# RE - 102

x86-64 (Windows)

# Disclaimers

Content, views, anything expressed are strictly my own and not that of my employer.

# Purpose

A CPU can understand machine code. Programmers can’t. And here we are!

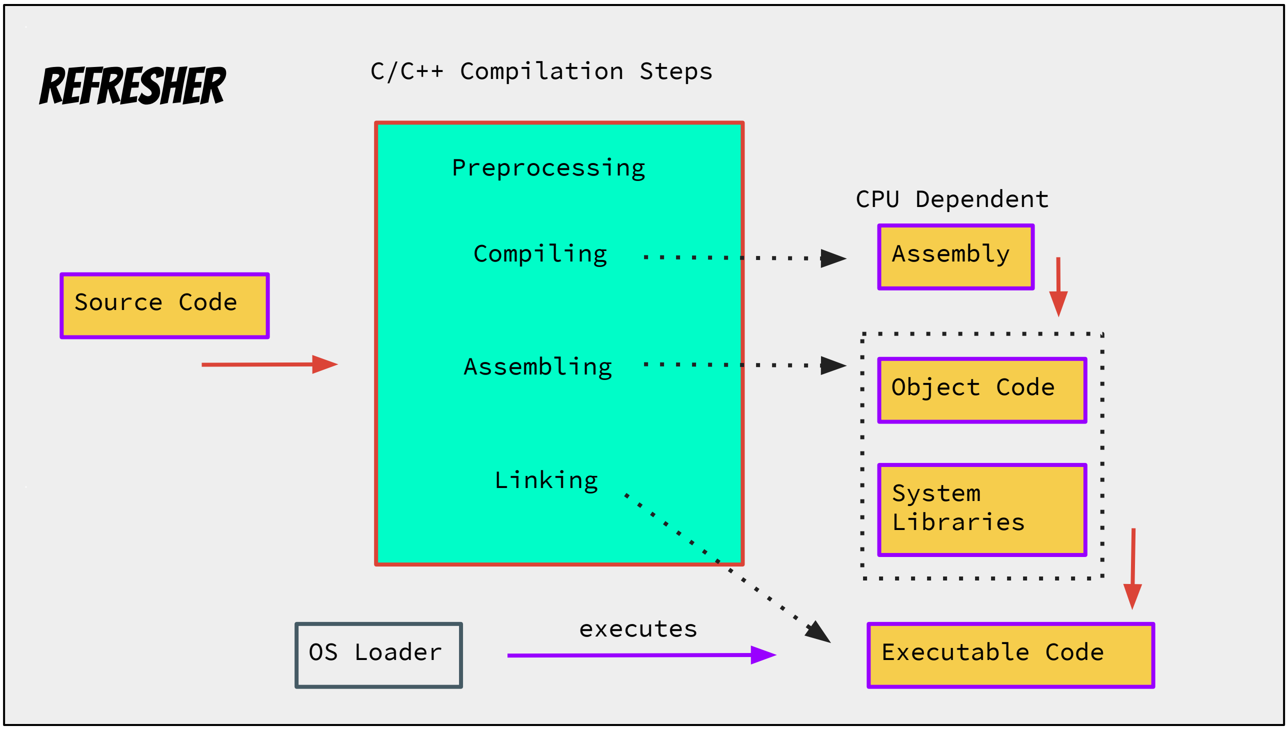
A better appreciation for Computers.

Curiosity :)

WHEN I STARTED PROGRAMMING, WE DIDN'T HAVE ANY OF THESE SISSY "ICONS" AND "WINDOWS."
© 1992 United Feature Syndicate, Inc.
S. Adams
9-8
ALL WE HAD WERE ZEROS AND ONES -- AND SOMETIMES WE DIDN'T EVEN HAVE ONES.
I WROTE AN ENTIRE DATABASE PROGRAM USING ONLY ZEROS.
YOU HAD ZEROS? WE HAD TO USE THE LETTER "O."

# Contents

- Lab/Tools Prep
- Refreshers
  - Compilation steps
  - A Process in Memory
- x64 Assembly
  - Registers
  - Stack
  - Instructions
- Exercise



REFRESHER

C/C++ Compilation Steps

Source Code

Preprocessing

Compiling

Assembling

Linking

CPU Dependent

Assembly

Object Code

System Libraries

OS Loader

executes

Executable Code

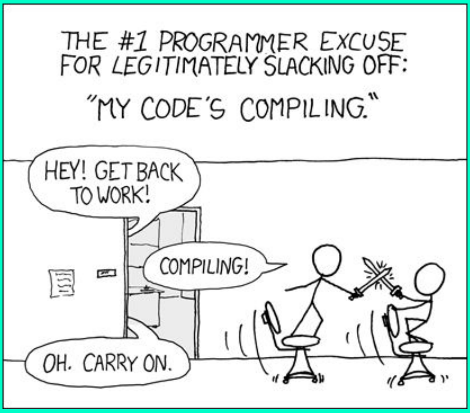
What is assembly?

