Lecture #19 – Data Movement
COMP201 Topic 6: How does a computer interpret and execute C programs?
Learning Assembly

This Lecture

Moving data around

Lecture 20

Arithmetic and logical operations

Lecture 21-23

Control flow

Lecture 24-26

Function calls
Lecture Plan

• Recap: mov so far
• Data and Register Sizes

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
Lecture Plan

- **Recap:** `mov` so far
- Data and Register Sizes
- The `lea` Instruction

Source: [https://www.strchr.com/x86_machine_code_statistics](https://www.strchr.com/x86_machine_code_statistics)
The **mov** instruction copies bytes from one place to another; it is similar to the assignment operator ( = ) in C.

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

- Immediate (constant value, like a number) *(only src)*

  
  
  $$\text{mov} \quad \text{src, dst}$$

- Register

- Memory Location *(at most one of src, dst)*

  
  $\text{mov} \quad \text{src, dst}$

  
  $\text{mov} \quad \%rbx$

  
  Direct address 0x6005c0

  
  Immediate (constant value, like a number) $0x104$
Operand Forms: Immediate

`mov $0x104,_____`

Copy the value 0x104 into some destination.
Operand Forms: Registers

Copy the value in register `%rbx` into some destination.

```assembly
mov  `%rbx, _____
```

Copy the value from some source into register `%rbx`.

```assembly
mov  _____, `%rbx
```
Operand Forms: Absolute Addresses

- `mov 0x104,_____`  
  Copy the value at address 0x104 into some destination.

- `mov _____,0x104`  
  Copy the value from some source into the memory at address 0x104.
Operand Forms: Indirect

Copy the value at the address stored in register %rbx into some destination.

```plaintext
mov (%rbx),_____ 
```

Copy the value from some source into the memory at the address stored in register %rbx.

```plaintext
mov _____,(%rbx) 
```
Operand Forms: Base + Displacement

- `mov` 0x10(%rax),_________
  
  Copy the value at the address (0x10 plus what is stored in register %rax) into some destination.

- `mov` __________,0x10(%rax)
  
  Copy the value from some source into the memory at the address (0x10 plus what is stored in register %rax).
Copy the value at the address which is (the sum of the values in registers %rax and %rdx) into some destination.

Copy the value from some source into the memory at the address which is (the sum of the values in registers %rax and %rdx).
Operand Forms: Indexed

Copy the value at the address which is (the sum of $0x10$ plus the values in registers $%rax$ and $%rdx$) into some destination.

```plaintext
mov 0x10(%rax,%rdx),______
```

Copy the value from some source into the memory at the address which is (the sum of $0x10$ plus the values in registers $%rax$ and $%rdx$).

```plaintext
mov ______,0x10(%rax,%rdx)
```
Practice #1: 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,
the value 0xAB is stored at address 0x104,
0x100 is stored in register %rax and 0x3 is stored in %rdx.

1. `mov $0x42,(%rax)` Move 0x42 to memory address 0x100
2. `mov 4(%rax),%rcx` Move 0xAB into %rcx
3. `mov 9(%rax,%rdx),%rcx` Move 0x11 into %rcx

\[ \text{Imm}(r_b, r_i) \text{ is equivalent to address } \text{Imm} + R[r_b] + R[r_i] \]

**Displacement:** positive or negative constant (if missing, = 0)
**Base:** register (if missing, = 0)
**Index:** register (if missing, = 0)
Operand Forms: Scaled Indexed

mov \((,%rdx,4)\),______

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

mov ______,\((,%rdx,4)\)

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

The scaling factor (e.g. 4 here) must be hardcoded to be either 1, 2, 4 or 8.
Operand Forms: Scaled Indexed

