x86-64 Programming II

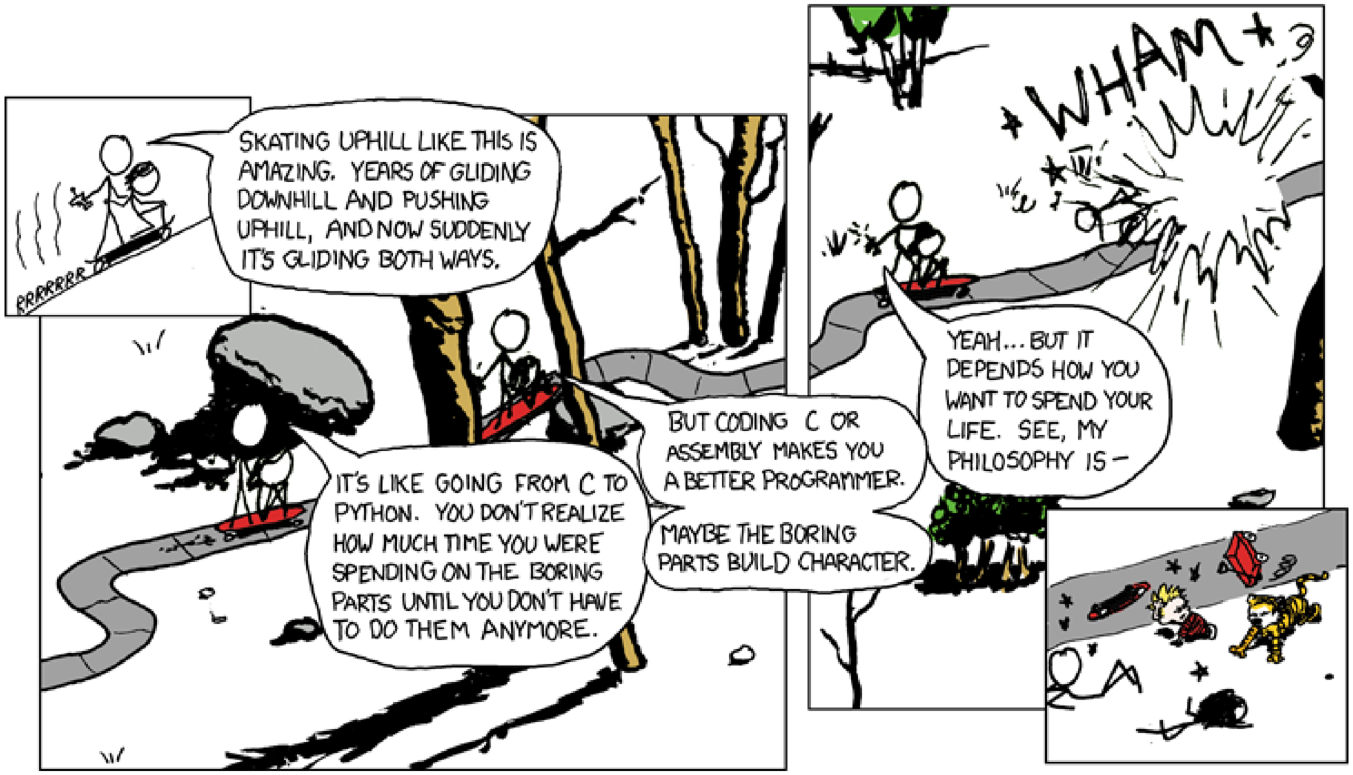
CSE 351 Spring 2019

Instructor:

Ruth Anderson

Teaching Assistants:

Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh


RRRRRRR
SKATING UPHILL LIKE THIS IS AMAZING. YEARS OF GLIDING DOWNHILL AND PUSHING UPHILL, AND NOW SUDDENLY IT'S GLIDING BOTH WAYS.
IT'S LIKE GOING FROM C TO PYTHON. YOU DON'T REALIZE HOW MUCH TIME YOU WERE SPENDING ON THE BORING PARTS UNTIL YOU DON'T HAVE TO DO THEM ANYMORE.
BUT CODING C OR ASSEMBLY MAKES YOU A BETTER PROGRAMMER.
MAYBE THE BORING PARTS BUILD CHARACTER.
YEAH... BUT IT DEPENDS HOW YOU WANT TO SPEND YOUR LIFE. SEE, MY PHILOSOPHY IS –
WHAM

http://xkcd.com/409/

Administrivia

- Lab 1b due Monday (4/22)
  - Submit `bits.c` and `lab1Breflect.txt`
- Homework 2 due Wednesday (4/24)
  - On Integers, Floating Point, and x86-64
- Lab 2 (x86-64) coming soon, due Wednesday (5/01)

- Midterm in two weeks (Fri 5/03, 4:30pm in KNE 130)
  - No lecture that day

Address Computation Instruction

- `leaq src, dst` — "Mem" Reg
  - "`lea`" stands for *load effective address*
  - `src` is address expression (any of the formats we've seen)
    ↳ calculates Reg[Rb]+Reg[Ri]*S+D
  - `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`
    = %rdx + 4*%rcx rather than Memory [%rdx + 4*%rcx] like before
- Uses:
  - Computing addresses without a memory reference
    - *e.g.* translation of `p = &x[i];` address-of operator
  - Computing arithmetic expressions of the form `x+k*i+d` Reg[Rb]+Reg[Ri]*S + D
    - Though k can only be 1, 2, 4, or 8

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

4

```
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;   ← replaced by lea & shift
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax     # rax = x+y (t1)
  addq    %rdx, %rax            # rax = x+y+z (t2)
  leaq    (%rsi,%rsi,2), %rdx   # rdx = 3y
  salq    $4, %rdx              # rdx = 48y (t4)
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax            ← multiplying two variables
  ret
```

calculates t3 as part of this step
4 + x + 48y = t5

- Interesting Instructions
  - leaq: "address" computation
  - salq: shift shift four spots = multiply by 2^4 = 16
  - imulq: multiplication
    - Only used once!

Arithmetic Example

```
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;
}
```

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

limited registers means they often get reused!

comment

```
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
```

Peer Instruction Question

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

$s \in \{1,2,4,8\}$

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

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

$\boxed{\text{C. } \texttt{leaq (\%rdi,\%rdi,8), \%rax}}$ $\rightarrow$ %rax = 9 * %rdi

D. `movq (%rdi,%rdi,8), %rax` %rax = *(9 * %rdi)

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:
  ???         x
  movq    %rdi, %rax
  ???
  ???         y
  movq    %rsi, %rax
  ???
  ret
```

Control Flow

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

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

Conditional jump

Unconditional jump

```
max:
            if TRUE
  if x <= y then jump to else
      ~~if FALSE~~
  movq    %rdi, %rax
  jump to done
else:
            y
  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

