CS107, Lecture 12
Assembly: Arithmetic, Logic and Condition Codes

Reading: B&O 3.5-3.6

CS107 Topic 6: How does a computer interpret and execute C programs?

Learning Assembly

Moving data around	→	Arithmetic and logical operations	→	Control flow	→	Function calls
5/6		**Today**		**5/17**		**5/20**

Learning Goals

- Learn how to perform arithmetic and logical operations in assembly

Plan For Today

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

Plan For Today

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

Assembly

- **Assembly code** is a human-readable form of the machine code your computer actually executes when running your programs.
- Assembly works at a lower level of abstraction than C code. It works with 64-bit spaces called **registers** that act as “scratch paper” for the processor.
- Operations in your C program ultimately are converted to operations that read or write to registers and perform calculations on these registers.

Our First Assembly

```
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                   jl     4005c2 <sum_array+0xc>
  4005cf:     f3 c3                   repz retq
```

Our First Assembly

```
0000000004005b6 <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                   jl     4005c2 <sum_array+0xc>
  4005cf:     f3 c3                   repz retq
```

Each instruction has an operation name ("opcode").

Our First Assembly

```
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                   jl     4005c2 <sum_array+0xc>
  4005cf:     f3 c3
```

Each instruction can also have arguments (“operands”).

mov

The **mov** instruction copies bytes from one place to another.

```
mov        src,dst
```

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

- Immediate (constant value, like a number)
- 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 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.

Plan For Today

- **Recap:** Assembly and **mov**
- Data and Register Sizes
- The **lea** Instruction
- Logical and Arithmetic Operations
- Control
  - Condition Codes
  - Assembly Instructions
- **Practice**: Reverse-Engineering

Data Sizes

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

- A **byte** is….well, 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

63	31	15	7
%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

63	31	15	7
%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

63	31	15	7
%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

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

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

Instruction		Effect	Description
`MOVZ`	S, R	$R \longleftarrow \mathit{ZeroExtend}(S)$	Move with zero extension
`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

Instruction		Effect	Description
	S, R	$R \longleftarrow \mathit{SignExtend}(S)$	Move with sign extension
`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`		`%rax` $\longleftarrow \mathit{SignExtend}($`%eax`$)$	Sign-extend `%eax` to `%rax`

Plan For Today

- **Recap:** Assembly and **mov**
- Data and Register Sizes
- The **lea** Instruction
- Logical and Arithmetic Operations
- Control
  - Condition Codes
  - Assembly Instructions
- **Practice**: Reverse-Engineering

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.

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 <u>at</u> the address src to the destination, **lea** copies the value of src *itself* to the destination.

Plan For Today

- **Recap:** Assembly and **mov**
- Data and Register Sizes
- The **lea** Instruction
- Logical and Arithmetic Operations
- Control
  - Condition Codes
  - Assembly Instructions
- **Practice**: Reverse-Engineering

Unary Instructions

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

Instruction	Effect	Description
`inc D`	$D \longleftarrow D + 1$	Increment
`dec D`	$D \longleftarrow D - 1$	Decrement
`neg D`	$D \longleftarrow -D$	Negate
`not D`	$D \longleftarrow \sim D$	Complement

Examples:

```
incq 16(%rax)
dec %rdx
not %rcx
```

Binary Instructions

The following instructions operate on two operands (register or memory). Both cannot be memory locations. Read it as, e.g. "Subtract S from D":

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

`imul S, D` $D \longleftarrow D * S$ Multiply

- 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
`imulq S`	R[`%rdx`]:R[`%rax`] $\longleftarrow$ S $\times$ R[`%rax`]	Signed full multiply
`mulq S`	R[`%rdx`]:R[`%rax`] $\longleftarrow$ S $\times$ R[`%rax`]	Unsigned full multiply

Division and Remainder

Instruction	Effect	Description
`idivq S`	R[`%rdx`] $\longleftarrow$ R[`%rdx`]:R[`%rax`] mod S; R[`%rax`] $\longleftarrow$ R[`%rdx`]:R[`%rax`] $\div$ S	Signed divide
`divq S`	R[`%rdx`] $\longleftarrow$ R[`%rdx`]:R[`%rax`] mod S; R[`%rax`] $\longleftarrow$ R[`%rdx`]:R[`%rax`] $\div$ S	Unsigned divide

