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.

Swapxy for Ints

```
struct xy {
  int x;
  int y;
}
void swapxy(struct xy *p){
  int temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```

```
swap:
  movl (%rdi), %eax
  movl 4(%rdi), %edx
  movl %edx, (%rdi)
  movl %eax, 4(%rdi)
  ret
```

- **Pointers are 64 bits**
- **What they point to are 32 bits**

Here we have a simple function that swaps the two components of a structure that's passed to it. (Assume that %rdi contains the argument.) Note that even though we use the "e" form of the registers to hold the (32-bit) data, we need the "r" form to hold the 64-bit addresses.

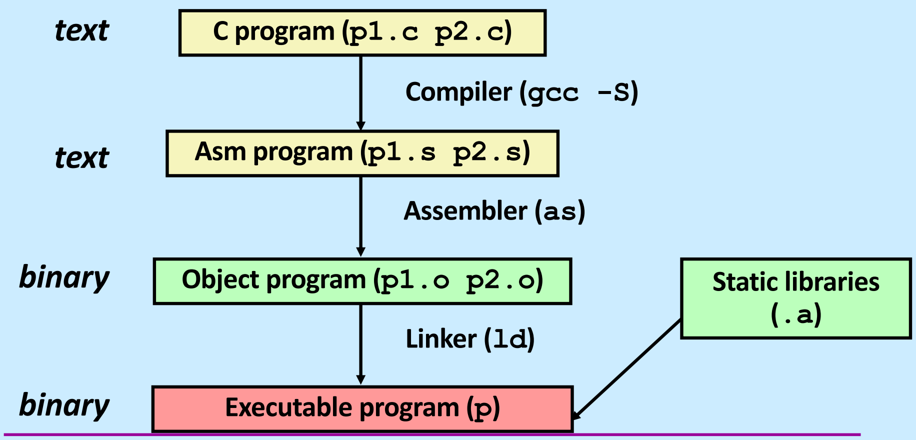
Bytes

- **Each register has a byte version**
  - **e.g., %r10: %r10b; see earlier slide for x86 registers**
- **Needed for byte instructions**
  - **movb (%rax, %rsi), %r10b**
  - **sets *only* the low byte in %r10**
    - » **other seven bytes are unchanged**
- **Alternatives**
  - **movzbq (%rax, %rsi), %r10**
    - » **copies byte to low byte of %r10**
    - » **zeroes go to higher bytes**
  - **movsbq (%rax, %rsi), %r10**
    - » **copies byte to low byte of %r10**
    - » **sign is extended to all higher bits**

Note that using single-byte versions of registers has a different behavior from using 4-byte versions of registers. Putting data into the latter using **mov** causes the upper bytes to be zeroed. But with the byte versions, putting data into them does not affect the upper bytes.

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc –O1 p1.c p2.c -o p`
  - use basic optimizations (`–O1`)
  - put resulting binary in file `p`



Supplied by CMU.

Note that normally one does not ask gcc to produce assembler code, but instead it compiles C code directly into machine code (producing an object file). Note also that the gcc command actually invokes a script; the compiler (also known as gcc) compiles code into either assembler code or machine code; if necessary, the assembler (as) assembles assembler code into object code. The linker (ld) links together multiple object files (containing object code) into an executable program.

Example

```
long ASum(long *a, unsigned long size) {
   long i, sum = 0;
   for (i=0; i<size; i++)
      sum += a[i];
   return sum;
}

int main() {
   long array[3] = {2,117,-6};
   long sum = ASum(array, 3);
   return sum;
}
```

Assembler Code

```
ASum:
        testq   %rsi, %rsi
        je      .L4
        movq    %rdi, %rax
        leaq    (%rdi,%rsi,8), %rcx
        movl    $0, %edx
.L3:
        addq    (%rax), %rdx
        addq    $8, %rax
        cmpq    %rcx, %rax
        jne     .L3
.L1:
        movq    %rdx, %rax
        ret
.L4:
        movl    $0, %edx
        jmp     .L1
```

```
main:
        subq    $32, %rsp
        movq    $2, (%rsp)
        movq    $117, 8(%rsp)
        movq    $-6, 16(%rsp)
        movq    %rsp, %rdi
        movl    $3, %esi
        call    ASum
        addq    $32, %rsp
        ret
```


Here is the assembler code produced by gcc from the C code of the previous slide. Note that the two movl instructions are ostensibly just copying a zero into %edx (a 32-bit register). However, what it's really doing is copying a zero in the 64-bit register %rdx (the 64-bit extension of %edx). This happens because, as we discussed earlier, when one copies something into a 32-bit register, the high-order 32 bits of its extension is filled with 0s.

Object Code

Code for `ASum`

