x86-64 Programming III
CSE 351 Summer 2018

Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee
Natalie Andreeva
Teagan Horkan

http://xkcd.com/1652/
 Administrivia

- Homework 2 due Wednesday (7/11)
- Lab 2 (x86-64) due next Monday (7/16)
  - Learn to read x86-64 assembly and use GDB
- Midterm is next Wednesdays (7/18 in lecture)
  - 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!
GDB Demo

- Examine the `movz` and `movs` examples from last lecture on a real machine!
  - `movzbq %al, %rbx`
  - `movsbl (%rax), %ebx`

- You will need to use GDB to get through Lab 2
  - Useful debugger in this class and beyond!

- Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step/next` and `stepe/nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining memory (`x`)
Choosing instructions for conditionals

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

| Instruction | Condition | Example
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>d (op) s == 0</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>d (op) s != 0</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>d (op) s &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>d (op) s &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>d (op) s &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>d (op) s &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>d (op) s &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>d (op) s &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>d (op) s &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>d (op) s &lt; 0U</td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

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

<table>
<thead>
<tr>
<th></th>
<th><code>cmp a, b</code></th>
<th><code>test a, b</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code></td>
<td>“Equal”</td>
<td><code>b == a</code></td>
</tr>
<tr>
<td><code>jne</code></td>
<td>“Not equal”</td>
<td><code>b != a</code></td>
</tr>
<tr>
<td><code>js</code></td>
<td>“Sign” (negative)</td>
<td><code>b-a &lt; 0</code></td>
</tr>
<tr>
<td><code>jns</code></td>
<td>(non-negative)</td>
<td><code>b-a &gt;= 0</code></td>
</tr>
<tr>
<td><code>jg</code></td>
<td>“Greater”</td>
<td><code>b &gt; a</code></td>
</tr>
<tr>
<td><code>jge</code></td>
<td>“Greater or equal”</td>
<td><code>b &gt;= a</code></td>
</tr>
<tr>
<td><code>jl</code></td>
<td>“Less”</td>
<td><code>b &lt; a</code></td>
</tr>
<tr>
<td><code>jle</code></td>
<td>“Less or equal”</td>
<td><code>b &lt;= a</code></td>
</tr>
<tr>
<td><code>ja</code></td>
<td>“Above” (unsigned &gt;)</td>
<td><code>b &gt; a</code></td>
</tr>
<tr>
<td><code>jb</code></td>
<td>“Below” (unsigned &lt;)</td>
<td><code>b &lt; a</code></td>
</tr>
</tbody>
</table>

- `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_{\text{LSB}} == 0`
  - `jne: a_{\text{LSB}} == 1`
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Comparison</th>
<th>Register Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>b == a</td>
<td>%rdi: argument x</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>b != a</td>
<td>%rsi: argument y</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>b-a &lt; 0</td>
<td>%rax: return value</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>b-a &gt;=0</td>
<td></td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>b &gt; a</td>
<td></td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>b &gt;= a</td>
<td></td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>b &lt; a</td>
<td></td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>b &lt;= a</td>
<td></td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>b &gt; a</td>
<td></td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>b &lt; a</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Register Use(s):
- %rdi: argument x
- %rsi: argument y
- %rax: return value
**Question**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

A. `cmpq %rsi, %rdi` \(x-y\)
   `jle .L4`

B. `cmpq %rsi, %rdi` \(x-y\)
   `jg .L4`

C. `testq %rsi, %rdi` \(x\&y\)
   `jle .L4`

D. `testq %rsi, %rdi` \(x\&y\)
   `jg .L4`

E. We’re lost...

---

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

```assembly
absdiff:

____________________________________
# x > y:
    movq %rdi, %rax
    subq %rsi, %rax
    ret

.L4:
# x <= y:
    movq %rsi, %rax
    subq %rdi, %rax
    x-y <= 0
    ret
```

Register Use(s)

