Novel Synthesis Methodology in Digital IC Design and Automation to Reduce NRE costs and Time-to-Market

Basireddy Karunakar Reddy

A Dissertation Submitted to
Indian Institute of Technology Hyderabad
In Partial Fulfillment of the Requirements for
The Degree of Master of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Department of Electrical Engineering

July, 2015

Declaration

I declare that this written submission represents my ideas in my own words, and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

[illegible]

(Signature)
Basireddy Karunakar Reddy
EE12M1006

Approval Sheet

This thesis entitled "Novel Synthesis Methodology in Digital IC Design and Automation to Reduce NRE costs and Time-to-Market" by Basireddy Karunakar Reddy is approved for the degree of Master of Technology from IIT Hyderabad.

[illegible]

External Examiner

[illegible]

Internal Examiner

[illegible]

Adviser

Co-Adviser

[illegible]

Chairman

Acknowledgements

I am thankful to Dr. Amit Acharyya for patiently listening to my foolish ideas, and guiding and supporting me to make them innovative ideas.

I would like to thank Dr. Asudeb Dutta, Dr. Shiv Govind Singh, and Dr. Siva Rama Krishna for providing me the required theoretical knowledge through the courses and projects.

I also want to thank Mr. Srinivas Sabbavarapu for helping me through out in my project.

Finally, I would like to thank my friends who always helped me to shape my character.

Dedicated to

Kṛṣṇa, My Family and Friends………

Abstract

The number of incremental and iterative steps in the digital IC design & automation methodology will decide the non-recurring-engineering (NRE) costs and time-to-market (TTM). Since the aforementioned factors are the major driving factors of the IC design industry, many algorithms were proposed in the last few decades to minimize/optimize the number of design steps in the conventional digital IC design & automation methodology. However, the conventional frontend and backend designs have been carried out separately, which has limited the further minimization of design steps. Here we propose a novel digital IC design & automation methodology, which reduces the NRE costs and TTM by merging the frontend and backend designs partially. It maps the input RTL description directly to their corresponding physical layouts (derived using the existing CAD tools and stored in a pre-computed library) without going through the all the steps in conventional logic and physical synthesis process. As part of the proposed methodology, we use a pre-computed library which stores all required physical layouts and their Boolean functions. We have exploited the functional symmetry and negation-permutation-negation (NPN) class representations to decoct the library size and number of comparisons. The functional symmetry reduced the number of required pre-computed circuits in our experiments from 1031 to 222 (464.4% reduction in the memory size) and helps in maintaining the regularity in the design, which is a major concern for engineering change order.

To minimize the number steps further, complexity and memory requirements, we have proposed a new graph data structure called Shannon Factor Graph (SFG) and cut-less mapping technique for technology dependent mapping. The preliminary findings show the effectiveness of the proposed methodology.

Contents

Chapter 1

Introduction

1.1 Conventional digital IC design and automation methodology

The Non-Recurring Engineering (NRE) costs, which is one time investment for Computer-Aided Design (CAD) tools, design effort, mask generation, and design time (which is part of the NRE cost) have been considered as major factors to decide the product cost and productivity. With continued technology scaling, the design of current and future generations of digital integrated circuits (ICs) is increasingly becoming more complex. Due to such complexity, meeting design cost constraints with high productivity at the same time is a truly daunting task [1]. According to [2] "cost of design is the greatest challenge to continuation of the semiconductor roadmap" and the annual tool costs (part of NRE) per designer increase 3.9% per year. Hence if the NRE costs can be cut down, overall price of the product will reduce, leading to increased demand for the product. On the other hand, when the design time of a product goes up, consequently its demand and revenue earned will be reduced. As for example, delay of half of the design time in productivity will lead to reduction in demand and revenue by 75% [2]. Therefore significant amount of research is going on in the IC design industry and academia to reduce the NRE cost and time-to-market.

These two factors are majorly influenced by design & automation methodologies/flows that one follow during the design of any system (digital in our case). Therefore, we start our discussion with conventional digital IC design and automation methodology [3].

Conventional industry-standard IC design methodology includes hierarchical design and automated synthesis steps. The design abstraction usually starts from the register transfer-level (RTL) with an aim to obtain the final IC layout through logic synthesis (frontend design) and physical synthesis (backend design). The design methodology follows the original Y-chart representation proposed by Gajski and Kuhn [3], also known as the Gajski Chart (shown in Fig. 1.1). The basic premise of this methodology is to use separate the frontend and backend CAD tools in incremental and iterative steps to achieve the final design layout. Frontend CAD tools convert the RTL description of a design to the gate level netlist through translation, optimization and technology mapping [4, 5] and the backend converts the netlist into final IC layout through floor planning, placement and routing [1, 3]. This whole process is termed as clockwise Gajski chart flow. The clockwise Gajski chart flow represents the design in three hierarchical domains, functional/behavioral, structural and physical/geometrical, and refinement is conducted in steps as shown in Fig. 1.1.

Functional representation: At highest level of abstraction, the designer will consider what the chip does. For instance, the Boolean expression y=ab + (ab)' tells us about only its function where its output is 'y' and inputs are 'a' and 'b', but does not say anything about its implementation and structure of the cell. Functional representation of a design may be captured on several levels.

Structural representation: It's a bridge between functional representation and geometrical representation. The Boolean function is mapped to the corresponding components (ex: NAND) and connections based some objective function (are, delay... etc.) however it doesn't say anything about the physical parameters, like positions of the components on the printed circuit board silicon chip. Depending upon the design entry structural representation may serve the functional/behavioral representation.

Geometrical representation: The final representation ignores, as much as possible, what the design is supposed to do and binds its structure in space (physical design) or to silicon (geometrical design).

Each of the above representations may have several local representations as shown in the Fig. 1.2.

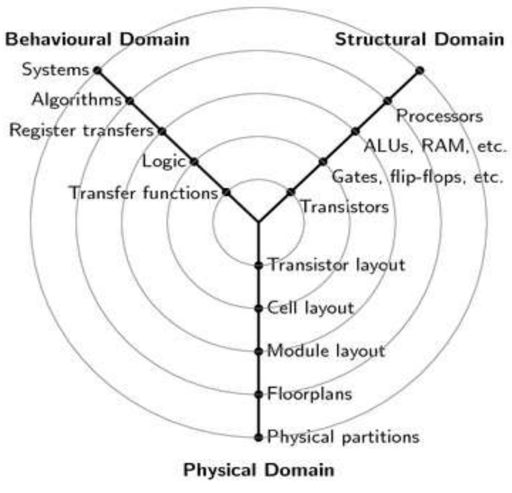


Figure 1.1: Gajski-Kuhn chart

The conventional design automation methodology, shown in Fig. 2, consists of incremental and iterative steps from RTL description to final physical. At each design step a separate algorithm should be run by the respective CAD tools to convert the RTL design description to corresponding gate level netlist (fig.1). The translated logic is optimized to get best performance in terms of area, delay and power. Before mapping the optimized logic to the target technology gates, a technology independent mapping is done, later possible logic optimizations are done through local transformations. Based on the given specifications this process is carried iteratively. Mapped logic (netlist) is fed as the input to the backend tools

