CS 33

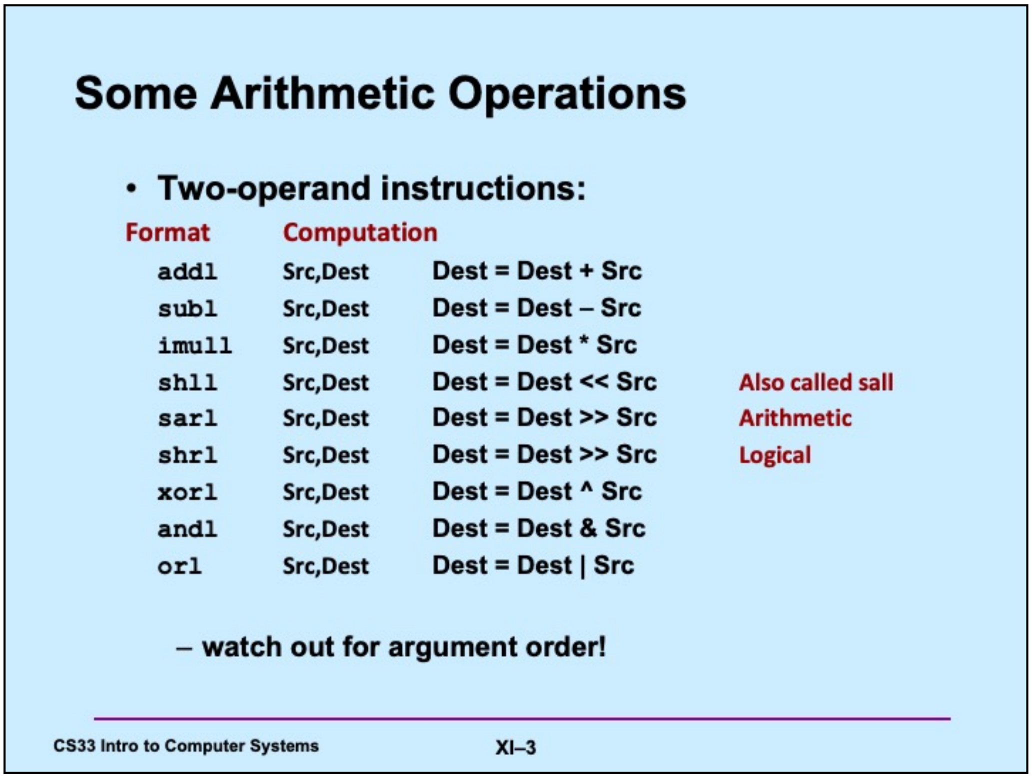
Machine Programming (3)

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.

How Many Instructions are There?

- We cover ~30
- Implemented by Intel:
  - 80 in original 8086 architecture
  - 7 added with 80186
  - 17 added with 80286
  - 33 added with 386
  - 6 added with 486
  - 6 added with Pentium
  - 1 added with Pentium MMX
  - 4 added with Pentium Pro
  - 8 added with SSE
  - 8 added with SSE2
  - 2 added with SSE3
  - 14 added with x86-64
  - 10 added with VT-x
  - 2 added with SSE4a
- Total: 198
- Doesn't count:
  - floating-point instructions
    - » ~100
  - SIMD instructions
    - » lots
  - AMD-added instructions
  - undocumented instructions

The source for this is http://en.wikipedia.org/wiki/X86_instruction_listings, viewed on 6/20/2017, which came with the caveat that it may be out of date. While it's likely that more instructions have been added since then, we won't be covering them in 33!



Supplied by CMU.

Note that for shift instructions, the Src operand (which is the size of the shift) must either be an immediate operand or be a designator for a one-byte register (e.g., %cl – see the slide on general-purpose registers for IA32).

Also note that what's given in the slide are the versions for 32-bit operands. There are also versions for 8-, 16-, and 64-bit operands, with the "l" replaced with the appropriate letter ("b", "s", or "q").

Some Arithmetic Operations

- **One-operand Instructions**

`incl`	Dest	= Dest + 1
`decl`	Dest	= Dest – 1
`negl`	Dest	= – Dest
`notl`	Dest	= ~Dest

- **See textbook for more instructions**
- **See Intel documentation for even more**

Adapted from a slide supplied by CMU.

Arithmetic Expression Example

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
arith:
    leal   (%rdi,%rsi), %eax
    addl   %edx, %eax
    leal   (%rsi,%rsi,2), %edx
    shll   $4, %edx
    leal   4(%rdi,%rdx), %ecx
    imull  %ecx, %eax
    ret
```


Supplied by CMU, but converted to x86-64.

Understanding `arith`

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

%rdx	z
%rsi	y
%rdi	x

```
leal   (%rdi,%rsi), %eax
addl   %edx, %eax
leal   (%rsi,%rsi,2), %edx
shll   $4, %edx
leal   4(%rdi,%rdx), %ecx
imull  %ecx, %eax
ret
```


Supplied by CMU, but converted to x86-64.

Understanding `arith`

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

%rdx	z
%rsi	y
%rdi	x

```
leal   (%rdi,%rsi), %eax     # eax = x+y      (t1)
addl   %edx, %eax            # eax = t1+z     (t2)
leal   (%rsi,%rsi,2), %edx   # edx = 3*y      (t4)
shll   $4, %edx              # edx = t4*16    (t4)
leal   4(%rdi,%rdx), %ecx    # ecx = x+4+t4   (t5)
imull  %ecx, %eax            # eax *= t5      (rval)
ret
```


Supplied by CMU, but converted to x86-64.

By convention, the first three arguments to a function are placed in registers **rdi**, **rsi**, and **rdx**, respectively. Note that, also by convention, functions put their return values in register **eax**/**rax**.

Observations about `arith`

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

- Instructions in different order from C code
- Some expressions might require multiple instructions
- Some instructions might cover multiple expressions

```
leal   (%rdi,%rsi), %eax     # eax = x+y     (t1)
addl   %edx, %eax            # eax = t1+z    (t2)
leal   (%rsi,%rsi,2), %edx   # edx = 3*y     (t4)
shll   $4, %edx              # edx = t4*16   (t4)
leal   4(%rdi,%rdx), %ecx    # ecx = x+4+t4  (t5)
imull  %ecx, %eax            # eax *= t5     (rval)
ret
```


Supplied by CMU, but converted to x86-64.

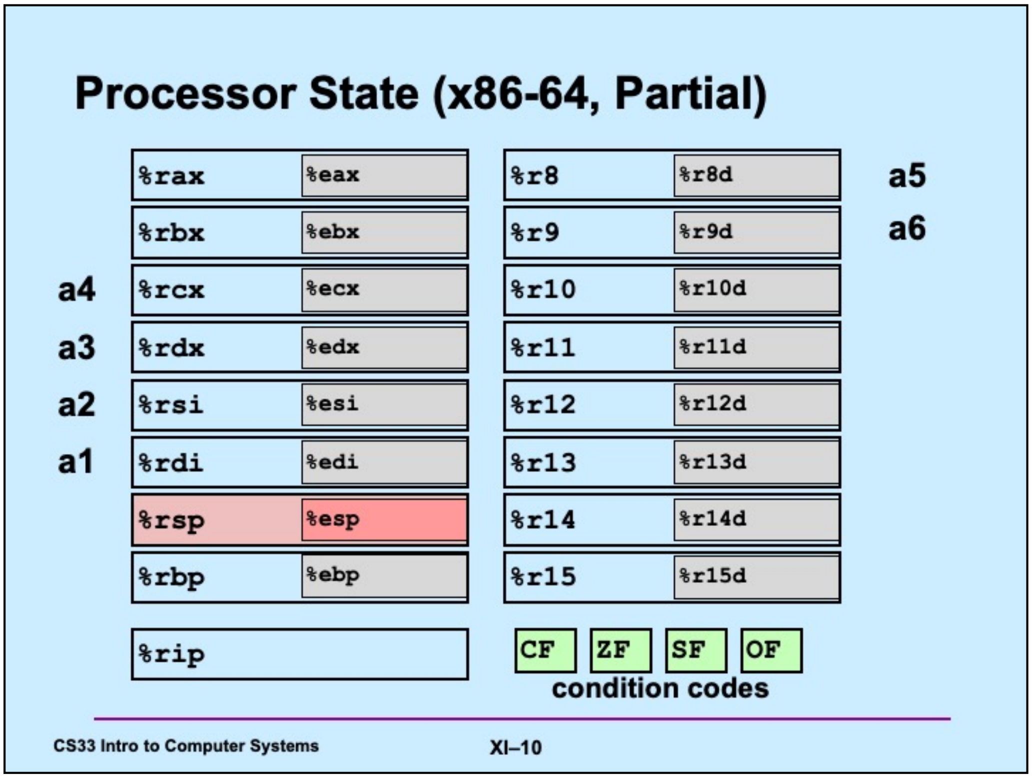
Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192$, $2^{13} - 7 = 8185$

