is compiled into assembly code. The body of the code is as follows:

```
  xp at %ebp+8, yp at %ebp+12, zp at %ebp+16
    movl    8(%ebp), %edi
    movl    12(%ebp), %edx
    movl    16(%ebp), %ecx
    movl    (%edx), %ebx
    movl    (%ecx), %esi
    movl    (%edi), %eax
    movl    %eax, (%edx)
    movl    %ebx, (%ecx)
    movl    %esi, (%edi)
```

Parameters `xp`, `yp`, and `zp` are stored at memory locations with offsets 8, 12, and 16, respectively, relative to the address in register `%ebp`.

Write C code for `decode1` that will have an effect equivalent to the assembly code above.

3.5 Arithmetic and Logical Operations

Figure 3.7 lists some of the integer and logic operations. Most of the operations are given as instruction classes, as they can have different variants with different operand sizes. (Only `leal` has no other size variants.) For example, the instruction class ADD consists of three addition instructions: `addb`, `addw`, and `addl`, adding bytes, words, and double words, respectively. Indeed, each of the instruction classes shown has instructions for operating on byte, word, and double-word data. The operations are divided into four groups: load effective address, unary, binary, and shifts. *Binary* operations have two operands, while *unary* operations have one operand. These operands are specified using the same notation as described in Section 3.4.

3.5.1 Load Effective Address

The *load effective address* instruction `leal` is actually a variant of the `movl` instruction. It has the form of an instruction that reads from memory to a register, 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 3.7 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 `%edx` contains value x, then the instruction `leal 7(%edx,%edx,4), %eax` will set register `%eax` to $5x + 7$. Compilers often find clever uses of `leal` that have nothing to do with effective address computations. The destination operand must be a register.

Instruction		Effect	Description
`leal`	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	$D \leftarrow D \text{ ^ } S$	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	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Figure 3.7 **Integer arithmetic operations.** The load effective address (`leal`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $>>_A$ and $>>_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Practice Problem 3.6

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

Instruction	Result
`leal 6(%eax), %edx`	________
`leal (%eax,%ecx), %edx`	________
`leal (%eax,%ecx,4), %edx`	________
`leal 7(%eax,%eax,8), %edx`	________
`leal 0xA(,%ecx,4), %edx`	________
`leal 9(%eax,%ecx,2), %edx`	________

3.5.2 Unary and Binary Operations

Operations in the second group are unary operations, with the single operand serving as both source and destination. This operand can be either a register or

```
    } else if (__________)
        val = __________;
    return val;
}
```

GCC, with the command-line setting '-march=i686', generates the following assembly code:

```
  x at %ebp+8, y at %ebp+12
   movl    8(%ebp), %ebx
   movl    12(%ebp), %ecx
   testl   %ecx, %ecx
   jle     .L2
   movl    %ebx, %edx
   subl    %ecx, %edx
   movl    %ecx, %eax
   xorl    %ebx, %eax
   cmpl    %ecx, %ebx
   cmovl   %edx, %eax
   jmp     .L4
 .L2:
   leal    0(,%ebx,4), %edx
   leal    (%ecx,%ebx), %eax
   cmpl    $-2, %ecx
   cmovge  %edx, %eax
 .L4:
```

Fill in the missing expressions in the C code.

3.6.7 Switch Statements

A `switch` statement provides a multi-way 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, 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 3.18(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

(a) Switch statement

```
int switch_eg(int x, int n) {
    int result = x;

    switch (n) {

    case 100:
        result *= 13;
        break;

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

    case 103:
        result += 11;
        break;

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

    default:
        result = 0;
    }

    return result;
}
```

(b) Translation into extended C

```
int switch_eg_impl(int x, int n) {
    /* Table of code pointers */
    static void *jt[7] = {
        &&loc_A, &&loc_def, &&loc_B,
        &&loc_C, &&loc_D, &&loc_def,
        &&loc_D
    };

    unsigned index = n - 100;
    int result;

    if (index > 6)
        goto loc_def;

    /* Multiway branch */
    goto *jt[index];

 loc_def:  /* Default case*/
    result = 0;
    goto done;

 loc_C:    /* Case 103 */
    result = x;
    goto rest;

 loc_A:    /* Case 100 */
    result = x * 13;
    goto done;

 loc_B:    /* Case 102 */
    result = x + 10;
    /* Fall through */

 rest:     /* Finish case 103 */
    result += 11;
    goto done;

 loc_D:    /* Cases 104, 106 */
    result = x * x;
    /* Fall through */

 done:
    return result;
}
```

