CS 33

Machine Programming (2)

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," 2^{nd} 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.

Processor State (x86-64, Partial)

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

%rip

CF ZF SF OF

condition codes

Condition Codes (Implicit Setting)

- **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: *`addl/addq`* *Src,Dest* ↔ `t = a+b`

  CF set if carry out from most significant bit or borrow (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) || (a<0 && b<0 && t>=0)`

- **Not set by `lea` instruction**

Supplied by CMU.

Condition Codes (Explicit Setting: Compare)

- **Explicit setting by compare instruction**

  `cmpl/cmpq` ***src2, src1***

      compares src1:src2

  `cmpl b,a` **like computing** `a-b` **without setting destination**

  CF set if carry out from most significant bit or borrow (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)`

Supplied by CMU.

Condition Codes (Explicit Setting: Test)

- **Explicit setting by test instruction**

  `testl/testq` ***src2*, *src1***
  `testl 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`

Supplied by CMU.

Reading Condition Codes

- **SetX instructions**
  - set single byte based on combinations of condition codes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Supplied by CMU.

Reading Condition Codes (Cont.)

- **SetX instructions:**
  - set single byte based on combination of condition codes
- **Uses byte registers**
  - does not alter remaining 7 bytes
  - typically use `movzbl` to finish job

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

%rax	%eax	%ah	%al

Body

```
cmpl %esi, %edi     # compare x : y
setg %al            # %al = x > y
movzbl %al, %eax    # zero rest of %eax/%rax
```


Supplied by CMU, but converted to x86-64.

Jumping

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

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Supplied by CMU.

Quiz 1

What would be an appropriate description if the condition is ~CF?

a) above or equal (unsigned)
b) not less (signed)
c) incomparable

Jumping

- jX instruction
  - Jump to differ

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Supplied by CMU.

Conditional-Branch Example

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

```
absdiff:
    movl   %esi, %eax
    cmpl   %esi, %edi
    jle    .L6
    subl   %eax, %edi
    movl   %edi, %eax
    jmp .L7
.L6:
    subl %edi, %eax
.L7:
    ret
```

Body1

Body2a

Body2b

x in %edi

y in %esi

Supplied by CMU, but converted to x86-64.

Conditional-Branch Example (Cont.)

```c
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    movl   %esi, %eax
    cmpl   %esi, %edi       Body1
    jle    .L6
    subl   %eax, %edi
    movl   %edi, %eax       Body2a
    jmp .L7
.L6:
    subl %edi, %eax         Body2b
.L7:
    ret
```

- C allows “goto” as means of transferring control
  - closer to machine-level programming style
- Generally considered bad coding style

Supplied by CMU, but converted to x86-64.

General Conditional-Expression Translation

C Code

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

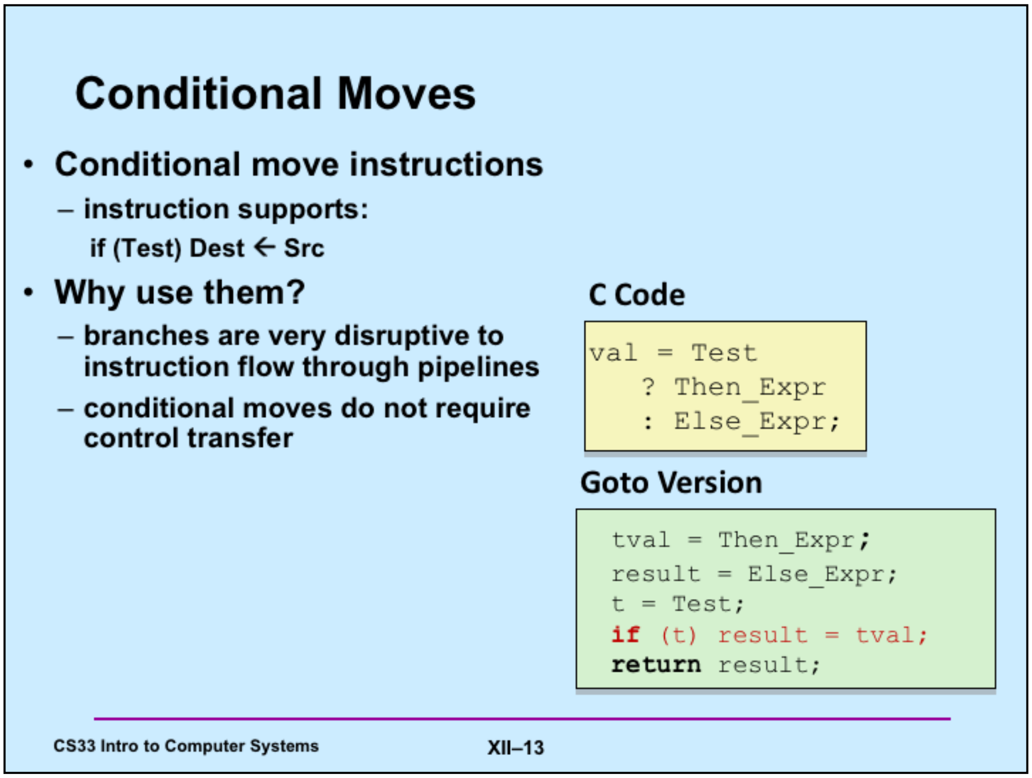
```
val = x>y ? x-y : y-x;
```

- Test is expression returning integer
  - == 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Goto Version

