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

Midterm Examination Solutions May 5th, 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 are finished with 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 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.
- 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 two outputs. You may only use two-input And and Or gates and single-input Not gates.

   Data in the form of 0s and 1s transmitted across computer networks often become erroneous in the transmission process (one or more bits are flipped one or more times) due to noise in its environment. One such error-detecting code utilizes a **parity bit** prepended to the **payload** (the stream of 0s and 1s intended to be sent across the network). Before the sender transmits the payload, the parity bit is set to 0 if the number of 1s in the value is even and set to 1 if the number of 1s in the value is odd (not including the parity bit itself). For example:

   - **0**1011010 has a parity bit of 0, as there are an even number of 1s in the payload
     ↳ parity bit
   - **1**1101011 has a parity bit of 1, as there are an odd number of 1s in the payload
     ↳ parity bit

   When the value arrives to the receiver, the receiver will know that there was an error in the transmission if the parity bit doesn't reflect the characteristics of the original payload (e.g., the receiver notices the parity bit is 1, but there are an even number of 1s in the payload).

   Design a circuit called CheckError that has three inputs (a, b, and c) and an output called error. The input bits represent the parity bit and payload in binary, where a is the most significant bit of the number and the parity bit and b and c represent a two-bit payload with c being the least significant bit. The output error has an output of 1 if the parity bit is inconsistent with the characteristics of the original payload and 0 otherwise.

   a. Write a truth table for the circuit with three inputs and one output.

a	**b**	**c**	**error**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

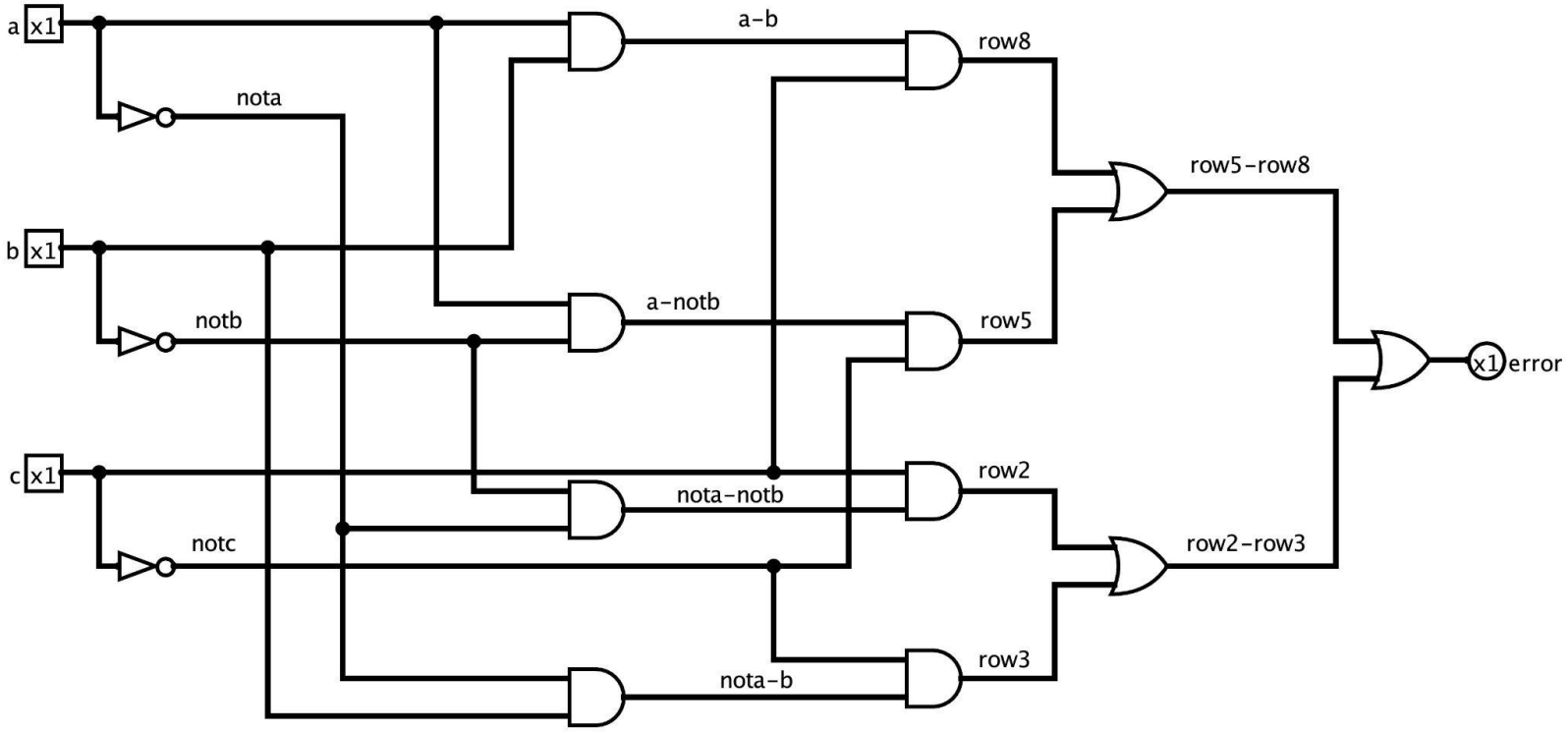
b. Based on the truth table you filled out, write a Boolean expression for error.

