Lecture 06
Arithmetic Instructions

CS213 – Intro to Computer Systems
Branden Ghena – Fall 2023

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

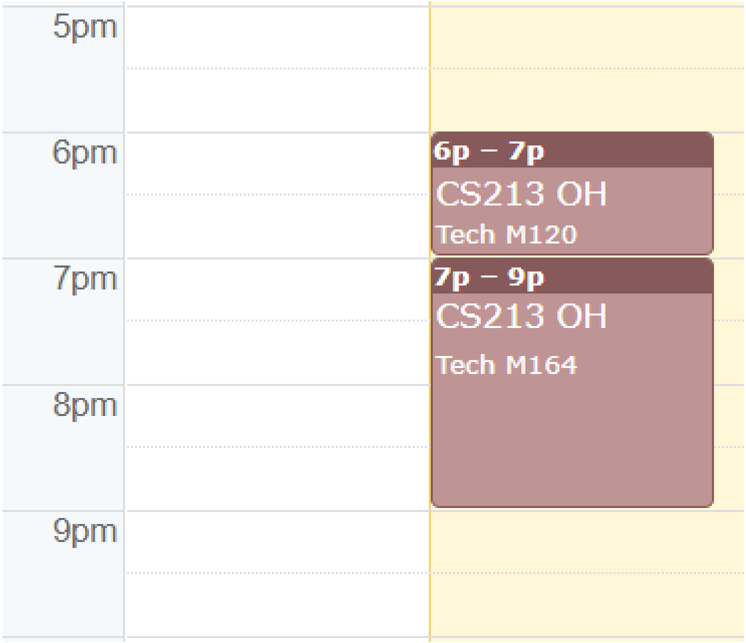
Administrivia

- Pack Lab due tonight by midnight
  - Warning: office hours today are going to be **very** full
  - Slip days (3 total) start to apply after the deadline
    - You don’t have to ask, we’ll use them automatically as best helps you

- Bomb Lab releases later today
  - Practice interpreting assembly code
  - Due after the midterm exam
    - But we strongly recommend you start it early as assembly practice
  - Partnership survey on Piazza

One more (hopefully last) office hour change

- Starting today!
  - Tuesdays 6-7 has moved to Tech M120
  - Tuesdays 7-9 is still in Tech M164



Administrivia

- Midterm exam in 1.5 weeks
  - Class time next week Thursday (Oct 19th)

  - Covers everything from the start of class through Control Flow
    - Does NOT cover function calls in assembly ("Procedures" lecture)

  - Bring a pencil and one 8.5"x11" inch paper with notes
    - Notes can be on both sides, handwritten or typed
  - No calculators

  - Practice exams (and solutions) are on the Canvas home page
  - Also good practice: Homework 2 (due Tuesday), phases 1-3 of Bomb Lab

- ANU students: I'll reach out this week with details

Instruction Set Architecture sits at software/hardware interface

Source code

Different applications or algorithms

Compiler

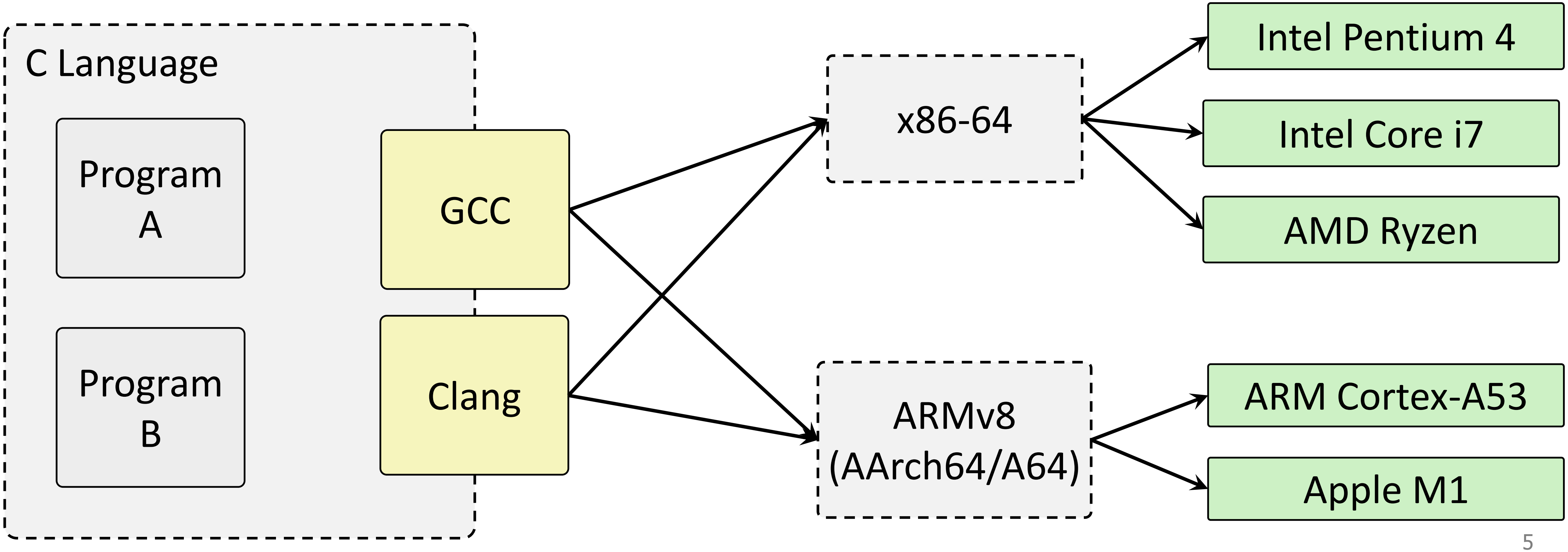
Perform optimizations, generate instructions