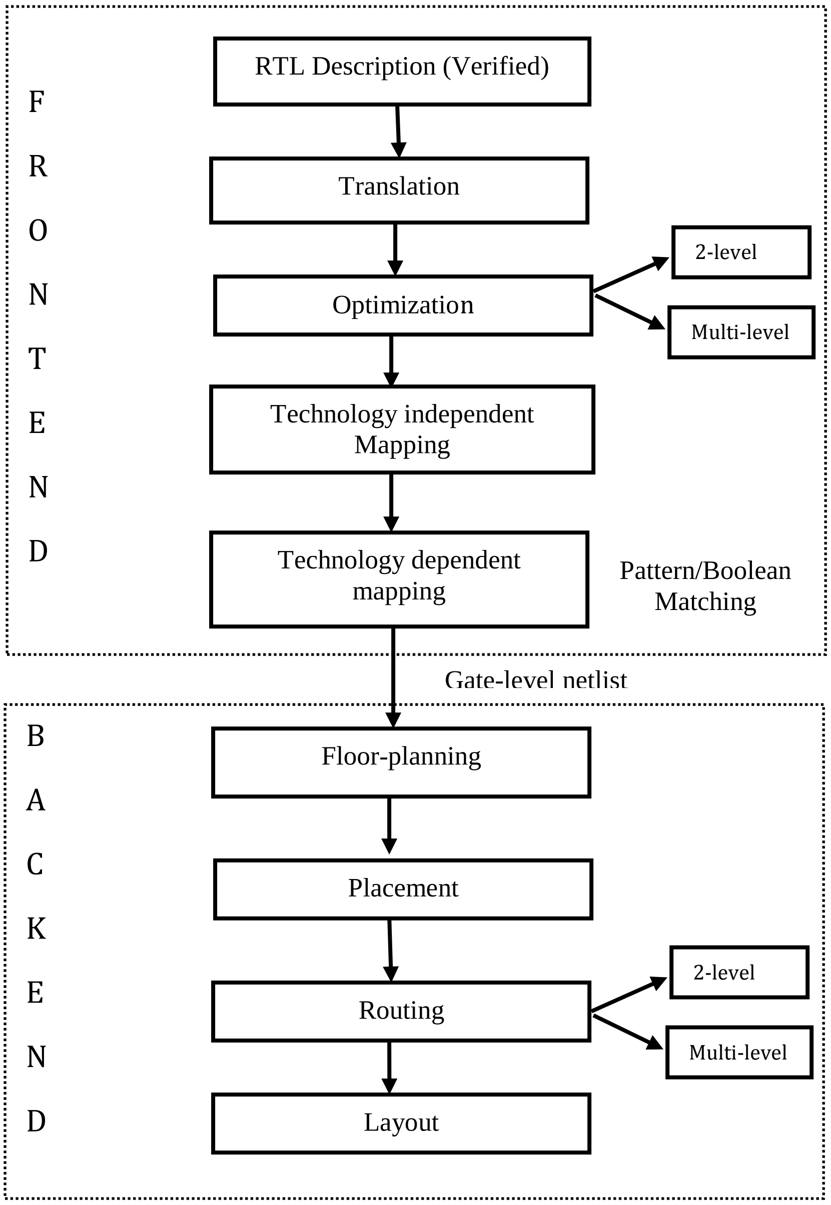


Figure 1.2: Simplified conventional digital IC design & automation methodology

and they mostly carry out the floorplan, placement and routing iteratively. Initial floorplan and placement influence the final routing congestion. The routing tool has to route wires among the gates of a block (intra routing) and also among the different blocks (inter routing). Routing within the block decides the congestion probability. To do all this, different algorithms have to be run by the tool at each step iteratively. So we can conclude that we need different tools for frontend and backend, and large design time to get a quality product.

Therefore with high productivity and low cost target, there has been growing research interest in CAD tools development domain to reduce the NRE costs for complex systems design and also to simplify design steps to improve time-to-market. However meeting such conflicting design trade-offs among the design complexity, high productivity and low cost requirements is highly challenging.

To address the design complexity various techniques have been proposed so far. A unified model of the design representation is proposed in [3] where the design is described in three hierarchical domains (behavioral, structural, physical) with different levels of abstraction. The circuit-performance including power consumption, area and delay depends on the quality of the logic synthesis [6]. Therefore the main research focus has been on the optimization of logic synthesis process [4, 5, 7-9], mainly the Graph based logic synthesis using Binary Decision Diagrams (BDD) and And-Invert-Graphs (AIG) which have already been proven to be an effective way for logic optimization and technology mapping. Recently in [10], a shift from the logic synthesis is observed, where the new circuit/netlist structure is retrieved from the pre-computed library, instead deriving it each time by skipping the optimization phase. Although this is an unconventional attempt, but the number of circuits to be stored using this approach is significantly large. Moreover the corresponding library has to be loaded with new circuits, when they are not found in the existing library. Therefore large memory is required for such design strategy to store all industrial benchmark circuits, which increases search space and design time. Therefore, we envisage a design automation methodology with reduced number of steps combining these two to reduce NRE costs and time-to-market, which is discussed next.

1.2 Proposed Digital IC Design and Automation Methodologies

We have proposed two methodologies, one is cut-based and another one is cut-less.

1.2.1 Cut-based methodology

We propose a novel, unconventional and unified design methodology [12] by merging the logic synthesis (frontend) with physical design (backend) partially. Our proposed methodology benefits from reduced number of design steps by creating a direct link from the RTL/behavioral description to the physical design in the original Gajski chart [3]. The direct physical design step is facilitated through the previously stored and pre-computed technology libraries. To minimize the storage requirements only the physical designs of negation-permutation-negation (NPN) class functions along with their Boolean expressions [11] are stored instead of the extensive industrial benchmarks as proposed in [10]. Thus it drastically reduces the iterative loading and re-loading overheads used in the conventional design methodology. As for an example, 1478 NPN class functions are needed in [10] to represent only the 4-input frequently appearing industrial circuits, whereas our proposed methodology reduced this number to 1031 (a reduction of 30%), which can be used to map any Boolean expression without any limitation on number of input variables of the function.

We have also exploited the functional symmetry along with the NPN class representation to minimize the library size further and to improve the regularity of the We show that due to such unified design methodology with reduced complexity, NRE costs and Time-to-market are reduced significantly. The functional symmetry reduced the number of required pre-computed circuits in our experiments from 1031 to 222 (464.4% reduction in the memory size) and helps in maintaining the regularity in the design, which is a major concern for engineering change order.

The aforementioned methodology uses the cut-based technology mapping technique for mapping the input RTL/Boolean expressions/truth-tables to the cells of the pre-computed library. The following proposed methodology points out the drawbacks of the cut-based methodology and shows a possible solution for solving them.

1.2.2 Cut-less methodology

The methodology proposed in [11-14] are cut-based methodologies, which require cut-enumeration, storing of cuts and computation of canonical form for mapping. The cut-enumeration is computationally complex and requires more memory for storing the cuts. If there are n number of nodes and k is the cut size, then the possible number of cuts is $O(n^k)$, which is exponentially growing with the cut size and number of nodes. Due to this the runtime and required memory grows exponentially, thereby it affects the overall design cycle for highly complex designs. Here, we propose a cut-less mapping technique and a new graph data structure [15], called Shannon Factor Graph (SFG), for mapping the input RTL/behavioral description directly to their final physical layouts. The information stored at each node of the SFG can be used to map the nodes to the technology dependent cells without enumerating the cuts. The embedded intelligence of the SFG takes the advantage of the library cell's size. This feature of the SFG minimizes the graph building time and required memory significantly. The proposed cut-less mapping technique can improve the design time and required memory further, which will reduce the time to market and NRE costs significantly.

1.3 Thesis Outline

The thesis is organized as follows. Chapter 2 describes the AIG and its construction, functionally reduced AIG (FRAIG), then proposed Shannon Factor Graph (SFG) for cut-less mapping. Chapter 3 talks about the cut-based technology mapping and its drawbacks. Chapter 4 explains the Negation-Permutation-Negation (NPN) class representation and functional symmetry along with the proposed modification. Chapter 5 presents the proposed cut-based design and automation methodology. Chapter 6 describes the proposed cut-less design and automation methodology. Chapter 7 concludes the discussion with the future scope of work.

Chapter 2

And-Inverter Graph (AIG) and Shannon Factor Graph (SFG)

2.1 And-Inverter Graph (AIG)

AIGs were introduced by Kuehlmann and Krohm for combinational equivalence checking [16]. There are several data structures used for representation of the subject graph, which is technology independent one used for mapping to the technology dependent standard cells. Here the subject graph is represented as AIG for proposed cut-based methodology.

2.1.1 Definition

Let (X, A), A≤X×X, be a directed acyclic graph where each node n€X has either no incoming arcs or exactly two incoming arc, are called Inputs. Nodes with two incoming arcs are called 'And' nodes. Let inv be a function from A to the set {0, 1}. The tuple G = (X, A, inv) is called an And Inverter Graph (AIG). If inv(a) = 1 then arc a is said to be inverted. Inverted arcs indicated with a bubble on solid line, and uninverted arcs with a solid line (see Fig. 2.1).

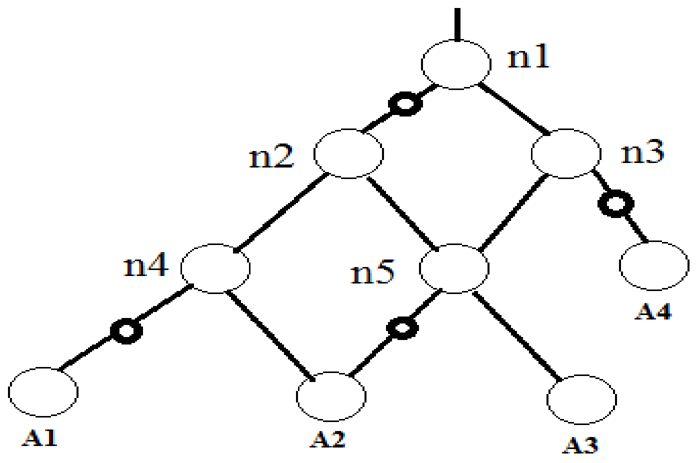


Figure 2.1: AIG graph

Every node and edge in an AIG corresponds to a Boolean function, called the function of the node or edge. Every Input node n of an AIG is associated with a formal Boolean variable Xn. The functions of the other nodes and edges are defined in terms of these Boolean variables.

