

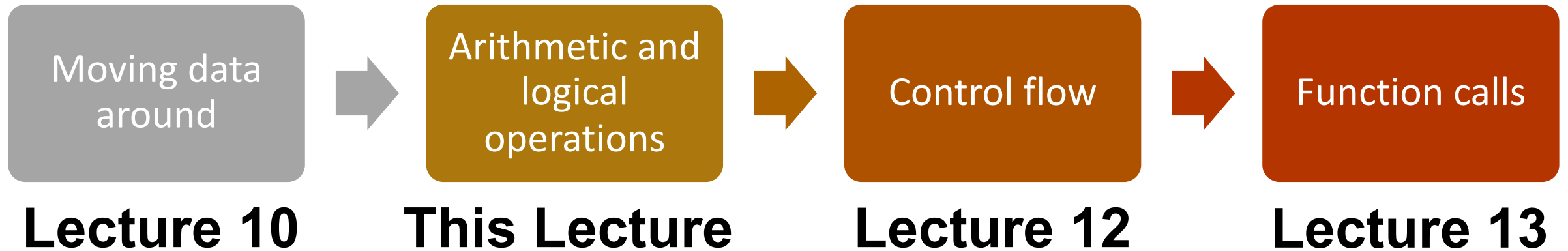
CS107, Lecture 11

Assembly: Arithmetic and Logic

Reading: B&O 3.5-3.6

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

Learning Assembly



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

Learning Goals

- Learn how to perform arithmetic and logical operations in assembly
- Begin to learn how to read assembly and understand the C code that generated it

Lecture Plan

- **Recap: mov** so far 7
- Data and Register Sizes 11
- The **lea** Instruction 24
- Logical and Arithmetic Operations 30
- Practice: Reverse Engineering 38

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

Helpful Assembly Resources

- **Course textbook** (reminder: see relevant readings for each lecture on the Schedule page, <http://cs107.stanford.edu/schedule.html>)
- **CS107 Assembly Reference Sheet:** <http://cs107.stanford.edu/resources/x86-64-reference.pdf>
- **CS107 Guide to x86-64:** <http://cs107.stanford.edu/guide/x86-64.html>

Lecture Plan

- **Recap: mov so far** 7
- Data and Register Sizes 11
- The **lea** Instruction 24
- Logical and Arithmetic Operations 30
- Practice: Reverse Engineering 38

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

mov

The **mov** instruction copies bytes from one place to another; it is similar to the assignment operator (=) in C.

mov **src, dst**

The **src** and **dst** can each be one of:

- Immediate (constant value, like a number) (*only src*)
- Register
- Memory Location
(*at most one of src, dst*)

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no \$)
(%rax)	What's in %rax
4(%rax)	What's in %rax, plus 4
(%rax, %rdx)	Sum of what's in %rax and %rdx
4(%rax, %rdx)	Sum of values in %rax and %rdx, plus 4
(, %rcx, 4)	What's in %rcx, times 4 (multiplier can be 1, 2, 4, 8)
(%rax, %rcx, 2)	What's in %rax, plus 2 times what's in %rcx
8(%rax, %rcx, 2)	What's in %rax, plus 2 times what's in %rcx, plus 8

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 from the book: “Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.”

Lecture Plan

- **Recap: mov** so far 7
- **Data and Register Sizes** 11
- The **lea** Instruction 24
- Logical and Arithmetic Operations 30
- Practice: Reverse Engineering 38

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

Data Sizes

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

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

Assembly instructions can have suffixes to refer to these sizes:

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

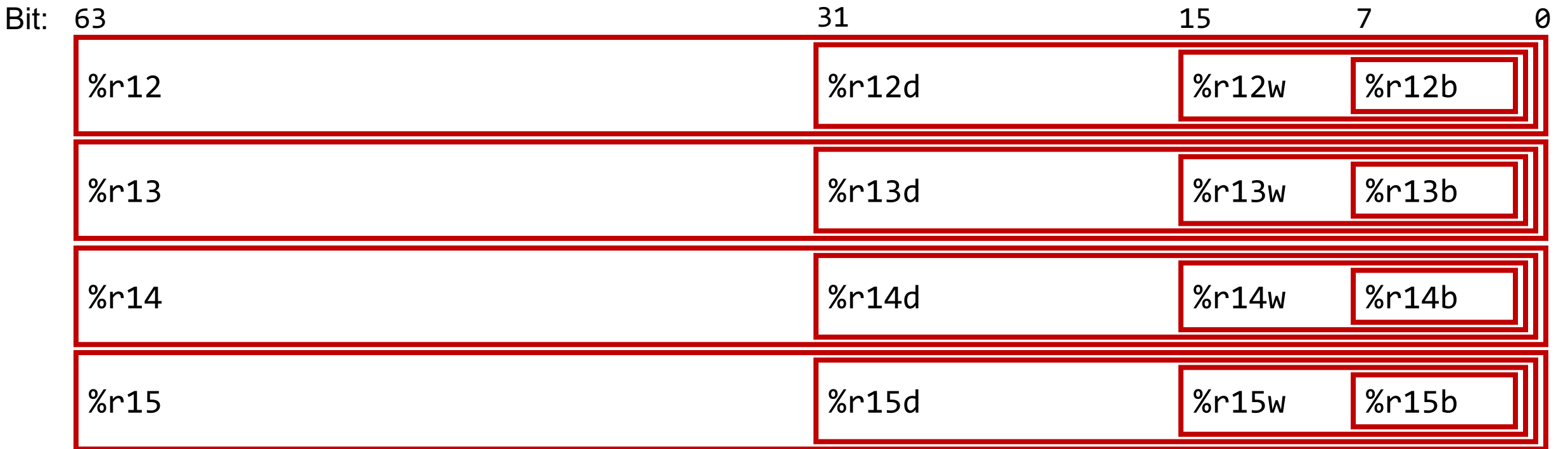
Register Sizes

Bit:	63	31	15	7	0
%rax	%eax		%ax	%al	
%rbx	%ebx		%bx	%bl	
%rcx	%ecx		%cx	%cl	
%rdx	%edx		%dx	%dl	
%rsi	%esi		%si	%sil	
%rdi	%edi		%di	%dil	

Register Sizes

Bit:	63	31	15	7	0
%rbp	%ebp		%bp	%bpl	
%rsp	%esp		%sp	%spl	
%r8	%r8d		%r8w	%r8b	
%r9	%r9d		%r9w	%r9b	
%r10	%r10d		%r10w	%r10b	
%r11	%r11d		%r11w	%r11b	

Register Sizes



Register Responsibilities

Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

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

mov Variants

- **mov** can take an optional suffix (b,w,l,q) that specifies the size of data to move:
movb, movw, movl, movq
- **mov** only updates the specific register bytes or memory locations indicated.
 - **Exception: movl** writing to a register will also set high order 4 bytes to 0.

Practice: mov And Data Sizes

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. **movb**, **movw**, **movl** or **movq**).

1. mov__ %eax, (%rsp)
2. mov__ (%rax), %dx
3. mov__ \$0xff, %bl
4. mov__ (%rsp,%rdx,4),%dl
5. mov__ (%rdx), %rax
6. mov__ %dx, (%rax)

Practice: mov And Data Sizes

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. **movb**, **movw**, **movl** or **movq**).