```
xorl %esi, %edi        # edi = x^y          (t1)
sarl $17, %edi         # edi = t1>>17       (t2)
movl %edi, %eax        # eax = edi
andl $8185, %eax       # eax = t2 & mask (rval)
```


Supplied by CMU, but converted to x86-64.



%rip is the instruction-pointer register. It contains the address of the next instruction to be executed. CF, ZF, SF, and OF are the condition codes, referring to carry flag, zero flag, sign flag, and overflow flag.

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)

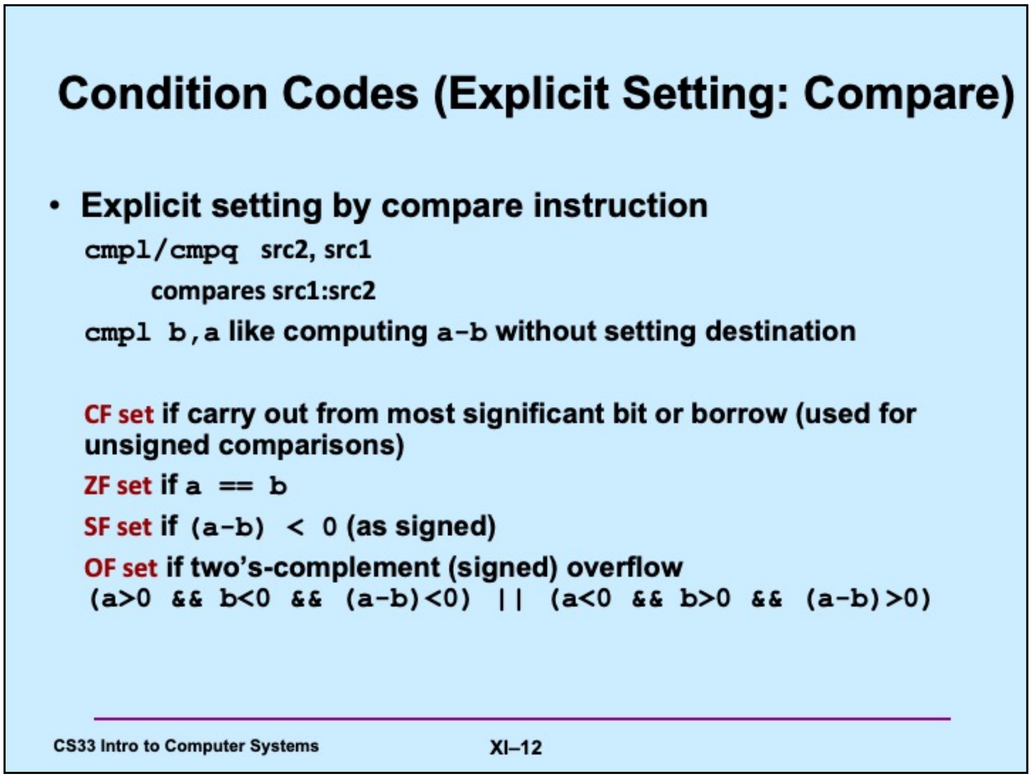
  ZF set if `t == 0`

  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.



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.

Note that if a&b<0, what is meant is that the most-significant bit is 1.

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)

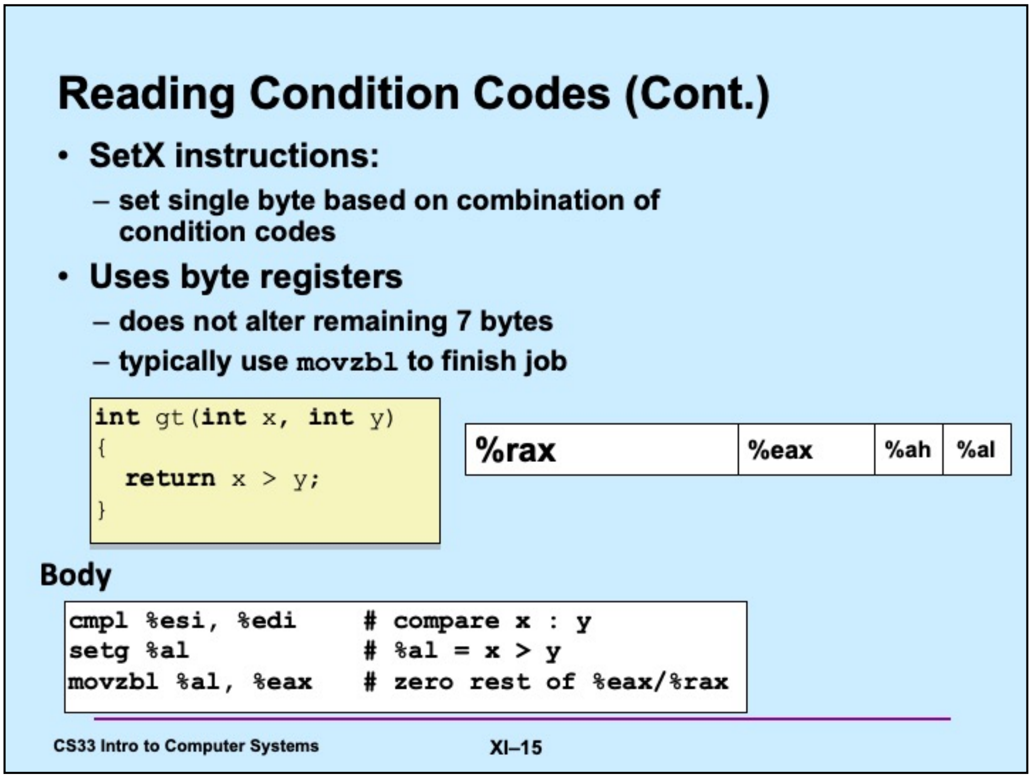
Supplied by CMU.

These operations allow one to set a byte depending on the values of the condition codes.

Some of these conditions aren't all that obvious. Suppose we are comparing A with B (cmpl B,A). Thus the condition codes would be set as if we computed A-B. For signed arithmetic, If A >= B, then the true result is non-negative. But we have to deal with two's complement arithmetic with a finite word size. If overflow does not occur, then the sign flag should not be set. If overflow does occur, then even though the true result should have been positive, the actual result is negative. So, if both the sign flag and the overflow flag are not set, we know that A >= B. If both flags are set, we know the true result of the subtraction is positive and thus A>=B. But if one of the two flags is set and the other isn't, then A must be less than B. Thus if ~(SF^OF) is 1, we know that A>=B. If ZF (zero flag) is set, we know that A==B. Thus for A>B, ZF is not set.

For unsigned arithmetic, if A>B, then subtracting B from A doesn't require a borrow and thus CF is not set; and since A is not equal to B, ZF is not set. If A<B, then subtracting B from A requires a borrow and thus CF is set.

The other cases can be worked out similarly.



Supplied by CMU, but converted to x86-64.

Recall that the first argument to a function is passed in %rdi (%edi) and the second in %rsi (%esi).

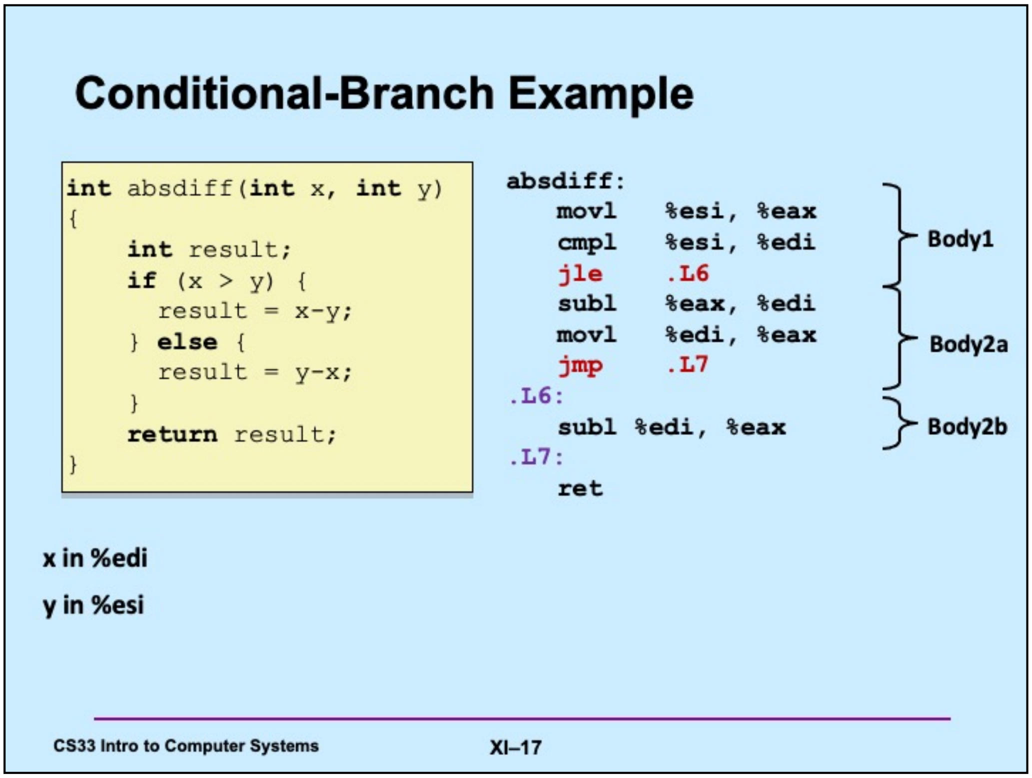
Jumping

- **jX instructions**
  - **Jump to different part of program 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.

See the notes for slide 14.



Supplied by CMU, but converted to x86-64.

The function computes the absolute value of the difference of its two arguments.

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;
}
```

