

Lecture 06

Arithmetic Instructions

CS213 – Intro to Computer Systems
Branden Ghen a – Winter 2025

Slides adapted from:

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

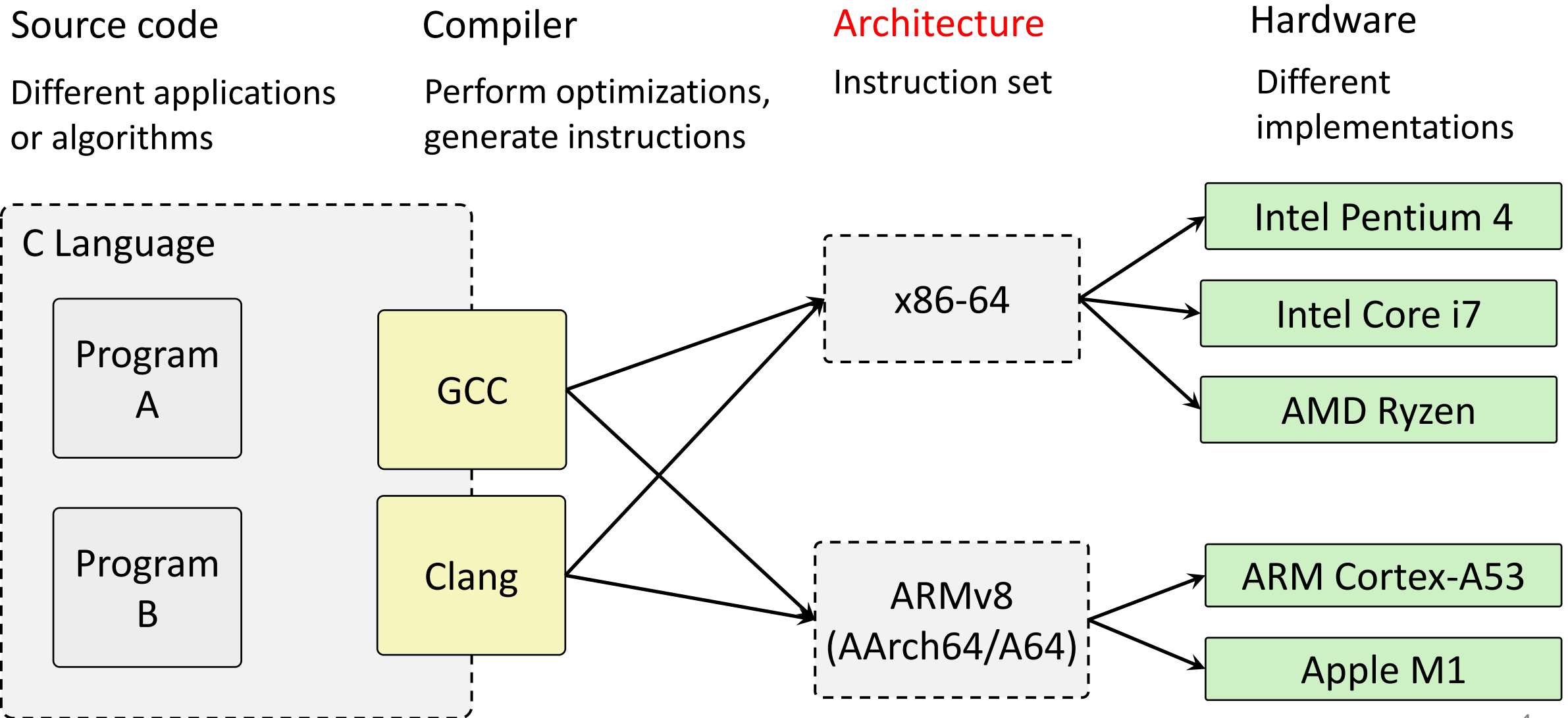
Administrivia

- Pack Lab due Tuesday by midnight
 - Expect office hours to start getting very busy
- Homework 2
 - Releases later today
 - Covers material through today's lecture
 - Due in 1 week (Thursday, January 30th)
- Bomb Lab
 - Comes out next week
 - Partnership survey sometime this weekend

