```
double complex c_sub(double complex x, double complex y) {
    return x - y;
}
```

When compiled, GCC generates the following assembly code for these functions:

```
  double c_imag(double complex x)
c_imag:
  movapd  %xmm1, %xmm0
  ret

  double c_real(double complex x)
c_real:
  rep; ret

  double complex c_sub(double complex x, double complex y)
c_sub:
  subsd   %xmm2, %xmm0
  subsd   %xmm3, %xmm1
  ret
```

Based on these examples, determine the following:

A. How are complex arguments passed to a function?

B. How are complex values returned from a function?

Solutions to Practice Problems

Solution to Problem 3.1 (page 218)

This exercise gives you practice with the different operand forms.

Operand	Value	Comment
`%rax`	`0x100`	Register
`0x104`	`0xAB`	Absolute address
`$0x108`	`0x108`	Immediate
`(%rax)`	`0xFF`	Address `0x100`
`4(%rax)`	`0xAB`	Address `0x104`
`9(%rax,%rdx)`	`0x11`	Address `0x10C`
`260(%rcx,%rdx)`	`0x13`	Address `0x108`
`0xFC(,%rcx,4)`	`0xFF`	Address `0x100`
`(%rax,%rdx,4)`	`0x11`	Address `0x10C`

Solution to Problem 3.2 (page 221)

As we have seen, the assembly code generated by GCC includes suffixes on the instructions, while the disassembler does not. Being able to switch between these two forms is an important skill to learn. One important feature is that memory references in x86-64 are always given with quad word registers, such as `%rax`, even if the operand is a byte, single word, or double word.

Here is the code written with suffixes:

```
movl   %eax, (%rsp)
movw   (%rax), %dx
movb   $0xFF, %bl
movb   (%rsp,%rdx,4), %dl
movq  (%rdx), %rax
movw   %dx, (%rax)
```

Solution to Problem 3.3 (page 222)

Since we will rely on GCC to generate most of our assembly code, being able to write correct assembly code is not a critical skill. Nonetheless, this exercise will help you become more familiar with the different instruction and operand types.

Here is the code with explanations of the errors:

```
movb $0xF, (%ebx)         Cannot use %ebx as address register
movl %rax, (%rsp)         Mismatch between instruction suffix and register ID
movw (%rax),4(%rsp)       Cannot have both source and destination be memory references
movb %al,%sl              No register named %sl
movl %eax,$0x123          Cannot have immediate as destination
movl %eax,%dx             Destination operand incorrect size
movb %si, 8(%rbp)         Mismatch between instruction suffix and register ID
```

Solution to Problem 3.4 (page 223)

This exercise gives you more experience with the different data movement instructions and how they relate to the data types and conversion rules of C. The nuances of conversions of both signedness and size, as well as integral promotion, add challenge to this problem.

`src_t`	`dest_t`	Instruction	Comments
`long`	`long`	`movq (%rdi), %rax`	Read 8 bytes
		`movq %rax, (%rsi)`	Store 8 bytes
`char`	`int`	`movsbl (%rdi), %eax`	Convert char to int
		`movl %eax, (%rsi)`	Store 4 bytes
`char`	`unsigned`	`movsbl (%rdi), %eax`	Convert char to int
		`movl %eax, (%rsi)`	Store 4 bytes
`unsigned char`	`long`	`movzbl (%rdi), %eax`	Read byte and zero-extend
		`movq %rax, (%rsi)`	Store 8 bytes

`int`	`char`	`movl (%rdi), %eax`	Read 4 bytes
		`movb %al, (%rsi)`	Store low-order byte
`unsigned`	`unsigned`	`movl (%rdi), %eax`	Read 4 bytes
	`char`	`movb %al, (%rsi)`	Store low-order byte
`char`	`short`	`movsbw (%rdi), %ax`	Read byte and sign-extend
		`movw %ax, (%rsi)`	Store 2 bytes

Solution to Problem 3.5 (page 225)

Reverse engineering is a good way to understand systems. In this case, we want to reverse the effect of the C compiler to determine what C code gave rise to this assembly code. The best way is to run a "simulation," starting with values x, y, and z at the locations designated by pointers xp, yp, and zp, respectively. We would then get the following behavior:

```
  void decode1(long *xp, long *yp, long *zp)
  xp in %rdi, yp in %rsi, zp in %rdx
decode1:
  movq    (%rdi), %r8     Get x = *xp
  movq    (%rsi), %rcx    Get y = *yp
  movq    (%rdx), %rax    Get z = *zp
  movq    %r8, (%rsi)     Store x at yp
  movq    %rcx, (%rdx)    Store y at zp
  movq    %rax, (%rdi)    Store z at xp
  ret
```

From this, we can generate the following C code:

```
void decode1(long *xp, long *yp, long *zp)
{
    long x = *xp;
    long y = *yp;
    long z = *zp;

    *yp = x;
    *zp = y;
    *xp = z;
}
```

Solution to Problem 3.6 (page 228)

This exercise demonstrates the versatility of the `leaq` instruction and gives you more practice in deciphering the different operand forms. Although the operand forms are classified as type "Memory" in Figure 3.3, no memory access occurs.

Instruction	Result
`leaq 9(%rdx), %rax`	$9 + q$
`leaq (%rdx,%rbx), %rax`	$q + p$
`leaq (%rdx,%rbx,3), %rax`	$q + 3p$
`leaq 2(%rbx,%rbx,7), %rax`	$2 + 8p$
`leaq 0xE(,%rdx,3), %rax`	$14 + 3q$
`leaq 6(%rbx,%rdx,7), %rdx`	$6 + p + 7q$

Solution to Problem 3.7 (page 229)

Again, reverse engineering proves to be a useful way to learn the relationship between C code and the generated assembly code.

The best way to solve problems of this type is to annotate the lines of assembly code with information about the operations being performed. Here is a sample:

```
  short scale3(short x, short y, short z)
  x in %rdi, y in %rsi, z in %rdx
scale3:
  leaq    (%rsi,%rsi,9), %rbx      10 * y
  leaq    (%rbx,%rdx), %rbx        10 * y + z
  leaq    (%rbx,%rdi,%rsi), %rbx   10 * y + z + y * x
  ret
```

From this, it is easy to generate the missing expression:

```
    short t = 10 * y + z + y * x;
```

Solution to Problem 3.8 (page 230)

This problem gives you a chance to test your understanding of operands and the arithmetic instructions. The instruction sequence is designed so that the result of each instruction does not affect the behavior of subsequent ones.

