CS 33

Machine Programming (3)

Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

Today

- **Loops**
- **Switch statements**

Supplied by CMU.

"Do-While" Loop Example

C Code

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

Goto Version

```
int pcount_do(unsigned x)
{
  int 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 either to continue looping or to exit loop**

Supplied by CMU.

"Do-While" Loop Compilation

Goto Version

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

Registers:

%edx	x
%eax	result

```
  movl   $0, %eax      #   result = 0
.L2:                   # loop:
  movl   %edx, %ecx
  andl   $1, %ecx      #   t = x & 1
  addl   %ecx, %eax    #   result += t
  shrl   %edx          #   x >>= 1
  jne    .L2           #   if !0, goto loop
```


Supplied by CMU.

Note that the condition codes are set as part of the execution of the shrl instruction.

General "Do-While" Translation

C Code

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

Goto Version

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

- **Body:**

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

- **Test returns integer**
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

Supplied by CMU.

“While” Loop Example

C Code

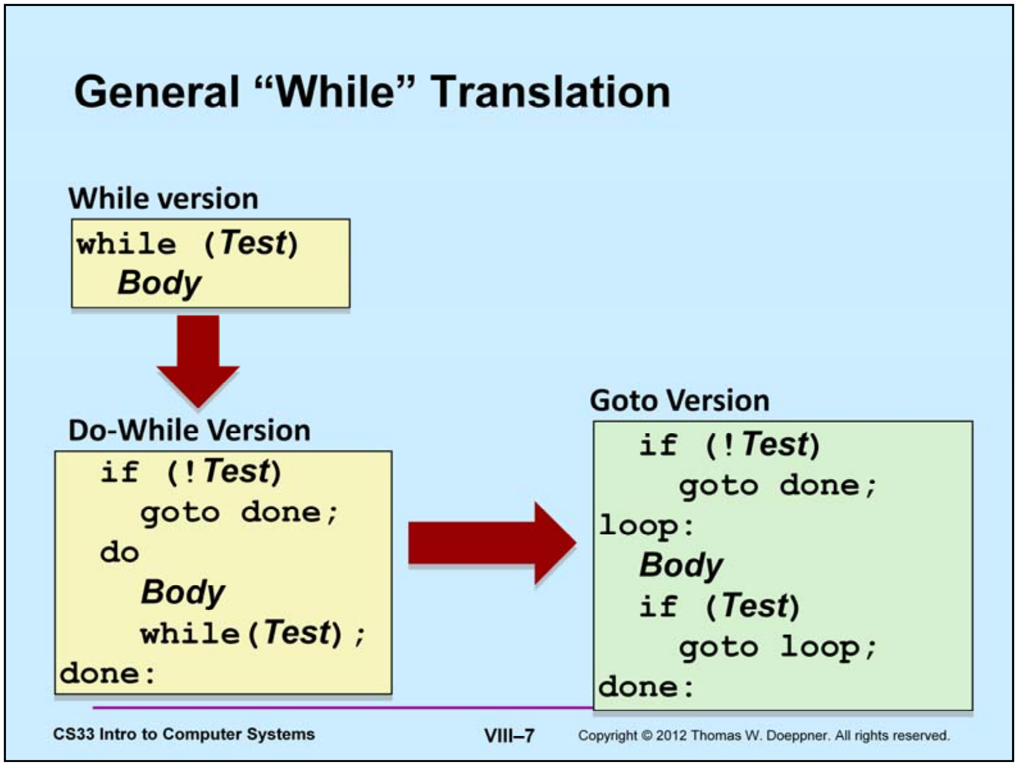
```
int pcount_while(unsigned x) {
  int result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Goto Version

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

- **Is this code equivalent to the do-while version?**
  - **must jump out of loop if test fails**

Supplied by CMU.



Supplied by CMU.

“For” Loop Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- **Is this code equivalent to other versions?**

Supplied by CMU.

