x86 Programming III
CSE 351 Autumn 2016

http://xkcd.com/648/
Administrivia
Peer Instruction Question

Which conditional statement properly fills in the following blank?

```c
if(__________) {...} else {...}
```

- `cmpq $1, %rsi` # %rsi = j
- `setg %dl` # %dl =
- `cmpq %rdi, %rsi` # %rdi = i
- `setl %al` # %al =
- `orb %al, %dl` # arithmetic operation
- `je .else` # sets flags!

(A) $j > 1 \mathbin{||} j < i$
(B) $j > 1 \mathbin{&&} j < i$
(C) $j \leq 1 \mathbin{||} j \geq i$
(D) $j \leq 1 \mathbin{&&} j \geq i$
Jumping

- **j* Instructions**
  - Jumps to `target` (argument – actually just an address)
  - Conditional jump relies on special *condition code registers*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>je <code>target</code></td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne <code>target</code></td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js <code>target</code></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns <code>target</code></td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg <code>target</code></td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge <code>target</code></td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl <code>target</code></td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle <code>target</code></td>
<td>(SF^OF) | ZF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja <code>target</code></td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb <code>target</code></td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
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
Compiling Loops

C/Java code:

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

Assembly code:

```
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:

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

Goto version

```
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/Java code:

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

Assembly code:

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

**Do-while loop**

C/Java code:

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

Assembly code:

```assembly
loopTop:    <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```
Do-While Loop Example

C Code

```c
long pcount_do(unsigned long x)
{
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

Goto Version

```c
long pcount_goto(unsigned long x)
{
    long result = 0;
    loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

- Count number of 1’s in argument \( x \) (“popcount”)
- Use backward branch to continue looping
- Only take branch when “while” condition holds
**Do-While Loop Compilation**

### Goto Version

```c
long pcount_goto(unsigned long x) {
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (x) goto loop;
    return result;
}
```

### Assembly

```
movl  $0, %eax       # result = 0
.L2:                   # loop:
    movq  %rdi, %rdx
    andl  $1, %edx      # t = x & 0x1
    addq  %rdx, %rax    # result += t
    shrq  %rdi          # x >>= 1
    jne   .L2           # if (x) goto loop
    rep ret             # return (rep weird)
```

### 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>%rax</td>
<td>ret val (result)</td>
</tr>
</tbody>
</table>
General **Do-While Loop** Translation

<table>
<thead>
<tr>
<th>C Code</th>
<th>Goto Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>do</strong></td>
<td><strong>loop:</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Body</strong></td>
</tr>
<tr>
<td><strong>while</strong> (Test);</td>
<td>if (Test)</td>
</tr>
<tr>
<td></td>
<td>goto loop</td>
</tr>
</tbody>
</table>

- **Body:**
  
  ```
  \{ 
  \quad \text{Statement}_1; 
  \quad \ldots 
  \quad \text{Statement}_n; 
  \}
  ```

- **Test** returns integer
  - = 0 interpreted as false, ≠ 0 interpreted as true
General **While Loop** - Translation #1

- “Jump-to-middle” translation
- Used with `-Og`

**While version**

```c
while (Test)  
  Body
```

**Goto Version**

```c
goto test;  
loop:  
  Body  
test:  
  if (Test)  
    goto loop;  
  done:
```
While Loop Example – Translation #1

C Code

```c
long pcount_while
(unsigned long x)
{
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Jump to Middle

```c
long pcount_goto_jtm
(unsigned long x)
{
    long result = 0;
    goto test;
    loop:
    result += x & 0x1;
    x >>= 1;
    test:
    if (x) goto loop;
    return result;
}
```

- Used with `-Og`
- Compare to do-while version of function
- Initial `goto` starts loop at `test`
**General While Loop - Translation #2**

**While version**

```c
while (Test)  
  Body  
```

**Do-While Version**

```c
if (!Test)  
  goto done;  
 do  
  Body  
  while (Test);  
done:  
```

**Goto Version**

```c
if (!Test)  
  goto done;  
loop:  
  Body  
  if (Test)  
    goto loop;  
done:  
```

- “Do-while” conversion
- Used with `-O1`
While Loop Example – Translation #2

C Code

```c
long pcount_while (unsigned long x)
{
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Do-While Version

```c
long pcount_goto_dw (unsigned long x)
{
    long result = 0;
    if (!x) goto done;
    loop:
    result += x & 0x1;
    x >>= 1;
    if (x) goto loop;
    done:
    return result;
}
```

- Used with \(-O1\)
- Compare to do-while version of function (one less jump?)
- Initial conditional guards entrance to loop
For Loop Form

General Form

\[
\text{for} \ (\text{Init}; \ \text{Test}; \ \text{Update}) \\
\text{Body}
\]