Instruction	Destination	Value
`addq %rcx,(%rax)`	`0x100`	`0x100`
`subq %rdx,8(%rax)`	`0x108`	`0xA8`
`imulq $16,(%rax,%rdx,8)`	`0x118`	`0x110`
`incq 16(%rax)`	`0x110`	`0x14`
`decq %rcx`	`%rcx`	`0x0`
`subq %rdx,%rax`	`%rax`	`0xFD`

Solution to Problem 3.9 (page 231)

This exercise gives you a chance to generate a little bit of assembly code. The solution code was generated by GCC. By loading parameter `n` in register `%ecx`, it can then use byte register `%cl` to specify the shift amount for the `sarq` instruction. It might seem odd to use a `movl` instruction, given that `n` is eight bytes long, but keep in mind that only the least significant byte is required to specify the shift amount.

```
  long shift_left4_rightn(long x, long n)
  x in %rdi, n in %rsi
shift_left4_rightn:
  movq    %rdi, %rax     Get x
  salq    $4, %rax       x <<= 4
  movl    %esi, %ecx     Get n (4 bytes)
  sarq    %cl, %rax      x >>= n
```

Solution to Problem 3.10 (page 232)

This problem is fairly straightforward, since the assembly code follows the structure of the C code closely.

```
short p1 = y | z;
short p2 = p1 >> 9;
short p3 = ~p2;
short p4 = y - p3;
```

Solution to Problem 3.11 (page 233)

A. This instruction is used to set register `%rcx` to zero, exploiting the property that x ^ $x = 0$ for any x. It corresponds to the C statement `x = 0`.

B. A more direct way of setting register `%rcx` to zero is with the instruction `movq $0,%rcx`.

C. Assembling and disassembling this code, however, we find that the version with `xorq` requires only 3 bytes, while the version with `movq` requires 7. Other ways to set `%rcx` to zero rely on the property that any instruction that updates the lower 4 bytes will cause the high-order bytes to be set to zero. Thus, we could use either `xorl %ecx,%ecx` (2 bytes) or `movl $0,%ecx` (5 bytes).

Solution to Problem 3.12 (page 236)

We can simply replace the `cqto` instruction with one that sets register `%rdx` to zero, and use `divq` rather than `idivq` as our division instruction, yielding the following code:

```
void uremdiv(unsigned long x, unsigned long y,
             unsigned long *qp, unsigned long *rp)
x in %rdi, y in %rsi, qp in %rdx, rp in %rcx
uremdiv:
  movq    %rdx, %r8       Copy qp
  movq    %rdi, %rax      Move x to lower 8 bytes of dividend
  movl    $0, %edx        Set upper 8 bytes of dividend to 0
  divq    %rsi            Divide by y
  movq    %rax, (%r8)     Store quotient at qp
  movq    %rdx, (%rcx)    Store remainder at rp
  ret
```

Solution to Problem 3.13 (page 240)

It is important to understand that assembly code does not keep track of the type of a program value. Instead, the different instructions determine the operand sizes and whether they are signed or unsigned. When mapping from instruction sequences back to C code, we must do a bit of detective work to infer the data types of the program values.

A. The suffix '`l`' and the register identifiers indicate 32-bit operands, while the comparison is for a two's-complement <. We can infer that `data_t` must be `int`.

B. The suffix '`w`' and the register identifiers indicate 16-bit operands, while the comparison is for a two's-complement >=. We can infer that `data_t` must be `short`.

C. The suffix '`b`' and the register identifiers indicate 8-bit operands, while the comparison is for an unsigned <=. We can infer that `data_t` must be `unsigned char`.

D. The suffix '`q`' and the register identifiers indicate 64-bit operands, while the comparison is for `!=`, which is the same whether the arguments are signed, unsigned, or pointers. We can infer that `data_t` could be either `long`, `unsigned long`, or some form of pointer.

Solution to Problem 3.14 (page 241)

This problem is similar to Problem 3.13, except that it involves TEST instructions rather than CMP instructions.

A. The suffix '`q`' and the register identifiers indicate a 64-bit operand, while the comparison is for >=, which must be signed. We can infer that `data_t` must be `long`.

B. The suffix '`w`' and the register identifier indicate a 16-bit operand, while the comparison is for ==, which is the same for signed or unsigned. We can infer that `data_t` must be either `short` or `unsigned short`.

C. The suffix '`b`' and the register identifier indicate an 8-bit operand, while the comparison is for unsigned >. We can infer that `data_t` must be `unsigned char`.

D. The suffix '`l`' and the register identifier indicate 32-bit operands, while the comparison is for <. We can infer that `data_t` must be `int`.

Solution to Problem 3.15 (page 245)

This exercise requires you to examine disassembled code in detail and reason about the encodings for jump targets. It also gives you practice in hexadecimal arithmetic.

A. The `je` instruction has as its target `0x4003fc` + `0x02`. As the original disassembled code shows, this is `0x4003fe`:

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

B. The `je` instruction has as its target `0x0x400431` − 12 (since `0xf4` is the 1-byte two's-complement representation of −12). As the original disassembled code shows, this is `0x400425`:

```
40042f: 74 f4                    je     400425
400431: 5d                       pop    %rbp
```

C. According to the annotation produced by the disassembler, the jump target is at absolute address `0x400547`. According to the byte encoding, this must be at an address `0x2` bytes beyond that of the `pop` instruction. Subtracting these gives address `0x400545`. Noting that the encoding of the `ja` instruction requires 2 bytes, it must be located at address `0x400543`. These are confirmed by examining the original disassembly:

```
400543: 77 02                    ja     400547
400545: 5d                       pop    %rbp
```

D. Reading the bytes in reverse order, we can see that the target offset is `0xffffff73`, or decimal −141. Adding this to `0x0x4005ed` (the address of the `nop` instruction) gives address `0x400560`:

```
4005e8: e9 73 ff ff ff           jmpq   400560
4005ed: 90                       nop
```

Solution to Problem 3.16 (page 248)

Annotating assembly code and writing C code that mimics its control flow are good first steps in understanding assembly-language programs. This problem gives you practice for an example with simple control flow. It also gives you a chance to examine the implementation of logical operations.

