Homework 1 (100 points)
Due Tuesday, February 1 @11:59pm
Gradescope

The following homework problems are to be completed and turned in on Gradescope on the due date. **Please follow the Gradescope tutorial on Canvas when submitting homework.**

1. Properties of Boolean Difference [15 pts]

(i) [10 pts] Use Boolean algebra and the basic properties of Shannon cofactors from the notes to show that this identity is true. Again, f and g are functions of $x_1, x_2, \ldots x_n$, and x refers to some arbitrary variable in $x_1, x_2, \ldots x_n$.

$$\frac{\partial(f+g)}{\partial x} = \left[\bar{f} \cdot \frac{\partial g}{\partial x}\right] \oplus \left[\bar{g} \cdot \frac{\partial f}{\partial x}\right] \oplus \left[\frac{\partial f}{\partial x} \cdot \frac{\partial g}{\partial x}\right]$$

Hints: (a) Notice that there are no "x" variables in the functions on the left-hand side of this equation (since we cofactored them out), yet there are "x" variables on the right side, since there are both f function and g function there. The only way this can be true is if the equation works both when $x = 0$ and when $x = 1$. So, the first thing to do is set x to a constant on the right-hand side, and then simplify. You have to show this equation works for both $x = 0$ and $x = 1$.

(b) If you first blast all of the EXORs down into their sum of products (SOP) form, this is **not** necessarily the easiest way to do the derivation. Use what you know about cofactors, i.e., the "cofactor of an EXOR is the EXOR of the cofactors" for all simplifications, and so on.

(c) It's helpful to recall that AND distributes over EXOR, i.e., $a \cdot (b \oplus c) = ab \oplus ac$

(d) It's also helpful to recall that $a \oplus a = 0$ and $a \oplus \bar{a} = 1$ for any Boolean expression "a".

(e) It's probably easiest to simplify the right-hand side until it looks like the left, and not the other way around.

(ii) [5 pts] Use ordinary Bolean algebra to show that this identity is true: if function *f* does *not* depend on variable x, then:

$$\frac{\partial(f+g)}{\partial x} = \bar{f} \cdot \frac{\partial g}{\partial x}$$

Hints: think carefully about what it means that f does not depend on x. This means something very specific in terms of the Shannon decomposition.)

2. Properties of cofactors [10 pts]

Let p and q be input variables of a function $f(p, q, x, y, z)$. Let g, h be two other Boolean functions that are independent of p and q, i.e., they only depend on x, y and z. Using Boolean algebra and what you know about cofactors, prove this:

> f can be represented as $f = (p+q)g(x,y,z) + \bar{p}\bar{q}h(x,y,z)$ for $g(*), h(*)$ functions independent of p, q ***if and only if*** $f_p = f_q$

Hints: The trick is to take cofactors for both sides of the equation here, and remember their properties. Remember also that you have to prove the implication *both* ways since this is *if-and-only-if*. This means you have to assume each "side" of the equation is true and prove that the *other* side is true. One way is pretty easy. The other way requires a little thought, since you have to actually **create** some new functions $g(x, y, z)$ and $h(x, y, z)$ and demonstrate with Boolean algebra that you can indeed represent f as it says above. In other words, when you are doing the "if" part, and assuming $f_p = f_q$, you just need to show that you can find *some* functions g and h that make $f = (p+q)g + \bar{p}\bar{q}h$. g and h will end up being related to f ***(and its cofactors!)*** in a straightforward way.

3. Shannon Expansion(s) [10 pts]

Shannon expansion says: $F = xF_x + \bar{x}F_{\bar{x}}$. Are there any *other* ways of representing this same fact?

