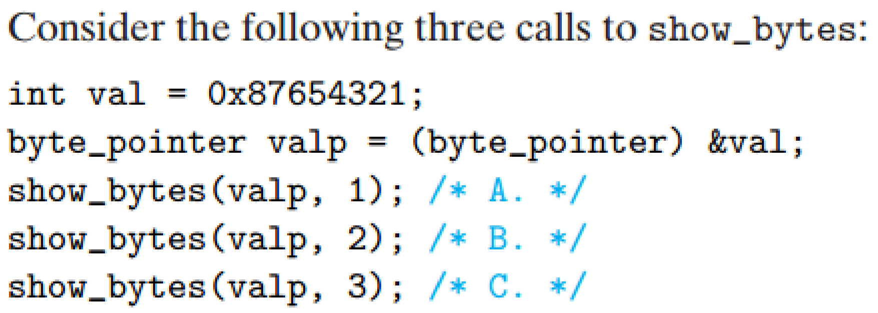
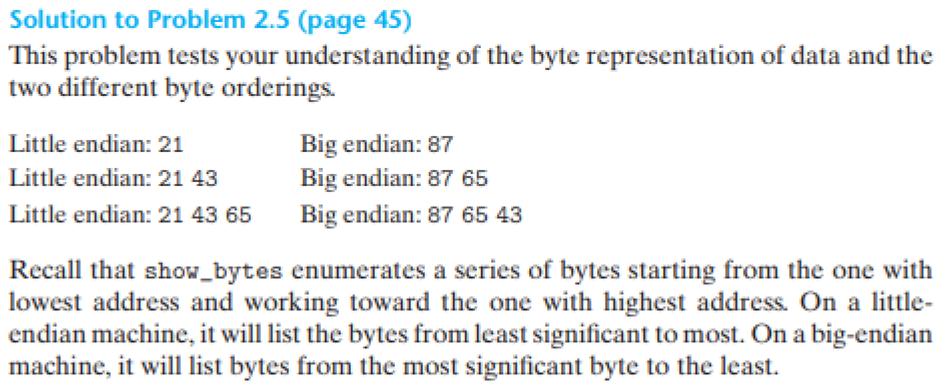
Exercícios do livro

2.5:



Consider the following three calls to show_bytes:

```
int val = 0x87654321;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```



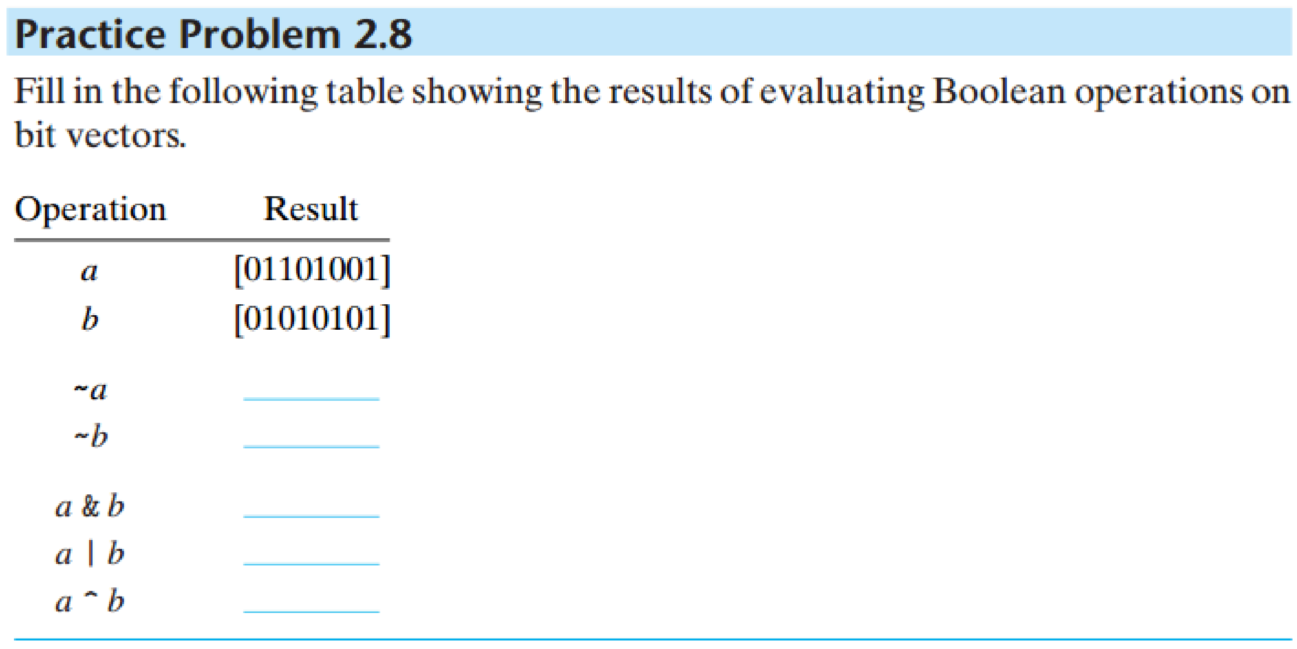
Solution to Problem 2.5 (page 45)

This problem tests your understanding of the byte representation of data and the two different byte orderings.

Little endian: 21	Big endian: 87
Little endian: 21 43	Big endian: 87 65
Little endian: 21 43 65	Big endian: 87 65 43

Recall that show_bytes enumerates a series of bytes starting from the one with lowest address and working toward the one with highest address. On a little-endian machine, it will list the bytes from least significant to most. On a big-endian machine, it will list bytes from the most significant byte to the least.

2.8



Practice Problem 2.8

Fill in the following table showing the results of evaluating Boolean operations on bit vectors.

Operation	Result
a	[01101001]
b	[01010101]
~a	
~b	
a & b	
a | b	
a ^ b	

Operation	Result
a	[01101001]
b	[01010101]
~a	[10010110]
~b	[10101010]
a & b	[01000001]
a | b	[01111101]
a ^ b	[00111100]

2.12:

Practice Problem 2.12

Write C expressions, in terms of variable x, for the following values. Your code should work for any word size $w \geq 8$. For reference, we show the result of evaluating the expressions for x = 0x87654321, with $w = 32$.

A. The least significant byte of x, with all other bits set to 0. [0x00000021].

B. All but the least significant byte of x complemented, with the least significant byte left unchanged. [0x789ABC21].

C. The least significant byte set to all 1s, and all other bytes of x left unchanged. [0x876543FF].

Solution to Problem 2.12 (page 53)

Here are the expressions:

A. `x & 0xFF`

B. `x ^ ~0xFF`

C. `x | 0xFF`

2.13

Practice Problem 2.13

The Digital Equipment VAX computer was a very popular machine from the late 1970s until the late 1980s. Rather than instructions for Boolean operations And and Or, it had instructions `bis` (bit set) and `bic` (bit clear). Both instructions take a data word `x` and a mask word `m`. They generate a result `z` consisting of the bits of `x` modified according to the bits of `m`. With `bis`, the modification involves setting `z` to 1 at each bit position where `m` is 1. With `bic`, the modification involves setting `z` to 0 at each bit position where `m` is 1.

To see how these operations relate to the C bit-level operations, assume we have functions `bis` and `bic` implementing the bit set and bit clear operations, and that we want to use these to implement functions computing bit-wise operations `|` and `^`, without using any other C operations. Fill in the missing code below. **Hint:** Write C expressions for the operations `bis` and `bic`.

