Introduction to Reverse Engineering

Phish 'n' Chips Team

Outline

- What is Reverse Engineering?
- Interpreted vs. Compiled Languages
- Executable file formats
- Memory regions
- X86 crash course

What is Reverse Engineering?

Reverse engineering is the process of determining how a product is constructed by analyzing it, *without having access to the original blueprints*

Reverse Engineering an Executable

In the computer science domain, reverse engineering is reconstructing an high level representation *(source code)* of a program from its optimized form *(assembly)*

From Source Code to Executable

Source code is not directly executed by the CPU. It has to be *transformed* first

Compiled Languages

Source code is transformed to an executable **run directly** by the target operating system by a *compiler*. This means the program is written in some *assembly* language

A few examples:

- C / C++
- Rust

Interpreted Languages

Source code is transformed into an intermediate representation executed by an **interpreter**. This representation is usually called *bytecode*

A few examples:

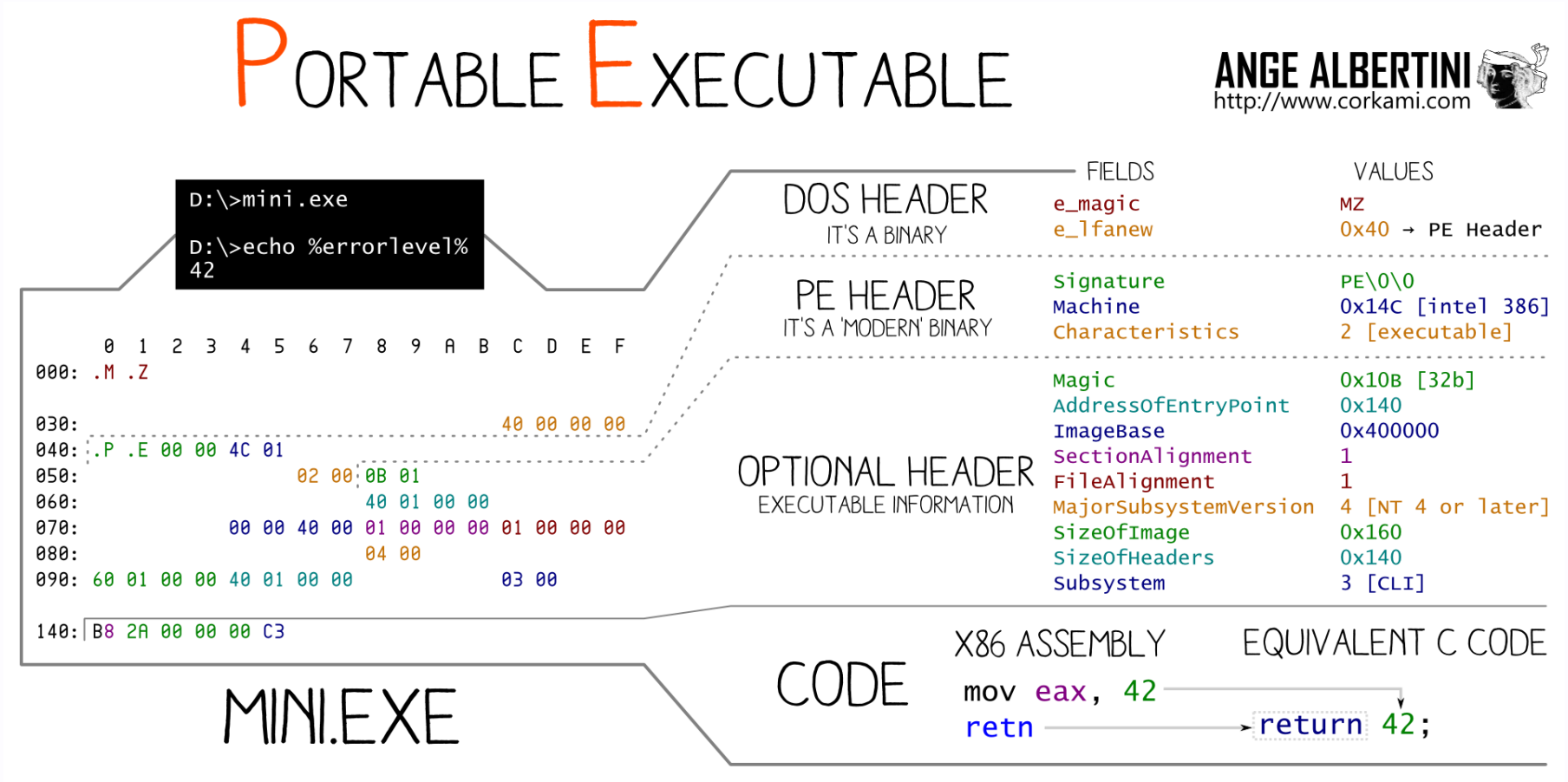
- Python
- Ruby

Executable Formats

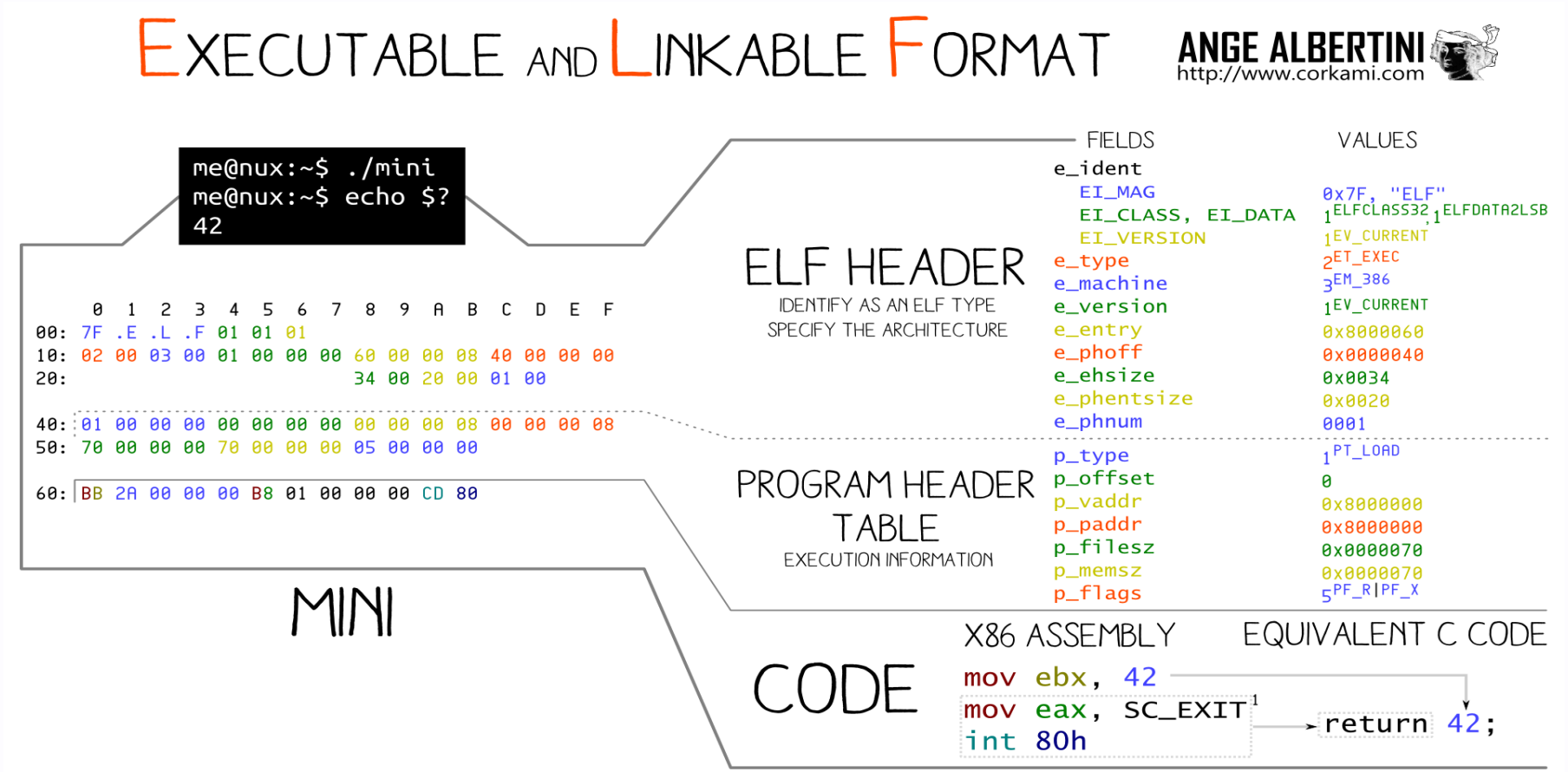
Each operating system executes programs in different ways. A program has to be "packaged" for a specific OS before being executed

