RIT

جامعة روتشستر الأمريكية للتكنولوجيا في دبي

A Global American University in Dubai

CSEC 202 Reverse Engineering Fundamentals

Module 0x04
Basics of assembly
{a lot of x86-32} & {some of x86-64}

Eng. Emad Abu Khousa

Sections: 600 | 601 | 602

00110100110011 1

February 12, 2024

Learning Objectives*

1. Recognize and define x86 assembly language
2. Distinguish the difference between 16/32/64-bit assembly code
3. Explore sections of executable file
4. Differentiate Intel and AT&T syntax
5. Explain Stack and Heap memory allocations.

* Adopted from: Saltaformaggio, B. (2022). *Introduction to assembly language*. Georgia Tech.

1) Assembly Language

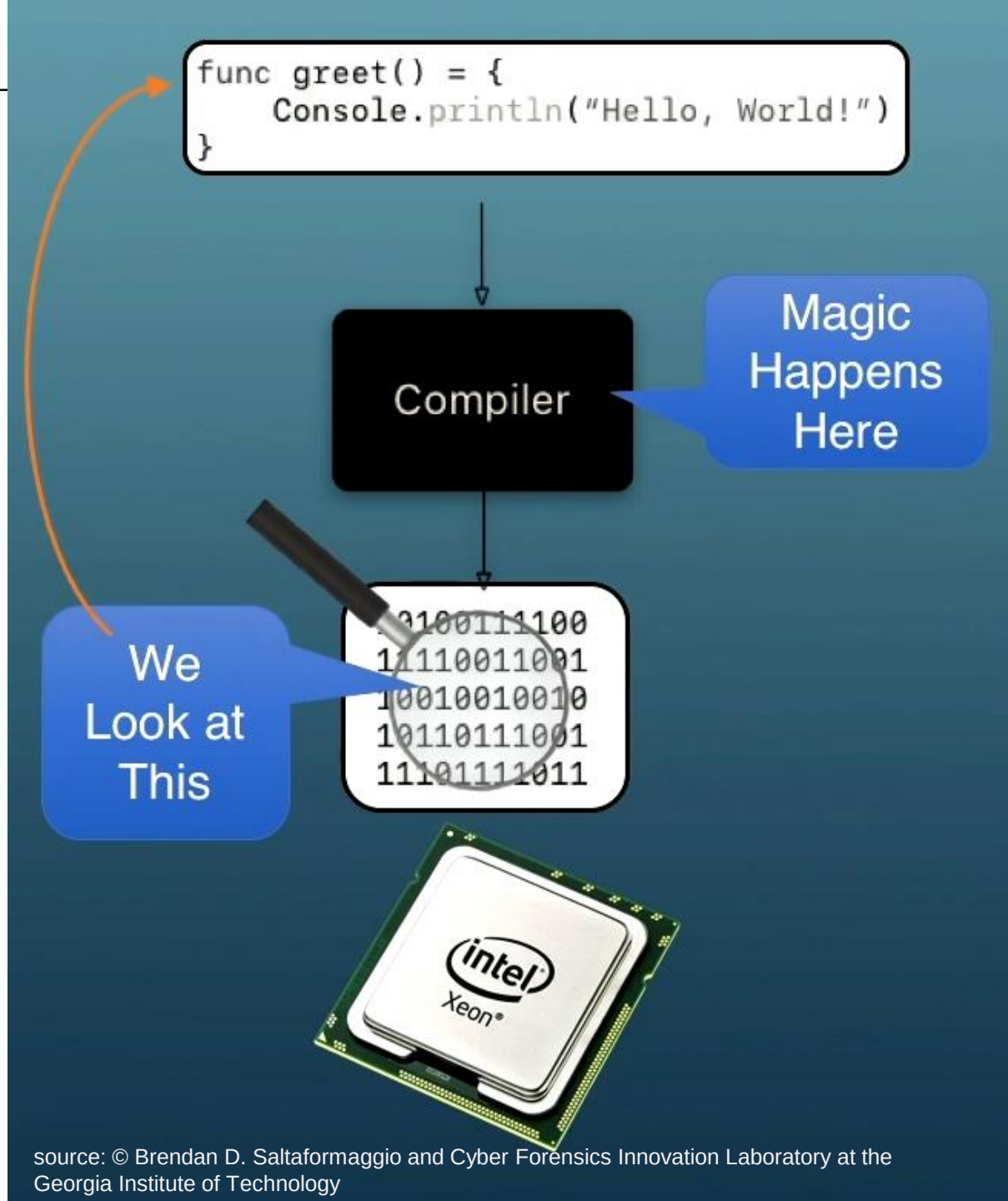
Binary Analysis

- Malware does not come with source code
- So we are left analyzing only malware executables (a.k.a. binaries)

❑ **Is it hard? Yes and NO**

- An executable program is just a sequence of 1's and 0's which the CPU understands as instructions

- **Reverse Engineering:** The process of analyzing a subject binary program to create representations of the program's logic at a higher level of abstraction

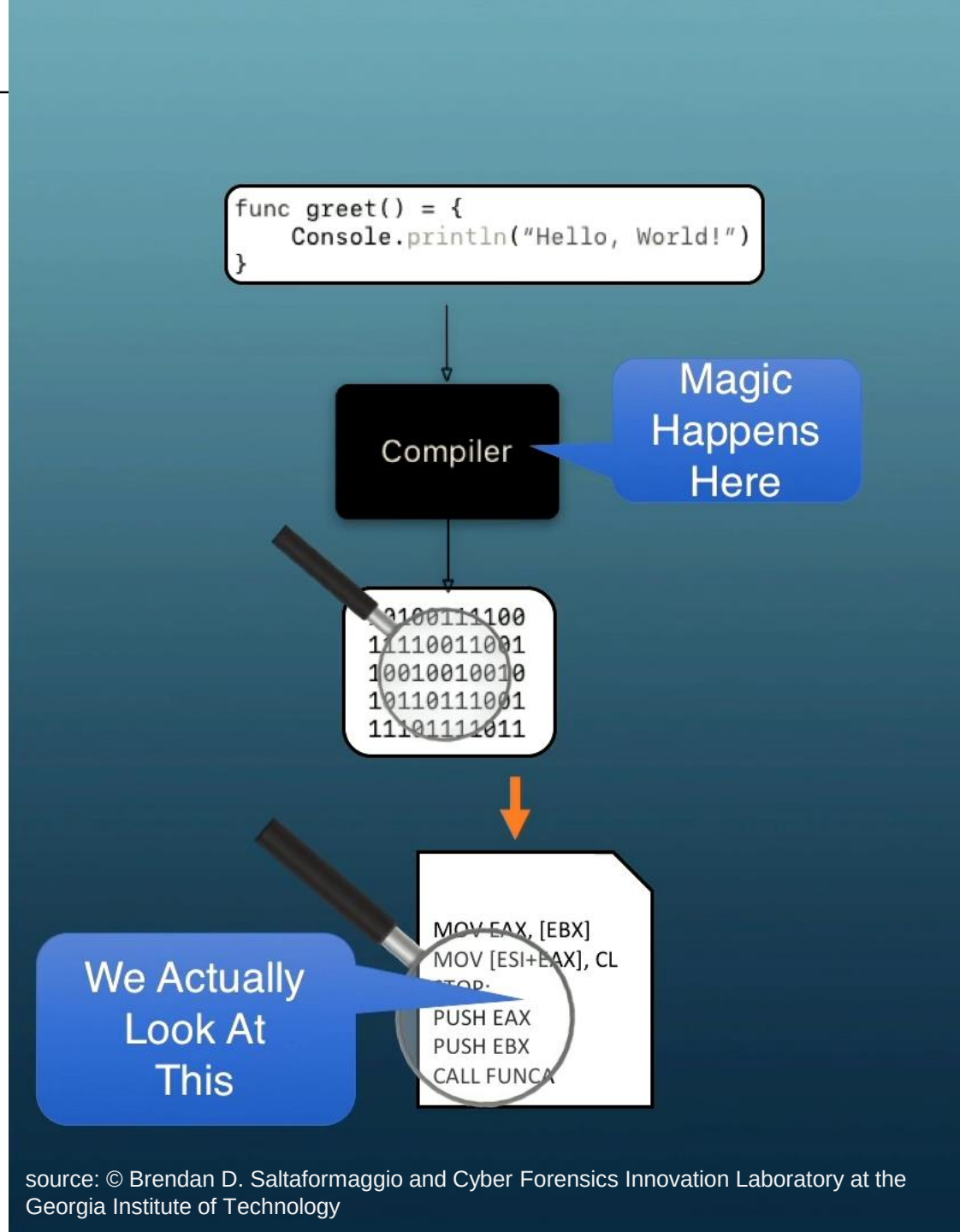


source: © Brendan D. Saltaformaggio and Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology

Binary Analysis

- Lucky for us, binary programs can be disassembled back to Assembly Language

- “An executable program is just a sequence of 1’s and 0’s which the CPU understands as instructions”



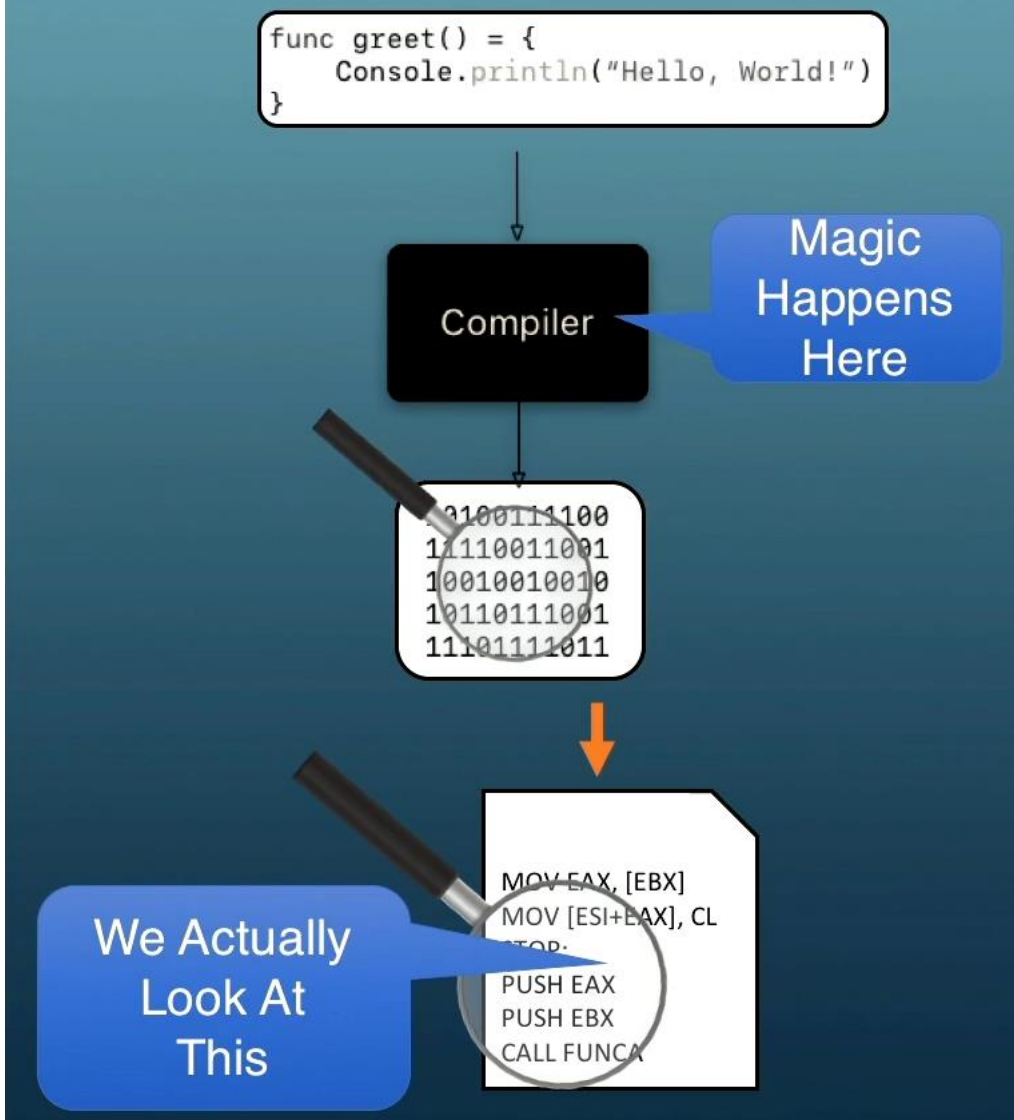
source: © Brendan D. Saltaformaggio and Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology

Binary Analysis

An executable program is just a sequence of 1's and 0's which the CPU understands as instructions

1:1

Assembly code is just a sequence of **mnemonics** which represent each processor instruction (called an **opcode**)

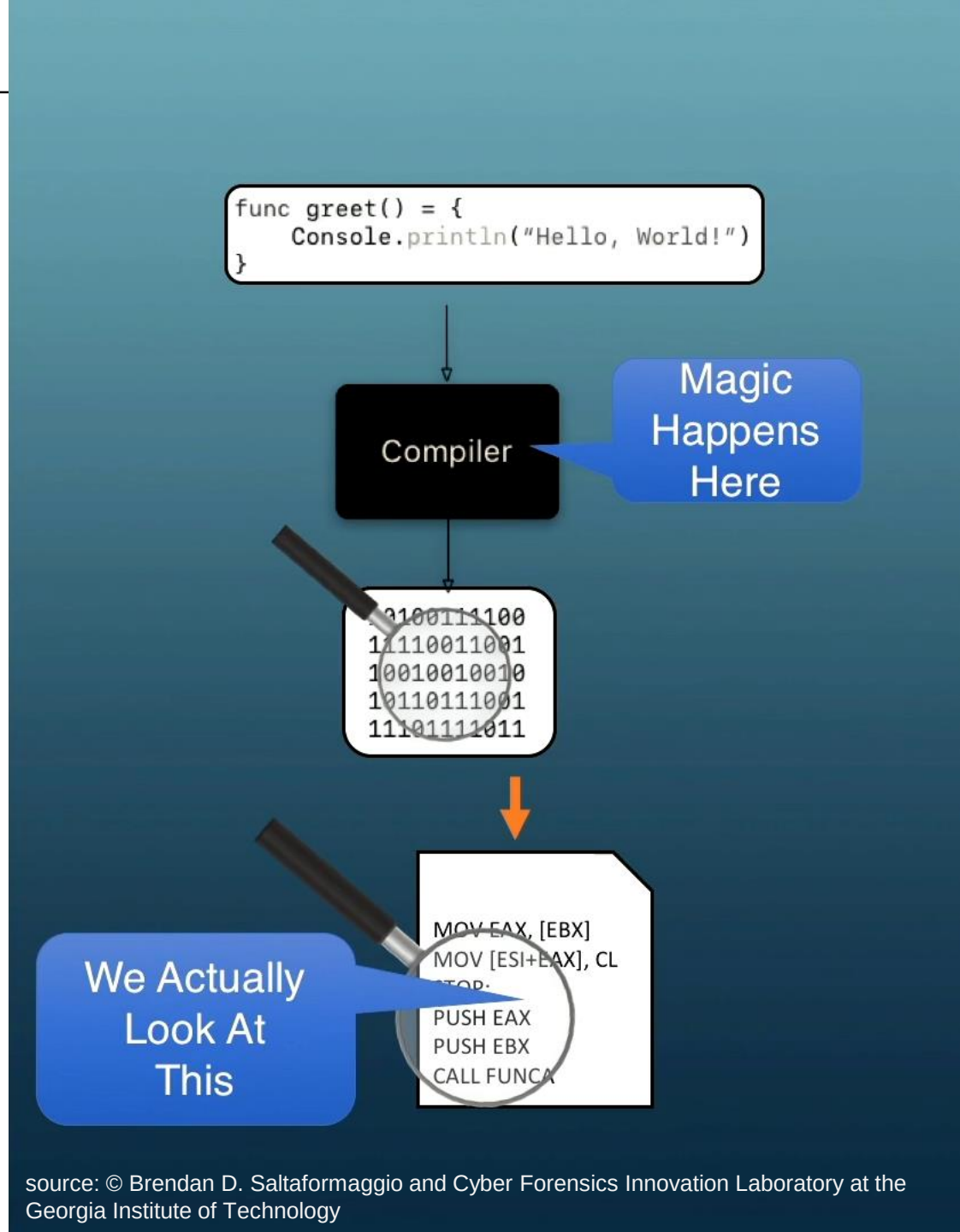


source: © Brendan D. Saltaformaggio and Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology

Binary Analysis

Opcode Disassembly Example:

0000 0100 0000 1010 = **040**Ah = **add al**,10



source: © Brendan D. Saltaformaggio and Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology

Your New Hobby: Assembly Language

Recommendations for success:

1. Compile programs that you're familiar with and examine the resulting assembler to understand what's going on
2. Write native assembler applications and verify that they work properly
3. Practice ..Practice.. Practice.

or

4. Read and memories the Intel architecture manuals (5000+ pages)

This module is only a quick tutorial on Intel assembly, there is so much more to learn!

Native – vs. - Compiled

```
1  section .text
2     global _start      ;must be declared for linker (ld)
3
4  _start:               ;tells linker entry point
5     mov  edx,len       ;message length
6     mov  ecx,msg       ;message to write
7     mov  ebx,1         ;file descriptor (stdout)
8     mov  eax,4         ;system call number (sys_write)
9     int  0x80          ;call kernel
10
11    mov  eax,1         ;system call number (sys_exit)
12    int  0x80          ;call kernel
13
14 section .data
15 msg db 'Hello, world!', 0xa  ;string to be printed
16 len equ $ - msg       ;length of the string
```

Live Demo

```
segment .text       ;code segment
   global _start     ;must be declared for linker

_start:             ;tell linker entry point
   mov edx,len      ;message length
   mov ecx,msg      ;message to write
   mov ebx,1        ;file descriptor (stdout)
   mov eax,4        ;system call number (sys_write)
   int 0x80         ;call kernel

   mov eax,1        ;system call number (sys_exit)
   int 0x80         ;call kernel

segment .data       ;data segment
msg     db 'Hello, world!',0xa   ;our dear string
len     equ     $ - msg          ;length of our dear string
```

https://www.tutorialspoint.com/assembly_programming/index.htm

Native – vs. - Compiled

```
section .text
   global _start     ;must be declared for linker (ld)

_start:              ;tells linker entry point
   mov  edx,len      ;message length
   mov  ecx,msg      ;message to write
   mov  ebx,1        ;file descriptor (stdout)
   mov  eax,4        ;system call number (sys_write)
   int  0x80         ;call kernel

   mov  eax,1        ;system call number (sys_exit)
   int  0x80         ;call kernel

section .data
msg db 'Hello, world!', 0xa  ;string to be printed
len equ $ - msg      ;length of the string
```

```
#include <stdio.h>

int main() {
    printf("Hello, world!");
 return 0;
}
```

```
    .file   "hellow.c"
    .intel_syntax noprefix
    .text
    .section    .rodata
.LC0:
    .string "Hello, world!"
    .text
    .globl  main
    .type   main, @function
main:
    push    rbp
    mov rbp, rsp
    lea rax, .LC0[rip]
    mov rdi, rax
    mov eax, 0
    call    printf@PLT
    mov eax, 0
    pop rbp
    ret
```

Your New Hobby: Assembly Language

Recommendations for success:

1. Compile programs that you're familiar with and examine the resulting assembler to understand what's going on
2. Write native assembler applications and verify that they work properly
3. Practice ..Practice.. Practice.

or

4. Read and memories the Intel architecture manuals (5000+ pages)

This module is only a quick tutorial on Intel assembly, there is so much more to learn!

Back to hello.c

- How much code is generated?
- How complex is the executable?