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

```plaintext
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`).

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

\[
\text{mov } (%rax, %rdx, 2), _________
\]

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).

\[
\text{mov } _________, (%rax, %rdx, 2)
\]
Operand Forms: Scaled Indexed

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

\texttt{mov 0x4(\%rax,\%rdx,2),_____}

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

\texttt{mov _____,0x4(\%rax,\%rdx,2)}
Most General Operand Form

\[ \text{Imm}(r_b, r_i, s) \]

is equivalent to...

\[ \text{Imm} + R[r_b] + R[r_i]*s \]
Most General Operand Form

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

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

**Index:** register
- (if missing, = 0)

**Base:** register
- (if missing, = 0)

**Scale** must be 1, 2, 4, or 8
- (if missing, = 1)
# Memory Location Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td>Address 0x104 (no $)</td>
</tr>
<tr>
<td>(%rax)</td>
<td>What’s in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>What’s in %rax, plus 4</td>
</tr>
<tr>
<td>(%rax, %rdx)</td>
<td>Sum of what’s in %rax and %rdx</td>
</tr>
<tr>
<td>4(%rax, %rdx)</td>
<td>Sum of values in %rax and %rdx, plus 4</td>
</tr>
<tr>
<td>(, %rcx, 4)</td>
<td>What’s in %rcx, times 4 (multiplier can be 1, 2, 4, 8)</td>
</tr>
<tr>
<td>(%rax, %rcx, 2)</td>
<td>What’s in %rax, plus 2 times what’s in %rcx</td>
</tr>
<tr>
<td>8(%rax, %rcx, 2)</td>
<td>What’s in %rax, plus 2 times what’s in %rcx, plus 8</td>
</tr>
</tbody>
</table>
## Operand Forms

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand Value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$\text{Imm}$</td>
<td>$\text{Imm}$</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$r_a$</td>
<td>$\text{R}[r_a]$</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}$</td>
<td>$\text{M}[\text{Imm}]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_a$)</td>
<td>$\text{M}[\text{R}[r_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}(r_b)$</td>
<td>$\text{M}[\text{Imm} + \text{R}[r_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_b, r_i$)</td>
<td>$\text{M}[\text{R}[r_b] + \text{R}[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}(r_b, r_i)$</td>
<td>$\text{M}[\text{Imm} + \text{R}[r_b] + \text{R}[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>($, r_i, s$)</td>
<td>$\text{M}[\text{R}[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}(, r_i, s)$</td>
<td>$\text{M}[\text{Imm} + \text{R}[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_b, r_i, s$)</td>
<td>$\text{M}[\text{R}[r_b] + \text{R}[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}(r_b, r_i, s)$</td>
<td>$\text{M}[\text{Imm} + \text{R}[r_b] + \text{R}[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

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 #2: 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
- the value 0x11 is stored at address 0x10C.

1. mov $0x42,0xfc(,%rcx,4)  
   Move 0x42 to memory address 0x100

2. mov (%rax,%rdx,4),%rbx  
   Move 0x11 into %rbx
Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!
Extra Practice
Extra Practice

Fill in the blank to complete the code that generated the assembly below.

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

// %rdi stores arr, %rcx stores 3, and %rax stores num
mov (%rdi, %rcx, 8),%rax
Extra Practice

Fill in the blank to complete the code that generated the assembly below.

```c
long arr[5];
...
long num = arr[3];
```

// %rdi stores arr, %rcx stores 3, and %rax stores num
mov (%rdi, %rcx, 8),%rax
Extra Practice

Fill in the blank to complete the code that generated the assembly below.

```c
int x = ...
int *ptr = malloc(...);
___???____ = x;

// %ecx stores x, %rax stores ptr
mov %ecx,(%rax)
```
Extra Practice

Fill in the blank to complete the code that generated the assembly below.

```c
int x = ...
int *ptr = malloc(...);
*ptr = x;
```

// %ecx stores x, %rax stores ptr
mov %ecx,(%rax)
Extra Practice

Fill in the blank to complete the code that generated the assembly below.

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

// %rcx stores str, %rdx stores 2
mov $0x63,(%rcx,%rdx,1)
```
Extra Practice

Fill in the blank to complete the code that generated the assembly below.

```c
char str[5];
...
str[2] = 'c';
```

```
// %rcx stores str, %rdx stores 2
mov $0x63,(%rcx,%rdx,1)
```
Lecture Plan

• Recap: mov so far

• Data and Register Sizes
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**
• 1 means **double word**
• q means **quad word**
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.

<table>
<thead>
<tr>
<th>C Type</th>
<th>Suffix</th>
<th>Byte</th>
<th>Intel Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>b</td>
<td>1</td>
<td>Byte</td>
</tr>
<tr>
<td>short</td>
<td>w</td>
<td>2</td>
<td>Word</td>
</tr>
<tr>
<td>int</td>
<td>l</td>
<td>4</td>
<td>Double word</td>
</tr>
<tr>
<td>long</td>
<td>q</td>
<td>8</td>
<td>Quad word</td>
</tr>
<tr>
<td>char *</td>
<td>q</td>
<td>8</td>
<td>Quad word</td>
</tr>
<tr>
<td>float</td>
<td>s</td>
<td>4</td>
<td>Single precision</td>
</tr>
<tr>
<td>double</td>
<td>l</td>
<td>8</td>
<td>Double precision</td>
</tr>
</tbody>
</table>
Register Sizes

<table>
<thead>
<tr>
<th>Bit: 63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%al</td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bl</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%cl</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dl</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td>%sil</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td>%dil</td>
<td></td>
</tr>
</tbody>
</table>
## Register Sizes

<table>
<thead>
<tr>
<th>Bit: 63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td>%bpl</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td>%spl</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
<td>%r8b</td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
<td>%r9b</td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
<td>%r10b</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
<td>%r11b</td>
<td></td>
</tr>
</tbody>
</table>
## Register Sizes