Architecture

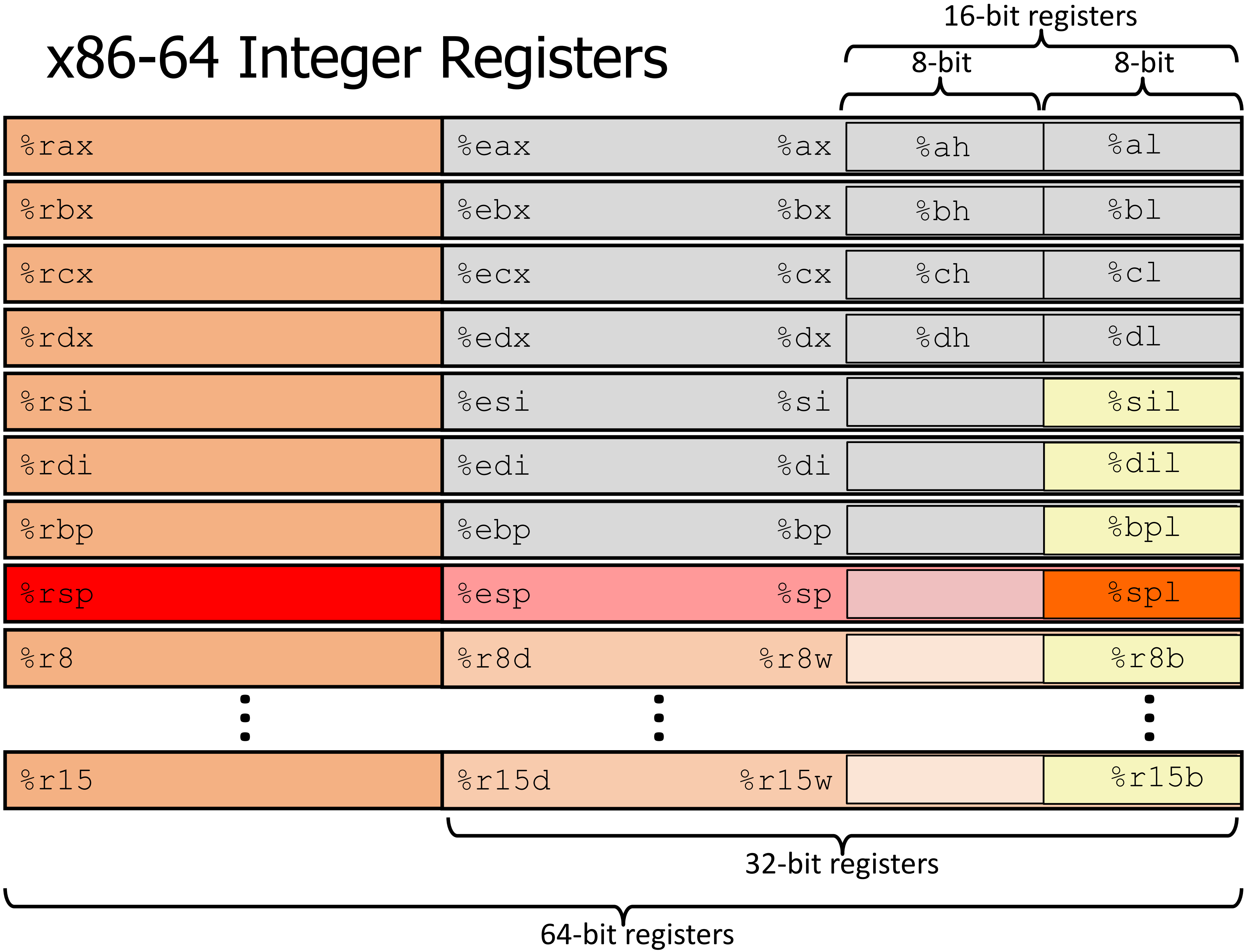
Instruction set

Hardware

Different implementations



x86-64 Integer Registers



%rax	%eax	%ax	%ah	%al
%rbx	%ebx	%bx	%bh	%bl
%rcx	%ecx	%cx	%ch	%cl
%rdx	%edx	%dx	%dh	%dl
%rsi	%esi	%si		%sil
%rdi	%edi	%di		%dil
%rbp	%ebp	%bp		%bpl
%rsp	%esp	%sp		%spl
%r8	%r8d	%r8w		%r8b
⋮	⋮			⋮
%r15	%r15d	%r15w		%r15b

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
`addq` S, D	D ← D + S	Add
`subq` S, D	D ← D – S	Substract
`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
- Register
- Memory

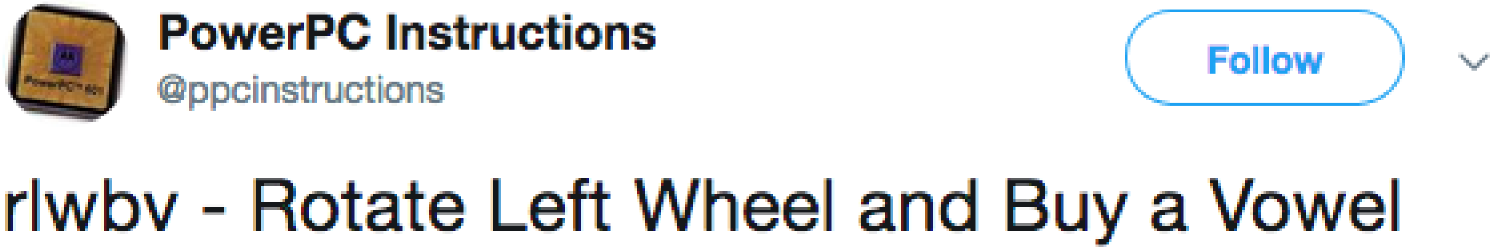
(Only one can be memory)

Be careful with operand order!!!

(Matters for some operations)

A note on instruction names

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



PowerPC Instructions
@ppcinstructions

rlwbv - Rotate Left Wheel and Buy a Vowel

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

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

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 slower

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?

No: overwrites C which could still be used later in code!