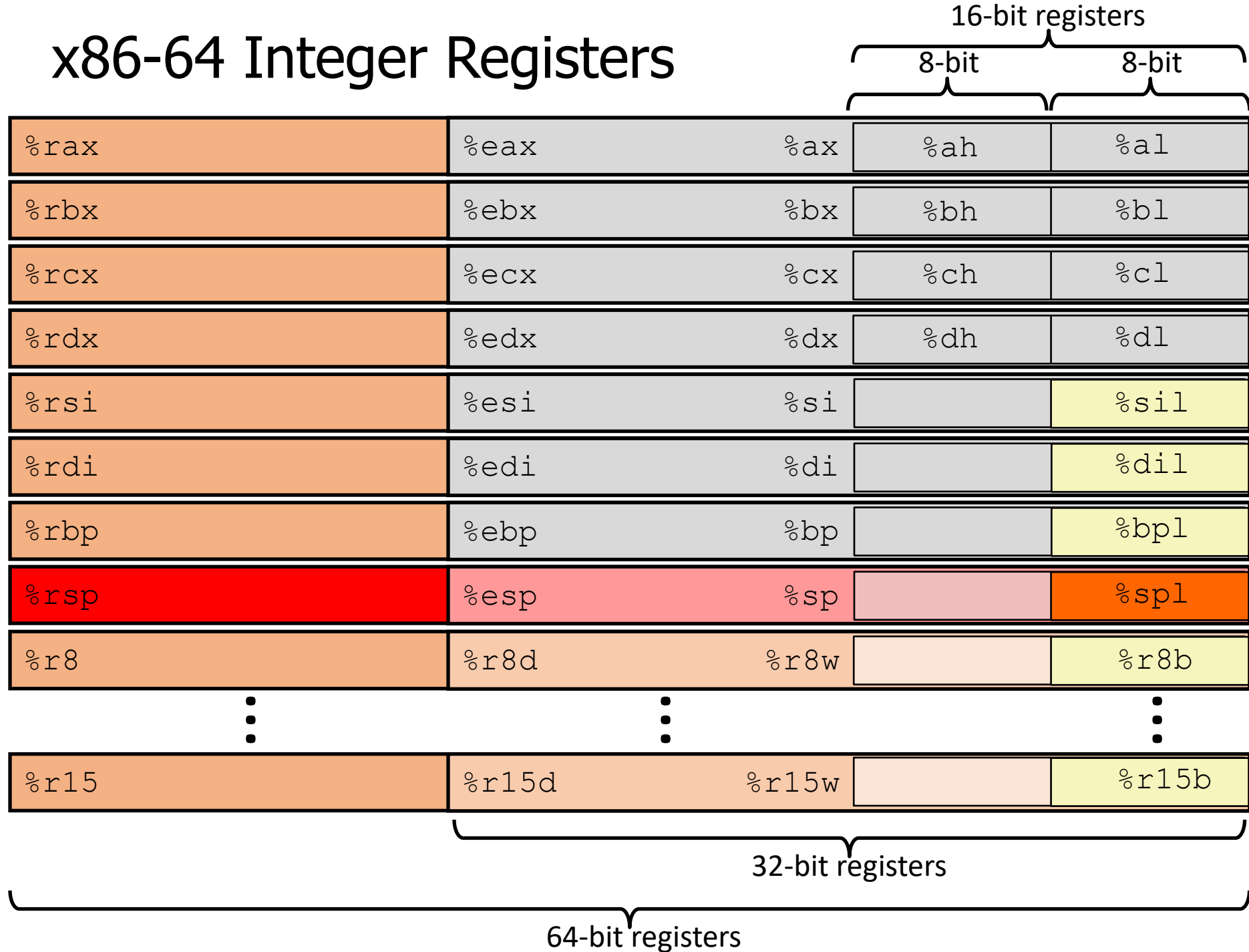
Administrivia

- Midterm exam
 - Exactly two weeks from today (February 6th)
 - Taken here in the classroom
 - Details next week

Instruction Set Architecture sits at software/hardware interface



x86-64 Integer Registers



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

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

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

`rlwbv` - Rotate Left Wheel and Buy a Vowel

The youth still get
Wheel of Fortune
jokes, right?

