# Lecture 08
# Arithmetic Instructions

Comp_Sci 213

Introduction to Computer Systems

Monday, Oct. 19, 2026

Kate Smith – Fall 2026

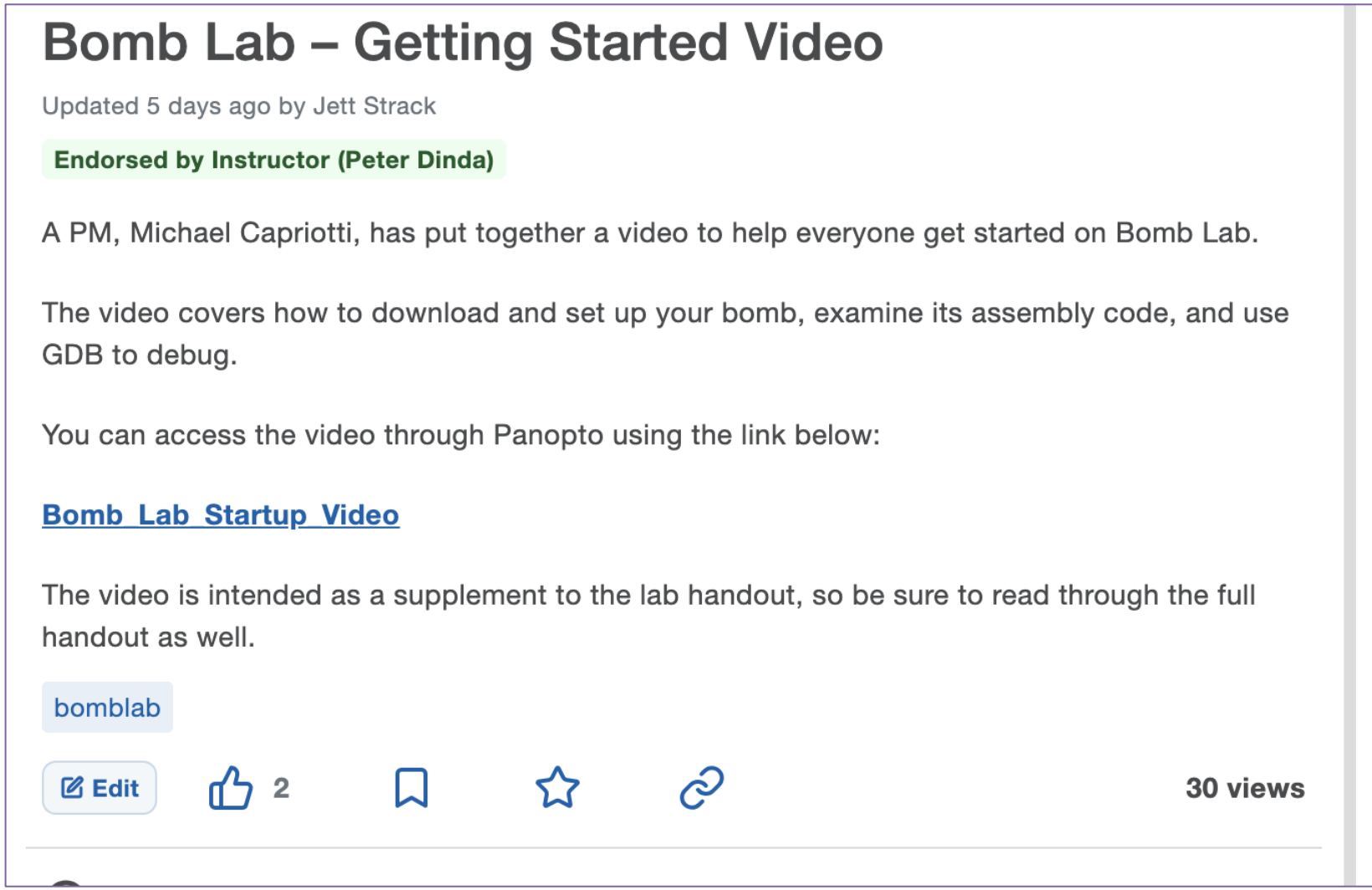
Slides adapted from:
Ghena, St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

# Midterm exam

- Midterm on Monday, Oct. 26
  - If you need accommodations, please contact AccessibleNU by today!!
- In class, pencil and paper
- You can have 1 cheat sheet, double sided
  - Handwritten or typed!
- You can have a calculator
- Our exam will be different from the other 213 section
- I will save some time during Wed. Oct 21 class (~20 minutes) for exam review

# Bomb Lab (Understanding Machine-level Programs)

- Due Thursday, Oct. 29
  - Requires today's material (arithmetic instructions in assembly) and control flow (next time)
  - Requires Procedures, Pointers, Arrays, and Structs
- There is a video on Piazza that's very helpful!



## Bomb Lab – Getting Started Video

Updated 5 days ago by Jett Strack

**Endorsed by Instructor (Peter Dinda)**

A PM, Michael Capriotti, has put together a video to help everyone get started on Bomb Lab.

The video covers how to download and set up your bomb, examine its assembly code, and use GDB to debug.

You can access the video through Panopto using the link below:

**Bomb_Lab_Startup_Video**

The video is intended as a supplement to the lab handout, so be sure to read through the full handout as well.

bomblab

Edit 2 **30 views**

# Topics

- Arithmetic Instructions
- Special Cases
  - Non 64-bit data
  - Load effective address
- Condition Codes
- Viewing x86-64 assembly

# Topics

- **Arithmetic Instructions**
- Special Cases
  - Non 64-bit data
  - Load effective address
- Condition Codes
- Viewing x86-64 assembly

# Some arithmetic operations

- Two-operand instructions

| Instruction | Effect | Description |
|---|---|---|
| `addq` S, D | D ← D + S | Add |
| `subq` S, D | D ← D – S | Subtract |
| `imulq` S, D | D ← D * S | Multiply |
| `xorq` S, D | D ← D ^ S | Exclusive or |
| `orq` S, D | D ← D \| S | Or |
| `andq` S, D | D ← D & S | And |

- Shifts

| Instruction | Effect | Description |
|---|---|---|
| `sarq` k, D | D ← D >> k | Shift arithmetic right |
| `shrq` k, D | D ← D >> k | Shift logical right |
| `salq` k, D | D ← D << k | Shift left |
| `shlq` k, D | D ← D << k | Shift left (same as salq) |

***Operand types***

- Immediate (e.g. constants)
- Register
- Memory

(Only one can be memory)

Be careful with operand order!!!

