

x86-64 Programming II

CSE 351 Winter 2023

Instructor:

Sam Wolfson

Teaching Assistants:

Aman Mohammed

Angela Xu

Armin Magness

Clare Edmonds

David Dai

Jenny Peng

Maggie Jiang

Mara Kirdani-Ryan

Nayha Auradkar

Yoonseo Song



Relevant Course Information

- ❖ hw6 due tonight
- ❖ Lab 2 (x86-64) released today
 - Learn to trace x86-64 assembly and use GDB
 - We'll give some tips & tricks in section this week

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

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

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

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

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

Example of Basic Addressing Modes

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

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

Compiler Explorer:

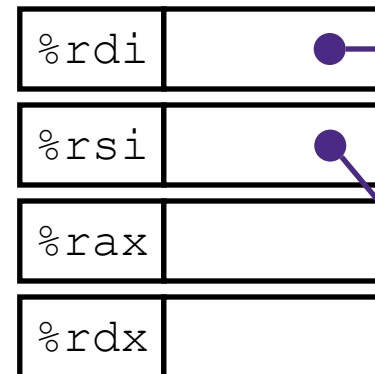
<https://godbolt.org/z/zc4Pcq>

Understanding `swap()`

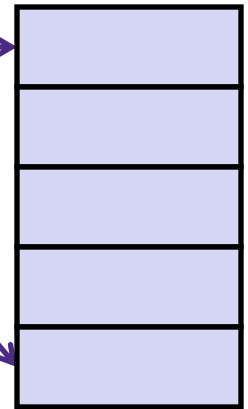
```
void swap(long* xp, long* yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

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

Registers



Memory



Register

Variable

| | | |
|------|---|----|
| %rdi | ⇔ | xp |
| %rsi | ⇔ | yp |
| %rax | ⇔ | t0 |
| %rdx | ⇔ | t1 |

Understanding `swap()`

Registers

| | |
|-------------------|--------------------|
| <code>%rdi</code> | <code>0x120</code> |
| <code>%rsi</code> | <code>0x100</code> |
| <code>%rax</code> | |
| <code>%rdx</code> | |

Memory

| Word Address |
|--------------|
| 123 |
| |
| |
| |
| 456 |

0x120

0x118

0x110

0x108

0x100

```
swap:
```

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Complete Memory Addressing Modes

❖ General:

- $D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$
 - Rb : Base register (any register)
 - Ri : Index register (any register except `%rsp`)
 - S : Scale factor (1, 2, 4, 8) – *why these numbers?*
 - D : Constant displacement value (a.k.a. immediate)

❖ Special cases (see CSPP Figure 3.3 on p.181)

- $D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \quad (S=1)$
- $(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S] \quad (D=0)$
- $(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \quad (S=1, D=0)$
- $(, Ri, S) \quad \text{Mem}[\text{Reg}[Ri] * S] \quad (Rb=0, D=0)$

Address Computation Examples

| | |
|-------------------|---------------------|
| <code>%rdx</code> | <code>0xf000</code> |
| <code>%rcx</code> | <code>0x0100</code> |

$D(Rb, Ri, S) \rightarrow$

$\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$

↑ ignore the memory access for now

| Expression | Address Computation | Address (8 bytes wide) |
|----------------------------|---------------------|---------------------------|
| <code>0x8(%rdx)</code> | | |
| <code>(%rdx,%rcx)</code> | | |
| <code>(%rdx,%rcx,4)</code> | | |
| <code>0x80(,%rdx,2)</code> | | |

Reading Review

❖ Terminology:

- Address Computation Instruction (`leaq`)
- 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

- ❖ Which of the following x86-64 instructions correctly calculates `%rax=9*%rdi`?
 - A. `leaq (,%rdi,9), %rax`
 - B. `movq (,%rdi,9), %rax`
 - C. `leaq (%rdi,%rdi,8), %rax`
 - D. `movq (%rdi,%rdi,8), %rax`
- ❖ If `%rsi` is `0x B0BACAFE 1EE7 F0 0D`, what is its value after executing `movswl %si, %esi`?

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 $x + k * i + d$
 - Though `k` can only be 1, 2, 4, or 8

Example: lea vs. mov

Registers

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

Memory 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

