VLSI CAD: LOGIC TO LAYOUT:
Programming Assignment 1:
Unate Recursive *Complement* & Boolean Calculator Engine

Rob A. Rutenbar
University of Illinois at Champaign-Urbana

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, represented in Positional Cube Notation (PCN). It turns out that many common Boolean computations can be done using URP ideas. In this problem, we'll first extend these ideas 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.

(We will also provide an optional Extra Credit part – you don't have to do it! – that extends these further to a full *Boolean Calculator Engine*.)

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 specific 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:

$$F' = x \bullet (F_x)' + x' \bullet (F_{x'})' \;=\; \mathrm{OR}(\ \mathrm{AND}(\mathbf{x}, \mathbf{P}),\ \mathrm{AND}(\mathbf{x'}, \mathbf{N})\ )$$

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 **20 variables**.

- Second line of the file is a number: how many cubes in this cube list. This is a positive integer. If there are 10 cubes in this file, this is a "10". You need to be able to handle at least $\mathbf{2^{20}}$ different cubes in your program.

- Each of the subsequent lines of the 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" for the **required** part of this assignmnt, 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 **20** variables, with up to **2^{20}** 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 a few 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.

Mechanical Details: Submission to Coursera Site & Auto Grading

Please see the Coursera website for this assignment, and we will describe these in detail.

ILLINOIS

VLSI CAD: LOGIC TO LAYOUT:
Programming Assignment 1 Extra Credit:

Boolean Calculator Engine

Congratulations! If you got this far, we assume you have a working URP Complement code, that can read a PCN cubelist file, and write a PCN cubelist file. So, what can do next? We can extend these ideas, with a little bit of work, to build a *complete* Boolean Calculator Engine.

What do we need to build this complete engine?

- **Manage multiple functions:** Your previous code just worked with one input function, complemented it, and wrote out the PCN result. You need to be able to manage several Boolean functions now.
- **Mutiple function input:** Your previous code could read just one PCN file, to specify one Boolean function. You need to extend your code to be able to read several files, each specifying a different Boolean function.
- **NOT**(***F***): You need to be able to complement an arbitrary function. You *already* have this – this was the required part of this assignment.
- **OR**(***F, G***): You need to be able to do a Boolean OR on two functions **F1, F2**. You *already* know how to do this – it was in your URP complement code.
- **AND**(***F, G***): You need to be able to do a Boolean AND on two functions **F1, F2**.
- **Boolean Calculator command language:** You need to be able to read a file that tells you what Boolean operations to perform, on what Boolean functions.
- **Function output:** You need to be able to write one of the Boolean functions you are calculating to the output, in the standard PCN ASCII format.

As we shall see – these are really not too difficult. It's just more code, and you can decide if this is something you would like to do.

Managing Multiple Boolean Functions

To make things easy, we are just going to use *numbers* for all the functions your program will read, write, or manipulate. So, our functions are named: **F0, F1, F2, F3, F4, …** But, to specify them, we just to tell you: **0, 1, 2, 3, 4, …** etc.

This means we can use a number to specify a Boolean function to read as input, to create inside your calculator engine, or to write to the output as a file. Function numbers start at **0**. You need to be able to support **32** functions, maximum.

Multiple Boolean Function Input

Since our functions are named, **F0**, **F1, F2, F3, …,** we will give you a set of ASCII input files in our standard PCN format, named **0.pcn, 1.pcn, 2.pcn, …** etc. These will be located on the Coursera site, for you to download. Your program will need to be able read some of these inputs, as specified by a simple Boolean command language.

Boolean Function Output

Since our functions are named, **F0, F1, F2, F3, …,** you need to be able to choose one the functions you have represented inside your code, call it **Fn**, and write it to an output file in our standard PCN format, named **n.pcn**. The only difference in this new code is you need to be able to manage several functions, and pick one to write to the output.

Boolean Calculator Command Language

Your program will read a separate file that has a very simple command language, in ASCII, that tells your Boolean engine what to do. Again, our goal is to make this very simple, so you can focus on the Boolean parts, and not on this input/output stuff. The command file has this format:

- **Command format:** Every line specifies a single operation: an input operation, a Boolean operation, or an output operation. Each line starts with a single ASCII character to specify the operation, and then either one, two, or three integers, to specify what Boolean function to do.
- **Input operation:** a line of the form "`r n`" (`n` is an integer) tells you to read input file `n.pcn`, and store it in your program as Boolean function **Fn**.
- **NOT operation:** a line of the form "`! k n`" (`k,n` are integers) tells you to complement Boolean function **Fn**, and store the result **Fn'** into function **Fk**. In other words, this operation does: **Fk = Fn'**. It is always OK to just overwrite **Fk**, if it already exists.
- **OR operation:** a line of the form "`+ k n m`" (`k,n,m` are integers) tells you to perform this operation: **Fk = Fn + Fm** (Boolean OR). It is always OK to just overwrite **Fk**, if it already exists.
- **AND operation:** a line of the form "`& k n m`" (`k,n,m` are integers) tells you to perform this operation: **Fk = Fn • Fm** (Boolean AND). It is always OK to just overwrite **Fk**, if it already exists.
- **Output operation:** a line of the form "`p n`" (`n` is an integer) tells you to write Boolean function **Fn** to an output file called "**n.pcn**", in our standard ASCII output format. We will never ask you to output a function that you have not yet created.
- **Quit operation:** the line "`q`" tells you to stop parsing the command file, you are done with processing commands.