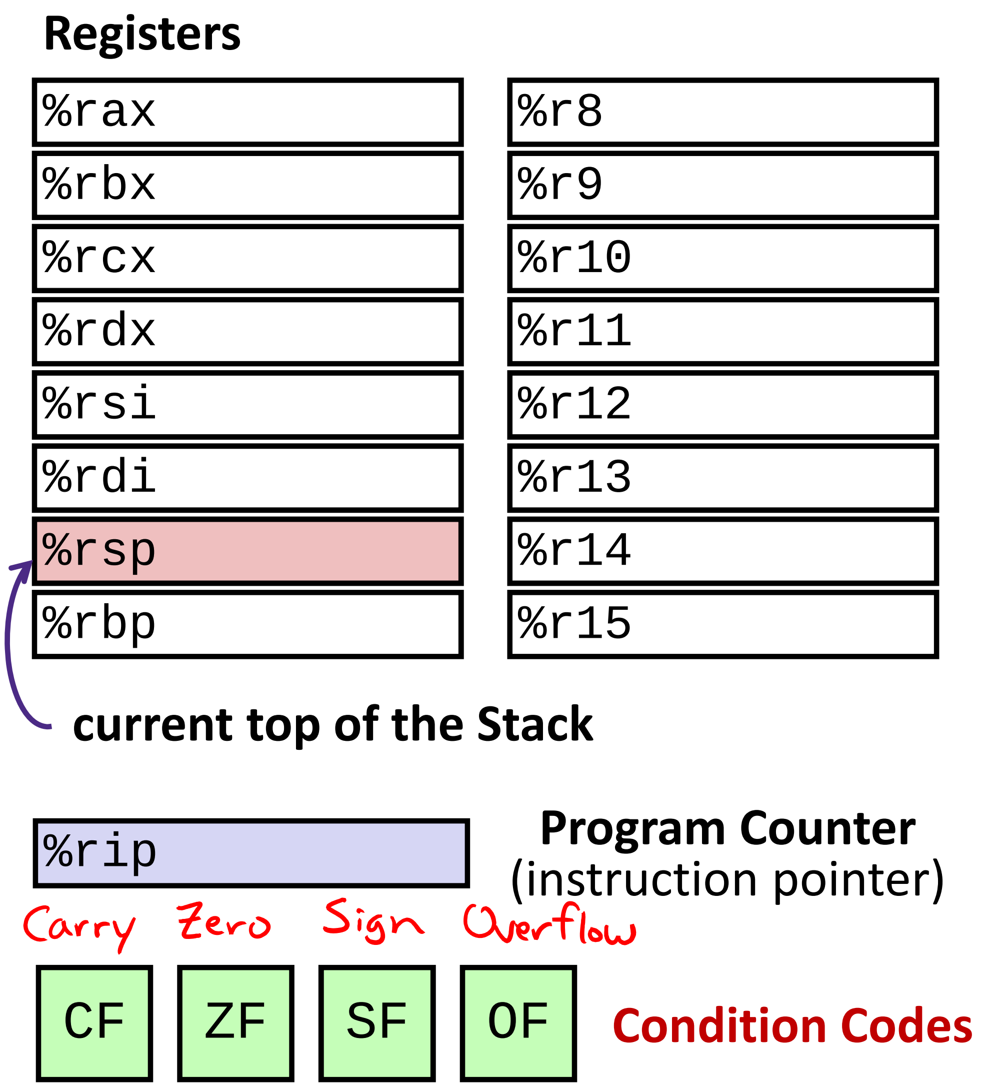
- Together, they can implement most control flow constructs in high-level languages:
  - **`if`** (*condition*) **`then`** {…} **`else`** {…}
  - **`while`** (*condition*) {…}
  - **`do`** {…} **`while`** (*condition*)
  - **`for`** (*initialization*; *condition*; *iterative*) {…}
  - **`switch`** {…}

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)

%rcx ↓

By implicit, they mean they are set automatically any time you perform other operations

- *Implicitly* set by **arithmetic** operations
  - (think of it as side effects)
  - Example: **addq** src, dst ↔ r = d+s

    %rcx, %rcx

```
  0x80 -- 0
  0b 10000.. 0
+    100 ... -0
 ---------------
   1 0000 .. 0
```

  - **CF=1** if carry out from MSB (*unsigned* overflow)   CF = 1
  - **ZF=1** if r==0   ZF = 1
  - **SF=1** if r<0 (if MSB is 1)   SF = 0
  - **OF=1** if *signed* overflow   OF = 1

    (s>0 && d>0 && r<0)||(s<0 && d<0 && r>=0)

    signs don't match!
  - *Not* set by `lea` instruction (beware!)

