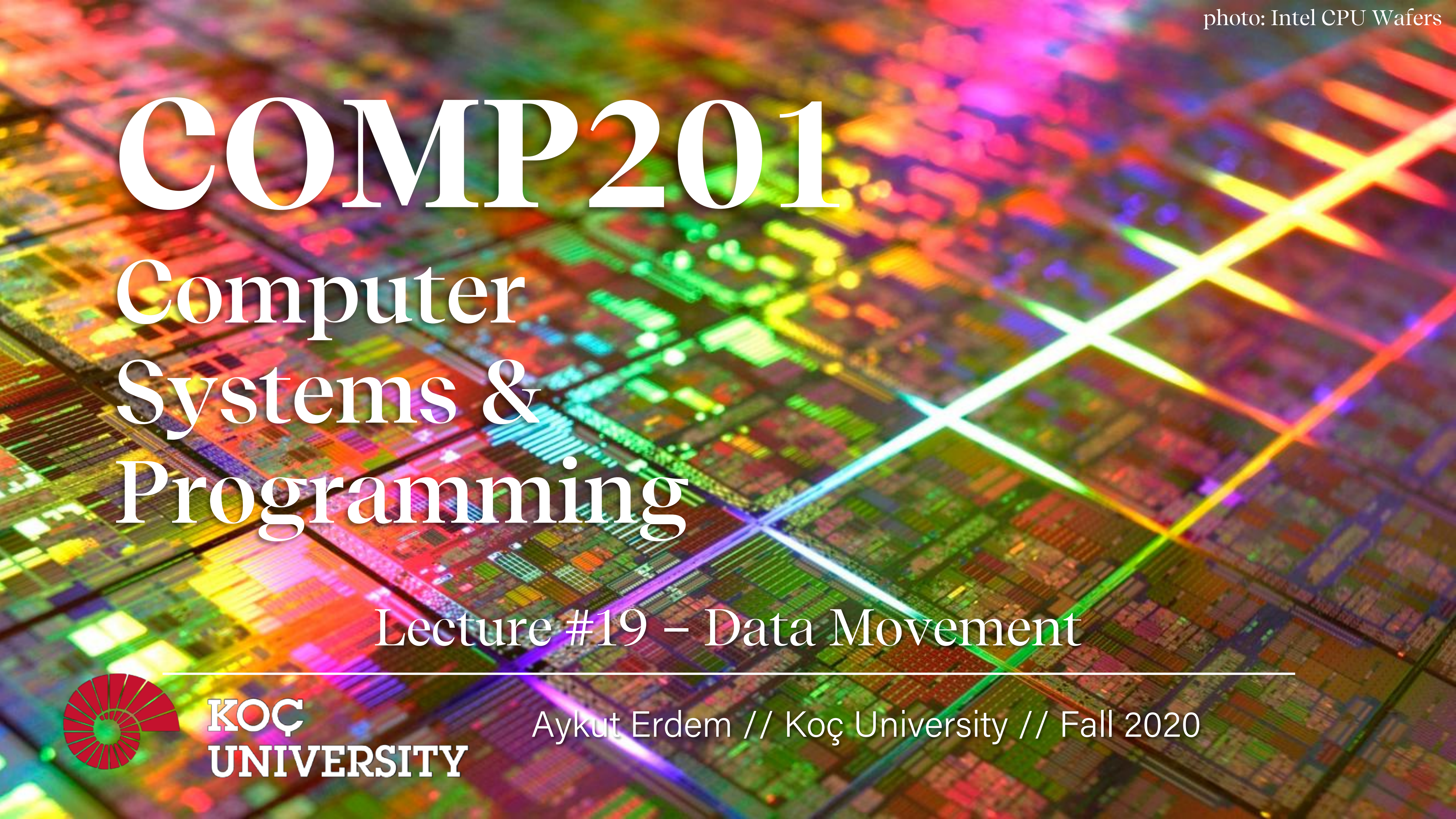

photo: Intel CPU Wafers

COMP201

Computer Systems & Programming

Lecture #19 – Data Movement

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

COMP201 Topic 6: How does a computer interpret and execute C programs?