<table>
<thead>
<tr>
<th>Bit: 63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
<td>%r12b</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
<td>%r13b</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%r14w</td>
<td>%r14b</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
<td>%r15b</td>
<td></td>
</tr>
</tbody>
</table>
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 Stanford CS107 x86-64 Reference Sheet on Resources page of the course website! [https://aykuterdem.github.io/classes/comp201/index.html#div_resources](https://aykuterdem.github.io/classes/comp201/index.html#div_resources)
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 #3: 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)
   movl_%eax, (%rsp)
2. mov_%(%rax), %dx
   movw_%(%rax), %dx
3. mov_%$0xff, %bl
   movb_$0xff, %bl
4. mov_%(%rsp,%rdx,4),%dl
   movb_%(%rsp,%rdx,4),%dl
5. mov_%(%rdx), %rax
   movq_%(%rdx), %rax
6. mov_%%dx, (%rax)
   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
```
Practice #4: mov And Data Sizes

For each of the following mov instructions, determine how data movement instructions modify the upper bytes of a destination register.

1. movabs $0x0011223344556677, %rax  \%rax = 0011223344556677
2. movb $-1, %al  \%rax = 00112233445566FF
3. movw $-1, %ax  \%rax = 001122334455FFFF
4. movl $-1, %eax  \%rax = 00000000FFFFFF
5. movq $-1, %rax  \%rax = FFFFFFFFFFFFFFFF
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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movzbw</td>
<td>Move zero-extended byte to word</td>
</tr>
<tr>
<td>movzbl</td>
<td>Move zero-extended byte to double word</td>
</tr>
<tr>
<td>movzw1</td>
<td>Move zero-extended word to double word</td>
</tr>
<tr>
<td>movzbq</td>
<td>Move zero-extended byte to quad word</td>
</tr>
<tr>
<td>movzwq</td>
<td>Move zero-extended word to quad word</td>
</tr>
</tbody>
</table>
movz and movs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movsbw</td>
<td>Move sign-extended byte to word</td>
</tr>
<tr>
<td>movsbl</td>
<td>Move sign-extended byte to double word</td>
</tr>
<tr>
<td>movswl</td>
<td>Move sign-extended word to double word</td>
</tr>
<tr>
<td>movsbq</td>
<td>Move sign-extended byte to quad word</td>
</tr>
<tr>
<td>movswq</td>
<td>Move sign-extended word to quad word</td>
</tr>
<tr>
<td>movslq</td>
<td>Move sign-extended double word to quad word</td>
</tr>
<tr>
<td>cltq</td>
<td>Sign-extend %eax to %rax</td>
</tr>
<tr>
<td></td>
<td>%rax ← SignExtend(%eax)</td>
</tr>
</tbody>
</table>

MOVS S,R  \quad R ← SignExtend(S)
Recap

• mov Instruction
• Data and Register Sizes

Next Time: Logical and Arithmetic Operations
The story of Mel

Source: scottt@israeli, May 21, 1983.

A recent article devoted to the *maverick* side of programming made the bald and unvarnished statement:

Real Programmers write in Fortran.

Maybe they do now, in this decadent era of Line bar, hand calculators and "user-friendly" software but back in the Good Old Days, when the term "software" sounded funny and Real Computers were made out of drums and vacuum tubes, Real Programmers wrote in machine code. Not Fortran. Not KATFOR. Not even, assembly language. Machine Code. Raw, unadorned, incunabular hexadecimal numbers. Directly.

Let this whole new generation of programmers grow up in ignorance of this glorious past. I feel duty-bound to describe, as best I can through the generation gap, how a Real Programmer wrote code. I'll call him Mel, because that was his name.

I first met Mel when I went to work for Royal McBee Computer Corp., a now-defunct subsidiary of the typewriter company. The firm manufactured the LPG-30, a small, cheap-by-the-standards-of-the-day drum-memory computer, and had just started to manufacture the RPC-4000, a much-improved, bigger, better, faster — drum-memory computer. Cress cost too much, and weren't here to stay, anyway. (That's why you haven't heard of the company, or the computer.)

I had been hired to write a Fortran compiler for this new marvel and Mel was my guide to its wonders. Mel didn't approve of compilers.

"If a program can't rewrite its own code," he asked, "what good is it?"

Mel had written, in hexadecimal, the most popular computer program the company owned. It ran on the LPG-30 and played Blackjack with potential customers at computer shows. Its effect was always dramatic. The LPG-30 booth was packed at every show, and the IBM salesmen stood around talking to each other. Whether or not this actually sold computers was a question we never discussed.

Mel's job was to re-write the blackjack program for the RPC-4000. (Port? What does that mean?) The new computer had a one-plus-one addressing scheme, in which each machine instruction, in addition to the operation code and the address of the needed operand, had a second address that indicated where, on the revolving drum, the next instruction was located. In modern parlance, every single instruction was followed by a GO TO. Put "that" in Pascal's pipe and smoke it.

Mel loved the RPC-4000 because he could optimize his code: that is, locate instructions on the drum so that just as soon as one finished its job, the next would be just arriving at the "read head" and available for immediate execution. There was a program to do that job, an "optimizing assembler", but Mel refused to use it.

"You never know where it's going to put things," he explained, "so you'd have to use separate constants".

It was a long time before I understood that remark. Since Mel knew the numerical value of every operation code, and assigned his own drum addresses, every instruction he wrote could also be considered a numerical constant. He could pick up an earlier "old" instruction, say, and multiply by it, if it had the right numerical value. His code was not easy, for someone else to modify.

http://www.pbm.com/~lindahl/mel.html

Annotated: https://www.cs.utah.edu/~elb/folklore/mel-annotated/mel-annotated.html