```
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
```

| Register | Use(s) |
|----------|------------------------------|
| %rdi | 1 st argument (x) |
| %rsi | 2 nd argument (y) |
| %rdx | 3 rd argument (z) |

❖ 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 |

```

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`

`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:

`movzbq %al, %rbx`

| | | | | | | | | |
|------|------|------|------|------|------|------|------|-------|
| 0x?? | 0x?? | 0x?? | 0x?? | 0x?? | 0x?? | 0x?? | 0xFF | ←%rax |
| 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0xFF | ←%rbx |

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

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

Conditionals and Control Flow

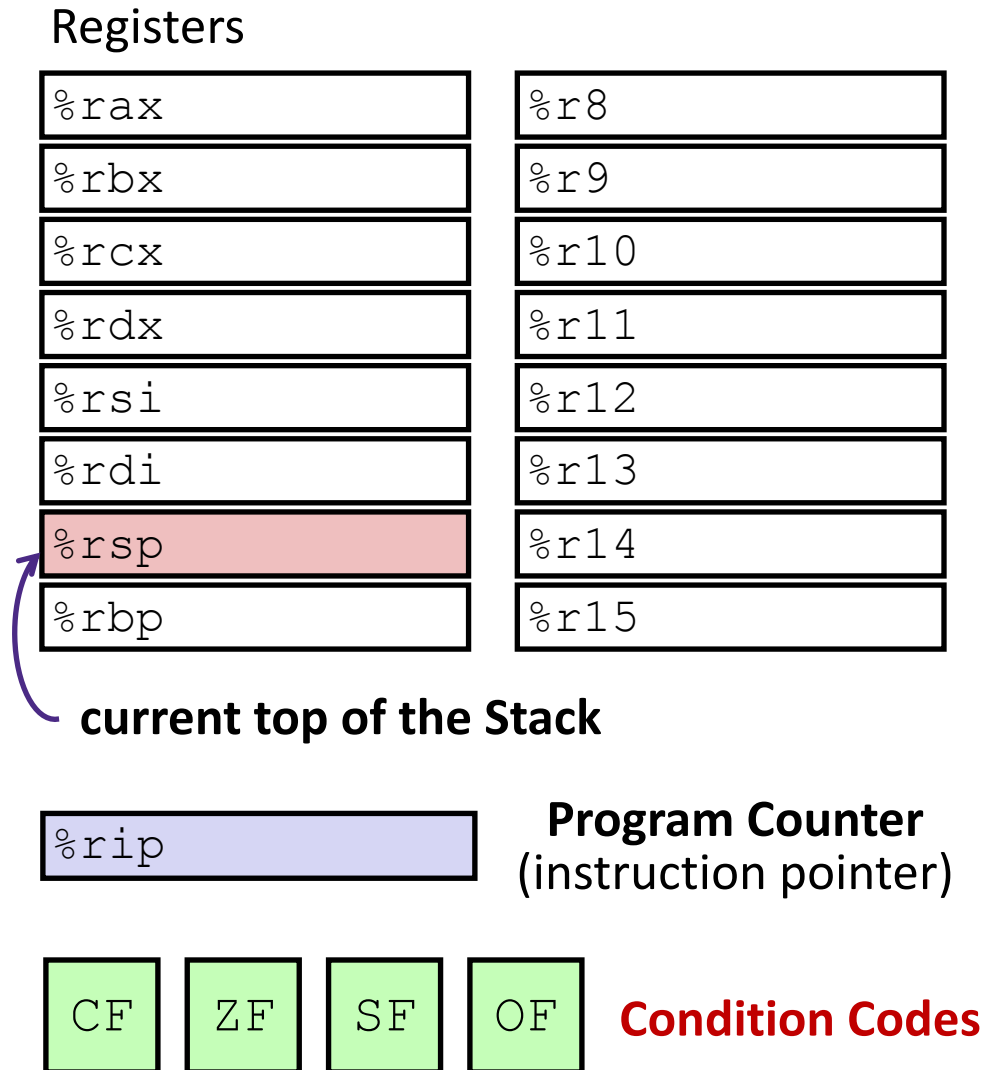
- ❖ Conditional branch/*jump*
 - Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- ❖ Unconditional branch/*jump*
 - *Always* jump when you get to this instruction
- ❖ Together, they can implement most control flow constructs in high-level languages:
 - **if** (*condition*) **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)