A. Here is the C code:

```
void goto_cond(short a, short *p) {
    if (a == 0)
        goto done;
    if (a >= *p)
        goto done;
    *p = a;
 done:
    return;
}
```

B. The first conditional branch is part of the implementation of the `&&` expression. If the test for `a` being non-null fails, the code will skip the test of `a >= *p`.

Solution to Problem 3.17 (page 248)

This is an exercise to help you think about the idea of a general translation rule and how to apply it.

A. Converting to this alternate form involves only switching around a few lines of the code:

```
long gotodiff_se_alt(long x, long y) {
    long result;
    if (x < y)
        goto x_lt_y;
    ge_cnt++;
    result = x - y;
    return result;
 x_lt_y:
    lt_cnt++;
    result =  y - x;
    return result;
}
```

B. In most respects, the choice is arbitrary. But the original rule works better for the common case where there is no `else` statement. For this case, we can simply modify the translation rule to be as follows:

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

A translation based on the alternate rule is more cumbersome.

Solution to Problem 3.18 (page 249)

This problem requires that you work through a nested branch structure, where you will see how our rule for translating `if` statements has been applied. On the whole, the machine code is a straightforward translation of the C code.

```
short test(short x, short y, short z) {
    short val = z+y-x;
    if (z > 5) {
        if (y > 2)
            val = x/z;
        else
            val = x/y;
    } else if (z < 3)
        val = z/y;
    return val;
}
```

Solution to Problem 3.19 (page 252)

This problem reinforces our method of computing the misprediction penalty.

A. We can apply our formula directly to get $T_{\mathrm{MP}} = 2(45 - 25) = 40$.

B. When misprediction occurs, the function will require around $25 + 40 = 65$ cycles.

Solution to Problem 3.20 (page 255)

This problem provides a chance to study the use of conditional moves.

A. The operator is '/'. We see this is an example of dividing by a power of 4 by right shifting (see Section 2.3.7). Before shifting by $k = 4$, we must add a bias of $2^k - 1 = 15$ when the dividend is negative.

B. Here is an annotated version of the assembly code:

```
  short arith(short x)
  x in %rdi
arith:
  leaq    15(%rdi), %rbx    temp = x+15
  testq   %rdi, %rdi        Text x
  cmovns  %rdi, %rbx        If x>= 0, temp = x
  sarq    $4, %rbx          result = temp >> 4 (= x/16)
  ret
```

The program creates a temporary value equal to $x + 15$, in anticipation of x being negative and therefore requiring biasing. The `cmovns` instruction conditionally changes this number to x when $x \geq 0$, and then it is shifted by 4 to generate $x/16$.

Solution to Problem 3.21 (page 255)

This problem is similar to Problem 3.18, except that some of the conditionals have been implemented by conditional data transfers. Although it might seem daunting to fit this code into the framework of the original C code, you will find that it follows the translation rules fairly closely.

```
short test(short x, short y) {
    short val = y + 12;
    if (x < 0) {
        if (x < y)
            val = x * y;
        else
            val = x | y;
    } else if (y > 10)
        val = x / y;
    return val;
}
```

Solution to Problem 3.22 (page 257)

A. The computation of 14! would overflow with a 32-bit `int`. As we learned in Problem 2.35, when we get value x while attempting to compute $n!$, we can test for overflow by computing x/n and seeing whether it equals $(n - 1)!$

(assuming that we have already ensured that the computation of $(n - 1)!$ did not overflow). In this case we get $1{,}278{,}945{,}280/14 = 91353234.286$. As a second test, we can see that any factorial beyond 10! must be a multiple of 100 and therefore have zeros for the last two digits. The correct value of 14! is 87,178,291,200.

Further, we can build up a table of factorials computed through 14! with data type `int`, as shown below:

n	$n!$	OK?
1	1	Y
2	2	Y
3	6	Y
4	24	Y
5	120	Y
6	720	Y
7	5,040	Y
8	40,320	Y
9	362,880	Y
10	3,628,800	Y
11	39,916,800	Y
12	479,001,600	Y
13	1,932,053,504	N
14	1,278,945,280	N

B. Doing the computation with data type `long` lets us go up to 20!, thus the 14! computation does not overflow.

Solution to Problem 3.23 (page 258)

The code generated when compiling loops can be tricky to analyze, because the compiler can perform many different optimizations on loop code, and because it can be difficult to match program variables with registers. This particular example demonstrates several places where the assembly code is not just a direct translation of the C code.

A. Although parameter `x` is passed to the function in register `%rdi`, we can see that the register is never referenced once the loop is entered. Instead, we can see that registers `%rbx`, `%rcx`, and `%rdx` are initialized in lines 2–5 to `x`, `x/9`, and `4*x`. We can conclude, therefore, that these registers contain the program variables.

B. The compiler determines that pointer `p` always points to `x`, and hence the expression `(*p)+=5` simply increments `x`. It combines this incrementing by 5 with the increment of `y`, via the `leaq` instruction of line 7.

C. The annotated code is as follows:

```
    short dw_loop(short x)
    x initially in %rdi
    dw_loop:
      movq    %rdi, %rbx             Copy x to %rbx
      movq    %rdi, %rcx
      idivq   $9, %rcx               Compute y = x/9
      leaq    (,%rdi,4), %rdx        Compute n = 4*x
    .L2:                           loop:
      leaq    5(%rbx,%rcx), %rcx     Compute y += x + 5
      subq    $2, %rdx               Decrement n by 2
      testq   %rdx, %rdx             Test n
      jg      .L2                    If > 0, goto loop
      rep; ret                     Return
```

Solution to Problem 3.24 (page 260)

This assembly code is a fairly straightforward translation of the loop using the jump-to-middle method. The full C code is as follows:

```
short loop_while(short a, short b)
{
    short result = 0;
    while (a > b) {
        result = result + (a*b);
        a = a-1;
    }
    return result;
}
```

Solution to Problem 3.25 (page 262)

While the generated code does not follow the exact pattern of the guarded-do translation, we can see that it is equivalent to the following C code:

```
long loop_while2(long a, long b)
{
    long result = b;
    while (b > 0) {
        result = result * a;
        b = b-a;
    }
    return result;
}
```

We will often see cases, especially when compiling with higher levels of optimization, where GCC takes some liberties in the exact form of the code it generates, while preserving the required functionality.

