# CMU Fall 01 18-760 VLSI CAD

## [120 pts[1]] Homework 1 (v1). Out Wed Aug 30, Due Wed, Sep 13.

### 1. Properties of cofactors [10 pts]

Use "ordinary" Boolean algebra to show that each of the following identities about operations with the Shannon Cofactors actually works. Assume *f* refers to *f(x1, x2, .., xn)*, *g* is *g(x1,x2,...xn)*, and *x* refers to some arbitrary variable in *x1,x2,...xn*.

- $(\overline{f})_x = \overline{(f_x)}$
- $(f \oplus g)_x = (f_x) \oplus (g_x)$ ("⊕" means Boolean "exclusive or" here.)

(**Note**: in particular, this means you start with the equation on one side of the equality, and do standard Boolean operations until you get the result on the other side. These identities may look easy enough for you to be tempted to just write a couple of sentences in English saying "look--it's obvious". **Don't** do this; do it formally, using Boolean algebra. So, for the first part here, you should think about taking that function "f" and expanding it out via the Shannon expansion, then looking at what that complement operation and positive-x-cofactor operation are going to do to it, via ordinary Boolean algebra.)

### 2. Properties of Boolean difference [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 *x1,x2,...xn*, and *x* refers to some arbitrary variable in *x1,x2,...xn*.

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

**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 is both an *f* function and a *g* function there. The only way this can be true is if the equation works when x=0 and it also works when x=1. So, the first thing to do is set x to a constant on the right hand side, and then simplify. So--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, ie, the "cofactor of an EXOR is the EXOR of the cofactors" on the left, and so on.

1. Homeworks don't always add to "100"; some are longer and are more points, some are shorter.

(c) It's helpful to recall that AND distributes over EXOR, ie $a \bullet (b \oplus c) = ab \oplus ac$

(d) It's also helpful to recall that $a \oplus a = 0$ 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.

**3. Shannon Expansion [10 pts]**

So far, we know that the Shannon expansion identity says: $F = x\,F_x + x'\,F_{x'}$

Of course, this is a s*um of products* style formula, i.e., it ORs together some terms that are each products. 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.)

**4. More 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 $f = (p+q)\,g(x,y,z) + p'\,q'\,h(x,y,z)$
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 $f = (p+q)\,g + p'\,q'\,h$. *g* and *h* will end up being related to *f* ***(and its cofactors!)*** in a straightforward way.

## 5. Unate functions: complements [ 10 pts]

Suppose we have a function $F(...)$ of several variables, and we have a SOP form for this function. We are interested in the *complement* of function F. Clearly, we could just apply the Shannon expansion in a straightforward way and write:

$$F' = x \bullet (F')_x + x' \bullet (F')_x ,$$

which is correct, but doesn't look very useful.

It turns out that if $F$ has an SOP representation (i.e., a "cover") that is **unate** in variable $x$, we can simplify this even further. If the SOP form of $F$ is *positive unate* in variable $x$, then this means that all the product terms in $F$ either have an "$x$" literal, or no "$x$" literal; no product terms can have the complemented $x$' literal. If the SOP form of $F$ is *negative unate* in variable $x$, then this means that all the product terms in $F$ either have an "$x'$" literal, or no "$x$" literal; no product terms can have the *un*complemented $x$ literal. If we have a unate variable, then we can represent the complement of $F$ like this:

If variable x is *positive* unate in SOP form of F: $F' = (F')_x + x' \bullet (F')_x ,$

If variable x is *negative* unate in SOP form of F: $F' = x \bullet (F')_x + (F')_x ,$

Note that if the variable is unate in the SOP form of $F$, then the Shannon expansion for the complement is a little simpler: one of the $x$ variables in the expansion disappears.

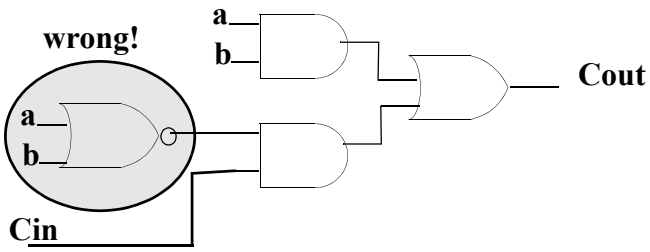
**Do this:** use Boolean algebra to show that these two "simpler" formulas for the Shannon expansion of the complement are correct. (**Hint**: think *carefully* about the SOP form of function $F$ when you know one of the variables is unate; you can (and will want to) write a simpler, more specialized Shannon form for $F$ directly, and this leads directly to the above results.)

