

Lecture 06

Arithmetic Instructions

CS213 – Intro to Computer Systems
Branden Ghen a – Spring 2021

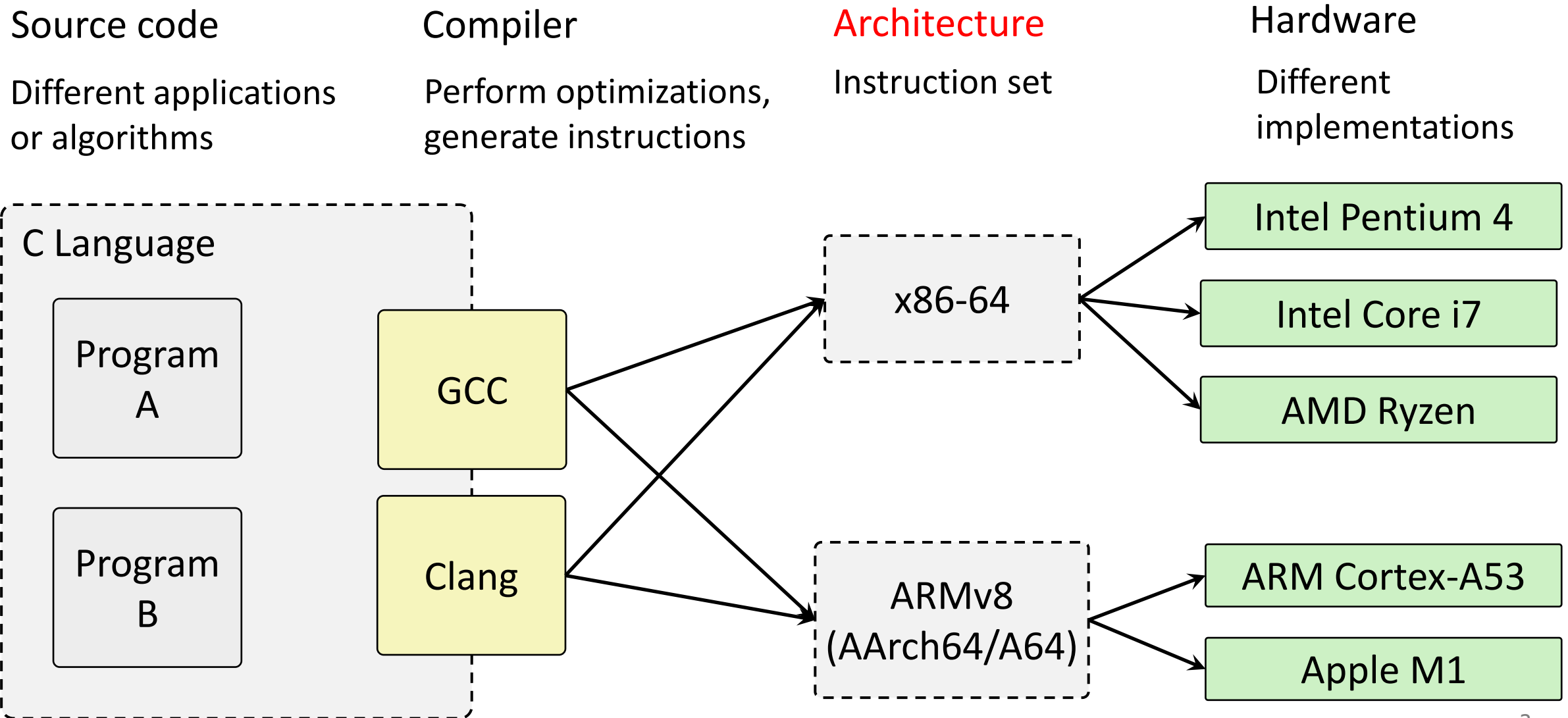
Slides adapted from:

St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

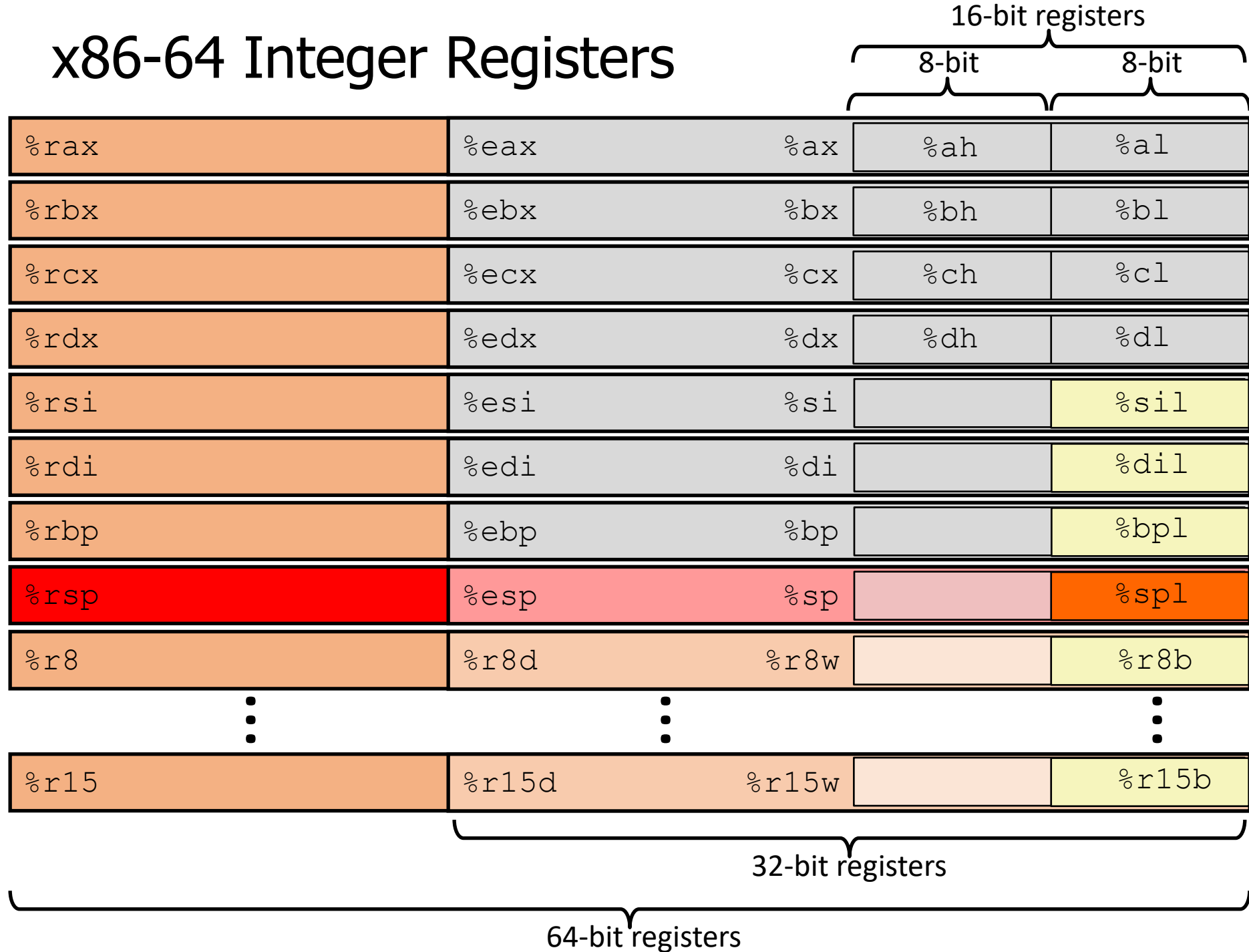
Administrivia

- Data Lab due tonight
 - 10% penalty per late day (or portion thereof)
- Homework 2 released. Bomb Lab released
- Exam next week Thursday (during class)
 - Details on this Thursday
 - There will be an alternate time for students abroad

