x86-64 Programming II

CSE 351 Summer 2022

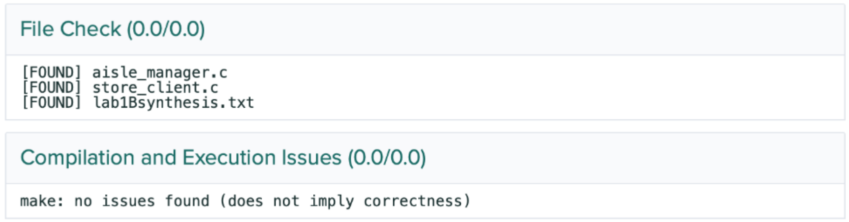
Instructor:
Kyrie Dowling

Teaching Assistants:
Aakash Srazali
Allie Pfleger
Ellis Haker

THE PEOPLE WHO ENCOUNTERED THE SAME BUG AS ME 10 YEARS AGO ON STACKOVERFLOW
ME
imgflip.com

Relevant Course Information

- hw6 due tonight
- hw7 due Wed



File Check (0.0/0.0)

```
[FOUND] aisle_manager.c
[FOUND] store_client.c
[FOUND] lab1Bsynthesis.txt
```

Compilation and Execution Issues (0.0/0.0)

```
make: no issues found (does not imply correctness)
```

- Lab 1b due tonight
  - Submit on Gradescope before 11:59pm
  - Make sure that your code compiles, and that you submitted all the files!

- Lab 2 (x86-64) is out
  - Check out the announcement post on Ed for more details
  - Section this week has more info!

Relevant Course Information

- Unit Portfolio 1 is out
  - Due Friday at 11:59 pm
  - Submit on *Canvas* ***not*** on Gradescope!
  - Low-stakes, record your submission how ever is most comfortable for you
  - More info on the course website

Extra Credit

- All labs starting with Lab 2 have extra credit portions
  - These are meant to be fun extensions to the labs

- Extra credit points *don't* affect your lab grades
  - From the course policies: "they will be accumulated over the course and will be used to bump up borderline grades at the end of the quarter."
  - Make sure you finish the rest of the lab before attempting any extra credit

Arithmetic Example

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

Convention!

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

don't actually need new variables!

