3.5.1 Load Effective Address

The *load effective address* instruction `leaq` is actually a variant of the `movq` instruction. It has the form of an instruction that reads from memory to a register,

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D+1$	Increment
DEC	D	$D \leftarrow D-1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_{\mathrm{A}} k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_{\mathrm{L}} k$	Logical right shift

Figure 3.10 Integer arithmetic operations. The load effective address (`leaq`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $>>_{\mathrm{A}}$ and $>>_{\mathrm{L}}$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

but it does not reference memory at all. Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination. We indicate this computation in Figure 3.10 using the C address operator $\&S$. This instruction can be used to generate pointers for later memory references. In addition, it can be used to compactly describe common arithmetic operations. For example, if register `%rdx` contains value x, then the instruction `leaq 7(%rdx,%rdx,4), %rax` will set register `%rax` to $5x + 7$. Compilers often find clever uses of `leaq` that have nothing to do with effective address computations. The destination operand must be a register.

Practice Problem 3.6 (solution page 327)

Suppose register `%rax` holds value x and `%rcx` holds value y. Fill in the table below with formulas indicating the value that will be stored in register `%rdx` for each of the given assembly-code instructions:

Instruction	Result
`leaq 6(%rax), %rdx`	_____
`leaq (%rax,%rcx), %rdx`	_____
`leaq (%rax,%rcx,4), %rdx`	_____
`leaq 7(%rax,%rax,8), %rdx`	_____

```
leaq 0xA(,%rcx,4), %rdx      _____
leaq 9(%rax,%rcx,2), %rdx    _____
```

As an illustration of the use of `leaq` in compiled code, consider the following C program:

```
long scale(long x, long y, long z) {
    long t = x + 4 * y + 12 * z;
    return t;
}
```

When compiled, the arithmetic operations of the function are implemented by a sequence of three `leaq` functions, as is documented by the comments on the right-hand side:

```
  long scale(long x, long y, long z)
  x in %rdi, y in %rsi, z in %rdx
scale:
  leaq    (%rdi,%rsi,4), %rax     x + 4*y
  leaq    (%rdx,%rdx,2), %rdx     z + 2*z = 3*z
  leaq    (%rax,%rdx,4), %rax     (x+4*y) + 4*(3*z) = x + 4*y + 12*z
  ret
```

The ability of the `leaq` instruction to perform addition and limited forms of multiplication proves useful when compiling simple arithmetic expressions such as this example.

Practice Problem 3.7 (solution page 328)

Consider the following code, in which we have omitted the expression being computed:

```
long scale2(long x, long y, long z) {
    long t = ____________________;
    return t;
}
```

Compiling the actual function with GCC yields the following assembly code:

```
  long scale2(long x, long y, long z)
  x in %rdi, y in %rsi, z in %rdx
scale2:
  leaq    (%rdi,%rdi,4), %rax
  leaq    (%rax,%rsi,2), %rax
  leaq    (%rax,%rdx,8), %rax
  ret
```

Fill in the missing expression in the C code.

3.6.6 Implementing Conditional Branches with Conditional Moves

The conventional way to implement conditional operations is through a conditional transfer of *control*, where the program follows one execution path when a condition holds and another when it does not. This mechanism is simple and general, but it can be very inefficient on modern processors.

An alternate strategy is through a conditional transfer of *data*. This approach computes both outcomes of a conditional operation and then selects one based on whether or not the condition holds. This strategy makes sense only in restricted cases, but it can then be implemented by a simple *conditional move* instruction that is better matched to the performance characteristics of modern processors. Here, we examine this strategy and its implementation with x86-64.

Figure 3.17(a) shows an example of code that can be compiled using a conditional move. The function computes the absolute value of its arguments `x` and `y`, as did our earlier example (Figure 3.16). Whereas the earlier example had side effects in the branches, modifying the value of either `lt_cnt` or `ge_cnt`, this version simply computes the value to be returned by the function.