Solution to Problem 3.26 (page 264)

Being able to work backward from assembly code to C code is a prime example of reverse engineering.

A. We can see that the code uses the jump-to-middle translation, using the `jmp` instruction on line 3.

B. Here is the original C code:

```
short test_one(unsigned short x) {
    short val = 1;
    while (x) {
        val ^= x;
        x >>= 1;
    }
    return val & 0;
}
```

C. This code computes the *parity* of argument `x`. That is, it returns 1 if there is an odd number of ones in `x` and 0 if there is an even number.

Solution to Problem 3.27 (page 267)

This exercise is intended to reinforce your understanding of how loops are implemented.

```
long fibonacci_gd_goto(long n)
{
    long i = 2;
    long next, first = 0, second = 1;
    if (n <= 1)
        goto done;
 loop:
    next = first + second;
    first = second; second = next;
    i++;
    if (i <= n)
        goto loop;
 done:
    return n;
}
```

Solution to Problem 3.28 (page 267)

This problem is trickier than Problem 3.26, since the code within the loop is more complex and the overall operation is less familiar.

A. Here is the original C code:

```
long fun_b(unsigned long x) {
    long val = 0;
    long i;
    for (i = 64; i != 0; i--) {
        val = (val << 1) | (x & 0x1);
        x >>= 1;
    }
    return val;
}
```

B. The code was generated using the guarded-do transformation, but the compiler detected that, since i is initialized to 64, it will satisfy the test $i \neq 0$, and therefore the initial test is not required.

C. This code reverses the bits in `x`, creating a mirror image. It does this by shifting the bits of `x` from left to right, and then filling these bits in as it shifts `val` from right to left.

Solution to Problem 3.29 (page 268)

Our stated rule for translating a `for` loop into a `while` loop is just a bit too simplistic—this is the only aspect that requires special consideration.

A. Applying our translation rule would yield the following code:

```
/* Naive translation of for loop into while loop */
/* WARNING: This is buggy code */
long sum = 0;
long i = 0;
while (i < 10) {
    if (i & 1)
        /* This will cause an infinite loop */
        continue;
    sum += i;
    i++;
}
```

This code has an infinite loop, since the `continue` statement would prevent index variable `i` from being updated.

B. The general solution is to replace the `continue` statement with a `goto` statement that skips the rest of the loop body and goes directly to the update portion:

```
/* Correct translation of for loop into while loop */
long sum = 0;
long i = 0;
while (i < 10) {
    if (i & 1)
        goto update;
    sum += i;
update:
    i++;
}
```

Solution to Problem 3.30 (page 272)

This problem gives you a chance to reason about the control flow of a `switch` statement. Answering the questions requires you to combine information from several places in the assembly code.

- Line 2 of the assembly code adds 2 to x to set the lower range of the cases to zero. That means that the minimum case label is -2.
- Lines 3 and 4 cause the program to jump to the default case when the adjusted case value is greater than 8. This implies that the maximum case label is $-2 + 8 = 6$.
- In the jump table, we see that the entry on lines 6 (case value 2) and 9 (case value 5) have the same destination (`.L2`) as the jump instruction on line 4, indicating the default case behavior. Thus, case labels 2 and 5 are missing in the `switch` statement body.
- In the jump table, we see that the entries on lines 3 and 10 have the same destination. These correspond to cases -1 and 6.
- In the jump table, we see that the entries on lines 5 and 7 have the same destination. These correspond to cases 1 and 3.

From this reasoning, we draw the following conclusions:

A. The case labels in the `switch` statement body have values -2, -1, 0, 1, 3, 4, and 6.

B. The case with destination `.L5` has labels -1 and 6.

C. The case with destination `.L7` has labels 1 and 3.

Solution to Problem 3.31 (page 273)

The key to reverse engineering compiled `switch` statements is to combine the information from the assembly code and the jump table to sort out the different cases. We can see from the `ja` instruction (line 3) that the code for the default case has label `.L2`. We can see that the only other repeated label in the jump table is `.L5`, and so this must be the code for the cases C and D. We can see that the code falls through at line 8, and so label `.L7` must match case A and label `.L3` must match case B. That leaves only label `.L6` to match case E.

The original C code is as follows:

```
void switcher(long a, long b, long c, long *dest)
{
    long val;
    switch(a) {
    case 5:
        c = b ^ 15;
        /* Fall through */
    case 0:
        val = c + 112;
        break;
    case 2:
    case 7:
        val = (c + b) << 2;
        break;
    case 4:
        val = a;
        break;
    default:
        val = b;
    }
    *dest = val;
}
```

Solution to Problem 3.32 (page 280)

Tracing through the program execution at this level of detail reinforces many aspects of procedure call and return. We can see clearly how control is passed to the function when it is called, and how the calling function resumes upon return. We can also see how arguments get passed through registers `%rdi` and `%rsi`, and how results are returned via register `%rax`.

Instruction			State values (at beginning)					
Label	PC	Instruction	`%rdi`	`%rsi`	`%rax`	`%rsp`	`*%rsp`	Description
M1	`0x400560`	`callq`	10	—	—	`0x7fffffffe820`	—	Call `first(10)`
F1	`0x400548`	`lea`	10	—	—	`0x7fffffffe818`	`0x400565`	Entry of `first`
F2	`0x40054c`	`sub`	10	11	—	`0x7fffffffe818`	`0x400565`	
F3	`0x400550`	`callq`	9	11	—	`0x7fffffffe818`	`0x400565`	Call `last(9, 11)`
L1	`0x400540`	`mov`	9	11	—	`0x7fffffffe810`	`0x400555`	Entry of `last`
L2	`0x400543`	`imul`	9	11	9	`0x7fffffffe810`	`0x400555`	
L3	`0x400547`	`retq`	9	11	99	`0x7fffffffe810`	`0x400555`	Return 99 from `last`
F4	`0x400555`	`repz repq`	9	11	99	`0x7fffffffe818`	`0x400565`	Return 99 from `first`
M2	`0x400565`	`mov`	9	11	99	`0x7fffffffe820`	—	Resume `main`

Solution to Problem 3.33 (page 282)

This problem is a bit tricky due to the mixing of different data sizes.

