1. Book Problem 3.54

This is an example of a problem that requires students to reverse engineer the actions of GCC. We have found that reverse engineering is a good way to learn about both compilers and machine-level programs.

```c
1 int decode2(int x, int y, int z)
2 {
3     int t1 = y - z;
4     int t2 = x * t1;
5     int t3 = (t1 << 31) >> 31;
6     int t4 = t3 ^ t2;
7
8     return t4;
9 }
```

2. 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.)

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.

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
1 int loop(int x, int n)
2 {
3     int result = -1;
4     int mask;
5     for (mask = 0x1; mask != 0; mask = mask << n) {
6         result ^= (x & mask);
7     }
8     return result;
9 }
```
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
#include <stdio.h>

int switch3(int *p1, int *p2, mode_t action)
{
    int result = 0;
    switch(action) {
        case MODE_A:
            result = *p1;
            *p1 = *p2;
            break;
        case MODE_B:
            *p2 += *p1;
            result = *p2;
            break;
        case MODE_C:
            *p2 = 15;
            result = *p1;
            break;
        case MODE_D:
            *p2 = *p1;
            /* Fall Through */
        case MODE_E:
            result = 17;
            break;
        default:
            result = -1;
    }
    return result;
}
```

4. Consider the following function’s assembly code:
int bar(int x)
{
    int y = 0;
    int z = ______________;
    for(   ;  ___________  ;  ___________)
    {
        z = ______________;
    }
    return  ______________;
}

int bar2(int x)
{
    int y = 0;
    int z = x/4;
    for( ; z > 0 ; y++)
        z = z/4;
    return y;
}
5. 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

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```
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;
}
```

= means shaded
7. 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.

```c
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;
}
```

```assembly
000000af <doSomething>:  push %ebp
    mov %esp,%ebp
    sub $0xc,%esp
    mov 0x8(%ebp),%ecx
    mov $0x1,%eax
    test %ecx,%ecx
    je de <doSomething+0x2f>
    mov %ecx,%edx
    shr $0x1f,%edx
    lea (%ecx,%edx,1),%edx
    sar %edx
    mov 0x10(%ebp),%eax
    mov %eax,0x8(%esp)
    mov %ecx,0x4(%esp)
    mov %edx,(%esp)
    call da <doSomething+0x2b>
    leave
    ret
```

```
ret Addr to caller
old ebp
eax (c)
ecx (b)
edx (a)
0xde
old ebp
eax (c)
ecx (b)
edx (a)
0xde
old ebp <- %ebp
blank
blank
black <- %esp
```
8. 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 0xffffffff8(%ebp), %ebx
E. push $0x804857b
F. test %eax, %eax
G. pop %ebp
H. ret

```
80484ac <check_password>:
80484ac: 55 push %ebp
80484ad: 89 e5 mov %esp,%ebp
80484af: 83 ec 24 sub $0x24,%esp
80484b2: 53 push %ebx
80484b3: 83 c4 f8 add $0xffffffff8,%esp
80484b6: 8d 5d f8 lea 0xffffffff8(%ebp),%ebx
80484b9: 53 push %ebx
80484ba: 68 78 85 04 08 push $0x8048578
80484bf: e8 a0 ff ff ff call 8048364 <scanf>
80484c4: 83 c4 f8 add $0xffffffff8,%esp
80484c7: 68 7b 85 04 08 push $0x8048578
80484cc: 53 push %ebx
80484cd: e8 be ff ff ff call 8048490 <string_compare>
80484d2: 83 c4 20 add $0x20,%esp
80484d5: 85 c0 test %eax,%eax
80484d7: 74 0a je 80484e3 <check_password+0x37>
80484d9: 83 c4 f4 add $0xffffffff4,%esp
80484dc: 6a 01 push $0x1
80484de: e8 c1 fe ff ff call 80483a4 <exit>
80484e3: 8b 5d d8 mov 0xffffffff8(%ebp),%ebx
80484e6: 89 ec mov %ebp,%esp
80484e8: 5d pop %ebp
80484e9: c3 ret
```