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

The carry output of a 1-bit adder has the Boolean equation $Cout = a \bullet b + (a+b) \bullet Cin$ , where $Cin$ is the input carry bit, and $a$ and $b$ are the 1-bit numbers we want to add together. Suppose we implemented this design incorrectly, as shown below. Instead of the correct OR gate to make the $(a+b)$ term, we used a NOR gate instead and got $\overline{(a+b)}$ instead. Use the network repair idea from the notes to do gate debugging on this network. You know the correct equation for the network, and assume you know that it's the NOR gate that is wrong. Construct the Z( ) function (replace the NOR with the 4-input MUX, etc.) and show the Boolean algebra for how to apply the quantification operator to derive the four *d0, d1, d2, d3* inputs to the MUX. Do this:

- Show that this technique can in fact fix the network and describe what the MUX says you should implement to fix things

- Can you have more than one solution here? If so, what are they and what do they mean?.

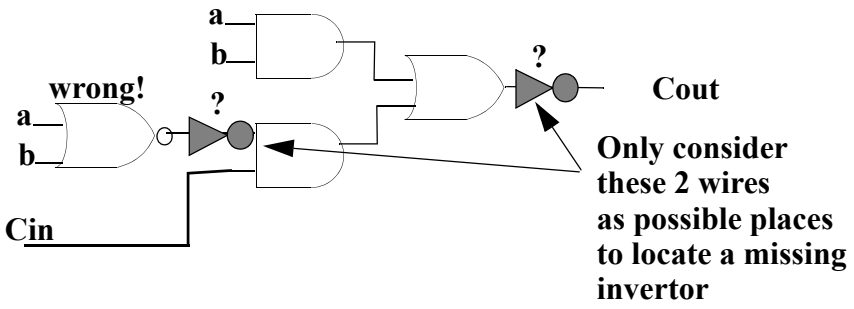



**7. More gate level debugging via quantification [10 pts]**

Consider a related 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 the same quantification approach of the previous problem 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 *d0, d1, d2, d3* variables above) and what exactly you need to quantify away.

Show how to use your strategy on the same gate level network at the previous problem:

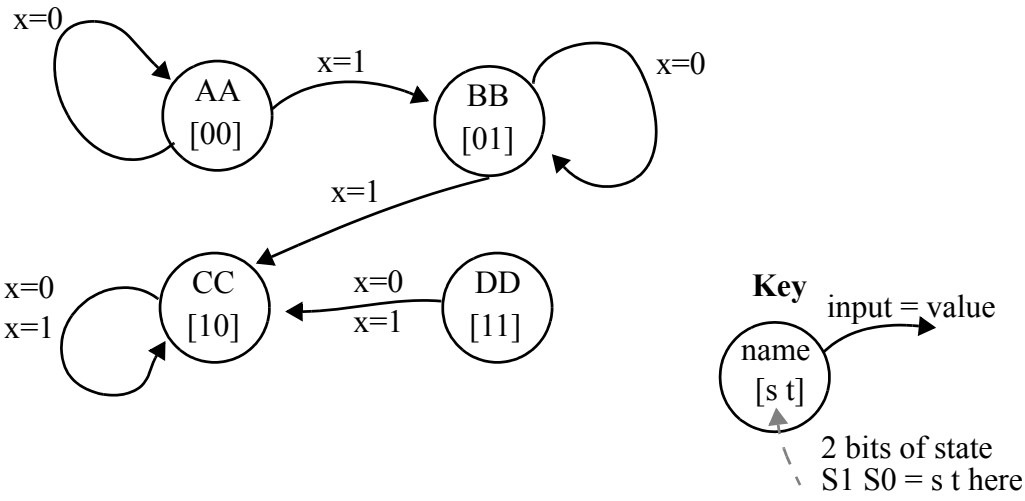



To keep the algebra simpler, only consider the 2 wires shown above as possible locations for the missing invertor. Note--you know what the right answer is by looking at this problem--but the point is to show how, by using purely Boolean manipulations, you can identify the right invertor to add. Show 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**: 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 *everywhere* in your modified network, on *every* single gate output wire, 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 the programming input. Remember: $x \oplus 0$=what? $x \oplus 1$=what? How would you design logic to enable exactly *one* of the many inverters in this network? What logical widget do you know that can be controlled to turn on just a particular selected *one* of its many outputs...? How would you add this to the problem and use it to make an analogous Z( ) function for this task?)

