CSE390B Spring 2020, Midterm Examination
May 7, 2020

You will have 50 minutes to complete the exam.

Rules:

- The exam is closed-book, closed-note, etc.
- However, you have been provided a reference sheet as a separate file.
- There are **100 points**, distributed **unevenly** among **6** questions (most with multiple parts).

Advice:

- Read questions 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, ask via a private chat message to Dan or Aaron.
- Relax. You are here to learn.

1. (**20** points) In this problem, you will build Boolean circuits with three inputs and one output. You may use two-input *And* and *Or* gates, and single-input *Not* gates.

   (a) In this part, you will build a circuit AtLeastTwo where the output is 1 if and only if 2 or 3 of the inputs are 1.

      i. Write a truth table for this circuit. You can call the inputs a, b, c, and the output d.

      ii. Draw an implementation of this circuit using the standard symbols for the gates you use.

      iii. Complete the following HDL template to implement this circuit:

```
CHIP AtLeastTwo {
 IN  a, b, c;
 OUT d;
 PARTS:
 // Write the lines that would be inserted here
}
```

   (b) In this part, you will build a circuit ExactlyTwo where the output is 1 if and only if exactly 2 of the inputs are 1. You can use AtLeastTwo where it is helpful.

      i. Write a truth table for this circuit. You can call the inputs a, b, c, and the output d.

      ii. Draw an implementation of this circuit using the standard symbols for the gates you use.

      iii. Complete the following HDL template to implement this circuit:

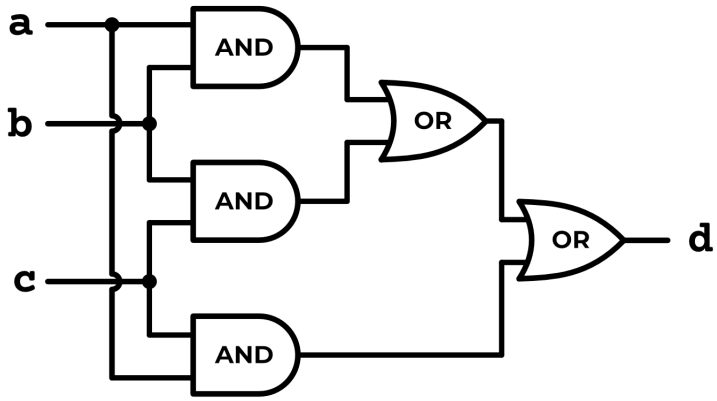
```
CHIP ExactlyTwo {
   IN  a, b, c;
   OUT d;
   PARTS:
   // Write the lines that would be inserted here
}
```

Solution:

(a) i.

a	b	c	d
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

ii. (Other answers possible)

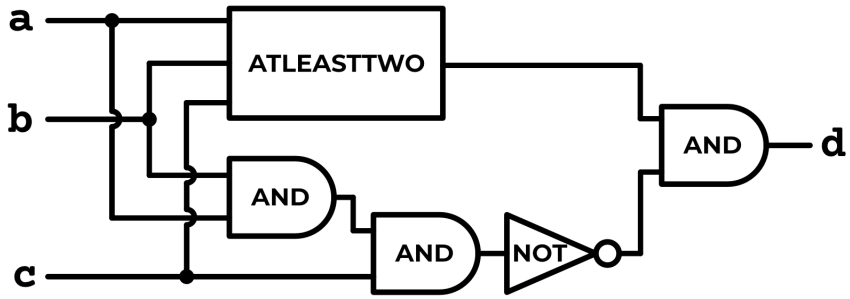


iii.
```
CHIP AtLeastTwo {
  IN  a, b, c;
  OUT d;
  PARTS:
  And (a=a, b=b, out=and1);
  And (a=b, b=c, out=and2);
  And (a=a, b=c, out=and3);
  Or (a=and1, b=and2, out=or1);
  Or (a=or1, b=and3, out=d);
}
```

(b) i.

a	b	c	d
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

ii. (Other answers possible)



iii.
```
CHIP ExactlyTwo {
  IN  a, b, c;
  OUT d;
  PARTS:
  AtLeastTwo (a=a, b=b, c=c, d=atleasttwo);
  And (a=a, b=b, out=and1);
```

```
  And (a=and1, b=c, out=allthree);
  Not (in=allthree, out=notallthree);
  And (a=atleasttwo, b=notallthree, out=d);
}
```

2. (**10** points) In this problem, you can use only *And*, *Or*, and *Not* gates. Your *And* and *Or* gates can take any number of inputs.

   (a) Explain at a high level in *1–2 English sentences* how you would build a circuit that takes a twos-complement 16-bit number and outputs 1 if and only if the number is ≥ 0. You may include a picture if you find it helpful.

