Introduction to Software Reverse Engineering

Pascal Junod (HEIG-VD)

SU 2016

Part I

Administrativia

Teacher

- Pascal Junod, professor @ HEIG-VD since end of 2008
- Co-founder of `strong.codes SA`, a startup active in the domain of sotware protection.
- Gone through ETH Zurich, EPF Lausanne, Nagravision (groupe Kudelski)
- Main research interests:
  - Industrial cryptography
  - Software protection
  - Ethical hacking
- `http://crypto.junod.info`
- Twitter: `@cryptopathe`

Prerequisites

- Good skills in C/C++ and Java programming
- Some familiarity with a script language (e.g. Python)
- Good knowledge of the Linux operating system
- Willingness to put your hands dirty

Lecture Web Page

- A web page is available at `http://crypto.junod.info/files/SU16`.
- Username: `SU16`
- Password given in class

Calendar

Day	Contents	Labs
08-07-2016 AM	Introduction	L1
08-07-2016 PM	x86 Architecture	L2
11-07-2016 PM	ARM Architecture	L3
12-07-2016 PM	Interpreted Languages	L4
13-07-2016 AM	Structure & Life of an Executable	Test

Evaluation

- Written report (one or two students) on one of the labs' solution, no more than 2 pages (50%)
- Send your report in PDF format to `pascal.junod@heig-vd.ch` before Tuesday 18h00 CET.
- Multiple-choice on Wednesday morning (50%)

Recommended Tools

- Students are free to use the software and tools they like.
- A good toolbox is Kali Linux (cf. `https://www.kali.org/` and `http://docs.kali.org/general-use/kali-linux-virtual-box-guest`)

Books for Further Study

- Reverse engineering can hardly be learnt in books: one must practice it!
- Nevertheless, the two following books can be recommended for further study:
  - Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, Wiley, 2015, ISBN 978-0764574818.
  - Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sébastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, Wiley, 2014, ISBN 978-1118787311.

Part II

Introduction

Objectives

At the end of this part, the student will be able to ...

- ... describe the concept of software reverse engineering and related activities.
- ... name the most important tools that help the reverse engineering process.
- ... explain the difference between static and dynamic reverse engineering.

Basics

What is Reverse Engineering?

Definition of Wikipedia:

> *Reverse engineering, also called back engineering, is the processes of extracting knowledge or design information from anything man-made and re-producing it or reproducing anything based on the extracted information. The process often involves disassembling something (a mechanical device, electronic component, computer program, or biological, chemical, or organic matter) and analyzing its components and workings in detail.*
> *The reasons and goals for obtaining such information vary widely from everyday or socially beneficial actions, to criminal actions, depending upon the situation.*

Is Software Reverse Engineering Legal?

- USA: according to Wikipedia, "*Reverse engineering of computer software in the US often falls under both contract law as a breach of contract as well as any other relevant laws. This is because most EULA's (end user license agreement) specifically prohibit it [...] the DMCA (17 U.S.C. § 1201 (f)) says that a person who is in legal possession of a program, is permitted to reverse-engineer and circumvent its protection if this is necessary in order to achieve "interoperability" [...]*"

Is Software Reverse Engineering Legal?

- Europe: according to Wikipedia, "*[...] in general, "unauthorised reproduction, translation, adaptation or transformation" is unlawful. An exemption exists for reverse engineering (as in the U.S.) when this is performed for interoperability purposes but the law prohibits use of the knowledge gained, in a way that prejudices the rightsholder's position or legitimate interests (for example, reverse engineering performed to create a competing product). It also prohibits the public release of information obtained through reverse engineering of software.*"

Is Software Reverse Engineering Legal?

- Rule of thumb: in general, performing reverse engineering is lawful, but be *very* careful with what is done with the acquired information!
- There are many aspects of software reverse engineering that are justified: malware analysis, interoperability, etc.

Goals of Reverse Engineering

- Interfacing (for interoperability purposes)
- Military or commercial espionage
- Improve documentation shortcomings
- Obsolescence, software modernization
- Product security analysis, backdoor identification
- Bug fixing
- Academic/learning purposes
- etc.

Intellectual Property

- Publishing some intellectual property («patent application») allows to gain some legal protection on its use.
- Reverse engineering is one traditional way to prove that a competitor infringes a patent.
- Dissymetric process, as reverse engineering is costly in terms of time and money.

Malware Analysis

- Malware authors virtually never publish the source code of their products.
- Reverse engineering is a necessary step to:
  - understand the inner workings of a piece of malware;
  - design an identification/protection method;
  - perform attribution.

Vulnerability Analysis

- Reverse engineering is often an obligatory step when performing a software vulnerability analysis:
  - vulnerability discovery;
  - vulnerability analysis;
  - exploit development.

Interoperability

- Software is often released in binary form, without any published specifications.
- Reverse engineering is the only possible way to write interoperable software:
  - PC-BIOS
  - Samba
  - LibreOffice
  - Wine
- Important counter-example: Skype protocol

Compiler Validation

- Verification of the results of a compiler's work:
  - Correctness
  - Optimization
  - Security (branch-free, constant-time implementations, backdoor insertion, etc.)

Malicious Aspects

- License management system cracking
- Rogue analysis/modification of an executable:
  - game cheating;
  - code carving;
  - confidential data leak;
  - DRM cracking;
  - security check bypass;
  - security token emulation;
  - etc.

Overall Process

From the Code Source to an Executable

Let us consider the following piece of C code:

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

int main (void)
{
  int i, sum = 0;

  for (i = 1; i <= 63; i++) {
    sum += i;
  }
  fprintf (stdout, "The sum of 1 to 63 is %d.\n", sum);

  return EXIT_SUCCESS;
}
```

From the Code Source to an Executable (2)

The compiler's frontend (here, LLVM's `clang`) generates some intermediate representation :

```
; ModuleID = 'exec_build.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.11.0"

; [...]

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %i = alloca i32, align 4
  %sum = alloca i32, align 4
  store i32 0, i32* %1
  store i32 0, i32* %sum, align 4
  store i32 1, i32* %i, align 4
  br label %2

; [...]
```

From the Code Source to an Executable (3)

The intermediate representation is then transformed into assembler code by the compiler's backend:

```
1         .section        __TEXT,__text,regular,pure_instructions
2         .globl          _main
3         .align          4, 0x90
4 _main:                                  ## @main
5         .cfi_startproc
6 ## BB#0:
7         pushq           %rbp
8 Ltmp2:
9         .cfi_def_cfa_offset 16
10 Ltmp3:
11        .cfi_offset %rbp, -16
12        movq            %rsp, %rbp
13 Ltmp4:
14        .cfi_def_cfa_register %rbp
15        subq            $16, %rsp
16        movl            $0, -4(%rbp)
17        movl            $0, -12(%rbp)
18        movl            $1, -8(%rbp)
19 LBB0_1:                                ## =>This Inner Loop Header: Depth=1
20
21 ## [...]
```

From the Code Source to an Executable (4)

The assembler is then transformed into machine code, that can be directly executed on the target architecture.

```
1  0000000    cf  fa  ed  fe  07  00  00  01  03  00  00  80  02  00  00  00
2  0000020    10  00  00  00  60  05  00  00  85  00  20  00  00  00  00  00
3  0000040    19  00  00  00  48  00  00  00  5f  5f  50  41  47  45  5a  45
4  0000060    52  4f  00  00  00  00  00  00  00  00  00  00  00  00  00  00
5  0000100    00  00  00  00  01  00  00  00  00  00  00  00  00  00  00  00
6  0000120    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
7  0000140    00  00  00  00  00  00  00  00  19  00  00  00  28  02  00  00
8  0000160    5f  5f  54  45  58  54  00  00  00  00  00  00  00  00  00  00
9  0000200    00  00  00  00  01  00  00  00  00  10  00  00  00  00  00  00
10 0000220    00  00  00  00  00  00  00  00  00  10  00  00  00  00  00  00
11 0000240    07  00  00  00  05  00  00  00  06  00  00  00  00  00  00  00
12 0000260    5f  5f  74  65  78  74  00  00  00  00  00  00  00  00  00  00
13 0000300    5f  5f  54  45  58  54  00  00  00  00  00  00  00  00  00  00
14 0000320    f0  0e  00  00  01  00  00  00  70  00  00  00  00  00  00  00
15 0000340    f0  0e  00  00  04  00  00  00  00  00  00  00  00  00  00  00
16 0000360    00  04  00  80  00  00  00  00  00  00  00  00  00  00  00  00
17 0000400    5f  5f  73  74  75  62  73  00  00  00  00  00  00  00  00  00
18 0000420    5f  5f  54  45  58  54  00  00  00  00  00  00  00  00  00  00
19 0000440    60  0f  00  00  01  00  00  00  06  00  00  00  00  00  00  00
20 [...]
```

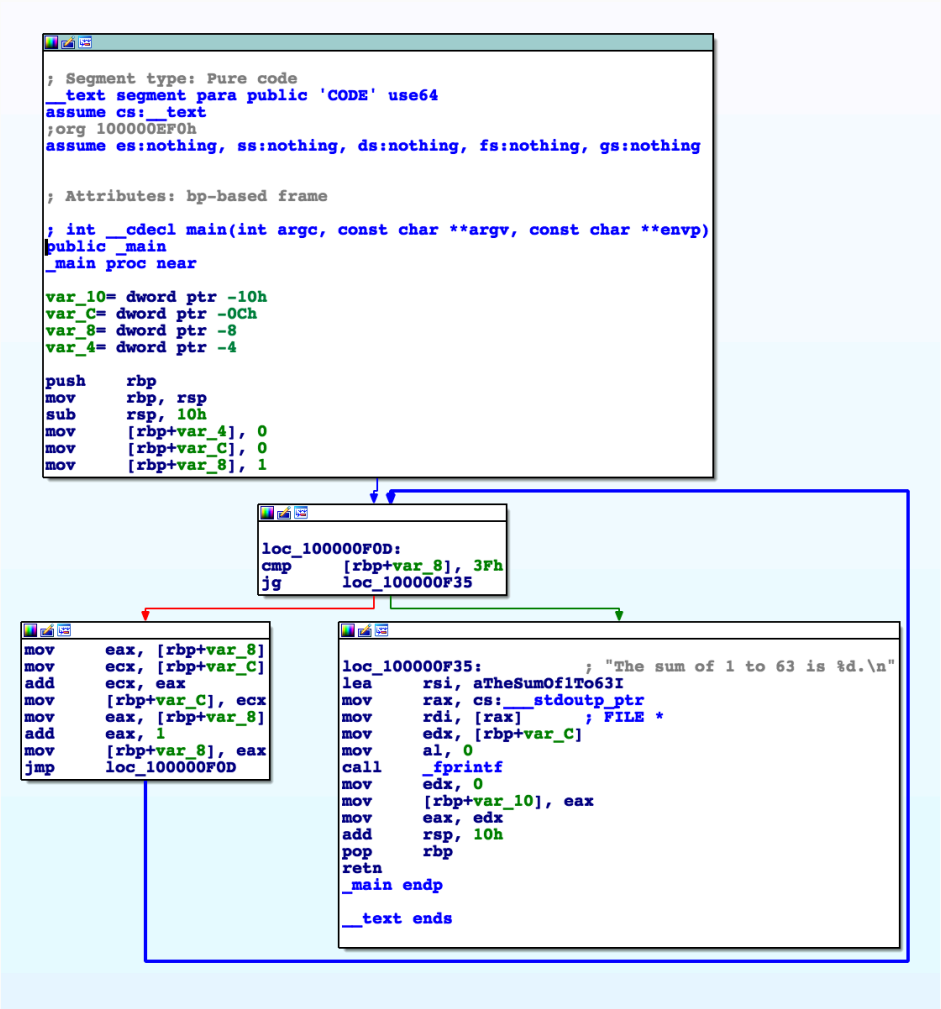
From an Executable to Code Source

A simple disassembler (`otool -tV` on OS X) is able to transform the executable into assembler code:

```
exec_build:
(__TEXT,__text) section
_main:
0000000100000ef0        pushq        %rbp
0000000100000ef1        movq        %rsp, %rbp
0000000100000ef4        subq        $0x10, %rsp
0000000100000ef8        movl        $0x0, -0x4(%rbp)
0000000100000eff        movl        $0x0, -0xc(%rbp)
0000000100000f06        movl        $0x1, -0x8(%rbp)
0000000100000f0d        cmpl        $0x3f, -0x8(%rbp)
0000000100000f14        jg        0x100000f35
0000000100000f1a        movl        -0x8(%rbp), %eax
0000000100000f1d        movl        -0xc(%rbp), %ecx
0000000100000f20        addl        %eax, %ecx
0000000100000f22        movl        %ecx, -0xc(%rbp)
0000000100000f25        movl        -0x8(%rbp), %eax
0000000100000f28        addl        $0x1, %eax
[...]
```

From an Executable to Code Source (2)

A more advanced disassembler (like IDA Pro) gives a better overview of the disassembled code:



From an Executable to Code Source (3)

A good decompiler (like Hexrays) is often able to give a very good version in C:

```
 1 int __cdecl main(int argc, const char **argv, const char **envp)
 2 {
 3   unsigned int v4; // [rsp+4h] [rbp-Ch]@1
 4   signed int i; // [rsp+8h] [rbp-8h]@1
 5
 6   v4 = 0;
 7   for ( i = 1; i <= 63; ++i )
 8     v4 += i;
 9   fprintf(*(FILE **)__stdoutp_ptr, "The sum of 1 to 63 is %d.\n", v4);
10   return 0;
11 }
```

Static vs. Dynamic Analysis

- Static analysis is all what can be done without executing the binary:
  - partially disassembly (but what to do with `jmp [eax]`?)
  - list of dynamic imports/exports with respect to dynamic libraries (`.so`, `.dll`, `.dylib`, ...)
- Dynamic analysis englobes additional information provided at runtime:
  - contents of memory and registers;
  - listing and results of interactions with the operating system;
  - etc.

Disassembly

- Disassembly is a 4-stage process:
  1. Find an entry point (not always easy when mixed data and code)
  2. Decode an opcode (not always trivial in case of variable length instructions, or instruction prefixes)
  3. Print instruction
  4. Find next instruction

Linear Sweep Disassembly

- Linear sweep is the simplest way to proceed
- Technique mainly used by tools like `gdb`, `objdump`, `otool`, etc.
- Main idea: `«Where one instruction ends, another begins!»`
  - No effort is made to understand the control flow, hence no recognition of branches, loops, etc.
  - Provides a complete coverage of a binary code section.
  - Fails to account the fact that data can be commingled with code (switch constructs, jump tables, templating, virtual functions, etc.)

Recursive Disassembly

- Recursive disassembly focuses on the concept of control flow.
- It determines whether an instruction should be dissassembled or not based on whether it is referenced from another instruction.
- Strategy:
  1. Sequential flow: pass execution to the instruction right after
  2. Conditional branches: disassemble both possible paths
  3. Unconditional branches: try to determine next instruction
  4. Function calls are similar to unconditional instructions, but with a `ret` (return instruction)
  5. Return instruction: gives no information about what instruction will be executed next, as the return address is red on the stack on most architectures.

Recursive Disassembly (2)

- Superior ability to distinguish code from data, but still unable to follow indirect code paths

Decompilation

- In addition to disassembly:
  - Perform semantic analysis to recover data types, like `long` variables;
  - Perform data flow analysis to remove lower-level aspects such as registers, condition codes, stack references.
  - Perform control flow analysis to recover control structures (loops, conditional branches, switch structures, etc.)
  - Perform type analysis to recover high-level data types, such as arrays, structures and classes.
- For more information on decompilation, see e.g. `http://www.program-transformation.org/Transform/DeCompilation`.

Tools

Small Utilities: `strings`

Out of `man strings`:

> *For each file given,* `strings` *prints the printable character sequences that are at least 4 characters long (or the number given with the options below) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.* `strings` *is mainly useful for determining the contents of non-text files.*

For instance:

```
$ strings code/a.out
The sum of 1 to 63 is %d.
```

Small Utilities: `file`

Out of `man file`:

> `file` - *determine file type*

For instance:

```
$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18,
stripped
$ file /bin/ls
/bin/ls: Mach-O 64-bit executable x86_64
```

Small Utilities on Linux

- `nm`: list the symbols in an executable
- `readelf`: displays information about ELF files
- `ldd`: displays the shared objects dependencies
- `lsof`: list files (and other system ressources) opened by a process
- `ltrace`: traces the calls to (dynamic) library functions of a process
- `strace`: traces the systems calls of a process
- ...

Disassemblers

- `objdump` (for OS X:`otool`)
- `disas` command of `gdb`
- `Radare` portable reversing framework
- `Capstone` lightweight multi-platform, multi-architecture disassembly framework.
- `Hopper` (commercial, free trial version)
- `IDA Pro` (free and commercial)
- ...

Decompilers

- `Boomerang`
- `Hexrays` is plugin of IDA Pro (commercial)
- `.NET Reflector`
- `JD` (Java Decompiler)
- `uncompyle2` is a Python decompiler
- `JPEXS Flash Decompiler`
- ...

Debuggers

- `gdb` is the GNU open-source debugger
- `lldb` is an open-source debugger from the LLVM framework
- `WinDbg` is a multipurpose debugger for the Microsoft Windows computer operating system, distributed by Microsoft
- `OllyDbg` is a free debugger for Microsoft Windows
- `x64dbg` is an open-source debugger for Microsoft Windows
- ...

Emulators

- `qemu` is the Swiss army knife of emulators
- `Bochs` is open-source `x86-32` emulator
- ...

Binary Instrumentation Frameworks

- `PIN` is a free, but closed-source instrumentation framework provided by Intel
- Various compilers offer instrumentation capabilities.

Lab L1

- `http://crypto.junod.info/files/SU16/L1`
- *Hint*: basic tools are sufficient to break it!

Part III

x86 Architecture

Objectives

At the end of this part, the student will be able to ...

- ... describe the main characteristics of the Intel x86 architecture.
- ... name important registers of the Intel x86 architecture.
- ... name the most important instructions of the Intel x86 instruction set.
- ... recognize typical control-flow constructs in x86 assembler.
- ... explain what is a calling convention and name a few of them.

Basics

A Bit of (Non-Exhaustive) History

Year	Micro-Architecture	Main Feature
1978	8086	First **x86** processor
1982	80286	Protected mode
1985	Intel386	32-bit architecture
1989	Intel486	Built-in floating point
1993	Intel Pentium	Super-scaling, branch predictor
1995	Intel Pentium II	SIMD
2000	Netburst	First 64-bit architecture
2003	Pentium M	SSE2
2006	Intel Core	**x86-64** SSE
2008	Nehalem	Embedded memory controller
2011	Sandy Bridge	AVX
2013	Haswell	AVX2
2015	Skylake	AVX-512

Basics of x86

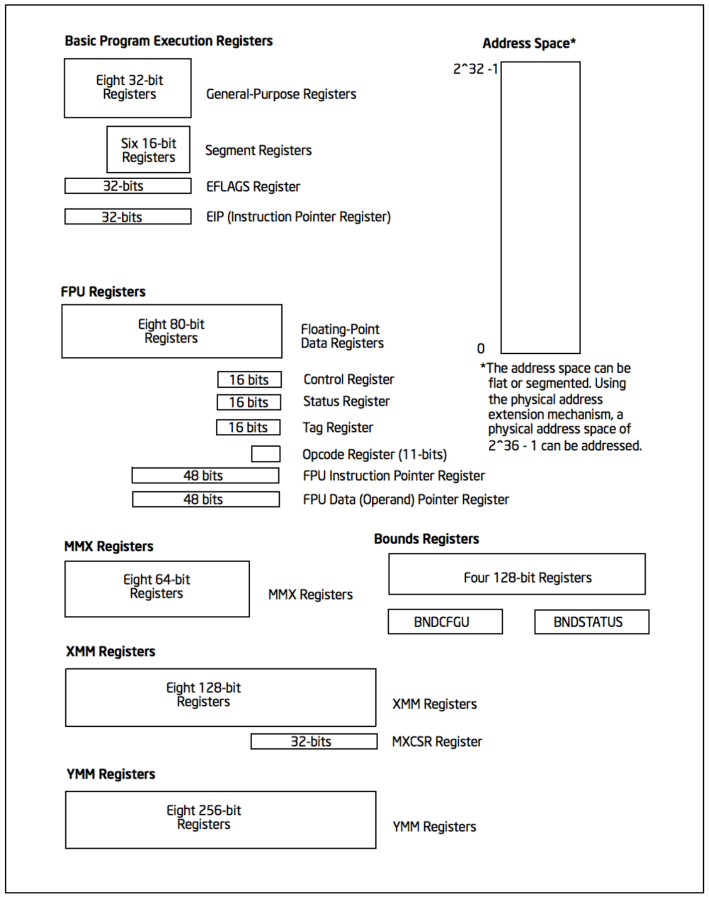
- Little-endian CISC architecture
- 32-bit version is called **x86** or IA-32
- 64-bit version is called **x64** or **x86-64**
