CSE 390B: Building Academic Success Through Bottom-Up Computing Winter 2022

Midterm Examination Solutions February 10th, 2022, at 1:30pm

Name: _____

UW NetID: _____

Instructions for those taking the exam <u>in person</u>:

- Make sure you have included your name on the exam (first & last) & your student ID #.
- When you are finished with the exam, turn in your exam to the course staff.

Instructions for students taking the exam <u>virtually</u>:

- You will be writing your exam answers on a blank sheet of paper.
- Make sure to include your name (first & last), your student ID # as well as for each exam question, clearly indicate the exam question number and your answer on your paper.
- Please have your zoom video on and audio muted during the exam.
- When you are finished with the exam, take a picture of your answers and upload it to Gradescope.

General Instructions:

- You will have 50 minutes to complete the exam.
- Questions are not necessarily in order of difficulty.
- This exam is closed-note, closed-book (except for the given reference sheet).
- There are 100 points distributed unevenly among 5 questions (most with multiple parts).

Advice:

- Read each question carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.
- The questions are not necessarily in order of difficulty. Skip around. Make sure you get to all the questions.
- If you have questions, please raise your hand and the course staff will get to you shortly.
- Relax. You are here to learn.

Question	1	2	3	4	5	Total
Possible Points	25	10	20	20	25	100

1. (25 points) In this problem, you will build Boolean circuits with three inputs and two outputs. You may only use two-input And and Or gates and single-input Not gates.

   A prime number is a whole number that's greater than one with only two factors (the number 1 and the factor itself). A number is composite if the number is a whole number greater than one and not a prime number. The first few prime numbers are 2, 3, 5, 7, 11, and so on.

   Design a circuit called NumberTypes that has three inputs (a, b, and c) and two outputs (isPrime and isComposite). The input bits represent the number to check the type for, where a is the most significant bit of the number and c is the least significant bit of the number. The output isPrime has an output of 1 when the number represented by the three input bits is a prime number and 0 otherwise. The output isComposite has an output of 1 when the number represented by the three input bits is a composite number and 0 otherwise.

   a. Write a truth table for the circuit. The circuit should have three inputs and two outputs. You can call the three inputs a, b, and c and the outputs isPrime for if the number is prime and isComposite if the number is composite.

	a	b	c		isPrime	isComposite
Row 1	0	0	0		0	0
Row 2	0	0	1		0	0
Row 3	0	1	0		1	0
Row 4	0	1	1		1	0
Row 5	1	0	0		0	1
Row 6	1	0	1		1	0
Row 7	1	1	0		0	1
Row 8	1	1	1		1	0

b. Based on the truth table you filled out, write a boolean expression for the output isComposite.

Sum of Products: isComposite = (a & ~b & ~c) | (a & b & ~c)
Simplified: isComposite = a & ~c

c. Implement the boolean expression you came up with from part b for the isComposite output only by completing the following HDL template or drawing a circuit diagram using the standard symbols. You do not need to do both.

```
CHIP NumberType {
      // a is the MSB and c is the LSB
      IN a, b, c;

      // isPrime is 1 if input is a prime number
      // isComposite is 1 if input is a composite number
      OUT isPrime, isComposite;

      PARTS:
      // Your code or circuit diagram here:
      // Sum of Products
      Not (in=b, out=notb);
      Not (in=c, out=notc);
      And (a=a, b=notb, out=anotb);
      And (a=anotb, b=notc, out=row5);
      And (a=a, b=b, out=ab);
      And (a=ab, b=notc, out=row7);
      Or (a=row5, b=row7, out=isComposite);

      // Simplified
      Not (in=c, out=notc);
      And (a=a, b=notc, out=isComposite);
}
```

d. In less than a paragraph, describe at a high level how you would implement the output logic for isPrime using the logic that you implemented for isComposite from part b. (Hint: A Mux gate will be helpful here.)

We can negate the isComposite output using a Not gate and call the output notComposite. However, this will cause the inputs of 0 to 1 to become marked as prime when 0 and 1 are not prime numbers. To address this, we can use a Mux gate. We can pass into the Mux gate false (a zero) and notComposite. The select bit would check if the input is greater than 1. If so, the Mux will output notComposite and false (a zero) otherwise.

2. (10 points) In this problem, you may only use And, Or, and Not gates. Your And and Or gates can take any number of inputs. For the problems below, briefly describe which gates you would use and how they contribute to the desired output.

    a. In less than a paragraph, explain at a high level how you would build a circuit that takes a two's complement 16-bit number that outputs 1 if the input is the maximum possible two's complement value or the minimum possible two's complement value.

    In two's complement, the maximum possible value is 0b0111_1111_1111_1111 and the minimum value is 0b1000_0000_0000_0000.