Instruction Set Architecture sits at software/hardware interface



x86-64 Integer Registers



Operand Combinations

| | Source | Dest | Src, Dest | C Analog |
|------|--------|------|---------------------|----------------|
| movq | Imm | Reg | movq \$0x4, %rax | var_a = 0x4; |
| | | Mem | movq \$-147, (%rax) | *p_a = -147; |
| | Reg | Reg | movq %rax, %rdx | var_d = var_a; |
| | | Mem | movq %rax, (%rdx) | *p_d = var_a; |
| | Mem | Reg | movq (%rax), %rdx | var_d = *p_a; |

Cannot do memory-memory transfer with a single instruction

- **How would you do it?** 1) Mem->Reg, 2) Reg->Mem

Three Basic Kinds of Instructions

1. Transfer data between memory and register

- *Load* data from memory into register
 - `%reg = Mem[address]`
- *Store* register data into memory
 - `Mem[address] = %reg`

Remember: Memory is indexed just like an array of bytes!

2. Perform arithmetic operation on register or memory data

- `c = a + b; z = x << y; i = h & g;`

3. Control flow: what instruction to execute next

- Unconditional jumps to/from procedures
- Conditional branches

In x86-64 these basic types can often be combined

Today's Goals

- Continue exploring x86-64 assembly
 - Arithmetic
- Discuss real-world x86-64
 - Special cases
 - Generating assembly
- Understand condition codes
 - Method for testing Boolean conditions

Outline

- **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 |
|------------------------|---------------------------|--------------|
| <code>addq S,D</code> | $D \leftarrow D + S$ | Add |
| <code>subq S,D</code> | $D \leftarrow D - S$ | Subtract |
| <code>imulq S,D</code> | $D \leftarrow D * S$ | Multiply |
| <code>xorq S,D</code> | $D \leftarrow D \wedge S$ | Exclusive or |
| <code>orq S,D</code> | $D \leftarrow D S$ | Or |
| <code>andq S,D</code> | $D \leftarrow D \& S$ | And |

Operand types

- Immediate
 - Register
 - Memory
- (Only one can be memory)

Shifts

| Instruction | Effect | Description |
|------------------------|------------------------|---|
| <code>sarq k, D</code> | $D \leftarrow D \gg k$ | Shift arithmetic right |
| <code>shrq k, D</code> | $D \leftarrow D \gg k$ | Shift logical right |
| <code>salq k, D</code> | $D \leftarrow D \ll k$ | Shift left |
| <code>shlq k, D</code> | $D \leftarrow D \ll k$ | Shift left (same as <code>salq</code>) |

Be careful with operand order!!!
(Matters for some operations)

A note on instruction names

- Instruction names can look somewhat arcane
 - `shlq?` `movzb1?`



PowerPC Instructions

@ppcinstructions

Follow



rlwbv - Rotate Left Wheel and Buy a Vowel

5:06 PM - 20 Jan 2015

- But, good news: names (usually) follow conventions
 - Common prefixes (**a**dd), suffixes (**b**, **w**, **l**, **q**), etc.
 - So you can understand pieces separately
 - Then combine their meanings

Some Arithmetic Operations

- Unary (one-operand) Instructions:

| Instruction | Effect | Description |
|---------------------|-----------------------|-------------|
| <code>incq D</code> | $D \leftarrow D + 1$ | Increment |
| <code>decq D</code> | $D \leftarrow D - 1$ | Decrement |
| <code>negq D</code> | $D \leftarrow -D$ | Negate |
| <code>notq D</code> | $D \leftarrow \sim D$ | Complement |

- See Section 3.5.5 for more instructions:
`mulq`, `cqto`, `idivq`, `divq`

Converting C to Assembly

- Suppose $a \rightarrow \%rax$, $b \rightarrow \%rbx$, $c \rightarrow \%rcx$
Convert the following C statement to x86-64:

$a = b + c;$

Converting C to Assembly

- Suppose $a \rightarrow \%rax$, $b \rightarrow \%rbx$, $c \rightarrow \%rcx$
Convert the following C statement to x86-64:

$a = b + c;$

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

Converting C to Assembly

- Suppose $a \rightarrow \%rax$, $b \rightarrow \%rbx$, $c \rightarrow \%rcx$
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

- Suppose $a \rightarrow \%rax$, $b \rightarrow \%rbx$, $c \rightarrow \%rcx$
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 slower

Converting C to Assembly

- Suppose $a \rightarrow \%rax$, $b \rightarrow \%rbx$, $c \rightarrow \%rcx$
Convert the following C statement to x86-64:

$a = b + c;$

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

Is this okay?

