x86 Programming II

CSE 351 Autumn 2016

Instructor:

Justin Hsia

Teaching Assistants:

Chris Ma

Hunter Zahn

John Kaltenbach

Kevin Bi

Sachin Mehta

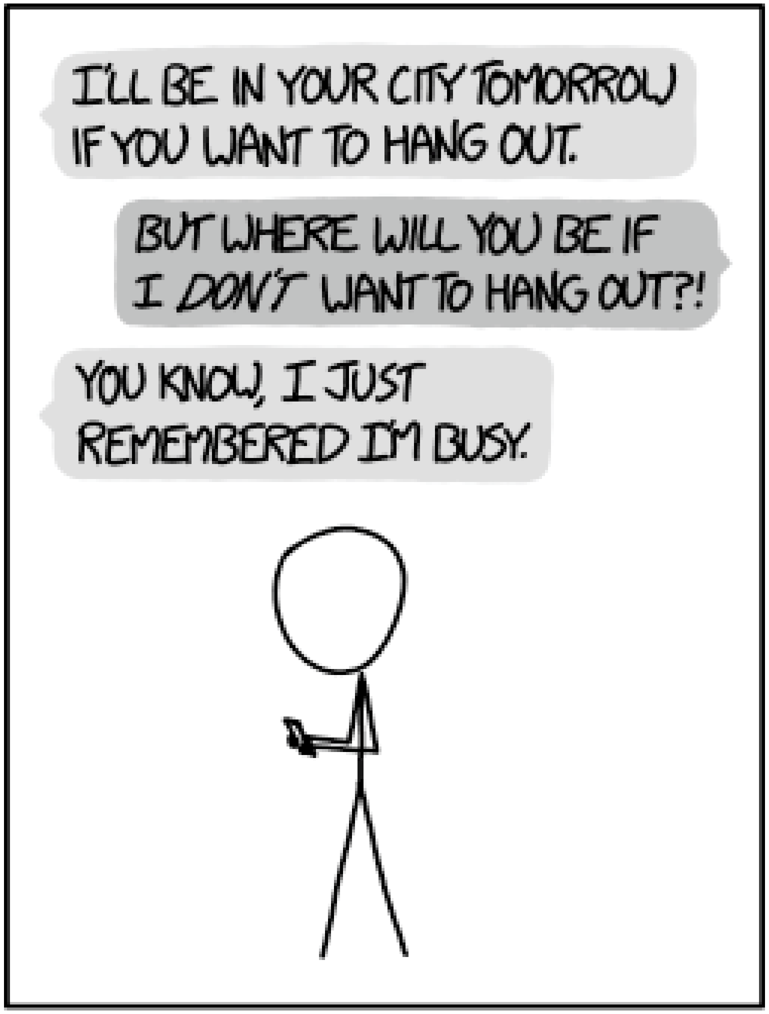
Suraj Bhat

Thomas Neuman

Waylon Huang

Xi Liu

Yufang Sun



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Administrivia

- Lab 2 released tomorrow
  - Learn to use gdb and look at assembly code
- Homework 1 due on Friday (10/21)

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`
    - Though k can only be 1, 2, 4, or 8