Let us first describe one answer and then explain the second possibility. If we assume the first addition (line 3) implements `*u += a`, while the second (line 4) implements `v += b`, then we can see that `a` was passed as the first argument in `%edi` and converted from 4 bytes to 8 before adding it to the 8 bytes pointed to by `%rdx`. This implies that `a` must be of type `int` and `u` must be of type `long *`. We can also see that the low-order byte of argument `b` is added to the byte pointed to by `%rcx`. This implies that `v` must be of type `char *`, but the type of `b` is ambiguous—it could be 1, 2, 4, or 8 bytes long. This ambiguity is resolved by noting the return value of

6, computed as the sum of the sizes of a and b. Since we know a is 4 bytes long, we can deduce that b must be 2.

An annotated version of this function explains these details:

```
int procprobl(int a, short b, long *u, char *v)
a in %edi, b in %si, u in %rdx, v in %rcx
procprob:
  movslq  %edi, %rdi        Convert a to long
  addq    %rdi, (%rdx)      Add to *u (long)
  addb    %sil, (%rcx)      Add low-order byte of b to *v
  movl    $6, %eax          Return 4+2
  ret
```

Alternatively, we can see that the same assembly code would be valid if the two sums were computed in the assembly code in the opposite ordering as they are in the C code. This would result in interchanging arguments a and b and arguments u and v, yielding the following prototype:

```
int procprob(int b, short a, long *v, char *u);
```

Solution to Problem 3.34 (page 288)

This example demonstrates the use of callee-saved registers as well as the stack for holding local data.

A. We can see that lines 9–14 save local values a0–a5 into callee-saved registers %rbx, %r15, %r14, %r13, %r12, and %rbp, respectively.

B. Local values a6 and a7 are stored on the stack at offsets 0 and 8 relative to the stack pointer (lines 16 and 18).

C. After storing six local variables, the program has used up the supply of callee-saved registers. It stores the remaining two local values on the stack.

Solution to Problem 3.35 (page 290)

This problem provides a chance to examine the code for a recursive function. An important lesson to learn is that recursive code has the exact same structure as the other functions we have seen. The stack and register-saving disciplines suffice to make recursive functions operate correctly.

A. Register %rbx holds the value of parameter x, so that it can be used to compute the result expression.

B. The assembly code was generated from the following C code:

```
long rfun(unsigned long x) {
    if (x == 0)
        return 0;
    unsigned long nx = x>>2;
    long rv = rfun(nx);
    return x + rv;
}
```

Solution to Problem 3.36 (page 292)

This exercise tests your understanding of data sizes and array indexing. Observe that a pointer of any kind is 8 bytes long. Data type `short` requires 2 bytes, while `int` requires 4.

Array	Element size	Total size	Start address	Element i
P	4	20	$x_{\mathtt{P}}$	$x_{\mathtt{P}} + 4i$
Q	2	4	$x_{\mathtt{Q}}$	$x_{\mathtt{Q}} + 2i$
R	8	72	$x_{\mathtt{R}}$	$x_{\mathtt{R}} + 8i$
S	8	80	$x_{\mathtt{S}}$	$x_{\mathtt{S}} + 8i$
T	8	16	$x_{\mathtt{T}}$	$x_{\mathtt{T}} + 8i$

Solution to Problem 3.37 (page 294)

This problem is a variant of the one shown for integer array `E`. It is important to understand the difference between a pointer and the object being pointed to. Since data type `short` requires 2 bytes, all of the array indices are scaled by a factor of 2. Rather than using `movl`, as before, we now use `movw`.

Expression	Type	Value	Assembly Code
`P[1]`	`short`	$M[x_{\mathtt{P}} + 2]$	`movw 2(%rdx),%ax`
`P+3+i`	`short *`	$x_{\mathtt{P}} + 6 + 2i$	`leaq 6(%rdx,%rcx,2),%rax`
`P[i*6-5]`	`short`	$M[x_{\mathtt{P}} + 12i - 10]$	`movw -10(%rdx,%rcx,12),%ax`
`P[2]`	`short`	$M[x_{\mathtt{P}} + 4]$	`movw 4(%rdx),%ax`
`&P[i+2]`	`short *`	$x_{\mathtt{P}} + 2i + 4$	`leaq 4(%rdx,%rcx,2),%rax`

Solution to Problem 3.38 (page 295)

This problem requires you to work through the scaling operations to determine the address computations, and to apply Equation 3.1 for row-major indexing. The first step is to annotate the assembly code to determine how the address references are computed:

```
    long sum_element(long i, long j)
    i in %rdi, j in %rsi
sum_element:
  leaq    0(,%rdi,8), %rdx         Compute 8i
  subq    %rdi, %rdx               Compute 7i
  addq    %rsi, %rdx               Compute 7i + j
  leaq    (%rsi,%rsi,4), %rax      Compute 5j
  addq    %rax, %rdi               Compute i + 5j
  movq    Q(,%rdi,8), %rax         Retrieve M[x_Q + 8 (5j + i)]
  addq    P(,%rdx,8), %rax         Add M[x_P + 8 (7i + j)]
  ret
```

We can see that the reference to matrix `P` is at byte offset $8 \cdot (7i + j)$, while the reference to matrix `Q` is at byte offset $8 \cdot (5j + i)$. From this, we can determine that `P` has 7 columns, while `Q` has 5, giving $M = 5$ and $N = 7$.