(Matters for some operations)

# Review: Left Shift: `x << y`

| Argument `x` | 00000010 |
|---|---|
| << 3 | ~~000~~00010***000*** |

| Argument `x` | 10100010 |
|---|---|
| << 3 | ~~101~~00010***000*** |

- Shift bit-vector `x` left by `y` positions
  - Truncate back to maximum size
    - Throws away bits on the left
  - Fill empty bits with 0
    - Same behavior for signed or unsigned

- Equivalent to multiplying by $2^y$
  - And then taking modulo (i.e. truncating overflow bits)

- Undefined behavior in C when:
  - `y < 0`, or `y ≥ bit_width(x)`
  - Also when some non-0 bits get shifted off (*probably* they get truncated)

# Review: Right Shift: `x >> y`

- Shift bit-vector x right y positions
  - Throw away extra bits on right
- But how to fill the new bits that open up?
  - Will depend on signed vs unsigned

| Argument x | 01100010 |
|---|---|
| Logi. >> 2 | 00011000 |
| Arith. >> 2 | 00011000 |

| Argument x | 10100010 |
|---|---|
| Logi. >> 2 | 00101000 |
| Arith. >> 2 | 11101000 |

- Unsigned: ***Logical shift***
  - Always fill with 0's on left
- Signed: ***Arithmetic shift***
  - Replicate most significant bit on left
  - Necessary for two's complement integer representation (sign extension!)
- Undefined behavior in C when:
  - `y < 0,  or  y ≥ bit_width(x)`

# Instruction naming

- Instruction names can seem odd…
- Names (usually) follow conventions
  - Common prefixes (**`add`**), suffixes (**`b`**, **`w`**, **`l`**, **`q`**), etc.
  - So you can understand pieces separately
  - Then combine their meanings
- Suffix reminder:
  - **`b`**: 1-byte, “byte”
  - **`w`**: 2-byte, “word
  - **`l`**: 4-byte, “long word”
  - **`q`**: 8-byte. “quad word”

# Some Arithmetic Operations

- Unary (one-operand) Instructions:

| Instruction | Effect | Description |
|---|---|---|
| **`incq`** D | D ← D + 1 | Increment |
| **`decq`** D | D ← D – 1 | Decrement |
| **`negq`** D | D ← -D | Negate |
| **`notq`** D | D ← ~D | Complement |

  - Unary operations have a single operand serving as the source and destination
- See textbook Section 3.5.5 for more instructions:
  - `mulq`: unsigned full multiply
  - `cqto`: convert to oct word (16 bytes)
  - `idivq`: signed divide
  - `divq`: unsigned divide

# Converting C to Assembly

`op src, dst`

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

# Converting C to Assembly

`op src, dst`

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
movq %rbx, %rax    (a = b;)
addq %rcx, %rax    (a += c;)
```

# Converting C to Assembly

`op src, dst`

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
movq   $0, %rax
addq %rbx, %rax
addq %rcx, %rax
```

Is this okay?

# Converting C to Assembly

`op src, dst`

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
movq   $0, %rax
addq %rbx, %rax
addq %rcx, %rax
```

Is this okay?

Yes: just a little slow

# Converting C to Assembly

`op src, dst`

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
addq %rbx, %rcx
movq %rcx, %rax
```

Is this okay?

# Converting C to Assembly

`op src, dst`

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
addq %rbx, %rcx
movq %rcx, %rax
```

Is this okay?

Probably not: it overwrites C which might still be needed later in code!

`addq, src, dst → dst = dst + src`

# Practice

*Reminder*

`addq, src, dst → dst = dst + src`

- Suppose `a → %rax, b → %rbx, c→ %rcx`

  Convert the following C statement to x86-64:

```
c = (a-b)+5;
```

[A]
```
movq %rax, %rcx
subq %rbx, %rcx
addq   $5, %rcx
```

[B]
```
movq %rax, %rcx
subq %rbx, %rcx
movq   $5, %rcx
```

[C]
```
subq %rcx, %rax, %rbx
addq %rcx, %rcx, $5
```

[D]
```
subq %rbx, %rax
addq $5, %rax
movq %rax, %rcx
```

# Practice

*Reminder*

`addq, src, dst → dst = dst + src`

- Suppose `a → %rax, b → %rbx, c→ %rcx`

  Convert the following C statement to x86-64:

```
c = (a-b)+5;
```

```
[A]
movq %rax, %rcx
subq %rbx, %rcx
addq   $5, %rcx
```

```
[B]
movq %rax, %rcx
subq %rbx, %rcx
movq   $5, %rcx
```

c = 5

```
[C]
subq %rcx, %rax, %rbx
addq %rcx, %rcx, $5
```

Not x86

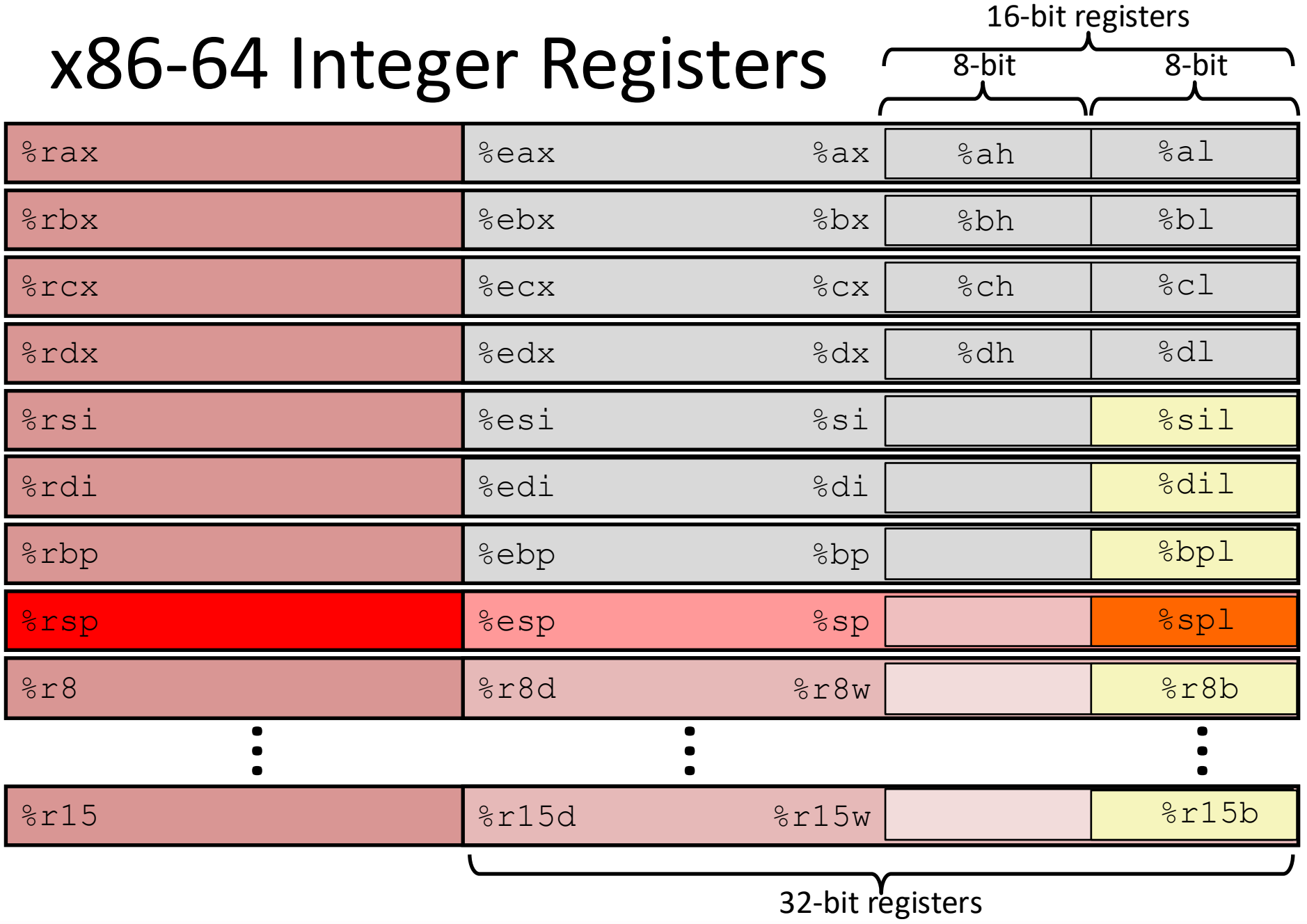
```
[D]
subq %rbx, %rax
addq $5, %rax
movq %rax, %rcx
```

Overwrites a

# Topics

- Arithmetic Instructions
- **Special Cases**
  - **Non 64-bit data**
  - Load effective address
- Condition Codes
- Viewing x86-64 assembly

# x86-64 Integer Registers



# Moving data of different sizes

`op src, dst`

- "Vanilla" move can only move between source and destination of the same size
  - Larger → smaller: use the smaller version of registers, which truncates
  - Smaller → larger: extension! We have two options: zero-extend or sign-extend

| Instruction | Effect | Description |
|---|---|---|
| **mov**X S,D<br>X ∈ {**q**, **l**, **w**, **b**} | D ← S | Copy quad-word (8B), long-word (4B), word (2B) or byte (1B) |
| **movs**XX S,D<br>XX ∈ {**bw**, **bl**, **wl**, **bq**, **wq**, **lq**} | D ← SignExtend(S) | Copy sign-extended byte to word, byte to long-word, etc. |
| **movz**XX S,D<br>XX ∈ {**bw**, **bl**, **wl**, **bq**, **wq**, **lq**} | D ← ZeroExtend(S) | Copy zero-extended byte to word, byte to long-word, etc. |
| **cltq**<br>(convert long to quad) | **%rax** ← SignExtend(**%eax**) | Sign-extend %eax to %rax<br>Identical to `movslq %eax, %rax` |

# Example: moving byte data

`op src, dst`

- Note the differences between `movb`, `movsbl` and `movzbl`
- Assume `%dl = 0xCD`, `%eax = 0x98765432`

| | |
|---|---|
| | *%eax = 0x987654CD* |
| `movb %dl,%al` | *%eax = 0xFFFFFFCD* |
| `movsbl %dl,%eax` | *%eax = 0x000000CD* |
| `movzbl %dl,%eax` | |

# 32-bit Instruction Peculiarities

`op src, dst`

- Instructions that move or generate 32-bit values also set the upper 32 bits of the respective 64-bit register to zero, while 16 or 8 bit instructions don't.