1. `movl %eax, (%rsp)`
2. `movw (%rax), %dx`
3. `movb $0xff, %bl`
4. `movb (%rsp,%rdx,4),%dl`
5. `movq (%rdx), %rax`
6. `movw %dx, (%rax)`

mov

- The **movabsq** instruction is used to write a 64-bit Immediate (constant) value.
- The regular **movq** instruction can only take 32-bit immediates.
- 64-bit immediate as source, only register as destination.

```
movabsq $0x0011223344556677, %rax
```

movz and movs

- There are two mov instructions that can be used to copy a smaller source to a larger destination: **movz** and **movs**.
- **movz** fills the remaining bytes with zeros
- **movs** fills the remaining bytes by sign-extending the most significant bit in the source.
- The source must be from memory or a register, and the destination is a register.

movz and movs

MOVZ S, R

$R \leftarrow \text{ZeroExtend}(S)$

Instruction	Description
movzbw	Move zero-extended byte to word
movzbl	Move zero-extended byte to double word
movzwl	Move zero-extended word to double word
movzbq	Move zero-extended byte to quad word
movzwq	Move zero-extended word to quad word

movz and movs

MOVS S, R

$R \leftarrow \text{SignExtend}(S)$

Instruction	Description
movsbw	Move sign-extended byte to word
movsbl	Move sign-extended byte to double word
movswl	Move sign-extended word to double word
movsbq	Move sign-extended byte to quad word
movswq	Move sign-extended word to quad word
movslq	Move sign-extended double word to quad word
cltq	Sign-extend %eax to %rax $\%rax \leftarrow \text{SignExtend}(\%eax)$

Lecture Plan

- **Recap: mov** so far 7
- Data and Register Sizes 11
- **The lea Instruction** 24
- Logical and Arithmetic Operations 30
- Practice: Reverse Engineering 38

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

lea

The **lea** instruction copies an “effective address” from one place to another.

lea **src, dst**

Unlike **mov**, which copies data at the address **src** to the destination, **lea** copies the value of **src** *itself* to the destination.

The syntax for the destinations is the same as **mov**. The difference is how it handles the **src**.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
6(%rax), %rdx	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
6(%rax), %rdx	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
(%rax, %rcx), %rdx	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
6(%rax), %rdx	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
(%rax, %rcx), %rdx	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
(%rax, %rcx, 4), %rdx	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
<code>6(%rax), %rdx</code>	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
<code>(%rax, %rcx), %rdx</code>	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
<code>(%rax, %rcx, 4), %rdx</code>	Go to the address ($\%rax + 4 * \%rcx$) and copy data there into %rdx.	Copy ($\%rax + 4 * \%rcx$) into %rdx.
<code>7(%rax, %rax, 8), %rdx</code>	Go to the address ($7 + \%rax + 8 * \%rax$) and copy data there into %rdx.	Copy ($7 + \%rax + 8 * \%rax$) into %rdx.

Unlike **mov**, which copies data at the address **src** to the destination, **lea** copies the value of **src** *itself* to the destination.

Lecture Plan

- **Recap: mov** so far 7
- Data and Register Sizes 11
- The **lea** Instruction 24
- **Logical and Arithmetic Operations** 30
- Practice: Reverse Engineering 38

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

Unary Instructions

The following instructions operate on a single operand (register or memory):

Instruction	Effect	Description
<code>inc D</code>	$D \leftarrow D + 1$	Increment
<code>dec D</code>	$D \leftarrow D - 1$	Decrement
<code>neg D</code>	$D \leftarrow -D$	Negate
<code>not D</code>	$D \leftarrow \sim D$	Complement

Examples:

```
incq 16(%rax)
```

```
dec %rdx
```

```
not %rcx
```

Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations. Read it as, e.g. “Subtract S from D”:

Instruction	Effect	Description
add S, D	$D \leftarrow D + S$	Add
sub S, D	$D \leftarrow D - S$	Subtract
imul S, D	$D \leftarrow D * S$	Multiply
xor S, D	$D \leftarrow D \wedge S$	Exclusive-or
or S, D	$D \leftarrow D \mid S$	Or
and S, D	$D \leftarrow D \& S$	And