```
/* Declarations of functions implementing operations bis and bic */
int bis(int x, int m);
int bic(int x, int m);

/* Compute x|y using only calls to functions bis and bic */
int bool_or(int x, int y) {
  int result = _________;
  return result;
}

/* Compute x^y using only calls to functions bis and bic */
int bool_xor(int x, int y) {
  int result = _________;
  return result;
}
```

Solution to Problem 2.13 (page 53)

These problems help you think about the relation between Boolean operations and typical ways that programmers apply masking operations. Here is the code:

```
/* Declarations of functions implementing operations bis and bic */
int bis(int x, int m);
int bic(int x, int m);

/* Compute x|y using only calls to functions bis and bic */
int bool_or(int x, int y) {
  int result = bis(x,y);
  return result;
}

/* Compute x^y using only calls to functions bis and bic */
int bool_xor(int x, int y) {
  int result = bis(bic(x,y), bic(y,x));
  return result;
}
```

The `bis` operation is equivalent to Boolean Or—a bit is set in `z` if either this bit is set in `x` or it is set in `m`. On the other hand, `bic(x, m)` is equivalent to `x&~m`; we want the result to equal 1 only when the corresponding bit of `x` is 1 and of `m` is 0.

Given that, we can implement `|` with a single call to `bis`. To implement `^`, we take advantage of the property

$$x \mathbin{\hat{}} y = (x \mathbin{\&} {\sim}y) \mid ({\sim}x \mathbin{\&} y).$$

2.14:

Practice Problem 2.14

Suppose that `x` and `y` have byte values `0x66` and `0x39`, respectively. Fill in the following table indicating the byte values of the different C expressions:

Expression	Value	Expression	Value
`x & y`	______	`x && y`	______
`x | y`	______	`x || y`	______
`~x | ~y`	______	`!x || !y`	______
`x & !y`	______	`x && ~y`	______

Solution to Problem 2.14 (page 54)

This problem highlights the relation between bit-level Boolean operations and logic operations in C. A common programming error is to use a bit-level operation when a logic one is intended, or vice versa.

Expression	Value	Expression	Value
`x & y`	`0x20`	`x && y`	`0x01`
`x | y`	`0x7F`	`x || y`	`0x01`
`~x | ~y`	`0xDF`	`!x || !y`	`0x00`
`x & !y`	`0x00`	`x && ~y`	`0x01`

2.15:

Practice Problem 2.15

Using only bit-level and logical operations, write a C expression that is equivalent to `x == y`. In other words, it will return 1 when `x` and `y` are equal, and 0 otherwise.

Solution to Problem 2.15 (page 54)

The expression is `!(x ^ y)`.

That is, `x^y` will be zero if and only if every bit of `x` matches the corresponding bit of `y`. We then exploit the ability of `!` to determine whether a word contains any nonzero bit.

There is no real reason to use this expression rather than simply writing `x == y`, but it demonstrates some of the nuances of bit-level and logical operations.

2.16:

Practice Problem 2.16

Fill in the table below showing the effects of the different shift operations on single-byte quantities. The best way to think about shift operations is to work with binary representations. Convert the initial values to binary, perform the shifts, and then convert back to hexadecimal. Each of the answers should be 8 binary digits or 2 hexadecimal digits.

x		x << 3		(Logical) x >> 2		(Arithmetic) x >> 2	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
0xC3	______	______	______	______	______	______	______
0x75	______	______	______	______	______	______	______
0x87	______	______	______	______	______	______	______
0x66	______	______	______	______	______	______	______

Solution to Problem 2.16 (page 56)

This problem is a drill to help you understand the different shift operations.

x		x << 3		(Logical) x >> 2		(Arithmetic) x >> 2	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
0xC3	[11000011]	[00011000]	0x18	[00110000]	0x30	[11110000]	0xF0
0x75	[01110101]	[10101000]	0xA8	[00011101]	0x1D	[00011101]	0x1D
0x87	[10000111]	[00111000]	0x38	[00100001]	0x21	[11100001]	0xE1
0x66	[01100110]	[00110000]	0x30	[00011001]	0x19	[00011001]	0x19

3.1:

Practice Problem 3.1

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the values for the indicated operands:

Operand	Value
%eax	______
0x104	______
$0x108	______
(%eax)	______
4(%eax)	______
9(%eax,%edx)	______
260(%ecx,%edx)	______
0xFC(,%ecx,4)	______
(%eax,%edx,4)	______

Solution to Problem 3.1 (page 170)

This exercise gives you practice with the different operand forms.

Operand	Value	Comment
%eax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%eax)	0xFF	Address 0x100
4(%eax)	0xAB	Address 0x104
9(%eax,%edx)	0x11	Address 0x10C
260(%ecx,%edx)	0x13	Address 0x108
0xFC(,%ecx,4)	0xFF	Address 0x100
(%eax,%edx,4)	0x11	Address 0x10C

3.2

Practice Problem 3.2

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, `mov` can be rewritten as `movb`, `movw`, or `movl`.)