```
movabsq $0xffffffffffffffff, %rax   # rax = 0xffffffffffffffff
movb $0, %al                        # rax = 0xffffffffffffff00
movw $0, %ax                        # rax = 0xffffffffffff0000
movl $0, %eax                       # rax = 0x00000000000000000
```

- This includes 32-bit arithmetic! (e.g., `addl`)

# Topics

- Arithmetic Instructions
- **Special Cases**
  - Non 64-bit data
  - **Load effective address**
- Condition Codes
- Viewing x86-64 assembly

# Memory Addressing Modes: Basic

- Common need: interact with memory
  - Exact address might be made of multiple parts

- **Indirect:** `(R)` Mem[Reg[R]]
  - Data in register `R` specifies the memory address
  - Like pointer dereference in C
  - Example: `movq (%rcx), %rax`

- **Displacement:** `D(R)` Mem[Reg[R]+D]
  - Data in register `R` specifies the *start* of some memory region
  - Constant displacement `D` specifies the offset from that address
  - Example: `movq 8(%rbp), %rdx`

# Complete Memory Addressing Modes

- **General:**
  - `D(Rb,Ri,S)` Mem[Reg[Rb]+Reg[Ri]*S+D]
    - **Rb**: Base register (any register)
    - **Ri**: Index register (any register except `%rsp`)
    - **S**: Scale factor (1, 2, 4, 8) – *why these numbers?*
    - D: Constant displacement value

*The scale factors match the sizes in bytes of the common C types!*

- **Special cases** (see textbook Figure 3.3 or next slide)
  - `D(Rb,Ri)` Mem[Reg[Rb]+Reg[Ri]+D] `(S=1)`
  - `(Rb,Ri,S)` Mem[Reg[Rb]+Reg[Ri]*S] `(D=0)`
  - `(Rb,Ri)` Mem[Reg[Rb]+Reg[Ri]] `(S=1,D=0)`
  - `(,Ri,S)` Mem[Reg[Ri]*S] `(Rb=0,D=0)`

# Full list of addressing mode forms

| Type | Form | Operand value | Name |
|---|---|---|---|
| Immediate | $\$Imm$ | $Imm$ | Immediate |
| Register | $\mathtt{r}_a$ | $\mathsf{R}[\mathtt{r}_a]$ | Register |
| Memory | $Imm$ | $\mathsf{M}[Imm]$ | Absolute |
| Memory | $(\mathtt{r}_a)$ | $\mathsf{M}[\mathsf{R}[\mathtt{r}_a]]$ | Indirect |
| Memory | $Imm(\mathtt{r}_b)$ | $\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b]]$ | Base + displacement |
| Memory | $(\mathtt{r}_b,\mathtt{r}_i)$ | $\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$ | Indexed |
| Memory | $Imm(\mathtt{r}_b,\mathtt{r}_i)$ | $\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$ | Indexed |
| Memory | $(,\mathtt{r}_i,s)$ | $\mathsf{M}[\mathsf{R}[\mathtt{r}_i] \cdot s]$ | Scaled indexed |
| Memory | $Imm(,\mathtt{r}_i,s)$ | $\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_i] \cdot s]$ | Scaled indexed |
| Memory | $(\mathtt{r}_b,\mathtt{r}_i,s)$ | $\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$ | Scaled indexed |
| Memory | $Imm(\mathtt{r}_b,\mathtt{r}_i,s)$ | $\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$ | Scaled indexed |

Figure 3.3 **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.

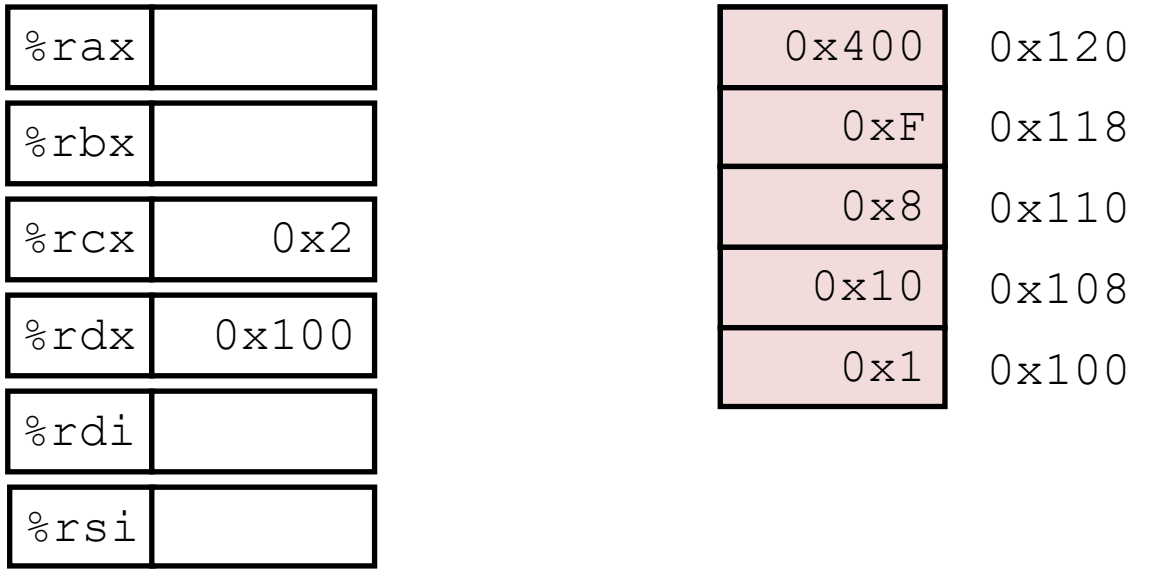
# Saving computed addresses

- Generally, any instruction with `()` in it, accesses memory
  - Address is computed first
  - Loads from memory if in a source operand
  - Stores into memory if in a destination operand
- But what if what you really want *is* the address?
  - **`lea`** – load effective address
  - Exception to `()` rule. Does NOT access memory
  - Also used for arbitrary arithmetic
    - This is the compiler's *favorite* instruction (will explain in a bit)

# Address computation instruction

- **`leaq src, dst`**
  - **`"lea"`** stands for *load effective address*
  - **`src`** MUST be an address expression (any of the formats we've seen)
  - **`dst`** is a register
  - Sets **`dst`** to the *address* computed by the **`src`** expression (does not go to memory! – it just does math)
  - Example: `leaq (%rdx,%rcx,4), %rax`
- Uses:
  - Computing addresses without a memory reference
    - *e.g.* translation of `p = &x[i];`
  - Computing arithmetic expressions of the form `x+k*i+d`
    - Though `k` can only be 1, 2, 4, or 8

# Example: `lea` vs. `mov`




D(**Rb**,**Ri**,**S**) →
Mem[Reg[Rb]+Reg[Ri]***S**+D]

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

# Example: `lea` vs. `mov`

Registers

| | |
|---|---|
| %rax | **0x108** |
| %rbx | |
| %rcx | 0x2 |
| %rdx | 0x100 |
| %rdi | |
| %rsi | |

| Memory | Word Address |
|---|---|
| 0x400 | 0x120 |
| 0xF | 0x118 |
| 0x8 | 0x110 |
| 0x10 | 0x108 |
| 0x1 | 0x100 |

D(**Rb**,**Ri**,**S**) →
Mem[Reg[Rb]+Reg[Ri]***S**+D]

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

# Example: `lea` vs. `mov`

Registers

| | |
|---|---|
| %rax | **0x108** |
| %rbx | **0x10** |
| %rcx | 0x2 |
| %rdx | 0x100 |
| %rdi | |
| %rsi | |

| Memory | Word Address |
|---|---|
| 0x400 | 0x120 |
| 0xF | 0x118 |
| 0x8 | 0x110 |
| 0x10 | 0x108 |
| 0x1 | 0x100 |

D(**Rb**,**Ri**,**S**) →
Mem[Reg[Rb]+Reg[Ri]***S**+D]

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

# Example: `lea` vs. `mov`

Registers

| | |
|---|---|
| %rax | **0x108** |
| %rbx | **0x10** |
| %rcx | 0x2 |
| %rdx | 0x100 |
| %rdi | **0x100** |
| %rsi | |

| Memory | Word Address |
|---|---|
| 0x400 | 0x120 |
| 0xF | 0x118 |