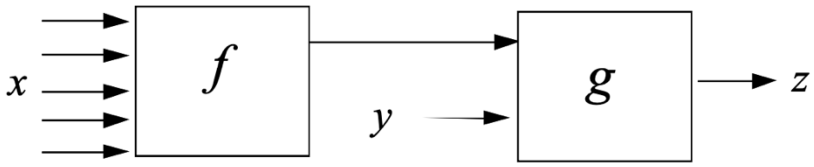
(i) [6 pts] Of course, the above formula is a sum of products style formula, i.e., it ORs together some terms that are each product. Clearly, there has got to be an equivalent form of the Shannon expansion that yields a product of sums. What is it? Draw the direct "gate level network" with ANDs and ORs, x's and cofactors as the inputs, that is implied by your POS formula. In one or two sentences, argue why this network makes physical sense.

Hints: the SOP form we did in class can be thought of making function $F = 1$ "in all the right places." The POS for is more easily thought of as making $F = 0$ "in all the right places."

(ii) [4 pts] How about this proposed formula, does this work as well? $F = xF_x \oplus \bar{x}F_{\bar{x}}$. Tell us yes/no and use Boolean algebra to prove it.

4. Properties of Boolean difference: Simple Chain Rule [10 pts]

It turns out there is yet more "calculus-like" behavior in the Boolean difference. Consider the diagram below:



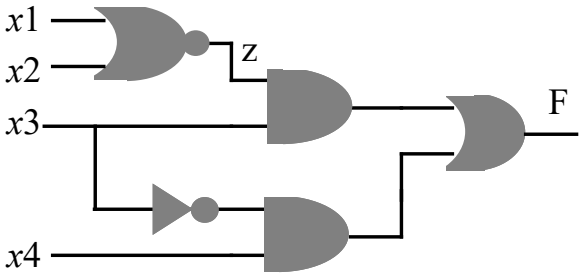
Assume, f is a function of $x_1, x_2, \ldots x_n$, and x refers to some arbitrary variable in $x_1, x_2, \ldots x_n$. Assume g is a function of just two variables, and y is *different* from any of the x_i variables. Use Boolean algebra and the basic properties of Shannon cofactors from the notes to show that this identity is true.

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial f} \cdot \frac{\partial f}{\partial x}$$

Hints: the "hardware diagram" view of what the Shannon expansion theorem represents, with the two copies of each function plus multiplexor, is one good way of thinking about what's going on in this problem.

5. Using Boolean Difference in Testing [10 pts]

Consider this small logic network:



(i) [8 pts] Use the Boolean difference to find a pattern of inputs that tests z for stuck-at-1. Show your work to find a Boolean equation that must be satisfied to find the test vectors. How about the pattern of inputs that tests z for stuck-at-0?

(ii) [2 pts] Repeat, but now use the result from Problem 4 instead of calculating $\partial F/\partial z$ directly, use the chain rule with respect to signal z, and show your work.

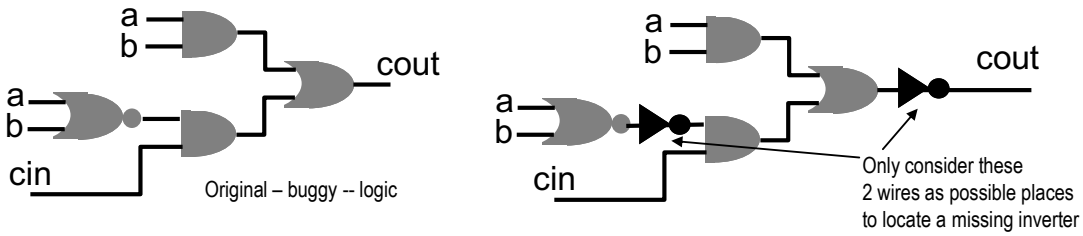
Hint: finish your work by properly choosing an intermediate variable between z and F

6. Gate-level debugging using quantification [10 pts]

Consider a new logic debugging scenario. You have a gate level network implemented, and you are sure that the reason it does not work correctly is that **one** wire is missing an inverter. What you don't know is, ***which*** wire is the one missing the inverter.