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    if (argc == 2)
        printf("Hello %s\n", argv[1]);
return 0;
}
```

gcc -m32 -Wall -fno-asynchronous-unwind-tables -fno-pie -fno-pic -masm=intel -S hello.c -o hello.s

Back to hello.c

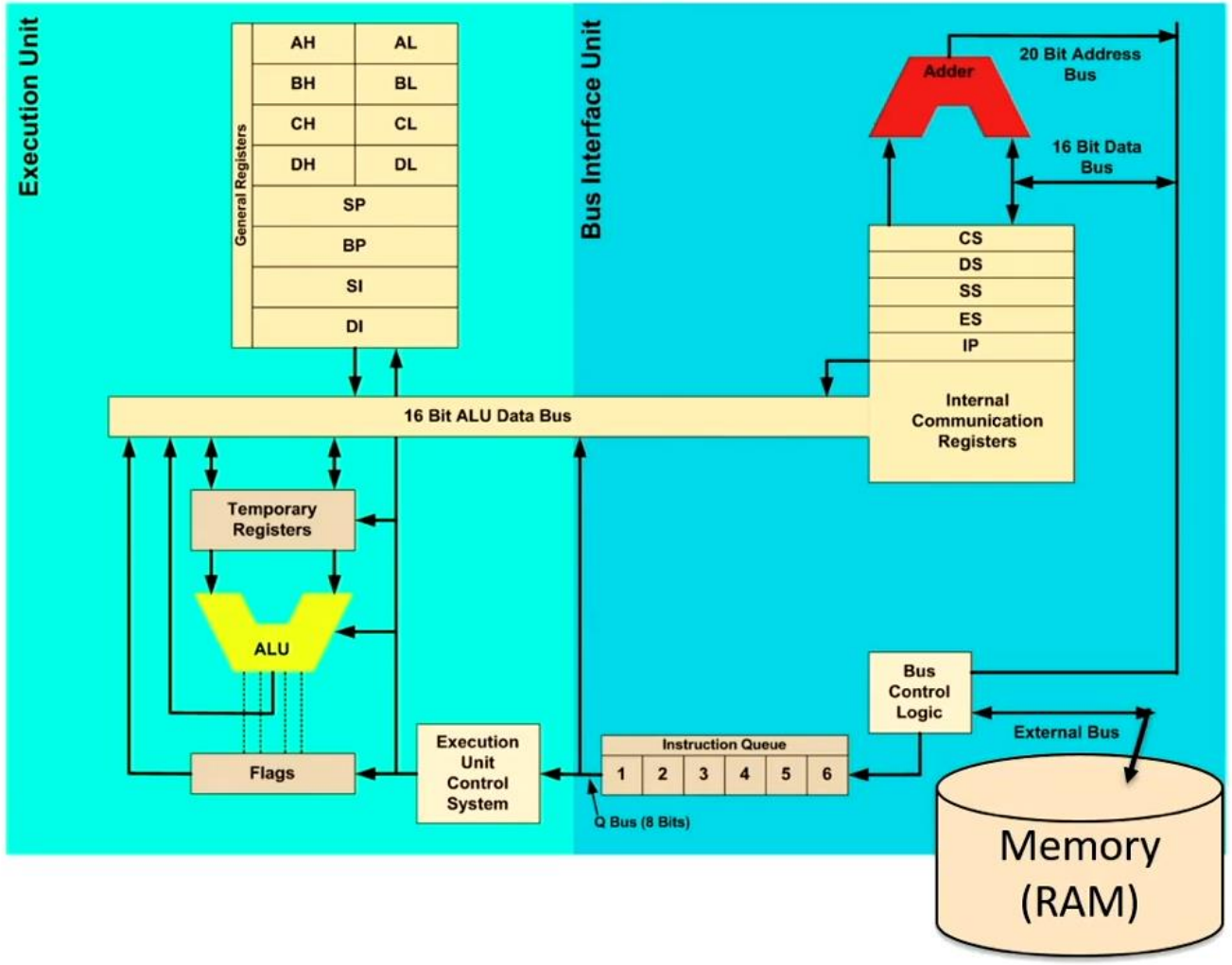
By the end of these slides, you will understand every line of this ☺

