x86-64 Programming II

CSE 351 Winter 2018

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan

http://xkcd.com/409/

Administrative

- Homework 2 (Ints and Floats) due Today!
  - By 11:59 pm, no late submissions!
  - x86-64 part due 1/29
- GDB Tutorial Session
  - Tomorrow, Thursday 1/25, 5 – 6 pm
  - Room : SIG 134

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** {…}

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:

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)

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`
  - **CF=1** if carry out from MSB (unsigned overflow)
  - **ZF=1** if `r==0`
  - **SF=1** if `r<0` (assuming signed, actually just if MSB is 1)
  - **OF=1** if two's complement (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`
  - **`cmpq`** `a, b` sets flags based on `b-a`, but doesn't store
  - **CF=1** if carry out from MSB (used for unsigned comparison)
  - **ZF=1** if `a==b`
  - **SF=1** if `(b-a)<0` (signed)
  - **OF=1** if two's complement (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***
  - **ZF=1** if `a&b==0`
  - **SF=1** if `a&b<0` (signed)
  - **CF** and **OF** set to 0
  - 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*

Using Condition Codes: Jump

- `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: Set

- `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 "<")

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

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 to y (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

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

Using Condition Codes: Jump

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

Instruction	Condition	Description
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 "<")

Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation (op)
  - Conditionals are comparisons against 0

```
addq 5, (p)
*p = *p + 5;
```

```
addq 5, (p)
je:   *p+5 == 0
jne:  *p+5 != 0
jg:   *p+5 >  0
jl:   *p+5 <  0
```

addq (op) s, d

je	"Equal"	d (op) s == 0
jne	"Not equal"	d (op) s != 0
js	"Sign" (negative)	d (op) s < 0
jns	(non-negative)	d (op) s >= 0
jg	"Greater"	d (op) s > 0
jge	"Greater or equal"	d (op) s >= 0
jl	"Less"	d (op) s < 0
jle	"Less or equal"	d (op) s <= 0
ja	"Above" (unsigned >)	d (op) s > 0U
jb	"Below" (unsigned <)	d (op) s < 0U

Choosing instructions for conditionals

- Reminder: `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

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

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

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

```
testb a, 0x1
je:   a_LSB == 0
jne:  a_LSB == 1
```


Choosing instructions for conditionals

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

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

%al %bl

If either %al or %bl are false

```
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
```


Choosing instructions for conditionals

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

```
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
```

- https://godbolt.org/g/KntpyG

Question

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
```

A. `cmpq %rsi, %rdi` `jle .L4`

B. `cmpq %rsi, %rdi` `jg .L4`

C. `testq %rsi, %rdi` `jle .L4`

D. `testq %rsi, %rdi` `jg .L4`

E. We're lost…

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