This compiles to the following assembly code:

```
   void remdiv(long x, long y, long *qp, long *rp)
   x in %rdi, y in %rsi, qp in %rdx, rp in %rcx
1  remdiv:
2    movq    %rdx, %r8        Copy qp
3    movq    %rdi, %rax       Move x to lower 8 bytes of dividend
4    cqto                     Sign-extend to upper 8 bytes of dividend
5    idivq   %rsi             Divide by y
6    movq    %rax, (%r8)      Store quotient at qp
7    movq    %rdx, (%rcx)     Store remainder at rp
8    ret
```

In this code, argument rp must first be saved in a different register (line 2), since argument register %rdx is required for the division operation. Lines 3–4 then prepare the dividend by copying and sign-extending x. Following the division, the quotient in register %rax gets stored at qp (line 6), while the remainder in register %rdx gets stored at rp (line 7).

Unsigned division makes use of the divq instruction. Typically, register %rdx is set to zero beforehand.

Practice Problem 3.12 (solution page 365)

Consider the following function for computing the quotient and remainder of two unsigned 64-bit numbers:

```
void uremdiv(unsigned long x, unsigned long y,
             unsigned long *qp, unsigned long *rp) {
    unsigned long q = x/y;
    unsigned long r = x%y;
    *qp = q;
    *rp = r;
}
```

Modify the assembly code shown for signed division to implement this function.

3.6 Control

So far, we have only considered the behavior of *straight-line* code, where instructions follow one another in sequence. Some constructs in C, such as conditionals, loops, and switches, require conditional execution, where the sequence of operations that get performed depends on the outcomes of tests applied to the data. Machine code provides two basic low-level mechanisms for implementing conditional behavior: it tests data values and then alters either the control flow or the data flow based on the results of these tests.

Data-dependent control flow is the more general and more common approach for implementing conditional behavior, and so we will examine this first. Normally, both statements in C and instructions in machine code are executed *sequentially*, in the order they appear in the program. The execution order of a set of machine-code instructions can be altered with a *jump* instruction, indicating that control should pass to some other part of the program, possibly contingent on the result of some test. The compiler must generate instruction sequences that build upon this low-level mechanism to implement the control constructs of C.

In our presentation, we first cover the two ways of implementing conditional operations. We then describe methods for presenting loops and `switch` statements.

3.6.1 Condition Codes

In addition to the integer registers, the CPU maintains a set of single-bit *condition code* registers describing attributes of the most recent arithmetic or logical operation. These registers can then be tested to perform conditional branches. These condition codes are the most useful:

`CF`: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

`ZF`: Zero flag. The most recent operation yielded zero.

`SF`: Sign flag. The most recent operation yielded a negative value.

`OF`: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

For example, suppose we used one of the ADD instructions to perform the equivalent of the C assignment `t = a+b`, where variables `a`, `b`, and `t` are integers. Then the condition codes would be set according to the following C expressions:

`CF`	`(unsigned) t < (unsigned) a`	Unsigned overflow
`ZF`	`(t == 0)`	Zero
`SF`	`(t < 0)`	Negative
`OF`	`(a < 0 == b < 0) && (t < 0 != a < 0)`	Signed overflow

The `leaq` instruction does not alter any condition codes, since it is intended to be used in address computations. Otherwise, all of the instructions listed in Figure 3.10 cause the condition codes to be set. For the logical operations, such as XOR, the carry and overflow flags are set to zero. For the shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to zero. For reasons that we will not delve into, the INC and DEC instructions set the overflow and zero flags, but they leave the carry flag unchanged.

In addition to the setting of condition codes by the instructions of Figure 3.10, there are two instruction classes (having 8-, 16-, 32-, and 64-bit forms) that set condition codes without altering any other registers; these are listed in Figure 3.13. The CMP instructions set the condition codes according to the differences of their two operands. They behave in the same way as the SUB instructions, except that they set the condition codes without updating their destinations. With ATT format,

Instruction		Based on	Description
CMP	S_1, S_2	$S_2 - S_1$	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word
TEST	S_1, S_2	S_1 & S_2	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

Figure 3.13 Comparison and test instructions. These instructions set the condition codes without updating any other registers.

the operands are listed in reverse order, making the code difficult to read. These instructions set the zero flag if the two operands are equal. The other flags can be used to determine ordering relations between the two operands. The TEST instructions behave in the same manner as the AND instructions, except that they set the condition codes without altering their destinations. Typically, the same operand is repeated (e.g., `testq %rax,%rax` to see whether `%rax` is negative, zero, or positive), or one of the operands is a mask indicating which bits should be tested.

3.6.2 Accessing the Condition Codes

Rather than reading the condition codes directly, there are three common ways of using the condition codes: (1) we can set a single byte to 0 or 1 depending on some combination of the condition codes, (2) we can conditionally jump to some other part of the program, or (3) we can conditionally transfer data. For the first case, the instructions described in Figure 3.14 set a single byte to 0 or to 1 depending on some combination of the condition codes. We refer to this entire class of instructions as the SET instructions; they differ from one another based on which combinations of condition codes they consider, as indicated by the different suffixes for the instruction names. It is important to recognize that the suffixes for these instructions denote different conditions and not different operand sizes. For example, instructions `setl` and `setb` denote "set less" and "set below," not "set long word" or "set byte."