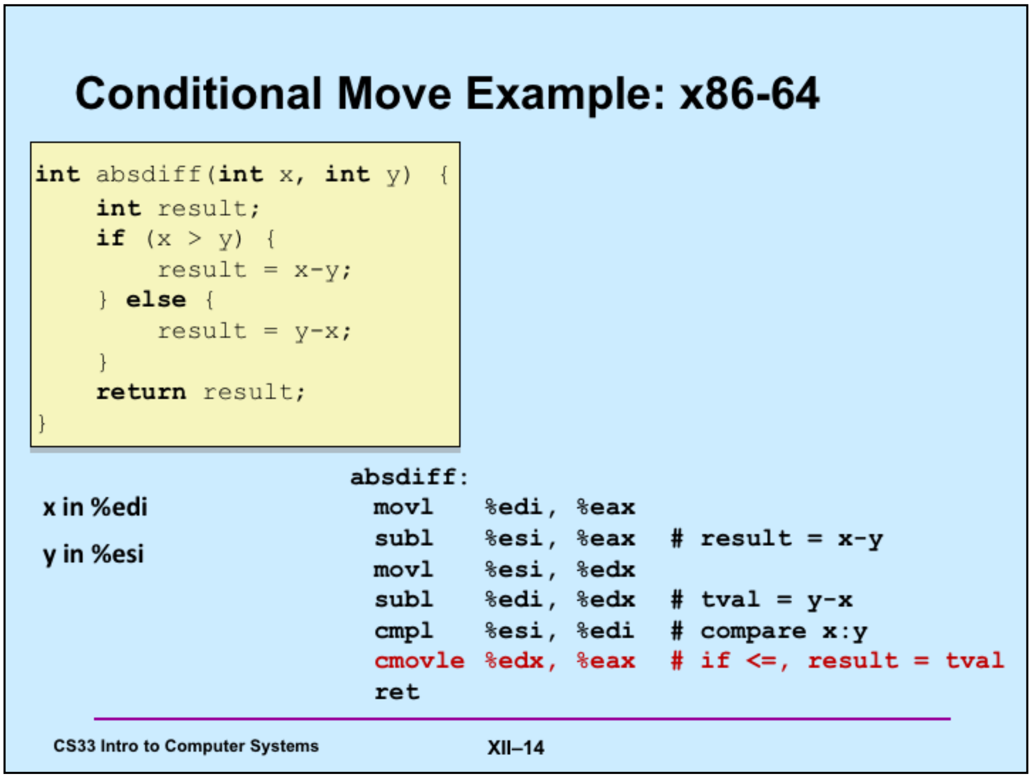
```
  nt = !Test;
  if (nt) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```


Supplied by CMU.



Supplied by CMU.

Note that, as shown in the goto version, both *then_expr* and *else_expr* are computed before the test is performed.



Supplied by CMU.

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

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:

%edi	x
%eax	result

```
    movl   $0, %eax      #   result = 0
.L2:       # loop:
    movl   %edi, %ecx
    andl   $1, %ecx      #   t = x & 1
    addl   %ecx, %eax    #   result += t
    shrl   %edi          #   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:**

```
{
  Statement_1;
  Statement_2;
    …
  Statement_n;
}
```

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

Supplied by CMU.

"While" Loop Example

C Code

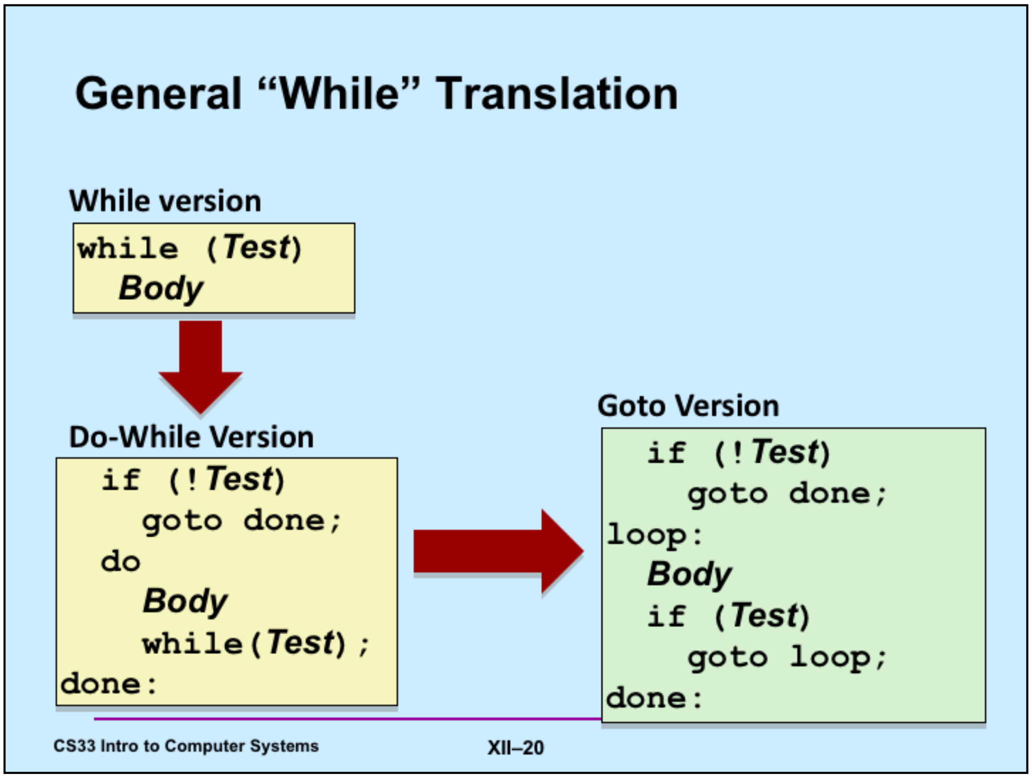
```
int pcount_while(unsigned x) {
  int result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

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
```

