x86-64 Programming III
CSE 351 Spring 2018

I'll be in your city tomorrow if you want to hang out.
But where will you be if I don't want to hang out?!
You know, I just remembered I'm busy.

Why I try not to be pedantic about conditionals.

http://xkcd.com/1652/
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;
}
```

```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;
}
```

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

C/Java code:

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

Assembly code:

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

- 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>
            jne    loopTop
loopDone:
```

**Do-while Loop:**

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

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

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

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

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

For Version

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

While 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
Labels: What the heck are they?

- A jump changes the program counter (%rip), which is just the *address of an instruction to execute*.
- So we need a way to say *the address of that instruction over there*
- A *label* is an assembly-language “thing” for *the address of this next instruction*
  - Because we don’t know where the *assembler* will put that instruction in the big array of bytes
  - It will (a) choose a place and (b) go change the *uses* of the label “appropriately” for us

So: *labels are address of instructions*, nbd 😊
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

```c
long switch_ex
(long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```
### Jump Table Structure

#### Switch Form

```java
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

<table>
<thead>
<tr>
<th>Targ0:</th>
<th>Code Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ1:</td>
<td>Code Block 1</td>
</tr>
<tr>
<td>Targ2:</td>
<td>Code Block 2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Targn-1:</td>
<td>Code Block n-1</td>
</tr>
</tbody>
</table>

#### Approximate Translation

```java
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 Statement 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 Code
```
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

Take a look!
[https://godbolt.org/g/DnOmXb](https://godbolt.org/g/DnOmXb)
Switch Statement Example

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

Jump table

```asm
.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 statement example:

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

Indirect jump
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 $0 \leq x \leq 6$

```asm
.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

```
.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
```

this data is 64-bits wide

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;
}

declaring data, not instructions

8-byte memory alignment

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;
    . . .
}
```

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

```c
case 3:
    w = 1;
```

merge:
    w += z;

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

• Example compilation trade-off
Code Blocks (x == 2, x == 3)

long w = 1;

switch (x) {

    case 2: // .L5
        w = y/z;
        /* Fall Through */
    case 3: // .L9
        w += z;
        break;

}

.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
    addq %rcx, %rax # w += z
    ret

Register | Use(s)
---|---
%rdi | 1\textsuperscript{st} argument (x)
%rsi | 2\textsuperscript{nd} argument (y)
%rdx | 3\textsuperscript{rd} argument (z)
%rax | Return value
Code Blocks (rest)

```c
switch (x) {
    . . .
    case 5:  // .L7
        w -= z;
        break;
    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>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>