```
0x1125 <ASum>:
    0x48
    0x85
    0xf6
    0x74
    0x1c
    0x48
    0x89
    0xf8
    0x48
    0x8d
    0x0c
    0xf7
     .
     .
     .
```

- Total of 39 bytes
- Each instruction: 1, 2, or 3 bytes
- Starts at address `0x1125`

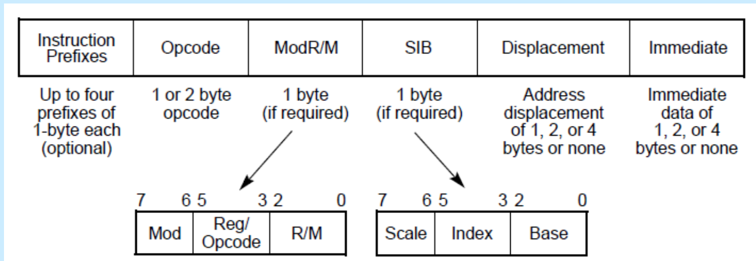
- **Assembler**
  - translates `.s` into `.o`
  - binary encoding of each instruction
  - nearly complete image of executable code
  - missing linkages between code in different files
- **Linker**
  - resolves references between files
  - combines with static run-time libraries
    - » e.g., code for `printf`
  - some libraries are *dynamically linked*
    - » linking occurs when program begins execution

Adapted from a slide supplied by CMU.

The lefthand column shows the object code produced by gcc. This was produced either by assembling the code of the previous slide, or by compiling the C code of the slide before that.

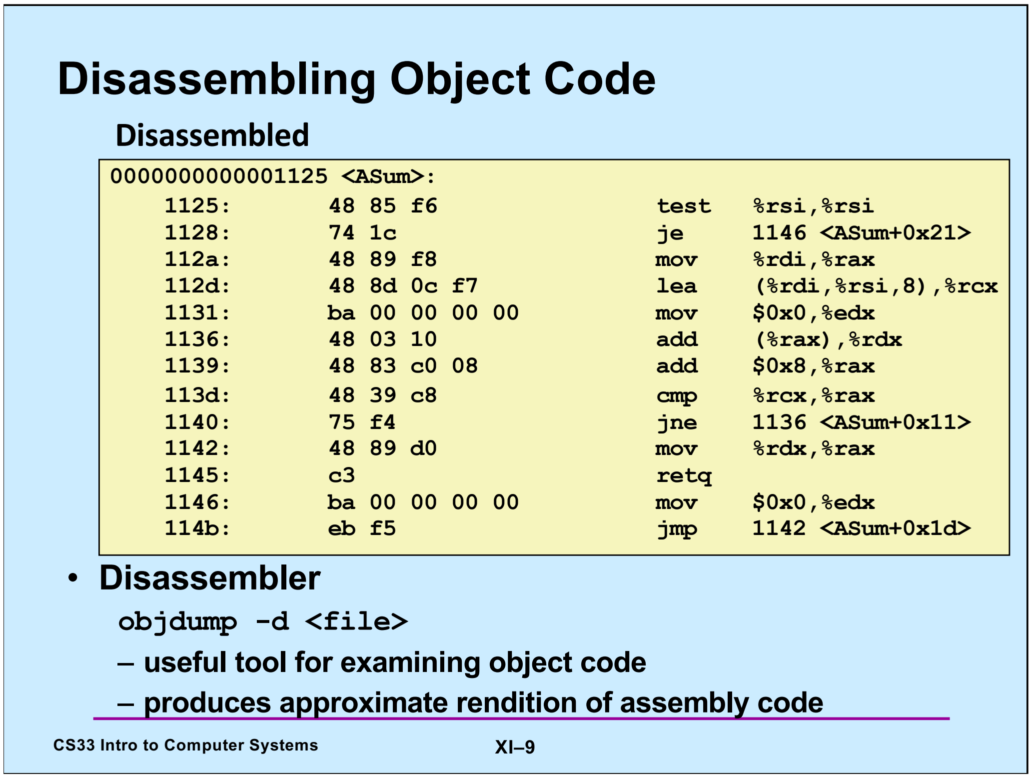
Suppose that all we have is the object code – we don't have the assembler code and the C code. Can we translate for object code to assembler code? (This is known as disassembling.)

Instruction Format



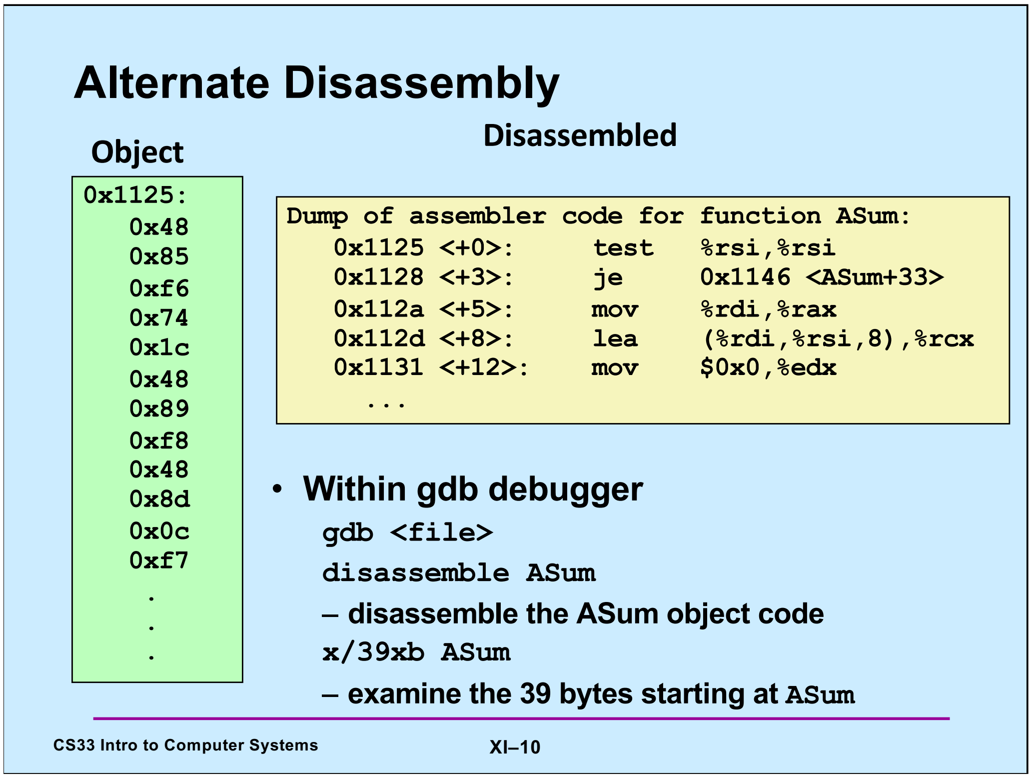
This is taken from Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 2: Instruction Set Reference; Order Number 325462-043US, Intel Corporation, May 2012 (https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4)

The point of the slide is that the instruction format is complicated, too much so for a human to deal with. Which is why we talk about **disassemblers** in the next slides.