Converting C to Assembly

- Suppose $a \rightarrow \%rax$, $b \rightarrow \%rbx$, $c \rightarrow \%rcx$
Convert the following C statement to x86-64:

$a = b + c;$

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

Is this okay?

No: overwrites C

Question + Break

Reminder

`addq, src, dst` \rightarrow `dst = dst + src`

- Suppose `a` \rightarrow `%rax`, `b` \rightarrow `%rbx`, `c` \rightarrow `%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
```

Question + Break

Reminder

`addq, src, dst` \rightarrow `dst = dst + src`

- Suppose `a` \rightarrow `%rax`, `b` \rightarrow `%rbx`, `c` \rightarrow `%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]

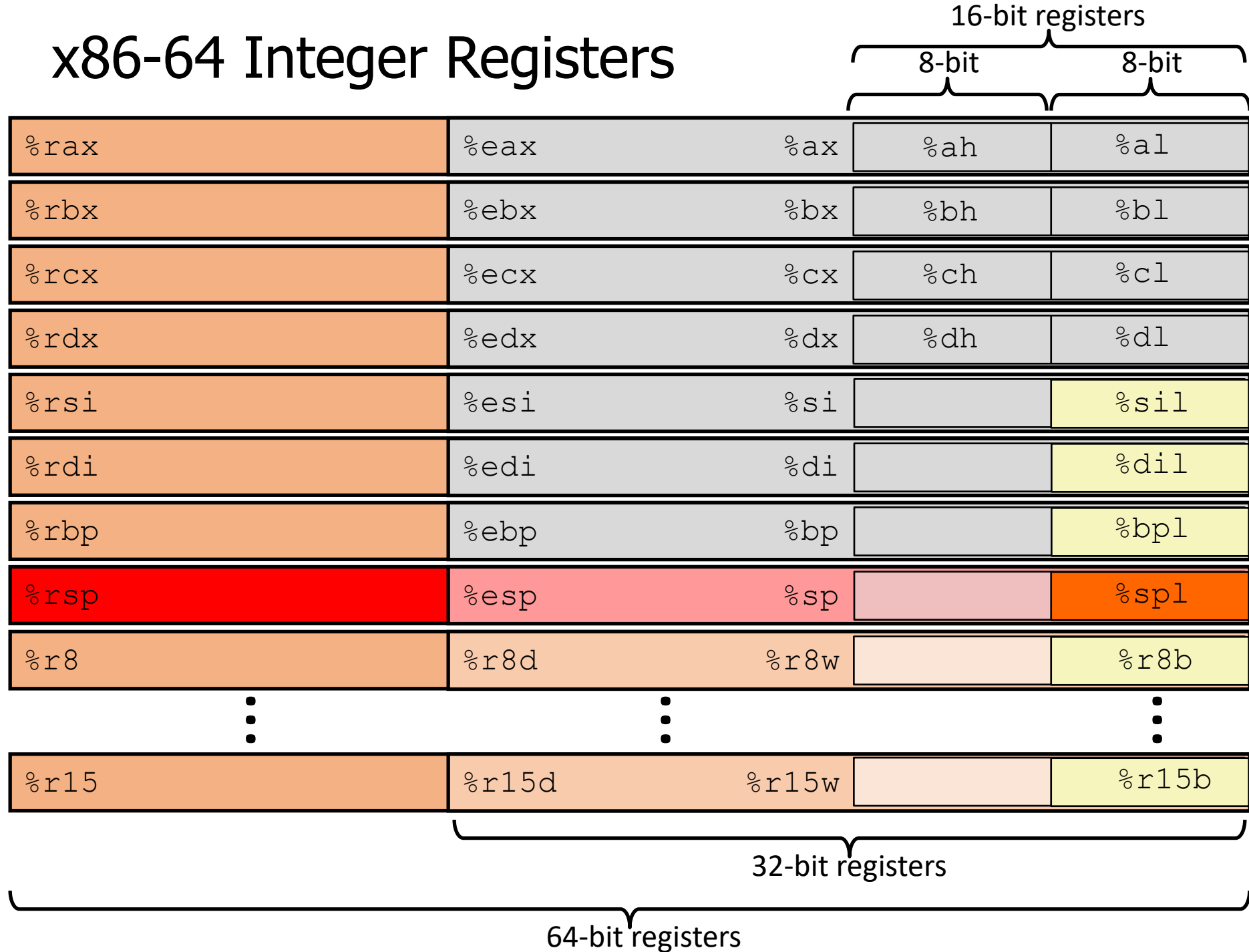
```
subq %rbx, %rax
addq $5, %rax
movq %rax, %rcx
```

Overwrites
`a`

Outline

- 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

- “Vanilla” move can only move between source and dest of the same size
 - Larger → smaller: use the smaller version of registers
 - Smaller → larger: extension! We have two options!

| Instruction | Effect | Description |
|--|---|---|
| <code>movX S,D</code> $X \in \{q, l, w, b\}$ | $D \leftarrow S$ | Copy quad-word (8B), long-word (4B), word (2B) or byte (1B) |
| <code>movsXX S,D</code> $XX \in \{bw, bl, wl, bq, wq, lq\}$ | $D \leftarrow \text{SignExtend}(S)$ | Copy sign-extended byte to word, byte to long-word, etc. |
| <code>movzXX S,D</code> $XX \in \{bw, bl, wl, bq, wq, lq\}$ | $D \leftarrow \text{ZeroExtend}(S)$ | Copy zero-extended byte to word, byte to long-word, etc. |
| <code>c1tq</code> | $\%rax \leftarrow \text{SignExtend}(\%eax)$ | Sign-extend %eax to %rax |

Example: moving byte data

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

`movb %dl, %al` `%eax = 0x987654CD`

`movsbl %dl, %eax` `%eax = 0xFFFFFFFFCD`

`movzbl %dl, %eax` `%eax = 0x000000CD`

32-bit Instruction Peculiarities

- 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 = 0xfffffffffff00
movw $0, %ax                         # rax = 0xfffffffff0000
movl $0, %eax                        # rax = 0x0000000000000000
```