Learning Assembly

Moving data around → Arithmetic and logical operations → Control flow → Function calls

This Lecture | Lecture 20 | Lecture 21-23 | Lecture 24-26

Lecture Plan

- **Recap:** `mov` so far
- Data and Register Sizes
- The `lea` Instruction

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Lecture Plan

- **Recap:** `mov` so far
- Data and Register Sizes
- The **lea** Instruction

`mov`

The **`mov`** instruction <u>copies</u> bytes from one place to another;
it is similar to the assignment operator (=) in C.

```
mov        src,dst
```

The **src** and **dst** can each be one of:

- Immediate (constant value, like a number) (*only* ***src***) **`$0x104`**
- Register **`%rbx`**
- Memory Location (*at most one of* ***src, dst***) Direct address **`0x6005c0`**

Operand Forms: Immediate

```
mov        $0x104,_____
```

Copy the value 0x104 into some destination.

Operand Forms: Registers

Copy the value in register %rbx into some destination.

```
mov     %rbx,____
```

```
mov     ____,%rbx
```

Copy the value from some source into register %rbx.

Operand Forms: Absolute Addresses

Copy the value at address 0x104 into some destination.

```
mov    0x104,_____
```

```
mov    _____,0x104
```

Copy the value from some source into the memory at address 0x104.

Operand Forms: Indirect

Copy the value at the address stored in register %rbx into some destination.

```
mov        (%rbx),_____
```

```
mov        _____,(%rbx)
```

Copy the value from some source into the memory at the address stored in register %rbx.

Operand Forms: Base + Displacement

*Copy the value at the address (**0x10 plus** what is stored in register %rax) into some destination.*

```
mov     0x10(%rax),__________
```

```
mov     __________,0x10(%rax)
```

*Copy the value from some source into the memory at the address (**0x10 plus** what is stored in register %rax).*

Operand Forms: Indexed

Copy the value at the address which is (the sum of the values in registers %rax and %rdx) into some destination.

`mov (%rax,%rdx),__________`

`mov __________,(%rax,%rdx)`

Copy the value from some source into the memory at the address which is (the sum of the values in registers %rax and %rdx).

Operand Forms: Indexed

*Copy the value at the address which is (the sum of **0x10 plus** the values in registers %rax and %rdx) into some destination.*

```
mov     0x10(%rax,%rdx),______
```

```
mov     ______,0x10(%rax,%rdx)
```

*Copy the value from some source into the memory at the address which is (the sum of **0x10 plus** the values in registers %rax and %rdx).*

Practice #2: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume

the value *0x11* is stored at address *0x10C*,
the value *0xAB* is stored at address *0x104*,
0x100 is stored in register %rax and *0x3* is stored in %rdx.

1. mov	**`$0x42,(%rax)`**	Move 0x42 to memory address 0x100
2. mov	**`4(%rax),%rcx`**	Move 0xAB into %rcx
3. mov	**`9(%rax,%rdx),%rcx`**	Move 0x11 into %rcx

`Imm(`r_b`, `r_i`)` is equivalent to address **`Imm + R[`r_b`] + R[`r_i`]`**

Displacement: positive or negative constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**4 times** the value in register %rdx) into some destination.*

```
mov    (,%rdx,4),________

mov    ________,(,%rdx,4)
```

The *scaling factor* (e.g. 4 here) must be hardcoded to be either 1, 2, 4 or 8.