    Maximum value: To check for the maximum possible value, we can use a 16-bit **And** gate. We would pass the 16-bit input through the 16-bit And gate with the exception of the most-significant bit, which would go through a **Not** gate before entering the 16-bit And gate.

    Minimum value: We can use a 16-bit **Or** gate. We would pass the 16-bit input through the 16-bit Or gate with the exception of the most-significant bit, which would go through a Note gate before entering the 16-bit Or gate. Currently, the 16-bit Or gate outputs 0 if the input is the minimum value, but we want the output to be 1 if the input value is the minimum value. We can connect the output of the 16-bit Or gate through a **Not** gate to achieve this.

    Combining the two: We can use a two-bit Or gate to connect the logic for checking the maximum and minimum values.

    b. In less than a paragraph, explain at a high level how you would build a circuit that takes an unsigned 16-bit number as its input and outputs 1 if the input number is even.

    The only bit we would have to check for is the least-significant bit. If the LSB is 0, then the number is even. Otherwise, the number is odd. We can pass the LSB of the input through a two-bit Or gate and make the other input false. The output of this Or gate would specify whether the number is odd. We can pass the output of the Or gate to a Not gate to specify whether it's even or not.

3. (20 points) In this problem, you may only use two-input muxes, DFFs, and combinational logic gates.

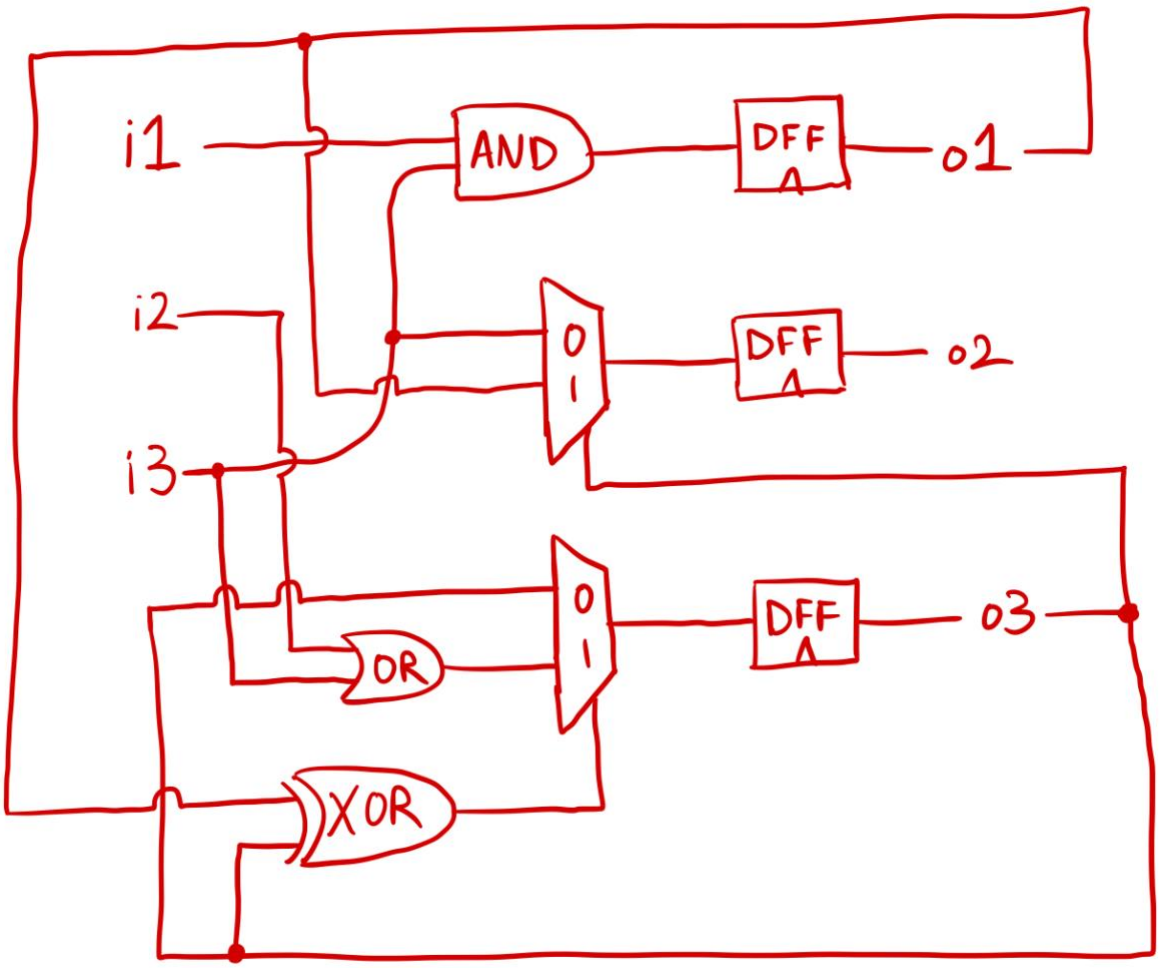
   Draw the following circuit specification using conventional notation, omitting implicit clock signals.

