CS107 Lecture 15
Introduction to Assembly, Take II

Reading: B&O 3.1-3.4

Move Operations

The **mov** instruction **copies bytes from one location to another**.
It's akin to the assignment (=) in C where the **arguments are reversed**.

```
mov src,dst
```

src can be any one of:

- **Immediate** `$0x314`
- **Register** `%rbx`
- **Memory Location** `0x6005c0`

dst can **always** be a register, but it can **never** be an immediate. It can also be a **memory location**, though **only one** of **src** and **dst** can be a memory location in any one **mov** instruction. x86-64 **doesn't support arbitrary memory-to-memory moves**.

Operand Forms: Immediate

```
mov     $0x104,______
```

Copy the value **`0x104`** into some destination.

Operand Forms: Registers

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

```
mov     %rbx,____
```

```
mov     ____,%rcx
```

Copy the value from some **source** into register **`%rcx`**.

Operand Forms: Absolute Addresses

Copy the value at address **`0x104`** into some **destination**.

```
mov     0x104,_____
```

```
mov     _____,0x104
```

Copy the value from some **source** into the memory at address **`0x104`**.

Practice: Operand Forms

What do **each of the following `mov` instructions do**? Assume the value **5** is stored at address **`0x42`**, and the value **8** is stored in **`%rbx`**.

1. `mov $0x42,%rax`

2. `mov 0x42,%rax`

3. `mov %rbx,0x55`

Operand Forms: Indirect

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

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

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

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

Operand Forms: Base + Displacement

Copy the value at the address **`0x10`** more than what is stored in register **`%rax`** into some **destination**.

```
mov    0x10(%rax),_____
```

```
mov    _____,0x10(%rax)
```

Copy the value from some **source** into the memory at the address that is **`0x10`** more than what is stored in register **`%rax`**.

Operand Forms: Indexed

Copy the value at the address **`%rax + %rdx`** into some **destination**.

```
mov     (%rax,%rdx),_____

mov     _____,(%rax,%rdx)
```

Copy the value from some **source** into the memory at the address **`%rax + %rdx`**.

Operand Forms: Indexed

Copy the value at the address **`%rbx + %rdx + 0x10`** into some **destination**.

```
mov     0x10(%rbx,%rdx),_____

mov     ______,0x40(%r8,%r9)
```

Copy the value from some **source** into the memory at the address **`%r8 + %r9 + 0x40`** .

Practice: Operand Forms

What do **each of the following** **`mov`** **instructions do**? 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 **`%rcx + 8 * %rax`** into some **destination**.

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

```
mov     _____,(%rdi,%rsi,4)
```

Copy the value from some **source** into the memory at the address **`%rdi + 4 * %rsi`** .

Operand Forms: Scaled Indexed

Copy the value at the address
`%rax + 8 * %r11 + 0x4` into some **destination**.

```
mov     0x4(%rax,%r11,8),_____

mov     ______,0x1(%rbx,%rdx,4)
```

Copy the value from some **source** into the memory at the address **`%rbx + 4 * %rdx + 0x1`** .

Most General Operand Form

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

Practice: Operand Forms

What do **each of the following** **`mov`** **instructions do**? 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`**.

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

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

Baby's First Assembly: Revisited

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

We're finally understanding some **real assembly**! **What makes sense at this point?**

- Registers store addresses and values
- **`mov src, dst`** copies value from **`src`** to **`dst`**
- **`sizeof(int)`** is **4**
- Instructions executed sequentially by default

```
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
  4005ce:    7c f2                jl     4005c2 <sum_array+0xc>
  4005d0:    f3 c3                repz retq
```

We'll come back to this example in future lectures!

From Assembly to C

Spend a few minutes thinking about where the **mov** instruction might come into play. **What line of C might compile to each of the following**?

- Examples:

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

```
long y = 0;
long offset = y;
arr[0] = 66;
long w = arr[offset];
```

Indirect addressing is essentially pointer arithmetic and dereference.