- Supports real and protected modes
- In protected mode, supports the concept of privilege separation through rings (ring 0 to ring 3).
  - Ring 0 is the highest privilege level and can modify all system settings (usually, kernel land).
  - Ring 3 is the lowest privilege level and can only read/modify a subset of system settings (usually, user land).
- The architecture is fully documented in the *Intel 64 and IA-32 architectures software developer's manual* (3800+ pages).

Registers and Memory

Main registers

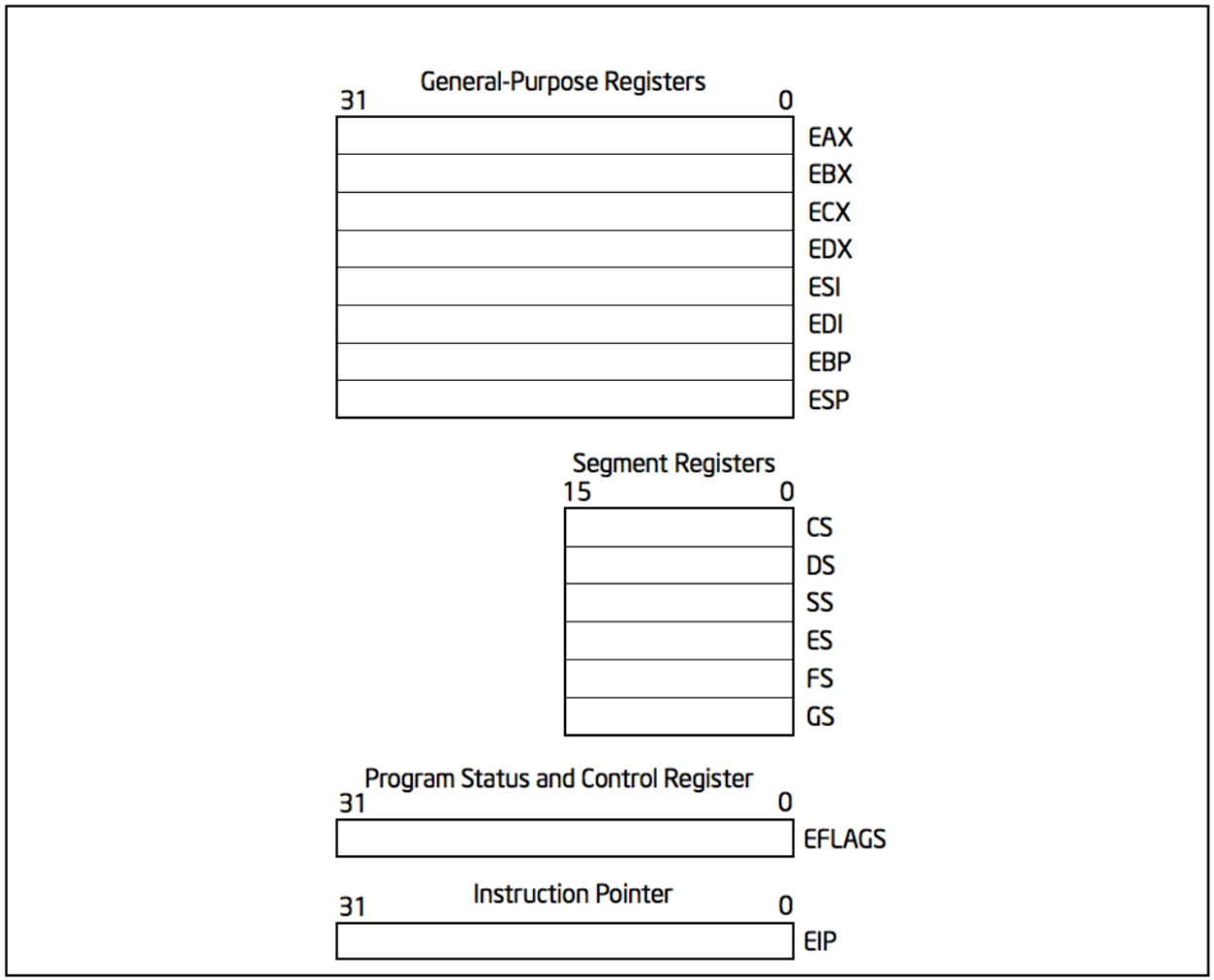
- In 32-bit protected mode, there are 8 general-purpose registers: `EAX`, `EBX`, `ECX`, `EDX`, `EDI`, `ESI`, `EBP` and `ESP`.
- In 64-bit protected mode, there are 16 general-purpose registers: `RAX`, `RBX`, `RCX`, `RDX`, `RDI`, `RSI`, `RBP`, `RSP` and `R8` to `R15`.
- Special purposes:
  - `ECX`: counter in loops
  - `ESI`: source in string/memory operations
  - `EDI`: destination in string/memory operations
  - `EBP`: base frame pointer
  - `ESP`: stack pointer

x86 Registers



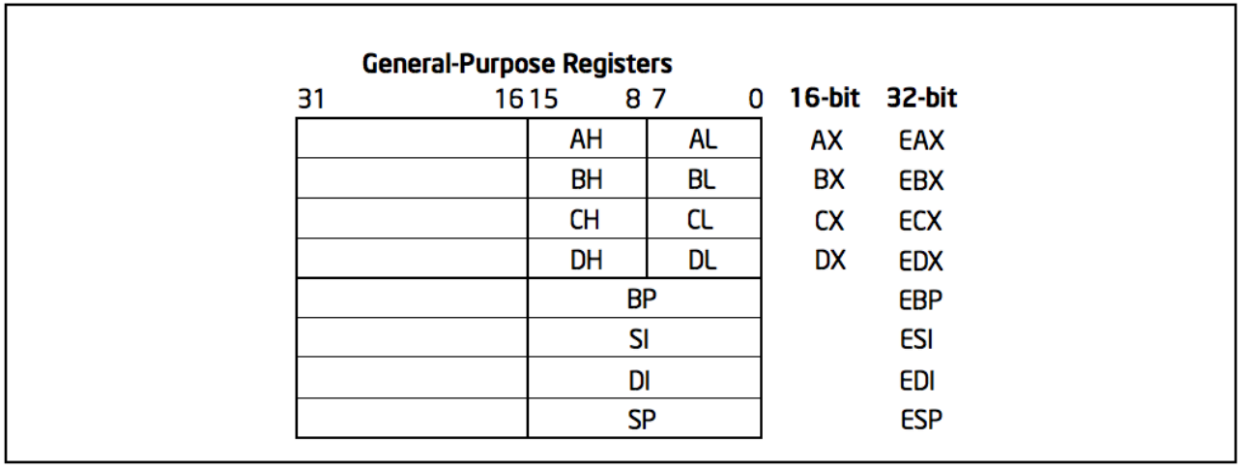
Picture source: *Intel 64 and IA-32 architectures software developer's manual*

x86 Registers (2)



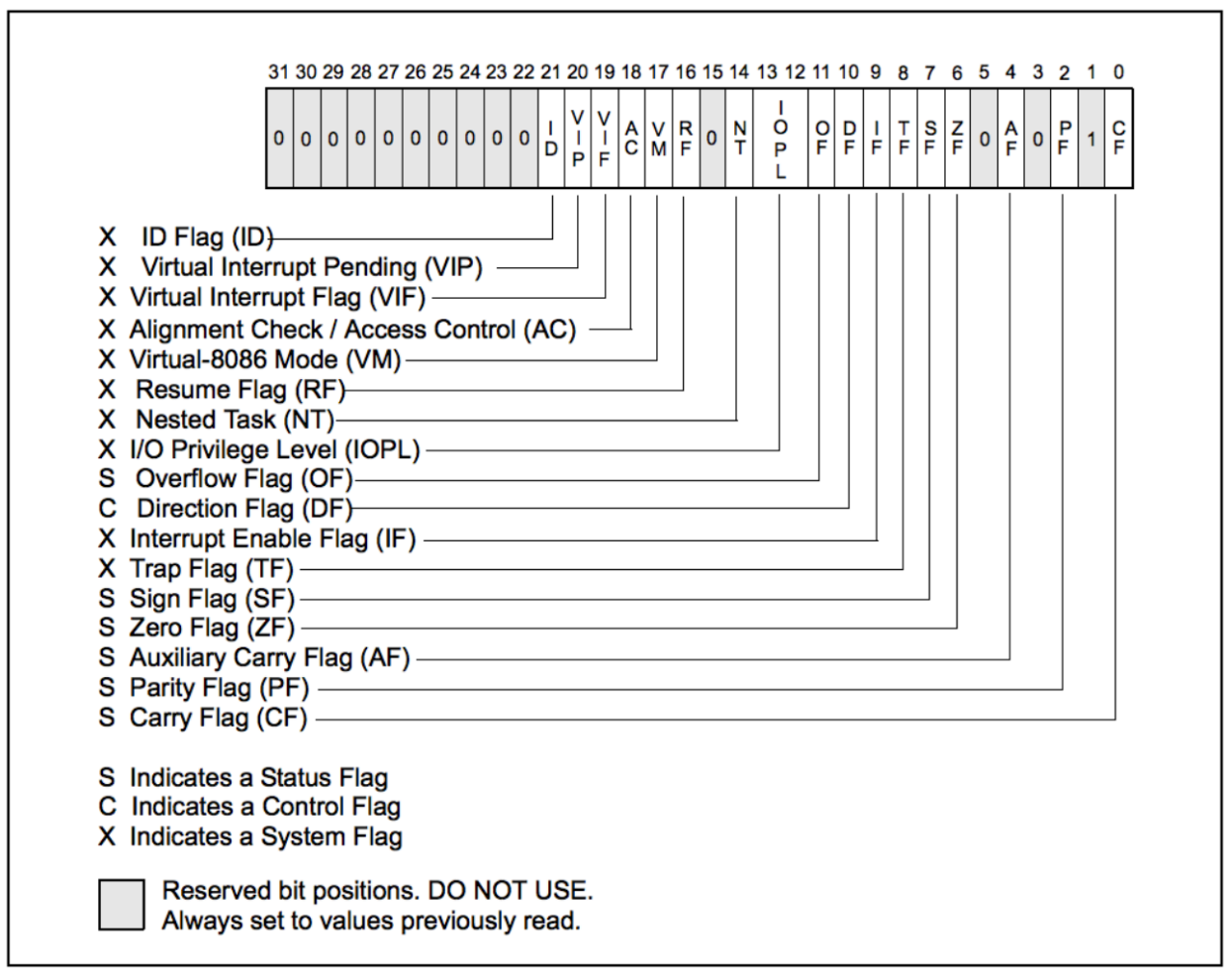
Picture source: *Intel 64 and IA-32 architectures software developer's manual*

x86 Registers (3)



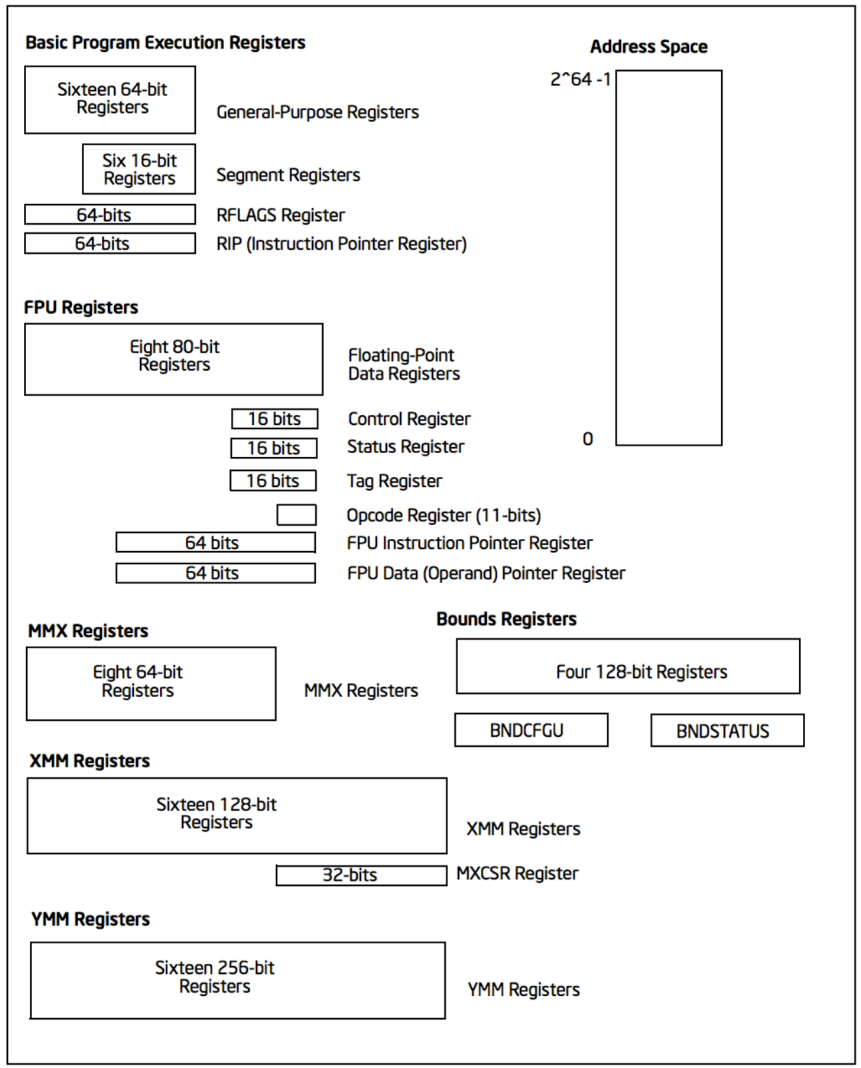
Picture source: *Intel 64 and IA-32 architectures software developer's manual*

x86 Flags



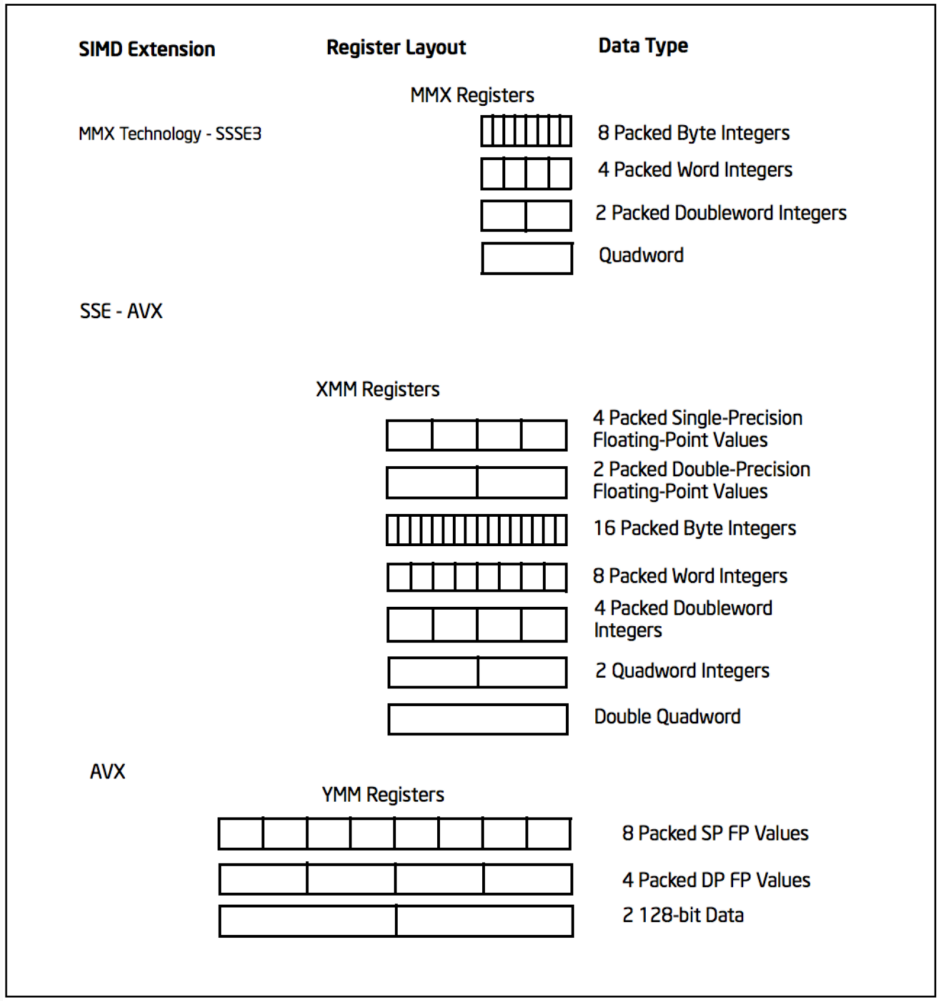
Picture source: *Intel 64 and IA-32 architectures software developer's manual*

x86-64 Registers



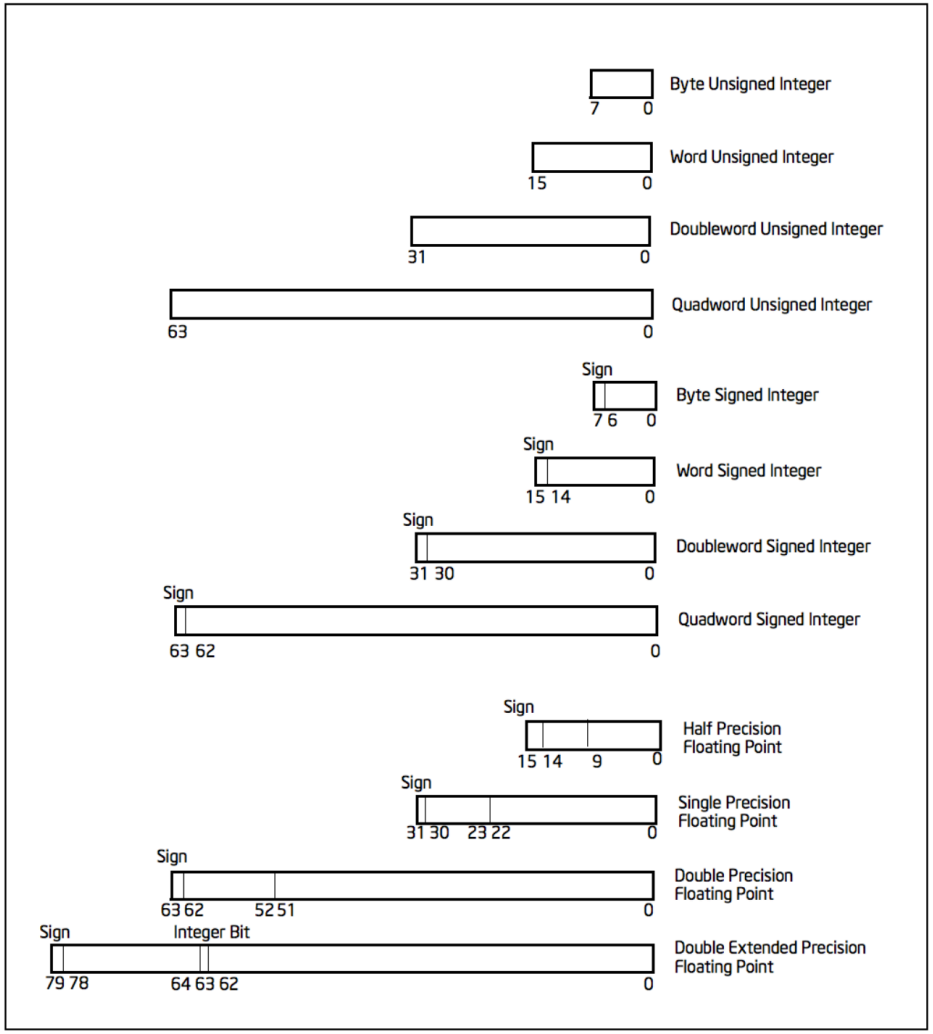
Picture source: *Intel 64 and IA-32 architectures software developer's manual*

Data Types



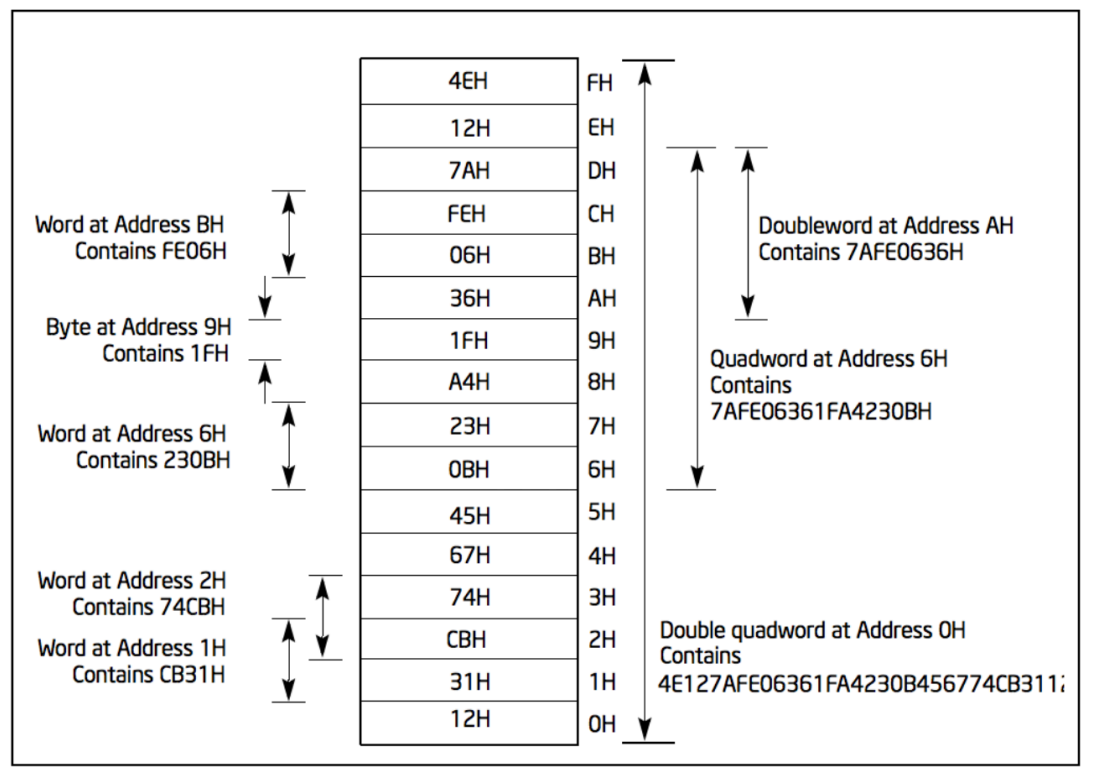
Picture source: *Intel 64 and IA-32 architectures software developer's manual*

Data Types (2)



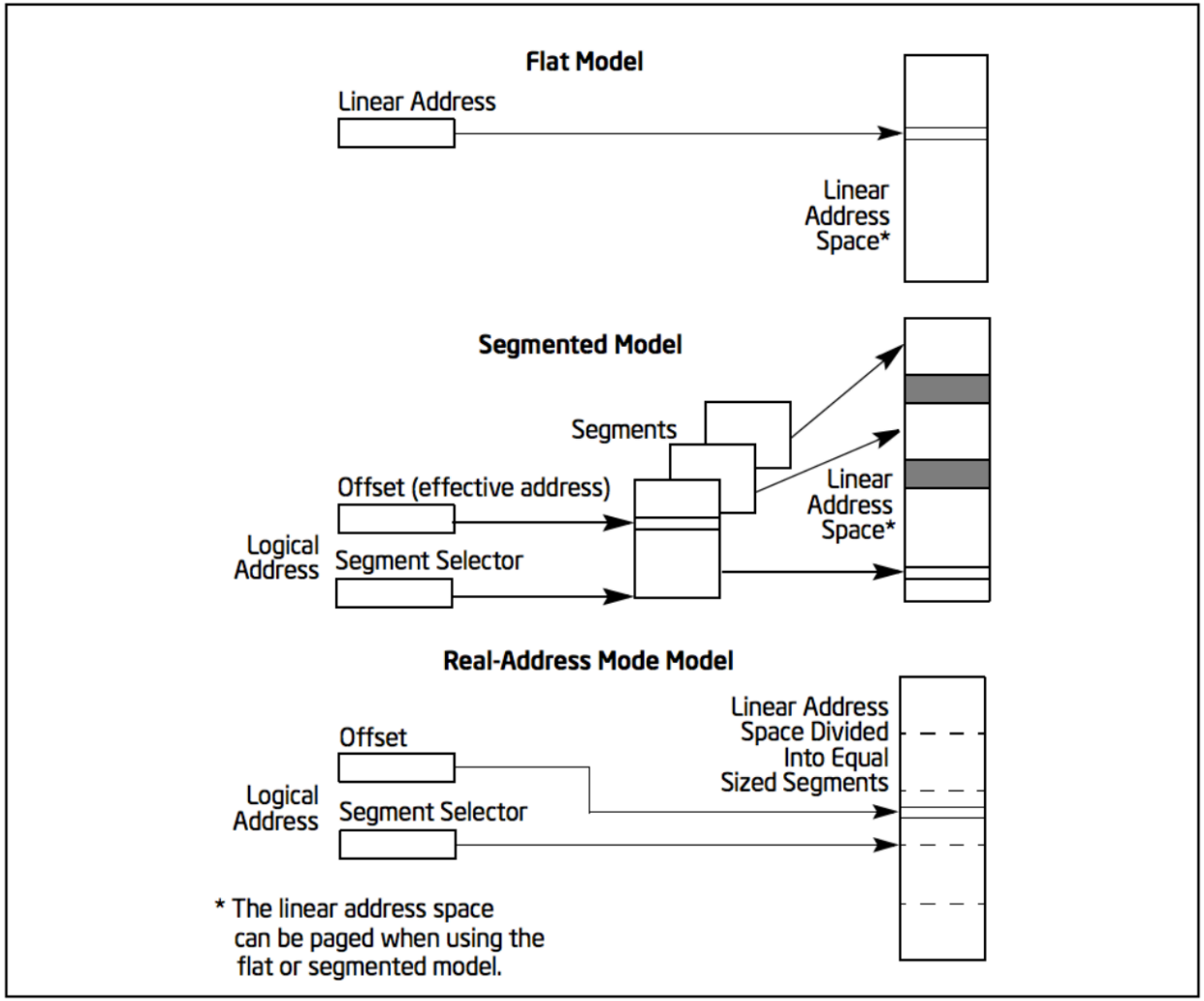
Picture source: *Intel 64 and IA-32 architectures software developer's manual*

Data Types (3)



Picture source: *Intel 64 and IA-32 architectures software developer's manual*

Memory Models



Picture source: *Intel 64 and IA-32 architectures software developer's manual*

Instructions

Data Movements

- The Intel instruction set allows a variety of data movements:
  - immediate value to register;
  - register to register;
  - immediate value to memory;
  - register to memory;
  - memory to register;
  - memory to memory.

Data Movements (2)

- In a RISC architecture, incrementing a counter stored in memory typically requires three instructions:
  - read the data from memory to a register;
  - increment the value in the register;
  - store the data from the register to memory.
- On `x86`, such an operation will require a single instruction such as `ADD` or `INC`, as they can directly address the memory.

Two Assembler Syntaxes

- Depending on the assembler/disassembler, there are two different syntaxes for x86 assembly code: Intel and AT&T.

