Lec 6: Machine Code

CS492E: Introduction to Software Security

Sang Kil Cha

Our goal in software security is to find out whether a program is secure or not.

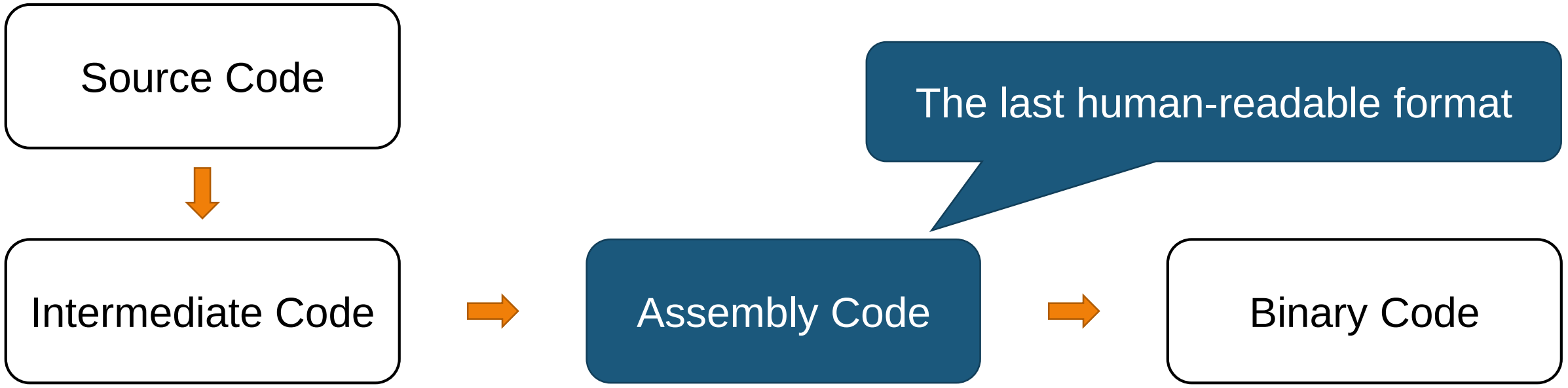
To do so, we need to see how the program ***binary*** (= executable code) executes on a machine.

Compilation

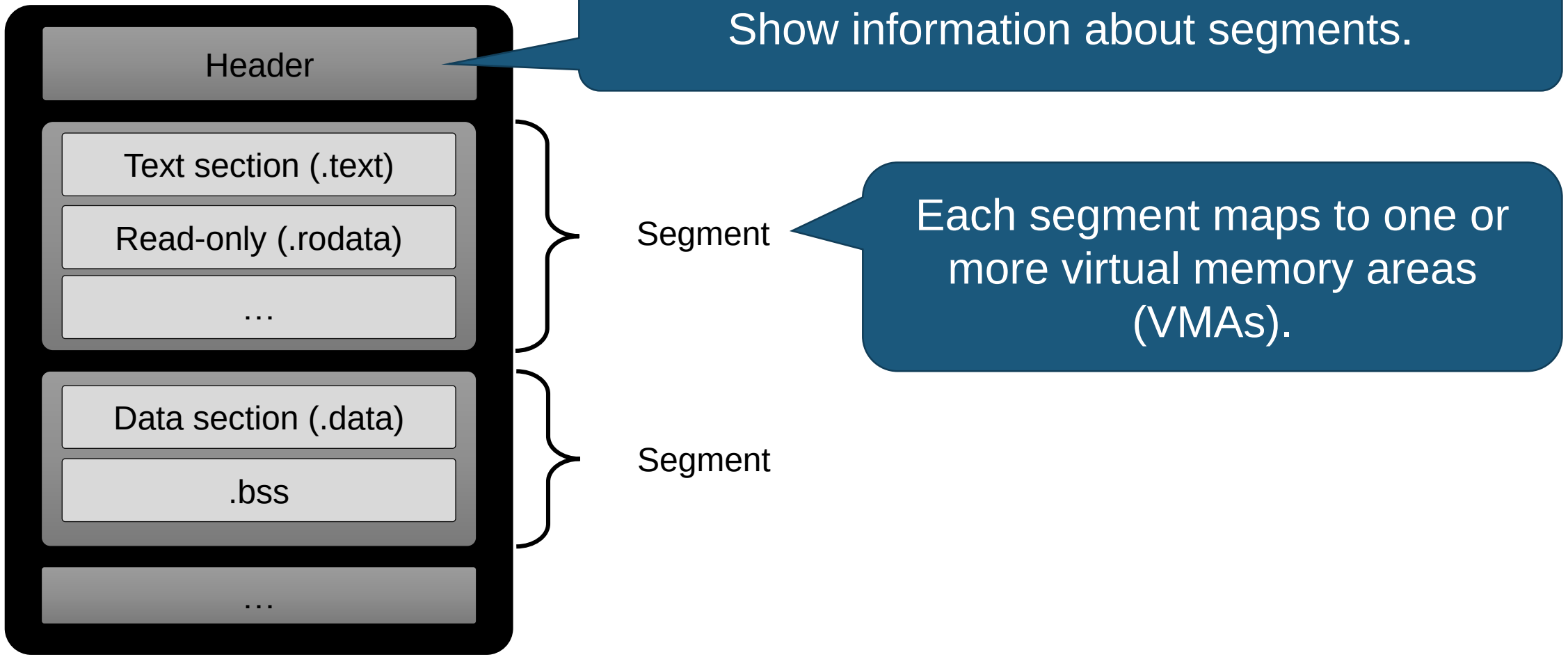
```
#include <stdio.h>
int someFunction(int a, int b)
{
  int s = a + b;
  printf("%d\n", s);
  return s;
}

int main(int argc, char* argv[])
{
  int x = 0;
  return someFunction(x, 42);
}
```

```
01010101010111111010
10101010101010010001
00100100011111111010
11111010010101000010
00101101000101101000
01010010010100101111
11101010100000010100
10110000010000010111
```