error = (~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 error output by completing the following HDL template or drawing a circuit diagram. You do not need to do both.

```
CHIP CheckError {
      // a is the parity bit
      // {b, c} is the two-bit payload
      IN a, b, c;

      // error is 1 if input if parity bit is inconsistent
      with the characteristics of the payload
      OUT error;

      PARTS:
      // Your code or circuit diagram here:

      Not (in=a, out=nota);
      Not (in=b, out=notb);
      Not (in=c, out=notc);

      And (a=nota, b=notb, out=nota-notb);
      And (a=nota-notb, b=c, out=row2);

      And (a=nota, b=b, out=nota-b);
      And (a=nota-b, b=notc, out=row3);

      And (a=a, b=notb, out=a-notb);
      And (a=a-notb, b=notc, out=row5);

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

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

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

Circuit diagram for 1(c):



2. (10 points) In this problem, you may only use And, Or, Not, Mux, and DMux 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 two 16-bit values and outputs 1 if the two values are equal to each other.

    Use an Xor gate on the two 16-bit values. If the output of the Xor gate is 0, then the values are equal. Since we want to output 1 when the values are equal, we can add a Not gate to the output of the Xor gate.

    Since we don't have an Xor gate available to us, we can create one using And, Or, and Not gates using the Boolean function synthesis method. In the Xor truth table, we would create a Boolean function for each of the rows where only a single input is true using And and Not gates, and then combine those two rows using Or gates.

    b. In less than a paragraph, explain at a high level how you would build a circuit that takes two 16-bit values and outputs the minimum of the two values using the two's complement binary number interpretation. Assume that the two values will never be equal and you have a chip that can subtract two 16-bit numbers and a chip that can check for any inequality (less than, greater than or equal to, etc.) against 0.

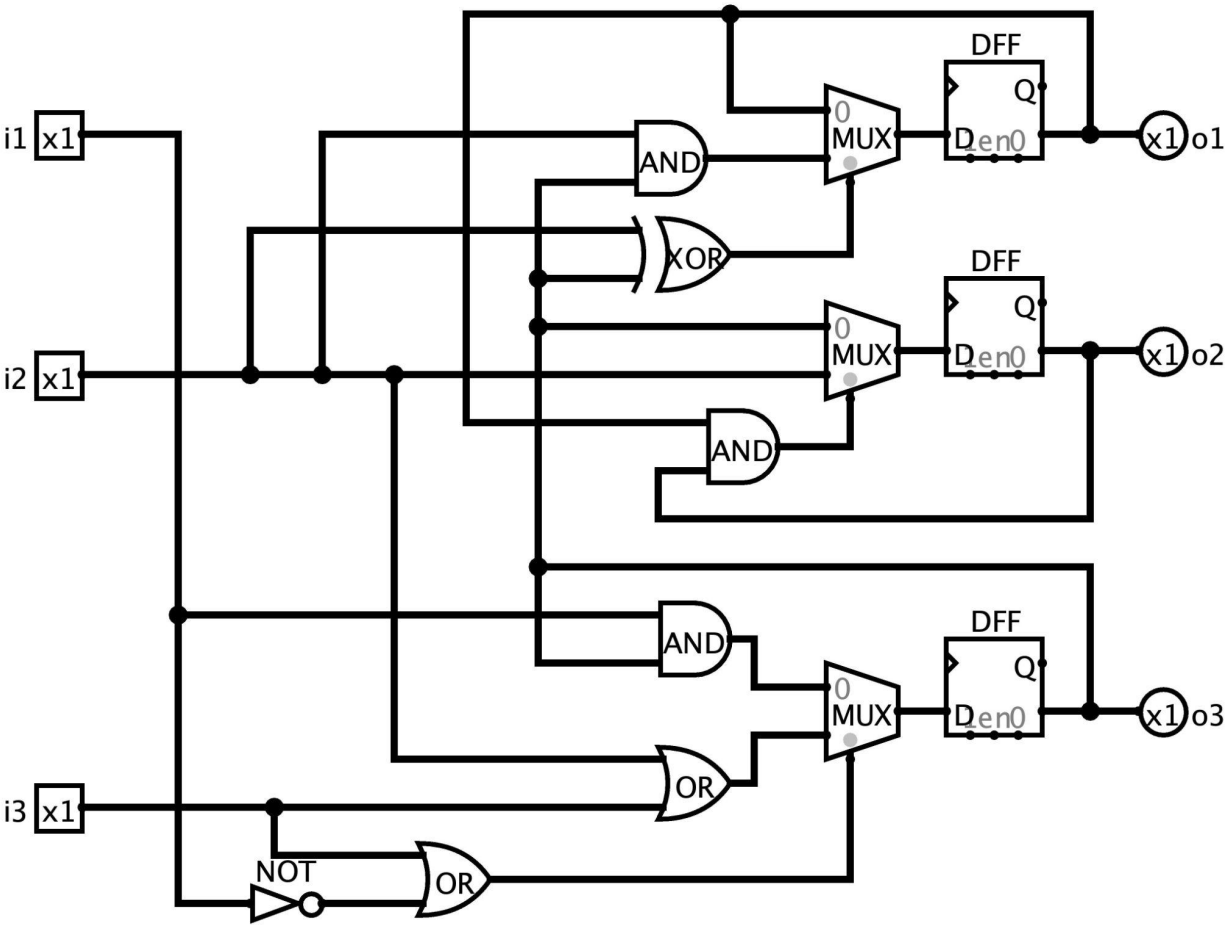
    Assume that the first 16-bit value is called a and the other 16-bit value is called b. We can use the subtract chip to subtract a from b and call this out (the output of the subtract chip would be b - a). We can use the less than inequality to check if out is less than 0 (if so, b is the minimum) and pass this into the select bit of a Mux. We can pass b into the input associated with a select bit of 1 and pass a into the input associated with a select bit of 0.

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(t) == o3(t)):  o1(t + 1) = o1(t)
       else:                 o1(t + 1) = i2(t) & o3(t)

     ○ if (o1(t) & o2(t)):   o2(t + 1) = i2(t)
       else:                 o2(t + 1) = o3(t)

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



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