```
    .file   "hello.c"
    .intel_syntax noprefix
    .text
    .section    .rodata
.LC0:
    .string "Hello %s\n"
    .text
    .globl  main
    .type   main, @function
main:
    lea ecx, [esp+4]
    and esp, -16
    push    DWORD PTR [ecx-4]
    push    ebp
    mov ebp, esp
    push    ecx
    sub esp, 4
    mov eax, ecx
    cmp DWORD PTR [eax], 2
    jne .L2
    mov eax, DWORD PTR [eax+4]
    add eax, 4
    mov eax, DWORD PTR [eax]
    sub esp, 8
    push    eax
    push    OFFSET FLAT:.LC0
    call    printf
    add esp, 16
.L2:
    mov eax, 0
    mov ecx, DWORD PTR [ebp-4]
    leave
    lea esp, [ecx-4]
    ret
    .size   main, .-main
    .ident  "GCC: (Debian 13.2.0-7) 13.2.0"
    .section    .note.GNU-stack,"",@progbits
```

Intel Assembler: Need to Know

You must begin thinking like a processor…

Basic 16-bit CPU Architecture



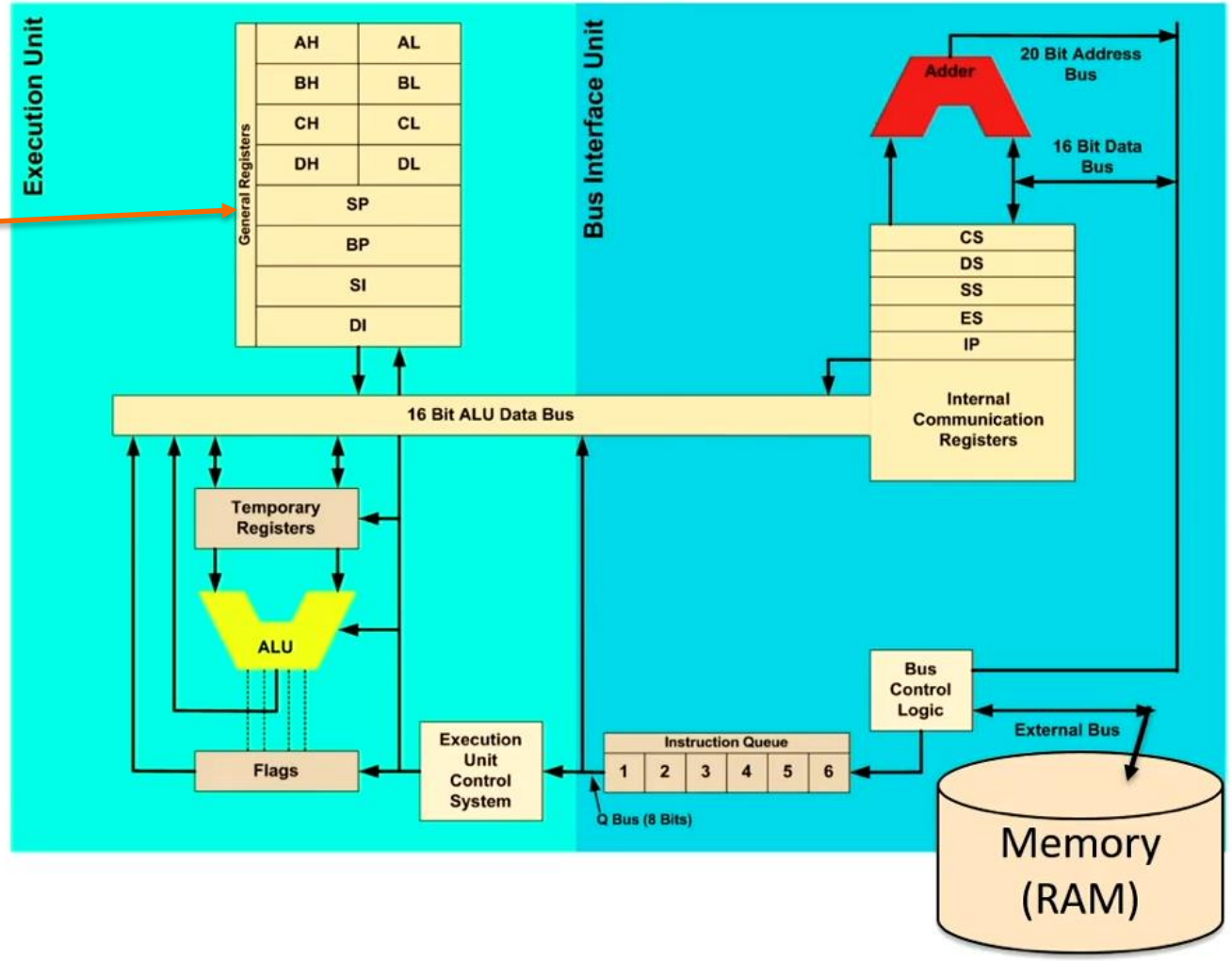
source: © Brendan D. Saltaformaggio and Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology

Intel Assembler: Need to Know

You must begin thinking like a processor…

- Registers
  Store “Live” Data

Basic 16-bit CPU Architecture



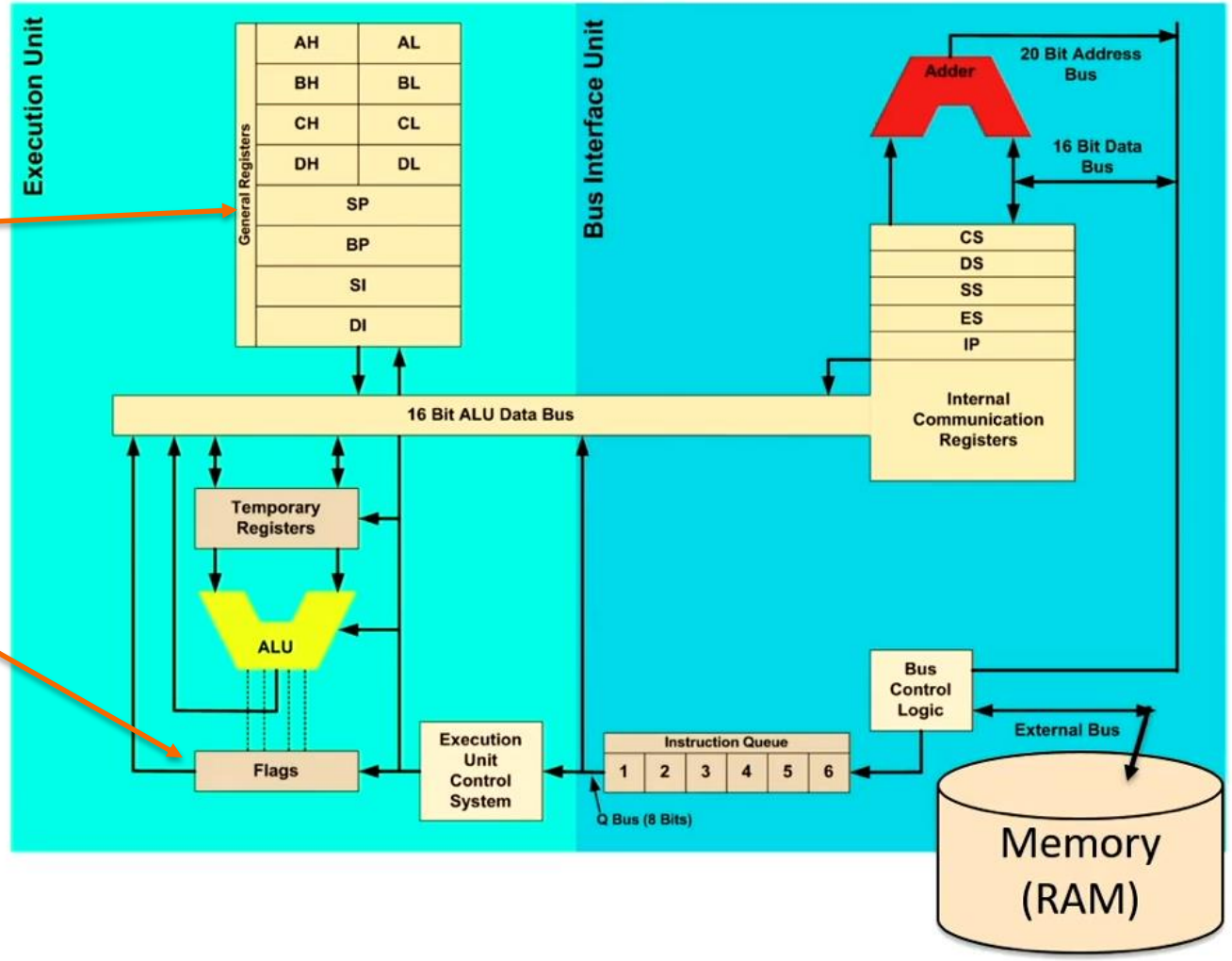
source: © Brendan D. Saltaformaggio and Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology

Intel Assembler: Need to Know

You must begin thinking like a processor…

- Registers
  Store “Live” Data
- Flags
  Control the CPU’s decisions

Basic 16-bit CPU Architecture



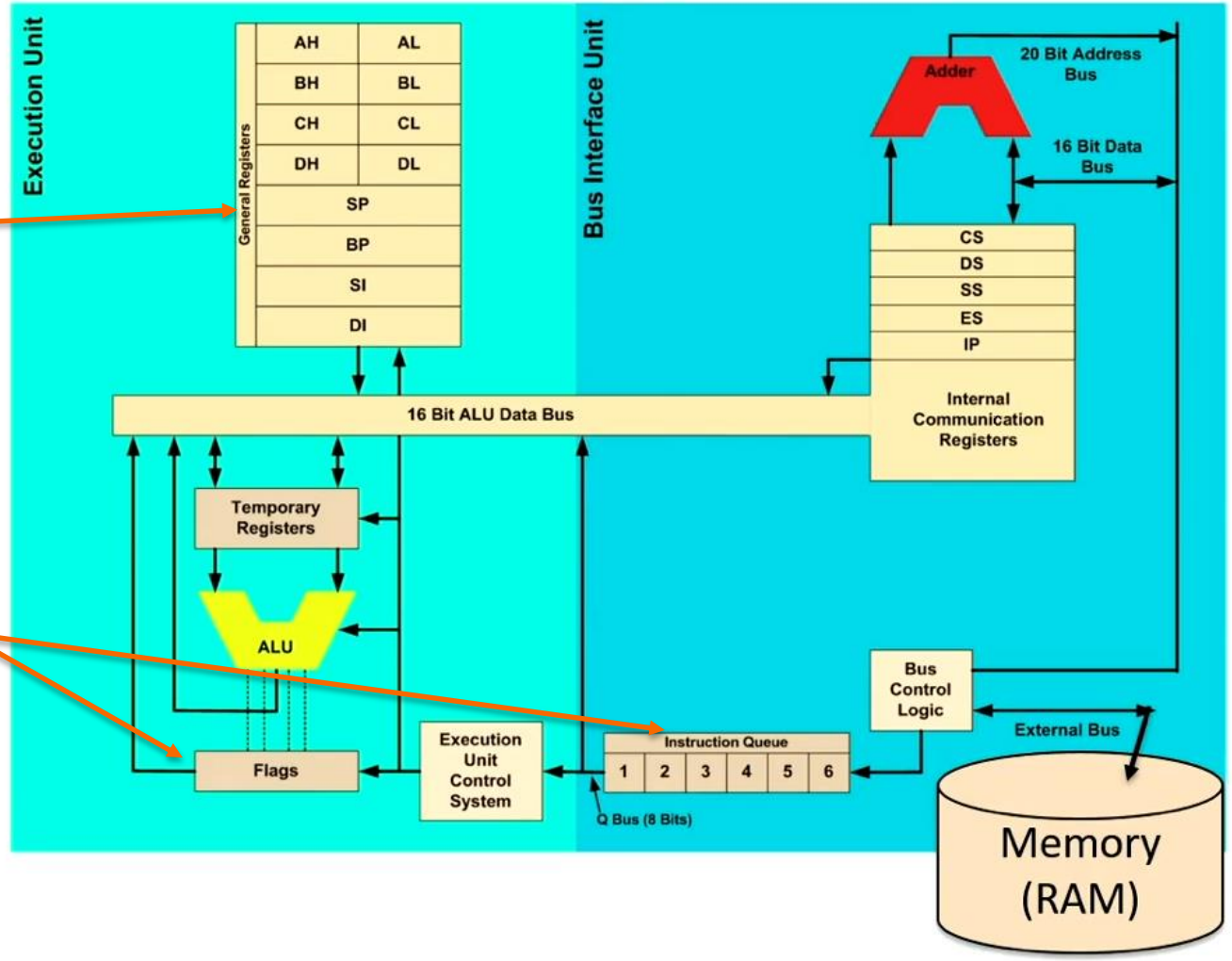
source: © Brendan D. Saltaformaggio and Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology

Intel Assembler: Need to Know

You must begin thinking like a processor…

- Registers
  Store “Live” Data
- Flags
  Control the CPU’s decisions
- Instructions
  Tell the CPU what to do

Basic 16-bit CPU Architecture



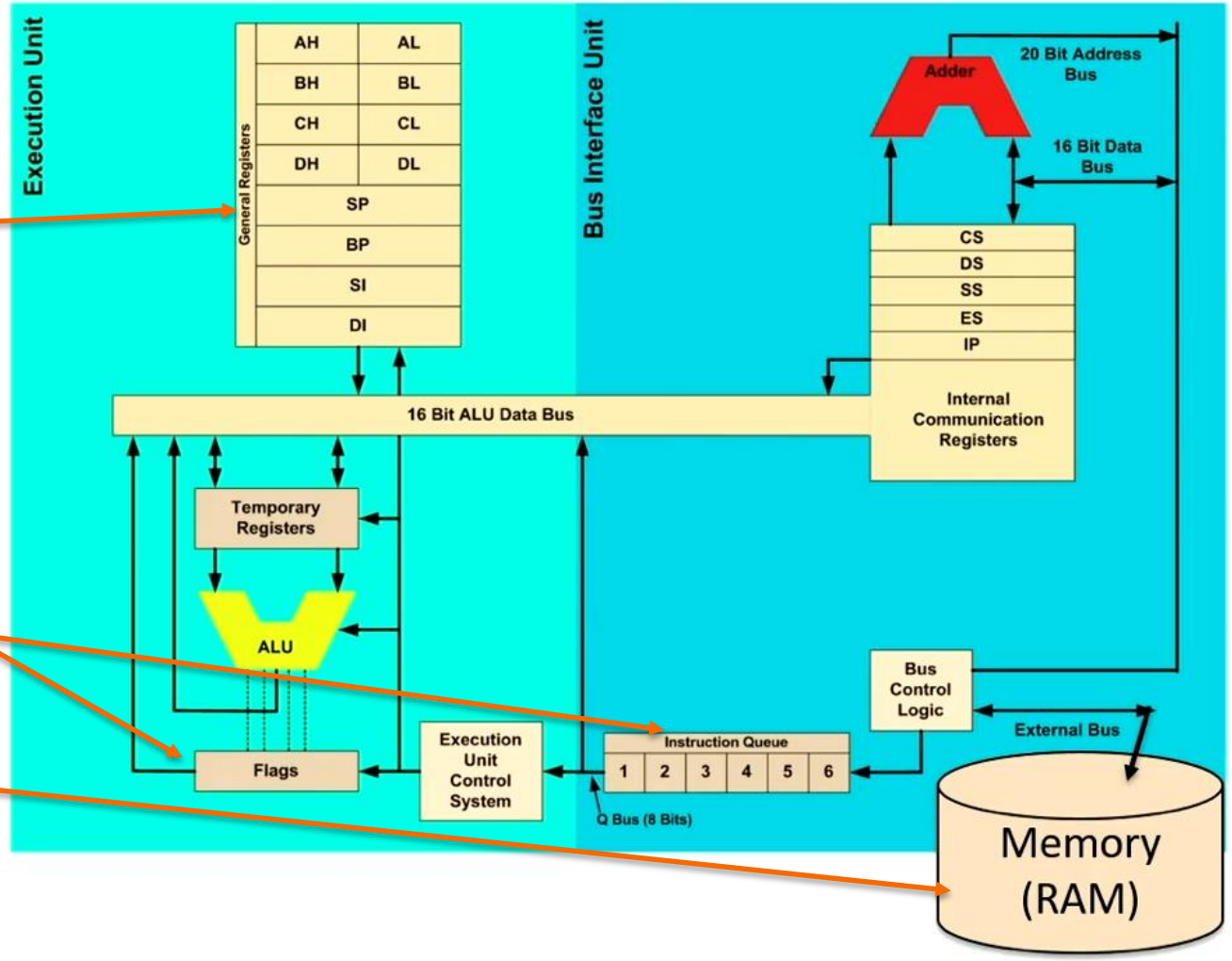
source: © Brendan D. Saltaformaggio and Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology

Intel Assembler: Need to Know

You must begin thinking like a processor…

- Registers
  Store "Live" Data
- Flags
  Control the CPU's decisions
- Instructions
  Tell the CPU what to do
- Data Formats
  How data is stored
- Stack & Heap
  How memory is accessed

Basic 16-bit CPU Architecture



source: © Brendan D. Saltaformaggio and Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology

2) Basics of Assembly Code

x86 Assembler: Registers (again!)

Maximum register size depends on the CPU:
8088/8086/80186/80188 / 80286:

➤ 16-bit registers

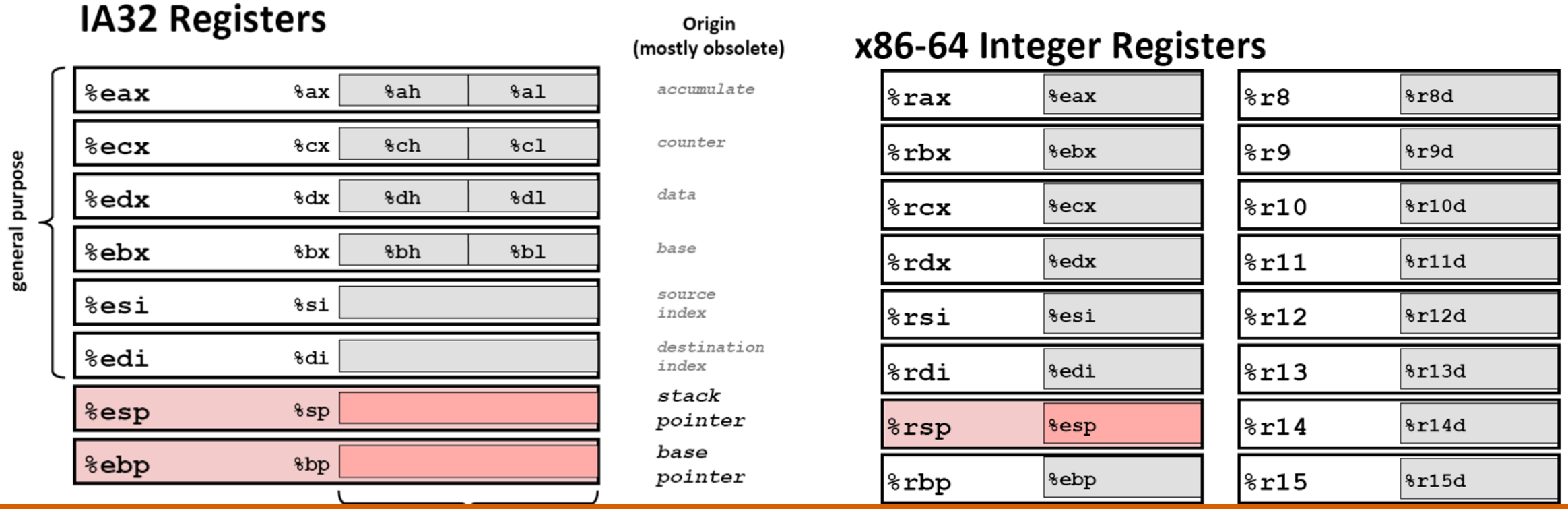
80386/80486 / Pentium / Pentium Pro / Pentium MMX / Pentium II / Pentium M/Pentium III / Pentium 4:

➤ 32-bit registers

Pentium 4 [later] / Pentium D / Pentium Extreme / Core2 (i3/i5/i7):

➤ 64-bit registers

x86 / 64 Assembler: Registers



You will need to understand the differences between 16-bit, 32-bit, and 64-bit processors. Because the register names will be quite different, the feature sets will be quite different, and malware in the wild will abuse these differences in order to make it difficult to reverse engineer what they are doing.

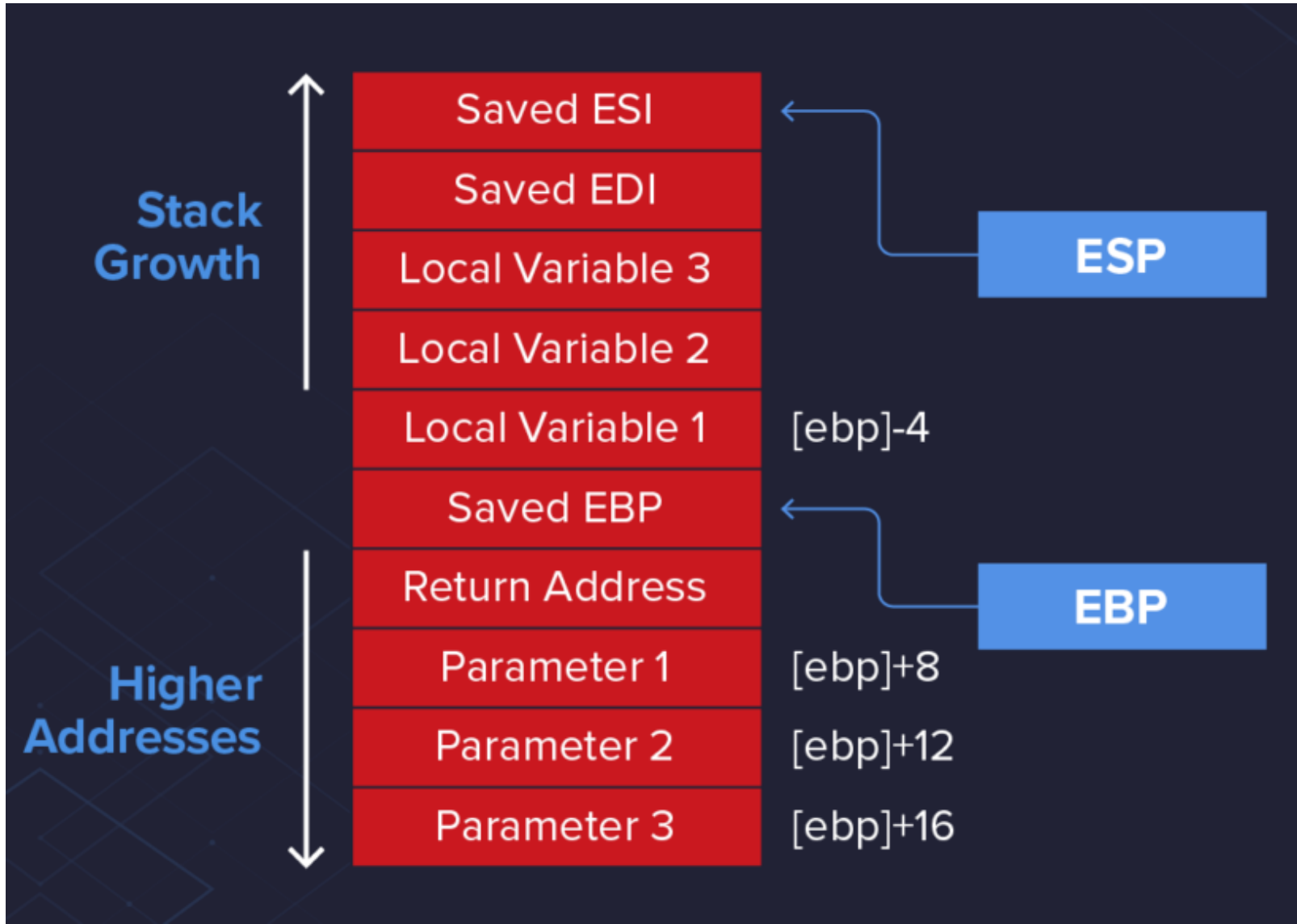
Assembler general-purpose Registers

1. EAX - Known as the **accumulator register**. Often used to store **the return value of a function.**
2. EBX - Sometimes known as the **base register**, not to be confused with the base pointer. Sometimes used as a base pointer for memory access.
3. EDX - Sometimes known as the **data register**.
4. ECX - Sometimes known as the **counter register**. Used as a loop counter.
5. ESI - Known as the **source index**. Used as the source pointer in string operations.
6. EDI - Known as the **destination index**. Used as the destination pointer in string operations.
7. E**BP** {**Base Pointer = Frame Pointer**: **specific for keeping track of the stack**}
8. E**SP** {**Stack Pointer: specific for keeping track of the stack**}
9. EFLAGS register

Assembler general-purpose Registers

- ESP - The **stack pointer**. Holds the address of the top of the stack.

- EBP - The **base pointer**. Holds the address of the base (bottom) of the stack frame. EBP is used to keep track of the function's base address, which is essential for accessing local variables and the function's return address.

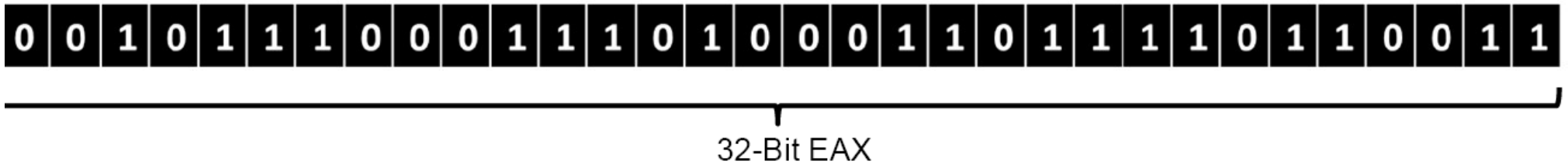


Assembler: Registers

On a 32-bit processor, each register holds exactly 32 bits.

- This is used to store any binary value the program needs

Consider storing the number **775,567,283** in the EAX register:

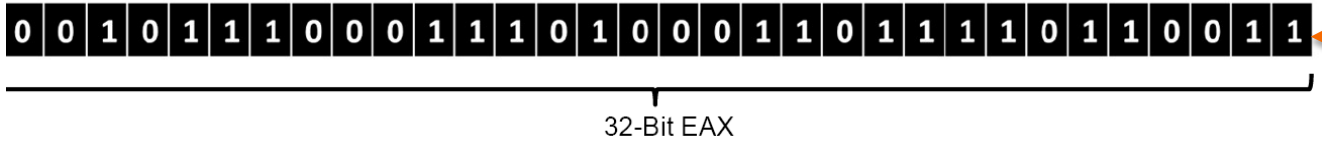


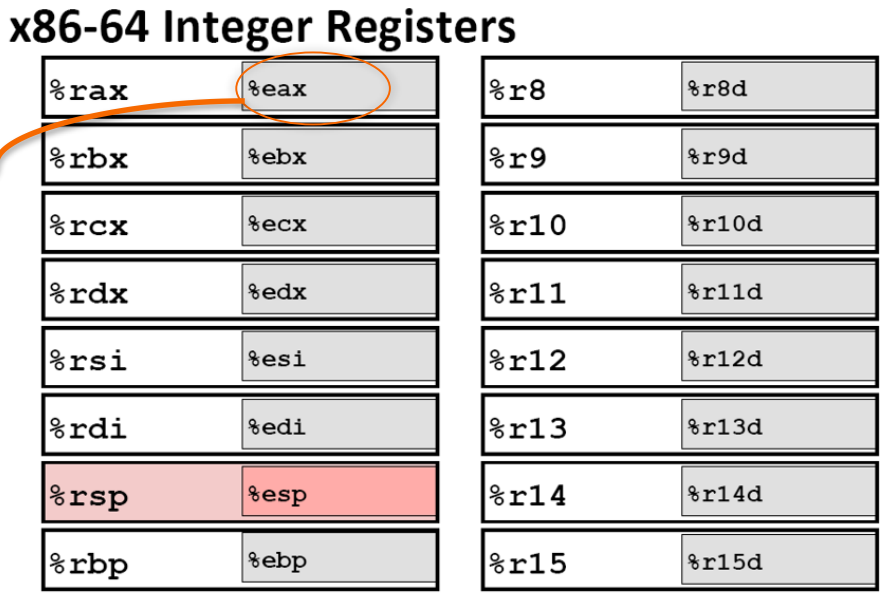
Assembler: Registers

On a 32-bit processor, each register holds exactly 32 bits.

- This is used to store any binary value the program needs

Consider storing the number 775,567,283
in the EAX register:





On 64-bit processors, you have

- **64-bit registers:** rax, rbx, rcx, rdx
- **32-bit registers:** eax, ebx, ecx, edx
  - Are actually the low order 32 bits of the 64-bit registers
- **16-bit registers:** ax, bx, cx, dx
  - Are the low order 16 bits of the 32-bit registers eax, ebx, ecx, edx

Example - Assembler: Registers

EAX=A9DC81F5

8-bit registers: ah, al, bh, bl, ch, cl, dh, dl: are bits 8-15 and 0-7 of the 16-bit registers ax, bx, cx, dx

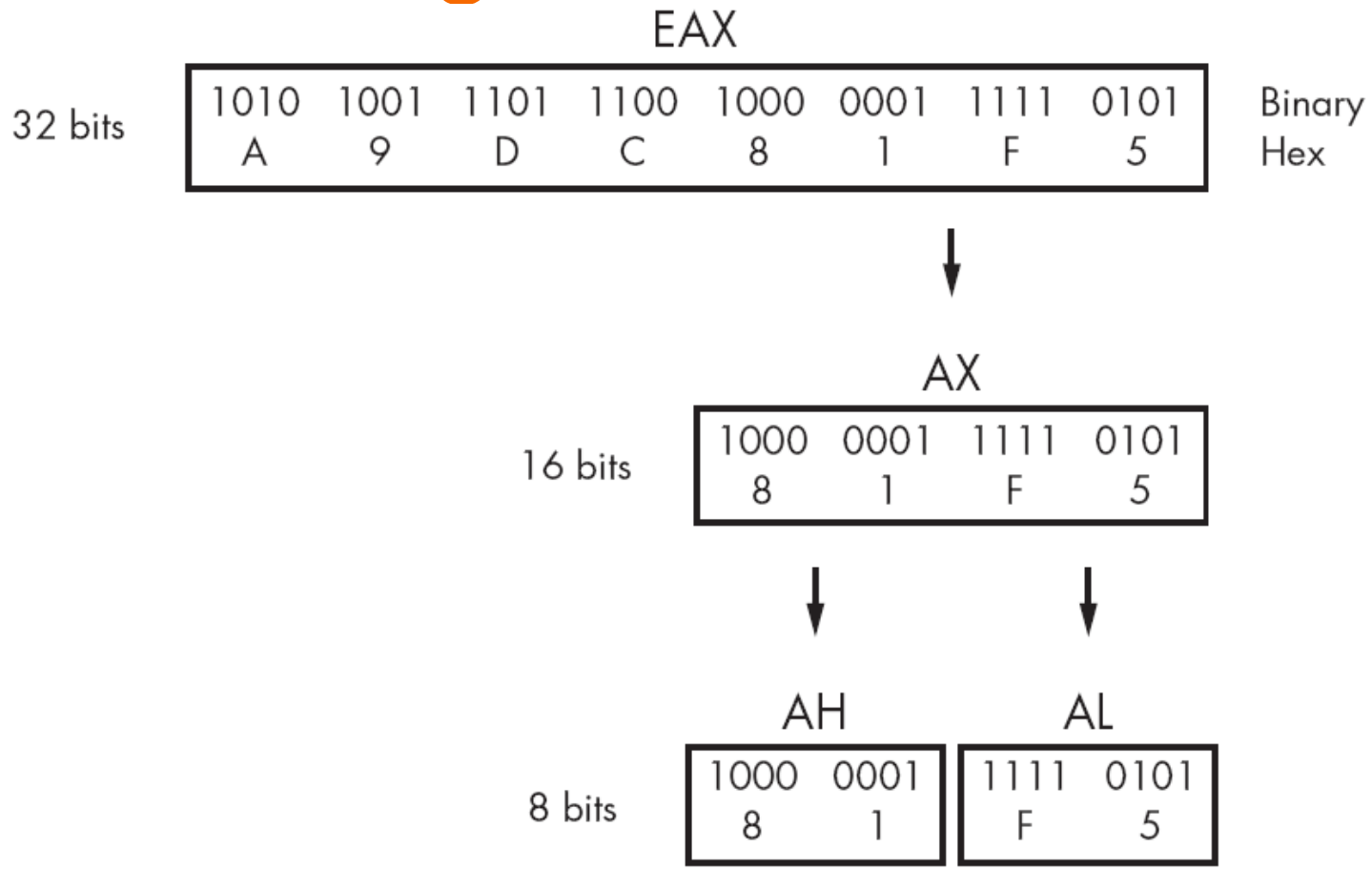
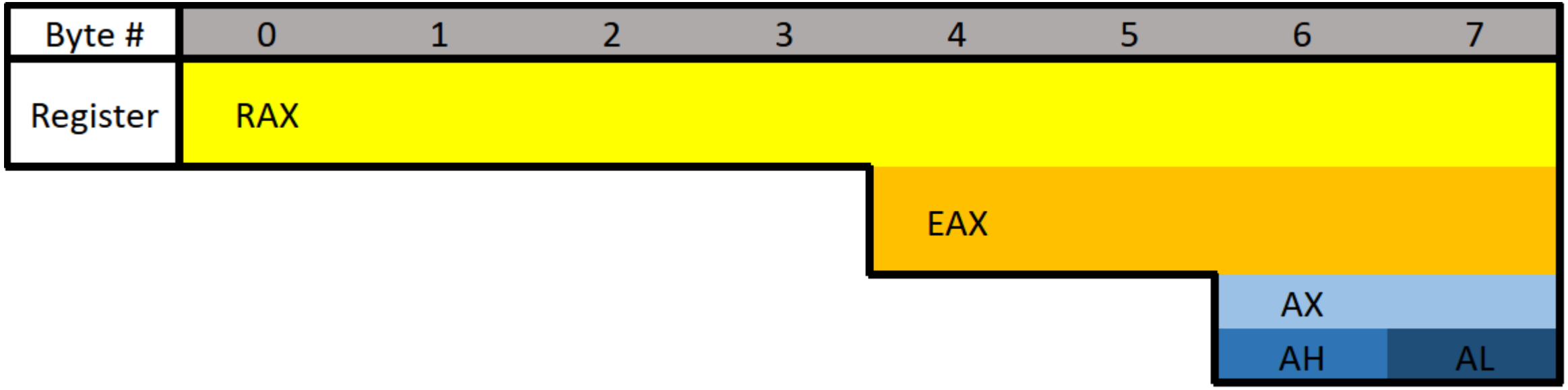


Figure 4-4: x86 EAX register breakdown

Assembler: Registers - Break Downs



Assembler: Registers - Break Downs

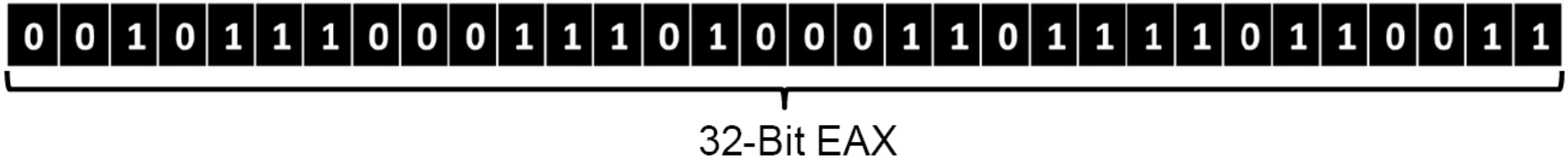
- 64-bit mode introduces an additional 8 new 64-bit general purpose registers R8 – R15

- R8 - Full 64-bit (8 bytes) register.
- R8D - Lower double word (4 bytes).
- R8W - Lower word (2 bytes)
- R8B - Lower byte.

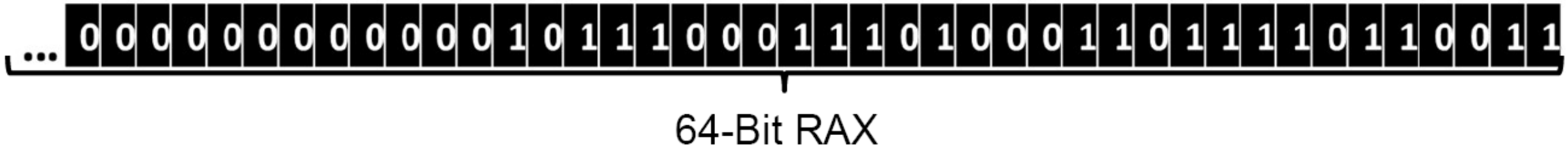
- No “h” mode for these (i.e., no direct access to bits **8-15**)

8/16/32/64-Bit Madness

- Consider storing the number 775,567,283 in the EAX register:

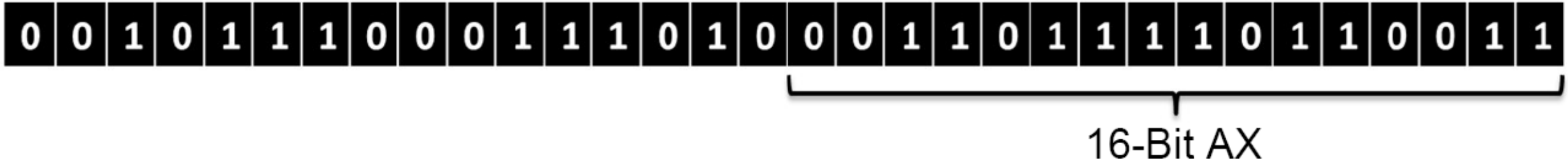


- If you read this from RAX, it still equals 775,567,283:

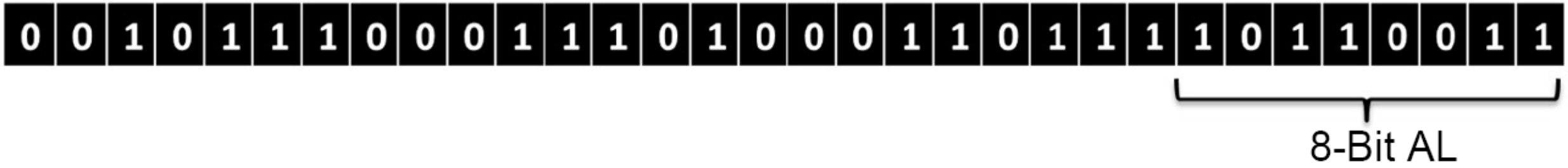


8/16/32/64-Bit Madness

- If you read this from AX, it equals 14,259:



- If you read this from AL, it equals 179 or **-77** (signed integer, read up on 2's compliment)



EFLAGS

Control Flags (how instructions are carried out):

DF: Direction	IF: Interrupt	TF: Trap Flag
Controls the direction of string operations.1 means string operations process down from high to low address	Whether interrupts can occur, 1 means enabled	Allows users to single-step through programs.

EFLAGS

Control Flags (how instructions are carried out):

Status Flags (result of operations):

CF: Carry result of unsigned op. is too large or below zero**. 1= carry/borrow**

OF: Overflow result of signed op. is too large or small. **1=overflow/underflow**

SF: Sign of result. Reasonable for Integer only. **1= neg. /0= pos.**

ZF: Zero result of operation is zero. {**It is 1 when the result is zero**}

AF: Auxiliary carry similar to Carry but restricted to **the low nibble only**

PF: Parity 1= result has **even number of set bits**

Yes, More

- Also control registers that support, e.g.,
  - processor features,
  - the debugging and
  - virtualization architectures

- CR0 – CR8 (see Intel manuals, debugging architecture)

We'll go over each flag and register when we need to, as we're looking at them in malware.

Intel manuals are a great resource for how these different processor features work.

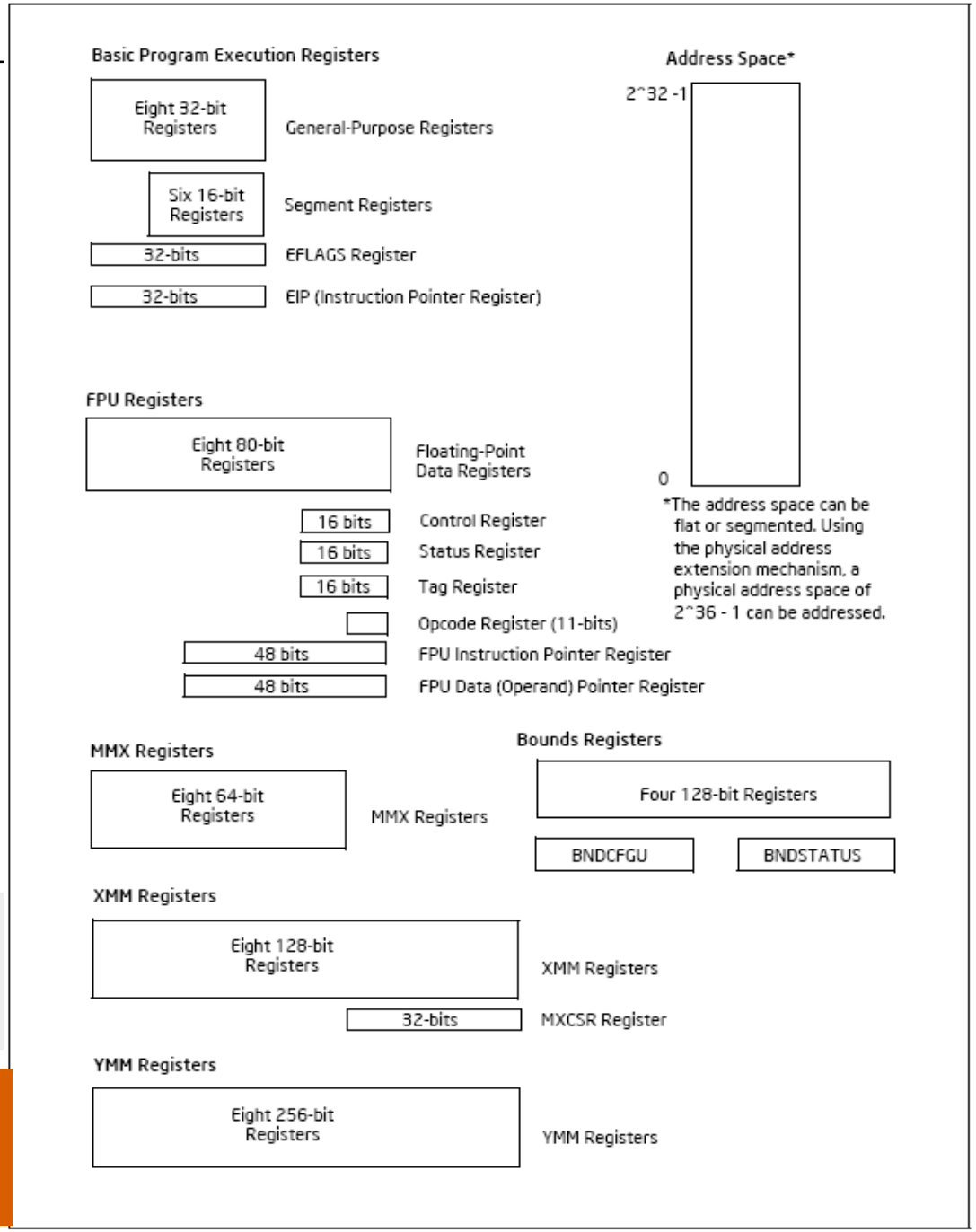
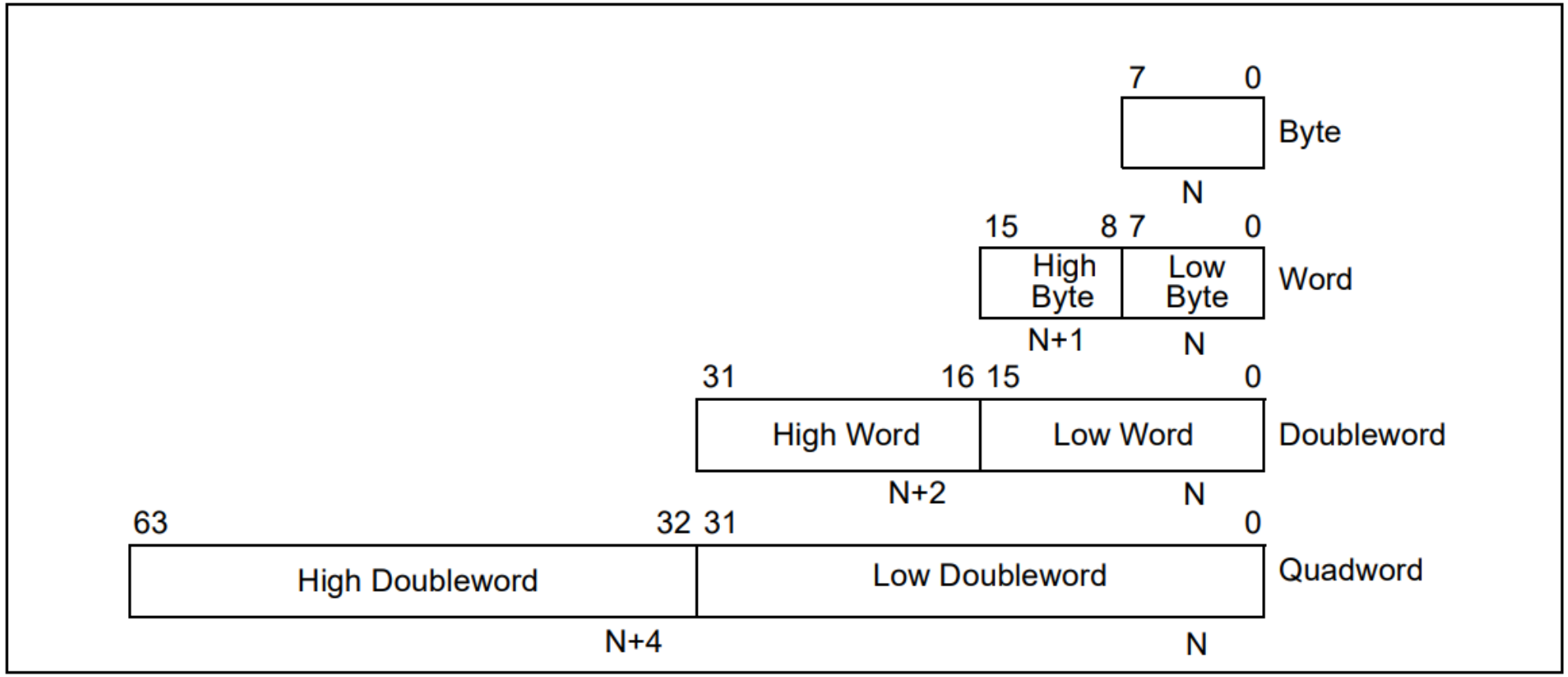


Figure 3-1. IA-32 Basic Execution Environment for Non-64-Bit Modes

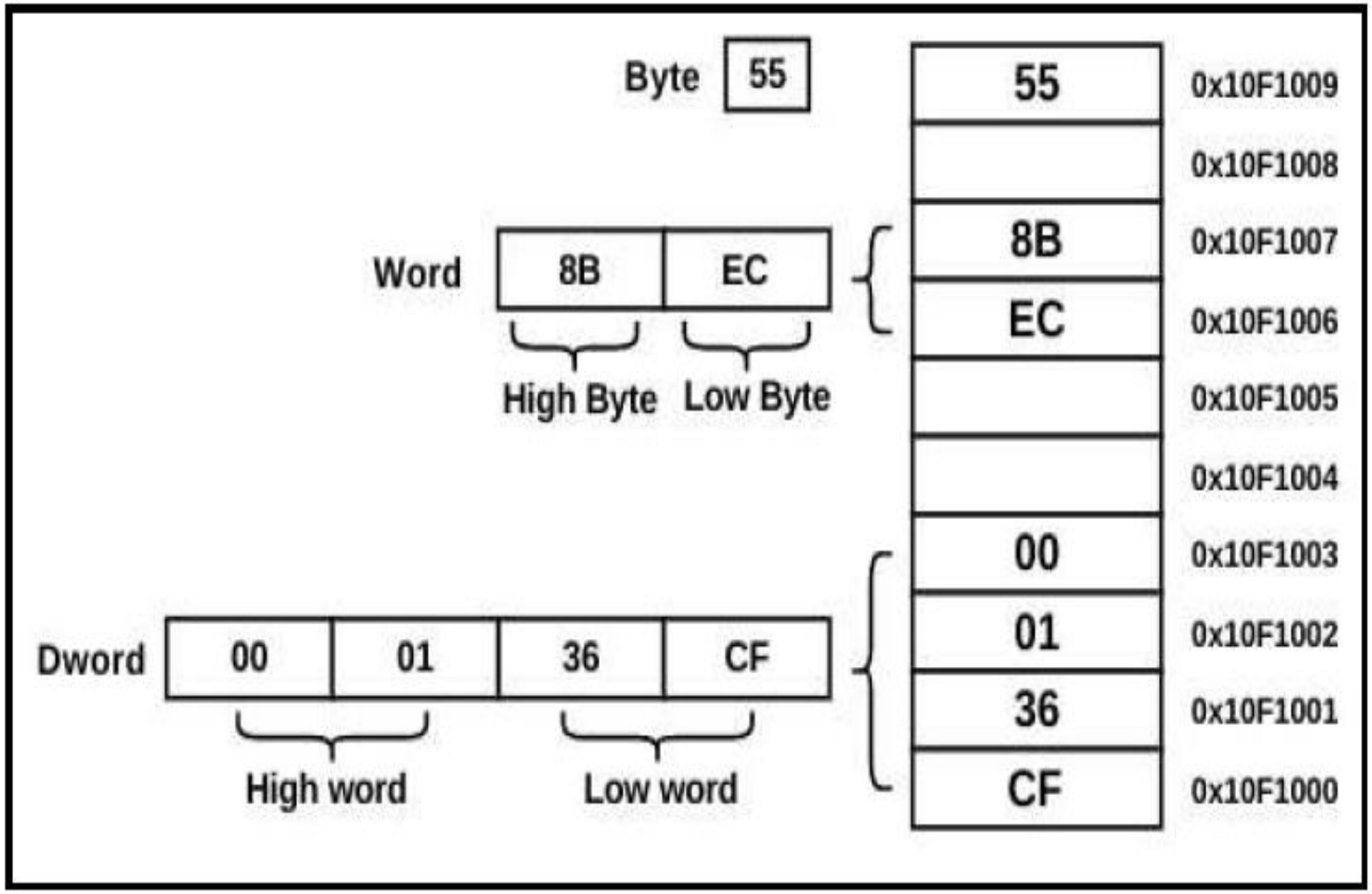
Fundamental Data Types



Modern processors understand bytes, which are essentially 8 bits. A word consists of two bytes. Similarly, a double word is made up of two words, and a quad word comprises two double words, reaching up to 64 bits in size. These are the only data types that modern processors recognize, with everything being composed of bytes, words, double words, and quad words

Endianness: How Data Resides In Memory

x86 architecture uses the little-endian format. In memory, the data is stored in the little-endian format; that is, a low-order byte is stored at the lower address, and subsequent bytes are stored in successively higher addresses in the memory.



Instructions Formats

[LABEL:] **INSTRUCTION** destop **[, sourceop] [; comment]**

- e.g.:

```
HERE: cmp ebx, 0xBEEF ; does ebx contain magic #?
push ebx
push eax
xor ebx, ebx          ; ebx = 0
xor eax, eax          ; eax = 0
```

[ ] = Optional

Data Addressing Modes

- **Immediate**
  - Value is a constant

```
mov rax, 0xBEEF ; store 0xBEEF in rax
```

- **Register addressing**
  - Use value in register

```
mov eax, ebx  ; store the value held in ebx into eax
```

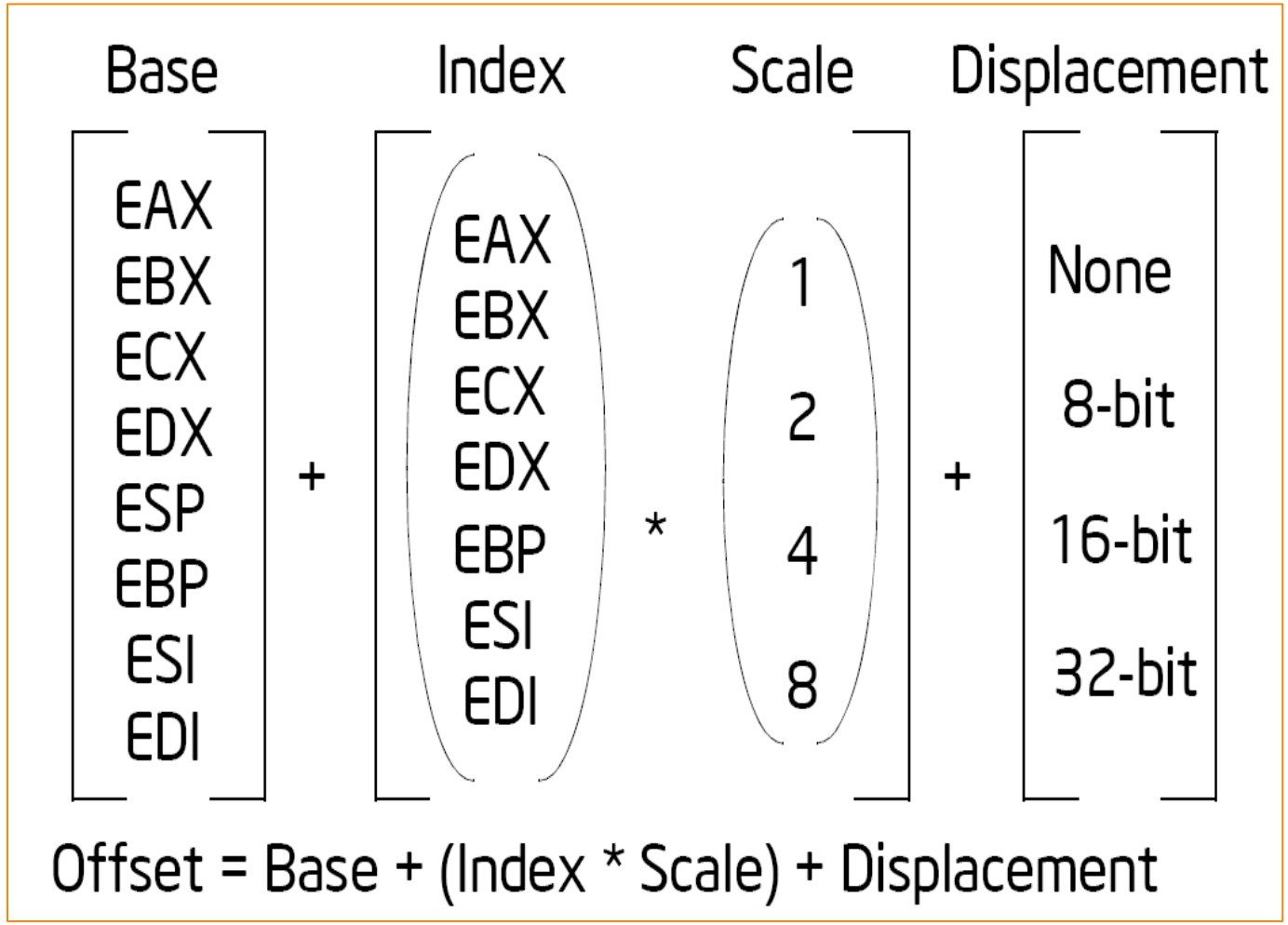
- **Memory to register or Register to memory: YES**
  - Indexed / Memory operands (next slide)
- **Memory to Memory: NO!**

Data Addressing Modes

