Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the test supplement and the permitted one-page formula sheet. No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 8 questions, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name Solution
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

signed

1. a) [8 points] Write an unsimplified Boolean expression for the function F defined by the following truth table.

```
x  y  z  F
----------
0  0  0  0
0  0  1  1  → ~x * ~y *  z
0  1  0  0
0  1  1  1  → ~x *  y *  z
1  0  0  0
1  0  1  1  →  x * ~y *  z
1  1  0  0
1  1  1  1  →  x *  y *  z
----------
```

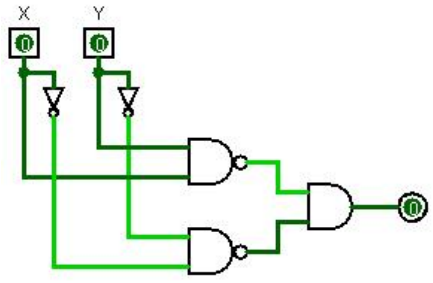
So, F = ~x * ~y * z + ~x * y * z + x * ~y * z + x * y * z

b) [12 points] Simplify the expression you created in part a completely. Cite the axiom or theorem you use in each stage of the simplification. See the test supplement for axioms and theorems.

```
F = ~x*~y*z + ~x*y*z + x*~y*z + x*y*z          original expression
  = (~x*~y*z + ~x*y*z) + (x*~y*z + x*y*z)      associative law
  = ~x*z*(~y + y) + x*z*(~y + y)               commutative and distributive laws
  = ~x*z*1 + x*z*1                             law of complements
  = ~x*z + x*z                                 boundedness law
  = (~x + x)*z                                 distributive law
  = 1*z                                        law of complements
  = z                                          boundedness law
```

---

2. [10 points] The circuit below is equivalent to a single logic gate that was discussed in class. Which one? Justify your conclusion. (Those are two NAND gates and one AND gate.)




```
Output = ~(x * y)*~(~x*~y)
       = (~x + ~y) * (x + y)
       = ~x*x + ~x*y + ~y*x + ~y*y
       = ~x*y + x*~y
```

And that is precisely the definition of XOR.

Alternatively, you could construct the truth table for the circuit and observe that it's the table for XOR.

3. Consider the problem of implementing a circuit to add two 32-bit operands representing signed integers in 2's complement form. Represent the operands *A* and *B* and the resulting sum *S* as:

$$A = a_{31}a_{30}a_{29}\cdots a_2a_1a_0$$
$$B = b_{31}b_{30}b_{29}\cdots b_2b_1b_0$$
$$S = s_{31}s_{30}s_{29}\cdots s_2s_1s_0$$

The addition circuit must include a component that detects the occurrence of overflow.

a) [6 points] Give a one-sentence definition of *overflow*. A definition is not the same thing as a description of how you would detect overflow!

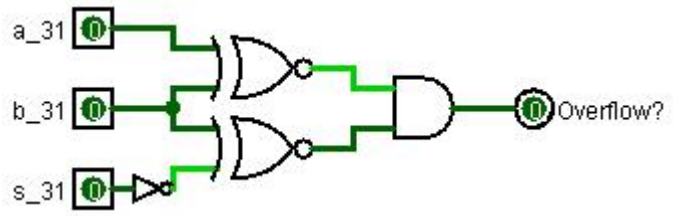
Overflow occurs when the mathematically-correct result of an arithmetic computation, such as addition, cannot be represented in the format used to store the result.

b) [8 points] Write a sentence or two describing how you could determine whether overflow has occurred when two operands *A* and *B*, as described above, are added.

From the discussion in class, the occurrence of overflow in addition of signed 2's complement numbers is only possible if the operands have the same sign, and is indicated by the fact that the sign of the result does not match the signs of the operands… so overflow can be detected by comparing the sign bits of the operands, and one of those sign bits with the sign bit of the result.

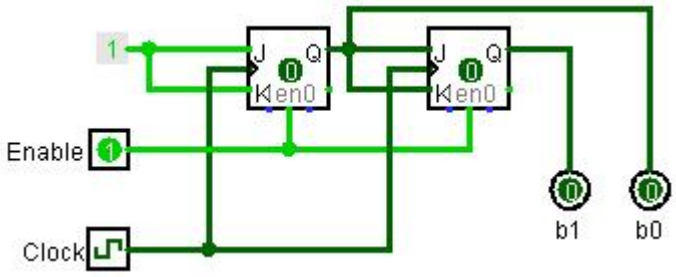
c) [10 points] Draw a circuit that implements the overflow detection logic you described in part b. You may use any of the logic gates shown on the test supplement, and use any of the bits of the operands and/or sum as inputs.

