CSCI-UA.0201

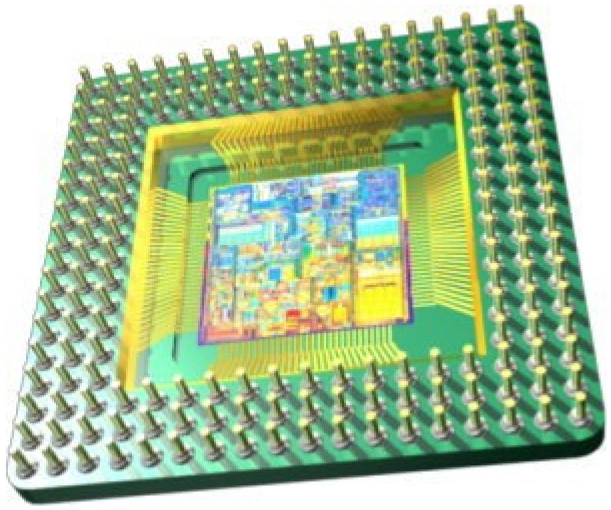
Computer Systems Organization

Machine-Level Programming III

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:

- Clark Barrett
- Jinyang Li
- Randy Bryant
- Dave O'Hallaron



How to implement loops?

- do-while
- while
- for

“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
- Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

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

Register	Use(s)
%rdi	Argument x
%rax	result

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

General “Do-While” Translation

C Code

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

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

General “While” Translation #1

- “Jump-to-middle” translation

While version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

While Loop Example #1

C Code

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

Jump to Middle

```
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” conversion

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

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

Do-While

```
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

```
for (Init; Test; Update )

      Body
```

```
#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 < WSIZE
```

Update

```
i++
```

Body

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

“For” Loop □ 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;
}
```

Switch statement

Example

```
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;
}
```

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

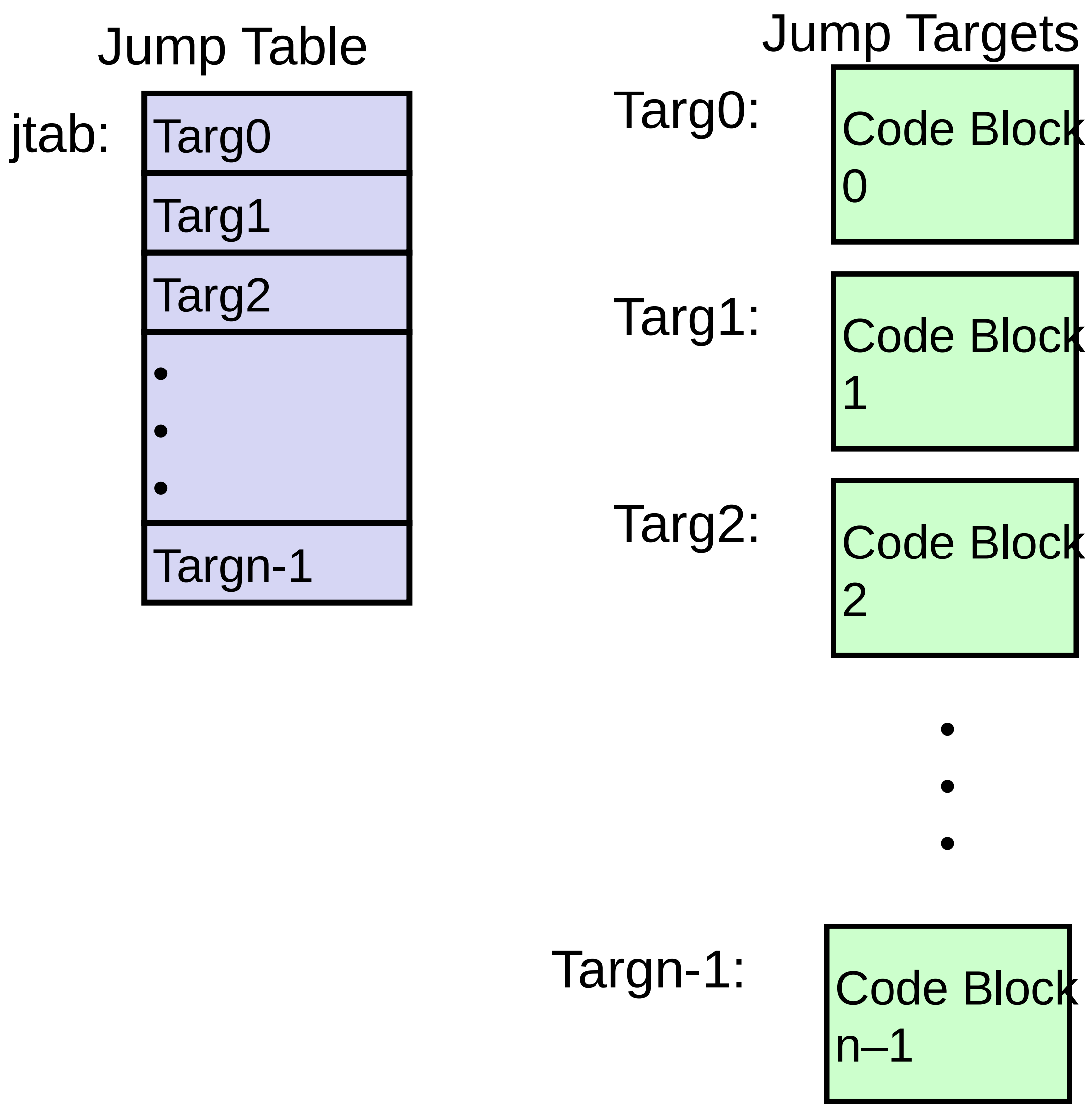
Jump Table Structure

Switch Form

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

Translation (Extended C)

