

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

Binary

Code Segment
(.text)

Data Segment
(.data)

...

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
- ...

Binary

Code Segment
(.text)

Data Segment
(.data)

...

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

add r/m encoding
 of register

offset

- 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

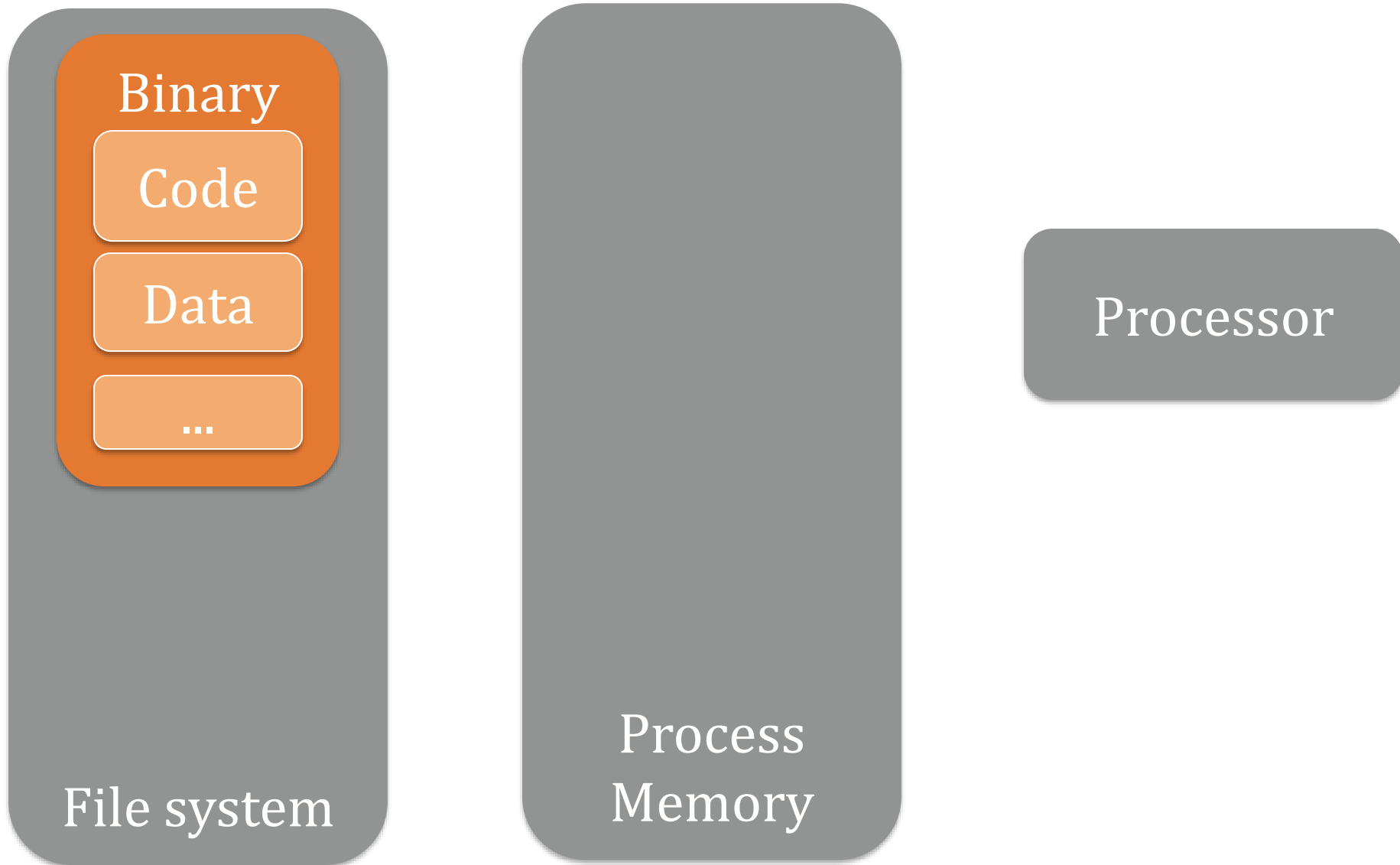
- Stored at address **0x80483ca**

Agenda

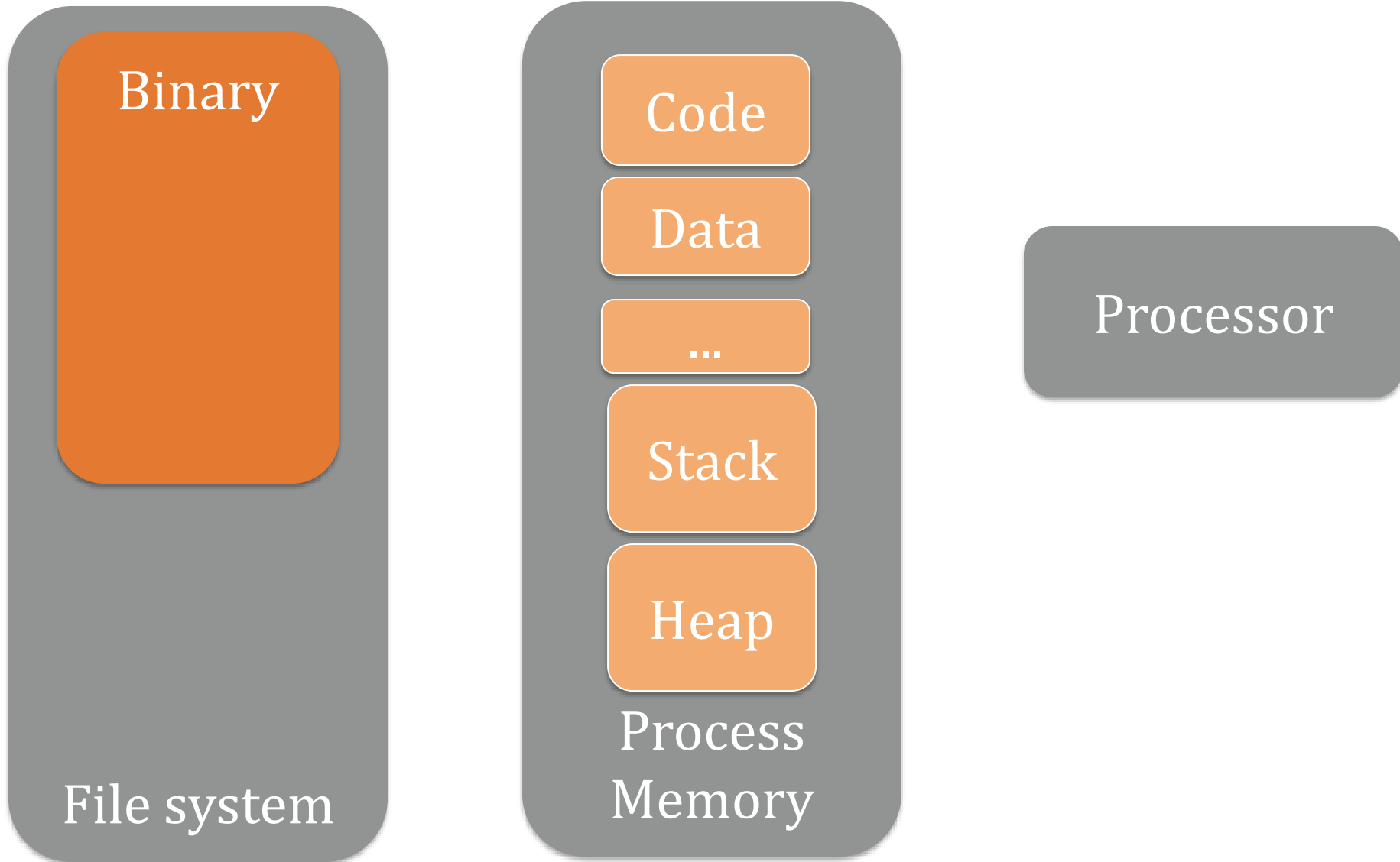
- Compilation Workflow
- x86 Execution Model
 - Basic Execution
 - Memory Operation
 - Control Flow
 - Memory Organization



