EECS 151/251A Homework 2

Due Friday, September 16th, 2022

Problem 1: Verilog

It's 102 degrees outside and the Cory administrators won't turn on the AC in the lab unless you figure out some Verilog problems.

(a) Rewrite the following statement to different forms.

```
assign out = sel[0] ? in0 : (~sel[1] ? in1 : in2);
```

Assume the relevant signals are already defined correctly.

(i) **Always block** using **if** statements.

Solution:

```
always @(*) begin
    if (sel[0])
        out = in0;
    else if (~sel[1])
        out = in1;
    else
        out = in2;
end
```

(ii) **Always block** using **case** statements.

Solution:

Note that the case must be a full case.

```
always @(*) begin
    case (sel)
        2'b00: out = in1;
        2'b01: out = in0;
        2'b10: out = in2;
        2'b11: out = in0;
    endcase
end
```

or:

```
always @(*) begin
    case (sel)
        2'b00: out = in1;
        2'b10: out = in2;
        default: out = in0; // can also be 2'b?1 if using casez
```

```
    endcase
end
```

(iii) **Structural Verilog** (i.e. AND, OR, NOT, etc). Writing out the logic in a boolean expression first might be helpful. You may also find the boolean expression for a MUX from HW1 useful.

Solution:

Solution may vary. In general students should implement
out = (sel[0] & in0) | (∼ sel[0] & ((∼ sel[1] & in1) | (sel[1] & in2))).

Example solution in verilog:

```
wire sel0_inv, sel1_inv;
wire and_in0_out, and_in1_out, and_in2_out, and_sel1_out, or_sel1_out;
and and_in0 (and_in0_out, sel[0], in0);
not not_sel0 (sel0_inv, sel[0]);
not not_sel1 (sel1_inv, sel[1]);
and and_in1 (and_in1_out, sel1_inv, in1);
and and_in2 (and_in2_out, sel[1], in2);
or or_sel1 (or_sel1_out, and_in1_out, and_in2_out);
and and_sel1 (and_sel1_out, sel0_inv, or_sel1_out);
or or_out (out, and_in0_out, and_sel1_out);
```

(b) Consider the following code block. Logic for *clk* has been omitted.

```
reg [3:0] a, b, c;
initial begin
    a = 4'd0;
    b = 4'd0;
    c = 4'd0;
end
always @(posedge clk) begin
    a <= 4'd1;
    b <= a + 4'd2;
    c <= b + 4'd3;
    c <= c + 4'd4;
end
```

(i) After 1 clock cycle, what are the values for a, b, and c? Describe how you arrived at your solution.

Solution:

$a = 1$, $b = 0 + 2$, $c = 0 + 4$. Note that the second non-blocking assignment for c overwrote the first.

(ii) If blocking assignments were used instead in the always block, what do the values become after 1 cycle? Why?

Solution:

$a = 1$, $b = 1 + 2 = 3$, $c = (3 + 3) + 4 = 10$. Each assignment happens after previous lines finish assigning.

Problem 2: Boolean Expressions

Unfortunately it turns out the ACs in Cory 111 stopped working. However, the maintenance crew said once a few things with this Boolean expression is figured out, everything will be OK (according to them).

$$(a + (bc)')' + [(a' + b + c')(a' + b' + c)]' + ab(c(1 + c'))$$

(a) **Simplify the expression.** You should produce every step of your simplification and mention what laws of Boolean algebra you used for each step. ***Hint:*** *one of the intermediate expressions is $a'bc + ab'c + abc' + abc$. However, this is not the simplest form.*

Solution:

$$
\begin{aligned}
&(a + (bc)')' + [(a' + b + c')(a' + b' + c)]' + ab(c(1 + c')) \\
&= (a + (bc)')' + (a' + b + c')' + (a' + b' + c)' + ab(c(1)) && \text{DeMorgan, Identity, Associative} \\
&= (a + b' + c')' + ab'c + abc' + abc && \text{DeMorgan, Identity, Associative} \\
&= \boldsymbol{a'bc + ab'c + abc' + abc} && \text{DeMorgan} \\
&= a'bc + ab'c + abc' + abc + abc + abc && \text{Inverse Idempotence} \\
&= a'bc + abc + ab'c + abc + abc' + abc && \text{Commutative} \\
&= bc(a' + a) + ac(b' + b) + ab(c' + c) && \text{Inverse Distributive} \\
&= bc + ac + ab && \text{Complements}
\end{aligned}
$$

(b) Negate the simplified expression and rewrite into Product of Sums.

