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

Midterm Examination Solutions November 10th, 2022, at 2:30pm

Name: _____

UW NetID: _____

Instructions:

- Make sure you have included your name (first & last) and your UW NetID on this page.
- When you finish the exam, turn in your exam to the course staff.
- You will have 60 minutes to complete the exam.
- Questions are not necessarily in order of difficulty.
- This exam is closed-note, closed-book (except for the reference sheet).
- There are 100 points distributed unevenly among five questions (most with multiple parts).

Advice:

- Read each question carefully. Understand a question before you start writing.
- When applicable, elaborate on your answer, explain your thought process, and write down the intermediate steps for possible partial credit. However, clearly indicate what your final answer is.
- 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.

Technical Details:

- For Boolean expressions, use &, |, and ~ to specify And, Or, and Not, respectively. If you use different symbols, explicitly specify what they mean.
- When using a Mux or DMux gate, explicitly show or describe the select bits that the inputs are connected to (e.g., the a input of the Mux is connected to the select bit of 0).

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 one output. You may only use two-input And and Or gates and single-input Not gates.

   An electronics company specializes in creating unique electronics chips that build on many of the same fundamental logic gates that we have learned together in CSE 390B, and they have requested your help with the Xor3Way chip. This chip has the same idea as the XOR gate. As we might expect, the Xor3Way chip has an output of 1 when exclusively one of its inputs is 1. However, there is actually another row in which the Xor3Way gate has an output of 1.

   It turns out that an n-input XOR gate has the following specification: the XOR of one column with the XOR of the (n-1) other columns. For example, for an XOR gate with three inputs, we can calculate the output for Xor3Way using the following Boolean expression specification: a XOR (b XOR c).

   a. Given the specification of Xor3Way described in the paragraph above, write a truth table for the circuit with three inputs and one output.

	a	**b**	**c**	**out**
Row 1	0	0	0	0
Row 2	0	0	1	1
Row 3	0	1	0	1
Row 4	0	1	1	0
Row 5	1	0	0	1
Row 6	1	0	1	0
Row 7	1	1	0	0
Row 8	1	1	1	1

   b. Based on the truth table you filled out, write a Boolean expression for out.

   out = Row 2 | Row 3 | Row 5 | Row 8

   out = (~a & ~b & c) | (~a & b & ~c) | (a & ~b & ~c) | (a & b & c)

c. Implement the Boolean expression you came up with from part b for the out output by completing the following HDL template or drawing a circuit diagram. You do not need to do both.

```
CHIP Xor3Way {
      // a, b, and c are the inputs
      IN a, b, c;

      // out is the result of the following specification:
      // a XOR (b XOR c).
      OUT out;

      PARTS:
      Not (in=a, out=not-a);
      Not (in=b, out=not-b);
      Not (in=c, out=not-c);

      And (a=not-a, b=not-b, out=not-a-and-not-b);
      And (a=not-a-and-not-b, b=c, out=row2);

      And (a=not-a, b=b, out=not-a-and-b);
      And (a=not-a-and-b, b=not-c, out=row3);

      And (a=a, b=not-b, out=a-and-not-b);
      And (a=a-and-not-b, b=not-c, out=row5);

      And (a=a, b=b, out=a-and-b);
      And (a=a-and-b, b=c, out=row8);

      Or (a=row2, b=row3, out=row2-or-row3);
      Or (a=row5, b=row8, out=row5-or-row8);
      Or (a=row2-or-row3, b=row5-or-row8, out=out);
}
```

d. An n-input Xor logic gate is an **odd function** (odd in the sense of an even or odd number). Explain why you think the Xor logic gate is called an odd function. (Hint: Look back at your truth table, specifically at the characteristics of the rows with an output of 1.)

The Xor logic gate is called an odd function because given any number of inputs and any set of inputs, the Xor gate will always return 1 when there are an odd number of inputs that are 1. Looking at the truth table above for Xor3Way, we observe that the rows with an odd number of inputs have an output of 1.

2. (10 points) Free response questions. Describe the answers to the following questions in a paragraph.

    a. Explain at a high level how you would build a circuit that takes three inputs and returns 1 if an even number of the inputs are 1 and 0 otherwise. You may use any number of And, Or, Not, or Xor gates that have two inputs. Describe which gates you would use, how to wire them together, and how they contribute to the specified output. You may write a Boolean expression or draw a circuit diagram in addition to your explanation if you find that helpful. (Hint: Refer to problem 1.)

    From problem 1, we observed that an Xor gate has an output of 1 when an odd number of inputs are 1. We can take that same idea and apply it here to build a circuit that returns 1 if there are an even number of inputs by flipping the result of the three-way Xor gate using a Not gate.

