Complementary of Boolean Function Using Unate Recursive Paradigm

Sidi M. Aourid

Erin Mills Group

Research & Development

5205 Glen Erin Dr, Mississauga, ON, L5M 5N6

2023-05-23

Contents

Unate Recurisve Complement

Objective

The idea here is to use the Unate Recursive Paradigm (URP) to determine a complement of a Boolean function F. For that, we use the complement version of Shannon's expansion version. This is similar to the idea to determine *Tautology* for a boolean function using URP. The function F is represented by Positional Cube Notation (PCN) cube list in a file. The complement code returns a new boolean function $\overline{F}$ as a PCN cube list as well.

---

1 Boolean Algebra

The boolean algebra is the basic mathematics needed for the logic design for any digital systems. It is defined by the set $B = \{0, 1\}$ and two operations denoted by $+$ and $\cdot$. They are also called disjunction and conjunction, and satisfy the commutative and distributive laws with identity elements are 0 and 1, respectively. B^n is the multi-dimensional space spanned by n boolean variables. Each variable x or its complement x' (denoted by x' or $\bar{x}$) is referred to as a literal and products of literals is often called cubes (i.e: $x_1.x_2'$).

1.1 Basic Theorems

1. Idempotent Laws:

$$x + x = x \qquad x.x = x \tag{1}$$

2. Involution law: $(x')' = x$

3. Laws of complementarity:

$$x + x' = 1 \tag{2}$$

$$x.x' = 0 \tag{3}$$

4. Commutativity law:

$$x.y = y.x \tag{4}$$

$$x + y = y + x \tag{5}$$

5. Associative Law

$$(xy)z = z(yz) = xyz \tag{6}$$

$$(x + y) + z = x + (y + z) \tag{7}$$

$$= x + y + z$$

6. Distributive Laws:

$$x(y + z) = xy + xz \tag{8}$$

$$x + yz = (x + y)(x + z) \tag{9}$$

7. DeMorgan's Laws

$$(x + y)' = x'y' \tag{10}$$

$$(xy)' = x' + y' \tag{11}$$

8. Elimination Law:

9. consensus Law:

$$xy + x'z + yz = xy + x'z \tag{12}$$

The exclusive OR defined as follows:

$$0 \oplus 0 = 0 \qquad 1 \oplus 1 = 0 \tag{13}$$

$$1 \oplus 0 = 1 \qquad 0 \oplus 1 = 1 \tag{14}$$

Note that, $X \oplus Y = 0 \iff X = Y$. The exclusive OR can be used to compare 2 circuits. In fact, if 2 circuits have the same functionality (ie, *equivalent*), the exclusive OR of their outputs should be 0.

1.2 Boolean Functions

A completely specified Boolean function is a mapping between 2 Boolean spaces B^n and B^m defined by:

$$F : B^n \to B^m$$
$$x \mapsto F(x)$$

When the function is not completely specified, there exists some points where the function is not defined and called don't care conditions. These points correspond to some input patterns that can never occur and for which the output is not observed. In this case the output space can be represented by $\{0, 1, X\}^m$, where X denotes a don't care condition.
On-set, *Off-set* and *DC set* are subsets of the domain $\{0, 1, X\}^m$ where the function takes the values 1, 0 and X respectively.
The different laws we mentioned are used to simplify the expression of the boolean function. Simplifying an expression reduces the cost of its implementation using gates. Two different class of simplification methods exist: Graphical methods and Algebraic methods. Graphical methods are used for small function, usually with less than 5 variables.

Example. *Let us consider a three-dimensional Boolean expression.*

$$F(x, y, z) = xy + xz' + y(x'z + z')$$

The boolean function defined above with 3 variables, has 8 literals. A vertex in the B^3 space, is any product of literals, for example, xyz'. The function F, can be evaluated at each vertex

by substituting a value of 0 and 1 to each variable. When, the number of variables is small, we can use a truth table to specify the output values of the boolean function. A truth table for an n-variable expression will have 2^n rows.

1.3 Normal Forms of a Boolean Expression:

Any Boolean function can be expressed either as:

- **Sum-of-Products: SOP** called also DNF, and consists of a disjunction (+) of conjunctions ($\cdot$) of literals. (i.e: $F = xyz + x'y'$)
- **Product-of-Sum: POS** called also CNF, if it consists of conjunctions ($\cdot$) of a disjunction (+) of literals, (i.e: $F = (x + y + z)(x' + y')$).

