

x86-64 Programming II

CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker

Adithi Raghavan

Aman Mohammed

Brenden Page

Celestine Buendia

Chloe Fong

Claire Wang

Hamsa Shankar

Maggie Jiang

Malak Zaki

Naama Amiel

Nikolas McNamee

Shananda Dokka

Stephen Ying

Will Robertson



*** An Amendment**

Announcements, Reminders

- ❖ HW6 due tonight! (10 Apr)
 - HW7 due 15 Apr
 - HW8 due 17 Apr
 - HW 9/10 due 19 Apr
- ❖ No reading due for next class! Naama will be instructing!
- ❖ Lab 1b due 15 Apr by 11:59 PM
- ❖ Lab 2 releasing 12 Apr; due 26 Apr

Lab Extras

- ❖ All labs starting with Lab 2 have extra “components”
 - These are meant to be **fun extensions to the labs**
- ❖ Extra components aren't graded, nor do we expect you to do them
 - Make sure you finish the lab before even attempting any extra component

Last Time: Practice Question

❖ Which of the following are valid implementations of $rcx = rax + rbx$?

- `addq %rax, %rcx`
`addq %rbx, %rcx`

- `movq %rax, %rcx`
`addq %rbx, %rcx`

- `movq $0, %rcx`
`addq %rbx, %rcx`
`addq %rax, %rcx`

- `xorq %rax, %rax`
`addq %rax, %rcx`
`addq %rbx, %rcx`

Arithmetic Example

```

long simple_arith(long x, long y)
{
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
    
```

Register	Use(s)
%rdi	1 st argument (x)
%rsi	2 nd argument (y)
%rax	return value

Compiler Optimization & Generation

```

y += x;
y *= 3;
long r = y;
return r;
    
```

```

simple_arith:
    addq    %rdi, %rsi
    imulq   $3, %rsi
    movq    %rsi, %rax
    ret
    
```

Example of Basic Addressing Modes

```
void swap(long* xp, long* yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

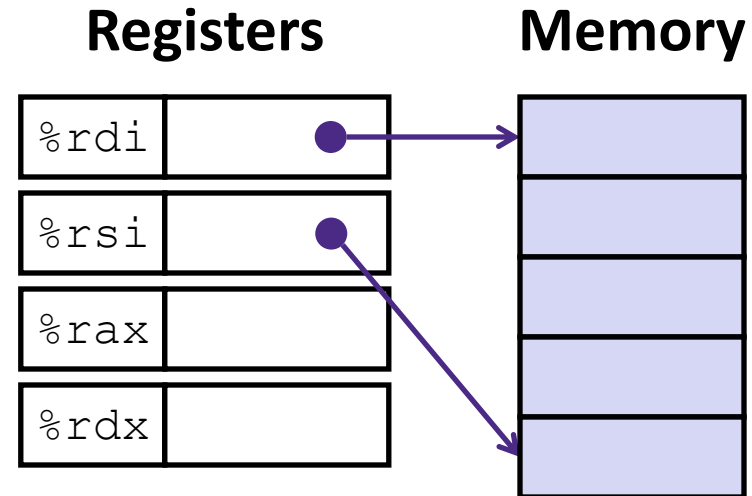
Compiler Explorer:

<https://godbolt.org/z/8sMcnasjx>

Understanding swap ()

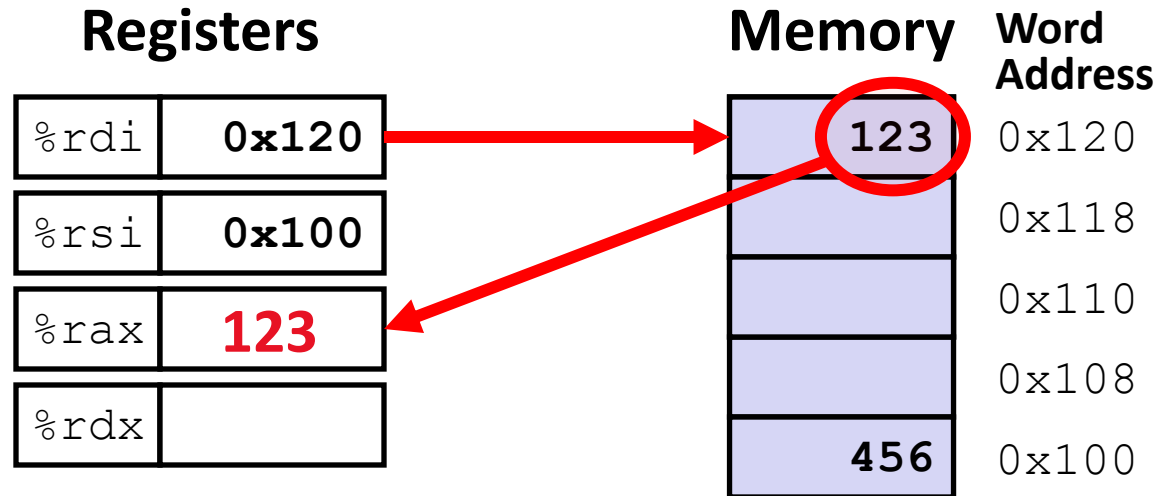
```
void swap(long* xp, long* yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```



