CS107, Lecture 16
Assembly: Arithmetic and Logic

Reading: B&O 3.5-3.6

CS107 Topic 5

How does a computer interpret and execute C programs?

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- We can learn how to reverse engineer and exploit programs at the assembly level

assign5: find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.

Learning Assembly

Moving data around → Arithmetic and logical operations → Control flow → Function calls

Last Lecture **This Lecture**

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 Calendar page, http://cs107.stanford.edu/calendar.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

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

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

Lecture Plan

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

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

We calculate this value and *then* go to that address.

Practice #2: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value *0x11* is stored at address *0x10C*, *0xAB* is stored at address *0x104*, *0x100* is stored in register %rax and *0x3* is stored in %rdx.

```
1. mov    $0x42,(%rax)
2. mov    4(%rax),%rcx
3. mov    9(%rax,%rdx),%rcx
```

Imm(r_b, r_i) is equivalent to address **Imm + R[r_b] + R[r_i]**

Displacement: positive or negative constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**4 times** the value in register %rdx) into some destination.*

```
mov    (,%rdx,4),________
```

```
mov    ________,(,%rdx,4)
```

The *scaling factor* (e.g. 4 here) must be hardcoded to be either 1, 2, 4 or 8.

*Copy the value from some source into the memory at the address which is (**4 times** the value in register %rdx).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (4 times the value in register %rdx, **plus 0x4**), into some destination.*

```
mov      0x4(,%rdx,4),______
```

```
mov      ______,0x4(,%rdx,4)
```

*Copy the value from some source into the memory at the address which is (4 times the value in register %rdx, **plus 0x4**).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**the value in register %rax** plus 2 times the value in register %rdx) into some destination.*

```
mov        (%rax,%rdx,2),________
```

```
mov        _________,(%rax,%rdx,2)
```

*Copy the value from some source into the memory at the address which is (**the value in register %rax** plus 2 times the value in register %rdx).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**<u>0x4 plus</u>** the value in register %rax plus 2 times the value in register %rdx) into some destination.*

```
mov     0x4(%rax,%rdx,2),______

mov     ______,0x4(%rax,%rdx,2)
```

*Copy the value from some source into the memory at the address which is (**<u>0x4 plus</u>** the value in register %rax plus 2 times the value in register %rdx).*

Most General Operand Form

$\texttt{Imm}(r_b, r_i, s)$

is equivalent to…

$\texttt{Imm} + R[r_b] + R[r_i]*s$

Most General Operand Form

$\texttt{Imm(}r_b\texttt{, }r_i\texttt{, s)}$ is equivalent to address $\texttt{Imm + R[}r_b\texttt{] + R[}r_i\texttt{]*s}$

Displacement: pos/neg constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Scale must be 1,2,4, or 8 (if missing, = 1)

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."

Practice #3: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value *0x1* is stored in register %rcx, the value *0x100* is stored in register %rax, the value *0x3* is stored in register %rdx, and value *0x11* is stored at address *0x10C*.

For #2, respond with your thoughts on PollEv: pollev.com/cs107 or text CS107 to 22333 once to join.

```
1. mov    $0x42,0xfc(,%rcx,4)

2. mov    (%rax,%rdx,4),%rbx
```

Imm(r_b, r_i, s) is equivalent to address **Imm + R[r_b] + R[r_i]*s**

Displacement | Base | Index | Scale (1,2,4,8)

Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!

From Assembly to C

What might be the equivalent C-like operation?

1. `mov $0x0,%rdx`
2. `mov %rdx,%rcx`
3. `mov $0x42,(%rdi)`
4. `mov (%rax,%rcx,8),%rax`

From Assembly to C

What might be the equivalent C-like operation?

Indirect addressing is like pointer arithmetic/deref!

1. `mov $0x0,%rdx -> maybe long x = 0`
2. `mov %rdx,%rcx -> maybe long x = y;`
3. `mov $0x42,(%rdi) -> maybe *ptr = 0x42;`
4. `mov (%rax,%rcx,8),%rax -> maybe long x = arr[i];`

Lecture Plan

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

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

Bit: 63	31	15	7	0
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

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

Sometimes, you might see mov suffixes that specify the amount of data being moved. Other times, they are omitted if we can deduce the size from the arguments.

```
movl %eax,(%rsp)
movw (%rax),%dx
movb (%rsp,%rdx,4),%dl
mov  $0x0,%eax
```

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 ← 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 ← 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 <- SignExtend(%eax)`

Register Sizes

- The operand forms with parentheses (e.g. **mov (%rax)**) require that registers in parentheses be the 64-bit registers.
- For that reason, you may see smaller registers extended with e.g. **movs** into the larger registers before these kinds of instructions.