The way a program is prepared is called **executable format**

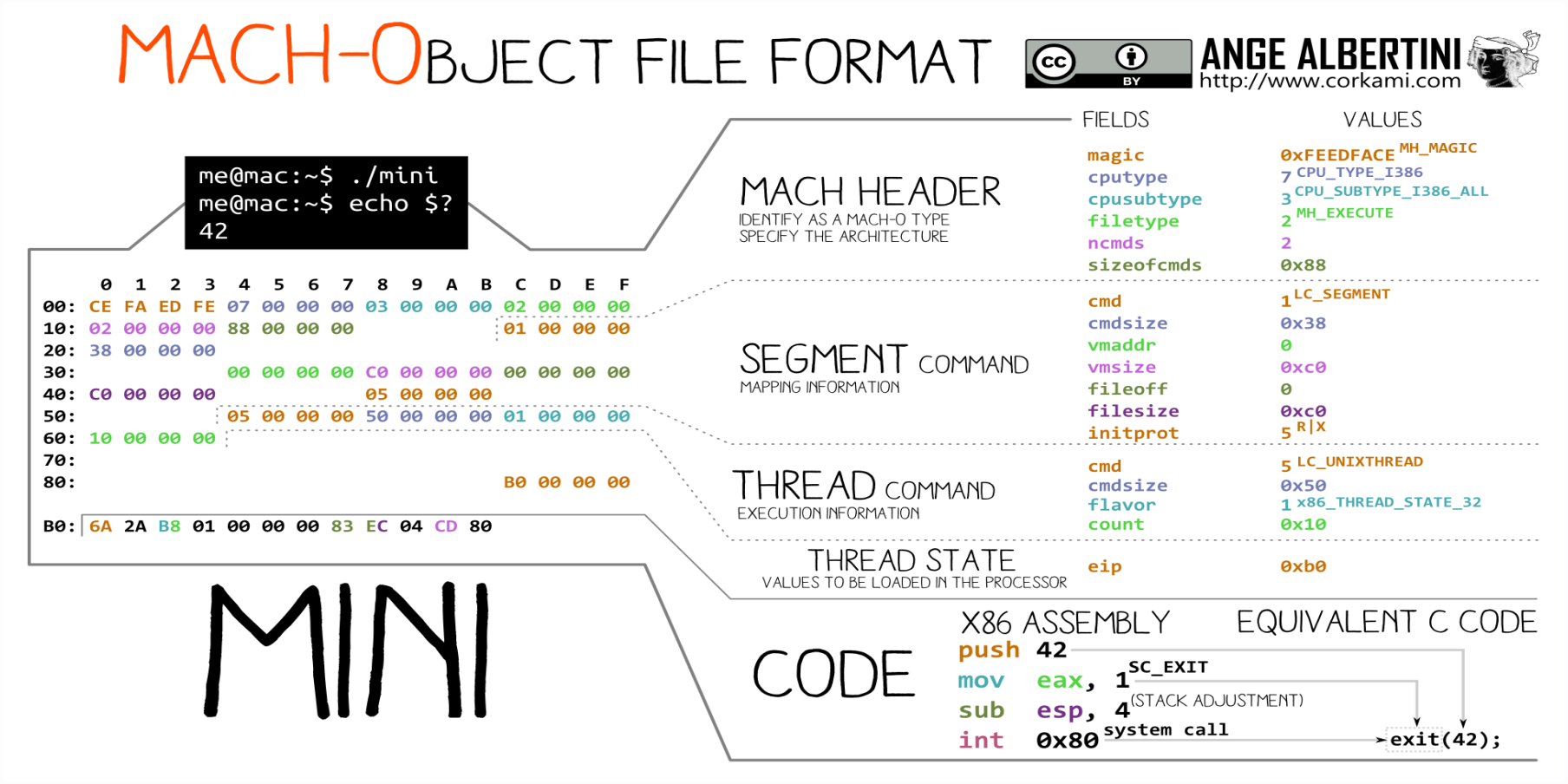
Windows: Portable Executable (PE)