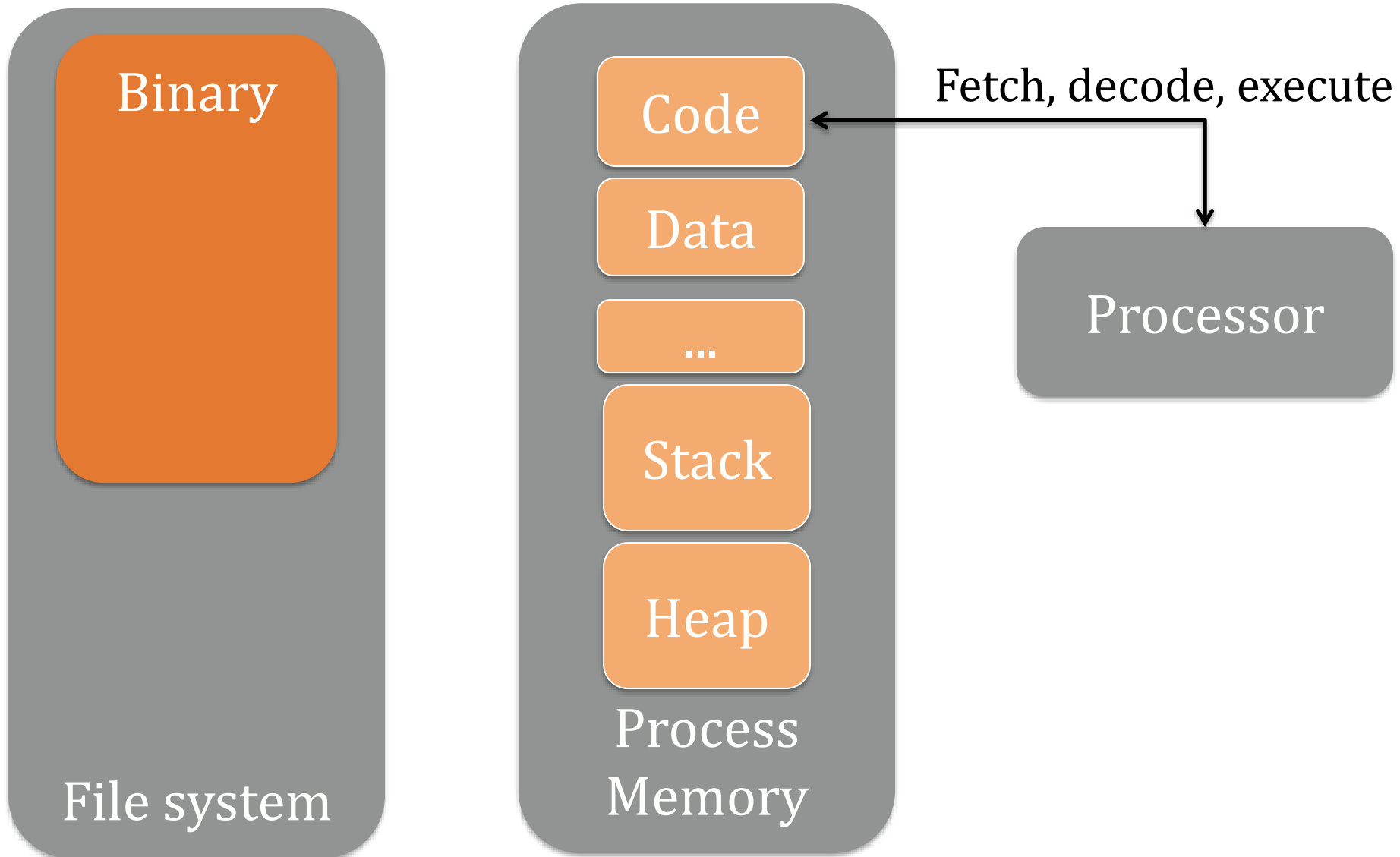
Basic Execution



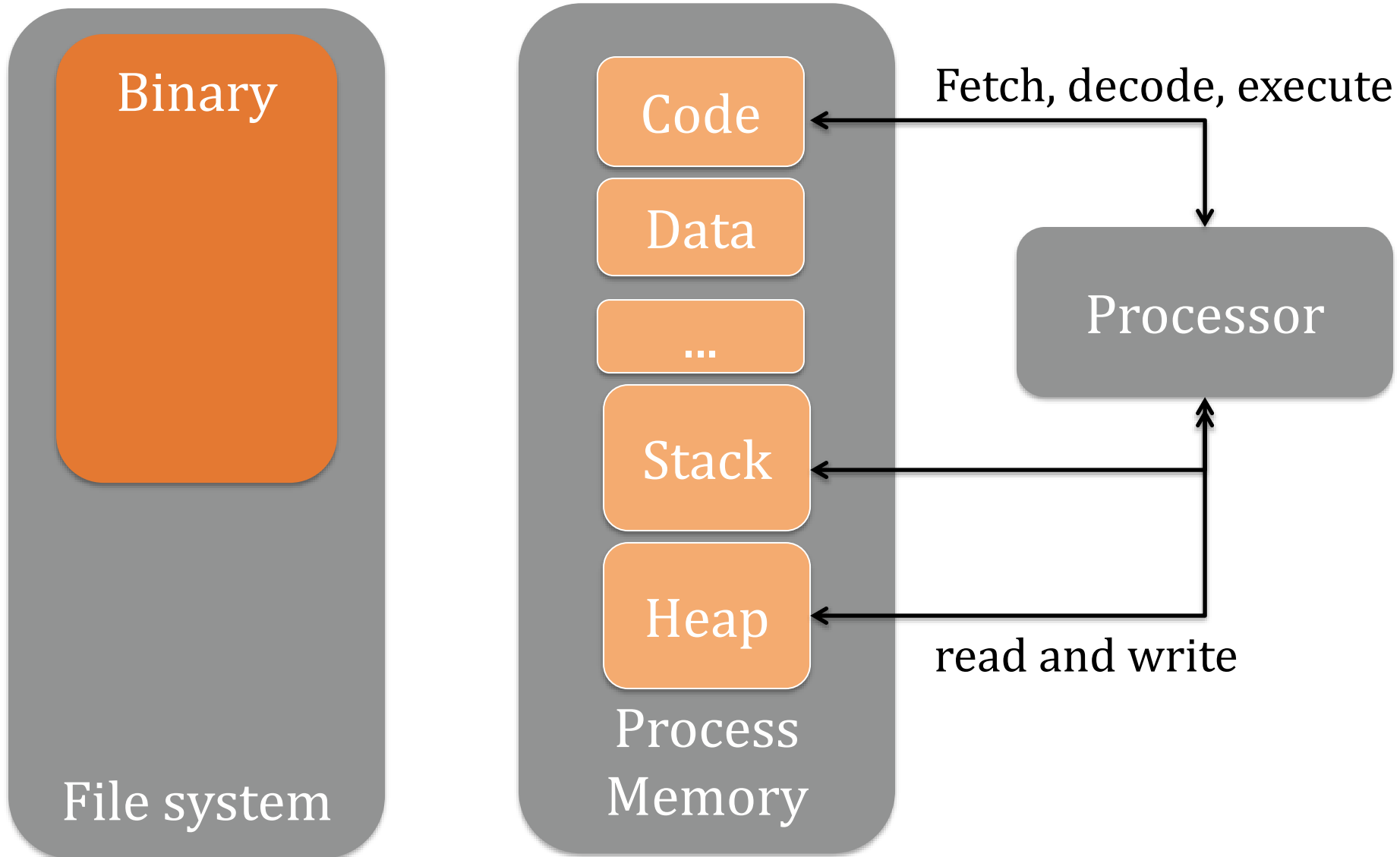
Basic Execution



Basic Execution



Basic Execution



x86 Processor

EIP

EFLAGS

EAX

EDX

ECX

EBX

ESP

EBP

ESI

EDI

x86 Processor

EAX

EDX

ECX

EBX

ESP

EBP

ESI

EDI

EIP

EFLAGS

Address of
next
instruction

x86 Processor

EAX

EBX

ECX

EDX

ESP

EBP

ESI

EDI