Adapted from a slide supplied by CMU.

objdump's rendition is approximate because it assumes everything in the file is assembly code, and thus translates data into (often really weird) assembly code. Also, it leaves off the suffix at the end of each instruction, assuming it can be determined from context.



Adapted from a slide supplied by CMU.

The "x/35xb" directive to gdb says to examine (first x, meaning print) 35 bytes (b) viewed as hexadecimal (second x) starting at ASum.

The format of the output has been modified a bit from what gdb actually produces, so that it will fit on the slide. In the dump of the assembler code, the addresses are actually 64-bit values (in hex) – we have removed the leading 0s. The output of the x command is actually displayed in multiple columns. We have reorganized it into one column.

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!

Some Arithmetic Operations

- **Two-operand instructions:**

Format		Computation	
`addl`	Src,Dest	Dest = Dest + Src	
`subl`	Src,Dest	Dest = Dest – Src	
`imull`	Src,Dest	Dest = Dest * Src	
`shll`	Src,Dest	Dest = Dest << Src	Also called sall
`sarl`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrl`	Src,Dest	Dest = Dest >> Src	Logical
`xorl`	Src,Dest	Dest = Dest ^ Src	
`andl`	Src,Dest	Dest = Dest & Src	
`orl`	Src,Dest	Dest = Dest | Src	

  – **watch out for argument order!**

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