Suggest a strategy for using a quantification-based approach like we described in class to solve this debugging problem. The output here should tell you which single wire in the network, if any, can add an inverter to fix the network. Be clear about any extra inputs you need (e.g., whatever behaves like the d_0, d_1, d_2, d_3 variables from the notes) and what exactly you need to quantify away.

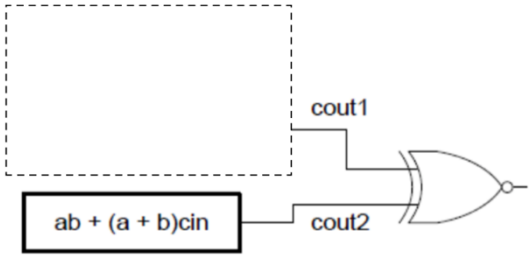
Show how to use your strategy on this proposed (but buggy) gate level design for the carry-out (cout) from a 1-bit adder:



To keep the algebra simpler, only consider the 2 wires shown above as possible locations for the missing invertor. Note, you theoretically know what the *right* answer is by looking at the Boolean equation of c_{out} in terms of a, b and c_{in}, but the point is to show how, by using purely Boolean manipulations, you can identify the correct single invertor to add. Show any extra logic you need to add to make the network you need to apply quantification to. Show all the steps of the analysis.

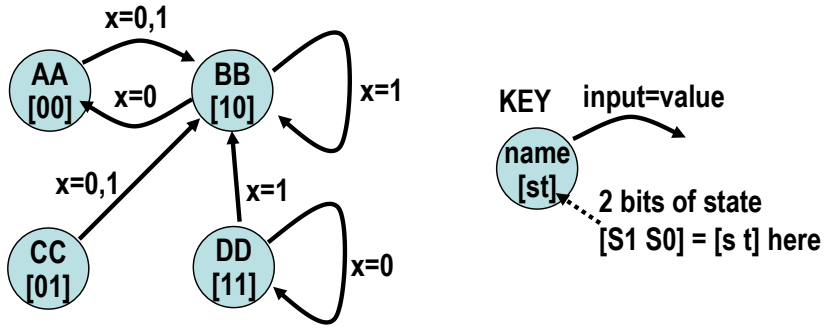
Hints: (a) the new trick in this one is that you *don't* know, in general, where to look to add the inverter. So, add an inverter on *every* single gate output wire (i.e., both of these two suggested places), and figure out how to *enable only one* of these inverters. You can add a programmable inverter on a wire by passing the wire through a 2-input EXOR gate; the second input is an *enabling* input. Remember: $x \oplus 0 = x, x \oplus 1 = \bar{x}$. How would you design logic to enable exactly *one* of the many inverters in this network? Note that EXOR gate can be controlled to turn on just a particular selected *one* of its many outputs.

(b) What you need to find is just a modification to the original buggy logic that can be put inside the dashed box



7. Finite State Machine analysis [10 pts]

It is very natural to think about representing a finite state machine using some kind of a graph: the *nodes* represent the states, and the *edges* represent the transitions from state to state under input changes. The common state diagram notation is exactly this kind of a graph:



The machine above has four states AA, BB, CC, DD, represented by 2 bits of state (2 separate flip flops) whose values appear in brackets (e.g., [10] represents state BB) in each state bubble. This machine has just one input x. We are ignoring any outputs for this machine.

Unfortunately, these graphs can get very big very easily. Imagine a state machine with 20 bits of state (i.e., 20 separate flip flops). It has 2^{20} states ~1,000,000 states. Suppose also there are 20 input variables. Then each of these 1,000,000 states has 1,000,000 transition arrows leaving it. Our little machine with 20 flip flops has one trillion state transitions!

