CSCI-UA.0201-003

Computer Systems Organization

Exam 1 Fall 2018 (time: 60 minutes)

Last name: First name: NetID:

Notes:

- **If you perceive any ambiguity in any of the questions, state your assumptions clearly.**
- **Questions vary in difficulty; it is strongly recommended that you do not spend too much time on any one question.**
- **The exam consists of 5 pages, 5 questions, and a total of 50 points. Last paper is left intentionally blank, for you to use if you want.**
- **You answer on the question sheet.**

---

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 you write a C program that consists of one C file, then we don't need a linker:

1. True 2. False 3. depends on OS 4. depends on the hardware

(B) By seeing the following signed number: 0xFF0070FF we know for sure that it is:

1. negative int 2. negative short int
3. negative floating point 4. We do not know for sure ← will be considered correct too

(C) Suppose we have a 32-bit machine. The size of "double" is:

1. 4 bytes 2. 8 bytes
3. 2 bytes 4. Depends on the compiler.

(D) Suppose we have a 64-bit machine. The size of "double" is:

1. 4 bytes 2. 8 bytes
3. 2 bytes 4. Depends on the compiler.

(E) If we see a binary file that cannot be opened by a text editor, then it has been generated by (choose the most precise):

1. the compiler 2. the assembler 3. the linker
4. 1 or 2 5. 1 or 3 6. 2 or 3

2. [6 points] For single precision floating point numbers, calculate the following and show all your steps (correct final answer without steps will not earn you any credits). You can leave your answer at the form: $xyz*2^{abc}$

- The largest positive number (infinity is not counted):

sign bit = 0, exponent = 11111110 (as it cannot be all 1s) = 254
real exponent = 254 – 127 = 127
largest fraction = 1111….11 (that is 23 1s)
number = $+1.1111.11* 2^{127} = \sim +2^{128}$

- The smallest positive non-zero number:

We need denormalized form
sign bit = 0 exponent = 00000000 → real exponent = 1-127 = -126
fraction = 0000….1
This makes the number: $+ 0.0000…1*2^{-126} = 2^{-23} * 2^{-126} = 2^{-149}$

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 right of x is set to 1.

- What value **mask** must have in binary?

  00000100

- What value **mask** must have in hexadecimal?

  0x04

- Suppose x = 1, will the condition be true? Show the value of x, mask, and x & mask in binary to justify.

  x = 00000001 mask = 00000100
  x & mask = 00000000 → condition is false

- What if x = -1? Show the value of x, mask, and x & mask in binary to justify.

  x = 11111111 mask = 00000100
  x & mask = 00000100 → condition is true

4. Suppose we have the following piece of C code (%p in printf just prints the address in hex):

```
int main()
{
   foo(2);
}
foo(int x)
{
   printf("The address of x is %p\n", &x);
   foo1(x-1);
}
foo1(int y)
{
   char k;
   printf("The address of y is %p\n", &y);
   scanf("%c", &k);
}
```

a. [3 points] If x is stored at address A1, and y at address A2, will A1>A2 always be true? or will A2 > A1 always be true? Or can it sometime go this way and sometimes the other in different executions? Justify your answer.

main → foo → foo1 So the top of the stack is the stack frame of foo1 that contains k. Below is the stack frame of foo that contains x.
The stack grows toward the address zero. So A1 (where x is stored) will always be higher than A2 (where y is stored).

b. [4 points] k in foo1 is a char. Specify the range of numbers that k can present [Hint: You already know whether k is signed or unsigned, so don't ask!]

Since it is char and not unsigned char, then k is signed. A char is 8 bits. So the range goes from: -2^{8-1} to $+2^{8-1}-1$ → -128 to 127

c. [2 points] If the user inputs a number larger than the range that k can present, what will happen?

The extra bits will be truncated and what is stored in k will be wrong.

d. [2 points] When we reach the scanf line, how many stack frames exist in the memory?

Three. Stack frames for main, foo, and foo1. By the time we reach scanf(), the stack frame of printf() has already been removed from the stack. Those who wrote 4 to include scanf() stack frame will be considered correct too.

5. **Given the following piece of code:**

```
void process(int *x){
   int y[10];

   x = (int *)malloc(10*sizeof(int));

 }
```

a. [4 points] How many bytes does the array of y[] consume? and how many does x[] consume?

- Array x[] needs: **10 elements * 4 bytes each = 40 bytes**
- Array y[] needs: **also 40 bytes**

b. [4 points] Where in memory is the array y[] stored? and where is x[] stored? (data, text, heap, or stack)

- x[] is stored in: **the heap**
- y[] is stored in: **stack**

c. [4 points] What happens to arrays x[] and y[] when the function process() returns?

y[] will disappear and x[] stays in the heap.

d. [2 points] Suppose, instead of the malloc(), we want to make x points to array y[]. Write the C statement to do so.

x = y
or x = &y[0]

e. [2 points] Why do we need the typecasting with malloc?

Because malloc returns a generic pointer (void *) and we need the compiler to know the type of the data that this pointer points to in order to do pointer arithmetic correctly.

f. [4 points] Suppose that, after malloc is executed, x[0] is stored at address 1000. And assume that some extra code is written to initialize the array x[] as follows: x[0] = 0, x[1] = 1, …. till x[9] = 9. So element i of array x contains value i. What are the values stored in the following:

Variable	**Value**
x	**1000**
x+2	**1000 + (2*4) = 1008**
*(x+3)	**X[3] = 3**
x+10	**1000 + (10*4) = 1040**

The compiler will do the pointer arithmetic and the CPU will execute no matter whether the address is outside the array or not, as shown in the last item of the table.