A SET instruction has either one of the low-order single-byte register elements (Figure 3.2) or a single-byte memory location as its destination, setting this byte to either 0 or 1. To generate a 32-bit or 64-bit result, we must also clear the high-order bits. A typical instruction sequence to compute the C expression `a < b`, where `a` and `b` are both of type `long`, proceeds as follows:

Instruction		Synonym	Effect	Set condition
`sete`	D	`setz`	$D \leftarrow$ `ZF`	Equal / zero
`setne`	D	`setnz`	$D \leftarrow$ `~ ZF`	Not equal / not zero
`sets`	D		$D \leftarrow$ `SF`	Negative
`setns`	D		$D \leftarrow$ `~ SF`	Nonnegative
`setg`	D	`setnle`	$D \leftarrow$ `~ (SF ^ OF) & ~ZF`	Greater (signed >)
`setge`	D	`setnl`	$D \leftarrow$ `~ (SF ^ OF)`	Greater or equal (signed >=)
`setl`	D	`setnge`	$D \leftarrow$ `SF ^ OF`	Less (signed <)
`setle`	D	`setng`	$D \leftarrow$ `(SF ^ OF) | ZF`	Less or equal (signed <=)
`seta`	D	`setnbe`	$D \leftarrow$ `~ CF & ~ZF`	Above (unsigned >)
`setae`	D	`setnb`	$D \leftarrow$ `~ CF`	Above or equal (unsigned >=)
`setb`	D	`setnae`	$D \leftarrow$ `CF`	Below (unsigned <)
`setbe`	D	`setna`	$D \leftarrow$ `CF | ZF`	Below or equal (unsigned <=)

Figure 3.14 The SET instructions. Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," that is, alternate names for the same machine instruction.

```
  int comp(data_t a, data_t b)
  a in %rdi, b in %rsi
1 comp:
2   cmpq    %rsi, %rdi      Compare a:b
3   setl    %al             Set low-order byte of %eax to 0 or 1
4   movzbl  %al, %eax       Clear rest of %eax (and rest of %rax)
5   ret
```

Note the comparison order of the `cmpq` instruction (line 2). Although the arguments are listed in the order `%rsi` (`b`), then `%rdi` (`a`), the comparison is really between `a` and `b`. Recall also, as discussed in Section 3.4.2, that the `movzbl` instruction (line 4) clears not just the high-order 3 bytes of `%eax`, but the upper 4 bytes of the entire register, `%rax`, as well.

For some of the underlying machine instructions, there are multiple possible names, which we list as "synonyms." For example, both `setg` (for "set greater") and `setnle` (for "set not less or equal") refer to the same machine instruction. Compilers and disassemblers make arbitrary choices of which names to use.

Although all arithmetic and logical operations set the condition codes, the descriptions of the different SET instructions apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation `t = a-b`. More specifically, let a, b, and t be the integers represented in two's-complement form by variables `a`, `b`, and `t`, respectively, and so $t = a -^{t}_{w} b$, where w depends on the sizes associated with `a` and `b`.

Consider the `sete`, or "set when equal," instruction. When $a = b$, we will have $t = 0$, and hence the zero flag indicates equality. Similarly, consider testing for signed comparison with the `setl`, or "set when less," instruction. When no overflow occurs (indicated by having `OF` set to 0), we will have $a < b$ when $a -^{t}_{w} b < 0$, indicated by having `SF` set to 1, and $a \geq b$ when $a -^{t}_{w} b \geq 0$, indicated by having `SF` set to 0. On the other hand, when overflow occurs, we will have $a < b$ when $a -^{t}_{w} b > 0$ (negative overflow) and $a > b$ when $a -^{t}_{w} b < 0$ (positive overflow). We cannot have overflow when $a = b$. Thus, when `OF` is set to 1, we will have $a < b$ if and only if `SF` is set to 0. Combining these cases, the EXCLUSIVE-OR of the overflow and sign bits provides a test for whether $a < b$. The other signed comparison tests are based on other combinations of `SF ^ OF` and `ZF`.

For the testing of unsigned comparisons, we now let a and b be the integers represented in unsigned form by variables `a` and `b`. In performing the computation `t = a-b`, the carry flag will be set by the CMP instruction when $a - b < 0$, and so the unsigned comparisons use combinations of the carry and zero flags.

It is important to note how machine code does or does not distinguish between signed and unsigned values. Unlike in C, it does not associate a data type with each program value. Instead, it mostly uses the same instructions for the two cases, because many arithmetic operations have the same bit-level behavior for unsigned and two's-complement arithmetic. Some circumstances require different instructions to handle signed and unsigned operations, such as using different versions of right shifts, division and multiplication instructions, and different combinations of condition codes.

Practice Problem 3.13 (solution page 366)

The C code

```
int comp(data_t a, data_t b) {
    return a COMP b;
}
```

shows a general comparison between arguments `a` and `b`, where `data_t`, the data type of the arguments, is defined (via `typedef`) to be one of the integer data types listed in Figure 3.1 and either signed or unsigned. The comparison `COMP` is defined via `#define`.

Suppose `a` is in some portion of `%rdx` while `b` is in some portion of `%rsi`. For each of the following instruction sequences, determine which data types `data_t` and which comparisons `COMP` could cause the compiler to generate this code. (There can be multiple correct answers; you should list them all.)

A.
```
cmpl    %esi, %edi
setl    %al
```

B.
```
cmpw    %si, %di
setge   %al
```

C.
```
cmpb    %sil, %dil
setbe    %al
```

D.
```
cmpq    %rsi, %rdi
setne   %a
```

Practice Problem 3.14 (solution page 366)

The C code

```
int test(data_t a) {
    return a TEST 0;
}
```

shows a general comparison between argument `a` and 0, where we can set the data type of the argument by declaring `data_t` with a `typedef`, and the nature of the comparison by declaring `TEST` with a `#define` declaration. The following instruction sequences implement the comparison, where `a` is held in some portion of register `%rdi`. For each sequence, determine which data types `data_t` and which comparisons `TEST` could cause the compiler to generate this code. (There can be multiple correct answers; list all correct ones.)