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

must return in %rax

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

#return

Example of Basic Addressing Modes

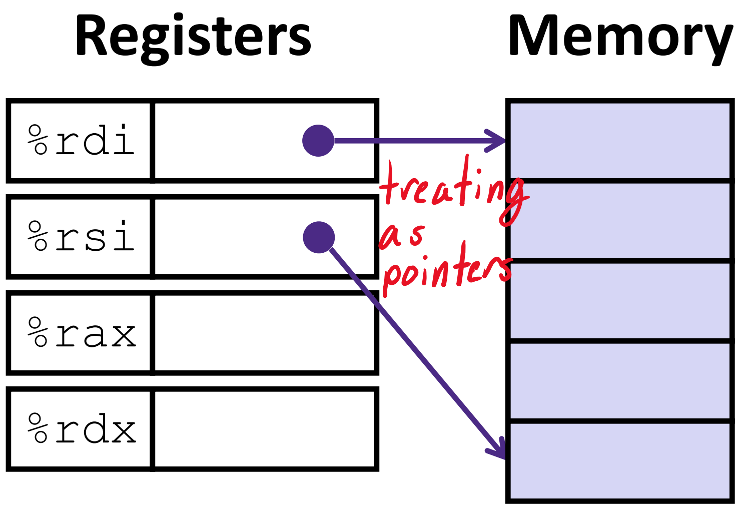
```
void swap(long* xp, long* yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

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

Compiler Explorer:
https://godbolt.org/z/zc4Pcq

Understanding `swap()`

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

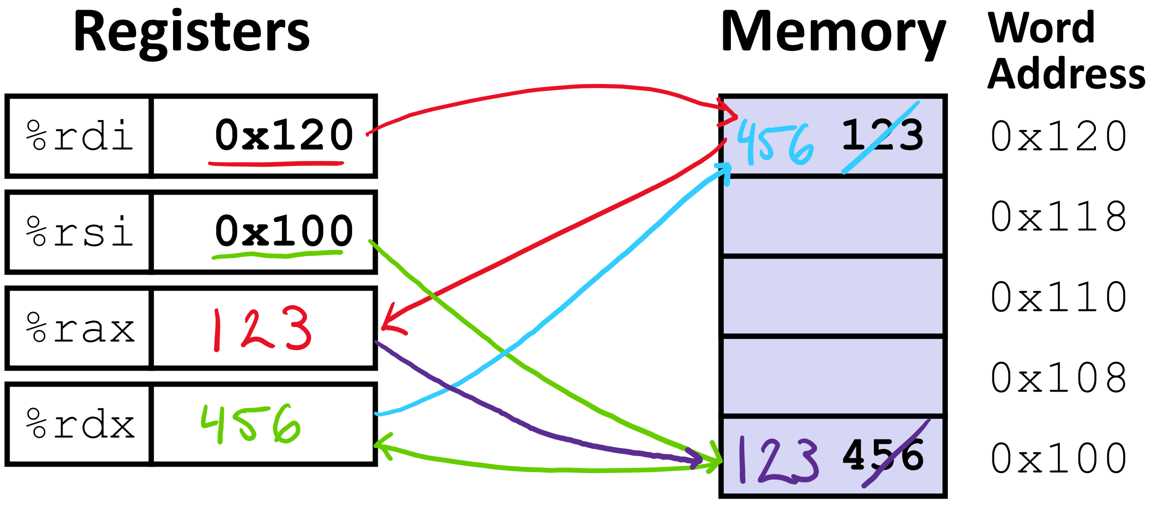


memory operands   register operands

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

Register		**Variable**
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

Understanding swap()



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

Complete Memory Addressing Modes

ar[i] ↔ *(ar+i) → Mem[ar + i * sizeof(data type)]

- **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?* data type width
    - `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)`

↑ so reg name not interpreted as Rb

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

`D(Rb,Ri,S)` → Mem[Reg[`Rb`]+Reg[`Ri`]`*S+D`]

ignore the memory access for now

Expression	Address Computation	**Address** (8 bytes wide)
$\overset{D}{\texttt{0x8}}\texttt{(\%}\overset{Rb}{\texttt{rdx}}\texttt{)}$	Reg[Rb]+D = 0xf000+0x08	0xf008
$\texttt{(\%}\overset{Rb}{\texttt{rdx}}\texttt{,\%}\overset{Ri}{\texttt{rcx}}\texttt{)}$	Reg[Rb]+Reg[Ri]*1	0xf100
$\texttt{(\%}\overset{Rb}{\texttt{rdx}}\texttt{,\%}\overset{Ri}{\texttt{rcx}}\texttt{,}\overset{S}{\texttt{4}}\texttt{)}$	Reg[Rb]+Reg[Ri]*4	0xf400
$\overset{D}{\texttt{0x80}}\texttt{(,\%}\overset{Ri}{\texttt{rdx}}\texttt{,}\overset{S}{\texttt{2}}\texttt{)}$	Reg[Ri]*2 +D	0x1e080

0xf000 *2
0xf000 << 1 = 0x1e000

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

- Questions from the Reading?

Review Questions

no memory access so must be lea

S∈{1, 2, 4, 8}

- Which of the following x86-64 instructions correctly calculates `%rax=9*%rdi`?
  - **A.** `leaq (,%rdi,9), %rax` invalid syntax
  - **B.** `movq (,%rdi,9), %rax` invalid syntax
  - **C.** `leaq (%rdi,%rdi,8), %rax` %rax = 9 * %rdi
  - **D.** `movq (%rdi,%rdi,8), %rax` %rax = Mem[9 * %rdi]

MSB of %si is a 1

- If `%rsi` is `0x B0BACAFE 1EE7 F0 0D`, what is its value after executing **`movswl %si, %esi`**?

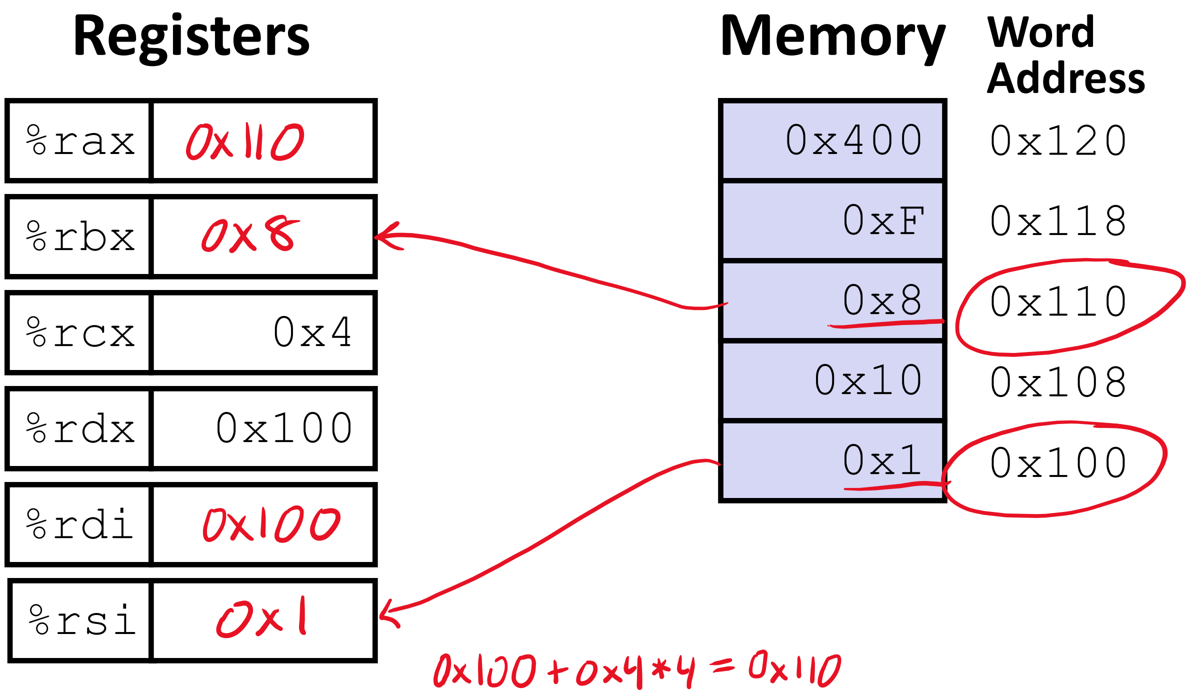
sign extension → destination is 4 bytes; source is 2 bytes

$\underbrace{\text{0x 0000 0000}}_{\text{x86-64 rule for when dst is 32 bits}}\ \underbrace{\text{FFFF}}_{\text{sign extension}}\ \underbrace{\text{F00D}}_{\text{original data}}$

Address Computation Instruction

- `leaq src, dst` ("Mem" over src, Reg over dst)
  - `"lea"` stands for *load effective address*
  - `src` is address expression (any of the formats we've seen)
  - `dst` is a register   calculates Reg[Rb]+Reg[Ri]*S +D (no Mem[ ])
  - Sets `dst` to the *address* computed by the `src` expression (does not go to memory! – it just does math)
  - <u>Example</u>: `leaq (%rdx,%rcx,4), %rax`
- Uses:
  - Computing addresses without a memory reference
    - *e.g.*, translation of `p = &x[i];` ← address of op
  - 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`