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

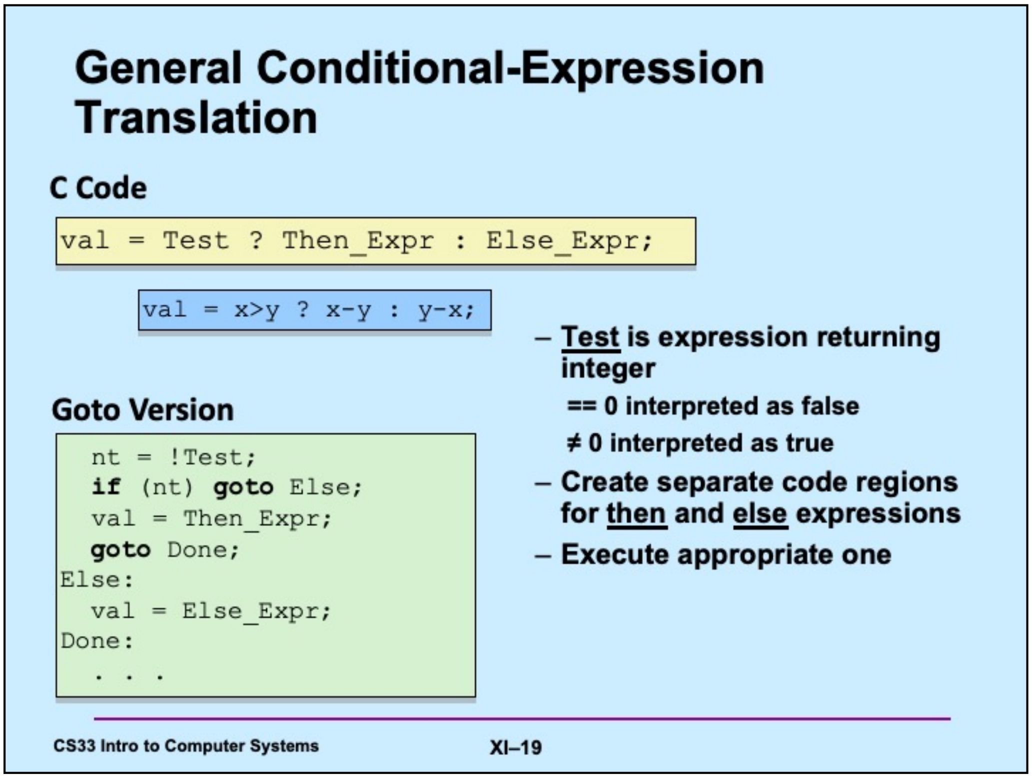
```
absdiff:
    movl   %esi, %eax
    cmpl   %esi, %edi
    jle    .L6
    subl   %eax, %edi
    movl   %edi, %eax
    jmp .L7
.L6:
    subl %edi, %eax
.L7:
    ret
```

Body1

Body2a

Body2b

Supplied by CMU, but converted to x86-64.



Supplied by CMU.

C's conditional expression, as shown in the slide, is sometimes useful, but often results in really difficult-to-read code.