Example: `lea` vs. `mov`

Registers

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

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
```

x86 Control Flow

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

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`** {…}

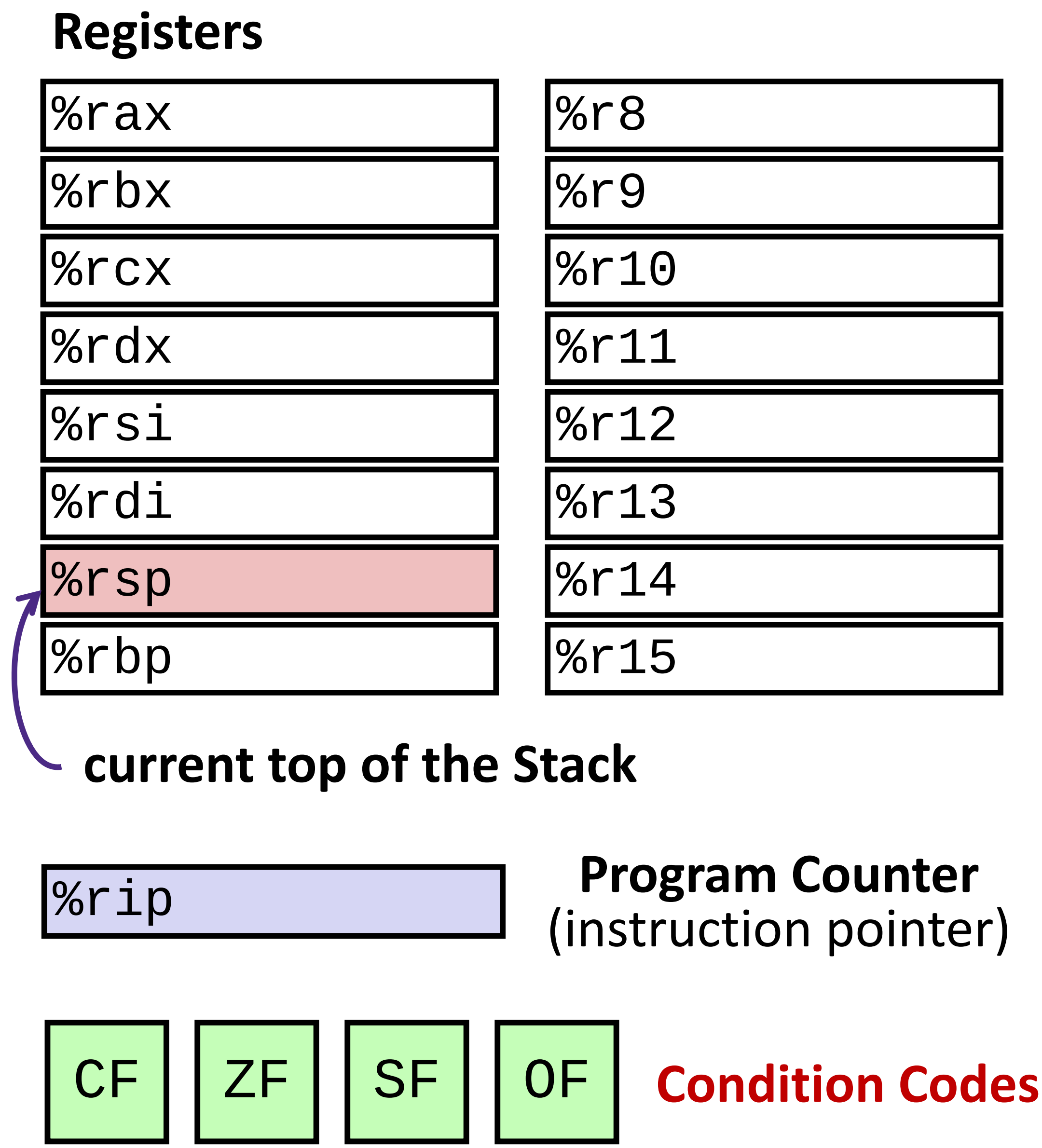
Jumping

- `j*` Instructions
  - Jumps to ***target*** (argument – actually just an address)
  - Conditional jump relies on special *condition code registers*

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)

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` ↔ `t = a+b`

  - **`CF=1`** if carry out from MSB (unsigned overflow)
  - **`ZF=1`** if `t==0`
  - **`SF=1`** if `t<0` (assuming signed, actually just if MSB is 1)
  - **`OF=1`** if two's complement (signed) overflow
    `(a>0 && b>0 && t<0)||(a<0 && b<0 && t>=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 src2, src1`
  - `cmpq b, a` sets flags based on `a-b`, but doesn't store

  - **CF=1** if carry out from MSB (used for unsigned comparison)
  - **ZF=1** if `a==b`
  - **SF=1** if `(a-b)<0` (signed)
  - **OF=1** if two's complement (signed) overflow
    `(a>0 && b<0 && (a-b)<0) ||`
    `(a<0 && b>0 && (a-b)>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 b, a` 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)

  - Example: `testq %rax, %rax`
    - Tells you if (+), 0, or (−) based on ZF and SF

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

Reading Condition Codes

- `set*` Instructions
  - Set a low-order byte 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 “<”)

x86-64 Integer Registers

- Accessing the low-order byte:

Register	Low byte	Register	Low 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

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`)

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`)

`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

Choosing instructions for conditionals

		`cmp b,a`	`test a,b`
`je`	"Equal"	a == b	a&b == 0
`jne`	"Not equal"	a != b	a&b != 0
`js`	"Sign" (negative)		a&b < 0
`jns`	(non-negative)		a&b >= 0
`jg`	"Greater"	a > b	a&b > 0
`jge`	"Greater or equal"	a >= b	a&b >= 0
`jl`	"Less"	a < b	a&b < 0
`jle`	"Less or equal"	a <= b	a&b <= 0
`ja`	"Above" (unsigned >)	a > b	
`jb`	"Below" (unsigned <)	a < b	