1.4 Boole-Shannon Expansion Theorem

Here, we review a class of a fundamental result known as Shannon Expansion (For the sake of simplicity, we consider single-output functions, i.e m = 1).
Let $F(x_1, x_2, \ldots, x_i, \ldots, x_n)$ a boolean function of n variables, then the Shannon's expansion function is given by:

$$F(x) = x_i F_{x_i} + x_i' F_{x_i'} \tag{15}$$

where:

$$\begin{aligned} F_{x_i'} &= F_{x_i=0}(x) = F(x_1, x_2, \ldots, 0, \ldots, x_n) \\ F_{x_i} &= F_{x_i=1}(x) = F(x_1, x_2, \ldots, 1, \ldots, x_n) \end{aligned} \tag{16}$$

- F_x and $F_{x'}$ are called respectively the **Positive** and **Negative Cofactors** of $F(x)$ with respect to variable x_i.
- F_{x_i} and F'_{x_i} are boolean functions and don't depend on variable x_i.

Example. *Let us consider the same Boolean function :*

$$F(x, y, z) = xy + xz' + y(x'z + z')$$

Then:

$$F_y = F(x, 1, z) = x + xz' + (x'z + z') = x + z$$

$$F_{y'} = F(x, 0, z) = xz'$$

Both F_y and $F_{y'}$ are independent on y.

Remark: Shannon expansion can be expanded to multiple variables. In this case, the Shannon Cofactor with respect of x_i and x_j can be expressed as: $F_{x_i x_j}$, $F_{x_i x'_j}$, $F_{x'_i x_j}$, $F_{x'_i x'_j}$.

For our previous example, $F(x, y, z)$ can be written as:

$$F(x, y, z) = xyF_{xy} + x'yF_{x'y} + xy'F_{xy'} + x'y'F_{x'y'}$$

which is a Boolean function of z.

The expansion theorem can be used to prove or disprove interesting identities.

Example. *We can show that for every boolean function :*

$$F(x, y) = F(x + y) \cdot F(x' + y) = F(1) \cdot F(y)$$

By using the Shannon expansion, we can write:

$$F(x, y) = x' \cdot F_{x=0} + x \cdot F_{x=1} = x' \cdot F(1) \cdot F(y) + x \cdot F(1) \cdot F(y) = F(1) \cdot F(y)$$

Remark: The Shannon expansion can be extended with respect to a function $f(x)$ instead of variable x. The following illustrates this fact:

$$F\Big(g(x)\Big) = g(x).F(1) + \overline{g(x)}.F(0)$$

Example. *Let consider the following example , where* $g(x, y) = x + y$*:*

$$F(x, y) = (x + y).F(1) + \overline{(x + y)}.F(0) = (x + y)F(1) + \bar{x}.\bar{y}.F(0)$$

1.5 Cofactors Properties

The following properties are very useful, they can help in getting cofactors of complex formulas.

1. Complements:

   $F'_x = (F_x)'$ The cofactor of complement is complement of cofactor

2. Boolean Operations

   a. $(F \cdot G)_x = F_x \cdot G_x$ *(Cofactor of AND is AND of cofactors)*

   b. $(F + G)_x = F_x + G_x$ *(Cofactor of OR is OR of cofactors)*

   c. $(F \oplus G)_x = F_x \oplus G_x$ *(Cofactor of XOR is XOR of cofactors)*

Three operators are important in Boolean algebra: Boolean difference, consensus (universal quantifier) and smoothing (existential quantifier) [1]

1.6 Boolean Difference

The boolean difference of a function $F(x)$ with respect to variable x_i is given by:

$$\frac{\partial F(x)}{\partial x_i} = F_{x_i} \oplus F_{x'_i} \tag{17}$$

The boolean difference w.r.t. x_i compares the value of F, when $x_i = 0$ against when $x_i = 1$. $\frac{\partial F(x)}{\partial x_i} = 0 \iff F_{x_i} = F'_{x_i}$. In this case, F is not sensitive to changes in input x_i, we say it is *unobservable*.