(a) Original C code

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

(b) Implementation using conditional assignment

```
1  long cmovdiff(long x, long y)
2  {
3      long rval = y-x;
4      long eval = x-y;
5      long ntest = x >= y;
6      /* Line below requires
7         single instruction: */
8      if (ntest) rval = eval;
9      return rval;
10 }
```

(c) Generated assembly code

```
    long absdiff(long x, long y)
    x in %rdi, y in %rsi
1   absdiff:
2     movq    %rsi, %rax
3     subq    %rdi, %rax      rval = y-x
4     movq    %rdi, %rdx
5     subq    %rsi, %rdx      eval = x-y
6     cmpq    %rsi, %rdi      Compare x:y
7     cmovge  %rdx, %rax      If >=, rval = eval
8     ret                     Return tval
```

Figure 3.17 Compilation of conditional statements using conditional assignment. (a) C function `absdiff` contains a conditional expression. The generated assembly code is shown (c), along with (b) a C function `cmovdiff` that mimics the operation of the assembly code.

For this function, GCC generates the assembly code shown in Figure 3.17(c), having an approximate form shown by the C function `cmovdiff` shown in Figure 3.17(b). Studying the C version, we can see that it computes both `y-x` and `x-y`, naming these `rval` and `eval`, respectively. It then tests whether x is greater than or equal to y, and if so, copies `eval` to `rval` before returning `rval`. The assembly code in Figure 3.17(c) follows the same logic. The key is that the single `cmovge` instruction (line 7) of the assembly code implements the conditional assignment (line 8) of `cmovdiff`. It will transfer the data from the source register to the destination, only if the `cmpq` instruction of line 6 indicates that one value is greater than or equal to the other (as indicated by the suffix `ge`).

To understand why code based on conditional data transfers can outperform code based on conditional control transfers (as in Figure 3.16), we must understand something about how modern processors operate. As we will see in Chapters 4 and 5, processors achieve high performance through *pipelining*, where an instruction is processed via a sequence of stages, each performing one small portion of the required operations (e.g., fetching the instruction from memory, determining the instruction type, reading from memory, performing an arithmetic operation, writing to memory, and updating the program counter). This approach achieves high performance by overlapping the steps of the successive instructions, such as fetching one instruction while performing the arithmetic operations for a previous instruction. To do this requires being able to determine the sequence of instructions to be executed well ahead of time in order to keep the pipeline full of instructions to be executed. When the machine encounters a conditional jump (referred to as a "branch"), it cannot determine which way the branch will go until it has evaluated the branch condition. Processors employ sophisticated *branch prediction logic* to try to guess whether or not each jump instruction will be followed. As long as it can guess reliably (modern microprocessor designs try to achieve success rates on the order of 90%), the instruction pipeline will be kept full of instructions. Mispredicting a jump, on the other hand, requires that the processor discard much of the work it has already done on future instructions and then begin filling the pipeline with instructions starting at the correct location. As we will see, such a misprediction can incur a serious penalty, say, 15–30 clock cycles of wasted effort, causing a serious degradation of program performance.

As an example, we ran timings of the `absdiff` function on an Intel Haswell processor using both methods of implementing the conditional operation. In a typical application, the outcome of the test `x < y` is highly unpredictable, and so even the most sophisticated branch prediction hardware will guess correctly only around 50% of the time. In addition, the computations performed in each of the two code sequences require only a single clock cycle. As a consequence, the branch misprediction penalty dominates the performance of this function. For x86-64 code with conditional jumps, we found that the function requires around 8 clock cycles per call when the branching pattern is easily predictable, and around 17.50 clock cycles per call when the branching pattern is random. From this, we can infer that the branch misprediction penalty is around 19 clock cycles. That means time required by the function ranges between around 8 and 27 cycles, depending on whether or not the branch is predicted correctly.

Aside How did you determine this penalty?

Assume the probability of misprediction is p, the time to execute the code without misprediction is T_{OK}, and the misprediction penalty is T_{MP}. Then the average time to execute the code as a function of p is $T_{avg}(p) = (1 - p)T_{OK} + p(T_{OK} + T_{MP}) = T_{OK} + pT_{MP}$. We are given T_{OK} and T_{ran}, the average time when $p = 0.5$, and we want to determine T_{MP}. Substituting into the equation, we get $T_{ran} = T_{avg}(0.5) = T_{OK} + 0.5T_{MP}$, and therefore $T_{MP} = 2(T_{ran} - T_{OK})$. So, for $T_{OK} = 8$ and $T_{ran} = 17.5$, we get $T_{MP} = 19$.

On the other hand, the code compiled using conditional moves requires around 8 clock cycles regardless of the data being tested. The flow of control does not depend on data, and this makes it easier for the processor to keep its pipeline full.

Practice Problem 3.19 (solution page 332)

Running on an older processor model, our code required around 16 cycles when the branching pattern was highly predictable, and around 31 cycles when the pattern was random.