Extra Practice 1

Fill in the blank to complete the C code that

1. mystery line compiles to this assembly
2. registers hold these values

```
int x = ...
int *ptr = malloc(…);
...
___???___ = _???_;
```

```
mov %ecx,(%rax)
```

<val of x>	<val of ptr>
%ecx	%rax

Try subbing in <x> and <ptr> with actual values, like 4 and 0x7fff80

Extra Practice 1

Fill in the blank to complete the C code that

```
int x = ...
int *ptr = malloc(…);
...
___???___ = _???_;    *ptr = x;
```

```
mov %ecx,(%rax)
```

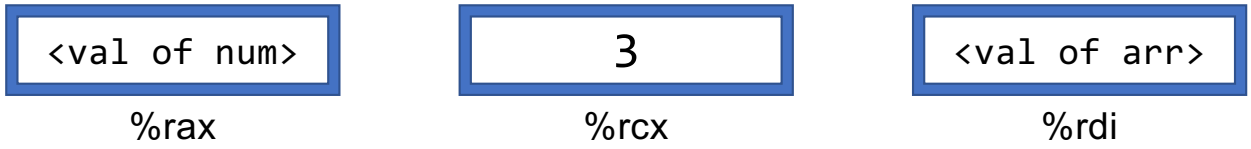
Extra Practice 2

Fill in the blank to complete the C code that

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

```
long *arr = malloc(…);
...
long num = ____???___;
```

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



Extra Practice 2

Fill in the blank to complete the C code that

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

```
long *arr = malloc(…);
...
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>

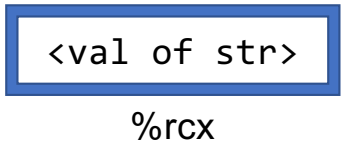
Extra Practice 3

Fill in the blank to complete the C code that

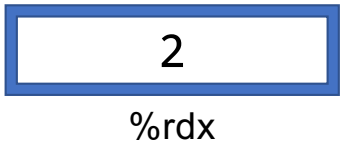
1. generates this assembly
2. has this register layout

```
char *str = malloc(…);
long i = 2;
___???___ = 'c';
```

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



%rcx



%rdx

Extra Practice 3

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
char *str = malloc(…);
long i = 2;
___???___ = 'c';
```

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

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

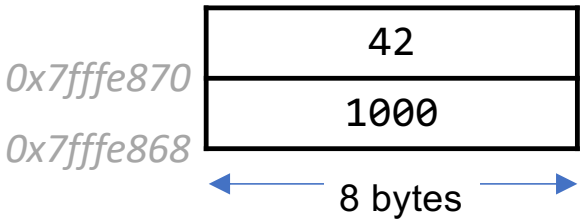
Bonus: Sneak peek into next week

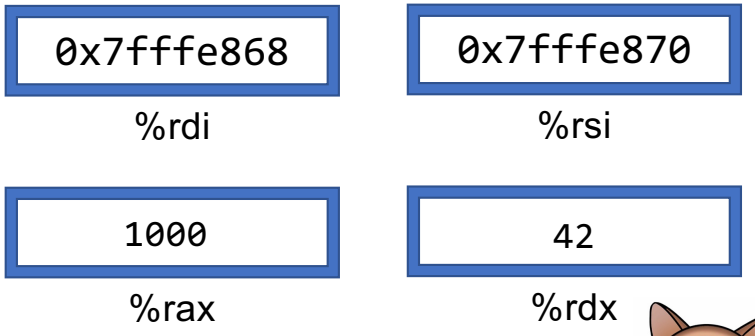
- The below code is the **`objdump`** of a C function, **`foo`**.
  - `foo` keeps its 1st and 2nd parameters are in registers %rdi and %rsi, respectively.

```
0x4005b6 <foo>      mov    (%rdi),%rax
0x4005b9 <foo+3>    mov    (%rsi),%rdx
0x4005bc <foo+6>    mov    %rdx,(%rdi)
0x4005bf <foo+9>    mov    %rax,(%rsi)
```

1. **What does this function do?**
2. **What C code could have generated this assembly?**

(Hints: make up C variable names as needed, assume all regs 64-bit)





Bonus: Sneak peek into next week

- The below code is the **`objdump`** of a C function, **`foo`**.
  - `foo` keeps its 1st and 2nd parameters are in registers %rdi and %rsi, respectively.

```
0x4005b6 <foo>      mov    (%rdi),%rax
0x4005b9 <foo+3>    mov    (%rsi),%rdx
0x4005bc <foo+6>    mov    %rdx,(%rdi)
0x4005bf <foo+9>    mov    %rax,(%rsi)
```

```
void foo(long *xp, long *yp) {
  long a = *xp;
  long b = *yp;
  *yp = a;
  *xp = b;
  ...
```