Machine-Level Programming II: Control

CS351: Systems Programming
Day 7: Sep. 13, 2022

Instructor:
Nik Sultana

Slides adapted from Bryant and O’Hallaron
Remember to check the course page

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aug 22</td>
<td>Aug 23&lt;br&gt;LEC 1: Introduction&lt;br&gt;Preparation: Read CS:APP Chapter 1</td>
<td>Aug 24</td>
<td>Aug 25&lt;br&gt;LEC 2: C and x86_64 toolchains&lt;br&gt;Preparation: Read K&amp;R Chapter 1, and work through Roy Toft’s NASM tutorial.</td>
</tr>
<tr>
<td>Aug 29 LAB</td>
<td>Aug 30&lt;br&gt;LEC 3: Bits, Bytes, and Ints: Part 1&lt;br&gt;Preparation: Read CS:APP 2.1&lt;br&gt;Assigned: Lab 1: Preliminaries</td>
<td>Aug 31</td>
<td>Sep 01&lt;br&gt;LEC 4: Bits, Bytes, and Ints: Part 2&lt;br&gt;Preparation: Read CS:APP 2.2-2.3</td>
</tr>
<tr>
<td>Sep 05 Labor Day</td>
<td>Sep 06&lt;br&gt;LEC 5: Floating Point&lt;br&gt;Preparation: Read CS:APP 2.4</td>
<td>Sep 07</td>
<td>Sep 08&lt;br&gt;LEC 6: Machine Prog: Basics&lt;br&gt;Preparation: Read CS:APP 3.1-3.5</td>
</tr>
<tr>
<td>Sep 12 LAB</td>
<td>Sep 13&lt;br&gt;LEC 7: Machine Prog: Control&lt;br&gt;Preparation: Read CS:APP 3.6&lt;br&gt;Assigned: Lab 2: Datalab and Data Representations</td>
<td>Sep 14</td>
<td>Sep 15&lt;br&gt;LEC 8: Machine Prog: Procedures&lt;br&gt;Preparation: Read CS:APP 3.7</td>
</tr>
</tbody>
</table>

- See course webpage for:
  - New lab assignment.
  - Confirm office hours in case there are changes.
  - Make sure to carry out the Preparation.
Questions follow-up

■ What do the different parameters to gcc do?
  ▪ Use “man gcc”
  ▪ Or “gcc --help” (Note: two dashes)

■ Showcasing assignments on GitHub for job search?
  ▪ Not much benefit, for the following reasons:
    ▪ Many do this, so it’s hard to stand out with these assignments.
    ▪ Learning-quality code != your best code.
  ▪ Quality > Quantity: One interesting self-motivated or open-source project contribution > 10 lab assignments.
  ▪ Where to start? Some ideas:
    ▪ [https://hacktoberfest.com/](https://hacktoberfest.com/)
    ▪ [https://summerofcode.withgoogle.com/](https://summerofcode.withgoogle.com/) (Note CoI)
    ▪ Look around the Web for more, let me know what you find!
Questions follow-up

- **What if we need more lab sessions to complete a lab?**
  - Work on the lab assignment outside of the “lab”!
  - Reach out to your TA – many students do this on Discord.

- **What if we’ve finished the lab early?**
  - (Labs are timed to avoid overlapping assignments; some students need the last “Monday” to finish the lab if they’re very stuck.)
  - Do the exercises in the CS:APP3e book and ask your TA if stuck.
  - Practice C – do the exercises in the K&R book!
  - See resources on the course webpage, in particular: [https://missing.csail.mit.edu/](https://missing.csail.mit.edu/)
  - Work on “above and beyond” problems.
Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements
Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (\%rax, \ldots)
  - Location of runtime stack (\%rsp)
  - Location of current code control point (\%rip, \ldots)
  - Status of recent tests (CF, ZF, SF, OF)

Registers

\%
rax
\%
rbx
\%
rcx
\%dx
\%si
\%di
\%sp
\%bp
\%
r8
\%
r9
\%10
\%11
\%12
\%13
\%14
\%15