*Copy the value from some source into the memory at the address which is (**4 times** the value in register %rdx).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (4 times the value in register %rdx, **plus 0x4**), into some destination.*

```
mov     0x4(,%rdx,4),______
```

```
mov     _______,0x4(,%rdx,4)
```

*Copy the value from some source into the memory at the address which is (4 times the value in register %rdx, **plus 0x4**).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (__**the value in register %rax**__ plus 2 times the value in register %rdx) into some destination.*

```
mov     (%rax,%rdx,2),________
```

```
mov     _________,(%rax,%rdx,2)
```

*Copy the value from some source into the memory at the address which is (__**the value in register %rax**__ plus 2 times the value in register %rdx).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**0x4 plus** the value in register %rax plus 2 times the value in register %rdx) into some destination.*

```
mov    0x4(%rax,%rdx,2),_____
```

```
mov    ______,0x4(%rax,%rdx,2)
```

*Copy the value from some source into the memory at the address which is (**0x4 plus** the value in register %rax plus 2 times the value in register %rdx).*

Most General Operand Form

$$\mathtt{Imm}(r_b, r_i, s)$$

is equivalent to...

$$\mathtt{Imm} + R[r_b] + R[r_i]*s$$

Most General Operand Form

`Imm(`r_b`, `r_i`, s)` is equivalent to address **`Imm + R[`r_b`] + R[`r_i`]*s`**

Displacement: pos/neg constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Scale must be 1,2,4, or 8 (if missing, = 1)

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 from the book: "Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either. 1, 2, 4, or 8."

Practice #3: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume

the value *0x1* is stored in register %rcx,
the value *0x100* is stored in register %rax,
the value *0x3* is stored in register %rdx, and
the value *0x11* is stored at address *0x10C*.

1. `mov $0x42,0xfc(,%rcx,4)`

Move 0x42 to memory address 0x100

2. `mov (%rax,%rdx,4),%rbx`

Move 0x11 into %rbx

$\mathtt{Imm}(r_b,\ r_i,\ s)$ *is equivalent to address* $\mathtt{Imm} + R[r_b] + R[r_i]*s$

Displacement Base Index Scale

(1,2,4,8)

Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!

Extra Practice

Extra Practice

slido **Event code: 73165**

Fill in the blank to complete the code that generated the assembly below.

```
long arr[5];
...
long num = ____???___;
```

```
// %rdi stores arr, %rcx stores 3, and %rax stores num
mov (%rdi, %rcx, 8),%rax
```

Extra Practice

slido Event code: 73165

Fill in the blank to complete the code that generated the assembly below.

```
long arr[5];
...
long num = arr[3];
```

```
// %rdi stores arr, %rcx stores 3, and %rax stores num
mov (%rdi, %rcx, 8),%rax
```

Extra Practice

Event code: 73165

Fill in the blank to complete the code that generated the assembly below.

```
int x = ...
int *ptr = malloc(…);
___???___ = x;
```

```
// %ecx stores x, %rax stores ptr
mov %ecx,(%rax)
```

Extra Practice

slido Event code: 73165

Fill in the blank to complete the code that generated the assembly below.

```
int x = ...
int *ptr = malloc(…);
*ptr = x;
```

```
// %ecx stores x, %rax stores ptr
mov %ecx,(%rax)
```

Extra Practice

Event code: 73165

Fill in the blank to complete the code that generated the assembly below.

```
char str[5];
...
___???___ = 'c';
```

```
// %rcx stores str, %rdx stores 2
mov $0x63,(%rcx,%rdx,1)
```

Extra Practice

Event code: 73165

Fill in the blank to complete the code that generated the assembly below.

```
char str[5];
...
str[2] = 'c';
```

```
// %rcx stores str, %rdx stores 2
mov $0x63,(%rcx,%rdx,1)
```