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

Outline

- Arithmetic Instructions
- **Special Cases**
 - Non 64-bit Data
 - **Load Effective Address**
- Condition Codes
- Viewing x86-64 Assembly

Complete Memory Addressing Modes

- **General:**

- $D(Rb, Ri, S)$
 - **Rb**: Base register (any register)
 - **Ri**: Index register (any register except `%rsp`)
 - **S**: Scale factor (1, 2, 4, 8) (sizes of common C types)
 - **D**: Constant displacement value (a.k.a. immediate)

- $Mem[Reg[Rb] + Reg[Ri] * S + D]$

Saving computed addresses

- Generally, any instruction with `()` in it, accesses memory
 - Address is computed first
 - Load if in a source operand
 - Store if in a destination operand
- But what if what you really want is the address?
 - `leal` – load effective address
 - Exception to `()` rule. Does NOT load from memory
 - Also generally useful for arithmetic

Address computation instruction

- **leaq src, dst**
 - "leaq" stands for *load effective address*
 - **src** is 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`

Registers

| | |
|-------------------|--------------------|
| <code>%rax</code> | |
| <code>%rbx</code> | |
| <code>%rcx</code> | <code>0x4</code> |
| <code>%rdx</code> | <code>0x100</code> |
| <code>%rdi</code> | |
| <code>%rsi</code> | |

Memory

| Word Address | |
|--------------------|--------------------|
| <code>0x120</code> | <code>0x400</code> |
| <code>0x118</code> | <code>0xF</code> |
| <code>0x110</code> | <code>0x8</code> |
| <code>0x108</code> | <code>0x10</code> |
| <code>0x100</code> | <code>0x1</code> |