Linux: Executable and Linkable Format (ELF)



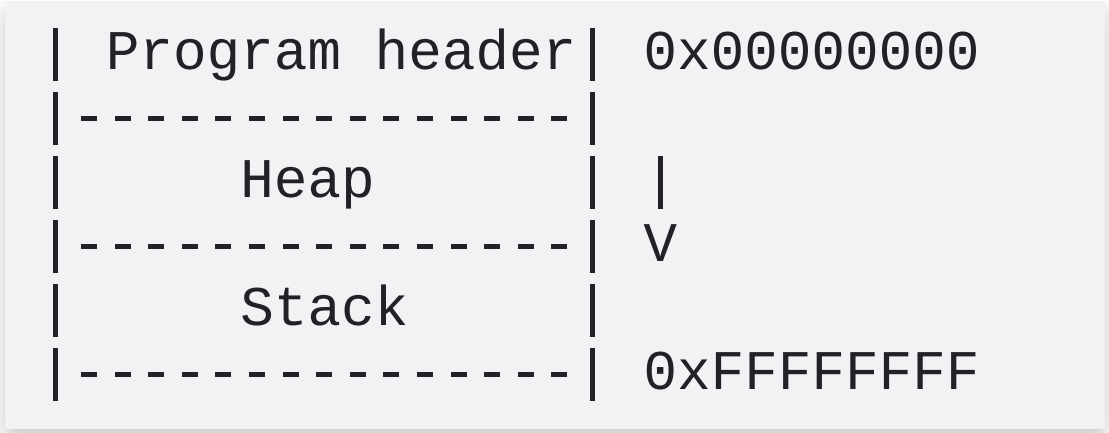
MacOS: Mach Object File Format (Mach-O)



Memory regions

An executable usually has three memory regions.

The **program header**, the **heap** and the **stack**.



The Program Header

The program header contains various information on how the OS has to load the executable. The locations of different memory segments are defined here

It includes the specification of the *base address*, the *entrypoint*, the *import table* and more

The Heap

The heap is where data structures generated **at runtime** are stored. This memory region expands and shrinks as the developer requests/releases memory

The Stack

The stack is a *LIFO (last-in first-out)* memory region. It's used by the program to store runtime information and it's crucial for function invocations

Pushing and Popping on the Stack

The data on the stack may be handled with two operations: *push* and *pop*

Push

```
Pushing A, then B

|     Empty      |
|----------------|
|        B       |
|----------------|
|        A       |
|----------------|
```