It turns out there is a more elegant way of representing things here. Imaging that we create a new Boolean function, called $\boldsymbol{R}$, the *state transition relation function*. In general $\boldsymbol{R}$ has 3 kinds of inputs:

$\boldsymbol{R}$(*state variables for starting state*, *input variables*, *state variables for ending state*)

The way to think about the $\boldsymbol{R}$ function is that it answers a simple question: *can I get from a specific starting state to a specified ending state via specified input value?* For our little example, here are a few values of **R**:

$\boldsymbol{R}(AA, 0, BB) = 1$ means "*yes, from state AA, an input x=0 takes you to state BB*"

$\boldsymbol{R}(BB, 1, AA) = 0$ means "*NO, from state BB, input x=1 does NOT take you to AA*"

Of course, you can't just input the state "names", you have to use the state assignment bits that represent each state. This means that R is a function of 5 variables for our little example:

$\boldsymbol{R}(S_1, S_0, x, E_1, E_0)$

where S_1, S_0 is the state assignment for the starting state, and E_1, E_0 is the assignment for the ending state. So, for example:

$\boldsymbol{R}(AA, 0, BB) = 1$ really means that $\boldsymbol{R}(0,0,0,1,0) = 1$, since state AA is represented by the assignment S_1S_0=00, and similarly the state BB has the assignment 10.

(i) [6 pts] Complete a truth table for $\boldsymbol{R}(*)$ for this little 4-state example, and create a sum of products Boolean expression for $\boldsymbol{R}(*)$ using a 5-variable Kmap.

(ii) [4 pts] Let's create a new function, called $\boldsymbol{G}(E_1, E_0)$. $\boldsymbol{G}$ is again a function of the states, but it answers a different question: *is there ANY way to reach this state* E_1E_0? For example, it is clear from our little state diagram that $\boldsymbol{G}(CC) = 0$, i.e., there is just NO way to start from some state and take a transition that gets you to state CC. But $\boldsymbol{G}(AA) = 1$ because it is possible to get to state AA (for example, by being in state BB and taking the $x = 0$ transition). Show how to use the *quantification* operators (i.e.,$\{\exists, \forall\}$) to transform the $\boldsymbol{R}(*)$ function into the $\boldsymbol{G}(*)$ function. *Write* a general expression for what you have to do to $\boldsymbol{R}(*)$ to turn it into $\boldsymbol{G}(*)$. *Explain* in 1-2 sentences why it works. Then, actually use Boolean algebra to *perform* the appropriate quantification on the SOP form for $\boldsymbol{R}(*)$ you already derived and show that the resulting equation for $\boldsymbol{G}(*)$ makes sense for this problem.

Hints: don't just write down the $\boldsymbol{G}(*)$ function by inspection. But the idea of this problem is that if you had a big machine with a trillion states and edges, as long as you can represent the Boolean function $\boldsymbol{R}(*)$ in some efficient way, you can mechanically derive the $\boldsymbol{G}(*)$ function. This is a fabulously *useful* result: you can mechanically determine which states in your finite state machine are unreachable *entirely with Boolean algebra*, you don't have to make the state graph explicitly.

8. Unate Recursive Complement Algorithm [20 pts]

In class, we talked about how to use the Unate Recursive Paradigm (URP) idea to determine tautology for a Boolean equation available as a SOP cube list. It turns out that many common Boolean computations can be done using URP ideas. In this problem, we'll extend these ideas to do unate recursive *complement*.

The overall skeleton for URP complement is very similar to the one for URP tautology. The biggest difference is that instead of just a yes/no answer from each recursive call to the algorithm, URP complement actually returns a Boolean equation represented as a cube list. We use "cubeList" as a data type in the pseudocode below. A simple version of the algorithm is below:

```
1.  cubeList  Complement( cubeList  F ) {
2.      // check if F is simple enough to complement it directly and quit
3.      if ( F is simple and we can complement it directly )
4.          return( directly computed complement of F );
5.      else  {
6.          // do  recursion idea
7.          let x = most binate variable for splitting
8.          cubeList P = Complement( positiveCofactor( F, x) )
9.          cubeList N = Complement( negativeCofactor( F, x ) )
10.         P = AND(x, P)
11.         N = AND(x' , N)
12.         return(  OR(P, N) )
13.     } // end recursion
14. } // end function
```

There are a few new ideas here, but they are mechanically straightforward.

