1. Book Problem 3.56

One way to analyze assembly code is to try to reverse the compilation process and produce C code that would look “natural” to a C programmer. For example, we wouldn’t want any goto statements, since these are seldom used in C. Most likely, we wouldn’t use a do-while statement either. This exercise forces students to reverse the compilation into a particular framework. It requires thinking about the translation of for loops.

A. We can see that result must be in register %edi, since this value gets copied to %eax at the end of the function as the return value (line 13). We can see that %esi and %ebx get loaded with the values of x and n (lines 1 and 2), leaving %edx as the one holding variable mask (line 4).

Looking for:
result in %edi
x in %esi
n in %ebx
mask in %edx

B. Register %edi (result) is initialized to −1 and %edx (mask) to 1.

C. The condition for continuing the loop (line 12) is that mask is nonzero i.e. mask != 0.

D. The shift instruction on line 10 updates mask to be mask << n.

E. Lines 6–8 update result to be result ^ (x & mask).

F. Here is the original code:

```c
int loop(int x, int n)
{
    int result = -1;
    int mask;
    for (mask = 0x1; mask != 0; mask = mask << n) {
        result ^= (x & mask);
    }
    return result;
}
```

2. Book Problem 3.58

This problem requires students to reason about the code fragments that implement the different branches of a switch statement. For this code, it also requires understanding different forms of pointer dereferencing.

A. In line 34, register %edx is copied to register %eax as the return value. From this, we can infer that %edx holds result.

B. The original C code for the function is as follows:

```c
/* Enumerated type creates set of constants numbered 0 and upward */
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

int switch3(int *p1, int *p2, mode_t action)
```
5 {  
6     int result = 0;  
7     switch(action) {  
8         case MODE_A:  
9             result = *p1;  
10            *p1 = *p2;  
11            break;  
12         case MODE_B:  
13            *p2 += *p1;  
14            result = *p2;  //if combine result=*p1*p2, then *p2 does not change  
15            break;  
16         case MODE_C:  
17            *p2 = 15;  
18            result = *p1;  
19            break;  
20         case MODE_D:  
21            *p2 = *p1;  
22            /* Fall Through - watch for no break statement */  
23            case MODE_E:  
24            result = 17;  
25            break;  
26            default:  
27                result = -1;  
28            }  
29        return result;  
30    }

3. Consider the following function’s assembly code:

```
bar2:
    push   %ebp
    movl   %esp,%ebp
    movl   8(%ebp),%eax
    leal   3(%eax),%edx
    testl  %eax,%eax
    cmovns %eax,%edx
    sarl   $2,%edx
    movl   $0,%eax
    testl  %edx,%edx
    jle    .L3

.L6:
    leal   3(%edx),%ecx
    testl  %edx,%edx
    cmovns %edx,%ecx
    sarl   $2,%edx
    addl   $1,%eax
    testl  %edx,%edx
    jg     .L6

.L3:
    popl   %ebp
    ret
```

```
00000000 <bar2>:
  0:    55            push   %ebp
  1:    89 e5         mov    %esp,%ebp
  3:    8b 45 08      mov     0x8(%ebp),%eax
  6:    8d 50 03      lea     0x3(%eax),%edx
  9:    85 c0         test    %eax,%eax
  1b:   0f 49 d1      cmovs   %ecx,%edx
  1e:   c1 fa 02      sar     $0x2,%edx
  1f:   b8 00 00 00 00 mov     $0x0,%eax
  6f:   85 d2         test    %edx,%edx
  77:   7e 12         jle     2c <bar2+0x2c>
  81:   8d 4a 03      lea     0x3(%edx),%ecx
  8e:   85 d2         test    %edx,%edx
  98:   0f 48 d1      cmovs   %ecx,%edx
  a0:   c1 fa 02      sar     $0x2,%edx
  a3:   83 c0 01      add     $0x1,%eax
  b0:   85 d2         test    %edx,%edx
  b4:   7f ee         jg      1a <bar2+0x1a>
  c0:   5d            pop     %ebp
  c2:   c3            ret
```
Fill in the corresponding C code:

```c
int bar(int x)
{
    int y = 0;
    int z = _________________;
    for(   ;  ___________  ;  ___________)
    {
        z = _________________;
    }
    return _________________;
}
```

4. Given the following assembly code:

```assembly
xorl %eax,%eax
leal 16(%ecx),%ebx
.L59:
    leal (%eax,%eax,4),%edx
    movl (%ecx),%eax
    addl $4,%ecx
    leal (%eax,%edx,2),%eax
    cmpl %ebx,%ecx
    jle .L59
```

Put comments on each assembly statement as it pertains to the C code then give equivalent C code. Here is some additional information that will help:

Register assignments:
- %ecx is z
- %eax is zi
- %ebx is zend

Computations:
- 10*zi + *z is implemented as *z + 2*(zi+4*zi)
- Z++ increments by 4
5. Using the template below (allowing a maximum of 40 bytes, indicate the allocation of data for struct my_struct. Mark off and label the areas for each individual element (arrays may be labeled as a single element) where each cell in the template is 1 byte. Shade the boxes used for padding i.e. the ones that are allocated, but not used; and be sure to clearly indicate the end of the structure. Use the letter of the variable to designate the space filled for each byte.

```
struct my_struct {
    short b;
    int x;
    short s;
    long z;
    char c[5]
    long long a;
    char q;
}
```

It was not stated if using x86 or ia32 (although I hope the majority assumed ia32 since that is the assembly language covered). There’s a practice problem that specifies 4 bytes for long and 8 bytes for long long for ia32; but 8 bytes for long and 8 bytes for long long for x86. Thus the change to the solution would be 8 “z” letters and the long long falling on the 4 byte boundaries pushing everything “over” by 4 bytes with no “empty” entries at the end.

<table>
<thead>
<tr>
<th>b</th>
<th>b</th>
<th>=</th>
<th>=</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>s</td>
<td>=</td>
<td>=</td>
<td>z</td>
<td>z</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>q</td>
<td>=</td>
<td>=</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
</tr>
</tbody>
</table>

= means shaded

Grade the comments:

NOTE: *z is the dereferencing operator (not multiply)
6. Below is the C code and assembly code for simple function. Draw a detailed stack diagram for this function, starting with a function that calls this function and continuing for 2 recursive calls of this function i.e. at least two stack frames that belong to this function. Be sure to label everything you can so your solution is understandable.

```
000000af <doSomething>:
    int doSomething(int a, int b, int c) {
        int d;
        if (a == 0) { return 1; }
        d = a/2;
        c = doSomething(d, a, c);
        return c;
    }
```

This stack starts from the top of the given list and “grows down” toward decreasing addresses.
7. Give the IA32 instruction format for each of the following assembly statements.

A. `push %ebp`
B. `sub $0x24, %esp`
C. `add $0xffffffff8, %esp`
D. `lea 0xf(%ebp), %ebx`
E. `push $0x804857b`
F. `test %eax, %eax`
G. `pop %ebp`
H. `ret`

**ANSWERS:**

A. 55
B. 83 ec 24
C. 83 c4 f8  
   okay if have f8 followed by ff ff
D. 8d 5d f8
E. 68 7b 85 04 08  
   Constant value different than below  
   i.e. 7b instead of 78
F. 85 c0
G. 5d
H. c3

This was a practice problem converted to a practice encoding problem :o)