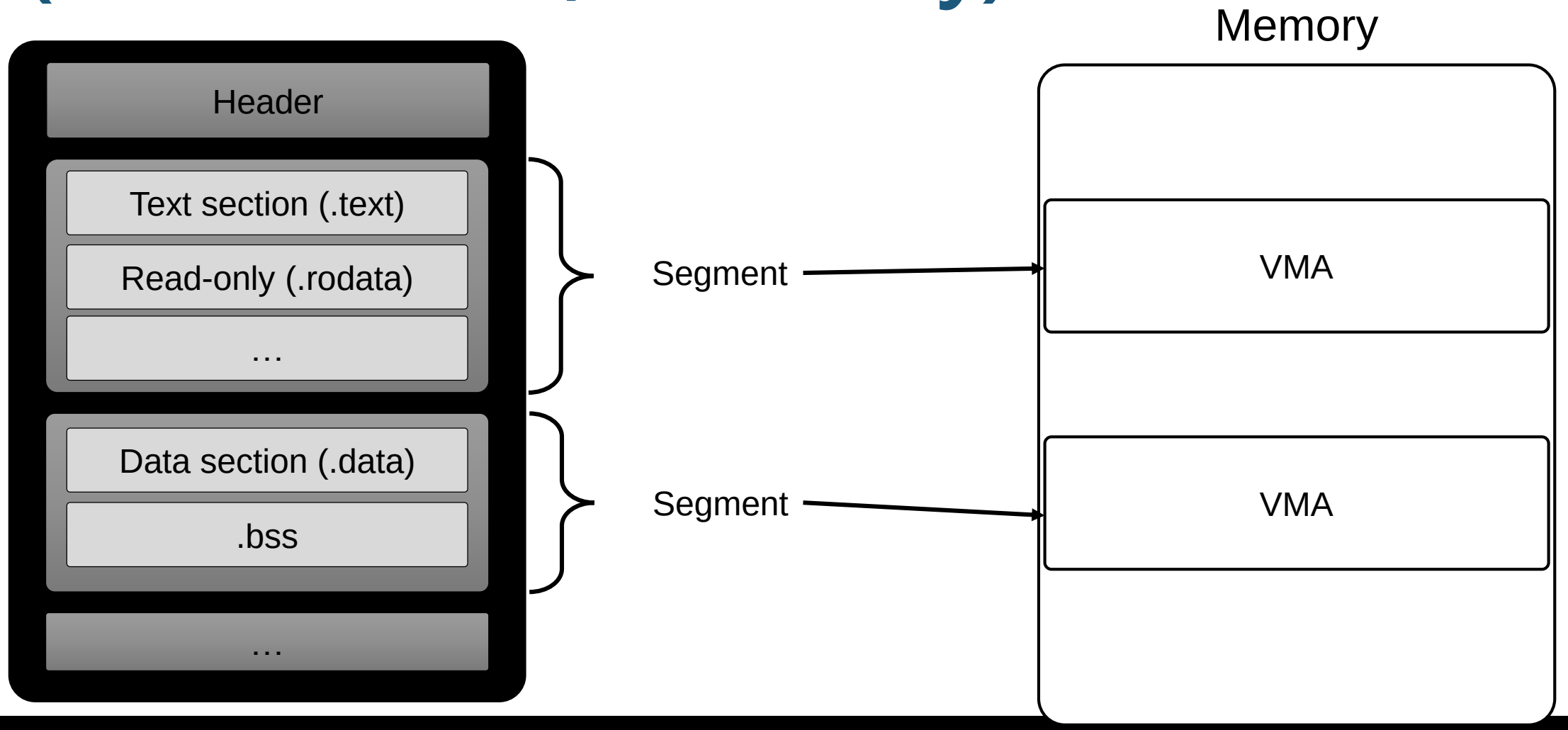
Compilation



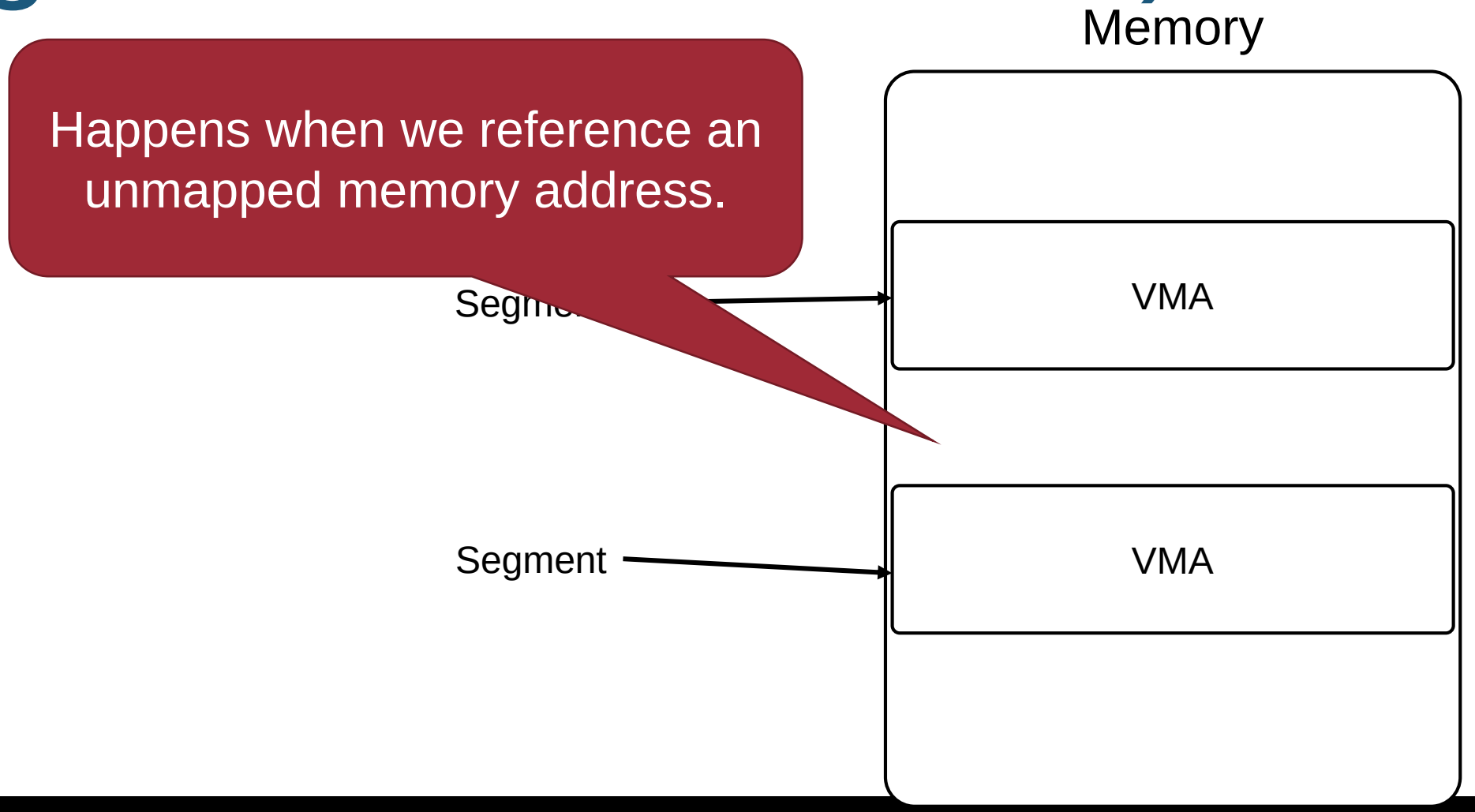
Executable Binary (= Executable, or Binary)



Executable Binary (= Executable, or Binary)



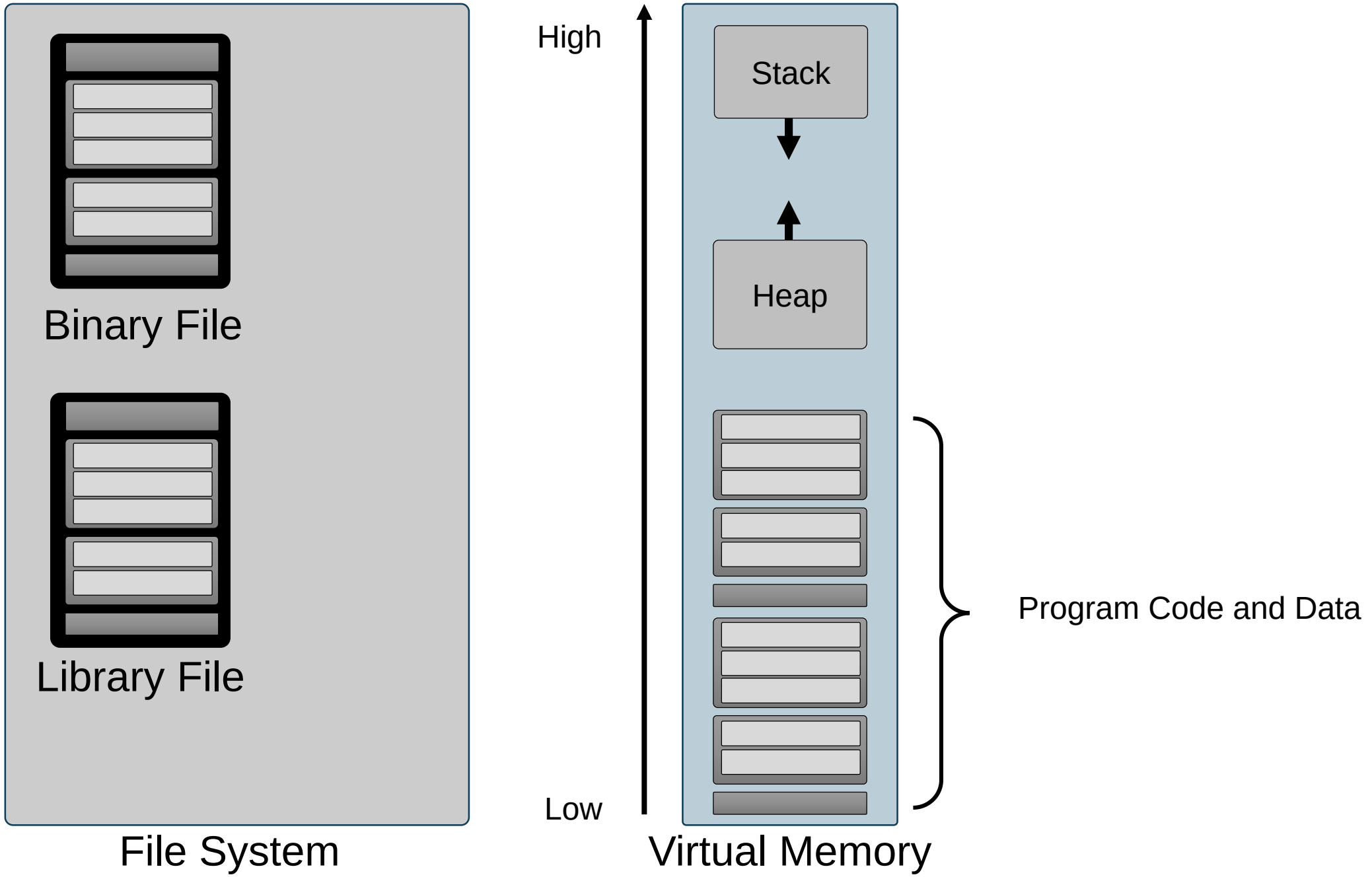
Segmentation Fault
(= SegFault or Access Violation)