Memory Location Syntax

Syntax	Meaning
`0x104`	Address `0x104` (no `$`)
`(%rax)`	What's in `%rax`
`4(%rax)`	What's in `%rax`, plus `4`
`(%rax, %rdx)`	Sum of what's in `%rax` and `%rdx`
`4(%rax, %rdx)`	Sum of values in `%rax` and `%rdx`, plus 4
`(, %rcx, 4)`	What's in `%rcx`, times 4 (multiplier can be 1, 2, 4, 8)
`(%rax, %rcx, 2)`	What's in `%rax`, plus 2 times what's in `%rcx`
`8(%rax, %rcx, 2)`	What's in `%rax`, plus 2 times what's in `%rcx`, plus 8

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$\mathrm{R}[r_a]$	Register
Memory	Imm	$\mathrm{M}[Imm]$	Absolute
Memory	(r_a)	$\mathrm{M}[\mathrm{R}[r_a]]$	Indirect
Memory	$Imm(r_b)$	$\mathrm{M}[Imm + \mathrm{R}[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$\mathrm{M}[\mathrm{R}[r_b] + \mathrm{R}[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$\mathrm{M}[Imm + \mathrm{R}[r_b] + \mathrm{R}[r_i]]$	Indexed
Memory	$(, r_i, s)$	$\mathrm{M}[\mathrm{R}[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$\mathrm{M}[\mathrm{R}[r_b] + \mathrm{R}[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_b] + \mathrm{R}[r_i] \cdot s]$	Scaled indexed

Figure 3.3 from the book: “Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either. 1, 2, 4, or 8.”

Lecture Plan

- **Recap:** `mov` so far
- Data and Register Sizes
- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

Data Sizes

Data sizes in assembly have slightly different terminology to get used to:

- A **byte** is 1 byte.
- A **word** is 2 bytes.
- A **double word** is 4 bytes.
- A **quad word** is 8 bytes.

Assembly instructions can have suffixes to refer to these sizes:

- `b` means **byte**
- `w` means **word**
- `l` means **double word**
- `q` means **quad word**

Register Sizes

Bit: 63	31	15	7 0
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil

Register Sizes

Bit: 63	31	15	7	0
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	
%r8	%r8d	%r8w	%r8b	
%r9	%r9d	%r9w	%r9b	
%r10	%r10d	%r10w	%r10b	
%r11	%r11d	%r11w	%r11b	

Register Sizes

Bit:	63	31	15	7	0
	%r12	%r12d	%r12w	%r12b	
	%r13	%r13d	%r13w	%r13b	
	%r14	%r14d	%r14w	%r14b	
	%r15	%r15d	%r15w	%r15b	

Register Responsibilities

Some registers take on special responsibilities during program execution.

- **`%rax`** stores the return value
- **`%rdi`** stores the first parameter to a function
- **`%rsi`** stores the second parameter to a function
- **`%rdx`** stores the third parameter to a function
- **`%rip`** stores the address of the next instruction to execute
- **`%rsp`** stores the address of the current top of the stack

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

`mov` Variants

- **`mov`** can take an optional suffix (b,w,l,q) that specifies the size of data to move: `movb, movw, movl, movq`
- **`mov`** only updates the specific register bytes or memory locations indicated.
  - **Exception: `movl`** writing to a register will also set high order 4 bytes to 0.

Practice: `mov` And Data Sizes

For each of the following `mov` instructions, determine the appropriate suffix based on the operands (e.g. `movb`, `movw`, `movl` or `movq`).

```
1. mov__ %eax, (%rsp)
2. mov__ (%rax), %dx
3. mov__ $0xff, %bl
4. mov__ (%rsp,%rdx,4),%dl
5. mov__ (%rdx), %rax
6. mov__ %dx, (%rax)
```

Practice: `mov` And Data Sizes

For each of the following `mov` instructions, determine the appropriate suffix based on the operands (e.g. `movb`, `movw`, `movl` or `movq`).

```
1. movl %eax, (%rsp)
2. movw (%rax), %dx
3. movb $0xff, %bl
4. movb (%rsp,%rdx,4),%dl
5. movq (%rdx), %rax
6. movw %dx, (%rax)
```

`mov`

- The **`movabsq`** instruction is used to write a 64-bit Immediate (constant) value.
- The regular **`movq`** instruction can only take 32-bit immediates.
- 64-bit immediate as source, only register as destination.

```
movabsq $0x0011223344556677, %rax
```

`movz` and `movs`

- There are two `mov` instructions that can be used to copy a smaller source to a larger destination: **`movz`** and **`movs`**.
- **`movz`** fills the remaining bytes with zeros
- **`movs`** fills the remaining bytes by sign-extending the most significant bit in the source.
- The source must be from memory or a register, and the destination is a register.