"Do-While" Loop Example

C Code

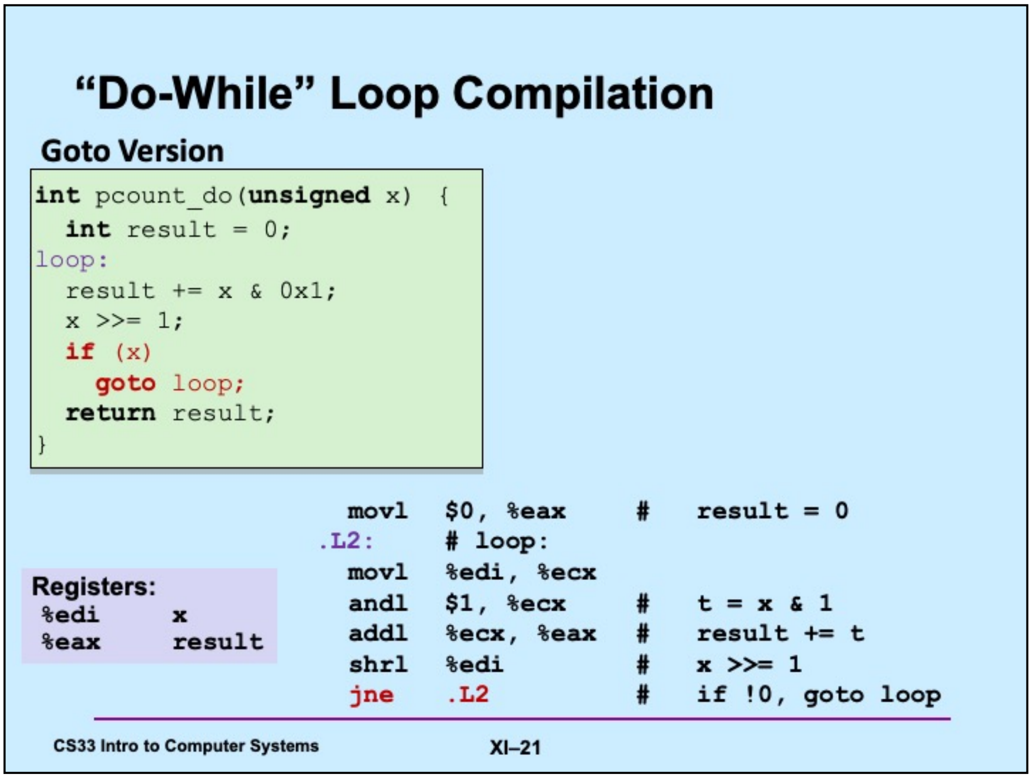
```
int pcount_do(unsigned x)
{
  int result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

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.



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

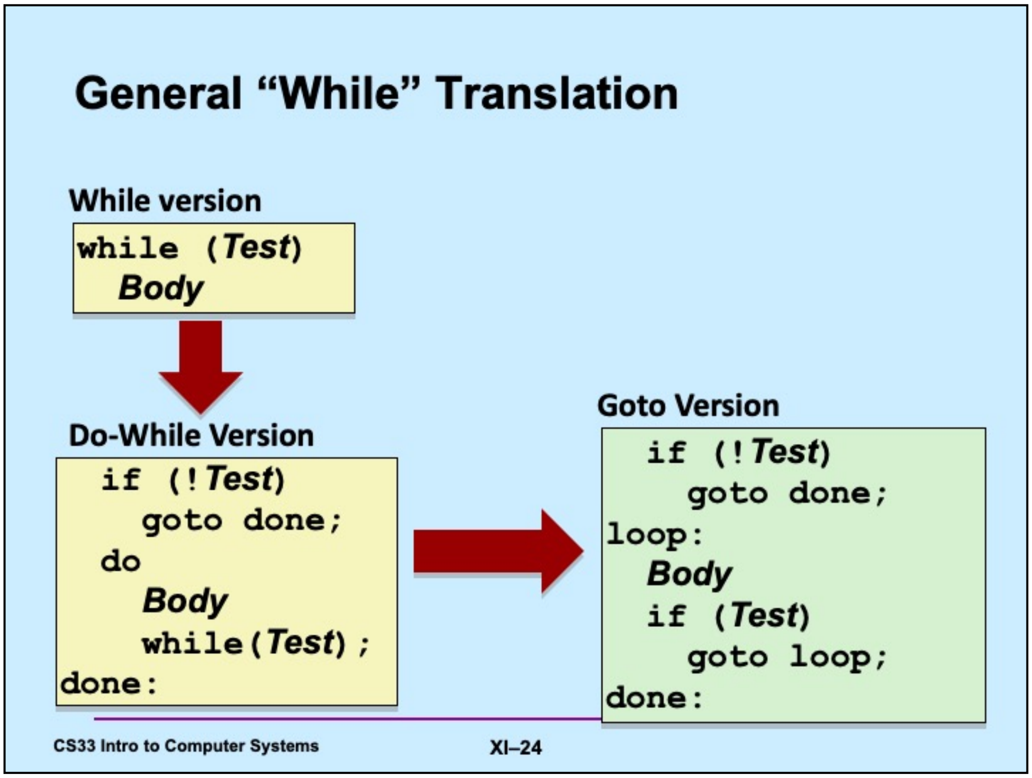
```
int pcount_while(unsigned x) {
  int result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

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
```

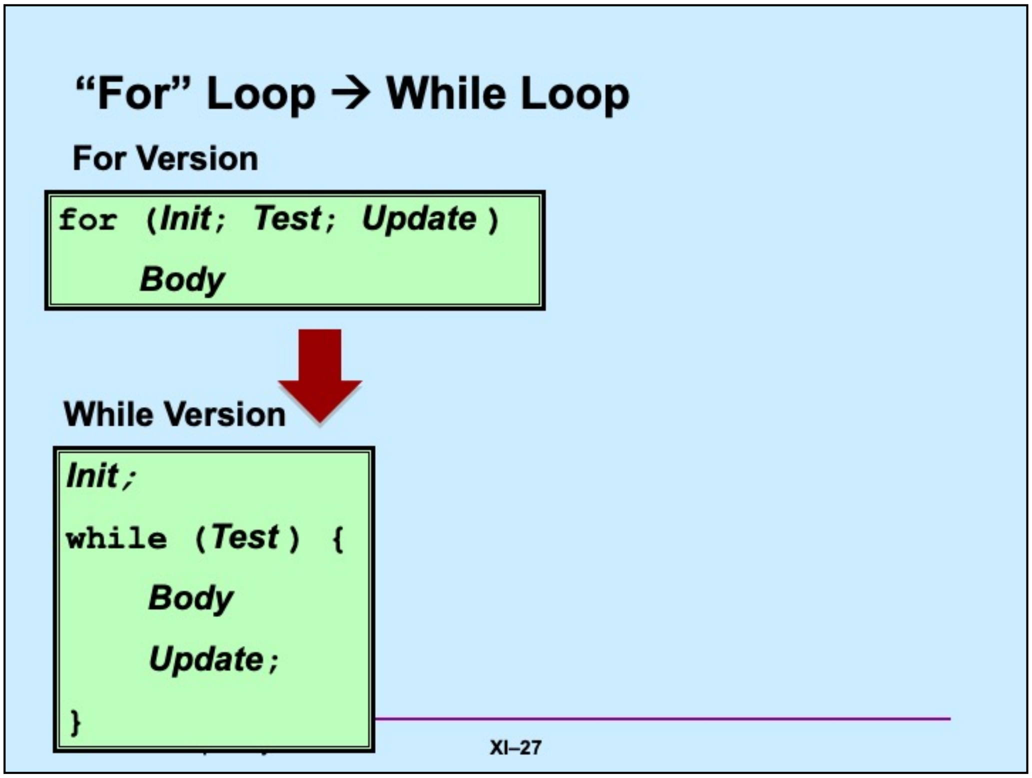
Update