   - The circuit takes three data inputs (`i1`, `i2`, and `i3`)
   - The circuit has three outputs (`o1`, `o2`, and `o3`)
   - Each output at time `t + 1` is defined as follows:

     - `o1(t + 1) = i1(t) & i3(t)`

     - ```
       if (o3(t)): o2(t + 1) = i3(t)
       else:       o2(t + 1) = o1(t)
       ```

     - ```
       if (o1(t) != o3(t)): o3(t + 1) = i2(t) | i3(t)
       else:                o3(t + 1) = o3(t)
       ```



4. (20 points) Below is a sample program written in high-level pseudocode:

```
if (R0 > 2 || R1 == -1) {
      R2 = 0
} else if (R0 == 0) {
      R2 = R0 & R1
} else {
      R2 = R1 - 5
}
```

Write an equivalent Hack assembly program using the virtual registers R0, R1, and R2. R0, R1, and R2 correspond to the values in memory at addresses 0, 1, and 2, respectively.

```
(START)
      @R0
      D = M
      @2
      D = D - A
      @BRANCH1
      D; JGT
      @R1
      D = M
      D = D + 1
      @BRANCH1
      D; JEQ
      @R0
      D = M
      @BRANCH2
      D; JEQ
      @BRANCH3
      0; JMP
```

Continued on right column →

```
(BRANCH1)
      @R2
      M = 0
      @END
      0; JMP
(BRANCH2)
      @R0
      D = M
      @R1
      D = D & M
      @R2
      M = D
      @END
      0; JMP
(BRANCH3)
      @R1
      D = M
      @5
      D = D - A
      @R2
      M = D
      @END
      0; JMP
(END)
      @END
      0; JMP
```

5. (25 points) Below is a Hack assembly program with a single bug. R0, R1, and R2 correspond to the values in memory at addresses 0, 1, and 2, respectively.

```
01.     (START)
02.         @R0
03.         M = 0
04.         @R2
05.         M = 3
06.     (LOOP)
07.         @R2
08.         D = M
09.         @8
10.         D = D - A
11.         @END
12.         D; JGT
13.         @R2
14.         A = M;
15.         // PART I
16.         D = M;
17.         @R1
18.         D = M - D
19.         @CONTAINS
20.         D; JEQ
21.         @R2
22.         M = M + 1
23.         // PART II
24.         @LOOP
25.         0; JMP
26.     (CONTAINS)
27.         @R0
28.         M = 1
29.         @END
30.         0; JMP
31.     (END)
32.         @END
33.         0; JMP
```

Here is the state of memory before the Hack Assembly code to the left runs (we will use this to answer later parts of this problem):

Address	Value
0	39
1	3
2	17
3	5
4	-9
5	25
6	2
7	5
8	-2
9	29
10	37
11	3
12	-7
13	1
14	18
15	-31

a. Trace through the code starting with the state of memory given in the table. Indicate the value of the registers M, A, and D, at each of the following locations commented “PART #” the first time you reach that location when executing the code.

   i. Values of M, A, and D when first reaching comment with “PART I”

      M = 5

      A = 3

      D = -5

   ii. Values of M, A, and D when first reaching comment with “PART II”

      M = 4

      A = 2

      D = -2

b. Starting with the state of memory given in the table, what are the values stored at address 0, address 1, and address 2 in memory after the Hack Assembly code runs to completion, (enters the END infinite loop)?

      Value at address 0 = 0

      Value at address 1 = 3

      Value at address 2 = 9

c. The Hack Assembly code should check if the values stored in memory from addresses R3 to R7, inclusive, contain the value stored at address R1. If so, the value stored at address R0 should be 1. Otherwise, the value stored at address R0 should be 0. The return statement on line 5 immediately terminates the program. The program attempts to be equivalent to the following pseudocode:

```
1.      ram[0] = 0
2.      for (i = 3; i < 8; i++) {
3.          if (ram[1] == ram[i]) {
4.              ram[0] = 1
5.              return
6.          }
7.      }
```

The Hack Assembly program has a single bug. The program can set ram[0] to 1 even when the value specified in ram[1] doesn't exist between the values stored from R3 to R7, inclusive. Provide the memory address and value that would cause the program to incorrectly set ram[0] to 1.

i. Memory address: 8

ii. Value at memory address: 3

d. The bug in the Hack Assembly code can be fixed by changing a single line. Provide the line # for the line of code that you would change and the change that you would make to fix the program.

i. Line # that should be fixed: 12

ii. New line of code: D; JGE or D; JEQ

e. In one sentence, what is the purpose of the value at address R2 in this problem?

The value at address R2 stores the index i of the for loop.