++;

    #pragma omp critical
    if ((checked % 100000)==0)
        printf ("checked=\%d\n", checked);
int main()
{
    int32_t i;
    int threads=omp_get_max_threads();
    printf ("threads=%d\n", threads);

    __min=(int32_t*)malloc(threads*sizeof(int32_t));
    __max=(int32_t*)malloc(threads*sizeof(int32_t));
    for (i=0; i<threads; i++)
        __min[i]=__max[i]=-1;

    start=time(NULL);

    #pragma omp parallel for
    for (i=0; i<INT32_MAX; i++)
        check_nonce (i);

    for (i=0; i<threads; i++)
        printf ("__min[%d]=0x%08x __max[%d]=0x%08x\n", i, __min[i], i, __max[i]);

    free(__min); free(__max);
}

The check_nonce() function just adds a number to the string, hashes it with the SHA512 algorithm and checks for 3 zero bytes in the result.

A very important part of the code is:

```
#pragma omp parallel for
for (i=0; i<INT32_MAX; i++)
    check_nonce (i);
```

Yes, that simple, without #pragma we just call check_nonce() for each number from 0 to INT32_MAX (0x7fffffff or 2147483647). With #pragma, the compiler adds some special code which slices the loop interval into smaller ones, to run them on all CPU cores available.

The example can be compiled in MSVC 2012:

```
cl openmp_example.c sha512.obj /openmp /O1 /Zi /Faopenmp_example.asm
```

2. N.B.: This is intentionally simplest possible example, but in practice, the usage of OpenMP can be harder and more complex

3. sha512.(c|h) and u64.h files can be taken from the OpenSSL library: http://go.yurichev.com/17324
Or in GCC:

```
gcc -fopenmp 2.c sha512.c -S -masm=intel
```

## 92.1 MSVC

Now this is how MSVC 2012 generates the main loop:

```
push OFFSET _main$omp$1
push 0
push 1
call __vcomp_fork
add esp, 16 ;
```

All functions prefixed by vcomp are OpenMP-related and are stored in the vcomp*.dll file. So here a group of threads is started.

Let's take a look on _main$omp$1:

```
$T1 = -8 ; size
= 4
$T2 = -4 ; size
= 4
_main$omp$1 PROC
; COMDAT
push ebp
mov ebp, esp
push ecx
push ecx
push esi
lea eax, DWORD PTR $T2[ebp]
push eax
lea eax, DWORD PTR $T1[ebp]
push eax
push 1
push 1
push 2147483646 ; 7
push 0
call __vcomp_for_static_simple_init
mov esi, DWORD PTR $T1[ebp]
```
This function is to be started \( n \) times in parallel, where \( n \) is the number of CPU cores. vcomp_for_static_simple_init() calculates the interval for the for() construct for the current thread, depending on the current thread's number. The loop's start and end values are stored in the $T1$ and $T2$ local variables. You may also notice 7ffffffeh (or 2147483646) as an argument to the vcomp_for_static_simple_init() function—this is the number of iterations for the whole loop, to be divided evenly.

Then we see a new loop with a call to the check_nonce() function, which does all the work.

Let's also add some code in the beginning of the check_nonce() function to gather statistics about the arguments with which the function was called.

This is what we see when we run it:

```plaintext
threads=4
...
checked=2800000
checked=3000000
checked=3200000
checked=3300000
found (thread 3): [hello, world!_1611446522]. seconds spent=3
__min[0]=0x00000000 __max[0]=0x1fffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7fffffff
```

Yes, the result is correct, the first 3 bytes are zeroes:
The running time is \( \approx 2.3 \) seconds on 4-core Intel Xeon E3-1220 3.10 GHz. In the task manager we see 5 threads: 1 main thread + 4 more. No further optimizations are done to keep this example as small and clear as possible. But probably it can be done much faster. My CPU has 4 cores, that is why OpenMP started exactly 4 threads.

By looking at the statistics table we can clearly see how the loop was sliced in 4 even parts. Oh well, almost even, if we don’t consider the last bit.

There are also pragmas for atomic operations.

Let’s see how this code is compiled:

```c
#pragma omp atomic
checked++;

#pragma omp critical
if ((checked % 100000) == 0)
   printf ("checked=%d\n", checked);
```

Listing 92.3: MSVC 2012

```assembly
push    edi
push OFFSET _checked
call    __vcomp_atomic_add_i4
; Line 55
push OFFSET _$vcomp$critsect$
call    __vcomp_enter_critsect
add esp, 12
; Line 56
mov     ecx, DWORD PTR _checked
mov     eax, ecx
cdq
mov     esi, 100000
; Line 57
idiv    esi
test    edx, edx
jne     SHORT $LN1@check_nonc
push    ecx
push OFFSET ??_C@_0M@NPNHLIO0@checked?$DN?$CFd?6?
; Line 58
```

1269
As it turns out, the vcomp_atomic_add_i4() function in the vcomp*.dll is just a tiny function with the LOCK XADD instruction in it.

vcomp_enter_critsect() eventually calling win32 API function EnterCriticalSection.

### 92.2 GCC

GCC 4.8.1 produces a program which shows exactly the same statistics table, so, GCC’s implementation divides the loop in parts in the same fashion.

#### Listing 92.4: GCC 4.8.1

```assembly
mov edi, OFFSET FLAT:main._omp_fn.0
call GOMP_parallel_start
mov edi, 0
call main._omp_fn.0
call GOMP_parallel_end
```

Unlike MSVC’s implementation, what GCC code does is to start 3 threads, and run the fourth in the current thread. So there are 4 threads instead of the 5 in MSVC.

Here is the main._omp_fn.0 function:

#### Listing 92.5: GCC 4.8.1

```
main._omp_fn.0:
push rbp
mov rbp, rsp
push rbx
sub rsp, 40
mov QWORD PTR [rbp-40], rdi
call omp_get_num_threads
mov ebx, eax
call omp_get_thread_num
mov esi, eax
```

---

4 Read more about LOCK prefix: A.6.1 on page 1301  
5 You can read more about critical sections here: 68.4 on page 1033
Here we see the division clearly: by calling `omp_get_num_threads()` and `omp_get_thread_num()` we get the number of threads running, and also the current thread's number, and then determine the loop's interval. Then we run `check_nonce()`.

GCC also inserted the `LOCK ADD` instruction right in the code, unlike MSVC, which generated a call to a separate DLL function:

**Listing 92.6: GCC 4.8.1**

```assembly
lock add DWORD PTR checked[rip], 1
call GOMP_critical_start
mov ecx, DWORD PTR checked[rip]
```
The functions prefixed with GOMP are from GNU OpenMP library. Unlike vcomp*.dll, its source code is freely available: GitHub.
Chapter 93

Itanium

Although almost failed, Intel Itanium (IA64) is a very interesting architecture. While OOE CPUs decide how to rearrange their instructions and execute them in parallel, EPIC\(^1\) was an attempt to shift these decisions to the compiler: to let it group the instructions at the compile stage.

This resulted in notoriously complex compilers.

Here is one sample of IA64 code: simple cryptographic algorithm from the Linux kernel:

```
Listing 93.1: Linux kernel 3.2.0.4

#define TEA_ROUNDS 32
#define TEA_DELTA 0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
    u32 y, z, n, sum = 0;
    u32 k0, k1, k2, k3;
    struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
    const __le32 *in = (const __le32 *)src;
    __le32 *out = (__le32 *)dst;

    y = le32_to_cpu(in[0]);
    z = le32_to_cpu(in[1]);

    k0 = ctx->KEY[0];
    k1 = ctx->KEY[1];
    k2 = ctx->KEY[2];
```

\(^1\)Explicitly parallel instruction computing
k3 = ctx->KEY[3];

n = TEA_ROUNDS;

while (n-- > 0) {
    sum += TEA_DELTA;
    y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1);
    z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3);
}

out[0] = cpu_to_le32(y);
out[1] = cpu_to_le32(z);

Here is how it was compiled:

Listing 93.2: Linux Kernel 3.2.0.4 for Itanium 2 (McKinley)

```
0090| tea_encrypt:
0090|     08 80 80 41 00 21 adds r16 = 96, r32
      // ptr to ctx->KEY[2]
0096|     80 C0 82 00 42 00 adds r8 = 88, r32
      // ptr to ctx->KEY[0]
009C|     00 00 04 00 nop.i 0
00A0|     09 18 70 41 00 21 adds r3 = 92, r32
      // ptr to ctx->KEY[1]
00A6|     F0 20 88 20 28 00 ld4 r15 = [r34], 4
      // load z
00AC|     44 06 01 84 adds r32 = 100, r32
      // ptr to ctx->KEY[3]
00B0|     08 98 00 20 10 10 ld4 r19 = [r16]
      // r19=k2
00B6|     00 01 00 00 42 40 mov r16 = r0
      // r0 always contain zero
00BC|     00 08 CA 00 mov.i r2 = ar.lc
      // save lc register
00C0|     05 70 00 44 10 10 9E FF FF FF 7F 20 ld4 r14 = [r34]
      // load y
00CC|     92 F3 CE 6B movl r17 = 0
      // xFFFFFFFFF9E3779B9;; // TEA_DELTA
00D0|     08 00 00 00 01 00 nop.m 0
00D6|     50 01 20 20 20 00 ld4 r21 = [r8]
      // r21=k0
00DC|     F0 09 2A 00 mov.i ar.lc = 31
      // TEA_ROUNDS is 32
00E0|     0A A0 00 06 10 10 ld4 r20 = [r3];
```
00E6|20 01 80 20 20 00          ld4 r18 = [r32]  \\
00EC|00 00 04 00                  nop.i 0  \\
00F0|                                  loc_F0:  \\
00F0|09 80 40 22 00 20            add r16 = r16, r17  \\
      // r16=sum, r17=TEA_DELTA  \\
00F6|D0 71 54 26 40 80         shladd r29 = r14, 4,  \\
     r21    // r14=y, r21=k0  \\
00FC|A3 70 68 52 27            extr.u r28 = r14, 5,  \\
0100|03 F0 40 1C 00 20            add r30 = r16, r14  \\
0106|B0 E1 50 00 40 40            add r27 = r28, r20;  \\
010C|D3 F1 3C 80 27            xor r26 = r29, r30;  \\
0110|0B C8 6C 34 0F 20            xor r25 = r27, r26;  \\
0116|F0 78 64 00 40 00        add r15 = r15, r25  \\
      // r15=z  \\
0120|00 00 04 00                  nop.i 0;  \\
0126|80 51 3C 34 29 60 27        extr.u r24 = r15, 5,  \\
0130|0B B8 3C 20 00 20            add r23 = r15, r16;  \\
0136|A0 C0 48 00 40 00            add r10 = r24, r18  \\
013C|00 00 04 00                  nop.i 0;  \\
0140|0B 48 28 16 0F 20            xor r9 = r10, r11;  \\
0146|60 B9 24 1E 40 00            xor r22 = r23, r9  \\
014C|00 00 04 00                  nop.i 0;  \\
0150|11 00 00 00 01 00            nop.m 0  \\
0156|E0 70 58 00 40 A0            add r14 = r14, r22  \\
015C|A0 FF FF 48              br.cloop.sptk.few  \\
      // loc_F0;  \\
0160|09 20 3C 42 90 15          st4 [r33] = r15, 4  \\
      // store z  \\
0166|00 00 00 02 00 00          nop.m 0  \\
016C|20 08 AA 00              mov.i ar.lc = r2;  \\
      // restore lc register  \\
0170|11 00 38 42 90 11          st4 [r33] = r14  \\
      // store y  \\
0176|00 00 00 02 00 80        nop.i 0  \\
017C|08 00 84 00                     br.ret.sptk.many b0;  \\
      ;  \\
1275
First of all, all IA64 instructions are grouped into 3-instruction bundles. Each bundle has a size of 16 bytes (128 bits) and consists of template code (5 bits) + 3 instructions (41 bits for each). IDA shows the bundles as 6+6+4 bytes—you can easily spot the pattern.

All 3 instructions from each bundle usually executes simultaneously, unless one of instructions has a “stop bit”.

Supposedly, Intel and HP engineers gathered statistics on most frequent instruction patterns and decided to bring bundle types (AKA “templates”): a bundle code defines the instruction types in the bundle. There are 12 of them. For example, the zeroth bundle type is MII, which implies the first instruction is Memory (load or store), the second and third ones are I (integer instructions). Another example is the bundle of type 0x1d: MFB: the first instruction is Memory (load or store), the second one is Float (FPU instruction), and the third is Branch (branch instruction).

If the compiler cannot pick a suitable instruction for the relevant bundles slot, it may insert a NOP: you can see here the nop . i instructions (NOP at the place where the integer instruction might be) or nop . m (a memory instruction might be at this slot). NOPs are inserted automatically when one uses assembly language manually.

And that is not all. Bundles are also grouped. Each bundle may have a “stop bit”, so all the consecutive bundles with a terminating bundle which has the “stop bit” can be executed simultaneously. In practice, Itanium 2 can execute 2 bundles at once, resulting in the execution of 6 instructions at once.

So all instructions inside a bundle and a bundle group cannot interfere with each other (i.e., must not have data hazards). If they do, the results are to be undefined.

Each stop bit is marked in assembly language as two semicolons (; ;) after the instruction. So, the instructions at [90-ac] may be executed simultaneously: they do not interfere. The next group is [b0-cc].

We also see a stop bit at 10c. The next instruction at 110 has a stop bit too. This implies that these instructions must be executed isolated from all others (as in CISC). Indeed: the next instruction at 110 uses the result from the previous one (the value in register r26), so they cannot be executed at the same time. Apparently, the compiler was not able to find a better way to parallelize the instructions, in other words, to load CPU as much as possible, hence too much stop bits and NOPs.

Manual assembly programming is a tedious job as well: the programmer has to group the instructions manually.

The programmer is still able to add stop bits to each instructions, but this will degrade the performance that Itanium was made for.

An interesting examples of manual IA64 assembly code can be found in the Linux kernel’s sources:

Another introductory paper on Itanium assembly: [Bur], [haq].

Another very interesting Itanium feature is the *speculative execution* and the NaT (“not a thing”) bit, somewhat resembling NaN numbers: [MSDN].
Chapter 94

8086 memory model

When dealing with 16-bit programs for MS-DOS or Win16 (78.3 on page 1101 or 53.5 on page 846), we can see that the pointers consist of two 16-bit values. What do they mean? Oh yes, that is another weird MS-DOS and 8086 artefact.

8086/8088 was a 16-bit CPU, but was able to address 20-bit address in RAM (thus being able to access 1MB of external memory). The external memory address space was divided between RAM (640KB max), ROM, windows for video memory, EMS cards, etc.

