

# x86-64 Programming III

CSE 351 Winter 2020

## Instructor:

Ruth Anderson

## Teaching Assistants:

Jonathan Chen

Justin Johnson

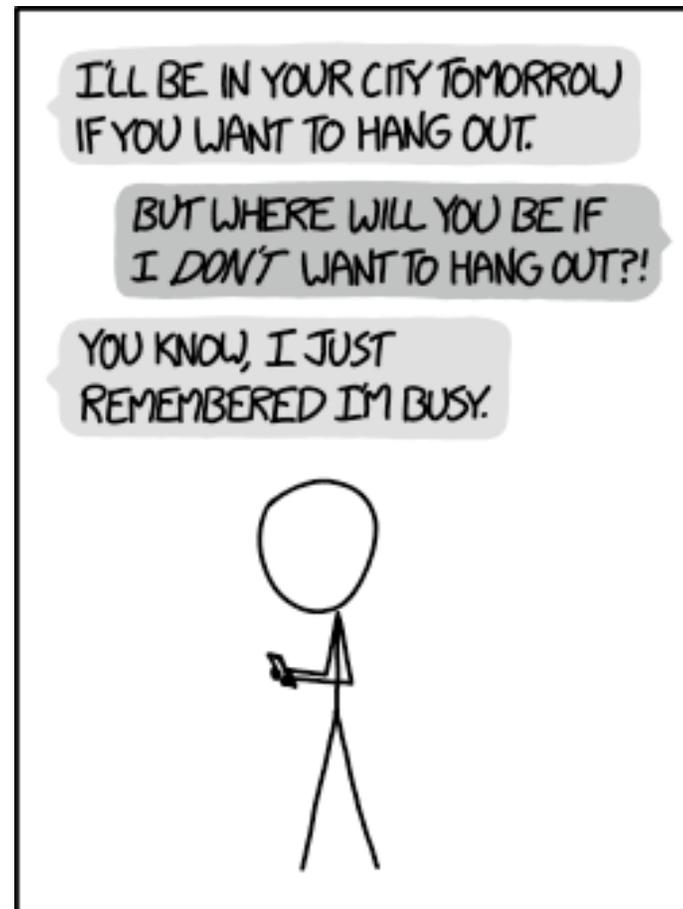
Porter Jones

Josie Lee

Jeffery Tian

Callum Walker

Eddy (Tianyi) Zhou



WHY I TRY NOT TO BE  
PEDANTIC ABOUT CONDITIONALS.

<http://xkcd.com/1652/>

# Administrivia

- ❖ Lab 2 due next Friday (2/07)
  - Ideally want to finish well before the midterm
  - Optional GDB Tutorial homework on Gradescope
  
- ❖ Section tomorrow on Assembly
  - Bring your midterm reference sheets!
  - If possible, bring your laptop to get some GDB practice
  - BEFORE section, please download this file on attu or CSE VM  
(Copy and paste this command to a terminal):  
wget <https://courses.cs.washington.edu/courses/cse351/20wi/sections/01/code/calculator.c>
  
- ❖ Midterm: 2/10, during lecture
  - You will be provided a fresh reference sheet

# GDB Demo

- ❖ The `movz` and `movs` examples 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 `stepi/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 (*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
    
```

		(op) s, d
<b>je</b>	"Equal"	d (op) s == 0
<b>jne</b>	"Not equal"	d (op) s != 0
<b>js</b>	"Sign" (negative)	d (op) s < 0
<b>jns</b>	(non-negative)	d (op) s >= 0
<b>jg</b>	"Greater"	d (op) s > 0
<b>jge</b>	"Greater or equal"	d (op) s >= 0
<b>jl</b>	"Less"	d (op) s < 0
<b>jle</b>	"Less or equal"	d (op) s <= 0
<b>ja</b>	"Above" (unsigned >)	d (op) s > 0U
<b>jb</b>	"Below" (unsigned <)	d (op) s < 0U

# Choosing instructions for conditionals

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

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

```

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:   aLSB == 0
jne:  aLSB == 1
    