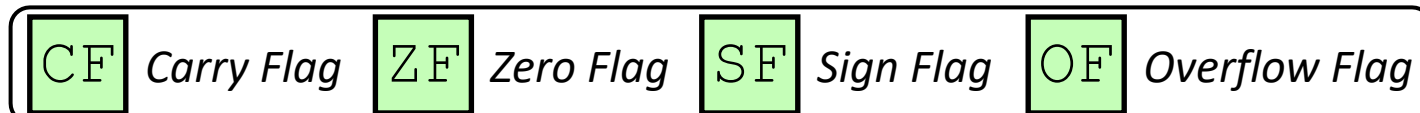
- ❖ Information about currently executing program
 - Temporary data (`%rax`, ...)
 - Location of runtime stack (`%rsp`)
 - Location of current code control point (`%rip`, ...)
 - Status of recent tests (**CF**, **ZF**, **SF**, **OF**)
 - Single bit registers:



Condition Codes (Implicit Setting)

❖ *Implicitly* set by **arithmetic** operations

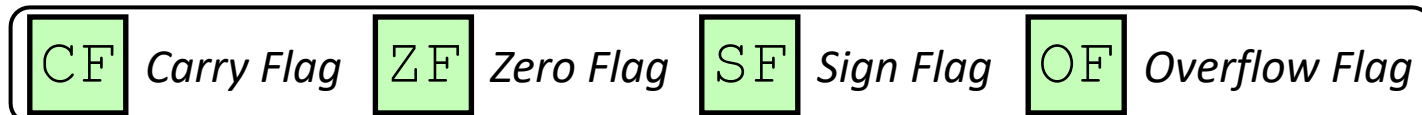
- (think of it as side effects)
- Example: **addq** src, dst \leftrightarrow $r = d + s$
- **CF=1** if carry out from MSB (*unsigned* overflow)
- **ZF=1** if $r == 0$
- **SF=1** if $r < 0$ (if MSB is 1)
- **OF=1** if *signed* overflow
 $(s > 0 \ \&\& \ d > 0 \ \&\& \ r < 0) \ || \ (s < 0 \ \&\& \ d < 0 \ \&\& \ r \geq 0)$
- *Not set by lea instruction (beware!)*



Condition Codes (Explicit Setting: Compare)

❖ Explicitly set by **Compare** instruction

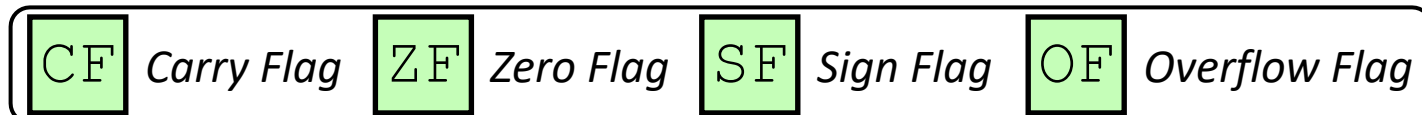
- **cmpq** src1, src2
- **cmpq** a, b sets flags based on $b-a$, but doesn't store
- **CF=1** if carry out from MSB (good for *unsigned* comparison)
- **ZF=1** if $a==b$
- **SF=1** if $(b-a) < 0$ (if MSB is 1)
- **OF=1** if *signed* overflow
 $(a > 0 \ \&\& \ b < 0 \ \&\& \ (b-a) > 0) \ ||$
 $(a < 0 \ \&\& \ b > 0 \ \&\& \ (b-a) < 0)$



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)



Example Condition Code Setting

- ❖ Assuming that `%a1 = 0x80` and `%b1 = 0x81`, which flags (CF, ZF, SF, OF) are set when we execute **`cmpb %a1, %b1`**?