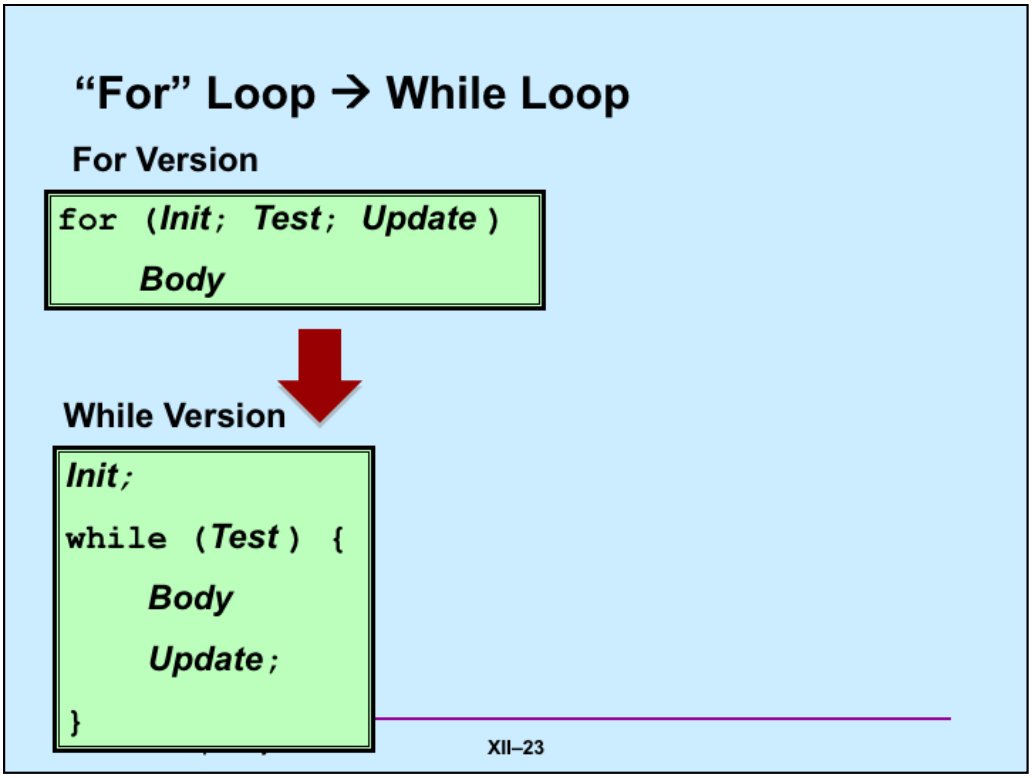
Update

```
i++
```

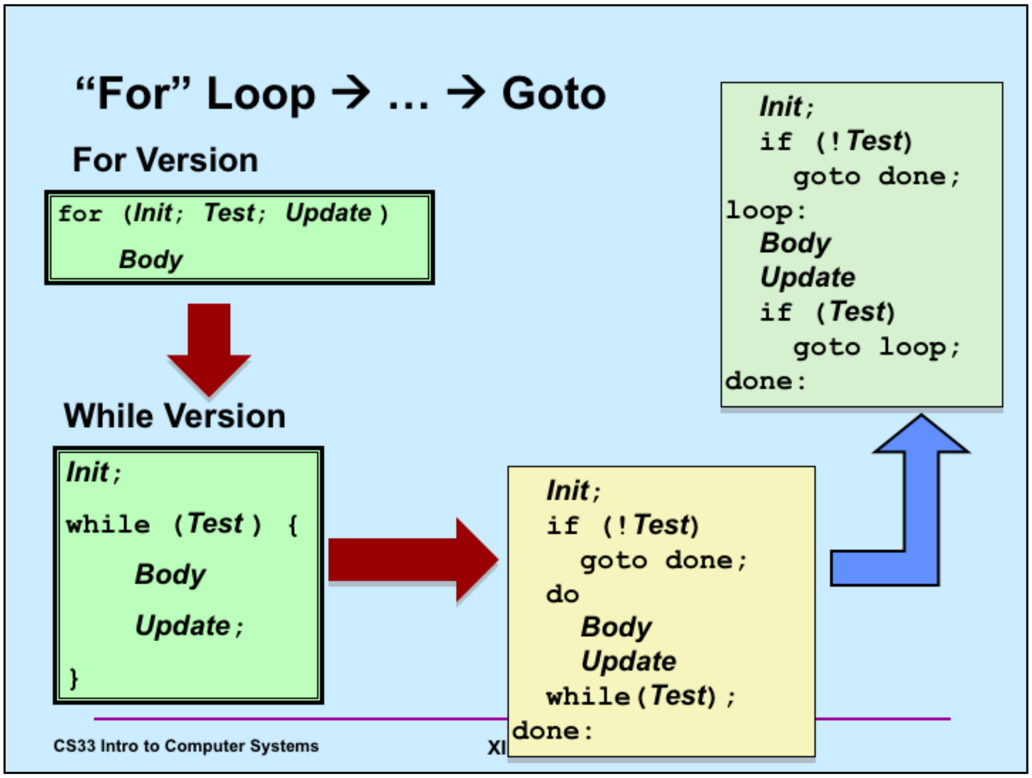
Body

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