Figure 3.18 **Switch statement example with 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.

```
   x at %ebp+8, n at %ebp+12
   movl    8(%ebp), %edx            Get x
   movl    12(%ebp), %eax           Get n
   Set up jump table access
   subl    $100, %eax               Compute index = n-100
   cmpl    $6, %eax                 Compare index:6
   ja      .L2                      If >, goto loc_def
   jmp     *.L7(,%eax,4)            Goto *jt[index]
   Default case
 .L2:                             loc_def:
   movl    $0, %eax                   result = 0;
   jmp     .L8                        Goto done
   Case 103
 .L5:                             loc_C:
   movl    %edx, %eax                 result = x;
   jmp     .L9                        Goto rest
   Case 100
 .L3:                             loc_A:
   leal    (%edx,%edx,2), %eax        result = x*3;
   leal    (%edx,%eax,4), %eax        result = x+4*result
   jmp     .L8                        Goto done
   Case 102
 .L4:                             loc_B:
   leal    10(%edx), %eax             result = x+10
   Fall through
 .L9:                             rest:
   addl    $11, %eax                  result += 11;
   jmp     .L8                        Goto done
   Cases 104, 106
 .L6:                             loc_D
   movl    %edx, %eax                 result = x
   imull   %edx, %eax                 result *= x
   Fall through
 .L8:                             done:
   Return result
```

Figure 3.19 **Assembly code for `switch` statement example in Figure 3.18.**

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 3.19 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 3.18(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 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 `int`. 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` (`.L4`), `loc_C` (`.L5`), `loc_D` (`.L6`), and `loc_def` (`.L2`), 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 6, where the `jmp` instruction's operand is prefixed with '`*`', indicating 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 `index`, corresponding to values 100–106 of `n`. Observe 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:

```
  .section        .rodata
  .align 4         Align address to multiple of 4
.L7:
  .long   .L3      Case 100: loc_A
  .long   .L2      Case 101: loc_def
  .long   .L4      Case 102: loc_B
  .long   .L5      Case 103: loc_C
```

```
8     .long   .L6      Case 104: loc_D
9     .long   .L2      Case 105: loc_def
10    .long   .L6      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 "long" (4-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 `.L7` marks the start of this allocation. The address associated with this label serves as the base for the indirect jump (line 6).

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 `result` and then go to the end of the function. Similarly, the assembly-code blocks compute a value for register `%eax` and jump to the position indicated by label `.L8` at the end of the function. Only the code for case labels 102 and 103 do not follow this pattern, to account for the way that case 102 falls through to 103 in the original C code. This is handled in the assembly code and `switch_eg_impl` by having separate destinations for the two cases (`loc_C` and `loc_B` in C, `.L5` and `.L4` in assembly), where both of these blocks then converge on code that increments `result` by 11 (labeled `rest` in C and `.L9` in assembly).

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 3.28

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.

```
int switch2(int x) {
    int result = 0;
    switch (x) {
        /* Body of switch statement omitted */
    }
    return result;
}
```

In compiling the function, GCC generates the assembly code that follows for the initial part of the procedure and for the jump table. Variable `x` is initially at offset 8 relative to register `%ebp`.

```
  x at %ebp+8
    movl    8(%ebp), %eax
  Set up jump table access
    addl    $2, %eax
    cmpl    $6, %eax
    ja      .L2
    jmp     *.L8(,%eax,4)
```

```
  Jump table for switch2
  .L8:
    .long   .L3
    .long   .L2
    .long   .L4
    .long   .L5
    .long   .L6
    .long   .L6
    .long   .L7
```

Based on this information, answer the following questions:

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

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

Practice Problem 3.29

For a C function `switcher` with the general structure

```
int switcher(int a, int b, int c)
{
    int answer;
    switch(a) {
    case ________:          /* Case A */
        c = ________;
        /* Fall through */
    case ________:          /* Case B */
        answer = ________;
        break;
    case ________:          /* Case C */
    case ________:          /* Case D */
        answer = ________;
        break;
    case ________:          /* Case E */
        answer = ________;
        break;
    default:
        answer = ________;
    }
    return answer;
}
```

GCC generates the assembly code and jump table shown in Figure 3.20.

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 at %ebp+8, b at %ebp+12, c at %ebp+16
  movl    8(%ebp), %eax
  cmpl    $7, %eax
  ja      .L2
  jmp     *.L7(,%eax,4)
.L2:
  movl    12(%ebp), %eax
  jmp     .L8
.L5:
  movl    $4, %eax
  jmp     .L8
.L6:
  movl    12(%ebp), %eax
  xorl    $15, %eax
  movl    %eax, 16(%ebp)
.L3:
  movl    16(%ebp), %eax
  addl    $112, %eax
  jmp     .L8
.L4:
  movl    16(%ebp), %eax
  addl    12(%ebp), %eax
  sall    $2, %eax
.L8:
```

```
.L7:
  .long   .L3
  .long   .L2
  .long   .L4
  .long   .L2
  .long   .L5
  .long   .L6
  .long   .L2
  .long   .L4
```