Let’s also recall that 8086/8088 was in fact an inheritor of the 8-bit 8080 CPU. The 8080 has a 16-bit memory space, i.e., it was able to address only 64KB. And probably because of old software porting reason\(^1\), 8086 can support many 64KB windows simultaneously, placed within the 1MB address space. This is some kind of a toy-level virtualization. All 8086 registers are 16-bit, so to address more, special segment registers (CS, DS, ES, SS) were introduced. Each 20-bit pointer is calculated using the values from a segment register and an address register pair (e.g. DS:BX) as follows:

\[
\text{real_address} = (\text{segment_register} \ll 4) + \text{address_register}
\]

For example, the graphics (EGA\(^2\), VGA\(^3\)) video RAM window on old IBM PC-compatibles has a size of 64KB. To access it, a value of 0xA000 has to be stored in one of the segment registers, e.g. into DS. Then DS:0 will address the first byte of video RAM and DS:0xFFFF — the last byte of RAM. The real address on the 20-bit address bus, however, will range from 0xA0000 to 0xAFFFF.

The program may contain hard-coded addresses like 0x1234, but the OS may need

---

\(^1\) The author is not 100% sure here
\(^2\) Enhanced Graphics Adapter
\(^3\) Video Graphics Array
to load the program at arbitrary addresses, so it recalculates the segment register values in a way that the program does not have to care where it’s placed in the RAM.

So, any pointer in the old MS-DOS environment in fact consisted of the segment address and the address inside segment, i.e., two 16-bit values. 20-bit was enough for that, though, but we needed to recalculate the addresses very often: passing more information on the stack seemed a better space/convenience balance.

By the way, because of all this it was not possible to allocate a memory block larger than 64KB.

The segment registers were reused at 80286 as selectors, serving a different function.

When the 80386 CPU and computers with bigger RAM were introduced, MS-DOS was still popular, so the DOS extenders emerged: these were in fact a step toward a “serious” OS, switching the CPU in protected mode and providing much better memory APIs for the programs which still needed to run under MS-DOS. Widely popular examples include DOS/4GW (the DOOM video game was compiled for it), Phar Lap, PMODE.

By the way, the same way of addressing memory was used in the 16-bit line of Windows 3.x, before Win32.
Chapter 95

Basic blocks reordering

95.1 Profile-guided optimization

This optimization method can move some basic blocks to another section of the executable binary file.

Obviously, there are parts of a function which are executed more frequently (e.g., loop bodies) and less often (e.g., error reporting code, exception handlers).

The compiler adds instrumentation code into the executable, then the developer runs it with a lot of tests to collect statistics. Then the compiler, with the help of the statistics gathered, prepares final the executable file with all infrequently executed code moved into another section.

As a result, all frequently executed function code is compacted, and that is very important for execution speed and cache usage.

An example from Oracle RDBMS code, which was compiled with Intel C++:

Listing 95.1: orageneric11.dll (win32)

```assembly
public _skgfsync
 proc near
 ; address 0x6030D86A

 db 66h
 nop
 push ebp
 mov ebp, esp
 mov ebp, esp
 mov edx, [ebp+0Ch]
 test edx, edx
```


jz short loc_6030D884
mov eax, [edx+30h]
test eax, 400h
jnz __VInfreq__skgfsync ; write to log
continue:
mov eax, [ebp+8]
mov edx, [ebp+10h]
mov dword ptr [eax], 0
lea eax, [edx+0Fh]
and eax, 0FFFFFFFCh
mov ecx, [eax]
cmp ecx, 45726963h
jnz error ; exit with error
mov esp, ebp
pop ebp
retn

__VInfreq__skgfsync:

mov eax, [edx]
test eax, eax
jz continue
mov ecx, [ebp+10h]
push ecx
mov ecx, [ebp+8]
push edx
push ecx
push offset ...

push dword ptr [edx+4]
call dword ptr [eax] ; write to log
add esp, 14h
jmp continue

error:

mov edx, [ebp+8]
mov dword ptr [edx], 69AAh ; 27050 "/rcurvearrowse

\( x, \text{iov}=0x%\text{x}\)\n"
push dword ptr [edx+4]
call dword ptr [eax] ; write to log
add esp, 14h
jmp continue
The distance of addresses between these two code fragments is almost 9 MB.

All infrequently executed code was placed at the end of the code section of the DLL file, among all function parts. This part of the function was marked by the Intel C++ compiler with the VInfreq prefix. Here we see that a part of the function that writes to a log file (presumably in case of error or warning or something like that) which was probably not executed very often when Oracle’s developers gathered statistics (if it was executed at all). The writing to log basic block eventually returns the control flow to the “hot” part of the function.

Another “infrequent” part is the basic block returning error code 27050.

In Linux ELF files, all infrequently executed code is moved by Intel C++ into the separate text.unlikely section, leaving all “hot” code in the text.hot section.

From a reverse engineer’s perspective, this information may help to split the function into its core and error handling parts.
Part XI

Books/blogs worth reading
Chapter 96

Books

96.1 Windows

[RA09].

96.2 C/C++

[ISO13].

96.3 x86 / x86-64

[Int13], [AMD13a]

96.4 ARM

ARM manuals: http://go.yurichev.com/17024

96.5 Cryptography

[Sch94]
Chapter 97

Blogs

97.1 Windows

- Microsoft: Raymond Chen
- nynaeve.net
Chapter 98

Other

There are two excellent RE-related subreddits on reddit.com: [reddit.com/r/ReverseEngineering](https://www.reddit.com/r/ReverseEngineering/) and [reddit.com/r/remath](https://www.reddit.com/r/remath) (on the topics for the intersection of RE and mathematics).

There is also a RE part of the Stack Exchange website: [reverseengineering.stackexchange.com](https://reverseengineering.stackexchange.com).

On IRC there's a ##re channel on FreeNode\(^1\).

---

\(^1\)freenode.net
Afterword
Chapter 99

Questions?

Do not hesitate to mail any questions to the author:<dennis(a)yurichev.com>

Do you have any suggestion on new content for the book?

Please, do not hesitate to send any corrections (including grammar (you see how horrible my English is?)), etc.

The author is working on the book a lot, so the page and listing numbers, etc. are changing very rapidly.

Please, do not refer to page and listing numbers in your emails to me.

There is a much simpler method: make a screenshot of the page, in a graphics editor underline the place where you see the error, and send it to me. He'll fix it much faster.

And if you familiar with git and \LaTeX you can fix the error right in the source code: GitHub.

Do not worry to bother me while writing me about any petty mistakes you found, even if you are not very confident. I'm writing for beginners, after all, so beginners' opinions and comments are crucial for my job.
Appendix
Appendix A

x86

A.1 Terminology

Common for 16-bit (8086/80286), 32-bit (80386, etc), 64-bit.

**byte** 8-bit. The DB assembly directive is used for defining variables and arrays of bytes. Bytes are passed in the 8-bit part of registers: AL/BL/CL/DL/AH/BH/CH/DH.

**word** 16-bit. DW assembly directive —”. Words are passed in the 16-bit part of the registers: AX/BX/CX/DX/SI/DI/R*W.

**double word** (“dword”) 32-bit. DD assembly directive —”. Double words are passed in registers (x86) or in the 32-bit part of registers (x64). In 16-bit code, double words are passed in 16-bit register pairs.

**quad word** (“qword”) 64-bit. DQ assembly directive —”. In 32-bit environment, quad words are passed in 32-bit register pairs.

**tbyte** (10 bytes) 80-bit or 10 bytes (used for IEEE 754 FPU registers).

**paragraph** (16 bytes)— term was popular in MS-DOS environment.

Data types of the same width (BYTE, WORD, DWORD) are also the same in Windows API.

A.2 General purpose registers

It is possible to access many registers by byte or 16-bit word parts. It is all inheritance from older Intel CPUs (up to the 8-bit 8080) still supported for backward
compatibility. Older 8-bit CPUs (8080) had 16-bit registers divided by two. Programs written for 8080 could access the low byte part of 16-bit registers, high byte part or the whole 16-bit register. Probably, this feature was left in 8086 as a helper for easier porting. This feature is usually not present in RISC CPUs.

Registers prefixed with R- appeared in x86-64, and those prefixed with E— in 80386. Thus, R-registers are 64-bit, and E-registers—32-bit.

8 more GPR’s were added in x86-86: R8-R15.

N.B.: In the Intel manuals the byte parts of these registers are prefixed by L, e.g.: R8L, but IDA names these registers by adding the B suffix, e.g.: R8B.

**A.2.1 RAX/EAX/AX/AL**

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX(^{x64})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AKA accumulator. The result of a function if usually returned via this register.

**A.2.2 RBX/EBX/BX/BL**

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBX(^{x64})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A.2.3 RCX/ECX/CX/CL**

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCX(^{x64})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AKA counter: in this role it is used in REP prefixed instructions and also in shift instructions (SHL/SHR/RxL/RxR).
### A.2.4 RDX/EDX/DX/DL

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RDX(^{x64})</td>
<td>EDX</td>
<td>DX</td>
<td>DH</td>
<td>DL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.2.5 RSI/ESI/SI/SIL

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RSI(^{x64})</td>
<td>ESI</td>
<td>SI</td>
<td>SIL(^{x64})</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AKA “source index”. Used as source in the instructions REP MOVsx, REP CMPSx.

### A.2.6 RDI/EDI/DI/DIL

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RDI(^{x64})</td>
<td>EDI</td>
<td>DI</td>
<td>DIL(^{x64})</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AKA “destination index”. Used as a pointer to the destination in the instructions REP MOVsx, REP STOSx.

### A.2.7 R8/R8D/R8W/R8L

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R8</td>
<td>R8D</td>
<td>R8W</td>
<td>R8L</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### A.2.8 R9/R9D/R9W/R9L

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th (byte number)</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>R9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.2.9 R10/R10D/R10W/R10L

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th (byte number)</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>R10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.2.10 R11/R11D/R11W/R11L

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th (byte number)</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>R11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.2.11 R12/R12D/R12W/R12L

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th (byte number)</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>R12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.2.12 R13/R13D/R13W/R13L

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th (byte number)</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>R13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### A.2.13 R14/R14D/R14W/R14L

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>R14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.2.14 R15/R15D/R15W/R15L

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>R15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.2.15 RSP/ESP/SP/SPL

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AKA stack pointer.** Usually points to the current stack except in those cases when it is not yet initialized.

### A.2.16 RBP/EBP/BP/BPL

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AKA frame pointer.** Usually used for local variables and accessing the arguments of the function. More about it: ([7.1.2 on page 92](#)).
A.2.17  RIP/EIP/IP

<table>
<thead>
<tr>
<th>7th (byte number)</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIPx64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AKA** “instruction pointer” \(^1\). Usually always points to the instruction to be executed right now. Cannot be modified, however, it is possible to do this (which is equivalent):

```
MOV EAX, ...
JMP EAX
```

Or:

```
PUSH value
RET
```

A.2.18  CS/DS/ES/SS/FS/GS

16-bit registers containing code selector (CS), data selector (DS), stack selector (SS).

FS in win32 points to TLS, GS took this role in Linux. It is done for faster access to the TLS and other structures like the TIB.

In the past, these registers were used as segment registers (94 on page 1278).

A.2.19  Flags register

**AKA** EFLAGS.

\(^1\)Sometimes also called “program counter”
### APPENDIX A. X86

#### Bit (mask) | Abbreviation (meaning) | Description
--- | --- | ---
0 (1) | CF (Carry) | The CLC/STC/CMC instructions are used for setting/resetting/toggling this flag.
2 (4) | PF (Parity) | (17.7.1 on page 320).
4 (0x10) | AF (Adjust) | Setting to 0 if the last operation’s result was zero.
6 (0x40) | ZF (Zero) | Setting to 0 if the last operation’s result was zero.
7 (0x80) | SF (Sign) | Used for debugging. If turned on, an exception is to be generated after each instruction.
8 (0x100) | TF (Trap) | Used for debugging. If turned on, an exception is to be generated after each instruction.
9 (0x200) | IF (Interrupt enable) | Are interrupts enabled. The CLI/STI instructions are used for setting/resetting the flag.
10 (0x400) | DF (Direction) | A direction is set for the REP MOVSx, REP CMPSx, REP LODSx, REP SCASx instructions.
11 (0x800) | OF (Overflow) | The CLD/STD instructions are used for setting/resetting the flag.
12, 13 (0x3000) | IOPL (I/O privilege level) | 80286
14 (0x4000) | NT (Nested task) | 80286
16 (0x10000) | RF (Resume) | 80386
17 (0x20000) | VM (Virtual 8086 mode) | 80386
18 (0x40000) | AC (Alignment check) | 80486
19 (0x80000) | VIF (Virtual interrupt) | Pentium
20 (0x100000) | VIP (Virtual interrupt pending) | Pentium
21 (0x200000) | ID (Identification) | Pentium

All the rest flags are reserved.

### A.3 FPU registers

8 80-bit registers working as a stack: ST(0)-ST(7). N.B.: IDA calls ST(0) as just ST. Numbers are stored in the IEEE 754 format.

`long double` value format:
## A.3.1 Control Word

Register controlling the behaviour of the FPU.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Abbreviation (meaning)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IM (Invalid operation Mask)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>DM (Denormalized operand Mask)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ZM (Zero divide Mask)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>OM (Overflow Mask)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>UM (Underflow Mask)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PM (Precision Mask)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IEM (Interrupt Enable Mask)</td>
<td>Exceptions enabling, 1 by default (disabled)</td>
</tr>
<tr>
<td>8, 9</td>
<td>PC (Precision Control)</td>
<td>00 — 24 bits (REAL4) 10 — 53 bits (REAL8) 11 — 64 bits (REAL10)</td>
</tr>
<tr>
<td>10, 11</td>
<td>RC (Rounding Control)</td>
<td>00 — (by default) round to nearest 01 — round toward $-\infty$ 10 — round toward $+\infty$ 11 — round toward 0</td>
</tr>
<tr>
<td>12</td>
<td>IC (Infinity Control)</td>
<td>0 — (by default) treat $+\infty$ and $-\infty$ as unsigned 1 — respect both $+\infty$ and $-\infty$</td>
</tr>
</tbody>
</table>

The PM, UM, OM, ZM, DM, IM flags define if to generate exception in the case of a corresponding error.

## A.3.2 Status Word

Read-only register.
## APPENDIX A. X86

### Bit Abbreviation (meaning) Description

<table>
<thead>
<tr>
<th>Bit</th>
<th>Abbreviation (meaning)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>B (Busy)</td>
<td>Is FPU do something (1) or results are ready (0)</td>
</tr>
<tr>
<td>14</td>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>13, 12, 11</td>
<td>TOP</td>
<td>points to the currently zeroth register</td>
</tr>
<tr>
<td>10</td>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>C0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IR (Interrupt Request)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>SF (Stack Fault)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>P (Precision)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>U (Underflow)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>O (Overflow)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Z (Zero)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>D (Denormalized)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>I (Invalid operation)</td>
<td></td>
</tr>
</tbody>
</table>

The SF, P, U, O, Z, D, I bits signal about exceptions.

About the C3, C2, C1, C0 you can read more here: (17.7.1 on page 319).

N.B.: When ST(x) is used, the FPU adds \( x \) to TOP (by modulo 8) and that is how it gets the internal register's number.