```c
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.

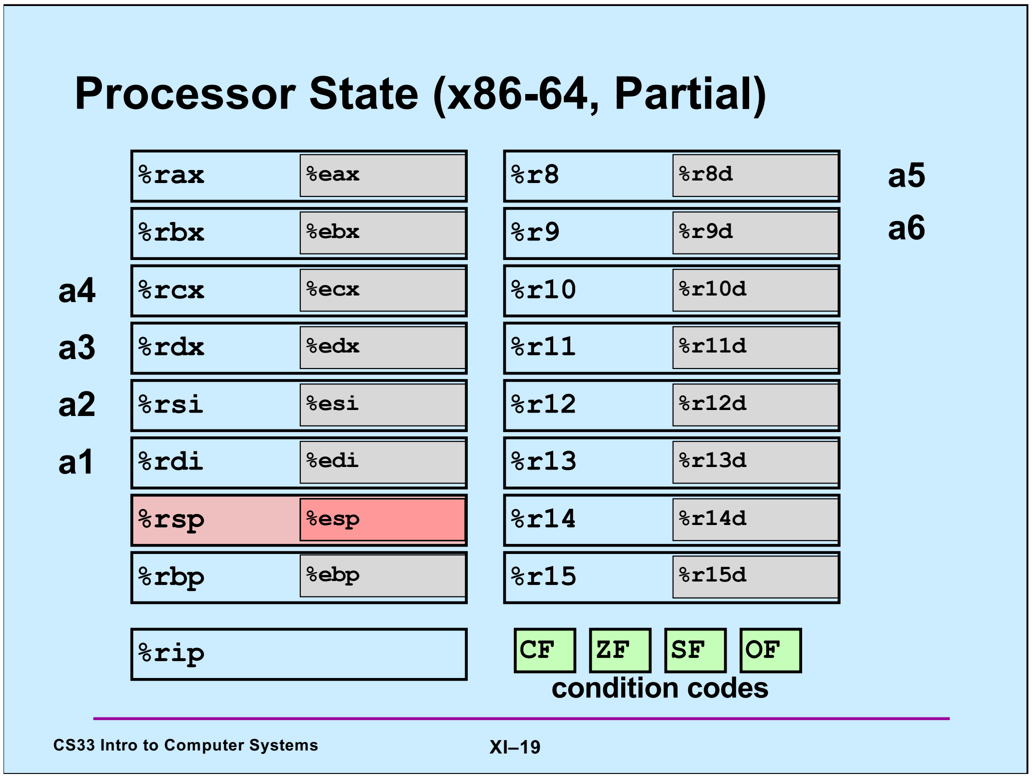
Another Example

```
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.



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.

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.

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 some issues come up because of 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 (because A is positive, B is negative, and A-B is a large positive number that does not fit in an int), 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.

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.

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 23.

Conditional-Branch Example

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

x in %edi

y in %esi

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


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;
```

Goto Version

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

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

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.

“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);
```

- **Body:**

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

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

Goto Version

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


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.

General “While” Translation

While version

```
while (Test)
  Body
```

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:
```


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.

“For” Loop → While Loop

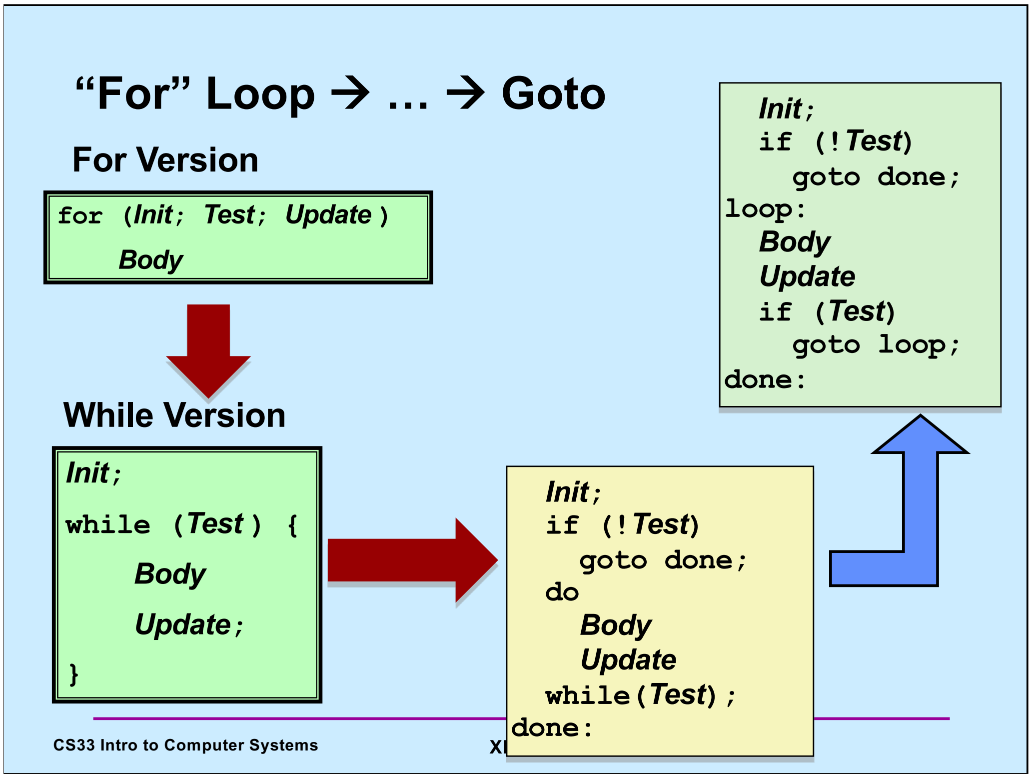
For Version

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

While Version

```
Init;
while (Test ) {
     Body
     Update;
}
```


Supplied by CMU.



Supplied by CMU.