Question + Break

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
```

Question + Break

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

Outline

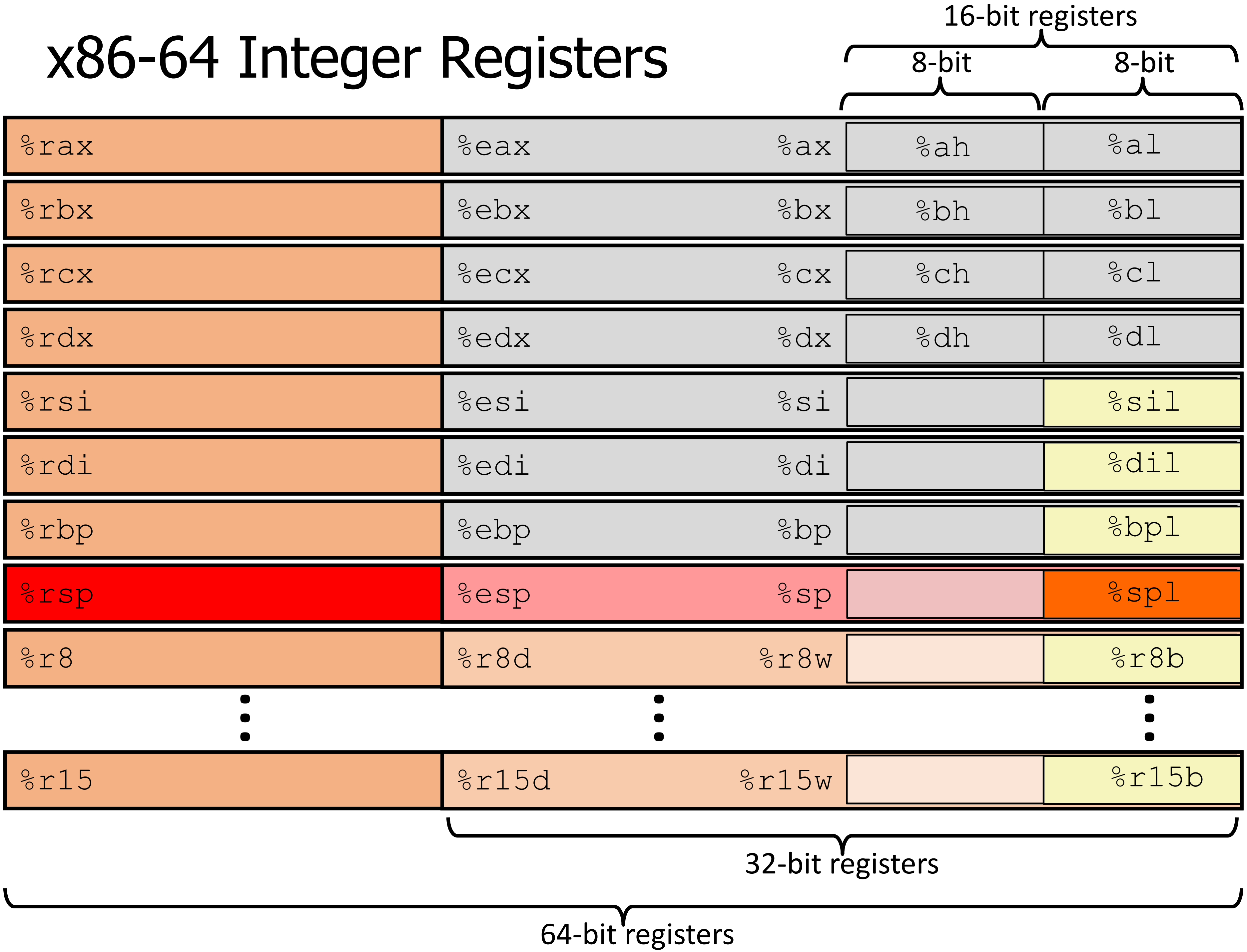
- Arithmetic Instructions

- **Special Cases**
    - **Non 64-bit Data**
    - Load Effective Address

- Condition Codes

- Viewing x86-64 Assembly

x86-64 Integer Registers



%rax	%eax	%ax	%ah	%al
%rbx	%ebx	%bx	%bh	%bl
%rcx	%ecx	%cx	%ch	%cl
%rdx	%edx	%dx	%dh	%dl
%rsi	%esi	%si		%sil
%rdi	%edi	%di		%dil
%rbp	%ebp	%bp		%bpl
%rsp	%esp	%sp		%spl
%r8	%r8d	%r8w		%r8b
⋮	⋮			⋮
%r15	%r15d	%r15w		%r15b

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

Example: moving byte data

`op src, dst`

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

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

movsbl %dl,%eax      %eax = 0xFFFFFFCD

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 = 0xffffffffffffff00
movw $0, %ax                       # rax = 0xffffffffffff0000
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?
    - **`lea`** – load effective address

    - Exception to `()` rule. Does NOT load from 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

%rax	
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

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

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

Example: `lea` vs. `mov`

Registers

%rax	**0x110**
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

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

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

Example: `lea` vs. `mov`

Registers

`%rax`	**`0x110`**
`%rbx`	**`0x8`**
`%rcx`	`0x4`
`%rdx`	`0x100`
`%rdi`	
`%rsi`	

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

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

Example: `lea` vs. `mov`

Registers

%rax	**0x110**
%rbx	**0x8**
%rcx	0x4
%rdx	0x100
%rdi	**0x100**
%rsi	

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

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

Example: `lea` vs. `mov`

Registers

`%rax`	**0x110**
`%rbx`	**0x8**
`%rcx`	0x4
`%rdx`	0x100
`%rdi`	**0x100**
`%rsi`	**0x1**

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

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

Why does the compiler love `lea`?

- Sometimes it’s good for computing addresses

- 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;
  ....
}
```