A. What is the approximate miss penalty?

B. How many cycles would the function require when the branch is mispredicted?

Figure 3.18 illustrates some of the conditional move instructions available with x86-64. Each of these instructions has two operands: a source register or memory location S, and a destination register R. As with the different SET (Section 3.6.2) and jump (Section 3.6.3) instructions, the outcome of these instructions depends on the values of the condition codes. The source value is read from either memory or the source register, but it is copied to the destination only if the specified condition holds.

The source and destination values can be 16, 32, or 64 bits long. Single-byte conditional moves are not supported. Unlike the unconditional instructions, where the operand length is explicitly encoded in the instruction name (e.g., `movw` and `movl`), the assembler can infer the operand length of a conditional move instruction from the name of the destination register, and so the same instruction name can be used for all operand lengths.

Unlike conditional jumps, the processor can execute conditional move instructions without having to predict the outcome of the test. The processor simply reads the source value (possibly from memory), checks the condition code, and then either updates the destination register or keeps it the same. We will explore the implementation of conditional moves in Chapter 4.

To understand how conditional operations can be implemented via conditional data transfers, consider the following general form of conditional expression and assignment:

Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *R*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *R*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *R*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *R*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *R*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *R*	`cmovna`	`CF | ZF`	Below or equal (unsigned <=)

Figure 3.18 The conditional move instructions. These instructions copy the source value *S* to its destination *R* when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

`v = `*test-expr*` ? `*then-expr*` : `*else-expr*`;`

The standard way to compile this expression using conditional control transfer would have the following form:

```
    if (!test-expr)
        goto false;
    v = then-expr;
    goto done;
false:
    v = else-expr;
done:
```

This code contains two code sequences—one evaluating *then-expr* and one evaluating *else-expr*. A combination of conditional and unconditional jumps is used to ensure that just one of the sequences is evaluated.

For the code based on a conditional move, both the *then-expr* and the *else-expr* are evaluated, with the final value chosen based on the evaluation *test-expr*. This can be described by the following abstract code:

```
    v  = then-expr;
    ve = else-expr;
    t  = test-expr;
    if (!t) v = ve;
```

The final statement in this sequence is implemented with a conditional move—value `ve` is copied to `v` only if test condition `t` does not hold.

Not all conditional expressions can be compiled using conditional moves. Most significantly, the abstract code we have shown evaluates both *then-expr* and *else-expr* regardless of the test outcome. If one of those two expressions could possibly generate an error condition or a side effect, this could lead to invalid behavior. Such is the case for our earlier example (Figure 3.16). Indeed, we put the side effects into this example specifically to force GCC to implement this function using conditional transfers.

As a second illustration, consider the following C function:

```
long cread(long *xp) {
    return (xp ? *xp : 0);
}
```

At first, this seems like a good candidate to compile using a conditional move to set the result to zero when the pointer is null, as shown in the following assembly code:

```
    long cread(long *xp)
    Invalid implementation of function cread
    xp in register %rdi
1   cread:
2     movq    (%rdi), %rax    v = *xp
3     testq   %rdi, %rdi      Test x
4     movl    $0, %edx        Set ve = 0
5     cmove   %rdx, %rax      If x==0, v = ve
6     ret                     Return v
```

This implementation is invalid, however, since the dereferencing of `xp` by the `movq` instruction (line 2) occurs even when the test fails, causing a null pointer dereferencing error. Instead, this code must be compiled using branching code.

Using conditional moves also does not always improve code efficiency. For example, if either the *then-expr* or the *else-expr* evaluation requires a significant computation, then this effort is wasted when the corresponding condition does not hold. Compilers must take into account the relative performance of wasted computation versus the potential for performance penalty due to branch misprediction. In truth, they do not really have enough information to make this decision reliably; for example, they do not know how well the branches will follow predictable patterns. Our experiments with GCC indicate that it only uses conditional moves when the two expressions can be computed very easily, for example, with single add instructions. In our experience, GCC uses conditional control transfers even in many cases where the cost of branch misprediction would exceed even more complex computations.

Overall, then, we see that conditional data transfers offer an alternative strategy to conditional control transfers for implementing conditional operations. They can only be used in restricted cases, but these cases are fairly common and provide a much better match to the operation of modern processors.