As a simple example, suppose we want to calculate the exclusive OR of function **F2** and **F3**, and store the result in **F0**: **F0 = F2 exor F3**. Here is the command file. We have added comments for readabilty – but these will *not* appear in the actual file.

```
COMMAND FILE
r 2               // read in F2
r 3               // read in F3
! 4 2             // F4 = F2′
! 5 3             // F5 = F3′
& 6 2 5           // F6 = F2 * F5, ie, F2 F3′
& 7 3 4           // F7 = F3 * F4, ie, F3 F2′
+ 0 6 7           // F0 = F6 + F7, ie F2 F3′ + F3 F2′
p 0               // output F0 = F2 exor F3
q                 // we are done
```

Technical Extensions: PCN OR and AND

The description above is just to allow us to specify several Boolean functions as PCN ASCII files, give you a sequence of Boolean commands to process, and tell how to write out a more general result. This is just code – there's nothing new here.

So, what is *new* here, that you don't know yet? Answer: Boolean OR and Boolean AND. The required part of this code is the complicated new stuff: URP based PCN complement. But this will only do the "**!**" operation; how do you do "**+**" and "**&**".

It turns out, these are both surprisingly easy:

- **Boolean OR:** Actually, you already know this. This was line 12 in the URP complement pseudocode at the start of this document. Recall: to perform **Fn + Fm**, just **concatenate** the PCN cube lists for **Fn** and **Fm**. This means, take the cube list for **Fm**, and put at *after* the last cube in the list for Fn. So, if **Fn** is a list of 100 cubes, and **Fm** is a list of 537 cubes, then **Fn+Fm** is a list of 100+537=637 cubes. Its first cube is the first cube of **Fn**; its last cube is the last cube of **Fm**. So, the only "new" thing here is that you must expose the OR operation as a subroutine that you can call for an arbitrary pair of functions.

- **Boolean AND:** This is new. Be careful to note that the AND operation on lines 10-11 of the complement pseudocode only does the AND of a *single variable* and a cube, e.g., **x•xyz'.** To perform the general operation **Fn•Fm**, use DeMorgan's Law. We can write this so that it looks like code as follows:
  AND(Fn, Fm) = NOT(OR(NOT(Fn), NOT(Fm)))
  In words: use your URP complement routine to compute **!Fn** and **!Fm**, and store them someplace. OR these two complements together to produce **!Fn + !Fm**. Then complement the result; **!(!Fn + !Fm).** So, once you add an OR( ) routine to your NOT( ) routine, you can implement AND( ) easily as a DeMorgan Law set of calls to these routines!

So, the reason we make this an Extra Credit assignment is that there's not a lot of new ideas to deal with – just some extra code, if you feel you want to do a little more work, to build a *much* more functional Boolean engine.

Doing This Extra Credit Assignment: Overview

You will extend your URP program so that it can read an ASCII format Boolean Command File. Commands in this file will instruct your program how to read other PCN files, of the form **1.pcn, 2.pcn, 3.pcn**, etc, and how to write PCN files of the same format.

We will give you the following, to do this extra credit assignment:

- Several Boolean functions, specified in PCN ASCII format, in files named with the **n.pcn** convention. You can download these from our website.
- A few small command files, in the new Boolean Command Language format, that will work these **n.pcn** files. You run your code on each command file, let it read and write as specified. Each of the command files will write out exactly **ONE NEW** Boolean function at the end, which will *always* be **0.pcn**. You will upload your results for **0.pcn** for each command file you try to run to the Coursera site, and our auto-grader will check it and score it.

Three more details, just to be clear:

- To keep things simple, we also guarantee that each command input file will only use Boolean function files **n.pcn, m.pcn**, etc, that have *exactly the same number* of Boolean variables. It is just more book-keeping to make you handle functions that each have a different number of variables, and it doesn't really add anything technically interesting. So, one command file might only use Boolean function files that all have 10 variable. Another command file might use different Boolean function files that all have 12 variables. Once you read the first input file, you know how many variables all the remaining possible input files will have.
- We still guarantee no "extreme" Boolean functions, like **Fn=0** or **Fn=1** for all inputs. You can handle these edge-cases if you like, but we won't make any command files that require them.
- Although this is a correct way to describe and manipulate Boolean function, it is not a very efficient way. You may have many duplicate cubes in your representation, for example. There is nothing in our algorithm description for dealing with the fact that, e.g, **xyz' + xyz' +xyz' = xyz'**. It turns out that the techniques we describe in later parts of the course are aimed at dealing with this problem. But URP methods on PCN data structures are still an important, foundational idea to understand, because this is a nice introduction into thinking about Boolean functions as data structures, that are always maniplulated by recursive algorithms.