- 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
leaq (%rsi,%rdi),%rcx     # 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;
  ....
}
```

- 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
leaq (%rsi,%rdi),%rcx     # rcx = x+y   (t1)
leaq (%rsi,%rsi,2),%rsi   # rsi = y + 2*y = 3*y
salq $4,%rsi              # 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;
  ....
}
```

- 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
leaq (%rsi,%rdi),%rcx    # rcx = x+y   (t1)
leaq (%rsi,%rsi,2),%rsi  # rsi = y + 2*y = 3*y
salq $4,%rsi             # rsi = (3*y)*16 = 48*y (t4)
addq %rdx,%rcx           # 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;
  ....
}
```

- 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
leaq (%rsi,%rdi),%rcx    # rcx = x+y   (t1)
leaq (%rsi,%rsi,2),%rsi  # rsi = y + 2*y = 3*y
salq $4,%rsi             # rsi = (3*y)*16 = 48*y (t4)
addq %rdx,%rcx           # rcx = z+t1 (t2)
leaq 4(%rsi,%rdi),%rdi   # 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;
   ....
}
```

- 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
leaq (%rsi,%rdi),%rcx     # rcx = x+y   (t1)
leaq (%rsi,%rsi,2),%rsi   # rsi = y + 2*y = 3*y
salq $4,%rsi              # rsi = (3*y)*16 = 48*y (t4)
addq %rdx,%rcx            # rcx = z+t1 (t2)
leaq 4(%rsi,%rdi),%rdi    # rdi = t4+x+4 (t5)
imulq %rcx,%rdi           # 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: ✗
  - 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: Boolean Operations