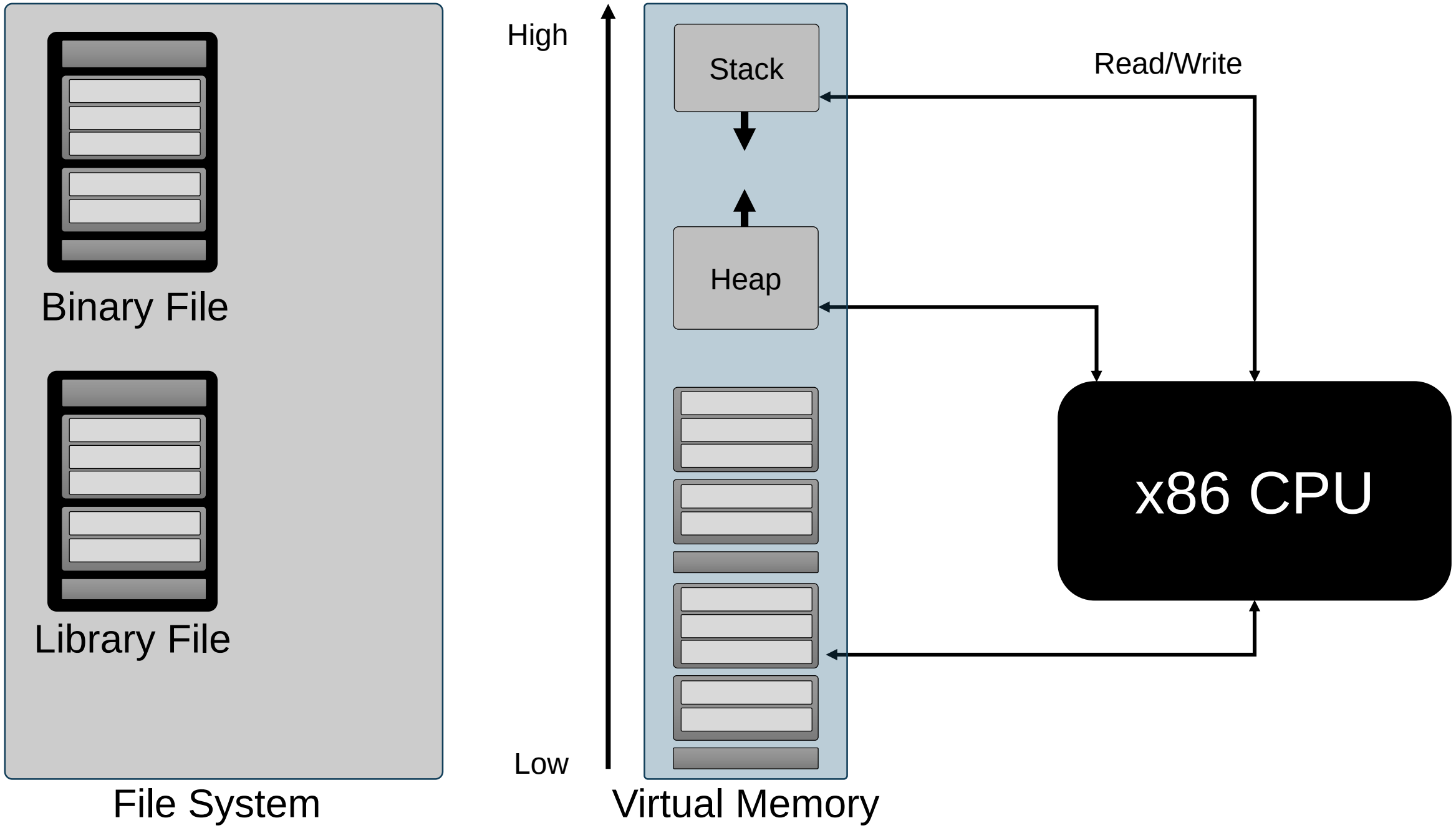


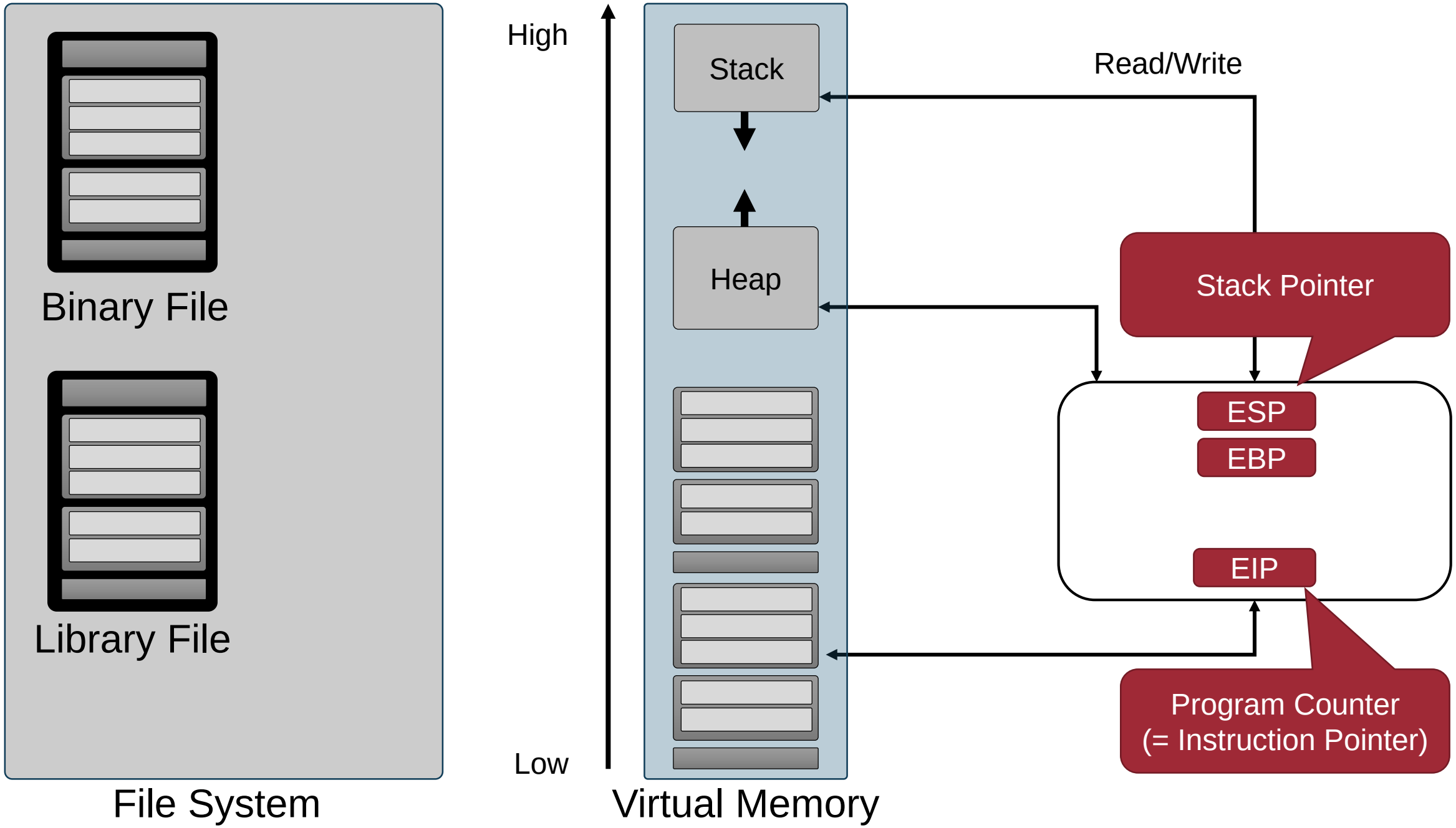
x86 (IA-32) Architecture

x86

- Developed by Intel in 1985
- 32-bit address space
- One of the most common architecture


Binary File
Library File
File System
High
Low
Stack
Heap
Program Code and Data
Virtual Memory


Binary File
Library File
File System
High
Low
Stack
Heap
Virtual Memory
Read/Write
x86 CPU


Binary File
Library File
File System
High
Low
Stack
Heap
Virtual Memory
Read/Write
Stack Pointer
ESP
EBP
EIP
Program Counter
(= Instruction Pointer)

Registers in x86

- General Purpose Registers
  - **EAX**, **EBX**, **ECX**, **EDX**
- Pointers
  - **ESI**, **EDI**
- Stack Pointers
  - **ESP**: points to the top of the stack
  - **EBP**: points to the base of the current stack frame
- Special Registers:
  - **EIP**: instruction pointer
  - **EFLAGS**: holds the state of the processor

All of them have a size of a ***double word*** (= 32 bit)

Wait, Double Word?

- A word is the natural ***unit*** of data used by a processor.
- Typically, a word size is 32 bits on a 32-bit machine, and 64 bits on a 64-bit machine.

However, in x86, we say a word is 16 bits and a double word is 32 bits even though it is a 32-bit processor.

History of Intel/AMD Processors

Word size was 16-bit

1978: 8086
1982: 80286
} 16-bit processor, Registers (SP, BP, IP, …)

1985: 80386
1989: 80486
} 32-bit processor, Registers (ESP, EBP, EIP, …)

…

x86 or IA-32

2003: Opteron
2005: Prescott
2006: Core 2
} 64-bit processor, Registers (RSP, RBP, RIP, …)