Formally, the semantics of an AIG G is specified by defining a valuation function f that maps every node and edge of G to a Boolean function. If e is the edge (n, i), we define

$$f(e) = \begin{cases} f(n) & \text{if } i=0 \\ \sim f(n) & \text{if } i=1 \end{cases}$$

If n is a node, we define

$$If(n) = \begin{cases} 0 & \text{if } \textit{n is the zero node} \\ X_n & \text{if } \textit{n is an input mode} \\ f(e_l).f(e_r) & \textit{otherwise} \text{ (} n \text{ is an And node with input edges } e_l \text{ and } e_r\text{)} \end{cases}$$

This recursive definition of f is well-formed since the nodes in an AIG can be topologically ordered (as it is a directed acyclic graph). An AIG G that has exactly one node n with no out-going arcs is called a single output AIG. In this case we often abuse terminology and talk of the function of G. This should be understood to be the function of n.

2.1.2 AIG construction

AIGs for Boolean functions can be constructed starting from different functional representation [17]

SOP: Given an SOP representation of a function, it can be factored [18] and the factored form can be converted into the AIGs. Each two-input OR-gate is converted into a two-input AND-gate using the DeMorgan rule.

Circuit: Given a circuit representation of a (multi-output) Boolean function, the (multi-output) AIG is constructed in a bottom-up fashion, by calling a recursive construction procedure for each PO of the circuit. When called for a PI node, the procedure returns the elementary AIG variable. Otherwise, it first calls itself for the fanins of a node and then builds the AIG for the node using the factored form of the node. When an AIG is constructed from a circuit, the number of AIG nodes does not exceed the number of literals in the factored forms.

When the AIG is constructed from a BDD, the number of AIG nodes does not exceed three times the BDD number since each MUX can be represented using three ANDs. It follows that the size of the AIG is proportional to the size of the circuit or BDD.

Quantifications performed on AIGs have an exponential complexity in the number of quantified variables because quantifying each variable is done by ORing the cofactors and can potentially duplicate the graph size. Except for quantification, Boolean computation is more robust with AIGs than with BDDs. This is because Boolean operations on AIGs lead to the resulting graphs whose size is bounded by the sum of the sizes of their arguments, while in the case of BDDs the worst case complexity of the result is equal to the product of the sizes of the arguments.

2.2 Functionally Reduced AIGs (FRAIGs) [17]

FRAIGs are “semi-canonical” because no two nodes in a FRAIG structure have the same function in terms of the primary inputs, but the same function may have different FRAIGs structures.

Canonicity: A representation of a Boolean function is canonical if, for any function, the representation is unique.

AIGs are not canonical, that is, the same function can be represented by two functionally equivalent AIGs with different structure. Although in general it is computationally expensive to remove redundancies in an AIG (detecting if two nodes compute the same function is Co-NP Complete), in practical implementations some easily detected redundancies are prohibited by enforcing the following conditions:

Structural Hashing: There is at most only one And node with a given pair of edges as inputs (since two nodes with same inputs compute the same function).

Redundant And Elimination: There is no And node with both inputs the same.

Constant Zero Elimination: There is no And node with an edge and its complement (since it computes the same function as the Zero node).

Constant Zero Propagation: There is no And node with the Zero edge as an input (since it computes the same function as the Zero node).

Constant One Propagation: There is no And node with the One edge as an input (since it computes the same function as the other input).

We abuse terminology and refer to these conditions collectively as structural hashing conditions. An AIG that satisfies these conditions is said to be structurally hashed. We generally assume that an AIG is structurally hashed.

The aforementioned AIG based subject graph can be used for cut-based technology mapping, but, when it comes to the cut-less technology mapping the AIGs give inferior results [19]. However, there were attempts to minimize the number of cuts [20] to reduce the complexity and memory requirement (will be discussed in chapter 4).

We propose a new graph data structure for representing the subject graph called, Shannon Factor Graph (SFG), to facilitate the cut-less technology mapping, which is equally applicable to LUT-based and standard cell based mapping.

2.3 Shannon Decomposition Theorem [21]

A Shannon Decomposition is a method to represent any Boolean function as the sum of two sub-functions of the original function. A cofactor is a sub element of a Shannon decomposition generated by setting the value of a given variable to either "0" or "1". A cofactor, which is generated for a function F by setting a variable xi to 0 is called the negative cofactor of the function F with respect to xi, otherwise it is called positive cofactor (setting to "1"). A cube-cofactor is obtained by setting more than one variable to "0" and/or "1", i.e a cube-factor is a cofactor from a cofactor. Equation (1) shows the mathematical representation of Shannon decomposition theorem.

F(x0, x1, x2, …, xi) = x0 * F(1, x1, x2…., xi) + x0' * F(0, x1, x2, ……, xi)-----------------(1)

Where * and ' represent the AND and NOT functions respectively. For an example, the negative cofactor of the function F(x0, x1, x2, x3) = x0*(x1+x2) +x3 with respect to x0 is x3, whereas the cube-cofactor with respect to x0(= 0) and x3(= 1) is 1.

2.4 Brief Introduction to Binary Decision (BDD)

For BDD construction, we can start with the truth-table (2n values) or with more compact representations, like Boolean expression. In the case of Boolean expressions, a top-down procedure can be used to derive the diagram by repeated applications of the classical Shannon decompostion formula (1).

Consider a 5-variable function, G=x1*(x0'*x2+x2'*x4') + x4'*(x0'*x1+x1'*x3).

We begin by setting x0=0 in G (without considering the proper variable ordering for BDD size reduction) to obtain the function G0, which must be realized below the x0=0 branch. We then do for x0=1 to obtain the function G1. Similarly the procedure should be repeated until the Shannon cofactor values are constant "1" or "0". The size of the BDD and construction time increases exponentially with the number of variables and converting them to a canonical representation is again an exponential complex problem. The other problem with the BDDs is, its structure does not represent the final target graph. So to overcome the problems associated with the AIG and BDD representation we come up with a new graph representation, whose construction is more like a BDD. More details on BDD can be found in [22-26]

2.5 Proposed Shannon Factor Graph (SFG) [15]

The nodes of the SFG represent the Cofactors or cube cofactors value and the level, which will be used as node ID for finding the appropriate cell in the pre-computed library, and edges represent the connecting wires among the nodes. Unlike Binary Decision Diagrams (BDD) [22-26] and AIG [6, 27], the structure of the SFG helps in eliminating the cut-enumeration and pruning, computation of truth-tables and Negation-permutation-Negation (NPN) class representatives [11, 28] for each cut. We propose two different algorithms for SFG construction, which can be used for cut-less mapping, to improve the runtime, graph size and required memory. Thereby it improves the runtime and reduces the required memory drastically.

We found that, computation of constant zero and one, non-decomposed, and shared nodes (explained in next section) is critical in minimizing the runtime and graph size for complex circuits. By considering the nature of nodes while constructing the graph makes the SFG semi-canonical, because the nodes at each level of the SFG will have uniquely represented nodes. The proposed SFG construction algorithm (algorithm 2) computes the constant one and zero, non-decomposed and shared nodes on-the-fly, thereby it improves the overall size of the graph, which in turn reduces the final area and building time.

2.5.1 Non-decomposed, constant one, constant zero and shared nodes

Non-decomposed, constant one and zero, and shared nodes represent the nature of SFG nodes. A non-decomposed node is the node which has similar node(s) in the SFG, which have the same cofactor or cube cofactor value and level. Non-decomposed nodes improve the logic sharing, minimize the graph size and final area. If the building time of a non-decomposed is t_b and the number of non-decomposed nodes in the SFG is L, then the building time and graph size will be reduced by a factor $(L-1)*t_b$ and L-1 respectively. A node which receives two constant zeros (ones) is called constant zero (one) node, i.e. the constant zero (one) node will have all zeros (ones) in its corresponding truth-table. If there are no constant zeros or ones as input to the node, then the node represents function of a typical Shannon cofactor (two AND gates, one inverter and one OR gate), otherwise the node represents an AND gate or AND gate followed by an OR gate. If the two primary outputs have the nodes, which have the same functionality and level, then those nodes are called shared nodes. Shared nodes minimize the size of the SFG.

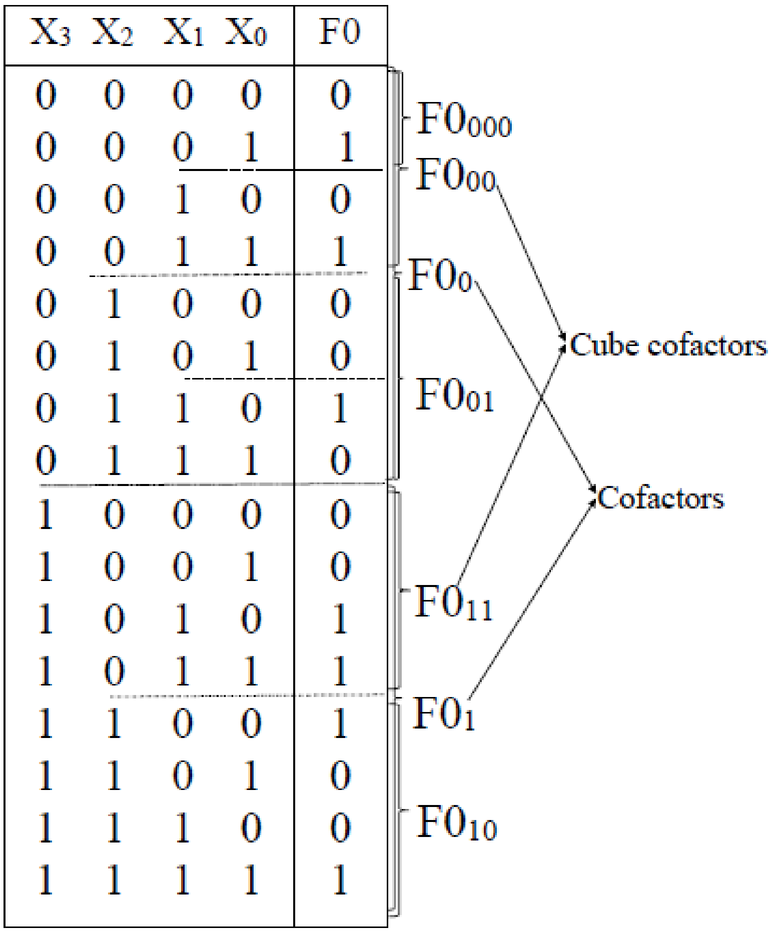
2.5.2 Construction of proposed SFG

The proposed SFG can be built with and without considering the nature of nodes.

