SP 2023-2 Assignment-02 Problems

3-1.
Consider the following assembly code:

```
long loop(long x, int n)
x in %rdi, n in %esi
loop:
   movl %esi, %ecx
   movl $1, %edx
   movl $0, %eax
   jmp .L2
.L3:
   movq %rdi, %r8
   andq %rdx, %r8
   addq %r8, %rax
   salq %cl, %rdx
.L2:
   testq %rdx, %rdx
   jne .L3
   rep; ret
```

The preceding code was generated by corμpiling C code that had the following overall form:

```
long loop(long x, long n)
{
   long result = __________;
   long mask;
   for (mask = __________; mask __________; mask = __________)
      result = __________;
   return result;
}
```

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register %rax. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables. Fill in all the missing parts of the C code.

#

3-2.

The code that follows shows an example of branching on an enumerated type value in a switch statement. Recall that enumerated types in C are simply a way to introduce a set of names having associated integer values. By default, the values assigned to the names count from zero upward. In our code, some of the actions associated with the different case labels have been omitted.

```
/* 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 switch_ex(long *p1, long *p2, mode_t action)
{
   long result = 0;
   switch(action) {
   case MODE_A:
      result = *p2;
      [ ① ];
      break;
   case MODE_B:
      [ ② ]
      result = *p2;
      break;
   case MODE_C:
      *p1 = 59;
      [ ③ ];
      break;
   case MODE_D:
      [ ④ ];
      /* Fall Through */
   case MODE_E:
      [ ⑤ ];
      break;
   default:
      result = 12;
   }
   return result;
}
```

The part of the generated assembly code implementing the different actions is shown as follows. The annotations indicate the argument locations, the register values, and the case labels for the different jump destinations. Fill in the missing parts of the C code.

```
p1 in %rdi, p2 in %rsi, action in %rdx
.L8:                          MODE_E
   movl $27, %eax
   ret
.L3:                          MODE_A
   movq (%rsi), %rax
   movq (%rdi), %rdx
   movq %rdx, (%rsi)
   ret
.L5:                          MODE_B
   movq (%rdi), %rax
   addq (%rsi), %rax
   movq %rax, (%rsi)
   ret
.L6:                          MODE_C
   movq $59, (%rdi)
   movq 8(%rdi), %rax
   ret
.L7:                          MODE_D
   movq (%rsi), %rax
   movq %rax, (%rdi)
   movl $27, %eax
   ret
.L9:                          default
   movl $12, %eax
   ret
```

#

3-3.
Consider the following source code, where R, S, and T are constants declared with #define.

```
long A[R][S][T];

long store_elet(long i, long j, long k, long *dest) {
   *dest = A[i][j][k];
   return sizeof(A);
}
```

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

```
long store_elet(long i, long j, long k, long *dest)
i in %rdi, j in %rsi, k in %rdx, dest in %rcx
store_elet:
   leaq (%rsi,%rsi,2), %rax
   leaq (%rsi,%rax,4), %rax
   movq %rdi, %rsi
   salq $6, %rsi
   addq %rsi, %rdi
   addq %rax, %rdi
   addq %rdi, %rdx
   movq A(,%rdx,8), %rax
   movq %rax, (%rcx)
   movl $5200, %eax
   ret
```

A. Extend the following equation from two dimensions to three to provide a formula for the location of array element A[i][j][k].

T A[R][C];
&A[i][j] = x_A + sizeof(T)·(C·i + j)

B. Use your reverse engineering skills to determine the values of R, S, and T based on the assembly code.

#

3-4.
The following code transposes the elements of an M × M array, where M is a constant defined by #define:

```
void transpose_mat(long A[M][M]) {
   long i, j;
   for (i = 0; i < M; i++)
      for (j = 0; j < i; j++) {
         long t = A[i][j];
         A[i][j] = A[j][i];
         A[j][i] = t;
      }
}
```

When compiled with optimization level –O1, gcc generates the following code for the inner loop of the function:

```
.L6:
   movq (%rdx), %rcx
   movq (%rax), %rsi
   movq %rsi, (%rdx)
   movq %rcx, (%rax)
   addq $8, %rdx
   addq $160, %rax
   cmpq %rdi, %rax
   jne .L6
```

We can see that gcc has converted the array indexing to pointer code. What is the value of M?

#

3-5.
Consider the following source code, where NR and NC are macro expressions declared with #define that compute the dimensions of array A in terms of parameter n. This code computes the sum of the elements of column j of the array.

```
long sum_col(long n, long A[NR(n)][NC(n)], long j) {
   long i;
   long result = 0;
   for (i = 0; i < NR(n); i++)
      result += A[i][j];
   return result;
}
```

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

```
long sum_col(long n, long A[NR(n)] [NC(n)], long j)
n in %rdi, A in %rsi, j in %rdx
sum_col:
   leaq 2(,%rdi,8), %r8
   leaq (%rdi,%rdi,4), %rax
   movq %rax, %rdi
   testq %rax, %rax
   jle .L4
   salq $3, %r8
   leaq (%rsi,%rdx,8), %rcx
   movl $0, %eax
   movl $0, %edx
.L3:
   addq (%rcx), %rax
   addq $1, %rdx
   addq %r8, %rcx
   cmpq %rdi, %rdx
   jne .L3
   rep; ret
.L4:
   movl $0, %eax
   ret
```

Use your reverse engineering skills to determine the definitions of NR and NC. Show the macro definitions of NR and NC with #define.