…

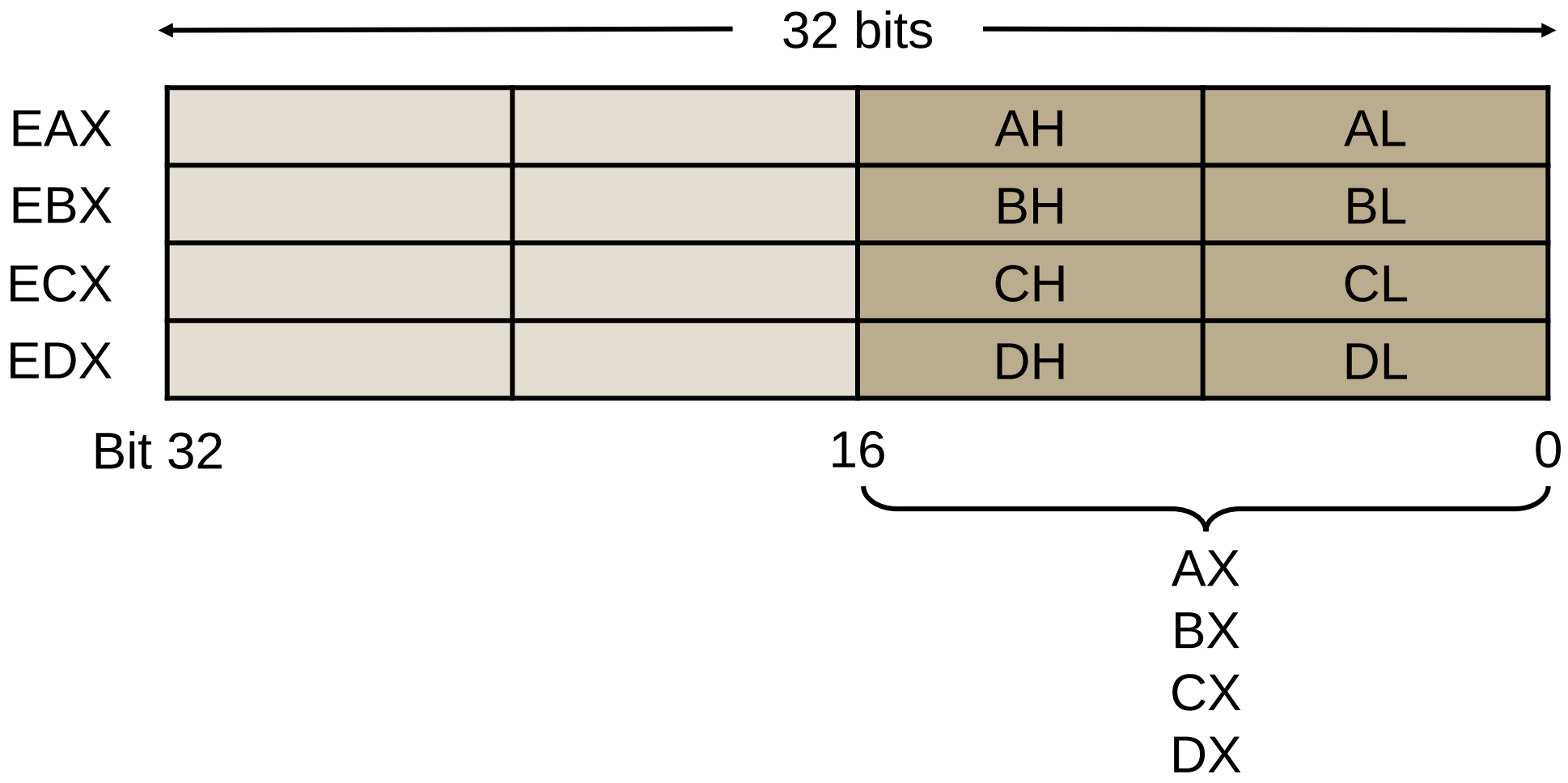
x86-64 or AMD64

x86 Convention

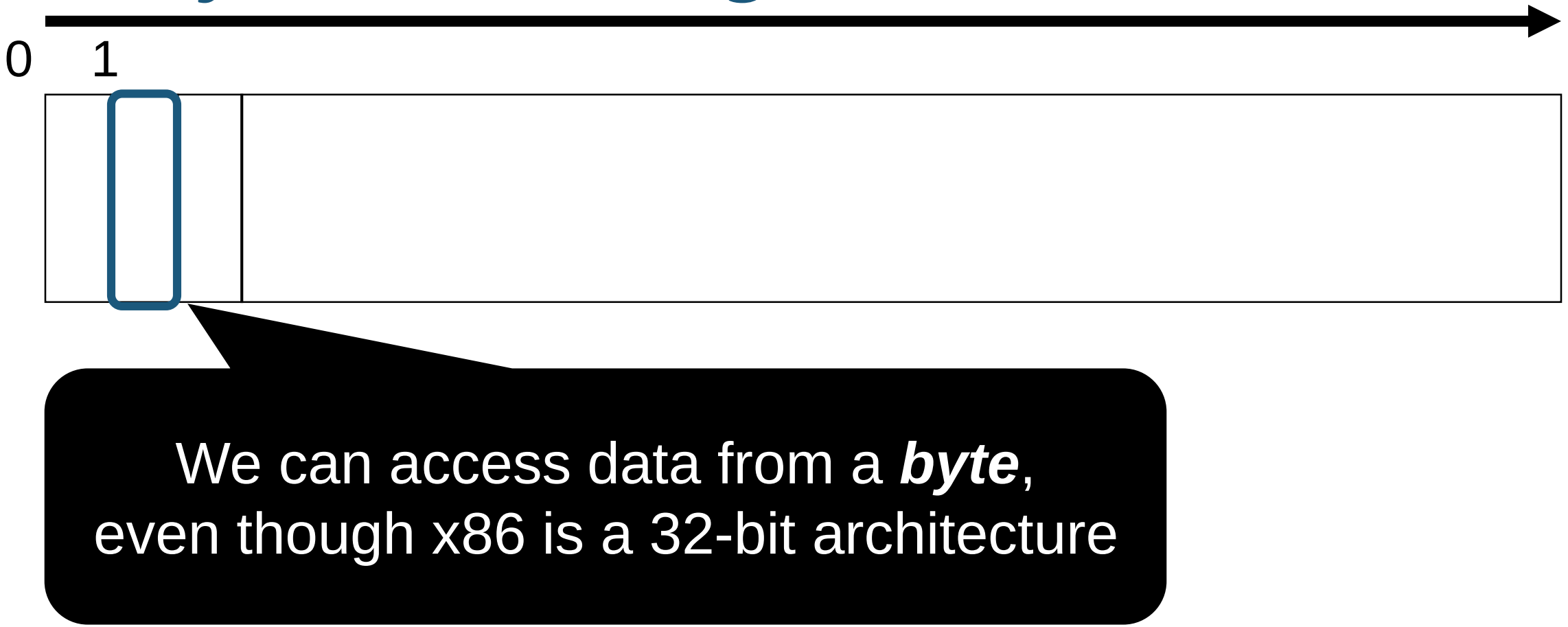
- Word = 16 bits
- Double Word (DWORD) = 32 bits
- Quad Word (QWORD) = 64 bits