<u>Register</u>		<u>Variable</u>
%rdi	↔	xp
%rsi	↔	yp
%rax	↔	t0
%rdx	↔	t1

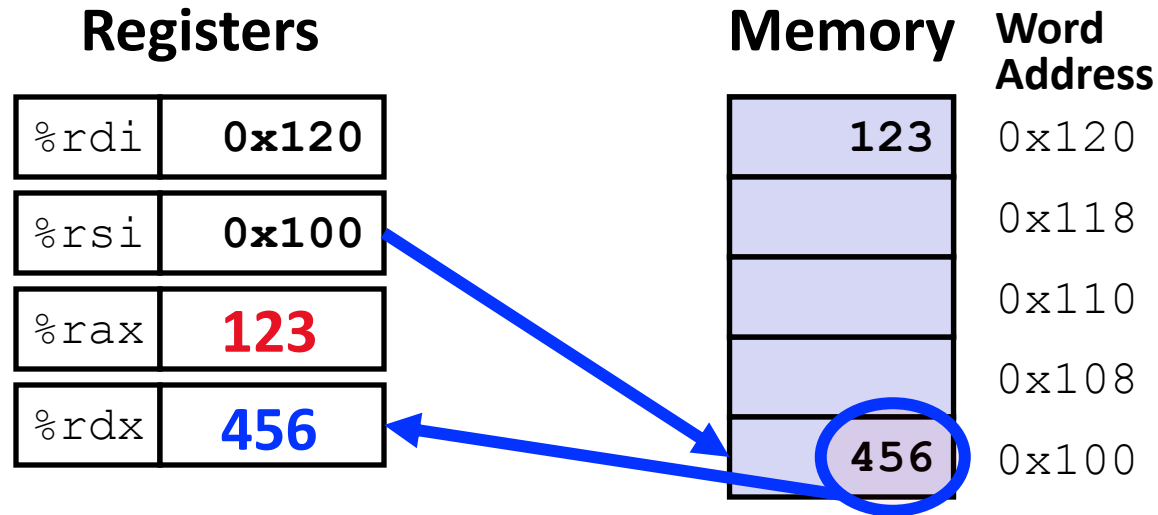
Understanding swap ()



```
swap:
```

```
 1 movq  (%rdi), %rax  # t0 = *xp
   movq  (%rsi), %rdx  # t1 = *yp
   movq  %rdx,  (%rdi) # *xp = t1
   movq  %rax,  (%rsi) # *yp = t0
   ret
```