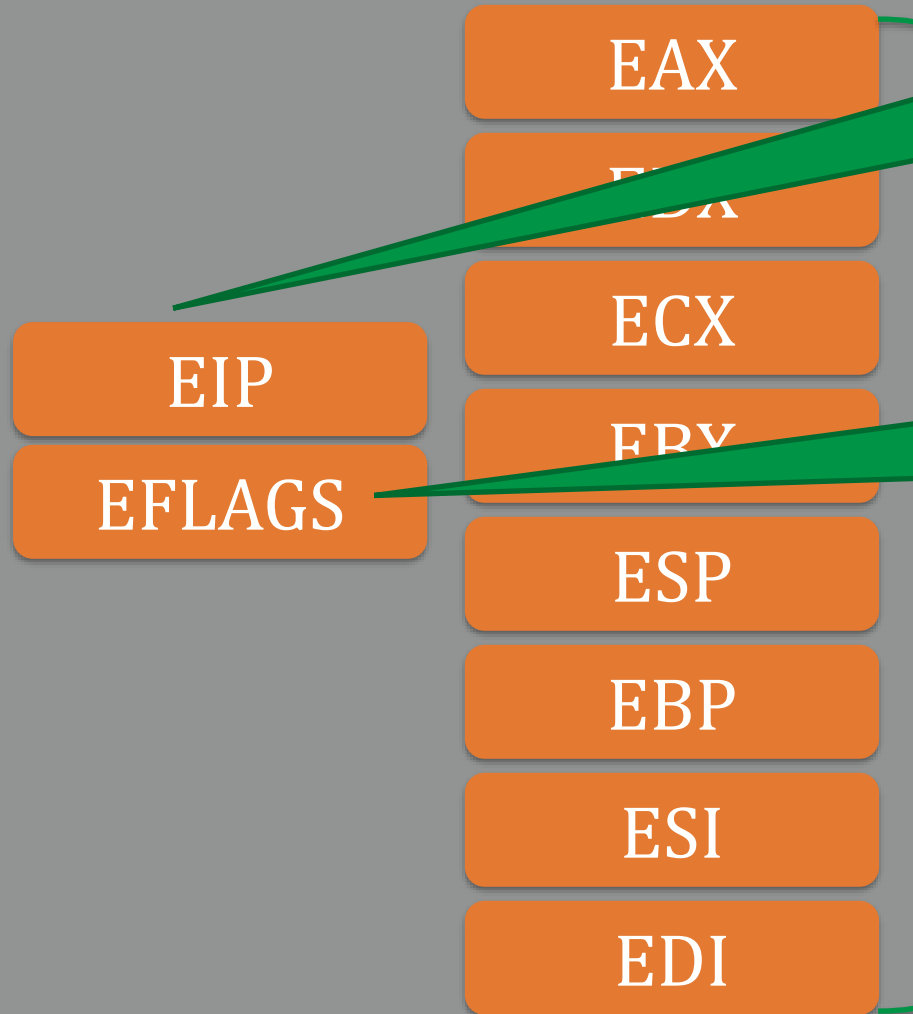
EIP

EFLAGS

Address of
next
instruction

Condition
codes

x86 Processor



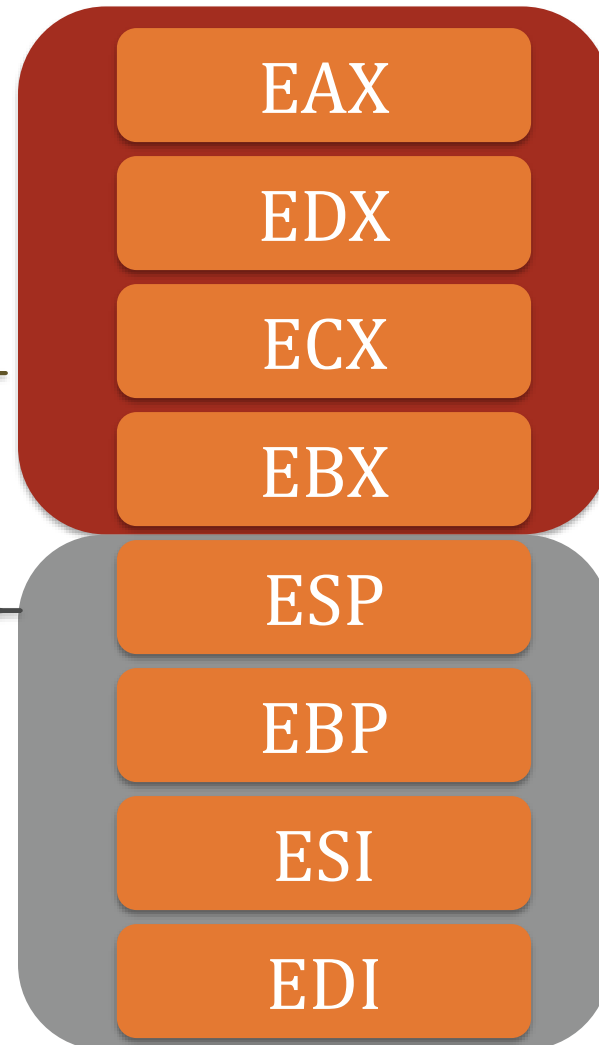
Address of
next
instruction

Condition
codes

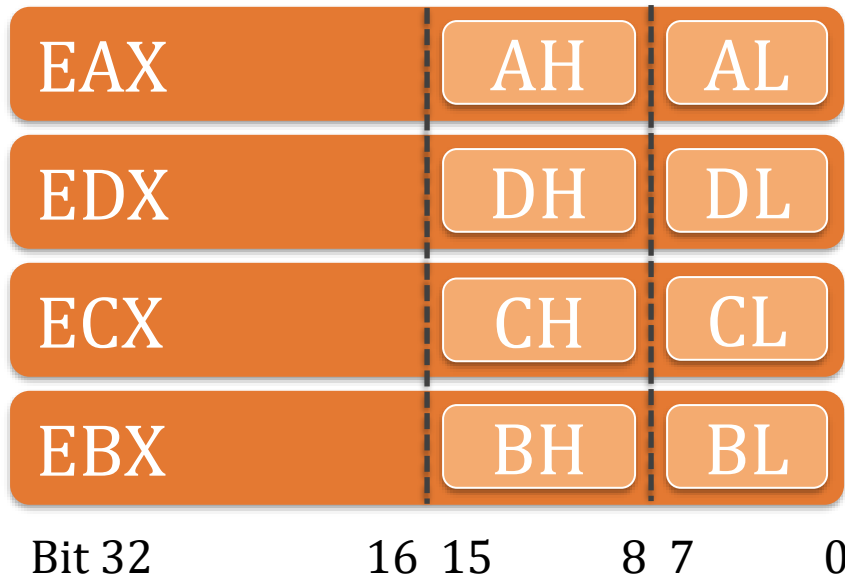
General
Purpose

Registers have up to 4 addressing modes

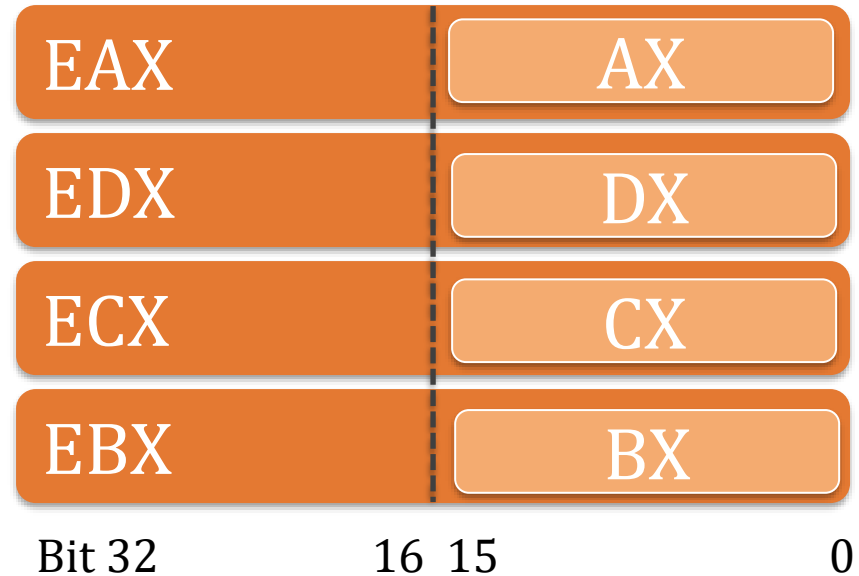
1. Lower 8 bits
2. Mid 8 bits
3. Lower 16 bits
4. Full register



