CS107, Lecture 11
Assembly: Arithmetic and Logic

Reading: B&O 3.5-3.6

CS107 Topic 5: How does a computer interpret and execute C programs?

CS107 Topic 5

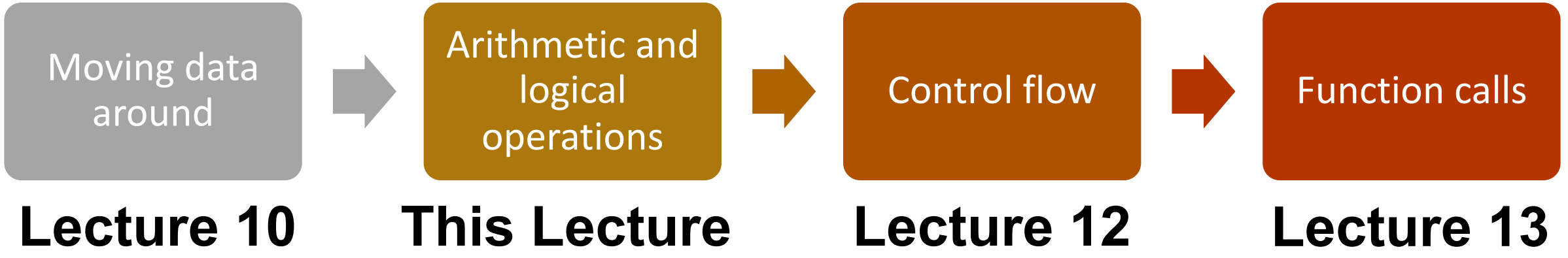
How does a computer interpret and execute C programs?

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



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
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

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
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

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**)*

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."

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)

Lecture Plan

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

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

Data Sizes

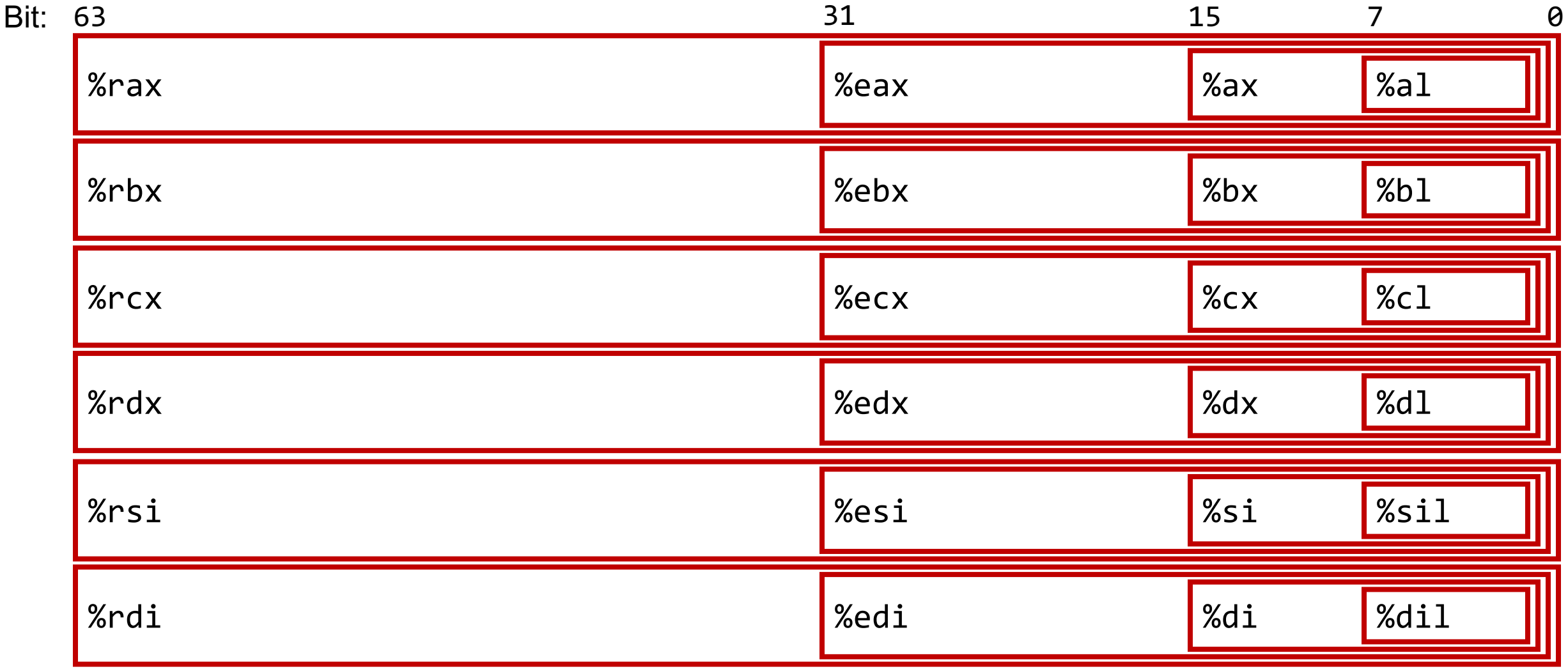
Data sizes in assembly have slightly different terminology to get used to:

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

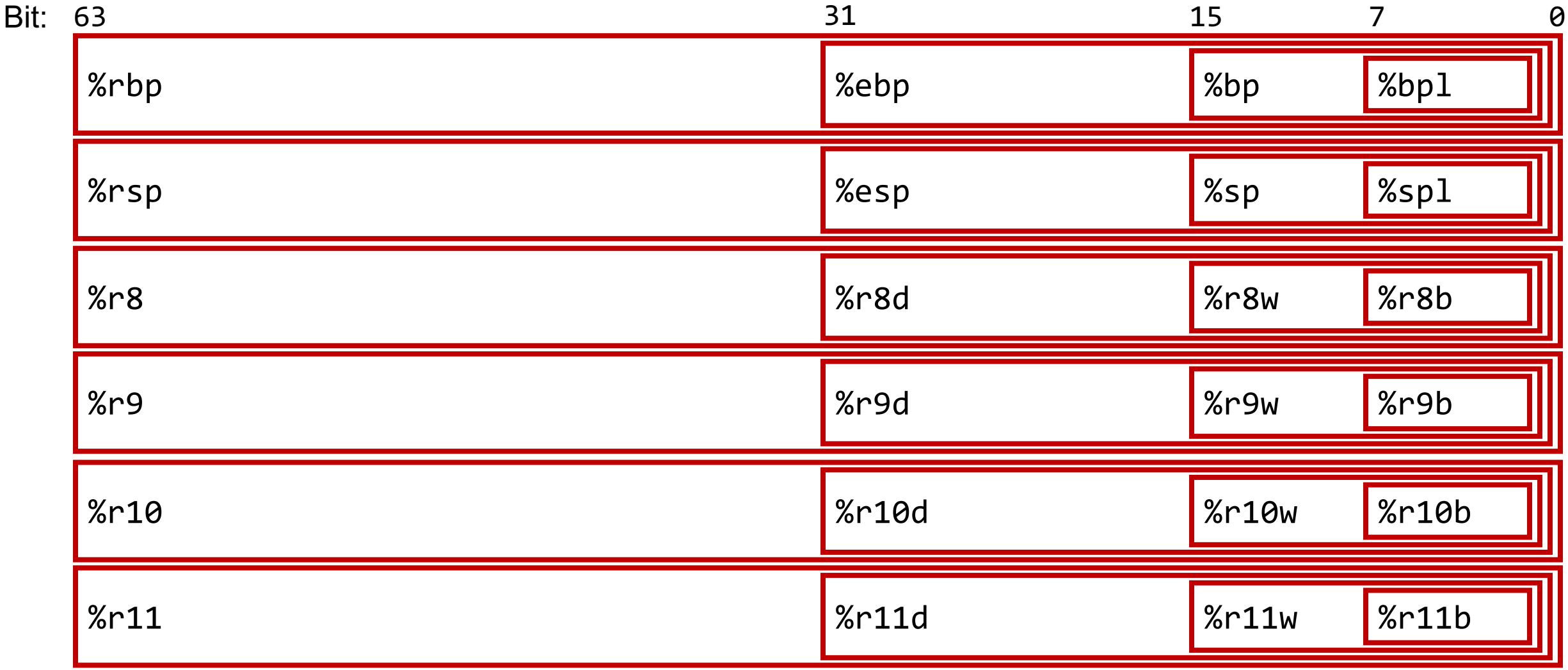
Assembly instructions can have suffixes to refer to these sizes:

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

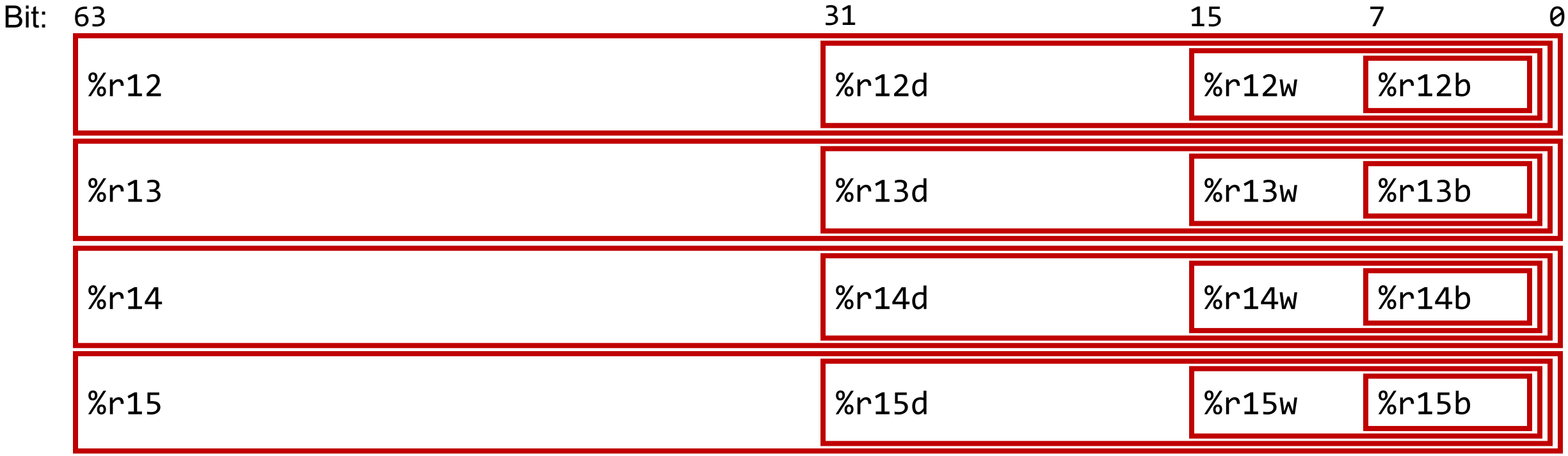
Register Sizes



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

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)`

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**
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

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.

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(int x, int y, int *ptr) {
    ____?____;
}

----------

calculate:
  leal (%rdi,%rsi,2), %eax
  movl %eax, (%rdx)
  ret
```

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

Reverse Engineering Practice

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

----------

calculate:
  leal (%rdi,%rsi,2), %eax
  movl %eax, (%rdx)
  ret
```

Lecture Plan

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

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

A Note About Operand Forms

- Many instructions 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

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)
```

Shift Instructions

The following instructions have two operands: the shift amount **k** and the destination to shift, **D**. **k** can be either an immediate value, or the byte register **%cl** (and only that register!)

Instruction	Effect	Description
`sal k, D`	D ← D << k	Left shift
`shl k, D`	D ← D << k	Left shift (same as `sal`)
`sar k, D`	D ← D $>>_A$ k	Arithmetic right shift
`shr k, D`	D ← D $>>_L$ k	Logical right shift

Examples:

```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

Shift Amount

Instruction	Effect	Description
`sal k, D`	D ← D << k	Left shift
`shl k, D`	D ← D << k	Left shift (same as `sal`)
`sar k, D`	D ← D $>>_A$ k	Arithmetic right shift
`shr k, D`	D ← D $>>_L$ k	Logical right shift

- When using **%cl**, the width of what you are shifting determines what portion of **%cl** is used.
- For **w** bits of data, it looks at the low-order **log2(w)** bits of **%cl** to know how much to shift.
  - If **%cl** = 0xff, then: **shlb** shifts by 7 because it considers only the low-order log2(8) = 3 bits, which represent 7. **shlw** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.

Assembly Exploration

- Let’s pull these commands together and see how some C code might be translated to assembly.
- Compiler Explorer is a handy website that lets you quickly write C code and see its assembly translation. Let’s check it out!
- https://godbolt.org/z/Ecbde99e3

Code Reference: `calculate`

```
int calculate(int x, int arr[]) {
    int sum = x;
    sum += arr[0];
    sum <<= x;
    sum &= 512;
    return sum;
}

----------

calculate:
  movl %edi, %ecx
  movl %edi, %eax
  addl (%rsi), %eax
  sall %cl, %eax
  andl $512, %eax
  ret
```

Large Multiplication

- Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?
- If you specify two operands to **imul**, it multiplies them together and truncates until it fits in a 64-bit register.

```
imul S, D      D ← D * S
```

- If you specify one operand, it multiplies that by **%rax**, and splits the product across **2** registers. It puts the high-order 64 bits in **%rdx** and the low-order 64 bits in **%rax**.

Instruction	Effect	Description
`imulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Signed full multiply
`mulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Unsigned full multiply

Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide

- Terminology: **dividend / divisor = quotient with remainder**
- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- The high-order 64 bits of the dividend are in **%rdx**, and the low-order 64 bits are in **%rax**. The divisor is the operand to the instruction.
- The quotient is stored in **%rax**, and the remainder in **%rdx**.

Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide
`cqto`	`R[%rdx]:R[%rax] ← SignExtend(R[%rax])`	Convert to oct word

- Terminology: **dividend / divisor = quotient with remainder**
- The high-order 64 bits of the dividend are in **%rdx**, and the low-order 64 bits are in **%rax**. The divisor is the operand to the instruction.
- Most division uses only 64-bit dividends. The **cqto** instruction sign-extends the 64-bit value in **%rax** into **%rdx** to fill both registers with the dividend, as the division instruction expects.

Compiler Explorer Demo

https://godbolt.org/z/4cT75M4nd

Code Reference: full_divide

```
// Returns x/y, stores remainder in location stored in remainder_ptr
long full_divide(long x, long y, long *remainder_ptr) {
    long quotient = x / y;
    long remainder = x % y;
    *remainder_ptr = remainder;
    return quotient;
}

-------

full_divide:
  movq %rdi, %rax
  movq %rdx, %rcx
  cqto
  idivq %rsi
  movq %rdx, (%rcx)
  ret
```

Lecture Plan

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

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

Assembly Exercise 1

```
000000000040116e <sum_example1>:
  40116e: 8d 04 37          lea  (%rdi,%rsi,1),%eax
  401171: c3                retq
```

Which of the following is most likely to have generated the above assembly?

```
// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}
```

```
// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
```

```
// B)
int sum_example1(int x, int y) {
    return x + y;
}
```

Assembly Exercise 2

```
0000000000401172 <sum_example2>:
      401172: 8b 47 0c        mov  0xc(%rdi),%eax
      401175: 03 07           add  (%rdi),%eax
      401177: 2b 47 18        sub  0x18(%rdi),%eax
      40117a: c3              retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly above represents the C code's **sum** variable?

%eax

Assembly Exercise 3

```
0000000000401172 <sum_example2>:
      401172: 8b 47 0c        mov  0xc(%rdi),%eax
      401175: 03 07           add  (%rdi),%eax
      401177: 2b 47 18        sub  0x18(%rdi),%eax
      40117a: c3              retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly code above represents the C code's **6** (as in **arr[6]**)?