- But, good news: names (usually) follow conventions
 - Common prefixes (`add`), 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 textbook Section 3.5.5 for more instructions:
`mulq`, `cqto`, `idivq`, `divq`

Converting C to Assembly

op src, dst

- 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

op src, dst

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

op src, dst

- 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

op src, dst

- 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

op src, dst

- 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?

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

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]

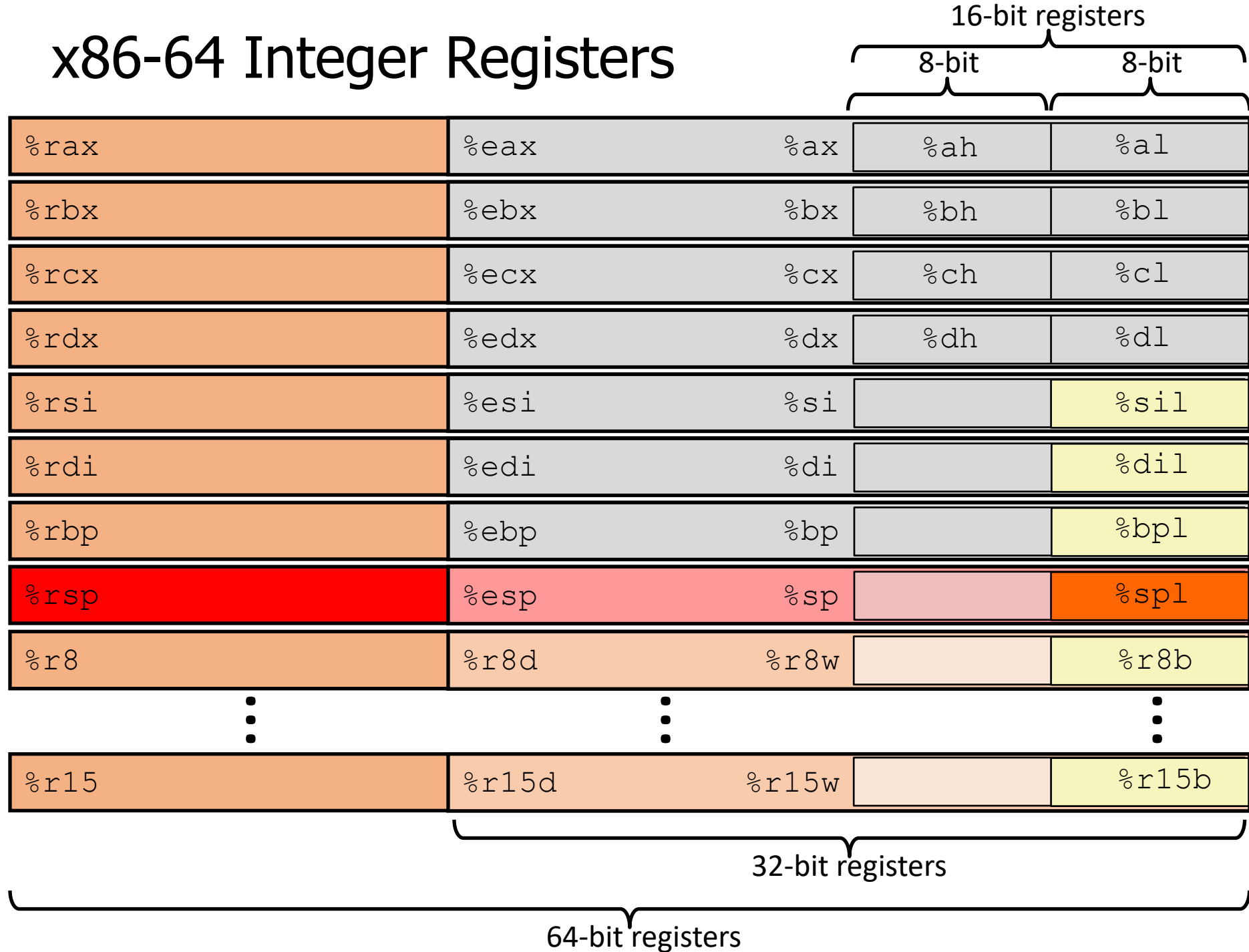
```
subq %rbx, %rax
addq $5, %rax
movq %rax, %rcx
```