“For” Loop Conversion 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;
}
```

Initial test can be optimized away

Goto Version

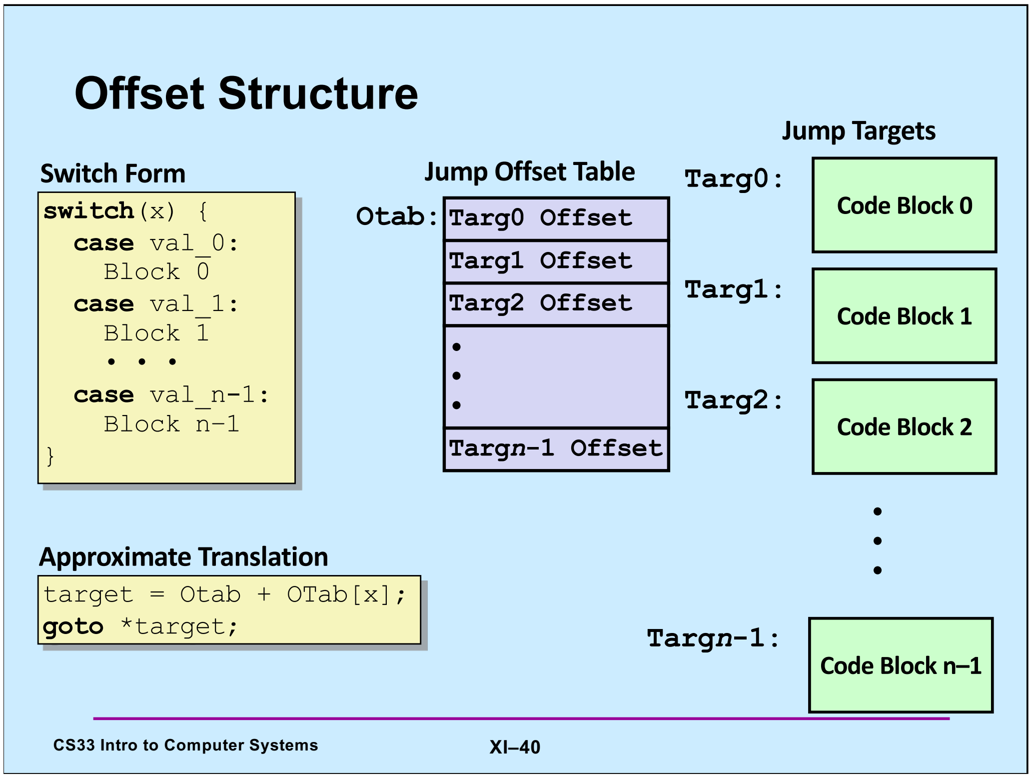
```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                      Init
  if (!(i < WSIZE))           !Test
    goto done;
 loop:
  {                           Body
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;  Update
  if (i < WSIZE)  Test
    goto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


Supplied by CMU.

Switch-Statement Example

```
long switch_eg (long m, long d) {
    if (d < 1) return 0;
    switch(m) {
    case 1: case 3: case 5:
    case 7: case 8: case 10:
    case 12:
        if (d > 31) return 0;
        else return 1;
    case 2:
        if (d > 28) return 0;
        else return 1;
    case 4: case 6: case 9:
    case 11:
        if (d > 30) return 0;
        else return 1;
    default:
        return 0;
    }
    return 0;
}
```


Code very much like this appears in level three of the traps project.



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
        movl    $0, %eax                          .align 4
        testq   %rsi, %rsi                  .L4:
        jle     .L1                               .long   .L8-.L4
        cmpq    $12, %rdi                         .long   .L3-.L4
        ja      .L8                               .long   .L6-.L4
        leaq    .L4(%rip), %rdx                   .long   .L3-.L4
        movslq  (%rdx,%rdi,4), %rax               .long   .L5-.L4
        addq    %rdx, %rax                        .long   .L3-.L4
        jmp     *%rax                             .long   .L5-.L4
                                                  .long   .L3-.L4
                                                  .long   .L3-.L4
                                                  .long   .L5-.L4
                                                  .long   .L3-.L4
                                                  .long   .L5-.L4
                                                  .long   .L3-.L4
                                                  .text
```


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)

```
.L3:
        cmpq    $31, %rsi
        setle   %al
        movzbl  %al, %eax
        ret
.L6:
        cmpq    $28, %rsi
        setle   %al
        movzbl  %al, %eax
        ret
.L5:
        cmpq    $30, %rsi
        setle   %al
        movzbl  %al, %eax
        ret
.L8:
        movl    $0, %eax
.L1:
        ret
```

Assembler Code Explanation (1)

```
switch_eg:
        movl    $0, %eax    # return value set to 0
        testq   %rsi, %rsi  # sets cc based on %rsi & %rsi
        jle     .L1         # go to L1, where it returns 0
        cmpq    $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

- **testq %rsi, %rsi**
  - **sets cc based on the contents of %rsi (d)**
  - **jle**
    - **jumps if `(SF^OF)|ZF`**
    - **OF is not set**
    - **jumps if SF or ZF is set (i.e., < 1)**

The first three instructions cause control to go to .L1 if the second argument (d) is less than 1. At .L1 is code that simply returns (with a return value of 0).