```
goto *JTab[x];
```



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
    jg      .L8
    jmp     *.L4(,%rdi,8)
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	Return value

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;
}
```

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
```

Setup:

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

Indirect jump

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

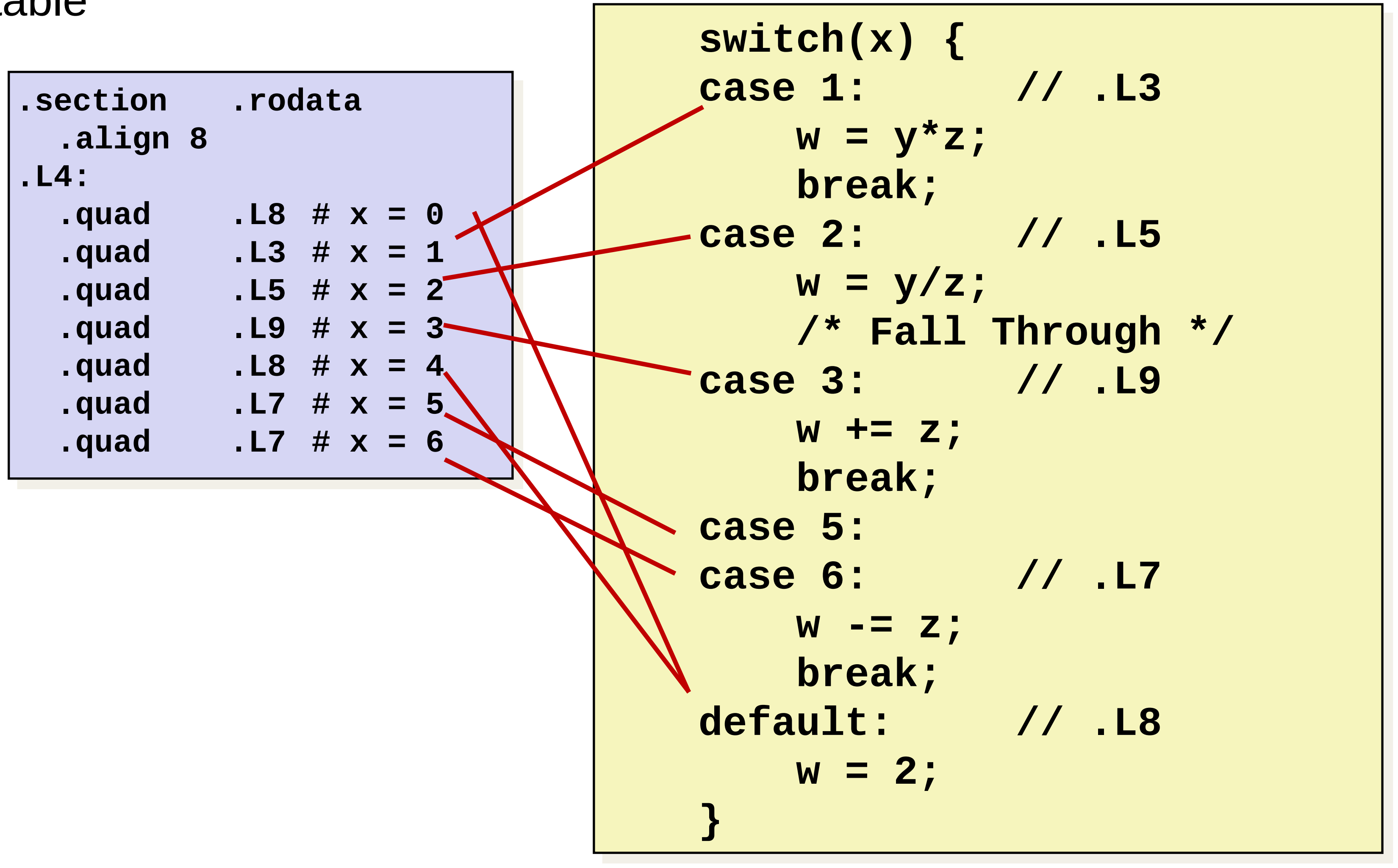
  - Indirect: **`jmp *.L4(,%rdi,8)`**
  - Start of jump table: .L4
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address .L4 + x*8
    - Only for $0 \le x \le 6$

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

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;
}
```