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: zero-extend or sign-extend

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

Example: moving byte data

op src, dst

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

`movb %dl, %al` `%eax = 0x987654``CD`

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

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

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 = 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
 - 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

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`

Registers

<code>%rax</code>	
<code>%rbx</code>	
<code>%rcx</code>	<code>0x2</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>	0x108
<code>%rbx</code>	
<code>%rcx</code>	0x2
<code>%rdx</code>	0x100
<code>%rdi</code>	
<code>%rsi</code>	

Memory

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

```
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>0x108</code>
<code>%rbx</code>	<code>0x10</code>
<code>%rcx</code>	<code>0x2</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>	0x108
<code>%rbx</code>	0x10
<code>%rcx</code>	0x2
<code>%rdx</code>	0x100
<code>%rdi</code>	0x100
<code>%rsi</code>	

Memory

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

```
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>0x108</code>
<code>%rbx</code>	<code>0x10</code>
<code>%rcx</code>	<code>0x2</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
```

Why does the compiler love `leaq`?

- 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
 - `leaq` adds up to two registers and an immediate, AND stores to a different register!

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;  
    ....  
}
```

- 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

```
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)
```

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

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

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

Operation	Address Loaded	%rcx Value
movq (%rax, %rbx), %rcx	0x2020	0xE3

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

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

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

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

Break + Say hi to your neighbors

- Things to share
 - Name
 - Major
 - One of the following
 - Favorite Candy
 - Favorite Pokemon
 - Favorite Emoji

Break + Say hi to your neighbors

- Things to share
 - Name -Branden
 - Major -Electrical and Computer Engineering, and Computer Science
 - One of the following
 - Favorite Candy - Twix
 - Favorite Pokemon - Eevee
 - Favorite Emoji - 🍷

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?