Pop

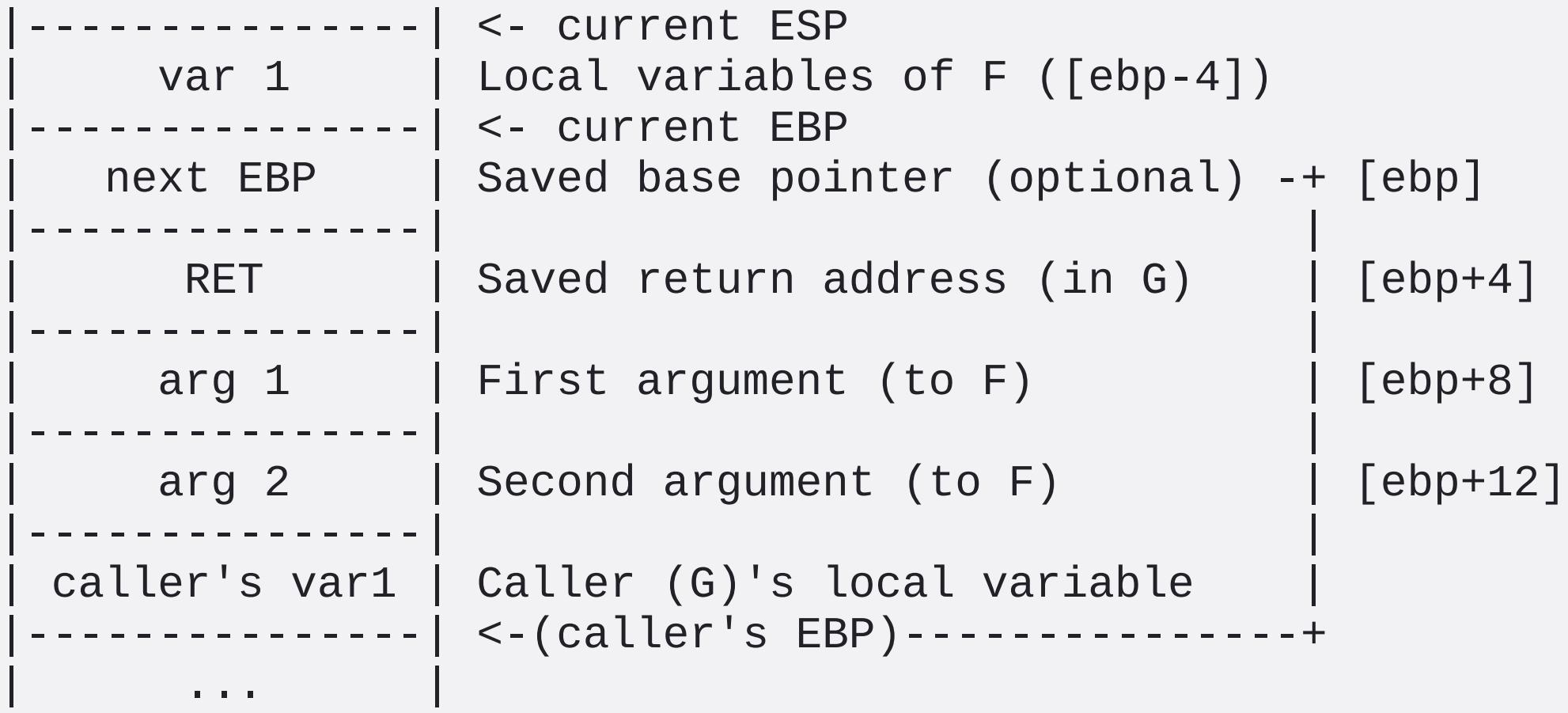
```
To get A, we must pop B first

|     Empty      |
|----------------|
|     Empty      |
|----------------|
|        A       |
|----------------|
```

Stack frame

A stack frame contains information about the active function in the program. It stores the *arguments*, *local variables* and saved registers to keep track of the execution

Structure of the stack frame



The `x86` Instruction Set

The `x86` instruction set is one of the most used CPU architectures and it belongs to the *CISC (Complex Instruction Set Computer)* family

Some history of the `x86` family

The `x86` is pretty old. It started as a 16-bit architecture with Intel shipping the `8086` CPU

Later it evolved to support 32-bit and, finally, 64-bit instructions

The Syntax War™: AT&T vs Intel

The instructions may be disassembled following two different syntax: **AT&T** (the wrong one) and **Intel**

Syntax Examples

The main difference is that the AT&T syntax puts the source operand *first* while the Intel one puts the destination operand *first*