2.5.2.1 SFG construction without considering the nature of nodes

Algorithm 1 shows the pseudo code for the SFG construction without considering the non-decomposed, constant one and zero, and shared nodes. The input to the SFG construction algorithm is truth-table and input size of the largest cells (MaxCellSize) available in the library. Using the equation (1) the Shannon cofactors and cube cofactors of the given truth-tables will be computed by successively dividing the truth-table.

Consider the 4- input truth-table given in the Fig. 2.2 The variables will be selected depending the order they appear in the truth-table. For calculating the Shannon cofactors of the variable X3, the truth-table will be partitioned into two equal. The first half values in the output (F0) represent the negative cofactor F00 and the remaining half represent the positive cofactor F01. Now each half (F00 and F01) will be decomposed, which is nothing but finding the cofactor of the cofactor (cube cofactor). First half of the F00 is the cube cofactor F000 of the function F0, when the variables X3=0 and X2=0.



X_3	X_2	X_1	X_0	F0
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 2.2: Shannon Cofactor and cube cofactor computation by successively decomposing the Truth-table.

This process continues till the cofactors/cube cofactors of all variables are calculated. Decimal values of the computed Shannon cofactors/ cube cofactors and their levels (number of primary input variables that are in the fan-in cone of a node) will be used as the node IDs. Table 2.1 helps to classify the nodes and to determine the leaf value (carries the bottom node value to the node from which it is decomposed). Column I represents the positive/negative cofactor value of the node (leaves), the Column II and III determine the leaf value when the number of primary inputs are more than two and equal to two respectively.

Table 2.1 Determining the Values of the Leaves of the Node

Cofactor value (binary)	**Leaf value (n>2)**	**Leaf value (n=2)**
All zeros	**Constant zero**	**Constant zero**
All ones	**Constant one**	**Constant one**
01	**-**	**a’ (literal)**
10	**-**	**a (literal)**

Since the size of the truth-table grows exponentially with the number of variables, Boolean expressions are taken as input for the large functions (number of variables >16). In case of Boolean expressions, cofactors/cube cofactors are computed by substituting 0/1 in place of the selected variable (s). Then, Boolean expression of the cofactors and cube cofactors are used as the node IDs, which will be used for mapping and finding the nature of nodes (whether nodes are receiving constant one or zero, non-decomposed nodes and shared nodes) [15]. The SFG has embedded intelligence, it considers the input size of the pre-computed library cells and constructs the graph till the number of inputs to the bottom nodes of the graph is equal to the input size of the pre-computed library cells. Once the whole SFG is constructed IDs of the nodes, except for bottom nodes, will be deleted to free up the memory (explained in detail in next section). Unlike the Binary Decision Diagrams (BDD), the proposed SFG contains fewer nodes (see Table 2.2) and requires less graph building time due to its on-the-fly size reduction [15] and its structure reflects the final target graph, which is technology dependent. Fig. 2.3 (a) shows the basic structure of the SFG, without considering the nature of nodes, for an arbitrary 6-input Boolean function F.

SFG construction algorithm takes the advantage of the size of the library cells during graph construction to improve the graph size and runtime. The SFG is constructed to a level that the bottom most nodes of the SFG will have a level of MaxCellSize. This is because of the fact that the nodes which receive inputs from MaxCellSize number of primary input variables can be mapped with the library cells whose size is not less than MaxCellSize. For an instance, if F is a 10-input Boolean function and MaxCellSize is 4, then the bottom most nodes of SFG of the Boolean function F will have a level of 4, i.e. only 6-variables are considered for the Shannon decomposition. Since bottom most nodes have a level 4, they can be mapped directly with the 4-input library cells. In this way, by considering the size of the library cells, the SFG construction algorithm reduces the graph size and improves the graph building time. The SFG constructed using the algorithm 2.1 is not canonical, because it has nodes which are redundant. In order to make the SFG semi-canonical, there should not be any constant one and zero, shared and non-decomposed nodes. The proposed algorithm 2.2 takes this into consideration and makes the SFG semi-canonical.

Algorithm 2.1 SFG Construction without considering the non-decomposed, constant one and zero, and shared nodes

```
1: int Truth2ShannonFactorGraph(truthtable, MaxCellSize)
2: {
3: int j,f, numvar, truthlength;
4: tempTruth=truthtable;
5: truthlength=length of the tempTruth;
6: numvar=log2(truthlength);
7: compute the decimal value of tempTruth save in f;
8: for j=1 to numvar-MaxCellSize
9: {
10: compute the Shannon cofactors and cube cofactors by successively dividing the tempTruth;
11: compute the decimal values of the cofactors and cube-cofactors and save in f;
12: }
13: return f;
14: }
```

2.5.2.2 SFG construction-considering the nature of nodes

Algorithm 2.2 shows the pseudo code for the SFG construction and computing the non-decomposed, constant one and zero, and shared nodes on-the-fly. It considers the size of the library cells and nature of nodes to minimize the graph building time and size. Fig. 3.3 (b) shows the basic structure of the SFG, with considering the nature of nodes, for an arbitrary 6-input Boolean function F. At every level, the value of Shannon cofactors or cube cofactors are checked to find the non-decomposed, constant one and zero nodes before proceeding to the next phase of decomposition. The cofactor or cube cofactor values of the nodes having the same level will be compared, then the nodes which are having the same cofactor or cube cofactor value in their truth-table (output decimal value) are classified as non-decomposed nodes and only one out of all non-decomposed nodes will be considered for the next phase of Shannon decomposition. If there are m number of non-decomposed nodes, then only one out of 'm' will be considered for the decomposition and the remaining 'm-1' will be implemented from the decomposed node. So, there is no need to spend time to decompose all 'm' non-decomposed nodes, which is runtime overhead. As the number of non-decomposed nodes increases, the graph building time and size decrease.

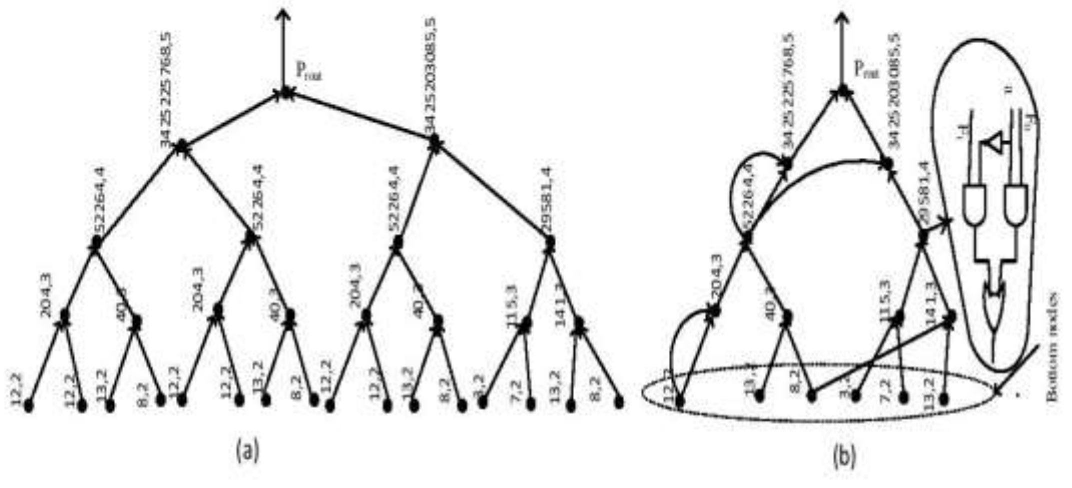


Figure 2.3: Basic structure of the Shannon Factor Graph (SFG), assuming that size of the largest cells in the pre-computed library is 2 (MaxCellSize) (a) without considering the nature of nodes and (b) considering the nature of nodes of a 6-input Boolean function F.