Figure 3.20 **Assembly code and jump table for Problem 3.29.**

3.7 Procedures

A procedure call involves passing both data (in the form of procedure parameters and return values) and control from one part of a program to another. In addition, it must allocate space for the local variables of the procedure on entry and deallocate them on exit. Most machines, including IA32, provide only simple instructions for transferring control to and from procedures. The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.

3.7.1 Stack Frame Structure

IA32 programs make use of the program stack to support procedure calls. The machine uses the stack to pass procedure arguments, to store return information, to save registers for later restoration, and for local storage. The portion of the stack allocated for a single procedure call is called a *stack frame*. Figure 3.21 diagrams the general structure of a stack frame. The topmost stack frame is delimited by two pointers, with register `%ebp` serving as the *frame pointer*, and register `%esp`

```
Arguments: p1 at %ebp+8, p2 at %ebp+12, action at %ebp+16
Registers: result in %edx (initialized to -1)
The jump targets:
.L17:                                MODE_E
  movl    $17, %edx
  jmp     .L19
.L13:                                MODE_A
  movl    8(%ebp), %eax
  movl    (%eax), %edx
  movl    12(%ebp), %ecx
  movl    (%ecx), %eax
  movl    8(%ebp), %ecx
  movl    %eax, (%ecx)
  jmp     .L19
.L14:                                MODE_B
  movl    12(%ebp), %edx
  movl    (%edx), %eax
  movl    %eax, %edx
  movl    8(%ebp), %ecx
  addl    (%ecx), %edx
  movl    12(%ebp), %eax
  movl    %edx, (%eax)
  jmp     .L19
.L15:                                MODE_C
  movl    12(%ebp), %edx
  movl    $15, (%edx)
  movl    8(%ebp), %ecx
  movl    (%ecx), %edx
  jmp     .L19
.L16:                                MODE_D
  movl    8(%ebp), %edx
  movl    (%edx), %eax
  movl    12(%ebp), %ecx
  movl    %eax, (%ecx)
  movl    $17, %edx
.L19:                                default
  movl    %edx, %eax                 Set return value
```

Figure 3.43 **Assembly code for Problem 3.58.** This code implements the different branches of a `switch` statement.

3.59 ◆◆

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

```
int switch_prob(int x, int n)
{
    int result = x;

    switch(n) {

        /* Fill in code here */
    }

    return result;
}
```

Figure 3.44 shows the disassembled machine code for the procedure. We can see in lines 4 and 5 that parameters x and n are loaded into registers %eax and %edx, respectively.

The jump table resides in a different area of memory. We can see from the indirect jump on line 9 that the jump table begins at address 0x80485d0. Using the GDB debugger, we can examine the six 4-byte words of memory comprising the jump table with the command x/6w 0x80485d0. GDB prints the following:

```
(gdb) x/6w 0x80485d0
0x80485d0: 0x08048438 0x08048448 0x08048438 0x0804843d
0x80485e0: 0x08048442 0x08048445
```

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

```
1   08048420 <switch_prob>:
2    8048420:  55                      push   %ebp
3    8048421:  89 e5                   mov    %esp,%ebp
4    8048423:  8b 45 08                mov    0x8(%ebp),%eax
5    8048426:  8b 55 0c                mov    0xc(%ebp),%edx
6    8048429:  83 ea 32                sub    $0x32,%edx
7    804842c:  83 fa 05                cmp    $0x5,%edx
8    804842f:  77 17                   ja     8048448 <switch_prob+0x28>
9    8048431:  ff 24 95 d0 85 04 08    jmp    *0x80485d0(,%edx,4)
10   8048438:  c1 e0 02                shl    $0x2,%eax
11   804843b:  eb 0e                   jmp    804844b <switch_prob+0x2b>
12   804843d:  c1 f8 02                sar    $0x2,%eax
13   8048440:  eb 09                   jmp    804844b <switch_prob+0x2b>
14   8048442:  8d 04 40                lea    (%eax,%eax,2),%eax
15   8048445:  0f af c0                imul   %eax,%eax
16   8048448:  83 c0 0a                add    $0xa,%eax
17   804844b:  5d                      pop    %ebp
18   804844c:  c3                      ret
```

Figure 3.44 **Disassembled code for Problem 3.59.**