```
mov   %eax, (%esp)
mov   (%eax), %dx
mov   $0xFF, %bl
mov   (%esp,%edx,4), %dh
push  $0xFF
mov   %dx, (%eax)
pop   %edi
```

```
movl   %eax, (%esp)
movw   (%eax), %dx
movb   $0xFF, %bl
movb   (%esp,%edx,4), %dh
pushl  $0xFF
movw   %dx, (%eax)
popl   %edi
```

3.3

Practice Problem 3.3

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

```
movb $0xF, (%bl)
movl %ax, (%esp)
movw (%eax),4(%esp)
movb %ah,%sh
movl %eax,$0x123
movl %eax,%dx
movb %si, 8(%ebp)
```

```
movb $0xF, (%bl)       Cannot use %bl as address register
movl %ax, (%esp)       Mismatch between instruction suffix and register ID
movw (%eax),4(%esp)    Cannot have both source and destination be memory references
movb %ah,%sh           No register named %sh
movl %eax,$0x123       Cannot have immediate as destination
movl %eax,%dx          Destination operand incorrect size
movb %si, 8(%ebp)      Mismatch between instruction suffix and register ID
```

3.4

Practice Problem 3.4

Assume variables v and p declared with types

```
src_t v;
dest_t *p;
```

where src_t and dest_t are data types declared with typedef. We wish to use the appropriate data movement instruction to implement the operation

```
*p = (dest_t) v;
```

where v is stored in the appropriately named portion of register %eax (i.e., %eax, %ax, or %al), while pointer p is stored in register %edx.

For the following combinations of src_t and dest_t, write a line of assembly code that does the appropriate transfer. Recall that when performing a cast that involves both a size change and a change of "signedness" in C, the operation should change the signedness first (Section 2.2.6).

src_t	dest_t	Instruction
int	int	movl %eax, (%edx)
char	int	________
char	unsigned	________
unsigned char	int	________
int	char	________
unsigned	unsigned char	________
unsigned	int	________

src_t	dest_t	Instruction
int	int	movl %eax, (%edx)
char	int	movsbl %al, (%edx)
char	unsigned	movsbl %al, (%edx)
unsigned char	int	movzbl %al, (%edx)
int	char	movb %al, (%edx)
unsigned	unsigned char	movb %al, (%edx)
unsigned	int	movl %eax, (%edx)

3.5

Practice Problem 3.5

You are given the following information. A function with prototype

```
void decode1(int *xp, int *yp, int *zp);
```

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

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

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

From this, we can generate the following C code:

```
void decode1(int *xp, int *yp, int *zp)
{
    int tx = *xp;
    int ty = *yp;
    int tz = *zp;

    *yp = tx;
    *zp = ty;
    *xp = tz;
}
```

3.6

Practice Problem 3.6

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

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

Instruction	Result
`leal 6(%eax), %edx`	$6 + x$
`leal (%eax,%ecx), %edx`	$x + y$
`leal (%eax,%ecx,4), %edx`	$x + 4y$
`leal 7(%eax,%eax,8), %edx`	$7 + 9x$
`leal 0xA(,%ecx,4), %edx`	$10 + 4y$
`leal 9(%eax,%ecx,2), %edx`	$9 + x + 2y$

3.7

Practice Problem 3.7

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the effects of the following instructions, both in terms of the register or memory location that will be updated and the resulting value:

Instruction	Destination	Value
`addl %ecx,(%eax)`		
`subl %edx,4(%eax)`		
`imull $16,(%eax,%edx,4)`		
`incl 8(%eax)`		
`decl %ecx`		
`subl %edx,%eax`		

Solution to Problem 3.7 (page 179)

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
`addl %ecx,(%eax)`	0x100	0x100
`subl %edx,4(%eax)`	0x104	0xA8
`imull $16,(%eax,%edx,4)`	0x10C	0x110
`incl 8(%eax)`	0x108	0x14
`decl %ecx`	%ecx	0x0
`subl %edx,%eax`	%eax	0xFD

3.8

Practice Problem 3.8

Suppose we want to generate assembly code for the following C function:

```
int shift_left2_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```

The code that follows is a portion of the assembly code that performs the actual shifts and leaves the final value in register %eax. Two key instructions have been omitted. Parameters x and n are stored at memory locations with offsets 8 and 12, respectively, relative to the address in register %ebp.

```
movl    8(%ebp), %eax     Get x
_____________             x <<= 2
movl    12(%ebp), %ecx    Get n
_____________             x >>= n
```