Lines 2,3,4 are the termination conditions for the recursion, the cases where we can just compute the solution directly. There are only 3 cases:

- If the cubeList F is empty, and has no cubes in it, then this represents the Boolean equation "0". The complement is clearly "1", which is represented as a single cube with all its variable slots set to don't cares. For example, if our variables were x, y, z, w, then this cube representing the Boolean equation "1" would be: [11 11 11 11].
- If the cubeList F contains the all don't care cube [11 11 ... 11], then clearly $F = 1$. Note, there might be other cubes in this list, but if you have $F = (\text{stuff} + 1)$, it's still true that $F = 1$, and $\bar{F} = 0$. In this case, the right result is to return an empty cubeList.
- If the cubeList F contains just one cube, not all don't cares, you can complement it directly using the DeMorgan Laws. For example, if we have the cube [11 01 10 01] which is $y\bar{z}w$, the complement is clearly: $(\bar{y} + z + \bar{w}) = \{[11\ 10\ 11\ 11], [11\ 11\ 01\ 11], [11\ 11\ 11\ 10]\}$, which is easy to compute. You get one new cube for each non-don't-care slot in the F cube. Each new cube has don't cares in all slots but one, and that one variable is the complement of the value in the F cube.

Lines 6,7,8,9 are just like the tautology algorithm from class, and work exactly the same.

Lines 10,11,12 are new. The tautology code just returned yes/no answers and combined them logically. The complement code actually computes a new Boolean function, using the complement version of the Shannon expansion:

$$\bar{F} = x\bar{F_x} + \bar{x}\overline{F_{\bar{x}}} = OR(AND(x, P), AND(\bar{x}, N))$$

The AND(variable, cubeList) operation is simple. Remember that in this application, you are ANDing in a variable into a cubelist that *lacks that variable*, i.e., if you do AND(x, P) we know that P has no x variables in it. To do AND, you just i*nsert the variable back* into the right slot in each cube of the cubeList. For example: AND($x, yz + z\bar{w}$) = $xyz + xz\bar{w}$ mechanically becomes AND(x, {[11 01 01 11], [11 11 01 10]}) = {[*01* 01 01 11], [*01* 11 01 10]}.

The OR(P, N) operation is equally simple. Remember that "OR" in a cubeList just means putting all the cubes in the same list. So, this just concatenates the two cubeLists into one single cubeList.

There are a few other tricks people do in *real* versions of this algorithm (e.g., using the idea of unate functions more intelligently), that we will ignore. Note that the cubeList results you get back may not be minimal, and may have some redundant cubes in them.

(i) [15 pts] Show by hand a recursion tree (like for the tautology examples from class, except now you also have to show Boolean functions going back UP the tree) for the above algorithm URP Complement, running on the function:

$$F(x, y, z, w, p, q) = pyw + \bar{y}\bar{w}z + xy\bar{w}\bar{z} + \bar{p}q$$

(ii) [5 pts] Show with a simple Karnaugh map that your URP algorithm got the right answer for the complement.

9. Unate Recursive Algorithms for General Boolean Computations [5 pts]

The previous problem worked out the development of a URP algorithm for complement, so if we had a general cube-list form of a function $f(*)$, we could complement it. If we could do OR and AND on arbitrary functions, we could also do any arbitrary Boolean computation. OR is easy, as discussed above. But what about AND? Discuss how you could do "AND" operations on an arbitrary pair of Boolean functions represented as SOP cube-lists. We don't need to see detailed pseudo-code, a paragraph will do. How could you do this simply? It doesn't need to be efficient, it just needs to work.