- Consists of processor instructions. Processor being the CPU. These are translated by the assembler into machine language (Instruction Set Architecture) instructions that can be loaded into memory and executed.
- Textual representation of the binary instructions

Source code is compiled - purpose of the compiler to create an executable program from a high level language.

https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Compiled_Programs

# Demo



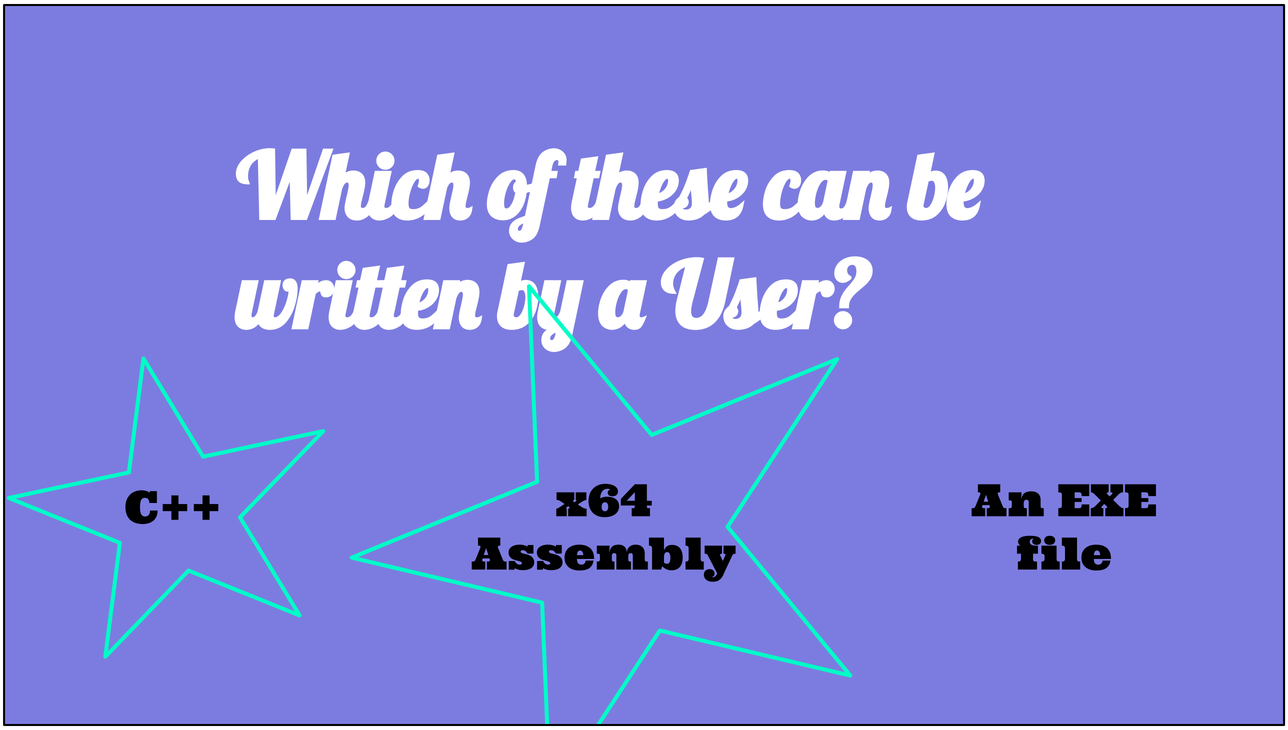
Godbolt - https://godbolt.org/z/Gj8Wrh

# Which of these can be written by a User?

C++

x64 Assembly

An EXE file

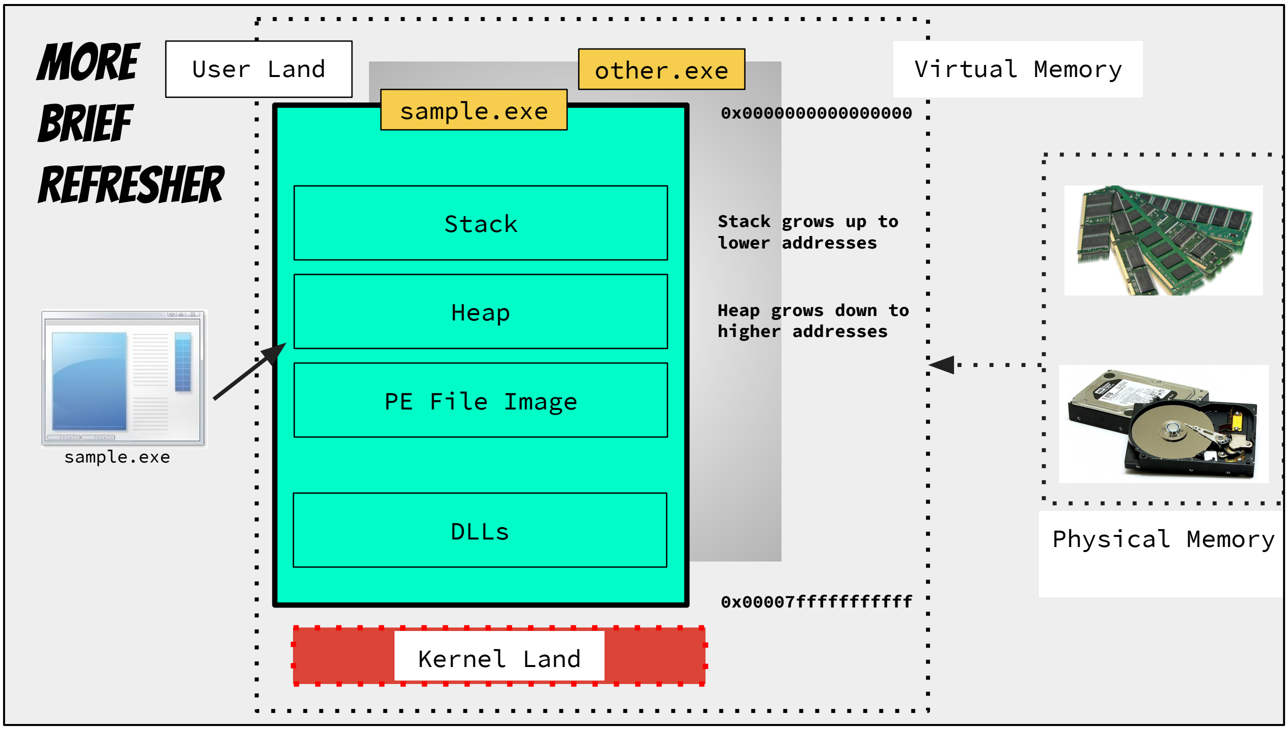


# Which of these can be written by a User?

C++

x64 Assembly

An EXE file



