CSE 390B: Building Academic Success Through Bottom-Up Computing Winter 2023

Midterm Exam Solutions February 9th, 2023, 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).
- This exam contains 100 points distributed unevenly among five questions (some 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. Feel free to skip around. Do your best to get to all the questions.
- If you have a question, please raise your hand, and the course staff will get to you shortly.
- Take deep breaths and relax. Remember that you are here to learn.

Technical Details:

- You may specify Boolean operations using symbols or words (e.g., for the And gate, you may use the symbol & or "And").
- 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.

   Recall that a C-instruction in Hack Assembly contains a destination, a computation, and a jump component. There are three computations that are just values (i.e., do not contain any operations on any of the registers). Sometimes, a Hack Assembly programmer can forget the limitation that C-instructions only have three valid values and provide an invalid computation, leading to an error in the assembly process. Implement a chip called ValidComputation that accepts three bits as a Two's Complement number and returns 1 if the number from the input bits represents a valid Hack Assembly computation and 0 otherwise.

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

a	**b**	**c**	**out**
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
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 out.

      out = (~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 ValidComputation {
        // a, b, and c represent the value in Two's Complement
        IN a, b, c;

        // out is the result of whether the input represents a
        // valid computation in Hack Assembly
        OUT out;

        PARTS:
        // Your code or circuit diagram here:

        // Setting up nots of the input bits
        Not (in=a, out=nota);
        Not (in=b, out=notb);
        Not (in=c, out=notc);

        // Row 1 Boolean expression
        And (a=nota, b=notb, out=notab);
        And (a=notab, b=notc, out=out1);

        // Row 2 Boolean expression
        And (a=notab, b=c, out=out2);

        // Row 8 Boolean expression
        And (a=a, b=b, out=ab);
        And (a=ab, b=c, out=out3);

        // Combining all the Boolean expressions using Or
        Or (a=out1, b=out2, out=out1orout2);
        Or (a=out1orout2, b=out3, out=out);
}
```

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

    a. Is it possible (i.e., assemble without error) to provide a negative value to an A-instruction in Hack Assembly? If so, explain what happens when you provide a negative value to an A-instruction. If not, explain why you cannot provide a negative value to an A-instruction.

       Yes, it is possible to provide a negative value to an A-instruction in Hack Assembly. However, it is not practical for a Hack Assembly programmer to use a negative A-instruction. When you do so, the most-significant bit of the 16-bit instruction becomes 1 instead of 0, which causes the instruction to become a C-instruction. A negative A-instruction will be meaningless to the programmer unless the corresponding C-instruction in machine code is decoded to an intelligible C-instruction in Hack Assembly.

    b. Describe one benefit and one drawback of symbols in Hack Assembly.

       One benefit of symbols in Hack Assembly is that it enables programmers to write Hack Assembly code more efficiently and makes their code more legible as well. One drawback of symbols in Hack Assembly is that it requires additional work for implementing the Hack Assembly language. Instead of using just numbers for A-instructions, there would need to be an additional symbol table that maps from a symbol to an address value.

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

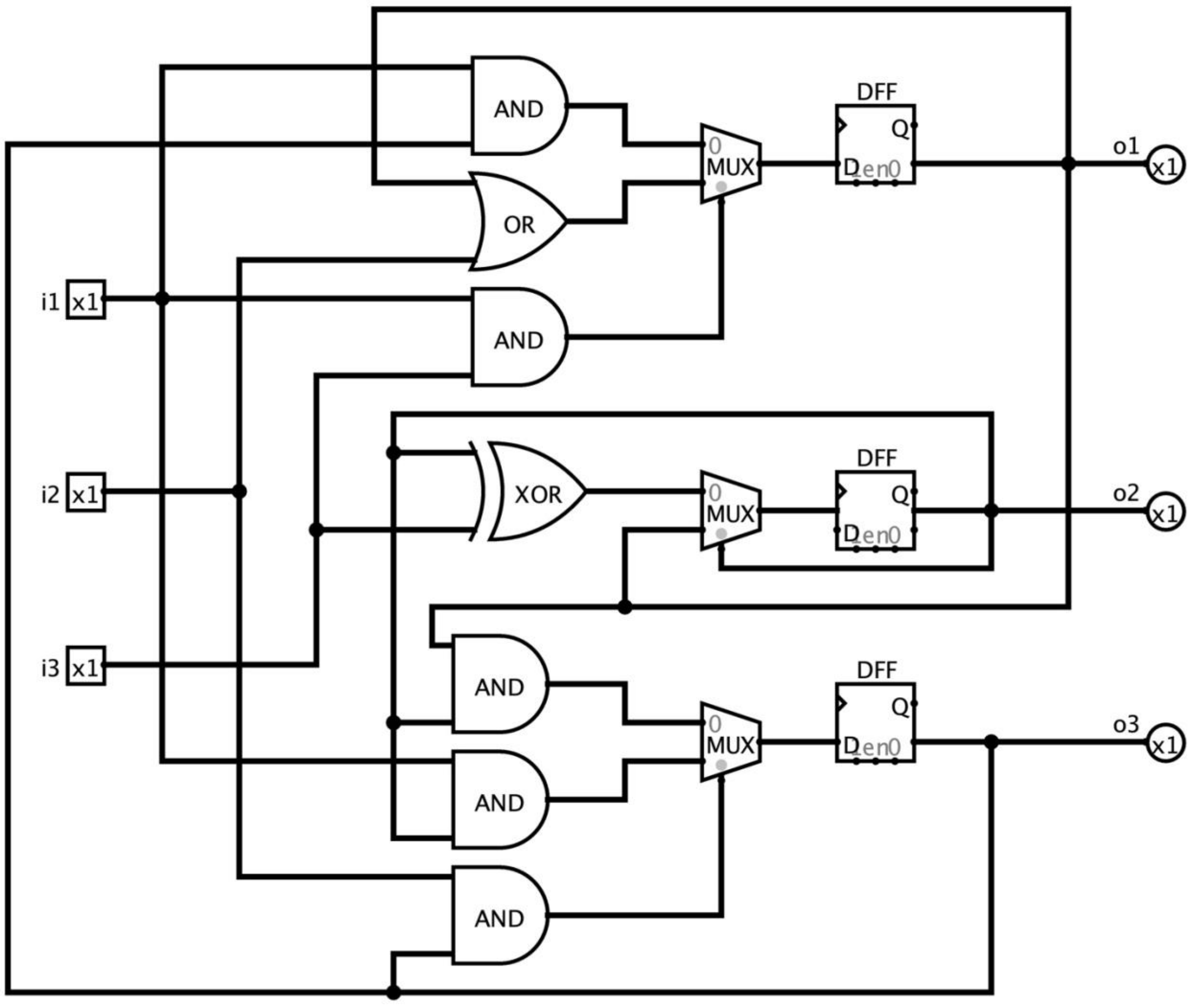
   Draw the following circuit specification using conventional notation, omitting implicit clock signals.

   - 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 (i1 & i3):          o1(t + 1) = o1 | i2
  else:                  o1(t + 1) = o3 & i1

○ if (o2):               o2(t + 1) = o1
  else:                  o2(t + 1) = o2 ^ i3

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



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