Instruction pointer
\%
rip

Current stack top

Condition codes

CF ZF SF OF
Condition Codes (Implicitly set)

- Single bit registers
  - CF  Carry Flag (for unsigned)  SF  Sign Flag (for signed)
  - ZF  Zero Flag
  - OF  Overflow Flag (for signed)

- Implicitly set (think of it as side effect) by arithmetic operations

  **Example:** \texttt{addq Src,Dest} \rightarrow t = a+b

  - **CF set** if carry out from most significant bit (unsigned overflow)
  - **ZF set** if \( t = 0 \)
  - **SF set** if \( t < 0 \) (as signed)
  - **OF set** if two’s-complement (signed) overflow
    \( (a>0 \ \&\& \ b>0 \ \&\& \ t<0) \ \text{||} \ (a<0 \ \&\& \ b<0 \ \&\& \ t>=0) \)

- Not set by \texttt{leaq} instruction
Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - `cmpq` Src2, Src1
  - `cmpq` b, a like computing `a - b` without setting destination

- CF set if carry out from most significant bit (used for unsigned comparisons)
- ZF set if `a == b`
- SF set if `(a - b) < 0` (as signed)
- OF set if two’s-complement (signed) overflow
  
  \[(a > 0 && b < 0 && (a - b) < 0) || (a < 0 && b > 0 && (a - b) > 0)\]
Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - `testq Src2, Src1`
    - `testq b, a` like computing `a & b` without setting destination

- Sets condition codes based on value of `Src1 & Src2`
- Useful to have one of the operands be a mask

- ZF set when `a & b == 0`
- SF set when `a & b < 0`
Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>Registers</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%al</td>
</tr>
<tr>
<td>%rbx</td>
<td>%bl</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
</tr>
<tr>
<td>%rsi</td>
<td>%sil</td>
</tr>
<tr>
<td>%rdi</td>
<td>%sil</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
</tr>
<tr>
<td>%rbp</td>
<td>%pl</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8b</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15b</td>
</tr>
</tbody>
</table>

- Can reference low-order byte
Reading Condition Codes (Cont.)

- **SetX Instructions:**
  - Set single byte based on combination of condition codes

- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use `movzbl` to finish job
    - 32-bit instructions also set upper 32 bits to 0

```c
int gt (long x, long y)
{
    return x > y;
}
```

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

```assembly
    cmpq   %rsi, %rdi  # Compare x:y
    setg   %al        # Set when >
    movzbl %al, %eax  # Zero rest of %rax
    ret
```
Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements
Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Conditional Branch Example

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

```
unix> gcc -Og -S -fno-if-conversion control.c
```

```
absdiff:
    cmpq  %rsi, %rdi  # x:y  
    jle   .L4
    movq  %rdi, %rax
    subq  %rsi, %rax  
    ret

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

<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>
Expressing with Goto Code

- C allows goto statement
- Jump to position designated by label

```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;
}
```
General Conditional Expression Translation (Using Branches)

C Code

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

```c
val = 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:
    ...
```

- Create separate code regions for then & else expressions
- Execute appropriate one
Using Conditional Moves

- Conditional Move Instructions
  - 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?
  - 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;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
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` # eval = y - x
- `cmpq %rsi, %rdi` # x:y
- `cmovle %rdx, %rax` # if <=, result = eval
- `ret`
Bad Cases for Conditional Move

Expensive Computations

```java
val = Test(x) ? Hard1(x) : Hard2(x);
```

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

Risky Computations

```java
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```java
val = x > 0 ? x*=7 : x+=3;
```

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

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements
“Do-While” Loop Example

C Code

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

Goto Version

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 conditional branch to either continue looping or to exit loop
“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;
}
```

```
long pcount_goto (unsigned long x) {
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if(x) goto loop;
    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>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

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

From Day 6
General “Do-While” Translation

C Code

