x86-64 Programming II

CSE 351 Summer 2020

Instructor:

Porter Jones

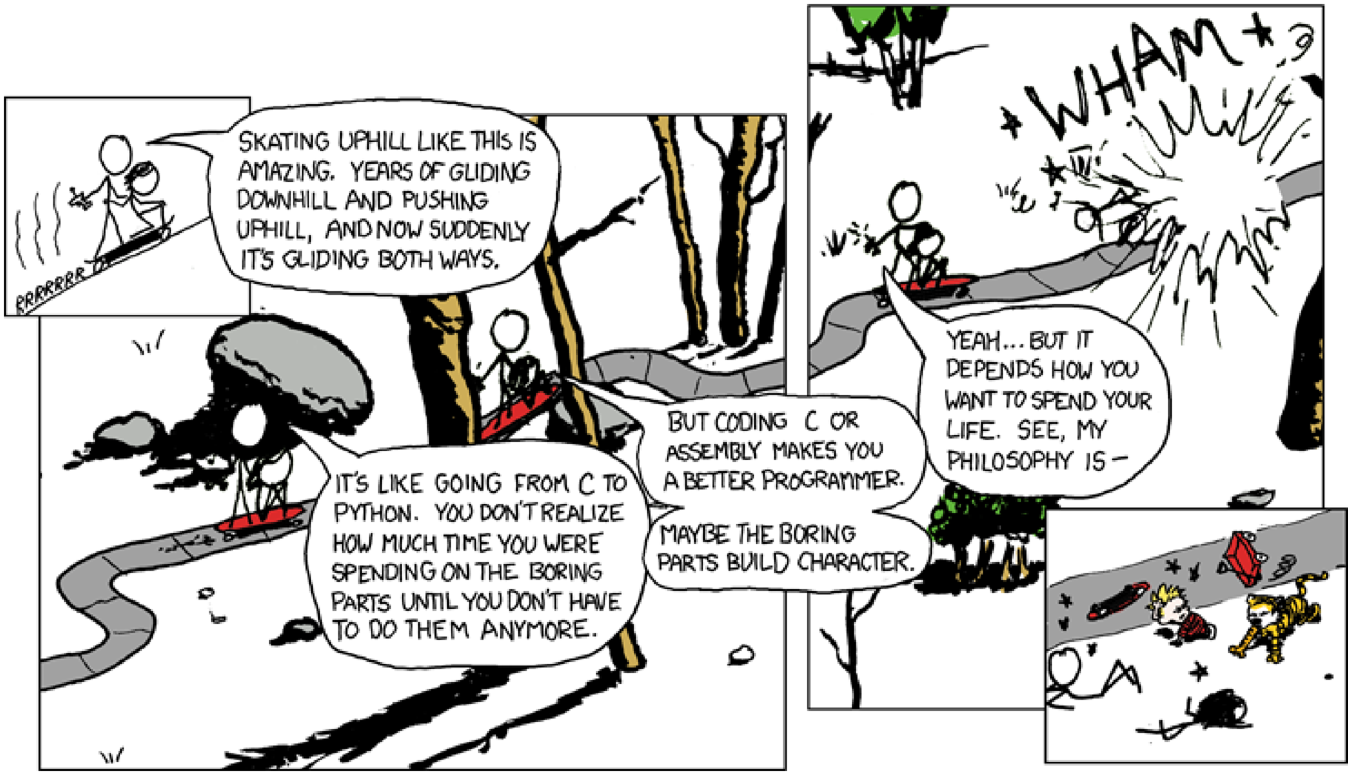
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk



http://xkcd.com/409/

Administrivia

- Questions doc: https://tinyurl.com/CSE351-7-10
- **See my email about accommodations!**
- Lab 1b now due Monday at 11:59pm (7/13)
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Breflect.txt`
  - Can still use late days until 7/15
- hw6, hw7 now due Monday (7/13) – 10:30am
- Unit Summary 1 now due Friday (7/17) – 11:59pm
  - Can still use late days until 7/20
- Mid-quarter Survey still due Friday (7/17) – 11:59pm
- hw8, hw9, hw10 now due Monday (7/20) – 10:30am

Administrivia

- Lab1a grades released later today
  - Talk to us about any questions you have!
  - Regrades open 24 hours after an assignment is due, stay open usually for about a week
- Lab 2 released later today!
  - Debugging x86-64 assembly using gdb
- I will now drop your lowest homework score (see Syllabus for more details).
  - Essentially will bump your homework total up by 11.5 points (the largest single homework total).