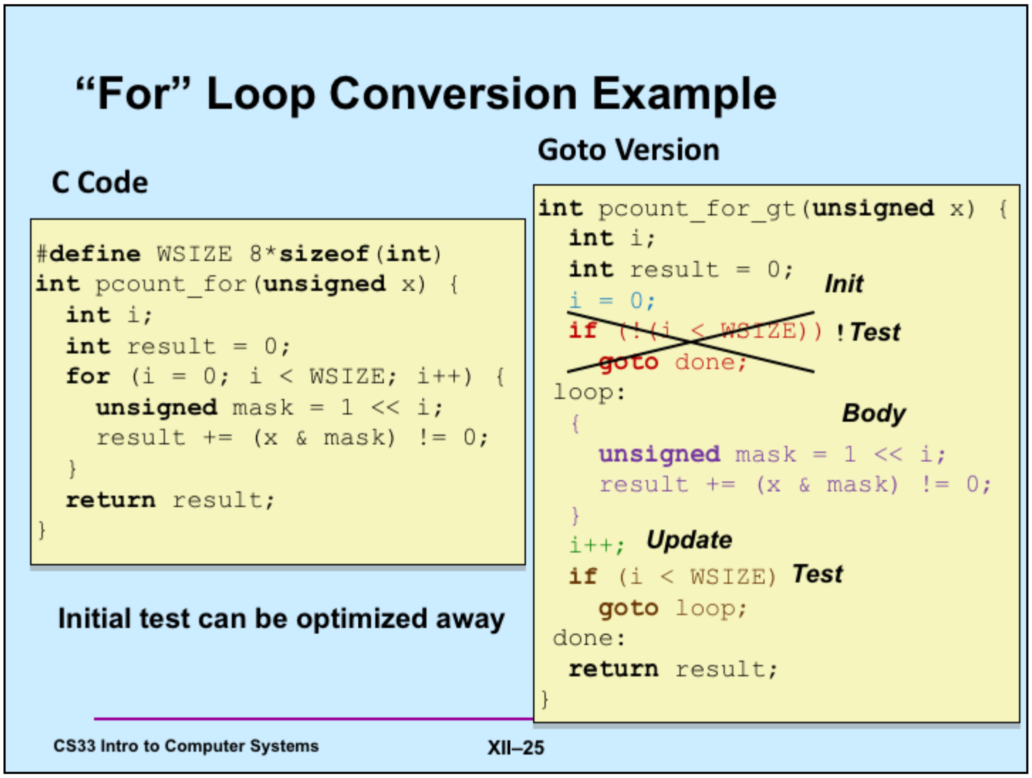

Supplied by CMU.



Supplied by CMU.



Supplied by CMU.



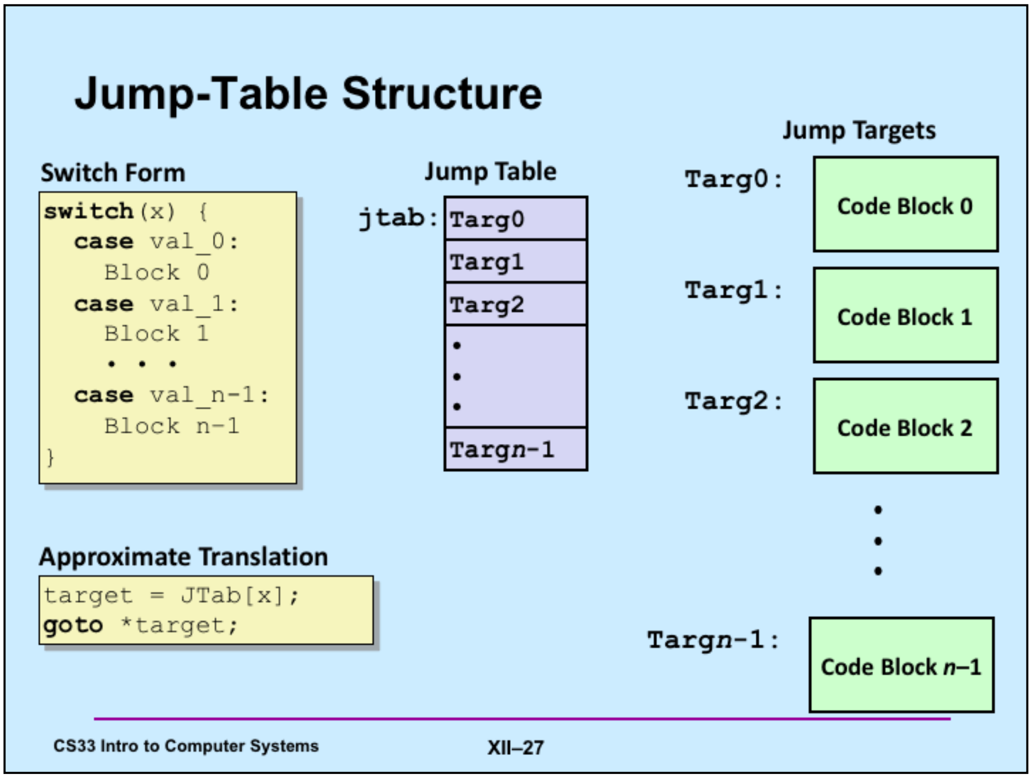
Supplied by CMU.