   (b) Explain at a high level in *1–2 English sentences* how you would build a circuit that takes a twos-complement 16-bit number and outputs 1 if and only if the number is > 0. You may include a picture if you find it helpful. (Hint: Use your answer to part (a) and additional logic.)

   Solution:

   (a) Ignore all the bits except the leftmost (highest) bit and simply negate it (with a not gate).

   (b) Use an or16 (or a collection of or gates) with all the bits of the number to produce a wire that is 1 if and only if the number is not 0. Now use an and gate with inputs that are this wire and the output of part (a), and use the and's output as the answer.

   (This way the output is 1 if the input is ≥ 0 and not equal to 0.)

3. (**12** points) In this problem, use only two-input muxes, basic flip-flops, and, if necessary, combinational logic.

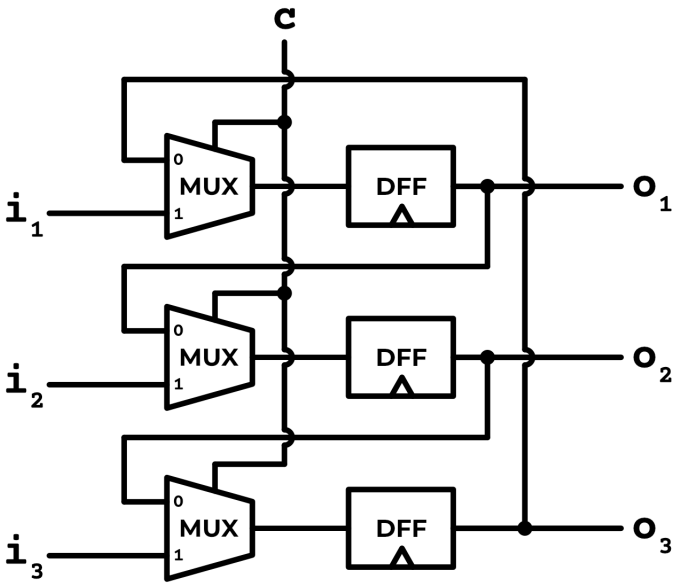
   Draw using conventional notation and omitting the implicit clock signal this circuit:

   - It takes four inputs: three data inputs $i1$, $i2$, and $i3$ and a control signal c.
   - It has three outputs: $o1$, $o2$, and $o3$.
   - When the control signal is set, the output at time $t+1$ is the input at time t, with $o1$ matching $i1$, etc.
   - When the control signal is not set, the output at time $t+1$ is the output at time t but rotated, with $o2$ being the "old" $o1$, $o3$ the "old" $o2$, and $o1$ the "old" $o3$.

   So, for example, if at time t we have $i1 = 0$, $i2 = 1$, $i3 = 1$, and $c = 1$ and then from time $t+1$ onward we have $c = 0$, then the outputs $o1$, $o2$, $o3$ would be:

   - at time t: unknown given the information available
   - at time $t+1$: $0, 1, 1$
   - at time $t+2$: $1, 0, 1$
   - at time $t+3$: $1, 1, 0$
   - at time $t+4$: $0, 1, 1$
   - ...

   Solution:



4. (**20** points) In this problem, you will write Hack assembly language (not machine language) programs. The R0 and R1 "virtual registers" will be reserved for program inputs. You can use the other "virtual registers" to store temporary data.

   (a) Write a sequence of assembly instructions that swaps the contents of two adjacent 16-bit memory locations. The two locations are the address (call it x) that is stored in R0 when the code starts running and one more than that address (i.e., $x + 1$). Hint: Sample solution is about 15 lines long.

   (b) Now write a sequence of assembly instructions (containing a loop) that rotates the contents of a set of adjacent 16-bit memory locations. In terms of part (a), the locations are x, $x + 1$, ... $x + i$ where x is the value in R0 and i is the value in R1 when the code starts running. Assume i is a positive number. After the loop finishes, the program counter should be just after the instructions, and what started at x should now be at $x + 1$, what started at $x + 1$ should be at $x + 2$, and so on, with what started at $x + i$ being at x. Hints:
   - By working backwards through memory, most of your program can be your answer to part (a) where the data starting at $x + i$ is repeatedly moved one word earlier until its final resting place of x. This is simpler than other algorithms.
   - You do not need to copy your part (a) solution — where you want to use it, you can just write, "*part (a) solution here.*"

Solution:

(a)
```
@R0
A=M
D=M
@R2
M=D
@R0
A=M+1
D=M
A=A-1
M=D
@R2
D=M
@R0
A=M+1
M=D
```

(b)
```
  @R1
  D=M-1
  @R0
  M=M+D
  @CONDITION
  0;JMP
(LOOP)
  // part (a) solution here
  @R1
  M=M-1
  @R0
  M=M-1
(CONDITION)
  @R1
  D=M
  @LOOP
  D;JGT
```

5. (**20** points) This problem uses Hack assembly to write to the screen. Recall:

   - The screen pixels correspond to positions in memory where a 0-bit means white and a 1-bit means black.
   - The screen pixels start at address `@SCREEN` and end immediately before `@KBD`.
   - Hack words are 16-bits long, i.e., each address contains 16 bits.