0x18

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}

----------
// x in %edi, arr in %rsi, i in %edx
add_to:
  movslq %edx, %rdx
  movl %edi, %eax
  addl (%rsi,%rdx,4), %eax
  ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}
```

```
// x in %edi, arr in %rsi, i in %edx
add_to:
  movslq %edx, %rdx            // sign-extend i into full register
  movl %edi, %eax              // copy x into %eax
  addl (%rsi,%rdx,4), %eax     // add arr[i] to %eax
  ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = x;
    sum += arr[i];
    return sum;
}

----------
// x in %edi, arr in %rsi, i in %edx
add_to:
  movslq %edx, %rdx            // sign-extend i into full register
  movl %edi, %eax              // copy x into %eax
  addl (%rsi,%rdx,4), %eax     // add arr[i] to %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax
  imull (%rdi), %eax
  subl 4(%rdi), %eax
  sarl $2, %eax
  addl $2, %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax          // copy y into %eax
  imull (%rdi), %eax       // multiply %eax by nums[0]
  subl 4(%rdi), %eax       // subtract nums[1] from %eax
  sarl $2, %eax            // shift %eax right by 2
  addl $2, %eax            // add 2 to %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[0] * y;
    z -= nums[1];
    z >>= 2;
    return z + 2;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax          // copy y into %eax
  imull (%rdi), %eax       // multiply %eax by nums[0]
  subl 4(%rdi), %eax       // subtract nums[1] from %eax
  sarl $2, %eax            // shift %eax right by 2
  addl $2, %eax            // add 2 to %eax
  ret
```

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;
}
```

We're 1/2 of the way to understanding assembly!
What looks understandable right now?

```
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
```

Recap

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

Lecture 11 takeaway: There are assembly instructions for arithmetic and logical operations. They share the same operand form as mov, but lea interprets them differently. There are also different register sizes that may be used in assembly instructions.

Next Time: control flow in assembly (while loops, if statements, and more)

Extra Practice

https://godbolt.org/z/hGKPWszq4

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------
// x in %rdi, ptr in %rsi
func:
  movq %rdi, %rax
  leaq 1(%rdi), %rcx
  movq %rcx, (%rsi)
  cqto
  idivq %rcx
  movq %rdx, %rax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------
// x in %rdi, ptr in %rsi
func:
  movq %rdi, %rax          // copy x into %rax
  leaq 1(%rdi), %rcx       // put x + 1 into %rcx
  movq %rcx, (%rsi)        // copy %rcx into *ptr
  cqto                     // sign-extend x into %rdx
  idivq %rcx               // calculate x / (x + 1)
  movq %rdx, %rax          // copy the remainder into %rax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}
----------
// x in %rdi, ptr in %rsi
func:
  movq %rdi, %rax          // copy x into %rax
  leaq 1(%rdi), %rcx       // put x + 1 into %rcx
  movq %rcx, (%rsi)        // copy %rcx into *ptr
  cqto                     // sign-extend x into %rdx
  idivq %rcx               // calculate x / (x + 1)
  movq %rdx, %rax          // copy the remainder into %rax
  ret
```

Side Note: Old GCC Output

```
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}
----------
// x in %rdi, ptr in %rsi
func:
  leaq 1(%rdi), %rcx        // put x + 1 into %rcx
  movq %rcx, (%rsi)         // copy %rcx into *ptr
  movq %rdi, %rax           // copy x into %rax
  cqto                      // sign-extend x into %rdx
  idivq %rcx                // calculate x / (x + 1)
  movq %rdx, %rax           // copy the remainder into %rax
  ret
```