Fill in the missing instructions, following the annotations on the right. The right shift should be performed arithmetically.

```
movl    8(%ebp), %eax     Get x
sall    $2, %eax          x <<= 2
movl    12(%ebp), %ecx    Get n
sarl    %cl, %eax         x >>= n
```

3.9

Practice Problem 3.9

In the following variant of the function of Figure 3.8(a), the expressions have been replaced by blanks:

```
int arith(int x,
          int y,
          int z)
{
    int t1 = __________;
    int t2 = __________;
    int t3 = __________;
    int t4 = __________;
    return t4;
}
```

The portion of the generated assembly code implementing these expressions is as follows:

```
    x at %ebp+8, y at %ebp+12, z at %ebp+16
  movl    12(%ebp), %eax
  xorl    8(%ebp), %eax
  sarl    $3, %eax
  notl    %eax
  subl    16(%ebp), %eax
```

Based on this assembly code, fill in the missing portions of the C code.

Solution to Problem 3.9 (page 181)

This problem is fairly straightforward, since each of the expressions is implemented by a single instruction and there is no reordering of the expressions.

```
    int t1 = x^y;
    int t2 = t1 >> 3;
    int t3 = ~t2;
    int t4 = t3-z;
```

3.15

Practice Problem 3.15

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

A. What is the target of the `je` instruction below? (You don't need to know anything about the `call` instruction here.)

```
804828f:       74 05                   je      XXXXXXX
8048291:       e8 1e 00 00 00          call    80482b4
```

B. What is the target of the `jb` instruction below?

```
8048357:       72 e7                   jb      XXXXXXX
8048359:       c6 05 10 a0 04 08 01    movb    $0x1,0x804a010
```

C. What is the address of the `mov` instruction?

```
XXXXXXX:        74 12                   je     8048391
XXXXXXX:        b8 00 00 00 00          mov    $0x0,%eax
```

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 IA32. What is the address of the jump target?

```
80482bf:        e9 e0 ff ff ff          jmp    XXXXXXX
80482c4:        90                      nop
```

E. Explain the relation between the annotation on the right and the byte coding on the left.

```
80482aa:        ff 25 fc 9f 04 08       jmp    *0x8049ffc
```

A. The `je` instruction has as target `0x8048291` + `0x05`. As the original disassembled code shows, this is `0x8048296`:

```
804828f:        74 05                   je     8048296
8048291:        e8 1e 00 00 00          call   80482b4
```

B. The `jb` instruction has as target `0x8048359` – 25 (since `0xe7` is the 1-byte, two's-complement representation of –25). As the original disassembled code shows, this is `0x8048340`:

```
8048357:        72 e7                   jb     8048340
8048359:        c6 05 10 a0 04 08 01    movb   $0x1,0x804a010
```

C. According to the annotation produced by the disassembler, the jump target is at absolute address `0x8048391`. According to the byte encoding, this must be at an address `0x12` bytes beyond that of the `mov` instruction. Subtracting these gives address `0x804837f`, as confirmed by the disassembled code:

```
804837d:        74 12                   je     8048391
804837f:        b8 00 00 00 00          mov    $0x0,%eax
```

D. Reading the bytes in reverse order, we see that the target offset is `0xffffffe0`, or decimal –32. Adding this to `0x80482c4` (the address of the `nop` instruction) gives address `0x80482a4`:

```
80482bf:        e9 e0 ff ff ff          jmp    80482a4
80482c4:        90                      nop
```

E. An indirect jump is denoted by instruction code `ff 25`. The address from which the jump target is to be read is encoded explicitly by the following 4 bytes. Since the machine is little endian, these are given in reverse order as `fc 9f 04 08`.

3.18

Practice Problem 3.18

Starting with C code of the form

```
int test(int x, int y) {
    int val = ________;
    if (________) {
        if (________)
            val = ________;
        else
            val = ________;
    } else if (________)
        val = ________;
    return val;
}
```

GCC generates the following assembly code:

GCC generates the following assembly code:

```
    x at %ebp+8, y at %ebp+12
      movl     8(%ebp), %eax
      movl     12(%ebp), %edx
      cmpl     $-3, %eax
      jge      .L2
      cmpl     %edx, %eax
      jle      .L3
      imull    %edx, %eax
      jmp      .L4
    .L3:
      leal     (%edx,%eax), %eax
      jmp      .L4
    .L2:

      cmpl     $2, %eax
      jg       .L5
      xorl     %edx, %eax
      jmp      .L4
    .L5:
      subl     %edx, %eax
    .L4:
```

3.22

Practice Problem 3.22

A function, fun_a, has the following overall structure:

```
int fun_a(unsigned x) {
    int val = 0;
    while ( ________ ) {
        ________;
    }
    return ________;
}
```

The GCC C compiler generates the following assembly code:

```
   x at %ebp+8
     movl    8(%ebp), %edx
     movl    $0, %eax
     testl   %edx, %edx
     je      .L7
   .L10:
     xorl    %edx, %eax
     shrl    %edx                 Shift right by 1
     jne     .L10
   .L7:
     andl    $1, %eax
```

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. Describe in English what this function computes.

Solution to Problem 3.22 (page 202)

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

A. Here is the original C code:

```
int fun_a(unsigned x) {
    int val = 0;
    while (x) {
        val ^= x;
        x >>= 1;
    }
    return val & 0x1;
}
```

B. 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.

3.23

Practice Problem 3.23

A function `fun_b` has the following overall structure:

```
int fun_b(unsigned x) {
    int val = 0;
    int i;
    for ( ________ ; ________ ; ________ ) {
        ________
    }
    return val;
}
```

The GCC C compiler generates the following assembly code:

```
    x at %ebp+8
    movl    8(%ebp), %ebx
    movl    $0, %eax
    movl    $0, %ecx
  .L13:
    leal    (%eax,%eax), %edx
    movl    %ebx, %eax
    andl    $1, %eax
    orl     %edx, %eax
    shrl    %ebx              Shift right by 1
    addl    $1, %ecx
    cmpl    $32, %ecx
    jne     .L13
```

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. Describe in English what this function computes.