   (a) Here is a *buggy* Hack assembly program that attempts to flip all the pixels on the screen, so each white becomes black and black becomes white. There are *two bugs*. For each bug, give the line number and describe what needs to be fixed.

```
1    @SCREEN
2    D=A
3    @R0
4    M=D
5    @CONDITION
6    0;JMP
7
8  (LOOP)
9    @R0
10   A=M
11   M=-M
12   @R0
13   M=M+1
14
15 (CONDITION)
16   @R0
17   D=M
18   @KBD
19   D=A-D
20   @LOOP
21   D;JLT
```

   (b) Suppose the entire screen were quite small, just 32 pixels wide and 32 pixels tall. What number would `@KBD` minus `@SCREEN` be?

   For the rest of this problem, we now consider changing Hack so that each pixel is represented by 2 bits, so the first word at `@SCREEN` is the memory for 8 pixels instead of 16. Pixels now have 4 colors, where 00 is white, 01 is light-gray, 10 is dark-gray, and 11 is black.

   (c) Now, while still assuming 32 pixels wide and 32 pixels tall, what number would `@KBD` minus `@SCREEN` be?

   (d) If you took the *assembly program* in part (a) *after your bug fixes*, converted it to machine language for this new machine, and executed it, what would happen to the screen? Explain *what* happens in English, no need to explain *why*.

   (e) Now suppose you took the program from part (a) after your bug fixes, but consider the corresponding machine language program for the old 2-color-screen machine. If you execute that old machine-language program on the new 4-color-screen machine, what would happen to the screen? Explain *what* happens in English, no need to explain *why*.

Solution:

(a) One bug is on line 11, where the wrong operator is used — bits are interpreted as a number and negated (-), rather than the correct behavior of flipping each bit (!). The other bug is on line 21, where the wrong jump condition is used which causes the loop to never execute. The following is a fixed version of the code:

```
@SCREEN
  D=A
  @R0
  M=D
  @CONDITION
  0;JMP

(LOOP)
  @R0
  A=M
  M=!M
  @R0
  M=M+1

(CONDITION)
  @R0
  D=M
  @KBD
  D=A-D
  @LOOP
  D;JNE
```

(b) 64 (which is $2^5 \cdot 2^5/16$)

(c) 128 (we need twice as much space for representing the screen)

(d) Each pixel would change color as follows: white to black, light gray to dark gray, dark gray to light gray, black to white. So the same code still works to take the screen and produce its negative image.

(e) The pixels for the top half of the screen would change as in part (d) but the bottom half would stay unchanged. (This is because `@KBD` would have its value for the old machine.)

6. (**18** points) For each question below, *give your a/b/c answer **and** briefly (at most 1 sentence) explain why.* Each question is independent of the others (the changes aren't combined).

   (a) Suppose we change the Hack language and computer so that there is a third register Q that can be used like register D.

      i. Would the job of the assembly programmer be (a) much harder, (b) much easier, or (c) not much harder or easier?

      ii. Would the job of the chip implementor be (a) much harder, (b) much easier, or (c) not much harder or easier?

   (b) Suppose we change the Hack language and computer so that when a jump does not occur, the program counter is decremented (subtract 1) instead of incremented (add 1).

      i. Would the job of the assembly programmer be (a) much harder, (b) much easier, or (c) not much harder or easier?

      ii. Would the job of the chip implementor be (a) much harder, (b) much easier, or (c) not much harder or easier?

   (c) Suppose we change the Hack language by adding an instruction that puts the constant 42 into register D.

      i. Would the job of the assembly programmer be (a) much harder, (b) much easier, or (c) not much harder or easier?

      ii. Would the job of the chip implementor be (a) much harder, (b) much easier, or (c) not much harder or easier?

Solution:
We accepted multiple answers if your explanation justified it since "much" is somewhat subjective.

(a) i. Much easier – having another register lets us keep more data in places that we can compute with directly.

ii. Much harder – all our instructions would have to be re-encoded in machine language, with more circuitry connecting the additional register to the ALU.

(b) i. Not much harder or easier – we'd just have to write our programs in reverse order (and the first instruction might have to jump farther ahead). Okay to answer "much harder" just because this would be harder to read.

ii. Not much harder or easier – circuitry for +1 and -1 is about the same.

(c) i. Not much harder or easier – this instruction isn't very useful as we have other ways of getting its effect.

ii. Not much harder or easier – it is only one more instruction to implement via muxing the constant bits that encode 42 into the D register.