“For” Loop Form

General Form

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

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

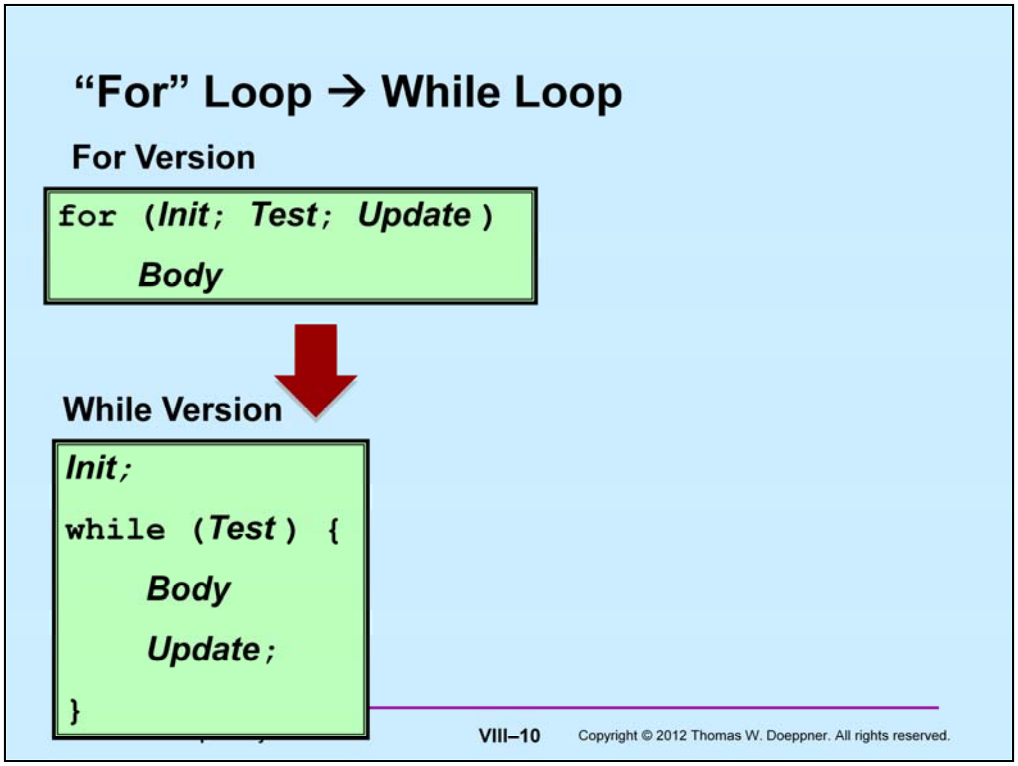
Update

```
i++
```

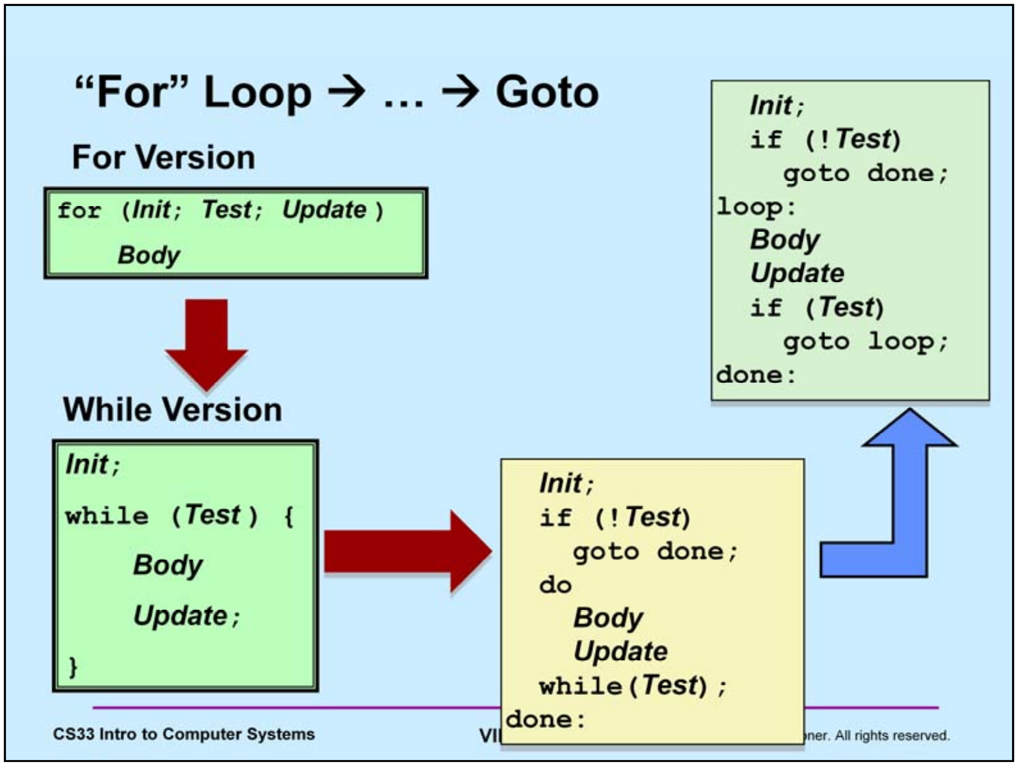
Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```


Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

"For" Loop Conversion Example

C Code

```c
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

Initial test can be optimized away

Goto Version

```c
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               !Test
    goto done;
 loop:
  {                               Body
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;  Update
  if (i < WSIZE)  Test
    goto loop;
 done:
  return result;
}
```


Supplied by CMU.

Today

- Loops
- Switch statements

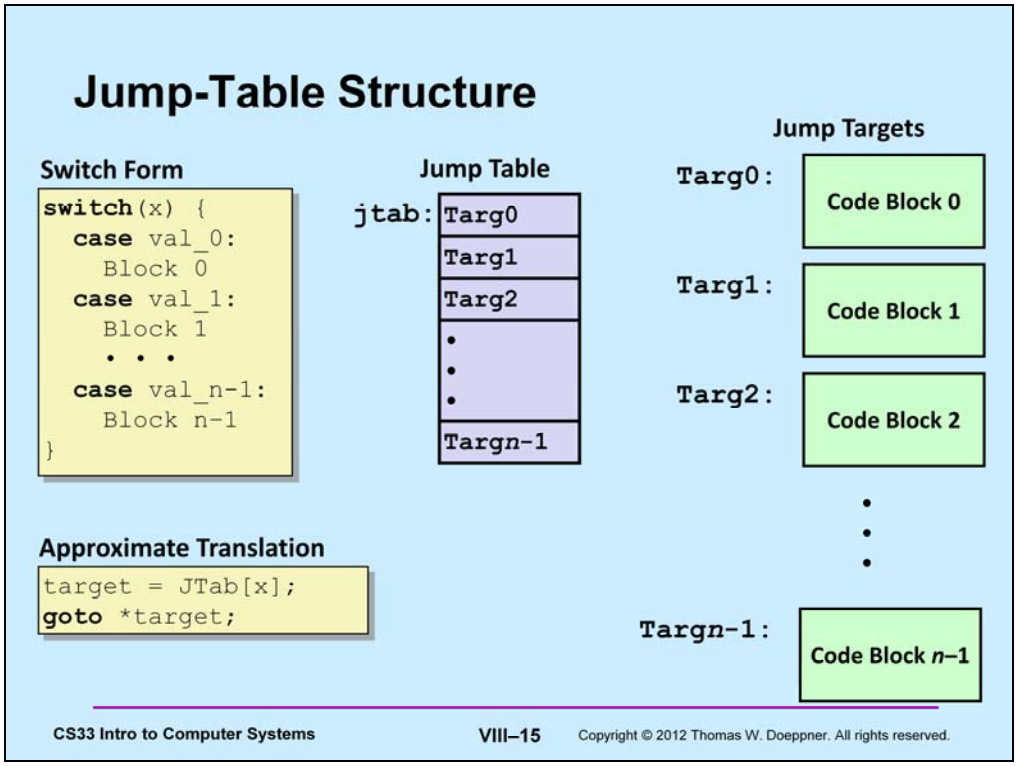
Supplied by CMU.