x86-64 Introduction

- Data transfer instruction (`mov`)
- Arithmetic operations
- **Memory addressing modes**
- **Address computation instruction (`lea`)**

Memory Addressing Modes: Basic

- **Indirect:** `(R)` Mem[Reg[R]]
  - Data in register `R` specifies the memory address
  - Like pointer dereference in C
  - Example: `movq (%rcx), %rax`

- **Displacement:** `D(R)` Mem[Reg[R]+D]
  - Data in register `R` specifies the *start* of some memory region
  - Constant displacement `D` specifies the offset from that address
  - Example: `movq 8(%rbp), %rdx`

Complete Memory Addressing Modes

- **General:**
  - `D(Rb,Ri,S)` Mem[Reg[`Rb`]+Reg[`Ri`]*`S`+`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)
  - `D(Rb,Ri)` Mem[Reg[`Rb`]+Reg[`Ri`]+`D`] `(S=1)`
  - `(Rb,Ri,S)` Mem[Reg[`Rb`]+Reg[`Ri`]*`S`] `(D=0)`
  - `(Rb,Ri)` Mem[Reg[`Rb`]+Reg[`Ri`]] `(S=1,D=0)`
  - `(,Ri,S)` Mem[Reg[`Ri`]*`S`] `(Rb=0,D=0)`

Address Computation Examples

%rdx	**0xf000**
%rcx	**0x0100**

D(Rb,Ri,S) →
Mem[Reg[Rb]+Reg[Ri]*S+D]

Expression	Address Computation	Address
0x8(%rdx)		
(%rdx,%rcx)		
(%rdx,%rcx,4)		
0x80(,%rdx,2)		

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

Memory

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

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

`lea` – "It just does math"

Arithmetic Example

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

```
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
```

- Interesting Instructions
  - `leaq`: "address" computation
  - `salq`: shift
  - `imulq`: multiplication
    - Only used once!

Arithmetic Example

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

```
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
```

Polling Question [Asm II – a]

- Which of the following x86-64 instructions correctly calculates `%rax = 9 * %rdi`?
  - Vote at http://pollev.com/pbjones

A. `leaq (,%rdi,9), %rax`

B. `movq (,%rdi,9), %rax`

C. `leaq (%rdi,%rdi,8), %rax`

D. `movq (%rdi,%rdi,8), %rax`

E. We're lost…

Control Flow

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

```
long max(long x, long y)
{
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

```
max:
   ???
   movq    %rdi, %rax
   ???
   ???
   movq    %rsi, %rax
   ???
   ret
```

Control Flow

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

```
long max(long x, long y)
{
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

Conditional jump

Unconditional jump

```
max:
   if x <= y then jump to else
   movq    %rdi, %rax
   jump to done
else:
   movq    %rsi, %rax
done:
   ret
```

Conditionals and Control Flow

- Conditional branch/*jump*
  - Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- Unconditional branch/*jump*
  - *Always* jump when you get to this instruction