### Virtual Memory

Virtual memory uses both computer hardware and software to create more memory than is physically available. A memory management unit (MMU - that is built into the hardware) maps virtual address to physical address. The OS will make and manage memory mappings by using page tables and other data structures. The MMU, which acts as an address translation hardware, will automatically translate the addresses.

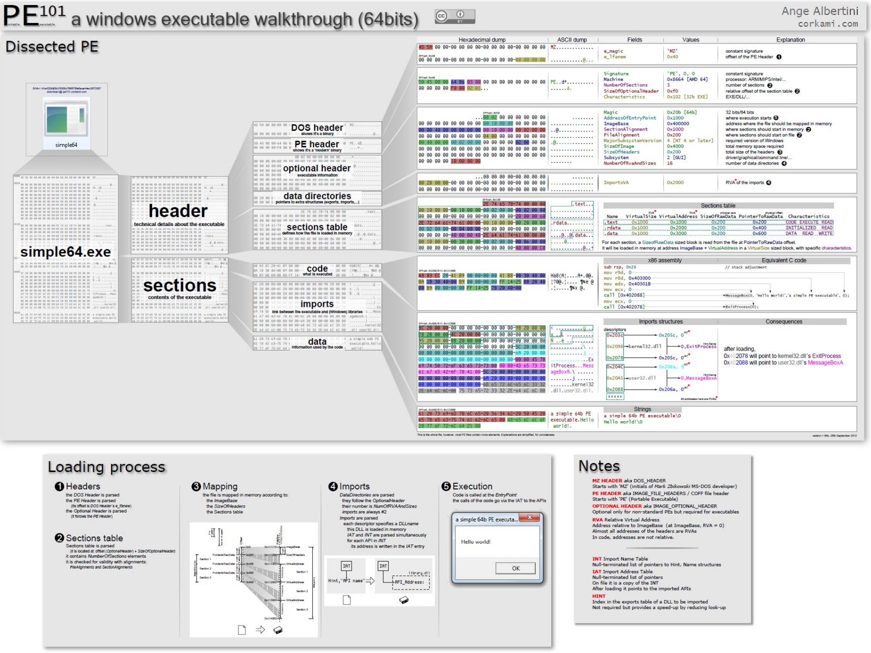
### User Land vs Kernel Land

- Applications and other user processes run in User land. Each process has their own Virtual Address Space.
- Core operating system components and drivers run in Kernel land.
- Code in user mode does have access to Kernel land.
- Code in kernel mode has access to User and Kernel land.