Solution to Problem 3.39 (page 298)

These computations are direct applications of Equation 3.1:

- For $L = 4$, $C = 16$, and $j = 0$, pointer `Aptr` is computed as $x_A + 4 \cdot (16i + 0) = x_A + 64i$.
- For $L = 4$, $C = 16$, $i = 0$, and $j = k$, `Bptr` is computed as $x_B + 4 \cdot (16 \cdot 0 + k) = x_B + 4k$.
- For $L = 4$, $C = 16$, $i = 16$, and $j = k$, `Bend` is computed as $x_B + 4 \cdot (16 \cdot 16 + k) = x_B + 1{,}024 + 4k$.

Solution to Problem 3.40 (page 298)

This exercise requires that you be able to study compiler-generated assembly code to understand what optimizations have been performed. In this case, the compiler was clever in its optimizations.

Let us first study the following C code, and then see how it is derived from the assembly code generated for the original function.

```
/* Set all diagonal elements to val */
void fix_set_diag_opt(fix_matrix A, int val) {
    int *Abase = &A[0][0];
    long i = 0;
    long iend = N*(N+1);
    do {
        Abase[i] = val;
        i += (N+1);
    } while (i != iend);
}
```

This function introduces a variable `Abase`, of type `int *`, pointing to the start of array `A`. This pointer designates a sequence of 4-byte integers consisting of elements of `A` in row-major order. We introduce an integer variable `index` that steps through the diagonal elements of `A`, with the property that diagonal elements i and $i + 1$ are spaced $N + 1$ elements apart in the sequence, and that once we reach diagonal element N (index value $N(N + 1)$), we have gone beyond the end.

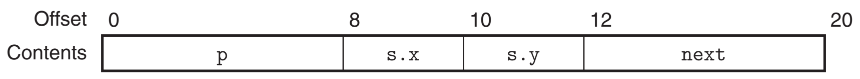
The actual assembly code follows this general form, but now the pointer increments must be scaled by a factor of 4. We label register `%rax` as holding a value `index4` equal to `index` in our C version but scaled by a factor of 4. For $N = 16$, we can see that our stopping point for `index4` will be $4 \cdot 16(16 + 1) = 1{,}088$.

```
fix_set_diag:
void fix_set_diag(fix_matrix A, int val)
A in %rdi, val in %rsi
  movl    $0, %eax                  Set index4 = 0
.L13:                              loop:
  movl    %esi, (%rdi,%rax)         Set Abase[index4/4] to val
  addq    $68, %rax                 Increment index4 += 4(N+1)
  cmpq    $1088, %rax              Compare index4: 4N(N+1)
  jne     .L13                     If !=, goto loop
  rep; ret                         Return
```

Solution to Problem 3.41 (page 304)

This problem gets you to think about structure layout and the code used to access structure fields. The structure declaration is a variant of the example shown in the text. It shows that nested structures are allocated by embedding the inner structures within the outer ones.

A. The layout of the structure is as follows:



B. It uses 20 bytes.

C. As always, we start by annotating the assembly code:

```
void st_init(struct test *st)
st in %rdi
st_init:
  movl    8(%rdi), %eax      Get st->s.x
  movl    %eax, 10(%rdi)     Save in st->s.y
  leaq    10(%rdi), %rax     Compute &(st->s.y)
  movq    %rax, (%rdi)       Store in st->p
  movq    %rdi, 12(%rdi)     Store st in st->next
  ret
```

From this, we can generate C code as follows:

```
void st_init(struct test *st)
{
    st->s.y    = st->s.x;
    st->p      = &(st->s.y);
    st->next   = st;
}
```

Solution to Problem 3.42 (page 305)

This problem demonstrates how a very common data structure and operation on it is implemented in machine code. We solve the problem by first annotating the assembly code, recognizing that the two fields of the structure are at offsets 0 (for v) and 2 (for p).

```
short test(struct ACE *ptr)
ptr in %rdi
test:
  movl    $1, %eax                 result = 1
  jmp     .L2                      Goto middle
```

```
.L3:                            loop:
  imulq   (%rdi), %rax            result *= ptr->v
  movq    2(%rdi), %rdi           ptr = ptr->p
.L2:                            middle:
  testq   %rdi, %rdi              Test ptr
  jne     .L3                     If != NULL, goto loop
  rep; ret
```

A. Based on the annotated code, we can generate a C version:

```
short test(struct ACE *ptr) {
    short val = 1;
    while (ptr) {
        val *= ptr->v;
        ptr  = ptr->p;
    }
    return val;
}
```

B. We can see that each structure is an element in a singly linked list, with field v being the value of the element and p being a pointer to the next element. Function fun computes the sum of the element values in the list.

Solution to Problem 3.43 (page 308)

Structures and unions involve a simple set of concepts, but it takes practice to be comfortable with the different referencing patterns and their implementations.

EXPR	TYPE	Code

`up->t1.u`	`long`	`movq (%rdi), %rax`
		`movq %rax, (%rsi)`

`up->t1.v`	`short`	`movw 8(%rdi), %ax`
		`movw %ax, (%rsi)`

`&up->t1.w`	`char *`	`addq $10, %rdi`
		`movq %rdi, (%rsi)`

`up->t2.a`	`int *`	`movq %rdi, (%rsi)`

`up->t2.a[up->t1.u]`	`int`	`movq (%rdi), %rax`
		`movl (%rdi,%rax,4), %eax`
		`movl %eax, (%rsi)`

`*up->t2.p`	`char`	`movq 8(%rdi), %rax`
		`movb (%rax), %al`
		`movb %al, (%rsi)`

Solution to Problem 3.44 (page 311)

Understanding structure layout and alignment is very important for understanding how much storage different data structures require and for understanding the code generated by the compiler for accessing structures. This problem lets you work out the details of some example structures.

A. `struct P1 { short i; int c; int *j; short *d; };`

`i`	`c`	`j`	`d`	Total	Alignment
0	2	6	14	16	8

B. `struct P2 { int i[2]; char c[8]; short [4]; long *j; };`

`i`	`c`	`d`	`j`	Total	Alignment
0	8	16	24	32	8

C. `struct P3 { long w[2]; int *c[2] };`

`w`	`c`	Total	Alignment
0	16	32	8

D. `struct P4 { char w[16]; char *c[2] };`

`w`	`c`	Total	Alignment
0	16	32	8

E. `struct P5 { struct P4 a[2]; struct P1 t };`

`a`	`t`	Total	Alignment
0	24	40	8

Solution to Problem 3.45 (page 311)

This is an exercise in understanding structure layout and alignment.