A.
```
testq  %rdi, %rdi
setge  %al
```

B.
```
testw   %di, %di
sete    %al
```

C.
```
testb    %dil, %dil
seta     %al
```

D.
```
testl    %edi, %edi
setle    %al
```

3.6.3 Jump Instructions

Under normal execution, instructions follow each other in the order they are listed. A *jump* instruction can cause the execution to switch to a completely new position in the program. These jump destinations are generally indicated in assembly code by a *label*. Consider the following (very contrived) assembly-code sequence:

```
  movq $0,%rax            Set %rax to 0
  jmp .L1                 Goto .L1
  movq (%rax),%rdx        Null pointer dereference (skipped)
.L1:
  popq %rdx               Jump target
```

Instruction	Synonym	Jump condition	Description
`jmp` *Label*		1	Direct jump
`jmp` **Operand*		1	Indirect jump
`je` *Label*	`jz`	`ZF`	Equal / zero
`jne` *Label*	`jnz`	`~ZF`	Not equal / not zero
`js` *Label*		`SF`	Negative
`jns` *Label*		`~SF`	Nonnegative
`jg` *Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge` *Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl` *Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle` *Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja` *Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae` *Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb` *Label*	`jnae`	`CF`	Below (unsigned <)
`jbe` *Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

The instruction `jmp .L1` will cause the program to skip over the `movq` instruction and instead resume execution with the `popq` instruction. In generating the object-code file, the assembler determines the addresses of all labeled instructions and encodes the *jump targets* (the addresses of the destination instructions) as part of the jump instructions.

Figure 3.15 shows the different jump instructions. The `jmp` instruction jumps unconditionally. It can be either a *direct* jump, where the jump target is encoded as part of the instruction, or an *indirect* jump, where the jump target is read from a register or a memory location. Direct jumps are written in assembly code by giving a label as the jump target, for example, the label `.L1` in the code shown. Indirect jumps are written using '*' followed by an operand specifier using one of the memory operand formats described in Figure 3.3. As examples, the instruction

```
jmp *%rax
```

uses the value in register `%rax` as the jump target, and the instruction

```
jmp *(%rax)
```

reads the jump target from memory, using the value in `%rax` as the read address.

The remaining jump instructions in the table are *conditional*—they either jump or continue executing at the next instruction in the code sequence, depending on some combination of the condition codes. The names of these instructions and the conditions under which they jump match those of the SET instructions (see Figure 3.14). As with the SET instructions, some of the underlying machine instructions have multiple names. Conditional jumps can only be direct.

3.6.4 Jump Instruction Encodings

For the most part, we will not concern ourselves with the detailed format of machine code. On the other hand, understanding how the targets of jump instructions are encoded will become important when we study linking in Chapter 7. In addition, it helps when interpreting the output of a disassembler. In assembly code, jump targets are written using symbolic labels. The assembler, and later the linker, generate the proper encodings of the jump targets. There are several different encodings for jumps, but some of the most commonly used ones are *PC relative*. That is, they encode the difference between the address of the target instruction and the address of the instruction immediately following the jump. These offsets can be encoded using 1, 2, or 4 bytes. A second encoding method is to give an "absolute" address, using 4 bytes to directly specify the target. The assembler and linker select the appropriate encodings of the jump destinations.

As an example of PC-relative addressing, the following assembly code for a function was generated by compiling a file `branch.c`. It contains two jumps: the `jmp` instruction on line 2 jumps forward to a higher address, while the `jg` instruction on line 7 jumps back to a lower one.

```
1    movq    %rdi, %rax
2    jmp     .L2
3  .L3:
4    sarq    %rax
5  .L2:
6    testq   %rax, %rax
7    jg      .L3
8    rep; ret
```

The disassembled version of the `.o` format generated by the assembler is as follows:

```
1     0:   48 89 f8              mov    %rdi,%rax
2     3:   eb 03                 jmp    8 <loop+0x8>
3     5:   48 d1 f8              sar    %rax
4     8:   48 85 c0              test   %rax,%rax
5     b:   7f f8                 jg     5 <loop+0x5>
6     d:   f3 c3                 repz retq
```

In the annotations on the right generated by the disassembler, the jump targets are indicated as `0x8` for the jump instruction on line 2 and `0x5` for the jump instruction on line 5 (the disassembler lists all numbers in hexadecimal). Looking at the byte encodings of the instructions, however, we see that the target of the first jump instruction is encoded (in the second byte) as `0x03`. Adding this to `0x5`, the

Aside What do the instructions `rep` and `repz` do?

Line 8 of the assembly code shown on page 243 contains the instruction combination `rep; ret`. These are rendered in the disassembled code (line 6) as `repz retq`. One can infer that `repz` is a synonym for `rep`, just as `retq` is a synonym for `ret`. Looking at the Intel and AMD documentation for the `rep` instruction, we find that it is normally used to implement a repeating string operation [3, 51]. It seems completely inappropriate here. The answer to this puzzle can be seen in AMD's guidelines to compiler writers [1]. They recommend using the combination of `rep` followed by `ret` to avoid making the `ret` instruction the destination of a conditional jump instruction. Without the `rep` instruction, the `jg` instruction (line 7 of the assembly code) would proceed to the `ret` instruction when the branch is not taken. According to AMD, their processors cannot properly predict the destination of a `ret` instruction when it is reached from a jump instruction. The `rep` instruction serves as a form of no-operation here, and so inserting it as the jump destination does not change behavior of the code, except to make it faster on AMD processors. We can safely ignore any `rep` or `repz` instruction we see in the rest of the code presented in this book.

address of the following instruction, we get jump target address `0x8`, the address of the instruction on line 4.

Similarly, the target of the second jump instruction is encoded as `0xf8` (decimal −8) using a single-byte two's-complement representation. Adding this to `0xd` (decimal 13), the address of the instruction on line 6, we get `0x5`, the address of the instruction on line 3.

As these examples illustrate, the value of the program counter when performing PC-relative addressing is the address of the instruction following the jump, not that of the jump itself. This convention dates back to early implementations, when the processor would update the program counter as its first step in executing an instruction.

The following shows the disassembled version of the program after linking:

```
1   4004d0:  48 89 f8          mov    %rdi,%rax
2   4004d3:  eb 03             jmp    4004d8 <loop+0x8>
3   4004d5:  48 d1 f8          sar    %rax
4   4004d8:  48 85 c0          test   %rax,%rax
5   4004db:  7f f8             jg     4004d5 <loop+0x5>
6   4004dd:  f3 c3             repz retq
```

The instructions have been relocated to different addresses, but the encodings of the jump targets in lines 2 and 5 remain unchanged. By using a PC-relative encoding of the jump targets, the instructions can be compactly encoded (requiring just 2 bytes), and the object code can be shifted to different positions in memory without alteration.