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;
}
```

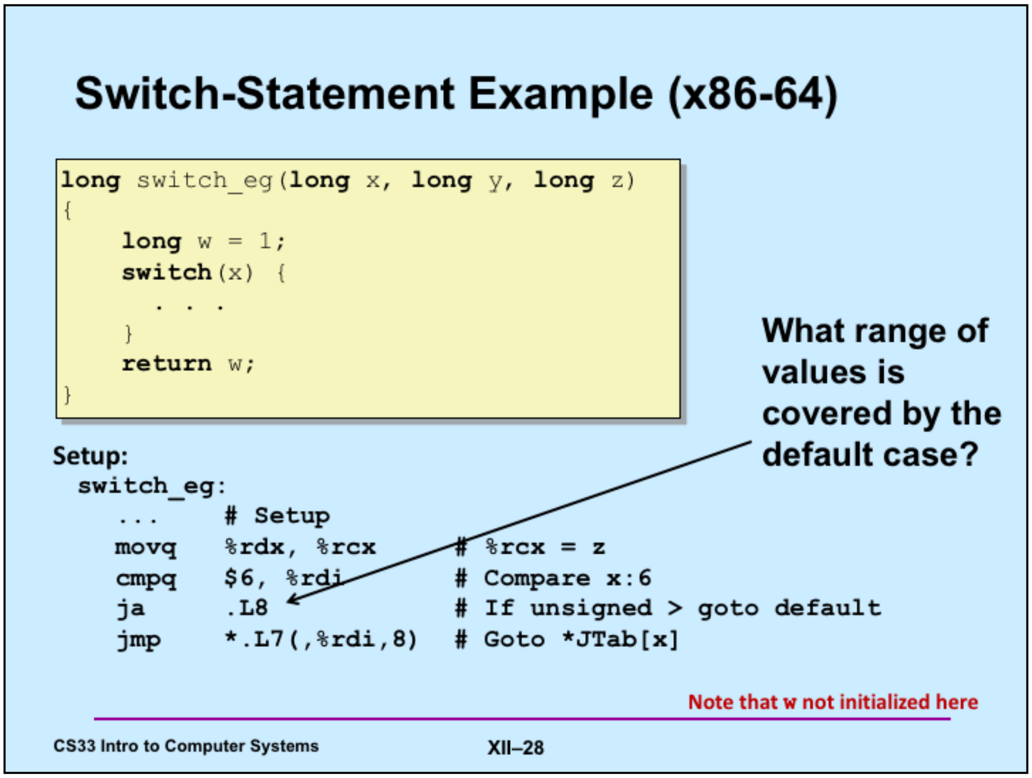
- **Multiple case labels**
  - here: 5 & 6
- **Fall-through cases**
  - here: 2
- **Missing cases**
  - here: 4

Supplied by CMU.



Supplied by CMU.

The translation is "approximate" because C doesn't have the notion of the target of a goto being a variable. But, if it did, then the translation is what we'd want!



Supplied by CMU, but converted to x86-64.

Note that the *ja* in the slide causes a jump to occur if the previous comparison is interpreted as being performed on unsigned values, and the result is that x is greater than (above) 6. Given that x is declared to be a *signed* value, for what range of values of x will *ja* cause a jump to take place?

Note that the assembler code shown in the examples was produced by compiling the C code using gcc with the "-O1" flag.

Switch-Statement Example

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

Jump table

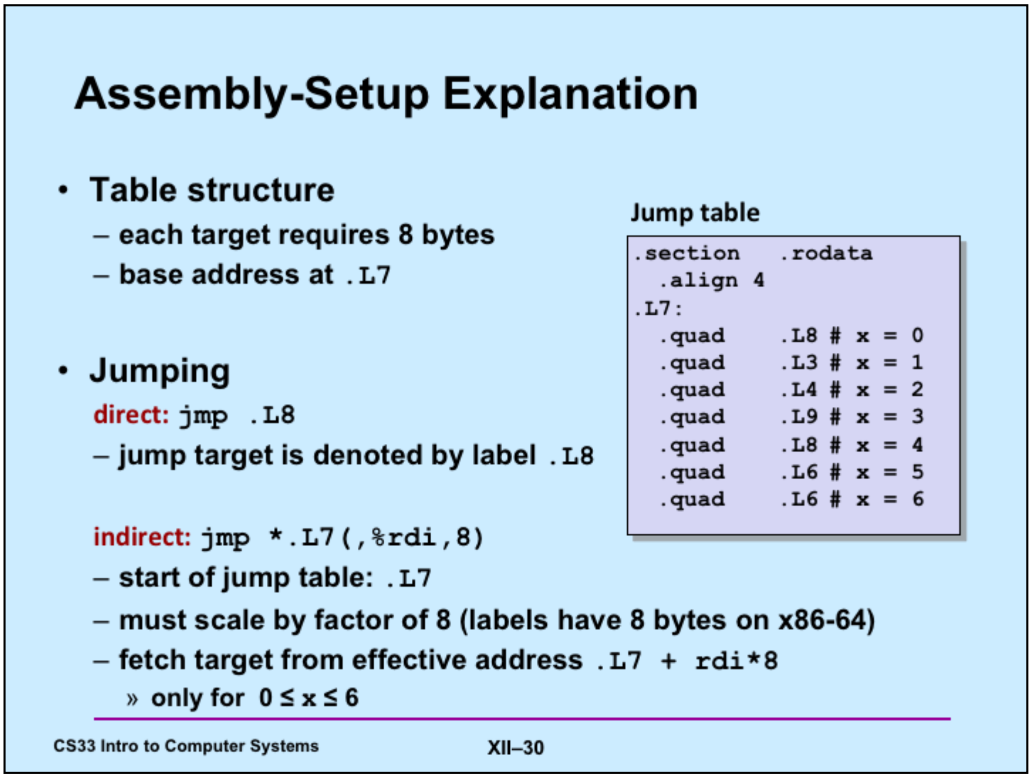
```
.section      .rodata
  .align 4
.L7:
  .quad       .L8  # x = 0
  .quad       .L3  # x = 1
  .quad       .L4  # x = 2
  .quad       .L9  # x = 3
  .quad       .L8  # x = 4
  .quad       .L6  # x = 5
  .quad       .L6  # x = 6
```

Setup:

```
switch_eg:
  ...      # Setup
  movq     %rdx, %rcx        # %rcx = z
  cmpq     $6, %rdi          # Compare x:6
  ja       .L8               # If unsigned > goto default
  jmp      *.L7(,%rdi,8)     # Goto *JTab[x]
```

Indirect jump

Supplied by CMU, but converted to x86-64.



Supplied by CMU, but converted to x86-64.

The *jmp* instruction is doing a couple things that require explanation: The asterisk means it's an *indirect jump* (such indirection is allowed only in jumps). The address specified after the asterisk is the address of an entry in the *jump table*. The asterisk means, rather than jumping directly to that entry, jump to the address that's in that table entry. ".L7" is a label that's being used as a displacement in the address computation. The value of .L7 is the address of the area of memory it labels. In this case, it's the address of the jump table. Thus, an unconditional jump is to take place to the address contained in the 8-byte entry of the jump table indexed by the contents of %rdi. Thus, if %rdi is, say, 2, then a jump will take place to address in the location starting 16 bytes beyond the beginning of the table. This will be a jump to .L4. .L4 itself is a label of code specified elsewhere, the reference to the label is replaced by the assembler with the address of the code labelled with .L4.

The jump table is separate from the code (it's not executable). This is specified by the ".section" directive, which also specifies that it should be placed in memory that's made read-only (".rodata" indicates this). The ".align 4") says that the address of the start of the table should be divisible by four (why this is important is something we'll get to in a week or two).

Jump Table

Jump table