```

# Choosing instructions for conditionals

		<code>cmp a,b</code>	<code>test a,b</code>
<code>je</code>	"Equal"	<code>b == a</code>	<code>b&amp;a == 0</code>
<code>jne</code>	"Not equal"	<code>b != a</code>	<code>b&amp;a != 0</code>
<code>js</code>	"Sign" (negative)	<code>b-a &lt; 0</code>	<code>b&amp;a &lt; 0</code>
<code>jns</code>	(non-negative)	<code>b-a &gt;= 0</code>	<code>b&amp;a &gt;= 0</code>
<code>jg</code>	"Greater"	<code>b &gt; a</code>	<code>b&amp;a &gt; 0</code>
<code>jge</code>	"Greater or equal"	<code>b &gt;= a</code>	<code>b&amp;a &gt;= 0</code>
<code>jl</code>	"Less"	<code>b &lt; a</code>	<code>b&amp;a &lt; 0</code>
<code>jle</code>	"Less or equal"	<code>b &lt;= a</code>	<code>b&amp;a &lt;= 0</code>
<code>ja</code>	"Above" (unsigned >)	<code>b &gt; a</code>	<code>b&amp;a &gt; 0U</code>
<code>jb</code>	"Below" (unsigned <)	<code>b &lt; a</code>	<code>b&amp;a &lt; 0U</code>

Register	Use(s)
<code>%rdi</code>	argument x
<code>%rsi</code>	argument y
<code>%rax</code>	return value

```

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

*x-3 ≥ 0*  
*x ≥ 3*

```

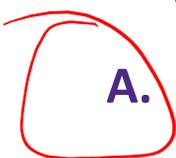
cmpq $3, %rdi // x-3
jge T2
T1: # x < 3:
    movq $1, %rax
    ret
T2: # !(x < 3):
    movq $2, %rax
    ret
    
```

# Polling Question

Register	Use(s)
%rdi	1 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> argument (y)
%rax	return value

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

Vote at <http://pollev.com/rea>



- A. `cmpq %rsi, %rdi`  
`jle .L4`
- B. `cmpq %rsi, %rdi`  
`jg .L4`
- C. `testq %rsi, %rdi`  
`jle .L4`
- D. `testq %rsi, %rdi`  
`jg .L4`
- E. We're lost...

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

# Choosing instructions for conditionals

		cmp a,b	test a,b
<b>je</b>	"Equal"	② <u>b<sup>x</sup> == y<sup>a</sup></u>	③ <u>b&amp;a == 0</u>
<b>jne</b>	"Not equal"	b != a	b&a != 0
<b>js</b>	"Sign" (negative)	b-a < 0	b&a < 0
<b>jns</b>	(non-negative)	b-a >= 0	b&a >= 0
<b>jg</b>	"Greater"	b > a	b&a > 0
<b>jge</b>	"Greater or equal"	b >= a	b&a >= 0
<b>jl</b>	"Less"	① <u>b<sup>x</sup> &lt; 3<sup>a</sup></u>	b&a < 0
<b>jle</b>	"Less or equal"	b <= a	b&a <= 0
<b>ja</b>	"Above" (unsigned >)	b > a	b&a > 0U
<b>jb</b>	"Below" (unsigned <)	b < a	b&a < 0U

❖ <https://godbolt.org/z/GNxpqv>

```

if (x < 3 && x == y) {
    return 1;
} else {
    return 2;
}
    
```

*Handwritten notes:* %al above x < 3, %bl above x == y. An arrow points to the else block with the text: "do this if either %al or %bl are False".

```

① cmpq $3, %rdi
   setl %al
   } %al = (x < 3)

② cmpq %rsi, %rdi
   sete %bl
   } %bl = (x == y)

③ testb %al, %bl
   je T2 ← jump to T2 if (%al & %bl) = 0

T1: # x < 3 && x == y:
    movq $1, %rax
    ret

T2: # else
    movq $2, %rax
    ret
    
```

# Labels

```
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
```

0x

- ❖ A jump changes the program counter (`%rip`)
  - `%rip` tells the CPU the *address* of the next instruction 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

```
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;
}
```

conditional jump  
unconditional jump  
labels (addresses)

cmp  
jle  
jmp

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

# Compiling Loops

C/Java code:

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

Assembly code:

```
loopTop:  testq %rax, %rax } !Test  
          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

*do {  
Body  
} while (Test);*

*"i=0" "i<n" "i++"  
for (Init; Test; Update) {  
Body  
}*

Do...while

*Loop: Body  
if (Test) goto Loop;*

For loop

*Init  
Loop: if (!Test) goto Exit;  
Body  
Update  
goto Loop;  
Exit:*

# Compiling Loops

*all jump instructions  
update the program counter (?rip)*

## While Loop:

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

x86-64:

```
loopTop:    testq %rax, %rax } ~Test
            je    loopDone
            <loop body code>
            jmp   loopTop

loopDone:
```

*sum == 0*

## Do-while Loop:

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

x86-64:

```
loopTop:
    <loop body code>
    testq %rax, %rax } Test
    jne   loopTop

loopDone:
```

## While Loop (ver. 2):

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

x86-64:

```
loopTop:    testq %rax, %rax } ~Test
            je    loopDone
            <loop body code>
            testq %rax, %rax } Test
            jne   loopTop

loopDone:
```

*do-while loop*

# 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**

```
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;
}
```

## 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

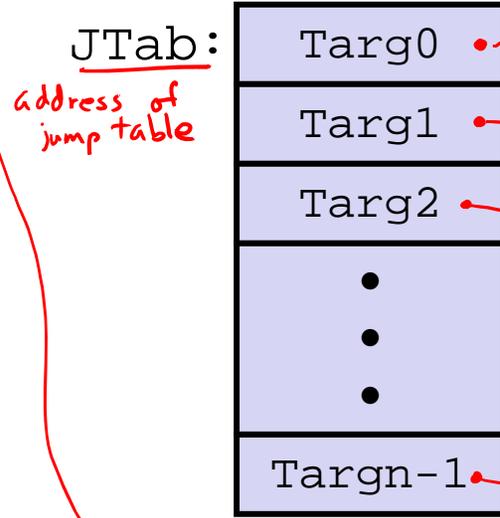
*Memory*

## Switch Form

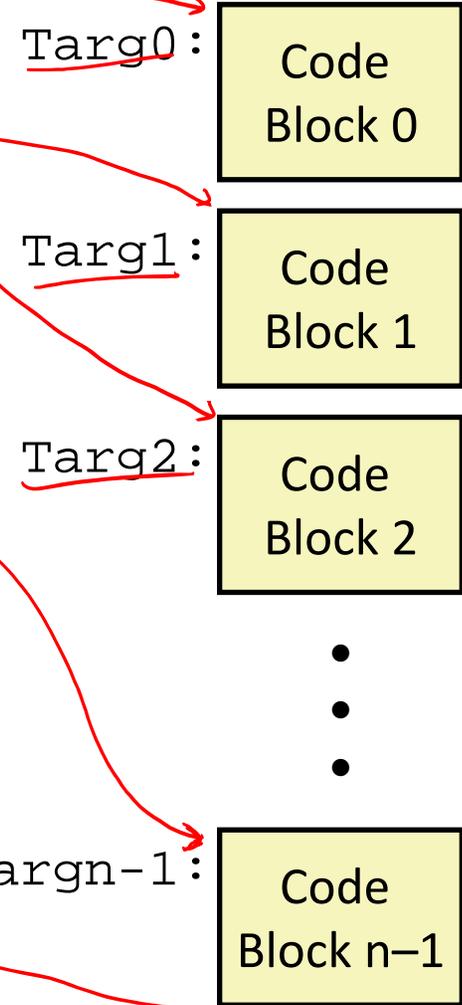
```