| 0x8 | 0x110 |
| 0x10 | 0x108 |
| 0x1 | 0x100 |

D(**Rb**,**Ri**,**S**) →
Mem[Reg[Rb]+Reg[Ri]***S**+D]

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

# Example: `lea` vs. `mov`

Registers

| | |
|---|---|
| %rax | **0x108** |
| %rbx | **0x10** |
| %rcx | 0x2 |
| %rdx | 0x100 |
| %rdi | **0x100** |
| %rsi | **0x1** |

| Memory | Word Address |
|---|---|
| 0x400 | 0x120 |
| 0xF | 0x118 |
| 0x8 | 0x110 |
| 0x10 | 0x108 |
| 0x1 | 0x100 |

D(**Rb**,**Ri**,**S**) →
Mem[Reg[Rb]+Reg[Ri]***S**+D]

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

# Practice Question #1

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0x2000 | B5 | B7 | DC | ED | 7D | 59 | 08 | 93 |
| 0x2008 | 1D | 23 | 58 | 46 | 9C | 22 | 2F | 5D |
| 0x2010 | C6 | 83 | 75 | 00 | 41 | 19 | 87 | 1C |
| 0x2018 | 24 | 0C | 26 | AA | C7 | BD | 03 | 1E |
| 0x2020 | E3 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x2028 | 8B | DB | 66 | D7 | 21 | 23 | 6B | 99 |

| Register | Value |
|---|---|
| %rax | 0x2000 |
| %rbx | 0x20 |
| %rcx | 0x8 |

D(**Rb**,**Ri**,**S**) → Mem[Reg[Rb]+Reg[Ri]***S**+D]

| Operation | Address Loaded | %rcx Value |
|---|---|---|
| **movq (%rax, %rbx), %rcx** | | |

First, determine whether an address is loaded, and if so, which address?

# Practice Question #1

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0x2000 | B5 | B7 | DC | ED | 7D | 59 | 08 | 93 |
| 0x2008 | 1D | 23 | 58 | 46 | 9C | 22 | 2F | 5D |
| 0x2010 | C6 | 83 | 75 | 00 | 41 | 19 | 87 | 1C |
| 0x2018 | 24 | 0C | 26 | AA | C7 | BD | 03 | 1E |
| 0x2020 | E3 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x2028 | 8B | DB | 66 | D7 | 21 | 23 | 6B | 99 |

| Register | Value |
|---|---|
| %rax | 0x2000 |
| %rbx | 0x20 |
| %rcx | 0x8 |

D(**Rb**,**Ri**,**S**) → Mem[Reg[Rb]+Reg[Ri]***S**+D]

| Operation | Address Loaded | %rcx Value |
|---|---|---|
| **movq (%rax, %rbx), %rcx** | 0x2020 | |

Second, determine the final value in %rcx

# Practice Question #1

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0x2000 | B5 | B7 | DC | ED | 7D | 59 | 08 | 93 |
| 0x2008 | 1D | 23 | 58 | 46 | 9C | 22 | 2F | 5D |
| 0x2010 | C6 | 83 | 75 | 00 | 41 | 19 | 87 | 1C |
| 0x2018 | 24 | 0C | 26 | AA | C7 | BD | 03 | 1E |
| 0x2020 | E3 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x2028 | 8B | DB | 66 | D7 | 21 | 23 | 6B | 99 |

| Register | Value |
|---|---|
| %rax | 0x2000 |
| %rbx | 0x20 |
| %rcx | 0x8 |

D(**Rb**,**Ri**,**S**) →
Mem[Reg[Rb]+Reg[Ri]***S**+D]

| Operation | Address Loaded | %rcx Value |
|---|---|---|
| **movq (%rax, %rbx), %rcx** | 0x2020 | 0x00000000000000E3 |

# Practice Question #2

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0x2000 | B5 | B7 | DC | ED | 7D | 59 | 08 | 93 |
| 0x2008 | 1D | 23 | 58 | 46 | 9C | 22 | 2F | 5D |
| 0x2010 | C6 | 83 | 75 | 00 | 41 | 19 | 87 | 1C |
| 0x2018 | 24 | 0C | 26 | AA | C7 | BD | 03 | 1E |
| 0x2020 | E3 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x2028 | 8B | DB | 66 | D7 | 21 | 23 | 6B | 99 |

| Register | Value |
|---|---|
| %rax | 0x2000 |
| %rbx | 0x20 |
| %rcx | 0x8 |

D(**Rb**,**Ri**,**S**) → Mem[Reg[Rb]+Reg[Ri]***S**+D]

| Operation | Address Loaded | %rcx Value |
|---|---|---|
| `leaq (%rax, %rbx), %rcx` | | |

First, determine whether an address is loaded, and if so, which address?

# Practice Question #2

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0x2000 | B5 | B7 | DC | ED | 7D | 59 | 08 | 93 |
| 0x2008 | 1D | 23 | 58 | 46 | 9C | 22 | 2F | 5D |
| 0x2010 | C6 | 83 | 75 | 00 | 41 | 19 | 87 | 1C |
| 0x2018 | 24 | 0C | 26 | AA | C7 | BD | 03 | 1E |
| 0x2020 | E3 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x2028 | 8B | DB | 66 | D7 | 21 | 23 | 6B | 99 |

| Register | Value |
|---|---|
| %rax | 0x2000 |
| %rbx | 0x20 |
| %rcx | 0x8 |

D(**Rb**,**Ri**,**S**) →
Mem[Reg[Rb]+Reg[Ri]***S**+D]

| Operation | Address Loaded | %rcx Value |
|---|---|---|
| **leaq (%rax, %rbx), %rcx** | None | |

Second, determine the final value in %rcx

# Practice Question #2

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0x2000 | B5 | B7 | DC | ED | 7D | 59 | 08 | 93 |
| 0x2008 | 1D | 23 | 58 | 46 | 9C | 22 | 2F | 5D |
| 0x2010 | C6 | 83 | 75 | 00 | 41 | 19 | 87 | 1C |
| 0x2018 | 24 | 0C | 26 | AA | C7 | BD | 03 | 1E |
| 0x2020 | E3 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x2028 | 8B | DB | 66 | D7 | 21 | 23 | 6B | 99 |

| Register | Value |
|---|---|
| %rax | 0x2000 |
| %rbx | 0x20 |
| %rcx | 0x8 |

D(**Rb**,**Ri**,**S**) →
Mem[Reg[Rb]+Reg[Ri]***S**+D]

| Operation | Address Loaded | %rcx Value |
|---|---|---|
| `leaq (%rax, %rbx), %rcx` | None | 0x2020 |

# Why does the compiler love `lea`?

- Sometimes it's good for computing addresses

- But usually, the compiler uses it to do math in fewer instructions
  - `addq` only adds a source and a destination, and overwrites destination
    - `addq, src, dst` → `dst = dst + src`
  - `leaq` adds up to two registers and an immediate, AND stores to a different register!

# Compiling Arithmetic Operations