- Linear address space = 0 ~ 2^{32} bits

x86 Register Accesses



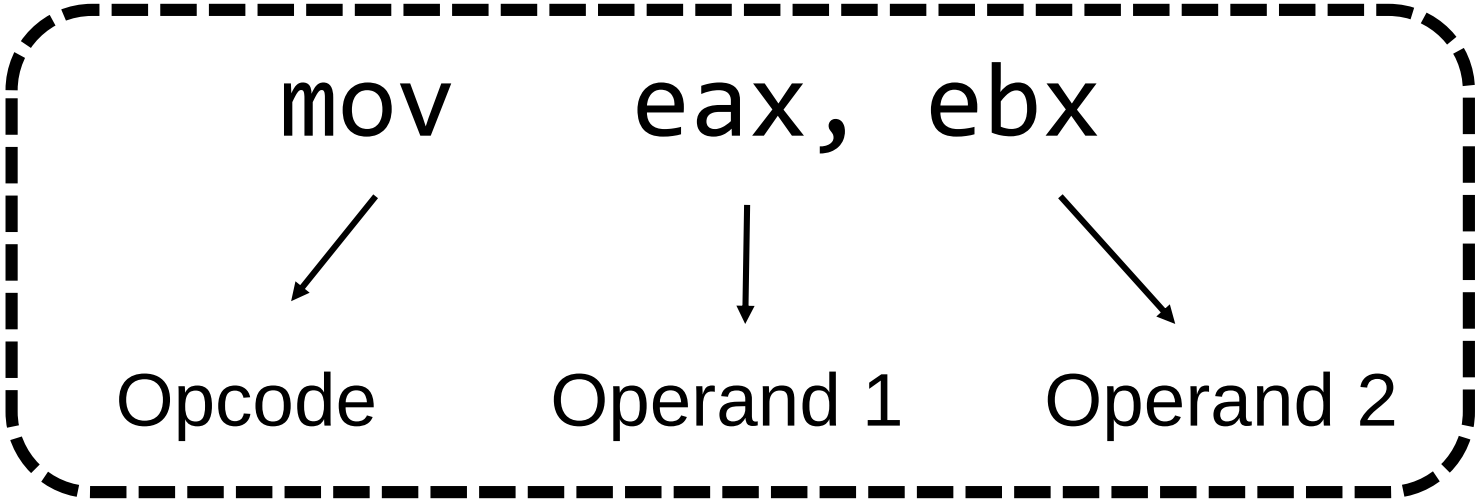
x86 Memory Access = Byte Addressing



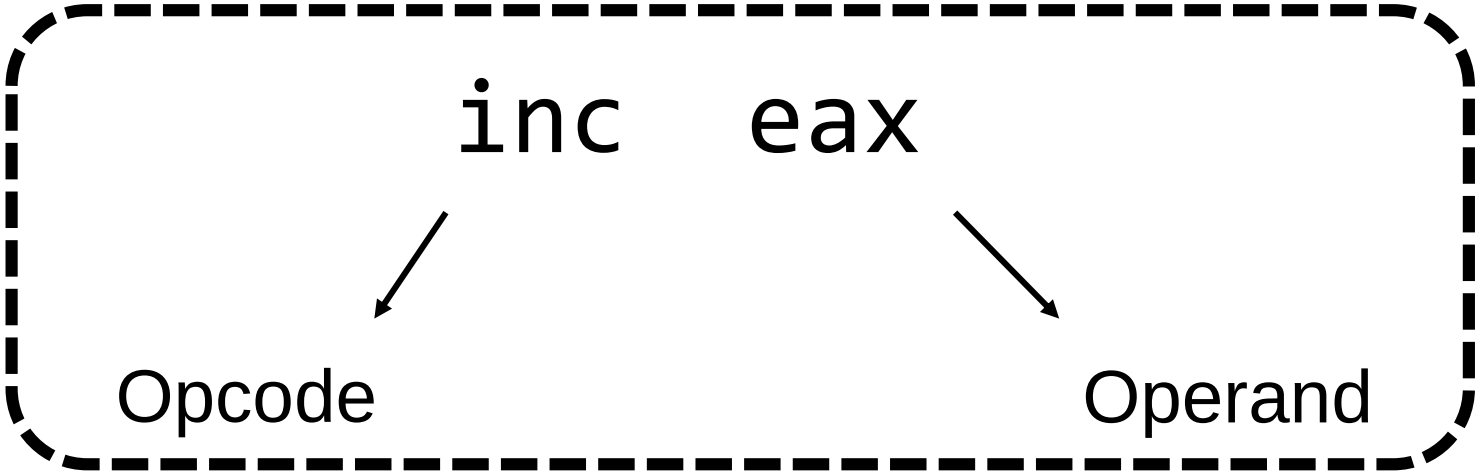
x86 Assembly

Basic Format of x86 Instructions

2 operands

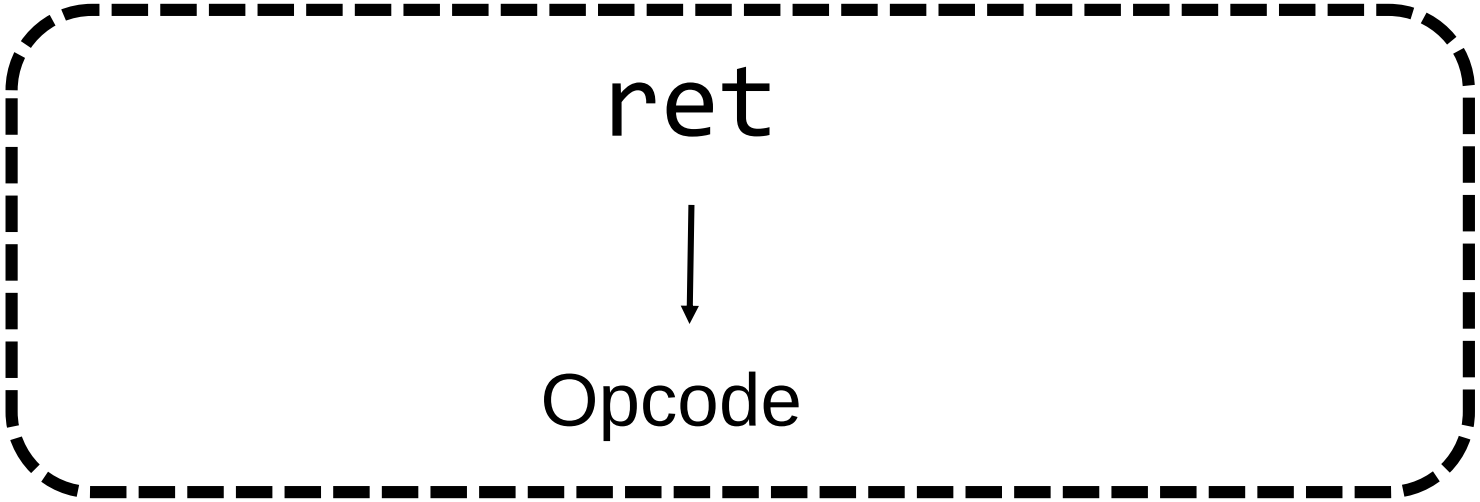


1 operand



Basic Format of x86 Instructions

0 operand



Opcode Decides Semantics

```
mov    eax, ebx
```

```
eax ← ebx
```

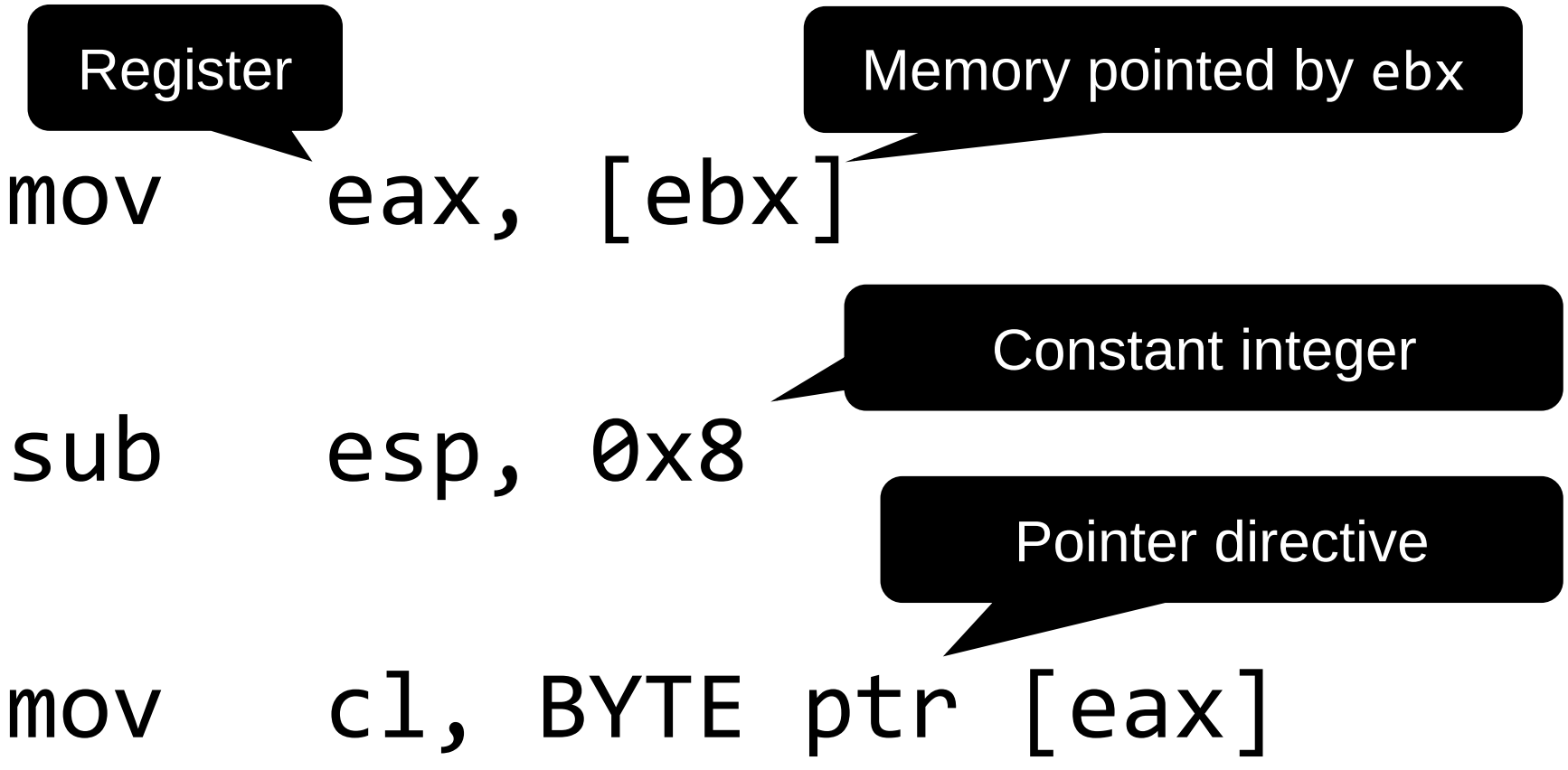
```
sub    esp, 0x8
```

```
esp ← esp – 0x8
```

```
inc    eax
```

```
eax ← eax + 1
```

Operand Types



Pointer Directive

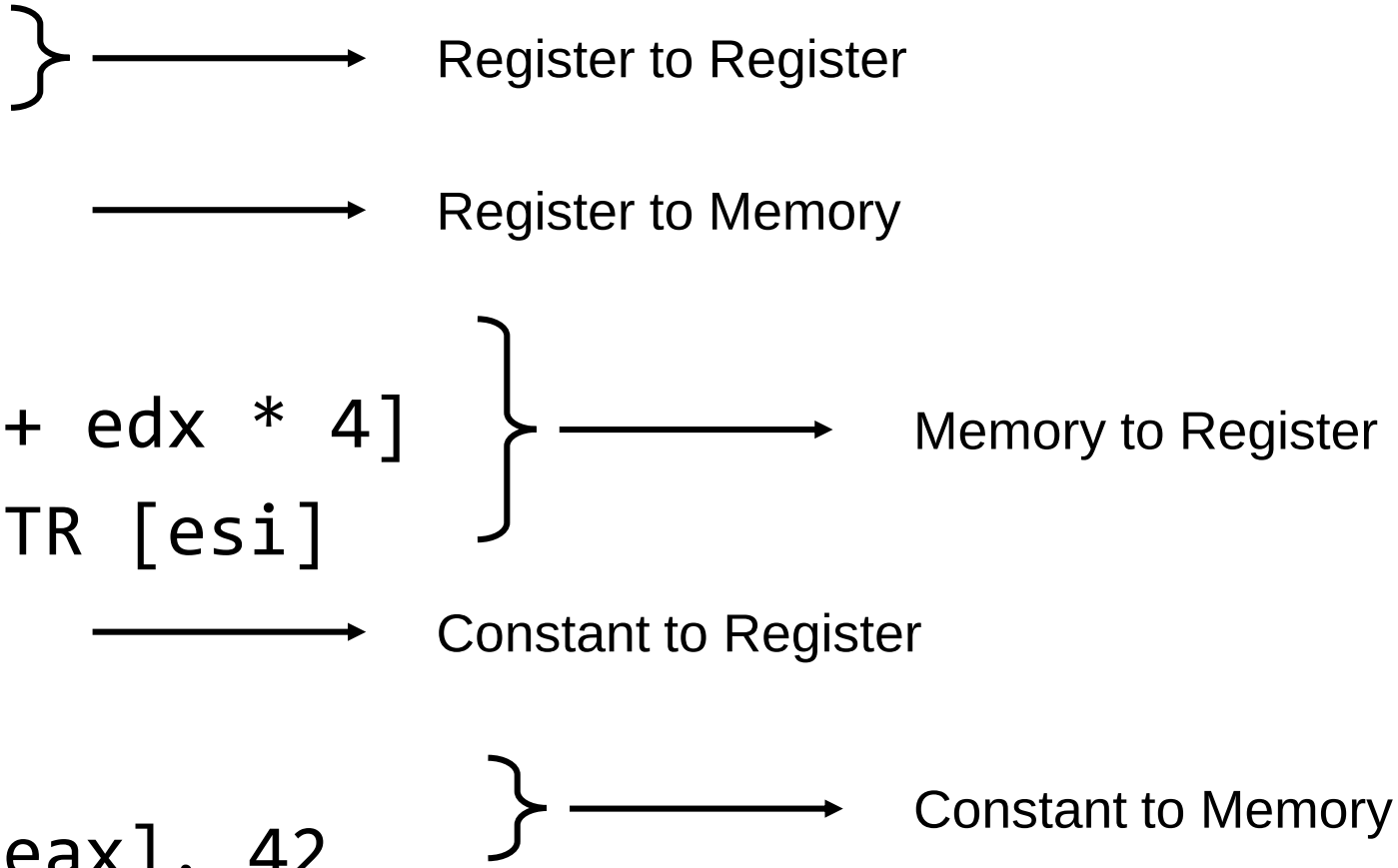
```
mov    [esi], al      ; ok
mov    [esi], 1       ; error (ambiguious)
```

```
mov    DWORD PTR [esi], 1
           or
mov    WORD PTR [esi], 1
           or
mov    BYTE PTR [esi], 1
```

Pointer directive is required!

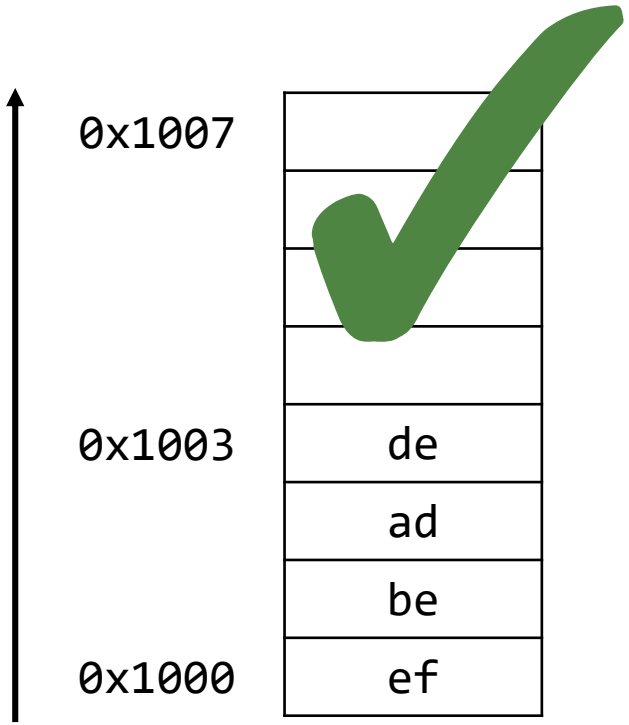
Moving Data Around (mov)

```
mov    eax, ebx
mov    al, bl
mov    [eax], ebx
mov    eax, [ebx]
mov    eax, [ebx + edx * 4]
mov    al, BYTE PTR [esi]
mov    eax, 42
mov    [ebx], 42
mov    BYTE PTR [eax], 42
```

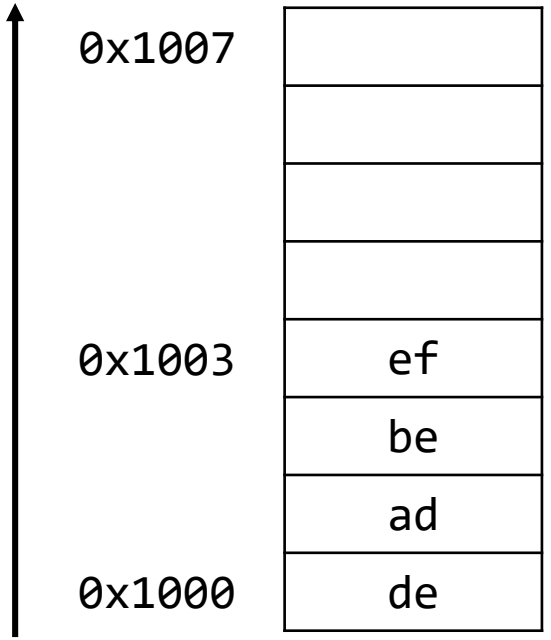


Example: Storing a DWORD in Memory

```
mov    [eax], 0xdeadbeef    ; eax = 0x1000
```



VS.