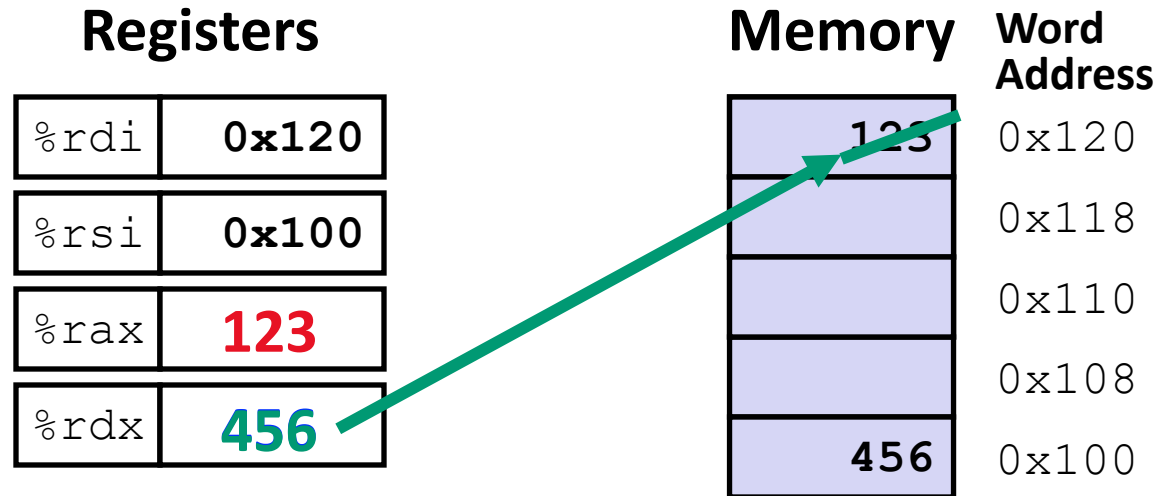

Understanding swap ()



```

swap:
1 movq  (%rdi), %rax  # t0 = *xp
2 movq  (%rsi), %rdx  # t1 = *yp
   movq  %rdx, (%rdi)  # *xp = t1
   movq  %rax, (%rsi)  # *yp = t0
   ret
    
```

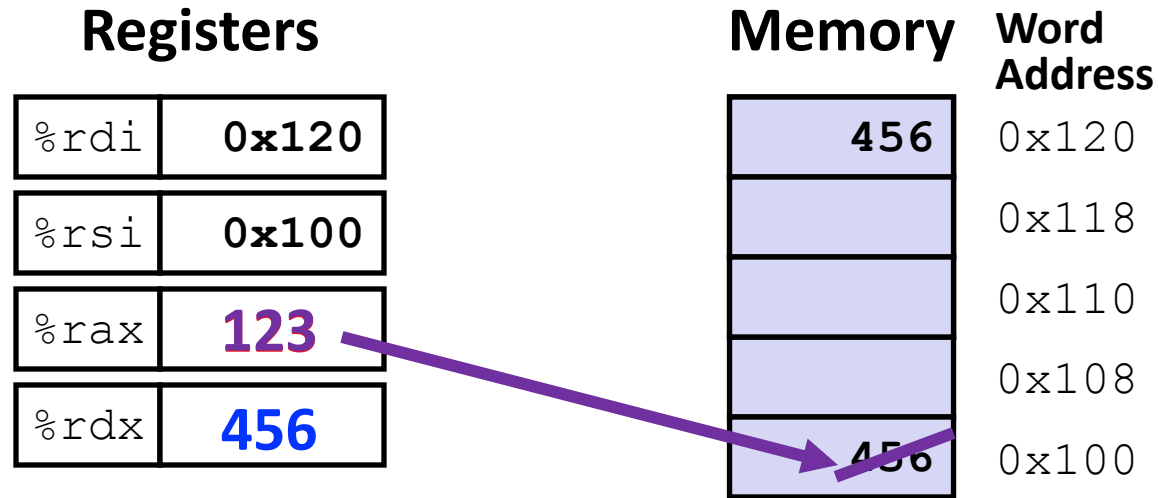
Understanding swap ()



```
swap:
```

```
1 movq  (%rdi), %rax  # t0 = *xp
2 movq  (%rsi), %rdx  # t1 = *yp
3 movq  %rdx, (%rdi)  # *xp = t1
   movq  %rax, (%rsi)  # *yp = t0
   ret
```

Understanding swap ()



```

swap:
1 movq  (%rdi), %rax  # t0 = *xp
2 movq  (%rsi), %rdx  # t1 = *yp
3 movq  %rdx, (%rdi)  # *xp = t1
4 movq  %rax, (%rsi)  # *yp = t0
ret
    
```

Understanding swap ()

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

	Word Address
456	0x120
	0x118
	0x110
	0x108
123	0x100

```
swap:
```

```
1 movq (%rdi), %rax # t0 = *xp
2 movq (%rsi), %rdx # t1 = *yp
3 movq %rdx, (%rdi) # *xp = t1
4 movq %rax, (%rsi) # *yp = t0
ret
```

Complete Memory Addressing Modes

❖ General:

$$D(\mathbf{Rb}, \mathbf{Ri}, \mathbf{S}) \rightarrow \text{Mem}[\text{Reg}[\mathbf{Rb}] + \text{Reg}[\mathbf{Ri}] * \mathbf{S} + \mathbf{D}]$$