3.6.8 Switch Statements

A `switch` statement provides a multiway branching capability based on the value of an integer index. They are particularly useful when dealing with tests where

there can be a large number of possible outcomes. Not only do they make the C code more readable, but they also allow an efficient implementation using a data structure called a *jump table*. A jump table is an array where entry i is the address of a code segment implementing the action the program should take when the switch index equals i. The code performs an array reference into the jump table using the switch index to determine the target for a jump instruction. The advantage of using a jump table over a long sequence of if-else statements is that the time taken to perform the switch is independent of the number of switch cases. GCC selects the method of translating a `switch` statement based on the number of cases and the sparsity of the case values. Jump tables are used when there are a number of cases (e.g., four or more) and they span a small range of values.

Figure 3.22(a) shows an example of a C `switch` statement. This example has a number of interesting features, including case labels that do not span a contiguous range (there are no labels for cases 101 and 105), cases with multiple labels (cases 104 and 106), and cases that *fall through* to other cases (case 102) because the code for the case does not end with a `break` statement.

Figure 3.23 shows the assembly code generated when compiling `switch_eg`. The behavior of this code is shown in C as the procedure `switch_eg_impl` in Figure 3.22(b). This code makes use of support provided by GCC for jump tables, as an extension to the C language. The array `jt` contains seven entries, each of which is the address of a block of code. These locations are defined by labels in the code and indicated in the entries in `jt` by code pointers, consisting of the labels prefixed by `&&`. (Recall that the operator '`&`' creates a pointer for a data value. In making this extension, the authors of GCC created a new operator `&&` to create a pointer for a code location.) We recommend that you study the C procedure `switch_eg_impl` and how it relates to the assembly-code version.

Our original C code has cases for values 100, 102–104, and 106, but the switch variable `n` can be an arbitrary integer. The compiler first shifts the range to between 0 and 6 by subtracting 100 from `n`, creating a new program variable that we call `index` in our C version. It further simplifies the branching possibilities by treating `index` as an *unsigned* value, making use of the fact that negative numbers in a two's-complement representation map to large positive numbers in an unsigned representation. It can therefore test whether `index` is outside of the range 0–6 by testing whether it is greater than 6. In the C and assembly code, there are five distinct locations to jump to, based on the value of `index`. These are `loc_A` (identified in the assembly code as `.L3`), `loc_B` (`.L5`), `loc_C` (`.L6`), `loc_D` (`.L7`), and `loc_def` (`.L8`), where the latter is the destination for the default case. Each of these labels identifies a block of code implementing one of the case branches. In both the C and the assembly code, the program compares `index` to 6 and jumps to the code for the default case if it is greater.

The key step in executing a `switch` statement is to access a code location through the jump table. This occurs in line 16 in the C code, with a `goto` statement that references the jump table `jt`. This *computed goto* is supported by GCC as an extension to the C language. In our assembly-code version, a similar operation occurs on line 5, where the `jmp` instruction's operand is prefixed with '`*`', indicating

(a) Switch statement

```
void switch_eg(long x, long n,
               long *dest)
{
    long val = x;

    switch (n) {

    case 100:
        val *= 13;
        break;

    case 102:
        val += 10;
        /* Fall through */

    case 103:
        val += 11;
        break;

    case 104:
    case 106:
        val *= val;
        break;

    default:
        val = 0;
    }
    *dest = val;
}
```

(b) Translation into extended C

```
1  void switch_eg_impl(long x, long n,
2                      long *dest)
3  {
4      /* Table of code pointers */
5      static void *jt[7] = {
6          &&loc_A, &&loc_def, &&loc_B,
7          &&loc_C, &&loc_D, &&loc_def,
8          &&loc_D
9      };
10     unsigned long index = n - 100;
11     long val;
12
13     if (index > 6)
14         goto loc_def;
15     /* Multiway branch */
16     goto *jt[index];
17
18  loc_A:    /* Case 100 */
19     val = x * 13;
20     goto done;
21  loc_B:    /* Case 102 */
22     x = x + 10;
23     /* Fall through */
24  loc_C:     /* Case 103 */
25     val = x + 11;
26     goto done;
27  loc_D:    /* Cases 104, 106 */
28     val = x * x;
29     goto done;
30  loc_def:  /* Default case */
31     val = 0;
32  done:
33     *dest = val;
34 }
```

Figure 3.22 Example `switch` statement and its translation into extended C. The translation shows the structure of jump table `jt` and how it is accessed. Such tables are supported by GCC as an extension to the C language.

an indirect jump, and the operand specifies a memory location indexed by register %eax, which holds the value of index. (We will see in Section 3.8 how array references are translated into machine code.)