```
; Intel syntax
mov ecx, DEADBEEFh
mov ecx, [eax]
mov eax, ecx

; AT&T syntax
movl $0xDEADBEEF, %ecx
movl (%eax), %ecx
movl %ecx, %eax
```

Two Assembler Syntaxes (2)

- Main differences:
  - AT&T prefixes the register with `%`, and immediates with `$`.
  - AT&T adds a prefix to the instruction to indicate operation width (`MOVL`, `MOVB`, …).
  - AT&T puts the source operand before the destination one. Intel uses the inverse order.
- Tools on Windows typically use the Intel notation, while AT&T is more popular on Unix and derivates.

Important Instructions

- `MOV`: move data between registers and/or memory
- `MOVSB`, `MOVSW`, `MOVSD`: move data with 1-, 2- and 4-byte granularity between two memory addresses.
- `SCASx`: implicitely compares `AL`/`AX`/`EAX` with data starting at address `EDI`, which is automatically increased depending on the `DF` bit.
- `STOSx`: same as `SCAS` except that it writes the value `AL`/`AX`/`EAX` to `EDI`.

Important Instructions (2)

- `AND`, `OR`, `XOR`, `NOT`: Boolean operators
- `MUL`: takes a register or a memory value, and multiplies it with `AL`, `AX` or `EAX`. The result is put in `AX`, `DX:AX` or `EDX:EAX`
- `IMUL regA, regB/mem`: `regA *= regB/mem`
- `IMUL regA, regB/mem, imm`: `regA = regB/mem*imm`
- `DIV`/`IDIV`: division, the dividend is put in `AX`, `DX:AX` or `EDX:EAX` and the resulting quotient/remainder is stored in `AL/AH`, `AX/DX` or `EAX/EDX`.

Important Instructions (3)

- `PUSH`/`POP`: push/pop a value onto the stack, and modify `ESP` at the same time.
- `CALL`: first, pushes the return address (i.e., the address immediately after the `CALL`) on the stack; then, changes `EIP` to the call destination.
- `RET`: pops the return address stored on the top of the stack and transfers control to it.

Control Flow

Control Flow Modification

- The instruction control flow can be modified through instructions such as `CMP`, `TEST`, `JMP` and `Jcc` instructions as well as the `EFLAGS` register:
  - `ZF` (zero flag): set if the result of the previous arithmetic operation is zero.
  - `SF` (sign flag): set to the most significant bit of the result.
  - `CF` (carry flag): set when the result requires a carry. Applies to *unsigned* integers.
  - `OF` (oveflow flag): set if the result overflows the maximal size. Applies to *signed* integers.

Control Flow Modification (2)

- The `CMP` instruction compares two operands by subtracting them and sets the appropriate flags.
- The `TEST` instruction compares two operands by computing a Boolean AND and sets the appropriate flags.
- `JMP`: inconditional branch operation.
- `Jcc`: conditional branch operation based on the condition code `cc`.

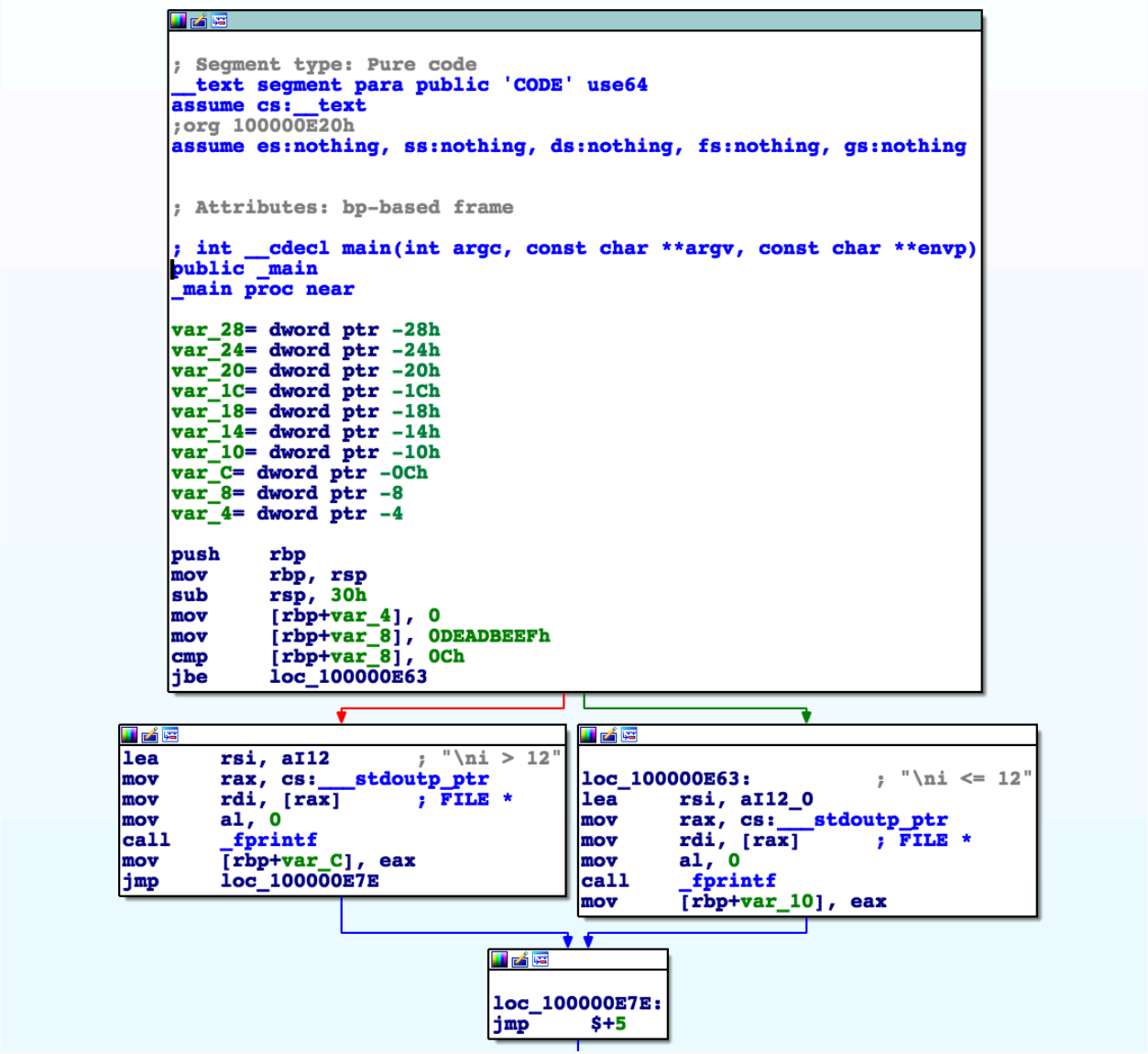
Control Flow Modification (3)

- Usual condition codes:
  - `B`/`NAE` (unsigned operations): below/neither above nor equal (`CF==1`).
  - `NB`/`AE` (unsigned operations): not below/above or equal (`CF==0`).
  - `E`/`Z`: equal/zero (`ZF==1`)
  - `NE`/`NZ`: not equal/non-zero (`ZF==0`)
  - `L` (signed operations): less than/neither greater nor equal (`SF⊕OF==1`).
  - `GE`/`NL` (signed operations): greater or equal/not less than (`SF⊕OF==0`).
  - `G`/`NLE` (signed operations): greater/not less nor equal (`(SF⊕OF)|ZF==0`).

Control Flow: Example Program

```c
#include <stdio.h>

int main (void)
{
  unsigned int i = 0xDEADBEEFUL;

  if (i > 12) { fprintf (stdout, "\ni > 12"); }
   else { fprintf (stdout, "\ni <= 12"); }

  while (i >= 12)
    i--;

  switch (i) {
  case 12:
    fprintf (stdout, "\ni == 12"); break;
  case 11:
    fprintf (stdout, "\ni == 11"); goto label;
  default:
    fprintf (stdout, "\ni != 12 and i != 11");
  }

 label:
  return 0;
}
```

Control Flow: Example Program (2)



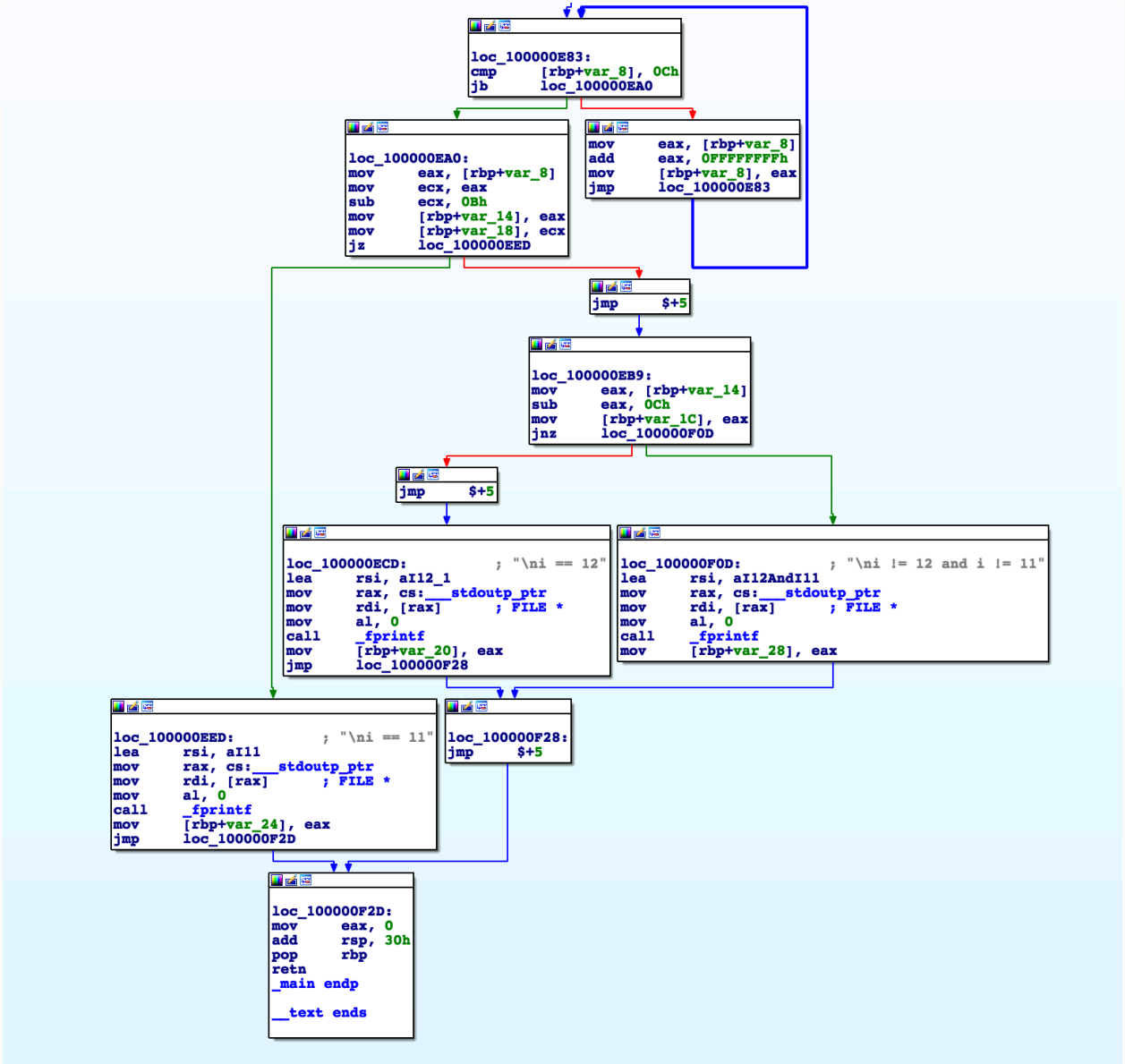
```
; Segment type: Pure code
__text segment para public 'CODE' use64
assume cs:__text
;org 100000E20h
assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing


; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public _main
_main proc near

var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     [rbp+var_4], 0
mov     [rbp+var_8], 0DEADBEEFh
cmp     [rbp+var_8], 0Ch
jbe     loc_100000E63
```

```
lea     rsi, aI12       ; "\ni > 12"
mov     rax, cs:___stdoutp_ptr
mov     rdi, [rax]      ; FILE *
mov     al, 0
call    _fprintf
mov     [rbp+var_C], eax
jmp     loc_100000E7E
```

```
loc_100000E63:          ; "\ni <= 12"
lea     rsi, aI12_0
mov     rax, cs:___stdoutp_ptr
mov     rdi, [rax]      ; FILE *
mov     al, 0
call    _fprintf
mov     [rbp+var_10], eax
```

```
loc_100000E7E:
jmp     $+5
```

Control Flow: Example Program (3)



Calling Conventions

Basics

- A calling convention is an implementation-level (low-level) scheme for how subroutines receive parameters from their caller and how they return a result.
- It specifies:
  - the order in which scalar parameters are allocated;
  - how parameters are passed (pushed on the stack, placed in registers, or a mix of both);
  - which registers the callee must preserve for the caller;
  - how the task of preparing the stack for, and restoring after, a function call is divided between the caller and the callee.

Calling convention `cdecl`

- Subroutine arguments are passed on the stack.
- Integer values and memory addresses are returned in the EAX register
- In context of the C programming language, function arguments are pushed on the stack in the reverse order.
- The caller cleans the stack after the function call returns.

Calling convention `cdecl` (2)

```c
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

```asm
caller:
    ; make new call frame
    push    ebp
    mov     ebp, esp
    ; push call arguments
    push    3
    push    2
    push    1
    ; call subroutine 'callee'
    call    callee
    ; remove arguments from frame
    add     esp, 12
    ; use subroutine result
    add     eax, 5
    ; restore old call frame
    pop     ebp
    ; return
    ret
```

Picture source: `https://en.wikipedia.org/wiki/X86_calling_conventions#cdecl`

Calling Convention `stdcall`

- `pascal`: the parameters are pushed on the stack in left-to-right order (opposite of `cdecl`), and the callee is responsible for balancing the stack before return.
- `stdcall`: variation on the `pascal` calling convention in which the callee is responsible for cleaning up the stack, but the parameters are pushed onto the stack in right-to-left order, as in the `cdecl` calling convention. Return values are stored in the EAX register.
- `stdcall` is the standard calling convention for the Microsoft Win32 API.

Calling Convention `fastcall/thiscall`

- Passes the first two arguments (evaluated left to right) that fit into `ECX` and `EDX`. Remaining arguments are pushed onto the stack from right to left.
- `thiscall` is used for calling C++ non-static member functions:
  - For the GCC compiler, `thiscall` is almost identical to `cdecl`: the caller cleans the stack, and the parameters are passed in right-to-left order. The difference is the addition of the `this` pointer, which is pushed onto the stack last, as if it were the first parameter in the function prototype.
  - On the Microsoft Visual C++ compiler, the `this` pointer is passed in `ECX` and it is the callee that cleans the stack, mirroring the `stdcall` convention used in C for this compiler and in Windows API functions.

Calling Convention Microsoft x64

- It uses registers RCX, RDX, R8, R9 for the first four integer or pointer arguments (in that order), and XMM0, XMM1, XMM2, XMM3 are used for floating point arguments.
- Additional arguments are pushed onto the stack (right to left).
- Integer return values (similar to x86) are returned in RAX if 64 bits or less. Floating point return values are returned in XMM0. Parameters less than 64 bits long are not zero extended; the high bits are not zeroed.

Calling Convention Microsoft x64 (2)

- It's the caller's responsibility to allocate 32 bytes of shadow space on the stack right before calling the function (regardless of the actual number of parameters used), and to pop the stack after the call. The shadow space is used to spill RCX, RDX, R8, and R9 but must be made available to all functions, even those with fewer than four parameters.
- For example, a function taking 5 integer arguments will take the first to fourth in registers, and the fifth will be pushed on the top of the shadow space. So when the called function is entered, the stack will be composed of (in ascending order) the return address, followed by the shadow space (32 bytes) followed by the fifth parameter.

System V AMD64 ABI

- This calling convention is followed on Solaris, Linux, FreeBSD, OS X, and other UNIX-like or POSIX-compliant operating systems.
- The first six integer or pointer arguments are passed in registers `RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `R9`, while `XMM0` to `XMM7` are used for floating point arguments.
- Additional arguments are passed on the stack and the return value is stored in `RAX`.
- A shadow space is not provided; on function entry, the return address is adjacent to the seventh integer argument on the stack.

System V AMD64 ABI Red Zone

- A red zone is a fixed-size area in a function's stack frame beyond the return address which is not preserved by that function.
- The callee function may use the red zone for storing local variables without the extra overhead of modifying the stack pointer.
- The x86-64 ABI used by System V mandates a 128-byte red zone which begins directly after the return address and includes the function's arguments.

Lab L2

- `http://crypto.junod.info/files/SU16/L2`
- *Hint*: basic tools are sufficient to break it!

Part IV

ARM Architecture

Objectives

At the end of this part, the student will be able to ...

- ... describe the main characteristics of the ARM architecture.
- ... name important registers of the ARM architecture.
- ... name the most important instructions of the ARM instruction set.
- ... recognize typical control-flow constructs in ARM assembler.

Basics

Basics of the ARM Architecture

- ARM originally stands from *Acorn RISC Machine*, then *Advanced RISC Machine*.
- 32-bit (ARMv1 to ARMv7) and 64-bit (ARMv8) RISC architecture
- A RISC-based architecture induces less costs, heat and power use.
- Many smartphones, tablets, laptops and embedded systems rely on the ARM architecture.
- Main business of ARM is to sell IP cores, i.e., they "*provide to all licensees an integratable hardware description of the ARM core as well as complete software development toolset (compiler, debugger, software development kit) and the right to sell manufactured silicon containing the ARM CPU.*" (source: Wikipedia).

ARM Architectures

Architecture	Core bit width	Cores designed by ARM Holdings	Cores designed by third parties	Profile	References
ARMv1	32[a 1]	ARM1			
ARMv2	32[a 1]	ARM2, ARM250, ARM3	Amber, STORM Open Soft Core[37]		
ARMv3	32[a 2]	ARM6, ARM7			
ARMv4	32[a 2]	ARM8	StrongARM, FA526		
ARMv4T	32[a 2]	ARM7TDMI, ARM9TDMI, SecurCore SC100			
ARMv5TE	32	ARM7EJ, ARM9E, ARM10E	XScale, FA626TE, Feroceon, PJ1/Mohawk		
ARMv6	32	ARM11			
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000		Microcontroller	
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300		Microcontroller	
ARMv7E-M	32	ARM Cortex-M4, ARM Cortex-M7		Microcontroller	
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8		Real-time	
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Krait, Scorpion, PJ4/Sheeva, Apple A6/A6X	Application	
ARMv8-A	32	ARM Cortex-A32		Application	
ARMv8-A	64	ARM Cortex-A35,[38] ARM Cortex-A53, ARM Cortex-A57,[39] ARM Cortex-A72[40]	X-Gene, Nvidia Project Denver, AMD K12, Apple A7/A8/A8X/A9/A9X, Cavium Thunder X,[41][42][43] Qualcomm Kryo	Application	[44][45]
ARMv8.1-A	64	TBA		Application	
ARMv8-R	32	TBA		Real-time	[46][47]
ARMv8-M	32	TBA		Microcontroller	[48]

Picture source: `https://en.wikipedia.org/wiki/ARM_architecture#Cores`

ARM Architecture Characteristics

- Load/store architecture
- Generally requires aligned memory accesses
- Uniform register file consisting of 16 32-bit registers
- Fixed instruction width (better decoding and pipelining capabilities, but decreased code density), however a 16-bit instruction set was later introduced (*Thumb*)
- Mostly single clock-cycle execution
- Conditional execution of most instructions is possible
- Arithmetic instructions alter condition codes only when asked
- 32-bit barrel shifter can be used with most arithmetic instructions and address computations
- etc.

ARM Modes

- **User mode**: non-privileged mode
- **FIQ mode**: privileged mode use to handle FIQ (*Fast Interrupt Request*) interrupts
- **IRQ mode**: privileged mode use to handle IRQs
- **Supervisor mode**: privileged mode entered whenever the CPU is reset or when an SVC instruction is executed
- + abort mode, undefined mode, system mode, monitor mode, hypervisor mode, etc.

Registers and Memory

Registers

- Registers `R0` through `R7` are the same across all CPU modes.
- Registers `R8` through `R12` are the same across all CPU modes, except FIQ mode, i.e., the FIQ mode has its own distinct `R8` through `R12` registers.
- Each mode that can be entered because of an exception has its own `R13` and `R14`. These registers generally contain the stack pointer and the return address from function calls, respectively.

Registers

- R13 is also referred to as SP (*Stack Pointer*).
- R14 is also referred to as LR (*Link Register*); it holds the address to return to when a function call completes.
- R15 is also referred to as PC (*Program Counter*).
- CPSR is a 32-bit register holding flags (*Current Program Status Register*)

Registers and CPU Modes

Registers across CPU modes

usr	sys	svc	abt	und	irq	fiq
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8_fiq
R9						R9_fiq
R10						R10_fiq
R11						R11_fiq
R12						R12_fiq
R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15						
CPSR						
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Picture source: https://en.wikipedia.org/wiki/ARM_architecture#Registers

Instructions

Conditional Execution

- Almost every ARM instruction has a conditional execution feature called *predication*, which is implemented with a 4-bit condition code selector (*predicate*).
- To allow for unconditional execution, one of the four-bit codes causes the instruction to be always executed.
- Most other CPU architectures only have condition codes on branch instructions (e.g., Intel's `Jcc`).

Conditional Execution / C

```c
while (i != j)  // We enter the loop when i<j or i>j, not when i==j
{
   if (i > j)   // When i>j we do that
       i -= j;
   else         // When i<j we do that (since i!=j is checked in while condition)
       j -= i;
}
```

Picture source: https://en.wikipedia.org/wiki/ARM_architecture#Conditional_execution

Conditional Execution / ARM-like C

```
do
{
   if       (i > j) // When i>j we do that
       i -= j;
   else if (i < j) // When i<j we do that
       j -= i;
   else
       ;           // When i==j we do nothing
}
while (i != j);    // We do nothing into the loop and leave the loop when i==j
```

Picture source: `https://en.wikipedia.org/wiki/ARM_architecture#Conditional_execution`

Conditional Execution / ASM

```
loop:   CMP    Ri, Rj       ; set condition "NE" if (i != j),
                            ;               "GT" if (i > j),
                            ;            or "LT" if (i < j)
        SUBGT  Ri, Ri, Rj   ; if "GT" (Greater Than), i = i-j;
        SUBLT  Rj, Rj, Ri   ; if "LT" (Less Than), j = j-i;
        BNE  loop           ; if "NE" (Not Equal), then loop
```

Picture source: `https://en.wikipedia.org/wiki/ARM_architecture#Conditional_execution`

Condition Codes

Table 5-1 Condition code suffixes and related flags

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.

Source: `http://infocenter.arm.com`

Use of the Barrel Shifter

On ARM, you can translate the C line

```
a += (j << 2);
```

in a single instruction:

```
ADD  Ra, Ra, Rj, LSL #2
```

Important Instructions

- `ADD` (`ADC`): add (with carry)
- `AND`: logical AND
- `B`: branch
- `BFC`/`BFI`/`BIC`: bit field clear / insert / bit clear
- `BL`/`BLX`: branch with link/branch with link, change instruction set
- `BX`: branch, change instruction set
- `CMP`: compare
- `EOR`: XOR

Important Instructions (2)

- `LDM`: load multiple registers
- `LDR`: load register with word
- `MLA`/`MLS`: multiply accumulate / multiply and substract
- `MOV`: move
- `MUL`: multiply
- `NOP`: no operation
- `ORN`: OR NOT
- `ORR`: OR
- `POP`/`PUSH`: pop/push from/to stack

Important Instructions (3)

- STM: store multiple registers
- STR: store register with word
- SUB: substract
- TST: test

Thumb/Thumb-2 Instruction Sets

- The Thumb instruction set is a compact 16-bit encoding for a subset of the ARM instruction set.
- Idea: sacrifce some functionalities while improving code density and save memory.
- Only branches can be conditional, and many opcodes are restricted to accessing only half of all of the CPU's general-purpose registers.
- Thumb-2 extends the limited 16-bit instruction set of Thumb with additional 32-bit instructions.
- Thumb-2 extends the Thumb instruction set with bit-field manipulation, table branches and conditional execution.

Other ARM Instruction Sets

- The NEON extension is a combined 64- and 128-bit SIMD instruction set that provides standardized acceleration for media and signal processing applications.
- Jazelle is a technique that allows Java bytecode to be executed directly in the ARM architecture as a third execution state (and instruction set) alongside the existing ARM and Thumb mode.

Procedure Call Standard

- The first four registers `R0`-`R3` are used to pass argument values into a subroutine and to return a result value from a function.
- Typically, the registers `R4`-`R8`, `R10` and `R11` are used to hold the values of a routine's local variables.
- A subroutine must preserve the contents of the registers `R4`-`R8`, `R10`, `R11` and `SP`.
- The stack is a contiguous area of memory that may be used for storage of local variables and for passing additional arguments to subroutines when there are insufficient argument registers available.

