CS107, Lecture 16
Assembly: Arithmetic and Logic

Reading: B&O 3.5-3.6

CS107 Topic 5

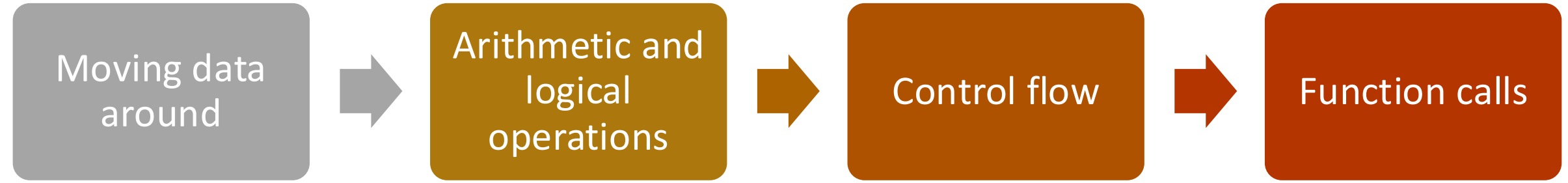
How does a computer interpret and execute C programs?

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- We can learn how to reverse engineer and exploit programs at the assembly level

assign5: find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.

Learning Assembly



Last Lecture **This Lecture**

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

Helpful Assembly Resources

- **Course textbook** (reminder: see relevant readings for each lecture on the Calendar page, http://cs107.stanford.edu/calendar.html)
- **CS107 Assembly Reference Sheet:** http://cs107.stanford.edu/resources/x86-64-reference.pdf
- **CS107 Guide to x86-64**: http://cs107.stanford.edu/guide/x86-64.html

Learning Goals

- Learn how to perform arithmetic and logical operations in assembly
- Begin to learn how to read assembly and understand the C code that generated it

Lecture Plan

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

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

Lecture Plan

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

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

mov

The **`mov`** instruction copies 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**)*
- Register
- Memory Location
  (*at most one of **src, dst**)*

mov

The **mov** instruction copies bytes from one place to another.

mov *src*, *dst*

Options include:

"Copy this exact value somewhere"

Immediate, e.g. $0x401

mov

The **mov** instruction copies bytes from one place to another.

mov *src*, *dst*

Options include:

"Copy this exact value somewhere" → *Immediate, e.g. $0x401*

"Copy what's in this register somewhere, or copy something into this register"

Register, e.g. %rax → *Register, e.g. %rax*

mov

The **mov** instruction copies bytes from one place to another.

mov *src*, *dst*

Options include:

src	*dst*
Immediate, e.g. $0x401	
Register, e.g. %rax	*Register, e.g. %rax*
Memory Location	*Memory Location*

"Copy this exact value somewhere"

"Copy what's in this register somewhere, or copy something into this register"

"Copy what's at this memory location somewhere, or copy something to this location in memory"

mov

The **mov** instruction copies bytes from one place to another.

mov *src*, *dst*

Options include:

"Copy this exact value somewhere"	*Immediate, e.g. $0x401*	
"Copy what's in this register somewhere, or copy something into this register"	*Register, e.g. %rax*	*Register, e.g. %rax*
	Memory Location	***Memory Location***

"Copy what's at this memory location somewhere, or copy something to this location in memory"

Memory Location Syntax

Hardcoded address (e.g. 0x104) or…

Memory Location Syntax

Hardcoded address (e.g. 0x104) or…

`Immediate (register1, register2, scale factor)`

e.g.

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

"Go to address **Immediate + register1 + (register2 * scale factor**)"

Key idea: any of the parts of this form can be omitted, in which case they are ignored.

Memory Location Syntax

Syntax	Go to this memory location:
0x104 (no $)	0x104
(%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
(, %rdx, 4)	What's in %rdx times 4
0x4(, %rdx, 4)	What's in %rdx times 4, plus 4
(%rax, %rdx, 2)	Sum of %rax and (what's in %rdx times 2)
0x4(%rax, %rdx, 2)	Sum of 0x4 and %rax and (what's in %rdx times 2)

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)

Practice #3: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value *0x5* is stored in register %rcx, the value *0x90* is stored in register %rax, the value *0x2* is stored in register %rdx, and value *0xc* is stored at address *0x98*.