```
        Rb     Ri     S
leaq  (%rdx,%rcx,4), %rax   → 0x110  ("addr")
movq  (%rdx,%rcx,4), %rbx   → 0x8    (data)
leaq  (%rdx), %rdi          → 0x100  ("addr")
movq  (%rdx), %rsi          → 0x1    (data)
```

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;      replaced by lea+shift!
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

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

- Interesting Instructions
  - `leaq`: "address" computation
  - `salq`: shift
  - `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, get reused!

```
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
```

Move extension: `movz` and `movs`

2 width specifiers (b, w, l, q)
1 2 4 8

`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`:

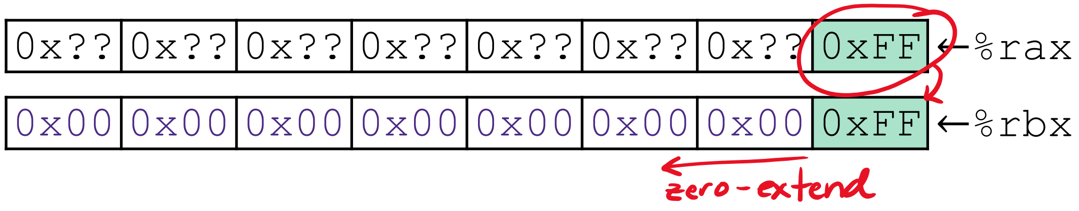
S – size of source (**`b`** = 1 byte, **`w`** = 2)

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

Example: 8 bytes