```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;
}
```

Switch-Statement Example

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

Supplied by CMU.



Supplied by CMU.

Switch-Statement Example (IA32)

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

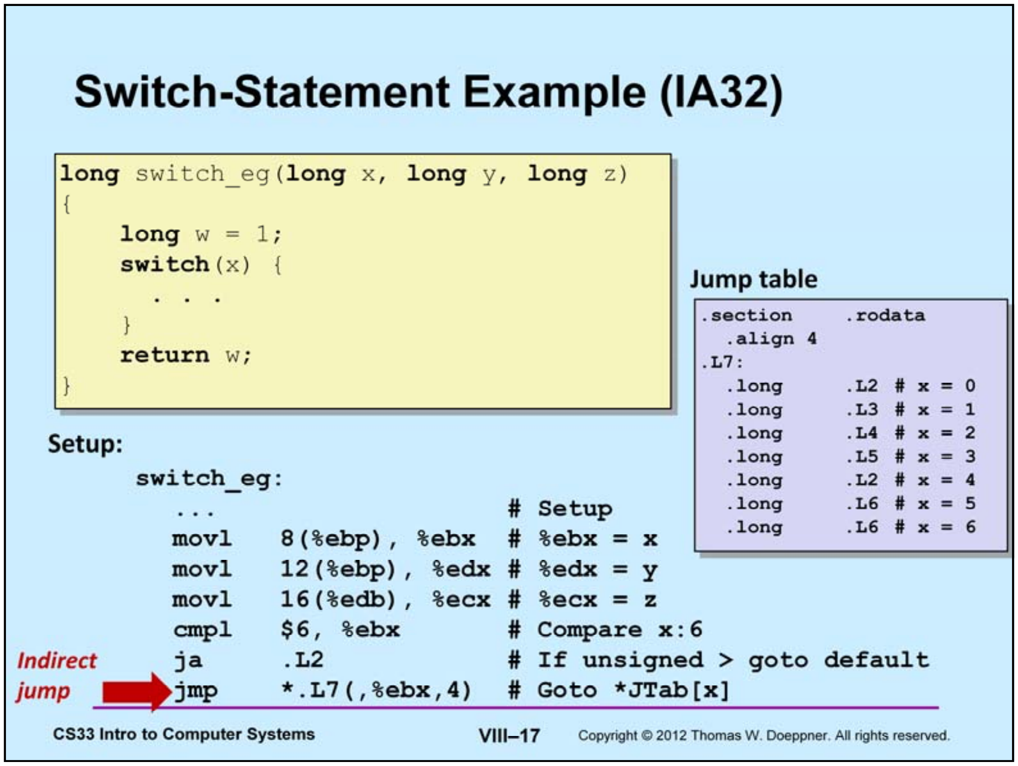
What range of values is covered by the default case?

Setup:

```
 switch_eg:
    ...                      # Setup
    movl   8(%ebp), %ebx     # %ebx = x
    movl   12(%ebp), %edx    # %edx = y
    movl   16(%edb), %ecx    # %ecx = z
    cmpl   $6, %ebx          # Compare x:6
    ja     .L2               # If unsigned > goto default
    jmp    *.L7(,%ebx,4)     # Goto *JTab[x]
```

Note that w not initialized here

Supplied by CMU.



Supplied by CMU.

Assembly-Setup Explanation

- **Table structure**
  - each target requires 4 bytes
  - base address at `.L7`

Jump table