- **Rb**: Base register (any register)
- **Ri**: Index register (any register except `%rsp`)
- **S**: Scale factor (1, 2, 4, 8) – *why these numbers?*
- **D**: Constant displacement value (a.k.a. immediate)

❖ Special cases (see CSPP Figure 3.3 on p.181)

Implies

- | | | |
|--|--|---------------------------------|
| ■ $D(\mathbf{Rb}, \mathbf{Ri})$ | $\text{Mem}[\text{Reg}[\mathbf{Rb}] + \text{Reg}[\mathbf{Ri}] + \mathbf{D}]$ | $(\mathbf{S}=1)$ |
| ■ $(\mathbf{Rb}, \mathbf{Ri}, \mathbf{S})$ | $\text{Mem}[\text{Reg}[\mathbf{Rb}] + \text{Reg}[\mathbf{Ri}] * \mathbf{S}]$ | $(\mathbf{D}=0)$ |
| ■ $(\mathbf{Rb}, \mathbf{Ri})$ | $\text{Mem}[\text{Reg}[\mathbf{Rb}] + \text{Reg}[\mathbf{Ri}]]$ | $(\mathbf{S}=1, \mathbf{D}=0)$ |
| ■ $(, \mathbf{Ri}, \mathbf{S})$ | $\text{Mem}[\text{Reg}[\mathbf{Ri}] * \mathbf{S}]$ | $(\mathbf{Rb}=0, \mathbf{D}=0)$ |

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

$$D(Rb, Ri, S) \rightarrow \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$$

we'll ignore the actual memory access for now

Expression	Address Computation	Address (8 bytes wide)
<code>0x8(%rdx)</code>		
<code>(%rdx,%rcx)</code>		
<code>(%rdx,%rcx,4)</code>		
<code>0x80(,%rdx,2)</code>		

Reading Review

❖ Terminology:

- Address Computation Instruction (`lea`)
- Condition codes: Carry Flag (CF), Zero Flag (ZF), Sign Flag (SF), and Overflow Flag (OF)
- Test (`test`) and compare (`cmp`) assembly instructions
- Jump (`j*`) and set (`set*`) families of assembly instructions

Review Questions

$$D (Rb, Ri, S) \rightarrow \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$$

- ❖ Which of the following x86-64 instructions correctly calculates:

`%rax=9*%rdi`

- A. `leaq (, %rdi, 9), %rax`
 - B. `movq (, %rdi, 9), %rax`
 - C. `leaq (%rdi, %rdi, 8), %rax`
 - D. `movq (%rdi, %rdi, 8), %rax`
- ❖ If `%rsi` is `0x B0BA CAFE 1EE7 F00D`, what is its value after executing `movswl %si, %esi`?

Address Computation Instruction

- ❖ `leaq src, dst`
 - "lea" stands for **load effective address**
 - `src` is address expression (any of the formats we've seen)
 - `dst` is a register
 - Sets `dst` to the **address** computed by the `src` expression
 - Does **not** go to memory! – it just does math!
 - Example: `leaq (%rdx,%rcx,4), %rax`
- ❖ Uses:
 - Computing addresses without a memory reference
 - *e.g.*, translation of `p = &x[i];`
 - Computing arithmetic expressions of the form $x+k*i+d$
 - Though `k` can only be 1, 2, 4, or 8