Endianness

The order in which a sequence of bytes are stored in memory

- Big Endian = The MSB goes to the lowest address
- Little Endian = The LSB goes to the lowest address

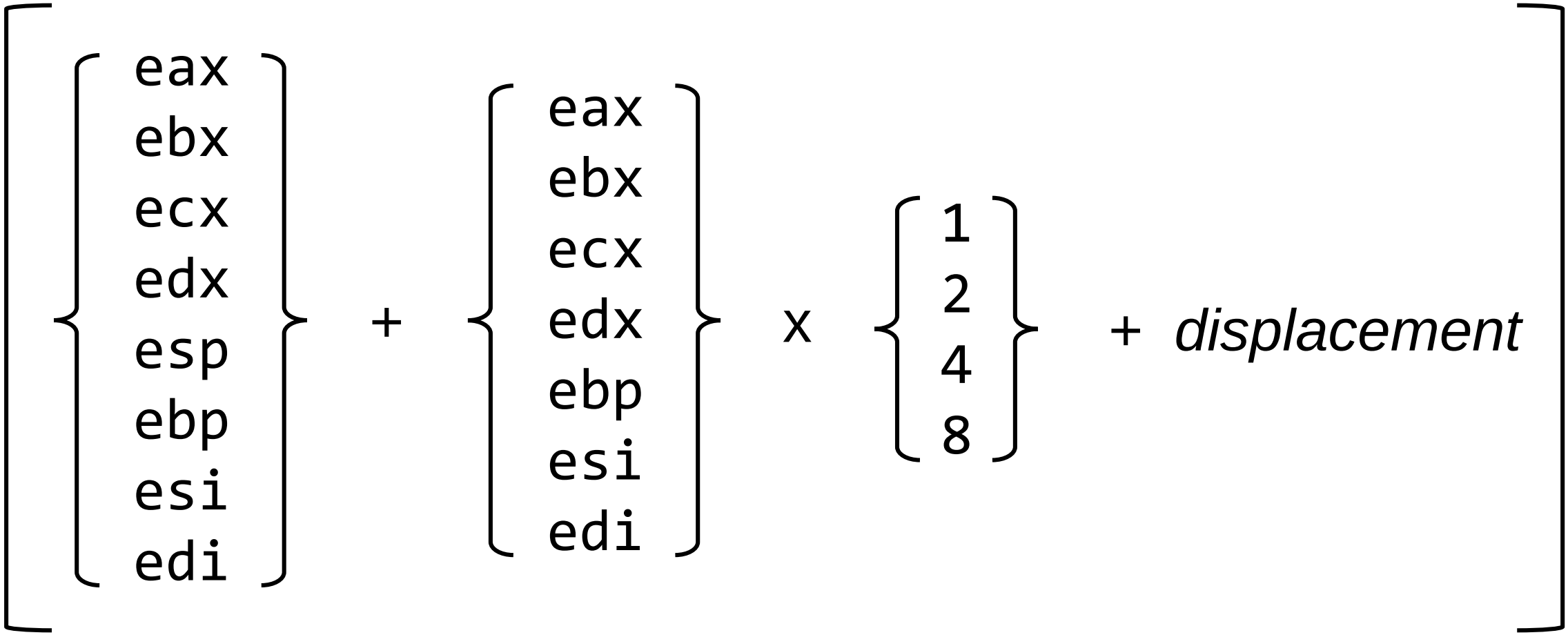
x86 uses Little Endian

Addressing Modes

- Addressing mode specifies how an instruction can access a memory location.
- There are many ways to represent a memory address other than just: `[register]`
- For example
  - `[register + register]`
  - `[register + register * num]`
  - `[register + register * num + num]`

  e.g., `mov eax, [edx + ebx * 4 + 8]`

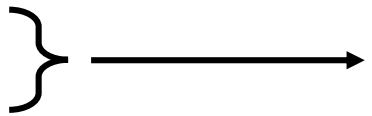
Addressing Modes



Loading Address (lea)

Load-effective Address

```
lea eax, [ebx]
lea eax, [ebp-0x8]
```



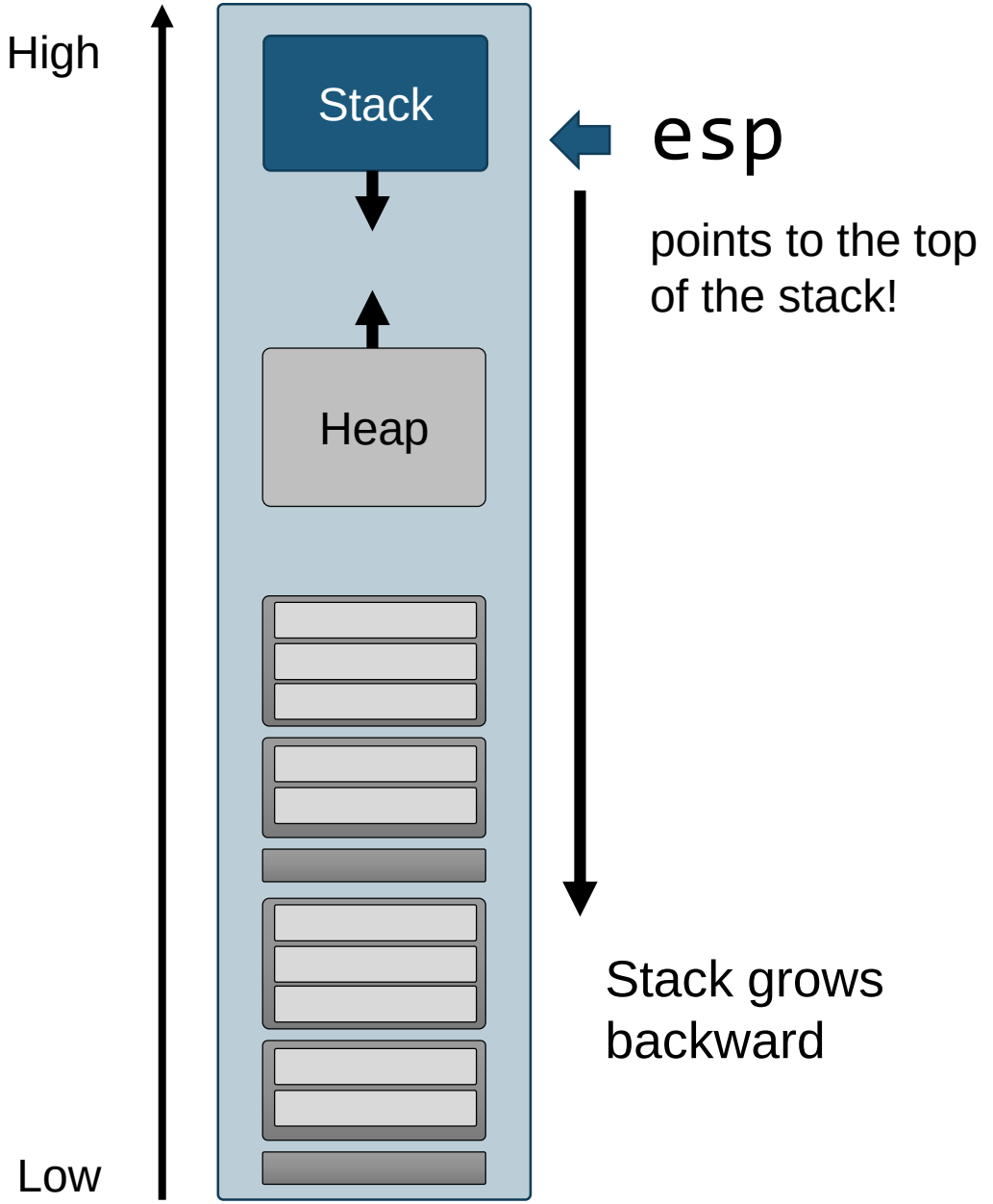
Memory address to Register

What's the Difference?

```
mov eax, [ebp + 0x10]
        vs.
lea eax, [ebp + 0x10]
```

```
eax ← *(ebp + 0x10)
        vs.
eax ← (ebp + 0x10)
```

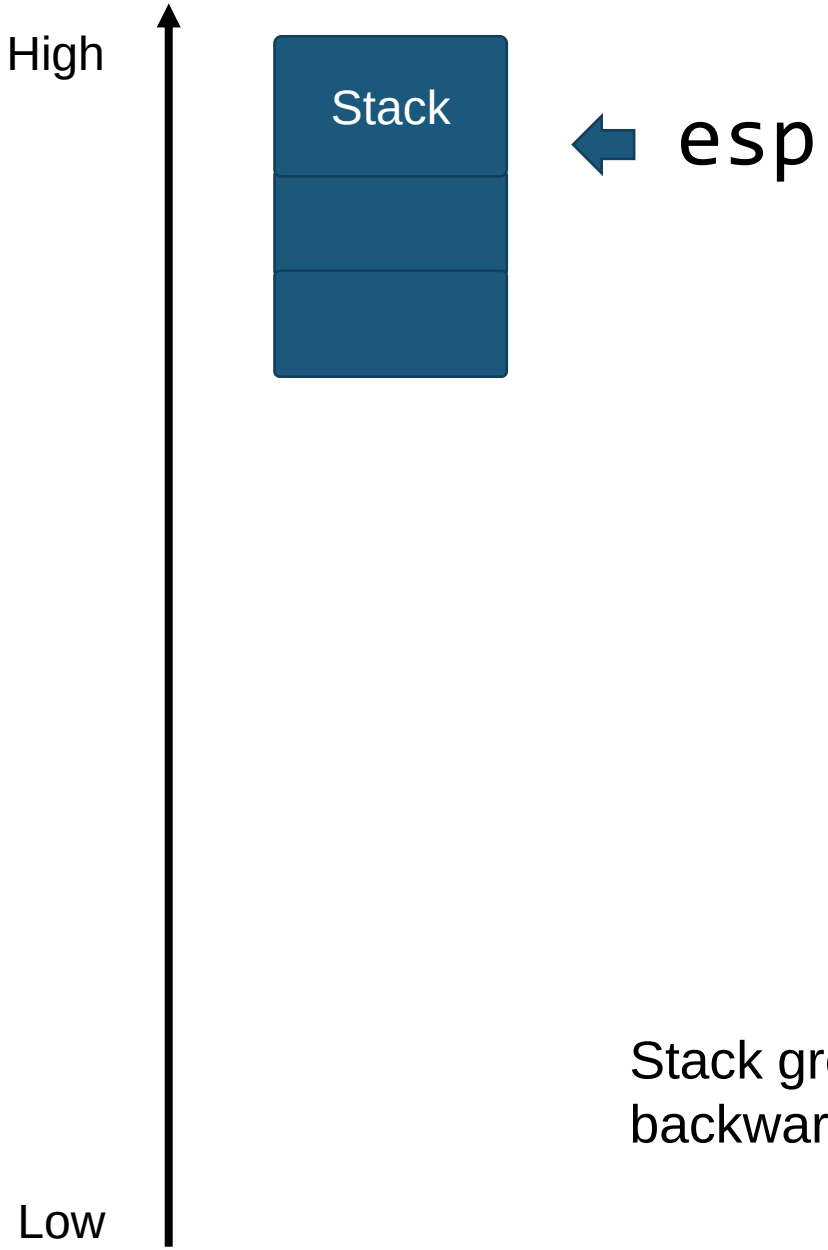
Stack Operations



Stack Operations

Push

Pop



Stack Operations (push)