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

Example: `lea` **vs.** `mov`

Registers

| | |
|-------------------|---------------------------|
| <code>%rax</code> | <code>0x110</code> |
| <code>%rbx</code> | <code>0x8</code> |
| <code>%rcx</code> | <code>0x4</code> |
| <code>%rdx</code> | <code>0x100</code> |
| <code>%rdi</code> | |
| <code>%rsi</code> | |

Memory

| Word Address | |
|--------------------|--------------------|
| <code>0x120</code> | <code>0x400</code> |
| <code>0x118</code> | <code>0xF</code> |
| <code>0x110</code> | <code>0x8</code> |
| <code>0x108</code> | <code>0x10</code> |
| <code>0x100</code> | <code>0x1</code> |

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

Example: `lea` **vs.** `mov`

Registers

| | |
|-------------------|---------------------------|
| <code>%rax</code> | <code>0x110</code> |
| <code>%rbx</code> | <code>0x8</code> |
| <code>%rcx</code> | <code>0x4</code> |
| <code>%rdx</code> | <code>0x100</code> |
| <code>%rdi</code> | <code>0x100</code> |
| <code>%rsi</code> | <code>0x1</code> |

Memory

| Word Address | |
|--------------------|--------------------|
| <code>0x120</code> | <code>0x400</code> |
| <code>0x118</code> | <code>0xF</code> |
| <code>0x110</code> | <code>0x8</code> |
| <code>0x108</code> | <code>0x10</code> |
| <code>0x100</code> | <code>0x1</code> |

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

Compiling Arithmetic Operations

```
int arith (long x, long y, long z) {  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    ....  
}
```

```
leaq (%rsi,%rdi),%rcx
```

- 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
```

```
# rcx = x+y (t1)
```


Compiling Arithmetic Operations

```
int arith (long x, long y, long z) {  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    ....  
}
```

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

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

Compiling Arithmetic Operations

```
int arith (long x, long y, long z) {  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    ....  
}
```

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

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

Compiling Arithmetic Operations

```
int arith (long x, long y, long z) {  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    ....  
}
```

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

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

Compiling Arithmetic Operations

```
int arith (long x, long y, long z) {  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    ....  
}
```

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

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

Outline

- 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: **X**
 - 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)

Condition codes

- Control is mediated via *Condition codes*
 - single-bit registers that record answers to questions about values
 - E.g., Is value x greater than value y? Are they equal? Is their sum even?
 - Let's keep "question" abstract for now. We'll see the details in a bit.
- **Terminology:**
 - a bit is ***set*** if it is 1
 - a bit is ***cleared*** (or ***reset***) if it is 0

Conditionals at the machine level

- 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, do the operation (or not)
- Can express Boolean operations, conditionals, loops, etc.
 - We will see the first today, and more control next lecture
- So now we need three things:
 1. Instructions that compare values and set condition codes
 2. Instructions that observe condition codes and do something (or not)
 3. A set of actual condition codes (what questions do we track answers to?)

Two-Step Conditional Process: Bool Ops

- Lots of new pieces
- Lets give an example first, then learn more about each
 - Translate C code on right into assembly

```
char 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: Bool Ops

- Step 1, **cmpq**: compare quad words
 - *compare* the values in `%rsi` and `%rdi`, keep track of *all* you can learn, and set the relevant condition codes
- Are the two equal? Set the condition codes that records they were equal
- Was the right one greater?
- Etc. We don't know yet which answer we are going to need!

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

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

The diagram shows assembly code with two callout boxes. One box labeled 'y' points to the `%rsi` register in the `cmpq` instruction. Another box labeled 'x' points to the `%rdi` register in the same `cmpq` instruction.