Solution to Problem 3.23 (page 205)

This problem is trickier than Problem 3.22, since the code within the loop is more complex and the overall operation is less familiar.

A. Here is the original C code:

```
int fun_b(unsigned x) {
    int val = 0;
    int i;
    for (i = 0; i < 32; i++) {
        val = (val << 1) | (x & 0x1);
        x >>= 1;
    }
    return val;
}
```

B. 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.

3.28

Practice Problem 3.28

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

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

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

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

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

Based on this information, answer the following questions:

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

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

A. The case labels in the `switch` statement body had values –2, 0, 1, 2, 3, and 4.

B. The case with destination `.L6` had labels 2 and 3.

3.29

Practice Problem 3.29

For a C function `switcher` with the general structure

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

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

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

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

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

```
int switcher(int a, int b, int c)
{
    int answer;
    switch(a) {
    case 5:
        c = b ^ 15;
        /* Fall through */
    case 0:
        answer = c + 112;
        break;
    case 2:
    case 7:
        answer = (c + b) << 2;
        break;
    case 4:
        answer = a;  /* equivalently, answer = 4 */
        break;
    default:
        answer = b;
    }
    return answer;
}
```

3.30

Practice Problem 3.30

The following code fragment occurs often in the compiled version of library routines:

```
    call next
  next:
    popl %eax
```

A. To what value does register `%eax` get set?

B. Explain why there is no matching `ret` instruction to this `call`.

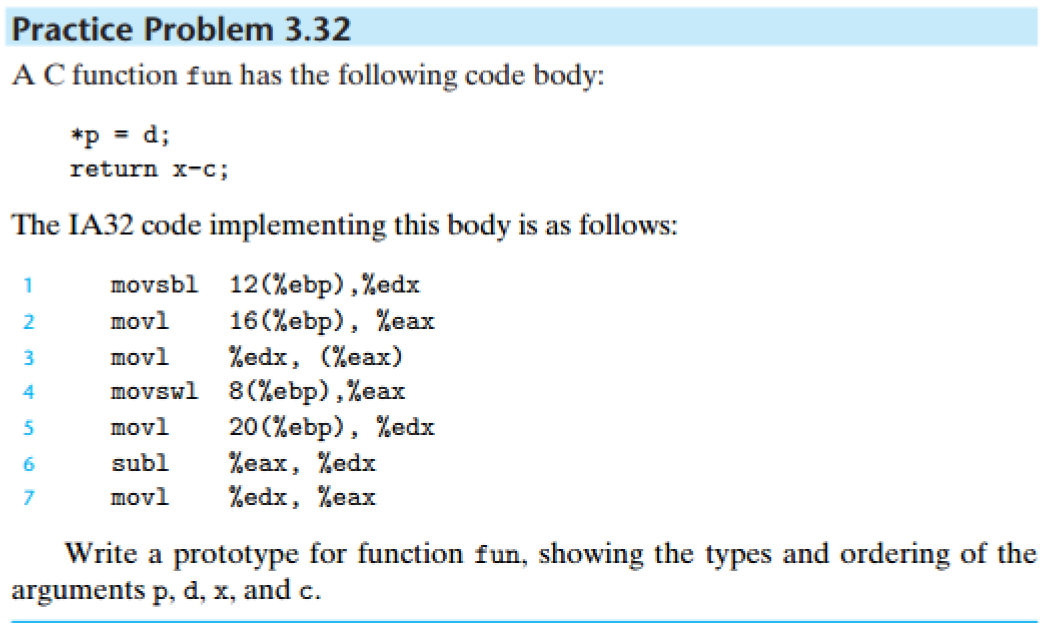
C. What useful purpose does this code fragment serve?

A. `%eax` is set to the address of the `popl` instruction.

B. This is not a true procedure call, since the control follows the same ordering as the instructions and the return address is popped from the stack.