```
if (R0 <= 3 || R1 != 1) {
      R0 = -5
} else if (R1 == -3) {
      R1 = R0 | R2
} else {
      R3 = R3 + 1
}
```

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

```
(START)
      @R0
      D = M
      @3
      D = D - A
      @BRANCH1
      D; JLE
      @R1
      D = M - 1
      @BRANCH1
      D; JNE
      @R1
      D = M
      @3
      D = D + A
      @BRANCH2
      D; JEQ
      // BRANCH 3
      @R3
      M = M + 1
      @END
      0; JMP
```

Continued on right column →

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

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

```
01.   (START)
02.       @R2
03.       D = M
04.       @R0
05.       M = D
06.       @3
07.       D = A
08.       @R1
09.       M = D
10.   (LOOP)
11.       @R1
12.       D = M
13.       @8
14.       D = D - A
15.       // PART I
16.       @END
17.       D; JGE
18.       @CHECK_MAX
19.       0; JMP
20.   (CHECK_MAX)
21.       @R1
22.       A = M
23.       D = M
24.       @0
25.       D = D - M
26.       @UPDATE_MAX
27.       D; JGT
28.       @R1
29.       M = M + 1
30.       @LOOP
31.       0; JMP
32.   (UPDATE_MAX)
33.       @R1
34.       D = M
35.       @0
36.       M = D
37.       @R1
38.       M = M + 1
39.       // PART II
40.       @LOOP
41.       0; JMP
42.   (END)
43.       @END
44.       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	13
1	-9
2	7
3	19
4	-9
5	13
6	21
7	5
8	2
9	-29
10	37
11	14
12	9
13	21
14	0
15	18

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 “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 = 8

      D = -5

      M = 2

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

      A = 1

      D = 3

      M = 4

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 = 6

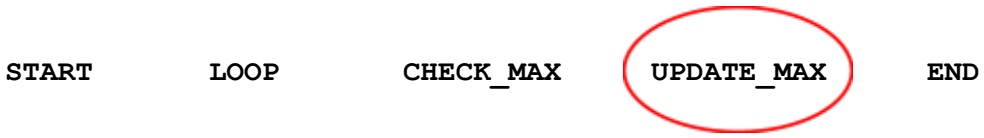
      Value at address 1 = 8

      Value at address 2 = 7

c. The Hack Assembly code is intended to find the maximum value stored in memory from addresses R2 to R7, inclusive. The maximum value should be stored at address R0. The Hack Assembly program above attempts to be equivalent to the following pseudocode:

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

The bug in the Hack Assembly code can be fixed by **adding** a single line of Hack Assembly. Circle the section of code indicated by the symbols in the Hack Assembly program in which the line should be added to fix the bug.



d. What is the line number you would add the new line of code to (shifting the original line of code in that line number and the subsequent lines of code down), and what is the additional line of code to fix the program?

i. Line number: 34

ii. New line of code: A = M

e. In less than a paragraph, explain why this additional line of code is necessary for the Hack Assembly program to have the same behavior as the pseudocode above.

Without the additional line of code, the program stores the address of the maximum value in memory. However, the pseudocode specifies that we should store the value itself. If we just have D = M in UPDATE_MAX, we store the address of the maximum value in D, since the R1 register stores the address it's currently on (the value in the R1 register functions as index i in the pseudocode).

By introducing A = M before D = M, we set the address of the maximum value into the A register, so when D = M is executed, D is set to the maximum value instead of the address of the maximum value.