Practice Problem 3.15 (solution page 366)

In the following excerpts from a disassembled binary, some of the information has been replaced by X's. Answer the following questions about these instructions.

A. What is the target of the je instruction below? (You do not need to know anything about the callq instruction here.)

```
4003fa: 74 02                    je     XXXXXX
4003fc: ff d0                    callq  *%rax
```

B. What is the target of the je instruction below?

```
40042f: 74 f4                    je     XXXXXX
400431: 5d                       pop    %rbp
```

C. What is the address of the ja and pop instructions?

```
XXXXXX: 77 02                    ja     400547
XXXXXX: 5d                       pop    %rbp
```

D. In the code that follows, the jump target is encoded in PC-relative form as a 4-byte two's-complement number. The bytes are listed from least significant to most, reflecting the little-endian byte ordering of x86-64. What is the address of the jump target?

```
4005e8: e9 73 ff ff ff           jmpq   XXXXXXX
4005ed: 90                       nop
```

The jump instructions provide a means to implement conditional execution (if), as well as several different loop constructs.

3.6.5 Implementing Conditional Branches with Conditional Control

The most general way to translate conditional expressions and statements from C into machine code is to use combinations of conditional and unconditional jumps. (As an alternative, we will see in Section 3.6.6 that some conditionals can be implemented by conditional transfers of data rather than control.) For example, Figure 3.16(a) shows the C code for a function that computes the absolute value of the difference of two numbers.[3] The function also has a side effect of incrementing one of two counters, encoded as global variables `lt_cnt` and `ge_cnt`. Gcc generates the assembly code shown as Figure 3.16(c). Our rendition of the machine code into C is shown as the function `gotodiff_se` (Figure 3.16(b)). It uses the `goto` statement in C, which is similar to the unconditional jump of

3. Actually, it can return a negative value if one of the subtractions overflows. Our interest here is to demonstrate machine code, not to implement robust code.

(a) Original C code

```
long lt_cnt = 0;
long ge_cnt = 0;

long absdiff_se(long x, long y)
{
    long result;
    if (x < y) {
        lt_cnt++;
        result = y - x;
    }
    else {
        ge_cnt++;
        result = x - y;
    }
    return result;
}
```

(b) Equivalent goto version

```
1   long gotodiff_se(long x, long y)
2   {
3       long result;
4       if (x >= y)
5           goto x_ge_y;
6       lt_cnt++;
7       result =  y - x;
8       return result;
9    x_ge_y:
10      ge_cnt++;
11      result = x - y;
12      return result;
13  }
```

(c) Generated assembly code

```
    long absdiff_se(long x, long y)
    x in %rdi, y in %rsi
absdiff_se:
  cmpq    %rsi, %rdi          Compare x:y
  jge     .L2                 If >= goto x_ge_y
  addq    $1, lt_cnt(%rip)    lt_cnt++
  movq    %rsi, %rax
  subq    %rdi, %rax          result = y - x
  ret                         Return
.L2:                        x_ge_y:
  addq    $1, ge_cnt(%rip)    ge_cnt++
  movq    %rdi, %rax
  subq    %rsi, %rax          result = x - y
  ret                         Return
```

Figure 3.16 **Compilation of conditional statements.** (a) C procedure `absdiff_se` contains an if-else statement. The generated assembly code is shown (c), along with (b) a C procedure `gotodiff_se` that mimics the control flow of the assembly code.

assembly code. Using `goto` statements is generally considered a bad programming style, since their use can make code very difficult to read and debug. We use them in our presentation as a way to construct C programs that describe the control flow of machine code. We call this style of programming "goto code."

In the goto code (Figure 3.16(b)), the statement `goto x_ge_y` on line 5 causes a jump to the label `x_ge_y` (since it occurs when $x \geq y$) on line 9. Continuing the

Aside Describing machine code with C code

Figure 3.16 shows an example of how we will demonstrate the translation of C language control constructs into machine code. The figure contains an example C function (a) and an annotated version of the assembly code generated by GCC (c). It also contains a version in C that closely matches the structure of the assembly code (b). Although these versions were generated in the sequence (a), (c), and (b), we recommend that you read them in the order (a), (b), and then (c). That is, the C rendition of the machine code will help you understand the key points, and this can guide you in understanding the actual assembly code.

execution from this point, it completes the computations specified by the `else` portion of function `absdiff_se` and returns. On the other hand, if the test `x >= y` fails, the program procedure will carry out the steps specified by the `if` portion of `absdiff_se` and return.

The assembly-code implementation (Figure 3.16(c)) first compares the two operands (line 2), setting the condition codes. If the comparison result indicates that x is greater than or equal to y, it then jumps to a block of code starting at line 8 that increments global variable `ge_cnt`, computes `x-y` as the return value, and returns. Otherwise, it continues with the execution of code beginning at line 4 that increments global variable `lt_cnt`, computes `y-x` as the return value, and returns. We can see, then, that the control flow of the assembly code generated for `absdiff_se` closely follows the goto code of `gotodiff_se`.

The general form of an if-else statement in C is given by the template

```
if (test-expr)
    then-statement
else
    else-statement
```

where *test-expr* is an integer expression that evaluates either to zero (interpreted as meaning “false”) or to a nonzero value (interpreted as meaning “true”). Only one of the two branch statements (*then-statement* or *else-statement*) is executed.

For this general form, the assembly implementation typically adheres to the following form, where we use C syntax to describe the control flow:

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

That is, the compiler generates separate blocks of code for *then-statement* and *else-statement*. It inserts conditional and unconditional branches to make sure the correct block is executed.

Practice Problem 3.16 (solution page 367)

When given the C code

```
void cond(short a, short *p)
{
    if (a && *p < a)
        *p = a;
}
```

GCC generates the following assembly code:

```
  void cond(short a, short *p)
  a in %rdi, p in %rsi
cond:
  testq   %rdi, %rdi
  je      .L1
  cmpq    %rsi, (%rdi)
  jle     .L1
  movq    %rdi, (%rsi)
.L1:
  rep; ret
```

A. Write a goto version in C that performs the same computation and mimics the control flow of the assembly code, in the style shown in Figure 3.16(b). You might find it helpful to first annotate the assembly code as we have done in our examples.