A. Here are the object sizes and byte offsets:

Field	`a`	`b`	`c`	`d`	`e`	`f`	`g`	`h`
Size	8	4	1	2	8	8	4	8
Offset	0	8	12	16	24	32	40	48

B. The structure is a total of 56 bytes long. The end of the structure does not require padding to satisfy the 8-byte alignment requirement.

C. One strategy that works, when all data elements have a length equal to a power of 2, is to order the structure elements in descending order of size. This leads to a declaration:

```
struct {
    int    *a;
    char   *h;
    double f;
    long   e;
    float  b;
    int    g;
    short  d;
    char   c;
} rec;
```

with the following offsets:

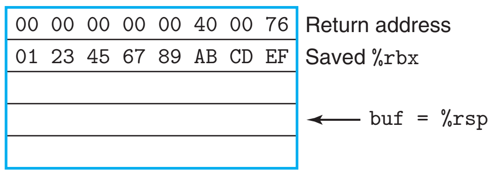
	Field							
	a	h	f	e	b	g	d	c
Size	8	8	8	8	4	4	2	1
Offset	0	8	16	24	32	36	40	42

The structure must be padded by 5 bytes to satisfy the 8-byte alignment requirement, giving a total of 48 bytes.

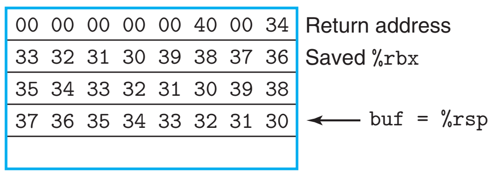
Solution to Problem 3.46 (page 318)

This problem covers a wide range of topics, such as stack frames, string representations, ASCII code, and byte ordering. It demonstrates the dangers of out-of-bounds memory references and the basic ideas behind buffer overflow.

A. Stack after line 3:



B. Stack after line 5:



C. The program is attempting to return to address `0x040034`. The low-order 2 bytes were overwritten by the code for character ‘`4`’ and the terminating null character.

D. The saved value of register `%rbx` was set to `0x3332313039383736`. This value will be loaded into the register before `get_line` returns.

E. The call to `malloc` should have had `strlen(buf)+1` as its argument, and the code should also check that the returned value is not equal to `NULL`.

Solution to Problem 3.47 (page 322)

A. This corresponds to a range of around 2^{13} addresses.

B. A 128-byte nop sled would cover 2^7 addresses with each test, and so we would only require around $2^6 = 64$ attempts.

This example clearly shows that the degree of randomization in this version of Linux would provide only minimal deterrence against an overflow attack.

Solution to Problem 3.48 (page 324)

This problem gives you another chance to see how x86-64 code manages the stack, and to also better understand how to defend against buffer overflow attacks.

A. For the unprotected code, we can see that lines 4 and 5 compute the positions of `v` and `buf` to be at offsets 24 and 0 relative to `%rsp`. In the protected code, the canary is stored at offset 40 (line 4), while `v` and `buf` are at offsets 8 and 16 (lines 7 and 8).

B. In the protected code, local variable `v` is positioned closer to the top of the stack than `buf`, and so an overrun of `buf` will not corrupt the value of `v`.

Solution to Problem 3.49 (page 329)

This code combines many of the tricks we have seen for performing bit-level arithmetic. It requires careful study to make any sense of it.

A. The `leaq` instruction of line 5 computes the value $8n + 22$, which is then rounded down to the nearest multiple of 16 by the `andq` instruction of line 6. The resulting value will be $8n + 8$ when n is odd and $8n + 16$ when n is even, and this value is subtracted from s_1 to give s_2.

B. The three instructions in this sequence round s_2 up to the nearest multiple of 8. They make use of the combination of biasing and shifting that we saw for dividing by a power of 2 in Section 2.3.7.

C. These two examples can be seen as the cases that minimize and maximize the values of e_1 and e_2.

n	s_1	s_2	p	e_1	e_2
5	2,065	2,017	2,024	1	7
6	2,064	2,000	2,000	16	0

D. We can see that s_2 is computed in a way that preserves whatever offset s_1 has with the nearest multiple of 16. We can also see that p will be aligned on a multiple of 8, as is recommended for an array of 8-byte elements.

Solution to Problem 3.50 (page 336)

This exercise requires that you step through the code, paying careful attention to which conversion and data movement instructions are used. We can see the values being retrieved and converted as follows:

- The value at `dp` is retrieved, converted to an `int` (line 4), and then stored at `ip`. We can therefore infer that `val1` is `d`.
- The value at `ip` is retrieved, converted to a `float` (line 6), and then stored at `fp`. We can therefore infer that `val2` is `i`.
- The value of `l` is converted to a `double` (line 8) and stored at `dp`. We can therefore infer that `val3` is `l`.
- The value at `fp` is retrieved on line 3. The two instructions at lines 10–11 convert this to double precision as the value returned in register `%xmm0`. We can therefore infer that `val4` is `f`.

Solution to Problem 3.51 (page 336)

These cases can be handled by selecting the appropriate entries from the tables in Figures 3.47 and 3.48, or using one of the code sequences for converting between floating-point formats.

T_x	T_y	Instruction(s)
`long`	`double`	`vcvtsi2sdq %rdi, %xmm0, %xmm0`
`double`	`int`	`vcvttsd2si %xmm0, %eax`
`float`	`double`	`vunpcklpd %xmm0, %xmm0, %xmm0`
		`vcvtpd2ps %xmm0, %xmm0`
`long`	`float`	`vcvtsi2ssq %rdi, %xmm0, %xmm0`
`float`	`long`	`vcvttss2siq %xmm0, %rax`

Solution to Problem 3.52 (page 337)

The basic rules for mapping arguments to registers are fairly simple (although they become much more complex with more and other types of arguments [77]).