Assembler Code Explanation (2)

```
switch_eg:
        movl    $0, %eax       # return value set to 0
        testq   %rsi, %rsi     # sets cc based on %rsi & %rsi
        jle     .L1            # go to L1, where it returns 0
        cmpq     $12, %rdi     # %rdi : 12
        ja       .L8           # go to L8 if %rdi > 12 or < 0
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

- **ja .L8**
  - **unsigned comparison, though m is signed!**
  - **jumps if %rdi > 12**
  - **also jumps if %rdi is negative**

The next two instructions simply check to make sure that %rsi (the first argument, m) is less than or equal to 12. If not, control goes to .L8, which sets the return value to 0 and returns. Of course, the return value (in %rax/%eax) is already zero, so setting it to zero again is unnecessary.

Note that we're using **ja** (jump if above), which is normally used after comparing unsigned values. The first argument, m, is a (signed) **long**. But if it is interpreted as an unsigned value, then if the leftmost bit (the sign bit) is set, it appears to be a very large unsigned value, and thus the jump is taken.

Assembler Code Explanation (3)

```
switch_eg:                                         .section    .rodata
        movl    $0, %eax                           .align 4
        testq   %rsi, %rsi                     .L4:
        jle     .L1                                .long   .L8-.L4 # m=0
        cmpq    $12, %rdi                          .long   .L3-.L4 # m=1
        ja      .L8                                .long   .L6-.L4 # m=2
        leaq    .L4(%rip), %rdx                    .long   .L3-.L4 # m=3
        movslq  (%rdx,%rdi,4), %rax                .long   .L5-.L4 # m=4
        addq    %rdx, %rax                         .long   .L3-.L4 # m=5
        jmp     *%rax                              .long   .L5-.L4 # m=6
                                                   .long   .L3-.L4 # m=7
                                                   .long   .L3-.L4 # m=8
                                                   .long   .L5-.L4 # m=9
                                                   .long   .L3-.L4 # m=10
                                                   .long   .L5-.L4 # m=11
                                                   .long   .L3-.L4 # m=12
                                                   .text
```


The table on the right is known as an **offset table**. Each line refers to the code to be executed for the corresponding value of m. Each entry in the table is a long (recall that in x86-64 assembler, long means 32 bits). The value of each entry is the difference between the address of the table (.L4) and the address of the code to be executed for a particular value of m (the other .L labels). Thus each entry is the distance (or offset) from the beginning of the table to the code for each case. Note that this offset might be negative. It's assumed that the offset fits in a 32-bit signed quantity (which the system guarantees to be true.)

One might ask why we put 32-bit offsets in the table rather than 64-bit addresses. The reason is to reduce the size of these tables – if we used addresses, they'd be twice the size.

This table is not executable (it just contains offsets), but it also should be treated as read-only – its contents will never change. The directive ".section .rodata" tells the assembler that we want this table to be located in memory that is read-only, but not executable. The directive at the end of the table (".text")tells the assembler that what follows is (again) executable code.

The highlighted code on the left is what interprets the table, We examine it next.

Assembler Code Explanation (4)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

indirect jump

```
        .section     .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```


The highlighted code makes use of an indirect jump instruction, indicated by having an asterisk before its register operand. The register contains an address, and the jump is made to the code at that address.

Assembler Code Explanation (5)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq    $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

```
        .section    .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```


The **leaq** instruction (load effective address, quad), performs an address computation, but rather than fetching the data at the address, it stores the address itself in %rdx.

What's unusual about the instruction is that it uses %rip (the instruction pointer) as the base register, and has a displacement that is a label. This is a special case for the assembler, which can compute the offset between the leaq instruction and the label, and use that value for the displacement field.

Assembler Code Explanation (6)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq    $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

```
        .section    .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```


The **movslq** instruction copies a long (32 bits) into a quad (64 bits), and does sign extension so as to preserve the sign of the value being copied.

%rdi contains m, the first argument, which is also the argument of the switch statement. We use it to index into the offset table: As we saw in the previous slide, %rdx contains the address of the table, whose entries are each 4 bytes long. Thus we use %rdi as an index register, with a scale factor of 4. The contents of that entry (which is the distance from the table to the code that should be executed to handle this case) is copied into %rax, using sign extension to fill the register.