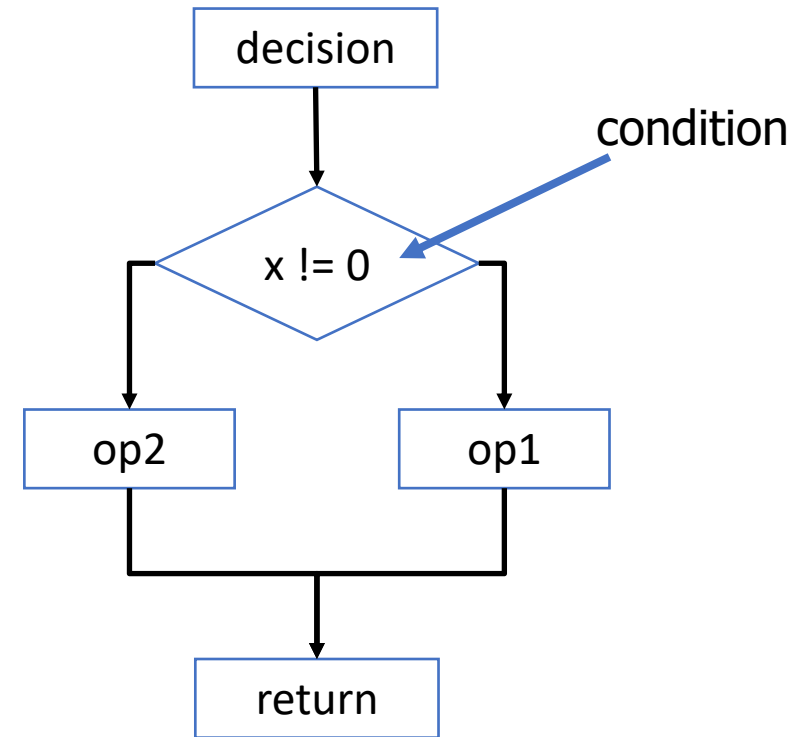
```
if (x > y)
    result = x-y;
else
    result = y-x;
```

```
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)

Control relies on “conditions”

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



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 even 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

- Result is: 00000000000000...000000000000

- Sign Flag

- Result is: 1xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxx

How are condition codes set?

- Overflow Flag

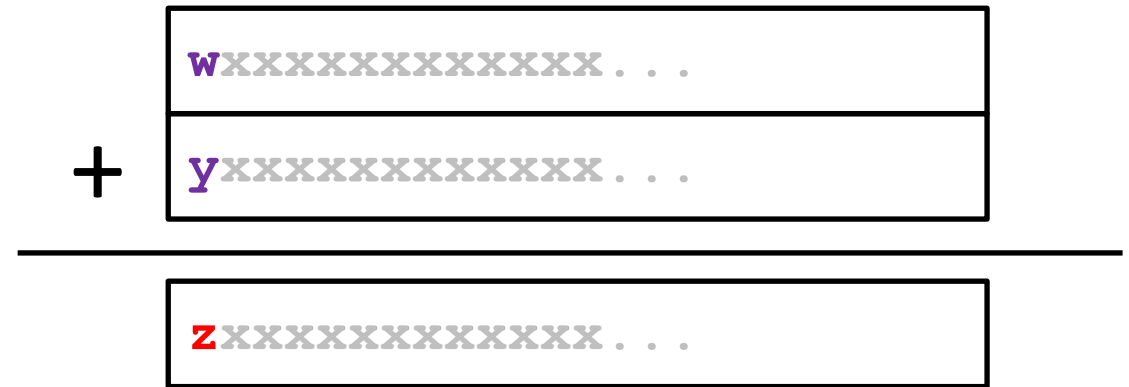
Either:

- Both numbers positive but result is negative

Or:

- Both numbers negative but result is positive

- Detects overflow for two's complement numbers

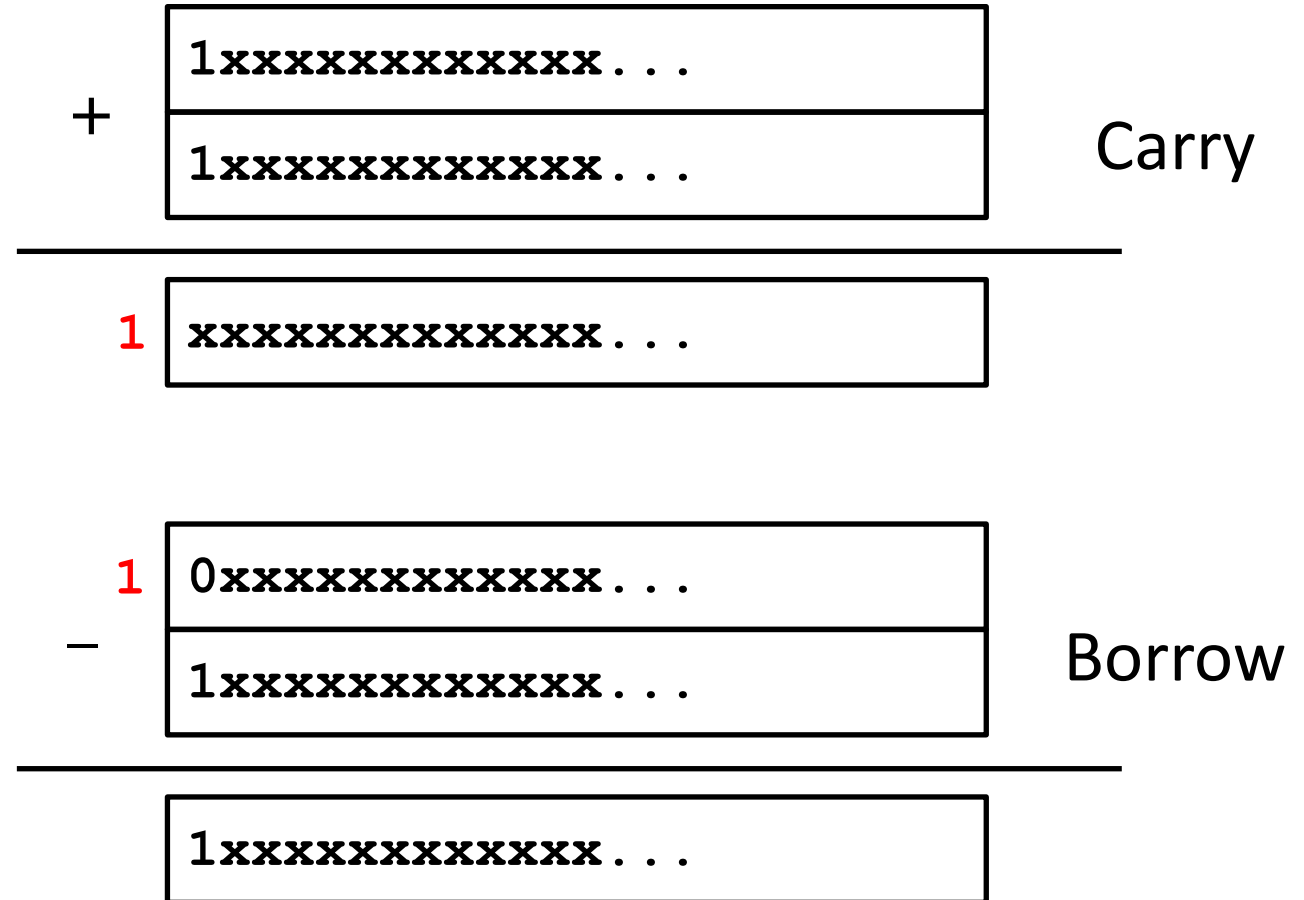


$$w == y \ \&\& \ w != z$$

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	$\sim ZF$
Negative	SF
Nonnegative	$\sim SF$
Greater (Signed)	$\sim (SF \wedge OF) \ \& \ \sim ZF$
Greater or Equal (Signed)	$\sim (SF \wedge OF)$
Less (Signed)	$(SF \wedge OF)$
Less or Equal (Signed)	$(SF \wedge OF) \ \ ZF$
Above (unsigned)	$\sim CF \ \& \ \sim ZF$
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
- 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)