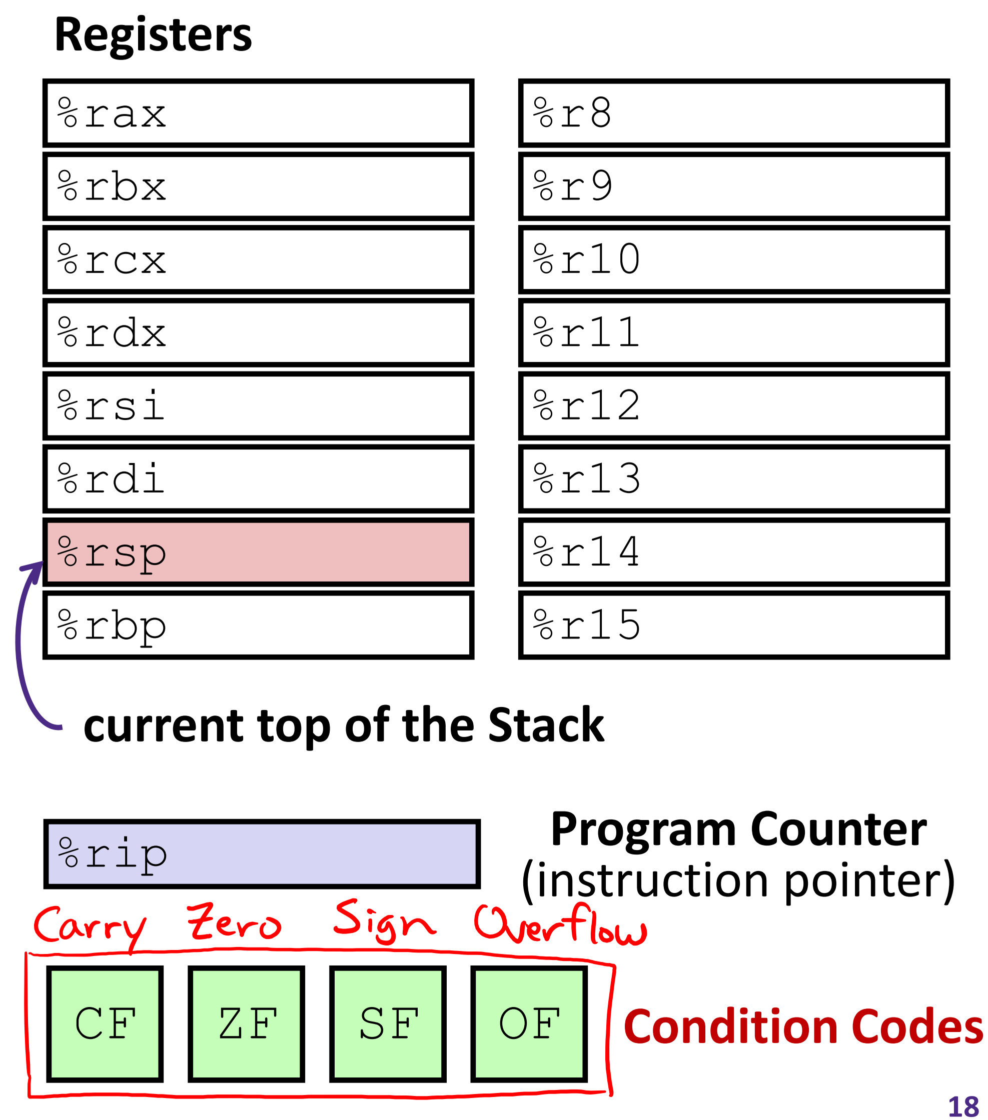
- Together, they can implement most control flow constructs in high-level languages:
  - **`if`** (*condition*) **`then`** {…} **`else`** {…}
  - **`while`** (*condition*) {…}
  - **`do`** {…} **`while`** (*condition*)
  - **`for`** (*initialization*; *condition*; *iterative*) {…}
  - **`switch`** {…}

x86 Control Flow

- **Condition codes**
- **Conditional and unconditional branches**
- Loops
- Switches

Processor State (x86-64, partial)

- Information about currently executing program
  - Temporary data (`%rax`, …)
  - Location of runtime stack (`%rsp`)
  - Location of current code control point (`%rip`, …)
  - Status of recent tests (**CF**, **ZF**, **SF**, **OF**) "flags"
    - Single bit registers:



Condition Codes (Implicit Setting)

- *Implicitly* set by **arithmetic** operations
  - (think of it as side effects)
  - Example: **`addq`** `src, dst` ↔ `r = d+s`
  - **`CF=1`** if carry out from MSB (*unsigned* overflow)
  - **`ZF=1`** if `r==0`
  - **`SF=1`** if `r<0` (if MSB is 1)
  - **`OF=1`** if *signed* overflow
    `(s>0 && d>0 && r<0)||(s<0 && d<0 && r>=0)`
  - *Not* set by `lea` instruction (beware!)

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Condition Codes (Explicit Setting: Compare)

- *Explicitly* set by **Compare** instruction
  - **`cmpq`** `src1, src2`
  - **`cmpq`** `a, b` sets flags based on `b-a`, but doesn't store `b-a`

  - **`CF=1`** if carry out from MSB (good for *unsigned* comparison)
  - **`ZF=1`** if `a==b`
  - **`SF=1`** if `(b-a)<0` (if MSB is 1)
  - **`OF=1`** if *signed* overflow

```
(a>0 && b<0 && (b-a)>0) ||
(a<0 && b>0 && (b-a)<0)
```

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Condition Codes (Explicit Setting: Test)

- *Explicitly* set by **Test** instruction
  - **`testq`** `src2, src1`
  - **`testq`** `a, b` sets flags based on `a&b`, but doesn't store `a&b`
    - Useful to have one of the operands be a ***mask***