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

## Jump Table



## Jump Targets



## Approximate Translation

```

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

*like an array of pointers*

# Jump Table Structure

C code:

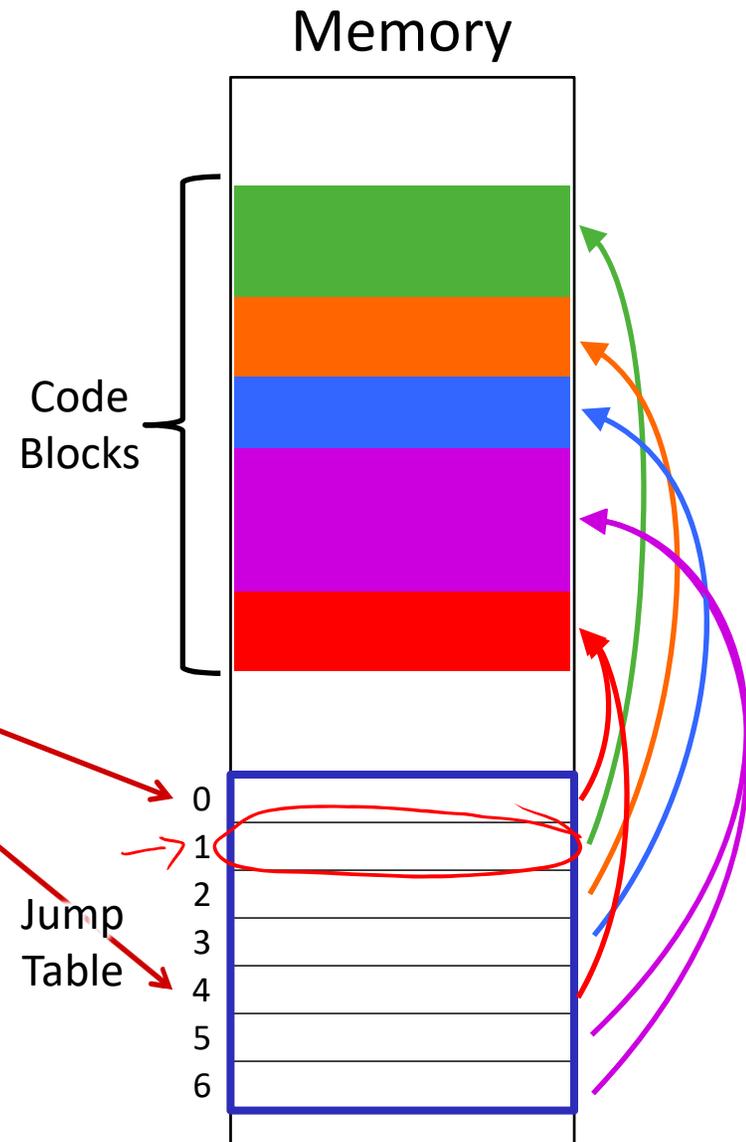
```

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  $x \leq 6$ :

```

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



# Switch Statement Example

```

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

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

Note compiler chose to not initialize w

```

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

jump to default case if x > 6 (unsigned)

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

Take a look!  
<https://godbolt.org/z/aY24el>

# Switch Statement Example

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

## Jump table

.section	.rodata	
.align 8		
<b>.L4:</b>		
.quad	<b>.L8</b>	# 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

following data is a "quad word" = 8 bytes

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

**Indirect jump** →  $D + R_i * S$   
 addr of jump table → D  
 x →  $R_i$   
 sizeof(void\*) → S

# 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`

*%rip* | ←

## ❖ 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  $0 \leq x \leq 6$

*Mem[D + Reg[Ri] \* 5]*

## 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
```

# BONUS SLIDES

Slides that expand on the simple switch code in assembly. **These slides expand on material covered today, so while you don't need to read these, the information is "fair game."**

# 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
    
```

```

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;  
  . . .  
}
```

Register	Use(s)
%rdi	1 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> argument (y)
%rdx	3 <sup>rd</sup> argument (z)
%rax	Return value

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

# Handling Fall-Through

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

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

```
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
.L6:                                # merge:
    addq    %rcx, %rax               # w += z
    ret

```

Register	Use(s)
%rdi	1 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> argument (y)
%rdx	3 <sup>rd</sup> argument (z)
%rax	Return value

# Code Blocks (rest)

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

Register	Use(s)
%rdi	1 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> argument (y)
%rdx	3 <sup>rd</sup> argument (z)
%rax	Return value

```
.L7:                # Case 5,6:  
    movl    $1, %eax    # w = 1  
    subq   %rdx, %rax   # w -= z  
    ret  
.L8:                # Default:  
    movl    $2, %eax    # 2  
    ret
```