```
00 int arith (long x, long y, long z){
01   long t1 = x+y;
02   long t2 = z+t1;
03   long t3 = x+4;
04   long t4 = y * 48;
05   long t5 = t3 + t4;
06   long rval = t2 * t5;
07   ....
08 }
```

- Compiler can reorder operations
- Can have one statement take multiple instructions
- Can have one instruction handle multiple statements

- **Don't expect a 1-1 mapping**

```
# rdi = x
# rsi = y
# rdx = z
```

# Compiling Arithmetic Operations

```
00 int arith (long x, long y, long z){
01   long t1 = x+y;
02   long t2 = z+t1;
03   long t3 = x+4;
04   long t4 = y * 48;
05   long t5 = t3 + t4;
06   long rval = t2 * t5;
07   ....
08 }
```

- Compiler can reorder operations
- Can have one statement take multiple instructions
- Can have one instruction handle multiple statements

- **Don't expect a 1-1 mapping**

Line 01 -
Line 04 -

```
leaq (%rsi,%rdi),%rcx
leaq (%rsi,%rsi,2),%rsi
salq $4,%rsi
```

```
# rdi = x
# rsi = y
# rdx = z
# rcx = x+y   (rcx = t1)
# rsi = y + 2*y = 3*y
# rsi = (3*y)*16 = 48*y (rsi = t4)
```

# Compiling Arithmetic Operations

```
00 int arith (long x, long y, long z){
01   long t1 = x+y;
02   long t2 = z+t1;
03   long t3 = x+4;
04   long t4 = y * 48;
05   long t5 = t3 + t4;
06   long rval = t2 * t5;
07   ....
08 }
```

- Compiler can reorder operations
- Can have one statement take multiple instructions
- Can have one instruction handle multiple statements

- **Don't expect a 1-1 mapping**

Line 01 -
Line 04 -
Line 02 -

```
leaq (%rsi,%rdi),%rcx
leaq (%rsi,%rsi,2),%rsi
salq $4,%rsi
addq %rdx,%rcx
```

```
# rdi = x
# rsi = y
# rdx = z
# rcx = x+y   (rcx = t1)
# rsi = y + 2*y = 3*y
# rsi = (3*y)*16 = 48*y (rsi = t4)
# rcx = z+t1 (rcx = t2)
```

# Compiling Arithmetic Operations

```
00 int arith (long x, long y, long z){
01   long t1 = x+y;
02   long t2 = z+t1;
03   long t3 = x+4;
04   long t4 = y * 48;
05   long t5 = t3 + t4;
06   long rval = t2 * t5;
07   ....
08 }
```

- Compiler can reorder operations
- Can have one statement take multiple instructions
- Can have one instruction handle multiple statements

- **Don't expect a 1-1 mapping**

Line 01 -
Line 04 -
Line 02 -
Lines 03 and 05 -

```
leaq (%rsi,%rdi),%rcx
leaq (%rsi,%rsi,2),%rsi
salq $4,%rsi
addq %rdx,%rcx
leaq 4(%rsi,%rdi),%rdi
```

```
# rdi = x
# rsi = y
# rdx = z
# rcx = x+y   (rcx = t1)
# rsi = y + 2*y = 3*y
# rsi = (3*y)*16 = 48*y (rsi = t4)
# rcx = z+t1 (rcx = t2)
# rdi = t4+x+4 (rdi = t5)
```

# Compiling Arithmetic Operations

```
00 int arith (long x, long y, long z){
01   long t1 = x+y;
02   long t2 = z+t1;
03   long t3 = x+4;
04   long t4 = y * 48;
05   long t5 = t3 + t4;
06   long rval = t2 * t5;
07   ....
08 }
```

- Compiler can reorder operations
- Can have one statement take multiple instructions
- Can have one instruction handle multiple statements

- **Don't expect a 1-1 mapping**

| | |
|---|---|
| Line 01 - | `leaq (%rsi,%rdi),%rcx` |
| Line 04 - | `leaq (%rsi,%rsi,2),%rsi` |
| | `salq $4,%rsi` |
| Line 02 - | `addq %rdx,%rcx` |
| Lines 03 and 05 - | `leaq 4(%rsi,%rdi),%rdi` |
| Line 06 - | `imulq %rcx,%rdi` |