The constant one and zero nodes are found by identifying the nodes whose cofactors have all zeros or ones in their truth-table or corresponding decimal value. Once the constant one or zero nodes are found, they are no more considered for the Shannon decomposition and will be used to simplify their parent nodes. Even if the constant one or zero nodes are decomposed, the resulting nodes (children) will also be constant one or zero nodes. So considering the constant one or zero nodes for decomposition is redundant, runtime overhead and increases the graph size. The shared nodes among the primary outputs are determined by comparing the nodes of one primary output with the other primary output. If there are any shared nodes, only one of them will be considered for the Shannon decomposition and remaining nodes will be implemented from the decomposed node. From the Fig. 2.3 it is clear that, the size of the SFG can be minimized significantly even for small functions by identifying the nature of nodes on-the-fly (from 31 to 15) and experimental results show that, the reduction in SFG size is more prominent for bigger functions.

2.5.2.3 SFG construction for multiple outputs

Generally the functionally equivalent nodes (Non-decomposed nodes), Non-Shannon nodes in the DAG will be calculated after constructing the graph, which is a runtime overhead. However, finding the functionally equivalent/non-decomposed nodes on-the-fly at every level of the SFG construction can reduce the graph building time and memory significantly. Algorithm 1 & 2 shows the pseudo code for constructing the proposed SFG for single output functions and finding the non-decomposed nodes and constant one or zero nodes on-the-fly and it is generalized for multi-output functions, which is presented in algorithm 2.3. The

Algorithm 2.2 SFG construction and computation of non-decomposed, constant one and zero nodes on-the-fly

```
1: graphconstruct(int truth_table, int MaxCellSize)
2: {
3: num_previousCubecofactor ←1
4: number_of_variables=log2(length(truth_table);
5: temp_truthtable=truth_table;
6: temTruth=truth_table;
7: for i = 2 to num_of_variables-MaxCellSize+1
8: {
9: if all Cube_cofactors are constant one/zeros
10: Break;
11: for j = 1 to num_previousCubecofactor
12: {
13: ShannonCofactor(i,k)=temp truthtable(1:end/2);
14: k=k+1;
15: ShannonCofactor(i,k)=temp truthtable((end/2+1):end);
16: k=k+1;
17: }
18: num_previousCubecofactor  ←number of co-factors in the Shannon Cofactor for i
19: Compare all the Cubecofactors and remove non-decomposed nodes and Constant one or
zero Nodes
20: }
21:}
```

nodes which have the same level will be compared to find the non-decomposed nodes. Similarly, the constant one and zero node will be computed based on their cofactor value. If a node has all zeros (ones) then it will be considered as constant zero (one) node. The constant one or zero nodes being found during the SFG construction, will not be decomposed further and these nodes will be used for simplifying the functionality of the parent node. A node which receives only one constant node represents an OR-gate and a node which receive the constant one and constant zero represents the primary input variable. Similarly a node receiving only one constant zero represents the AND gate. For common cofactors (non-decomposed nodes), which are having the same truth-table (output decimal value), only one of them will be decomposed and the remaining nodes will be implemented from the cube cofactors of the decomposed node. Computation of non-decomposed nodes

Algorithm 2.3 Algorithm for Multi-output functions to find Shared nodes

```
1: t←number of primary outputs
2: for j=1 to t
3: {
4: read truth_table(j);
5: Temp_truth=truth_truth(j);
6: graphconstruct(Temp_truth);
7: Multi_Cofactor(t)=ShannonCofactor;
8: }
9: for i=1 to t-1
10: for k=i + 1 to t   {
11: out1=Multi_Cofactor(i);
12: out2=Multi_Cofactor(k);
13: compare cofactor of out1 and out2
14: if common nodes are there, keep one and make all NULL
15: }
```

will take O(M) time, where M is the number of nodes in the SFG decomposition at a particular level.

For multi-output functions, the Shannon cofactors will be computed for each primary output. Then the cofactors and cube cofactors of each primary output will be compared with the cofactors and cube cofactors of the remaining primary outputs having the same level (lines 9-14 of the algorithm 2.3). If there are any shared nodes, which are having the truth-table will be grouped and only one from each group will be implemented. This will reduce the graph size, memory and improves the final circuit area, but the delay will remain the same. Algorithm 2.3 will take O(mn) time to find out the shared nodes, where m and n represent the number of primary outputs and the number of cofactors at each level of graph for each primary output.

2.5.2.4 Comparison of the algorithms 2.1 & 2.2

We extensively verified the proposed algorithms for SFG construction with the standard benchmark circuits [29,30]. The proposed algorithms are implemented in MATLAB running on a Xeon processor (3.4GHZ, 4GB RAM) operating in Linux-based environment. The benchmark circuits taken from [30] (circuits 10-14 in Table I), which are PLA format, converted into truth-tables manually and using the Simple-Solver [31]. Benchmark circuits taken from [29] (circuits 1-9 in Table I), which are in verilog format, converted into

Boolean equations using the ABC tool [32], then truth-tables are harvested from the Boolean equations.

Table 2.2 shows the variation of graph size and runtime (graph building time) of the SFG with and without considering the non-decomposed, constant one and zero, and shared nodes. Column 1 represents the standard benchmark circuit name. Column 2 shows the number of primary inputs and outputs of the benchmark circuit. Column 4 and 5 represent the time taken to build the SFG and number of nodes of the SFG (graph size) without considering the nature of the nodes (non-decomposed, constant one and zero, and shared) respectively. Column 6 and 7 gives the graph size and time taken to build the SFG, considering the nature of nodes.

We considered 3-input library cells, so the SFG is decomposed to a level where the bottom most nodes will have a level of 3. The size of the SFG graph increases drastically, when the non-decomposed, constant one (zero) and shared nodes are not considered in constructing the SFG. This is because, these nodes will also be considered for the Shannon decomposition, which augments the graph building time and size of the SFG. Since the non-decomposed nodes will have a representative node, which will be considered for the decomposition, all these nodes can be implemented from the decomposed node assuming that there is no fan-out limitation on a node. Thus, by considering only one representative for m nodes can improve the runtime and graph size significantly.

Assume that there are m sets of non-decomposed nodes, each set has n nodes (level and decimal values of the cofactors are same) and td is the time required to decompose each node. Now each set can be implemented ('n-1' nodes) from a single node, which is considered for the Shannon decomposition. So the total time taken to decompose the nodes will be m*td, which saves O(m*n) ((n-1)*m*td) time and reduces the graph size of similar amount. Same is applicable for shared nodes case also. Non-decomposed nodes represent the nodes within a primary output, whereas shared nodes represent among the primary outputs.

If any constant one or zero nodes are found during the SFG construction at any level, then those nodes will not be considered for the decomposition to minimize the graph building time and size of the SFG. The Shannon decomposition of constant one or zero nodes (parent) results in constant one or zero (children), which are redundant to consider for the further decomposition. Constant one and zero nodes reduce the size and graph building time of the SFG drastically compared to the non-decomposed and shared nodes. Column 8 shows the runtime ration with and without considering the nature of nodes. At an average non-decomposed, constant one and zero, and shared nodes minimize the runtime by a factor 5.5

(for few circuits it is around 100). But the interesting observation is for few circuits, the runtime ratio is 1, this is due to the presence of the constant one or zero and non-decomposed nodes near the bottom most nodes which increases the runtime. Similarly for

Table 2.2 Comparison of Runtime and Graph Sizes of the Proposed SFG With and Without Considering nature of nodes

S. No	Circuit name	No. of inputs/outputs	SFG without NCS		SFG with NCS		Speed up (R1/R2)	Reduction in Graph size (S1/S2)
			Runtime (R1)	Graph Size (S1)	Runtime (R2)	Graph Size (S2)		
1	cm138	6/9	0.013	135	0.005	27	2.6	5
2	cmb	16/4	0.58	65532	0.039	85	15.2	770
3	cm163a	16/5	0.72	81915	0.12	131	6.2	625
4	cm162a	14/5	0.17	20475	0.077	96	106	193
5	cm152a	11/1	0.007	511	0.006	88	1.2	5.8
6	alu2	10/6	0.011	1536	0.01	271	1.1	5.7
7	cm151a	12/2	0.02	2046	0.02	118	1	17
8	ex4	6/9	0.02	135	0.0083	101	2.2	1.4
9	ex1	9/19	0.64	2413	0.11	376	5.7	6.4
10	max46	9/1	0.002	127	0.003	72	0.67	1.8
11	7-bit even parity	7/1	5.7e-4	31	5.2e-4	5	1.12	6.2
12	mux4	6/1	3.7e-4	15	3.1e-4	9	1.2	1.67
13	majority	5/1	2.4e-4	7	2e-4	6	1.2	1.2
14	4gt13	4/1	2.4e-4	3	2.4e-4	3	1	1
	Total		~2.2	174881	~0.4	1388		

few circuits the variation in the graph size (circuits 8, 11-14) is minimum, this is because of the input size of the benchmark circuits is almost equal to the size of the library cells. At an average, considering the constant one and zero, shared, non-decomposed nodes during graph construction reduces the size of the SFG by a factor of 126 (for a few circuits it is around 700). So, by finding the nature of nodes on-the-fly makes the SFG semi-canonical, which in turn minimizes the graph size and runtime.

