CS-277-F2 Final Exam
Spring 2019
Name:

1. For the following, circle **TRUE** or **FALSE** (24 points)

   a. Two's complement allows us to represent the same number of positive integers as negative integers for a given word length. **TRUE** **FALSE**

   b. If `%edx = 0xABABABAB` and `%eax = 0x4455AA22`, the x86-64 instruction
   ```
   movsbl %dh, %eax
   ```
   would change the value of `%rax` to `0xFFFFFFAB` **TRUE** **FALSE**

   c. The variables local to a function are allocated from the stack. **TRUE** **FALSE**

   d. A string in C is encoded by an array of character terminated by the null character. **TRUE** **FALSE**

   e. x86-64 instructions can range in length from 1 to 15 bytes. **TRUE** **FALSE**

   f. x86-64 is a 64-bit instruction set architecture. **TRUE** **FALSE**

   g. In x86-64 `%rsp` is the stack register and you can use an add instruction to add an immediate value to it. **TRUE** **FALSE**

   h. A benefit of a fully associative cache is that it tends to reduce the miss rate. **TRUE** **FALSE**

   i. In x86-64 `cmpq %rdx, %rax` would not alter the condition codes. **TRUE** **FALSE**

   j. In x86-64 `cmpq %rdx, %rax` would leave the value in `%rdx` unchanged. **TRUE** **FALSE**

   k. In x86-64 `cmpq %rdx, %rax` would leave the value in %`rax` unchanged. **TRUE** **FALSE**

   l. In x86-64, both `sal` and `shl` shift instructions have the same effect. **TRUE** **FALSE**

2. In your own words, describe how parameters are passed to a procedure using the registers and the stack. (12 points)

3. Let x=0x87, give the hex for the following assuming a word size of 8 bits (6 Points)
   a. x << 3

   b. x >> 2 (logical)

   c. x >> 2 (arithmetic)

4. Illustrate the difference between the following two C functions by implementing them using Y86 assembly. (12 points)

```
long inc_a(long a) { return a++; }
long inc_b(long b) { return ++b; }
```

5. Assume a non-pipelined processor with a combinational circuit that takes 300 ps and a register element that takes 20 ps to latch a new value. (12 points)

   a. What is the cycle time for the non-pipelined processor?

   b. Suppose a computer architect proposes to break up the combinational circuit in 3 stages to build a pipelined version of the original design. Each stage takes 100 ps. A pipeline register follows each stage; each register taking 20 ps. What is the cycle time (after how much time can the next instruction start execution; not the total delay) of this 3-stage pipeline?

   c. Suppose the architect cannot evenly break the original 300 ps delay among the 3 stages. Instead stage 1 takes 50 ps, stage 2 takes 150 ps, and stage 3 takes 100 ps. We are still using the same pipeline registers. What is the fastest cycle time that this design can achieve?

6. Consider the following x86-64 assembly code for a C loop. (12 points)

```
loop:
        movl    $0x0, %eax
        jmp     .L2
.L3:
        subl    $1, %edi
        addl    $1, %esi
        addl    $1, %eax
.L2:
        cmpl    %esi, %edi
        jg      .L3
        ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use the symbolic variables `x`, `y`, and `result` in your expressions below — do not use register names.)

```
int loop(int x, int y)
{
    int result;

    for (_______________; _______________; result++ ) {

        _______________;

        _______________;

    }

    return result;
}
```

7. Consider the following C functions and x86-64 assembly code. (6 points)

```
int fun1(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}

int fun2(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

int fun3(int a, int b)
{
    if (a >= b)
        return a;
    else
        return b;
}
```

```
        movl    %esi, %eax
        cmpl    %esi, %edi
        jl      .L3
.L2
        ret
.L3
        movl    %edi, %eax
        jmp     .L2
```

Which of the C functions on the left represent the assembly code on the right?

8. The following problem concerns basic cache lookups. (16 points)

- The memory is byte addressable.
- Memory accesses are **1-byte words** (not 4-byte words).
- Memory addresses are 13-bits wide.
- The cache is 2-way set associative, with 4 byte block size and 16 total blocks.

In the following table, all numbers are given in hexadecimal. The contents of the cache are as follows:

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0	99	04	03	48
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	2F	81	FD	09	0B	0	8F	E2	05	BD
3	06	0	3D	94	9B	F7	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	B7	26	2D	F0	0	0C	71	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

a. The box below shows the format of a memory address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO The block offset with the cache block
CI The cache index
CT The cache tag

12	11	10	9	8	7	6	5	4	3	2	1	0

b. For the given memory address, indicate the cache entry accessed and the cache byte value returned in hex. Indicate whether a cache miss occurs.

If there is a cache miss, enter "–" for "Cache Byte Returned."

Memory address: **0xE34**

I. Memory address format (one bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0

II. Memory reference:

Parameter	Value
Block offset CO	0x
Cache Index CI	0x
Cache Tag CT	0x
Cache Hit? (Y/N)	
Cache Byte Returned	0x