#

3-6.
For this exercise, we will examine the code generated by GCC for functions that have structures as arguments and return values, and from this see how these language features are typically implemented.
The following C code has a function process having structures as argument and return values, and a function eval that calls process:

```
typedef struct {
   long a[2];
   long *p;
} strA;

typedef struct {
   long u[2];
   long q;
} strB;

strB process(strA s) {
   strB r;
   r.u[0] = s.a[1];
   r.u[1] = s.a[0];
   r.q = *s.p;
   return r;
}

long eval(long x, long y, long z) {
   strA s;
   s.a[0] = x;
   s.a[1] = y;
   s.p = &z;
   strB r = process(s);
   return r.u[0] + r.u[1] + r.q;
}
```

GCC generates the following code for these two functions:

```
strB process(strA s)
process:
    movq %rdi, %rax
    movq 24(%rsp), %rdx
    movq (%rdx), %rdx
    movq 16(%rsp), %rcx
    movq %rcx, (%rdi)
    movq 8(%rsp), %rcx
    movq %rcx, 8(%rdi)
    movq %rdx, 16(%rdi)
    ret

long eval(long x, long y, long z)
x in %rdi, y in %rsi, z in %rdx
eval:
    subq $104, %rsp
    movq %rdx, 24(%rsp)
    leaq 24(%rsp), %rax
    movq %rdi, (%rsp)
    movq %rsi, 8(%rsp)
    movq %rax, 16(%rsp)
    leaq 56(%rsp), %rdi
    call process
    movq 64(rsp), %rax
    addq 56(%rsp), %rax
    addq 72(%rsp), %rax
    addq $104, %rsp
    ret
```

A. We can see on line 2 of function `eval` that it allocates 104 bytes on the stack. Diagram the stack frame for `eval`, showing the values that it stores on the stack prior to calling `process`, and complete your diagram of the stack frame for `eval`, showing how `eval` accesses the elements of structure `r` following the return from `process`.
B. What general principles can you discern about how structure values are passed as function arguments and how they are returned as function results?

#

3-7.
You are charged with maintaining a large C program, and you come across the following code:

```
typedef struct {
   int first;
   a_struct a[CNT];
   int last;
} b_struct;

void test (long i, b_struct *bp)
{
   int n = bp->first + bp->last;
   a_struct *ap = &bp->a[i];
   ap->x[ap->idx] = n;
}
```

The declarations of the compile-time constant CNT and the structure a_struct are in a file for which you do not have the necessary access privilege. Fortunately, you have a copy of the .o version of code, which you are able to disassemble with the OBJDUMP program, yielding the following disassembly:

```
void test(long i, b_struct *bp)
i in %rdi, bp in %rsi
0000000000000000 <test>:
   0: 8b 8e 20 01 00 00        mov    0x170(%rsi),%ecx
   6: 03 0e                    add    (%rsi),%ecx
   8: 48 8d 04 bf              lea    (%rdi,%rdi,4),%rax
   c: 48 8d 04 c6              lea    (%rsi,%rax,8),%rax
  10: 48 8b 50 08              mov    0x8(%rax),%rdx
  14: 48 63 c9                 movslq %ecx,%rcx
  17: 48 89 4c d0 10           mov %rcx,0x10(%rax,%rdx,8)
  1c: c3                       retq
```

Using your reverse engineering skills, deduce the following:

A. The value of CNT.
B. The size of a_struct and the size of b_struct.
C. A complete declaration of structure a_struct. Assume that the only fields in this structure are idx and x, and that both of these contain signed values.

#

3-8.
Starting with C code of the form

```
long test(long x, long y, long z) {
   long val = __________;
   if (__________) {
      if (__________)
         val = __________;
      else
         val = __________;
   } else if (__________)
      val = __________;
   return val;
}
```

GCC generates the following assembly code:

```
      long test(long x, long y, long z)
      x in %rdi, y in %rsi , z in %rdx
   test:
      leaq      (%rdi,%rsi), %rax
      subq      %rdx, %rax
      cmpq      $6, %rdi
      jle       .L2
      cmpq      %rdx, %rsi
      jl        .L3
      leaq      (,%rax,2), %rax
      ret
   .L3:
      leaq      (,%rax,4), %rax
      imulq     %rdx, %rax
      ret
   .L2:
      cmpq      $0, %rdi
      jle       .L4
      imulq     %rsi, %rax
   .L4:
      rep; ret
```

Fill in the missing expressions in the C code.

#