### A.3.3 Tag Word

The register has current information about the usage of numbers registers.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Abbreviation (meaning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15, 14</td>
<td>Tag(7)</td>
</tr>
<tr>
<td>13, 12</td>
<td>Tag(6)</td>
</tr>
<tr>
<td>11, 10</td>
<td>Tag(5)</td>
</tr>
<tr>
<td>9, 8</td>
<td>Tag(4)</td>
</tr>
<tr>
<td>7, 6</td>
<td>Tag(3)</td>
</tr>
<tr>
<td>5, 4</td>
<td>Tag(2)</td>
</tr>
<tr>
<td>3, 2</td>
<td>Tag(1)</td>
</tr>
<tr>
<td>1, 0</td>
<td>Tag(0)</td>
</tr>
</tbody>
</table>

Each tag contains information about a physical FPU register (R(x)), not logical (ST(x)).

For each tag:
- **00** – The register contains a non-zero value
- **01** – The register contains 0
• 10 — The register contains a special value (NAN\(^2\), \(\infty\), or denormal)
• 11 — The register is empty

A.4 SIMD registers

A.4.1 MMX registers

8 64-bit registers: MM0..MM7.

A.4.2 SSE and AVX registers

SSE: 8 128-bit registers: XMM0..XMM7. In the x86-64 8 more registers were added: XMM8..XMM15.

AVX is the extension of all these registers to 256 bits.

A.5 Debugging registers

Used for hardware breakpoints control.

• DR0 — address of breakpoint #1
• DR1 — address of breakpoint #2
• DR2 — address of breakpoint #3
• DR3 — address of breakpoint #4
• DR6 — a cause of break is reflected here
• DR7 — breakpoint types are set here

\(^2\)Not a Number
### A.5.1 DR6

<table>
<thead>
<tr>
<th>Bit (mask)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>B0 — breakpoint #1 was triggered</td>
</tr>
<tr>
<td>1 (2)</td>
<td>B1 — breakpoint #2 was triggered</td>
</tr>
<tr>
<td>2 (4)</td>
<td>B2 — breakpoint #3 was triggered</td>
</tr>
<tr>
<td>3 (8)</td>
<td>B3 — breakpoint #4 was triggered</td>
</tr>
<tr>
<td>13 (0x2000)</td>
<td>BD — modification attempt of one of the DRx registers. may be raised if GD is enabled</td>
</tr>
<tr>
<td>14 (0x4000)</td>
<td>BS — single step breakpoint (TF flag was set in EFLAGS). Highest priority. Other bits may also be set.</td>
</tr>
<tr>
<td>15 (0x8000)</td>
<td>BT (task switch flag)</td>
</tr>
</tbody>
</table>

N.B. A single step breakpoint is a breakpoint which occurs after each instruction. It can be enabled by setting TF in EFLAGS (A.2.19 on page 1295).

### A.5.2 DR7

Breakpoint types are set here.

<table>
<thead>
<tr>
<th>Bit (mask)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>L0 — enable breakpoint #1 for the current task</td>
</tr>
<tr>
<td>1 (2)</td>
<td>G0 — enable breakpoint #1 for all tasks</td>
</tr>
<tr>
<td>2 (4)</td>
<td>L1 — enable breakpoint #2 for the current task</td>
</tr>
<tr>
<td>3 (8)</td>
<td>G1 — enable breakpoint #2 for all tasks</td>
</tr>
<tr>
<td>4 (0x10)</td>
<td>L2 — enable breakpoint #3 for the current task</td>
</tr>
<tr>
<td>5 (0x20)</td>
<td>G2 — enable breakpoint #3 for all tasks</td>
</tr>
<tr>
<td>6 (0x40)</td>
<td>L3 — enable breakpoint #4 for the current task</td>
</tr>
<tr>
<td>7 (0x80)</td>
<td>G3 — enable breakpoint #4 for all tasks</td>
</tr>
<tr>
<td>8 (0x100)</td>
<td>LE — not supported since P6</td>
</tr>
<tr>
<td>9 (0x200)</td>
<td>GE — not supported since P6</td>
</tr>
<tr>
<td>13 (0x2000)</td>
<td>GD — exception is to be raised if any MOV instruction tries to modify one of the DRx registers</td>
</tr>
<tr>
<td>16,17 (0x30000)</td>
<td>breakpoint #1: R/W — type</td>
</tr>
<tr>
<td>18,19 (0xC0000)</td>
<td>breakpoint #1: LEN — length</td>
</tr>
<tr>
<td>20,21 (0x300000)</td>
<td>breakpoint #2: R/W — type</td>
</tr>
<tr>
<td>22,23 (0xC00000)</td>
<td>breakpoint #2: LEN — length</td>
</tr>
<tr>
<td>24,25 (0x3000000)</td>
<td>breakpoint #3: R/W — type</td>
</tr>
<tr>
<td>26,27 (0xC000000)</td>
<td>breakpoint #3: LEN — length</td>
</tr>
<tr>
<td>28,29 (0x30000000)</td>
<td>breakpoint #4: R/W — type</td>
</tr>
<tr>
<td>30,31 (0xC0000000)</td>
<td>breakpoint #4: LEN — length</td>
</tr>
</tbody>
</table>

The breakpoint type is to be set as follows (R/W):
• 00 — instruction execution
• 01 — data writes
• 10 — I/O reads or writes (not available in user-mode)
• 11 — on data reads or writes

N.B.: breakpoint type for data reads is absent, indeed.

Breakpoint length is to be set as follows (LEN):
• 00 — one-byte
• 01 — two-byte
• 10 — undefined for 32-bit mode, eight-byte in 64-bit mode
• 11 — four-byte

A.6 Instructions

Instructions marked as (M) are not usually generated by the compiler: if you see one of them, it was probably a hand-written piece of assembly code, or is a compiler intrinsic (90 on page 1261).

Only the most frequently used instructions are listed here. You can read [Int13] or [AMD13a] for a full documentation.

Instruction’s opcodes has to be memorized? No, only those which are used for code patching (89.2 on page 1259). All the rest of the opcodes don’t need to be memorized.

A.6.1 Prefixes

**LOCK** forces CPU to make exclusive access to the RAM in multiprocessor environment. For the sake of simplification, it can be said that when an instruction with this prefix is executed, all other CPUs in a multiprocessor system are stopped. Most often it is used for critical sections, semaphores, mutexes. Commonly used with ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. You can read more about critical sections here (68.4 on page 1033).

**REP** is used with the MOVsx and STOSx instructions: execute the instruction in a loop, the counter is located in the CX/ECX/RCX register. For a detailed description, read more about the MOVsx (A.6.2 on page 1305) and STOSx (A.6.2 on page 1308) instructions.
The instructions prefixed by REP are sensitive to the DF flag, which is used to set the direction.

REPE/REPNE (AKA REPZ/REPNZ) used with CMPSx and SCASx instructions: execute the last instruction in a loop, the count is set in the CX/ECX/RCX register. It terminates prematurely if ZF is 0 (REPE) or if ZF is 1 (REPNE).

For a detailed description, you can read more about the CMPSx (A.6.3 on page 1310) and SCASx (A.6.2 on page 1307) instructions.

Instructions prefixed by REPE/REPNE are sensitive to the DF flag, which is used to set the direction.

A.6.2 Most frequently used instructions

These can be memorized in the first place.

ADC (add with carry) add values, increment the result if the CF flag is set. ADC is often used for the addition of large values, for example, to add two 64-bit values in a 32-bit environment using two ADD and ADC instructions. For example:

```
; work with 64-bit values: add val1 to val2.
; .lo mean lowest 32 bits, .hi means highest.
ADD val1.lo, val2.lo
ADC val1.hi, val2.hi ; use CF set or cleared at the previous instruction
```

One more example: 24 on page 543.

ADD add two values
AND logical “and”
CALL call another function: PUSH address_after_CALL_instruction; JMP label
CMP compare values and set flags, the same as SUB but without writing the result

DEC decrement. The CF flag is not modified.
IMUL signed multiply
INC increment. The CF flag is not modified.
Jxz, Jecz, Jrcxz (M) jump if CX/ECX/RCX=0
JMP jump to another address. The opcode has a jump offset.
APPENDIX A. X86

Jcc (where cc—condition code)

A lot of these instructions have synonyms (denoted with AKA), this was done for convenience. Synonymous instructions are translated into the same opcode. The opcode has a jump offset.

**JAE** AKA JNC: jump if above or equal (unsigned): CF=0

**JA** AKA JNBE: jump if greater (unsigned): CF=0 and ZF=0

**JBE** jump if lesser or equal (unsigned): CF=1 or ZF=1

**JB** AKA JC: jump if below (unsigned): CF=1

**JC** AKA JB: jump if CF=1

**JE** AKA JZ: jump if equal or zero: ZF=1

**JGE** jump if greater or equal (signed): SF=OF

**JG** jump if greater (signed): ZF=0 and SF=OF

**JLE** jump if lesser or equal (signed): ZF=1 or SF ≠ OF

**JL** jump if lesser (signed): SF ≠ OF

**JNAE** AKA JC: jump if not above or equal (unsigned) CF=1

**JNA** jump if not above (unsigned) CF=1 and ZF=1

**JNBE** jump if not below or equal (unsigned): CF=0 and ZF=0

**JNB** AKA JNC: jump if not below (unsigned): CF=0

**JNC** AKA JAE: jump CF=0 synonymous to JNB.

**JNE** AKA JNZ: jump if not equal or not zero: ZF=0

**JNGE** jump if not greater or equal (signed): SF ≠ OF

**JNG** jump if not greater (signed): ZF=1 or SF ≠ OF

**JNLE** jump if not lesser (signed): ZF=0 and SF=OF

**JNL** jump if not lesser (signed): SF=OF

**JNO** jump if not overflow: OF=0

**JNS** jump if SF flag is cleared

**JNZ** AKA JNE: jump if not equal or not zero: ZF=0

**JO** jump if overflow: OF=1

**JPO** jump if PF flag is cleared (Jump Parity Odd)

**JP** AKA JPE: jump if PF flag is set

**JS** jump if SF flag is set
APPENDIX A. X86

JZ  AKA JE: jump if equal or zero: ZF=1

LAHF  copy some flag bits to AH:

\[
\begin{array}{cccccc}
S & F & Z & F & P & F \\
7 & 6 & 4 & 2 & 0 \\
\end{array}
\]

LEAVE  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.

LEA  (Load Effective Address) form an address

This instruction was intended not for summing values and multiplication but for forming an address, e.g., for calculating the address of an array element by adding the array address, element index, with multiplication of element size\(^3\).

So, the difference between MOV and LEA is that MOV forms a memory address and loads a value from memory or stores it there, but LEA just forms an address.

But nevertheless, it is can be used for any other calculations.

LEA is convenient because the computations performed by it does not alter CPU flags. This may be very important for OOE processors (to create less data dependencies).

```c
int f(int a, int b)
{
    return a*8+b;
}
```

Listing A.1: Optimizing MSVC 2010

\[
\begin{array}{l}
_a$ = 8 \\
\_b$ = 12 \\
_f PROC
\begin{array}{l}
\text{mov eax, DWORD PTR } _b$[esp-4] \\
\text{mov ecx, DWORD PTR } _a$[esp-4] \\
\text{lea eax, DWORD PTR } [eax+ecx*8] \\
\text{ret 0}
\end{array}
\_f ENDP
\end{array}
\]

Intel C++ uses LEA even more:

\(^3\)See also: wikipedia
int f1(int a) {
    return a*13;
};

Listing A.2: Intel C++ 2011

_f1 PROC NEAR
    mov ecx, DWORD PTR [4+esp] ; ecx = a
    lea edx, DWORD PTR [ecx+ecx*8] ; edx = a*9
    lea eax, DWORD PTR [edx+ecx*4] ; eax = a*9
    jmp + a*4 = a*13
    ret

These two instructions performs faster than one IMUL.

**MOVSB/MOVSW/MOVSD/MOVSDQ** copy byte/16-bit word/32-bit word/64-bit word from the address which is in SI/ESI/RSI into the address which is in DI/EDI/RDI.

Together with the REP prefix, it is to be repeated in a loop, the count is to be stored in the CX/ECX/RCX register: it works like memcpy() in C. If the block size is known to the compiler in the compile stage, memcpy() is often inlined into a short code fragment using REP MOVSDx, sometimes even as several instructions.

The memcpy(EDI, ESI, 15) equivalent is:

```
; copy 15 bytes from ESI to EDI
CLD ; set direction to "forward"
MOV ECX, 3 ; set count to 15
REP MOVSD ; copy 12 bytes
MOVSW ; copy 2 more bytes
MOVSB ; copy remaining byte
```

(Supposedly, it works faster than copying 15 bytes using just one REP MOVSB).

**MOVEX** load with sign extension see also: (15.1.1 on page 274)

**MOVZX** load and clear all other bits see also: (15.1.1 on page 276)

**MOV** load value. this instruction name is misnomer, resulting in some confusion (data is not moved but copied), in other architectures the same instructions is usually named “LOAD” and/or “STORE” or something like that.

One important thing: if you set the low 16-bit part of a 32-bit register in 32-bit mode, the high 16 bits remains as they were. But if you modify the low 32-bit part of the register in 64-bit mode, the high 32 bits of the register will be cleared.

Supposedly, it was done to simplify porting code to x86-64.
MUL  unsigned multiply

NEG  negation: \( op = -op \)

NOP  NOP. Its opcode is 0x90, it is in fact the XCHG EAX, EAX idle instruction. This implies that x86 does not have a dedicated NOP instruction (as in many RISC). This book has at least one listing where GDB shows NOP as 16-bit XCHG instruction: 6.1.1 on page 62.

More examples of such operations: ( 88 on page 1256).

NOP may be generated by the compiler for aligning labels on a 16-byte boundary. Another very popular usage of NOP is to replace manually (patch) some instruction like a conditional jump to NOP in order to disable its execution.

NOT op1: \( op1 = \neg op1 \). logical inversion Feature — the instruction doesn’t change flags.

OR  logical “or”

POP get a value from the stack: value=SS:[ESP]; ESP=ESP+4 (or 8)

PUSH push a value into the stack: ESP=ESP-4 (or 8); SS:[ESP]=value

RET return from subroutine: POP tmp; JMP tmp.

In fact, RET is an assembly language macro, in Windows and *NIX environment it is translated into RETN (“return near”) or, in MS-DOS times, where the memory was addressed differently ( 94 on page 1278), into RETF (“return far”).

RET can have an operand. Then it works like this: POP tmp; ADD ESP op1; JMP tmp. RET with an operand usually ends functions in the stdcall calling convention, see also: 64.2 on page 952.

SAHF copy bits from AH to CPU flags:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>4</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>ZF</td>
<td>AF</td>
<td>PF</td>
<td>CF</td>
</tr>
</tbody>
</table>

SBB (subtraction with borrow) subtract values, decrement the result if the CF flag is set. SBB is often used for subtraction of large values, for example, to subtract two 64-bit values in 32-bit environment using two SUB and SBB instructions. For example:

```plaintext
; work with 64-bit values: subtract val2 from val1.
; .lo mean lowest 32 bits, .hi means highest.
SUB val1.lo, val2.lo
SBB val1.hi, val2.hi ; use CF set or cleared at the previous instruction
```
One more example: 24 on page 543.

