1. (On decompiling arithmetic operations.) Solve problem 3.58, on page 311, from our CMU text. *(Hint: you may want to introduce temporary C variables for different computations.)*

**Solution to 3.58:**

```c
1 long decode2(long x, long y, long z)
2 {
3    long t1 = y - z;
4    long t2 = x * t1;
5    long t3 = t1<<63>>63;
6    long t4 = t3 ^ t2;
7
8    return t4;
9 }
```

2. (On conditional statements.) Solve problem 3.61, on page 313, from our CMU text. *(Hint: you may need to experiment with running GCC on different versions of your code. Compile your code with the -O1 flag.)*

**Solution to 3.61:**

The idea of our solution is to set up a local variable having value 0, and then using a conditional move to overwrite xp with the address of this variable when xp is null.

```c
1 long cread_fix(long *xp) {
2    long zero = 0;
3    if (!xp) xp = &zero;
4    return *xp;
5 }
```

3. (On decompiling a C “for” loop.) Solve problem 3.60, on page 312, from our CMU text. *(Hint: note that register %cl is embedded in registers %ecx and %rcx.)*

**Solution to 3.60:**
A. We can see that result must be in register %rax, since this value gets returned as the final value. Parameter x is passed in %rdi. Parameter n is passed in %esi and then copied into %ecx. Register %edx is initialized to 1. We can infer that mask must be %rdx.

B. They are initialized to 0 and 1, respectively.

C. The condition for continuing the loop is that mask is nonzero.

D. The salq instruction updates mask to be

\[
\text{mask} \leftarrow \text{mask} \ll n.
\]

E. Variable result is updated to be

\[
\text{result} \leftarrow \text{result} \lor (x \& \text{mask}).
\]

F. Here is the original code:

```c
1 long loop(long x, int n)
2 {
3     long result = 0;
4     long mask;
5     for (mask = 0x1; mask != 0; mask = mask << n) {
6         result |= (x & mask);
7     }
8     return result;
9 }
```

4. (On decompiling a C “switch” statement.) Solve problem 3.62, on page 313, from our CMU text.

Solution to 3.62:

```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;

long switch3(long *p1, long *p2, mode_t action)
{
    long result = 0;

    switch(action) {

    case MODE_A:
        result = *p2;
        *p2 = *p1;
        break;
```
case MODE_B:
    *p1 += *p2;
    result = *p1;
    break;

case MODE_C:
    *p1 = 59;
    result = *p2;
    break;

case MODE_D:
    *p1 = *p2;
    /* Fall Through */

case MODE_E:
    result = 27;
    break;

default:
    result = 12;
}

return result;
}

5. (On array layout and access.) Solve problem 3.65, on page 317, from our CMU text.

Solution to 3.65:

A. We can see in the code that registers %rdx and %rax are being used as pointers. The only question is which one represents which matrix element. We can see in line 6 that %rdx gets incremented by 8. This must be a pointer to A[i][j].

B. That leaves %rax as a pointer to A[j][i].

C. Since %rax is incremented by 120 on each iteration (line 7), we must have M = 120/8 = 15.
6. (On structures and stack discipline.) Solve problem 3.67, on page 318, from our CMU text.

**Solution to 3.67:**

The best way to solve this problem is to examine the code for eval, seeing how it sets up the call of process. See annotated code for eval below.

```assembly
long eval(long x, long y, long z)
   x in %rdi, y in %rsi, z in %rdx
1   eval:
2   subq $104, %rsp       Allocate 104 bytes on stack
3   movq %rdx, 24(%rsp)   Store z at %rsp+24
4   leaq 24(%rsp), %rax   Compute &z
       Build s starting at %rsp
5   movq %rdi, (%rsp)     s.a[0] = x
6   movq %rsi, 8(%rsp)    s.a[1] = y
7   movq %rax, 16(%rsp)   s.p = &z
       Allocate space for r starting at %rsp+64
8   leaq 64(%rsp), %rdi   Compute &r
9   call process          Call process
10  movq 72(%rsp), %rax   Retrieve r.u[1]
11  addq 64(%rsp), %rax   Add r.u[0]
12  addq 80(%rsp), %rax   Add r.q
13  addq $104, %rsp       Deallocate stack space
14  ret
```

From this, we can draw a diagram of the stack frame for eval, as shown below.

![Stack Diagram](image)

With that information, the code for process is much easier to understand – see annotated code below.

A. We can see that eval passes s to process, using 24 bytes at the top of the stack, to store
s.a[0]=x, s.a[1]=y, and s.p=&z. It also stores argument z (in register %rdx) on the stack at offset 24.

B. Function eval allocates 24 bytes on the stack at offset 64 for the result structure r. It passes a pointer to this region as an argument to process in register %rdi.

C. Function process accesses the fields of argument structure s on the stack. Since the callq instruction pushed an 8-byte return address, the fields of s start at offset 8.

D. Function process sets the fields of the result structure via the pointer passed as an argument.

E. See above diagram, where r.u[0]=y, r.u[1]=x, and r.q=z.

F. Structure arguments are passed on the stack. When calling a function that will return a structure, the caller allocates space on its stack and passes a pointer to this region as an argument to the function.

```
process:
  movq  %rdi, %rax    ; Set return value
  movq  24(%rsp), %rdx ; Retrieve s.p
  movq  (%rdx), %rdx   ; Get *s.p
  movq  16(%rsp), %rcx ; Get s.a[1]
  movq  %rcx, (%rdi)   ; Copy to r.u[0]
  movq  8(%rsp), %rcx  ; Get s.a[0]
  movq  %rcx, 8(%rdi)  ; Copy to r.u[1]
  movq  %rdx, 16(%rdi) ; Set r.q to *s.p
ret
```