%rdi 1st argument (x)
%rsi 2nd argument (y)
%rax return value
A jump changes the program counter (%rip)
  - %rip tells the CPU the address of the next instr to execute

Labels give us a way to refer to a specific instruction in our assembly/machine code
  - Associated with the next instruction found in the assembly code (ignores whitespace)
  - Each use of the label will eventually be replaced with something that indicates the final address of the instruction that it is associated with
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- **Loops**
- Switches
Expressing with Goto Code

C allows `goto` as means of transferring control (jump)
- Closer to assembly programming style
- Generally considered bad coding style

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

```c
long absdiff_j(long x, long y) {
    long result;
    int ntest = (x <= y);
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```
Compiling Loops

- Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time

- Most important to consider:
  - When should conditionals be evaluated? (while vs. do-while)
  - How much jumping is involved?

C/Java code:

```java
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:

```assembly
loopTop: testq %rax, %rax
         je loopDone
         <loop body code>
      jmp loopTop
loopDone:
```

Test
Compiling Loops

C/Java code:

```c
while ( Test ) {
    Body
}
```

Goto version:

```c
Loop: if (!Test) goto Exit;
Body
goto Loop;
Exit:
```

- **What are the Goto versions of the following?**
  - Do...while: Test and Body
  - For-loop: Init, Test, Update, and Body
Compiling Loops

While Loop:

```c
C: while ( sum != 0 ) {
    <loop body>
}
```

```x86-64
loopTop: testq %rax, %rax
         je loopDone
         <loop body code>
         jmp loopTop
loopDone:
```

Do-while Loop:

```c
C: do {
    <loop body>
} while ( sum != 0 )
```

```x86-64
loopTop: testq %rax, %rax
         jne loopTop
         <loop body code>
         jmp loopTop
loopDone:
```

While Loop (ver. 2):

```c
C: while ( sum != 0 ) {
    <loop body>
}
```

```x86-64
loopTop: testq %rax, %rax
         je loopDone
         <loop body code>
         do-while loop
         testq %rax, %rax
         jne loopTop
loopDone:
```
For-Loop → While-Loop

For-Loop:

```c
for (Init; Test; Update) {
    Body
}
```

While-Loop Version:

```c
Init;
while (Test) {
    Body
    Update;
}
```

Caveat: C and Java have `break` and `continue`

- Conversion works fine for `break`
  - Jump to same label as loop exit condition
- But not continue: would skip doing `Update`, which it should do with for-loops
  - Introduce new label at `Update`
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- **Switches**
Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4
- Implemented with:
  - Jump table
  - Indirect jump instruction
Jump Table Structure

**Switch Form**

```c
switch (x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    ... ...
    case val_{n-1}:
        Block n-1
}
```

**Jump Table**

- **JTab**: address of jump table
- **Targ0**: Code Block 0
- **Targ1**: Code Block 1
- **Targ2**: Code Block 2
- **Targ_n-1**: Code Block n-1

**Jump Targets**

**Approximate Translation**

```c
target = JTab[x];
goto target;
```

*like an array of pointers*
Jump Table Structure

C code:

```c
switch (x) {
    case 1: <some code>
        break;
    case 2: <some code>
    case 3: <some code>
        break;
    case 5:
    case 6: <some code>
        break;
    default: <some code>
}
```

Use the jump table when $x \leq 6$:

```c
if (x <= 6)
    target = JTab[x];
else
    goto default;
```
Switch Example

```
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        ...
    }
    return w;
}
```

**Register Use(s)**
- `%rdi` 1st argument (x)
- `%rsi` 2nd argument (y)
- `%rdx` 3rd argument (z)
- `%rax` return value

Note compiler chose to not initialize `w`

**Jump Example**
```
switch_eg:
    movq %rdx, %rcx
    cmpq $6, %rdi  # x:6
    ja .L8        # default
    jmp *.L4(,%rdi,8)  # jump table
```

Jump above – unsigned > catches negative default cases
-1 > 6U → jump to default case