```
# rdi = x
# rsi = y
# rdx = z
# rcx = x+y   (rcx = t1)
# rsi = y + 2*y = 3*y
# rsi = (3*y)*16 = 48*y (rsi = t4)
# rcx = z+t1 (rcx = t2)
# rdi = t4+x+4 (rdi = t5)
# rdi = t2*t5 (rdi = rval)
```

# Topics

- Arithmetic Instructions
- Special Cases
  - Non 64-bit data
  - Load effective address
- **Condition Codes**
- Viewing x86-64 assembly

# What can instructions do?

- Move data: ✓
- Arithmetic: ✓
- Transfer control: ✗
  - Instead of executing next instruction, go somewhere else

- Let's back out. Why do we want that?

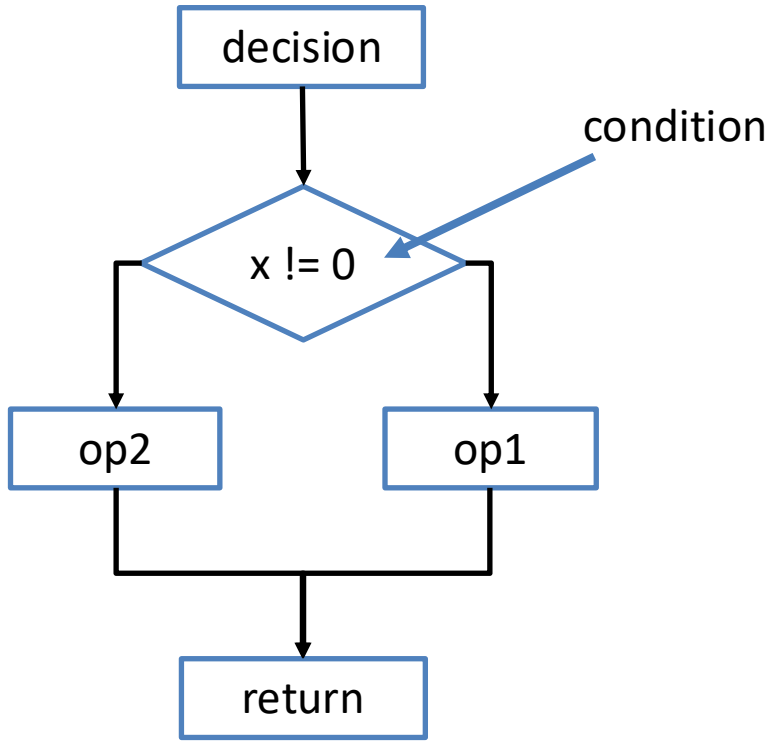
```
if (x > y)
    result = x-y;
else
    result = y-x;
```

