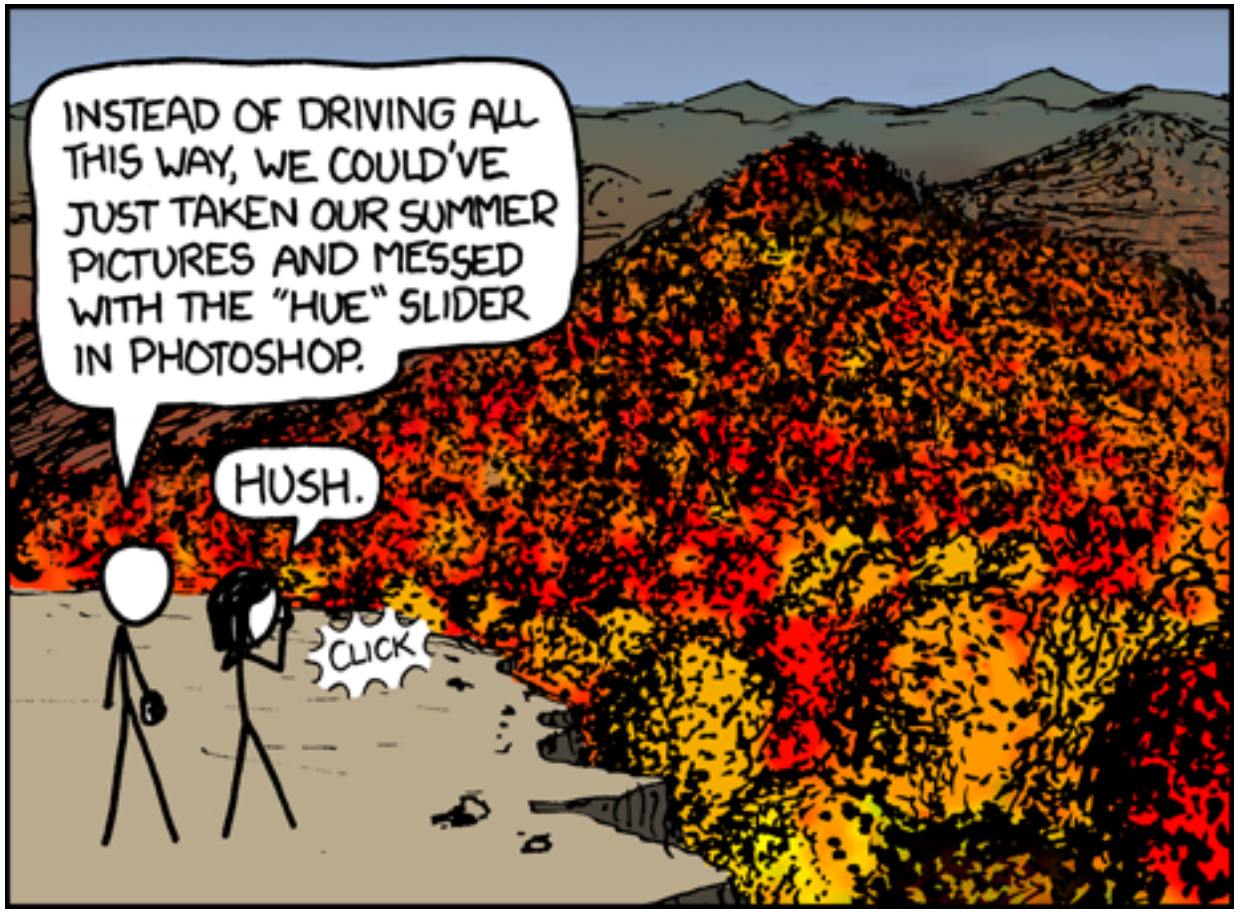
x86 Programming III

CSE 351 Autumn 2016



http://xkcd.com/648/

Administrivia

Peer Instruction Question

- Which conditional statement properly fills in the following blank?

```
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 || j < i` **(C)** `j ≤ 1 || j ≥ i`

(B) `j > 1 && j < i` **(D)** `j ≤ 1 && j ≥ i`

Jumping

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

Instruction	Condition	Description
`je` *target*	`ZF`	Equal / Zero
`jne` *target*	`~ZF`	Not Equal / Not Zero
`js` *target*	`SF`	Negative
`jns` *target*	`~SF`	Nonnegative
`jg` *target*	`~(SF^OF)&~ZF`	Greater (Signed)
`jge` *target*	`~(SF^OF)`	Greater or Equal (Signed)
`jl` *target*	`(SF^OF)`	Less (Signed)
`jle` *target*	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja` *target*	`~CF&~ZF`	Above (unsigned)
`jb` *target*	`CF`	Below (unsigned)

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;
}
```

- 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 != 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:

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

Assembly code:

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

Do-while loop

C/Java code:

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

Assembly code:

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

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 backward branch to continue looping
- Only take branch when "while" condition holds

Do-While Loop Compilation

Register	Use(s)
%rdi	1st argument (x)
%rax	ret val (result)