```
.section   .rodata
  .align 4
.L7:
  .long    .L2 # x = 0
  .long    .L3 # x = 1
  .long    .L4 # x = 2
  .long    .L5 # x = 3
  .long    .L2 # x = 4
  .long    .L6 # x = 5
  .long    .L6 # x = 6
```

- **Jumping**

  direct: `jmp .L2`
  - jump target is denoted by label `.L2`

  indirect: `jmp *.L7(,%ebx,4)`
  - start of jump table: `.L7`
  - must scale by factor of 4 (labels have 32 bits = 4 Bytes on IA32)
  - fetch target from effective address `.L7 + ebx*4`
    - » only for $0 \le x \le 6$

Supplied by CMU.

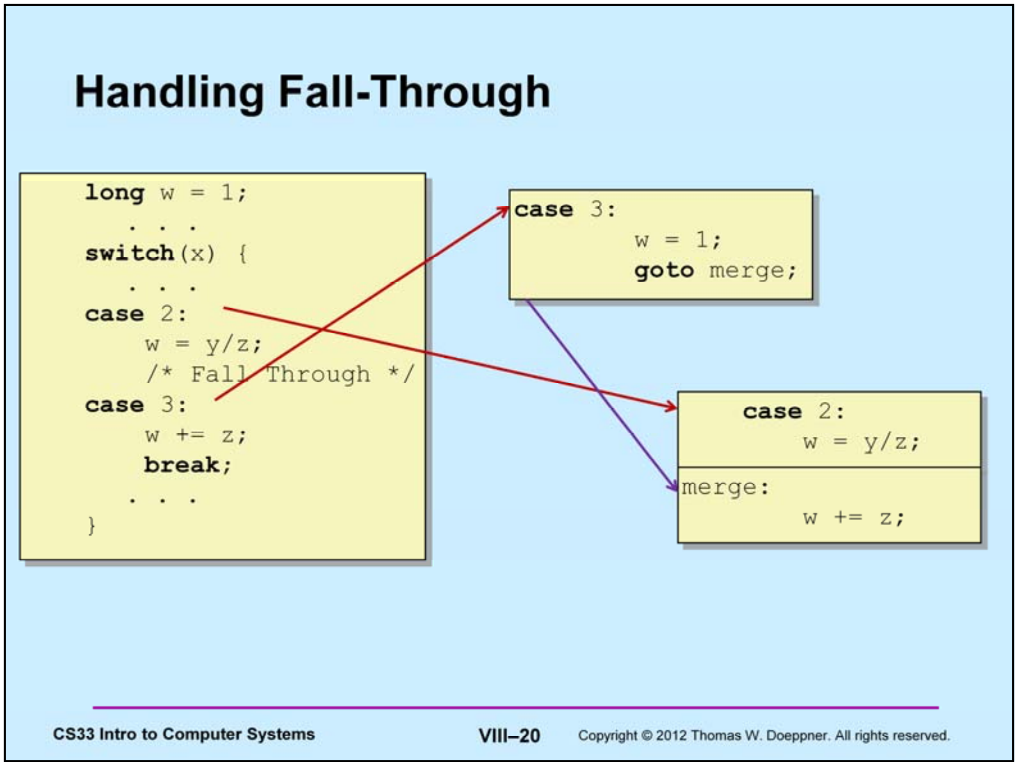
Jump Table

Jump table

```
.section   .rodata
  .align 4
.L7:
  .long    .L2 # x = 0
  .long    .L3 # x = 1
  .long    .L4 # x = 2
  .long    .L5 # x = 3
  .long    .L2 # x = 4
  .long    .L6 # x = 5
  .long    .L6 # x = 6
```