```
mov 100[ebx+4*ecx], eax
```

Data Addressing Modes

- **Base** — The value in a general-purpose register.
- **Index** — The value in a general-purpose register.
- **Scale factor** — A value of 2, 4, or 8 that is multiplied by the index value.
- **Displacement** — An 8, 16, or 32-bit value.



The offset which results from adding these components is called an **effective address.**

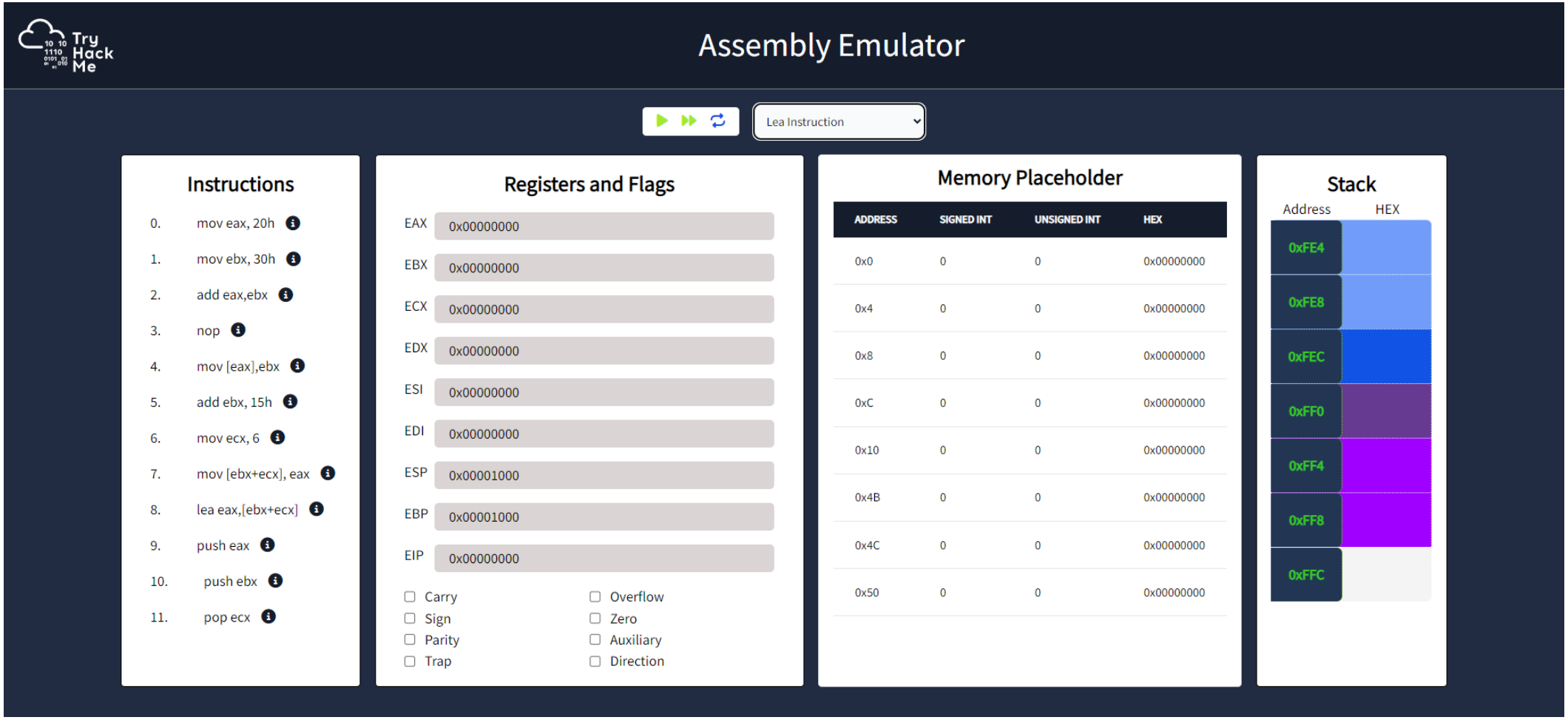
Data Addressing Modes

Offset = Base + (Index * Scale) + Displacement

ebx = 0x47EFFFFA

```
1)  mov BYTE PTR [ebx], 0       ; address in ebx (as a byte) <- 0
2)  mov DWORD PTR [ebx], 0 ; address in ebx (as a 32-bit) <- 0
3)  mov al, BYTE PTR FOO        ; set al to the byte pointed to by FOO
4)  mov [ebx+8*ecx], eax         ; address in ecx + 8*ebx <- eax
5)  mov [ebx+4*ecx + 100], eax
 or mov 100[ebx+4*ecx], eax    ; address in ecx + 4*ebx + 100 <- eax
```

Data Addressing Modes



https://static-labs.tryhackme.cloud/sites/assembly-emulator/

Data Section: Allocating Storage

friend: BYTE “**joe**”
friend: BYTE '**j**', '**o**', '**e**'

} Same effect

Data Section: Allocating Storage

```
friend: BYTE  “joe”
friend: BYTE  'j',  'o',  'e'
```

Same effect

```
gross: DWORD 144
gross: DWORD 12*12
gross: DWORD 10*15-7+1
gross: DWORD 90h          ; hex
gross: DWORD 10010000b    ; binary
gross : DWORD 220o        ; octal
```

Same effect

Data Section: Allocating Storage

```
friend: BYTE  “joe”
friend: BYTE  'j',  'o',  'e'
```

Same effect

```
gross: DWORD 144
gross: DWORD 12*12
gross: DWORD 10*15-7+1
gross: DWORD 90h              ; hex
gross: DWORD 10010000b        ; binary
gross : DWORD 220o            ; octal
```

Same effect

```
values: DWORD 10, 20, 30, 40          ; (4) 32-bit values
```

Data Section: Allocating Storage

```
friend: BYTE  "joe"
friend: BYTE  'j', 'o', 'e'
```

Same effect

```
gross: DWORD 144
gross: DWORD 12*12
gross: DWORD 10*15-7+1
gross: DWORD 90h          ; hex
gross: DWORD 10010000b    ; binary
gross : DWORD 220o        ; octal
```

Same effect

```
values: DWORD 10, 20, 30, 40        ; (4) 32-bit values
bigval: QWORD 99999999999           ; 64 bit
```

Back to hello.c

Let's see how much we understand now.☺

```
    .file   "hello.c"
    .intel_syntax noprefix
    .text
    .section    .rodata
.LC0:
    .string "Hello %s\n"
    .text
    .globl  main
    .type   main, @function
main:
    lea ecx, [esp+4]
    and esp, -16
    push    DWORD PTR [ecx-4]
    push    ebp
    mov ebp, esp
    push    ecx
    sub esp, 4
    mov eax, ecx
    cmp DWORD PTR [eax], 2
    jne .L2
    mov eax, DWORD PTR [eax+4]
    add eax, 4
    mov eax, DWORD PTR [eax]
    sub esp, 8
    push    eax
    push    OFFSET FLAT:.LC0
    call    printf
    add esp, 16
.L2:
    mov eax, 0
    mov ecx, DWORD PTR [ebp-4]
    leave
    lea esp, [ecx-4]
    ret
    .size   main, .-main
    .ident  "GCC: (Debian 13.2.0-7) 13.2.0"
    .section    .note.GNU-stack,"",@progbits
```

Back to hello.c

Let's see how much we understand now.☺

```
main:
    lea ecx, [esp+4]
    and esp, -16
    push    DWORD PTR [ecx-4]
    push    ebp
    mov ebp, esp
```

Back to hello.c

Let's see how much we understand now.☺

```
    cmp DWORD PTR [eax],2
    jne .L2
    mov eax, DWORD PTR [eax+4]
    add eax, 4
    mov eax, DWORD PTR [eax]
    sub esp, 8
    push    eax
    push    OFFSET FLAT:.LC0
    call    printf
    add esp, 16
.L2:
    mov eax, 0
    mov ecx, DWORD PTR [ebp-4]
    leave
    lea esp, [ecx-4]
    ret
```

3) Executable Files

Executable Files Have Sections

- The **.section** directive is used like this:

- **.section name [, "flags"[, @type[,flag_specific_arguments]]]**

- **a** - section is allocatable
- **e** - section is excluded from executable and shared library
- **w** - section is writable
- **x** - section is executable
- **M** - section is mergeable
- **S** - contains zero terminated strings
- **G** - section is a member of a section group
- **T** - section is used for thread-local-storage
- **?** - section is a member of the section's group, if any

Executable Files Have Sections

- The **.section** directive is used like this:

- **.section name [, "flags"[, @type[,flag_specific_arguments]]]**

@progbits section contains data

@nobits section w/o data (i.e., only occupies space)

@note section contains non-program data

@init_array section contains an array of ptrs to init functions

@fini_array section contains an array of ptrs to finish functions

@preinit_array section contains an array of ptrs to pre-init functions

Back to hello.c

Let's see how much we understand now.☺

```
    .file   "hello.c"
    .intel_syntax noprefix
    .text
    .section    .rodata
.LC0:
    .string "Hello %s\n"
    .text
    .globl  main
    .type   main, @function
main:
    lea ecx, [esp+4]
    and esp, -16
    push    DWORD PTR [ecx-4]
    push    ebp
    mov ebp, esp
    push    ecx
    sub esp, 4
    mov eax, ecx
    cmp DWORD PTR [eax], 2
    jne .L2
    mov eax, DWORD PTR [eax+4]
    add eax, 4
    mov eax, DWORD PTR [eax]
    sub esp, 8
    push    eax
    push    OFFSET FLAT:.LC0
    call    printf
    add esp, 16
.L2:
    mov eax, 0
    mov ecx, DWORD PTR [ebp-4]
    leave
    lea esp, [ecx-4]
    ret
    .size   main, .-main
    .ident  "GCC: (Debian 13.2.0-7) 13.2.0"
    .section    .note.GNU-stack,"",@progbits
```

Back to hello.c

Let's see how much we understand now.☺

```
    .file   "hello.c"
    .intel_syntax noprefix
    .text
    .section    .rodata
.LC0:
    .string "Hello %s\n"
    .text
    .globl  main
    .type   main, @function
main:
    lea ecx, [esp+4]
    and esp, -16
    push    DWORD PTR [ecx-4]
```

Common Intel Instructions

TRANSFER				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
MOV	Move (copy)	MOV Dest,Source	Dest:=Source									
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2 , Op2:=Op1									
STC	Set Carry	STC	CF:=1									1
CLC	Clear Carry	CLC	CF:=0									0
CMC	Complement Carry	CMC	CF:= ¬CF									±
STD	Set Direction	STD	DF:=1 (string op's downwards)		1							
CLD	Clear Direction	CLD	DF:=0 (string op's upwards)		0							
STI	Set Interrupt	STI	IF:=1			1						
CLI	Clear Interrupt	CLI	IF:=0			0						
PUSH	Push onto stack	PUSH Source	DEC SP, [SP]:=Source									
PUSHF	Push flags	PUSHF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL									
PUSHA	Push all general registers	PUSHA	AX, CX, DX, BX, SP, BP, SI, DI									
POP	Pop from stack	POP Dest	Dest:=[SP], INC SP									
POPF	Pop flags	POPF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL	±	±	±	±	±	±	±	±	±
POPA	Pop all general registers	POPA	DI, SI, BP, SP, BX, DX, CX, AX									
CBW	Convert byte to word	CBW	AX:=AL (signed)									
CWD	Convert word to double	CWD	DX:AX:=AX (signed)	±				±	±	±	±	±
CWDE	Conv word extended double	CWDE 386	EAX:=AX (signed)									
IN *i*	Input	IN Dest, Port	AL/AX/EAX := byte/word/double of specified port									
OUT *i*	Output	OUT Port, Source	Byte/word/double of specified port := AL/AX/EAX									

Be Very Careful!

- **These slides are NOT meant to exhaustively teach you everything about assembly!**

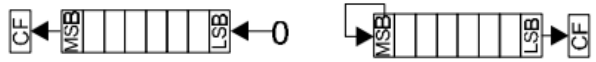
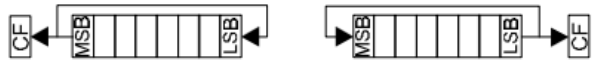
- You must read the Intel manuals & online references when reverse engineering
- For example:

mov eax, ebx

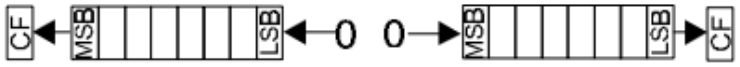
- On a 64-bit CPU?
- … automatically zeroes the upper 32 bits of RAX!

- **The manual says:**
  - 64-bit operands generate a 64-bit result in the destination 64-bit register
  - 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination register
  - 8-bit and 16-bit operands generate an 8-bit or 16-bit result in the destination 64-bit register
    - **Upper 56 bits or 48 bits (respectively) of the destination 64-bit register are left intact!**

Common Intel Instructions (2)

ARITHMETIC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
ADD	Add	ADD Dest,Source	Dest:=Dest+Source	±				±	±	±	±	±
ADC	Add with Carry	ADC Dest,Source	Dest:=Dest+Source+CF	±				±	±	±	±	±
SUB	Subtract	SUB Dest,Source	Dest:=Dest-Source	±				±	±	±	±	±
SBB	Subtract with borrow	SBB Dest,Source	Dest:=Dest-(Source+CF)	±				±	±	±	±	±
DIV	Divide (unsigned)	DIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
DIV	Divide (unsigned)	DIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
DIV 386	Divide (unsigned)	DIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
IDIV 386	Signed Integer Divide	IDIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
MUL	Multiply (unsigned)	MUL Op	Op=byte: AX:=AL*Op if AH=0 ♦	±				?	?	?	?	±
MUL	Multiply (unsigned)	MUL Op	Op=word: DX:AX:=AX*Op if DX=0 ♦	±				?	?	?	?	±
MUL 386	Multiply (unsigned)	MUL Op	Op=double: EDX:EAX:=EAX*Op if EDX=0 ♦	±				?	?	?	?	±
IMUL *i*	Signed Integer Multiply	IMUL Op	Op=byte: AX:=AL*Op if AL sufficient ♦	±				?	?	?	?	±
IMUL	Signed Integer Multiply	IMUL Op	Op=word: DX:AX:=AX*Op if AX sufficient ♦	±				?	?	?	?	±
IMUL 386	Signed Integer Multiply	IMUL Op	Op=double: EDX:EAX:=EAX*Op if EAX sufficient ♦	±				?	?	?	?	±
INC	Increment	INC Op	Op:=Op+1 (Carry not affected !)	±				±	±	±	±	
DEC	Decrement	DEC Op	Op:=Op-1 (Carry not affected !)	±				±	±	±	±	
CMP	Compare	CMP Op1,Op2	Op1-Op2	±				±	±	±	±	±
SAL	Shift arithmetic left (≡ SHL)	SAL Op,Quantity		*i*				±	±	?	±	±
SAR	Shift arithmetic right	SAR Op,Quantity		*i*				±	±	?	±	±
RCL	Rotate left through Carry	RCL Op,Quantity		*i*								±
RCR	Rotate right through Carry	RCR Op,Quantity		*i*								±
ROL	Rotate left	ROL Op,Quantity		*i*								±
ROR	Rotate right	ROR Op,Quantity		*i*								±

Common Intel Instructions (3)

LOGIC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
NEG	Negate (two-complement)	NEG Op	Op:=0-Op if Op=0 then CF:=0 else CF:=1	±				±	±	±	±	±
NOT	Invert each bit	NOT Op	Op:=¬Op (invert each bit)									
AND	Logical and	AND Dest,Source	Dest:=Dest∧Source	0				±	±	?	±	0
OR	Logical or	OR Dest,Source	Dest:=Dest∨Source	0				±	±	?	±	0
XOR	Logical exclusive or	XOR Dest,Source	Dest:=Dest (exor) Source	0				±	±	?	±	0
SHL	Shift logical left (≡ SAL)	SHL Op,Quantity		*i*				±	±	?	±	±
SHR	Shift logical right	SHR Op,Quantity		*i*				±	±	?	±	±

MISCELLANEOUS				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
NOP	No operation	NOP	No operation									
LEA	Load effective adress	LEA Dest,Source	Dest := address of Source									
INT	Interrupt	INT Nr	interrupts current program, runs spec. int-program			0	0					

Common Intel Instructions (4)

JUMPS (flags remain unchanged)							
Name	Comment	Code	Operation	Name	Comment	Code	Operation
CALL	Call subroutine	CALL Proc		RET	Return from subroutine	RET	
JMP	Jump	JMP Dest					
JE	Jump if Equal	JE Dest	(≡ JZ)	JNE	Jump if not Equal	JNE Dest	(≡ JNZ)
JZ	Jump if Zero	JZ Dest	(≡ JE)	JNZ	Jump if not Zero	JNZ Dest	(≡ JNE)
JCXZ	Jump if CX Zero	JCXZ Dest		JECXZ	Jump if ECX Zero	JECXZ Dest	386
JP	Jump if Parity (Parity Even)	JP Dest	(≡ JPE)	JNP	Jump if no Parity (Parity Odd)	JNP Dest	(≡ JPO)
JPE	Jump if Parity Even	JPE Dest	(≡ JP)	JPO	Jump if Parity Odd	JPO Dest	(≡ JNP)

Unsigned (Cardinal)				signed (Integer)			
JA	Jump if Above	JA Dest	(≡ JNBE)	JG	Jump if Greater	JG Dest	(≡ JNLE)
JAE	Jump if Above or Equal	JAE Dest	(≡ JNB ≡ JNC)	JGE	Jump if Greater or Equal	JGE Dest	(≡ JNL)
JB	Jump if Below	JB Dest	(≡ JNAE ≡ JC)	JL	Jump if Less	JL Dest	(≡ JNGE)
JBE	Jump if Below or Equal	JBE Dest	(≡ JNA)	JLE	Jump if Less or Equal	JLE Dest	(≡ JNG)
JNA	Jump if not Above	JNA Dest	(≡ JBE)	JNG	Jump if not Greater	JNG Dest	(≡ JLE)
JNAE	Jump if not Above or Equal	JNAE Dest	(≡ JB ≡ JC)	JNGE	Jump if not Greater or Equal	JNGE Dest	(≡ JL)
JNB	Jump if not Below	JNB Dest	(≡ JAE ≡ JNC)	JNL	Jump if not Less	JNL Dest	(≡ JGE)
JNBE	Jump if not Below or Equal	JNBE Dest	(≡ JA)	JNLE	Jump if not Less or Equal	JNLE Dest	(≡ JG)
JC	Jump if Carry	JC Dest		JO	Jump if Overflow	JO Dest	
JNC	Jump if no Carry	JNC Dest		JNO	Jump if no Overflow	JNO Dest	
				JS	Jump if Sign (= negative)	JS Dest	
				JNS	Jump if no Sign (= positive)	JNS Dest	

4) Intel vs. AT&T Syntax

Intel vs. AT&T Syntax

- There are actually two accepted representations for x86 assembly language
  - Intel Syntax and AT&T Syntax
- Virtually every tool for Windows uses Intel syntax
- gcc traditionally used AT&T syntax
  - Thus, virtually every tool for Linux uses AT&T syntax
- Can be overridden with appropriate switches
- **WE WILL USE BOTH**
  - So you need to be able to read either
- Here's a quick one slide cheat sheet…

Intel vs. AT&T Syntax

General Instructions:

Intel	AT&T	

push 4
add eax, 4
mov al, byte ptr FOO
call
jmp
ret
	pushl $4
	addl $4, %eax
	movb FOO, %al
	lcall
	ljmp
	lret
		• b = byte
		• s = short (16 bit int or 32-bit float)
		• w = word
		• l = long (32 bit int or 64-bit float)
		• q = quad
		• t = ten bytes

Intel vs. AT&T Syntax

Memory References:

Intel

```
displacement[base+index*scale]

mov eax, BYTE PTR [ebp-4]
mov ebx, BYTE PTR [foo+eax*4]
```

AT&T

```
displacement(base,index,scale)

movb -4(%ebp), %eax
movb foo(,%eax,4), %ebx
```

Note: Constants in memory references do not need a “$”

Listen to Drake

```
2   main:
3       leal    4(%esp), %ecx
4       andl    $-16, %esp
5       pushl   -4(%ecx)
6       pushl   %ebp
7       movl    %esp, %ebp
8       pushl   %ecx
9       subl    $4, %esp
10      movl    %ecx, %eax
11      cmpl    $2, (%eax)
12      jne .L2
13      movl    4(%eax), %eax
14      addl    $4, %eax
15      movl    (%eax), %eax
16      subl    $8, %esp
17      pushl   %eax
18      pushl   $.LC0
19      call    printf
20      addl    $16, %esp
```