### Resources

https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

https://sonictk.github.io/asm_tutorial/#windows:thewindowtothehardware


simple64
simple64.exe
header
technical details about the executable
sections
contents of the executable
DOS header
shows it's a binary
PE header
shows it's a 'modern' binary
optional header
executable information
data directories
pointers to extra structures (exports, imports...)
sections table
defines how the file is loaded in memory
code
what is executed
imports
link between the executable and (Windows) libraries
data
information used by the code
Hexadecimal dump
ASCII dump

# PE101 a windows executable walkthrough (64bits)

Ange Albertini
corkami.com

## Dissected PE

| Fields | Values | Explanation |
|---|---|---|
| e_magic | 'MZ' | constant signature |
| e_lfanew | 0x40 | offset of the PE Header ❶ |
| Signature | 'PE', 0, 0 | constant signature |
| Machine | 0x8664 [AMD 64] | processor: ARM/MIPS/Intel/... |
| NumberOfSections | 3 | number of sections ❷ |
| SizeOfOptionalHeader | 0xf0 | relative offset of the section table ❷ |
| Characteristics | 0x102 [32b EXE] | EXE/DLL/... |
| Magic | 0x20b [64b] | 32 bits/64 bits |
| AddressOfEntryPoint | 0x1000 | where execution starts ❺ |
| ImageBase | 0x400000 | address where the file should be mapped in memory |
| SectionAlignment | 0x1000 | where sections should start in memory ❷ |
| FileAlignment | 0x200 | where sections should start on file ❷ |
| MajorSubsystemVersion | 4 [NT 4 or later] | required version of Windows |
| SizeOfImage | 0x4000 | total memory space required |
| SizeOfHeaders | 0x200 | total size of the headers ❸ |
| Subsystem | 2 [GUI] | driver/graphical/command line/... |
| NumberOfRvaAndSizes | 16 | number of data directories ❹ |
| ImportsVA | 0x2000 | RVA* of the imports ❹ |

### Sections table

| Name | VirtualSize (RVA*) | VirtualAddress (RVA*) | SizeOfRawData (physical size) | PointerToRawData (physical offset) | Characteristics |
|---|---|---|---|---|---|
| .text | 0x1000 | 0x1000 | 0x200 | 0x200 | CODE EXECUTE READ |
| .rdata | 0x1000 | 0x2000 | 0x200 | 0x400 | INITIALIZED READ |
| .data | 0x1000 | 0x3000 | 0x200 | 0x600 | DATA READ WRITE |

For each section, a SizeofRawData sized block is read from the file at PointerToRawData offset.
It will be loaded in memory at address ImageBase + VirtualAddress in a VirtualSize sized block, with specific characteristics.

| x86 assembly | Equivalent C code |
|---|---|
| `sub rsp, 0x28` | `// stack adjustment` |
| `mov r9d, 0` | |
| `mov r8d, 0x403000` | |
| `mov edx, 0x40301B` | |
| `mov ecx, 0` | |
| `call [0x402088]` | `MessageBox(0, "Hello World!","a simple PE executable", 0);` |
| `mov ecx, 0` | |
| `call [0x402078]` | `ExitProcess(0);` |

### Imports structures

```
descriptors
0x203c  -> 0x205c, 0 (INT*)
0x2098  <- kernel32.dll       -> 0,ExitProcess (Hint,Name)
0x2078  -> 0x205c, 0 (IAT*)
0x204C  -> 0x206a, 0 (INT*)
0x20A5  <- user32.dll         -> 0,MessageBoxA (Hint,Name)
0x2088  -> 0x206a, 0 (IAT*)
```

All addresses here are RVAs.*

### Consequences

after loading,
0x402078 will point to kernel32.dll's ExitProcess
0x402088 will point to user32.dll's MessageBoxA

### Strings

```
a simple 64b PE executable\0
Hello world!\0
```

This is the whole file, however, most PE files contain more elements. Explanations are simplified, for conciseness.