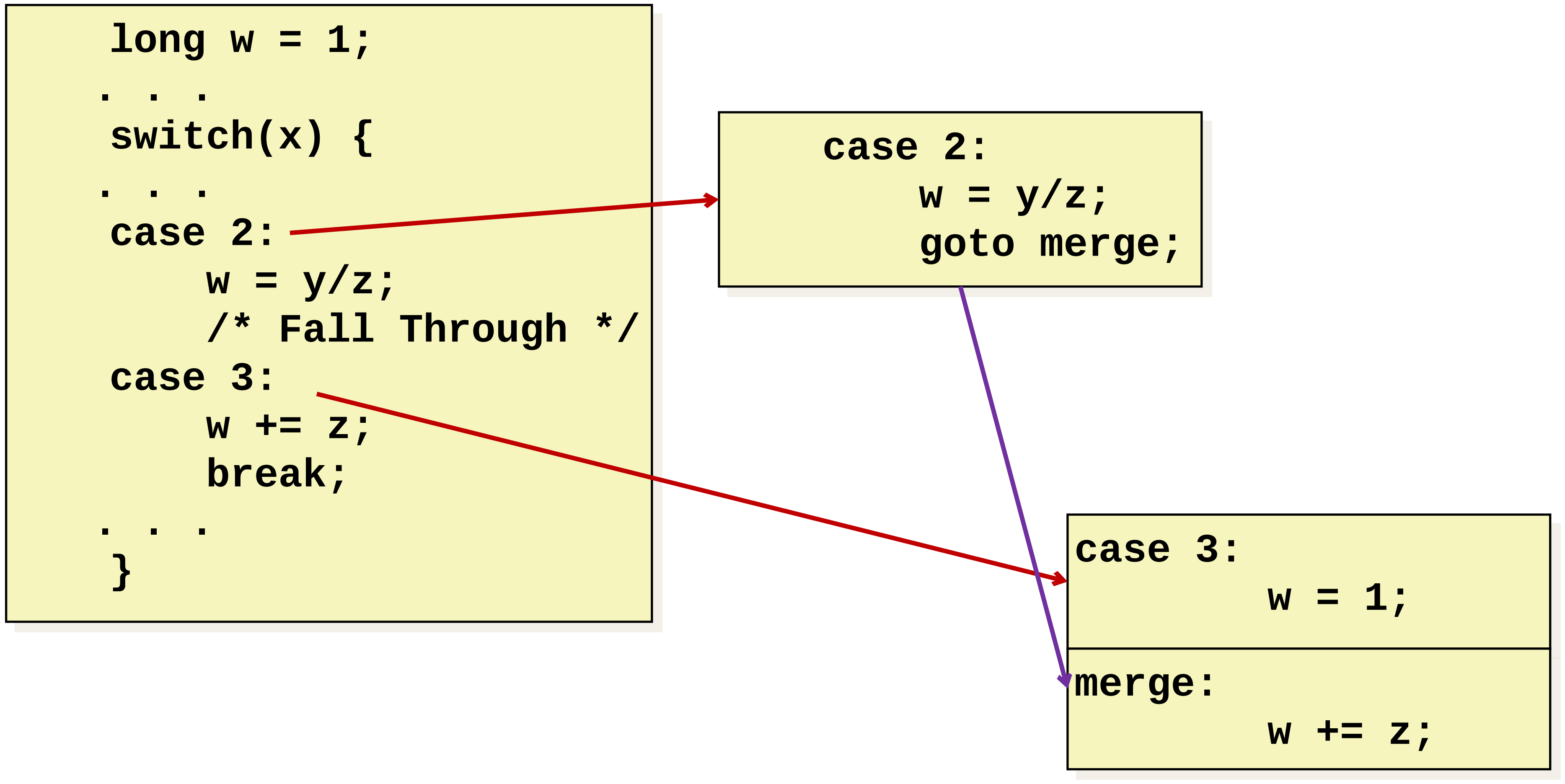
Code Blocks (x == 1)

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

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

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

Handling Fall-Through



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

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

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

```
merge:
        w += z;
```

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

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

```
.L5:                        # Case 2
   movq     %rsi, %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
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

Code Blocks (x == 5, x == 6, default)

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

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

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

Conclusions

- 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