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

Condition Codes (Explicit Setting: Compare)

If we want to set these condition codes without altering any of the info we have in the registers by performing an actual calculation

- *Explicitly* set by **Compare** instruction
  - **`cmpq`** `src1, src2` $\boxed{\text{like subq a, b} \longrightarrow b = b - a}$
  - **`cmpq`** `a, b` sets flags based on `b-a`, but doesn't store

  - **`CF=1`** if carry out from MSB (good for *unsigned* comparison)
  - **`ZF=1`** if `a==b` (b-a==0)
  - **`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` like $\boxed{\text{andq a, b}}$
  - **`testq`** `a, b` sets flags based on a&b, but doesn't store
    - Useful to have one of the operands be a ***mask***

- Can't have carry out (**CF**) or overflow (**OF**) $\text{CF}=0$, $\text{OF}=0$
- **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 e.g. this is where you would consider setting the condition codes. If you wanted to perform a while loop like "while n > 0" you would want to check if n > 0 before actually decrementing it or something
  - 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

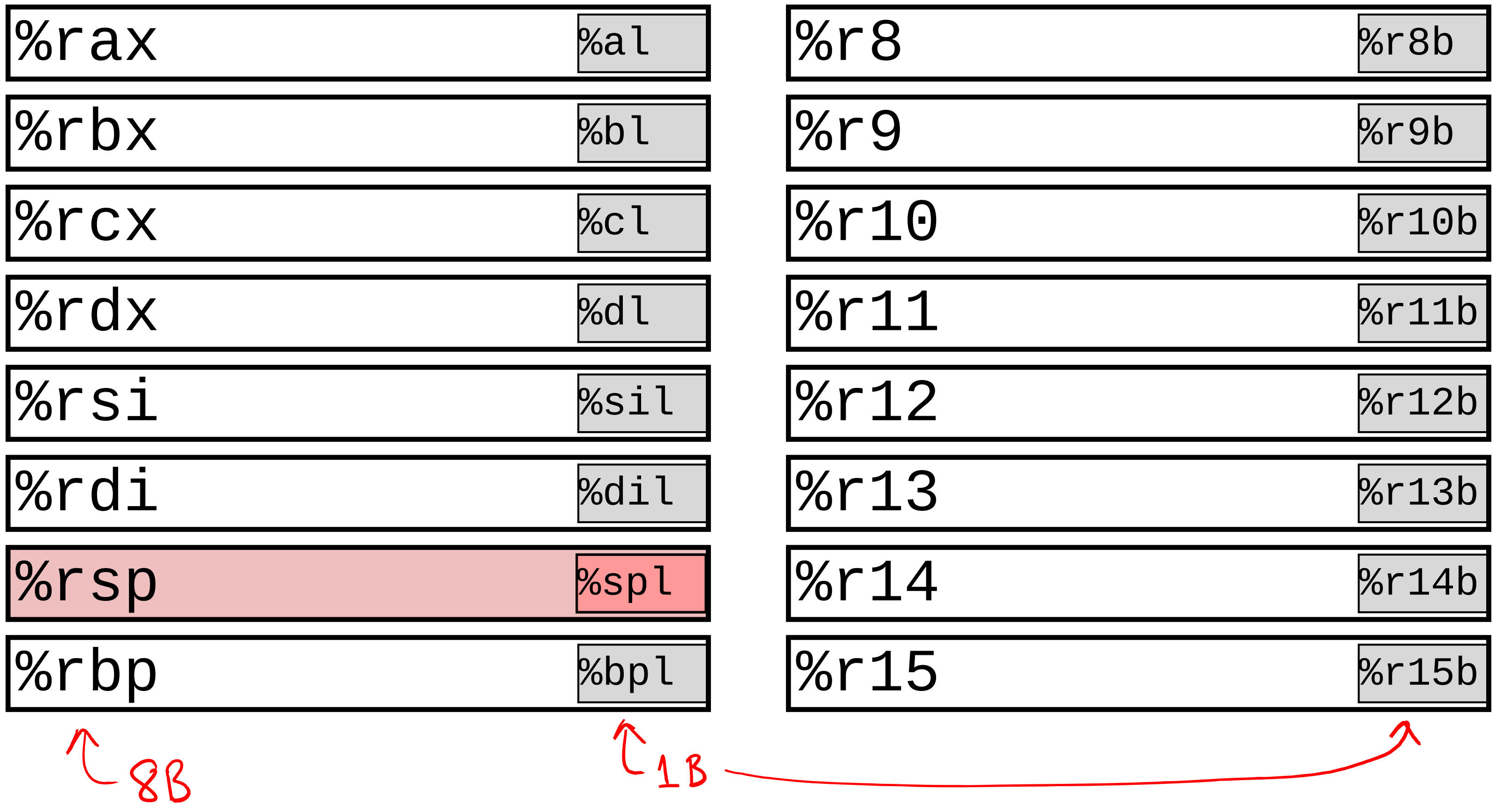
False → 0b 0000 0000 = 0x 00
True → 0b 0000 0001 = 0x 01

Same instruction suffixes as j* instructions!