version 1 64b, 25th September 2012

## Loading process

❶ Headers
the *DOS Header* is parsed
the *PE Header* is parsed
(its offset is *DOS Header*'s e_lfanew)
the *Optional Header* is parsed
(it follows the *PE Header*)

❷ Sections table
Sections table is parsed
(it is located at: offset (*OptionalHeader*) + *SizeOfOptionalHeader*)
it contains *NumberOfSections* elements
it is checked for validity with alignments:
*FileAlignments* and *SectionAlignments*

❸ Mapping
the file is mapped in memory according to:
the *ImageBase*
the *SizeOfHeaders*
the Sections table

❹ Imports
*DataDirectories* are parsed
they follow the *OptionalHeader*
their number is *NumOfRVAAndSizes*
*imports* are always #2
*Imports* are parsed
each descriptor specifies a *DLLname*
this DLL is loaded in memory
*IAT* and *INT* are parsed simultaneously
for each API in *INT*
its address is written in the *IAT* entry

❺ Execution
Code is called at the *EntryPoint*
the calls of the code go via the IAT to the APIs

a simple 64b PE executa...
Hello world!
OK

## Notes

**MZ HEADER** *aka* DOS_HEADER
Starts with 'MZ' (initials of *Mark Zbikowski* MS-DOS developer)

**PE HEADER** *aka* IMAGE_FILE_HEADERS / COFF file header
Starts with 'PE' (Portable Executable)

**OPTIONAL HEADER** *aka* IMAGE_OPTIONAL_HEADER
Optional only for non-standard PEs but required for executables

**RVA** Relative Virtual Address
Address relative to ImageBase (at ImageBase, RVA = 0)
Almost all addresses of the headers are RVAs
In code, addresses are *not* relative.

**INT** Import Name Table
Null-terminated list of pointers to Hint, Name structures

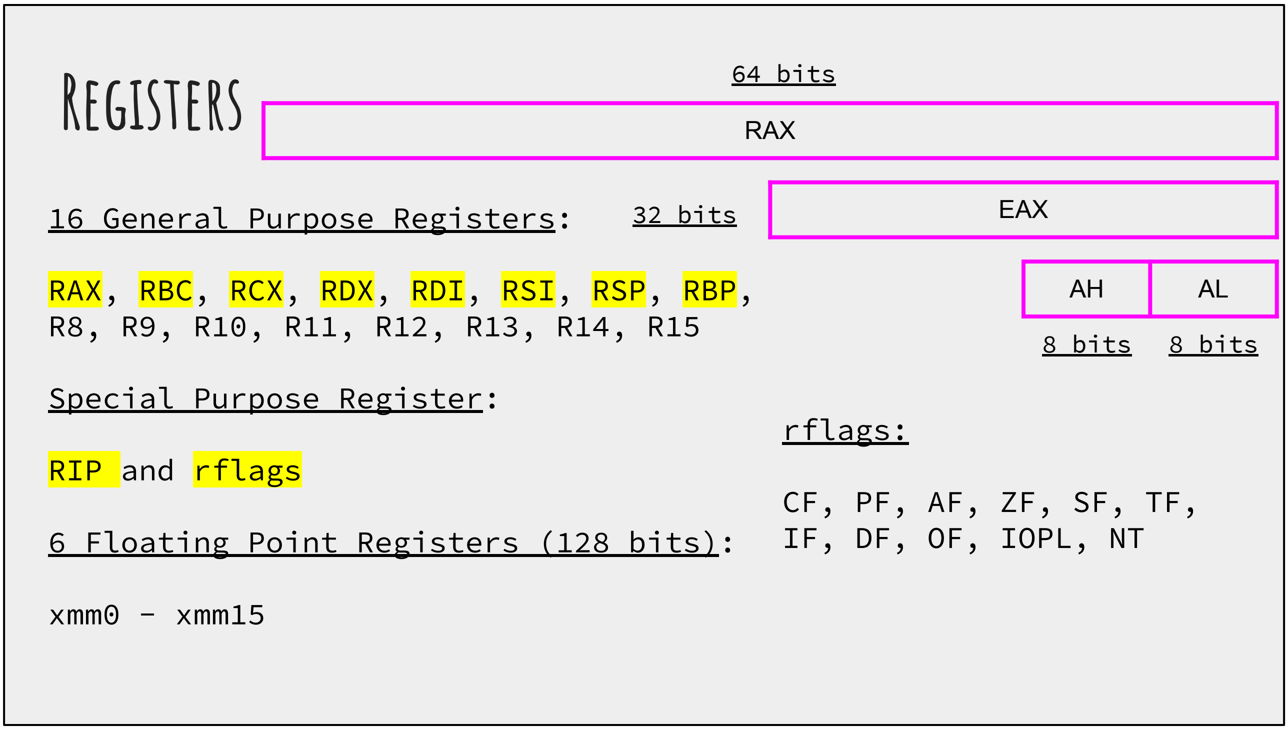
**IAT** Import Address Table
Null-terminated list of pointers
On file it is a copy of the INT
After loading it points to the imported APIs