```
      cmp 5,(p)
je:   *p == 5
jne:  *p != 5
jg:   *p >  5
jl:   *p <  5
```

```
      test a,a
je:    a == 0
jne:   a != 0
jg:    a >  0
jl:    a <  0
```

```
      test a,0x1
```
je: a_{LSB} == 0

jne: a_{LSB} == 1

Choosing instructions for conditionals

		`cmp b,a`	`test a,b`
`je`	"Equal"	a == b	a&b == 0
`jne`	"Not equal"	a != b	a&b != 0
`js`	"Sign" (negative)		a&b < 0
`jns`	(non-negative)		a&b >= 0
`jg`	"Greater"	a > b	a&b > 0
`jge`	"Greater or equal"	a >= b	a&b >= 0
`jl`	"Less"	a < b	a&b < 0
`jle`	"Less or equal"	a <= b	a&b <= 0
`ja`	"Above" (unsigned >)	a > b	
`jb`	"Below" (unsigned <)	a < b	

Register	Use(s)
`%rdi`	argument x
`%rsi`	argument y
`%rax`	return value

```
if (x < 3) {
  return 1;
}
return 2;
```

```
  cmpq $3, %rdi
  jge T2
T1: # x < 3:
  movq $1, %rax
  ret
T2: # !(x < 3):
  movq $2, %rax
  ret
```

Your Turn!

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

```
long absdiff(long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:

    ___________________

    ___________________
                            # x > y:
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                        # x <= y:
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

- Can view in provided `control.s`
  - `gcc –Og -S –fno-if-conversion control.c`

Choosing instructions for conditionals

		`cmp b,a`	`test a,b`
`je`	"Equal"	`a == b`	`a&b == 0`
`jne`	"Not equal"	`a != b`	`a&b != 0`
`js`	"Sign" (negative)		`a&b < 0`
`jns`	(non-negative)		`a&b >= 0`
`jg`	"Greater"	`a > b`	`a&b > 0`
`jge`	"Greater or equal"	`a >= b`	`a&b >= 0`
`jl`	"Less"	`a < b`	`a&b < 0`
`jle`	"Less or equal"	`a <= b`	`a&b <= 0`
`ja`	"Above" (unsigned >)	`a > b`	
`jb`	"Below" (unsigned <)	`a < b`	

```
if (x < 3 && x == y) {
  return 1;
} else {
  return 2;
}
```

```
  cmpq $3, %rdi
  setl %al
  cmpq %rsi, %rdi
  sete %bl
  testb %al, %bl
  je T2
T1: # x < 3 && x == y:
  movq $1, %rax
  ret
T2: # else
  movq $2, %rax
  ret
```

Summary

- `lea` is address calculation instruction
  - Does NOT actually go to memory
  - Used to compute addresses or some arithmetic expressions
- 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