C. This is the only way in IA32 to get the value of the program counter into an integer register.

3.32



Practice Problem 3.32

A C function `fun` has the following code body:

```
*p = d;
return x-c;
```

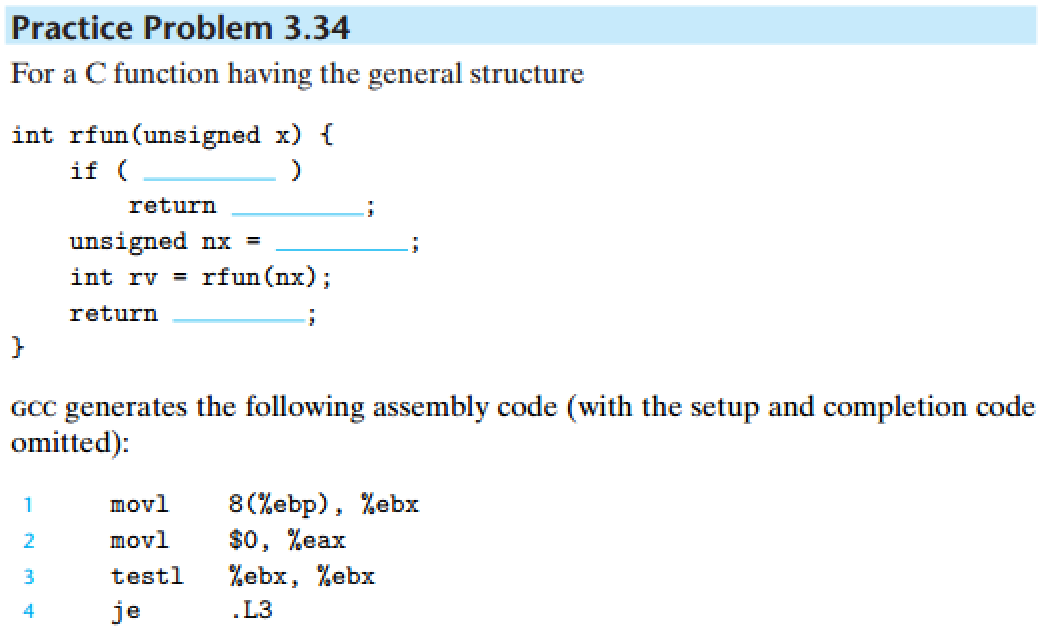
The IA32 code implementing this body is as follows:

```
    movsbl  12(%ebp),%edx
    movl    16(%ebp), %eax
    movl    %edx, (%eax)
    movswl  8(%ebp),%eax
    movl    20(%ebp), %edx
    subl    %eax, %edx
    movl    %edx, %eax
```

Write a prototype for function `fun`, showing the types and ordering of the arguments `p`, `d`, `x`, and `c`.

```
int fun(short c, char d, int *p, int x);
```

3.34



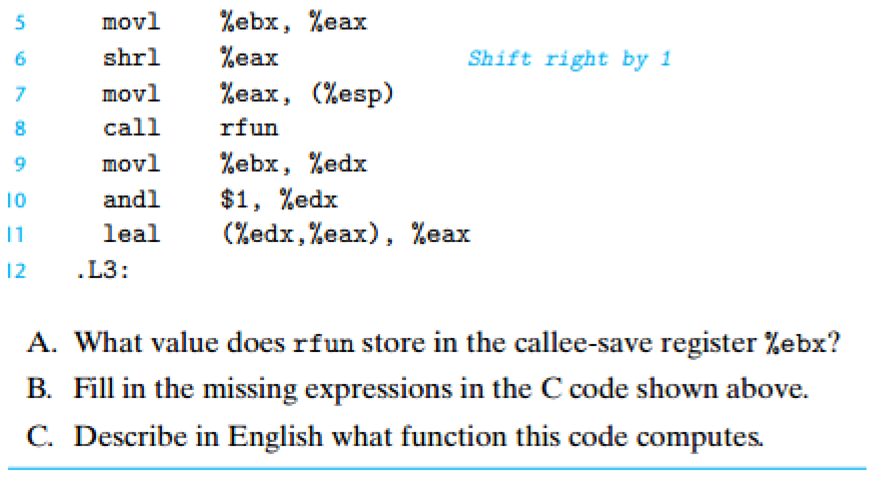
Practice Problem 3.34

For a C function having the general structure

```
int rfun(unsigned x) {
    if ( ________ )
        return ________;
    unsigned nx = ________;
    int rv = rfun(nx);
    return ________;
}
```

GCC generates the following assembly code (with the setup and completion code omitted):

```
    movl    8(%ebp), %ebx
    movl    $0, %eax
    testl   %ebx, %ebx
    je      .L3
```



```
    movl    %ebx, %eax
    shrl    %eax              Shift right by 1
    movl    %eax, (%esp)
    call    rfun
    movl    %ebx, %edx
    andl    $1, %edx
    leal    (%edx,%eax), %eax
  .L3:
```

A. What value does `rfun` store in the callee-save register `%ebx`?

B. Fill in the missing expressions in the C code shown above.

C. Describe in English what function this code computes.

Solution to Problem 3.34 (page 231)

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.

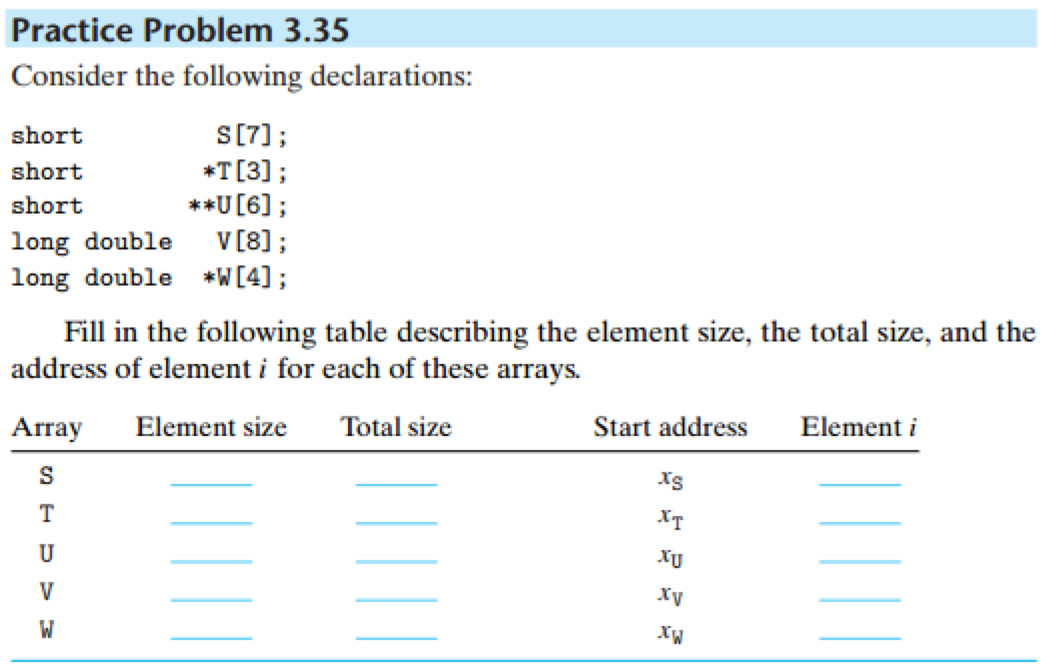
A. Register `%ebx` holds the value of parameter `x`, so that it can be used to compute the result expression.