**SCASB/SCASW/SCASD/SCASQ** (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word that’s stored in AX/EAX/RAX with a variable whose address is in DI/EDI/RDI. Set flags as CMP does.

This instruction is often used with the REPNE prefix: continue to scan the buffer until a special value stored in AX/EAX/RAX is found. Hence “NE” in REPNE: continue to scan while the compared values are not equal and stop when equal.

It is often used like the strlen() C standard function, to determine an ASCIIZ string’s length:

Example:

```assembly
lea   edi, string
mov   ecx, 0FFFFFFFFh ; scan $2^{32} - 1$ bytes, i.e., almost "infinitely"
xor   eax, eax ; 0 is the terminator
repne scasb
add   edi, 0FFFFFFFFh ; correct it

; now EDI points to the last character of the ASCIIZ string.

; lets determine string length'
; current ECX = -1-strlen
not   ecx
dec   ecx

; now ECX contain string length
```

If we use a different AX/EAX/RAX value, the function acts like the memchr() standard C function, i.e., it finds a specific byte.

**SHL** shift value left

**SHR** shift value right:
These instructions are frequently used for multiplication and division by $2^n$. Another very frequent application is processing bit fields: 19 on page 411.

**SHRD** $\text{op1, op2, op3}$: shift value in op2 right by op3 bits, taking bits from op1.

Example: 24 on page 543.

**STOSB/STOSW/STOSD/STOSQ** store byte/ 16-bit word/ 32-bit word/ 64-bit word from AX/EAX/RAX into the address which is in DI/EDI/RDI.

Together with the REP prefix, it is to be repeated in a loop, the counter is in the CX/ECX/RCX register: it works like memset() in C. If the block size is known to the compiler on compile stage, memset() is often inlined into a short code fragment using REP MOV$\times x$, sometimes even as several instructions.

memset(EDI, 0xAA, 15) equivalent is:

```
; store 15 0xAA bytes to EDI
CLD ; set direction to "forward"
MOV EAX, 0AAAAAAAAh
MOV ECX, 3
REP STOSD ; write 12 bytes
STOSW ; write 2 more bytes
STOSB ; write remaining byte
```

(Supposedly, it works faster than storing 15 bytes using just one REP STOSB).

**SUB** subtract values. A frequently occurring pattern is SUB $\text{reg, reg}$, which implies zeroing of reg.

**TEST** same as AND but without saving the result, see also: 19 on page 411

**XCHG** exchange the values in the operands

**XOR** $\text{op1, op2}$: XOR\footnote{\textit{eXclusive OR}} values. $\text{op1} = \text{op1} \oplus \text{op2}$. A frequently occurring pattern is XOR $\text{reg, reg}$, which implies zeroing of reg.

XOR is widely used when one needs just to flip specific bit(s).

Indeed, the XOR operation applied with 1 effectively inverts a bit:
And vice-versa, the XOR operation applied with 0 does nothing, i.e., it's an idle operation.

This is a very important property of the XOR operation and it's highly recommended to memorize it.

### A.6.3 Less frequently used instructions

- **BSF**  *bit scan forward*, see also: 25.2 on page 578
- **BSR**  *bit scan reverse*
- **BSWAP**  *(byte swap)*, change value *endianness*.
- **BTC**  *bit test and complement*
- **BTR**  *bit test and reset*
- **BTS**  *bit test and set*
- **BT**  *bit test*

**CBW/CWD/CWDE/CDQ/CDQE**  *Sign-extend value:*
  - **CBW**  convert byte in AL to word in AX
  - **CWD**  convert word in AX to doubleword in DX:AX
  - **CWDE**  convert word in AX to doubleword in EAX
  - **CDQ**  convert doubleword in EAX to quadword in EDX:EAX
  - **CDQE**  *(x64)* convert doubleword in EAX to quadword in RAX

These instructions consider the value's sign, extending it to high part of the newly constructed value. See also: 24.5 on page 557.

Interestingly to know these instructions was initially named as *SEX* *(Sign Extend)*, as Stephen P. Morse (one of Intel 8086 CPU designers) wrote in [Mor80]:

> The process of stretching numbers by extending the sign bit is called sign extension. The 8086 provides instructions (Fig. 3.29) to facilitate the task of sign extension. These instructions were initially named SEX (sign extend) but were
later renamed to the more conservative CBW (convert byte to word) and CWD (convert word to double word).

CLD  clear DF flag.
CLI  (M) clear IF flag
CMC  (M) toggle CF flag
CMOVcc  conditional MOV: load if the condition is true. The condition codes are the same as in the Jcc instructions (A.6.2 on page 1303).

CMPSB/CMPSW/CMPSD/CMPSQ  (M) compare byte/16-bit word/32-bit word/64-bit word from the address which is in SI/ESI/RSI with the variable at the address stored in DI/EDI/RDI. Set flags as CMP does.

Together with the REP prefix, it is to be repeated in a loop, the counter is stored in the CX/ECX/RCX register, the process will run until the ZF flag is zero (e.g., until the compared values are equal to each other, hence “E” in REPE).

It works like memcmp() in C.

Example from the Windows NT kernel (WRK v1.2):

Listing A.3: base\ntos\rtl\i386\movemem.asm

; ULONG
; RtlCompareMemory (  
;  IN PVOID Source1,  
;  IN PVOID Source2,  
;  IN ULONG Length  
;  )
;
; Routine Description:
;
;   This function compares two blocks of memory and returns the number of bytes that compared equal.
;
; Arguments:
;
;   Source1 (esp+4) - Supplies a pointer to the first block of memory to compare.
;
;   Source2 (esp+8) - Supplies a pointer to the second block of memory to compare.
APPENDIX A. X86

--

RcmSource1 equ [esp+12]
RcmSource2 equ [esp+16]
RcmLength equ [esp+20]

CODE_ALIGNMENT

; Length (esp+12) - Supplies the Length, in bytes, of the memory to be compared.

; Return Value:

; The number of bytes that compared equal is returned as the function value. If all bytes compared equal, then the length of the original block of memory is returned.

push esi ; save registers
push edi ;
cld ; clear direction
mov esi,RcmSource1 ; (esi) -> first block to compare
mov edi,RcmSource2 ; (edi) -> second block to compare

; Compare dwords, if any.

rcm10: mov ecx,RcmLength ; (ecx) = length in bytes
        shr ecx,2 ; (ecx) = length in dwords
        jz rcm20 ; no dwords, try bytes
        repe cmpsd ; compare dwords
        jnz rcm40 ; mismatch, go find byte

; Compare residual bytes, if any.

rcm20: mov ecx, RcmLength ; (ecx) = length
    in bytes
    and ecx, 3 ; (ecx) = length
    mod 4
    jz rcm30 ; 0 odd bytes, go
    do dwords
        repe cmpsb ; compare odd
        bytes
        jnz rcm50 ; mismatch, go
    report how far we got

; All bytes in the block match.

rcm30: mov eax, RcmLength ; set number of
    matching bytes
    pop edi ; restore
    registers
    pop esi ;
    stdRET _RtlCompareMemory

; When we come to rcm40, esi (and edi) points to the
; dword after the
; one which caused the mismatch. Back up 1 dword and
; find the byte.
; Since we know the dword didn't match, we can assume
; one byte won't.

rcm40: sub esi, 4 ; back up
    sub edi, 4 ; back up
    mov ecx, 5 ; ensure that ecx
    doesn't count out
        repe cmpsb ; find mismatch
    byte

; When we come to rcm50, esi points to the byte after
; the one that
; did not match, which is TWO after the last byte that
; did match.

rcm50: dec esi ; back up
       sub esi,RcmSource1 ; compute bytes ✓
       ▼ that matched
       mov   eax,esi    ;
       pop    edi ; restore ✓
       ▼ registers
       pop    esi    ;
       stdRET _RtlCompareMemory

stdENDP _RtlCompareMemory

N.B.: this function uses a 32-bit word comparison (CMPSD) if the block size
is a multiple of 4, or per-byte comparison (CMPSB) otherwise.

CPUID get information about the CPU's features. see also: (21.6.1 on page 503).

DIV unsigned division

IDIV signed division

INT (M): INT x is analogous to PUSHF; CALL dword ptr [x*4] in 16-bit
environment. It was widely used in MS-DOS, functioning as a syscall vector.
The registers AX/BX/CX/DX/SI/DI were filled with the arguments and then
the flow jumped to the address in the Interrupt Vector Table (located at the
beginning of the address space). It was popular because INT has a short
opcode (2 bytes) and the program which needs some MS-DOS services is not
bother to determine the address of the service's entry point. The interrupt
handler returns the control flow to caller using the IRET instruction.

The most busy MS-DOS interrupt number was 0x21, serving a huge part of
its API. See also: [Bro] for the most comprehensive interrupt lists and other
MS-DOS information.

In the post-MS-DOS era, this instruction was still used as syscall both in Linux
and Windows (66 on page 973), but was later replaced by the SYSENTER or
SYSCALL instructions.

INT 3 (M): this instruction is somewhat close to INT, it has its own 1-byte opcode
(0xCC), and is actively used while debugging. Often, the debuggers just write
the 0xCC byte at the address of the breakpoint to be set, and when an ex-
ception is raised, the original byte is restored and the original instruction at
this address is re-executed.

As of Windows NT, an EXCEPTION_BREAKPOINT exception is to be raised
when the CPU executes this instruction. This debugging event may be inter-
cepted and handled by a host debugger, if one is loaded. If it is not loaded,
Windows offers to run one of the registered system debuggers. If MSVS\(^5\) is
installed, its debugger may be loaded and connected to the process. In order

\(^5\)Microsoft Visual Studio
to protect from reverse engineering, a lot of anti-debugging methods check integrity of the loaded code.

MSVC has compiler intrinsic for the instruction: \texttt{\_\_debugbreak( )}\textsuperscript{6}.

There is also a win32 function in kernel32.dll named \texttt{DebugBreak( )}\textsuperscript{7}, which also executes INT 3.

\textbf{IN} (M) input data from port. The instruction usually can be seen in OS drivers or in old MS-DOS code, for example (\textit{78.3 on page 1101}).

\textbf{IRET} was used in the MS-DOS environment for returning from an interrupt handler after it was called by the INT instruction. Equivalent to POP tmp; POPF; JMP tmp.

\textbf{LOOP} (M) decrement CX/ECX/RCX, jump if it is still not zero.

\textbf{OUT} (M) output data to port. The instruction usually can be seen in OS drivers or in old MS-DOS code, for example (\textit{78.3 on page 1101}).

\textbf{POPA} (M) restores values of (R|E)DI, (R|E)SI, (R|E)BP, (R|E)BX, (R|E)DX, (R|E)CX, (R|E)AX registers from the stack.

\textbf{POPCNT} population count. Counts the number of 1 bits in the value. AKA “hamming weight”. AKA “NSA instruction” due to some rumors:

This branch of cryptography is fast-paced and very politically charged. Most designs are secret; a majority of military encryptions systems in use today are based on LFSRs. In fact, most Cray computers (Cray 1, Cray X-MP, Cray Y-MP) have a rather curious instruction generally known as “population count.” It counts the 1 bits in a register and can be used both to efficiently calculate the Hamming distance between two binary words and to implement a vectorized version of a LFSR. I’ve heard this called the canonical NSA instruction, demanded by almost all computer contracts.

\textbf{POPF} restore flags from the stack (AKA EFLAGS register)

\textbf{PUSHA} (M) pushes the values of the (R|E)AX, (R|E)CX, (R|E)DX, (R|E)BX, (R|E)BP, (R|E)SI, (R|E)DI registers to the stack.

\textbf{PUSHF} push flags (AKA EFLAGS register)

\textbf{RCL} (M) rotate left via CF flag:

\textsuperscript{6}MSDN
\textsuperscript{7}MSDN
**RCR** (M) rotate right via CF flag:

Despite the fact that almost all CPUs have these instructions, there are no corresponding operations in C/C++, so the compilers of these PLs usually do not generate these instructions.

For the programmer's convenience, at least **MSVC** has the pseudofunctions (compiler intrinsics) `_rotl()` and `_rotr`\(^8\), which are translated by the compiler directly to these instructions.

**SAL** Arithmetic shift left, synonymous to SHL

**SAR** Arithmetic shift right

\(^8\text{MSDN}\)
Hence, the sign bit always stays at the place of the MSB.

**SETcc** op: load 1 to operand (byte only) if the condition is true or zero otherwise. The condition codes are the same as in the Jcc instructions (A.6.2 on page 1303).

**STC** (M) set CF flag

**STD** (M) set DF flag. This instruction is not generated by compilers and generally rare. For example, it can be found in the ntoskrnl.exe Windows kernel file, in the hand-written memory copy routines.

**STI** (M) set IF flag

**SYSCALL** (AMD) call syscall (66 on page 973)

**SYSENTER** (Intel) call syscall (66 on page 973)

**UD2** (M) undefined instruction, raises exception. Used for testing.

### A.6.4 FPU instructions

-R in the mnemonic usually implies that the operands are reversed, -P implies that one element is popped from the stack after the instruction's execution, -PP implies that two elements are popped.

-P instructions are often useful when we do not need the value in the FPU stack to be present anymore after the operation.

**FABS** replace value in ST(0) by absolute value in ST(0)

**FADD** op: ST(0)=op+ST(0)

**FADD** ST(0), ST(i): ST(0)=ST(0)+ST(i)

**FADDP** ST(1)=ST(0)+ST(1); pop one element from the stack, i.e., the values in the stack are replaced by their sum

**FCHS** ST(0)=-ST(0)

**FCOM** compare ST(0) with ST(1)

**FCOM** op: compare ST(0) with op

**FCOMP** compare ST(0) with ST(1); pop one element from the stack

**FCOMPP** compare ST(0) with ST(1); pop two elements from the stack
FDIVR  op: ST(0)=op/ST(0)
FDIVR  ST(i), ST(j): ST(i)=ST(j)/ST(i)
FDIVRP op: ST(0)=op/ST(0); pop one element from the stack
FDIVRP ST(i), ST(j): ST(i)=ST(j)/ST(i); pop one element from the stack
FDIV  op: ST(0)=ST(0)/op
FDIV  ST(i), ST(j): ST(i)=ST(i)/ST(j)
FDIVP ST(1)=ST(0)/ST(1); pop one element from the stack, i.e., the dividend and divisor values in the stack are replaced by quotient
FILD  op: convert integer and push it to the stack.
FIST  op: convert ST(0) to integer op
FISTP op: convert ST(0) to integer op; pop one element from the stack
FLD1 push 1 to stack
FLDCW op: load FPU control word (A.3 on page 1297) from 16-bit op.
FLDZ push zero to stack
FLD  op: push op to the stack.
FMUL  op: ST(0)=ST(0)*op
FMUL  ST(i), ST(j): ST(i)=ST(i)*ST(j)
FMULP op: ST(0)=ST(0)*op; pop one element from the stack
FMULP ST(i), ST(j): ST(i)=ST(i)*ST(j); pop one element from the stack
FSINCOS : tmp=ST(0); ST(1)=sin(tmp); ST(0)=cos(tmp)
FSQRT : ST(0) = \sqrt{ST(0)}
FSTCW op: store FPU control word (A.3 on page 1297) into 16-bit op after checking for pending exceptions.
FNSTCW op: store FPU control word (A.3 on page 1297) into 16-bit op.
FSTSW op: store FPU status word (A.3.2 on page 1297) into 16-bit op after checking for pending exceptions.
FNSTSW op: store FPU status word (A.3.2 on page 1297) into 16-bit op.
FST  op: copy ST(0) to op
FSTP op: copy ST(0) to op; pop one element from the stack
FSUBR op: ST(0)=op-ST(0)
FSUBR ST(0), ST(i): ST(0)=ST(i)-ST(0)
FSUBRP \(\text{ST}(1) = \text{ST}(0) - \text{ST}(1)\); pop one element from the stack, i.e., the value in the stack is replaced by the difference

FSUB op: \(\text{ST}(0) = \text{ST}(0) - \text{op}\)

FSUB \(\text{ST}(0), \text{ST}(i): \text{ST}(0) = \text{ST}(0) - \text{ST}(i)\)

FSUBP \(\text{ST}(1) = \text{ST}(1) - \text{ST}(0); \) pop one element from the stack, i.e., the value in the stack is replaced by the difference

FUCOM \(\text{ST}(i): \) compare \(\text{ST}(0)\) and \(\text{ST}(i)\)

FUCOM compare \(\text{ST}(0)\) and \(\text{ST}(1)\)

FUCOMP compare \(\text{ST}(0)\) and \(\text{ST}(1)\); pop one element from stack.