Subroutine Calls

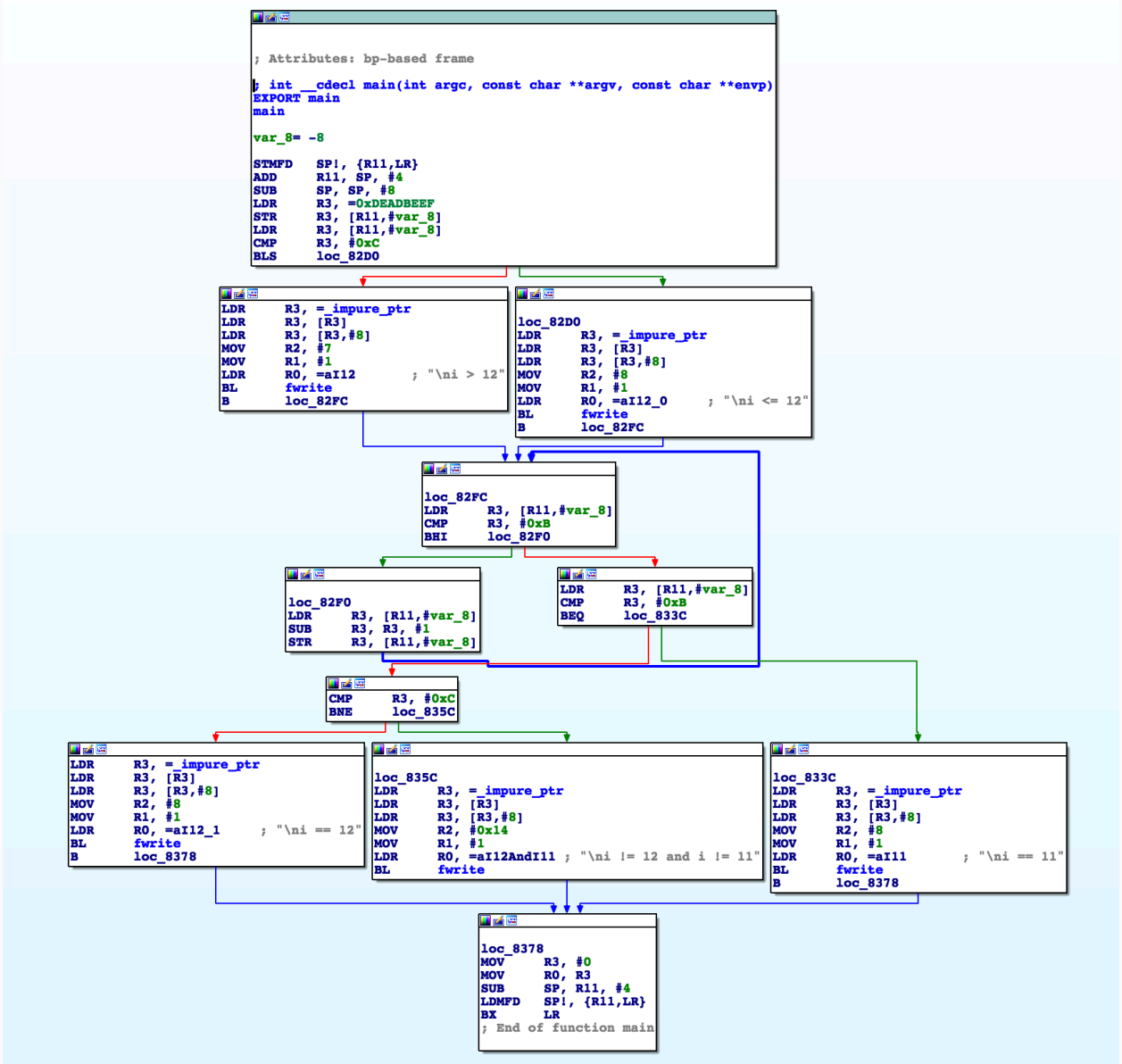
- Both the ARM and Thumb instruction sets contain a primitive subroutine call instruction, `BL`, which performs a branch-with-link operation.
- The effect of executing `BL` is to transfer the sequentially next value of the program counter (*the return address*) into the link register (`LR`) and the destination address into the program counter (`PC`).
- Bit 0 of the link register will be set to 1 if the `BL` instruction was executed from Thumb state, and to 0 otherwise.
- The result is to transfer control to the destination address, passing the return address in LR as an additional parameter to the called subroutine.
- Return values are returned in `R0`.

Control Flow

Control Flow: Example Program

```c
#include <stdio.h>

int main (void)
{
  unsigned int i = 0xDEADBEEFUL;

  if (i > 12) { fprintf (stdout, "\ni > 12"); }
   else { fprintf (stdout, "\ni <= 12"); }

  while (i >= 12)
    i--;

  switch (i) {
  case 12:
    fprintf (stdout, "\ni == 12"); break;
  case 11:
    fprintf (stdout, "\ni == 11"); goto label;
  default:
    fprintf (stdout, "\ni != 12 and i != 11");
  }

 label:
  return 0;
}
```

Control Flow: Example Program (2)



Lab L3

- `http://crypto.junod.info/files/SU16/L3`
- *Hint*: `arm-none-eabi-objdump` is sufficient to break it!

Part V

Interpreted Languages

Objectives

At the end of this part, the student will be able to ...

- ... understand the problematic of reverse engineering interpreted languages.
- ... name tools that allow to reverse engineer the most important interpreted languages.

Interpreted Languages

- According to Wikipedia, "*An interpreted language is a programming language for which most of its implementations execute instructions directly, without previously compiling a program into machine-language instructions. The interpreter executes the program directly, translating each statement into a sequence of one or more subroutines already compiled into machine code.*"
- Interpreted languages include Java, Python, Ruby, Perl, PHP, Postscript, Flash, etc.
- Typically, software engineering of interpreted languages is much easier than for native code.

Java and Derivates

Java Bytecode

- Java bytecode is the instruction set of the Java virtual machine.
- Each opcode is coded on 1 or 2 bytes plus 0 or more bytes for transmitted the parameters.
- Categories of opcodes:
  - Loading / storing
  - Arithmetic / Boolean
  - Type conversion
  - Object creation and manipulation
  - Operand stack management
  - Control transfer
  - Method invocation and return

Java Bytecode / Example (1)

```
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

Picture source: `https://en.wikipedia.org/wiki/Java_bytecode`

Java Bytecode / Example (2)

```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush  1000
6:   if_icmpge       44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge       31
16:  iload_1
17:  iload_2
18:  irem
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic       #84; // Field java/lang/System.out:Ljava/io/PrintStream;
34:  iload_1
35:  invokevirtual   #85; // Method java/io/PrintStream.println:(I)V
38:  iinc    1, 1
41:  goto    2
44:  return
```


Picture source: https://en.wikipedia.org/wiki/Java_bytecode

Java Bytecode Specifications

The full specifications of the Java bytecode are available at `https://docs.oracle.com/javase/specs/jvms/se8/html/index.html`.

Dalvik/ART Bytecode

- In the Android ecosystem, the Java bytecode is replaced by the Dalvik/ART bytecode.
- Main features:
  - the Dalvik VM uses a register-based architecture;
  - fewer, typically more complex, virtual machine instructions.
  - A tool called `dx` is used to convert Java `.class` files into the `.dex` format.
  - Java bytecode is converted into an alternative instruction set used by the Dalvik VM.
  - Standard Java bytecode executes 8-bit stack instructions. Local variables must be copied to or from the operand stack by separate instructions. Dalvik instead uses its own 16-bit instruction set that works directly on local variables.

Useful Tools

- `javap`: Java disassembler
- Java decompilers
  - CFR (`http://www.benf.org/other/cfr/`)
  - `jadx`: Dex to Java decompiler (`https://github.com/skylot/jadx`)
  - JDCore (`http://jd.benow.ca/`)
  - Procyon (`https://bitbucket.org/mstrobel/procyon`)
  - Fernflower (`https://github.com/fesh0r/fernflower`)
  - Cf. `http://www.javadecompilers.com/`

Python Bytecode

Python Bytecode

- CPython is the most widely used implementation of Python.
- CPython is a source code interpreter.
- Salient features:
  - Automatic memory management, reference counting
  - Execution model: bytecode interpretation by a stack-based virtual machine (VM)
  - Several data structures (maps, lists, tuples) are created and managed by the virtual machine
  - Multi-threaded, however only a single active interpreter thread at the same time
  - Late binding: search for method in class dictionnary by name only when needed for the first time.

Function Objects

- Functions are objects, like a list, a tuple or an instance of a class.
- Since functions are objects, it is possible to talk about without calling them: pass a function as a parameter, or assigning a function to another name.

```
def foo (a, b):
    return a+b
```

```
>>> foo
<function foo at 0x10eba0140>
>>> bar = foo
>>> bar
<function foo at 0x10eba0140>
>>> foo (1, 2)
3
>>> bar (1, 2)
3
```

Code Objects

There are several interesting attributes in a function object, including code objects:

```
>>> foo.func_code
<code object foo at 0x10eb90d30, file "<stdin>", line 1>
>>> foo.func_code.co_varnames
('a', 'b')
>>> foo.func_code.co_consts
(None,)
>>> foo.func_code.co_argcount
2
>>> foo.func_code.co_code
'|\x00\x00|\x01\x00\x17S'
```

Python Bytecode

- A code object contains Python bytecode. A list of possible bytecode instructions currently implemented by the Python compiler is available at `https://docs.python.org/3.5/library/dis.html#python-bytecode-instructions`
- It can be disassembled using the `dis` module.

```
>>> dis.dis (foo.func_code.co_code)
          0 LOAD_FAST                0 (0)
          3 LOAD_FAST                1 (1)
          6 BINARY_ADD
          7 RETURN_VALUE
```

Python Bytecode (2)

```
def foo (a, b):
    if a > b:
        while a > b:
            a -= 1
     elif a == b:
        a += 2
     else:
        while b < a:
            b -= 1
    return (a, b)
```

Python Bytecode (3)

```
>>> dis.dis(foo)
  2           0 LOAD_FAST                0 (a)
              3 LOAD_FAST                1 (b)
              6 COMPARE_OP               4 (>)
              9 POP_JUMP_IF_FALSE       44

  3          12 SETUP_LOOP              83 (to 98)
        >>   15 LOAD_FAST                0 (a)
             18 LOAD_FAST                1 (b)
             21 COMPARE_OP               4 (>)
             24 POP_JUMP_IF_FALSE       40
```

Python Bytecode (4)

```
  4           27 LOAD_FAST                0 (a)
              30 LOAD_CONST               1 (1)
              33 INPLACE_SUBTRACT
              34 STORE_FAST               0 (a)
              37 JUMP_ABSOLUTE           15
        >>    40 POP_BLOCK
              41 JUMP_FORWARD            54 (to 98)

  5     >>    44 LOAD_FAST                0 (a)
              47 LOAD_FAST                1 (b)
              50 COMPARE_OP               2 (==)
              53 POP_JUMP_IF_FALSE       69
[...]
```

Python Bytecode (5)

```
def bar (a):
    c = foo(a)
    return c + 3
```

Python Bytecode (6)

```
>>> dis.dis(bar)
  2           0 LOAD_GLOBAL              0 (foo)
              3 LOAD_FAST                0 (a)
              6 CALL_FUNCTION            1
              9 STORE_FAST               1 (c)