```
.section   .rodata
  .align 4
.L7:
  .quad     .L8 # x = 0
  .quad     .L3 # x = 1
  .quad     .L4 # x = 2
  .quad     .L9 # x = 3
  .quad     .L8 # x = 4
  .quad     .L6 # x = 5
  .quad     .L6 # x = 6
```

```
switch(x) {
case 1:        // .L3
    w = y*z;
    break;
case 2:        // .L4
    w = y/z;
    /* Fall Through */
case 3:        // .L9
    w += z;
    break;
case 5:
case 6:        // .L6
    w -= z;
    break;
default:       // .L8
    w = 2;
}
```

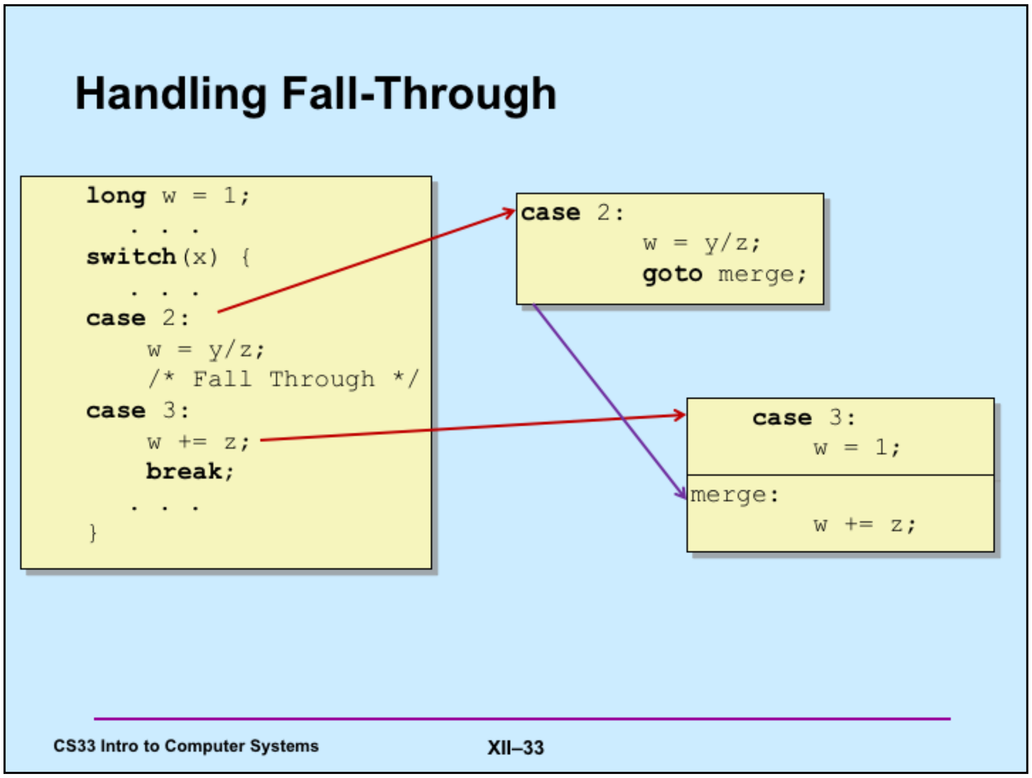

Supplied by CMU, but converted to x86-64.

Code Blocks (Partial)

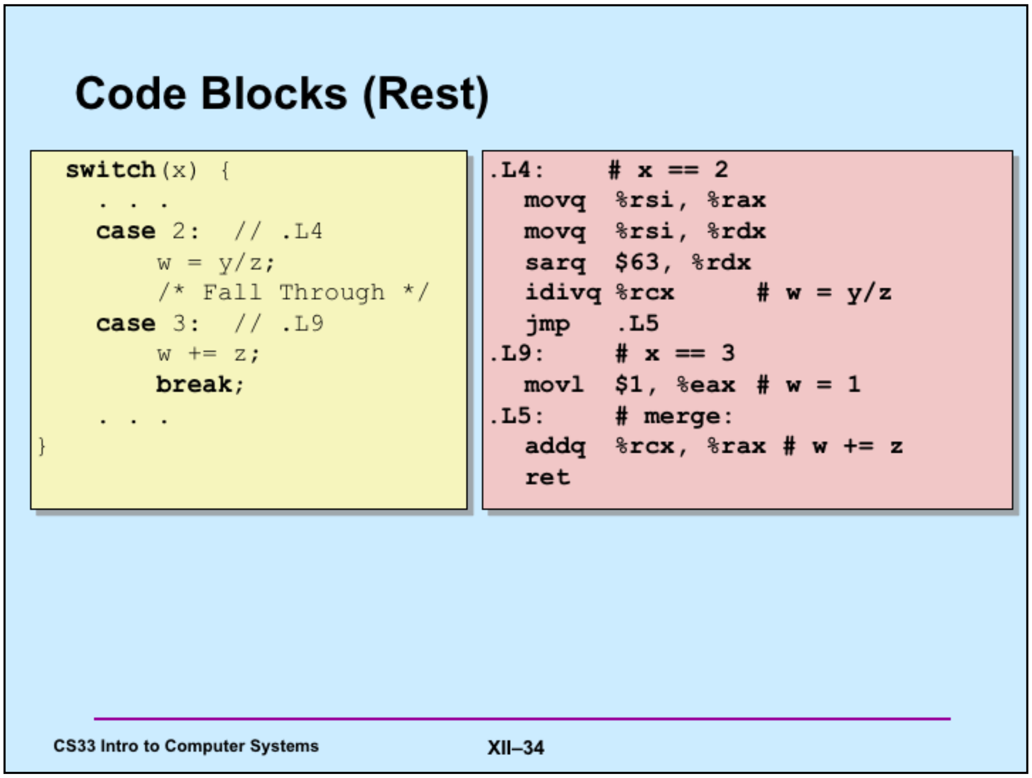
```
switch(x) {
case 1:        // .L3
      w = y*z;
      break;
 . . .
case 5:        // .L6
case 6:        // .L6
    w -= z;
    break;
default:       // .L8
    w = 2;
}
```