Using Condition Codes: Jumping

❖ j^* Instructions

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

| Instruction | Condition | Description |
|-------------------------------------|--------------------------------------|---------------------------|
| jmp <i>target</i> | 1 | Unconditional |
| je <i>target</i> | ZF | Equal / Zero |
| jne <i>target</i> | \sim ZF | Not Equal / Not Zero |
| js <i>target</i> | SF | Negative |
| jns <i>target</i> | \sim SF | Nonnegative |
| jg <i>target</i> | $\sim (SF \wedge OF) \ \& \ \sim ZF$ | Greater (Signed) |
| jge <i>target</i> | $\sim (SF \wedge OF)$ | Greater or Equal (Signed) |
| jl <i>target</i> | $(SF \wedge OF)$ | Less (Signed) |
| jle <i>target</i> | $(SF \wedge OF) \mid ZF$ | Less or Equal (Signed) |
| ja <i>target</i> | $\sim CF \ \& \ \sim ZF$ | Above (unsigned ">") |
| jb <i>target</i> | 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 <i>dst</i> | ZF | Equal / Zero |
| setne <i>dst</i> | \sim ZF | Not Equal / Not Zero |
| sets <i>dst</i> | SF | Negative |
| setns <i>dst</i> | \sim SF | Nonnegative |
| setg <i>dst</i> | $\sim (SF \wedge OF) \ \& \ \sim ZF$ | Greater (Signed) |
| setge <i>dst</i> | $\sim (SF \wedge OF)$ | Greater or Equal (Signed) |
| setl <i>dst</i> | $(SF \wedge OF)$ | Less (Signed) |
| setle <i>dst</i> | $(SF \wedge OF) \ \ ZF$ | Less or Equal (Signed) |
| seta <i>dst</i> | $\sim CF \ \& \ \sim ZF$ | Above (unsigned ">") |
| setb <i>dst</i> | CF | Below (unsigned "<") |

Reading Condition Codes

| Register | Use(s) |
|----------|------------------------------|
| %rdi | 1 st argument (x) |
| %rsi | 2 nd 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 | 1 st argument (x) |
| %rsi | 2 nd 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
```

Choosing instructions for conditionals

- ❖ All arithmetic instructions set condition flags based on result of operation (op)
 - Conditionals are comparisons against 0
- ❖ Come in instruction *pairs*

```

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

```

```

orq a, b
je:    b|a == 0
jne:   b|a != 0
jg:    b|a > 0
jl:    b|a < 0

```

| | | (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) | $b - a \geq 0$ | $b \& a \geq 0$ |
| jg | "Greater" | $b > a$ | $b \& a > 0$ |
| jge | "Greater or equal" | $b \geq a$ | $b \& a \geq 0$ |
| jl | "Less" | $b < a$ | $b \& a < 0$ |
| jle | "Less or equal" | $b \leq a$ | $b \& a \leq 0$ |
| ja | "Above" (unsigned >) | $b >_U a$ | $b \& a > 0U$ |
| jb | "Below" (unsigned <) | $b <_U a$ | $b \& a < 0U$ |

```

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 |
|--------------------------------|----------------|-----------------|
| jje "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) | $b - a \geq 0$ | $b \& a \geq 0$ |
| jg "Greater" | $b > a$ | $b \& a > 0$ |
| jge "Greater or equal" | $b \geq a$ | $b \& a \geq 0$ |
| jl "Less" | $b < a$ | $b \& a < 0$ |
| jle "Less or equal" | $b \leq a$ | $b \& a \leq 0$ |
| ja "Above" (unsigned >) | $b >_U a$ | $b \& a > 0_U$ |
| jb "Below" (unsigned <) | $b <_U a$ | $b \& a < 0_U$ |

| 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

```


Control Flow

| Register | Use(s) |
|----------|------------------------------|
| %rdi | 1 st argument (x) |
| %rsi | 2 nd 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
```

Practice Question 1

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

- 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...

```

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

```