- Lots of new pieces
- Lets give an example first, then learn more about each
  - Translate C code on right into assembly
  - We'll do this in the next steps

```
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`**, 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? 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

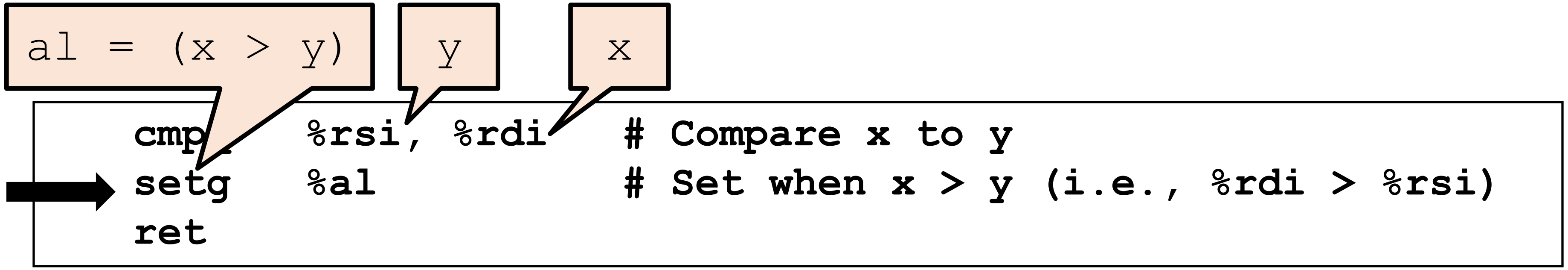
```
cmpq    %rsi, %rdi     # Compare x:y
setg    %al            # Set when x > y (i.e., %rdi > %rsi)
ret
```

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!)
    - There's also **`setl`** for less than, etc.
  - 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



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

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 **`lea`**; it’s not “officially” an arithmetic instruction

    - **Explicitly**: by instructions whose sole purpose is to set condition codes
        - E.g., **`cmpq`**
        - They don’t actually produce results (in registers or memory)

    - Condition codes are left unchanged by other operations (such as **`mov`**)

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 encoding)
  - **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!**

- Use cases
  - **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

- Use cases
  - **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 %al`**
      - means: **`%al`**=1 if flag ZF is set, **`%al`**=0 otherwise