Example: lea vs. mov

Registers

%rax	0x110
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Memory

Word Address

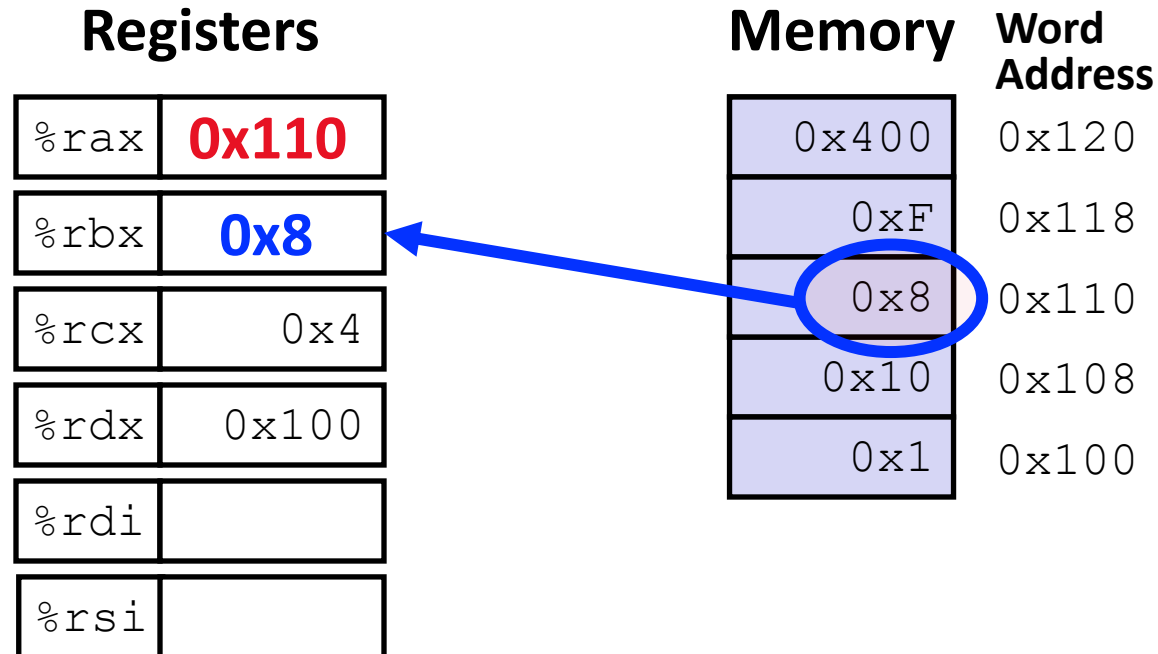
0x400	0x120
0xF	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```

1 leaq (%rdx,%rcx,4), %rax
    movq (%rdx,%rcx,4), %rbx
    leaq (%rdx), %rdi
    movq (%rdx), %rsi
    
```

0x100 + 0x4 * 4 = 0x110

Example: lea vs. mov



```

1 leaq (%rdx,%rcx,4), %rax
2 movq (%rdx,%rcx,4), %rbx
   leaq (%rdx), %rdi
   movq (%rdx), %rsi
    
```

0x100 + 0x4 * 4 = 0x110

Example: lea vs. mov

Registers

%rax	0x110
%rbx	0x8
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Memory Word Address

0x400	0x120
0xF	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
1 leaq (%rdx, %rcx, 4), %rax
2 movq (%rdx, %rcx, 4), %rbx
3 leaq (%rdx), %rdi
  movq (%rdx), %rsi
```

Example: lea vs. mov

Registers		Memory	Word Address
%rax	0x110	0x400	0x120
%rbx	0x8	0xF	0x118
%rcx	0x4	0x8	0x110
%rdx	0x100	0x10	0x108
%rdi	0x100	0x1	0x100
%rsi	0x1		

```
1 leaq (%rdx,%rcx,4), %rax
2 movq (%rdx,%rcx,4), %rbx
3 leaq (%rdx), %rdi
4 movq (%rdx), %rsi
```

Example: lea vs. mov

Registers

%rax	0x110
%rbx	0x8
%rcx	0x4
%rdx	0x100
%rdi	0x100
%rsi	0x1

Memory

	Word Address
0x400	0x120
0xF	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
1 leaq (%rdx,%rcx,4), %rax
2 movq (%rdx,%rcx,4), %rbx
3 leaq (%rdx), %rdi
4 movq (%rdx), %rsi
```

leaq used for Arithmetic

```

long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}

```

```

arith:
    leaq    (%rdi,%rsi), %rax
    addq    %rdx, %rax
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx
    leaq    4(%rdi,%rdx), %rcx
    imulq   %rcx, %rax
    ret

```

Register	Use(s)
%rdi	1 st argument (x)
%rsi	2 nd argument (y)
%rdx	3 rd argument (z)

Rather than using multiplication, use leaq and shifting!

❖ Interesting Instructions

- leaq: “address” computation
- salq: shift
- imulq: multiplication—only used once!

leaq used for Arithmetic