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

```
int rfun(unsigned x) {
    if (x == 0)
        return 0;
    unsigned nx = x>>1;
    int rv = rfun(nx);
    return (x & 0x1) + rv;
}
```

C. Like the code of Problem 3.49, this function computes the sum of the bits in argument `x`. It recursively computes the sum of all but the least significant bit, and then it adds the least significant bit to get the result.

3.35



Practice Problem 3.35

Consider the following declarations:

```
short        S[7];
short       *T[3];
short      **U[6];
long double  V[8];
long double *W[4];
```

Fill in the following table describing the element size, the total size, and the address of element i for each of these arrays.

Array	Element size	Total size	Start address	Element i
`S`	______	______	$x_{\mathtt{S}}$	______
`T`	______	______	$x_{\mathtt{T}}$	______
`U`	______	______	$x_{\mathtt{U}}$	______
`V`	______	______	$x_{\mathtt{V}}$	______
`W`	______	______	$x_{\mathtt{W}}$	______

Array	Element size	Total size	Start address	Element i
`S`	2	14	$x_{\mathtt{S}}$	$x_{\mathtt{S}} + 2i$
`T`	4	12	$x_{\mathtt{T}}$	$x_{\mathtt{T}} + 4i$
`U`	4	24	$x_{\mathtt{U}}$	$x_{\mathtt{U}} + 4i$
`V`	12	96	$x_{\mathtt{V}}$	$x_{\mathtt{V}} + 12i$
`W`	4	16	$x_{\mathtt{W}}$	$x_{\mathtt{W}} + 4i$

3.36

Practice Problem 3.36

Suppose the address of short integer array S and integer index i are stored in registers %edx and %ecx, respectively. For each of the following expressions, give its type, a formula for its value, and an assembly code implementation. The result should be stored in register %eax if it is a pointer and register element %ax if it is a short integer.

Expression	Type	Value	Assembly code
S+1			
S[3]			
&S[i]			
S[4*i+1]			
S+i-5			

Expression	Type	Value	Assembly
S+1	short *	$x_S + 2$	leal 2(%edx),%eax
S[3]	short	$M[x_S + 6]$	movw 6(%edx),%ax
&S[i]	short *	$x_S + 2i$	leal (%edx,%ecx,2),%eax
S[4*i+1]	short	$M[x_S + 8i + 2]$	movw 2(%edx,%ecx,8),%ax
S+i-5	short *	$x_S + 2i - 10$	leal -10(%edx,%ecx,2),%eax

3.37

Practice Problem 3.37

Consider the following source code, where M and N are constants declared with #define:

```
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j) {
    return mat1[i][j] + mat2[j][i];
}
```

In compiling this program, GCC generates the following assembly code:

```
    i at %ebp+8, j at %ebp+12
      movl    8(%ebp), %ecx
      movl    12(%ebp), %edx
      leal    0(,%ecx,8), %eax
      subl    %ecx, %eax
      addl    %edx, %eax
      leal    (%edx,%edx,4), %edx
      addl    %ecx, %edx
      movl    mat1(,%eax,4), %eax
      addl    mat2(,%edx,4), %eax
```

Use your reverse engineering skills to determine the values of M and N based on this assembly code.

```
movl    8(%ebp), %ecx          Get i
movl    12(%ebp), %edx         Get j
leal    0(,%ecx,8), %eax       8*i
subl    %ecx, %eax             8*i-i = 7*i
addl    %edx, %eax             7*i+j
leal    (%edx,%edx,4), %edx    5*j
addl    %ecx, %edx             5*j+i
movl    mat1(,%eax,4), %eax    mat1[7*i+j]
addl    mat2(,%edx,4), %eax    mat2[5*j+i]
```

We can see that the reference to matrix `mat1` is at byte offset $4(7i + j)$, while the reference to matrix `mat2` is at byte offset $4(5j + i)$. From this, we can determine that `mat1` has 7 columns, while `mat2` has 5, giving $M = 5$ and $N = 7$.

3.39

Practice Problem 3.39

Consider the following structure declaration:

```
struct prob {
    int *p;
    struct {
        int x;
        int y;
    } s;
    struct prob *next;
};
```

This declaration illustrates that one structure can be embedded within another, just as arrays can be embedded within structures, and arrays can be embedded within arrays.

The following procedure (with some expressions omitted) operates on this structure:

```
void sp_init(struct prob *sp)
{
    sp->s.x   = __________;
    sp->p     = __________;
    sp->next  = __________;
}
```

A. What are the offsets (in bytes) of the following fields?

```
   p: ________
 s.x: ________
 s.y: ________
next: ________
```

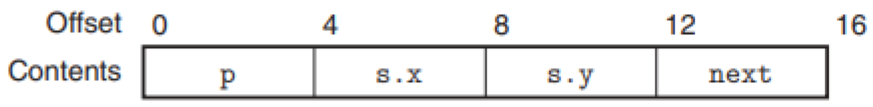
B. How many total bytes does the structure require?