B. Explain why the assembly code contains two conditional branches, even though the C code has only one `if` statement.

Practice Problem 3.17 (solution page 367)

An alternate rule for translating `if` statements into goto code is as follows:

```
    t = test-expr;
    if (t)
        goto true;
    else-statement
    goto done;
true:
    then-statement
done:
```

A. Rewrite the goto version of `absdiff_se` based on this alternate rule.

B. Can you think of any reasons for choosing one rule over the other?

Practice Problem 3.18 (solution page 368)

Starting with C code of the form

```
short test(short x, short y, short z) {
    short val = _________;
    if (_________) {
        if (_________)
            val = _________;
        else
            val = _________;
    } else if (_________)
        val = _________;
    return val;
}
```

GCC generates the following assembly code:

```
  short test(short x, short y, short z)
  x in %rdi, y in %rsi, z in %rdx
test:
  leaq    (%rdx,%rsi), %rax
  subq    %rdi, %rax
  cmpq    $5, %rdx
  jle     .L2
  cmpq    $2, %rsi
  jle     .L3
  movq    %rdi, %rax
  idivq   %rdx, %rax
  ret
.L3:
  movq    %rdi, %rax
  idivq   %rsi, %rax
  ret
.L2:
  cmpq    $3, %rdx
  jge     .L4
  movq    %rdx, %rax
  idivq   %rsi, %rax
.L4:
  rep; ret
```

Fill in the missing expressions 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

```
long cmovdiff(long x, long y)
{
    long rval = y-x;
    long eval = x-y;
    long ntest = x >= y;
    /* Line below requires
       single instruction: */
    if (ntest) rval = eval;
    return rval;
}
```

(c) Generated assembly code

```
  long absdiff(long x, long y)
  x in %rdi, y in %rsi
absdiff:
  movq    %rsi, %rax
  subq    %rdi, %rax      rval = y-x
  movq    %rdi, %rdx
  subq    %rsi, %rdx      eval = x-y
  cmpq    %rsi, %rdi      Compare x:y
  cmovge  %rdx, %rax      If >=, rval = eval
  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 368)

Running on a new processor model, our code required around 45 cycles when the branching pattern was random, and around 25 cycles when the pattern was highly predictable.

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.

Practice Problem 3.20 (solution page 369)

In the following C function, we have left the definition of operation `OP` incomplete:

```
#define OP __________ /* Unknown operator */

short arith(short x) {
    return x OP 16;
}
```

When compiled, GCC generates the following assembly code:

```
  short arith(short x)
  x in %rdi
arith:
  leaq    15(%rdi), %rbx
  testq   %rdi, %rdi
  cmovns  %rdi, %rbx
  sarq    $4, %rbx
  ret
```

A. What operation is `OP`?

B. Annotate the code to explain how it works.

Practice Problem 3.21 (solution page 369)

Starting with C code of the form

```
short test(short x, short y) {
    short val = __________;
    if (__________) {
        if (__________)
            val = __________;
        else
            val = __________;
    } else if (__________)
        val = __________;
    return val;
}
```

GCC generates the following assembly code:

```
  short test(short x, short y)
  x in %rdi, y in %rsi
test:
  leaq    12(%rsi), %rbx
  testq   %rdi, %rdi
  jge     .L2
```

```
  movq    %rdi, %rbx
  imulq   %rsi, %rbx
  movq    %rdi, %rdx
  orq     %rsi, %rdx
  cmpq    %rsi, %rdi
  cmovge  %rdx, %rbx
  ret
.L2:
  idivq   %rsi, %rdi
  cmpq    $10, %rsi
  cmovge  %rdi, %rbx
  ret
```

Fill in the missing expressions in the C code.

3.6.7 Loops

C provides several looping constructs—namely, `do-while`, `while`, and `for`. No corresponding instructions exist in machine code. Instead, combinations of conditional tests and jumps are used to implement the effect of loops. GCC and other compilers generate loop code based on the two basic loop patterns. We will study the translation of loops as a progression, starting with `do-while` and then working toward ones with more complex implementations, covering both patterns.

Do-While Loops

The general form of a `do-while` statement is as follows:

```
do
    body-statement
    while (test-expr);
```

The effect of the loop is to repeatedly execute *body-statement*, evaluate *test-expr*, and continue the loop if the evaluation result is nonzero. Observe that *body-statement* is executed at least once.

This general form can be translated into conditionals and `goto` statements as follows:

```
loop:
    body-statement
    t = test-expr;
    if (t)
       goto loop;
```

That is, on each iteration the program evaluates the body statement and then the test expression. If the test succeeds, the program goes back for another iteration.

(a) C code

```
long fact_do(long n)
{
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

(b) Equivalent goto version

```
long fact_do_goto(long n)
{
    long result = 1;
 loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
     long fact_do(long n)
     n in %rdi
fact_do:
  movl    $1, %eax          Set result = 1
.L2:                      loop:
  imulq   %rdi, %rax        Compute result *= n
  subq    $1, %rdi          Decrement n
  cmpq    $1, %rdi          Compare n:1
  jg      .L2               If >, goto loop
  rep; ret                  Return
```

Figure 3.19 **Code for `do-while` version of factorial program.** A conditional jump causes the program to loop.

As an example, Figure 3.19(a) shows an implementation of a routine to compute the factorial of its argument, written $n!$, with a `do-while` loop. This function only computes the proper value for $n > 0$.

Practice Problem 3.22 (solution page 369)

A. Try to calculate 14! with a 32-bit int. Verify whether the computation of 14! overflows.

B. What if the computation is done with a 64-bit long int?

The goto code shown in Figure 3.19(b) shows how the loop gets turned into a lower-level combination of tests and conditional jumps. Following the initialization of `result`, the program begins looping. First it executes the body of the loop, consisting here of updates to variables `result` and n. It then tests whether $n > 1$, and, if so, it jumps back to the beginning of the loop. Figure 3.19(c) shows

Aside Reverse engineering loops

A key to understanding how the generated assembly code relates to the original source code is to find a mapping between program values and registers. This task was simple enough for the loop of Figure 3.19, but it can be much more challenging for more complex programs. The C compiler will often rearrange the computations, so that some variables in the C code have no counterpart in the machine code, and new values are introduced into the machine code that do not exist in the source code. Moreover, it will often try to minimize register usage by mapping multiple program values onto a single register.