```
2   main:
3           cmp edi, 2
4           je .L6
5           xor eax, eax
6           ret
7           .L6:
8           push rax
9           mov rdx, QWORD PTR [rsi+8]
10          mov edi, 1
11          mov esi, OFFSET FLAT:.LC0
12          xor eax, eax
13          call __printf_chk
14          xor eax, eax
15          pop rdx
16          ret
```

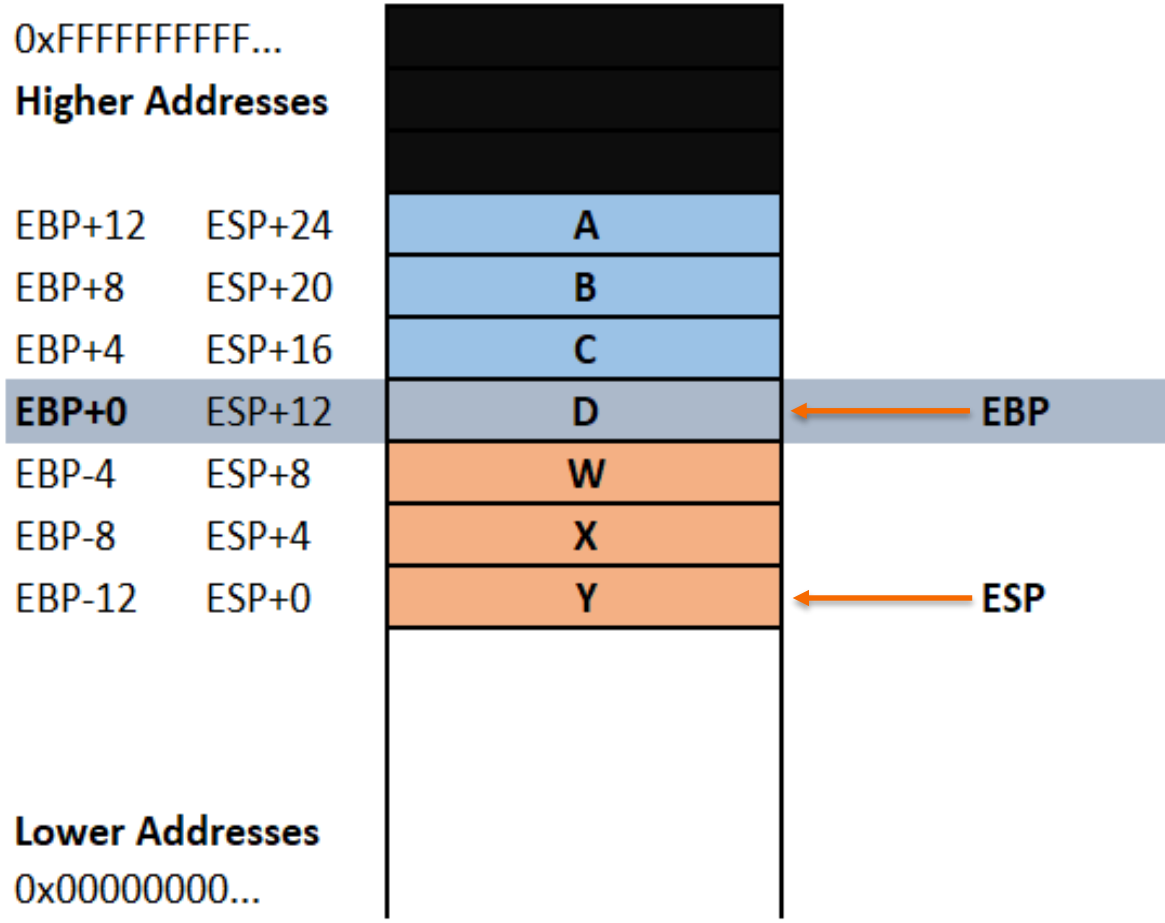
5) Memory …0x2BCON10U-ED

Now …

Memory Storage: Stack

The stack is considered an endless sequence of memory "slots"

- "Slots" are allocated in descending order!
- The EBP register points to the most recent "Stack Base Address"
- Stack base address is usually updated for each **function call**
- The ESP register points to the **bottom-most used** "slot"



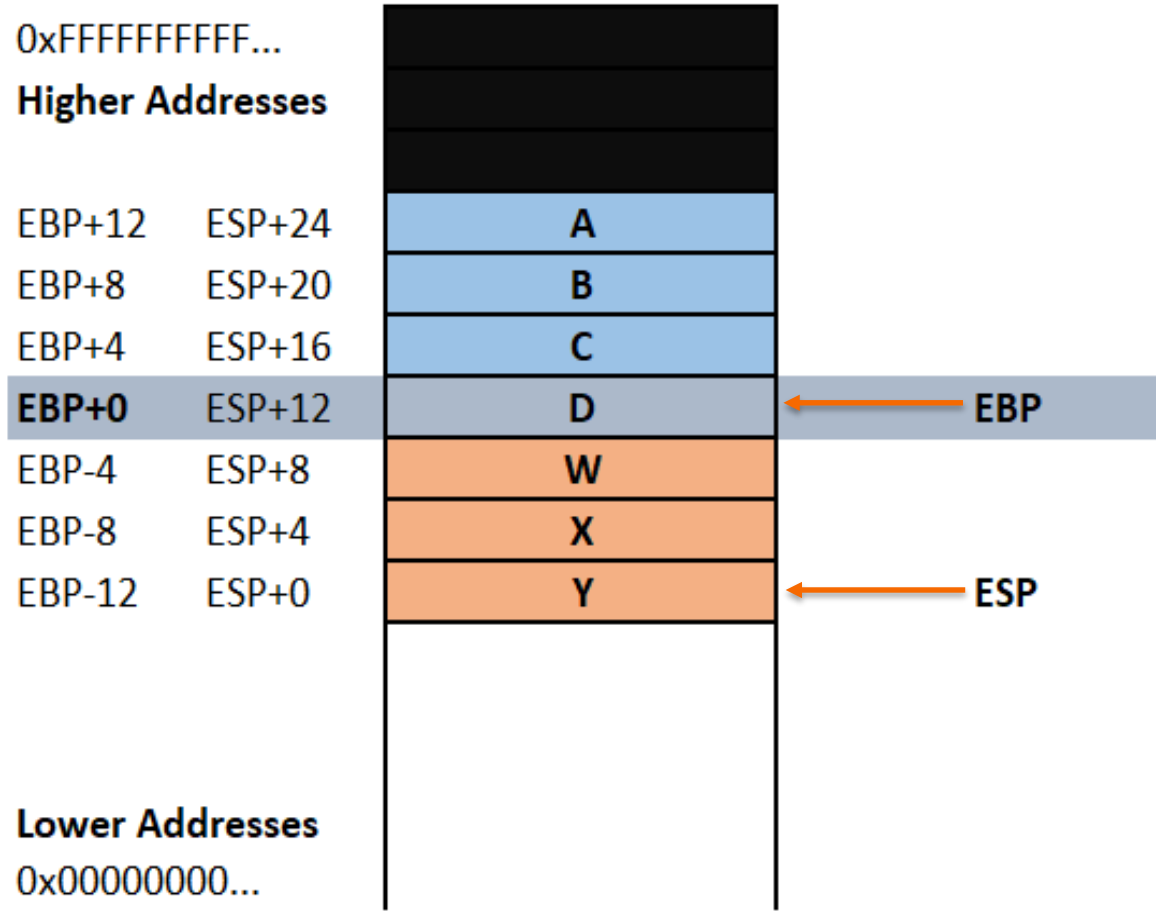
Memory Storage: Stack

Data (the size of a register) is pushed into a free "slot"

```
push eax
```

=

```
sub esp, 4
mov [esp], eax
```



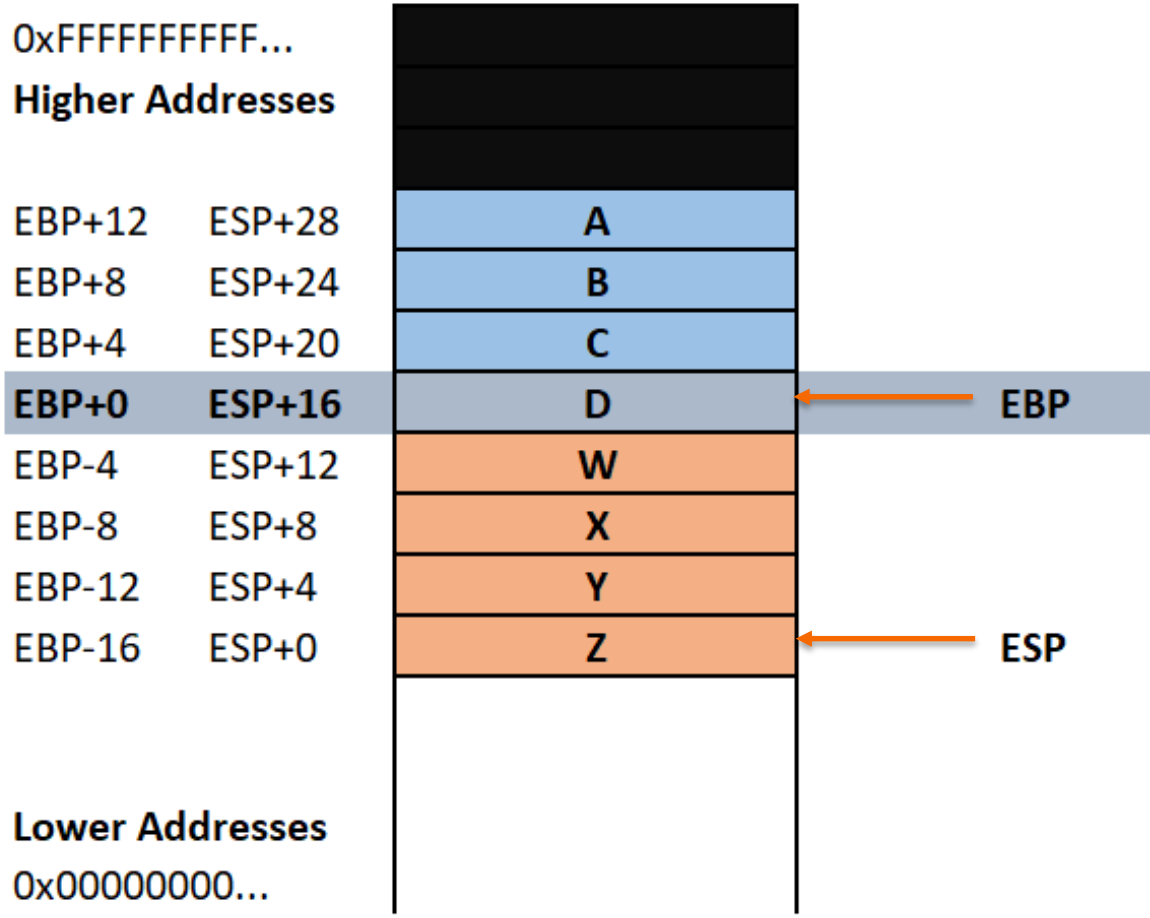
Memory Storage: Stack

Data (the size of a register) is pushed into a free "slot"

```
push eax
```

=

```
sub esp, 4
mov [esp], eax
```



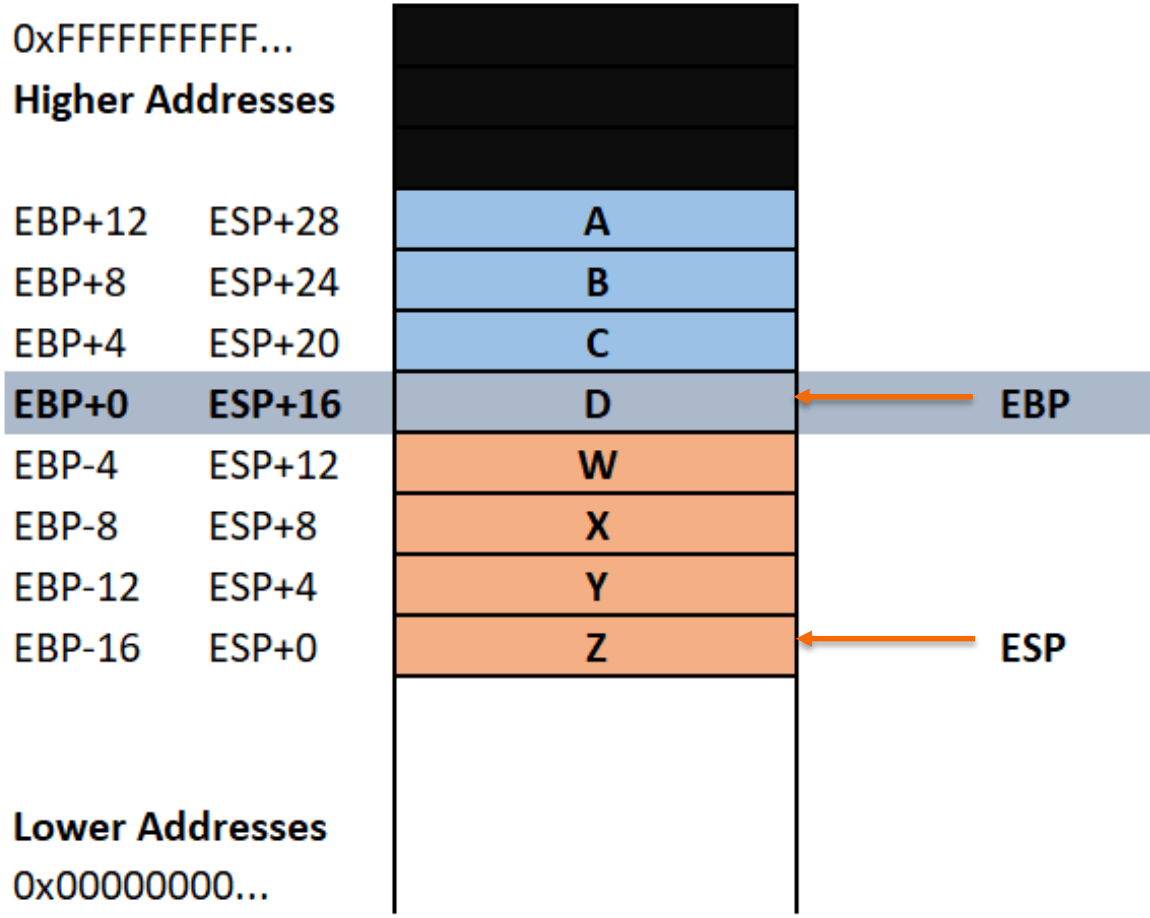
Memory Storage: Stack

Data (the size of a register) is pushed int a free "slot"

```
push eax
```
=
```
sub esp, 4
mov [esp], eax
```

Data (the size of a register) is popped from the "slot" pointed to by ESP

```
pop eax
```
=
```
mov eax, [esp]
add esp, 4
```



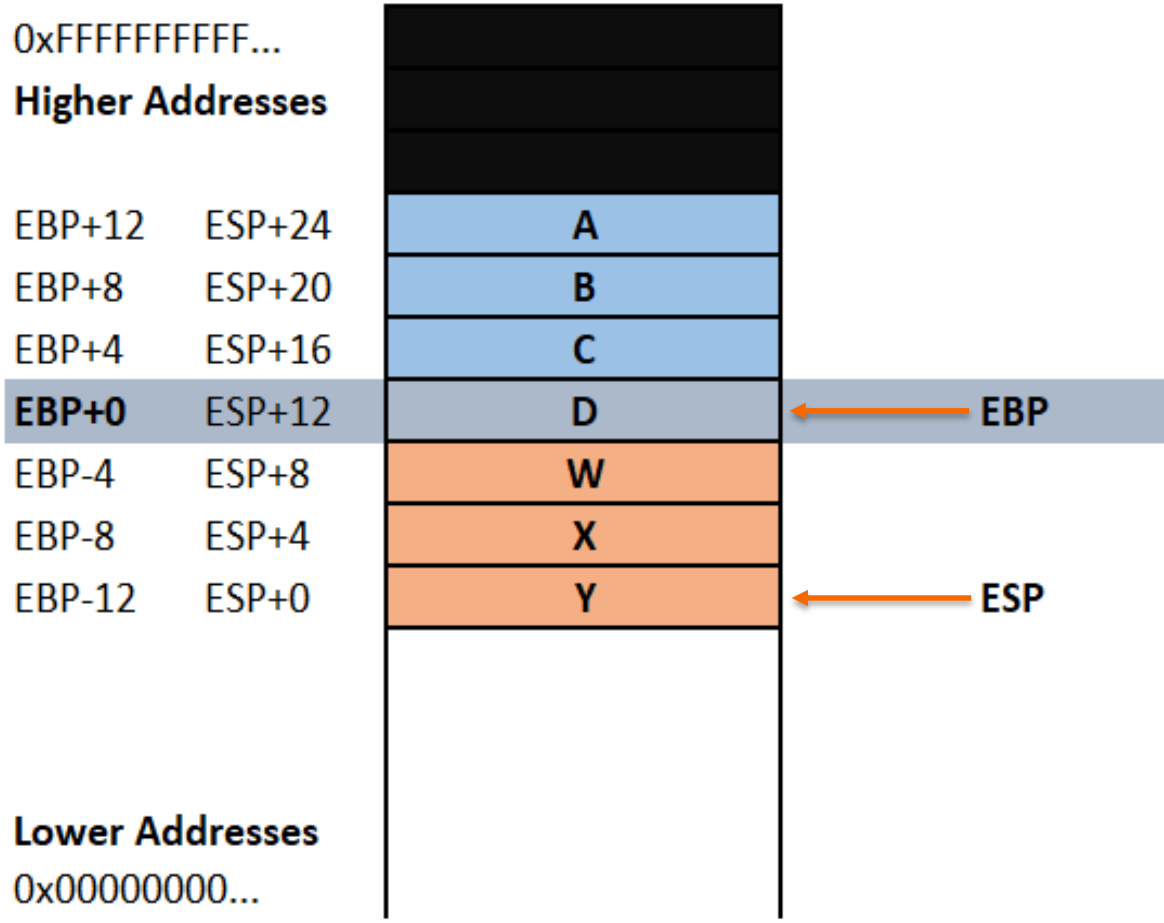
Memory Storage: Stack

Data (the size of a register) is pushed into a free "slot"

```
push eax
```
=
```
sub esp, 4
mov [esp], eax
```

Data (the size of a register) is popped from the "slot" pointed to by ESP

```
pop eax
```
=
```
mov eax, [esp]
add esp, 4
```



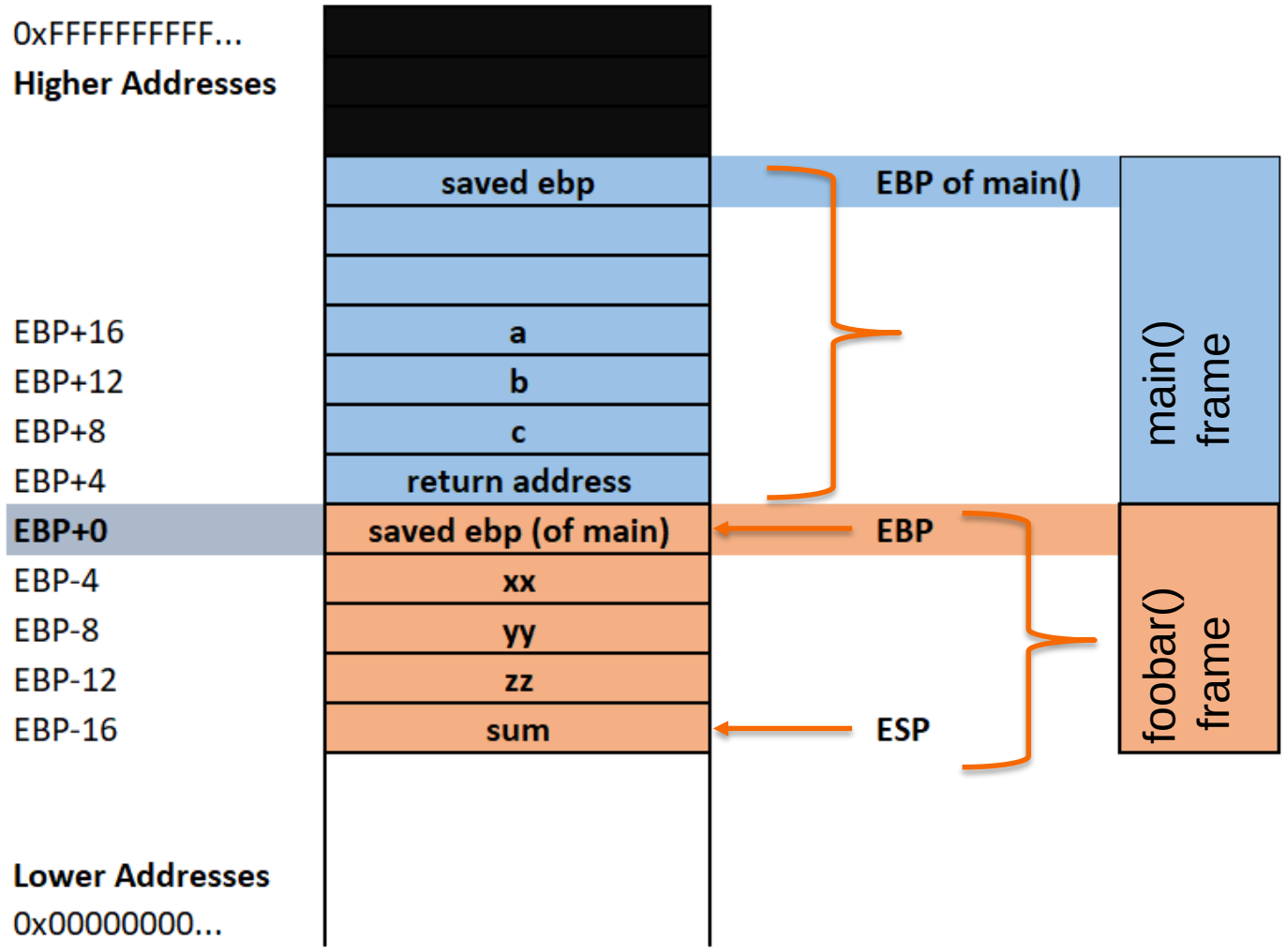
Stack and Function Call (16 & 32-bit)

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}
```

