CS165 – Computer Security

Understanding low-level program execution

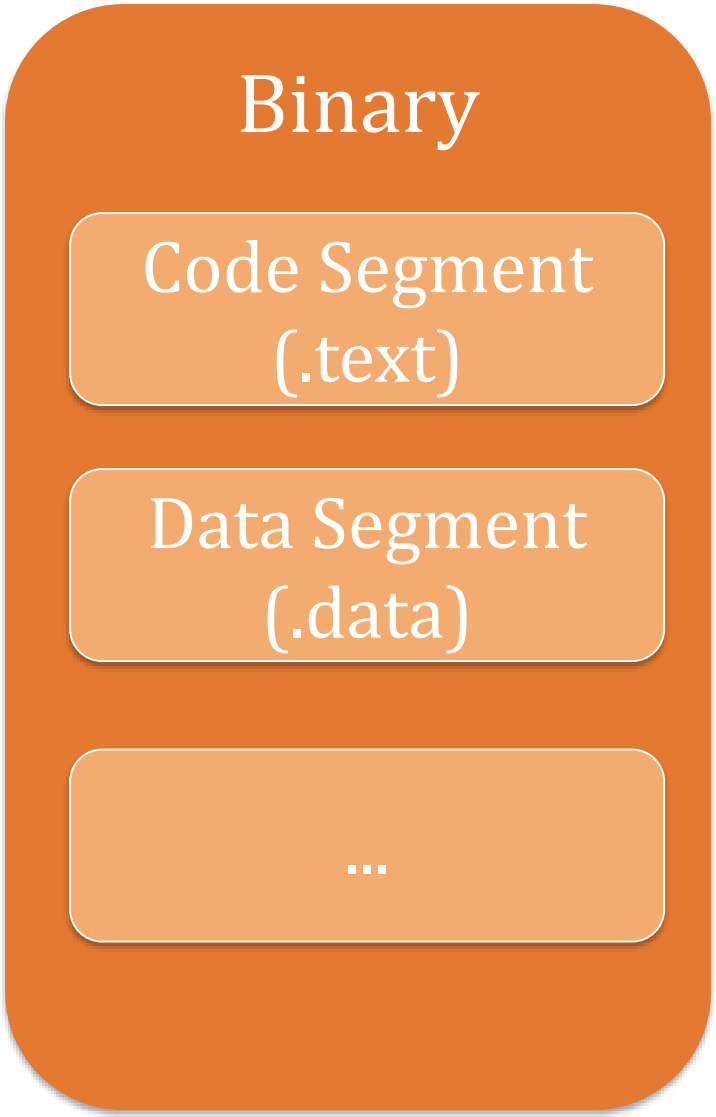
Sep 30, 2021

Refresher

- Understanding how a binary program gets to execute natively on hardware

Disassembling

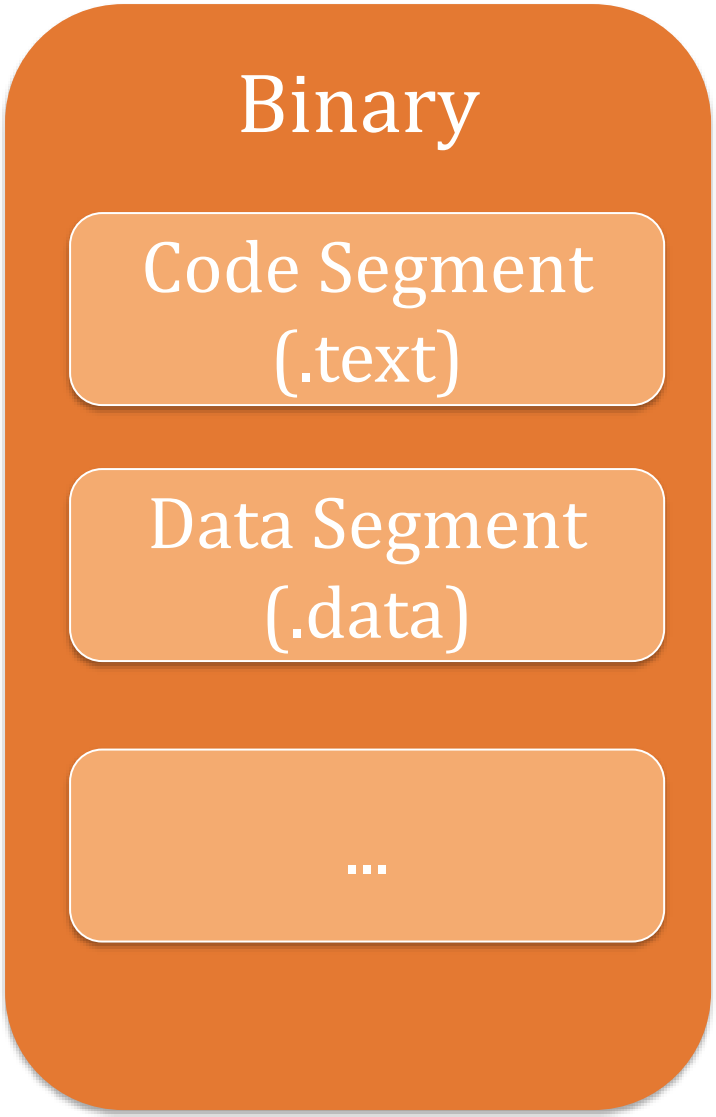
- Today: using objdump (part of binutils)
  - objdump –D <exe>
- If you compiled the source with “-g”, you will see more information
  - objdump –D –S
- GUI-based: IDA, Ghidra



The program ***binary*** (aka executable)

Final executable consists of several *segments*

- Text for code written
- Read-only data for constants such as “hello world” and globals
- …



The program ***binary*** (aka executable)

Final executable consists of several *segments*

- Text for code written
- Read-only data for constants such as "hello world" and globals
- …

```
$ readelf –S <file>
```

Machine Instruction Example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

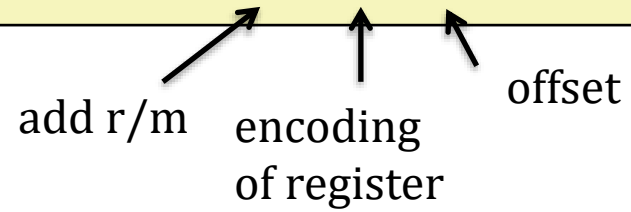
Similar to expression:

```
x += y
```

More precisely:

```
int eax;
int *ebp;
eax += *(ebp[2])
```

```
0x80483ca:  03 45 08
```

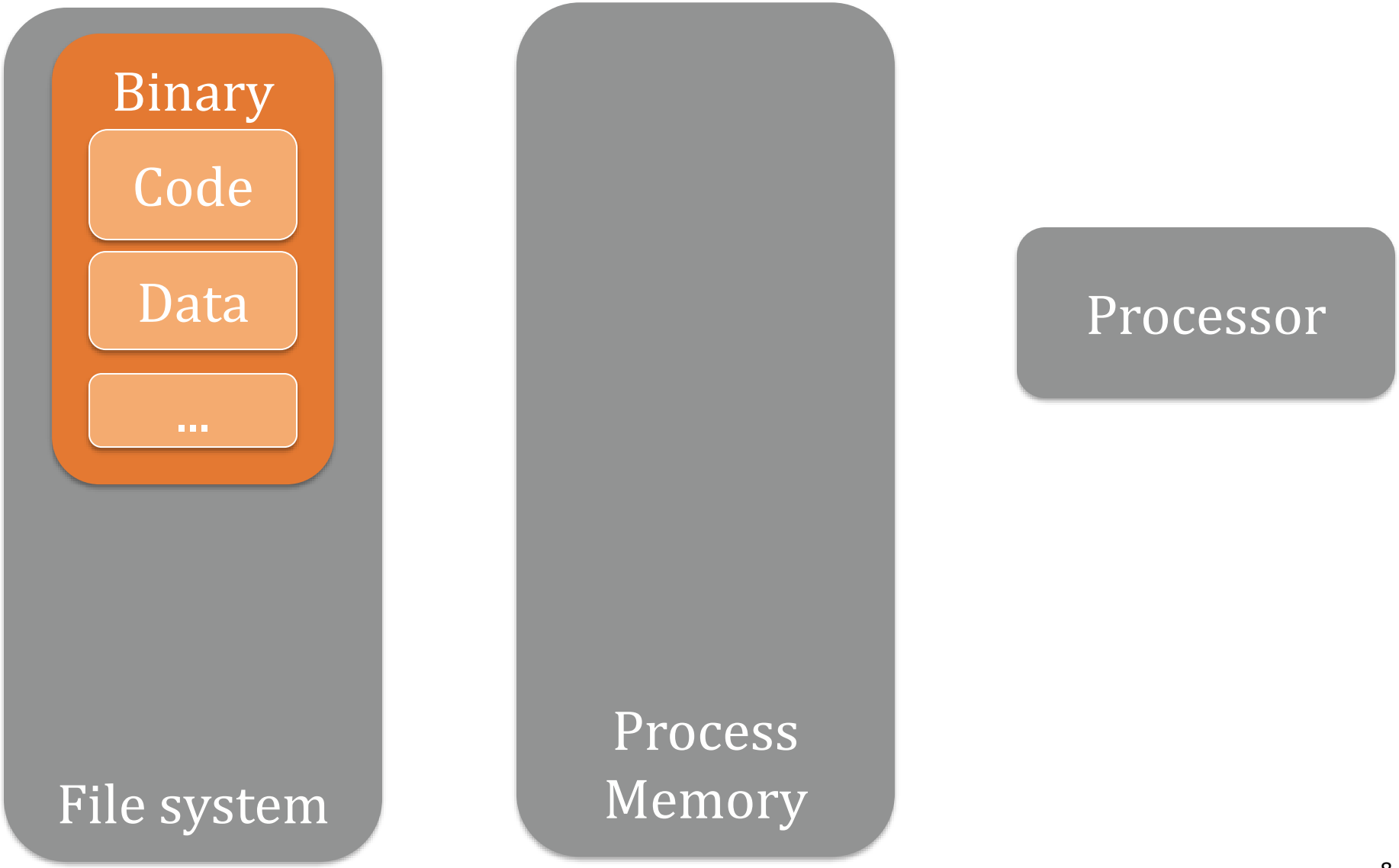


- C Code
  - Add two signed integers
- Assembly
  - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    - **x:** Register **%eax**
    - **y:** Memory **M[%ebp+8]**
    - **t:** Register **%eax**
      - function return value in **%eax**
- Object Code
  - 3-byte instruction
  - Stored at address **0x80483ca**

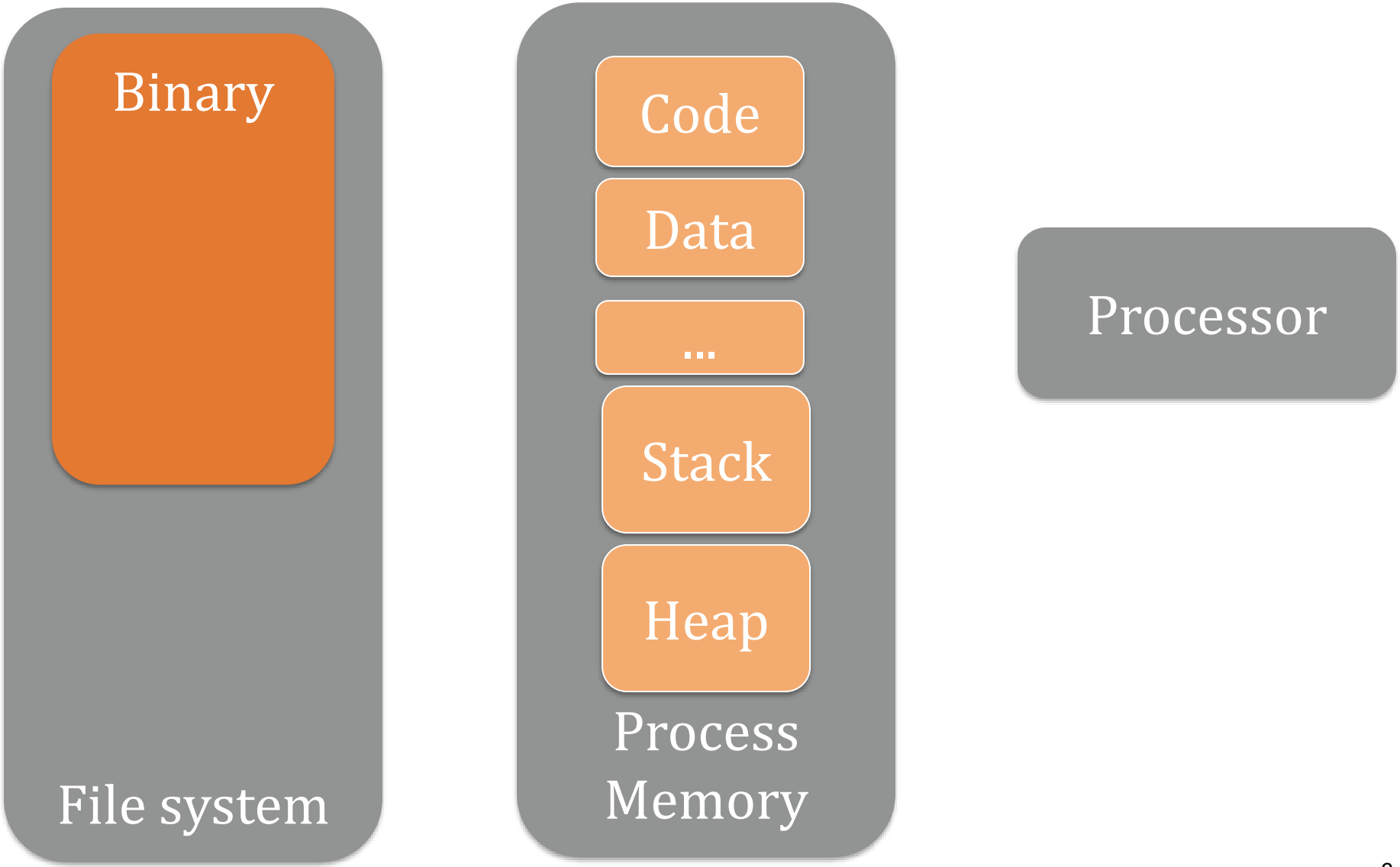
Agenda

- Compilation Workflow
- x86 Execution Model
  - Basic Execution
  - Memory Operation
  - Control Flow
  - Memory Organization

