1. (5 points) Circle the correct answer among the choices given. If you circle more than one answer, you will lose the grade of the corresponding question.

(A) If we write a C program that consists of 5 C files. The output of the assembler will be:
   1. five files  2. one file  3. depends on the type of the compiler  4. depends on OS

(B) By seeing the number: 0xFFFF700B we know for sure that it is a:
   1. negative number   2. positive number   3. unsigned number   4. We do not know for sure

(C) Suppose we have a 32-bit machine. The size of “long int *” is:
   1. 4 bytes   2. 8 bytes   3. 2 bytes   4. Depends on the OS.

(D) Suppose we have a 64-bit machine. The size of “long int *” is:
   1. 4 bytes   2. 8 bytes   3. 2 bytes   4. Depends on the OS.

(E) If we write a C program that includes a parenthesis that we opened but forgot to close. Then:
   1. the compiler will complain   2. the assembler will complain
   2. the linker will complain   3. the loader will complain
2. [4 points] We have seen that the floating point presentation has normalized encoding and denormalized encoding. State two reasons we need denormalized encoding. Every reason must not be more than one sentence.
   - **To be able to present 0**
   - **To present very small numbers.**

3. [8 points] Suppose you want to include this condition in your C code: `if( x & mask)`
   x is a char. You want the condition to be true if the third bit from the left of x is set to 1.
   - What value **mask** must have in binary?
     
     00100000
   - What value **mask** must have in hexadecimal?
     
     0x20
   - Suppose x = 5, will the condition be true? Show the value of x & mask in binary to justify.
     
     0000101 & 00100000 = 00000000 \rightarrow \text{condition is false}
   - What if x = -5? Show the value if x in binary to justify.
     
     11111011 & 00100000 = 00100000 \rightarrow \text{condition is true}
4. Suppose we have the following piece of C code (%p in printf just prints the address in hex):

```c
void foo(int i)
{
    char a[2];
    double d = 3.14;

    a[i] = 0xFFFFABCD;
    printf("%d", d);
}
```

a. [2 points] How many bytes does array `a` require?

2 bytes

b. [2 points] Suppose array `a` is stored at address A1. What is the address of `a[0]`? What is the address of `a[1]`?

`a[0]` will be stored in A1
`a[1]` will be stored in A1+1

c. [3 points] Suppose array `a` is stored at address A1. Will variable `d` always be stored in memory after array `a`? Justify.

Not necessarily. It depends on whether the OS will find an empty spot, big enough, for `d` after the array `a`.

d. [3 points] Something is wrong with the line: `a[i] = 0xFFFFABCD;` What is it?

`a` is an array of characters. The number assigned to `a[i]` is outside the range that a character can present.

e. If we call the function `foo` as follows: `foo(2);`

   - [3 points] What will happen?

   You will overwrite whatever is stored after array `a` in memory. The compiler may not complain especially if `foo()` is called at runtime with a number not known during compilation.

   - [3 points] Will this affect the value of `d`? Justify

   If `d` is stored just after array `a`, it will be affected. Otherwise, no.
5. Given the following C code:

```c
int compute(int a, int b)
{
    int c;

    while (b != 0) {
        c = (a & b) << 1;
        a=a^b;
        b=c;
    }

    return a;
}
```

a. [4 points] Suppose the two inputs to the above function are: a = 10 and b = 20. What will the function return (both in decimal and in binary)?

It will return 0000…011110 → 30 in decimal

b. [3 points] Given the above two inputs (a = 10, b = 20), how many times will the above loop execute? For each iteration, write down the value of c.

The loop will be executed once. c will have the value 0

c. [4 points] Assume x = 5 and we call compute(~x, 1), what will be the value returned (in binary and decimal). [Hint, you can use dots “…” to represent repeated bits when you state the binary result]

5 → 00000….0101 ~5 → 11111…1010
the output will be 111111..1011 → -5

d. [6 points] Suppose we have two pointer int * k and int * m declared inside compute. Write three statements: one to make k point to a, the second to make m point to b, and the third to execute statement a=a^b; using only k and m.

k = &a; m = &b; *k = (*k)^(*m);