- <u>Terminology:</u> **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
`cqto`	R[`%rdx`]:R[`%rax`] $\longleftarrow$ SignExtend(R[`%rax`])	Convert to oct word
`idivq S`	R[`%rdx`] $\longleftarrow$ R[`%rdx`]:R[`%rax`] mod S;	Signed divide
	R[`%rax`] $\longleftarrow$ R[`%rdx`]:R[`%rax`] $\div$ S	
`divq S`	R[`%rdx`] $\longleftarrow$ R[`%rdx`]:R[`%rax`] mod S;	Unsigned divide
	R[`%rax`] $\longleftarrow$ R[`%rdx`]:R[`%rax`] $\div$ S	

- 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 operate on two operands, one the shift amount and the other the destination to shift. The shift amount **k** can be either an immediate value, or the byte register **%cl** (and only that register!)

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

Examples:

```
shll $3,(%rax)
shr %cl,(%rax,%rdx,8)
sar $4,8(%rax)
```

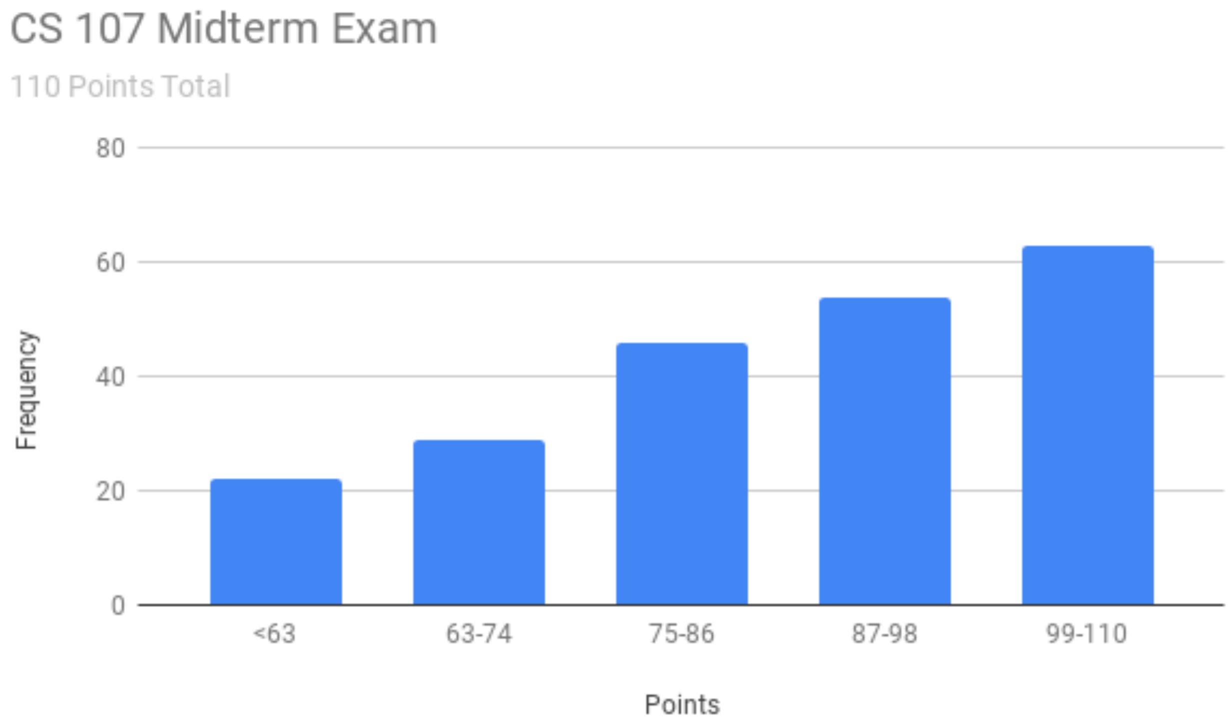
Shift Amount

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

- When using **%cl**, the width of what you are shifting determines how much of **%cl** it is shifted by.
- 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 log2(8) = 3 bits, which represent 7. **shlw** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.

Announcements

- Midterms have been graded, and will be emailed out this afternoon.
- Regrades accepted until the end of the day next Monday



Median: 89, Mean: 85, St. Dev. : 19

Assembly Exploration

- Let’s pull these different commands together and see how some C code we write may be translated into assembly.
- Compiler Explorer is a handy website that lets you write C code and see its assembly translation without having to log into Myth or compile/disassemble a program. Let’s check it out!
- https://godbolt.org/z/NLYhVf

Code Reference

```
// 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;
}

// 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;
}
```

Recap

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

Next time: Control flow in assembly