```
while (x > y)
    result = x-y;
return result;
```

  - Sometimes we want to go from the red code to the green code
  - But the blue code is what's next!
  - Need to transfer control! Execute an instruction that is not the next one
  - And ***conditionally***, too! (i.e., based on a condition)

# Control relies on “conditions”

- Equality
  - Equals and not equals
- Comparison
  - Greater than, less than
- Others C doesn’t directly support
  - Negative/Positive
  - Parity



# x86 tracks conditions with condition codes

- Whenever a comparison occurs
  - Modify some “flags” based on what happened
  - Flags are one-bit fields

  - Terminology:
    - A bit is ***set*** if it is 1
    - A bit is ***cleared*** (or ***reset***) if it is 0

- x86 “condition codes” are a set of one-bit flags for tracking conditions

# What are the x86 condition codes?

- **Zero Flag** (ZF)
  - Result is zero
- **Sign Flag** (SF)
  - MSb of result is one
- **Overflow Flag** (OF)
  - Overflow occurred during arithmetic
- **Carry Flag** (CF)
  - Carry occurred during arithmetic
- **Parity Flag** (PF) (rarely used in practice)
  - Result has an ever number of 1 bits

Each of these is a one-bit register

We can't write or read these manually, they're interacted with via specific assembly instructions!

# How are condition codes set?

- Zero Flag

  000000000000…00000000000

  - Result is:

- Sign Flag

  1xxxxxxxxxxxx…xxxxxxxxxxxx

  - Result is:

# How are condition codes set?

- Overflow Flag

  Either:

  - Both numbers positive but result is negative

  Or:

  - Both numbers negative but result is positive

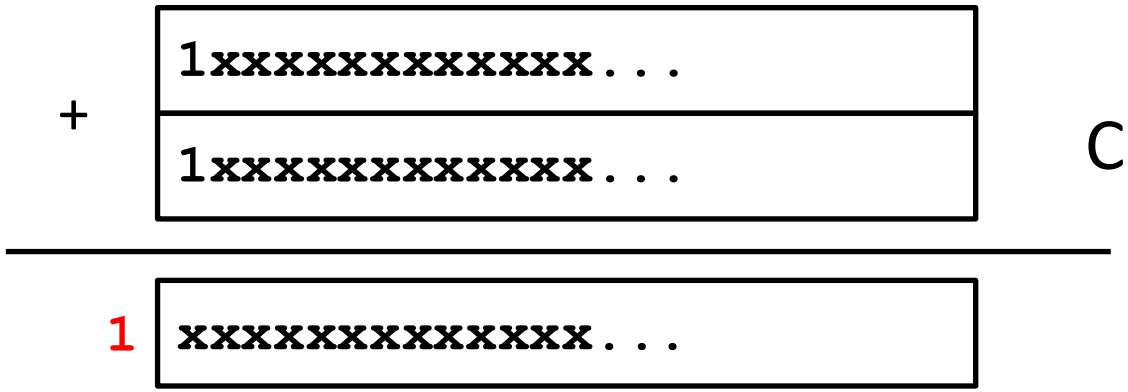



(w == y) && (w != z)

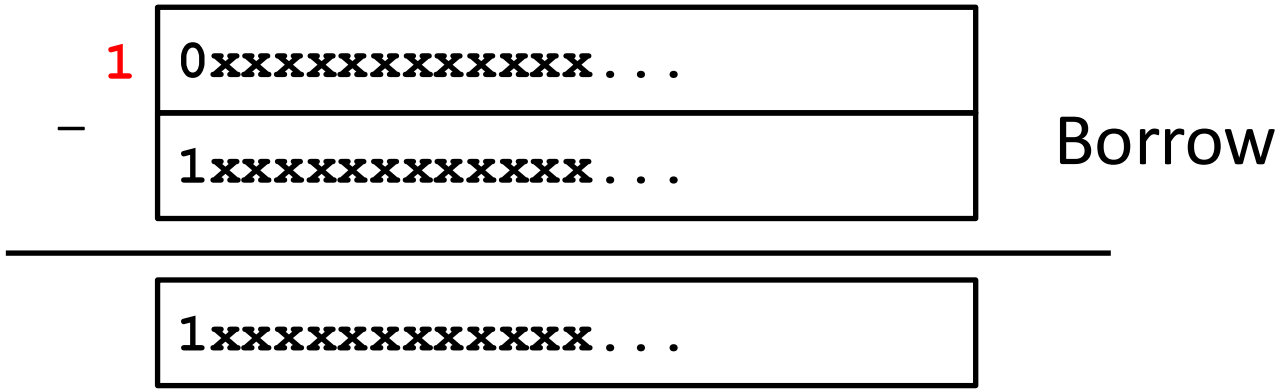
  - Detects overflow for two's complement numbers

# How are condition codes set?

- Carry Flag
  - An extra bit is produced or borrowed in calculation




  - Detects overflow for unsigned numbers
    - Below zero
    - Above max



# How are condition codes useful?

- The conditions we care about are boolean algebra combinations of the x86 condition codes
  - These combinations assume a **subtraction** operation set the condition codes

If you subtract two values, the condition codes let you compare the two values:

| Condition | Condition |
|---|---|
| Equal / Zero | `ZF` |
| Not Equal / Not Zero | `~ZF` |
| Negative | `SF` |
| Nonnegative | `~SF` |
| Greater (Signed) | `~(SF^OF)&~ZF` |
| Greater or Equal (Signed) | `~(SF^OF)` |
| Less (Signed) | `(SF^OF)` |
| Less or Equal (Signed) | `(SF^OF)\|ZF` |
| Above (unsigned) | `~CF&~ZF` |
| Below (unsigned) | `CF` |

# How are condition codes used?

- At machine level, conditional operations are a 2-step process:
  - Perform an operation that ***sets*** or ***clears*** condition codes (ask questions)
  - Then ***observe*** which condition codes are set

- Can use this observation for Boolean operations, conditionals, loops, etc.
  - We will see the first today, and more control next lecture…

# Necessary instructions for using condition codes

- We need two things:

1. **Instructions that compare values and set condition codes**

2. Instructions that observe condition codes and do something (or not)

# Explicitly Setting Condition Codes: Compare

- **`cmp{b,w,l,q}`** ***`Src2, Src1`***
- **`cmpq src, dst`** computes **`dst-src`**, then throws away the result
  - Sets condition codes along the way as a "subtraction" operation
  - Does NOT modify the destination register though
  - **Beware the order of the `cmp` operands!**
- Compare instruction is like 90% of all condition code usage
  - Often used for if statements and loop conditions

# Explicitly Setting Condition Codes: Test

- **`test{b,w,l,q}`** ***`Src2, Src1`***

- **`testq src, dst`** computes **`dst&src`**, then throws away the result!
  - Sets condition codes like an “and” operation
  - Again, doesn’t modify either register (not really a destination)
  - Order of arguments doesn’t matter here

- Useful when doing bit masking
  - E.g., **`x & 0x1`**, to know whether **`x`** is even or odd
  - If the result of the **`&`** is 0, it’s even, if 1, it’s odd

# Arithmetic instructions implicitly set condition codes

- All* arithmetic instructions set (and reset) condition codes in addition to producing a result
  - The same arithmetic instructions we learned earlier
  - They do still update the destination register
  - But also, they update the condition codes based on the result
    - We rarely care about this, but it matters occasionally

- *Exception: the `lea` instruction does not set condition codes
  - It's not "officially" an arithmetic instruction, but it is used for "cheap" math
  - Acts as an adder with a shifter

# Necessary instructions for using condition codes

- So now we need two things:

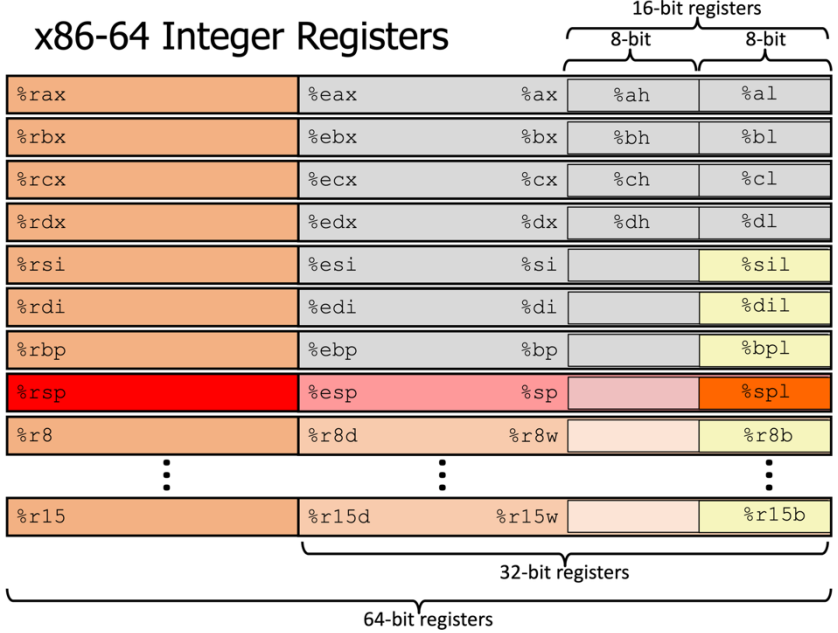
1. Instructions that compare values and set condition codes

2. **Instructions that observe condition codes and do something (or not)**

# Using condition codes to set registers

- **`setX`** family of instructions
  - Write single-byte destination register based on combinations of condition codes
    - **`set_ D`** where D is a 1-byte register
    - Example: **`sete %al`** means: **`%al`**=1 if the previous condition was "equal"

| SetX | Description | Condition |
|---|---|---|
| `sete` | Equal / Zero | `ZF` |
| `setne` | Not Equal / Not Zero | `~ZF` |
| `sets` | Negative | `SF` |
| `setns` | Nonnegative | `~SF` |
| `setg` | Greater (Signed) | `~(SF^OF)&~ZF` |
| `setge` | Greater or Equal (Signed) | `~(SF^OF)` |
| `setl` | Less (Signed) | `(SF^OF)` |
| `setle` | Less or Equal (Signed) | `(SF^OF)\|ZF` |
| `seta` | Above (unsigned) | `~CF&~ZF` |
| `setb` | Below (unsigned) | `CF` |




Note: suffixes do not indicate operand sizes, but rather conditions

These same suffixes will come back when we see other instructions that read condition codes.

Expect to be run after a **`cmp`** or else they won't make sense

# Explaining the set instructions

- Example `setle` – Less than or equal (signed)
  - Combination of condition codes: `(SF^OF)|ZF`
  - SF - Sign Flag (true if negative)
  - OF – Overflow Flag (true if signed overflow occurred)
  - ZF – Zero Flag (true if result is zero)

- All of the combos expect to be run after a `cmp src,dst`
  - `dst <= src` (runs `dst-src`)
    - If:
      - The result is zero – `src` and `dst` were equal
    - OR if one but not both:
      - The result is negative (and didn't overflow) – src was larger than dst
      - The result overflowed (and is positive) – dst is negative, src is positive

# What do you need to know?

- 90%+ of the time
  - `cmp` instruction followed by `setX` instruction (or a jump, next lecture)
- Don't have to think about condition codes at all! Use what `setX` tells you
  - Think of as `dst ??? src`
    - `dst <= src` or `dst != src` etc.
- 10% or less of the time
  - Arbitrary arithmetic instruction sets the condition codes
    - Or `testq` sets the condition codes
  - Followed by a `setX` or branch (next lecture)
  - And you actually have to think about which condition codes are set to figure out what the assembly is doing, which can be challenging

# Example of checking a Boolean condition

- Example: function that checks if first value is greater than the second value
  - Returns True or False (1 or 0 in a one-byte register)

```
bool gt (int x, int y)
{
  return x > y;
}
```

| Register | Use(s) |
|---|---|
| %rdi | Argument x |
| %rsi | Argument y |
| %rax | Return value |

```
    cmpq    %rsi, %rdi    # Compare x:y
    setg    %al           # Set when x > y (i.e., %rdi > %rsi)
    ret
```

# Two-Step Conditional Process: Boolean Operations

- Step 1, **`cmpq`**: compare quad words
  - *compare* the values in **`%rsi`** and **`%rdi`**, setting condition codes based on a subtraction
  - Are the two equal? Set the condition codes that records they were equal
  - Was the right one greater? Or less? Etc.
  - We don't know yet which answer we are going to need! So just save them all.

