VLSI CAD: Logic to Layout

Programming Assignment 1:

Unate Recursive *Complement* Algorithm

In the lectures, 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, representd in Positional Cube Notation (PCN). 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*. This means: we give you a file representing a Boolean function **F** as a PCN cube list, and you will **complement** it, and return **F'** as a PCN cube list.

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 PCNcube 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
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.

Termination Conditions

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:

- **Empty cube list:** 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]**.
- **Cube list contains All Don't Cares Cube:** 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.
- **Cube list contains just one cube:** 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.

Selection Criteria

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

However, let us be more precise about how to actually implement this. Our goal is to specify this so carefully that, if you implement it correct, you will get exactly the same answer as our course test code. This makes it possible for us to grade things more fairly, since everybody is "aiming" at the same function.

Here are the rules for picking the splitting variable, in order of priority:

1. Select the most binate variable. This means, look at variables that are not unate (so, they appear in both polarities across the cubes), and select the variable that appears in true or complemented form in the *most* cubes.

2. If there is tie for this "most binate variable" and more than one variable appears in the same number of cubes, break the tie in this manner: let **T** be the number of cubes where this variable appears in *true* form (i.e., as an **x**). Let **C** be the number of cubes where this variable appears in *complement* form (i.e., as an **x'**). Choose the variable that has the smallest value of **|T – C|**. This way, you pick a variable that has a roughly equal amount of work on each side of the recursion tree.

3. If there is a *still* a tie, and several variables have the same smallest value of **|T-C|**, then choose the variable with the *lowest index*. For simplicity, in this assignment, we will assume the input variables are numbered like this: **x1, x2, x3, x4,** etc. If there is a tie at this point (variables appear in same number of cubes, with identical minimum **|T-C|** value) then just pick the *first* variable in this list. So, for example, if you get to this point, and the variables are **x3, x5, x7, x23** – you pick **x3**, since 3 is the smallest index.

4. What if there is *no* binate variable at all? Then select the unate variable that appears in the most cubes in your cubelist.

5. What if there is a tie, and there are *many* such unate variables, that each appear in the *same* number of cubes? Then, again select the one with the lowest index.

We can say this more concisely like this:

- **if** (there are binate variables) {
  pick the binate variable in the most cubes, and if necessary,
  break ties with the smallest **|T-C|**, and then with smallest variable index;
  }
  else { *// there are no binate variables*
  pick the unate variable in the most cubes, and if necessary,
  break ties the smallest variable index;
  }

Working with Returned Cubelist Complements

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.

File Format

We are using a very simple text file format for this program. Your code will read a Boolean function specified in this format, complement this function, and then write out a file in exactly this *same* file format. The file format looks like this:

- First line of the file is a number: how many variables in the equation. This is a positive integer. We number the variables starting with index 1, so if this number was 6, the variables in your problem are: **x1, x2, x2, x4, x5, x6**. You should write your program to handle up to **10 variables**.

- Second line of the file is a number: how many cubes in your input cube list. This is a positive integer. If there are 10 cubes in your input, this is a "10". You need to be able to handle at least 2^{10} different cubes in your program.

- Each of the subsequent lines of your file describes one cube – you have the same number of lines as the second line of your file. Each of these lines also has a set of numbers: the first number on the line says how many variables are *not* don't cares in this cube. If this number is, e.g., 5, then the next 5 numbers on the line specify the true or complemented form of each variable in this cube. We use a simple convention: if variable **xk** appears in true form, then put integer **"k"** on the line; if variable **xk** appears in complement form **(xk')** then put integer **"-k"** on the line. The file will always order these variables in *increasing* index order. So, if your cube has x3x5x9', the line should say **"3 3 5 -9"** and not some other order, e.g., **"3 -9 3 5"**. Spaces on the line do not matter.

That's it. This is really very simple. Suppose we had this function as input:

F(x1, x2, x3, x4, x5, x6) =x2x4x5' + x2'x4'x6 + x1x2x3'x4' + x5x6

Then the input file format would look like this:

INPUT FILE

```
6
4
3 2 4 -5
3 -2 -4 6
4 1 2 -3 -4
2 5 6
```

Note: to keep things simple, your output file has exactly this *same* format. Also, we promise not to try to be "tricky" and so we will **not** give you either the Boolean function **F=1**, or **F=0,** as inputs. Just to be clear, **F=1** has just one cube in it, but with only don't cares in this cube. If **F** is a function of 6 variables like our previous example, the function **F(x1,x2,x3,x4,x5,x6)=1** is this file:

INPUT FILE

```
6
1
0
```

And, we promise **not** to give you the input function **F=0**, which is just an empty list with no cubes in it. **F(x1,x2,x3,x4,x5,x6)=0** is this file:

INPUT FILE

```
6
0
```

It's not hard to detect these as special cases and return the trivial complement, so we will just not do this case, to focus on the interesting part of the algorithm. But, we can give you any *other* function, of up to **10** variables, with up to $\mathbf{2^{10}}$ cubes, as input.

Doing This Program Assignment: Overview

You will write a program that takes a simple text file in this ASCII format, that specifies the input PCN cube list. You will read this file, and use these URP ideas to compute the complement, You will write an output file, in exactly this same text format, that represents your computed complement result. We will specify **five** files with test inputs that you will complement. You will upload these to the Coursera site, and our auto-grader will check them and score them.