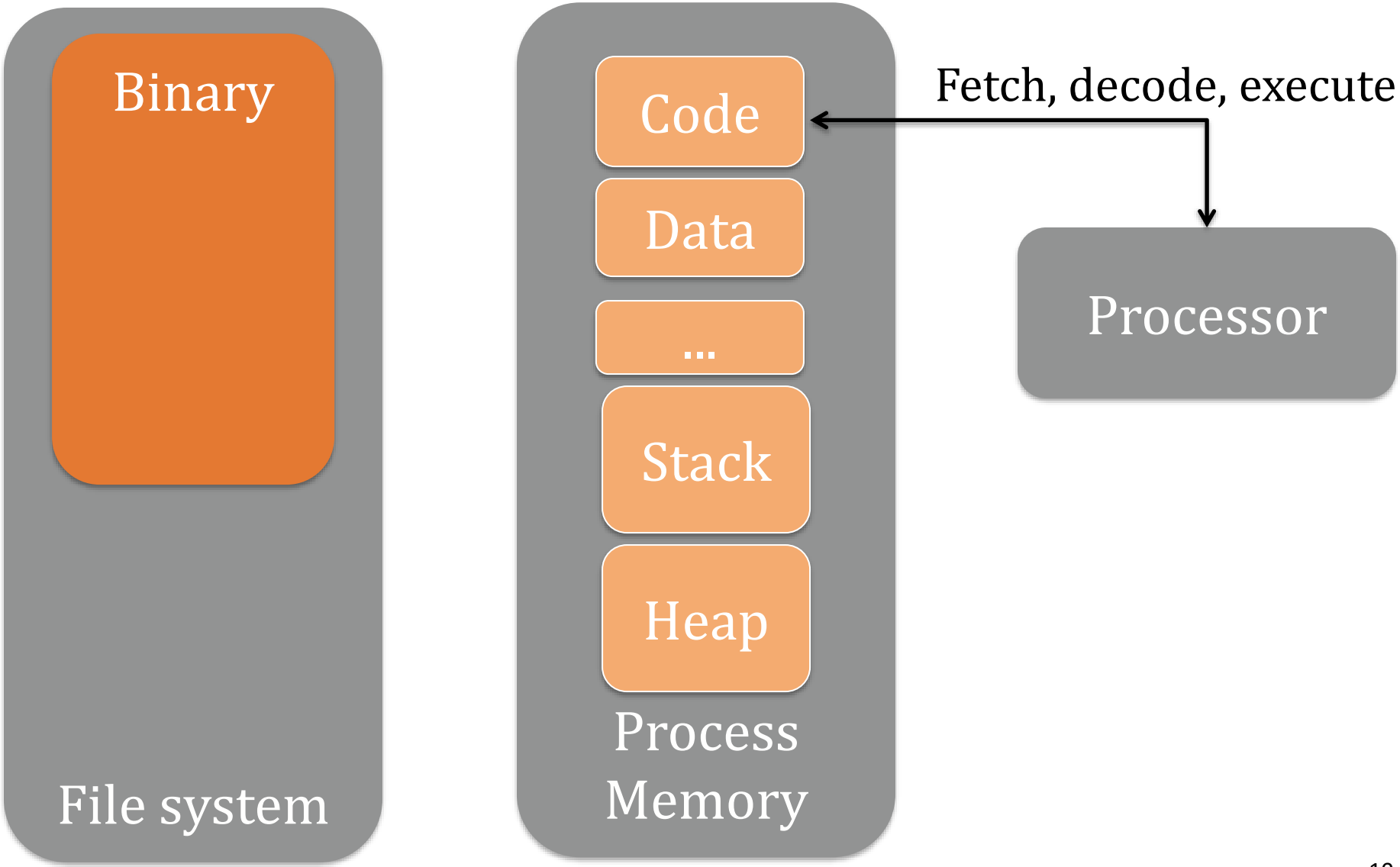
Basic Execution



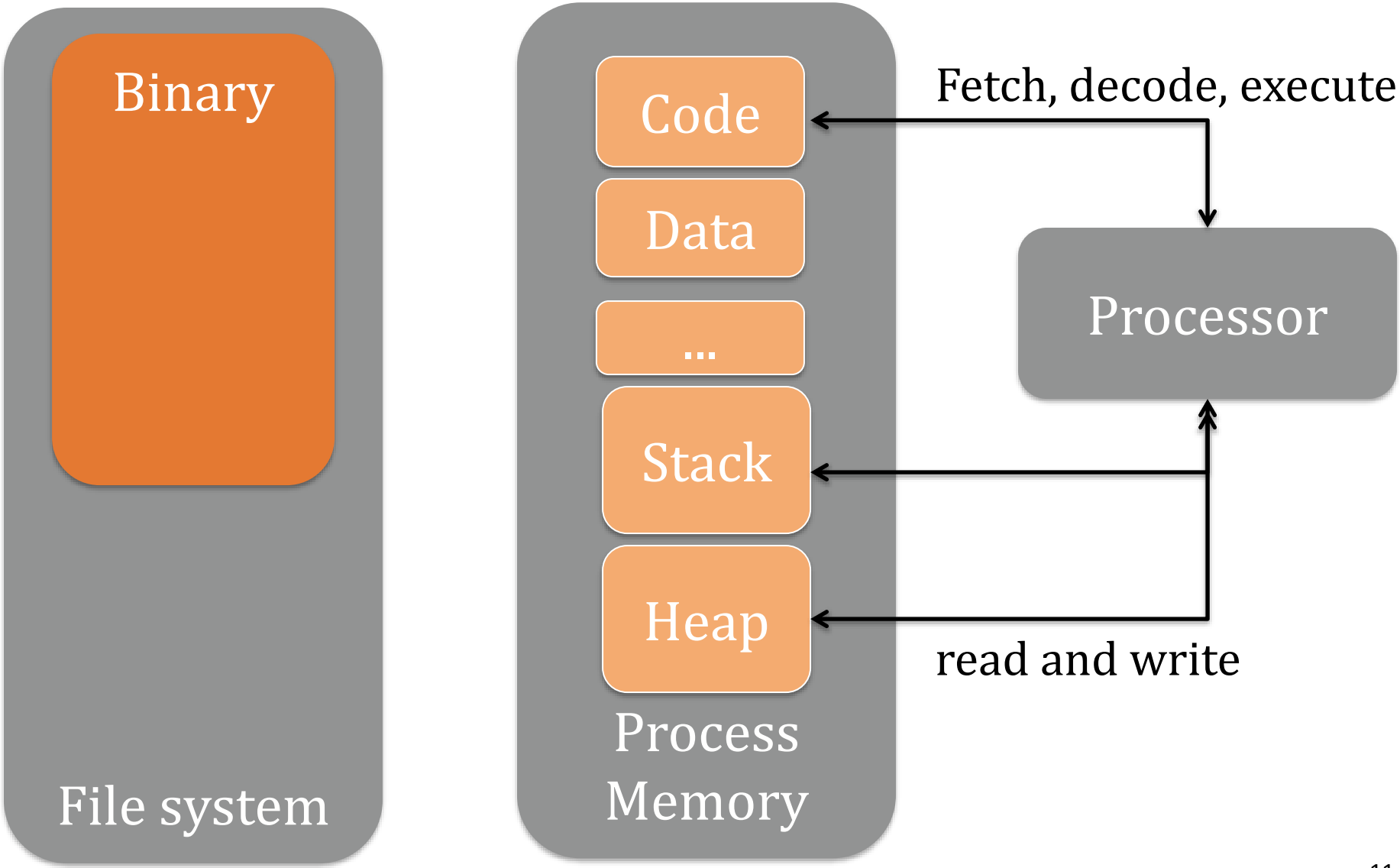
Basic Execution

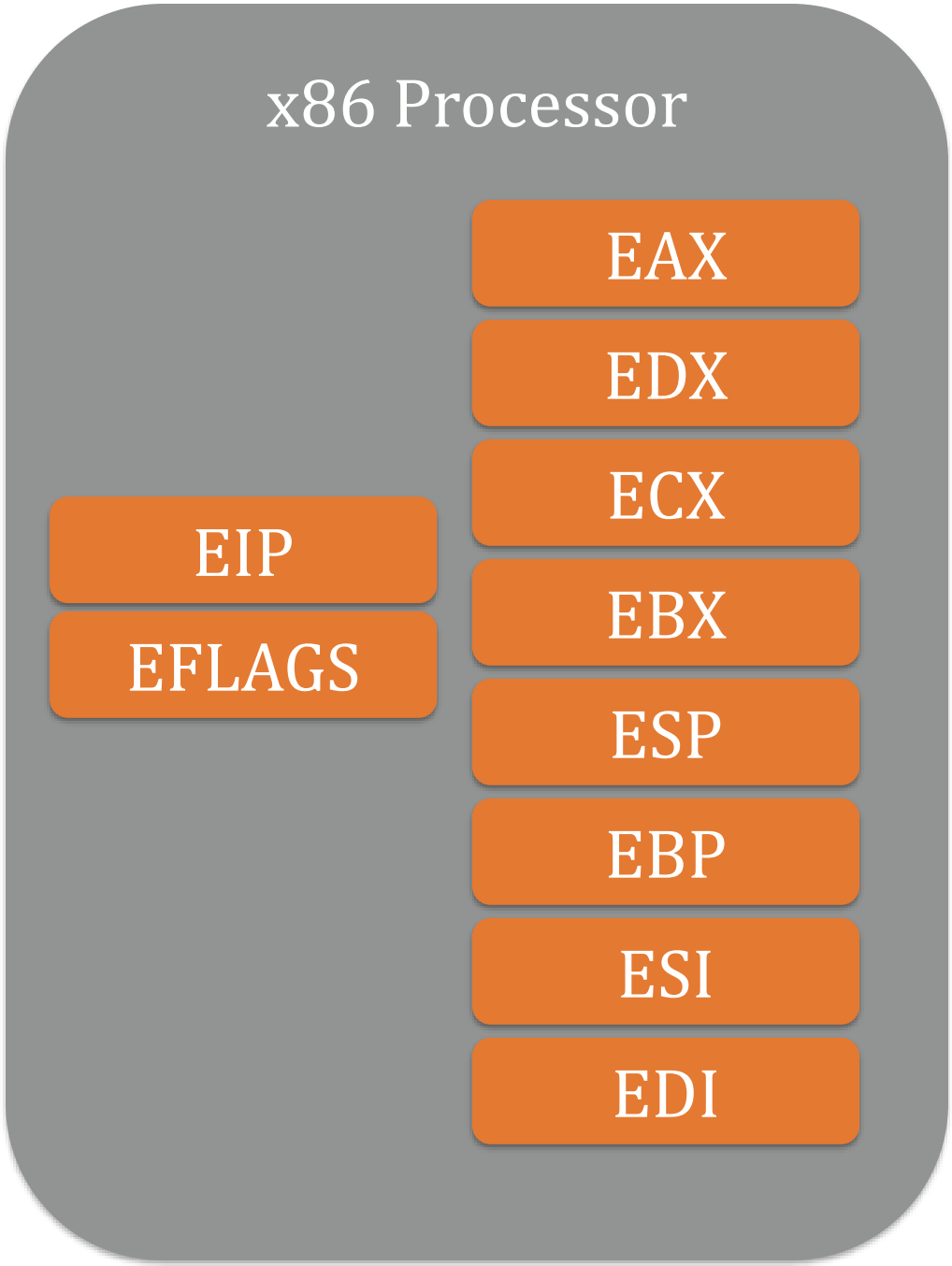


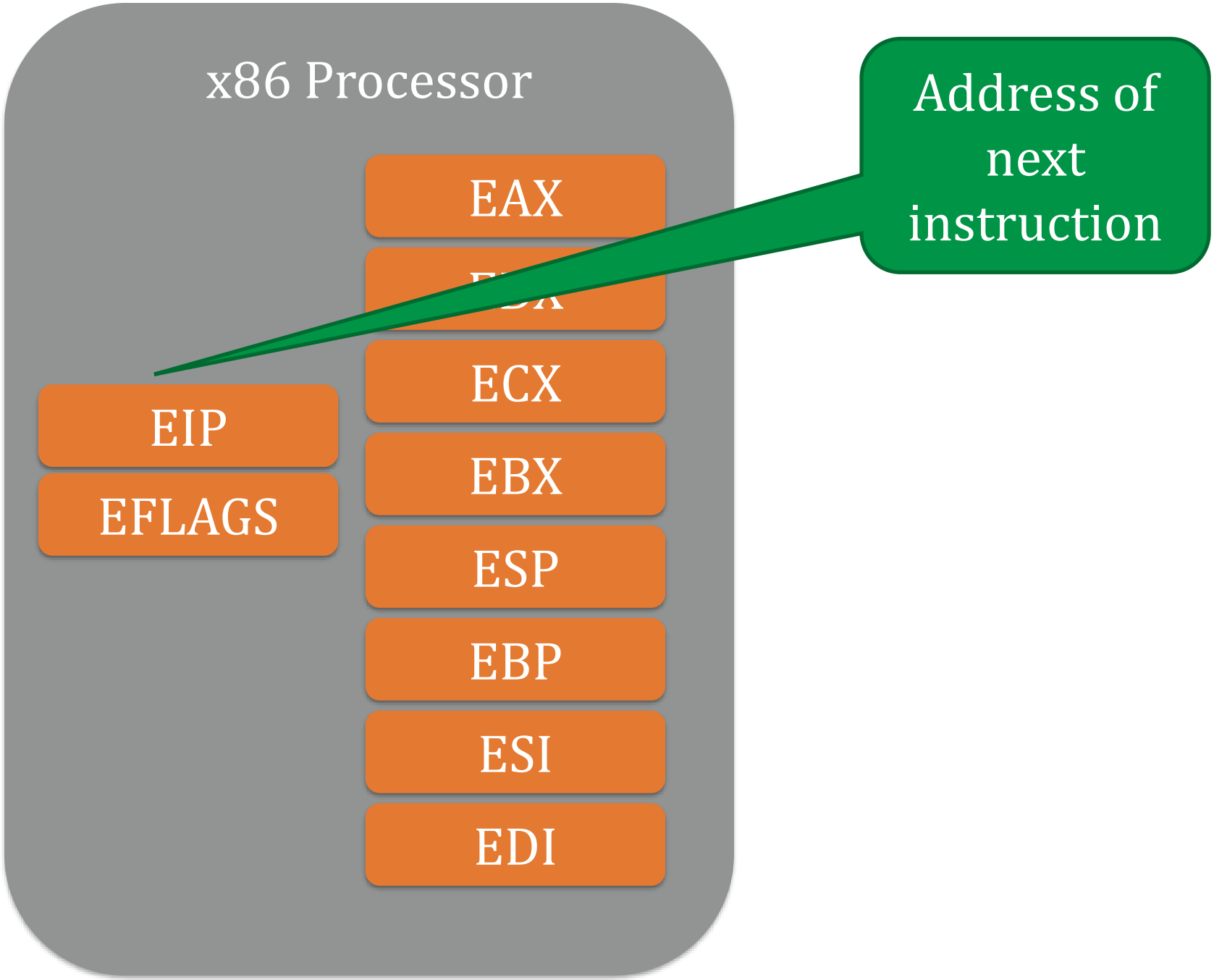
Basic Execution

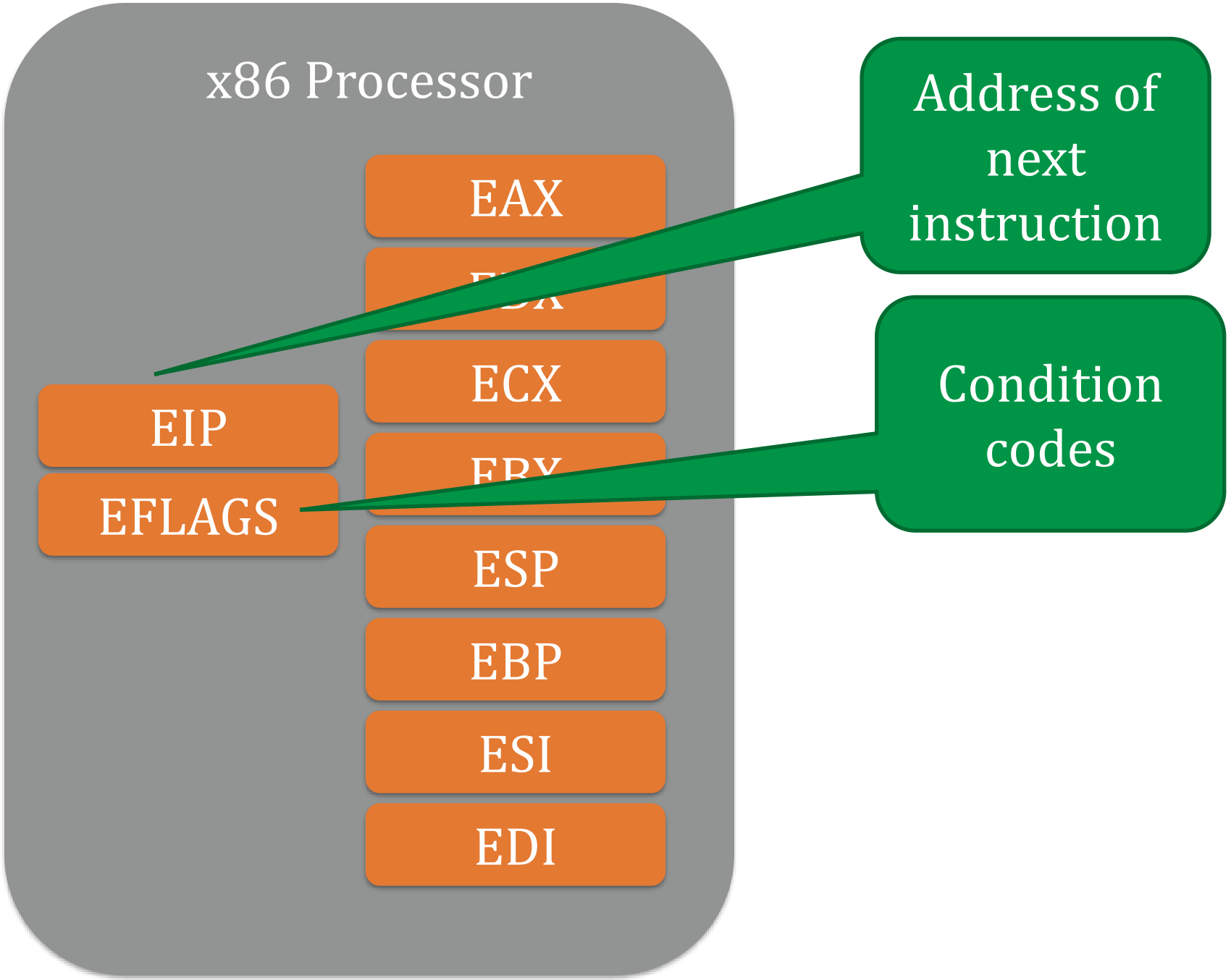


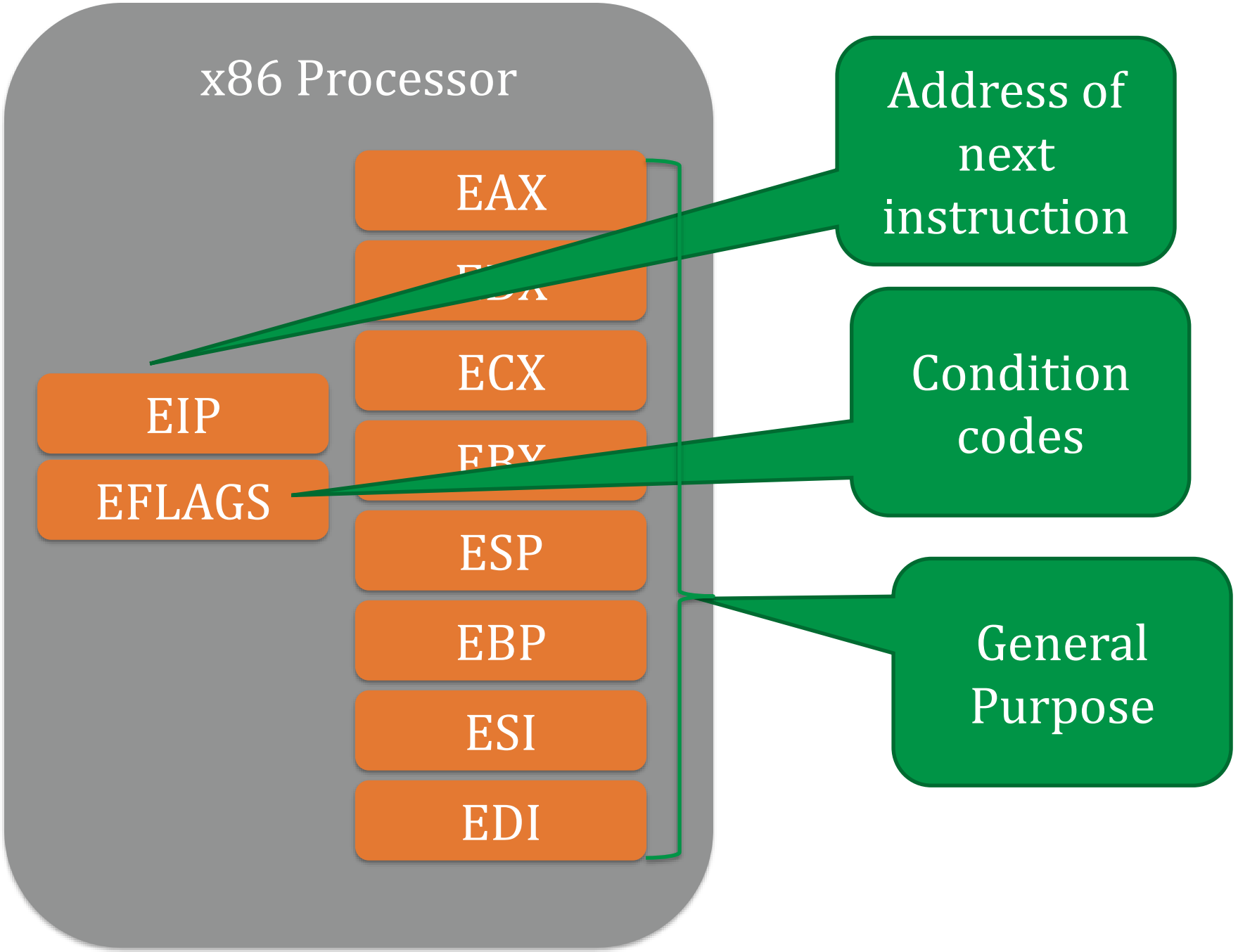
Basic Execution




x86 Processor
EAX
EDX
ECX
EBX
ESP
EBP
ESI
EDI
EIP
EFLAGS

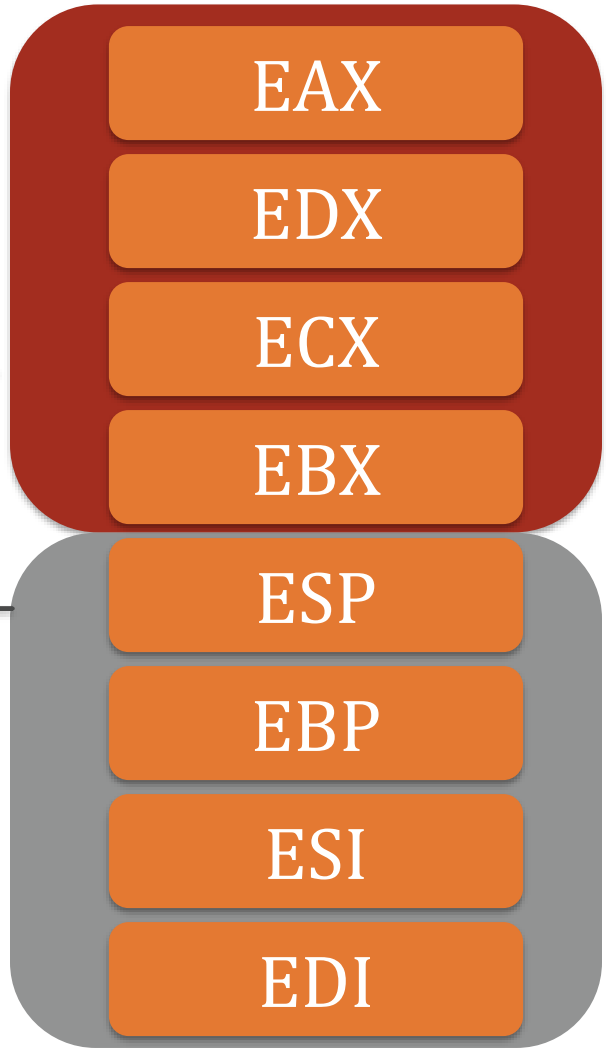

x86 Processor
EAX
ECX
EIP
EBX
EFLAGS
ESP
EBP
ESI
EDI
Address of next instruction


x86 Processor
EAX
ECX
EIP
EFLAGS
ESP
EBP
ESI
EDI
Address of next instruction
Condition codes

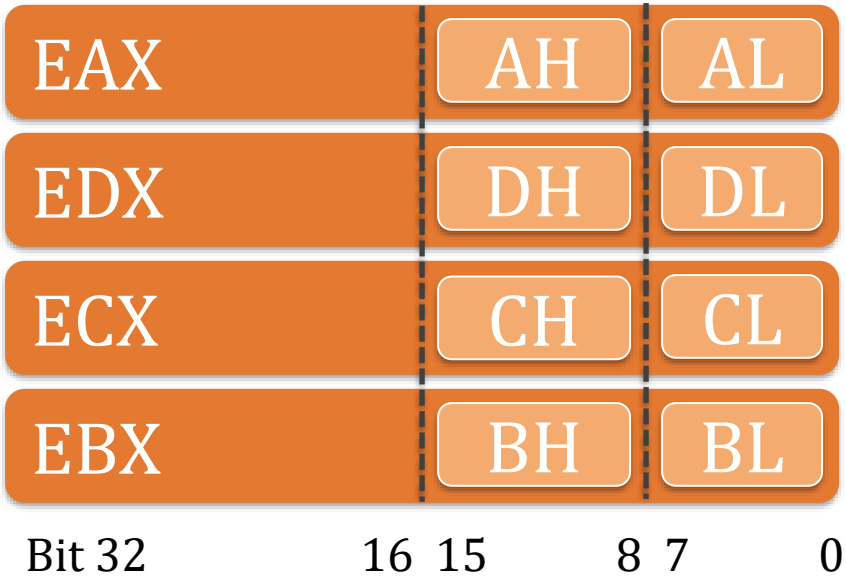

x86 Processor
EAX
ECX
EIP
EFLAGS
ESP
EBP
ESI
EDI
Address of next instruction
Condition codes
General Purpose

Registers have up to 4 addressing modes

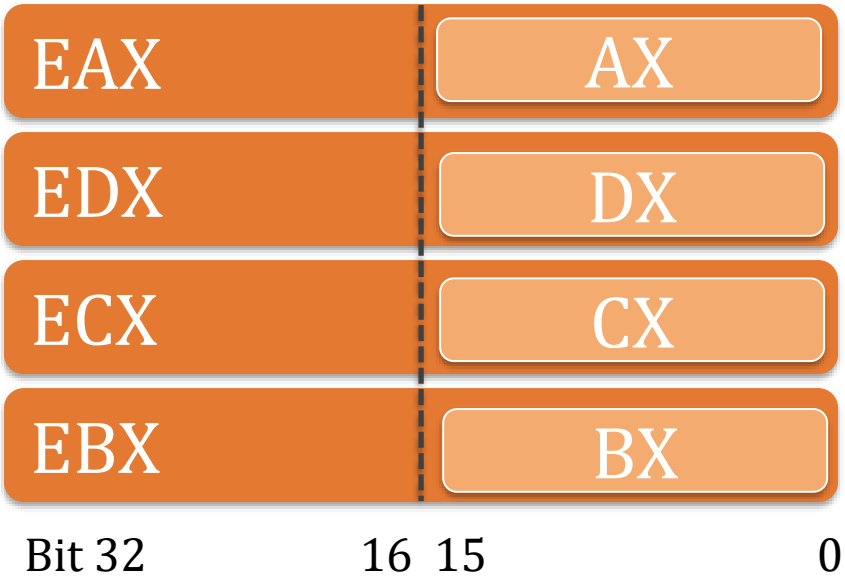
1. Lower 8 bits