Stack and Function Call (16 & 32-bit)

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}
```

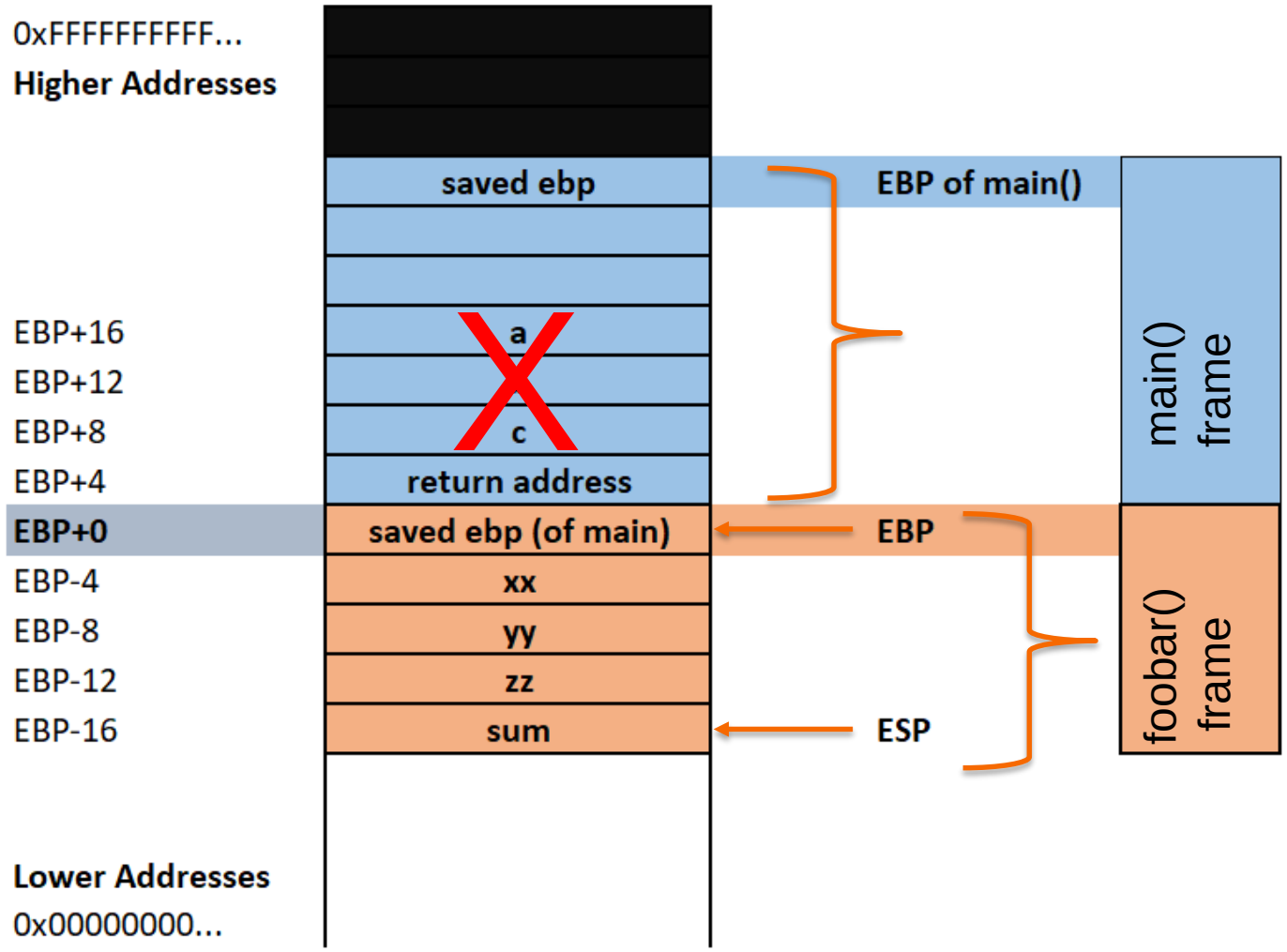
```
_foobar:
push ebp
mov ebp, esp
sub esp, 16
…
_main:
push ecx
push ebx
push eax
call _foobar
```



Stack and Function Call (16 & 32-bit)

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}
```

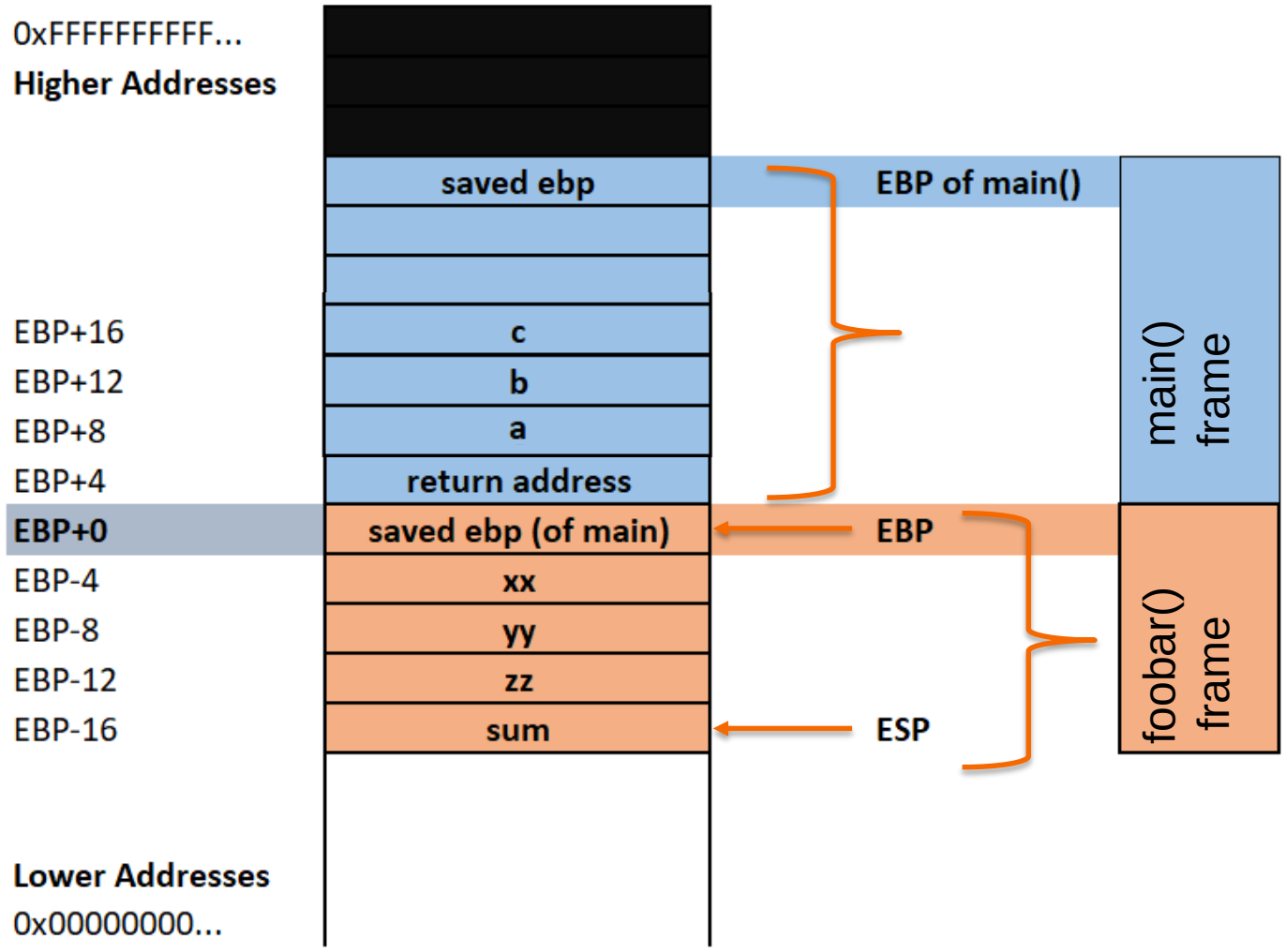
```
_foobar:
push ebp
mov ebp, esp
sub esp, 16
…
_main:
push ecx
push ebx
push eax
call _foobar
```



Stack and Function Call (16 & 32-bit)

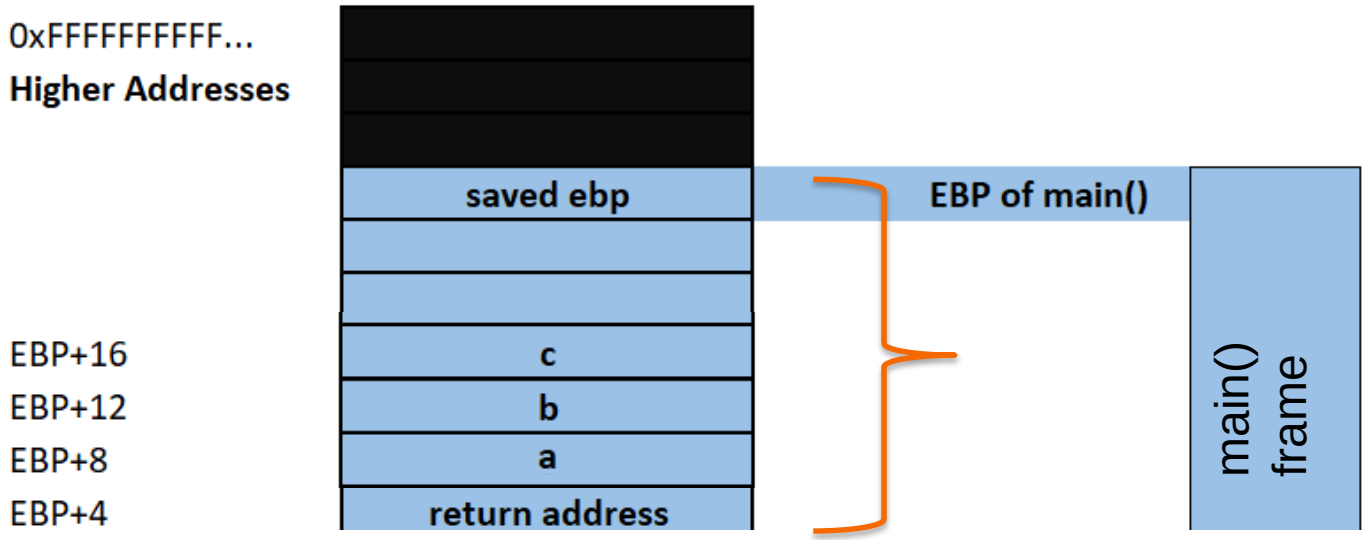
```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}
```

```
_foobar:
push ebp
mov ebp, esp
sub esp, 16
…
_main:
push ecx
push ebx
push eax
call _foobar
```



Stack and Function Call (16 & 32-bit)

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}
```




```
_foobar:
push ebp
mov ebp, esp
sub esp, 16
…
_main:
push ecx
push ebx
push eax
call _foobar
```

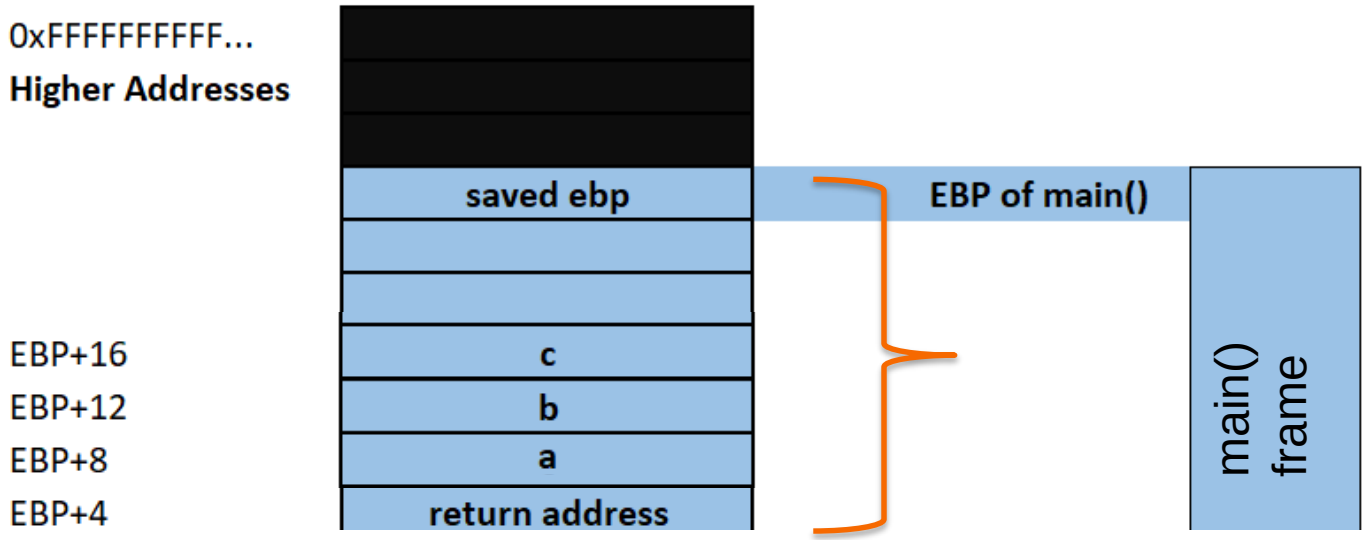
Call _foobar =

**push ebp + 5 *
Jmp _foobar**

- Push the return address or the address of the next instruction
- (* call _foobar is 5 bytes long)

Stack and Function Call (16 & 32-bit)

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}
```




```
_foobar:
push ebp
mov ebp, esp
sub esp, 16
…
_main:
push ecx
push ebx
push eax
call _foobar
```

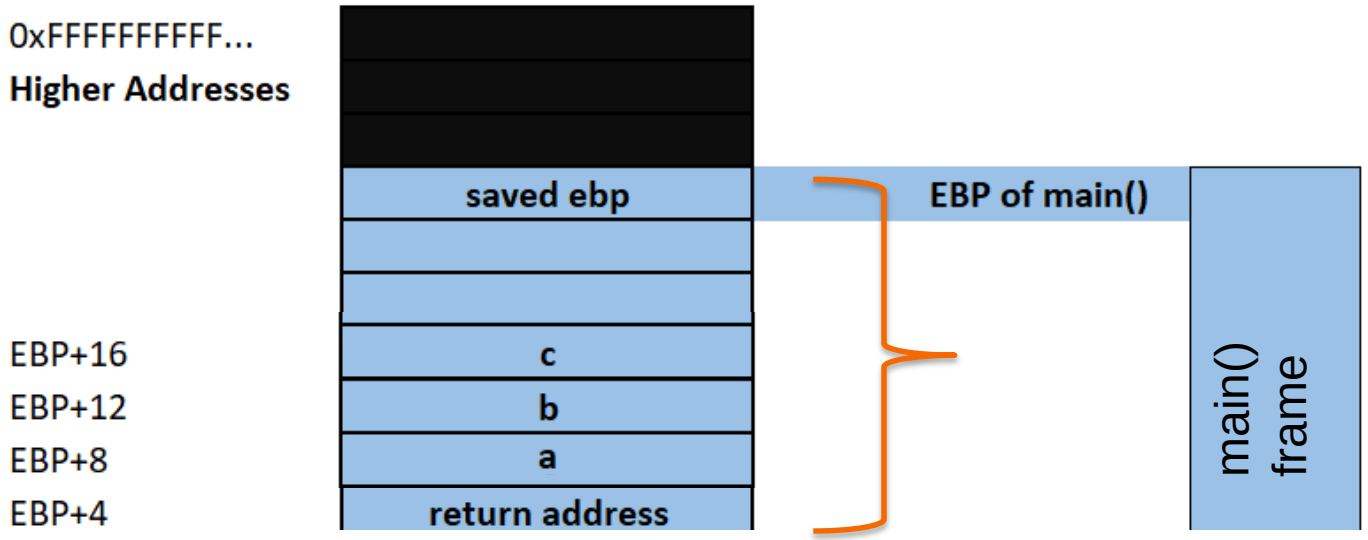
Call _foobar =

push ~~ebp~~ + 5 *
Jmp _foobar

- Push the return address or the address of the next instruction
- (* call _foobar is 5 bytes long)

Stack and Function Call (16 & 32-bit)

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}
```




```
_foobar:
push ebp
mov ebp, esp
sub esp, 16
…
_main:
push ecx
push ebx
push eax
call _foobar
```

Call _foobar =

**push eip + 5 *
Jmp _foobar**

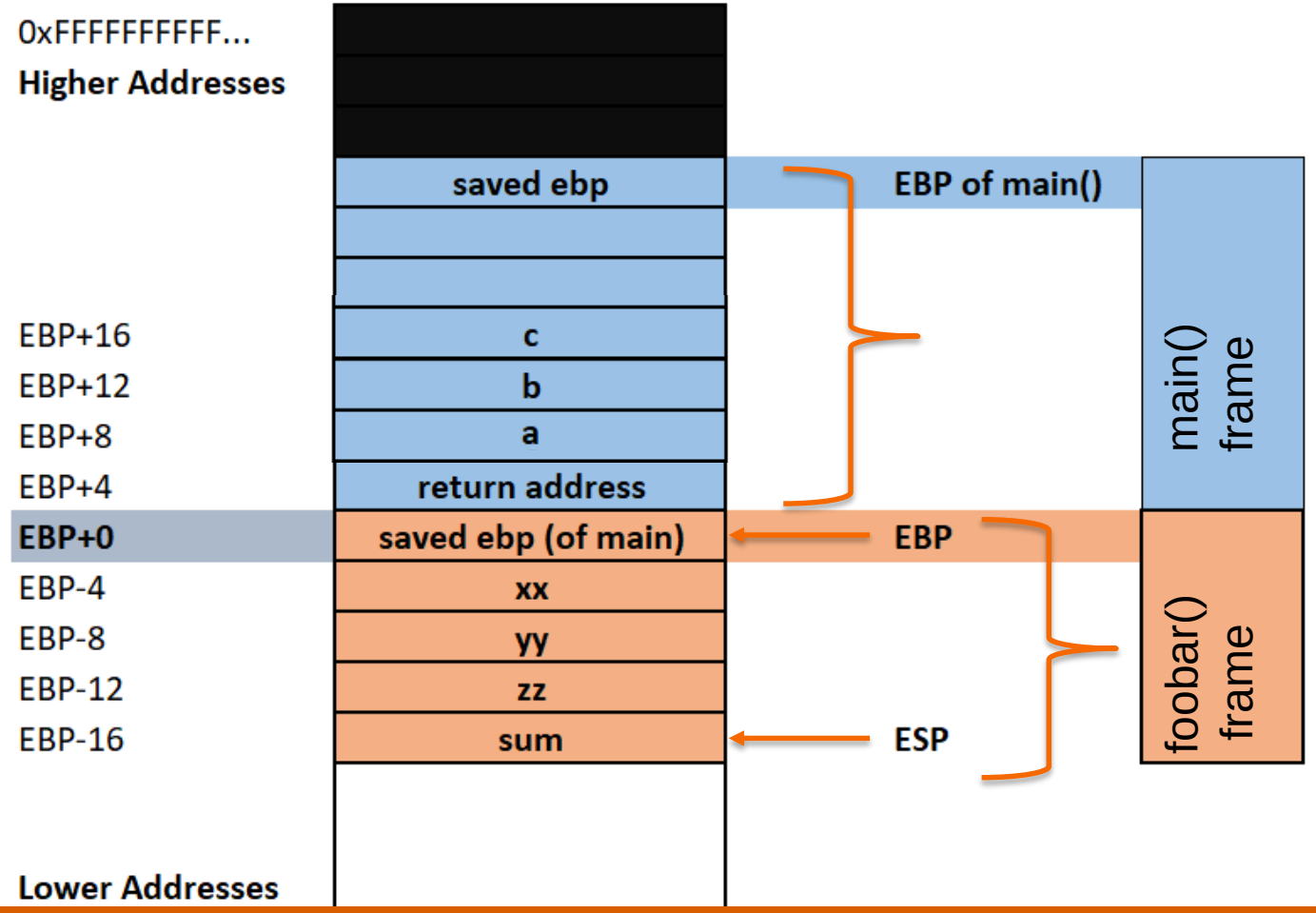
- Push the return address or the address of the next instruction
- (* call _foobar is 5 bytes long)

Stack and Function Call (16 & 32-bit)

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}
```

```
_foobar:
push ebp
mov ebp, esp
sub esp, 16
…
_main:
push ecx
push ebx
push eax
call _foobar
```