EAX, EDX, ECX, and EBX

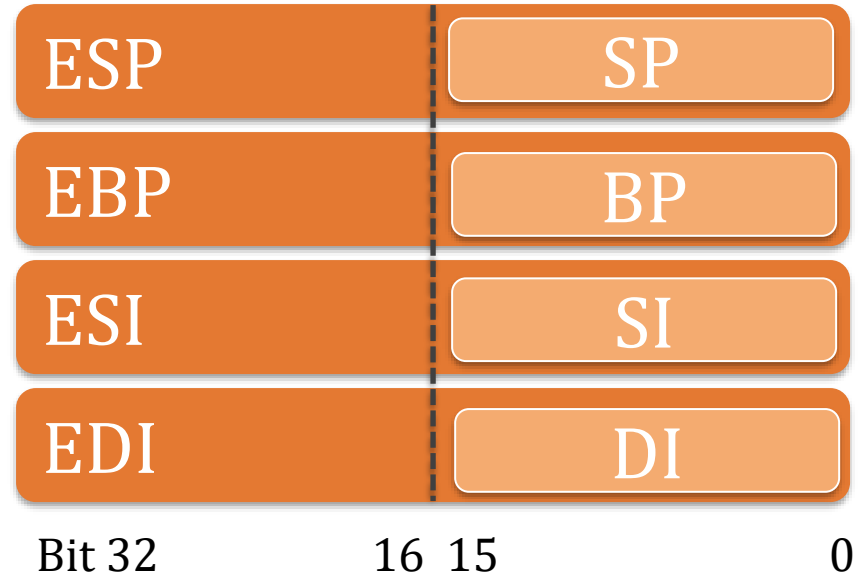
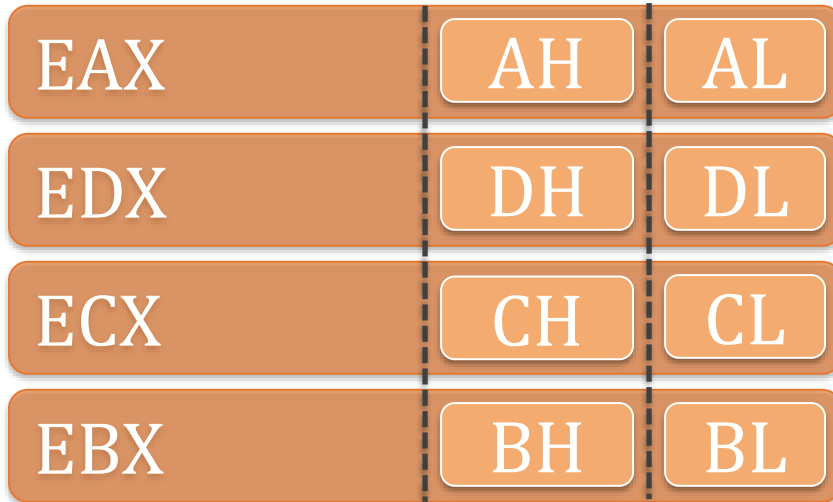


- 32 bit registers
(**three** letters)
- Lower bits (bits 0-7)
(two letters with **L** suffix)
- Mid-bits (bits 8-15)
(two letters with **H** suffix)



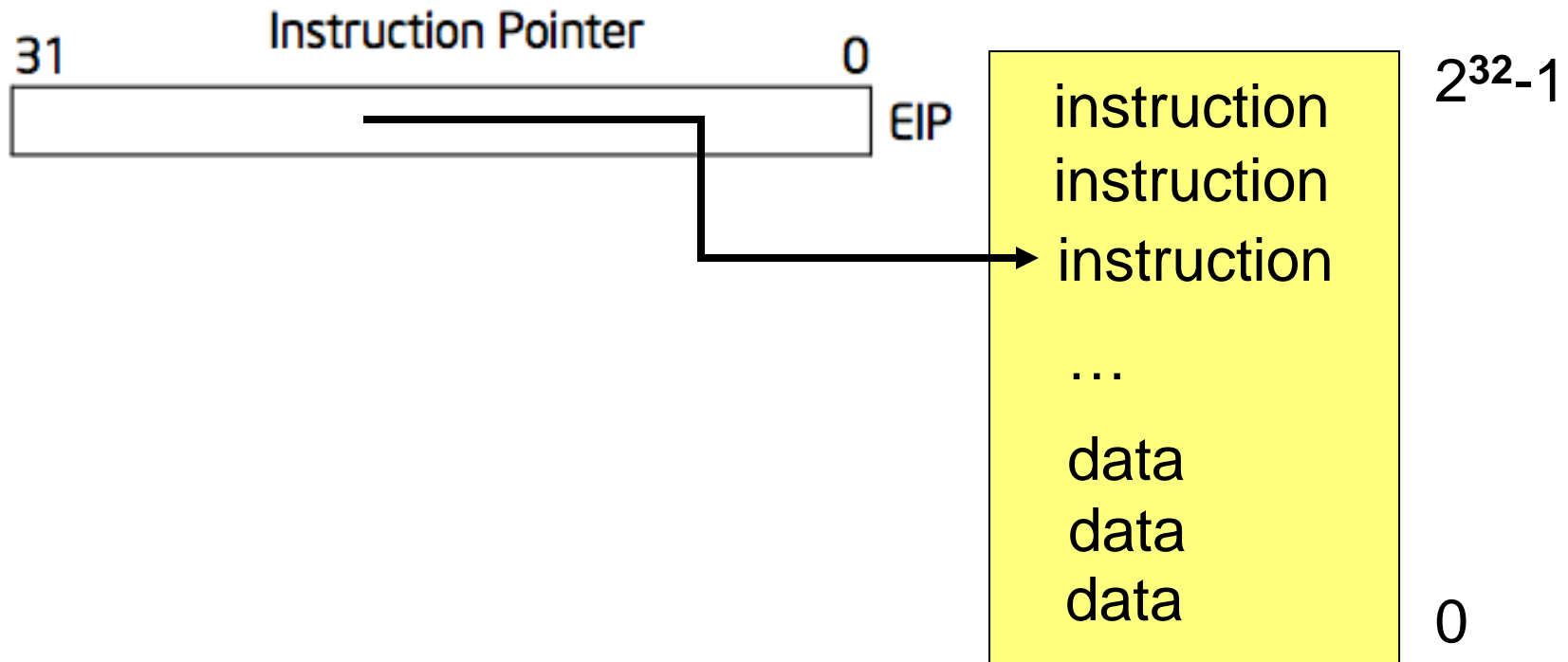
- Lower 16 bits (bits 0-15)
(2 letters with **X** suffix)

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

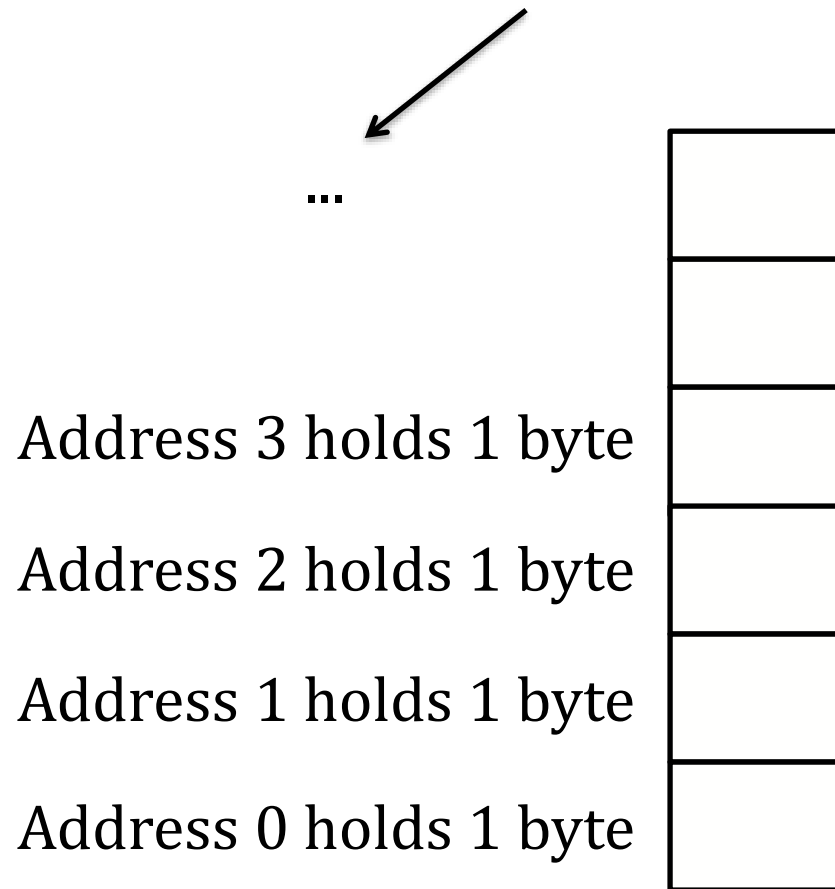
- Compilation Workflow
- x86 Execution Model
 - Basic Execution
 - Memory Operation
 - Control Flow
 - Memory Organization