  3          12 LOAD_FAST                1 (c)
             15 LOAD_CONST               1 (3)
             18 BINARY_ADD
             19 RETURN_VALUE
```

Python Decompilers

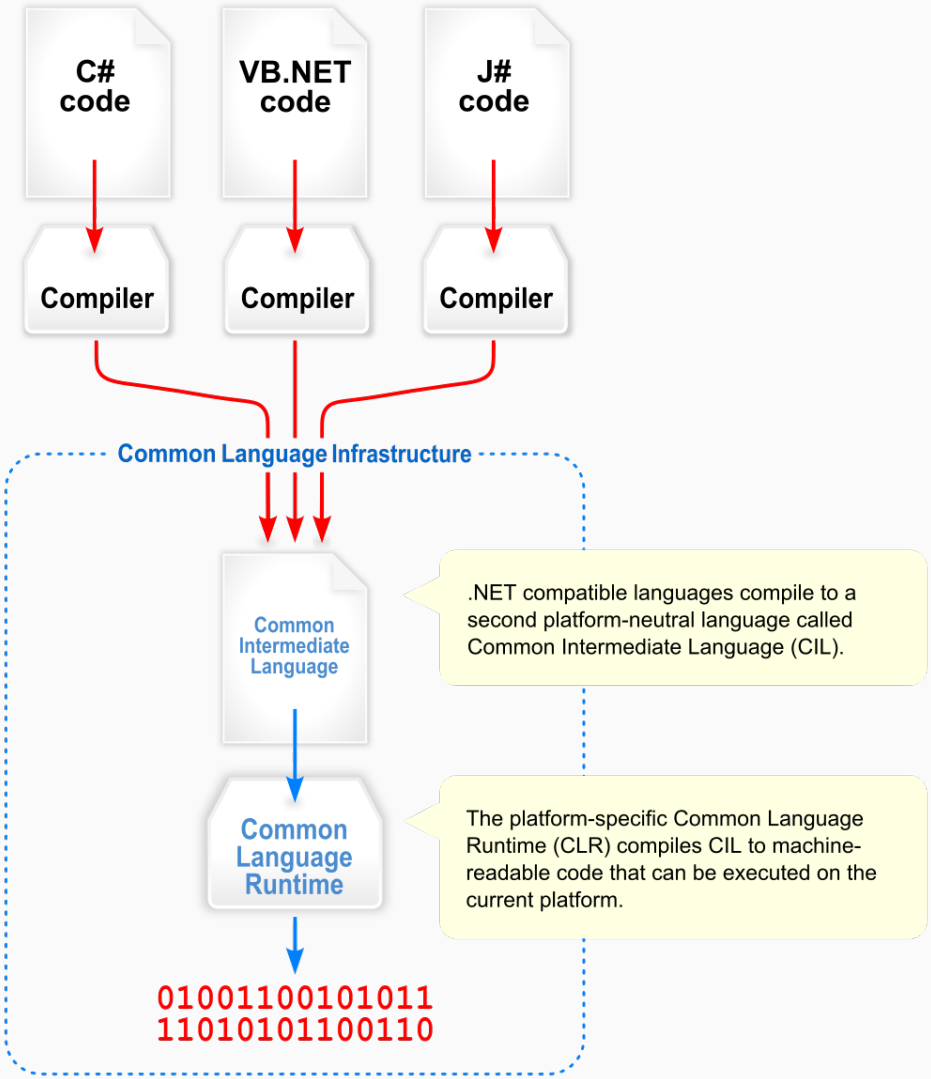
- `uncompyle2` is a (unmaintained) Python 2.7 decompiler written in Python (cf. `https://github.com/wibiti/uncompyle2`).
- `Decompyle++` is a Python disassembler/decompiler written in C++ supporting all versions of Python bytecode (cf. `https://github.com/zrax/pycdc`).

Microsoft .NET

Microsoft .NET

- The .NET Framework is a software framework developed by Microsoft that runs primarily on Microsoft Windows.
- Programs written for .NET Framework execute in a software environment, known as Common Language Runtime (CLR), an application virtual machine that provides services such as security, memory management and exception handling.

Microsoft .NET (2)



Picture source: `https://fr.wikipedia.org/wiki/Microsoft_.NET`

.NET Decompilers

- JetBrains dotPeek (`https://www.jetbrains.com/decompiler/`)
- ILSpy (`http://ilspy.net/`)
- Telerik JustDecompile (`http://www.telerik.com/products/decompiler.aspx`)
- ...

Lab L4

- `http://crypto.junod.info/files/SU16/L4`
- *Hint*: An `apk` file is just an archive.

Part VI

Structure and Life of an Executable

Objectives

At the end of this part, the student will be able to ...

- ... describe the important aspects of the structure of ELF, PE and Mach-O executables.
- ... explain how process loading and dynamic linking are implemented in practice on the Linux OS.

Executable Formats

Executable Formats

- According to Wikipedia, "*an executable file or executable program, or sometimes simply an executable, causes a computer "to perform indicated tasks according to encoded instructions," as opposed to a data file that must be parsed by a program to be meaningful. These instructions are traditionally machine code instructions for a physical CPU.*
- It is possible to extend the notion of executable to files containing bytecode targetting a software interpreter, or even to scripting languages.

Source: `https://en.wikipedia.org/wiki/Executable`

Executable Formats

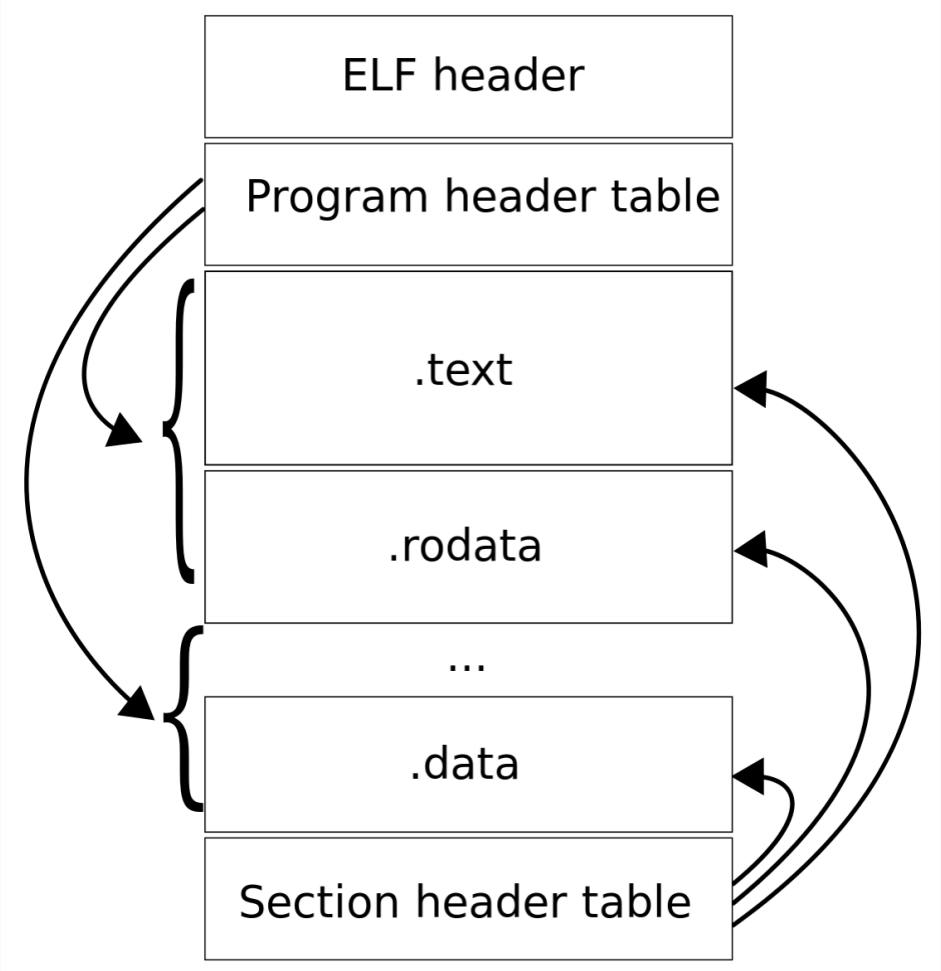
- Examples of well-known executable formats:
  - a.out, COFF, ELF: Unix-like OS
  - Mach-O: OS X, iOS
  - COM, MZ: DOS, old versions of Microsoft Windows
  - PE: recent versions of Microsoft Windows

Source: `https://en.wikipedia.org/wiki/Executable`

ELF Format Basics

- ELF stands for Executable and Linkable Format.
- File format supporting executables, object code, shared libraries and core files.
- High-level file layout:
  - ELF header;
  - Program header table, describing segments;
  - Section header table, describing sections;
  - Data referred to by entries from the program header table or the section header table.
- Segments contain information that is necessary for the runtime executation of the file.
- Sections contain information useful for linking and relocation.

ELF Format Basics (2)



Picture source: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

ELF Header (1)

Offset		Size (Bytes)		Field	Purpose
32-bit	64-bit	32-bit	64-bit		
0x00		4		`e_ident[EI_MAG0]` through `e_ident[EI_MAG3]`	`0x7F` followed by `ELF` (`45 4c 46`) in ASCII; these four bytes constitute the magic number.
0x04		1		`e_ident[EI_CLASS]`	This byte is set to either `1` or `2` to signify 32- or 64-bit format, respectively.
0x05		1		`e_ident[EI_DATA]`	This byte is set to either `1` or `2` to signify little or big endianness, respectively. This affects interpretation of multi-byte fields starting with offset `0x10` .
0x06		1		`e_ident[EI_VERSION]`	Set to `1` for the original version of ELF.
0x07		1		`e_ident[EI_OSABI]`	Identifies the target operating system ABI. Value / ABI 0x00 System V 0x01 HP-UX 0x02 NetBSD 0x03 Linux 0x06 Solaris 0x07 AIX 0x08 IRIX 0x09 FreeBSD 0x0C OpenBSD 0x0D OpenVMS It is often set to `0` regardless of the target platform.
0x08		1		`e_ident[EI_ABIVERSION]`	Further specifies the ABI version. Its interpretation depends on the target ABI. Linux kernel (after at least 2.6) has no definition of it.[5] In that case, offset and size of EI_PAD are `8` .
0x09		7		`e_ident[EI_PAD]`	currently unused
0x10		2		`e_type`	`1` , `2` , `3` , `4` specify whether the object is relocatable, executable, shared, or core, respectively.

Picture source: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

ELF Header (2)

0x12		2		`e_machine`	Specifies target instruction set architecture. Some examples are: **Value** / **ISA** 0x00 — No specific instruction set 0x02 — SPARC 0x03 — x86 0x08 — MIPS 0x14 — PowerPC 0x28 — ARM 0x2A — SuperH 0x32 — IA-64 0x3E — x86-64 0xB7 — AArch64
0x14		4		`e_version`	Set to `1` for the original version of ELF.
0x18		4	8	`e_entry`	This is the memory address of the entry point from where the process starts executing. This field is either 32 or 64 bits long depending on the format defined earlier.
0x1C	0x20	4	8	`e_phoff`	Points to the start of the program header table. It usually follows the file header immediately making the offset `0x40` for 64-bit ELF executables.
0x20	0x28	4	8	`e_shoff`	Points to the start of the section header table.
0x24	0x30	4		`e_flags`	Interpretation of this field depends on the target architecture.
0x28	0x34	2		`e_ehsize`	Contains the size of this header, normally 64 bytes for 64-bit and 52 for 32-bit format.
0x2A	0x36	2		`e_phentsize`	Contains the size of a program header table entry.
0x2C	0x38	2		`e_phnum`	Contains the number of entries in the program header table.
0x2E	0x3A	2		`e_shentsize`	Contains the size of a section header table entry.
0x30	0x3C	2		`e_shnum`	Contains the number of entries in the section header table.
0x32	0x3E	2		`e_shstrndx`	Contains index of the section header table entry that contains the section names.

Picture source: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

ELF Program Header

<table>
<tr><th>Offset</th><th>Size (Bytes)</th><th>Field</th><th>Purpose</th></tr>
<tr><td>0x00</td><td>4</td><td>p_type</td><td>Identifies the type of the segment.

<table><tr><th>Value</th><th>Name</th></tr><tr><td>0x00000000</td><td>PT_NULL</td></tr><tr><td>0x00000001</td><td>PT_LOAD</td></tr><tr><td>0x00000002</td><td>PT_DYNAMIC</td></tr><tr><td>0x00000003</td><td>PT_INTERP</td></tr><tr><td>0x00000004</td><td>PT_NOTE</td></tr><tr><td>0x00000005</td><td>PT_SHLIB</td></tr><tr><td>0x00000006</td><td>PT_PHDR</td></tr><tr><td>0x60000000</td><td>PT_LOOS</td></tr><tr><td>0x6FFFFFFF</td><td>PT_HIOS</td></tr><tr><td>0x70000000</td><td>PT_LOPROC</td></tr><tr><td>0x7FFFFFFF</td><td>PT_HIPROC</td></tr></table>
PT_LOOS to PT_HIOS (PT_LOPROC to PT_HIPROC) is an inclusive reserved ranges for operating system (processor) specific semantics.</td></tr>
<tr><td>0x04</td><td>4</td><td>p_offset</td><td>Offset of the segment in the file image.</td></tr>
<tr><td>0x08</td><td>4</td><td>p_vaddr</td><td>Virtual address of the segment in memory.</td></tr>
<tr><td>0x0C</td><td>4</td><td>p_paddr</td><td>On systems where physical address is relevant, reserved for segment's physical address.</td></tr>
<tr><td>0x10</td><td>4</td><td>p_filesz</td><td>Size in bytes of the segment in the file image. May be 0.</td></tr>
<tr><td>0x14</td><td>4</td><td>p_memsz</td><td>Size in bytes of the segment in memory. May be 0.</td></tr>
<tr><td>0x18</td><td>4</td><td>p_flags</td><td>Segment-dependent flags.</td></tr>
<tr><td>0x1C</td><td>4</td><td>p_align</td><td>0 and 1 specifies no alignment. Otherwise should be a positive, integral power of 2, with p_vaddr equating p_offset modulus p_align.</td></tr>
</table>

Picture source: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

ELF Section Header

Offset	Size (Bytes)	Field Name	Purpose
0x00	4	Name	An offset to a string in the **.shstrtab** section that represents the name of this section
0x04	4	Type	Identifies the type of this header. Some common examples include: **Value** / **Name** 0x00000000 NULL 0x00000001 PROGBITS 0x00000002 SYMTAB 0x00000003 STRTAB
0x08	4	Flags	
0x12	4	Address	
0x10	4	Offset	
0x14	4	Size	
0x18 - 0x28			Various other info found in the **readelf** command such as "ES", "Lk", "Inf" and "Al"

Picture source: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

A `readelf` Example

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4027e0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          107352 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
[...]
```

Sections and Segments

- The ELF formats defines two different interpretations:
  - one that targets the linking process;
  - one that targets the execution process.
- Sections refer to object code waiting to be linked.
- One or more sections map to a segment in the executable.

Symbols and Relocations

- Symbol tables are mappings of strings to locations into the ELF file.
- Symbols are required when linking: when the linker is looking for resolving a symbol `abc`, it will look for it into the symbol tables.
- A relocation is some blank space that needs to be filled later (typically, at execution time).

Program Headers

- Program headers describe to the operating system the information required to load the binary into memory and execute it, e.g.:
  - `PT_INTERP`: a string pointer to an interpreter for the binary file (typically the dynamic loader).
  - `PT_LOAD`: specifies the segment location into the ELF file, its base virtual address when living, as well as the initial permissions of the segment.

A `readelf` Example (2)

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R E    8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000183dc 0x00000000000183dc  R E    200000
  LOAD           0x0000000000019000 0x0000000000619000 0x0000000000619000
                 0x0000000000001240 0x0000000000001f60  RW     200000
  DYNAMIC        0x0000000000019a88 0x0000000000619a88 0x0000000000619a88
                 0x00000000000001d0 0x00000000000001d0  RW     8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000015d48 0x0000000000415d48 0x0000000000415d48
                 0x00000000000006b4 0x00000000000006b4  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8
```

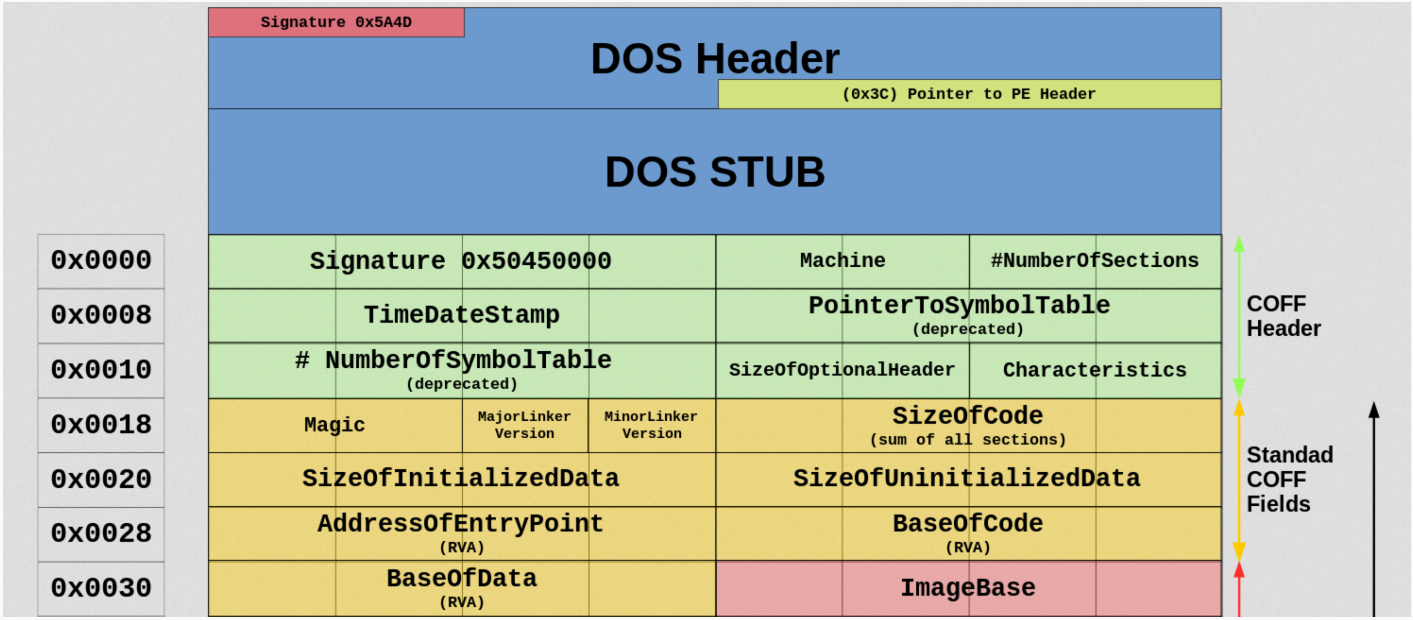
A `readelf` Example (3)