Using condition codes for comparison

- 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

Condition codes combinations

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`**

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

What do you need to know?

- 90%+ of the time
  - `cmp` instruction followed by `setX` instruction (or a branch, 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

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

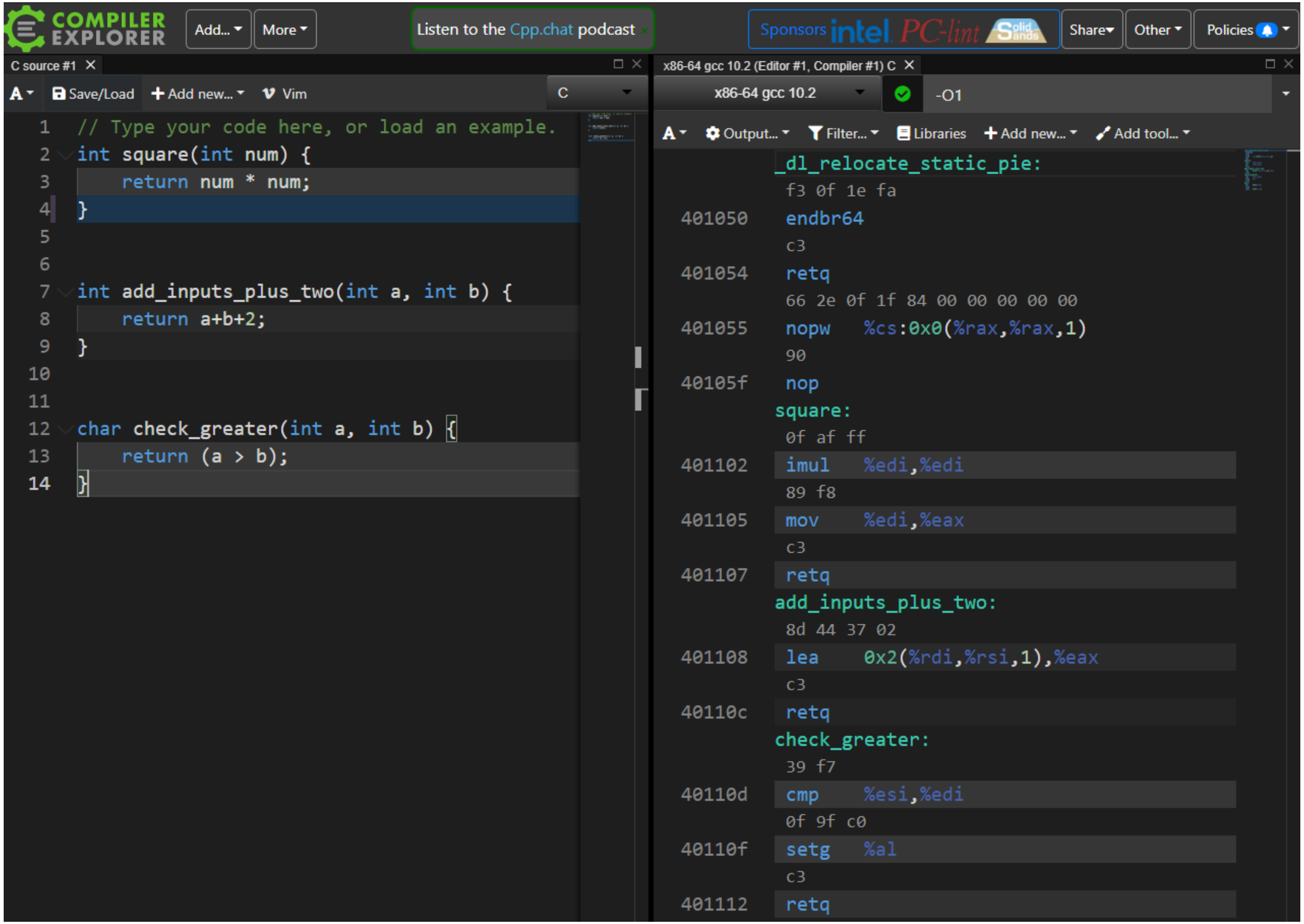
```
0000000000400595 <sum>:
  400595:   53                 push   %rbx
  400596:   48 89 d3           mov    %rdx,%rbx
  400599:   e8 f2 ff ff ff     callq  400590 <plus>
  40059e:   48 89 03           mov    %rax,(%rbx)
  4005a1:   5b                 pop    %rbx
  4005a2:   c3                 retq
```

Godbolt

Ignore section labeled:
“_dl_relocate_static_pie”

Play around with this to
try stuff on your own

https://godbolt.org/



```c
// Type your code here, or load an example.
int square(int num) {
    return num * num;
}


int add_inputs_plus_two(int a, int b) {
    return a+b+2;
}


char check_greater(int a, int b) {
    return (a > b);
}
```

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

- Godbolt example!

Outline

- Arithmetic Instructions

- Special Cases
  - Non 64-bit Data
  - Load Effective Address

- Condition Codes

- Viewing x86-64 Assembly