1.7 Complement

The complement of the function F by using Shannon expansion is given by:

$$\overline{F} = x.\overline{F_x} + \overline{x}.\overline{F_{\bar{x}}} \tag{18}$$

Proof

$$\begin{aligned} F &= x.F_x + \bar{x}.F_{\bar{x}} \\ \Rightarrow \overline{F} &= (\bar{x} + \overline{F_x}).(x + \overline{F_{\bar{x}}}) \\ &= \bar{x}.x + \bar{x}.\overline{F_{\bar{x}}} + x.\overline{F_x} + \overline{F_x}.\overline{F_{\bar{x}}} \\ &= \bar{x}.\overline{F_{\bar{x}}} + x.\overline{F_x} + (x + \bar{x}).\overline{F_x}.\overline{F_{\bar{x}}} \\ &= \bar{x}.\overline{F_{\bar{x}}}(1 + \overline{F_{\bar{x}}}) + x.\overline{F_x}.(1 + \overline{F_{\bar{x}}}) \\ &= x.\overline{F_x} + \overline{x}.\overline{F_{\bar{x}}} \end{aligned}$$

From this equation, we note that, to compute a complement of function F, we have to compute the complements of its cofactors. This is similar to URP **Tautology Check using Shannon's Expansion**, where both factors has to be a Tautology as well.

It is necessary to verify the correctness of a circuit design before producing chips. Let us denote the function realized by the new design by: G. The verification is the task to decide whether $G(x) = H(x)$.

For the equality test, we may design a representation of $F(x) = H \oplus G$. The equality test is equivalent to the satisfiability test for F(x).

$$F(x_1, \cdots, x_n) = 1$$

2 Representation of Boolean Functions

Boolean functions are related to hardware while data structures are important tools in algorithm design and therefore related to software. There are different ways to represent a

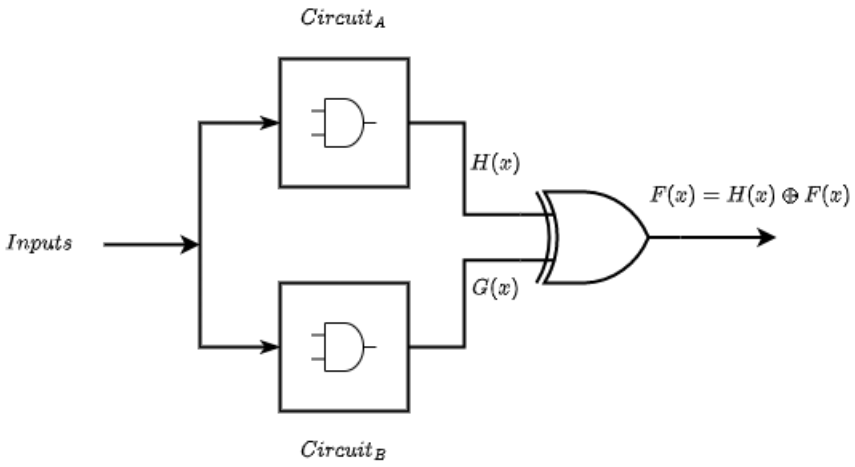

Figure 1: Boolean Difference

boolean function. We are considering here the Positional Cube Notation (PCN). So, for any boolean function in the form of a SOP, it can be represented by a list of PCN. The idea consists to create for each variable in a cube one slot, represented by 2 bits 0 or 1. So, each cube in the boolean function is represented by noting if the variable is:

- *True*: represented by slot **01**
- *Complemented*: represented by slot **10**
- *Absent*: represented by slot: **11**

Example. *Let considerer the following boolean function:*

$$F(a,b) = \bar{a} + bc$$

So: $\bar{a} = [10, 11, 11]$, since variable a **is complemented** and variables b and c **are absents**. The same for $bc = [11, 01, 01]$, a **is absent**, b and c **are *true***. $F(a,b)$ is then represented by the following list of cubes: $\{[10, 11, 11], [11, 01, 01]\}$. We can also write:

$$F(a,b) = \{[10, 11, 11], [11, 01, 01]\}$$

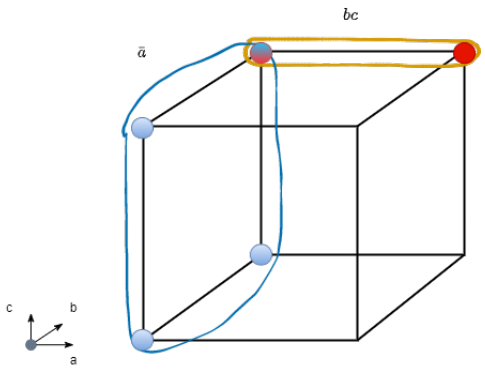

Figure 2: Function $F(a, b) = \bar{a} + bc$

It turns out that common boolean operations: *AND*, *OR*, *INV* *etc*$\cdots$ can be performed using PCN representation