2.5.3 Comparison of SFG with AIG and BDD

To verify the superiority of the proposed SFG, We compared with AIG and BDD using the standard benchmark circuits [29], taken from the different publicly available benchmarks. First, the Boolean equations of the benchmark circuits were computed using the ABC tool [32], then the truth-tables of each Boolean equations were harvested using a simple program. For the sequential circuits, only the combinational part of the logic is taken by cutting at the register edges.

Table 2. 3 shows the comparison of SFG with AIG and BDD in terms of graph size and variation of graph size and runtime for different variable sizes of the library circuits . Column I represents the benchmark circuit name, column II and column III show the graph size for AIG and BDD for different circuits respectively, computed using the ABC tool [32]. Column IV represents the SFG size and runtime for different values of m. In column IV, m represents the input size of the cells stored in the pre-computed library. The value m=0 signifies that the construction algorithm does not consider the variable size of the

Table 2.3 Comparison of the Proposed SFG with AIGI and BDD in Terms of Graph Size and Variation of Graph Size and Runtime for Different MaxCellSizes

Circuit name	**BDD size**	**AIG size**	**Proposed Shannon Factor Graph size**							
			m=0	**Runtime**	**m=3**	**Runtime**	**m=4**	**Runtime**	**m=6**	**Runtime**
alu2	**923**	**547**	**270**	**0.001**	**263**	$0.89e^{-3}$	**227**	$0.79e^{-3}$	**87**	$0.32e^{-3}$
alu4	**5059**	**1023**	**843**	**0.008**	**802**	**0.006**	**741**	**0.002**	**670**	**0.001**
bigkey	**4548**	**6805**	**5409**	**0.06**	**5020**	**0.06**	**4571**	**0.04**	**4302**	**0.03**
b9	**204**	**139**	**127**	**0.031**	**120**	**0.024**	**94**	**0.018**	**76**	**0.012**
c8	**255**	**312**	**291**	$78e^{-6}$	**214**	$71e^{-6}$	**201**	$60e^{-6}$	**180**	$52e^{-6}$
c499	**368**	**414**	**320**	**0.003**	**301**	**0.0026**	**273**	**0.002**	**241**	**0.002**
c880	**640**	**383**	**263**	**0.008**	**241**	**0.006**	**210**	**0.003**	**185**	**0.001**
cm150	**54**	**76**	**61**	**0.009**	**53**	**0.007**	**47**	**0.006**	**43**	**0.004**
cm151	**26**	**36**	**27**	**0.01**	**23**	**0.008**	**18**	**0.008**	**13**	**0.007**
cm162	**58**	**55**	**60**	**0.009**	**43**	**0.008**	**28**	**0.006**	**22**	**0.006**
cm163	**53**	**53**	**61**	**0.006**	**58**	**0.006**	**38**	**0.004**	**-**	**-**
ex1	**485**	**412**	**491**	**0.02**	**436**	**0.016**	**421**	**0.012**	**383**	**0.009**
ex2	**245**	**205**	**92**	**0.01**	**76**	**0.005**	**55**	**0.004**	**-**	**-**
ex3	**105**	**81**	**72**	**0.01**	**55**	**0.004**	**37**	**0.003**	**-**	**-**
ex4	**130**	**101**	**92**	**0.017**	**83**	**0.01**	**42**	**0.0029**	**-**	**-**
ex5	**104**	**82**	**52**	**0.016**	**40**	**0.012**	**26**	**0.008**	**-**	**-**
Cu	**83**	**60**	**26**	**0.01**	**16**	**0.01**	**10**	**-**	**-**	**-**
Total	**25343**	**17644**	**14070**	**0.358**	**12824**	**0.301**	**11441**	**0.203**	**10104**	**0.157**

pre-computed library cells while building the SFG (or assumes that there is no information available about the variable size of pre-computed library cells). Even without considering the size of the pre-computed library cells, the size of the proposed SFG is significantly less than the AIG and BDD. The reduction in the graph size simplifies the mapping problem and decreases the runtime and the required memory size. The graph size and runtime can be further minimized by considering input variable size of the pre-computed library cells. As the value of 'm' increases, then the depth to which the Boolean function to be decomposed will be reduced. This is because the cube cofactor(s) which has 'm' variables after decomposing the function with respect to 'n-m' number of variables (where n represents the number of variables of the function) can be mapped with the m-input pre-computed library cells. The algorithm finds out the common nodes at every level before decomposing it further, which helps in improving the graph building time. The average number of nodes of the proposed SFG is 74% and 150% less than AIG and BDD respectively.

In general the size of the AIG is less than the BDD, but for a few circuits (6 out of 22) BDD has fewer nodes than the AIG. However, for such cases also size of the proposed SFG is smaller than the AIG (except for ex1 circuit), but slightly higher than the BDD. This is due to the less number of non-decomposed nodes in the SFG of the circuits and we did not consider simplification of nodes while calculating the number of nodes.

2.6 SFG for Cut-less Technology Mapping

The unique representation of the nodes of SFG facilitates the cut-less technology mapping. The mapping of the SFG to the library cells starts from the bottom most nodes. The decimal value and level of the bottom nodes of the SFG will be compared with the output decimal value and size of the library cells to find the appropriate match for each node. Once the bottom nodes are mapped with the library cells, then the level of the graph will be reordered to get the actual level of each node. Now, the nodes above the bottom most nodes are selected based on the multiplexers size that are presented in the library. Every node of the SFG will have hidden variable which can be used as selection line of the multiplexer. Therefore, once the bottom most nodes are mapped with the library cells, the remaining nodes are mapped with the appropriate library cells (multiplexers) in a bottom-up fashion. Mapping continues till all nodes in the each primary output are covered. The detailed discussion is postponed to chapter 6.

Chapter 3

Cut-based Technology Mapping

3.1 Cut-computation

Given an AIG, the first step of matching is to enumerate cuts of size k or less, called k-feasible cuts, in G. Intuitively, a k-feasible cut corresponds to a single output subnetwork of G (with k inputs) which may be implemented by a gate or a LUT in the mapped network. For LUT-based FPGA mapping, cut enumeration is the only step for matching: each k-feasible cut can be implemented by a k-LUT. For standard cells, more work is needed to determine if a cut can be implemented by a library gate. This is discussed in Chapter 5. In this chapter, we discuss the complete flow involved in the cut-based technology mapping

3.1.1 k-feasible cut

Let n be a node in an AIG. A feasible cut of a node n in the AIG is a set of nodes {xi} in the transitive fan-in cone of n such that an arbitrary assignment of values to xi completely determines the value of n. A feasible cut is redundant if the value of a node in the cut is completely determined by an assignment of values to the other nodes in the cut. A k-feasible cut is a feasible cut of size at most k that is not redundant. The cut composed of node n alone is always a k-feasible cut of node n (for any k>1) and is called the trivial cut.

Example 3.1: The Fig. 3.1 illustrates the k-feasible cuts of an AIG. A1-A4 are primary input nodes. The 3-feasible cuts for n1 are {n1}, {n2, n3}, {n2, n5, A4}, {n4, n5, n3}, {n4, n5, A4}.

3.2 Technology Mapping [6]

We have used Boolean matching for technology mapping. The input to the mapping procedure is an And-Inverter graph (AIG) [27]. An AIG is a DAG whose nodes represent either AND gates or primary inputs (PIs). Its edges represent wires. Inverters are represented by bubbles on the edges. Given an AIG, the mapping is done in 5 steps.

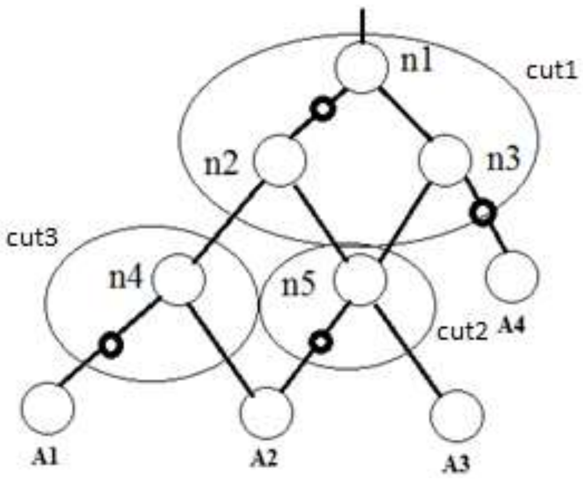


Figure 3.1: Illustration of K-feasible cuts of an AIG graph

Step 1. Compute k-feasible cuts: A feasible cut of a node N in the AIG is a set of nodes {Xi} in the transitive fan-in cone of N such that an arbitrary assignment of values to Xi completely determines the value of N. A feasible cut is redundant if the value of a node in the cut is completely determined by an assignment of values to the other nodes in the cut. A k-feasible cut is a feasible cut of size at most k that is not redundant. The cut {N} is always a k-feasible cut of node N (for any k) and is called the trivial cut. Let P(N) denote the set of k-feasible cuts of node N. If N is a PI, then P(N) = {{N}}. If N is a AND node with children A and B, then

P(N) = {{N}} U {u U v | u E P(A), v E P(B), |u U v | ≤ k}

We compute all 5-feasible cuts of every node in the network by the simple bottom-up traversal based on the above recursion. Although in general a graph may have O(n^5) 5-feasible cuts, we found that most test-cases have between 20 and 30 5-feasible cuts per node. We restrict our attention to 5-feasible cuts since our experiments show that the total number of cuts increases very quickly with k. Pruning techniques have to be applied, and the mapping results are not significantly better (since the pruning is quite arbitrary).