```
i++
```

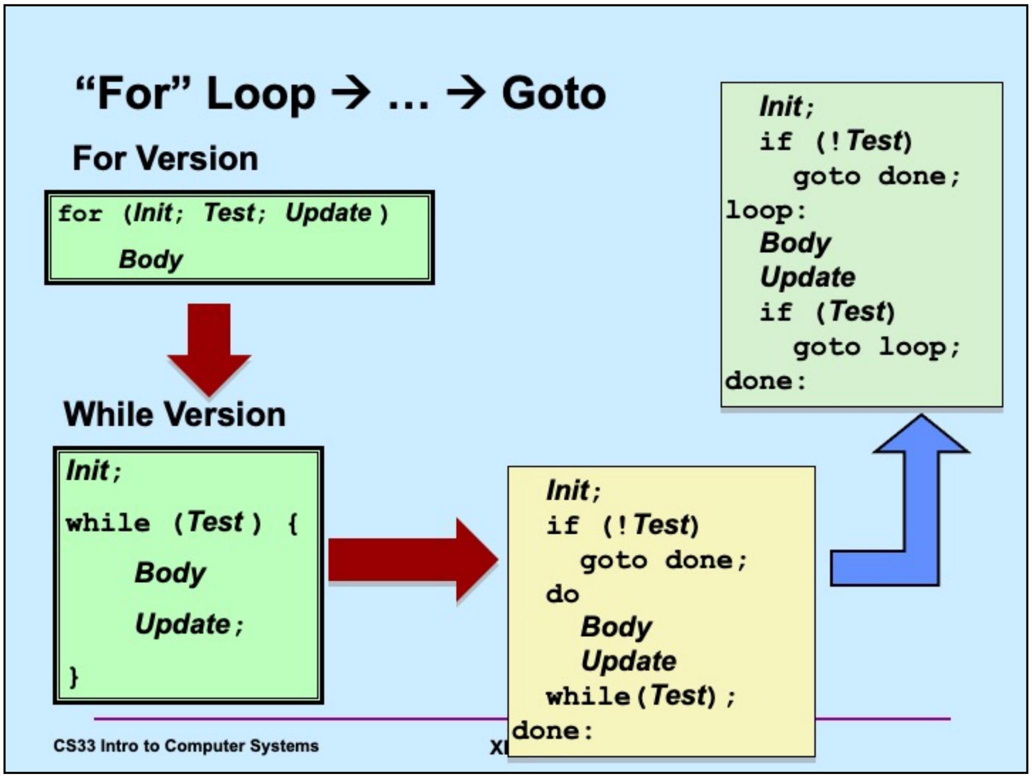
Body

```
{
  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;
}
```

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;
}
```

Initial test can be optimized away

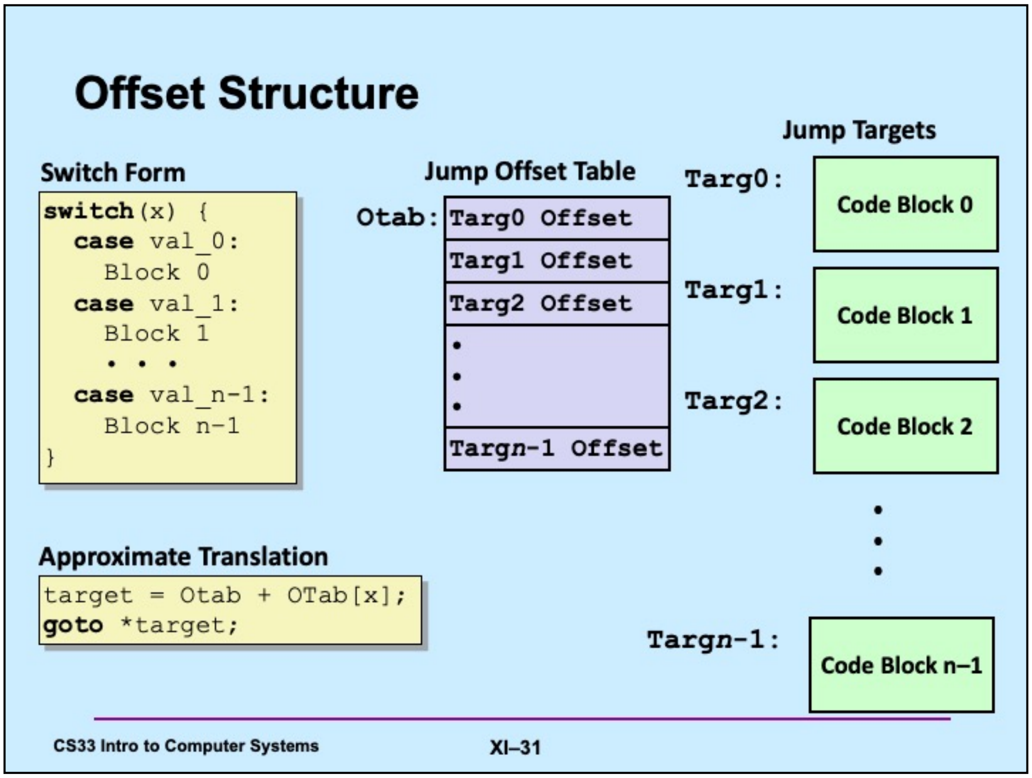
Supplied by CMU.

Switch-Statement Example

```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;
}
```

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

Adapted from slide supplied by CMU.



Adapted from slide supplied by CMU to account for changes in gcc.

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!

Otab (for "offset table") is a table of relative address of the jump targets. The idea is, given a value of x, **Otab[x]** contains a reference to the code block that should be handled for that case in the switch statement (this code block is known as the **jump target**). These references are offsets from the address **Otab**. In other words, **Otab** is an address, if we add to it the offset of a particular jump target, we get the absolute address of that jump target.

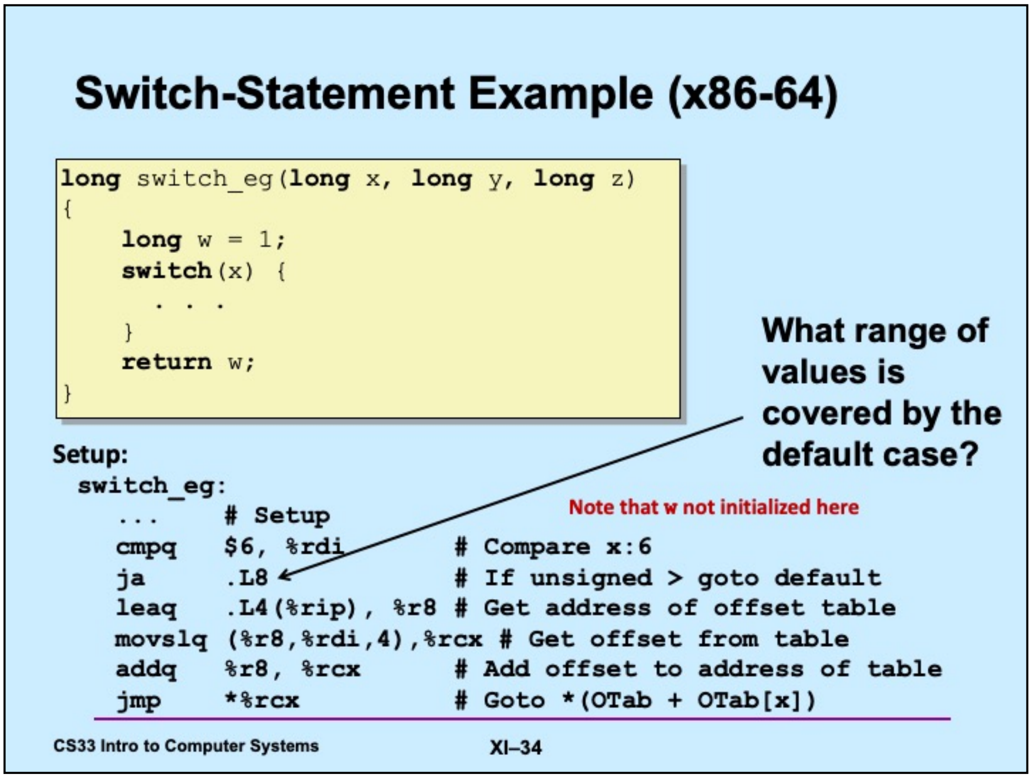
Assembler Code (1)

```
switch_eg:                                   .section        .rodata
        cmpq    $6, %rdi                     .align 4
        ja      .L8                     .L4:
        leaq    .L4(%rip), %r8               .long   .L8-.L4
        movslq  (%r8,%rdi,4), %rcx           .long   .L7-.L4
        addq    %r8, %rcx                    .long   .L6-.L4
        jmp     *%rcx                        .long   .L9-.L4
                                             .long   .L8-.L4
                                             .long   .L3-.L4
                                             .long   .L3-.L4
                                             .text
                                        .L7:
                                             movq    %rsi, %rax
                                             imulq   %rdx, %rax
                                             ret
```


Here's the assembler code obtained by compiling our C code in gcc with the –O1 optimization flag (specifying that some, but not lots of optimization should be done). We explain this code in subsequent slides. The jump offset table starts at label .L4.