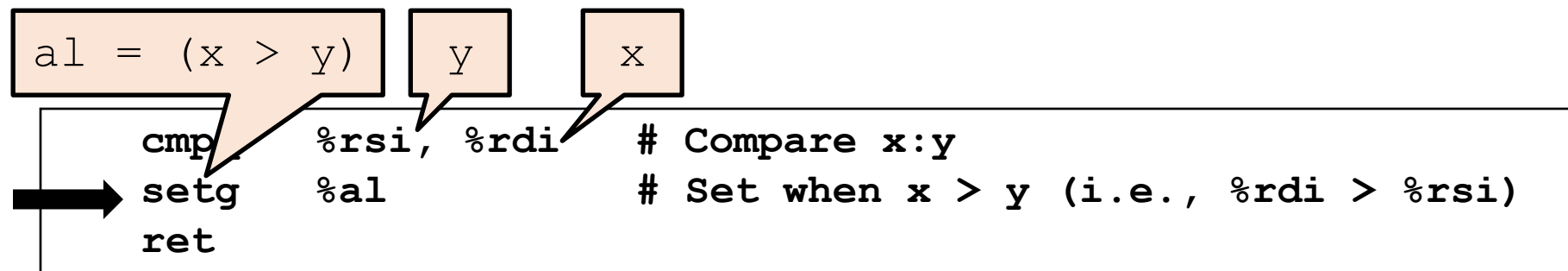
```
→ cmpq    %rsi, %rdi    # Compare x:y
   setg    %al          # Set when x > y (i.e., %rdi > %rsi)
   ret
```

Two-Step Conditional Process: Bool Ops

- 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!)
 - There's also **setl** for less than, etc.
 - Reads the condition codes that encodes the answer to that question
 - Set the 1-byte register **%a1** to 1 if true

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

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



Step 1: Setting condition codes

- Analogy: Asking ALL the possible questions at once
 - And recording the answers
 - We don't know yet which question is the one we care about!
- Done in one of two ways
 - **Implicitly**: all* arithmetic instructions set (and reset) condition codes in addition to producing a result
 - * except `leaq`; it's not "officially" an arithmetic instruction
 - **Explicitly**: by instructions whose sole purpose is to set condition codes
 - E.g., `cmprq`
 - They don't actually produce results (in registers or memory)
- Condition codes are left unchanged by other operations

Implicitly Setting Condition Codes

- Condition codes on x86
 - **CF** Carry Flag (for unsigned)
 - **ZF** Zero Flag
 - **PF** Parity Flag
 - **SF** Sign Flag (for signed)
 - **OF** Overflow Flag (for signed)
- Not an arbitrary set! By combining them, can keep track of answers to many useful questions! (We'll see exactly which in a bit.)

Implicitly Setting Condition Codes

CF (Carry) **SF** (Sign) **ZF** (Zero) **OF** (Overflow) **PF** (Parity)

- Set (or reset) based on the result of arithmetic operations

Example: `addq Src, Dest` # C-analog: `t = a+b`

- **ZF set** if `t == 0`
- **SF set** if `t < 0` (as signed)
- **CF set** if carry out from most significant bit (unsigned overflow)
also CF takes the value of the last bit shifted (left or right)
- **OF set** if twos-complement (signed) overflow (pos/neg overflow)
`(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
also, set if a 1-bit shift operation changes the sign of the result
- **PF set** if `t` has an even number of 1 bits

Explicitly Setting Condition Codes: Compare

- `cmp{b,w,l,q} Src2, Src1`
- `cmpq b, a` computes $t = a - b$, then throws away the result!
 - And sets condition codes along the way, like `subq` would!
 - Follows the rules we saw on the previous slide for arithmetic instructions!
 - **Beware the order of the `cmp` operands!**
- More directly
 - **ZF set** if $a == b$
 - **SF set** if $(a - b) < 0$ (as signed), i.e., $b > a$ in a signed comparison!
 - **CF and OF** used mostly in combinations with others (see in a few slides)

Explicitly Setting Condition Codes: Test

- `test{b,w,l,q} Src2,Src1`
- `testq b,a` computes `t = a&b`, then throws away the result!
 - And sets condition codes like `andq` would (order doesn't matter here)
 - So again, same rules as arithmetic instructions
- More directly
 - **ZF set** when `a&b == 0`, i.e., `a` and `b` have no bits in common
 - **SF set** when `a&b < 0`
- 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