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

e, ne, g, l, ...

- 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;    //x-y > 0
}
```

a(y), b(x)

will set the condition code

```
cmpq   %rsi, %rdi     # set flags based on x-y
setg   %al            # %al = (x>y)          ← lowest byte
movzbl %al, %eax      # %rax = (x>y)         ← whole register
ret
```

zero-extend →

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

2 width specifiers: b, w, l, q
1 2 4 8 bytes

`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`:

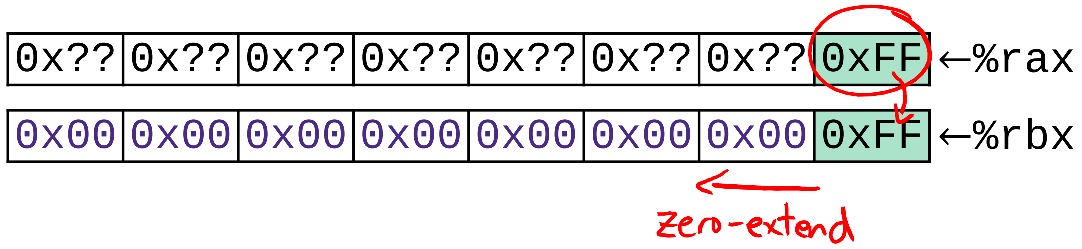
S – size of source (**`b`** = 1 byte, **`w`** = 2)

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

Example:

`movzbq %al, %rbx`

zero-extend
1 byte
8 bytes



0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF ←%rax
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF ←%rbx
zero-extend

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)

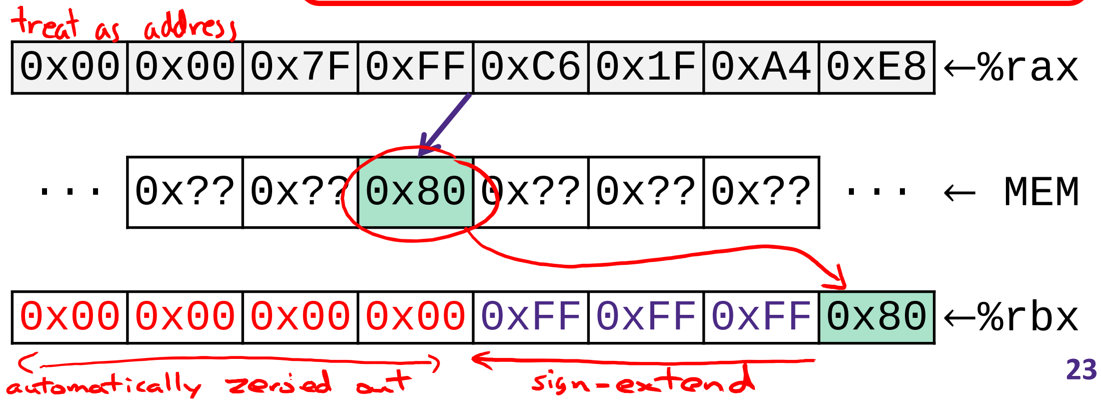
Note: In x86-64, *any instruction* that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.

Example: 1 byte

`movsbl (%rax), %ebx`

sign-extend 4 bytes

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



treat as address
0x00 0x00 0x7F 0xFF 0xC6 0x1F 0xA4 0xE8 ←%rax
... 0x?? 0x?? 0x80 0x?? 0x?? 0x?? ... ← MEM
0x00 0x00 0x00 0x00 0xFF 0xFF 0xFF 0x80 ←%rbx
automatically zeroed out
sign-extend

Summary

- 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