```
if (R0 + R1 == -5) {
      R0 = 8
} else if (R2 & R3 >= 3) {
      R0 = !R1 | R2
} else {
      R0 = R1 & R2
}
```

```
@R0
D = M
@R1
D = D + M
@5
D = D + A
@BRANCH1
D; JEQ
@R2
D = M
@R3
D = D & M
@3
D = D - A
@BRANCH2
D; JGE
@R1
D = M
@R2
D = D & M
@R0
M = D
@END
0; JMP
```

```
(BRANCH1)
   @8
   D = A
   @R0
   M = D
   @END
   0; JMP
(BRANCH2)
   @R1
   D = !M
   @R2
   D = D | M
   @R0
   M = D
(END)
   @END
   0; JMP
```

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

```
(START)
    @R0
    M = 0
    @2
    D = A
    @R1
    M = D
(LOOP)
    @R1
    D = M
    @8
    D = D - A
    // PART I
    @END
    D; JGE
(CHECK_NEGATIVE)
    @R1
    A = M
    D = M
    @UPDATE_INDEX
    D; JGE
    @R0
    M = !M
(UPDATE_INDEX)
    @R1
    M = M + 1
    // PART II
    @LOOP
    0; JMP
(END)
    @END
    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	5
1	1
2	3
3	-7
4	10
5	-1
6	2
7	3

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 = 8

      D = -6

      M = Unknown value at address 8 in the memory table

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

      A = 1

      D = 3

      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 = 0

   Value at address 1 = 8

   Value at address 2 = 3

c. The Hack Assembly code is supposed to check whether the elements stored in memory addresses R2 to R7 contain any negative values. The result is 1 if any of the values at the specified memory addresses contain any negative values and 0 otherwise. The result is stored in address R0. The Hack Assembly program above attempts to be equivalent to the following pseudocode:

```
R0 = 0
for (i = 2; i < 8; i++) {
    if (RAM[i] < 0) {
        R0 = 1
    }
}
```

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_NEGATIVE**] **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: M = 1

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, the buggy Hack Assembly program does return the correct output given the initial memory state show in the table. To cause the bug to appear, change the values in memory to include an odd number of negative numbers. The bug is that when a negative number is seen in memory, the R0 Boolean register is flipped instead of set to true.