2. Mid 8 bits
3. Lower 16 bits
4. Full register



EAX, EDX, ECX, and EBX

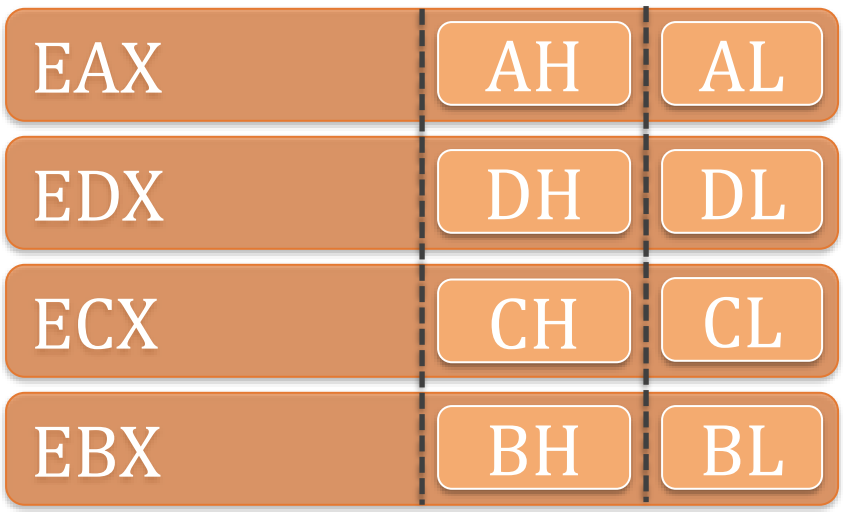


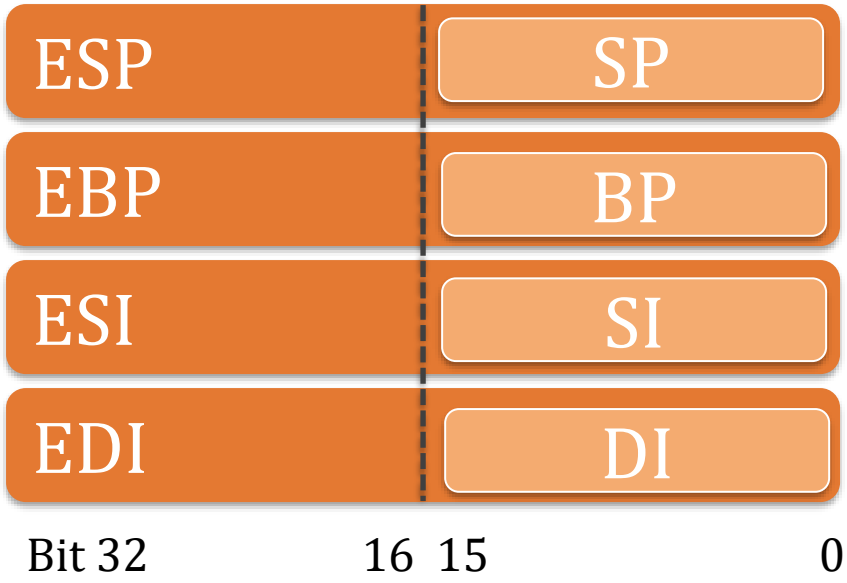
- 32 bit registers (three letters)
- Lower bits (bits 0-7) (two letters with L suffix)
- Mid-bits (bits 8-15) (two letters with H suffix)



- Lower 16 bits (bits 0-15) (2 letters with X suffix)

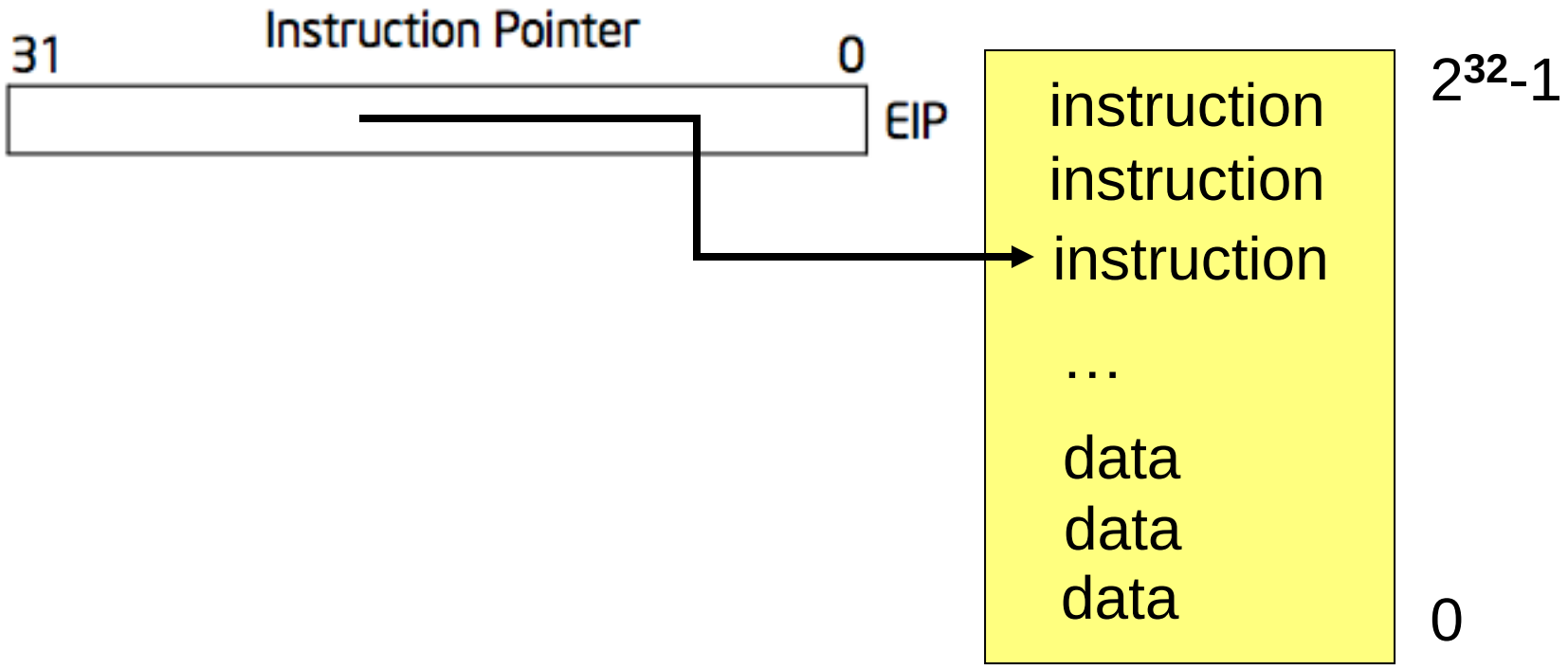
ESP, EBP, ESI, and EDI





- Lower 16 bits (bits 0-15) (2 letters)

x86 Implementation



- **EIP** is incremented after each instruction
- Instructions are different length
- EIP modified by CALL, RET, JMP, and cond. JMP

x86 Instruction Set

- Instructions classes:
  - Data Movement: MOV, PUSH, POP, …
  - Arithmetic: TEST, SHL, ADD, …
  - I/O: IN, OUT, …
  - Control: JMP, JZ, JNZ, CALL, RET
  - String: REP, MOVSB, …
  - System: IRET, INT, …