Step 2: Reading Condition Codes

- Cannot read condition codes directly; instead observe via instructions
 - And generally observe ***combinations*** of condition codes, not individual ones
- Example: the **setX** family of instructions
 - Write single-byte destination register based on combinations of condition codes
 - **set{e, ne, s, ...} D** where D is a 1-byte register
 - Example: **sete %a1**
 - means: %a1=1 if flag ZF is set, %a1=0 otherwise

Condition codes combinations

| SetX | Description | Condition |
|--------------|---------------------------|--------------------------------------|
| sete | Equal / Zero | ZF |
| setne | Not Equal / Not Zero | $\sim ZF$ |
| sets | Negative | SF |
| setns | Nonnegative | $\sim SF$ |
| setg | Greater (Signed) | $\sim (SF \wedge OF) \ \& \ \sim ZF$ |
| setge | Greater or Equal (Signed) | $\sim (SF \wedge OF)$ |
| setl | Less (Signed) | $(SF \wedge OF)$ |
| setle | Less or Equal (Signed) | $(SF \wedge OF) \ \ ZF$ |
| seta | Above (unsigned) | $\sim CF \ \& \ \sim ZF$ |
| 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.

Step 2: Reading Condition Codes

- `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)
- Usually you'll see a `cmpX` (or `testX`, or arithmetic) right before
 - But not always, so know what to do in general

Question + Break

- `%rax = 15, %rbx = 15`

```
cmpq %rax, %rbx
```

- Which flag(s) are set?

CF (Carry) **SF** (Sign) **ZF** (Zero) **OF** (Overflow) **PF** (Parity)

Question + Break

- `%rax = 15, %rbx = 15`

```
cmpq %rax, %rbx
```

- Which flag(s) are set?

CF (Carry) **SF** (Sign) **ZF** (Zero) **OF** (Overflow) **PF** (Parity)

- **ZF is set** (because the two are equal and subtracted)
- **PF is set** (because there are an even number of 1 bits, four total)

Outline

- 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`
 - Online compiler, shows asm output: <https://godbolt.org>
 - **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`
 - Within the gdb Debugger

```
linux> gdb prog
(gdb) disassemble sum
```

 - gdb tutorial coming soon!
 - **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 example!

Godbolt

Ignore
“_dl_relocate_static_pie”

Play around with this to
try stuff on your own

<https://godbolt.org/>

The screenshot displays the Godbolt Compiler Explorer interface. The left pane shows the C source code with three functions: `square`, `add_inputs_plus_two`, and `check_greater`. The right pane shows the assembly output for the `square` function, which is located at address `401050`. The assembly code includes instructions for `_dl_relocate_static_pie`, `endbr64`, `retq`, and `add_inputs_plus_two`. The assembly output is for the `square` function, which is located at address `401050`. The assembly code includes instructions for `_dl_relocate_static_pie`, `endbr64`, `retq`, and `add_inputs_plus_two`.

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
5
6
7 int add_inputs_plus_two(int a, int b) {
8     return a+b+2;
9 }
10
11
12 char check_greater(int a, int b) {
13     return (a > b);
14 }
```

Assembly output for `square`:

```
401050 _dl_relocate_static_pie:
      f3 0f 1e fa
      endbr64
      c3
      retq
401054 66 2e 0f 1f 84 00 00 00 00 00
401055 nopw    %cs:0x0(%rax,%rax,1)
      90
40105f nop
      square:
      0f af ff
401102 imul    %edi,%edi
      89 f8
401105 mov     %edi,%eax
      c3
401107 retq
      add_inputs_plus_two:
      8d 44 37 02
401108 lea     0x2(%rdi,%rsi,1),%eax
      c3
40110c retq
      check_greater:
      39 f7
40110d cmp     %esi,%edi
      0f 9f c0
40110f setg    %al
      c3
401112 retq
```

Outline

- Arithmetic Instructions
- Special Cases
 - Non 64-bit Data
 - Load Effective Address
- Condition Codes
- Viewing x86-64 Assembly