Step 2. Compute truth-tables of cuts: The next step is to compute the local function of a node in terms of its cut. This is done for every non-trivial k-feasible cut of every node in the network. Given a node N, and a cut {Xi} of that node, formal variables are assigned to the each cut node (in no particular order). Using these variables, the functionality of the node is computed symbolically. Since usually only 5-feasible cuts are considered, this symbolic function computation can be performed efficiently using 32-bit integers to represent the truth-tables. In what follows we use the words “function” and “truth-table” interchangeably.

Step 3. Boolean Matching: For each node in the network, for every cut, an appropriate gate (if one exists) is chosen from the library. Each gate thus chosen is called a match for the node. Our matching procedure uses the traditional approach, which is based on NPN class representation (detailed in Chapter 4)

Step 4. Compute best arrival time at each node: Starting from the PIs and working in topological order towards the outputs, the best arrival time is computed for each node from amongst all its matches.

Step 5. Choose the best cover: In the reverse topological order, the best gate for each primary output is chosen. Next, the best gates implementing the inputs of these gates are chosen and so on until all primary inputs have been reached.

3.3 Drawbacks of the Cut-based Technology Mapping

Computation of all cuts is typically a run-time and memory bottleneck [36]. Even though there are efficient algorithms to compute the cuts [36-39], the number of cuts increases exponentially with the cut size (number of leaves) and the number of nodes. So, the time to compute the cuts and finding the best cut among them increases exponentially with the cut size, for example if the size of the cut is k and the number of nodes in the AIG is n, then the number of possible cuts will be $O(n^K)$. In [19], a new technique has been proposed to reduce the number of cuts to be enumerated. However, still the cut computation is required. We will address this problem by proposing a cut-less technology mapping technique in chapter 6.

Chapter 4

Negation-Permutation-Negation (NPN) Classes and Functional Symmetry

4.1 Negation-Permutation-Permutation (NPN) Class [11]

The design of digital circuits involves a deep understanding of the concept of Boolean Functions. There are many operations usually applied in the digital circuit synthesis process. Many of these operations depend on the search space under consideration. This is the case of the matching phase performed during technology mapping [33], where a function (or only part of it) to be implemented is matched against cells from a library. NPN class representation provides a solution for reducing the search space.

For a given number of input variables, say n, there are 2^(2^n) well-defined number of functions. Each n variable function has 2^n possible minterms, resulting in a truth table with 2^n lines. This is shown in Table 4.1 for the case of 2-input Boolean functions. The 2^n possible minterms, or lines of the truth table, give the number of bits in each column of the truth table. This way, the output columns of the truth tables characterize a given Boolean function as a binary number of 2^n bits. As there are 2^(2^n) numbers of 2^n bits, there are 2^(2^n) possible different functions of n inputs. This is also shown in Table 4.1 for the case of 2- input functions.

Table 4.1 All the 16 Different 2-input Functions

AB	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
00	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1
01	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
10	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
11	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Table 4.2 shows the number of n-input functions for n varying from 2 to 4. We can see that for the case of 4-input functions the search space is almost intractable if many operations need to be repeated.

Table 4.2 Possible Number of Functions for 2, 3, 4 and n -Inputs

Number of inputs	**Number of functions**
2	**16**
3	**256**
4	**65536**
n	**2^(2^n)**

The n-input functions can be classified into different classes (set of functions) in order to reduce the search space. The number of functions is the first number to be considered in the classification of the set of n-input functions. Other two numbers that could be used in the classification of n-input functions are the numbers of P and NPN equivalence classes. As it will be discussed later, P classes group functions that are equivalent under input permutation while NPN classes group functions that are equivalent under input negation/permutation as well as output negation.

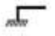
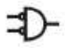
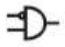
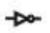
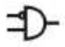
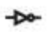
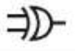
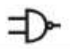
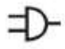
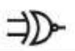
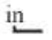
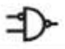
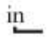
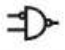
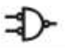
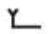
The functional equivalence can be verified using the Binary Decision Diagrams (BDDs) [23] or by manipulating the truth-tables. BDDs are the most used form to represent Boolean functions in Electronic Design Automation. There are many specific kinds of BDDs, depending on its use. Reduced Ordered Binary Decision Diagrams (ROBDDs) are commonly used to compare Boolean functions using the ROBDD strong canonical form (unique representation) presented in [24].

We use the truth-table based verification for checking the functional equivalence. The canonical form of the functions will be computed by manipulating the truth-table by using the Shannon decomposition theorem [21]. This canonical form is necessary to verify the equivalence of the functions and group them into a common equivalence class.

4.1.1 Implementations of n-input functions

Table 4.3 shows the gate implementations of all the 2-input functions.

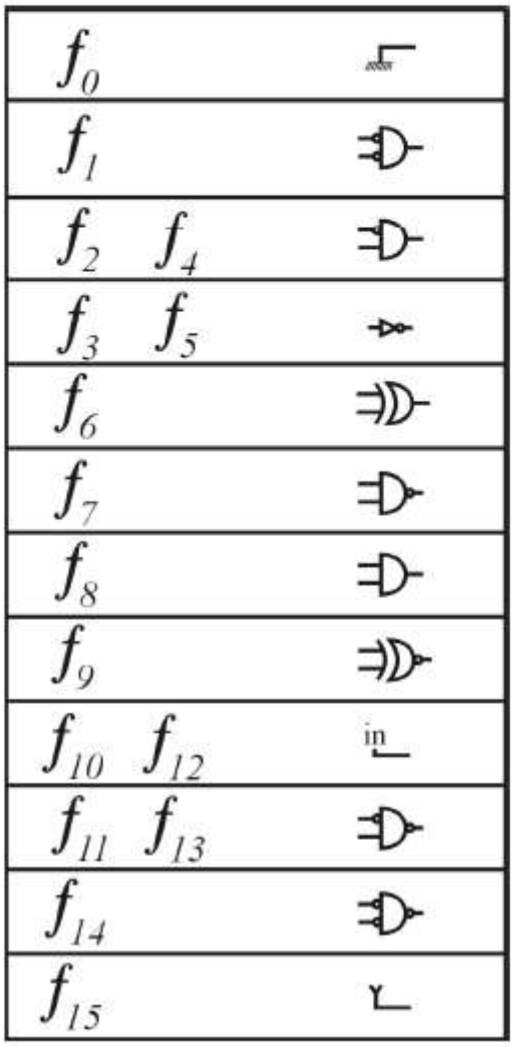
Table 4.3 Implementations for All the 16 Different 2-input Functions

$f_0 = 0$	
$f_1 = \overline{A} . \overline{B}$	
$f_2 = \overline{A} . B$	
$f_3 = \overline{A}$	
$f_4 = A . \overline{B}$	
$f_5 = \overline{B}$	
$f_6 = A \oplus B$	
$f_7 = \overline{A . B}$	
$f_8 = A . B$	
$f_9 = \overline{A \oplus B}$	
$f_{10} = B$	
$f_{11} = \overline{A . \overline{B}}$	
$f_{12} = A$	
$f_{13} = \overline{\overline{A} . B}$	
$f_{14} = \overline{\overline{A} . \overline{B}}$	
$f_{15} = 1$	

4.1.2 Concept of P class

The fact that many different 2-input functions may have the same gate-level implementations naturally introduces the concept of P equivalence. P equivalence between 2 functions is obtained when it is possible to achieve identical values for both truth table outputs by permuting the function inputs. Functions that are P equivalent can be grouped into P classes Table 4.4 shows all the 12 different P classes of 2-input functions. It is important to note that despite the existence of 16 different 2-input functions, there are only 12 different 2-input P classes. The circuits used to implement each P class are also shown in Table 4.4. Four P classes are composed by 2 functions, while eight P classes are composed by only one function. The most important property of P equivalent functions is that they can always be implemented with the same circuit (or cell from a library). Therefore, it is possible to implement any of the 2-input functions with a single cell from a library composed of one gate implementation for each P class.

