x86-64 Programming II

CSE 351 Winter 2021

Instructor:

Mark Wyse

Teaching Assistants:

Kyrie Dowling

Catherine Guevara

Ian Hsiao

Jim Limprasert

Armin Magness

Allie Pfleger

Cosmo Wang

Ronald Widjaja

http://xkcd.com/99/

Administrivia

- Lab 2 (x86-64) released today, due 2/5
  - Learn to read x86-64 assembly and use GDB
- Lecture readings – due at 11:00am PST
- Submissions that fail the autograder get a **ZERO**
  - No excuses – make full use of tools & Gradescope's interface
  - Some leeway was given on Lab 1, do not expect the same leniency moving forward
- hw8 due Wednesday, hw9 due Friday
- Study Guide 1 – due Friday 1/29

Extra Credit

- All labs starting with Lab 2 have extra credit portions
  - These are meant to be fun extensions to the labs
- Study Guides Task 1 and 2 can also be awarded extra credit, although this will be uncommon

- Extra credit points *don't* affect your lab/guide 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

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?

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?* – b, w, l, q
    - `D:` Constant displacement value (a.k.a. immediate)

(rb)

- **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 Instruction

(%rbx, %rcx) → mem[rbx+rcx]

- `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)
  - <u>Example</u>: `leaq (%rdx,%rcx,4), %rax` rax = rdx + (rcx*4)
- 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
```

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
```

Review Questions

- Which of the following x86-64 instructions correctly calculates %rax=9*%rdi? → not referencing memory

  D(Rb, Ri, S)

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

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

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

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

  (Rb + S*Ri)+D

  S ∈ 1,2,4,8

  rdi + 8*rdi = 9*rdi

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

  sign extend

  2 bytes %si ⇒ %rsi

  ↳ ⇒ %rsi, 4 low bytes

  0x FFFF F00D

  x86 convention → zero upper 32-bits of %rsi

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

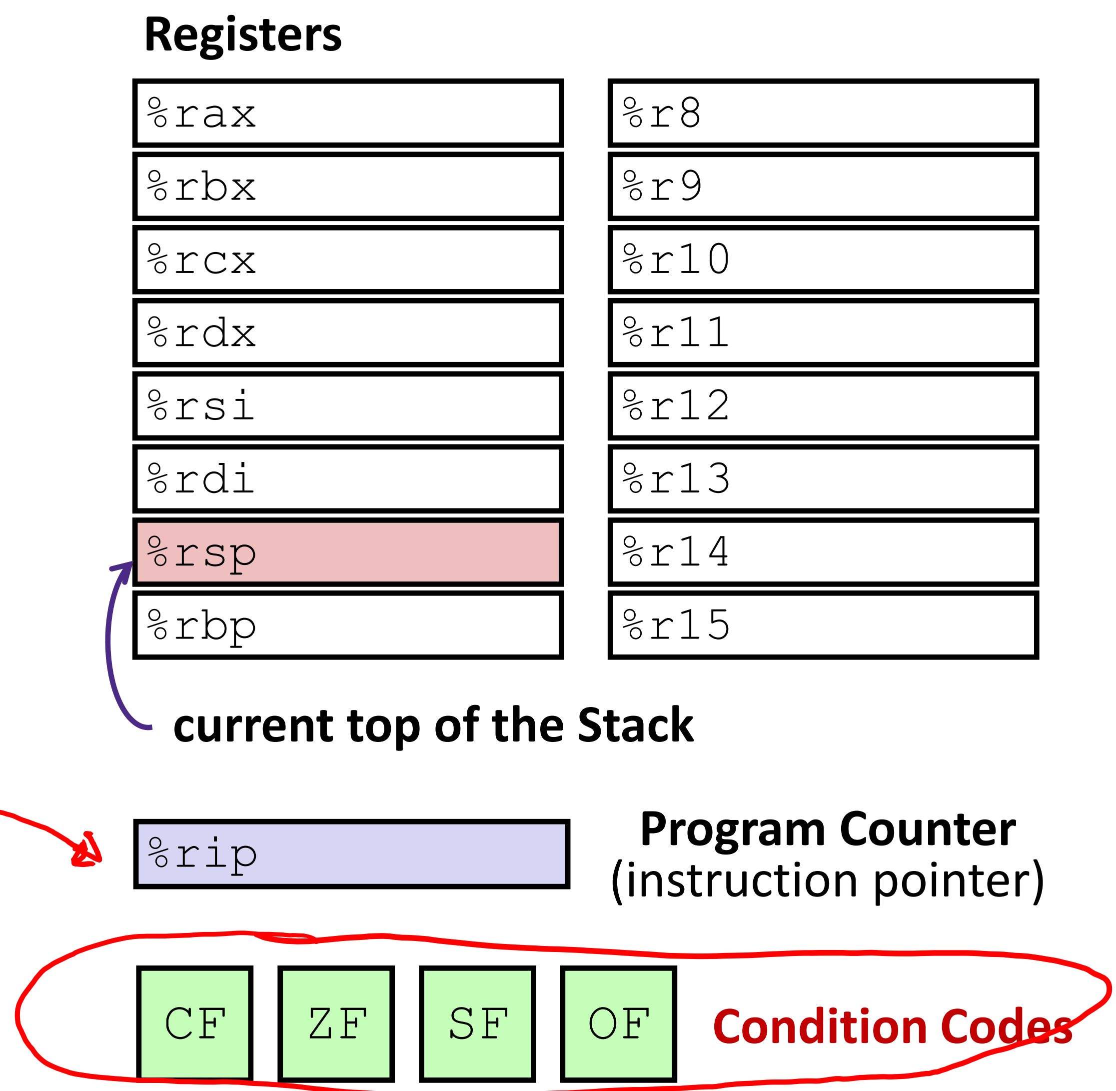
- Together, they can implement most control flow constructs in high-level languages:
  - **`if`** (*condition*) **`then`** {…} **`else`** {…}
  - **`while`** (*condition*) {…}
  - **`do`** {…} **`while`** (*condition*)
  - **`for`** (*initialization*; *condition*; *iterative*) {…}
  - **`switch`** {…}

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`**)
    - 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` → sub
  - **`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` → and
  - **`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**) → CF = 0, 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

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

e.g. generated by a sub instr.

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 "<")

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`) 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`) 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`)

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

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)

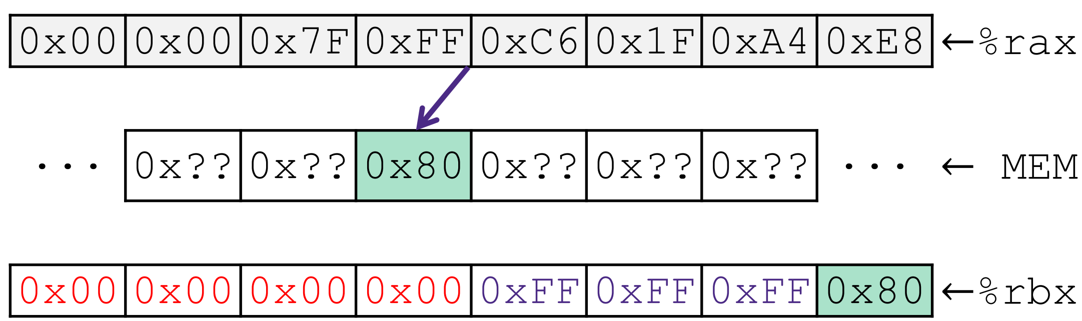
D – size of dest (**w** = 2 bytes, **l** = 4, **q** = 8)

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

- 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