`push eax` ⟶ Push register on the stack

`push 0x42` ⟶ Push constant on the stack

`push [eax]` ⟶ Push a value at the memory address on the stack

```
push x
```

=

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

Stack Operations (pop)

`pop eax` → Pop the top element of the stack into register

`pop [eax]` → Pop the top element of the stack into the memory address

```
pop x
```

=

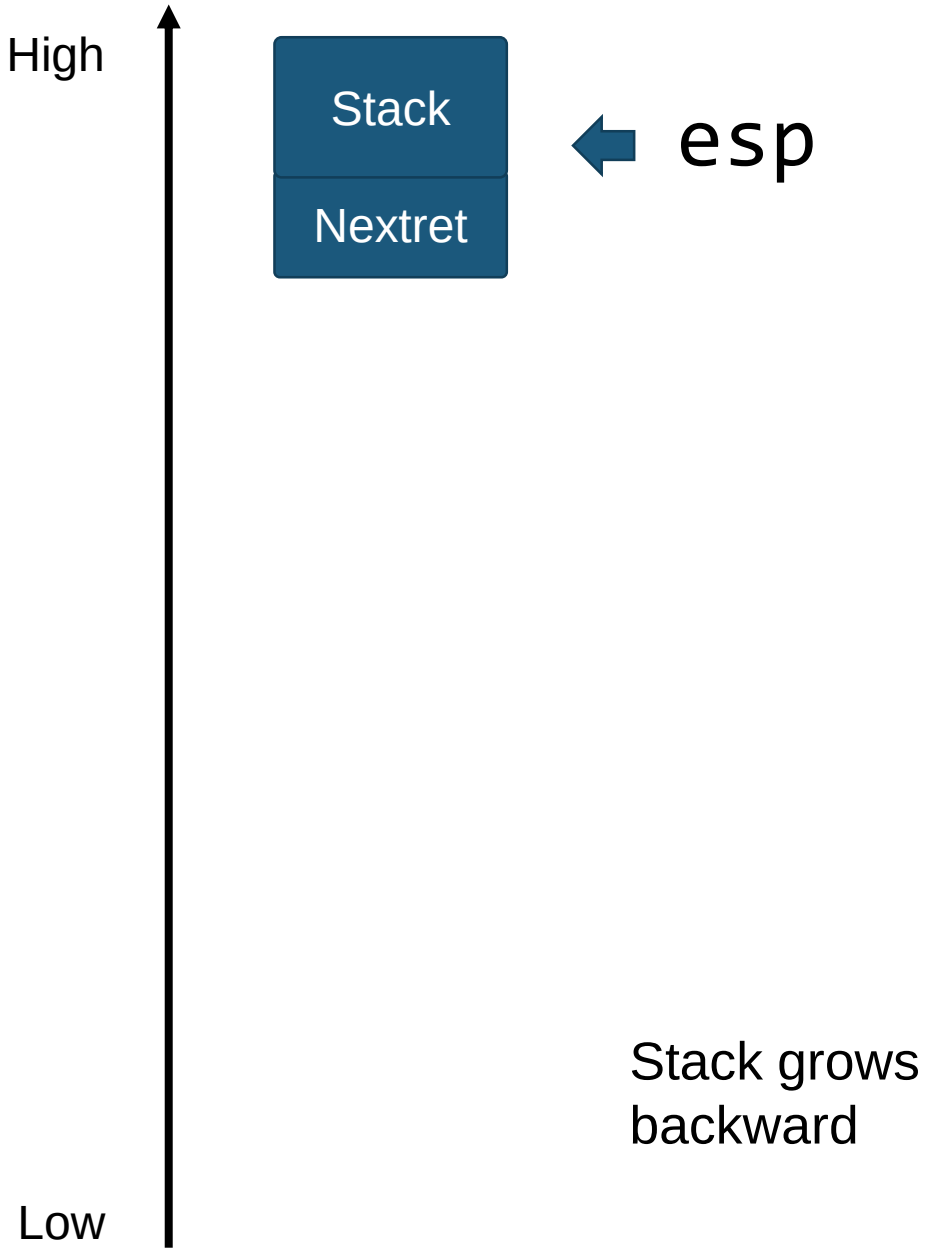
```
mov x, [esp]
add esp, 4
```

Stack Operations (leave)

```
leave     =     mov esp, ebp
                pop ebp
```

Call (call)

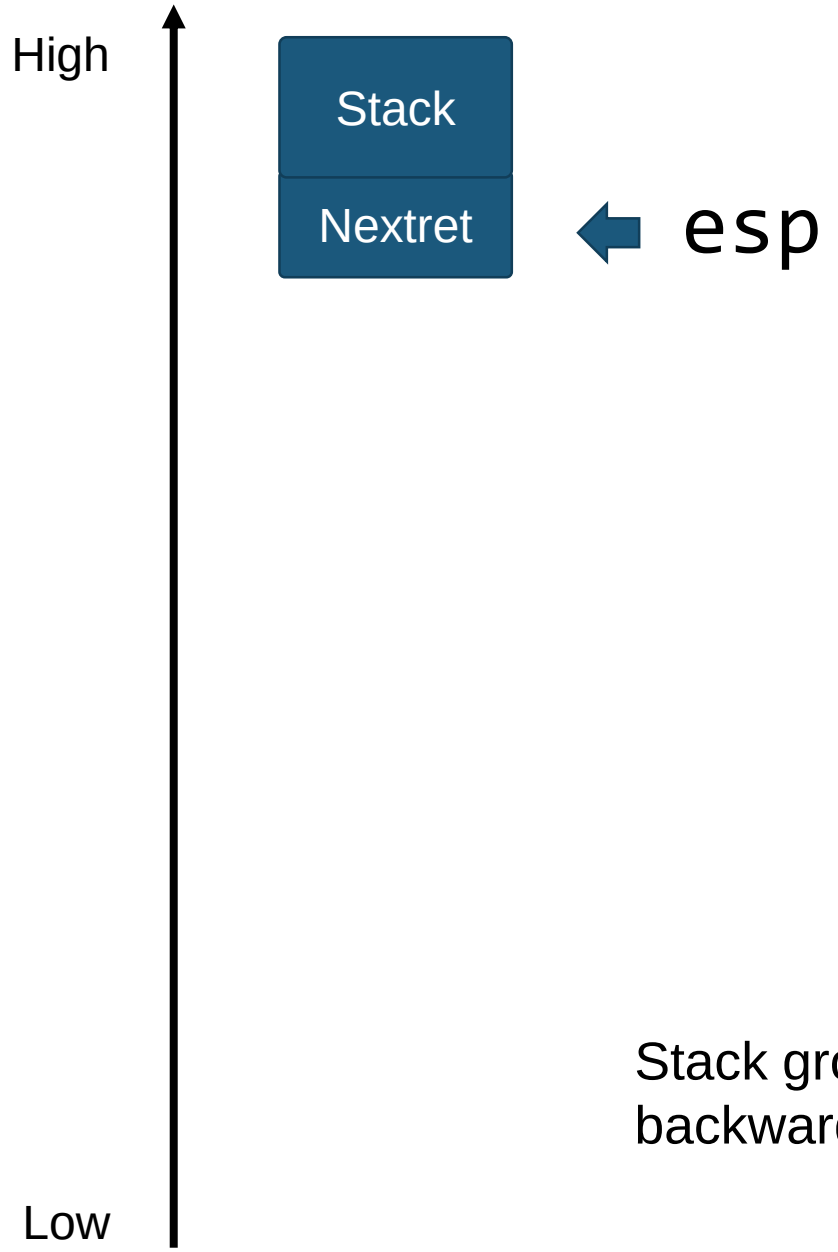
```
call foo ; call function foo
Nextret: ; next label after returning
         ; from foo

= push Nextret
  jmp foo
```



Return (ret)

```
call foo ; call function foo
Nextret: ; next label after returning
         ; from foo

= push Nextret
  jmp foo
```

```
ret      ; return to the caller

= pop eip
```



Arithmetic

```
add  eax, [ebx]
sub  esp, 0x40
inc  ecx
dec  edx
and  [eax + ecx], ebx
xor  edx, ebx
shl  eax, 1
…
```

Control Flow

```
if ( x ) A();
else B();
```

```
while ( x ) { }
```

```
for ( i = 0; i < n; i++ )
{ }
```

How to represent in assembly?