```
switch(x) {
case 1:        // .L3
     w = y*z;
     break;
case 2:        // .L4
     w = y/z;
     /* Fall Through */
case 3:        // .L5
     w += z;
     break;
case 5:
case 6:        // .L6
     w -= z;
     break;
default:       // .L2
     w = 2;
}
```


Supplied by CMU.



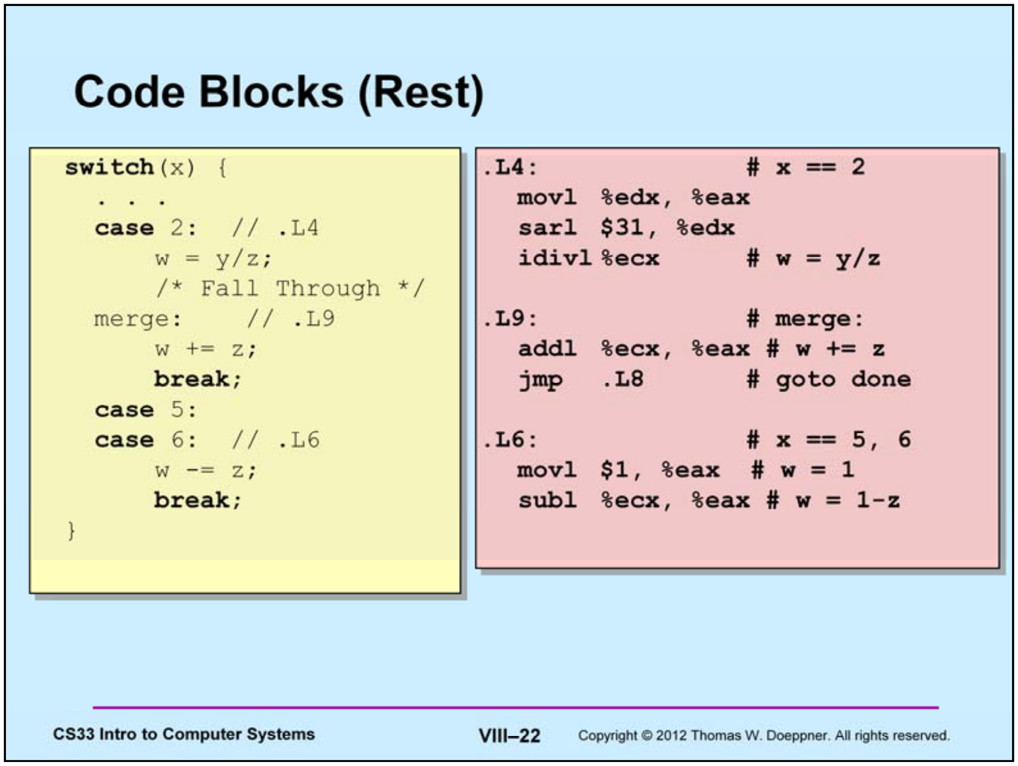
Supplied by CMU.

Code Blocks (Partial)

```
switch(x) {
case 1:       // .L3
      w = y*z;
      break;
 . . .
case 3:       // .L5
    w += z;
    break;
  . . .
default:      // .L2
    w = 2;
}
```

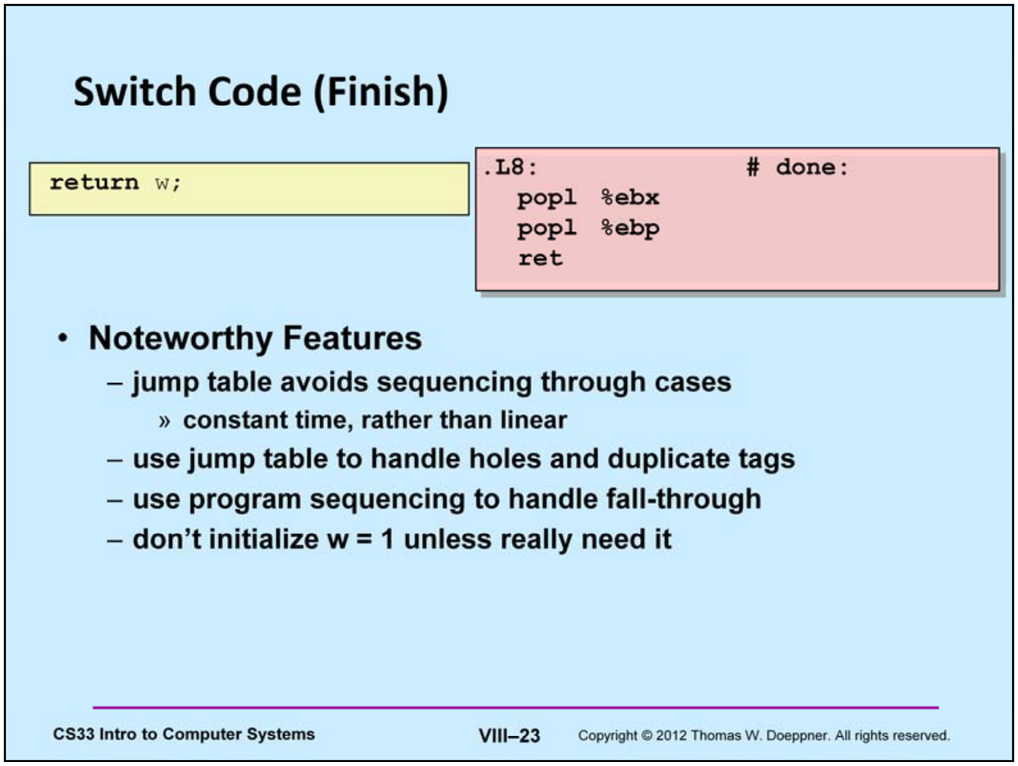
```
.L2:                # Default
  movl  $2, %eax    # w = 2
  jmp   .L8         # Goto done