- Volume 2A: Instruction Set Reference, A-M
  Volume 2B: Instruction Set Reference, N-Z
  - Intel syntax: OP DST, SRC
  - AT&T (gcc/gas) syntax: OP SRC, DST

Basic Ops and AT&T vs Intel Syntax

source first

destination first

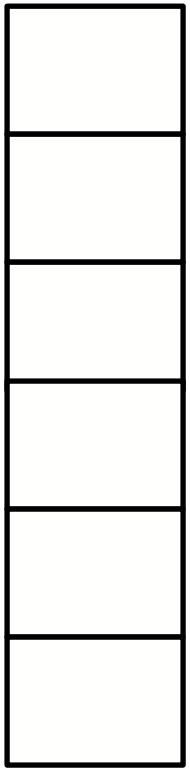
Meaning	AT&T	Intel
ebx = eax	movl %eax, %ebx	mov ebx, eax
eax = eax + ebx	addl %ebx, %eax	add eax, ebx
ecx = ecx << 2	shl $2, %ecx	shl ecx, 2

- AT&T is at odds with assignment order (e.g., in C). It is the default for objdump, and traditionally used for UNIX.
- Intel order *mirrors* assignment. Windows traditionally uses Intel, as is available via the objdump '-M intel' command line option. IDA uses Intel.

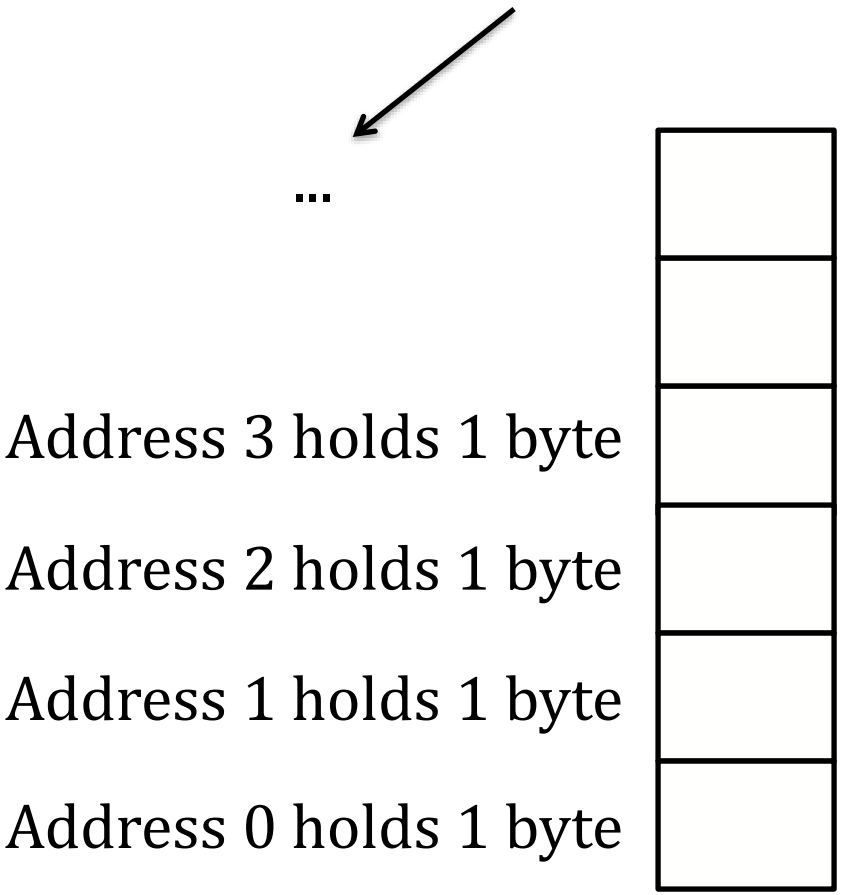
Agenda

- Compilation Workflow
- x86 Execution Model
  - Basic Execution
  - Memory Operation
  - Control Flow
  - Memory Organization

x86: Byte Addressable

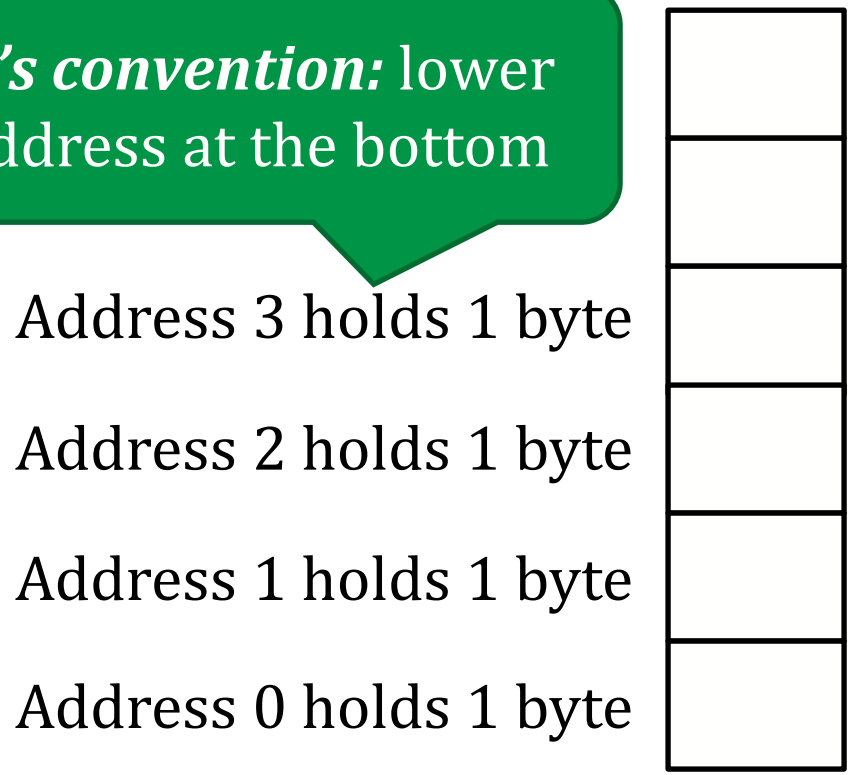


x86: Byte Addressable

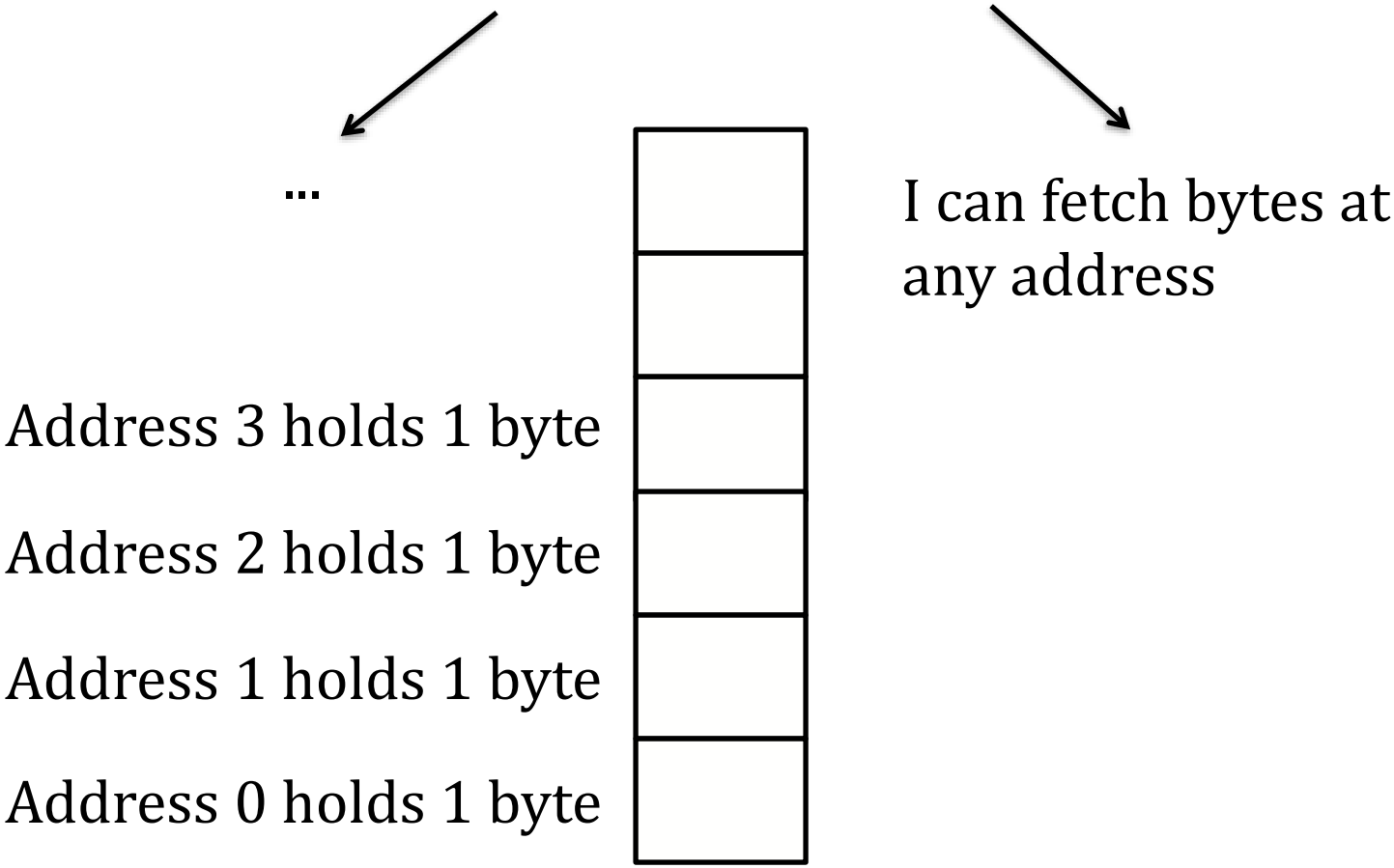


x86: Byte Addressable

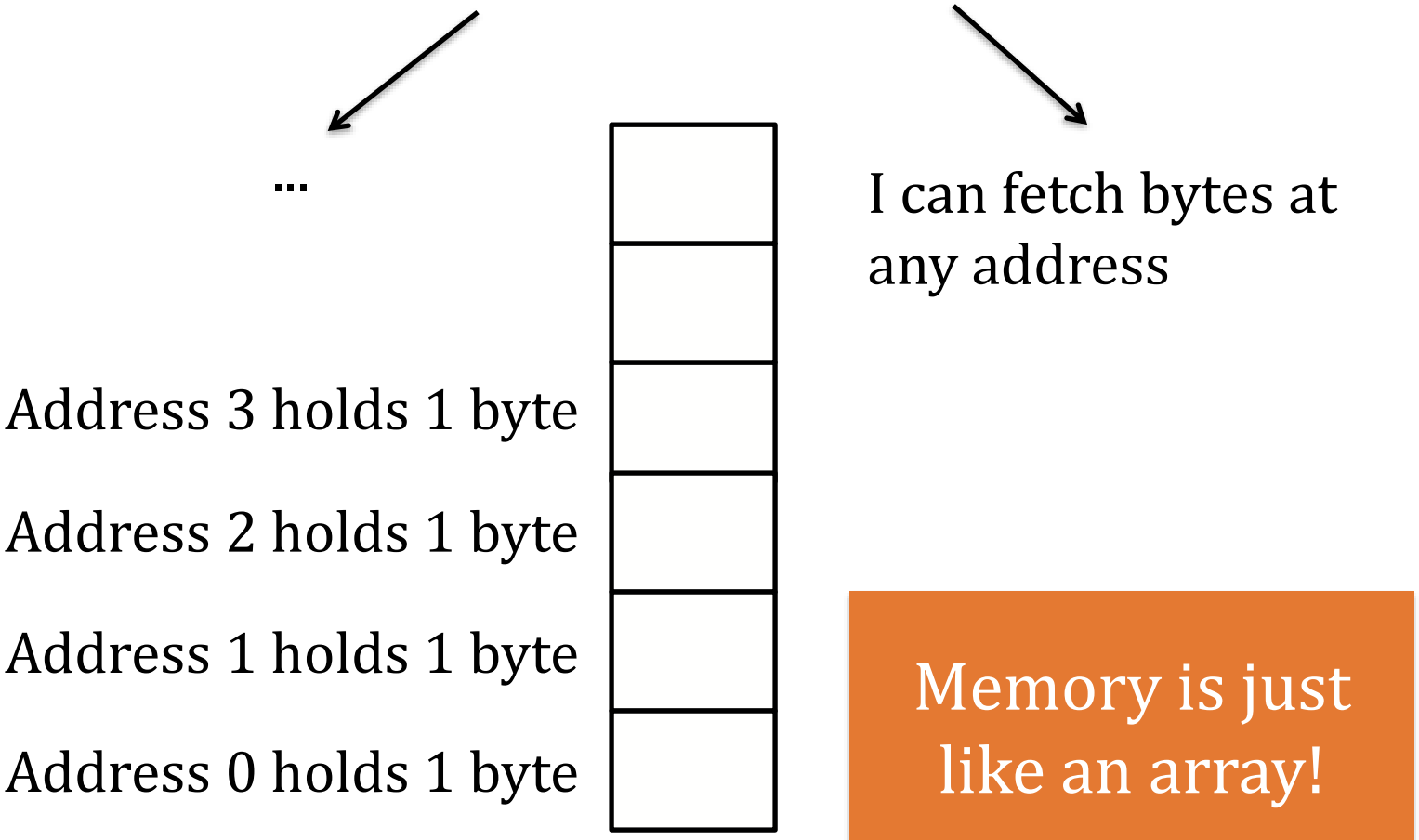
It's convention: lower address at the bottom



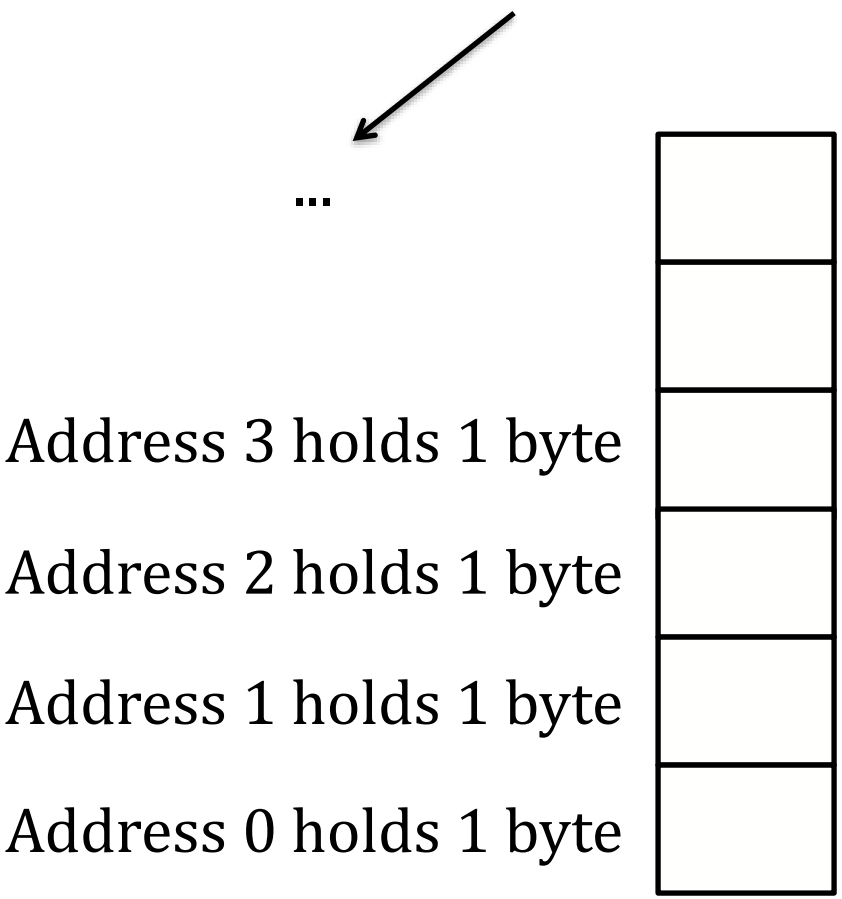
x86: Byte Addressable



x86: Byte Addressable



x86: Byte Addressable



I can fetch bytes at any address

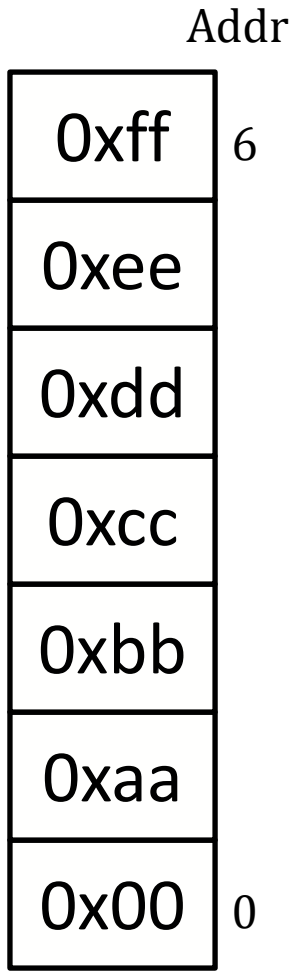
Memory is just like an array!