Goto Version

```
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)
```

General <u>Do-While Loop</u> Translation

C Code

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

Goto Version

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

- *Body:* {
    Statement$_1$;
    …
    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

```
while (Test)
  Body
```

Goto Version

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

While Loop Example – Translation #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;
}
```

- Used with `–Og`
- Compare to do-while version of function
- Initial `goto` starts loop at `test`

General While Loop - Translation #2

While version

```
while (Test)
  Body
```

Do-While Version

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

- "Do-while" conversion
- Used with -O1

Goto Version

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

While Loop Example – Translation #2

C Code

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

Do-While Version

```
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

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;
}
```

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

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
  (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;                          Init
   if (!(i < WSIZE))               !Test
     goto done;
 loop:
   unsigned bit =                  Body
     (x >> i) & 0x1;
   result += bit;
   i++;                            Update
   if (i < WSIZE)                  Test
     goto loop;
 done:
   return result;
}
```

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

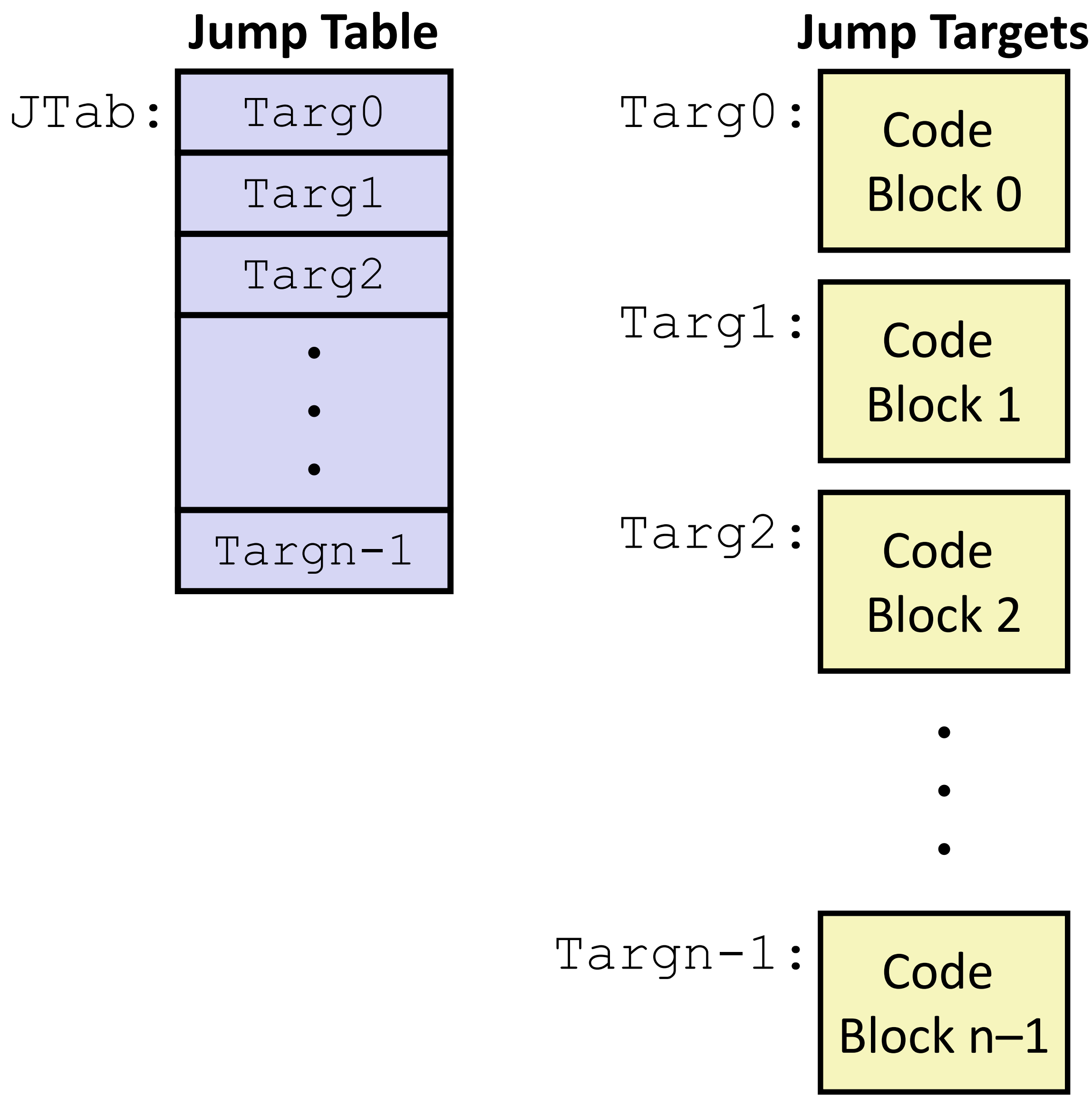
- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