.L5:                # x == 3
  movl  $1, %eax    # w = 1
  jmp   .L9         # Goto merge

.L3:                # x == 1
  movl  %ecx, %eax   # z
  imull %edx, %eax   # w = y*z
  jmp   .L8          # Goto done
```


Supplied by CMU.



Supplied by CMU.

The code following the .L4 label requires some explanation. The idivl instruction is of a special form in that it takes a 64-bit dividend, which is implicitly assumed to reside in registers edx and eax. y, which we want to be the dividend, is in edx. It is copied to eax by the movl instruction. The sarl instruction propagates the sign bit of edx across the entire register. Thus, if one considers edx to contain the most-significant bits of the dividend and eax to contain the least-significant bits, the pair of registers now contains the 64-bit version of y. The idivl instruction computes the quotient of dividing this 64-bit value by the 32-bit value contained in register ecx, which is z. The quotient goes into register eax (implicitly) and the remainder goes into register edx (and is ignored).



Supplied by CMU.

x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- Table entries 64 bits (pointers)
- Cases use revised code

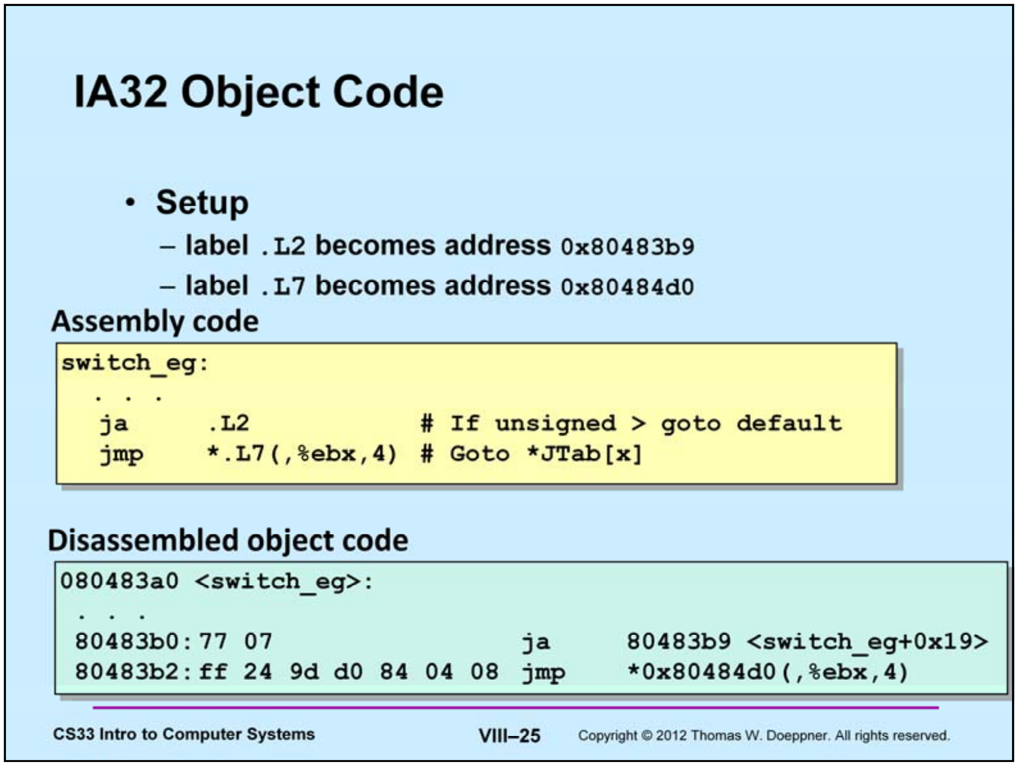
```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
  . . .
}
```