**HINT**
Index in the exports table of a DLL to be imported
Not required but provides a speed-up by reducing look-up

https://github.com/corkami/pics/blob/master/binary/pe101/README.md

# Demo

This brings back memories ;)



# Registers

64 bits

RAX

32 bits

EAX

AH | AL

8 bits | 8 bits

16 General Purpose Registers:

RAX, RBC, RCX, RDX, RDI, RSI, RSP, RBP,
R8, R9, R10, R11, R12, R13, R14, R15

Special Purpose Register:

RIP and rflags

6 Floating Point Registers (128 bits):

xmm0 - xmm15

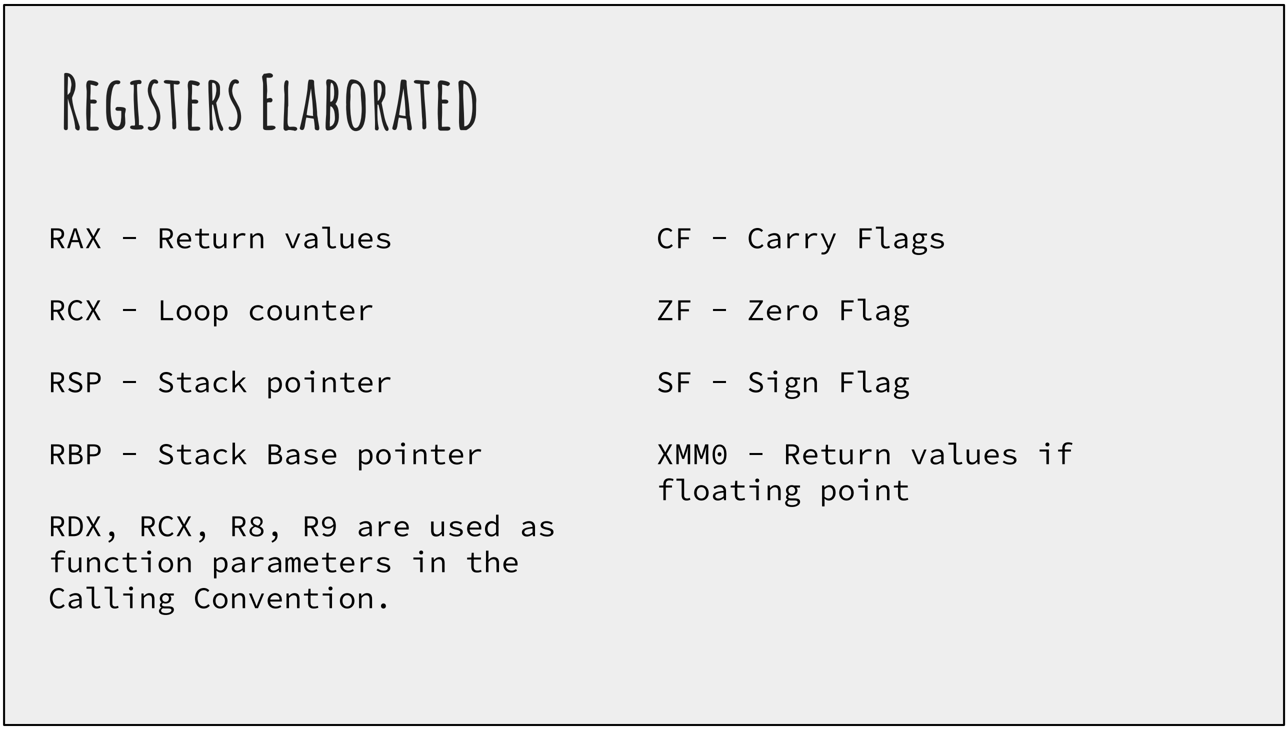
rflags:

CF, PF, AF, ZF, SF, TF,
IF, DF, OF, IOPL, NT

- The highlighted registers are an extension of the the 32-bit registers.
- Registers are memory locations that can be directly accessed on the CPU.
- Faster access than RAM.