```c
#define WSIZE 8*sizeof(int)
long pcount_for(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init
- \(i = 0\)

Test
- \(i < \text{WSIZE}\)

Update
- \(i++\)

Body
- \(\text{for} \ (i = 0; i < \text{WSIZE}; i++) \{\)
  - \(\text{unsigned bit} =\)
    - \((x >> i) & 0x1;\)
  - \(\text{result} += \text{bit};\)
- \} \)
  - \(\text{return} \text{result};\)
For Loop → While Loop

For Version

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

While Version

```plaintext
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
For Loop - While Conversion

```c
long pcount_for_while(unsigned long x)
{
    size_t i;
    long result = 0;
    i = 0;
    while (i < WSIZE) {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
        i++;
    }
    return result;
}
```
For Loop - Do-While Conversion

C Code

```c
long pcount_for (unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Goto Version

```c
long pcount_for_goto_dw (unsigned long x)
{
    size_t i;
    long result = 0;
    i = 0;
    if (!(i < WSIZE))
        goto done;
    loop:
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
        i++;
        if (i < WSIZE)
            goto loop;
    done:
        return result;
}
```

- Initial test can be optimized away!
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
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**

```
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 Statement Example

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

### Register Use(s)

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

Note compiler chose to not initialize `w`

Take a look!
[https://godbolt.org/g/NAxYV](https://godbolt.org/g/NAxYV)

**jump above** – unsigned > catches negative default cases
Switch Statement Example

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

Jump table

.switch .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_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:** \( \text{jmp} \ .L8 \)
  - Jump target is denoted by label \( .L8 \)

- **Indirect jump:** \( \text{jmp} \ * .L4 (, 8) \)
  - 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
```
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;
}

declaring data, not instructions

8-byte memory alignment

this data is 64-bits wide
Code Blocks (x == 1)

```
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
Code Blocks (x == 2, x == 3)

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

### 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>

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

The code block for x == 2 assigns `w = y/z`, but since it is a `Fall Through` case, control goes directly to the next case without returning the result. The code block for x == 3 adds `z` to `w` directly without division.
Code Blocks (rest)

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

### 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>

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

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

Would you implement this with a jump table?

```c
switch (x) {
    case 0:     <some code>
                break;
    case 10:    <some code>
                 break;
    case 32767: <some code>
                 break;
    default:    <some code>
                 break;
}
```

Probably not

- 32,768-entry jump table too big (256 KiB) for only 4 cases
- For comparison, text of this switch statement 193 B
Bonus content (nonessential). Does contain examples.

- Conditional Operator with Jumps
- Conditional Move
Conditional Operator with Jumps

C Code

```c
val = Test ? Then-Expr : Else-Expr;
```

Example:

```c
result = x>y ? x-y : y-x;
```

Goto Version

```c
ntest = !Test;
if (ntest) goto Else;
val = Then_Expr;
goto Done;
Else:
val = Else_Expr;
Done:
... 
```

- Ternary operator `?:`
- `Test` is expression returning integer
  - `0` interpreted as false
  - `≠ 0` interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

**Bonus Content (nonessential)**
Conditional Move

- Conditional Move Instructions: `cmovC src, dst`
  - Move value from `src` to `dst` if condition `C` holds
  - if(`Test`) `Dest ← Src`
  - GCC tries to use them (but only when known to be safe)

- Why is this useful?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

```c
long absdiff(long x, long y)
{
    return x>y ? x-y : y-x;
}
```

Bonus Content (nonessential) more details at end of slides
Using Conditional Moves

- Conditional Move Instructions
  - `cmovC src, dest`
  - Move value from src to dest if condition \( C \) holds
  - Instruction supports:
    - if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe

- Why is this useful?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```c
val = Test
? Then_Expr
: Else_Expr;
```

“Goto” Version

```c
result = Then_Expr;
else_val = Else_Expr;
nt = !Test;
if (nt) result = else_val;
return result;
```
Conditional Move Example

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

absdiff:

```
movq %rdi, %rax  # x
subq %rsi, %rax  # result = x - y
movq %rsi, %rdx
subq %rdi, %rdx  # else_val = y - x
cmpq %rsi, %rdi  # x:y
cmovle %rdx, %rax # if \(\leq\), result = else_val
ret
```
Bad Cases for Conditional Move

Expensive Computations

\[ \text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) \ : \ \text{Hard2}(x); \]

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

\[ \text{val} = p \ ? \ *p \ : \ 0; \]

- Both values get computed
- May have undesirable effects

Computations with side effects

\[ \text{val} = x > 0 \ ? \ x*=7 \ : \ x+=3; \]

- Both values get computed
- Must be side-effect free