```
.L3:
  movq    %rdx, %rax
  imulq   %rsi, %rax
  ret
```

Jump Table

```
  .section .rodata
  .align 8
.L7:
  .quad    .L2    # x = 0
  .quad    .L3    # x = 1
  .quad    .L4    # x = 2
  .quad    .L5    # x = 3
  .quad    .L2    # x = 4
  .quad    .L6    # X = 5
  .quad    .L6    # x = 6
```


Supplied by CMU.



Supplied by CMU.

IA32 Object Code (cont.)

- **Jump table**
  - doesn't show up in disassembled code
  - can inspect using gdb

```
gdb switch
(gdb) x/7xw 0x80484d0
```

  - » e*x*amine *7* he*x*adecimal format "*w*ords" (4-bytes each)
  - » use command "`help x`" to get format documentation

```
0x80484d0:    0x080483b9    0x080483ca    0x080483d2    0x080483c0
0x80484e0:    0x080483b9    0x080483dd    0x080483dd
```


Supplied by CMU.

IA32 Object Code (cont.)

- **Deciphering jump table**

```
0x80484d0:   0x080483b9   0x080483ca   0x080483d2   0x080483c0
0x80484e0:   0x080483b9   0x080483dd   0x080483dd
```

Address	Value	x
0x80484d0	0x80483b9	0
0x80484d4	0x80483ca	1
0x80484d8	0x80483d2	2
0x80484dc	0x80483c0	3
0x80484e0	0x80483b9	4
0x80484e4	0x80483dd	5
0x80484e8	0x80483dd	6

Supplied by CMU.

Disassembled Targets

```
80483b9:   b8 02 00 00 00        mov    $0x2,%eax
80483be:   eb 24                 jmp    80483e4 <switch_eg+0x44>
80483c0:   b8 01 00 00 00        mov    $0x1,%eax
80483c5:   8d 76 00              lea    0x0(%esi),%esi # no-op
80483c8:   eb 0f                 jmp    80483d9 <switch_eg+0x39>
80483ca:   89 c8                 mov    %ecx,%eax
80483cc:   0f af c2              imul   %edx,%eax
80483cf:   90                    nop                   # no-op
80483d0:   eb 12                 jmp    80483e4 <switch_eg+0x44>
80483d2:   89 d0                 mov    %edx,%eax
80483d4:   c1 fa 1f              sar    $0x1f,%edx
80483d7:   f7 f9                 idiv   %ecx
80483d9:   01 c8                 add    %ecx,%eax
80483db:   eb 07                 jmp    80483e4 <switch_eg+0x44>
80483dd:   b8 01 00 00 00        mov    $0x1,%eax
80483e2:   29 c8                 sub    %ecx,%eax
80483e4:   5b                    pop    %ebx
80483e5:   5d                    pop    %ebp
80483e6:   c3                    ret
```

Matching Disassembled Targets

Value
0x80483b9
0x80483ca
0x80483d2
0x80483c0
0x80483b9
0x80483dd
0x80483dd

```
80483b9:   mov    $0x2,%eax
80483be:   jmp    80483e4 <switch_eg+0x44>
80483c0:   mov    $0x1,%eax
80483c5:   lea    0x0(%esi),%esi # no-op
80483c8:   jmp    80483d9 <switch_eg+0x39>
80483ca:   mov    %ecx,%eax
80483cc:   imul   %edx,%eax
80483cf:   nop                  # no-op
80483d0:   jmp    80483e4 <switch_eg+0x44>
80483d2:   mov    %edx,%eax
80483d4:   sar    $0x1f,%edx
80483d7:   idiv   %ecx
80483d9:   add    %ecx,%eax
80483db:   jmp    80483e4 <switch_eg+0x44>
80483dd:   mov    $0x1,%eax
80483e2:   sub    %ecx,%eax
80483e4:   pop    %ebx
80483e5:   pop    %ebp
80483e6:   ret
```

Summarizing

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - conditional jump
  - conditional move
  - indirect jump
  - compiler generates code sequence to implement more complex control
- Standard Techniques
  - loops converted to do-while form
  - large switch statements use jump tables
  - sparse switch statements may use decision trees

Supplied by CMU.