Examples:

```
addq %rcx, (%rax)
```

```
xorq $16, (%rax, %rdx, 8)
```

```
subq %rdx, 8(%rax)
```


Large Multiplication

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

$$\text{imul } S, D \quad D \leftarrow D * S$$

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

Instruction	Effect	Description
<code>imulq S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Signed full multiply
<code>mulq S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Unsigned full multiply

Division and Remainder

Instruction	Effect	Description
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide

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

Division and Remainder

Instruction	Effect	Description
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide
<code>cqto</code>	$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word

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

Shift Instructions

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

Instruction	Effect	Description
<code>sal k, D</code>	$D \leftarrow D \ll k$	Left shift
<code>shl k, D</code>	$D \leftarrow D \ll k$	Left shift (same as <code>sal</code>)
<code>sar k, D</code>	$D \leftarrow D \gg_A k$	Arithmetic right shift
<code>shr k, D</code>	$D \leftarrow D \gg_L k$	Logical right shift

Examples:

```
shll $3, (%rax)
```

```
shrl %cl, (%rax, %rdx, 8)
```

```
sarl $4, 8(%rax)
```

Shift Amount

Instruction	Effect	Description
<code>sar k, D</code>	$D \leftarrow D \lll k$	Arithmetic right shift
<code>shr k, D</code>	$D \leftarrow D \ggg k$	Logical right shift
<code>shl k, D</code>	$D \leftarrow D \lll k$	Left shift (same as <code>sar</code>)
<code>sal k, D</code>	$D \leftarrow D \lll k$	Left shift

- When using **%cl**, the width of what you are shifting determines what portion of **%cl** is used.
- For **w** bits of data, it looks at the low-order **log₂(w)** bits of **%cl** to know how much to shift.
 - If **%cl** = 0xff, then: **shlb** shifts by 7 because it considers only the low-order $\log_2(8) = 3$ bits, which represent 7. **shlw** shifts by 15 because it considers only the low-order $\log_2(16) = 4$ bits, which represent 15.

Lecture Plan

- **Recap: mov** so far 7
- Data and Register Sizes 11
- The **lea** Instruction 24
- Logical and Arithmetic Operations 30
- **Practice: Reverse Engineering** 38

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

Assembly Exploration

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

Code Reference: add_to_first

```
// Returns the sum of x and the first element in arr
int add_to_first(int x, int arr[]) {
    int sum = x;
    sum += arr[0];
    return sum;
}
```

```
add_to_first:
    movl %edi, %eax
    addl (%rsi), %eax
    ret
```


Code Reference: full_divide

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

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

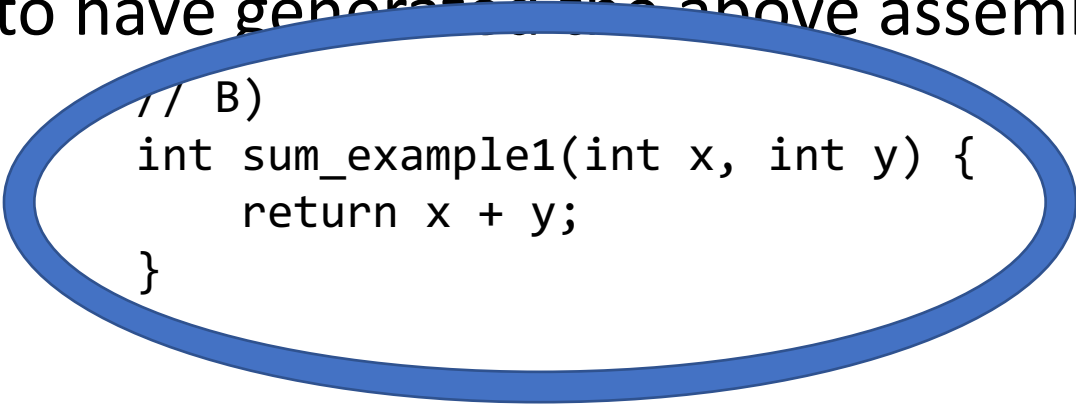
Assembly Exercise 1

```
00000000004005ac <sum_example1>:  
    4005bd:  8b 45 e8      mov  %esi,%eax  
    4005c3:  01 d0        add  %edi,%eax  
    4005cc:  c3          retq
```