**References:**

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture
https://software.intel.com/content/www/us/en/develop/articles/introduction-to-x64-assembly.html
https://sonictk.github.io/asm_tutorial/



## Registers Elaborated

RAX - Return values

RCX - Loop counter

RSP - Stack pointer

RBP - Stack Base pointer

RDX, RCX, R8, R9 are used as function parameters in the Calling Convention.

CF - Carry Flags

ZF - Zero Flag

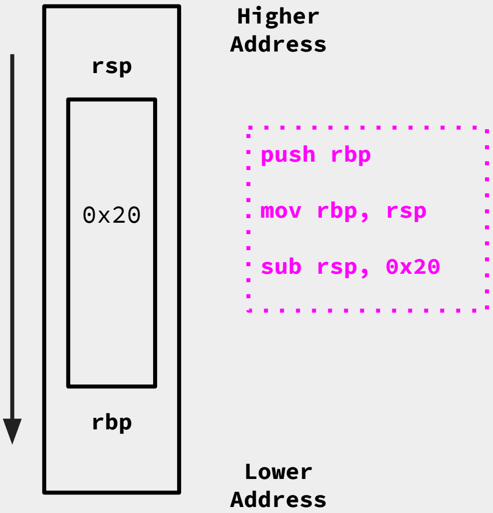
SF - Sign Flag

XMM0 - Return values if floating point

- Registers are memory locations that can be directly accessed on the CPU.
- Faster access than RAM.

**References:**
https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160

# Stack



```
push rbp

mov rbp, rsp

sub rsp, 0x20
```

The stack grows down.

It stores the return address.

It stores the function parameters (for any remaining parameters after the registers have been used).

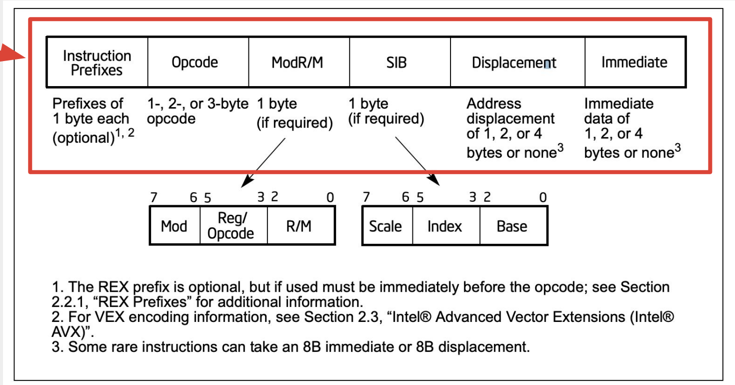
It stores the function's local variables.

RBP keeps track of the stack frame. RSP changes.

# CPU - Fetch -> Decode -> Execute

CPU fetches the instruction that is at the memory address stored in RIP.

The instruction is decoded and then executed.