1. `mov $0x42,0xf0(,%rcx,2)`

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

For #2, respond with your thoughts on PollEv:
pollev.com/cs107

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

Displacement Base Index Scale (1,2,4,8)

L16. For #2, what is in %rbx after this instruction?

Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!

From Assembly to C

What might be the equivalent C-like operation?

1. `mov $0x0,%rdx`
2. `mov %rdx,%rcx`
3. `mov $0x42,(%rdi)`
4. `mov (%rax,%rcx,8),%rax`

From Assembly to C

What might be the equivalent C-like operation?

1. `mov $0x0,%rdx -> maybe long x = 0`
2. `mov %rdx,%rcx -> maybe long x = y;`
3. `mov $0x42,(%rdi) -> maybe *ptr = 0x42;`
4. `mov (%rax,%rcx,8),%rax -> maybe long x = arr[i];`

Indirect addressing is like pointer arithmetic/deref!

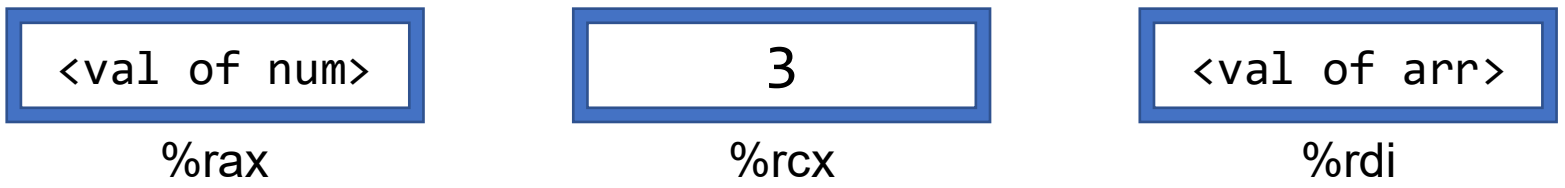
Scale Factor and Array Indexing

The scale factor is useful because in assembly, pointer arithmetic is done in terms of bytes, *not* number of places like in C. For example:

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

This last line of code in assembly could be:

```
mov (%rdi, %rcx, 8),%rax
```



Lecture Plan

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

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

Data Sizes

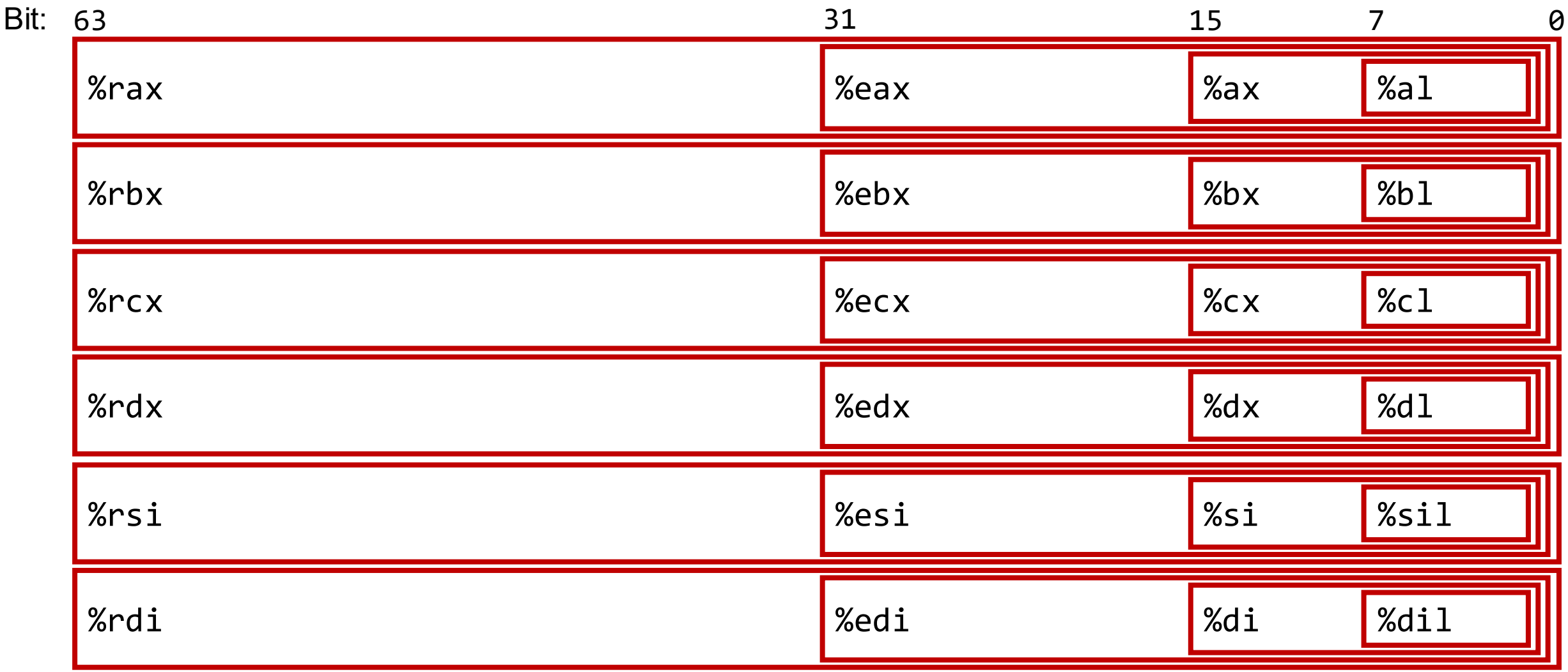
Data sizes in assembly have slightly different terminology to get used to:

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

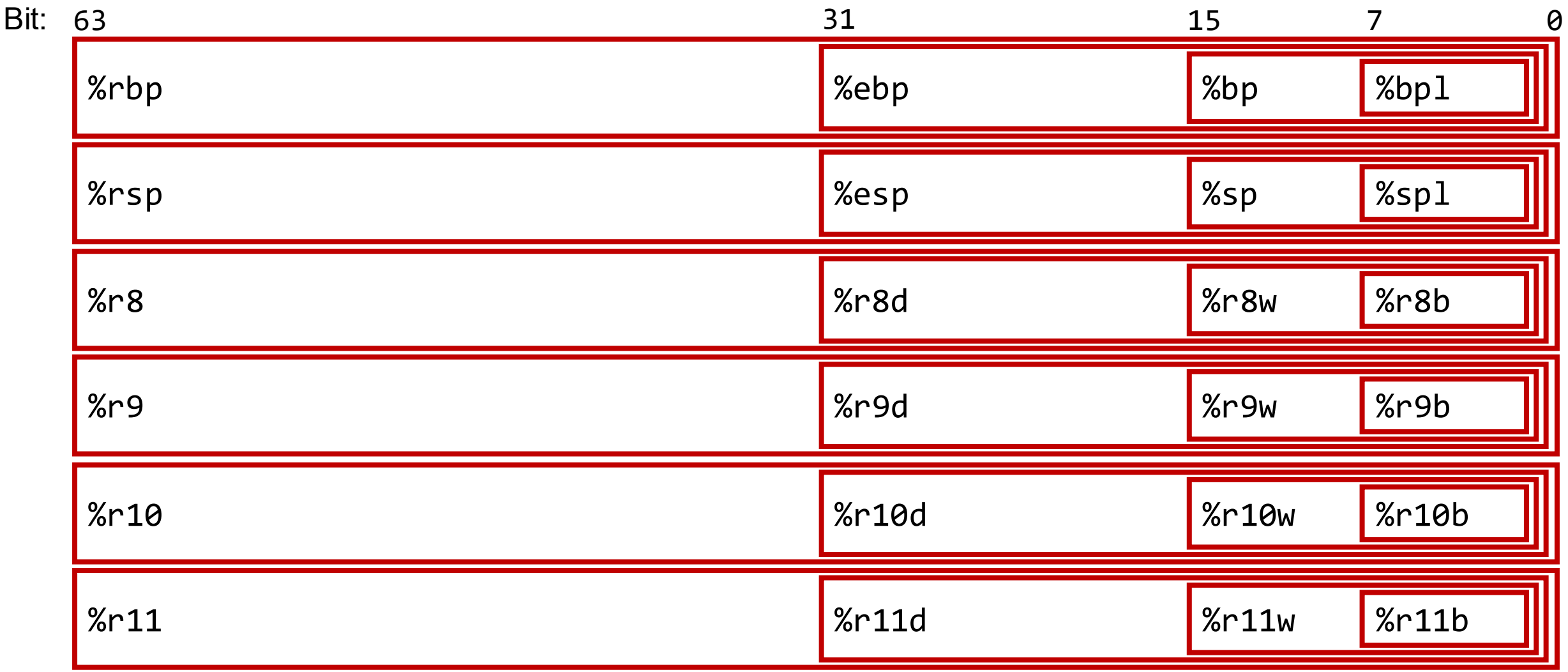
Assembly instructions can have suffixes to refer to these sizes:

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

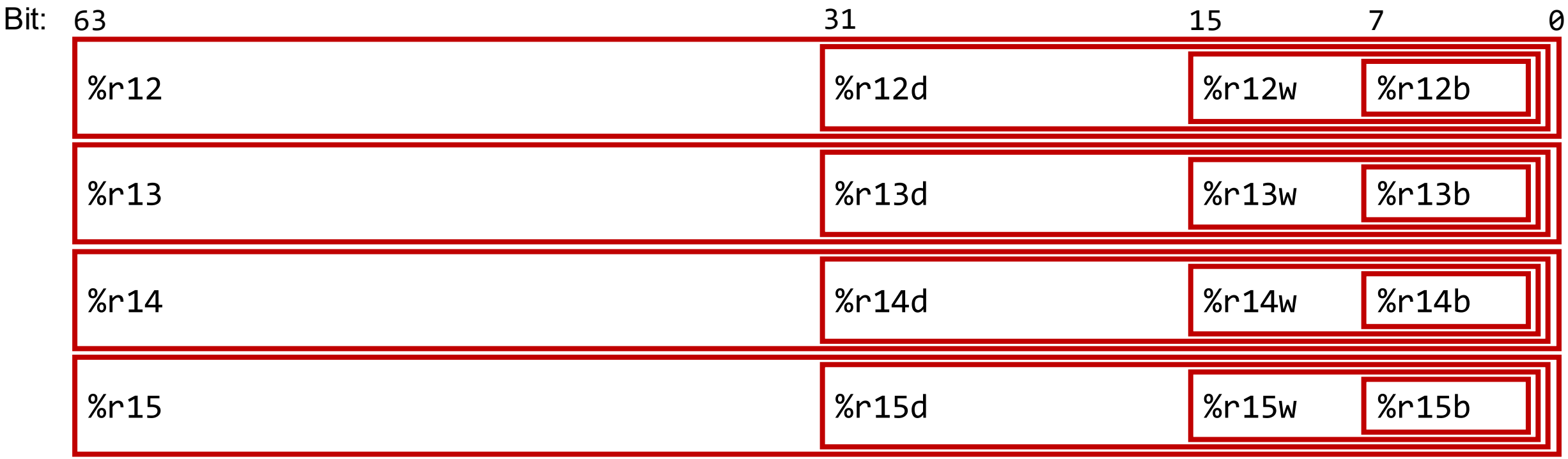
Register Sizes



Register Sizes



Register Sizes



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

Sometimes, you might see mov suffixes that specify the amount of data being moved. Other times, they are omitted if we can deduce the size from the arguments.

```
movl %eax,(%rsp)
movw (%rax),%dx
movb (%rsp,%rdx,4),%dl
mov  $0x0,%eax
```

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)`

Register Sizes

- The operand forms with parentheses (e.g. **mov (%rax)**) require that registers in parentheses be the 64-bit registers.
- For that reason, you may see smaller registers extended with e.g. **movs** into the larger registers before these kinds of instructions.

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

```
0000000000401136 <sum_array>:
  401136:    b8 00 00 00 00      mov    $0x0,%eax
  40113b:    ba 00 00 00 00      mov    $0x0,%edx
  401140:    39 f0               cmp    %esi,%eax
  401142:    7d 0b               jge    40114f <sum_array+0x19>
  401144:    48 63 c8            movslq %eax,%rcx
  401147:    03 14 8f            add    (%rdi,%rcx,4),%edx
  40114a:    83 c0 01            add    $0x1,%eax
  40114d:    eb f1               jmp    401140 <sum_array+0xa>
  40114f:    89 d0               mov    %edx,%eax
  401151:    c3                  retq
```

Lecture Plan

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

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

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. Destination must be a register.

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.

Reverse Engineering Practice

```
void calculate(long x, long y, long *ptr) {
    ____?____;
}

----------

calculate:
  leaq (%rdi,%rsi,2), %rax
  movq %rax, (%rdx)
  ret
```

Note: assume x is in %rdi, y is in %rsi and ptr is in %rdx.