Use Only “IF”s and “GOTO”s

```
if ( x ) A();
else B();
```

```
if ( !x ) goto F;
A(); goto E;
F:
B();
E: // next …
```

```
while ( x ) { }
```

```
WHILE:
if ( !x ) goto DONE;
...
goto WHILE;
DONE: // next …
```

```
for ( i = 0; i < n; i++ )
{ }
```

```
i = 0;
LOOP:
if ( i >= n ) goto DONE;
…
i++; goto LOOP;
```

Use Only “IF”s and “GOTO”s

This is roughly how assembly looks like

```
if ( !x ) goto F;
A(); goto E;
F:
B();
E: // next …


WHILE:
if ( !x ) goto DONE;
...
goto WHILE;
DONE: // next …


i = 0;
LOOP:
if ( i >= n ) goto DONE;
…
i++; goto LOOP;
```

Jump and Branch

```
cmp x, 0 ; test if x is zero
je F     ; if x = zero then goto F
```

```
cmp i, n ; test if i >= n
jge DONE  ; if x = zero then goto F
```

```
jmp  LOOP
```

```
if ( !x ) goto F;
A(); goto E;
F:
B();
E: // next …
```

```
WHILE:
if ( !x ) goto DONE;
...
goto WHILE;
DONE: // next …
```

```
i = 0;
LOOP:
if ( i >= n ) goto DONE;
…
i++; goto LOOP;
```

```
cmp x, 0 ; test if x is zero
je F     ; if x != zero then goto F
```

Where do we store the result of the comparison?

Implicitly fetch the stored result to perform conditional branch

EFLAGS: Storing the Processor State

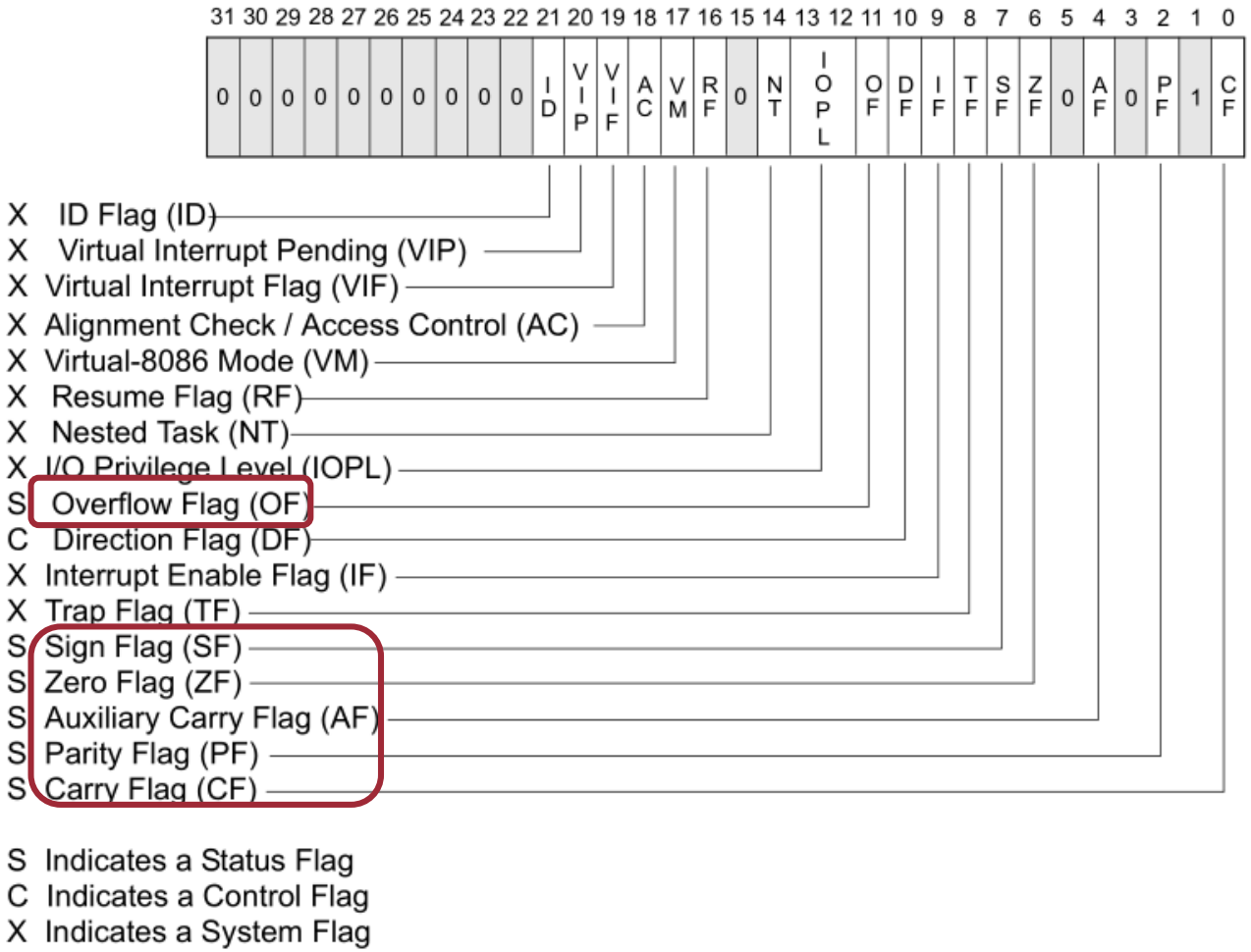


Figure from Intel 64 and IA-32 Architectu

Branch Instructions

Branch Instruction	Condition	Description
`ja`	`CF = 0 and ZF = 0`	Jump if above
`jb`	`CF = 1`	Jump if below
`je`	`ZF = 1`	Jump if equal
`jl`	`SF ≠ OF`	Jump if less
`jle`	`ZF = 1 or SF ≠ OF`	Jump if less or equal
`jna`	`CF = 1 or ZF = 1`	Jump if not above
`jnb`	`CF = 0`	Jump if not below
`jz`	`ZF = 1`	Jump if zero

… and many more

(For more information, see Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2, Chapter 3)

CMP Internals

```
cmp x, y
je F    ; if x = y then goto F
```

T := x – y
ZF := if T = 0 then 1 else 0
CF := if x < 0 then 1 else 0
…

If ZF = 1 then goto F
Else fall-through

So Far

- We learned how to move around data
  - How to load/store data from/to memory and registers
  - How to compute a pointer (address) for a memory
  - How to use stack (push/pop)
- We learned how to perform arithmetic operations
- We also learned how to control program's flow
  - Compare values and conditionally jump based on the comparison
  - Directly jump to a certain location

Already Turing Complete!

How to Know Instruction Semantics?

- Read the manual
- Or use ***B2R2***
  - https://github.com/B2R2-org/B2R2

Running B2R2

- Install .NET 6 (or above) SDK
  - https://dotnet.microsoft.com/en-us/download/dotnet/6.0
  - Choose the right OS (any OS should work)
  - After the installation, you should be able to run `dotnet` command from your terminal
- Type the following command in the terminal:

```
dotnet tool install --global B2R2.RearEnd.Launcher --version 0.6.0-alpha
```

- Now run B2R2 by typing “`b2r2`” in your terminal

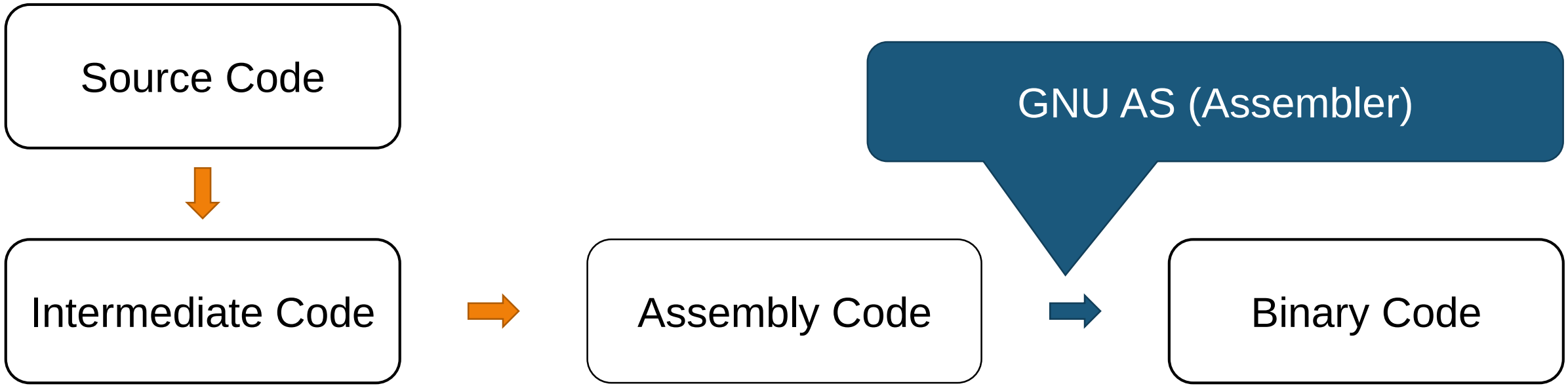
Example: CMP EAX, 0

```
$ b2r2 dump -i x86 -s 83f80074de --lift
[83 F8 00]
(3) {
T_1:I32 := EAX
T_2:I32 := 0x0:I32
T_3:I32 := (T_1:I32 - T_2:I32)
CF := (T_1:I32 < T_2:I32)
OF := (((T_2:I32 ^ T_1:I32) & (T_3:I32 ^ T_1:I32))[31:31])
AF := (((T_2:I32 ^ (T_1:I32 ^ T_3:I32)) & 0x10:I32) = 0x10:I32)
SF := (T_3:I32[31:31])
ZF := (T_3:I32 = 0x0:I32)
T_4:I32 := ((T_3:I32 >> 0x4:I32) ^ T_3:I32)
T_5:I32 := ((T_4:I32 >> 0x2:I32) ^ ((T_3:I32 >> 0x4:I32) ^ T_3:I32))
PF := (~ (((T_5:I32 >> 0x1:I32) ^ ((T_4:I32 >> 0x2:I32) ^ ((T_3:I32 >> 0x4:I32) ^ T_3:I32)))[0:0]))
} // 3
```

Example: `je -0x20`

```
$ b2r2 dump -i x86 -s 74de --lift
[74 DE]
(2) {
if ZF then ijmp (EIP + 0xfffffffe0:I32) else ijmp (EIP + 0x2:I32)
} // 2
```

Compilation



GNU AS (Assembler)

```
$ as file.s
$ ls a.out
```

```
.intel_syntax noprefix

mov  eax, ebx
…
```

When testing on a 64-bit machine, use --32 option:

```
$ as --32 file.s
```

Intel Syntax?

```
.intel_syntax noprefix
```

There are two ways to represent x86 assembly code.

- At&t: `mov %eax, %ebx ; src, dst in reverse`
- Intel: `mov ebx, eax`

We will only use Intel syntax!

Questions?