Assembler Code (2)

```
.L6:
        leaq    (%rsi,%rdx), %rax
        jmp     .L5
.L9:
        movl    $1, %eax
.L5:
        addq    %rdx, %rax
        ret
.L3:
        movl    $1, %eax
        subq    %rdx, %rax
        ret
.L8:
        movl    $2, %eax
        ret
```



Much modified from a slide supplied by CMU.

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? The answer is that the jump will take place if x, interpreted as a **signed** value, is greater than 6 or less than zero.

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 Offset Table

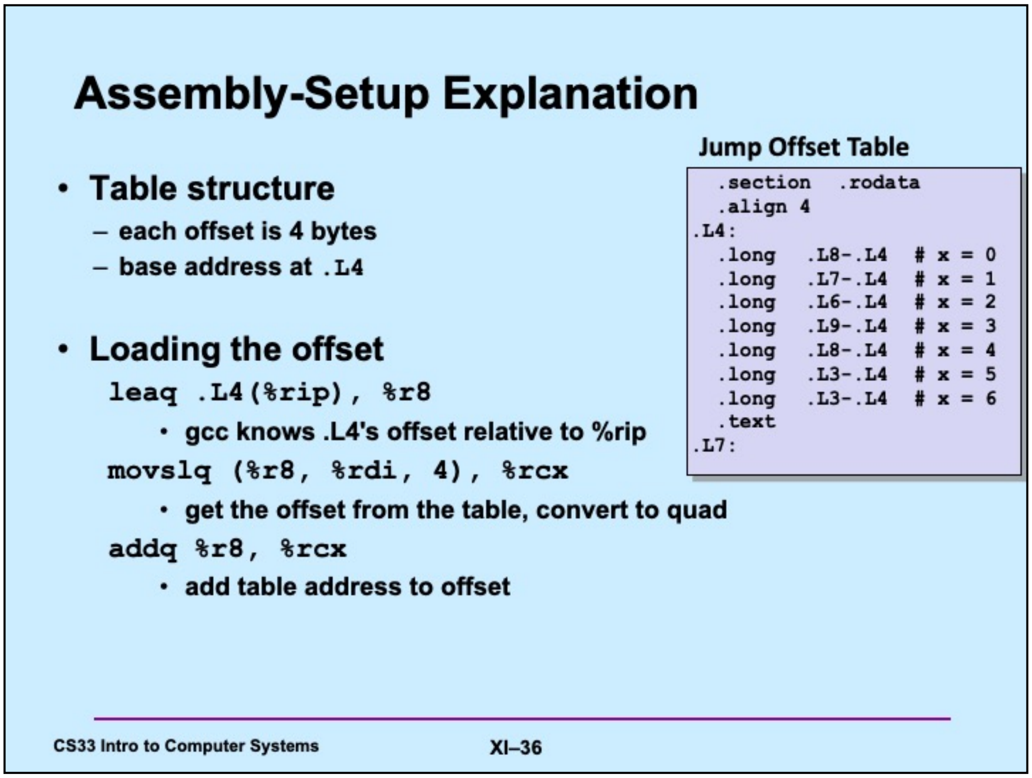
```
  .section  .rodata
  .align 4
.L4:
  .long   .L8-.L4   # x = 0
  .long   .L7-.L4   # x = 1
  .long   .L6-.L4   # x = 2
  .long   .L9-.L4   # x = 3
  .long   .L8-.L4   # x = 4
  .long   .L3-.L4   # x = 5
  .long   .L3-.L4   # x = 6
  .text
.L7:
```

Setup:

```
switch_eg:
   ...     # Setup
   cmpq   $6, %rdi       # Compare x:6
   ja     .L8            # If unsigned > goto default
   leaq   .L4(%rip), %r8 # Get address of offset table
   movslq (%r8,%rdi,4),%rcx # Get offset from table
   addq   %r8, %rcx      # Add offset to address of table
   jmp    *%rcx          # Goto *(OTab + OTab[x])
```

Indirect jump

Much modified from a slide supplied by CMU.



Much modified from a slide supplied by CMU.

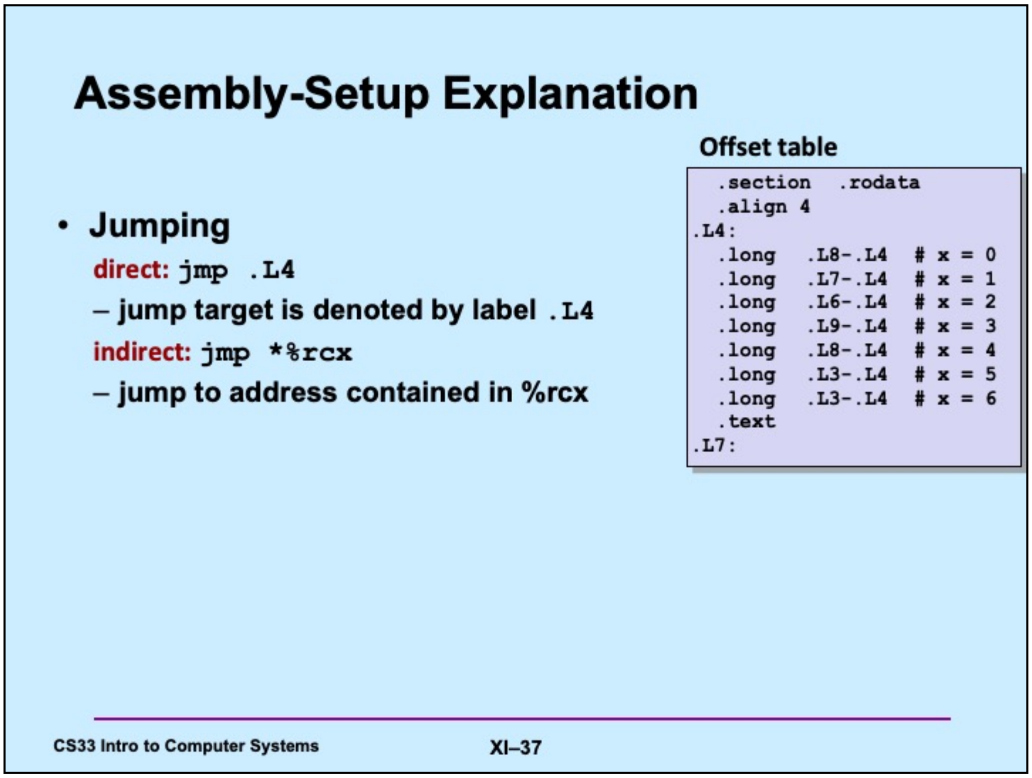
The jump offset table is different from the code in that it's not executable but is pure data (i.e. no instructions). This is specified by the ".section" directive, which also specifies that it should be placed in memory that's read-only and not executable (".rodata" indicates this). Thus, even though the assembler code for the table is listed inline with the assembler code for the executable part of the program, the table will be stored in a separate part of memory (whose address is referenced by ".L4"). The ".text" directive says that what follows is executable code (again).

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 a bit later).

The instruction pointer (%rip) is used as the base register since the compiler, with help from the linker, can figure out where .L4 is relative to the current address that's in %rip. Thus, the leaq instruction puts the address of .L4 into %r8, regardless of where the code was actually loaded into memory.