Our C code declares the jump table as an array of seven elements, each of which is a pointer to a code location. These elements span values 0–6 of

```
    void switch_eg(long x, long n, long *dest)
    x in %rdi, n in %rsi, dest in %rdx
1   switch_eg:
2     subq    $100, %rsi              Compute index = n-100
3     cmpq    $6, %rsi                Compare index:6
4     ja      .L8                     If >, goto loc_def
5     jmp     *.L4(,%rsi,8)           Goto *jg[index]
6   .L3:                            loc_A:
7     leaq    (%rdi,%rdi,2), %rax     3*x
8     leaq    (%rdi,%rax,4), %rdi     val = 13*x
9     jmp     .L2                     Goto done
10  .L5:                            loc_B:
11    addq    $10, %rdi               x = x + 10
12  .L6:                            loc_C:
13    addq    $11, %rdi               val = x + 11
14    jmp     .L2                     Goto done
15  .L7:                            loc_D:
16    imulq   %rdi, %rdi              val = x * x
17    jmp     .L2                     Goto done
18  .L8:                            loc_def:
19    movl    $0, %edi                val = 0
20  .L2:                            done:
21    movq    %rdi, (%rdx)            *dest = val
22    ret                             Return
```

Figure 3.23 Assembly code for `switch` statement example in Figure 3.22.

`index`, corresponding to values 100–106 of `n`. Observe that the jump table handles duplicate cases by simply having the same code label (`loc_D`) for entries 4 and 6, and it handles missing cases by using the label for the default case (`loc_def`) as entries 1 and 5.

In the assembly code, the jump table is indicated by the following declarations, to which we have added comments:

```
1     .section        .rodata
2     .align 8          Align address to multiple of 8
3   .L4:
4     .quad   .L3       Case 100: loc_A
5     .quad   .L8       Case 101: loc_def
6     .quad   .L5       Case 102: loc_B
7     .quad   .L6       Case 103: loc_C
8     .quad   .L7       Case 104: loc_D
9     .quad   .L8       Case 105: loc_def
10    .quad   .L7       Case 106: loc_D
```

These declarations state that within the segment of the object-code file called `.rodata` (for "read-only data"), there should be a sequence of seven "quad" (8-byte) words, where the value of each word is given by the instruction address associated with the indicated assembly-code labels (e.g., `.L3`). Label `.L4` marks the start of this allocation. The address associated with this label serves as the base for the indirect jump (line 5).

The different code blocks (C labels `loc_A` through `loc_D` and `loc_def`) implement the different branches of the `switch` statement. Most of them simply compute a value for `val` and then go to the end of the function. Similarly, the assembly-code blocks compute a value for register `%rdi` and jump to the position indicated by label `.L2` at the end of the function. Only the code for case label 102 does not follow this pattern, to account for the way the code for this case falls through to the block with label 103 in the original C code. This is handled in the assembly-code block starting with label `.L5`, by omitting the `jmp` instruction at the end of the block, so that the code continues execution of the next block. Similarly, the C version `switch_eg_impl` has no `goto` statement at the end of the block starting with label `loc_B`.

Examining all of this code requires careful study, but the key point is to see that the use of a jump table allows a very efficient way to implement a multiway branch. In our case, the program could branch to five distinct locations with a single jump table reference. Even if we had a `switch` statement with hundreds of cases, they could be handled by a single jump table access.

Practice Problem 3.30 (solution page 338)

In the C function that follows, we have omitted the body of the `switch` statement. In the C code, the case labels did not span a contiguous range, and some cases had multiple labels.

```
void switch2(long x, long *dest) {
    long val = 0;
    switch (x) {
        .
        .   Body of switch statement omitted
        .
    }
    *dest = val;
}
```

In compiling the function, GCC generates the assembly code that follows for the initial part of the procedure, with variable `x` in `%rdi`:

```
    void switch2(long x, long *dest)
    x in %rdi
1   switch2:
2     addq    $1, %rdi
3     cmpq    $8, %rdi
4     ja      .L2
5     jmp     *.L4(,%rdi,8)
```

It generates the following code for the jump table:

```
1   .L4:
2     .quad   .L9
3     .quad   .L5
4     .quad   .L6
5     .quad   .L7
6     .quad   .L2
7     .quad   .L7
8     .quad   .L8
9     .quad   .L2
10    .quad   .L5
```