```
.L3:    # x == 1
  movl  %rsi, %rax   # y
  imulq %rdx, %rax   # w = y*z
  ret
.L6:              # x == 5, x == 6
  movl  $1, %eax # w = 1
  subq  %rdx, %rax   # w -= z
  ret
.L8:              # Default
  movl  $2, %eax # w = 2
  ret
```


Supplied by CMU, but converted to x86-64.



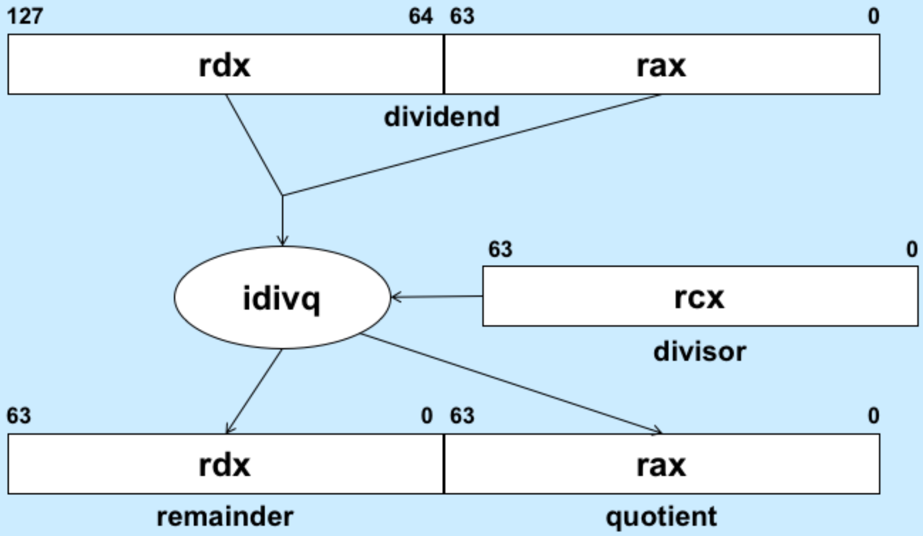
Supplied by CMU, but converted to x86-64.

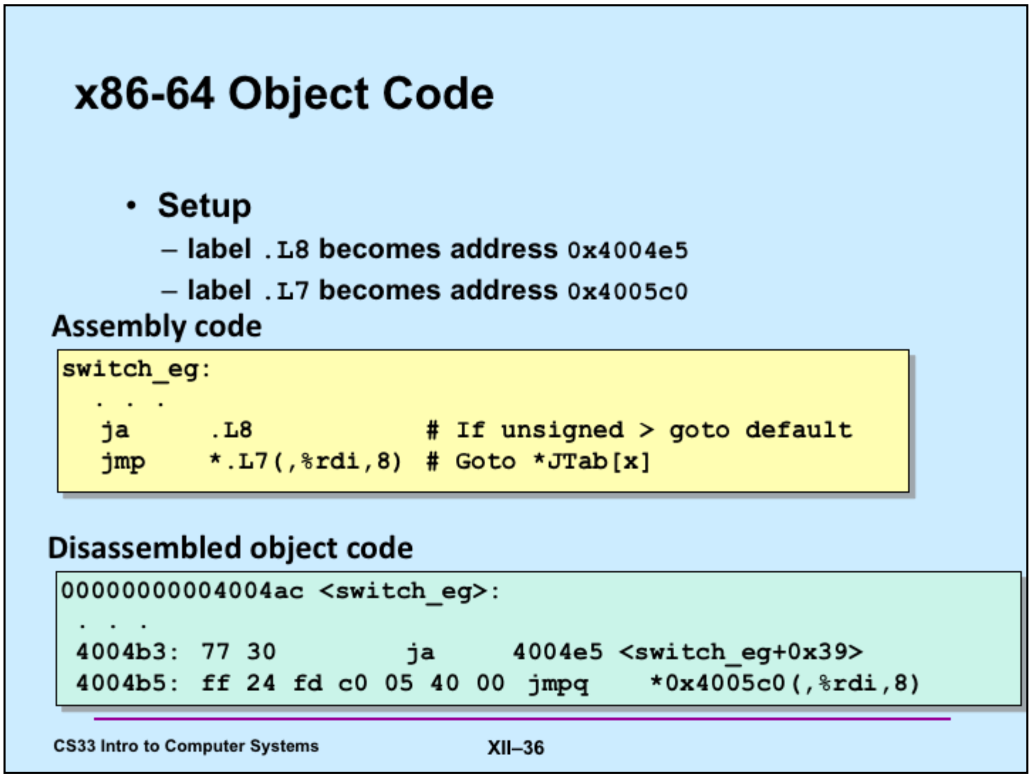


Supplied by CMU, but converted to x86-64.

The code following the .L4 label requires some explanation. The *idivq* instruction is special in that it takes a 128-bit dividend that is implicitly assumed to reside in registers *rdx* and *rax*. Its single operand specifies the divisor. The quotient is always placed in the *rax* register, and the remainder in the *rdx* register. In our example, *y*, which we want to be the dividend, is copied into both the *rax* and *rdx* registers. The *sarq* (shift arithmetic right quadword) instruction propagates the sign bit of *rdx* across the entire register, replacing its original contents. Thus, if one considers *rdx* to contain the most-significant bits of the dividend and *rax* to contain the least-significant bits, the pair of registers now contains the 128-bit version of *y*. The *idivq* instruction computes the quotient from dividing this 128-bit value by the 64-bit value contained in register *rcx* (containing *z*). The quotient is stored register *rax* (implicitly) and the remainder is stored in register *rdx* (and is ignored in our example). This illustrated in the next slide.

idivq


127
64
63
0
rdx
rax
dividend
idivq
63
0
rcx
divisor
63
0
63
0
rdx
rax
remainder
quotient



Supplied by CMU, but converted to x86-64.

Disassembly was accomplished using “objdump –d”. Note that the text enclosed in angle brackets (“<“, “>”) is essentially a comment, relating the address (4004e5) to a symbolic location (0x39 bytes after the beginning of *switch_eg*).

x86-64 Object Code (cont.)

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