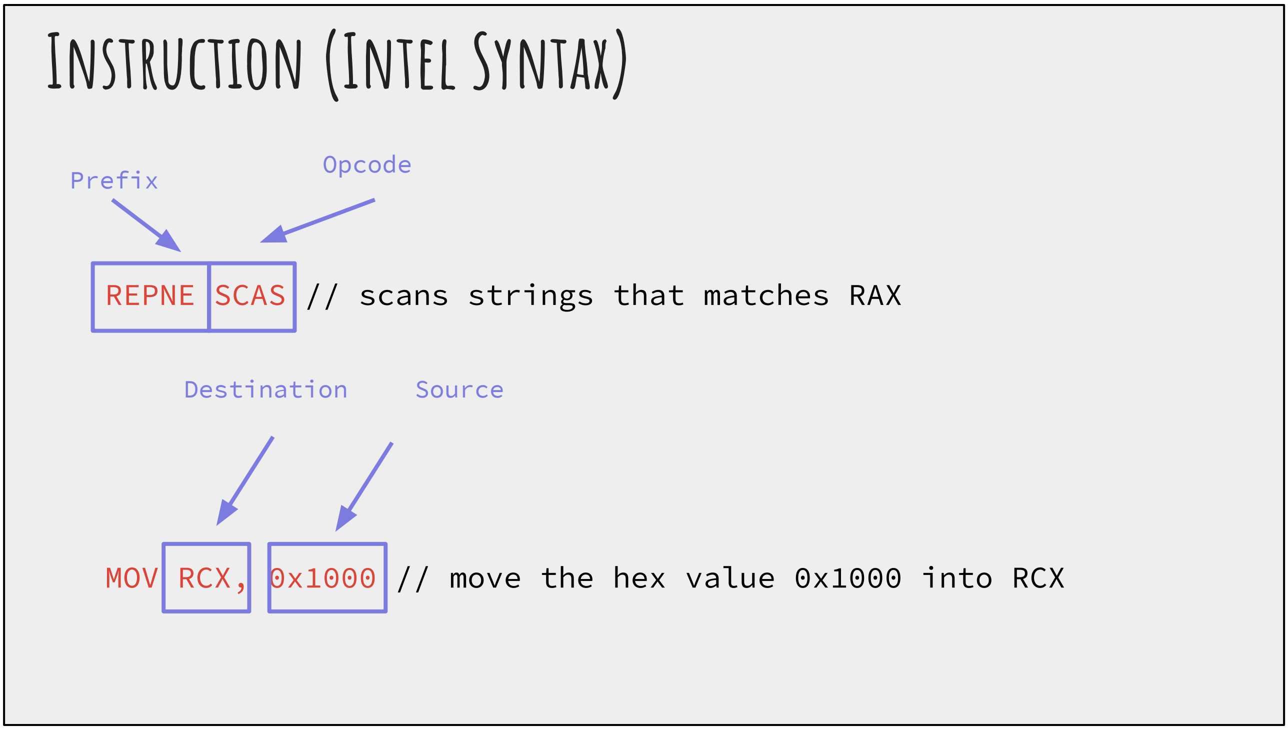


1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, "REX Prefixes" for additional information.
2. For VEX encoding information, see Section 2.3, "Intel® Advanced Vector Extensions (Intel® AVX)".
3. Some rare instructions can take an 8B immediate or 8B displacement.

**Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format**

Listing:
https://www.felixcloutier.com/x86/index.html
https://en.wikipedia.org/wiki/X86_instruction_listings#Original_8086/8088_instructions
(x86)

References:
https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf



# Instruction (Intel Syntax)

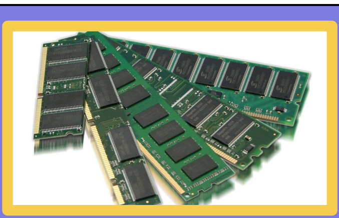
Prefix
Opcode

```
REPNE SCAS // scans strings that matches RAX
```

Destination
Source

```
MOV RCX, 0x1000 // move the hex value 0x1000 into RCX
```

Heap

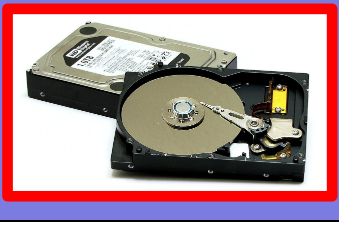


Running Process

# Where in Memory?

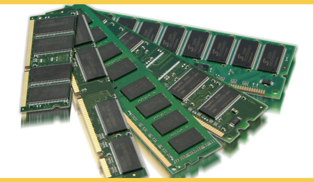
Stack

General Purpose Registers



Heap

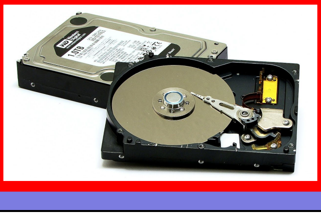
Running Process



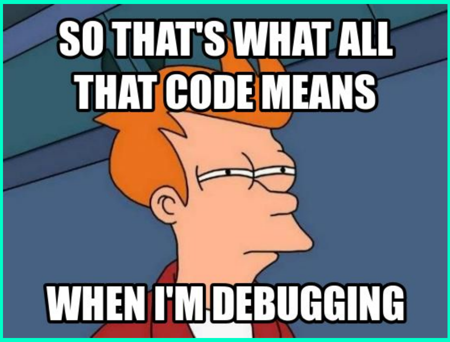
# Where in Memory?

Stack

General Purpose Registers



# Demo



# Exercise - Exploring the FILE

1. Password for exercise.7z - solvere102
2. Is there a main?
3. What are some of the visible strings?

# Exercise - API Land

1. What APIs does the file call?
2. Why is the API WideCharToMultiByte called twice?
3. Explain what CreateFileA does based on the parameters passed.

# Exercise - Finding the Key

1. Is the key encoded? Can you find the encoded bytes?
2. What routine is used to encrypt the key?
3. How else can you solve to decrypt the key?

# Resources

Other awesome learning resources

- https://www.begin.re/
- https://malwareunicorn.org/workshops/re101.html#0