```
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```
arith:
    leaq    (%rdi,%rsi), %rax    # rax/t1    = x + y
    addq    %rdx, %rax          # rax/t2    = t1 + z
    leaq    (%rsi,%rsi,2), %rdx  # rdx       = 3 * y
    salq    $4, %rdx           # rdx/t4    = (3*y) * 16
    leaq    4(%rdi,%rdx), %rcx   # rcx/t5    = x + t4 + 4
    imulq   %rcx, %rax          # rax/rval  = t5 * t2
    ret
```

Register	Use(s)
%rdi	x
%rsi	y
%rdx	z, t4
%rax	t1, t2, rval
%rcx	t5

Move extension: movz and movs

movz __ src, regDest # Move with zero extension

movs __ src, regDest # Move with sign extension

- Copy from a **smaller** source value to a **larger** destination
- Source can be memory or register; but Destination must be a register
- Fill remaining bits of dest with **zero** (movz) or **sign bit** (movs)

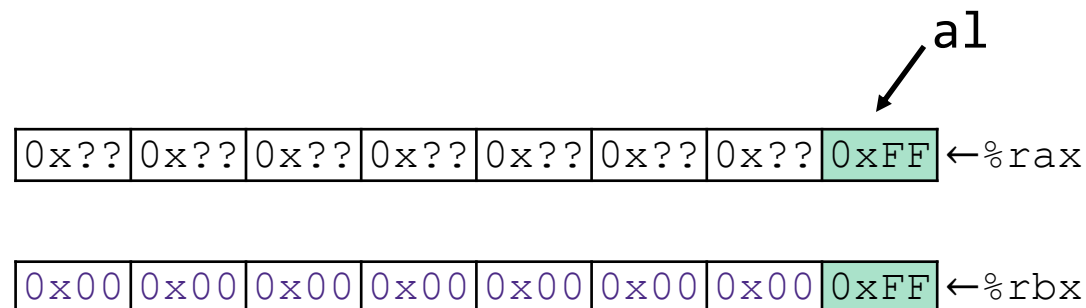
movzsd / movssd:

s – size of **source** (**b** = 1 byte, **w** = 2, **l** = 4)

d – size of **destination** (**w** = 2 bytes, **l** = 4, **q** = 8)

Example:

movzbq %al, %rbx



Move extension: movz and movs

```
movz __ src, regDest    # Move with zero extension
movs __ src, regDest    # Move with sign extension
```

- Copy from a **smaller** source value to a **larger** destination
- Source can be memory or register; but Destination **must** be a register
- Fill remaining bits of dest with **zero** (movz) or **sign bit** (movs)

Note: In x86-64, any instruction that generates a 32-bit (long word) value for a register **also** sets the high-order portion of the register to 0. Also the reason why **movz1q** is not defined! Good example on p. 184 in the textbook.

movzsd / movssd:

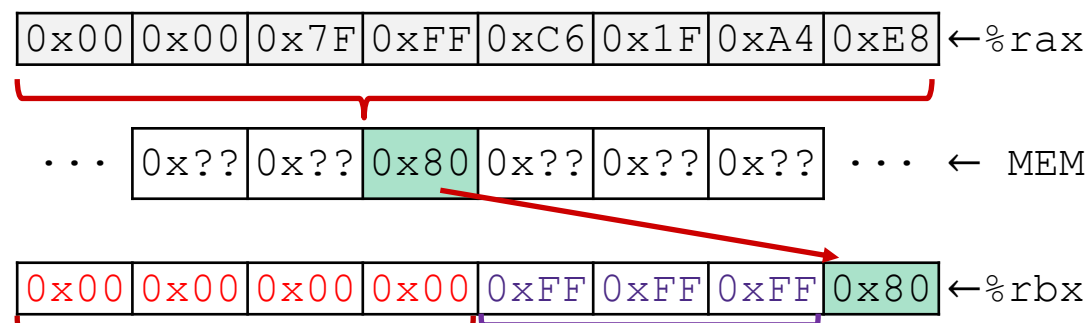
s – size of **source** (**b** = 1 byte, **w** = 2, **l** = 4)

d – size of **destination** (**w** = 2 bytes, **l** = 4, **q** = 8)

Example:

```
movsbl (%rax), %ebx
```

Copy 1 byte from memory into 8-byte register & sign extend it!



Summary

- ❖ **Memory Addressing Modes:** The addresses used for accessing memory in `MOV` (and other) instructions can be computed in several different ways
 - *Base register, index register, scale factor, and displacement* map well to pointer arithmetic operations
- ❖ Control flow in x86 determined by Condition Codes