x86: Byte Addressable



x86: **Byte** Addressable



x86: Byte Addressable

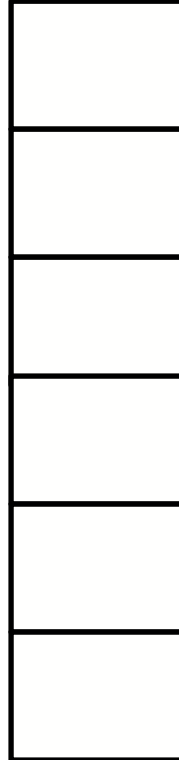
It's convention: lower address at the bottom

Address 3 holds 1 byte

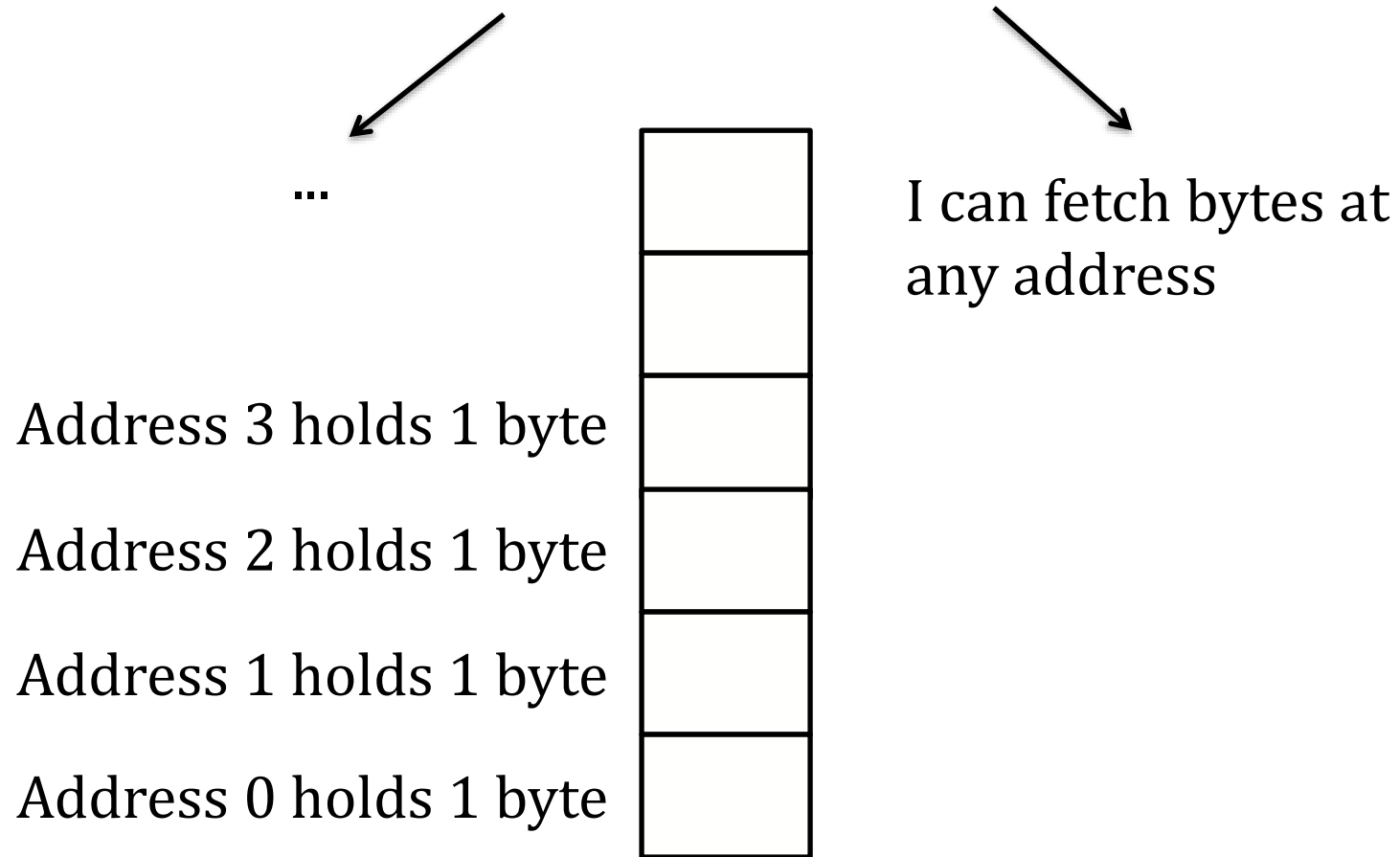
Address 2 holds 1 byte

Address 1 holds 1 byte

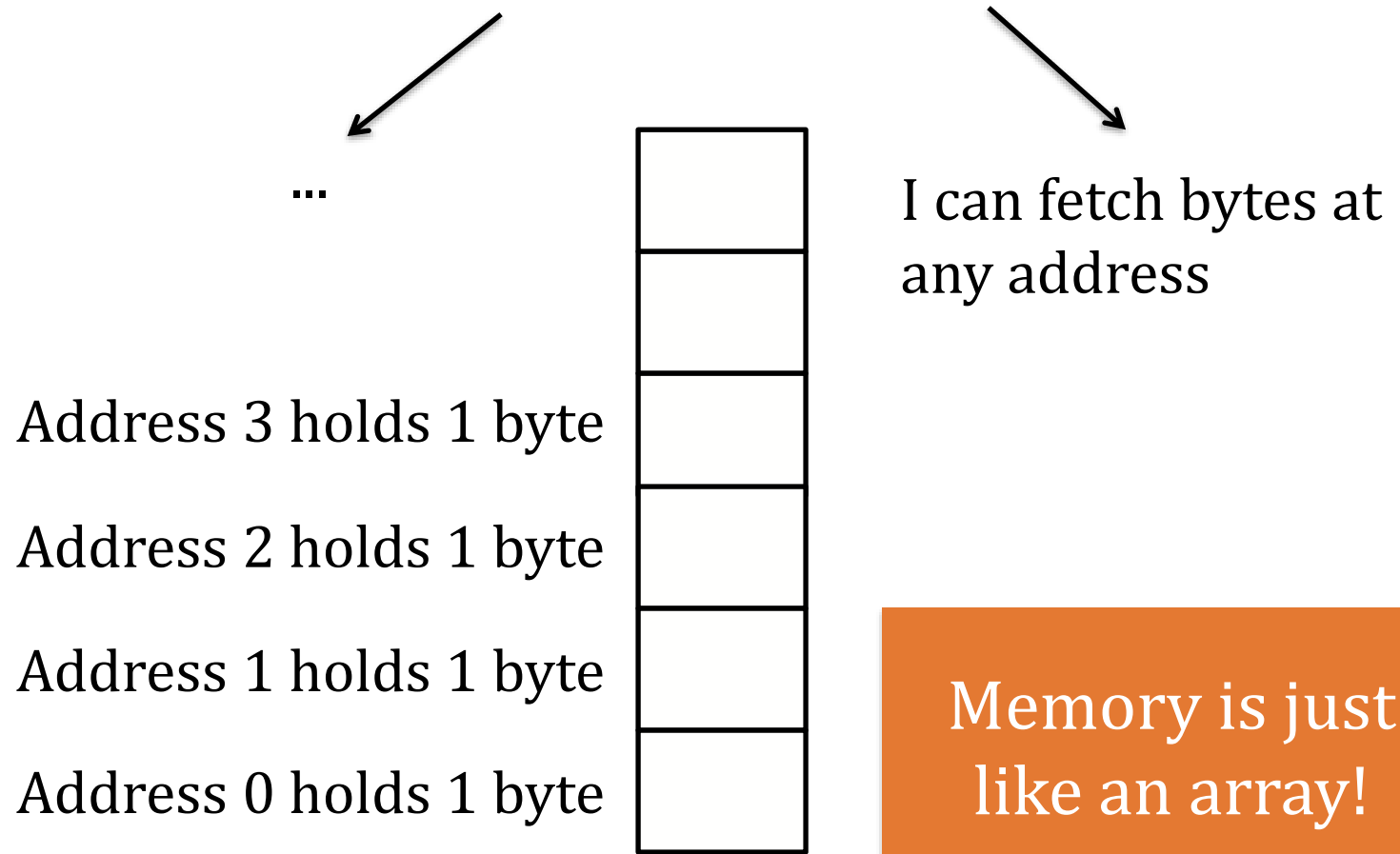
Address 0 holds 1 byte



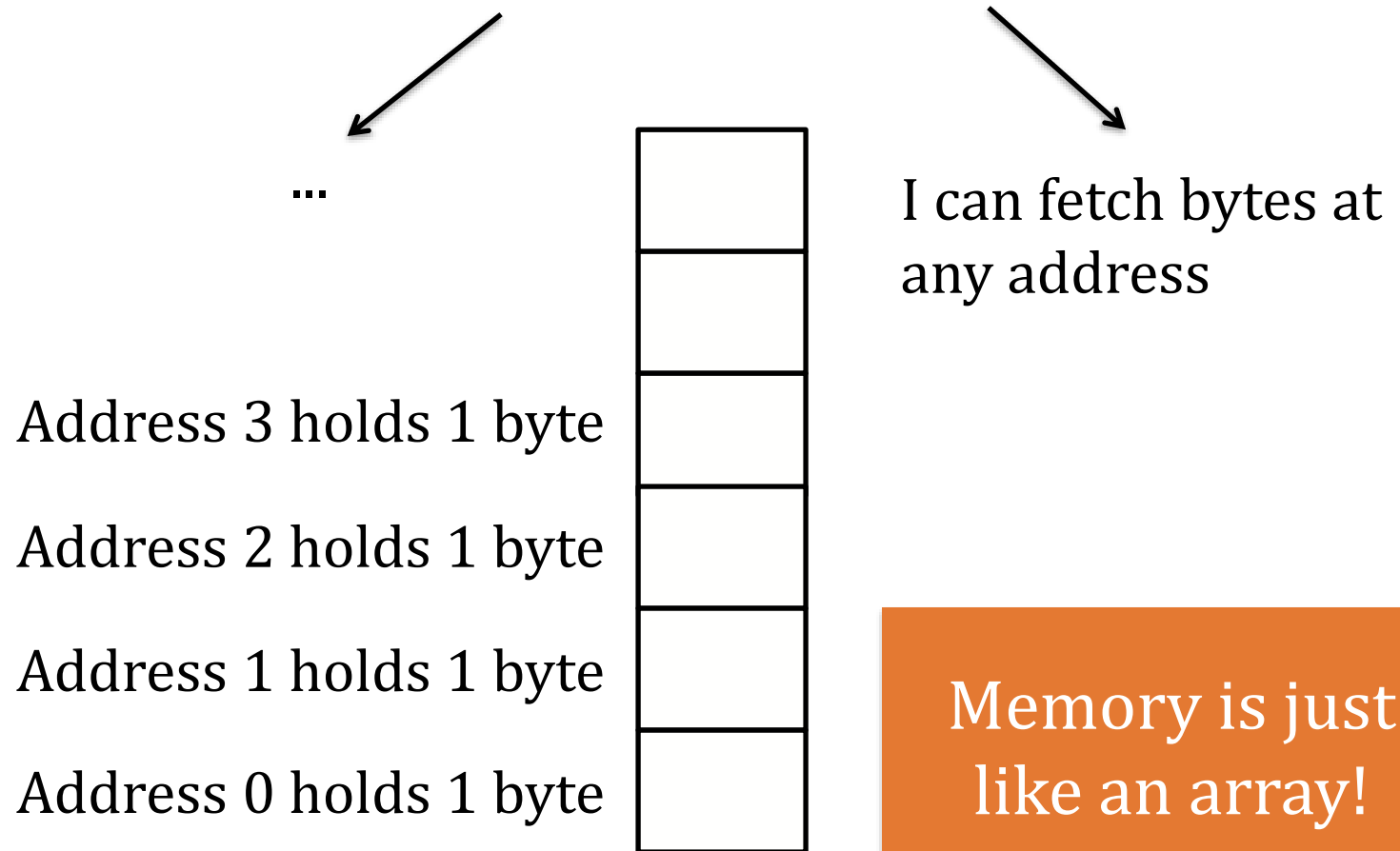
x86: **Byte** Addressable



