Midterm Exam

CSC 252
5 March 2020
Computer Science Department
University of Rochester

Instructor: Yuhao Zhu
TAs: Daniel Busaba, Sudhanshu Gupta, Mandar Juvekar, Max Kimmelman, Weituo Kong, Jiahao Lu, Vladimir Maksimovski, Nathan Reed, Yawo Alphonse Siatitse, Yudi Yang, Shuang Zhai, Prikshet Sharma

Name: _____

Problem 0 (2 points):	_____
Problem 1 (13 points):	_____
Problem 2 (14 points):	_____
Problem 3 (11 points):	_____
Problem 4 (24 points):	_____
Problem 5 (11 points):	_____
Total (75 points):	_____
Extra Credit (20 points)	_____

Remember "**I don't know**" is given 15% partial credit, but you must erase everything else. This does not apply to extra credit questions.

Your answers to all questions must be contained in the given boxes. Use spare space to show all supporting work to earn partial credit.

You have 75 minutes to work.

Please sign the following. I have not given nor received any unauthorized help on this exam.

Signature:_____

GOOD LUCK!!!

Problem 0: Warm-up (2 Points) What's your favourite instruction?

Problem 1: Fixed-Point Arithmetics (13 points)
Part a) (3 points) Represent the decimal number 683 in hexadecimal.

Part b) (3 points) Represent the binary value 10000111 in the base-6 number system.

Part c) (3 points) Represent the binary value 1101.101 in decimal.

Part d) (4 points) Is it possible to add two registers and set Carry Flag to 1, Zero Flag to 0, Signed Flag to 1, and Overflow Flag to 1? If yes, show an example; otherwise, explain.

Problem 2: Floating-Point Arithmetics (14 points + 4 points extra credit)

Part a) (4 points) Put $7\frac{19}{64}$ in the binary normalized form.

Part b) (4 points) According to the IEEE754 single-precision format, which of the following is NaN?

A. 0111 1111 1100 1010 0100 1001 0001 0010
B. 1111 1111 0100 1010 0100 1001 0001 0010
C. 0000 0000 0000 0000 0000 0000 0000 0000
D. 0111 1111 1000 0000 0000 0000 0000 0000

Part c) (6 points + 4 points extra credits) IEEE decided to add a new 12-bit representation, with its main characteristics consistent with the other IEEE standards. Under this 12-bit representation, the value $3\frac{25}{32}$ is represented exactly as `010000111001`.

(3 points) How many bits are needed for fraction?

(3 points) What is the bias?

(4 points extra credit) In this 12-bit representation, what is the result of the following operation?
1110 1011 0010 x 0111 0100 1001

Problem 3: Logic Design (11 points + 5 points extra credit)

The functionality of a two-input NOR gate is specified by the following truth table:

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

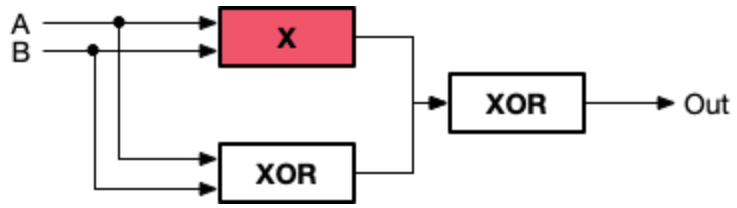
Part a) (9 points) Construct the binary NOT, OR and AND gates using only NOR gates.
(3 points) NOT Gate:

(3 points) OR Gate:

(3 points) AND Gate:

Part b) (2 points) A binary (2-input-1-output) logic gate is said to be "complete" if every other binary logic gate can be made using one or more copies of it. For instance, the NAND gate is known to be complete. Explain very briefly why the NOR gate is complete.

Part c) (5 points extra credit) The U.S. government wants to reconstruct a supercomputer developed by one of its enemies. For this they have asked their top spy, Jonathan, to go undercover looking for information. While snooping around, Jonathan recovered the following schematic. But to his dismay, part of the circuit was removed from the diagram. He knows, however, that the circuit takes two 1-bit inputs (`A` and `B`) and gives a single 1-bit output (`Out`). Furthermore, he knows that the circuit outputs TRUE for every input.



Help Jonathan recover the logic by expressing logic X using only NOT, OR, and AND operations. You don't have to draw the schematic; just show the logic expression.

Problem 4: Assembly Programming (24 points + 6 points extra credit)

For the following parts, the assembly shown uses the syntax **`opcode src, dst`** for instructions with two arguments where **`src`** is the source argument and **`dst`** is the destination argument. For example, this means that **`mov a, b`** moves the value **`a`** into **`b`** and **`sub a, b`** computes the value (**`b - a`**) and stores it in **`b`**.

Also, for functions that take two arguments, the first argument is stored in **`%rdi`** and the second is stored in **`%rsi`** at the time the function is called. The return value of this function is stored in **`%eax`** at the time the function returns.

Part a) (18 points) Below is the assembly code for a mystery function in C.

```
0x0000000000401170 <+0>:   mov    (%rdi),%eax
0x0000000000401172 <+2>:   mov    (%rsi),%edx
0x0000000000401174 <+4>:   mov    %edx,(%rdi)
0x0000000000401176 <+6>:   mov    %eax,(%rsi)
0x0000000000401178 <+8>:   add    (%rdi),%eax
0x000000000040117a <+10>:  retq
```

(3 points) What is one possible data type for the value in %rdi?

(3 points) What is one possible data type for the value in %eax when func returns?

(8 points) Suppose that the state of the memory before this function is called is as shown below, and that the registers **`%rdi`** = 0x48c and **`%rsi`** = 0x484.