C. The compiler generates the following assembly code for the body of sp_init:

```
    sp at %ebp+8
    movl    8(%ebp), %eax
    movl    8(%eax), %edx
    movl    %edx, 4(%eax)
    leal    4(%eax), %edx
    movl    %edx, (%eax)
    movl    %eax, 12(%eax)
```

On the basis of this information, fill in the missing expressions in the code for sp_init.

A. The layout of the structure is as follows:



B. It uses 16 bytes.

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

```
    sp at %ebp+8
    movl    8(%ebp), %eax      Get sp
    movl    8(%eax), %edx      Get sp->s.y
    movl    %edx, 4(%eax)      Store in sp->s.x
    leal    4(%eax), %edx      Compute &(sp->s.x)
    movl    %edx, (%eax)       Store in sp->p
    movl    %eax, 12(%eax)     Store sp in sp->next
```

From this, we can generate C code as follows:

```
void sp_init(struct prob *sp)
{
    sp->s.x    = sp->s.y;
    sp->p      = &(sp->s.x);
    sp->next   = sp;
}
```

3.40

Practice Problem 3.40

Suppose you are given the job of checking that a C compiler generates the proper code for structure and union access. You write the following structure declaration:

```
typedef union {
    struct {
        short  v;
        short  d;
        int    s;
    } t1;
    struct {
        int a[2];
        char  *p;
    } t2;
} u_type;
```

You write a series of functions of the form

```
void get(u_type *up, TYPE *dest) {
    *dest =  EXPR;
}
```

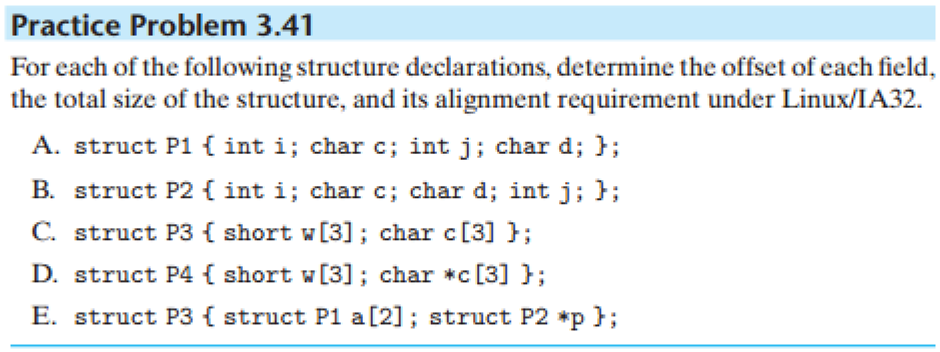
with different access expressions EXPR, and with destination data type TYPE set according to type associated with EXPR. You then examine the code generated when compiling the functions to see if they match your expectations.

Suppose in these functions that up and dest are loaded into registers %eax and %edx, respectively. Fill in the following table with data type TYPE and sequences of 1–3 instructions to compute the expression and store the result at dest. Try to use just registers %eax and %edx, using register %ecx when these do not suffice.

EXPR	TYPE	Code
up->t1.s	int	movl 4(%eax), %eax movl %eax, (%edx) ________
up->t1.v	____	________ ________ ________
&up->t1.d	____	________ ________ ________
up->t2.a	____	________ ________ ________
up->t2.a[up->t1.s]	____	________ ________ ________
*up->t2.p	____	________ ________ ________

EXPR	TYPE	Code
up->t1.s	int	movl 4(%eax), %eax movl %eax, (%edx)
up->t1.v	short	movw (%eax), %ax movw %ax, (%edx)
&up->t1.d	short *	leal 2(%eax), %eax movl %eax, (%edx)
up->t2.a	int *	movl %eax, (%edx)
up->t2.a[up->t1.s]	int	movl 4(%eax), %ecx movl (%eax,%ecx,4), %eax movl %eax, (%edx)
*up->t2.p	char	movl 8(%eax), %eax movb (%eax), %al movb %al, (%edx)

3.41



Practice Problem 3.41

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement under Linux/IA32.

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

B. `struct P2 { int i; char c; char d; int j; };`

C. `struct P3 { short w[3]; char c[3] };`

D. `struct P4 { short w[3]; char *c[3] };`

E. `struct P3 { struct P1 a[2]; struct P2 *p };`

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

`i`	`c`	`j`	`d`	Total	Alignment
0	4	8	12	16	4

B. `struct P2 { int i; char c; char d; int j; };`

`i`	`c`	`j`	`d`	Total	Alignment
0	4	5	8	12	4

C. `struct P3 { short w[3]; char c[3] };`

`w`	`c`	Total	Alignment
0	6	10	2

D. `struct P4 { short w[3]; char *c[3] };`

`w`	`c`	Total	Alignment
0	8	20	4

E. `struct P3 { struct P1 a[2]; struct P2 *p };`

`a`	`p`	Total	Alignment
0	32	36	4

3.42

Practice Problem 3.42

For the structure declaration

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

suppose it was compiled on a Windows machine, where each primitive data type of K bytes must have an offset that is a multiple of K.

A. What are the byte offsets of all the fields in the structure?

B. What is the total size of the structure?

C. Rearrange the fields of the structure to minimize wasted space, and then show the byte offsets and total size for the rearranged structure.

Solution to Problem 3.42 (page 251)

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	4	2	8	1	4	1	8	4
Offset	0	4	8	16	20	24	32	40

B. The structure is a total of 48 bytes long. The end of the structure must be padded by 4 bytes to satisfy the 8-byte alignment requirement.

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

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

with the following offsets, for a total of 32 bytes:

Field	c	g	e	a	h	b	d	f
Size	8	8	4	4	4	2	1	1
Offset	0	8	16	20	24	28	30	31