The process we described for `fact_do` works as a general strategy for reverse engineering loops. Look at how registers are initialized before the loop, updated and tested within the loop, and used after the loop. Each of these provides a clue that can be combined to solve a puzzle. Be prepared for surprising transformations, some of which are clearly cases where the compiler was able to optimize the code, and others where it is hard to explain why the compiler chose that particular strategy.

the assembly code from which the goto code was generated. The conditional jump instruction `jg` (line 7) is the key instruction in implementing a loop. It determines whether to continue iterating or to exit the loop.

Reverse engineering assembly code, such as that of Figure 3.19(c), requires determining which registers are used for which program values. In this case, the mapping is fairly simple to determine: We know that n will be passed to the function in register `%rdi`. We can see register `%rax` getting initialized to 1 (line 2). (Recall that, although the instruction has `%eax` as its destination, it will also set the upper 4 bytes of `%rax` to 0.) We can see that this register is also updated by multiplication on line 4. Furthermore, since `%rax` is used to return the function value, it is often chosen to hold program values that are returned. We therefore conclude that `%rax` corresponds to program value `result`.

Practice Problem 3.23 (solution page 370)

For the C code

```
short dw_loop(short x) {
    short y = x/9;
    short *p = &x;
    short n = 4*x;
    do {
        x += y;
        (*p) += 5;
        n -= 2;
    } while (n > 0);
    return x;
}
```

GCC generates the following assembly code:

```
short dw_loop(short x)
x initially in %rdi
dw_loop:
  movq    %rdi, %rbx
  movq    %rdi, %rcx
  idivq   $9, %rcx
  leaq    (,%rdi,4), %rdx
.L2:
  leaq    5(%rbx,%rcx), %rcx
  subq    $1, %rdx
  testq   %rdx, %rdx
  jg      .L2
  rep; ret
```

A. Which registers are used to hold program values `x`, `y`, and `n`?

B. How has the compiler eliminated the need for pointer variable `p` and the pointer dereferencing implied by the expression `(*p)+=5`?

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 3.19(c).

While Loops

The general form of a `while` statement is as follows:

```
while (test-expr)
    body-statement
```

It differs from `do-while` in that *test-expr* is evaluated and the loop is potentially terminated before the first execution of *body-statement*. There are a number of ways to translate a `while` loop into machine code, two of which are used in code generated by GCC. Both use the same loop structure as we saw for `do-while` loops but differ in how to implement the initial test.

The first translation method, which we refer to as *jump to middle*, performs the initial test by performing an unconditional jump to the test at the end of the loop. It can be expressed by the following template for translating from the general `while` loop form to goto code:

```
    goto test;
loop:
    body-statement
test:
    t = test-expr;
    if (t)
        goto loop;
```

As an example, Figure 3.20(a) shows an implementation of the factorial function using a `while` loop. This function correctly computes $0! = 1$. The adjacent

(a) C code

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

(b) Equivalent goto version

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
 loop:
    result *= n;
    n = n-1;
 test:
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
  long fact_while(long n)
  n in %rdi
fact_while:
  movl    $1, %eax        Set result = 1
  jmp     .L5             Goto test
.L6:                    loop:
  imulq   %rdi, %rax      Compute result *= n
  subq    $1, %rdi        Decrement n
.L5:                    test:
  cmpq    $1, %rdi        Compare n:1
  jg      .L6             If >, goto loop
  rep; ret                Return
```

Figure 3.20 C and assembly code for `while` version of factorial using jump-to-middle translation. The C function `fact_while_jm_goto` illustrates the operation of the assembly-code version.

function `fact_while_jm_goto` (Figure 3.20(b)) is a C rendition of the assembly code generated by GCC when optimization is specified with the command-line option `-Og`. Comparing the goto code generated for `fact_while` (Figure 3.20(b)) to that for `fact_do` (Figure 3.19(b)), we see that they are very similar, except that the statement `goto test` before the loop causes the program to first perform the test of `n` before modifying the values of `result` or `n`. The bottom portion of the figure (Figure 3.20(c)) shows the actual assembly code generated.

Practice Problem 3.24 (solution page 371)

For C code having the general form

```
short loop_while(short a, short b)
{
```

```
    short result = __________;
    while (__________) {
        result = __________;
        a = __________;
    }
    return result;
}
```

GCC, run with command-line option `-Og`, produces the following code:

```
     short loop_while(short a, short b)
     a in %rdi, b in %rsi
1    loop_while:
2      movl    $0, %eax
3      jmp     .L2
4    .L3:
5      leaq    (,%rsi,%rdi), %rdx
6      addq    %rdx, %rax
7      subq    $1, %rdi
8    .L2:
9      cmpq    %rsi, %rdi
10     jg      .L3
11     rep; ret
```

We can see that the compiler used a jump-to-middle translation, using the `jmp` instruction on line 3 to jump to the test starting with label `.L2`. Fill in the missing parts of the C code.

The second translation method, which we refer to as *guarded do*, first transforms the code into a `do-while` loop by using a conditional branch to skip over the loop if the initial test fails. GCC follows this strategy when compiling with higher levels of optimization, for example, with command-line option `-O1`. This method can be expressed by the following template for translating from the general `while` loop form to a `do-while` loop:

```
t = test-expr;
if (!t)
    goto done;
do
    body-statement
    while (test-expr);
done:
```

This, in turn, can be transformed into goto code as