  - Can't have carry out (**`CF`**) or overflow (**`OF`**)
  - **`ZF=1`** if `a&b==0`
  - **`SF=1`** if `a&b<0` (signed)

CF	*Carry Flag*	ZF	*Zero Flag*	SF	*Sign Flag*	OF	*Overflow Flag*

Using Condition Codes: Jumping

- `j*` Instructions
  - Jumps to ***target*** (an address) based on condition codes

don't worry about the details

Instruction	Condition	Description (always compared to 0)
jmp *target*	1	Unconditional
je *target*	ZF	Equal / Zero
jne *target*	~ZF	Not Equal / Not Zero
js *target*	SF	Negative
jns *target*	~SF	Nonnegative
jg *target*	~(SF^OF)&~ZF	Greater (Signed)
jge *target*	~(SF^OF)	Greater or Equal (Signed)
jl *target*	(SF^OF)	Less (Signed)
jle *target*	(SF^OF)|ZF	Less or Equal (Signed)
ja *target*	~CF&~ZF	Above (unsigned “>”)
jb *target*	CF	Below (unsigned “<“)

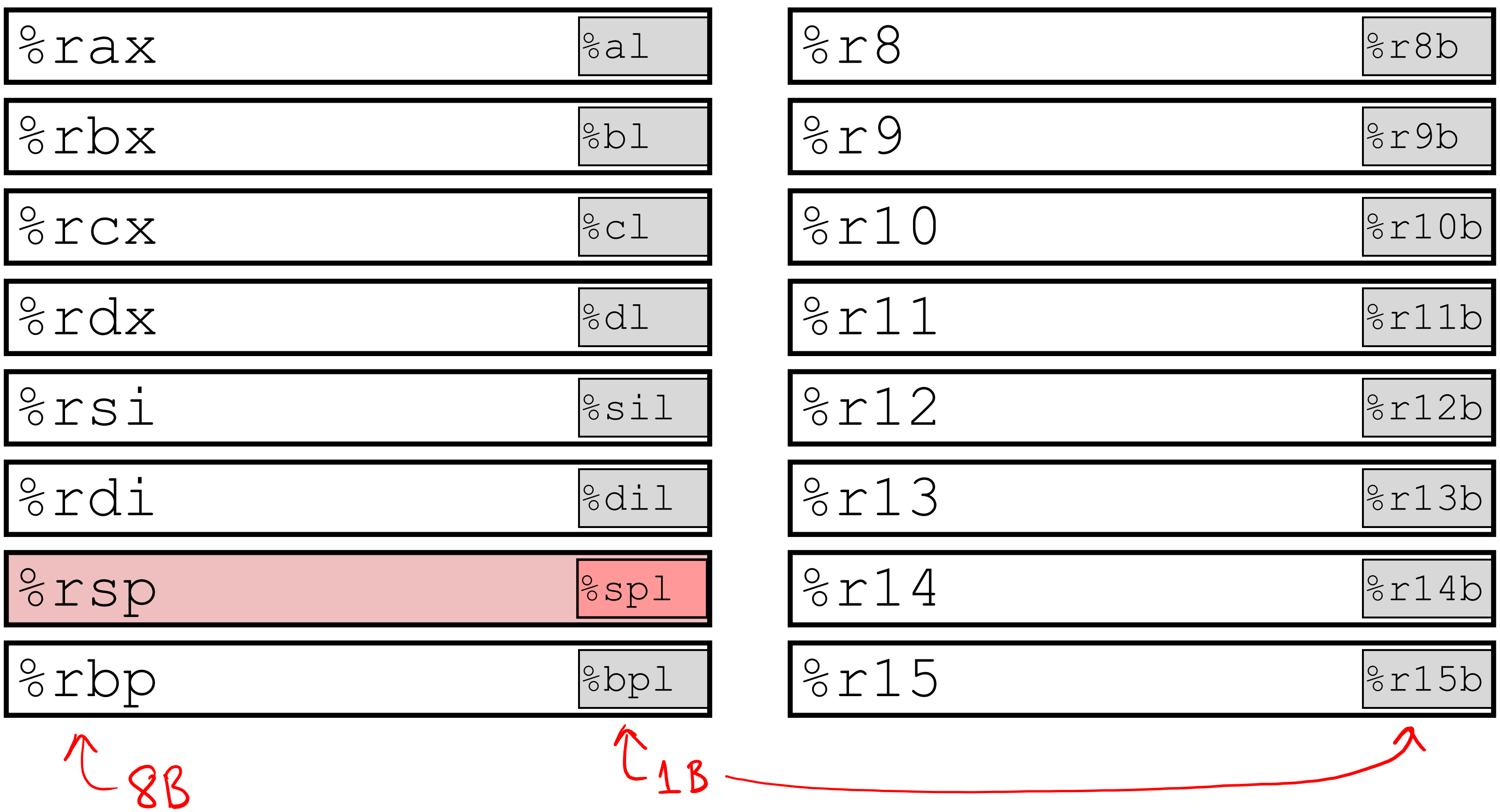
Using Condition Codes: Setting

- `set*` Instructions
  - Set low-order byte of `dst` to 0 or 1 based on condition codes
  - Does not alter remaining 7 bytes

Instruction	Condition	Description
`sete` *dst*	`ZF`	Equal / Zero
`setne` *dst*	`~ZF`	Not Equal / Not Zero
`sets` *dst*	`SF`	Negative
`setns` *dst*	`~SF`	Nonnegative
`setg` *dst*	`~(SF^OF)&~ZF`	Greater (Signed)
`setge` *dst*	`~(SF^OF)`	Greater or Equal (Signed)
`setl` *dst*	`(SF^OF)`	Less (Signed)
`setle` *dst*	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta` *dst*	`~CF&~ZF`	Above (unsigned “>”)
`setb` *dst*	`CF`	Below (unsigned “<”)

Reminder: x86-64 Integer Registers

- Accessing the low-order byte:



Reading Condition Codes

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

- `set*` Instructions
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. `al`, `dl`) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