Assembler Code Explanation (7)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq    $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

```
        .section    .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```


The offset of the code we want to jump to is in %rax. To convert this offset into an absolute address, we need to add to it the address of the table. That's what the **addq** instruction does.

We can now do the indirect jump, to the address contained in %rax.

Switch Statements and Traps

- The code we just looked at was compiled with gcc's O1 flag
  - a moderate amount of "optimization"
- Traps is compiled with the O0 flag
  - no optimization
- O0 often produces easier-to-read (but less efficient) code
  - not so for switch

O1 vs. O0 Code

```
switch_egO1:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq  (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
```

```
switch_eg00:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    cmpq    $0, -16(%rbp)
    jg      .L2
    movl    $0, %eax
    jmp     .L3
.L2:
    cmpq    $12, -8(%rbp)
    ja      .L4
    movq    -8(%rbp), %rax
    leaq    0(,%rax,4), %rdx
    leaq    .L6(%rip), %rax
    movl    (%rdx,%rax), %eax
    cltq
    leaq    .L6(%rip), %rdx
    addq    %rdx, %rax
    jmp     *%rax
```


On the left we have the O1 version of the code, on the right we have the O0.

O1 vs. O0 Code Explanation (1)

```
switch_eg01:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq  (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
```

```
switch_eg00:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    cmpq    $0, -16(%rbp)
    jg      .L2
    movl    $0, %eax
    jmp     .L3
.L2:
    cmpq    $12, -8(%rbp)
    ja      .L4
    movq    -8(%rbp), %rax
    leaq    0(,%rax,4), %rdx
    leaq    .L6(%rip), %rax
    movl    (%rdx,%rax), %eax
    cltq
    leaq    .L6(%rip), %rdx
    addq    %rdx, %rax
    jmp     *%rax
```


The highlighted code is not present in the O1 version. It links this function's stack frame to its caller, something we'll talk about in the next lecture. It also (rather inexplicably) copies the arguments from the registers to the stack frame.

O1 vs. O0 Code Explanation (2)

```
switch_egO1:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq  (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
```

```
switch_eg00:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    cmpq    $0, -16(%rbp)
    jg      .L2
    movl    $0, %eax
    jmp     .L3
.L2:
    cmpq    $12, -8(%rbp)
    ja      .L4
    movq    -8(%rbp), %rax
    leaq    0(,%rax,4), %rdx
    leaq    .L6(%rip), %rax
    movl    (%rdx,%rax), %eax
    cltq
    leaq    .L6(%rip), %rdx
    addq    %rdx, %rax
    jmp     *%rax
```


The next four instructions compare the second argument (d) with 0; if it's less than or equal to zero, it returns 0 (.L3 is the label of code that simply returns). Otherwise it jumps to .L2.

O1 vs. O0 Code Explanation (3)

```
switch_egO1:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq  (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
```

```
switch_eg00:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    cmpq    $0, -16(%rbp)
    jg      .L2
    movl    $0, %eax
    jmp     .L3
.L2:
    cmpq    $12, -8(%rbp)
    ja      .L4
    movq    -8(%rbp), %rax
    leaq    0(,%rax,4), %rdx
    leaq    .L6(%rip), %rax
    movl    (%rdx,%rax), %eax
    cltq
    leaq    .L6(%rip), %rdx
    addq    %rdx, %rax
    jmp     *%rax
```


The next two instructions do the same thing as the **cmpq** and **ja** instructions do in the O1 code. If the first argument (m) is greater than 12 or less than 0, these instructions cause a jump (in this case to .L4, which labels what's essentially the same code as is labelled by .L8 in the O1 code) to code that returns 0.

O1 vs. O0 Code Explanation (4)

```
switch_eg01:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq  (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
```

```
switch_eg00:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    cmpq    $0, -16(%rbp)
    jg      .L2
    movl    $0, %eax
    jmp     .L3
.L2:
    cmpq    $12, -8(%rbp)
    ja      .L4
    movq    -8(%rbp), %rax
    leaq    0(,%rax,4), %rdx
    leaq    .L6(%rip), %rax
    movl    (%rdx,%rax), %eax
    cltq
    leaq    .L6(%rip), %rdx
    addq    %rdx, %rax
    jmp     *%rax
```


In the next three instructions, the first one copies the first argument (m, which had been earlier copied to the stack) to %rax. Recall that m is the argument to the switch statement, and will be used as an index into the jump table.

The first **leaq** statement computes four times %rax, and puts the result into %rdx. The next **leaq** statement does the same thing as the **leaq** statement of the O1 code, it computes the address of the offset table (which is labelled with .L6 in this version) and stores it in %rax.

O1 vs. O0 Code Explanation (5)

```
switch_egO1:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq  (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
```