```
Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym
          .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
          .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .ctors .dtors .jcr .data.rel.ro .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
```

PE Format Basics

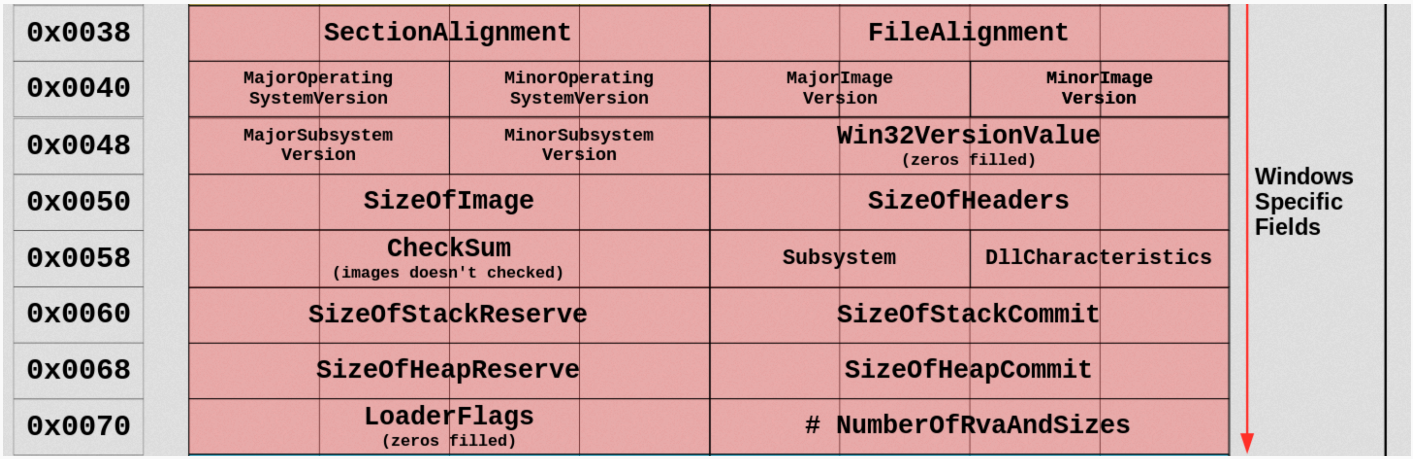
- PE stands for Portable Executable.
- The PE format is used in the Microsoft Windows ecosystem.
- It is a modified version of the Unix COFF (Common Object File Format) format.

PE Format (1)



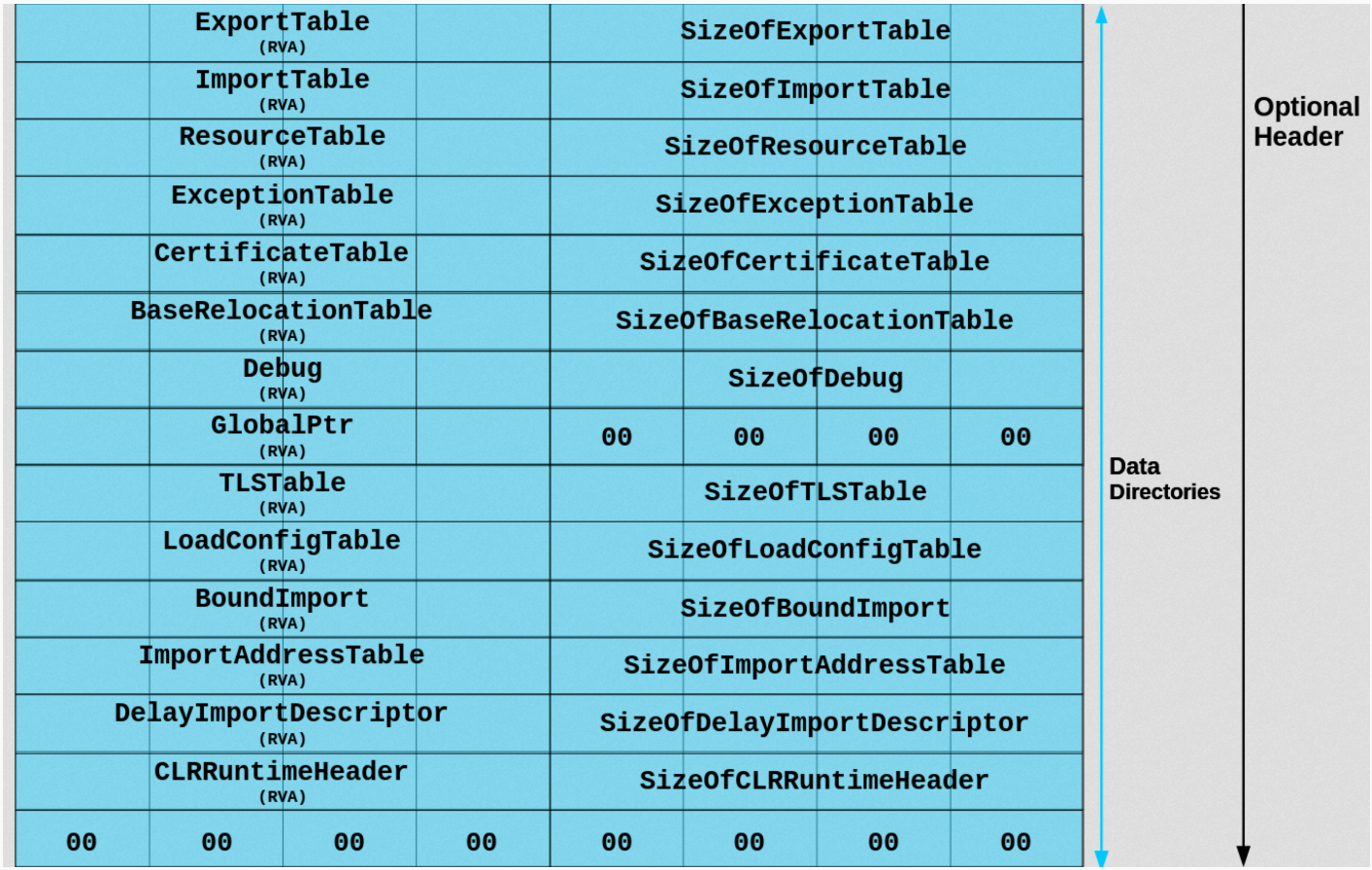
Picture source: `https://en.wikipedia.org/wiki/Portable_Executable`

PE Format (2)



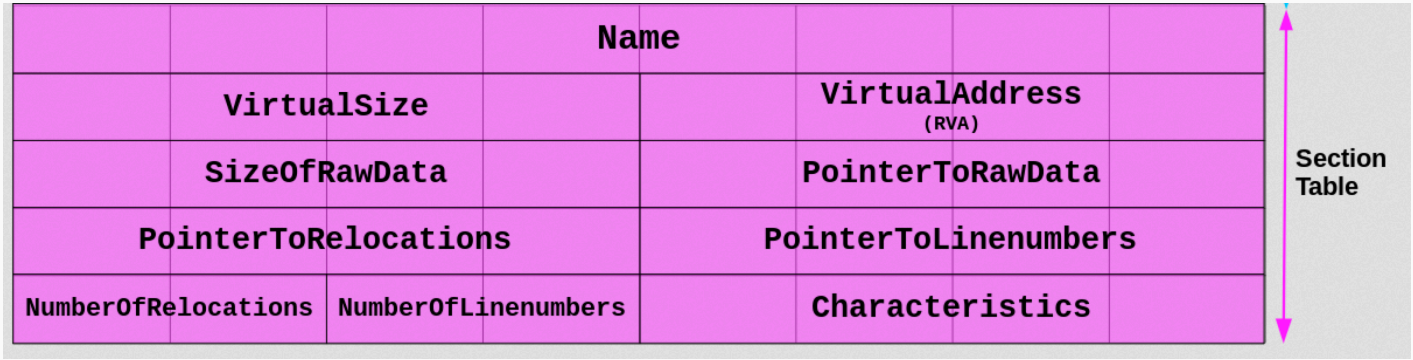
Picture source: `https://en.wikipedia.org/wiki/Portable_Executable`

PE Format (3)



Picture source: `https://en.wikipedia.org/wiki/Portable_Executable`

PE Format (4)



Picture source: `https://en.wikipedia.org/wiki/Portable_Executable`

Mach-O Format Basics

- Mach-O stands for Mach Object.
- Like ELF and PE, it is a format for executables, object codes, shared libraries and core dumps.
- Used in most systems based on the Mach kernel, including OS X and iOS.

Mach-O Format Basics (2)

- Made of one Mach-O header, followed by a sequences of load commands, followed by one or more segments, each of which contains between 0 and 255 sections.
- Supports multi-architecture binaries, i.e., a single binary file will store binary code for several hardware architectures.
- For instance, an iOS binary can contain code for the ARMv6, ARMv7, ARMv7s, ARMv8, x86 and x86-64 architectures.

Process Loading under Linux

Process Loading

- Under Linux, a process is loaded using the `exec` system call.
- After having allocated internal kernel structures, the kernel reads the specified ELF file.
- If the ELF file has a specified interpreter field (`PT_INTERP`), then it is executed. For executables using dynamic libraries, this program is the dynamic linker `ld.so` which will load the required libraries.

Process Loading (2)

- The kernel communicates some information to the launched process, such as the arguments of the program, environment variables, etc, by copying them onto the process' stack.
- The kernal add a virtual dynamic library, named `linux-gate.so.1`, to implement fast system calls, if the underlying hardware allows it.

Process Loading (3)

- The kernel then gives the control to the entry point of the interpreter (the dynamic linker).
- Once the dynamic linker has performed its duty, it will give the control to the entry point as specified in the ELF executable (typically the `_start` symbol).
- Then, the routine `__libc_start_main` is called, which will setting up certain C library mechanisms.

Process Loading (4)

- Then, the control is transfered to the `init` code, that will execute the global constructors stored into the `.ctors` section.
- Eventually, the `main` function is called.
- When the `main` function completes, `__libc_start_main` transfers the control to the `fini` code, that will execute the global destructors stored into the `.dtors` section.

Global Constructors and Destructors

```c
#include <stdio.h>

void __attribute__((constructor)) program_init(void)
{
      printf("\ninit\n");
}

void  __attribute__((destructor)) program_fini(void)
{
    printf("\nfini\n");
}

 int main(void)
{
    printf("\nmain\n");

   return 0;
}
```

Dynamic Linking under Linux

Dynamic Linking Basics

- The goal of dynamic linking is to save code space by sharing common (read-only) code between processes.
- The virtual memory mechanism allows to map shared code to the same physical memory page.
- Shared code is contained in dynamic libraries files (file extension is `.so` under Linux, `.dll` under Microsoft Windows and `.dylib` under OS X).
- Dynamic libraries have code and data sections, like executables.

At Compilation Time

- By default, programs use dynamic linking, unless compiled with the `-static` compilation flag.
- The compiler only needs to know the function prototypes.
- The (static) linker leaves information about which dynamic libraries have to be loaded in the `dynamic` section.

Required Dynamic Libraries

```
-bash-4.1$ readelf --dynamic /bin/ls

Dynamic section at offset 0x19a88 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libselinux.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [librt.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libcap.so.2]
 0x0000000000000001 (NEEDED)             Shared library: [libacl.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Required Dynamic Libraries (2)

```
-bash-4.1$ ldd /bin/ls
        linux-vdso.so.1 =>  (0x00007fffb7718000)
        libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f6083a13000)
        librt.so.1 => /lib64/librt.so.1 (0x00007f608380b000)
        libcap.so.2 => /lib64/libcap.so.2 (0x00007f6083606000)
        libacl.so.1 => /lib64/libacl.so.1 (0x00007f60833fe000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f608306a000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007f6082e65000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f6083c3a000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f6082c48000)
        libattr.so.1 => /lib64/libattr.so.1 (0x00007f6082a43000)
```

Dynamic Linker

- The dynamic linker manages shared dynamic libraries on behalf of the executable.
- The main job of the dynamic linker consists in resolving the missing addresses in the executable at run time.
- A relocation is a hint about the fact that a particular address needs to be “fixed” at runtime.
- Before running the code, the dynamic linker needs to go through all relocations and fix the refered addresses.
- Many different types of relocations exist and they are ABI-dependent.

Example of Relocation

```
$ cat a.c
   extern int x[100];
   int *y = x + 57;
$ cat b.c
   int x[100];
$ gcc -nostdlib -shared -fpic -s -o b.so b.c
$ gcc -nostdlib -shared -fpic -o a.so a.c ./b.so
```

Example of Relocation (2)

```
-bash-4.1$ readelf -r a.so

Relocation section '.rela.dyn' at offset 0x2c0 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000201000  000200000001 R_X86_64_64       0000000000000000 x + e4
```

Global Offset Table

- A shared library can be loaded at any address in a process space. So how does know the executable how to access to a global variable stored in it?
- A level of indirection is helpful here: it is called the Global Offset Table (GOT).
- When a dynamic library is loaded, it will find a relocation for the symbol that will write the symbol's address into the process' GOT.

Procedure Linkage Table

- A shared library can be loaded at any address in a process space. So how does know the executable how to access to some routine stored in it?
- The program does not call the external routine in a direct way, but through a Procedure Linkage Table (PLT) stub.

Lazy Binding

- When the dynamic linker loads a shared library, it will put an identifier and a resolution function at predetermined locations in the GOT.
- On the first call of the function, it falls into the default stub, which loads the identifier and calls the dynamic linker.
- The dynamic linker will patch the proper function address into the GOT, such that next time the original PLT entry will be called, the correct function address will be present.