`movz` and `movs`

`MOVZ S,R`   `R ← ZeroExtend(S)`

Instruction	Description
`movzbw`	Move zero-extended byte to word
`movzbl`	Move zero-extended byte to double word
`movzwl`	Move zero-extended word to double word
`movzbq`	Move zero-extended byte to quad word
`movzwq`	Move zero-extended word to quad word

`movz` and `movs`

`MOVS S,R`   `R ← SignExtend(S)`

Instruction	Description
`movsbw`	Move sign-extended byte to word
`movsbl`	Move sign-extended byte to double word
`movswl`	Move sign-extended word to double word
`movsbq`	Move sign-extended byte to quad word
`movswq`	Move sign-extended word to quad word
`movslq`	Move sign-extended double word to quad word
`cltq`	Sign-extend `%eax` to `%rax` `%rax <- SignExtend(%eax)`

Lecture Plan

- **Recap:** `mov` so far
- Data and Register Sizes
- The `lea` Instruction

`lea`

The `lea` instruction copies an “effective address” from one place to another.

```
lea        src,dst
```

Unlike **mov**, which copies data at the address src to the destination, **lea** copies the value of src *itself* to the destination.

The syntax for the destinations is the same as **mov**. The difference is how it handles the src.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in `%rax`), and copy data there into `%rdx`	Copy 6 + what's in `%rax` into `%rdx`.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in `%rax`), and copy data there into `%rdx`	Copy 6 + what's in `%rax` into `%rdx`.
`(%rax, %rcx), %rdx`	Go to the address (what's in `%rax` + what's in `%rcx`) and copy data there into `%rdx`	Copy (what's in `%rax` + what's in `%rcx`) into `%rdx`.

`lea` vs. `mov`

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in `%rax`), and copy data there into `%rdx`	Copy 6 + what's in `%rax` into `%rdx`.
`(%rax, %rcx), %rdx`	Go to the address (what's in `%rax` + what's in `%rcx`) and copy data there into `%rdx`	Copy (what's in `%rax` + what's in `%rcx`) into `%rdx`.
`(%rax, %rcx, 4), %rdx`	Go to the address (`%rax` + 4 * `%rcx`) and copy data there into `%rdx`.	Copy (`%rax` + 4 * `%rcx`) into `%rdx`.

`lea` vs. `mov`

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in `%rax`), and copy data there into `%rdx`	Copy 6 + what's in `%rax` into `%rdx`.
`(%rax, %rcx), %rdx`	Go to the address (what's in `%rax` + what's in `%rcx`) and copy data there into `%rdx`	Copy (what's in `%rax` + what's in `%rcx`) into `%rdx`.
`(%rax, %rcx, 4), %rdx`	Go to the address (`%rax` + 4 * `%rcx`) and copy data there into `%rdx`.	Copy (`%rax` + 4 * `%rcx`) into `%rdx`.
`7(%rax, %rax, 8), %rdx`	Go to the address (7 + `%rax` + 8 * `%rax`) and copy data there into `%rdx`.	Copy (7 + `%rax` + 8 * `%rax`) into `%rdx`.

Unlike **`mov`**, which copies data at the address src to the destination, **`lea`** copies the value of src itself to the destination.

Recap

- **Recap:** `mov` so far
- Data and Register Sizes
- The **lea** Instruction

Next Time: Logical and Arithmetic Operations