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

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

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



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

Assembly Exercise 2

```
0000000000400578 <sum_example2>:  
    400578:    8b 47 0c          mov    0xc(%rdi),%eax  
    40057b:    03 07           add   (%rdi),%eax  
    40057d:    2b 47 18        sub   0x18(%rdi),%eax  
    400580:    c3             retq
```

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

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

%eax

Assembly Exercise 3

```
0000000000400578 <sum_example2>:  
  400578:  8b 47 0c          mov  0xc(%rdi),%eax  
  40057b:  03 07            add  (%rdi),%eax  
  40057d:  2b 47 18        sub  0x18(%rdi),%eax  
  400580:  c3              retq
```

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

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

0x18

Our First Assembly

```
int sum_array(int arr[], int nelems) {  
    int sum = 0;  
    for (int i = 0; i < nelems; i++) {  
        sum += arr[i];  
    }  
    return sum;  
}
```

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

00000000004005b6 <sum_array>:

4005b6:	ba 00 00 00 00	mov	\$0x0,%edx
4005bb:	b8 00 00 00 00	mov	\$0x0,%eax
4005c0:	eb 09	jmp	4005cb <sum_array+0x15>
4005c2:	48 63 ca	movslq	%edx,%rcx
4005c5:	03 04 8f	add	(%rdi,%rcx,4),%eax
4005c8:	83 c2 01	add	\$0x1,%edx
4005cb:	39 f2	cmp	%esi,%edx
4005cd:	7c f3	j1	4005c2 <sum_array+0xc>
4005cf:	f3 c3	repz	retq



Recap

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

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

Live Session Slides

Post any questions you have to the lecture thread on the discussion forum for today's lecture!

Assembly reference sheet: <http://cs107.stanford.edu/resources/x86-64-reference.pdf>

Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss
- **15 minutes:** open Q&A
- **25 minutes:** extra practice

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

A Note About Operand Forms

- Many instructions share the same address operand forms that **mov** uses.
 - Eg. `7(%rax, %rcx, 2)`.
- These forms work the same way for other instructions, e.g. **sub**:
 - `sub 8(%rax,%rdx),%rcx` -> Go to $8 + \%rax + \%rdx$, subtract what's there from `%rcx`
- The exception is **lea**:
 - It interprets this form as just the calculation, *not the dereferencing*
 - `lea 8(%rax,%rdx),%rcx` -> Calculate $8 + \%rax + \%rdx$, put it in `%rcx`

Shift Amount

Instruction	Effect	Description
<code>sar k, D</code>	$D \leftarrow D \lll k$	Arithmetic right shift
<code>shr k, D</code>	$D \leftarrow D \ggg k$	Logical right shift
<code>shl k, D</code>	$D \leftarrow D \ll k$	Left shift (same as <code>sal</code>)
<code>sal k, D</code>	$D \leftarrow D \ll k$	Left shift

- When using **%cl**, the width of what you are shifting determines what portion of **%cl** is used.
- For **w** bits of data, it looks at the low-order **log₂(w)** bits of **%cl** to know how much to shift.
 - If **%cl** = 0xff, then: **shlb** shifts by 7 because it considers only the low-order $\log_2(8) = 3$ bits, which represent 7. **shlw** shifts by 15 because it considers only the low-order $\log_2(16) = 4$ bits, which represent 15.