FUCOMPP compare \(\text{ST}(0)\) and \(\text{ST}(1)\); pop two elements from stack.

The instructions perform just like FCOM, but an exception is raised only if one of the operands is SNaN, while QNaN numbers are processed smoothly.

FXCH \(\text{ST}(i): \) exchange values in \(\text{ST}(0)\) and \(\text{ST}(i)\)

FXCH exchange values in \(\text{ST}(0)\) and \(\text{ST}(1)\)

### A.6.5 Instructions having printable ASCII opcode

(In 32-bit mode).

These can be suitable for shellcode construction. See also: 82.1 on page 1198.

<table>
<thead>
<tr>
<th>ASCII character</th>
<th>hexadecimal code</th>
<th>x86 instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
<td>XOR</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>XOR</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>XOR</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>XOR</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>XOR</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>XOR</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
<td>AAA</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td>CMP</td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td>CMP</td>
</tr>
<tr>
<td>:</td>
<td>3a</td>
<td>CMP</td>
</tr>
<tr>
<td>;</td>
<td>3b</td>
<td>CMP</td>
</tr>
<tr>
<td>&lt;</td>
<td>3c</td>
<td>CMP</td>
</tr>
<tr>
<td>=</td>
<td>3d</td>
<td>CMP</td>
</tr>
<tr>
<td>?</td>
<td>3f</td>
<td>AAS</td>
</tr>
<tr>
<td>@</td>
<td>40</td>
<td>INC</td>
</tr>
<tr>
<td>A</td>
<td>41</td>
<td>INC</td>
</tr>
<tr>
<td>Character</td>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>B</td>
<td>INC 42</td>
<td>INC</td>
</tr>
<tr>
<td>C</td>
<td>INC 43</td>
<td>INC</td>
</tr>
<tr>
<td>D</td>
<td>INC 44</td>
<td>INC</td>
</tr>
<tr>
<td>E</td>
<td>INC 45</td>
<td>INC</td>
</tr>
<tr>
<td>F</td>
<td>INC 46</td>
<td>INC</td>
</tr>
<tr>
<td>G</td>
<td>INC 47</td>
<td>INC</td>
</tr>
<tr>
<td>H</td>
<td>DEC 48</td>
<td>DEC</td>
</tr>
<tr>
<td>I</td>
<td>DEC 49</td>
<td>DEC</td>
</tr>
<tr>
<td>J</td>
<td>DEC 4a</td>
<td>DEC</td>
</tr>
<tr>
<td>K</td>
<td>DEC 4b</td>
<td>DEC</td>
</tr>
<tr>
<td>L</td>
<td>DEC 4c</td>
<td>DEC</td>
</tr>
<tr>
<td>M</td>
<td>DEC 4d</td>
<td>DEC</td>
</tr>
<tr>
<td>N</td>
<td>DEC 4e</td>
<td>DEC</td>
</tr>
<tr>
<td>O</td>
<td>DEC 4f</td>
<td>DEC</td>
</tr>
<tr>
<td>P</td>
<td>PUSH 50</td>
<td>PUSH</td>
</tr>
<tr>
<td>Q</td>
<td>PUSH 51</td>
<td>PUSH</td>
</tr>
<tr>
<td>R</td>
<td>PUSH 52</td>
<td>PUSH</td>
</tr>
<tr>
<td>S</td>
<td>PUSH 53</td>
<td>PUSH</td>
</tr>
<tr>
<td>T</td>
<td>PUSH 54</td>
<td>PUSH</td>
</tr>
<tr>
<td>U</td>
<td>PUSH 55</td>
<td>PUSH</td>
</tr>
<tr>
<td>V</td>
<td>PUSH 56</td>
<td>PUSH</td>
</tr>
<tr>
<td>W</td>
<td>PUSH 57</td>
<td>PUSH</td>
</tr>
<tr>
<td>X</td>
<td>POP 58</td>
<td>POP</td>
</tr>
<tr>
<td>Y</td>
<td>POP 59</td>
<td>POP</td>
</tr>
<tr>
<td>Z</td>
<td>POP 5a</td>
<td>POP</td>
</tr>
<tr>
<td>[</td>
<td>POP 5b</td>
<td>POP</td>
</tr>
<tr>
<td>\</td>
<td>POP 5c</td>
<td>POP</td>
</tr>
<tr>
<td>]</td>
<td>POP 5d</td>
<td>POP</td>
</tr>
<tr>
<td>^</td>
<td>POP 5e</td>
<td>POP</td>
</tr>
<tr>
<td>_</td>
<td>POP 5f</td>
<td>POP</td>
</tr>
<tr>
<td>`</td>
<td>PUSH 60</td>
<td>PUSH</td>
</tr>
<tr>
<td>a</td>
<td>POPA 61</td>
<td>POPA</td>
</tr>
<tr>
<td>f</td>
<td></td>
<td>(in 32-bit mode) switch to 16-bit operand size</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td>(in 32-bit mode) switch to 16-bit address size</td>
</tr>
<tr>
<td>h</td>
<td>PUSH 68</td>
<td>PUSH</td>
</tr>
<tr>
<td>i</td>
<td>IMUL 69</td>
<td>IMUL</td>
</tr>
<tr>
<td>j</td>
<td>PUSH 6a</td>
<td>PUSH</td>
</tr>
<tr>
<td>k</td>
<td>IMUL 6b</td>
<td>IMUL</td>
</tr>
<tr>
<td>p</td>
<td>JO 70</td>
<td>JO</td>
</tr>
<tr>
<td>q</td>
<td>JNO 71</td>
<td>JNO</td>
</tr>
<tr>
<td>r</td>
<td>JB 72</td>
<td>JB</td>
</tr>
<tr>
<td>s</td>
<td>73</td>
<td>JAE</td>
</tr>
<tr>
<td>t</td>
<td>74</td>
<td>JE</td>
</tr>
<tr>
<td>u</td>
<td>75</td>
<td>JNE</td>
</tr>
<tr>
<td>v</td>
<td>76</td>
<td>JBE</td>
</tr>
<tr>
<td>w</td>
<td>77</td>
<td>JA</td>
</tr>
<tr>
<td>x</td>
<td>78</td>
<td>JS</td>
</tr>
<tr>
<td>y</td>
<td>79</td>
<td>JNS</td>
</tr>
<tr>
<td>z</td>
<td>7a</td>
<td>JP</td>
</tr>
</tbody>
</table>

In summary: AAA, AAS, CMP, DEC, IMUL, INC, JA, JAE, JB, JBE, JE, JNE, JNO, JNS, JO, JP, JS, POP, POPA, PUSH, PUSHA, XOR.
Appendix B

ARM

B.1 Terminology

ARM was initially developed as 32-bit CPU, so that’s why a word here, unlike x86, is 32-bit.

byte 8-bit. The DB assembly directive is used for defining variables and arrays of bytes.

halfword 16-bit. DCW assembly directive —““. 

word 32-bit. DCD assembly directive —““. 

doubleword 64-bit. 

quadword 128-bit.

B.2 Versions

- ARMv4: Thumb mode introduced.
- ARMv6: used in iPhone 1st gen., iPhone 3G (Samsung 32-bit RISC ARM 1176JZ(F)-S that supports Thumb-2)
- ARMv7: Thumb-2 was added (2003). was used in iPhone 3GS, iPhone 4, iPad 1st gen. (ARM Cortex-A8), iPad 2 (Cortex-A9), iPad 3rd gen.
- ARMv7s: New instructions added. Was used in iPhone 5, iPhone 5c, iPad 4th gen. (Apple A6).
• ARMv8: 64-bit CPU, AKA ARM64 AKA AArch64. Was used in iPhone 5S, iPad Air (Apple A7). There is no Thumb mode in 64-bit mode, only ARM (4-byte instructions).

### B.3 32-bit ARM (AArch32)

#### B.3.1 General purpose registers

- R0— function result is usually returned using R0
- R1...R12— GPRs
- R13— AKA SP (stack pointer)
- R14— AKA LR (link register)
- R15— AKA PC (program counter)

R0-R3 are also called “scratch registers”: the function’s arguments are usually passed in them, and the values in them are not required to be restored upon the function’s exit.

#### B.3.2 Current Program Status Register (CPSR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..4</td>
<td>M—processor mode</td>
</tr>
<tr>
<td>5</td>
<td>T—Thumb state</td>
</tr>
<tr>
<td>6</td>
<td>F—FIQ disable</td>
</tr>
<tr>
<td>7</td>
<td>I—IRQ disable</td>
</tr>
<tr>
<td>8</td>
<td>A—imprecise data abort disable</td>
</tr>
<tr>
<td>9</td>
<td>E—data endianness</td>
</tr>
<tr>
<td>10..15, 25, 26</td>
<td>IT—if-then state</td>
</tr>
<tr>
<td>16..19</td>
<td>GE—greater-than-or-equal-to</td>
</tr>
<tr>
<td>20..23</td>
<td>DNM—do not modify</td>
</tr>
<tr>
<td>24</td>
<td>J—Java state</td>
</tr>
<tr>
<td>27</td>
<td>Q—sticky overflow</td>
</tr>
<tr>
<td>28</td>
<td>V—overflow</td>
</tr>
<tr>
<td>29</td>
<td>C—carry/borrow/extend</td>
</tr>
<tr>
<td>30</td>
<td>Z—zero bit</td>
</tr>
<tr>
<td>31</td>
<td>N—negative/less than</td>
</tr>
</tbody>
</table>
B.3.3 VFP (floating point) and NEON registers

<table>
<thead>
<tr>
<th>0..31 bits</th>
<th>32..64</th>
<th>65..96</th>
<th>97..127</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q0</td>
<td>D0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>128 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S-registers are 32-bit, used for the storage of single precision numbers.
D-registers are 64-bit ones, used for the storage of double precision numbers.
D- and S-registers share the same physical space in the CPU—it is possible to access a D-register via the S-registers (it is senseless though).
Likewise, the NEON Q-registers are 128-bit ones and share the same physical space in the CPU with the other floating point registers.
In VFP 32 S-registers are present: S0..S31.
In VFPv2 there 16 D-registers are added, which in fact occupy the same space as S0..S31.
In VFPv3 (NEON or “Advanced SIMD”) there are 16 more D-registers, D0..D31, but the D16..D31 registers are not sharing space with any other S-registers.
In NEON or “Advanced SIMD” another 16 128-bit Q-registers were added, which share the same space as D0..D31.

B.4 64-bit ARM (AArch64)

B.4.1 General purpose registers

The register count was doubled since AArch32.

- X0— function result is usually returned using X0
- X0...X7—Function arguments are passed here.
- X8
- X9...X15—are temporary registers, the callee function can use and not restore them.
- X16
- X17
- X18
- X19...X29—callee function can use them, but must restore them upon exit.
• X29—used as FP (at least GCC)
• X30—“Procedure Link Register” AKA LR (link register).
• X31—register always contains zero AKA XZR or “Zero Register”. It’s 32-bit part is called WZR.
• SP, not a general purpose register anymore.

See also: [ARM13c].

The 32-bit part of each X-register is also accessible via W-registers (W0, W1, etc).

<table>
<thead>
<tr>
<th>High 32-bit part</th>
<th>low 32-bit part</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0</td>
<td>W0</td>
</tr>
</tbody>
</table>

**B.5 Instructions**

There is a -S suffix for some instructions in ARM, indicating that the instruction sets the flags according to the result. Instructions which lacks this suffix are not modify flags. For example ADD unlike ADDS will add two numbers, but the flags will not be touched. Such instructions are convenient to use between CMP where the flags are set and, e.g. conditional jumps, where the flags are used. They are also better in terms of data dependency analysis (because less number of registers are modified during execution).
## B.5.1 Conditional codes table

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>Z = 1</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
<td>Z = 0</td>
</tr>
<tr>
<td>CS AKA HS (Higher or Same)</td>
<td>Carry set / Unsigned, Greater than, equal</td>
<td>C = 1</td>
</tr>
<tr>
<td>CC AKA LO (LOwer)</td>
<td>Carry clear / Unsigned, Less than</td>
<td>C = 0</td>
</tr>
<tr>
<td>MI</td>
<td>Minus, negative / Less than</td>
<td>N = 1</td>
</tr>
<tr>
<td>PL</td>
<td>Plus, positive or zero / Greater than, equal</td>
<td>N = 0</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow</td>
<td>V =</td>
</tr>
<tr>
<td>VC</td>
<td>No overflow</td>
<td>V =</td>
</tr>
<tr>
<td>HI</td>
<td>Unsigned higher / Greater than</td>
<td>C =</td>
</tr>
<tr>
<td>LS</td>
<td>Unsigned lower or same / Less than or equal</td>
<td>C =</td>
</tr>
<tr>
<td>GE</td>
<td>Signed greater than or equal / Greater than or equal</td>
<td>N =</td>
</tr>
<tr>
<td>LT</td>
<td>Signed less than / Less than</td>
<td>N =</td>
</tr>
<tr>
<td>GT</td>
<td>Signed greater than / Greater than</td>
<td>Z =</td>
</tr>
<tr>
<td>LE</td>
<td>Signed less than or equal / Less than, equal</td>
<td>Z =</td>
</tr>
<tr>
<td>None / AL</td>
<td>Always</td>
<td>Any</td>
</tr>
</tbody>
</table>
Appendix C

MIPS

C.1 Registers

(O32 calling convention)

C.1.1 General purpose registers GPR

<table>
<thead>
<tr>
<th>Number</th>
<th>Pseudoname</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$ZERO</td>
<td>Always zero. Writing to this register is effectively an idle instruction (NOP).</td>
</tr>
<tr>
<td>$1</td>
<td>$AT</td>
<td>Used as a temporary register for assembly macros and pseudoinstructions.</td>
</tr>
<tr>
<td>$2 ...$3</td>
<td>$V0 ...$V1</td>
<td>Function result is returned here.</td>
</tr>
<tr>
<td>$4 ...$7</td>
<td>$A0 ...$A3</td>
<td>Function arguments.</td>
</tr>
<tr>
<td>$8 ...$15</td>
<td>$T0 ...$T7</td>
<td>Used for temporary data.</td>
</tr>
<tr>
<td>$16 ...$23</td>
<td>$S0 ...$S7</td>
<td>Used for temporary data*.</td>
</tr>
<tr>
<td>$24 ...$25</td>
<td>$T8 ...$T9</td>
<td>Used for temporary data.</td>
</tr>
<tr>
<td>$26 ...$27</td>
<td>$K0 ...$K1</td>
<td>Reserved for OS kernel.</td>
</tr>
<tr>
<td>$28</td>
<td>$GP</td>
<td>Global Pointer**.</td>
</tr>
<tr>
<td>$29</td>
<td>$SP</td>
<td>SP*.</td>
</tr>
<tr>
<td>$30</td>
<td>$FP</td>
<td>FP*.</td>
</tr>
<tr>
<td>$31</td>
<td>$RA</td>
<td>RA.</td>
</tr>
<tr>
<td>n/a</td>
<td>PC</td>
<td>PC.</td>
</tr>
<tr>
<td>n/a</td>
<td>HI</td>
<td>high 32 bit of multiplication or division remainder***.</td>
</tr>
<tr>
<td>n/a</td>
<td>LO</td>
<td>low 32 bit of multiplication and division remainder***.</td>
</tr>
</tbody>
</table>
C.2 Instructions

There are 3 kinds of instructions:

- R-type: those which have 3 registers. R-instruction usually have the following form:

```
instruction destination, source1, source2
```

One important thing to remember is that when the first and second register are the same, IDA may show the instruction in its shorter form:

```
instruction destination/source1, source2
```

That somewhat reminds us of the Intel syntax for x86 assembly language.