```
int gt(long x, long y)
{
  return x > y;
}
```

```
cmpq   %rsi, %rdi    #
setg   %al           #
movzbl %al, %eax     #
ret
```

Reading Condition Codes

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

- set* Instructions
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. al, dl) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use movzbl (zero-extended mov) to finish job

```
int gt(long x, long y)
{
  return x > y;
}
```

```
cmpq   %rsi, %rdi    # Compare x:y
setg   %al           # Set when >
movzbl %al, %eax     # Zero rest of %rax
ret
```

Aside: `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; 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)

D – size of dest (**`w`** = 2 bytes, **`l`** = 4, **`q`** = 8)

Example:

`movzbq %al, %rbx`

0x??	0x??	0x??	0x??	0x??	0x??	0x??	0xFF	←%rax
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0xFF	←%rbx

Aside: `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; Destination *must* be a register
- Fill remaining bits of dest with **zero** (`movz`) or **sign bit** (`movs`)

`movz`*SD* / **`movs`*SD*:**

S – size of source (**`b`** = 1 byte, **`w`** = 2)

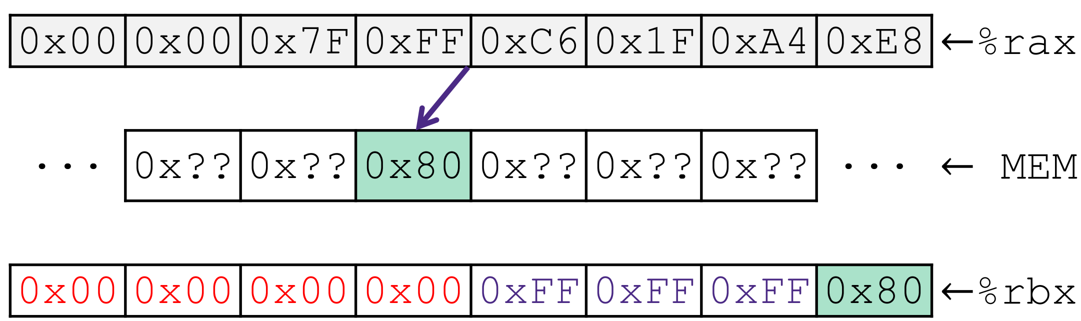
D – size of dest (**`w`** = 2 bytes, **`l`** = 4, **`q`** = 8)

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. Good example on p. 184 in the textbook.

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 status of Condition Codes
  - Showed **C**arry, **Z**ero, **S**ign, and **O**verflow, though others exist
  - Set flags with arithmetic instructions (implicit) or Compare and Test (explicit)
  - Set instructions read out flag values
  - Jump instructions use flag values to determine next instruction to execute