Alternative: **Word addressable**

Example: For 32-bit word size, it's valid to fetch 4 bytes from Mem[0], but not Mem[6] since 6 is not a multiple of 4

x86: Addressing bytes

Addresses are indicated by operands that have a bracket "[]" or paren "()", for Intel vs. AT&T, resp.



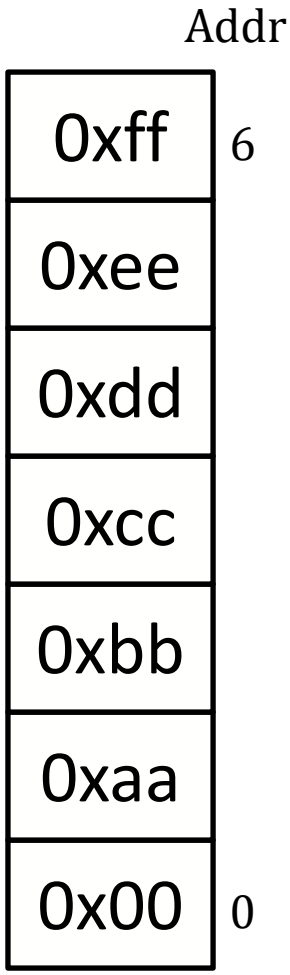
x86: Addressing bytes

Addresses are indicated by operands that have a bracket "[]" or paren "()", for Intel vs. AT&T, resp.

What does mov dl, [al] do?

Move the data at address specified in **al** to **dl**

Register	Value
eax	0x3
edx	0x0
ebx	0x5



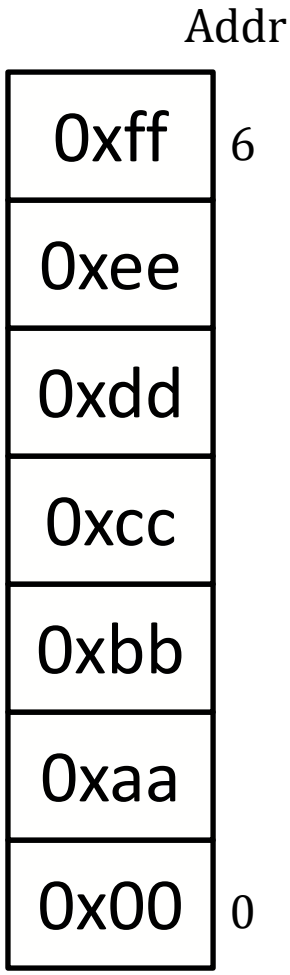
x86: Addressing bytes

Addresses are indicated by operands that have a bracket "[]" or paren "()", for Intel vs. AT&T, resp.

Register	Value
eax	0x3
edx	0x0
ebx	0x5

What does mov dl, [al] do?

Moves 0xcc into dl

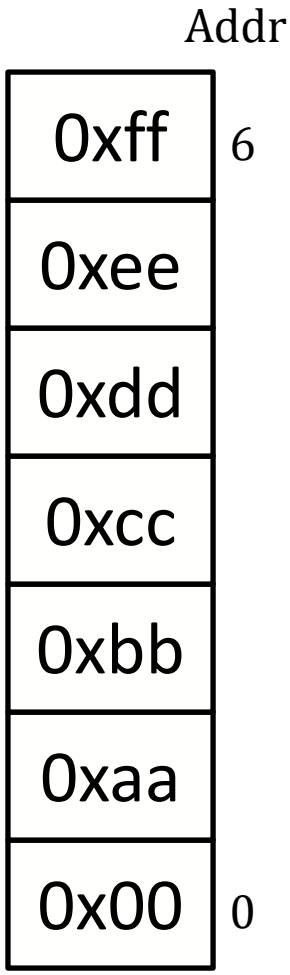


x86: Addressing bytes

Addresses are indicated by operands that have a bracket "[]" or paren "()", for Intel vs. AT&T, resp.

What does mov edx , [eax] do?

Register	Value
eax	0x3
edx	0xcc
ebx	0x5



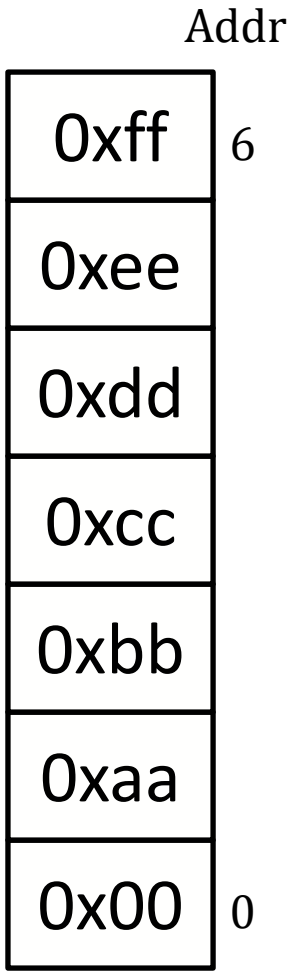
x86: Addressing bytes

Addresses are indicated by operands that have a bracket "[]" or paren "()", for Intel vs. AT&T, resp.

What does mov edx , [eax] do?

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Which 4 bytes get moved, and which is the LSB in edx?

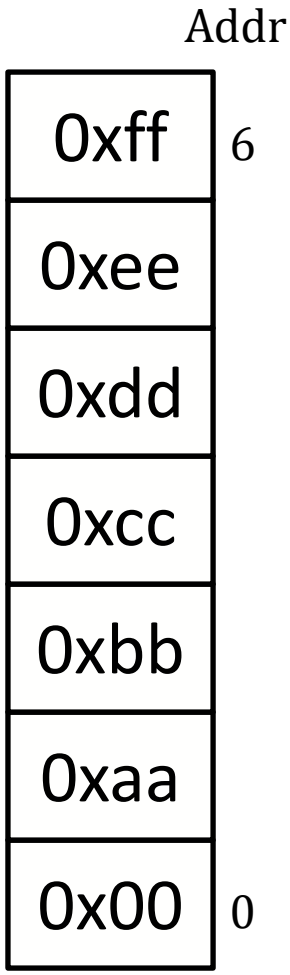


Endianess

- *Endianess*: Order of individually addressable units
- *Little Endian*: Least significant byte first

so address *a* goes in littlest byte (e.g., AL), *a+1* in the next (e.g., AH), etc.

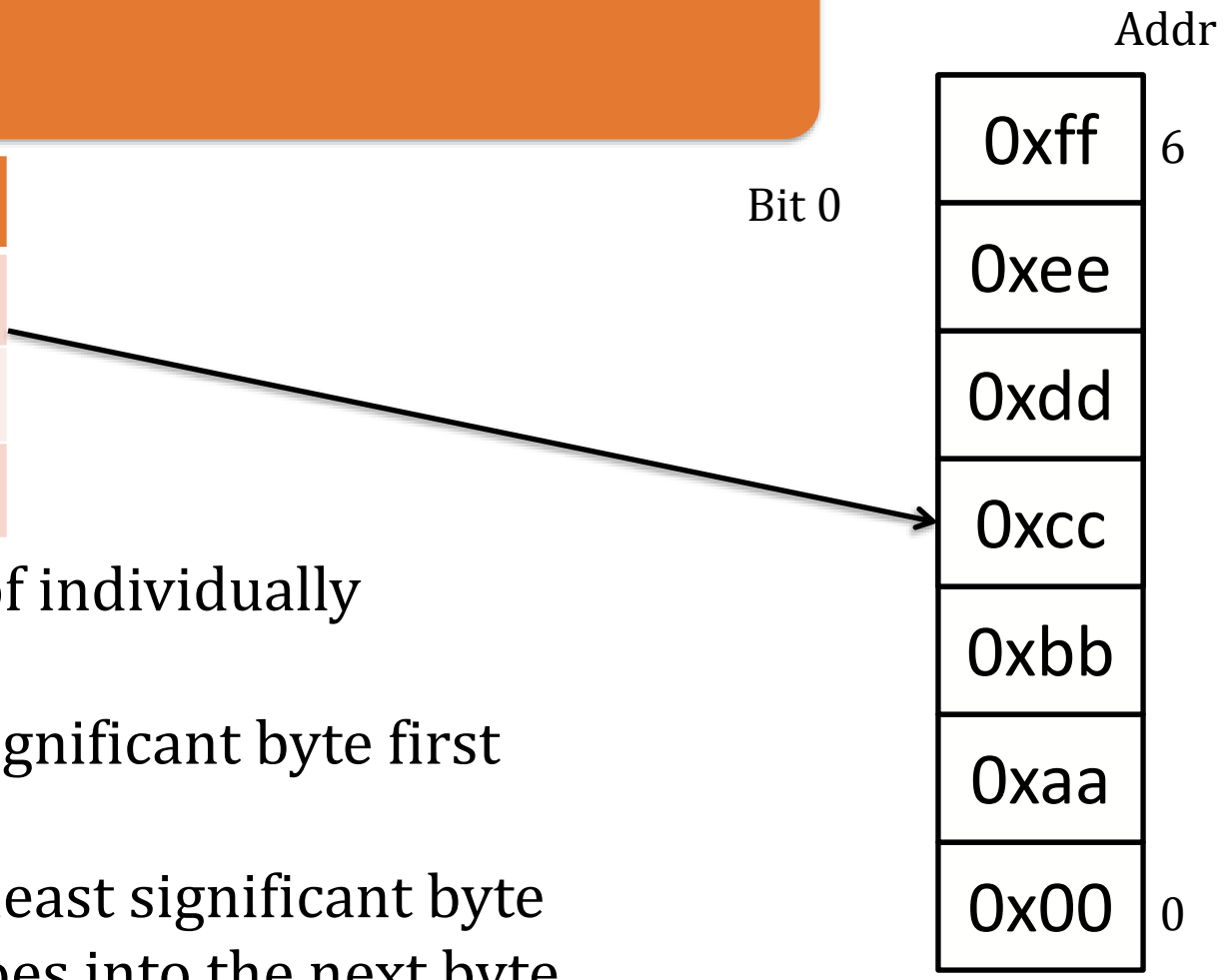
Register	Value
eax	0x3
edx	0xcc
ebx	0x5



mov edx, [eax]

EDX

Register	Value
eax	0x3
edx	0xcc
ebx	0x5



Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

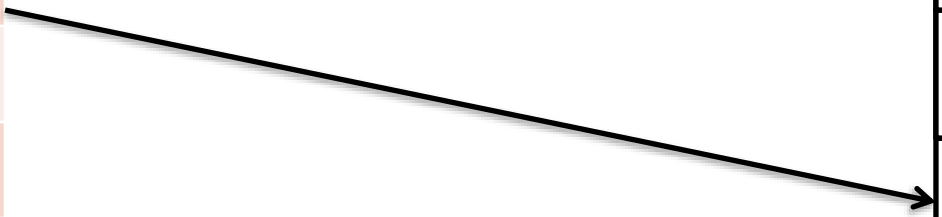
... so ...

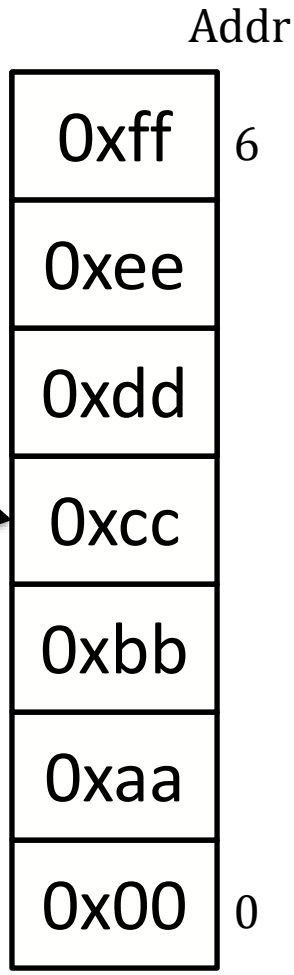
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov edx, [eax]

EDX 0xcc

Register	Value
eax	0x3
edx	0xcc
ebx	0x5





Endianess: Ordering of individually addressable units

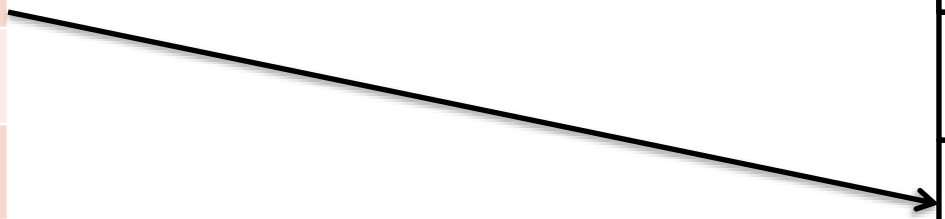
Little Endian: Least significant byte first

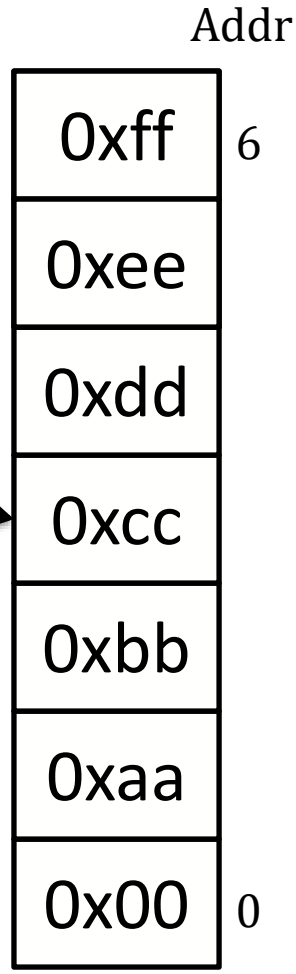
... so ...

address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov edx, [eax]

Register	Value
eax	0x3
edx	0xcc
ebx	0x5





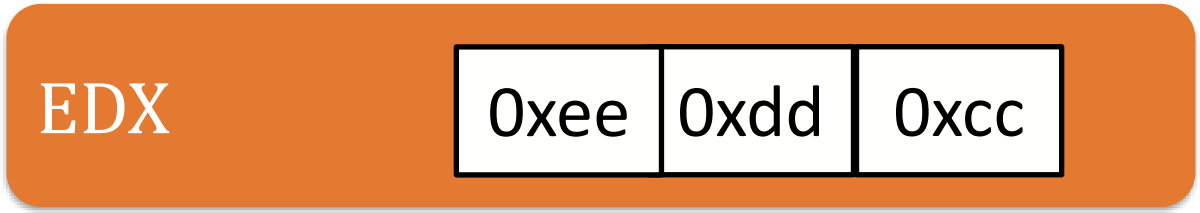
Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

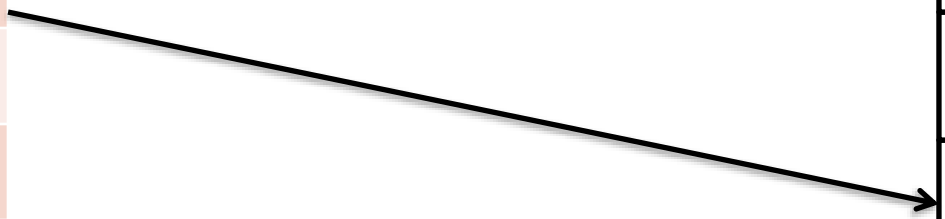
... so ...