```
movzbq %al, %rbx
```

zero-extend 1 byte


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

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; 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)

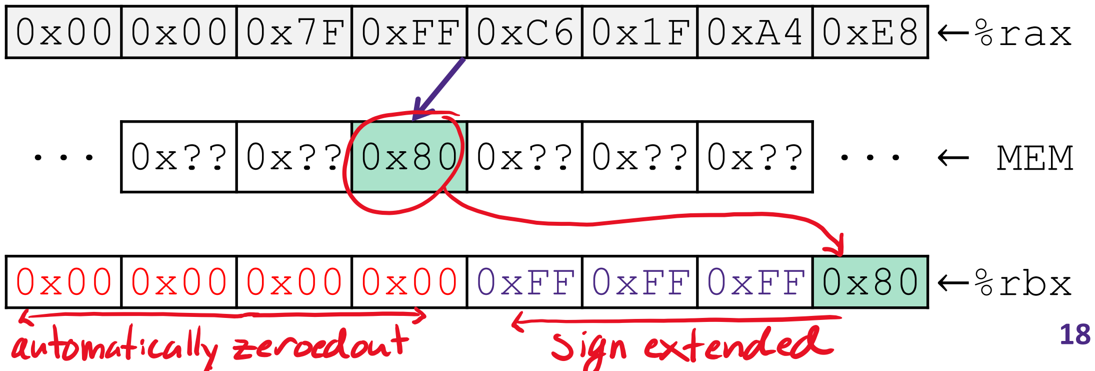
D – size of dest (**w** = 2 bytes, **l** = 4, **q** = 8)

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:

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

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



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 extended

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

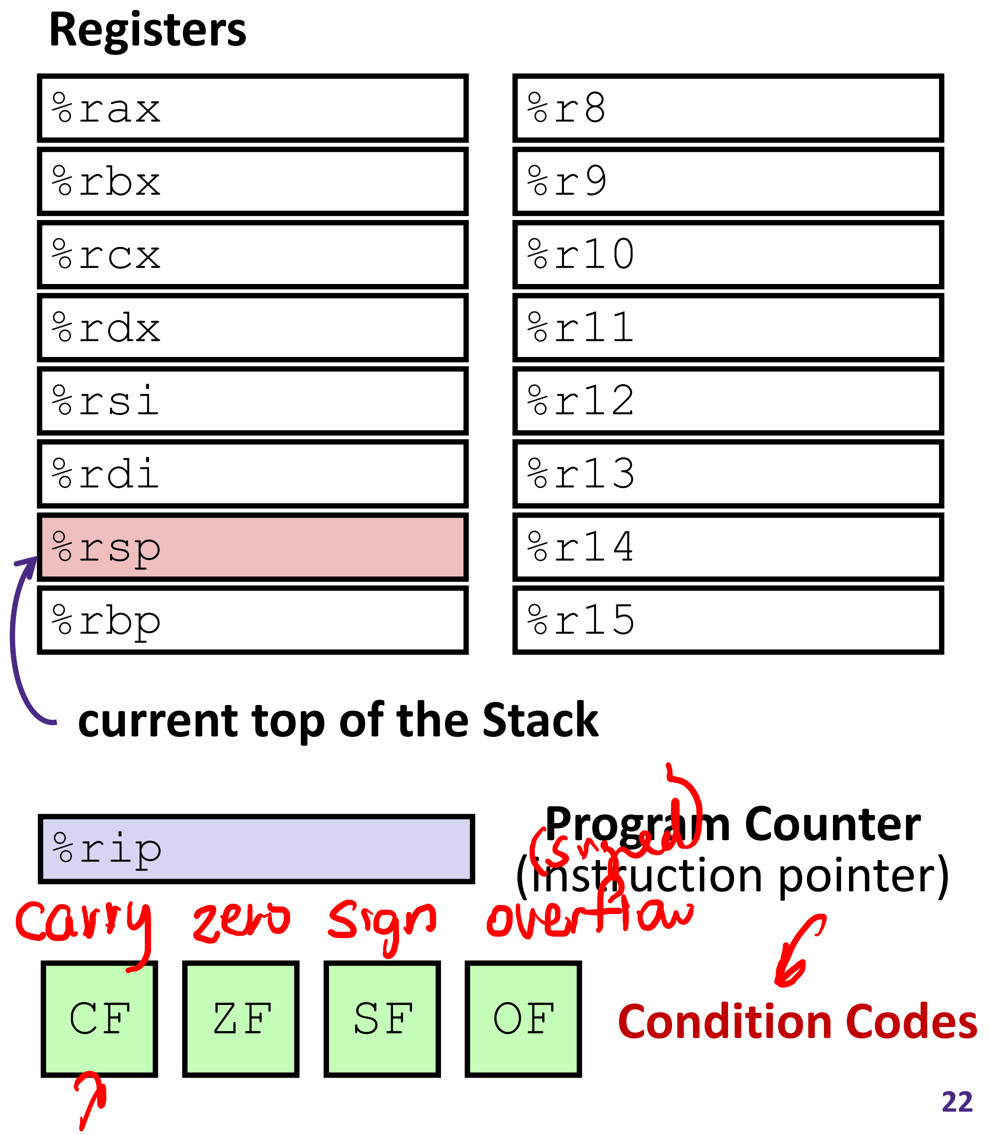
Conditionals and Control Flow

- 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` {…}

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**)
    - 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` $\Leftrightarrow$ subq src1, src2
  - **`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`
  - **`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
    - 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*

Example Condition Code Setting

- Assuming that `%al` = `0x80` and `%bl` = `0x81`, which flags (CF, ZF, SF, OF) are set when we execute **`cmpb %al, %bl`**?

Using Condition Codes: Jumping

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

Instruction	Condition	Description
`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 “<”)

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