x86-64 Programming III
CSE 351 Summer 2018

Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee
Natalie Andreeva
Teagan Horkan

Why I try not to be pedantic about conditionals.

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 `stepti/nextti`)
  - 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*

```
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
```

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

- Reminder: \texttt{cmp} is like \texttt{sub}, \texttt{test} is like \texttt{and}
  - Result is not stored anywhere

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>CMP Condition</th>
<th>TEST Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>( b == a )</td>
<td>( b &amp; a == 0 )</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>( b \neq a )</td>
<td>( b &amp; a \neq 0 )</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>( b - a &lt; 0 )</td>
<td>( b &amp; a &lt; 0 )</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>( b - a \geq 0 )</td>
<td>( b &amp; a \geq 0 )</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>( b &gt; a )</td>
<td>( b &amp; a &gt; 0 )</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>( b \geq a )</td>
<td>( b &amp; a \geq 0 )</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>( b &lt; a )</td>
<td>( b &amp; a &lt; 0 )</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>( b \leq a )</td>
<td>( b &amp; a \leq 0 )</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>( b &gt; a )</td>
<td>( b &amp; a &gt; 0U )</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>( b &lt; a )</td>
<td>( b &amp; a &lt; 0U )</td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

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

### Table

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

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

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

### A.

\[
\text{cmpq } \%\text{rsi}, \%\text{rdi} \\
\text{\textcolor{red}{jle}} \quad .L4
\]

### B.

\[
\text{cmpq } \%\text{rsi}, \%\text{rdi} \\
\text{\textcolor{red}{jg}} \quad .L4
\]

### C.

\[
\text{testq } \%\text{rsi}, \%\text{rdi} \\
\text{\textcolor{red}{jle}} \quad .L4
\]

### D.

\[
\text{testq } \%\text{rsi}, \%\text{rdi} \\
\text{\textcolor{red}{jg}} \quad .L4
\]

### E.

We’re lost...

### Register Use(s)

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

### Code Example

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

### Code Explanation

```asm
absdiff:

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

### Explanations

A. The comparison `cmpq %rsi, %rdi` checks if `x` is less than or equal to `y`. `jle` branches if the comparison is true, which is correct for finding the absolute difference.

B. The comparison `cmpq %rsi, %rdi` checks if `x` is greater than `y`. `jg` branches if the comparison is true, which is incorrect for finding the absolute difference.

C. The comparison `testq %rsi, %rdi` checks if `x` is less than or equal to `y`. `jle` branches if the comparison is true, which is correct for finding the absolute difference.

D. The comparison `testq %rsi, %rdi` checks if `x` is greater than `y`. `jg` branches if the comparison is true, which is incorrect for finding the absolute difference.

E. The statement `We’re lost...` is not correct for the problem.
Labels

- 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

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

```
max:
    movq %rdi, %rax
    cmpq %rsi, %rdi
    jg done
    movq %rsi, %rax

done:
    ret
```
x86 Control Flow

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

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

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;
}
```

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

C/Java code:

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

Assembly code:

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

loopDone:
```

- 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?
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
```

**Do-while Loop:**

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

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

**While Loop (ver. 2):**

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

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

```
loopDone:  testq %rax, %rax
          jne  loopTop
```
For-Loop → While-Loop

For-Loop:

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

While-Loop Version:

```
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

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

### Jump Table

<table>
<thead>
<tr>
<th>JTab:</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>Targn-1</th>
</tr>
</thead>
</table>

### Jump Targets

- **Targ0**: Code Block 0
- **Targ1**: Code Block 1
- **Targ2**: Code Block 2
- **Targn-1**: Code Block n-1

### Approximate Translation

```
target = JTab[x];
goto target;
```
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];
    goto target;
else
    goto default;
```
Switch Example

```c
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`

### Assembly Example

`switch_eg:`

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

**Take a look!**

https://godbolt.org/g/DnOmXb

**Jump above** – unsigned > catches negative default cases
Switch Example

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

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
```

Register Use(s)

<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>

Switch_eg:

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

```assembly
.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
```
Jump Table

declaring data, not instructions

8-byte memory alignment

Jump table

```assembly
.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

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)

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

.L3:
    movq  %rsi, %rax  # y
    imulq %rdx, %rax  # y*z
    ret

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

<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>

```
.L5:
    # Case 2:
    movq    %rsi, %rax # y in rax
cqto
    idivq    %rcx # y/z
jmp .L6 # goto merge
.L6:
    addq    %rcx, %rax # w += z
ret
```

Code Blocks (rest)

```c
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\textsuperscript{st} argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3\textsuperscript{rd} argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>
Roadmap

C:

```c
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:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111010000011111
```

Computer system:

OS:

- Windows 10
- macOS

Memory & data
Integers & floats
x86 assembly

Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
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

Memory Addresses

High Addresses

2^{N-1}

Memory

Stack

local variables; procedure context

Dynamic Data

(Heap)

variables allocated with new or malloc

Static Data

static variables (including global variables (C))

Literals

large constants (e.g. “example”)

Instructions

program code

Low Addresses

0

program code
Memory Permissions

- **Stack**: Managed “automatically” (by compiler).
  - Writable; not executable

- **Dynamic Data (Heap)**: Managed by programmer.
  - Writable; not executable

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

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

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

**segmentation faults?**
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
**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`**

- Stack “Bottom”
- Stack “Top”
- Stack Grows Down
- High Addresses
- Increasing Addresses
- Low Addresses
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`

---

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 same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (e.g. no arguments)
Procedure Call Overview

- 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;
}
```

Compiler Explorer: [https://godbolt.org/g/cKKDZn](https://godbolt.org/g/cKKDZn)

```assembly
00000000000400540 <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
```

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

```assembly
00000000000400550 <mult2>:
  400550: movq %rdi,%rax  # a
  400553: imulq %rsi,%rax # a * b
  400557: ret            # Return
```
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call**: `call label`
  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: \texttt{call label}

1) Push return address on stack \textit{(why? which address?)}
2) Jump to \textit{label}

Return address:
- Address of instruction immediately after \texttt{call} instruction
- Example from disassembly:

\begin{verbatim}
400544: call 400550 <mult2>
400549: movq %rax,(%rbx)
\end{verbatim}

Return address = \texttt{0x400549}

Procedure return: \texttt{ret}

1) Pop return address from stack
2) Jump to address