```c
do
    Body
while (Test);
```

- **Body:**
  ```c
  { 
    Statement\textsubscript{1};
    Statement\textsubscript{2};
    ...
    Statement\textsubscript{n};
  }
  ```

Goto Version

```c
loop:
    Body
    if (Test)
        goto loop
```
General “While” Translation #1

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

While version

```
while (Test)
  Body
```

Goto Version

```
goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```
While Loop Example #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;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test
General “While” Translation #2

While version

```
while (Test)
  Body
```

Do-While Version

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

Goto Version

```
if (!Test)
  goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```
## While Loop Example #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;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop
“For” Loop Form

General Form

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

\[
\text{Body}
\]

```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

```c
{
  unsigned bit =
    (x >> i) & 0x1;
  result += bit;
}
```
"For" Loop $\rightarrow$ While Loop

For Version

```
for (Init; Test; Update )

Body
```

While Version

```
Init;
while (Test) {

Body

Update ;
}
```
For-While Conversion

Init
i = 0

Test
i < WSIZE

Update
i++

Body
{
    unsigned bit =
        (x >> i) & 0x1;
    result += bit;
}

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

```
long pcount_for_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;
}
```

- Initial test can be optimized away

Goto Version

```
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

Init

Test

Body

Update
Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements
Switch Statement Example

- **Multiple case labels**
  - Here: 5 & 6

- **Fall through cases**
  - Here: 2

- **Missing cases**
  - Here: 4

```c
long switch_eg (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

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

Jump Table

```
jtab:
    Targ0
    Targ1
    Targ2
    •
    •
    •
    Targ_{n-1}
```

Jump Targets

```
Targ0:  Code Block 0
Targ1:  Code Block 1
Targ2:  Code Block 2
Targ_{n-1}:  Code Block n-1
```

Translation (Extended C)

```c
goto *JTab[x];
```
Switch Statement Example

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

Setup:

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

What range of values takes default?

<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>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

Note that w not initialized here
Switch Statement Example

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

Setup:

switch_eg:
    movq %rdx, %rcx
    cmpq $6, %rdi        # x:6
    ja .L8              # Use default
    jmp *.L4(,%rdi,8)    # goto *JTab[x]

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

Indirect jump

From Day 6
Assembly Setup Explanation

■ **Table Structure**
  - Each target requires 8 bytes
  - Base address at .L4

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

  - **Indirect:** `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 ≤ x ≤ 6
Jump Table

Jump table

```
.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;
}
```
Code Blocks ($x == 1$)

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

Register Use(s)

<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>%rdx</td>
<td>Argument $z$</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

.L3:

```asm
movq %rsi, %rax  # y
imulq %rdx, %rax  # y*z
ret
```
long w = 1;
    . . .
switch(x) {
    . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
    . . .
}
Code Blocks \((x == 2, x == 3)\)

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

### Register Use(s)

<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>%rdx</td>
<td>Argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

### Machine Code

- **L5:**
  - `movq %rsi, %rax`
  - `cqto`
  - `idivq %rcx`  \# \(y/z\)
  - `jmp .L6`  \# goto merge

- **L6:**
  - `merge:
    - `addq %rcx, %rax`  \# \(w += z\)
    - `ret`
Code Blocks (x == 5, x == 6, default)

```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>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch

- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control

- **Standard Techniques**
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)
Summary

■ Today
  ▪ Control: Condition codes
  ▪ Conditional branches & conditional moves
  ▪ Loops
  ▪ Switch statements

■ Next Time
  ▪ Stack
  ▪ Call / return
  ▪ Procedure call discipline
Per-lecture feedback

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- **The form is anonymous.**
  (It checks that you’re at Illinois Tech to filter abuse, but I don’t see who submitted any of the forms.)
- [https://forms.gle/qoeEbBuTYXo5FiU1A](https://forms.gle/qoeEbBuTYXo5FiU1A)
- I’ll remind about this at each lecture.