The XNOR gate implements the equality test on its inputs, so one simple solution looks like this:



Alternatively, you could omit the negation for `s_31` and replace the lower XNOR gate with an XOR.

4. [12 points] The circuit shown below uses two JK flip-flops, whose behavior is defined by the table below.



J	K	Q	~Q
0	0	no change	
0	1	0	1
1	0	1	0
1	1	opposite	

Describe the changes in the output bits b1 and b0 as the clock cycles. Your description should be sufficiently complete to explain the behavior of the circuit's outputs completely. Assume the flip-flops update on the rising edge of the clock cycle.

Cycle	Clock action	Circuit action	Right b1	Left b0
0	Initial state		0	0
1	Lo to Hi	Left FF sees both inputs at 1, toggles Right FF still sees both inputs at 0, due to propagation delay, and so holds.	0	1
	Hi to Lo	No state changes on falling edge		
2	Lo to Hi	Left FF sees both inputs at 1, toggles Right FF sees both inputs at 1, toggles	1	0
	Hi to Lo	No state changes on falling edge		
3	Lo to Hi	Left FF sees both inputs at 1, toggles Right FF sees both inputs at 0, holds	1	1
	Hi to Lo	No state changes on falling edge		
4	Lo to Hi	Left FF sees both inputs at 1, toggles Right FF sees both inputs at 1, toggles	0	0
	Hi to Lo	No state changes on falling edge		

(So this is just a mod-4 binary counter.)

For questions 5 through 8, refer to the datapath diagram on the test supplement.

5. [8 points] When a `sw` instruction is being executed, the ALUSrc control signal must be set to 1. Explain why. Be precise and complete.

Recall that the effect of `sw` is: `Memory[GPR[$rs] + Instr[15:0]] = GPR[$rt]`

ALUSrc determines whether the second operand to the ALU comes from the Register File (ALUSrc == 0) or from the Sign-extender (ALUSrc == 1).

For `sw`, the ALU is used to compute the address in Data Memory that is to be written to, and that is computed as the sum of a register value (the other operand to the ALU) and the sign-extended literal formed from the low 16 bits of the instruction itself.

6. [8 points] When a `lw` instruction is being executed, the RegDst control signal must be set to 0. Explain why. Be precise and complete.

Recall that the effect of `lw` is: `GPR[$rt] = Memory[GPR[$rs] + Instr[15:0]]`

RegDst determines whether the register to be written to is determined by bits 20:16 or bits 15:11 of the current instruction. In the case of `lw`, only two register numbers are stored in the instruction and the destination register is specified by bits 20:16.

Aside from all that, for a `lw` instruction bits 15:11 are part of the immediate field used to compute the memory address that will be accessed, and so do not specify a register number at all.

7. [10 points] When a `beq` instruction is being executed, it doesn't matter whether the MemtoReg control signal is set to 0 or to 1. Explain why. Be precise and complete, and explain what <u>other</u> control signal is involved in the situation.

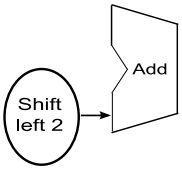
MemtoReg determines whether the value sent back to the Write data port on the Register File comes from the ALU (MemtoReg == 0) or from the Data memory (MemtoReg == 1).

Note that <u>some value will be sent</u> to the Write data port regardless of the setting of MemtoReg. So something else must be done to prevent the value from being written to a register.

But the `beq` instruction does not cause a value to be written into the register file, and so RegWrite will be set to 0 whenever a `beq` instruction is being executed.

Therefore, it does not matter what data is supplied to the Write data port because nothing will be written to a register in any case.

8. [8 points] Refer again to the datapath diagram on the test supplement. Consider the hardware components indicated below. Which of the supported instructions depend on these hardware components and why? Be precise and complete.



The hardware is used by the `beq` instruction, and no others depend upon it.

The hardware is used to calculate the branch target address (i.e., the address of the instruction that will be fetched and executed next IF the branch is taken).

More specifically, the `Shift left 2` unit is used to multiply the immediate from the instruction by 4, resulting in a word offset, which is necessary since all instructions are stored in memory on word boundaries.

Then, the `Add` unit is used to compute the sum of the shifted offset and the value PC + 4, resulting in the address of the instruction that will be executed next IF the branch is taken.