Reverse Engineering Practice

```
void calculate(long x, long y, long *ptr) {
    *ptr = x + 2 * y;
}

----------

calculate:
  leaq (%rdi,%rsi,2), %rax
  movq %rax, (%rdx)
  ret
```

A Note About Operand Forms

- Many instructions we will see share the same address operand forms that **mov** uses.
  - Eg. 7(%rax, %rcx, 2).
- These forms work the same way for other instructions, with the exception of **lea:**
  - It interprets this form as just the calculation, *not the dereferencing*
  - lea 8(%rax,%rdx),%rcx -> Calculate 8 + %rax + %rdx, put it in %rcx

Summary: parentheses means “dereference”, except for with **lea**.

Unary Instructions

The following instructions operate on a single operand (register or memory):

Instruction	Effect	Description
`inc D`	`D ← D + 1`	Increment
`dec D`	`D ← D - 1`	Decrement
`neg D`	`D ← -D`	Negate
`not D`	`D ← ~D`	Complement

Examples:

```
incq 16(%rax)
dec %rdx
not %rcx
```

Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations. Read it as, e.g. "Subtract S from D":

Instruction	Effect	Description
`add S, D`	`D ← D + S`	Add
`sub S, D`	`D ← D - S`	Subtract
`imul S, D`	`D ← D * S`	Multiply
`xor S, D`	`D ← D ^ S`	Exclusive-or
`or S, D`	`D ← D | S`	Or
`and S, D`	`D ← D & S`	And

Examples:

```
addq %rcx,(%rax)
xorq $16,(%rax, %rdx, 8)
subq %rdx,8(%rax)
```

Recap

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

Next Time: more arithmetic operations, and reverse engineering practice

Lecture 16 takeaway: mov has a variety of forms to specify an address – they are all variations of one overall form. There are also different register sizes that may be used in assembly instructions. **lea** is an instruction used for arithmetic that shares the same operand form as mov, but lea interprets them differently.

Extra Practice

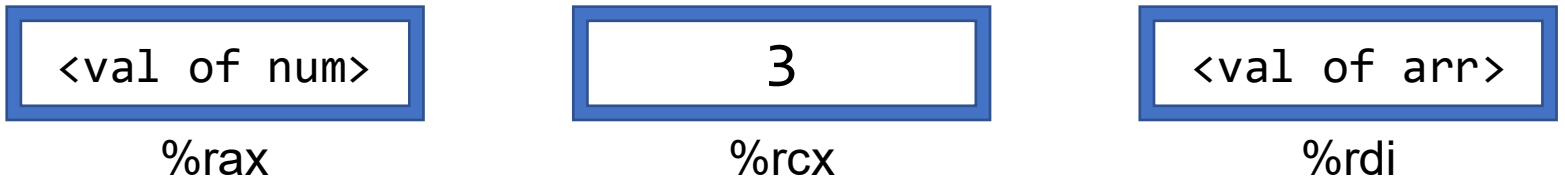
1. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. **results in** this register layout

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

```
mov (%rdi, %rcx, 8),%rax
```



1. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. **results in** this register layout

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

```
long num = arr[3];
long num = *(arr + 3);
long num = *(arr + y);
```

(assume `long y = 3;` declared earlier)

```
mov (%rdi, %rcx, 8),%rax
```

%rax	%rcx	%rdi
<val of num>	3	<val of arr>

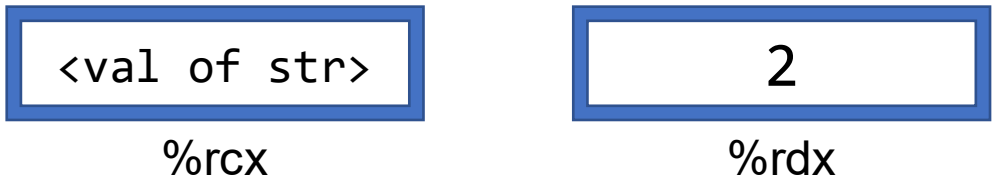
2. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

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

```
mov $0x63,(%rcx,%rdx,1)
```



2. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

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

```
str[2] = 'c';
*(str + 2) = 'c';
```

```
mov $0x63,(%rcx,%rdx,1)
```