```
    t = test-expr;
    if (!t)
        goto done;
```

```
loop:
    body-statement
    t = test-expr;
    if (t)
        goto loop;
done:
```

Using this implementation strategy, the compiler can often optimize the initial test, for example, determining that the test condition will always hold.

As an example, Figure 3.21 shows the same C code for a factorial function as in Figure 3.20, but demonstrates the compilation that occurs when GCC is given command-line option `-O1`. Figure 3.21(c) shows the actual assembly code generated, while Figure 3.21(b) renders this assembly code in a more readable C representation. Referring to this goto code, we see that the loop will be skipped if $n \leq 1$, for the initial value of n. The loop itself has the same general structure as that generated for the `do-while` version of the function (Figure 3.19). One interesting feature, however, is that the loop test (line 9 of the assembly code) has been changed from $n > 1$ in the original C code to $n \neq 1$. The compiler has determined that the loop can only be entered when $n > 1$, and that decrementing n will result in either $n > 1$ or $n = 1$. Therefore, the test $n \neq 1$ will be equivalent to the test $n \leq 1$.

Practice Problem 3.25 (solution page 371)

For C code having the general form

```
long loop_while2(long a, long b)
{
    long result = __________;
    while (__________) {
        result = __________;
        b = __________;
    }
    return result;
}
```

GCC, run with command-line option `-O1`, produces the following code:

```
    a in %rdi, b in %rsi
loop_while2:
  testq   %rsi, %rsi
  jle     .L8
  movq    %rsi, %rax
.L7:
  imulq   %rdi, %rax
  subq    %rdi, %rsi
  testq   %rsi, %rsi
```

(a) C code

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

(b) Equivalent goto version

```
long fact_while_gd_goto(long n)
{
    long result = 1;
    if (n <= 1)
        goto done;
 loop:
    result *= n;
    n = n-1;
    if (n != 1)
        goto loop;
 done:
    return result;
}
```

(c) Corresponding assembly-language code

```
     long fact_while(long n)
     n in %rdi
1    fact_while:
2      cmpq    $1, %rdi          Compare n:1
3      jle     .L7               If <=, goto done
4      movl    $1, %eax          Set result = 1
5    .L6:                      loop:
6      imulq   %rdi, %rax        Compute result *= n
7      subq    $1, %rdi          Decrement n
8      cmpq    $1, %rdi          Compare n:1
9      jne     .L6               If !=, goto loop
10     rep; ret                  Return
11   .L7:                      done:
12     movl    $1, %eax          Compute result = 1
13     ret                       Return
```

Figure 3.21 C and assembly code for `while` version of factorial using guarded-do translation. The `fact_while_gd_goto` function illustrates the operation of the assembly-code version.

```
9      jg      .L7
10     rep; ret
11   .L8:
12     movq    %rsi, %rax
13     ret
```

We can see that the compiler used a guarded-do translation, using the `jle` instruction on line 3 to skip over the loop code when the initial test fails. Fill in the missing parts of the C code. Note that the control structure in the assembly code does not exactly match what would be obtained by a direct translation of the C code according to our translation rules. In particular, it has two different `ret` instructions (lines 10 and 13). However, you can fill out the missing portions of the C code in a way that it will have equivalent behavior to the assembly code.

Practice Problem 3.26 (solution page 372)

A function `test_one` has the following overall structure:

```
short test_one(unsigned short x) {
    short val = 1;
    while ( ... ) {
        .
        .
        .
    }
    return ...;
}
```

The GCC C compiler generates the following assembly code:

```
     short test_one(unsigned short x)
     x in %rdi
test_one:
  movl    $1, %eax
  jmp     .L5
.L6:
  xorq    %rdi, %rax
  shrq    %rdi              Shift right by 1
.L5:
  testq   %rdi, %rdi
  jne     .L6
  andl    $0, %eax
  ret
```

Reverse engineer the operation of this code and then do the following:

A. Determine what loop translation method was used.

B. Use the assembly-code version to fill in the missing parts of the C code.

C. Describe in English what this function computes.

For Loops

The general form of a `for` loop is as follows:

`for (`*init-expr*`;` *test-expr*`;` *update-expr*`)`
    body-statement

The C language standard states (with one exception, highlighted in Problem 3.29) that the behavior of such a loop is identical to the following code using a `while` loop:

```
init-expr;
while (test-expr) {
    body-statement
    update-expr;
}
```

The program first evaluates the initialization expression *init-expr*. It enters a loop where it first evaluates the test condition *test-expr*, exiting if the test fails, then executes the body of the loop *body-statement*, and finally evaluates the update expression *update-expr*.

The code generated by GCC for a `for` loop then follows one of our two translation strategies for `while` loops, depending on the optimization level. That is, the jump-to-middle strategy yields the goto code

```
    init-expr;
    goto test;
loop:
    body-statement
    update-expr;
test:
    t = test-expr;
    if (t)
        goto loop;
```

while the guarded-do strategy yields

```
    init-expr;
    t = test-expr;
    if (!t)
        goto done;
loop:
    body-statement
    update-expr;
    t = test-expr;
    if (t)
        goto loop;
done:
```

As examples, consider a factorial function written with a `for` loop:

```
long fact_for(long n)
{
    long i;
    long result = 1;
```

```
    for (i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

As shown, the natural way of writing a factorial function with a `for` loop is to multiply factors from 2 up to n, and so this function is quite different from the code we showed using either a `while` or a `do-while` loop.

We can identify the different components of the `for` loop in this code as follows:

init-expr	`i = 2`
test-expr	`i <= n`
update-expr	`i++`
body-statement	`result *= i;`

Substituting these components into the template we have shown to transform a `for` loop into a `while` loop yields the following:

```
long fact_for_while(long n)
{
    long i = 2;
    long result = 1;
    while (i <= n) {
        result *= i;
        i++;
    }
    return result;
}
```

Applying the jump-to-middle transformation to the `while` loop then yields the following version in goto code:

```
long fact_for_jm_goto(long n)
{
    long i = 2;
    long result = 1;
    goto test;
 loop:
    result *= i;
    i++;
 test:
    if (i <= n)
        goto loop;
    return result;
}
```

Indeed, a close examination of the assembly code produced by GCC with command-line option -Og closely follows this template:

```
  long fact_for(long n)
  n in %rdi
fact_for:
  movl    $1, %eax        Set result = 1
  movl    $2, %edx        Set i = 2
  jmp     .L8             Goto test
.L9:                    loop:
  imulq   %rdx, %rax      Compute result *= i
  addq    $1, %rdx        Increment i
.L8:                    test:
  cmpq    %rdi, %rdx      Compare i:n
  jle     .L9             If <=, goto loop
  rep; ret                Return
```

Practice Problem 3.27 (solution page 372)

Write goto code for a function called `fibonacci` to print fibonacci numbers using a `while` loop. Apply the guarded-do transformation.