- I-type: those which have 2 registers and a 16-bit immediate value.

- J-type: jump/branch instructions, have 26 bits for encoding the offset.

C.2.1 Jump instructions

What is the difference between B- instructions (BEQ, B, etc) and J- ones (JAL, JALR, etc)?

The B-instructions have an I-type, hence, the B-instructions’ offset is encoded as a 16-bit immediate. JR and JALR are R-type and jump to an absolute address specified in a register. J and JAL are J-type, hence the offset is encoded as a 26-bit immediate.
In short, B-instructions can encode a condition (B is in fact pseudoinstruction for `BEQ $ZERO, $ZERO, LABEL`), while J-instructions can’t.
Appendix D

Some GCC library functions

<table>
<thead>
<tr>
<th>name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>__divdi3</td>
<td>signed division</td>
</tr>
<tr>
<td>__moddi3</td>
<td>getting remainder (modulo) of signed division</td>
</tr>
<tr>
<td>__udivdi3</td>
<td>unsigned division</td>
</tr>
<tr>
<td>__umoddi3</td>
<td>getting remainder (modulo) of unsigned division</td>
</tr>
</tbody>
</table>
Appendix E

Some MSVC library functions

11 in function name stands for “long long”, e.g., a 64-bit data type.

<table>
<thead>
<tr>
<th>name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>__alldiv</td>
<td>signed division</td>
</tr>
<tr>
<td>__allmul</td>
<td>multiplication</td>
</tr>
<tr>
<td>__allrem</td>
<td>remainder of signed division</td>
</tr>
<tr>
<td>__allshr</td>
<td>shift left</td>
</tr>
<tr>
<td>__aulldiv</td>
<td>unsigned division</td>
</tr>
<tr>
<td>__aullrem</td>
<td>remainder of unsigned division</td>
</tr>
<tr>
<td>__aullshr</td>
<td>unsigned shift right</td>
</tr>
</tbody>
</table>

Multiplication and shift left procedures are the same for both signed and unsigned numbers, hence there is only one function for each operation here.

The source code of these function can be found in the installed MSVS, in VC/crt/src/intel/*.asm.
Appendix F

Cheatsheets

F.1 IDA

Hot-keys cheatsheet:
<table>
<thead>
<tr>
<th>key</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>switch listing and graph view</td>
</tr>
<tr>
<td>C</td>
<td>convert to code</td>
</tr>
<tr>
<td>D</td>
<td>convert to data</td>
</tr>
<tr>
<td>A</td>
<td>convert to string</td>
</tr>
<tr>
<td>*</td>
<td>convert to array</td>
</tr>
<tr>
<td>U</td>
<td>undefine</td>
</tr>
<tr>
<td>O</td>
<td>make offset of operand</td>
</tr>
<tr>
<td>H</td>
<td>make decimal number</td>
</tr>
<tr>
<td>R</td>
<td>make char</td>
</tr>
<tr>
<td>B</td>
<td>make binary number</td>
</tr>
<tr>
<td>Q</td>
<td>make hexadecimal number</td>
</tr>
<tr>
<td>N</td>
<td>rename identificator</td>
</tr>
<tr>
<td>?</td>
<td>calculator</td>
</tr>
<tr>
<td>G</td>
<td>jump to address</td>
</tr>
<tr>
<td>:</td>
<td>add comment</td>
</tr>
<tr>
<td>Ctrl-X</td>
<td>show references to the current function, label, variable (incl. in local stack)</td>
</tr>
<tr>
<td>X</td>
<td>show references to the function, label, variable, etc.</td>
</tr>
<tr>
<td>Alt-I</td>
<td>search for constant</td>
</tr>
<tr>
<td>Ctrl-I</td>
<td>search for the next occurrence of constant</td>
</tr>
<tr>
<td>Alt-B</td>
<td>search for byte sequence</td>
</tr>
<tr>
<td>Ctrl-B</td>
<td>search for the next occurrence of byte sequence</td>
</tr>
<tr>
<td>Alt-T</td>
<td>search for text (including instructions, etc)</td>
</tr>
<tr>
<td>Ctrl-T</td>
<td>search for the next occurrence of text</td>
</tr>
<tr>
<td>Alt-P</td>
<td>edit current function</td>
</tr>
<tr>
<td>Enter</td>
<td>jump to function, variable, etc</td>
</tr>
<tr>
<td>Esc</td>
<td>get back</td>
</tr>
<tr>
<td>Num -</td>
<td>fold function or selected area</td>
</tr>
<tr>
<td>Num +</td>
<td>unfold function or area</td>
</tr>
</tbody>
</table>

Function/area folding may be useful for hiding function parts when you realize what they do. This is used in my script\(^1\) for hiding some often used patterns of inline code.

### F.2 OllyDbg

Hot-keys cheatsheet:

\(^1\) [GitHub](#)
F.3 MSVC

Some useful options which were used through this book.

<table>
<thead>
<tr>
<th>option</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/O1</td>
<td>minimize space</td>
</tr>
<tr>
<td>/Ob0</td>
<td>no inline expansion</td>
</tr>
<tr>
<td>/Ox</td>
<td>maximum optimizations</td>
</tr>
<tr>
<td>/GS-</td>
<td>disable security checks (buffer overflows)</td>
</tr>
<tr>
<td>/Fa(file)</td>
<td>generate assembly listing</td>
</tr>
<tr>
<td>/Zi</td>
<td>enable debugging information</td>
</tr>
<tr>
<td>/Zp(n)</td>
<td>pack structs on $n$-byte boundary</td>
</tr>
<tr>
<td>/MD</td>
<td>produced executable will use MSVCR*.DLL</td>
</tr>
</tbody>
</table>

Some information about MSVC versions: 55.1 on page 916.

F.4 GCC

Some useful options which were used through this book.

<table>
<thead>
<tr>
<th>option</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Os</td>
<td>code size optimization</td>
</tr>
<tr>
<td>-O3</td>
<td>maximum optimization</td>
</tr>
<tr>
<td>-regparm=</td>
<td>how many arguments are to be passed in registers</td>
</tr>
<tr>
<td>-o file</td>
<td>set name of output file</td>
</tr>
<tr>
<td>-g</td>
<td>produce debugging information in resulting executable</td>
</tr>
<tr>
<td>-S</td>
<td>generate assembly listing file</td>
</tr>
<tr>
<td>-masm=intel</td>
<td>produce listing in Intel syntax</td>
</tr>
<tr>
<td>-fno-inline</td>
<td>do not inline functions</td>
</tr>
</tbody>
</table>

F.5 GDB

Some of commands we used in this book:
<table>
<thead>
<tr>
<th>option</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>break filename.c:number</td>
<td>set a breakpoint on line number in source code</td>
</tr>
<tr>
<td>break function</td>
<td>set a breakpoint on function</td>
</tr>
<tr>
<td>break *address</td>
<td>set a breakpoint on address</td>
</tr>
<tr>
<td>p variable</td>
<td>print value of variable</td>
</tr>
<tr>
<td>run</td>
<td>run</td>
</tr>
<tr>
<td>cont</td>
<td>continue execution</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>bt</td>
<td>print stack</td>
</tr>
<tr>
<td>set disassembly-flavor intel</td>
<td>set Intel syntax</td>
</tr>
<tr>
<td>disas</td>
<td>disassemble current function</td>
</tr>
<tr>
<td>disas function</td>
<td>disassemble function</td>
</tr>
<tr>
<td>disas function,+50</td>
<td>disassemble portion</td>
</tr>
<tr>
<td>disas $eip,+0x10</td>
<td></td>
</tr>
<tr>
<td>disas/r</td>
<td>disassemble with opcodes</td>
</tr>
<tr>
<td>info registers</td>
<td>print all registers</td>
</tr>
<tr>
<td>info float</td>
<td>print FPU-registers</td>
</tr>
<tr>
<td>info locals</td>
<td>dump local variables (if known)</td>
</tr>
<tr>
<td>x/w ...</td>
<td>dump memory as 32-bit word</td>
</tr>
<tr>
<td>x/w $rdi</td>
<td>dump memory as 32-bit word at address stored in RDI</td>
</tr>
<tr>
<td>x/10w ...</td>
<td>dump 10 memory words</td>
</tr>
<tr>
<td>x/s ...</td>
<td>dump memory as string</td>
</tr>
<tr>
<td>x/i ...</td>
<td>dump memory as code</td>
</tr>
<tr>
<td>x/10c ...</td>
<td>dump 10 characters</td>
</tr>
<tr>
<td>x/b ...</td>
<td>dump bytes</td>
</tr>
<tr>
<td>x/h ...</td>
<td>dump 16-bit halfwords</td>
</tr>
<tr>
<td>x/g ...</td>
<td>dump giant (64-bit) words</td>
</tr>
<tr>
<td>finish</td>
<td>execute till the end of function</td>
</tr>
<tr>
<td>next</td>
<td>next instruction (don’t dive into functions)</td>
</tr>
<tr>
<td>step</td>
<td>next instruction (dive into functions)</td>
</tr>
<tr>
<td>set step-mode on</td>
<td>do not use line number information while stepping</td>
</tr>
<tr>
<td>frame n</td>
<td>switch stack frame</td>
</tr>
<tr>
<td>info break</td>
<td>list of breakpoints</td>
</tr>
<tr>
<td>del n</td>
<td>delete breakpoint</td>
</tr>
<tr>
<td>set args ...</td>
<td>set command-line arguments</td>
</tr>
</tbody>
</table>
Acronyms used
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Operating System</td>
<td>xxx</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
<td>757</td>
</tr>
<tr>
<td>PL</td>
<td>Programming language</td>
<td>4</td>
</tr>
<tr>
<td>PRNG</td>
<td>Pseudorandom number generator</td>
<td>xviii</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-only memory</td>
<td>109</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic logic unit</td>
<td>30</td>
</tr>
<tr>
<td>RA</td>
<td>Return Address</td>
<td>24</td>
</tr>
<tr>
<td>PE</td>
<td>Portable Executable: 68.2 on page 988</td>
<td>988</td>
</tr>
<tr>
<td>SP</td>
<td>stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.</td>
<td>19</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic-link library</td>
<td>988</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter. IP/EIP/RIP in x86/64. PC in ARM.</td>
<td>20</td>
</tr>
<tr>
<td>LR</td>
<td>Link Register</td>
<td>8</td>
</tr>
<tr>
<td>IDA</td>
<td>Interactive Disassembler and debugger developed by Hex-Rays</td>
<td>7</td>
</tr>
<tr>
<td>IAT</td>
<td>Import Address Table</td>
<td>989</td>
</tr>
<tr>
<td>INT</td>
<td>Import Name Table</td>
<td>989</td>
</tr>
<tr>
<td>RVA</td>
<td>Relative Virtual Address</td>
<td>989</td>
</tr>
<tr>
<td>VA</td>
<td>Virtual Address</td>
<td>989</td>
</tr>
<tr>
<td>OEP</td>
<td>Original Entry Point</td>
<td>972</td>
</tr>
</tbody>
</table>
APPENDIX F. CHEATSHEETS

**APPENDIX F. CHEATSHEETS**

**MSVC** Microsoft Visual C++

**MSVS** Microsoft Visual Studio ............................................ 1313

**ASLR** Address Space Layout Randomization ............................. 990

**MFC** Microsoft Foundation Classes ......................................... 994

**TLS** Thread Local Storage ................................................ xxvii

**AKA** Also Known As

**CRT** C runtime library : 68.1 on page 983 ............................ 11

**CPU** Central processing unit ................................................ xxx

**FPU** Floating-point unit .................................................. xi

**CISC** Complex instruction set computing ................................ 20

**RISC** Reduced instruction set computing ................................ 5

**GUI** Graphical user interface ............................................. 983

**RTTI** Run-time type information ........................................ 779

**BSS** Block Started by Symbol ............................................ 28

**SIMD** Single instruction, multiple data ................................. 266

**BSOD** Blue Screen of Death ............................................. 973

**DBMS** Database management systems ................................ xxvii

**ISA** Instruction Set Architecture ........................................ v
HPC  High-Performance Computing ........................................... 711

SEH  Structured Exception Handling : 68.3 on page 998 ................. 46

ELF  Executable file format widely used in *NIX systems including Linux........xxvii

TIB  Thread Information Block.................................................. 381

PIC  Position Independent Code: 67.1 on page 975 ...................... xxvii

NAN  Not a Number .................................................................. 1299

NOP  No OPeration ................................................................. 32

BEQ  (PowerPC, ARM) Branch if Equal .................................... 128

BNE  (PowerPC, ARM) Branch if Not Equal ............................... 285

BLR  (PowerPC) Branch to Link Register .................................. 1076

XOR  eXclusive OR............................................................... 1308

MCU  Microcontroller unit ....................................................... 673

RAM  Random-access memory .................................................. 110

EGA  Enhanced Graphics Adapter ............................................ 1278

VGA  Video Graphics Array..................................................... 1278