x86: **Byte** Addressable



x86: **Byte** Addressable

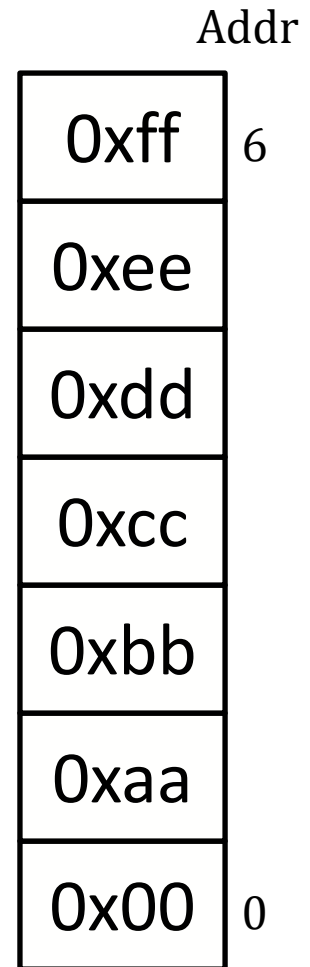


Alternative: **Word addressable**

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

	Addr
0xff	6
0xee	
0xdd	
0xcc	
0xbb	
0xaa	
0x00	0

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

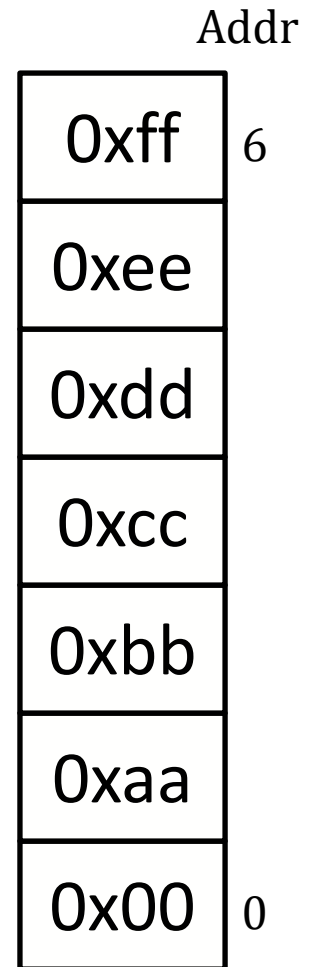
	Addr
0xff	6
0xee	
0xdd	
0xcc	
0xbb	
0xaa	
0x00	0

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?