- Implemented with:
  - *Jump table*
  - *Indirect jump instruction*

Jump Table Structure

Switch Form

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



Approximate Translation

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

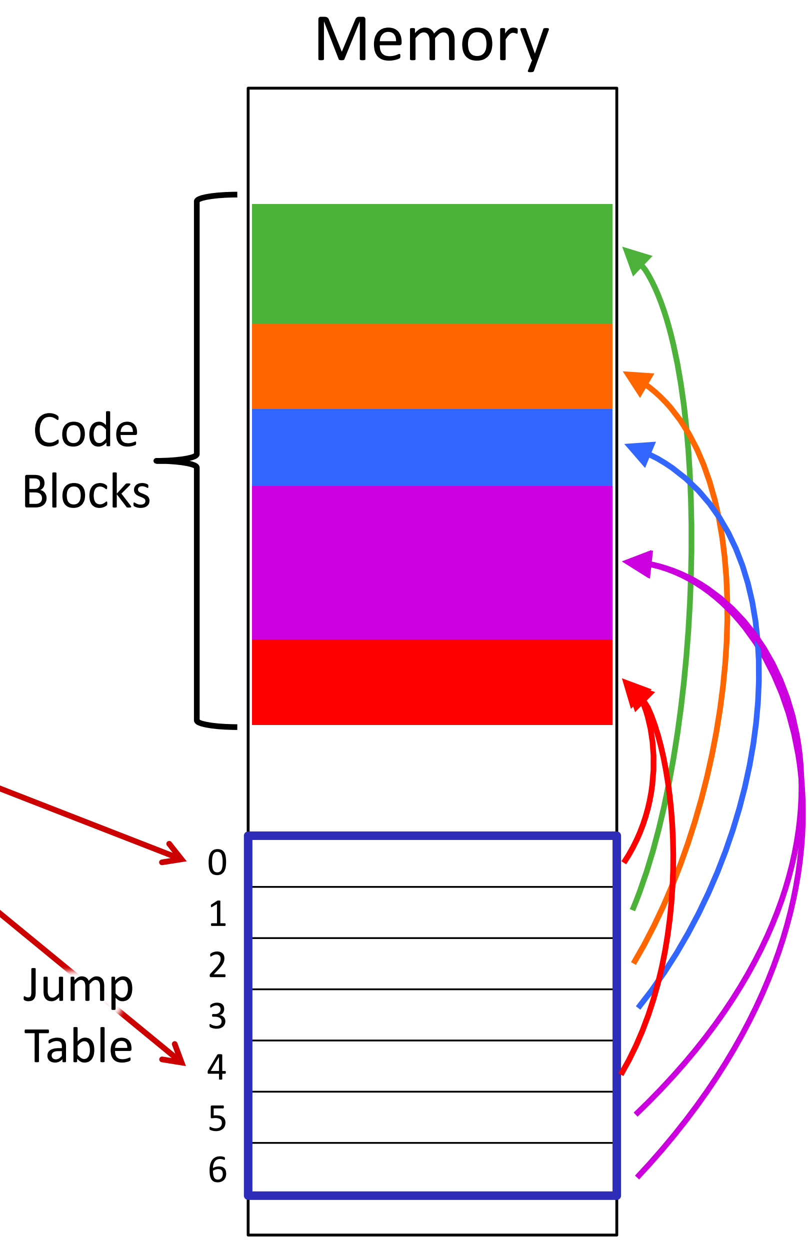
Jump Table Structure

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 <= 6:

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



Switch Statement Example

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

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

Note compiler chose to not initialize w

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

Take a look!
https://godbolt.org/g/NAxYVw

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;
}
```

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

Indirect jump

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
```

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes (address)
  - Base address at `.L4`

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
```

- **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 $0 \leq x \leq 6$

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;
}
```

Code Blocks (x == 1)

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

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

```
.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)

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

```
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
```

Code Blocks (rest)

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

```
switch (x) {
    . . .
    case 5:   // .L7
    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
```

Question

- Would you implement this with a jump table?

```
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 SLIDES

Bonus content (nonessential). Does contain examples.

- Conditional Operator with Jumps
- Conditional Move

Bonus Content (nonessential)

Conditional Operator with Jumps

C Code

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

```
if (Test)
   val = Then-Expr;
else
   val = Else-Expr;
```

Example:

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

Goto Version

```
   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

Conditional Move

Bonus Content (nonessential)
more details at end of slides

- 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

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

```
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 <=,
  ret                #    result=else_val
```

Bonus Content (nonessential)

Using Conditional Moves

- Conditional Move Instructions
  - **cmov*C*** 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

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

"Goto" Version

```
result = Then_Expr;
else_val = Else_Expr;
nt = !Test;
if (nt) result = else_val;
return result;
```

Bonus Content (nonessential)

Conditional Move Example

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

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

```
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 <=, result = else_val
   ret
```

Bonus Content (nonessential)

Bad Cases for Conditional Move

Expensive Computations

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

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

Risky Computations

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

- Both values get computed
- May have undesirable effects

Computations with side effects

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

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