Explicitly Setting Condition Codes: Compare

- `cmp{b,w,l,q} Src2, Src1`
- `cmpq src, dst` computes `dst-src`, then throws away the result
 - And 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 statement conditions or 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!
 - And 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
 - Not a different instruction, just the arithmetic instructions we learned earlier
 - They do still update destination register
 - But also, they update the condition codes based on the result
 - We rarely care about this, but it matters occasionally
- *Exception: the `leaq` instruction does not set condition codes
 - It's not "officially" an arithmetic instruction

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 %a1** means: **%a1=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: $(SF \wedge OF) \mid ZF$
 - 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!
 - Think of as `dst X 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

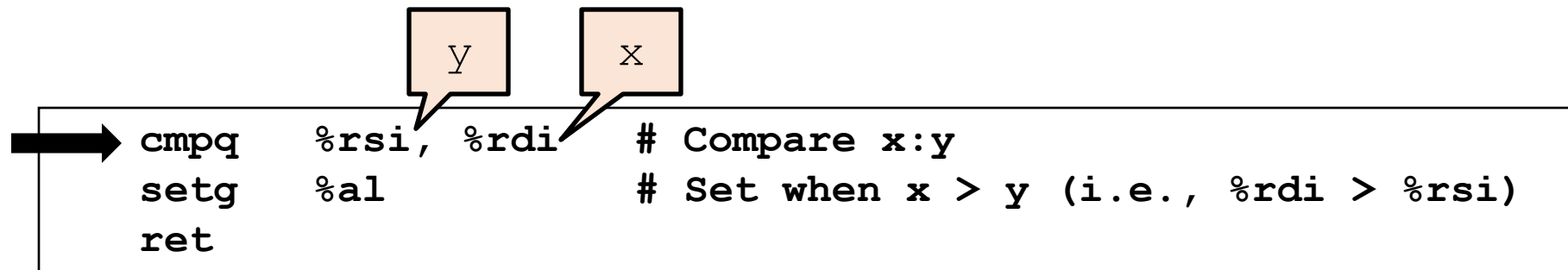
```
cmpq    %rsi, %rdi    # Compare x:y
setg     %al           # Set when x > y (i.e., %rdi > %rsi)
ret
```

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