Solution:

$$
\begin{aligned}
&(bc + ac + ab)' \\
&= (bc)'(ac)'(ab)' \\
&= (b' + c')(a' + c')(a' + b')
\end{aligned}
$$

Problem 3: Truth Tables and Circuits

Gah! The California state grid has been overloaded because of the heat wave. Sadly, Cory 111 is now subject to a rolling blackout. To protect the precious FPGA boards from overheating, you

asked PG&E if they could exempt the lab from the load shed. They said if you figure out this circuit, they will be able to restore power.

This question is concerned with the 3-way XOR function $out = a \oplus b \oplus c$. Note that this expression can be written as $out = ab'c' + a'bc' + a'b'c + abc$.

(a) **Write out the truth table.**

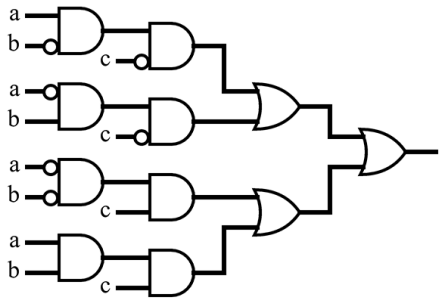
Solution:

a	b	c	out
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) **Draw a logic gate circuit implementation using 2-input AND, OR and NOT gates.** You may use a label (a, b, c) multiple times at gate inputs for clarity. For this part only, you may also use bubbles for negation in place of NOT gates at gate inputs and outputs.

Solution:

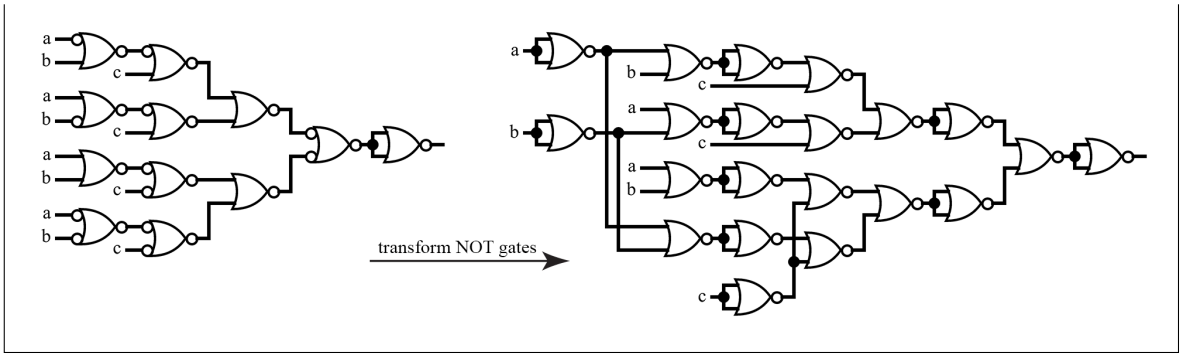
Solutions may vary.



(c) The PG&E employee you're in contact with said his nephew wants to make this circuit in Minecraft, where NOR gates are easy to build. **Transform your circuit to use 2-input NOR gates only.** Note that you can make a NOT gate by using the same signal for both inputs to a NOR.

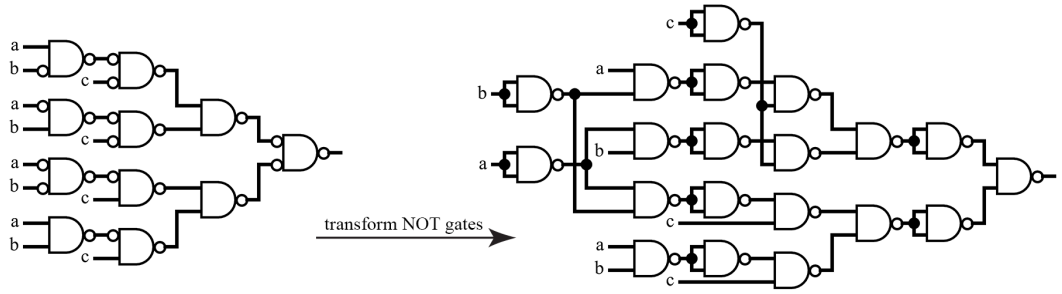
Solution:

General process: change AND gates to OR by negating all ports, introduce bubbles at OR outputs, then insert NOT gates for extra bubbles.



(d) *(251A only)* Also remake the circuit with 2-input NAND gates only.

Solution:



Problem 4: Adding Up

Finally, the AC breathes a cool breeze. However, fixing all the issues took so long that the heat wave has already passed, and it's really not that hot even without AC. You reflect on the problems you had to solve.

What is the connection between the expressions you looked at in Problem 2 and Problem 3? Explain how they are used. ***Hint:*** *think about what might be a common circuit element that uses these logic.*

Solution:

Single bit full-adder.

$out = ab'c' + a'bc' + a'b'c + abc$
$carry\_out = bc + ac + ab$