Now that we have the address of the offset table (in %r8), we can compute the address of a particular position in the table. To save space, the table consists of longs rather than quads. Each entry is the offset (in bytes) between some label (such as .L8) and the start of the table. Thus, for example, if the table starts at address 0x1000 and the address associated with .L8 is 0x1200, then what's in the first (index 0) entry of the table is 0x200. The movslq instruction (move signed long to quad), copies the 4-byte item specified by the source argument into the destination, sign-extending it to a quad (8 bytes).

Finally, the addq instruction adds the address of the beginning of the table to the value found in the referenced table entry. Thus, in our example of the offset in index 0 of the table, what's finally put in %rcx is 0x1000+0x200, or 0x1200, which is the address referenced by .L8.



Much modified from a slide supplied by CMU.

The "*" in the operand specification is allowed only in jmp instructions and indicates that the actual address of the target of the jump is not in the instruction, but, in this case, is in the indicated register. Thus, continuing with the example of the previous slide, if %rcx contains 0x1200, then a jump is made to location 0x1200.

Offset Table

Offset table

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

```
switch(x) {
case 1:        // .L7
    w = y*z;
    break;
case 2:        // .L6
    w = y+z;
    /* Fall Through */
case 3:        // .L9
    w += z;
    break;
case 5:
case 6:        // .L3
    w -= z;
    break;
default:       // .L8
    w = 2;
}
```

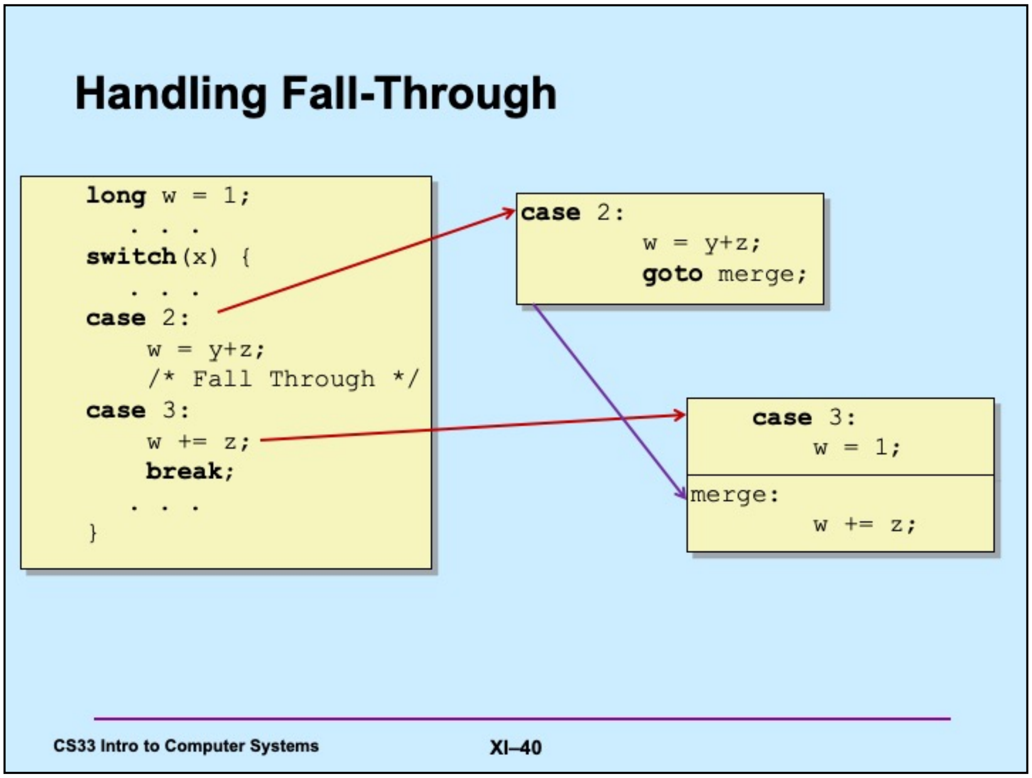

Much modified from a slide supplied by CMU.

Code Blocks (Partial)

```
switch(x) {
case 1:       // .L7
      w = y*z;
      break;
 . . .
case 5:       // .L3
case 6:       // .L3
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```

```
.L7:                # x == 1
  movq  %rsi, %rax   # y
  imulq %rdx, %rax   # w = y*z
  ret
.L3:                # x == 5, x == 6
  movl  $1, %eax     # w = 1
  subq  %rdx, %rax   # w -= z
  ret
.L8:                # Default
  movl  $2, %eax     # w = 2
  ret
```


Much modified from a slide supplied by CMU.



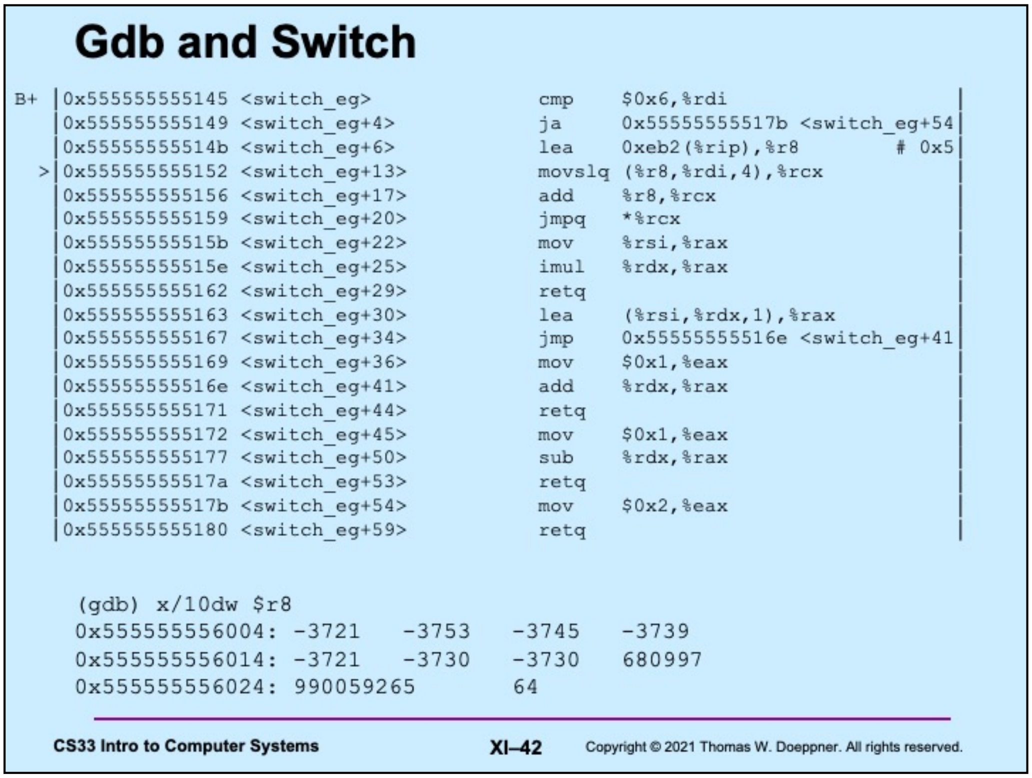
Adapted from a slide supplied by CMU.

Code Blocks (Rest)

```
  switch(x) {
    . . .
    case 2:  // .L6
        w = y+z;
        /* Fall Through */
    case 3:  // .L9
        w += z;
        break;
    . . .
}
```