Take a look!
https://godbolt.org/g/DnOmXb
Switch Example

long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        ...
    }
    return w;
}

Register Use(s)
%rdi 1st argument (x)
%rsi 2nd argument (y)
%rdx 3rd argument (z)
%rax return value

Jump table
.long switch_ex(
    long x,
    long y,
    long z)
{
    long w = 1;
    switch (x) {
        ...
    }
    return w;
}

Indirect jump
D + Ri + S
addr of jump table
size of (void*)

Jump table
.section .rodata
.align 8
.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6
Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes (address)
  - Base address at `.L4`

- **Direct jump:** `jmp .L8`
  - Jump target is denoted by label `.L8`

- **Indirect jump:** `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective address `.L4 + x*8`
    - Only for $0 \leq x \leq 6$

```
Jump table
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

Memo: $[	ext{D} + 	ext{Reg}][x*5]$
Jump Table

Declaring data, not instructions.

Jump table

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

8-byte memory alignment

```
switch(x) {
    case 1:     // .L3
        w = y*z;
        break;
    case 2:     // .L5
        w = y/z;
        /* Fall Through */
    case 3:     // .L9
        w += z;
        break;
    case 5:
    case 6:     // .L7
        w -= z;
        break;
    default:    // .L8
        w = 2;
}
```

This data is 64-bits wide.
Code Blocks \( (x == 1) \)

```c
switch(x) {
    case 1:    // .L3
        w = y*z;
        break;
    ...
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument ( (x) )</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument ( (y) )</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument ( (z) )</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

```
.L3:
    movq %rsi, %rax  # y
    imulq %rdx, %rax  # y*z
    ret
```
Handling Fall-Through

```c
long w = 1;
    . . .
switch (x) {
    . . .
    case 2: // .L5
        w = y/z;
        /* Fall Through */
    case 3: // .L9
        w += z;
        break;
    . . .
}
```

More complicated choice than “just fall-through” forced by “migration” of `w = 1;`

- Example compilation trade-off

```c
case 2:
    w = y/z;
    goto merge;
```

```c
case 3:
    w = 1;
merge:
    w += z;
```
Code Blocks \((x == 2, 3)\)

```c
long w = 1;
  ...
switch (x) {
  ...
  case 2: // .L5
    w = y/z;
    /* Fall Through */
  case 3: // .L9
    w += z;
    break;
  ...
}
```

Register Use(s)
- `%rdi` 1st argument \((x)\)
- `%rsi` 2nd argument \((y)\)
- `%rdx` 3rd argument \((z)\)
- `%rax` return value

```
  .L5:  # Case 2:
    movq %rsi, %rax # y in rax
    cqto # Div prep
    idivq %rcx # y/z
    jmp .L6 # goto merge
  .L9:  # Case 3:
    movl $1, %eax # w = 1
  .L6:  # merge:
    addq %rcx, %rax # w += z
    ret
```
Code Blocks (rest)

switch (x) {
    ...  
    case 5: // .L7
         case 6: // .L7
            w -= z;
        break;
    default: // .L8
        w = 2;
}

.L7:               # Case 5,6:
    movl $1, %eax   # w = 1
    subq %rdx, %rax # w -= z
    ret

.L8:               # Default:
    movl $2, %eax   # 2
    ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1\text{st} argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\text{nd} argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3\text{rd} argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>
Roadmap

C:

```c
#include <stdlib.h>
r
int main() {
    car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:

```assembly
get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret
```

Machine code:

```
0111010000011000
1000110100000100000000101000100111000010110000011111101000011111
```

Computer system:
Mechanisms required for procedures

1) Passing control
   - To beginning of procedure code
   - Back to return point

2) Passing data
   - Procedure arguments
   - Return value

3) Memory management
   - Allocate during procedure execution
   - Deallocate upon return

   All implemented with machine instructions!
   - An x86-64 procedure uses only those mechanisms required for that procedure