A. `double g1(double a, long b, float c, int d);`

Registers: `a` in `%xmm0`, `b` in `%rdi` `c` in `%xmm1`, `d` in `%esi`

B. `double g2(int a, double *b, float *c, long d);`

Registers: `a` in `%edi`, `b` in `%rsi`, `c` in `%rdx`, `d` in `%rcx`

C. `double g3(double *a, double b, int c, float d);`

Registers: `a` in `%rdi`, `b` in `%xmm0`, `c` in `%esi`, `d` in `%xmm1`

D. `double g4(float a, int *b, float c, double d);`

Registers: `a` in `%xmm0`, `b` in `%rdi`, `c` in `%xmm1`, `d` in `%xmm2`

Solution to Problem 3.53 (page 339)

We can see from the assembly code that there are two integer arguments, passed in registers `%rdi` and `%rsi`. Let us name these `i1` and `i2`. Similarly, there are two floating-point arguments, passed in registers `%xmm0` and `%xmm1`, which we name `f1` and `f2`.

We can then annotate the assembly code:

```
    Refer to arguments as i1 (%rdi), i2 (%esi)
                          f1 (%xmm0), and f2 (%xmm1)

    double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
    funct1:
      vcvtsi2ssq      %rsi, %xmm2, %xmm2    Get i2 and convert from long to float
      vaddss  %xmm0, %xmm2, %xmm0           Add f1 (type float)
      vcvtsi2ss       %edi, %xmm2, %xmm2    Get i1 and convert from int to float
      vdivss  %xmm0, %xmm2, %xmm0           Compute i1 / (i2 + f1)
      vunpcklps       %xmm0, %xmm0, %xmm0
      vcvtps2pd       %xmm0, %xmm0          Convert to double
      vsubsd  %xmm1, %xmm0, %xmm0           Compute i1 / (i2 + f1) - f2 (double)
      ret
```

From this we see that the code computes the value `i1/(i2+f1)-f2`. We can also see that `i1` has type `int`, `i2` has type `long`, `f1` has type `float`, and `f2` has type `double`. The only ambiguity in matching arguments to the named values stems from the commutativity of multiplication—yielding two possible results:

```
double funct1a(int p, float q, long r, double s);
double funct1b(int p, long q, float r, double s);
```

Solution to Problem 3.54 (page 339)

This problem can readily be solved by stepping through the assembly code and determining what is computed on each step, as shown with the annotations below:

```
    double funct2(double w, int x, float y, long z)
    w in %xmm0, x in %edi, y in %xmm1, z in %rsi
    funct2:
      vcvtsi2ss       %edi, %xmm2, %xmm2     Convert x to float
      vmulss  %xmm1, %xmm2, %xmm1            Multiply by y
      vunpcklps       %xmm1, %xmm1, %xmm1
      vcvtps2pd       %xmm1, %xmm2           Convert x*y to double
      vcvtsi2sdq      %rsi, %xmm1, %xmm1     Convert z to double
      vdivsd  %xmm1, %xmm0, %xmm0            Compute w/z
      vsubsd  %xmm0, %xmm2, %xmm0            Subtract from x*y
      ret                                    Return
```

We can conclude from this analysis that the function computes `y * x − w/z`.

Solution to Problem 3.55 (page 341)

This problem involves the same reasoning as was required to see that numbers declared at label `.LC2` encode 1.8, but with a simpler example.

We see that the two values are 0 and 1077936128 (`0x40400000`). From the high-order bytes, we can extract an exponent field of `0x404` (1028), from which we subtract a bias of 1023 to get an exponent of 5. Concatenating the fraction bits of the two values, we get a fraction field of 0, but with the implied leading value giving value 1.0. The constant is therefore $1.0 \times 2^5 = 32.0$.

Solution to Problem 3.56 (page 341)

A. We see here that the 16 bytes starting at address `.LC1` form a mask, where the low-order 8 bytes contain all ones, except for the most significant bit, which is the sign bit of a double-precision value. When we compute the AND of this mask with `%xmm0`, it will clear the sign bit of `x`, yielding the absolute value. In fact, we generated this code by defining `EXPR(x)` to be `fabs(x)`, where `fabs` is defined in `<math.h>`.

B. We see that the `vxorpd` instruction sets the entire register to zero, and so this is a way to generate floating-point constant 0.0.

C. We see that the 16 bytes starting at address `.LC2` form a mask with a single 1 bit, at the position of the sign bit for the low-order value in the XMM register. When we compute the EXCLUSIVE-OR of this mask with `%xmm0`, we change the sign of `x`, computing the expression `-x`.

Solution to Problem 3.57 (page 344)

Again, we annotate the code, including dealing with the conditional branch:

```
double funct3(int *ap, double b, long c, float *dp)
ap in %rdi, b in %xmm0, c in %rsi, dp in %rdx
funct3:
  vmovss   (%rdx), %xmm1                         Get d = *dp
  vcvtsi2sd       (%rdi), %xmm2, %xmm2           Get a = *ap and convert to double
  vucomisd        %xmm2, %xmm0                   Compare b:a
  jbe     .L8                                    If <=, goto lesseq
  vcvtsi2ssq      %rsi, %xmm0, %xmm0             Convert c to float
  vmulss  %xmm1, %xmm0, %xmm1                    Multiply by d
  vunpcklps       %xmm1, %xmm1, %xmm1
  vcvtps2pd       %xmm1, %xmm0                   Convert to double
  ret                                            Return
.L8:                                           lesseq:
  vaddss  %xmm1, %xmm1, %xmm1                    Compute d+d = 2.0 * d
  vcvtsi2ssq      %rsi, %xmm0, %xmm0             Convert c to float
  vaddss  %xmm1, %xmm0, %xmm0                    Compute c + 2*d
  vunpcklps       %xmm0, %xmm0, %xmm0
  vcvtps2pd       %xmm0, %xmm0                   Convert to double
  ret                                            Return
```

From this, we can write the following code for `funct3`:

```
double funct3(int *ap, double b, long c, float *dp) {
    int a = *ap;
    float d = *dp;
    if (a < b)
        return c*d;
    else
        return c+2*d;
}
```