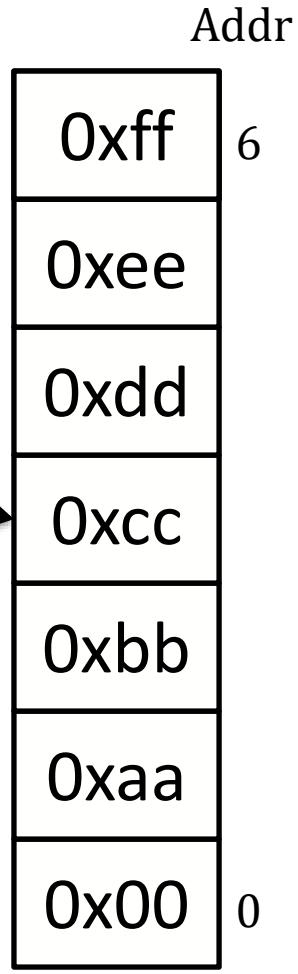
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov edx, [eax]



Register	Value
eax	0x3
edx	0xcc
ebx	0x5





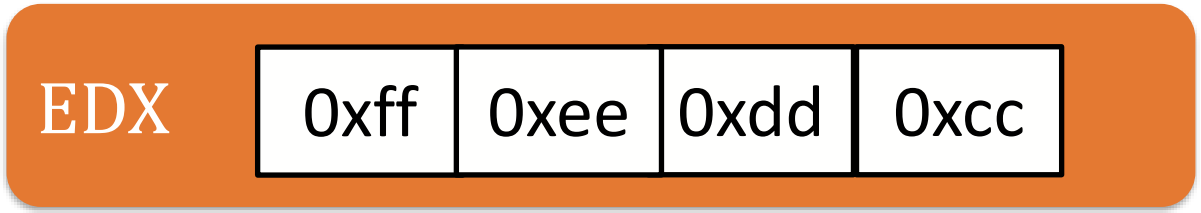
Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

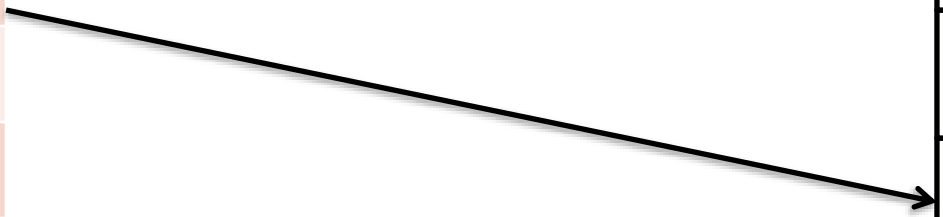
... so ...

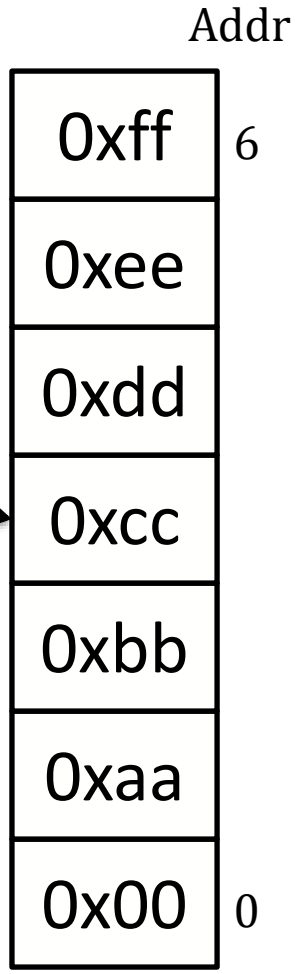
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov edx, [eax]



Register	Value
eax	0x3
edx	0xcc
ebx	0x5





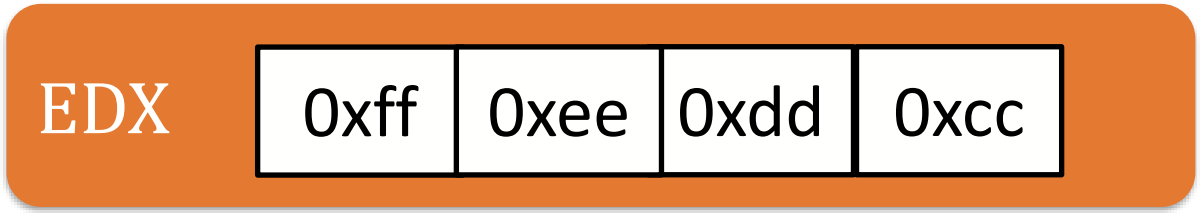
Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

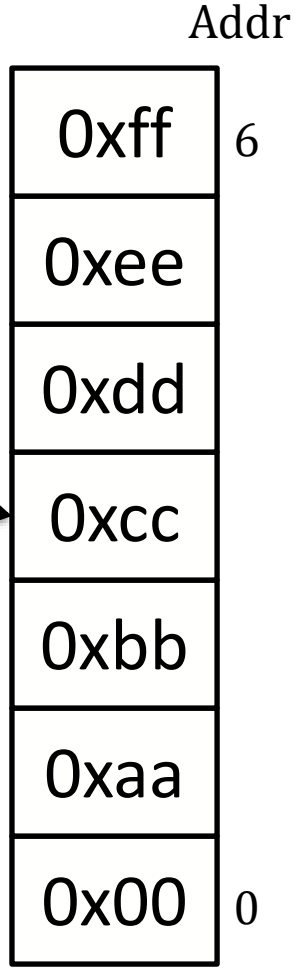
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov edx, [eax]



Register	Value
eax	
edx	
ebx	

EDX = 0xffeeddcc!



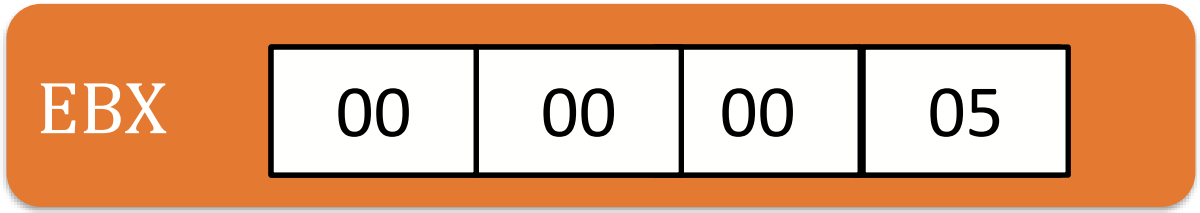
Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

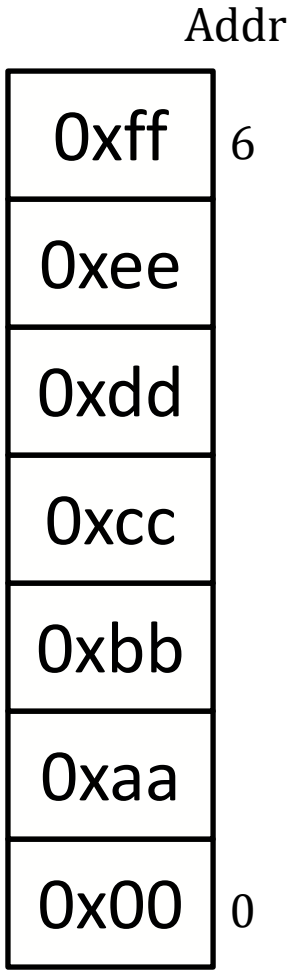
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov [eax], ebx



Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0



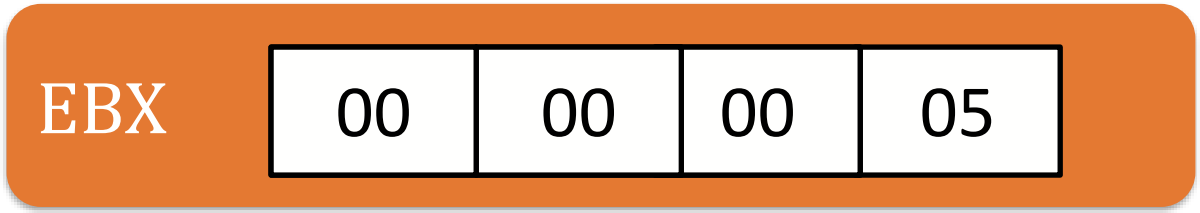
Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

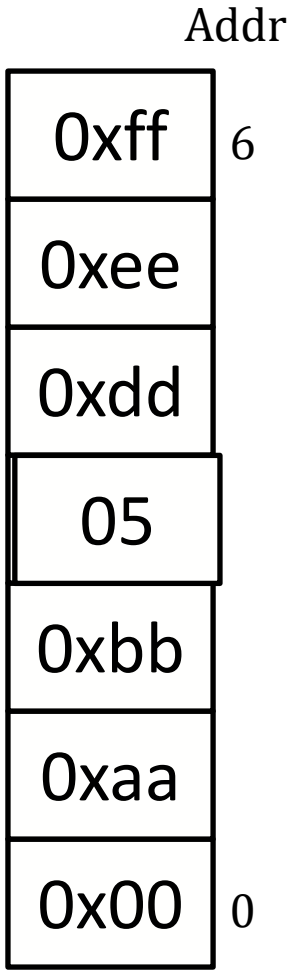
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov [eax], ebx



Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0



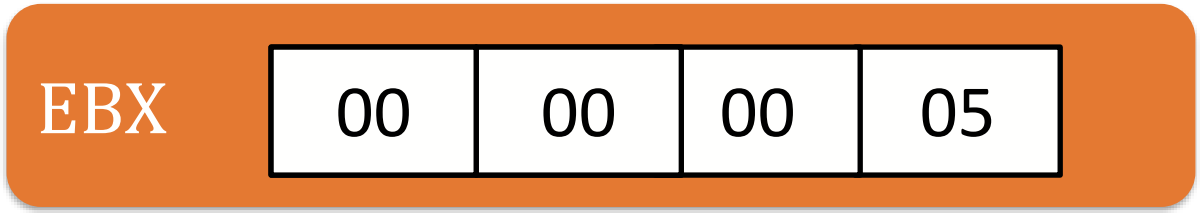
Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

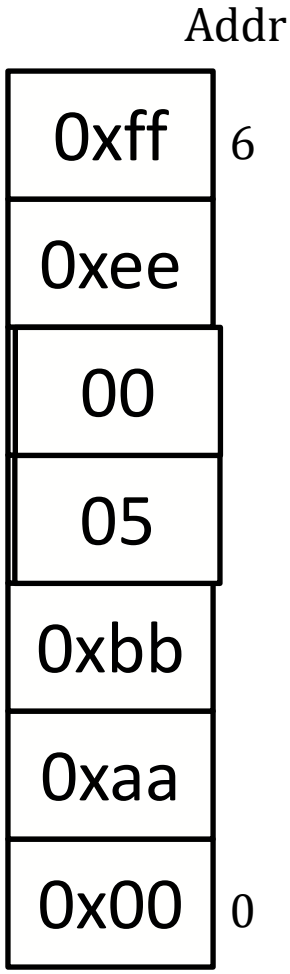
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov [eax], ebx



Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0



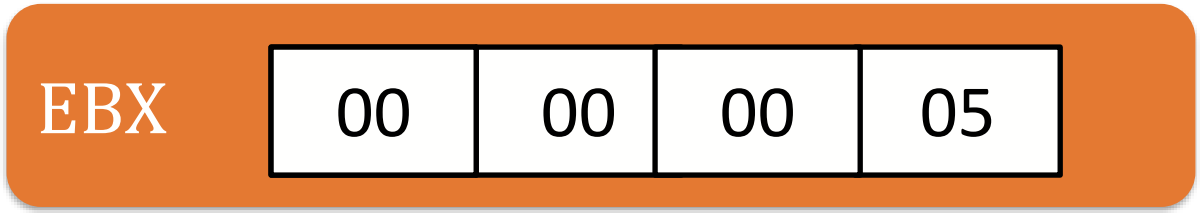
Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

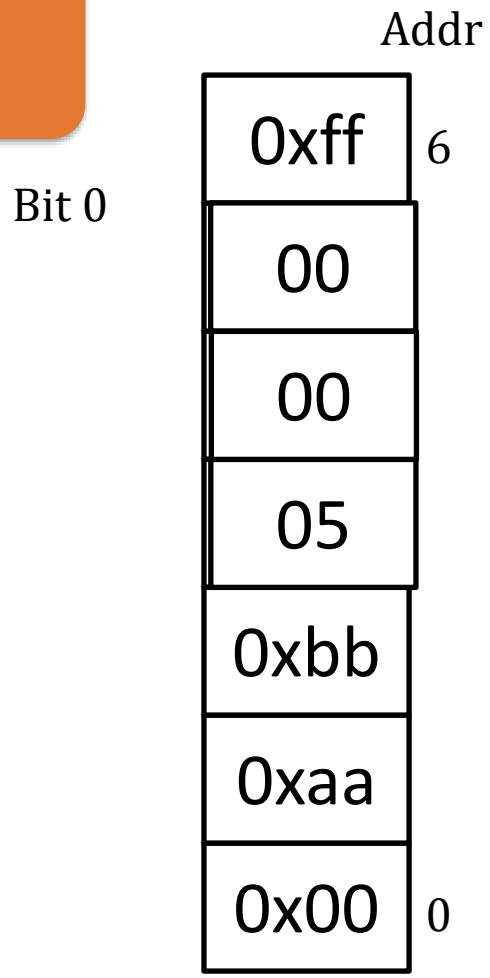
... so ...

address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov [eax], ebx



Register	Value
eax	0x3
edx	0xcc
ebx	0x5



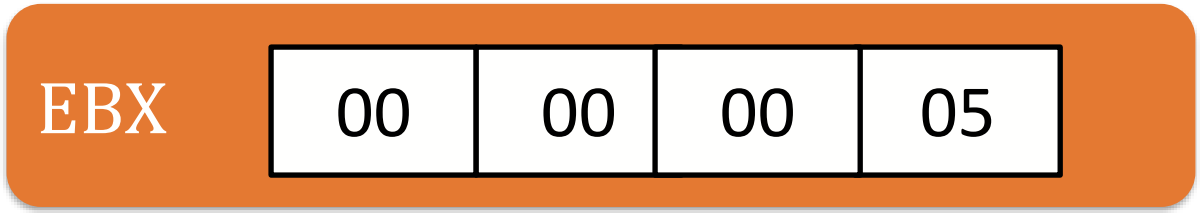
Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

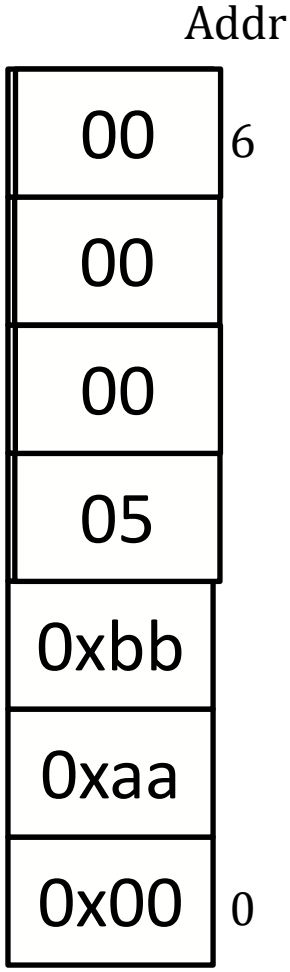
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov [eax], ebx



Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0



Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

There are other ways to address memory than just [register].