```
.L6:     # x == 2
   leaq  (%rsi,%rdx), %rax
   jmp   .L5
.L9:     # x == 3
   movl  $1, %eax # w = 1
.L5:     # merge:
   addq  %rdx, %rax # w += z
   ret
```


Much modified from a slide supplied by CMU.



So, now that we know how switch statements are implemented, how might we "reverse engineer" object code to figure out the switch statement it implements?

Here we're running gdb on a program that contains a call to *switch_eg*. We gave the command "layout asm" so that we can see the assembly listing at the top of the slide. We set a breakpoint at *switch_eg*.

Assuming no knowledge of the original source code, we look at the code for *switch_eg* and see an indirect jump instruction at switch_eg+20, which is a definite indication that the C code contained a switch statement. We can see that %r8 contains the address of the offset table, and that %rcx will be set to the entry in the table at the index given in %rdi. The contents of %r8 are added to %rcx, thus causing %rcx to point to the instruction the indirect jump will go to.

So, with all this in mind, after the breakpoint was reached, we issued the *stepi* (si) command 3 times so that the code giving %r8 a value is executed (switch_eg+6). We then used the x/10dw gdb command to print 10 entries of a jump offset table starting at the address contained in %r8. We had to guess how many entries there are – 10 seems reasonable in that it seems unlikely that a switch statement has more than 10 cases, though it might. We know that the table comes after the executable code, so the entries are negative. We see seven entries with values reasonably close to one another, while the remaining entries are very different, so we conclude that the jump table contains 7 entries.

Gdb and Switch

```
B+ |0x555555555145 <switch_eg>          cmp    $0x6,%rdi                       |
   |0x555555555149 <switch_eg+4>        ja     0x55555555517b <switch_eg+54|
   |0x55555555514b <switch_eg+6>        lea    0xeb2(%rip),%r8        # 0x5|
   |0x555555555152 <switch_eg+13>       movslq (%r8,%rdi,4),%rcx               |
  >|0x555555555156 <switch_eg+17>       add    %r8,%rcx                        |
   |0x555555555159 <switch_eg+20>       jmpq   *%rcx                           |
   |0x55555555515b <switch_eg+22>       mov    %rsi,%rax                       |
   |0x55555555515e <switch_eg+25>       imul   %rdx,%rax                       |
   |0x555555555162 <switch_eg+29>       retq                                   |
   |0x555555555163 <switch_eg+30>       lea    (%rsi,%rdx,1),%rax              |
   |0x555555555167 <switch_eg+34>       jmp    0x55555555516e <switch_eg+41|
   |0x555555555169 <switch_eg+36>       mov    $0x1,%eax                       |
   |0x55555555516e <switch_eg+41>       add    %rdx,%rax                       |
   |0x555555555171 <switch_eg+44>       retq                                   |
   |0x555555555172 <switch_eg+45>       mov    $0x1,%eax                       |
   |0x555555555177 <switch_eg+50>       sub    %rdx,%rax                       |
   |0x55555555517a <switch_eg+53>       retq                                   |
   |0x55555555517b <switch_eg+54>       mov    $0x2,%eax                       |
   |0x555555555180 <switch_eg+59>       retq                                   |

(gdb) x/10dw $r8
0x555555556004: -3721   -3753   -3745   -3739
0x555555556014: -3721   -3730   -3730   680997
0x555555556024: 990059265       64
```


The code for some case of the switch should come immediately after the jmp (what else would go there?!). So the smallest (most negative) offset in the jump offset table must be the offset for this first code segment. Thus offset -3753 corresponds to switch_eg+22 in the assembly listing. It's at index 1 of the table, so it's this code that's executed when the first argument of switch_eg is 1.

Knowing this, we can figure out the rest.

Gdb and Switch

```
B+ |0x555555555145 <switch_eg>          cmp    $0x6,%rdi                                |
   |0x555555555149 <switch_eg+4>        ja     0x55555555517b <switch_eg+54|
   |0x55555555514b <switch_eg+6>        lea    0xeb2(%rip),%r8         # 0x5|
   |0x555555555152 <switch_eg+13>       movslq (%r8,%rdi,4),%rcx                        |
  >|0x555555555156 <switch_eg+17>       add    %r8,%rcx                                 |
   |0x555555555159 <switch_eg+20>       jmpq   *%rcx                                    |
   |0x55555555515b <switch_eg+22>       mov    %rsi,%rax                                |
   |0x55555555515e <switch_eg+25>       imul   %rdx,%rax                                |
   |0x555555555162 <switch_eg+29>       retq                                            |
   |0x555555555163 <switch_eg+30>       lea    (%rsi,%rdx,1),%rax                       |
   |0x555555555167 <switch_eg+34>       jmp    0x55555555516e <switch_eg+41|
   |0x555555555169 <switch_eg+36>       mov    $0x1,%eax                                |
   |0x55555555516e <switch_eg+41>       add    %rdx,%rax                                |
   |0x555555555171 <switch_eg+44>       retq                                            |
   |0x555555555172 <switch_eg+45>       mov    $0x1,%eax                                |
   |0x555555555177 <switch_eg+50>       sub    %rdx,%rax                                |
   |0x55555555517a <switch_eg+53>       retq                                            |
   |0x55555555517b <switch_eg+54>       mov    $0x2,%eax                                |
   |0x555555555180 <switch_eg+59>       retq                                            |

    (gdb) x/10dw $r8
    0x555555556004: -3721    -3753    -3745    -3739
    0x555555556014: -3721    -3730    -3730    680997
    0x555555556024: 990059265         64
```


What's at index 0 of the table (-3721) is the offset of the code associated with that index. It's 32 greater than the smallest offset, so its code must start 32 bytes beyond the start of the code for the smallest offset. Thus, it starts at swtich_eg+22 +32, or switch_eg+54.

Gdb and Switch

```
B+ |0x555555555145 <switch_eg>         cmp    $0x6,%rdi                           |
   |0x555555555149 <switch_eg+4>       ja     0x55555555517b <switch_eg+54|
   |0x55555555514b <switch_eg+6>       lea    0xeb2(%rip),%r8        # 0x5|
   |0x555555555152 <switch_eg+13>      movslq (%r8,%rdi,4),%rcx           |
  >|0x555555555156 <switch_eg+17>      add    %r8,%rcx                    |
   |0x555555555159 <switch_eg+20>      jmpq   *%rcx                       |
   |0x55555555515b <switch_eg+22>      mov    %rsi,%rax                   |
   |0x55555555515e <switch_eg+25>      imul   %rdx,%rax                   |
   |0x555555555162 <switch_eg+29>      retq                               |
   |0x555555555163 <switch_eg+30>      lea    (%rsi,%rdx,1),%rax          |
   |0x555555555167 <switch_eg+34>      jmp    0x55555555516e <switch_eg+41|
   |0x555555555169 <switch_eg+36>      mov    $0x1,%eax                   |
   |0x55555555516e <switch_eg+41>      add    %rdx,%rax                   |
   |0x555555555171 <switch_eg+44>      retq                               |
   |0x555555555172 <switch_eg+45>      mov    $0x1,%eax                   |
   |0x555555555177 <switch_eg+50>      sub    %rdx,%rax                   |
   |0x55555555517a <switch_eg+53>      retq                               |
   |0x55555555517b <switch_eg+54>      mov    $0x2,%eax                   |
   |0x555555555180 <switch_eg+59>      retq                               |

(gdb) x/10dw $r8
0x555555556004: -3721   -3753   -3745   -3739
0x555555556014: -3721   -3730   -3730   680997
0x555555556024: 990059265       64
```


Taking this one step further, the code for index 2 is at offset -3745, which is 8 bytes beyond the code for index 1. Thus, the code for index 2 starts at switch_eg+22 +8, or switch_eg+30.

Quiz 1

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

```
        movq    $0, %rax
.L2:
        movq    %rax, a(,%rax,8)
        addq    $1, %rax
        cmpq    $10, %rax
        jne     .L2
        ret
```

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

a

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

b

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

c