    Assume that we have three inputs called a, b, and c. Since the problem specified that we could only use two-input gates, we can create a three-way Xor gate as so: a Xor (b Xor c). To create the circuit described in this problem, we would pass the inputs directly to this three-way Xor gate (which returns 1 if an *odd* number of inputs are 1) and then pass the output through a Not gate (since we want a circuit that returns 1 if an *even* number of inputs are 1).

    b. Describe two benefits of the two's complement number representation compared to the signed representation of binary numbers.

    First, the signed representation of binary numbers has two representations of 0: +0 and -0, whereas the two’s complement representation of binary numbers only has one representation of 0. Because of this, given a fixed-width binary number, the two’s complement representation can represent one additional number compared to the signed representation.

    Second, addition using the signed representation of binary numbers no longer works when overflow occurs. If a bit of 1 is carried over from the magnitude to the column with the sign (the MSB), the result will be incorrect. Additionally, if a bit of 1 is carried over from the column with the sign, the result will be incorrect.

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

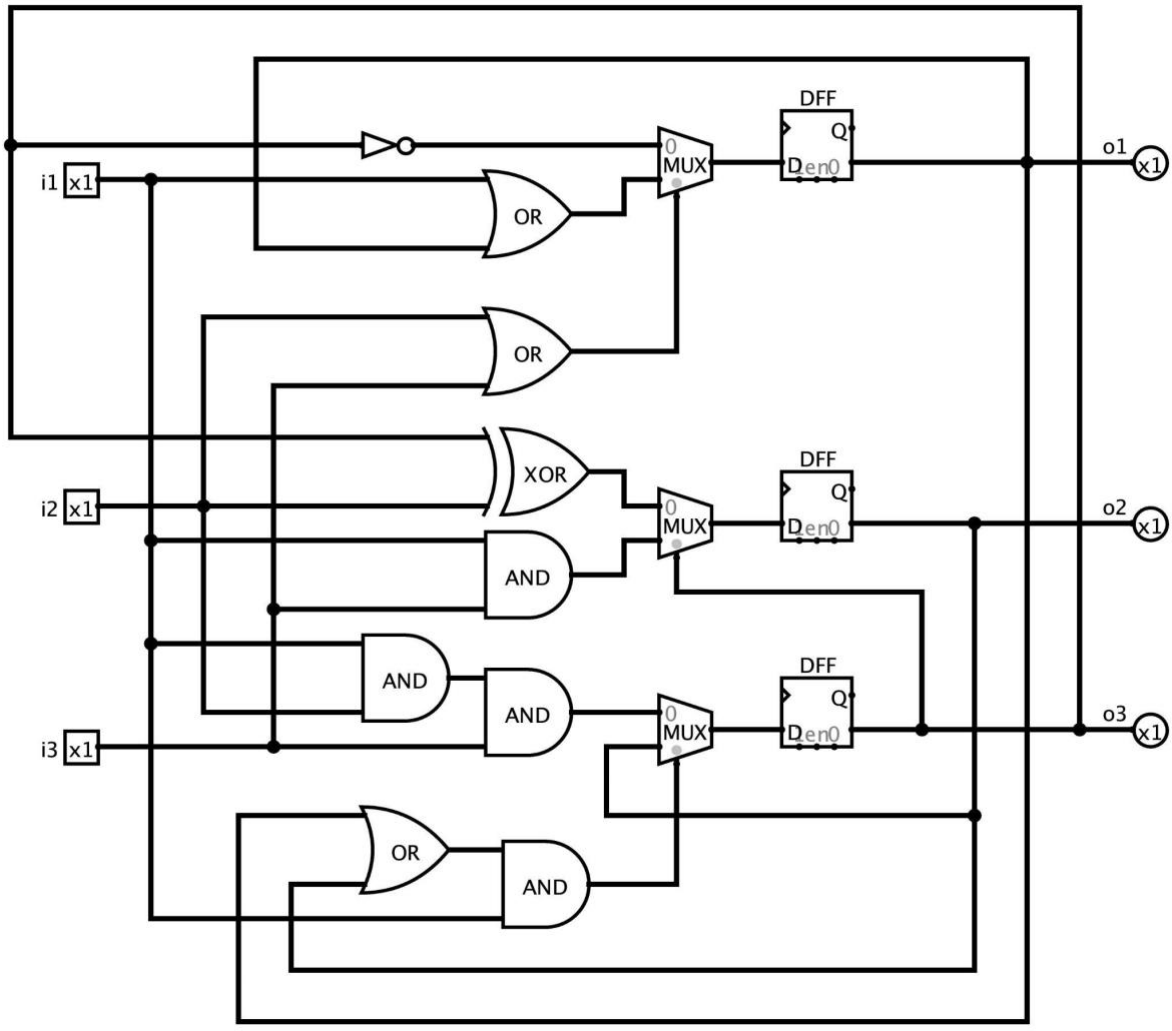
   Draw the following circuit specification using conventional notation, omitting implicit clock signals.