Division and Remainder

Instruction	Effect	Description
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide

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

Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss
- **15 minutes:** open Q&A
- **25 minutes:** extra practice

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

Extra Practice

<https://godbolt.org/z/hGKPWszq4>

Reverse Engineering 1

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

```
add_to:  
    movslq %edx, %rdx  
    movl %edi, %eax  
    addl (%rsi,%rdx,4), %eax  
    ret
```

Reverse Engineering 1

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

```
// x in %edi, arr in %rsi, i in %edx
```

```
add_to:
```

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

Reverse Engineering 1

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

```
// x in %edi, arr in %rsi, i in %edx
```

```
add_to:
```

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


Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {  
    int z = nums[___?___] * ___?___;  
    z -= ___?___;  
    z >>= ___?___;  
    return ___?___;  
}
```

```
elem_arithmetic:  
    movl %esi, %eax  
    imull (%rdi), %eax  
    subl 4(%rdi), %eax  
    sarl $2, %eax  
    addl $2, %eax  
    ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
```

// nums in %rdi, y in %esi

elem_arithmetic:

```
    movl %esi, %eax           // copy y into %eax
    imull (%rdi), %eax        // multiply %eax by nums[0]
    subl 4(%rdi), %eax        // subtract nums[1] from %eax
    sarl $2, %eax             // shift %eax right by 2
    addl $2, %eax             // add 2 to %eax
    ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {  
    int z = nums[0] * y;  
    z -= nums[1];  
    z >>= 2;  
    return z + 2;  
}
```

```
-----  
// nums in %rdi, y in %esi
```

```
elem_arithmetic:
```

```
    movl %esi, %eax           // copy y into %eax  
    imull (%rdi), %eax       // multiply %eax by nums[0]  
    subl 4(%rdi), %eax       // subtract nums[1] from %eax  
    sarl $2, %eax           // shift %eax right by 2  
    addl $2, %eax           // add 2 to %eax  
    ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {  
    *ptr = ____?____ + 1;  
    long result = x % ____?____;  
    return ____?____;  
}
```

```
func:  
    movq %rdi, %rax  
    leaq 1(%rdi), %rcx  
    movq %rcx, (%rsi)  
    cqto  
    idivq %rcx  
    movq %rdx, %rax  
    ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {  
    *ptr = ____?____ + 1;  
    long result = x % ____?____;  
    return ____?____;  
}
```

```
// x in %rdi, ptr in %rsi
```

```
func:
```

```
    movq %rdi, %rax           // copy x into %rax  
    leaq 1(%rdi), %rcx       // put x + 1 into %rcx  
    movq %rcx, (%rsi)       // copy %rcx into *ptr  
    cqto                    // sign-extend x into %rdx  
    idivq %rcx              // calculate x / (x + 1)  
    movq %rdx, %rax         // copy the remainder into %rax  
    ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {  
    *ptr = x + 1;  
    long result = x % *ptr; // or x + 1  
    return result;  
}
```

```
// x in %rdi, ptr in %rsi
```

```
func:
```

```
    movq %rdi, %rax           // copy x into %rax  
    leaq 1(%rdi), %rcx       // put x + 1 into %rcx  
    movq %rcx, (%rsi)       // copy %rcx into *ptr  
    cqto                    // sign-extend x into %rdx  
    idivq %rcx              // calculate x / (x + 1)  
    movq %rdx, %rax         // copy the remainder into %rax  
    ret
```

Side Note: Old GCC Output

```
long func(long x, long *ptr) {  
    *ptr = x + 1;  
    long result = x % *ptr; // or x + 1  
    return result;  
}
```

```
// x in %rdi, ptr in %rsi
```

```
func:
```

```
    leaq 1(%rdi), %rcx    // put x + 1 into %rcx  
    movq %rcx, (%rsi)    // copy %rcx into *ptr  
    movq %rdi, %rax      // copy x into %rax  
    cqto                 // sign-extend x into %rdx  
    idivq %rcx           // calculate x / (x + 1)  
    movq %rdx, %rax      // copy the remainder into %rax  
    ret
```