```
switch_eg00:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    cmpq    $0, -16(%rbp)
    jg      .L2
    movl    $0, %eax
    jmp     .L3
.L2:
    cmpq    $12, -8(%rbp)
    ja      .L4
    movq    -8(%rbp), %rax
    leaq    0(,%rax,4), %rdx
    leaq    .L6(%rip), %rax
    movl    (%rdx,%rax), %eax
    cltq
    leaq    .L6(%rip), %rdx
    addq    %rdx, %rax
    jmp     *%rax
```


Finally, the **movl** and **cltq** instructions, along with the first of the earlier **leaq** instructions, do what the **movslq** instruction did in the O1 version. The **movl** instruction copies the offset-table entry into (32-bit) %eax. The **cltq** is a rather obscure instruction that sign extends the value in %eax so that it fills the entire (64-bit) %rax. Then the address of the offset table is computed (again) via the **leaq** instruction.

The final two instructions do what the final two instructions do for the O1 code: they add the offset obtained from the table to the address of the table, then jump to the resulting address.

Gdb and Switch (1)

```
B+ 0x555555555169 <switch_eg+4>      mov    %rdi,-0x8(%rbp)
   0x55555555516d <switch_eg+8>      mov    %rsi,-0x10(%rbp)
   0x555555555171 <switch_eg+12>     cmpq   $0x0,-0x10(%rbp)
   0x555555555176 <switch_eg+17>     jg     0x55555555517f <nswitch+26>
   0x555555555178 <switch_eg+19>     mov    $0x0,%eax
   0x55555555517d <switch_eg+24>     jmp    0x5555555551ee <nswitch+137>
   0x55555555517f <switch_eg+26>     cmpq   $0xc,-0x8(%rbp)
   0x555555555184 <switch_eg+31>     ja     0x5555555551e9 <nswitch+132>
   0x555555555186 <switch_eg+33>     mov    -0x8(%rbp),%rax
   0x55555555518a <switch_eg+37>     lea    0x0(,%rax,4),%rdx
   0x555555555192 <switch_eg+45>     lea    0xe6b(%rip),%rax        # 0x5555555
   0x555555555199 <switch_eg+52>     mov    (%rdx,%rax,1),%eax
   0x55555555519c <switch_eg+55>     cltq
   0x55555555519e <switch_eg+57>     lea    0xe5f(%rip),%rdx        # 0x5555555
   0x5555555551a5 <switch_eg+64>     add    %rdx,%rax
  >0x5555555551a8 <switch_eg+67>     jmp    *%rax
```

```
(gdb) x/14dw $rdx
0x555555556004: -3611   -3674   -3653   -3674
0x555555556014: -3632   -3674   -3632   -3674
0x555555556024: -3674   -3632   -3674   -3632
0x555555556034: -3674   1734439765
```


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+67, which is a definite indication that the C code contained a switch statement. We can see that %rdx contains the address of the offset table, and that %rax will be set to the entry in the table at the index given in %rdi. The contents of %rdx are added to %rax, thus causing %rax 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 15 times so that we could see the values of all registers just before the indirect jmp. We then used the **x/14dw** gdb command to print 14 entries of a jump offset table starting at the address contained in %rdx. We had to guess how many entries there are – 14 seems reasonable in that it seems unlikely that a switch statement has more than 14 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 entry is very different, so we conclude that the jump table contains13 entries.

Gdb and Switch (2)

```
│  0x5555555551a5 <switch_eg+64>     add    %rdx,%rax                          │
│ >0x5555555551a8 <switch_eg+67>     jmp    *%rax                              │
│  0x5555555551aa <switch_eg+69>     cmpq   $0x1f,-0x10(%rbp)                  │
│  0x5555555551af <switch_eg+74>     jle    0x5555555551b8 <nswitch+83>        │
│  0x5555555551b1 <switch_eg+76>     mov    $0x0,%eax                          │
│  0x5555555551b6 <switch_eg+81>     jmp    0x5555555551ee <nswitch+137>       │
│  0x5555555551b8 <switch_eg+83>     mov    $0x1,%eax                          │
│  0x5555555551bd <switch_eg+88>     jmp    0x5555555551ee <nswitch+137>       │
│  0x5555555551bf <switch_eg+90>     cmpq   $0x1c,-0x10(%rbp)                  │
│  0x5555555551c4 <switch_eg+95>     jle    0x5555555551cd <nswitch+104>       │
```

```
(gdb) x/14dw $rdx
0x555555556004: -3611   -3674   -3653   -3674
0x555555556014: -3632   -3674   -3632   -3674
0x555555556024: -3674   -3632   -3674   -3632
0x555555556034: -3674   1734439765
```


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 -3674 corresponds to switch_eg+69 in the assembly listing. It's at indices 1, 3, 5, 7, 8, 10, and 12 of the table, so it's this code that's executed when the first argument of switch_eg is 1, 3, 5, 7, 8, 10, or 12.

Knowing this, we can figure out the rest.

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
```

a

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

b

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

c

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