x86-64 Programming II

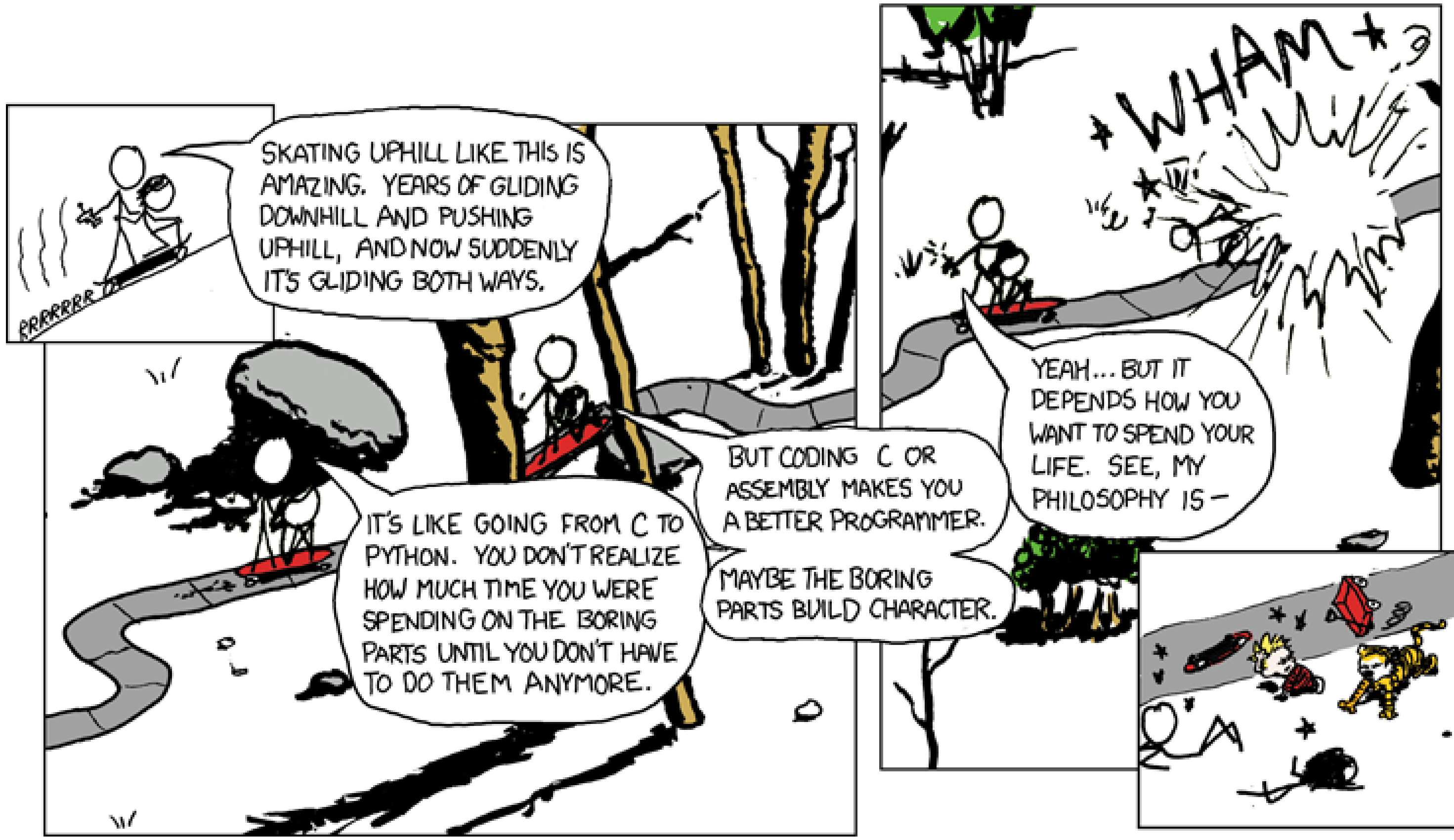
CSE 351 Autumn 2018

Instructor:

Justin Hsia

Teaching Assistants:

Akshat Aggarwal
An Wang
Andrew Hu
Brian Dai
Britt Henderson
James Shin
Kevin Bi
Kory Watson
Riley Germundson
Sophie Tian
Teagan Horkan



http://xkcd.com/409/

Administrivia

- Lab 2 (x86-64) released tonight
  - Learn to read x86-64 assembly and use GDB
- Homework 2 due Friday (10/19)

- Midterm is in two Mondays (10/29, 5 pm in KNE 120)
  - No lecture that day
  - You will be provided a fresh reference sheet
    - Study and use this NOW so you are comfortable with it when the exam comes around
  - You get 1 *handwritten*, double-sided cheat sheet (letter)
  - Find a study group! Look at past exams!

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 ~~Mem[ ]~~
  - `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];` 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

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

0x100 + 0x4 * 4 = $\boxed{0x110}$

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)
```

0x100

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
  ret
```

multiplying two variables

- 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

limited registers means they often get reused!

```
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;
}
```

comment (AT&T syntax)

```
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
```

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

Peer Instruction Question

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

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

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

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

C. `leaq (%rdi,%rdi,8), %rax` → %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:
  ???
  movq    %rdi, %rax   # if
  ???
  ???
  movq    %rsi, %rax   # else
  ???
  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
                if FALSE
   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:

Registers

`%rax`	`%r8`
`%rbx`	`%r9`
`%rcx`	`%r10`
`%rdx`	`%r11`
`%rsi`	`%r12`
`%rdi`	`%r13`
`%rsp`	`%r14`
`%rbp`	`%r15`

current top of the Stack

`%rip` **Program Counter** (instruction pointer)

Carry Zero Sign Overflow

CF	ZF	SF	OF

Condition Codes

Condition Codes (Implicit Setting)

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

    result = dst + src

  - **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)`

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

example if %eax holds 0x 80 00 00 00:

addl %eax, %eax # 0x0 stored in %eax

CF = 1

ZF = 1

SF = 0

OF = 1 (⊖ + ⊖ = ⊕)

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` like subq a, b → 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 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** = 0) or overflow (**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
  - 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

False → 0b 0000 0000 = 0x00
True → 0b 0000 0001 = 0x01

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

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:

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

8B

1B

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)

```
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:

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

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

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

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