These are called *Addressing Modes.*

An ***Addressing Mode*** specifies how to calculate the effective memory address of an operand by using information from registers and constants contained within the instruction or elsewhere.

Motivation: Addressing Buffers

```
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```

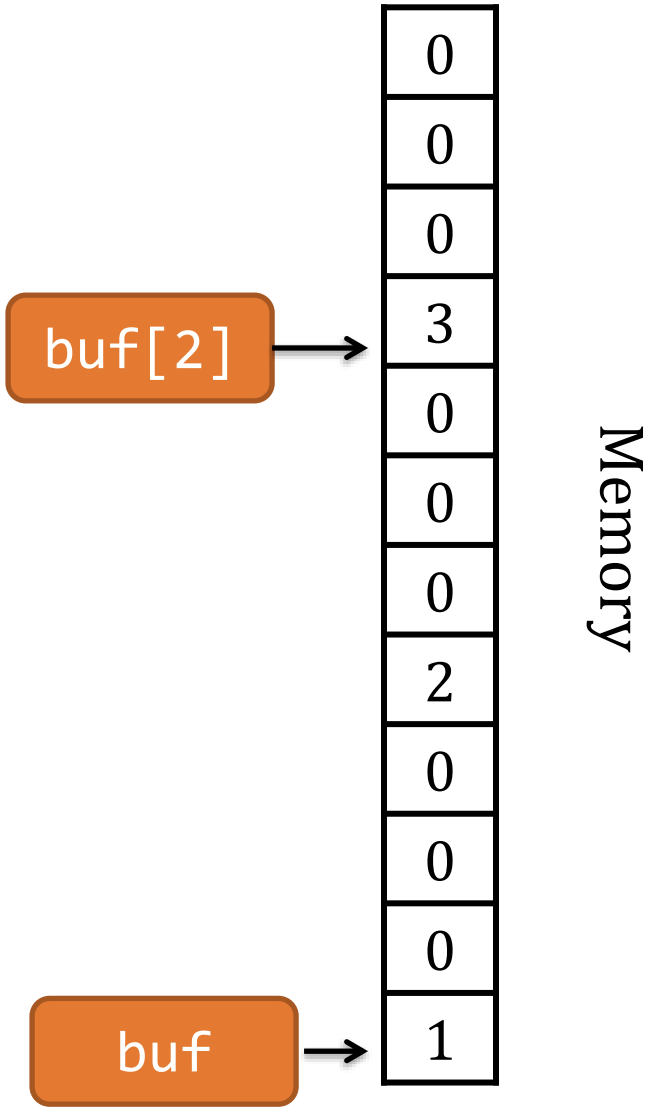
Can be done in a single x86 instruction!

Motivation: Addressing Buffers

```
typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;

w = buf[2];
x = *(buf + 2);
```

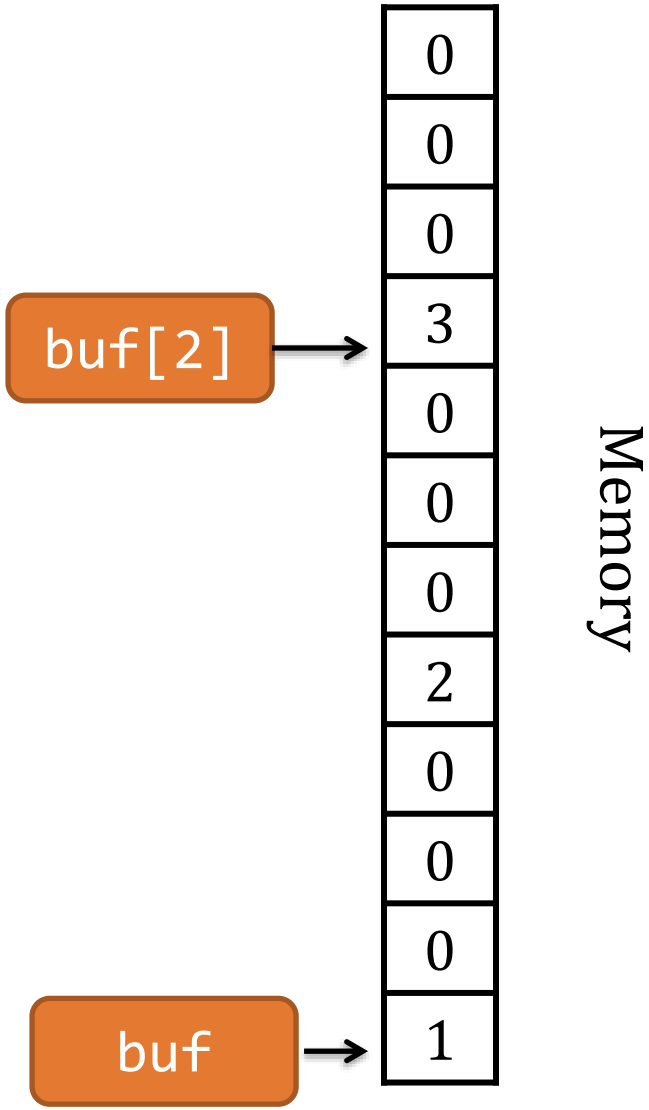
What is x? what memory cell does it ref?



Motivation: Addressing Buffers

```
typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;

w = buf[2];
x = *(buf + 2);
y = *( (uint32_t *) (ptr+8));
```



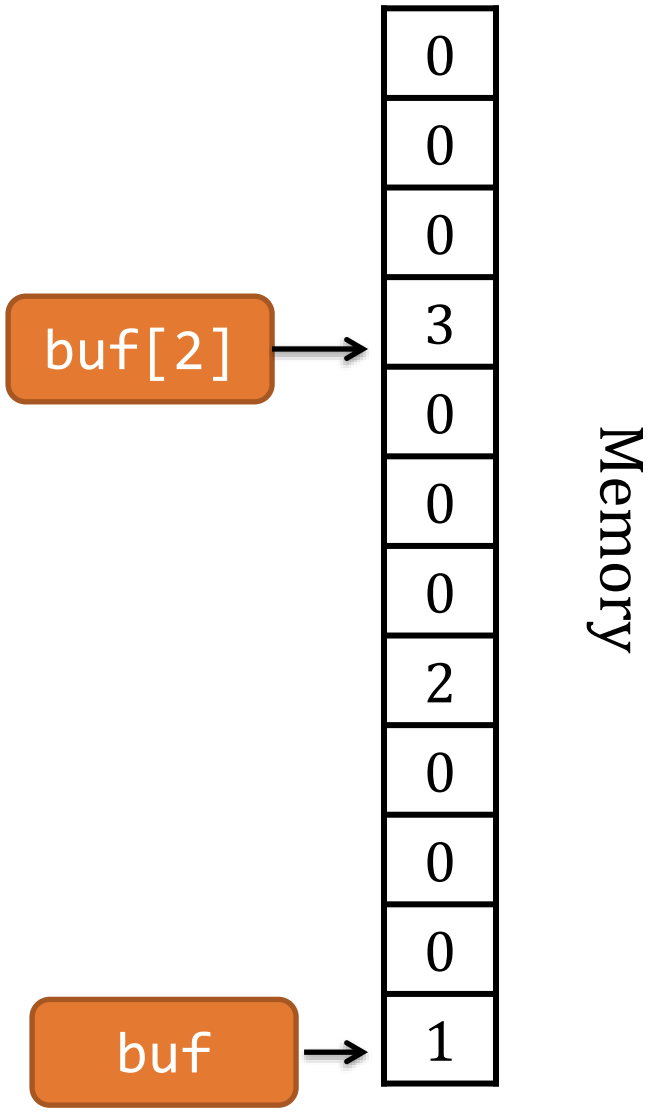
Motivation: Addressing Buffers

```
typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;
```

```
w = buf[2];
x = *(buf + 2);
y = *( (uint32_t *) (ptr+8));
```

Equivalent

(addr_t) (ptr + 8) = (uint32_t *) buf+2



Motivation: Addressing Buffers

```
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```

Motivation: Addressing Buffers

```
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```

Say at imm + r_1

Motivation: Addressing Buffers

```
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```

Say at imm + r_1

Constant ***scaling*** factor s, typically 1, 2, 4, or 8

Motivation: Addressing Buffers

```
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```

Say at imm +r_1

Constant ***scaling*** factor s, typically 1, 2, 4, or 8

Say in Register r_2

Motivation: Addressing Buffers

```
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```

Say at imm +r_1

Constant ***scaling*** factor s, typically 1, 2, 4, or 8

Say in Register r_2

$$imm + r_1 + s*r_2$$

AT&T: imm (r_1, r_2, s)

Intel: $r_1 + r_2*s + imm$

AT&T Addressing Modes for Common Codes

Form	Meaning on memory M
imm (r)	M[r + imm]
imm (r_1, r_2)	M[r_1 + r_2 + imm]
imm (r_1, r_2, s)	M[r_1 + r_2*s + imm]
imm	M[imm]

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x0100`

Expression	Address Computation	Address
`0x8(%edx)`		
`(%edx,%ecx)`		
`(%edx,%ecx,4)`		
`0x80(,%edx,2)`		

Referencing Memory

Loading a value from memory: mov

```
<eax> = *buf;
```

```
mov   -0x38(%ebp),%eax (A)
mov   eax, [ebp-0x38] (I)
```

Loading an address: lea

```
<eax> = buf;
```

```
lea   -0x38(%ebp),%eax (A)
lea   eax, [ebp-0x38] (I)
```

Suppose I want to access address **`0xdeadbeef`** directly

Loads the address

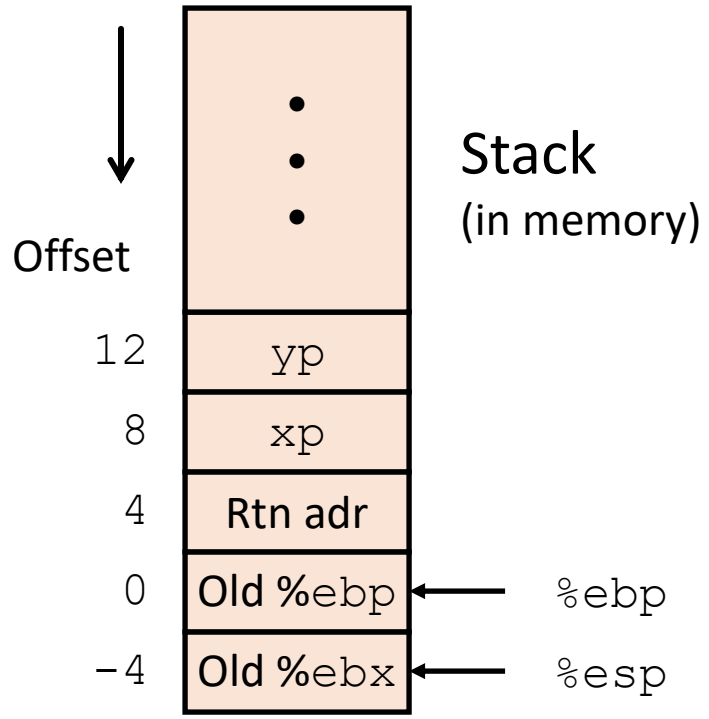
```
lea eax, 0xdeadbeef (I)
```

Deref the address

```
mov  eax, 0xdeadbeef (I)
mov  eax, [eax] (I)
```

Understanding Swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

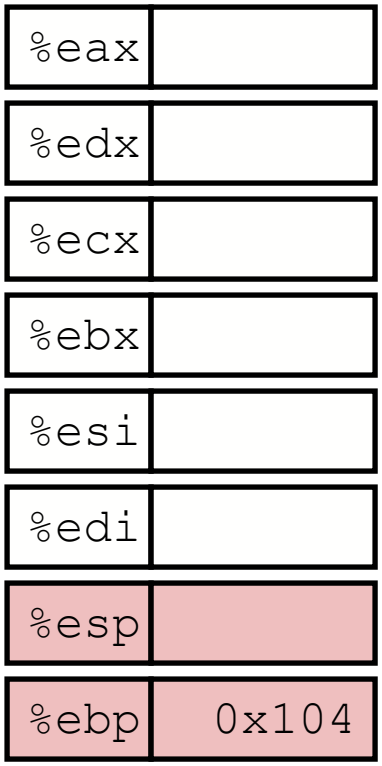


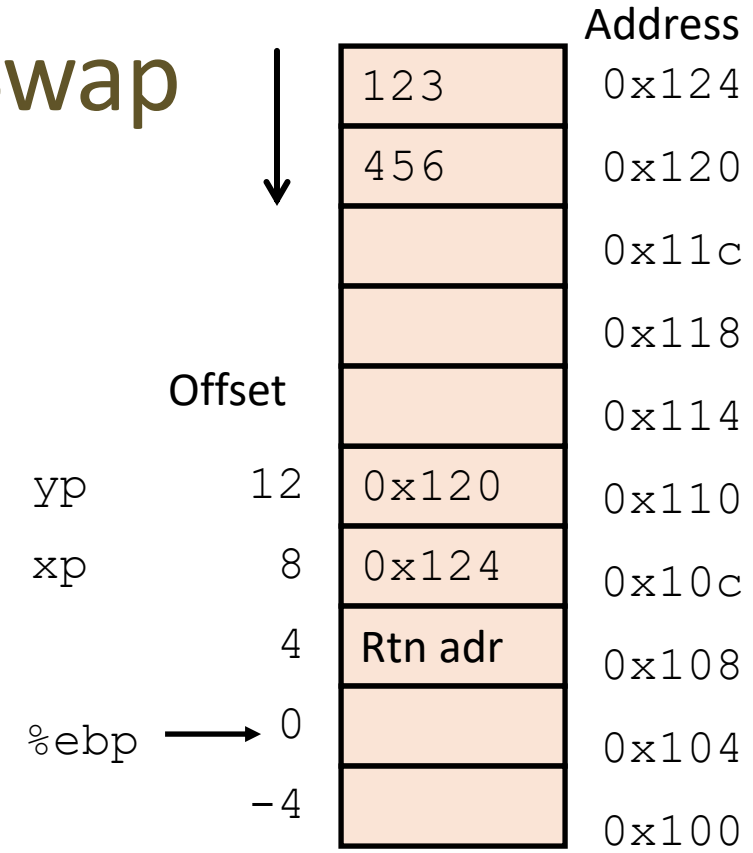
Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

AT&T format (src, dst)

```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

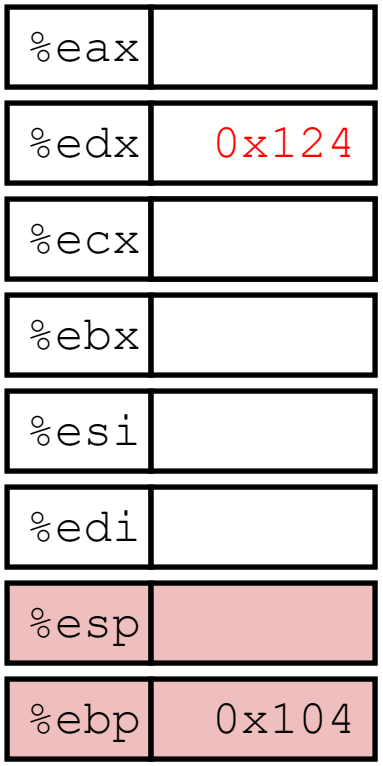
Understanding Swap

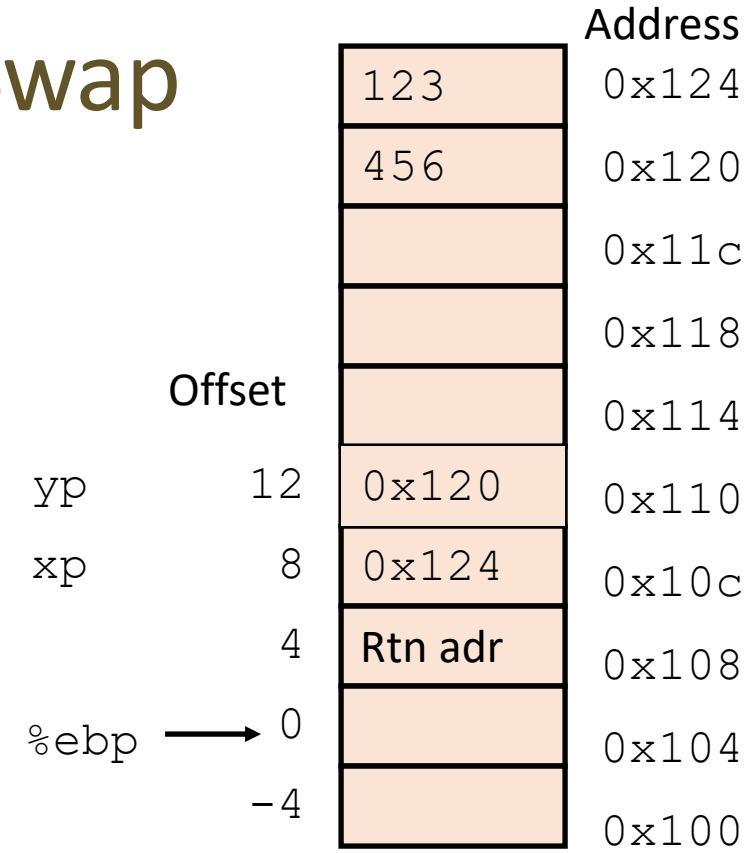