Based on this information, answer the following questions:

A. What were the values of the case labels in the `switch` statement?

B. What cases had multiple labels in the C code?

Practice Problem 3.31 (solution page 338)

For a C function `switcher` with the general structure

```
void switcher(long a, long b, long c, long *dest)
{
    long val;
    switch(a) {
    case __________:       /* Case A */
        c = __________;
        /* Fall through */
    case __________:       /* Case B */
        val = __________;
        break;
    case __________:       /* Case C */
    case __________:       /* Case D */
        val = __________;
        break;
    case __________:       /* Case E */
        val = __________;
        break;
    default:
        val = __________;
    }
    *dest = val;
}
```

GCC generates the assembly code and jump table shown in Figure 3.24.

Fill in the missing parts of the C code. Except for the ordering of case labels `C` and `D`, there is only one way to fit the different cases into the template.

(a) Code

```
    void switcher(long a, long b, long c, long *dest)
    a in %rsi, b in %rdi, c in %rdx, d in %rcx
1   switcher:
2     cmpq    $7, %rdi
3     ja      .L2
4     jmp     *.L4(,%rdi,8)
5     .section        .rodata
6   .L7:
7     xorq    $15, %rsi
8     movq    %rsi, %rdx
9   .L3:
10    leaq    112(%rdx), %rdi
11    jmp     .L6
12  .L5:
13    leaq    (%rdx,%rsi), %rdi
14    salq    $2, %rdi
15    jmp     .L6
16  .L2:
17    movq    %rsi, %rdi
18  .L6:
19    movq    %rdi, (%rcx)
20    ret
```

(b) Jump table

```
1   .L4:
2     .quad   .L3
3     .quad   .L2
4     .quad   .L5
5     .quad   .L2
6     .quad   .L6
7     .quad   .L7
8     .quad   .L2
9     .quad   .L5
```

Figure 3.24 Assembly code and jump table for Problem 3.31.

3.63 ◆◆

This problem will give you a chance to reverse engineer a `switch` statement from disassembled machine code. In the following procedure, the body of the `switch` statement has been omitted:

```
1  long switch_prob(long x, long n) {
2      long result = x;
3      switch(n) {
4          /* Fill in code here */
5
6      }
7      return result;
8  }
```

Figure 3.53 shows the disassembled machine code for the procedure.

The jump table resides in a different area of memory. We can see from the indirect jump on line 5 that the jump table begins at address `0x4006f8`. Using the GDB debugger, we can examine the six 8-byte words of memory comprising the jump table with the command `x/6gx 0x4006f8`. GDB prints the following:

```
(gdb) x/6gx 0x4006f8
0x4006f8:    0x00000000004005a1    0x00000000004005c3
0x400708:    0x00000000004005a1    0x00000000004005aa
0x400718:    0x00000000004005b2    0x00000000004005bf
```

Fill in the body of the `switch` statement with C code that will have the same behavior as the machine code.


```
     long switch_prob(long x, long n)
     x in %rdi, n in %rsi
1    0000000000400590 <switch_prob>:
2      400590:  48 83 ee 3c             sub    $0x3c,%rsi
3      400594:  48 83 fe 05             cmp    $0x5,%rsi
4      400598:  77 29                   ja     4005c3 <switch_prob+0x33>
5      40059a:  ff 24 f5 f8 06 40 00    jmpq   *0x4006f8(,%rsi,8)
6      4005a1:  48 8d 04 fd 00 00 00    lea    0x0(,%rdi,8),%rax
7      4005a8:  00
8      4005a9:  c3                      retq
9      4005aa:  48 89 f8                mov    %rdi,%rax
10     4005ad:  48 c1 f8 03             sar    $0x3,%rax
11     4005b1:  c3                      retq
12     4005b2:  48 89 f8                mov    %rdi,%rax
13     4005b5:  48 c1 e0 04             shl    $0x4,%rax
14     4005b9:  48 29 f8                sub    %rdi,%rax
15     4005bc:  48 89 c7                mov    %rax,%rdi
16     4005bf:  48 0f af ff             imul   %rdi,%rdi
17     4005c3:  48 8d 47 4b             lea    0x4b(%rdi),%rax
18     4005c7:  c3                      retq
```

Figure 3.53 Disassembled code for Problem 3.63.