API  Application programming interface .................................... 922

ASCII  American Standard Code for Information Interchange .......... 685

ASCIIZ  ASCII Zero (null-terminated ASCII string) ...................... 125
IA64  Intel Architecture 64 (Itanium): 93 on page 1273 ................. 620

EPIC  Explicitly parallel instruction computing .............................. 1273

OOE  Out-of-order execution ........................................................ 624

MSB  Most significant bit/byte .................................................... 428

LSB  Least significant bit/byte

STL  (C++) Standard Template Library: 51.4 on page 782 .................. 790

PODT  (C++) Plain Old Data Type .................................................. 808

HDD  Hard disk drive ................................................................. 826

VM  Virtual Memory

WRK  Windows Research Kernel .................................................... 944

GPR  General Purpose Registers .................................................. xxv

SSDT  System Service Dispatch Table .......................................... 973

RE  Reverse Engineering ............................................................ iv

BCD  Binary-coded decimal ......................................................... 1202

BOM  Byte order mark ................................................................. 929

GDB  GNU debugger ................................................................. 61

FP  Frame Pointer ................................................................. 27

MBR  Master Boot Record ........................................................... 936
Jump Parity Even (x86 instruction) ................................................. 325

Classless Inter-Domain Routing ................................................. 653

Store Multiple Full Descending (ARM instruction) ....................... 38

Load Multiple Full Descending (ARM instruction) ....................... 38

Store Multiple Empty Descending (ARM instruction) .................... 38

Load Multiple Empty Descending (ARM instruction) .................... 38

Store Multiple Full Ascending (ARM instruction) ....................... 38

Load Multiple Full Ascending (ARM instruction) ....................... 38

Store Multiple Empty Ascending (ARM instruction) ................... 38

Load Multiple Empty Ascending (ARM instruction) ................... 38

(ARM) Application Program Status Register .......................... 351

(ARM) Floating-Point Status and Control Register ................. 351

Program/process ID .......................................................... 1061

Line feed (10 or \n’ in C/C++) ............................................. 722

Carriage return (13 or \r’ in C/C++) ..................................... 722

Request for Comments ......................................................... 933

Top Of Stack ................................................................. 859

(Java) Local Variable Array ............................................... 868
JVM  Java virtual machine ............................................. xviii

JIT  Just-in-time compilation ........................................ 857

EOF  End of file ...................................................... 117
Glossary

**real number** numbers which may contain a dot. this is *float* and *double* in C/C++. 295

**decrement** Decrease by 1. 19, 249, 275, 605, 946, 1140, 1299, 1303, 1310

**increment** Increase by 1. 21, 249, 255, 275, 281, 439, 442, 605, 1135, 1299

**integral data type** usual numbers, but not a real ones. may be used for passing variables of boolean data type and enumerations. 314

**product** Multiplication result. 133, 303, 307, 558, 594, 678

**arithmetic mean** a sum of all values divided by their count. 711

**stack pointer** A register pointing to a place in the stack. 11, 12, 21, 38, 43, 54, 71, 73, 99, 135, 756, 838, 949, 951, 953, 954, 1291, 1301, 1318, 1332

**tail call** It is when the compiler (or interpreter) transforms the recursion (with which it is possible: *tail recursion*) into an iteration for efficiency: wikipedia. 643

**quotient** Division result. 295, 299, 301, 302, 307, 593, 673, 713, 1063

**anti-pattern** Generally considered as bad practice. 41, 103, 619

**atomic operation** “ατομικός” stands for “indivisible” in Greek, so an atomic operation is guaranteed not to be interrupted by other threads. 1031, 1266

**basic block** a group of instructions that do not have jump/branch instructions, and also don’t have jumps inside the block from the outside. In IDA it looks just like as a list of instructions without empty lines. 899, 1277, 1279

**callee** A function being called by another. 36, 42, 59, 91, 118, 132, 135, 139, 211, 578, 619, 756, 838, 949, 951–953, 957–959, 1323

**caller** A function calling another. 6–8, 11, 59, 118, 132–134, 137, 148, 211, 578, 625, 756, 949, 950, 953, 954, 959
**compiler intrinsic**  A function specific to a compiler which is not an usual library function. The compiler generates a specific machine code instead of a call to it. Often, it’s a pseudofunction for a specific CPU instruction. Read more: (90 on page 1261). 1310

**CP/M**  Control Program for Microcomputers: a very basic disk OS used before MS-DOS. 1196

**dongle**  Dongle is a small piece of hardware connected to LPT printer port (in past) or to USB. Its function was similar to a security token, it has some memory and, sometimes, a secret (crypto-)hashing algorithm. 1071

**endianness**  Byte order: 31 on page 619. 24, 106, 467, 1306

**GiB**  Gibibyte: $2^{30}$ or 1024 mebibytes or 1073741824 bytes. 17

**heap**  usually, a big chunk of memory provided by the OS so that applications can divide it by themselves as they wish. malloc()/free() work with the heap. 39, 42, 470, 781, 785, 805, 807, 984, 986

**jump offset**  a part of the JMP or Jcc instruction’s opcode, to be added to the address of the next instruction, and this is how the new PC is calculated. May be negative as well. 126, 177, 1299

**kernel mode**  A restrictions-free CPU mode in which the OS kernel and drivers execute. cf. user mode. 1340

**leaf function**  A function which does not call any other function. 34, 41

**link register**  (RISC)  A register where the return address is usually stored. This makes it possible to call leaf functions without using the stack, i.e., faster. 41, 1073, 1318, 1320

**loop unwinding**  It is when a compiler, instead of generating loop code for $n$ iterations, generates just $n$ copies of the loop body, in order to get rid of the instructions for loop maintenance. 253

**name mangling**  used at least in C++, where the compiler needs to encode the name of class, method and argument types in one string, which will become the internal name of the function. You can read more about it here: 51.1.1 on page 757. 754, 913–915

**NaN**  not a number: a special cases for floating point numbers, usually signaling about errors. 319, 341, 342, 1274
NEON  AKA “Advanced SIMD”—SIMD from ARM. 1319

NOP  “no operation”, idle instruction. 946

NTAPI API available only in the Windows NT line. Largely not documented by Microsoft. 1042

PDB  (Win32) Debugging information file, usually just function names, but sometimes also function arguments and local variables names. 912, 989, 1042, 1044, 1051, 1052, 1058, 1168

POKE  BASIC language instruction for writing a byte at a specific address. 946

register allocator  The part of the compiler that assigns CPU registers to local variables. 273, 412, 578

reverse engineering  act of understanding how the thing works, sometimes in order to clone it. vi, 1310

security cookie  A random value, different at each execution. You can read more about it here: 18.3 on page 378. 1017

stack frame  A part of the stack that contains information specific to the current function: local variables, function arguments, RA, etc. 92, 93, 133, 134, 638, 1017

stdout  standard output. 23, 44, 211

thunk function  Tiny function with a single role: call another function. 25, 534, 1073, 1086

tracer  My own simple debugging tool. You can read more about it here: 70.3 on page 1038. 256–258, 921, 937, 942, 943, 1010, 1024, 1171, 1181, 1188, 1189, 1192, 1257

user mode  A restricted CPU mode in which it all application software code is executed. cf. kernel mode. 1098, 1339

Windows NT  Windows NT, 2000, XP, Vista, 7, 8. 392, 573, 836, 926, 970, 987, 1030, 1201, 1310

word  data type fitting in GPR. In the computers older than PCs, the memory size was often measured in words rather than bytes. 790

xoring  often used in the English language, which implying applying the XOR operation. 1017, 1090, 1095
Index

.NET, 994
AT&T syntax, 14, 46
Buffer Overflow, 368, 1017
C language elements
  Pointers, 90, 99, 151, 524, 577
  Post-decrement, 604
  Post-increment, 604
  Pre-decrement, 604
  Pre-increment, 604
  C99, 149
    bool, 409
    restrict, 705
    variable length arrays, 382
const, 10, 110
for, 249, 647
if, 167, 210
return, 11, 118, 148
switch, 208, 210, 219
while, 271
C standard library
  alloca(), 43, 382, 619, 1003
  assert(), 390, 931
  atexit(), 789
  atoi(), 682, 1152
  calloc(), 1124
  close(), 977
  exit(), 625
  free(), 619
  fseek(), 1123
  ftell(), 1123
  getenv(), 1154
  localtime(), 850
  localtime_r(), 481
  longjmp(), 211
  malloc(), 471, 619
  memchr(), 1303
  memcmp(), 703, 934, 1306
  memcpy(), 14, 91, 699, 1302
  memset(), 697, 1188, 1305
  open(), 977
  pow(), 311
  puts(), 23
  qsort(), 524
  rand(), 456, 919, 1048, 1051, 1104
  read(), 977
  realloc(), 619
  scanf(), 90
  strcmp(), 693, 977
  strcpy(), 14, 697, 1105
  strlen(), 271, 571, 696, 721, 1303
  time(), 850
  tolower(), 1132
  toupper(), 733
  va_arg, 712
  va_list, 716
  vprintf, 716
Compiler’s anomalies, 199, 311, 405, 423, 447, 664, 729, 1259
C++, 1173
  exceptions, 1003
  C++11, 805, 962
  ostringstream, 777
References, 778
RTTI, 776
STL, 912
  std::forward_list, 804
| std::list | 790 |
| std::map | 815 |
| std::set | 815 |
| std::string | 779 |
| std::vector | 805 |
| Garbage collector | 889 |
| Global variables | 103 |
| grep usage | 258, 351, 912, 937, 942, 1170 |
| Hash functions | 622 |
| Intel syntax | 14, 19 |
| Mac OS X | 1037 |
| position-independent code | 20, 972 |
| RAM | 109 |
| ROM | 109, 110 |
| Base address | 986 |
| Binary tree | 815 |
| Dynamically loaded libraries | 24 |
| Linker | 110, 754 |
| RISC pipeline | 183 |
| Non-a-numbers (NaNs) | 341 |
| OOP Polymorphism | 754 |
| Reverse Polish notation | 356 |
| Buffer overflow | 375 |
| Entropy | 1218 |
| Recursion | 36, 40, 643 |
| Tail recursion | 643 |
| Stack | 38, 132, 211 |
| Stack overflow | 40 |
| Stack frame | 92 |
| Syntactic Sugar | 210 |
| Thumb-2 mode | 23 |
| iPod/iPhone/iPad | 18 |
| OllyDbg | 56, 94, 106, 134, 152, 171, 227, 255, 276, 300, 320, 331, 360, 370, 373, 394, 395, 437, 467, 493, 494, 500, 505, 529, 990, 1035, 1328 |
| Oracle RDBMS | 11, 557, 929, 999, 1174, 1187, 1189, 1231, 1244, 1259, 1277 |
| Ada | 145 |
| Alpha AXP | 5 |
| AMD | 957 |
| Angry Birds | 350, 351 |
| ARM | 281, 743, 1072, 1317 |
| ARM mode | 5 |

**Instructions**

| ADC | 545 |
| ADD | 23, 144, 182, 259, 432, 447, 676, 1320 |
| ADDAL | 182 |
| ADDCC | 233 |
| ADDS | 141, 545, 1320 |
| ADR | 20, 182 |
| ADRcc | 182, 219, 220, 621 |
| ADRP/ADD pair | 27, 72, 111, 388, 407, 608 |
| ANDcc | 736 |
| ASR | 452 |
| ASRS | 423, 676 |
| B | 71, 182, 184 |
| Bcc | 129, 131, 200 |
| BCS | 184, 353 |
| BEQ | 128, 220 |
| BGE | 184 |
| BIC | 422, 424, 430, 455 |
| BL | 20, 21, 23, 25, 27, 182, 609 |
| BLcc | 182 |
| BLE | 184 |
| BLS | 184 |
| BLT | 259 |
| BLX | 23, 24 |
| BNE | 184 |
| BX | 141, 236 |
| CMP | 128, 129, 182, 220, 233, 259, 447, 1320 |
| CSEL | 196, 203, 206, 449 |
| EOR | 430 |
| FCMPE | 353 |
| FCSEL | 353 |
| FMOV | 607 |
| FMRS | 432 |
| IT | 207, 350, 382 |
| LDMccFD | 182 |
| LDMEA | 38 |
LDMED, 38
LDMFA, 38
LDMFD, 20, 38, 182
LDP, 28
LDR, 74, 99, 110, 363, 386, 604
LDR.W, 402
LDRB, 495
LDRB.W, 282
LDRSB, 281
LSL, 448, 452
LSL.W, 448
LSLR, 737
LSLS, 364, 431, 737
LSR, 452
LSRS, 431
MADD, 142
MLA, 141
MOV, 7, 20, 22, 447, 676
MOVcc, 200, 206
MOVK, 606
MOVT, 22, 676
MOVT.W, 24
MOVW, 24
MUL, 144
MULS, 141
MVNS, 283
NEG, 691
ORR, 422
POP, 19, 20, 22, 38, 41
PUSH, 22, 38, 41
RET, 28
RSB, 191, 401, 447, 691
SBC, 545
SMMUL, 676
STMEA, 38
STMED, 38
STMFA, 38, 76
STMFD, 19, 38
STMIA, 74
STMIB, 76
STP, 26, 72
STR, 73, 363
SUB, 73, 401, 447
SUBcc, 736
SUBEQ, 283
SUBS, 545
SXTB, 495
SXTW, 407
TEST, 273
TST, 415, 447, 448
VADD, 306
VDIV, 306
VLDR, 306
VMOV, 306, 349
VMOVGT, 349
VMRS, 349
VMUL, 306
XOR, 192, 432
Pipeline, 234
Mode switching, 141, 236
Addressing modes, 604
mode switching, 24
Registers
  APSR, 349
  FPSCR, 349
  Link Register, 20, 41, 71, 236, 1318
  R0, 1318
  scratch registers, 282, 1318
  X0, 1319
  Z, 128, 1318
Thumb mode, 5, 184, 236
Thumb-2 mode, 5, 236, 350, 352
armel, 308
armhf, 308
Condition codes, 182
D-registers, 306, 1319
Data processing instructions, 676
DCB, 21
hard float, 308
if-then block, 350
Leaf function, 41
Optional operators
  ASR, 447, 676
  LSL, 363, 401, 447, 448, 606
  LSR, 447, 676
  ROR, 447
RRX, 447
Registers
R0, 146
S-registers, 306, 1319
soft float, 308
ARM64
lo12, 72
ASLR, 987
AWK, 940
Base64, 928
base64, 930
bash, 147
BASIC
POKE, 946
binary grep, 936, 1039
BIND.EXE, 994
binutils, 518
Bitcoin, 1261
Borland C++Builder, 915
Borland Delphi, 915, 924, 1256
BSoD, 970
BSS, 988
C11, 962
Callbacks, 524
Canary, 377
cdecl, 54, 949
COFF, 1083
column-major order, 393
Compiler intrinsic, 45, 1258
CRC32, 622, 644
CRT, 980, 1011
Cygwin, 914, 921, 995, 1037
DES, 557, 578
dlopen(), 977
dlsym(), 977
DOSBox, 1201
DosBox, 943
double, 297, 958
dtruss, 1037
Duff's device, 668
EICAR, 1195
ELF, 107
Error messages, 929
fastcall, 15, 89, 412, 951
float, 297, 958
Forth, 887
FORTRAN, 393, 705, 832, 914
FreeBSD, 935
Function epilogue, 36, 71, 74, 182, 495, 940
Function prologue, 12, 36, 41, 73, 377, 940
Fused multiply–add, 141, 142
Fuzzing, 691
GCC, 914, 1325, 1329
GDB, 34, 61, 66, 376, 535, 536, 1035, 1329
Glibc, 535, 970
HASP, 935
Hex-Rays, 1060
Hiew, 125, 177, 923, 989, 990, 995, 1257
IDA, 118, 518, 704, 904, 927, 1240, 1327
var_, 73, 99
IEEE 754, 297, 426, 512, 588, 1287
Inline code, 261, 422, 692, 763, 811
Integer overflow, 145
Intel
8080, 281
8086, 281, 422, 1098
Memory model, 848, 1275
8253, 1199
80286, 1098, 1276
80386, 422, 1276
80486, 296
FPU, 296
Intel C++, 11, 558, 1259, 1277, 1301
Itanium, 1270
Java, 854
jumptable, 226, 236
Keil, 18
kernel panic, 970
kernel space, 970