   - 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:

     ```
     ○ if (i2 | i3):          o1(t + 1) = i1 | o1
       else:                  o1(t + 1) = !o3

     ○ if (o3):               o2(t + 1) = i1 & i3
       else:                  o2(t + 1) = i2 ^ o3

     ○ if (i1 & (o1 | o2)):   o3(t + 1) = o2
       else:                  o3(t + 1) = i1 & i2 & i3
     ```



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

```
R2 = 0
R3 = 0
while (R0 >= R1) {
      R0 = R0 - R1
      R2 = R2 + 1
}
if (R0 == 0) {
      R3 = 1
}
```

   a. Write an equivalent Hack Assembly program using the virtual registers R0, R1, R2, and R3, each of which corresponds to the values in memory at addresses 0, 1, 2, and 3, respectively.

```
(START)
      @R2
      M = 0
      @R3
      M = 0
(LOOP)
      @R0
      D = M
      @R1
      D = D - M
      @IF_STATEMENT
      D; JLT
      @R1
      D = M
      @R0
      M = M - D
      @R2
      M = M + 1
      @LOOP
      0; JMP
```

```
(IF_STATEMENT)
      @R0
      D = M
      @END
      D; JNE
      @R3
      M = 1
(END)
      @END
      0; JMP
```

   b. What does this pseudocode do, and what do the registers in this problem represent?

      This pseudocode performs the operation R0 ÷ R1, storing the result in R2, the remainder in R0, and whether the remainder is 0 in R3.

5. (25 points) Below is a Hack Assembly program with a bug.

```
01.     (START)
02.         @R0
03.         M = 1
04.         @2
05.         D = A
06.         @R1
07.         M = D
08.     (LOOP)
09.         @R1
10.         D = M
11.         @7
12.         D = D - A
13.         // PART I
14.         @END
15.         D; JGE
16.     (CHECK_PAIR_SORTED)
17.         @R1
18.         A = M
19.         D = M
20.         A = A + 1
21.         D = M - D
22.         @UPDATE_INDEX
23.         D; JGT
24.         @R0
25.         M = 0
26.     (UPDATE_INDEX)
27.         @R1
28.         M = M + 1
29.         // PART II
30.         @LOOP
31.         0; JMP
32.     (END)
33.         @END
34.         0; JMP
```

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

Address	**Value**
0	0
1	2
2	5
3	6
4	7
5	9
6	10
7	11

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

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

      A = 7

      D = -5

      M = 11

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

      A = 1

      D = 1

      M = 3

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 (i.e., enters the END infinite loop)?

      Value at address 0 = 1

      Value at address 1 = 7

      Value at address 2 = 5

c. The Hack Assembly code is supposed to check whether the elements stored in memory address R2 to R7, inclusive, are sorted so that the values are non-decreasing as the memory addresses increase. The result, a boolean of whether the elements are non-decreasing, is stored in address R0. The Hack Assembly program above attempts to be equivalent to the following pseudocode:

```
1.      R0 = 1
2.      for (i = 2; i < 7; i++) {
3.          if (RAM[i] > RAM[i + 1]) {
4.              R0 = 0
5.          }
6.      }
```

We can fix the bug in the Hack Assembly code by **modifying** a single line of Hack Assembly. Circle the section of code indicated by the symbols in the Hack Assembly program in which we should modify the line to fix the bug.

```
START    LOOP    CHECK_PAIR_SORTED    UPDATE_INDEX    END
```

d. What line number would you modify to fix the bug in the Hack Assembly program, and what should the line of code be instead?

i. Line number: 23

ii. Line of Hack Assembly to fix the bug: `D; JGE`

e. Does the buggy Hack Assembly program return the correct output given the initial memory state shown in the table? If so, how could you change the initial memory state for the bug to appear? If not, how could you change the initial memory state for the buggy assembly program to produce the correct output?

Yes, given the initial memory state from the table, the buggy Hack Assembly program produces the correct output. The bug appears when there is a consecutive pair of numbers that are equal (the program will mark that pair as unsorted when it is sorted in non-decreasing order). Since there isn't a consecutive pair of numbers that are the same in the initial memory state, the buggy still produces the correct output. We can introduce the bug by adding a consecutive pair of equal numbers while maintaining non-decreasing order.