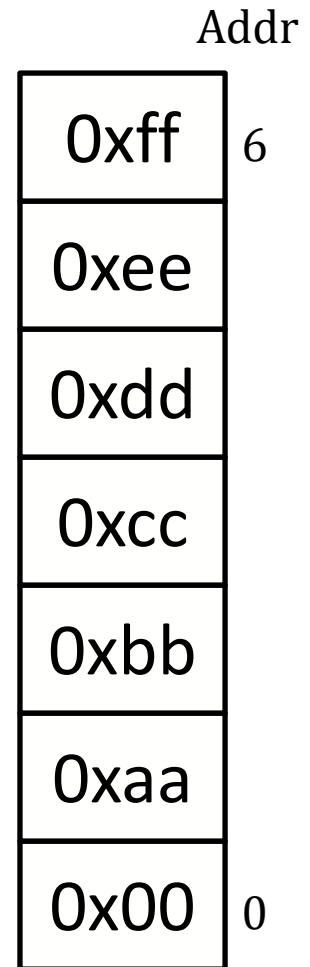
Addr	
6	0xff
	0xee
	0xdd
	0xcc
	0xbb
	0xaa
0	0x00

Endianness

- *Endianness*: 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

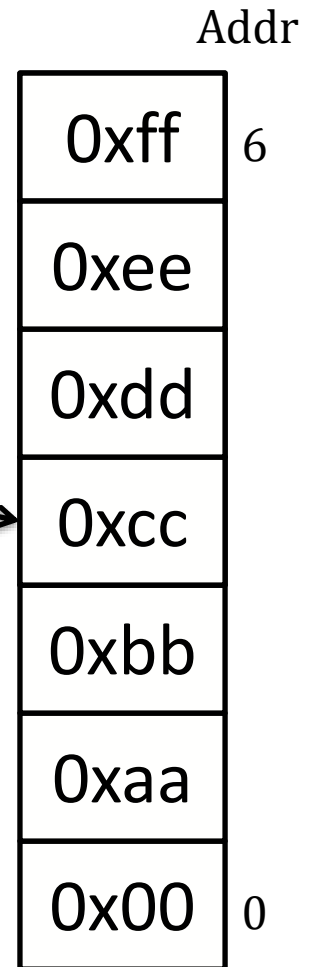


mov edx, [eax]

EDX

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

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.

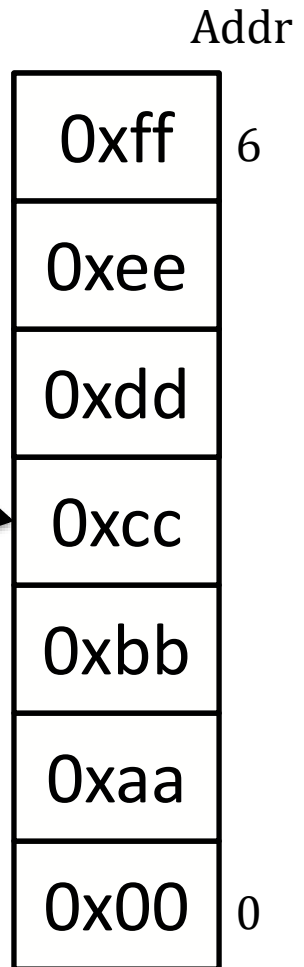
mov edx, [eax]

EDX

0xcc

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 edx, [eax]

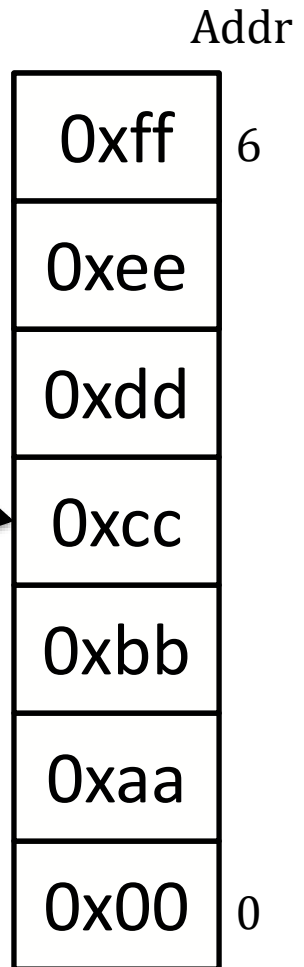
EDX

0xdd

0xcc

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 edx, [eax]

EDX

0xee

0xdd

0xcc

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0

0xff

6

0xee

0xdd

0xcc

0xbb

0xaa

0x00

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 edx, [eax]

EDX

0xff

0xee

0xdd

0xcc

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0

0xff

6

0xee

0xdd

0xcc

0xbb

0xaa

0x00

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 edx, [eax]`

EDX

0xff

0xee

0xdd

0xcc

Register	Value
eax	
edx	
ebx	

EDX = 0xffeeddccc!

Bit 0

Addr

0xff

6

0xee

0xdd

0xcc

0xbb

0xaa

0x00

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

EBX

00

00

00

05

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0

Addr

0xff

6

0xee

0xdd

0xcc

0xbb

0xaa

0x00

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

EBX

00

00

00

05

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0

Addr

0xff

6

0xee

0xdd

05

0xbb

0xaa

0x00

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

EBX

00

00

00

05

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0

Addr

0xff

6

0xee

00

05

0xbb

0xaa

0x00

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

EBX

00

00

00

05

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0

Addr

0xff

6

00

00

05

0xbb

0xaa

0x00

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

EBX

00

00

00

05

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0

Addr

00

6

00

00

05

0xbb

0xaa

0x00

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)
```

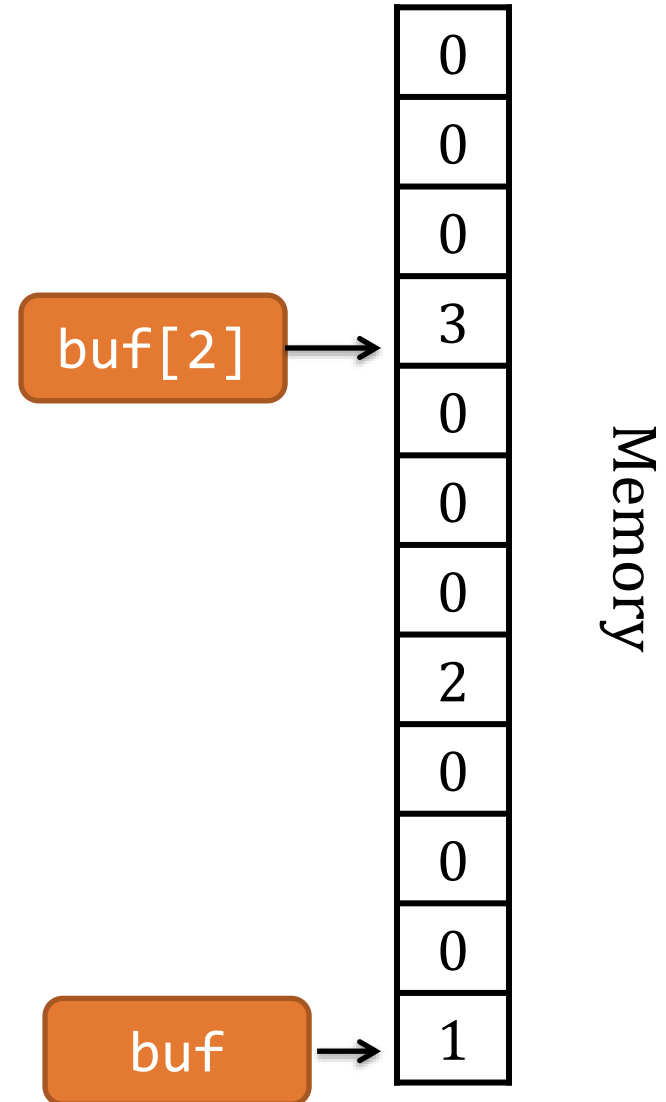
Can be done in a single x86 instruction!

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;
```