- **Complement:** The complement of a variable in PCN notation is very easy if it's 01: 10

- **PCN AND:**

  We suppose here that we are AND*ing* a variable into a cubelist. So the AND operation consists to insert the variable back into the right slot on each cube of the cube List. For example:

$$\begin{aligned} AND(x, yz + zw') &= AND\Big(x, \Big\{[11, 01, 01, 11], [11, 11, 01, 01]\Big\}\Big) \\ &= \Big\{[01, 01, 01, 11], [01, 11, 01, 01]\Big\} \end{aligned}$$

- **PCN OR:**

  The OR is just concatenates cubes into one single cube list. For example:

$$\begin{aligned} OR(xz, yz + zw') &= OR\Big([01, 11, 01, 11], \Big\{[11, 01, 01, 11], [11, 11, 01, 01]\Big\}\Big) \\ &= \Big\{[01, 11, 01, 11], [01, 01, 01, 11], [01, 11, 01, 01]\Big\} \end{aligned}$$

2.1 Complement Algorithm

The following *pseudocode* highlights the different steps of the Algorithm:

Algorithm 1 Complement (F)

Require: cubeList of F
  if (F is simple) **then**
    return F'
  else if **then**
    $x \leftarrow mostBinateVariable$
    $P \leftarrow Complement(postiveCofactor(F, x))$
    $N \leftarrow Complement(negativeCofactor(F, x))$
    $P \leftarrow AND(x, P)$
    $N \leftarrow OR(x^{'}, N)$
    return $OR(P, N)$
  end if

2.2 Termination and Selection

1. Termination Condition: They are only 3 cases:

   - Empty Cube List: if the cubeList is empty (i.e, this represents boolean function **0**), then the complement is **1**
   - CubeList contains all don't cares cube: if **F** contains only don't cares cube (i.e. [11, 11,..., 11], this represents boolean function **1**), then the complement is **0**
   - CubeList contains just one Cube: If the cubeList contains only One cube, the the complement is computed using Morgan law. For example: $F = y'z = [11, 10, 01] \Rightarrow F' = y + z' = \{[11, 01, 11], [11, 11, 10]\}$

2. Selection Criteria: The idea here is to pick the right splitting variable. For that we use the following scenarios:

   - Select the most binate variable. This corresponds to the variable that appears in true or complement from the most cubes

- In case of tie and more than one variable appears in the same cube, break the tie with this manner: Choose the variable with the smallest $|T - C|$, where T - Number of cubes where the varaibles appears in *true* and C = Number of cubes where this variables appears in *Complement*
- In case several varaibles has the same $|T - C|$, choose the variable woth the lowest index.
- In case, there is no binate varaible, choose the unate variable that appears in the most cubes.
- In case of tie, choose the one of lowest index

2.3 File Format

A function is represented by a text file format. The file format specify: number of variables, number of cubes and each cube by its literals. The following table shows the file format:

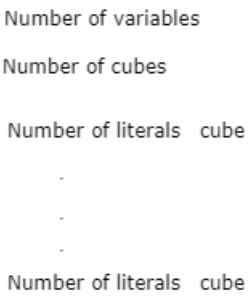

Figure 3: File format for a boolean function

Example. *Let us consider the Boolean function :*

$$F(x_1, x_2, x_3, x_4, x_5, x_6) = x_2x_4x_5' + x_2'x_4'x_6 + x_1x_2x_3'x_4' + x_5x_6 + x_1'x_6$$

Then the file format for the function F will be:

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

Table 1: Boolean function representation

3 References

[1] Giovanni De Michelli, Synthesis and Optimization of Digital Circuits

[2] D. Gary Hachtel, Fabio Somenzi, Logic Synthesis and Verification Algorithms

[3] Rob A. Rutenbar, VLSI CAD: Logic to Layout