```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

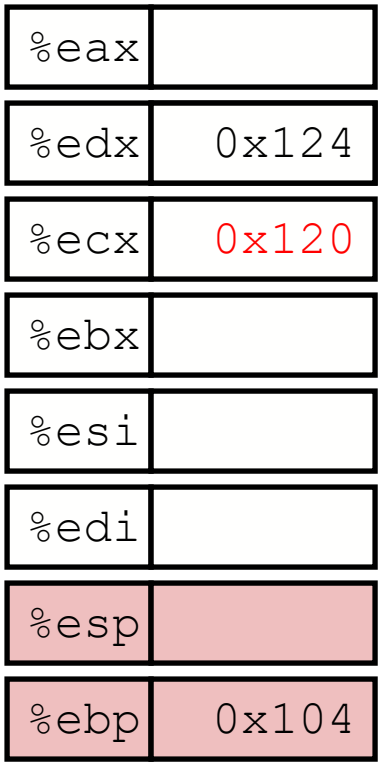
Understanding Swap

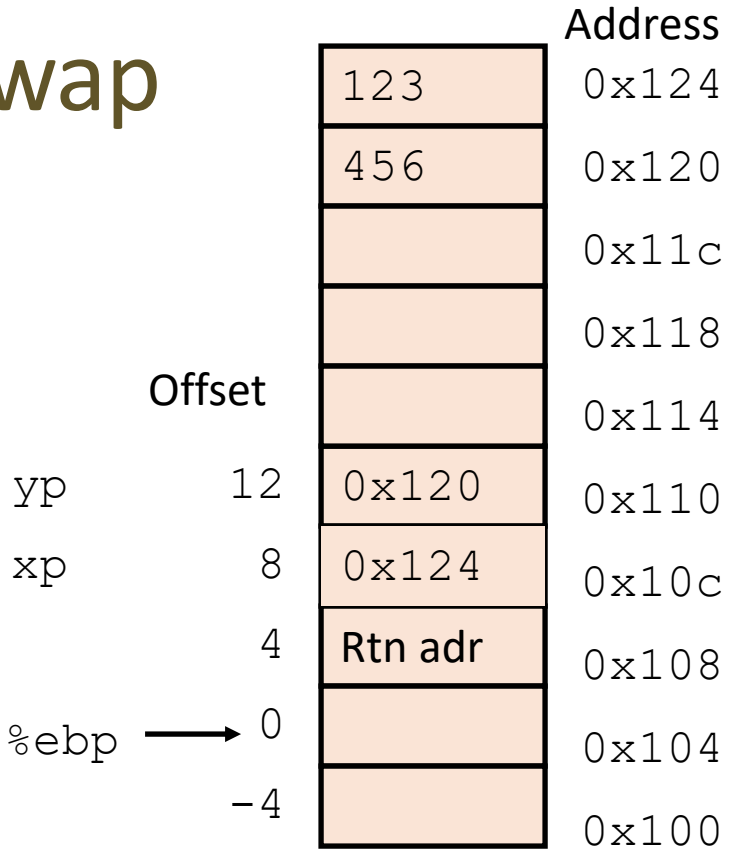





```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

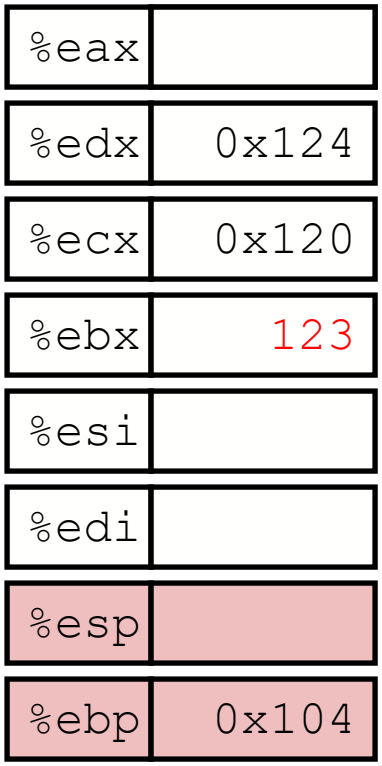
Understanding Swap

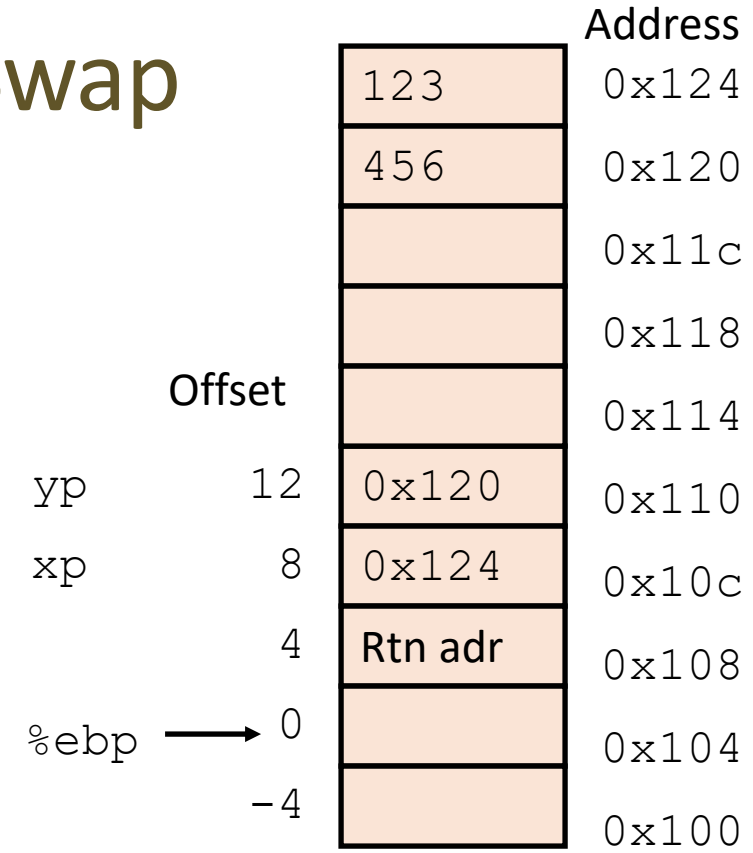





```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

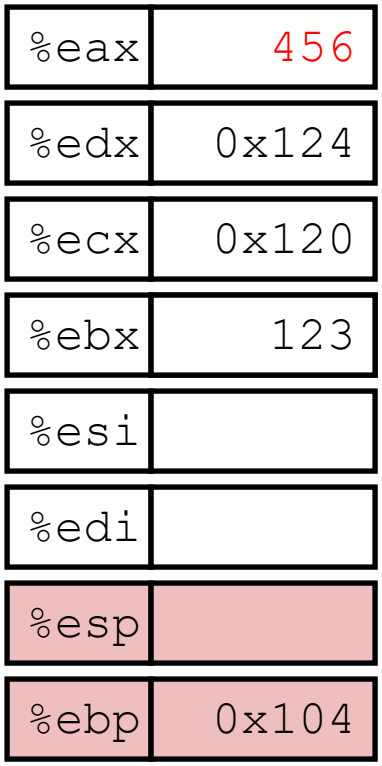
Understanding Swap

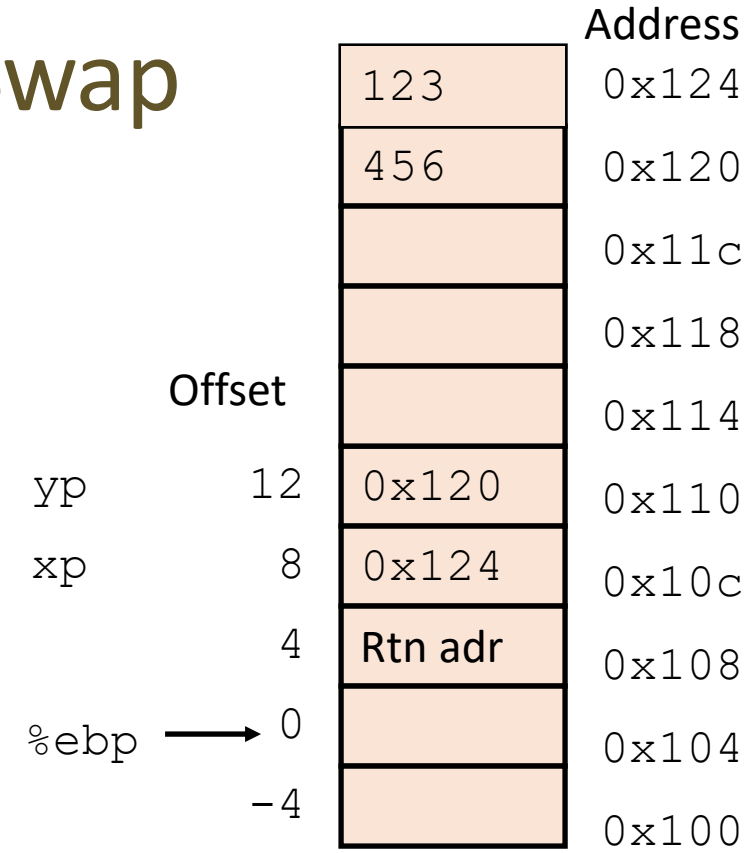





```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

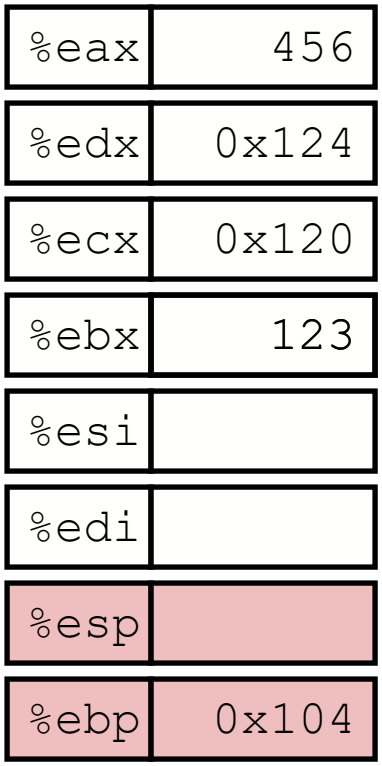
Understanding Swap

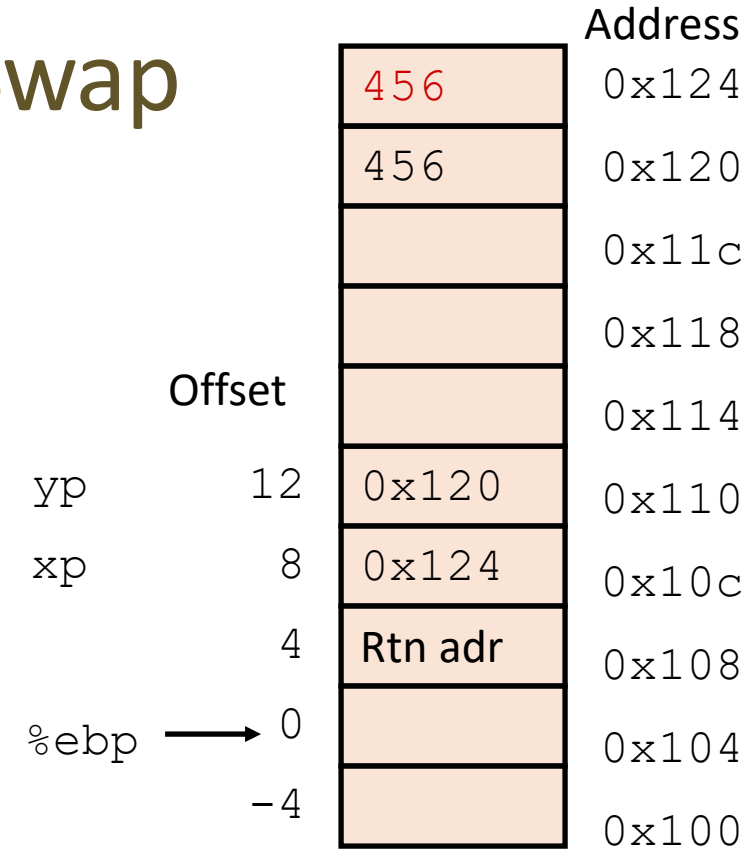





```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

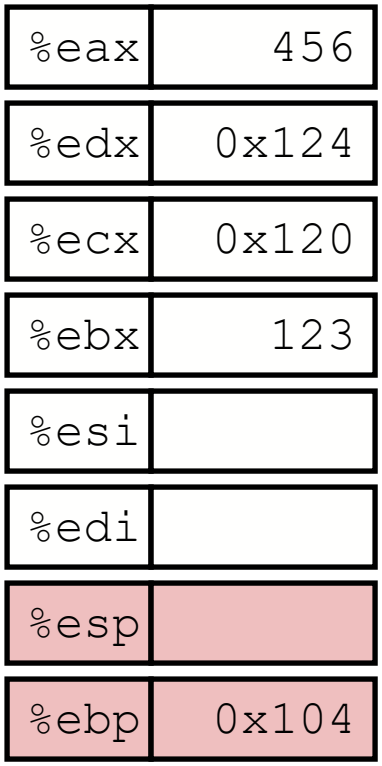
Understanding Swap

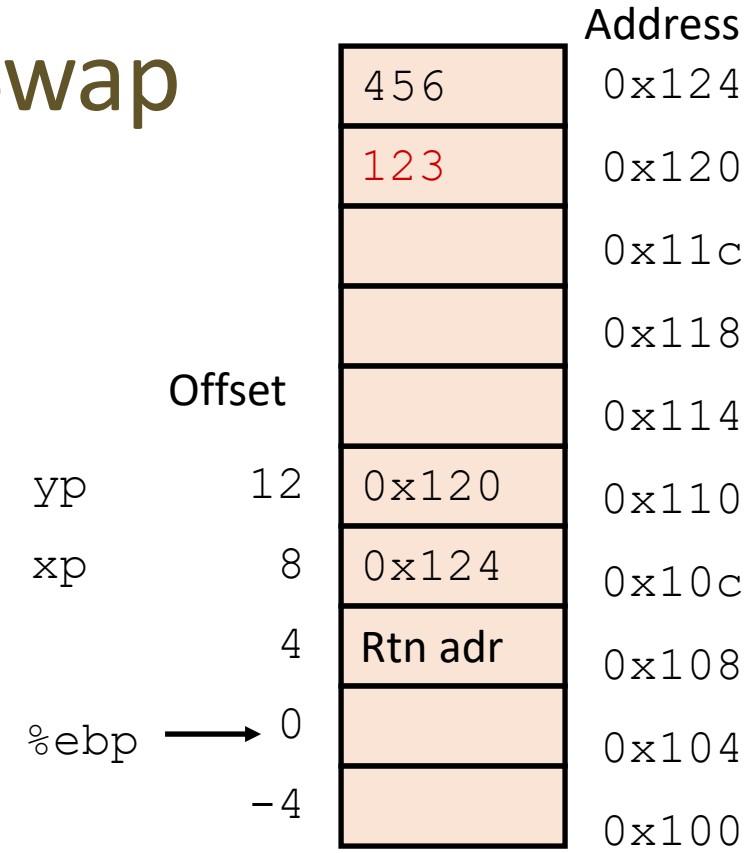





```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

Understanding Swap






```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```