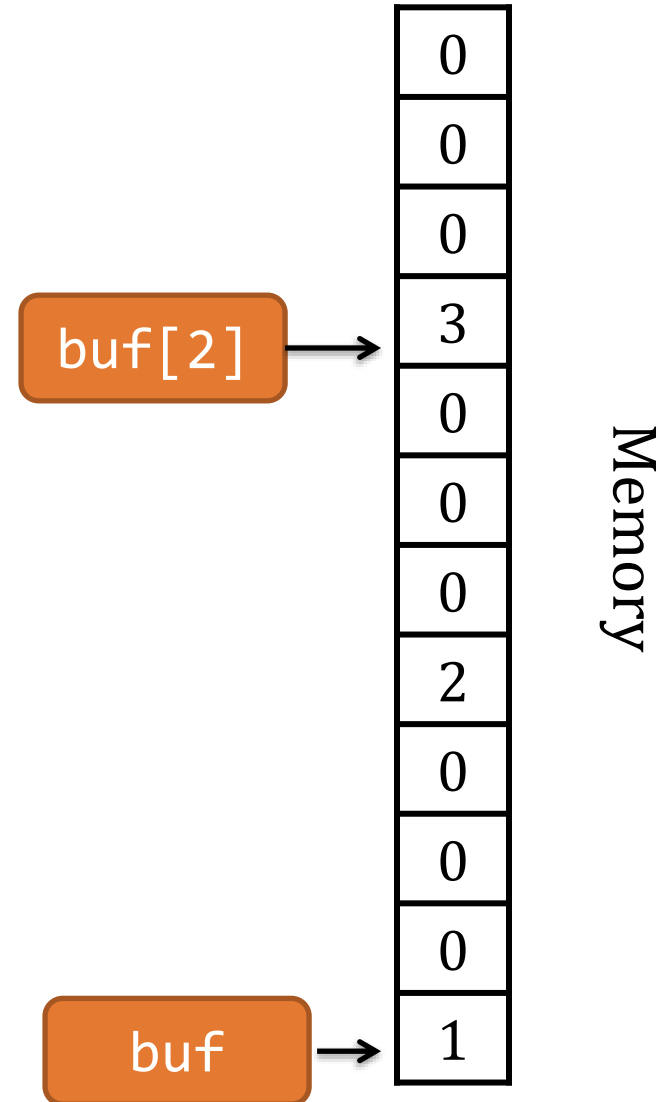
```
w = buf[2];  
x = *(buf + 2);
```

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));
```



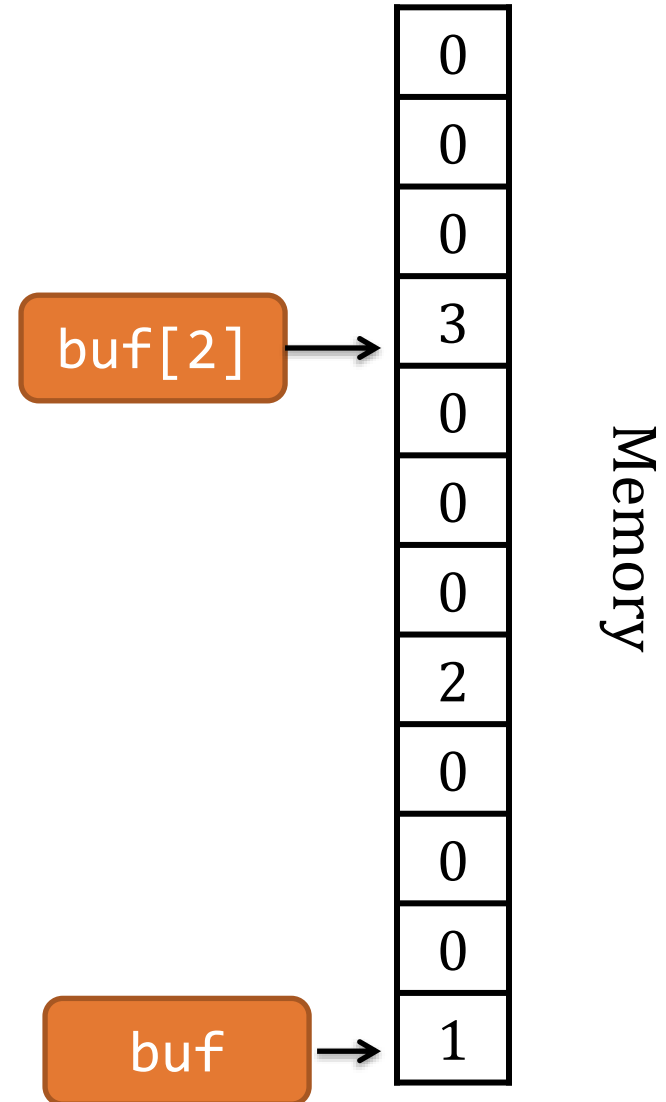
Motivation: Addressing Buffers

```
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

$(\text{addr_t}) (\text{ptr} + 8) = (\text{uint32_t} *) \text{buf} + 2$



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 $\text{imm} + r_1$

Motivation: Addressing Buffers

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

Say at $\text{imm} + r_1$

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 $\text{imm} + r_1$

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 $\text{imm} + r_1$

Constant
scaling factor
 s , typically
 $1, 2, 4, \text{ or } 8$

Say in Register
 r_2

$$\text{imm} + r_1 + s * r_2$$

AT&T: $\text{imm}(r_1, r_2, s)$

Intel: $r_1 + r_2 * s + \text{imm}$

AT&T Addressing Modes for Common Codes

Form	Meaning on memory M
$\text{imm}(r)$	$M[r + \text{imm}]$
$\text{imm}(r_1, r_2)$	$M[r_1 + r_2 + \text{imm}]$
$\text{imm}(r_1, r_2, s)$	$M[r_1 + r_2 * s + \text{imm}]$
imm	$M[\text{imm}]$

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

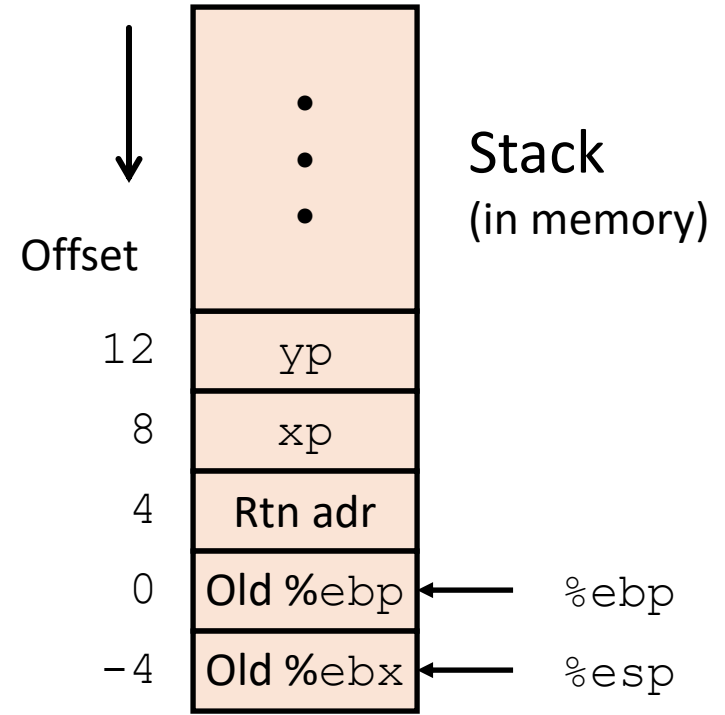
```
lea eax, 0xdeadbeef (I)
```

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

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address
<div> <div>↓</div> <div>Offset</div> <div>yp 12</div> <div>xp 8</div> <div>4</div> <div>%ebp → 0</div> <div>-4</div> </div>	123	0x124
	456	0x120
		0x11c
		0x118
		0x114
	0x120	0x110
	0x124	0x10c
	Rtn adr	0x108
		0x104
		0x100

```

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

%eax	
%edx	0x124
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	
	-4	
		0x104
		0x100

```

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

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x110
xp	8	0x10c
	4	Rtn adr
%ebp	0	0x108
	-4	0x104
		0x100

```

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

%eax	
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x110
xp	8	0x10c
	4	Rtn adr
%ebp	0	0x108
	-4	0x104
		0x100

```

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

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

The diagram illustrates a stack layout. On the left, a vertical stack of memory cells is shown, each with a value and an address. The addresses are 0x124, 0x120, 0x11c, 0x118, 0x114, 0x110, 0x10c, 0x108, 0x104, and 0x100. To the left of the stack, the offset from the current frame base is indicated: 12 for 0x110, 8 for 0x10c, 4 for 0x108, 0 for 0x104, and -4 for 0x100. The register %ebp is shown with an arrow pointing to the 0x104 address. The word 'wap' is written in a large, stylized font at the top left.

Register / Label	Offset	Value	Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
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

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x110
xp	8	0x10c
	4	Rtn adr
%ebp	0	0x108
	-4	0x104
		0x100

```

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

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		456
		0x124
		123
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
0x120		0x110
xp	8	0x124
0x124		0x10c
	4	Rtn adr
		0x108
%ebp	0	
		0x104
	-4	
		0x100

```

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

```