LD_PRELOAD, 976
Linux, 413, 972, 1174
libc.so.6, 411, 534
LLVM, 18
long double, 297
Loop unwinding, 252

Mac OS Classic, 1071
MD5, 622, 933
MFC, 991, 1154
MIDI, 934
MinGW, 914
minifloat, 607
MIPS, 5, 745, 944, 988, 1072
Global Pointer, 28

Instructions
ADD, 145
ADD.D, 311
ADDIU, 29, 115, 116
ADDU, 145
AND, 425
BC1F, 356
BC1T, 356
BEQ, 131, 186
BLTZ, 192
BNE, 186
BNEZ, 238
BREAK, 677
C.LT.D, 356
DIVD, 311
J, 7, 30
JAL, 145
JALR, 29, 145
JR, 224
LB, 268
LBU, 268
LI, 610
LUI, 29, 115, 116, 311, 430, 610
LW, 29, 100, 116, 224
LWC1, 311
MFC1, 316
MFHI, 145, 677, 1323
MFLO, 145, 677, 1323
MTC1, 521
MUL.D, 311
MULT, 145
NOR, 285
OR, 33
ORI, 425, 610
SB, 268
SLL, 238, 289, 451
SLLV, 451
SLT, 185
SLTIU, 238
SLTU, 185, 188, 238
SRL, 295
SUBU, 192
SW, 82

Pseudoinstructions
B, 264
BEQZ, 188
L.D, 311
LA, 33
LI, 7
MOVE, 30, 113
NEGU, 192
NOP, 33, 113
NOT, 285

Registers
FCCR, 355
HI, 677
LO, 677
Branch delay slot, 8
Global Pointer, 402
Load delay slot, 224
O32, 82, 88, 89, 1322
MS-DOS, 379, 843, 934, 943, 946, 985, 1098, 1195, 1197, 1224, 1256, 1275, 1287, 1303, 1310
DOS extenders, 1276
MSVC, 1326, 1329
Name mangling, 754
Native API, 987
NEC V20, 1201
objdump, 518, 975, 995
OEP, 985, 994
opaque predicate, 749
OpenMP, 918, 1261
OpenWatcom, 914, 953
Page (memory), 573
Pascal, 924
PDP-11, 604
PowerPC, 5, 28, 1071
puts() instead of printf(), 23, 97, 147, 179
Quake III Arena, 523
Raspberry Pi, 18
ReactOS, 1007
Register allocation, 578
Relocation, 24
row-major order, 393
RVA, 986
SAP, 912, 1168
SCO OpenServer, 1082
Scratch space, 955
Security cookie, 377, 1017
Security through obscurity, 930
SHA1, 622
SHA512, 1261
Shadow space, 137, 139, 590
Shellcode, 748, 970, 987, 1196, 1315
Signed numbers, 169, 614
SIMD, 588, 703
SSE, 588
SSE2, 588
stdcall, 949, 1256
strace, 976, 1037
syscall, 411, 970, 1037
TCP/IP, 618
thiscall, 754, 756, 953
thunk-functions, 25, 993, 1073, 1086
TLS, 379, 962, 988, 994, 1292
    Callbacks, 967
    Callbacks, 995
trace, 256, 531, 533, 921, 937, 942, 1010,
    1024, 1035, 1171, 1181, 1188,
    1189, 1191, 1257
UFS2, 935
Unicode, 924
Unrolled loop, 261, 382, 698
uptime, 976
USB, 1074
user space, 970
UTF-16LE, 924, 926
UTF-8, 924, 925
VA, 986
Watcom, 914
Windows, 1030
    API, 1287
    IAT, 986
    INT, 986
    KERNEL32.DLL, 410
    MSVCR80.DLL, 527
    NTAPI, 1042
    ntoskrnl.exe, 1174
    PDB, 912, 989, 1042, 1051, 1168
    Structured Exception Handling, 46, 995
TIB, 379, 996, 1292
Win32, 409, 926, 976, 985, 1276
    GetProcAddress, 994
    LoadLibrary, 994
    Ordinal, 990
    RaiseException(), 996
    SetUnhandledExceptionFilter(), 999
Windows 2000, 987
Windows 3.x, 836, 1276
Windows NT4, 987
Windows Vista, 985, 1042
Windows XP, 987, 994, 1051
Wine, 1007
Wolfram Mathematica, 680, 681, 1062,
    1218
x86
Flags
- CF, 1299, 1303, 1306, 1311, 1312
- DF, 1306, 1312
- IF, 1306, 1312

Instructions
- AAA, 1316
- AAS, 1316
- ADC, 543, 843, 1299
- ADD, 11, 54, 133, 684, 843, 1299
- ADDSD, 589
- ADDSS, 602
- ADRcc, 195
- AND, 12, 410, 416, 435, 453, 504, 1299, 1306
- BSF, 575, 1306
- BSR, 1306
- BSWAP, 618, 1306
- BT, 1306
- BTC, 429, 1306
- BTR, 429, 1031, 1306
- BTS, 429, 1306
- CALL, 10, 40, 740, 993, 1299
- CBW, 615, 1306
- CDQ, 554, 615, 1306
- CDQE, 615, 1306
- CLD, 1306
- CLI, 1306
- CMC, 1306
- CMOVcc, 183, 195, 197, 201, 206, 621, 1306
- CMP, 118, 1299, 1316
- CMPSB, 934, 1306
- CMPSD, 1306
- CMPSQ, 1306
- CMPSW, 1306
- COMISD, 599
- COMISS, 602
- CPUID, 500, 1309
- CWD, 615, 843, 1212, 1306
- CWDE, 615, 1306
- DEC, 275, 1299, 1316
- DIV, 615, 1309
- DIVSD, 589, 939
- FABS, 1313
- FADD, 1313
- FADDP, 299, 305, 1313
- FATRET, 445, 446
- FCHS, 1313
- FCMOVcc, 344
- FCOM, 330, 342, 1313
- FCOMP, 317, 1313
- FCOMPP, 1313
- FDIV, 299, 937, 938, 1313
- FDIVP, 299, 1313
- FDIVR, 305, 1313
- FDIVRP, 1313
- FILD, 1313
- FIST, 1313
- FISTP, 1313
- FLD, 312, 317, 1313
- FLD1, 1313
- FLDCW, 1313
- FLDZ, 1313
- FMUL, 299, 1313
- FMULP, 1313
- FNSTCW, 1314
- FNSTSW, 318, 342, 1314
- FSINCOS, 1314
- FSQRT, 1314
- FST, 1314
- FSTCW, 1314
- FSTCW, 1314
- FSTP, 312, 1314
- FSTSW, 1314
- FSUB, 1314
- FSUBP, 1314
- FSUBR, 1314
- FSUBRP, 1314
- FCOM, 342, 1314
- FCOMI, 344
- FUCOM, 1314
- FUCOMI, 341, 1314
- FWAIT, 296
- FXCH, 1314
- IDIV, 615, 673, 1309
- IMUL, 133, 405, 615, 1299, 1316
- IN, 740, 1098, 1199, 1310
INC, 275, 1257, 1299, 1316
INT, 1196, 1310
INT3, 921
IRET, 1310
JA, 169, 343, 615, 1299, 1316
JAE, 169, 1299, 1316
JB, 169, 615, 1299, 1316
JBE, 169, 1299, 1316
JC, 1299
Jcc, 131, 199
JCXZ, 1299
JE, 211, 1299, 1316
JE CXZ, 1299
JG, 169, 615, 1299
JGE, 169, 1299
JL, 169, 615, 1299
JLE, 168, 1299
JMP, 40, 71, 993, 1256, 1299
JNA, 1299
JNAE, 1299
JNB, 1299
JNBE, 342, 1299
JNC, 1299
JNE, 118, 169, 1299, 1316
JNG, 1299
JNGE, 1299
JNL, 1299
JNLE, 1299
JNO, 1299, 1316
JNS, 1299, 1316
JNZ, 1299
JO, 1299, 1316
JP, 319, 1200, 1299, 1316
JPO, 1299
JRCXZ, 1299
JS, 1299, 1316
JZ, 128, 211, 1259, 1299
LAHF, 1300
LEA, 93, 136, 474, 627, 649, 684, 956, 1046, 1301
LEAVE, 12, 1300
LES, 1105, 1211
LOCK, 1030
LODSB, 1200
LOOP, 249, 270, 940, 1211, 1310
MAXSD, 599
MOV, 7, 11, 15, 698, 699, 740, 990, 1257, 1302
MOVDQA, 562
MOVDQU, 562
MOVSB, 1302
MOVSD, 597, 701, 1131, 1302
MOVSDX, 597
MOVSS, 602
MOVSX, 1302
MOVSDX, 1302
MOVSSD, 272, 281, 493, 495, 615, 1302
MOVSXD, 384
MOVZX, 274, 471, 1072, 1302
MUL, 615, 1302
MULSD, 589
NEG, 689, 1302
NOP, 649, 1253, 1256, 1302
NOT, 280, 283, 1137, 1303
OR, 416, 721, 1303
OUT, 740, 1098, 1310
PADDD, 562
PCMP EQB, 574
PLMULHW, 557
PLMULLD, 557
PMOVMSKB, 575
POP, 11, 38, 40, 1303, 1316
POPA, 1311, 1316
POPCNT, 1311
POPF, 1199, 1311
PUSH, 10, 12, 38, 40, 92, 740, 1303, 1316
PUSHA, 1311, 1316
PUSHF, 1311
PXOR, 574
RCL, 940, 1311
RCR, 1311
RET, 6, 11, 40, 377, 756, 838, 1256, 1303
ROL, 446, 1258, 1311
<table>
<thead>
<tr>
<th>Instruction</th>
<th>DS, 1275</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAHF, 342, 1303</td>
<td>EAX, 118, 146</td>
</tr>
<tr>
<td>SAL, 1312</td>
<td>EBP, 92, 133</td>
</tr>
<tr>
<td>SALC, 1201</td>
<td>ECX, 754</td>
</tr>
<tr>
<td>SAR, 452, 615, 710, 1211, 1312</td>
<td>ES, 1211, 1275</td>
</tr>
<tr>
<td>SBB, 543, 1303</td>
<td>ESP, 54, 92</td>
</tr>
<tr>
<td>SCASB, 1200, 1303</td>
<td>FS, 965</td>
</tr>
<tr>
<td>SCASD, 1303</td>
<td>GS, 378, 965, 969</td>
</tr>
<tr>
<td>SCASQ, 1303</td>
<td>JMP, 231</td>
</tr>
<tr>
<td>SCASW, 1303</td>
<td>RIP, 975</td>
</tr>
<tr>
<td>SETALC, 1201</td>
<td>SS, 1275</td>
</tr>
<tr>
<td>SETcc, 185, 274, 342, 1312</td>
<td>ZF, 118, 410</td>
</tr>
<tr>
<td>SHL, 288, 359, 452, 1304</td>
<td>AVX, 556</td>
</tr>
<tr>
<td>SHR, 295, 452, 503, 1304</td>
<td>FPU, 1293</td>
</tr>
<tr>
<td>SHRD, 553, 1304</td>
<td>Instructions</td>
</tr>
<tr>
<td>STC, 1312</td>
<td>STOSB, 671</td>
</tr>
<tr>
<td>STD, 1312</td>
<td>MMX, 556</td>
</tr>
<tr>
<td>STI, 1312</td>
<td>SSE, 556</td>
</tr>
<tr>
<td>STOSB, 1305</td>
<td>SSE2, 556</td>
</tr>
<tr>
<td>STOSD, 1305</td>
<td>x86-64, 15, 16, 64, 91, 97, 126, 135, 577, 588, 742, 953, 975, 1288, 1296</td>
</tr>
<tr>
<td>STOSQ, 699, 1305</td>
<td>Xcode, 18</td>
</tr>
<tr>
<td>STOSW, 1305</td>
<td>Z3, 1059, 1064</td>
</tr>
<tr>
<td>SUB, 11, 12, 118, 211, 684, 1299, 1305</td>
<td></td>
</tr>
<tr>
<td>SYSCALL, 1310, 1312</td>
<td></td>
</tr>
<tr>
<td>SYSENTER, 971, 1310, 1312</td>
<td></td>
</tr>
<tr>
<td>TEST, 272, 410, 415, 453, 1305</td>
<td></td>
</tr>
<tr>
<td>UD2, 1312</td>
<td></td>
</tr>
<tr>
<td>XADD, 1032</td>
<td></td>
</tr>
<tr>
<td>XCHG, 1302, 1305</td>
<td></td>
</tr>
<tr>
<td>XOR, 11, 118, 280, 710, 940, 1090, 1257, 1305, 1316</td>
<td></td>
</tr>
<tr>
<td>Prefixes</td>
<td></td>
</tr>
<tr>
<td>LOCK, 1031, 1298</td>
<td></td>
</tr>
<tr>
<td>REP, 1298, 1302, 1305</td>
<td></td>
</tr>
<tr>
<td>REPE/REPNE, 1298</td>
<td></td>
</tr>
<tr>
<td>REPNE, 1303</td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td></td>
</tr>
<tr>
<td>Flags, 118, 171, 1292</td>
<td></td>
</tr>
<tr>
<td>AH, 1300, 1303</td>
<td></td>
</tr>
<tr>
<td>CS, 1275</td>
<td></td>
</tr>
<tr>
<td>DR6, 1297</td>
<td></td>
</tr>
<tr>
<td>DR7, 1297</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Available as</th>
</tr>
</thead>
<tbody>
<tr>
<td>[haq]</td>
<td>papasutraofhaquebright. “WRITING SHELLCODE FOR IA-64”. In: (). Also available as <a href="http://go.yurichev.com/17340">http://go.yurichev.com/17340</a>.</td>
<td></td>
</tr>
</tbody>
</table>