```c
int Q(int i) {
    int t = 3*i;
    int v[10];
    //...
    return v[t];
}
```

```c
P(...) {
    ...
    y = Q(x);
    print(y);
    ...
}
```
Procedures

- **Stack Structure**
- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data
- **Register Saving Conventions**
- **Illustration of Recursion**
Simplified Memory Layout

- **Instructions**
- **Literals**
- **Static Data**
  - variables allocated with `new` or `malloc`
  - `static` variables (including global variables (C))
- **Dynamic Data (Heap)**
  - large constants (e.g., “example”)
- **Stack**
  - local variables; procedure context

Memory Addresses:
- Low Addresses: `0x0...0`
- High Addresses: `2^N - 1`
Memory Permissions

- **Stack**
  - Writable; not executable
  - Managed "automatically" (by compiler)

- **Dynamic Data (Heap)**
  - Writable; not executable
  - Managed by programmer
  - Grow towards each other to maximize use of space

- **Static Data**
  - Writable; not executable
  - Initialized when process starts

- **Literals**
  - Read-only; not executable
  - Initialized when process starts

- **Instructions**
  - Read-only; executable
  - Initialized when process starts

Segmentation faults?

Accessing memory in a way that you are not allowed to
x86-64 Stack

- Region of memory managed with stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”

- Register %rsp contains lowest stack address
  - %rsp = address of top element, the most-recently-pushed item that is not-yet-popped

Last In, First Out (LIFO)
**x86-64 Stack: Push**

- **pushq** `src`  
  - Fetch operand at `src`  
    - `src` can be reg, memory, immediate  
  - **Decrement** `%rsp` by 8  
  - Store value at address given by `%rsp`

**Example:**

- **pushq** `%rcx`  
  - Adjust `%rsp` and store contents of `%rcx` on the stack

Stack Pointer: `%rsp`  

1. move `%rsp` down (subtract)  
2. store `src` at `%rsp`
x86-64 Stack: Pop

- **popq** \( dst \)
  - Load value at address given by \( %rsp \)
  - Store value at \( dst \)
  - **Increment** \( %rsp \) by 8

**Example:**
- **popq** \( %rcx \)
  - Stores contents of top of stack into \( %rcx \) and adjust \( %rsp \)

**Stack Pointer:** \( %rsp \)

Those bits are still there; we’re just not using them.
Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Procedure Call Overview

- **Callee** must know where to find **args**
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on the same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (*e.g.* no arguments)
The convention of where to leave/find things is called the calling convention (or procedure call linkage)

- Details vary between systems
- We will see the convention for x86-64/Linux in detail
- What could happen if our program didn’t follow these conventions?
# Code Example (Preview)

```c
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```c
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

Compiler Explorer: https://godbolt.org/g/cKKDZn

```
<mult2>:
400550: movq %rdi,%rax  # a
400553: imulq %rsi,%rax # a * b
400557: ret             # Return
```

```
<multstore>:
400540: push %rbx       # Save %rbx
400541: movq %rdx,%rbx  # Save dest
400544: call 400550 <mult2> # mult2(x,y)
400549: movq %rax,(%rbx) # Save at dest
40054c: pop %rbx        # Restore %rbx
40054d: ret             # Return
```

**Note:** The highlighted code is the expected output from the disassembly, illustrating the process of calling `mult2` and storing the result in the destination.
Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **call label**  (special push)
  1) Push return address on stack (*why? which address?*)
  2) Jump to **label**
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call**: `call label` (special push)
  1. Push return address on stack (why? which address?)
  2. Jump to `label`
- **Return address**:
  - Address of instruction immediately after `call` instruction
  - Example from disassembly:
    ```
    400544: call 400550 <mult2>
    400549: movq %rax,(%rbx)
    ```
    Return address = 0x400549
- **Procedure return**: `ret` (special pop)
  1. Pop return address from stack
  2. Jump to address