Our First Assembly

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

```
0000000000401136 <sum_array>:
  401136:     b8 00 00 00 00          mov    $0x0,%eax
  40113b:     ba 00 00 00 00          mov    $0x0,%edx
  401140:     39 f0                   cmp    %esi,%eax
  401142:     7d 0b                   jge    40114f <sum_array+0x19>
  401144:     48 63 c8                movslq %eax,%rcx
  401147:     03 14 8f                add    (%rdi,%rcx,4),%edx
  40114a:     83 c0 01                add    $0x1,%eax
  40114d:     eb f1                   jmp    401140 <sum_array+0xa>
  40114f:     89 d0                   mov    %edx,%eax
  401151:     c3                      retq
```

Lecture Plan

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

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
`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.
`7(%rax, %rax, 8), %rdx`	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.

Reverse Engineering Practice

```
void calculate(int x, int y, int *ptr) {
    ____?____;
}

----------

calculate:
  leal (%rdi,%rsi,2), %eax
  movl %eax, (%rdx)
  ret
```

Note: assume x is in %rdi, y is in %rsi and ptr is in %rdx.

Reverse Engineering Practice

```
void calculate(int x, int y, int *ptr) {
    *ptr = x + 2 * y;
}

----------

calculate:
  leal (%rdi,%rsi,2), %eax
  movl %eax, (%rdx)
  ret
```

Lecture Plan

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

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

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, except for **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

Unary Instructions

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

Instruction	Effect	Description
`inc D`	D ← D + 1	Increment
`dec D`	D ← D - 1	Decrement
`neg D`	D ← -D	Negate
`not D`	D ← ~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 ← D + S	Add
`sub S, D`	D ← D - S	Subtract
`imul S, D`	D ← D * S	Multiply
`xor S, D`	D ← D ^ S	Exclusive-or
`or S, D`	D ← D | S	Or
`and S, D`	D ← D & S	And

Examples:

```
addq %rcx,(%rax)
xorq $16,(%rax, %rdx, 8)
subq %rdx,8(%rax)
```

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
`sal k, D`	$D \leftarrow D \ll k$	Left shift
`shl k, D`	$D \leftarrow D \ll k$	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D \gg_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D \gg_L k$	Logical right shift

Examples:

```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

Shift Amount

Instruction	Effect	Description
`sal k, D`	D ← D << k	Left shift
`shl k, D`	D ← D << k	Left shift (same as `sal`)
`sar k, D`	D ← D $>>_A$ k	Arithmetic right shift
`shr k, D`	D ← D $>>_L$ k	Logical right shift

When a shift instruction uses **%cl**, it looks at only the number of bits in **%cl** that make sense for what is being shifted.

- E.g. when shifting 1 byte, it looks only at the lower 3 bits (storing at most 7)
- E.g. when shifting 2 bytes, it looks only at the lower 4 bits (storing at most 15)
- When shifting **w** bits, it looks at the low-order **log2(w)** bits of **%cl** for the shift amount.
- Why is this useful? Can specify shift amount as all 1s, but it will shift by the appropriate amount.

Recap

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

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.

Next Time: more arithmetic operations, and reverse engineering practice

Extra Practice

1. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. **results in** this register layout

```
long arr[5];
...
long num = ____???___;
```

```
mov (%rdi, %rcx, 8),%rax
```

<val of num>	3	<val of arr>
%rax	%rcx	%rdi

1. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. **results in** this register layout

```
long arr[5];
...
long num = ____???___;
```

```
long num = arr[3];
long num = *(arr + 3);
long num = *(arr + y);
```

(assume `long y = 3;` declared earlier)

```
mov (%rdi, %rcx, 8),%rax
```

%rax	%rcx	%rdi
<val of num>	3	<val of arr>

2. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
char str[5];
...
___???___ = 'c';
```

```
mov $0x63,(%rcx,%rdx,1)
```

%rcx	%rdx
<val of str>	2

2. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
char str[5];
...
___???___ = 'c';
```

```
str[2] = 'c';
*(str + 2) = 'c';
```

```
mov $0x63,(%rcx,%rdx,1)
```

%rcx	%rdx
<val of str>	2