State of memory before: (addresses on the left, values on the right)

0x480	0x5
0x484	0x2
0x488	0x20
0x48c	0x9

Fill in the state of the memory after the function is called as well as its return value below.

State of memory after: (addresses on the left, values on the right)	
0x480	
0x484	
0x488	
0x48c	

(4 points) Return value is:

Part b) (6 points) Below is the definition of a struct called **`student`** in C. Below the definition are three C functions that access certain fields or parts of fields from this struct as well as their disassembled assembly in random order. Refer to the struct definition to match these functions with their assembly counterparts in the table below. Assuming that this is a 64-bit machine.

```
typedef struct student{
      short year;
      char major [4];
      int *id;
      struct location {
            char country [3];
            int areacode;
      } home;
      struct student *nextstudent;
} student;
```

A	**B**	**C**

`mov 0x18(%rdi),%rax`
`mov 0x8(%rax),%rax`
`mov (%rax),%eax`
`retq`
	`movsbl 0x11(%rdi),%eax`
	`retq`
		`lea 0x14(%rdi),%eax`
		`retq`

C function	**Assembly (either A/B/C for each)**

`int* field1(student* s){`
`  return &((s -> home).areacode);`
`}`	

`char field2(student* s){`
`  return (s -> home).country[1];`
`}`	

`int field3(student* s){`
`  return *(s -> nextstudent -> id);`
`}`	

Part c) (6 points extra credit) Below is the assembly code for another mystery function in C called **`loop`**. Refer to this code when answering questions below.

```
0x000000000040119f <+0>:     push   %rbp
0x00000000004011a0 <+1>:     mov    %rsp,%rbp
0x00000000004011a3 <+4>:     movl   $0x0,-0x4(%rbp)
0x00000000004011aa <+11>:    movl   $0x5,-0x8(%rbp)
0x00000000004011b1 <+18>:    jmp    0x4011bc <loop+29>
0x00000000004011b3 <+20>:    mov    -0x8(%rbp),%eax
0x00000000004011b6 <+23>:    imul   %eax,%eax
0x00000000004011b9 <+26>:    add    %eax,-0x4(%rbp)
0x00000000004011bc <+29>:    subl   $0x1,-0x8(%rbp)
0x00000000004011c0 <+33>:    jg     0x4011b3 <loop+20>
0x00000000004011c2 <+35>:    mov    -0x4(%rbp),%eax
0x00000000004011c5 <+38>:    nop
0x00000000004011c6 <+39>:    pop    %rbp
0x00000000004011c7 <+40>:    retq
```

(3 points) What does **`loop()`** return?

(3 points) How many instructions are executed in the entire execution of **`loop()`** (including **`nop`**'s)?

Problem 5: ISA (11 points + 5 points extra credit)
The designers of a new ISA are thinking about how to encode jump instructions. Instead of having different opcodes for all the different kinds of jumps (jle, jg, jz, etc), they want to have one opcode for all jumps, and the kind of jump will be encoded in the instruction (see below).

In this ISA, there are 4 condition codes (C0, C1, C2, and C3), whose values can be either 0 or 1. These are similar to the status flags on x86 in that they reflect the status of the last instruction executed. The meanings of the condition codes for the **`add`** and **`sub`** (subtract) instructions in this ISA are given below. The **`mov`** instruction does not change the condition codes.

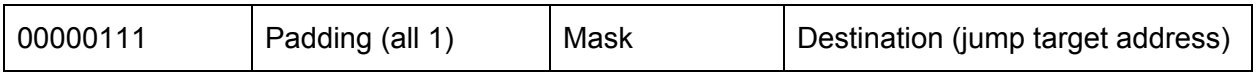
Condition Code	Meaning when codes are set for Add/Subtract instruction
C0	Result zero; no overflow
C1	Result less than zero; no overflow
C2	Result greater than zero; no overflow
C3	Overflow

The jump instruction encoding includes a 4 bit long mask as part of its encoding. The mask is from bits 12-15, as shown in the table.

Condition Code	Set Bit Position in the Instruction
C0	12
C1	13
C2	14
C3	15

A 1 in a certain bit position indicates that that condition code is selected when deciding whether to take the jump or not. To determine if the jump should be taken, the CPU computes the **`OR`** of the values of all the condition codes selected by the mask, and takes the jump if the result of the **`OR`** is 1. For example, a mask **`0110`** selects C1 and C2, and it indicates that the jump will be taken if C1 **`OR`** C2 is 1.

The entire jump instructions is 48-bit long, and it is encoded as follows:



0 7 8 11 12 15 16 47

Bits 0-7 for the opcode, bits 8-11 for the padding (these bits are all 1), bits 12-15 for the mask (as described above), and bits 16-47 for the destination address (i.e., the jump target).

Finally, all the registers in this ISA are 64-bit wide.

Part a) (8 points) Consider the following code (the syntax is **`opcode src, dest`**) for this hypothetical ISA:

```
add r2, r1
sub r1, r2
sub 0x0c, r1
mov r1,r2
```

Suppose for this part that when this code starts executing, the value **`0x0e`** is stored in **`r1`** and the value **`0x02`** is stored in **`r2`**. What is the value of each condition code bit after executing these instructions?

Condition Code	**Value**
C0	
C1	
C2	
C3	

Part b) (3 points) Give the complete encoding (in hexadecimal) of a jump instruction, which jumps to address **`0xffff3d00`** if the result of the previous instruction is less than or equal to 0. Assume that the target address is placed directly in the destination field in big-endian order.

Part c) (5 points extra credit) Now suppose that when the code in part a) starts executing, the value **`0x04`** is stored in **`r1`** and the value **`0x0d`** is stored in **`r2`**. Suppose that your jump instruction from part b) is executed after the instructions from part a). Will the jump be taken? Why or why not? Explain.