We see from this presentation that all three forms of loops in C—`do-while`, `while`, and `for`—can be translated by a simple strategy, generating code that contains one or more conditional branches. Conditional transfer of control provides the basic mechanism for translating loops into machine code.

Practice Problem 3.28 (solution page 372)

A function `test_two` has the following overall structure:

```
short test_two(unsigned short x) {
    short val = 0;
    short i;
    for ( ... ; ... ; ... ) {
        .
        .
        .
    }
    return val;
}
```

The GCC C compiler generates the following assembly code:

```
  test fun_b(unsigned test x)
  x in %rdi
test_two:
  movl    $1, %edx
```

```
    movl    $65, %eax
  .L10:
    movq    %rdi, %rcx
    andl    $1, %ecx
    addq    %rax, %rax
    orq     %rcx, %rax
    shrq    %rdi              Shift right by 1
    addq    $1, %rdx
    jne     .L10
    rep; ret
```

Reverse engineer the operation of this code and then do the following:

A. Use the assembly-code version to fill in the missing parts of the C code.

B. Explain why there is neither an initial test before the loop nor an initial jump to the test portion of the loop.

C. Describe in English what this function computes.

Practice Problem 3.29 (solution page 373)

Executing a `continue` statement in C causes the program to jump to the end of the current loop iteration. The stated rule for translating a `for` loop into a `while` loop needs some refinement when dealing with `continue` statements. For example, consider the following code:

```
/* Example of for loop containing a continue statement */
/* Sum even numbers between 0 and 9 */
long sum = 0;
long i;
for (i = 0; i < 10; i++) {
    if (i & 1)
        continue;
    sum += i;
}
```

A. What would we get if we naively applied our rule for translating the `for` loop into a `while` loop? What would be wrong with this code?

B. How could you replace the `continue` statement with a `goto` statement to ensure that the `while` loop correctly duplicates the behavior of the `for` loop?

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

```
void switch_eg_impl(long x, long n,
                    long *dest)
{
    /* Table of code pointers */
    static void *jt[7] = {
        &&loc_A, &&loc_def, &&loc_B,
        &&loc_C, &&loc_D, &&loc_def,
        &&loc_D
    };
    unsigned long index = n - 100;
    long val;

    if (index > 6)
        goto loc_def;
    /* Multiway branch */
    goto *jt[index];

 loc_A:    /* Case 100 */
    val = x * 13;
    goto done;
 loc_B:    /* Case 102 */
    x = x + 10;
    /* Fall through */
 loc_C:    /* Case 103 */
    val = x + 11;
    goto done;
 loc_D:    /* Cases 104, 106 */
    val = x * x;
    goto done;
 loc_def:  /* Default case */
    val = 0;
 done:
    *dest = val;
}
```

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
switch_eg:
  subq    $100, %rsi              Compute index = n-100
  cmpq    $6, %rsi                Compare index:6
  ja      .L8                     If >, goto loc_def
  jmp     *.L4(,%rsi,8)           Goto *jg[index]
.L3:                            loc_A:
  leaq    (%rdi,%rdi,2), %rax     3*x
  leaq    (%rdi,%rax,4), %rdi     val = 13*x
  jmp     .L2                     Goto done
.L5:                            loc_B:
  addq    $10, %rdi               x = x + 10
.L6:                            loc_C:
  addq    $11, %rdi               val = x + 11
  jmp     .L2                     Goto done
.L7:                            loc_D:
  imulq   %rdi, %rdi              val = x * x
  jmp     .L2                     Goto done
.L8:                            loc_def:
  movl    $0, %edi                val = 0
.L2:                            done:
  movq    %rdi, (%rdx)            *dest = val
  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:

```
  .section        .rodata
  .align 8           Align address to multiple of 8
.L4:
  .quad   .L3        Case 100: loc_A
  .quad   .L8        Case 101: loc_def
  .quad   .L5        Case 102: loc_B
  .quad   .L6        Case 103: loc_C
  .quad   .L7        Case 104: loc_D
  .quad   .L8        Case 105: loc_def
  .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 374)

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(short x, short *dest) {
    short 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(short x, short *dest)
    x in %rdi
switch2:
  addq    $2, %rdi
  cmpq    $8, %rdi
  ja      .L2
  jmp     *.L4(,%rdi,8)
```

It generates the following code for the jump table:

```
.L4:
  .quad   .L9
  .quad   .L5
  .quad   .L6
  .quad   .L7
  .quad   .L2
  .quad   .L7
  .quad   .L8
  .quad   .L2
  .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 374)

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
switcher:
  cmpq    $7, %rdi
  ja      .L2
  jmp     *.L4(,%rdi,8)
  .section        .rodata
.L7:
  xorq    $15, %rsi
  movq    %rsi, %rdx
.L3:
  leaq    112(%rdx), %rdi
  jmp     .L6
.L5:
  leaq    (%rdx,%rsi), %rdi
  salq    $2, %rdi
  jmp     .L6
.L2:
  movq    %rsi, %rdi
.L6:
  movq    %rdi, (%rcx)
  ret
```

(b) Jump table

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

Figure 3.24 Assembly code and jump table for Problem 3.31.

3.7 Procedures

Procedures are a key abstraction in software. They provide a way to package code that implements some functionality with a designated set of arguments and an optional return value. This function can then be invoked from different points in a program. Well-designed software uses procedures as an abstraction mechanism, hiding the detailed implementation of some action while providing a clear and concise interface definition of what values will be computed and what effects the procedure will have on the program state. Procedures come in many guises