Referred to as the "function prologue"



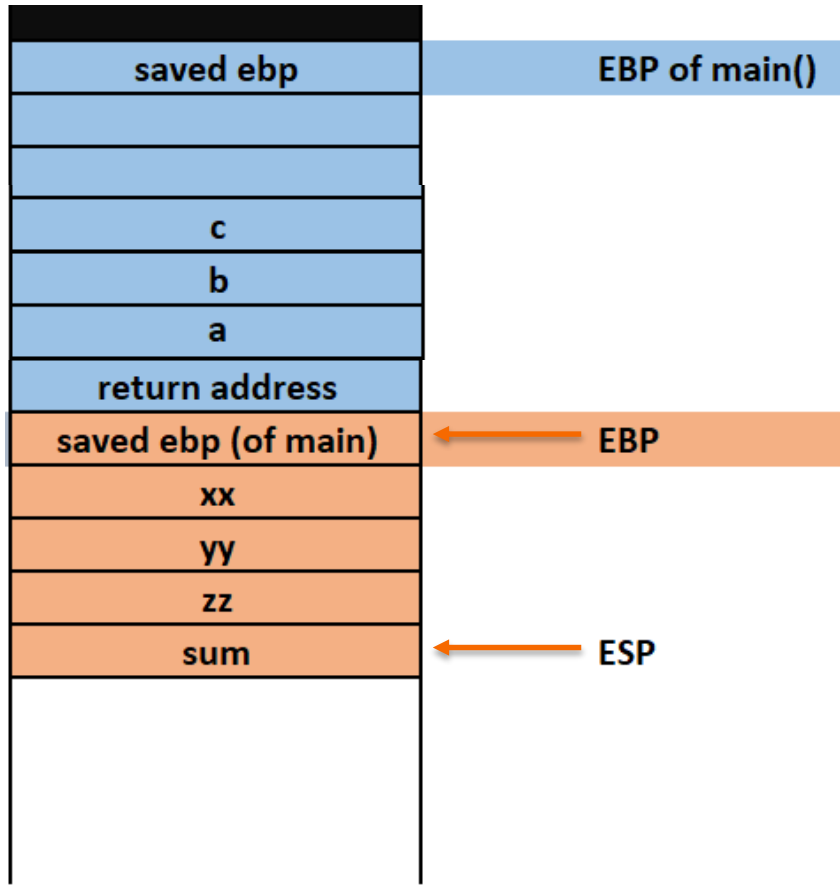
The prologue saves EBP, builds a new stack frame by saving ESP to EBP, and creates space for local variables.

Function Return (Same on Both 32- & 64-Bit)

The instruction ret performs a function return

- Prior to executing a ret instruction:
  1. The return value must be stored in EAX
  2. The stack must be cleaned up!

ret = pop ebp



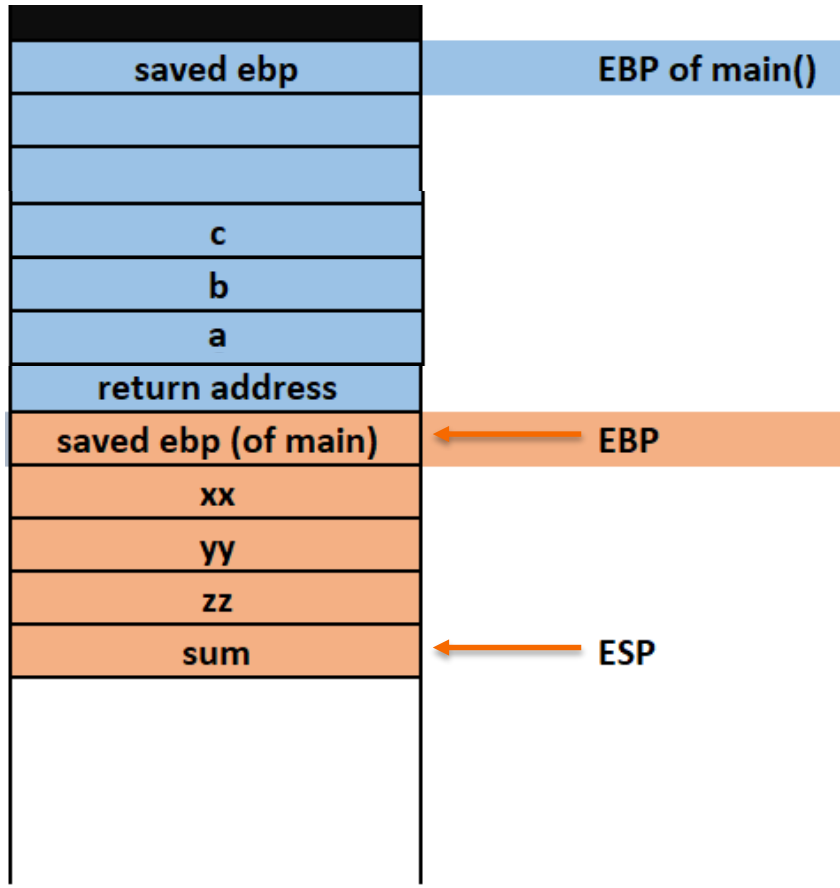
Function Return (Same on Both 32- & 64-Bit)

The instruction ret performs a function return

- Prior to executing a ret instruction:
  1. The return value must be stored in EAX
  2. The stack must be cleaned up!

ret = pop ebp

```
int foobar(int a, int b, int c)
{
    return xx * yy * zz + sum;
}
```



Function Return (Same on Both 32- & 64-Bit)

The instruction ret performs a function return

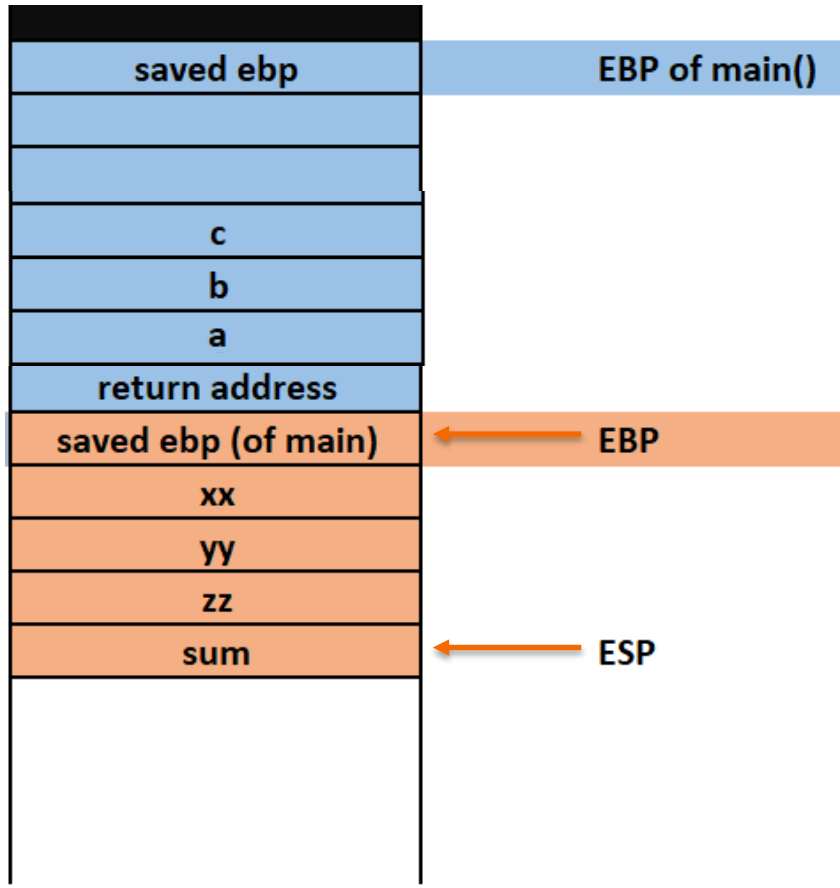
- Prior to executing a ret instruction:
  1. The return value must be stored in EAX
  2. The stack must be cleaned up!

ret = pop ebp

```
int foobar(int a, int b, int c)
{
    return xx * yy * zz + sum;
}
```

```
_foobar:
push ebp
mov ebp, esp
sub esp, 16
…
add eax, edx
add esp, 16
pop ebp
ret
```

Referred to as the "function epilogue"

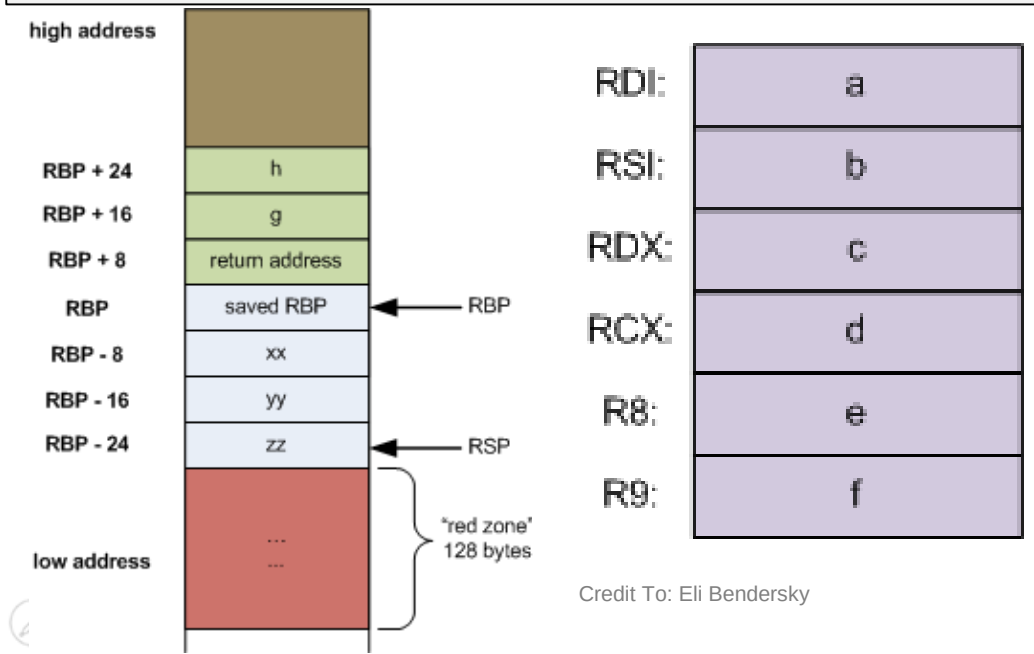


Stack and Function Call (64-Bit Linux)

```
long foobar(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = a - b - c - d - e - f - g - h;
return zz * xx * yy;
}
```

Stack and Function Call (64-Bit Linux)

```
long foobar(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = a - b - c - d - e - f - g - h;
return zz * xx * yy;
}
```



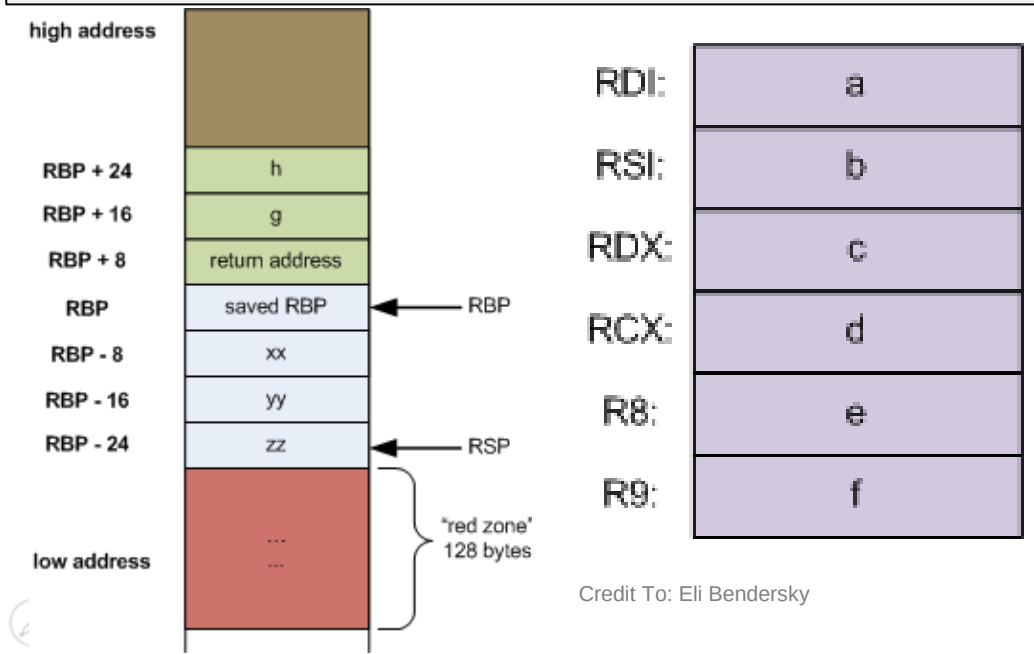
Credit To: Eli Bendersky

On Linux, you use rdi si dx cx r8, and r9 for storing these arguments and you can see It's going to be the first six arguments that you store in the registers.

After that, you go back to the old pushing in reverse order to the stack that we did before.

Stack and Function Call (64-Bit Linux)

```
long foobar(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = a - b - c - d - e - f - g - h;
return zz * xx * yy;
}
```

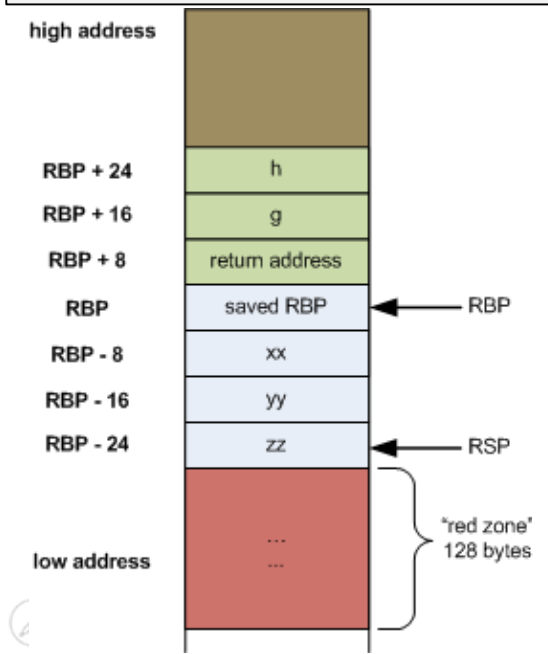


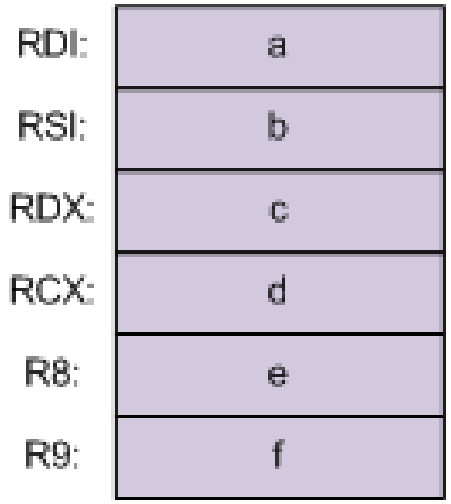
Credit To: Eli Bendersky

```
foobar:
push    rbp
mov     rbp, rsp
sub     rsp, 24
…
main:
push    8
push    7
mov     r9,  6
mov     r8,  5
mov     rcx, 4
mov     rdx, 3
mov     rsi, 2
mov     rdi, 1
call foobar
```

Stack and Function Call (64-Bit Linux)

```
long foobar(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = a - b - c - d - e - f - g - h;
return zz * xx * yy;
}
```





Credit To: Eli Bendersky

Windows is similar, except only uses 4 registers!
- a in RCX
- b in RDX
- c in R8
- d in R9

On Windows, things are a little bit different. If you're looking at a Windows binary only four registers are used for passing arguments. Those are cx, dx, r8 and r9. It's just the difference at how Windows versus Linux is designed.

Memory Storage Heap

- Heap is much simpler!
- The program will:
  1. Call **malloc** (or other heap allocation functions)
  2. Use the address that returns to access that malloc returns (recall: in EAX)

Memory Storage Heap

- Heap is much simpler!
- The program will:
  1. Call **malloc** (or other heap allocation functions)
  2. Use the address that returns to access that malloc returns (recall: in EAX)

```
1   int foobar()
2   {
3   int* x = malloc(sizeof(int));
4   *x = 3;
5   return *x;
6   }
```

Memory Storage Heap

- Heap is much simpler!
- The program will:
  1. Call **malloc** (or other heap allocation functions)
  2. Use the address that returns to access that malloc returns (recall: in EAX)

```
1   int foobar()
2   {
3   int* x = malloc(sizeof(int));
4   *x = 3;
5   return *x;
6   }
```

```
foobar:
        push ebp
        mov ebp, esp
        push 4
        call malloc
        mov DWORD PTR [eax], 3
        mov eax, [eax]
        add esp, 4
        pop ebp
        ret
```

Additional Reading (Optional):

- RE4B (Reverse Engineering 4 Beginners)
  - Covers Intel, ARM, MIPS assembler with concrete examples
  - Focus isn't on malware, but still a great reference

- Intel architecture manuals https://software.intel.com/en-us/articles/intel-sdm

- http://ref.x86asm.net/

- http://x86asm.net/articles/x86-64-tour-of-intel-manuals/index.html

- http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64

- https://godbolt.org/

Lesson Summary

- Recognize and define x86 assembly language
- Distinguish the difference between 16/32/64-bit assembly code
- Explore sections of executable file
- Differentiate Intel and AT&T syntax
- Explain Stack and Heap memory allocations.

6) Extra Credit 1 Solution Now …

Course Overview

- **Title: "CSEC 202 - Reverse Engineering Fundamentals"**

Instructor	Office	Phone	Email	Semester-Year
Emad Abu Khousa	D003		eakcad@rit.edu	Spring-2024
Office Hours:	M: 12:00-01:00 TR: 11:00-12:00			

- **600: TR 12:00-01:20, Room B-107**
- **601: MW 01:05-02:25, Room C-109**
- **602: TR 01:30-02:50, Room D-207**

Thank You and Q&A