AT&T

```
operation source, destination

add       $0x8  , %rsp
```

Intel

```
operation destination, source

add       rsp         , 0x8
```

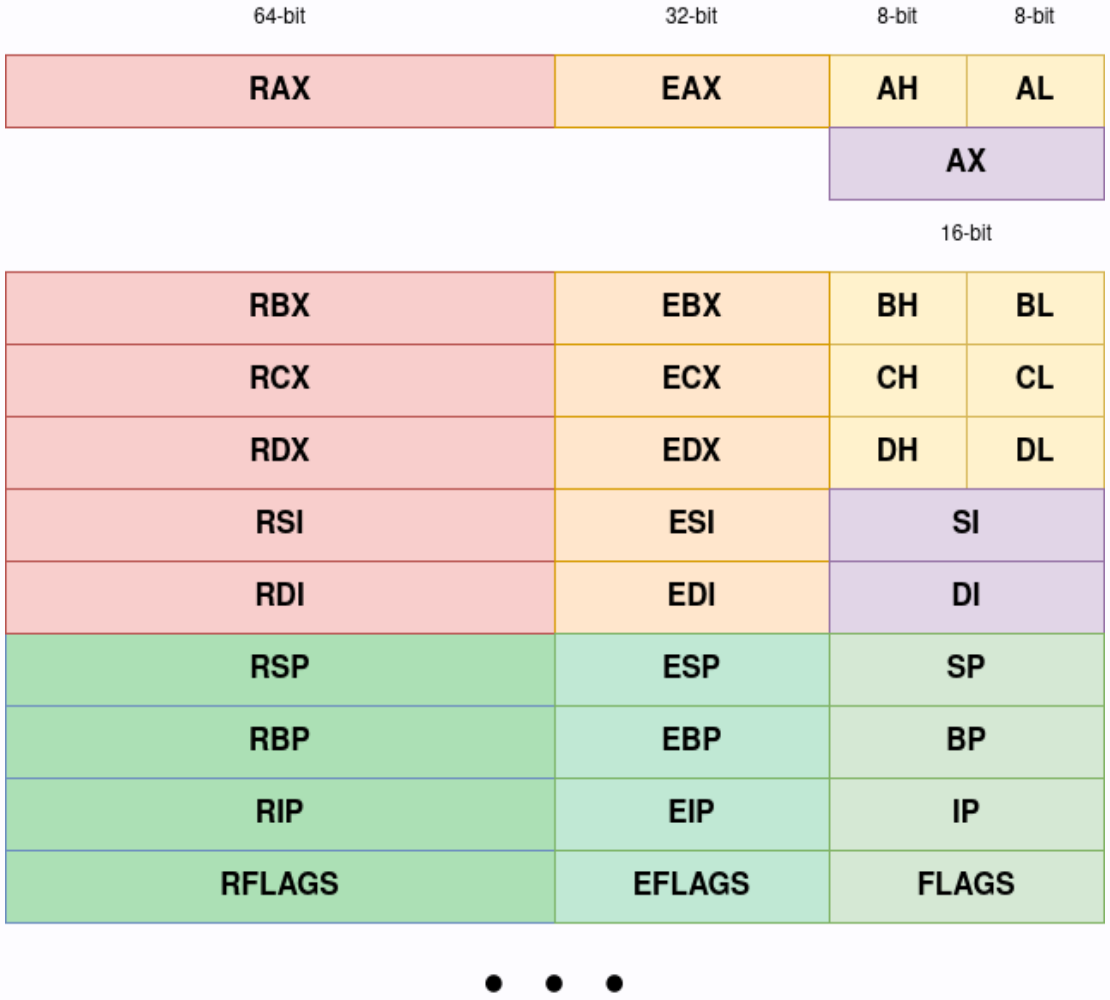
Anatomy of an instruction

An instruction is composed by an *operation* and zero or more *operands*

We refer to operations through *mnemonics*

The operands may be numbers, registers or memory locations

Registers



• • •

`RSP`, `RBP`, `RIP` and `RFLAGS` are specialized registers

The `x86` instruction set has several more registers!

Instruction Formats

Since `x86` is a CISC architecture, it has instruction variations that permit operations between:

- Register to Register
- Register to Memory
- Memory to Register
- Immediate to Register
- Immediate to Memory

Addressing Modes

An addressing mode provides a way to *calculate an address* the instruction has to operate on

Addressing Examples

- Register `mov rax, [rax]`
- Register + Offset `mov rax, [rax + 0x10]`

Common operations

`x86` has a huge set of instructions

The instructions range from moving values from a register to another to calculating a round of the AES keyschedule!

Some examples

- Add two registers `add rax, rbx`
- Move a register into another `mov rax, rbx`
- Move the *memory contents* pointed by a register into a register `mov rax, [rbx]`
- Do nothing `nop`
- Compare two values `cmp rax, rbx`
- Jump if result is zero `jz 0xb00bb00b`

Calling Conventions

A calling convention is a scheme that defines how functions receive their parameters and return their result to callers

Different *instruction set architectures (ISA)* use different calling conventions

`x86` Calling Convention (32-bit)

Parameters are passed **on the stack** in reverse order.
The result is stored in register `EAX`.

`x86_64` Calling Convention (64-bit)

Parameters are passed in **registers** and, if more are needed, on the stack. The result is stored in register

`RAX`

The order in which registers are read is:

`RDI, RSI, RDX, RCX, R8, R9`

That's it!

Questions?

Modifications by:

- 2020: Giulio De Pasquale (@peperunas)