```
bool gt (int x, int y)
{
  return x > y;
}
```

| Register | Use(s) |
|---|---|
| `%rdi` | Argument x |
| `%rsi` | Argument y |
| `%rax` | Return value |

y x

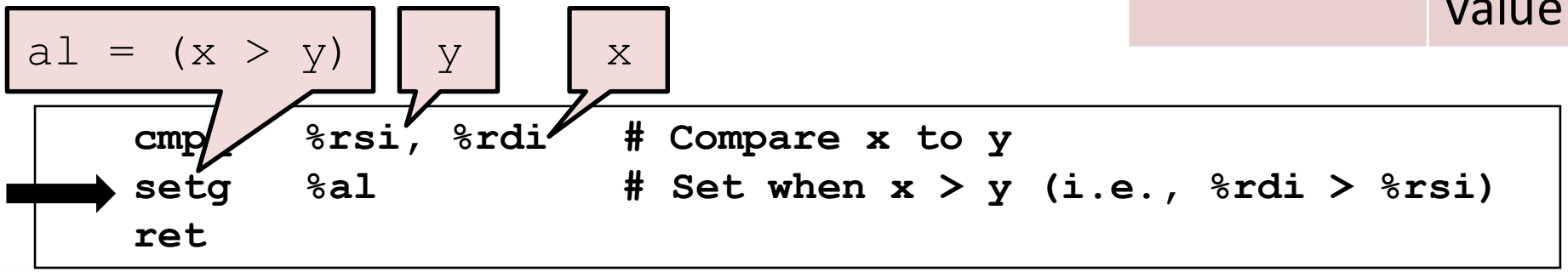
```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when x > y (i.e., %rdi > %rsi)
ret
```

# Two-Step Conditional Process: Boolean Operations

- Step 2, **`setX`**: set destination register to 1 if condition is met
  - **`setg`** = set if the 2nd operand is *greater than* the 1st (careful about the order!)
  - Reads the condition codes that encodes the answer to that question
  - Set the 1-byte register **`%al`** to 1 if true

```
bool gt (int x, int y)
{
  return x > y;
}
```

| Register | Use(s) |
|---|---|
| **`%rdi`** | Argument x |
| **`%rsi`** | Argument y |
| **`%rax`** | Return value |



# Understanding conditions

- `setX` (and others) read the current state of condition codes
  - Whatever it is, and whichever instruction changed it last
- So when you see (for example) `setne`, work backwards!
  - Look at previous instructions, to find the last one to change conditions
  - Then you'll know the two values that were compared
  - Ignore instructions that don't touch condition codes (like moves or lea)
- Usually you'll see a `cmpX` (or `testX`, or arithmetic) right before
  - But not always, so know what to do in general too

# Topics

- Arithmetic Instructions
- Special Cases
  - Non 64-bit data
  - Load effective address
- Condition Codes
- **Viewing x86-64 assembly**

# How to Get Your Hands on Assembly

- From C source code, using a compiler
  - **`gcc –O1 -S sum.c`**
    - Produces file **`sum.s`**
  - ***Warning***: May get very different results on different machines due to different versions of gcc and different compiler settings

**C Code:** sum.c

```
long plus(long x, long y);

void sum(long x, long y,
         long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 assembly: sum.s

```
sum:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

# How to Get Your Hands on Assembly

- From machine code, using a disassembler
  - **`objdump -d sum.o`**
  - ***Warning***: Disassemblers are approximate; some information is lost during translation from assembly to machine code
    - Label names are lost, what is just data (vs code) is lost, etc.
  - Useful if you don't have the source

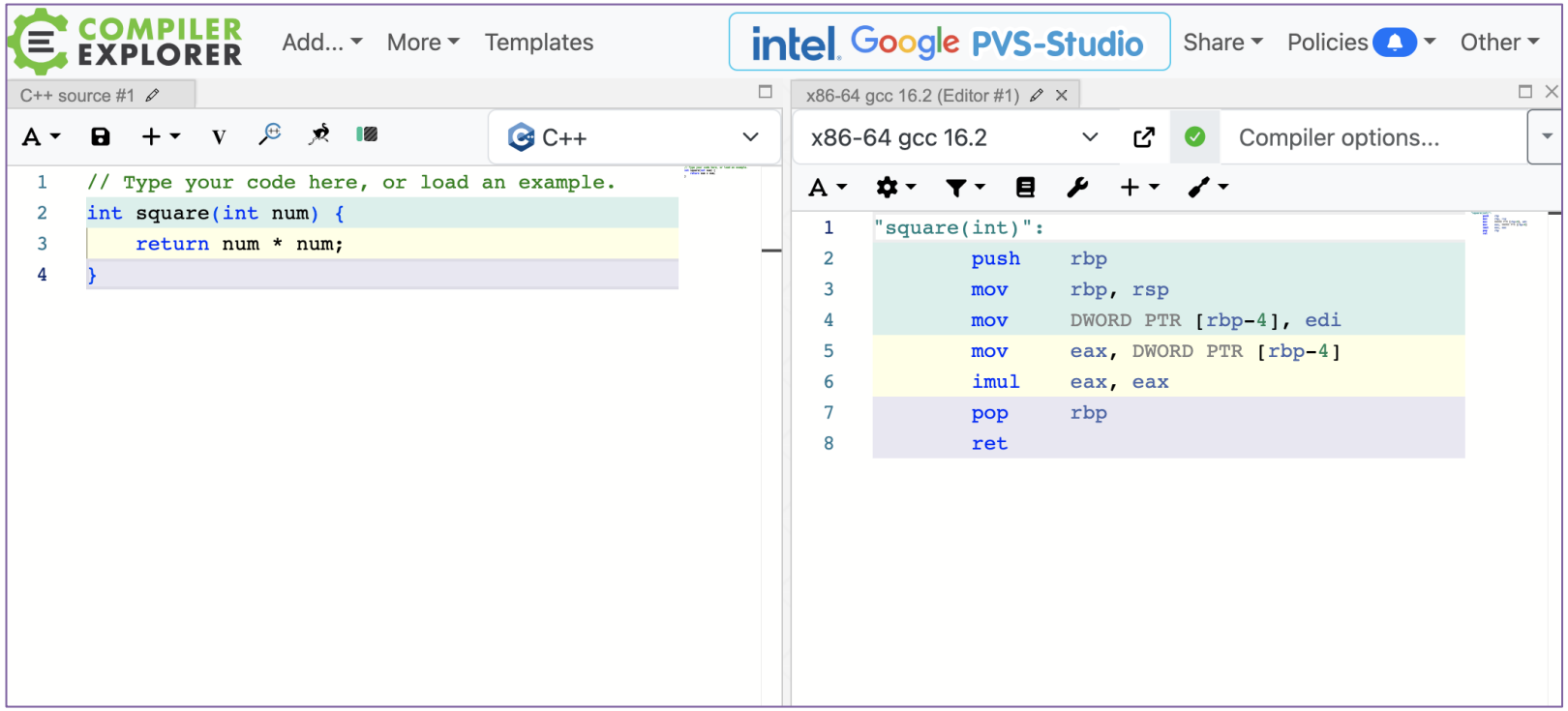
```
0000000000400595 <sum>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

# Godbolt

https://godbolt.org/

Allows you to write code and compile it into assembly code.

Really useful in seeing how code converts into assembly.



# Dive into System: ASM Visualizer

- Allows you to step through x86_64 instructions, functions, or even whole programs
  - Also support x86 (32-bit) and ARM64
- https://asm.diveintosystems.org/
- Example problem