**8. 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 flip flops) whose values appear in brackets (e.g., [10] represents state CC) in each state bubble. This machine has just 1 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 (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 **R**, the *state transition relation function*. In general **R** has 3 kinds of inputs:

**R**( *state variables for starting state*, *input variables*, *state variables for ending state*)

The way to think about the **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**:

**R**(AA, 0, AA) = 1 means "*yes, from state AA, an input x=0 takes you to state AA*"

**R**(BB, 1, CC) = 1 means "*yes, from state BB, an input x=1 takes you to state CC*"

**R**(DD, 0, BB) = 0 means "*NO!, from state DD, an input x=0 does NOT take you to state BB*"

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:

*R*( *S1, S0*, x, *E1,E0* )

where *S1 S0* is the state assignment for the starting state, and *E1 E0* is the assignment for the ending state. So, for example:

**R**(BB, 1, CC) = 1 really means that **R**(0,1, 1, 1,0) = 1, since state BB is represented by the assignment S1 S0 = 01, and similarly the state CC has the assignment 10.

Do the following:

- Complete a truth table for **R**( ) for this little 4-state example, and create a sum of products Boolean expression for **R**( ) using a 5-variable Kmap.
- Now, let's create a new function, called **G**(*E1, E0*). **G** is again a function of the states, but it answers a different question: *is there ANY way to reach this state E1 E0*? For example, it is clear from our little state diagram that **G**(DD) = 0, i.e., there is just NO way to start from some state and take a transition that gets you to state DD. But **G**(AA)=1 because it *is* possible to get to state AA (in this case, by already being in state AA and taking the x=0 transition).

  **Do this:** show how to use the *quantification* operators to transform the **R**( ) function into the **G**( ) function. *Write* a general expression for what you have to do to **R**( ) to turn it into **G**( ). *Explain* in 1-2 sentences why it works. Then, actually use Boolean algebra to *perform* the appropriate quantification on the SOP form for **R**( ) you already derived, and show that the resulting equation for **G**( ) makes sense for this problem.

  **Don't** just write down the **G**( ) function by inspection. It's obviously trivial here. But, the idea of this problem is that if you had a **big** machine with a trillion states+edges, as long as you can represent the Boolean function **R**( ) in some efficient way, you can mechanically derive the **G**( ) function. This is a neat 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.

**9. Unate recursive complement algorithm [10 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 ides 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=(stuff + 1 ) it's still true that F=1, and 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 yz'w, the complement is clearly:

  (y' + z + 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:

$F' = x \bullet (F_x)' + x' \bullet (F_{x'})'$ = OR( AND(x, P) , AND(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 + zw') = xyz + xzw' 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.

For this problem, **DO THIS**:

- 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 URPComplement, running on the function:

  F(x, y, z, w) = yz + y'w'z + xyw'z'

- Show with a simple Karnaugh map that your URP algorithm got the right answer for the complement.

**10. Unate recursive tautology code [30 pts]**

Write a simple program (in C, or C++, or JAVA) to do unate recursive tautology. To make things simple, assume that the sum-of-products form you will read in to test has this simple format:

- First line has 1 ints, call it VARS that tells you how many variables there are.
- The next lines each describe one product term. Assume the vars are just represented as single lower-case ASCII chars: if there are VARS=3 vars in your function, then the variables are "a" "b" and "c".
- We represent a true literal "a" as "+ a", i.e., a plus sign, a space, and an "a".
  We represent a complemented literal "a'" as "- a", i.e., a minus sign, a space, and an "a".
- A product term is just a sequence of space-delimited literals terminated with a semicolon.
- An empty product term (no literals, just a semicolon) terminates the entire input.

For example, for the function **f = a + c' d' + a 'b' + a b c + a b' d**, the input would look like this:

```
4              (4 vars a b c d)
+ a ;          (1st term is   a)
- c - d ;      (2nd term is c'd')
- a - b ;      (3rd term is a' b')
+ a + b + c ;  (4th term is a b c)
+ a - b + d ;  (5th term is a b' d)
;             (empty product term ends input)
```

Use the termination rules given in the notes. You should print out something "enlightening" each time your recursive tautology routine gets called, like what SOP form it got called on, and what the result was of checking each rule, and what variable you are splitting on. This will generate a simple trace that shows what the function is doing as it runs. Ultimately it should print out YES or NO for the overall function.

Don't get hung up in fancy data structures for the cubes, some simple arrays of chars or ints will do fine. The goal is for this thing to be *small*, and *simple*, 3-5 pages of code.

**Do the following for credit:**

- Turn in your code (yeah, we want to see it) Print it out, staple it to this homework assignment.
- Turn in the *output* of your code running on the function listed in this problem.
- Also, pick one other random example with 5 or 6 variables that **IS** a tautology, and show the Kmap (just draw it by hand) to show how the cubes make a tautology then run your code on the example and hand in the printout.
- Pick one other random example with 5 or 6 variables that **IS NOT** a tautology, and show the Kmap (just draw it by hand) to show how the cubes are arranged, then run your code on the example and hand in the printout.