```
gdb switch
(gdb) x/7xg 0x4005c0
```

  » e**x**amine **7** he**x**adecimal format "**g**iant" words (8-bytes each)

  » use command "`help x`" to get format documentation

```
0x4005c0:     0x00000000004004e5     0x00000000004004bc
0x4005d0:     0x00000000004004c4     0x00000000004004d3
0x4005e0:     0x00000000004004e5     0x00000000004004dc
0x4005f0:     0x00000000004004dc
```


Supplied by CMU, but converted to x86-64. We assume that the switch_eg function was included in a program whose name is *switch*. Hence, gdb is invoked from the shell with the argument "switch".

x86-64 Object Code (cont.)

- **Deciphering jump table**

```
0x4005c0:        0x00000000004004e5        0x00000000004004bc
0x4005d0:        0x00000000004004c4        0x00000000004004d3
0x4005e0:        0x00000000004004e5        0x00000000004004dc
0x4005f0:        0x00000000004004dc
```

Address	Value	x
0x4005c0	0x4004e5	0
0x4005c8	0x4004bc	1
0x4005d0	0x4004c4	2
0x4005d8	0x4004d3	3
0x4005e0	0x4004e5	4
0x4005e8	0x4004dc	5
0x4005f0	0x4004dc	6

Supplied by CMU, but converted to x86-64.

Disassembled Targets

```
(gdb) disassemble 0x4004bc,0x4004eb
 Dump of assembler code from 0x4004bc to 0x4004eb
   0x00000000004004bc <switch_eg+16>:   mov    %rsi,%rax
   0x00000000004004bf <switch_eg+19>:   imul   %rdx,%rax
   0x00000000004004c3 <switch_eg+23>:   retq
   0x00000000004004c4 <switch_eg+24>:   mov    %rsi,%rax
   0x00000000004004c7 <switch_eg+27>:   mov    %rsi,%rdx
   0x00000000004004ca <switch_eg+30>:   sar    $0x3f,%rdx
   0x00000000004004ce <switch_eg+34>:   idiv   %rcx
   0x00000000004004d1 <switch_eg+37>:   jmp    0x4004d8 <switch_eg+44>
   0x00000000004004d3 <switch_eg+39>:   mov    $0x1,%eax
   0x00000000004004d8 <switch_eg+44>:   add    %rcx,%rax
   0x00000000004004db <switch_eg+47>:   retq
   0x00000000004004dc <switch_eg+48>:   mov    $0x1,%eax
   0x00000000004004e1 <switch_eg+53>:   sub    %rdx,%rax
   0x00000000004004e4 <switch_eg+56>:   retq
   0x00000000004004e5 <switch_eg+57>:   mov    $0x2,%eax
   0x00000000004004ea <switch_eg+62>:   retq
```

Matching Disassembled Targets

Value	x
0x4004e5	0
0x4004bc	1
0x4004c4	2
0x4004d3	3
0x4004e5	4
0x4004dc	5
0x4004dc	6

```
0x00000000004004bc:    mov     %rsi,%rax
0x00000000004004bf:    imul    %rdx,%rax
0x00000000004004c3:    retq
0x00000000004004c4:    mov     %rsi,%rax
0x00000000004004c7:    mov     %rsi,%rdx
0x00000000004004ca:    sar     $0x3f,%rdx
0x00000000004004ce:    idiv    %rcx
0x00000000004004d1:    jmp     0x4004d8
0x00000000004004d3:    mov     $0x1,%eax
0x00000000004004d8:    add     %rcx,%rax
0x00000000004004db:    retq
0x00000000004004dc:    mov     $0x1,%eax
0x00000000004004e1:    sub     %rdx,%rax
0x00000000004004e4:    retq
0x00000000004004e5:    mov     $0x2,%eax
0x00000000004004ea:    retq
```

Quiz 2

What C code would you compile to get the following assembler code?

```
        movl    $0, %eax
.L2:
        movl    %eax, a(,%rax,4)
        addq    $1, %rax
        cmpq    $10, %rax
        jne     .L2
        ret
```

```
int a[10];
void func() {
  int i;
  for (i=0; i<10; i++)
    a[i]= 1;
}
```

a

```
int a[10];
void func() {
  int i=0;
  while (i<10)
    a[i]= i++;
}
```

b

```
int a[10];
void func() {
  int i=0;
  switch (i) {
case 0:
    a[i] = 0;
    break;
default:
    a[i] = 10
  }
}
```

c