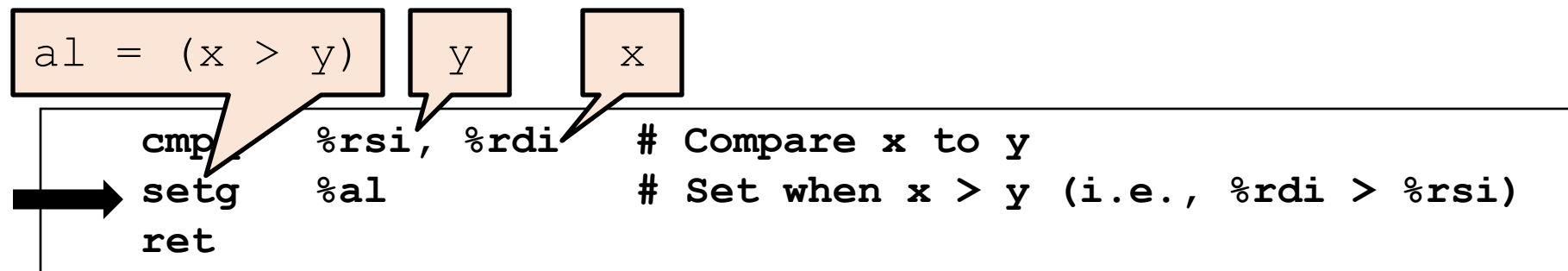
```
→ cmpq    %rsi, %rdi    # Compare x:y
   setg    %al          # Set when x > y (i.e., %rdi > %rsi)
   ret
```

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 **%a1** 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

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`
- **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

Ignore section labeled:
“_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 for three functions: `square`, `add_inputs_plus_two`, and `check_greater`. The right pane shows the corresponding x86-64 assembly output for gcc 10.2 with optimization level -O1. The assembly includes labels for `_dl_relocate_static_pie:`, `square:`, `add_inputs_plus_two:`, and `check_greater:`. The code is compiled for x86-64 using gcc 10.2, and the output is filtered to show only the relevant assembly instructions.

```
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 (x86-64 gcc 10.2, -O1):

```
_dl_relocate_static_pie:
f3 0f 1e fa
401050 endbr64
c3
401054 retq
66 2e 0f 1f 84 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
```

- Godbolt example!

Outline

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