Table 4.4 The 12 Different 2-input P Classes

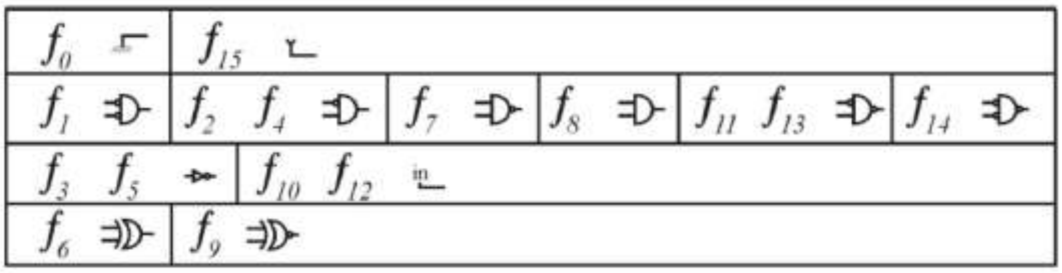


f_0	
f_1	
f_2 f_4	
f_3 f_5	
f_6	
f_7	
f_8	
f_9	
f_{10} f_{12}	in
f_{11} f_{13}	
f_{14}	
f_{15}	

4.1.3 The Concept of NPN class

From Fig. 4.5, it is possible to see that even P classes may have similar implementations. For instance, functions f_1, f_2, f_4, f_7, f_8, f_{11}, f_{13} and f_{14} have gate implementations based on a single nand gate plus some inverters. These functions may be grouped into a NPN equivalence class. NPN equivalence between 2 functions is obtained when it is possible to achieve identical values for both truth table outputs by permutation and/or negation of the function inputs and/or negation of the function output. Figure 4 shows all the 4 different NPN classes of 2-input functions, one NPN class per line. It is important to note that despite the existence of 16 different 2-input functions, there are only 4 different 2-input NPN classes. There are 2 NPN classes composed of 2 functions, one NPN class composed of 4 functions, and one NPN class composed of 8 functions. NPN equivalent functions can be implemented with the same circuit plus some inverters (used in the negation operations for the inputs and the output, if necessary). This way, it is possible to use a smaller library composed of one representative gate for each NPN class plus one inverter cell.

Table 4.5 The 4 Different 2-input NPN Classes



f_0	f_{15}				
f_1	f_2 f_4	f_7	f_8	f_{11} f_{13}	f_{14}
f_3 f_5	f_{10} f_{12}				
f_6	f_9				

This approach is especially useful when the cost of the inverter is very low. The NPN classes are also called semi-canonical representation.

4.1.4 NPN semi-canonical algorithm

The input to our NPN semi-canonical algorithm is truth-table. The number of 1s in the output bit stream of the truth-table is taken into consideration. The output phase is decided in such a way that the expression has lesser number of 1's. In case of a tie, the phase which results a lesser value of the decimal equivalent of the truth-table is considered.

The polarity of each variable is determined by considering the number of 1s in the Shannon cofactors of the function with respect to that variable. The negative cofactor is required to contain fewer 1s than the positive cofactor to keep the phase intact, otherwise the phase will be inverted. In case of a tie, the phase which results a lesser value of the decimal equivalent of the truth-table is considered.

The ordering of variables is determined using the number of 1s in the cofactors with respect to each variable. For this, we require that variables were ordered in the increasing order of the number of 1s in the negative cofactor. If any two variables have same number of 1s in the negative cofactor, then swap is done if swapping results in lower value of decimal equivalent of the truth-table. The pseudo code of the implemented algorithm is presented in algorithm 4.1.

4.2 Canonical Representation-Functional Symmetry

The NPN classes are not completely canonical representations as they can be further reduced. To make the representation canonical we exploit the functional symmetry along with the NPN classes

Algorithm 4.1 Pseudo code for semi-canonizing

```
truth table SemiCanonicize( truth table F, unsigned uCanonPhase, char * pCanonPerm )
{
 // output phase
count the number of 0s and 1s in the truth table of F;
if ( number of 1s is more than number of 0s in F )
 {
complement F;
record negation of the output in uCanonPhase;
}
if(number of 1s is equal to number of 0s in F)
{
F’=complement F;
if(decimal equivalent of F’ < decimal equivalent of F)
{
F=F’;
record negation of the output in uCanonPhase;
} }
// variable phase
count the number of 1s in the cofactors of F w.r.t. each variable;
for each input variable of F
if ( more 1s in negative cofactor than in positive cofactor )
{
change the variable’s phase in F;
record the change of variable’s phase in uCanonPhase;
}
if ( equal 1s in negative cofactor than in positive cofactor )
{
F’=change the variable’s phase in F;
if(decimal equivalent of F’ < decimal equivalent of F)
{
 F=F’;
record the change of variable’s phase in uCanonPhase;
} }
// variable permutation
sort input variables by the number of 1s in their negative cofactors;
permute inputs variables in F accordingly and record the resulting permutation in
pCanonPerm;
return F;
 if(equal 1s in cofactors of two variable)
swap the two variables in F accordingly
if(decimal equivalent of swapped F < decimal equivalent of F)
{
record the resulting permutation in pCanonPerm;
return F;
}}
```

4.2.1 Functional Symmetry along with the proposed modification

Concepts of Boolean matching and symmetry are closely related. Functional symmetries provide significant benefits for multiple tasks in synthesis and verification. For a function f(X) where X = (x1, x2,…,xn), two variables xi and xj are said to be symmetric, denoted as xi ≡ xj, if f(X) is invariant under an exchange of xi and xj. The number of transformations to be carried to convert an n-input variable function to an NPN class function is 2^{n+1}.n!, and the number of symmetric functions is 2^{n+1}. Detection of the symmetric variables of the semi-canonical NPN class functions and modifying their phase make the semi-canonical NPN representations to canonical [28, 34].

The functional symmetry of a Boolean function can be computed with a small modification to the algorithm 4.1. During the variable permutation computation, Variables, which are having equal number of ones in their negative co-factor, are grouped together and this group (argument 'grouped_variblelist' in algorithm 4.2) will be given as input to the algorithm 4.2, which finds symmetric variable to the first order and their optimal phase as follows. Based on the grouped variable list, variables are swapped with another variable, which is having the same number of 1's in the negative cofactor. If the truth-table value is same then these variables are placed in the symmetric group. To uniquely determine the phase of the variables in the symmetric group, we have to consider the eight different configurations (ab, a'b, ab', a'b', ba, b'a, ba', b'a') for each symmetric group, instead of only two (ab,ba) as given in [33]. But we have observed that for a symmetric group only four configuration (ba, b'a, ba', b'a') are sufficient to determine the phase of the variables. We considered the configuration which will give the smallest possible value in its truth-table.

Algorithm 4.2 determining the symmetric variables and their phase

```
find_symmetry (semicanonizedTT F, grouped_variablelist)
{
int sym_out;
for each group in the list
{
swap the position of the variables, save in F1;
if (F=F1)
{
then apply all four configurations (ba,b'a,ba',b'a') and consider the configuration with smallest F output value
}
Consider the pair with smallest value among;
save in sym_out;
Change F accordingly;
}}
```

Table 4.6 Comparison of Number of Functions for different classes

No. of inputs	**No. of functions**	**No. of NPN classes**	**NPN class + symmetry**
2	**16**	**4**	**-**
3	**256**	**14**	**-**
4	**65536**	**1031**	**222**
6	*****	**34225**	**2103**
8	*****	**271646**	**13932**

*****a very big number

Table 4.6 shows the reduction in the number of functions when they are classified as NPN classes and NPN classes along with Functional symmetry. The advantage of the aforementioned classifying will be clear in chapter 5.

Chapter 5

Proposed Cut-based Digital IC Design and Automation Methodology

In proposed cut-based design and automation methodology, shown in Fig. 5.1, the input truth-table/ Boolean expression is mapped to its physical design circuits through canonization and Boolean matching (based on input size of the function, truth-table to graph conversion, k-feasible cuts enumeration, and graph cuts to truth-table conversion are done). We consider only pure combinational networks. Sequential networks are handled as combinational networks by cutting at the register boundary, so the final network is a pure combinational circuits and registers. Here we are not considering the mapping of registers to their physical designs, because direct mapping will not reduce states, so area and timing can't be optimized.

5.1 Pre-Computed Library

To demonstrate the advantage of the functional symmetry, we maintained two pre-computed libraries one with semi-canonical forms of all the n-input Boolean functions computed using the NPN classes [12] and another one with NPN along with the functional symmetry classes [35]. Then these functions will be optimized, placed and routed using the existing CAD tools and stored in the precomputed library along with their output decimal value. Functional symmetry reduced the library size and the number of worst case comparisons required to find a match drastically. For an example, 1031 circuits and comparisons can be reduced to 222. Using the proposed methodology, pre-stored circuits can be used to map any combinational circuit without any limitation on the number of input variables and unlike [10], there is no need to update the library.

The library poses a limitation to the number of circuits that can be matched, i.e. the cases when the input expression is not in any of the Library circuit's NPN class. For such cases, an alternate procedure is suggested, which is not a method of reloading the library again. The Library is developed in such a way that for some number say 'm', if the number of input variables is less than or equal to m, then the library can match the input expression. That is, all representatives with number of variables less than or equal to ‘m’ are stored in the library.

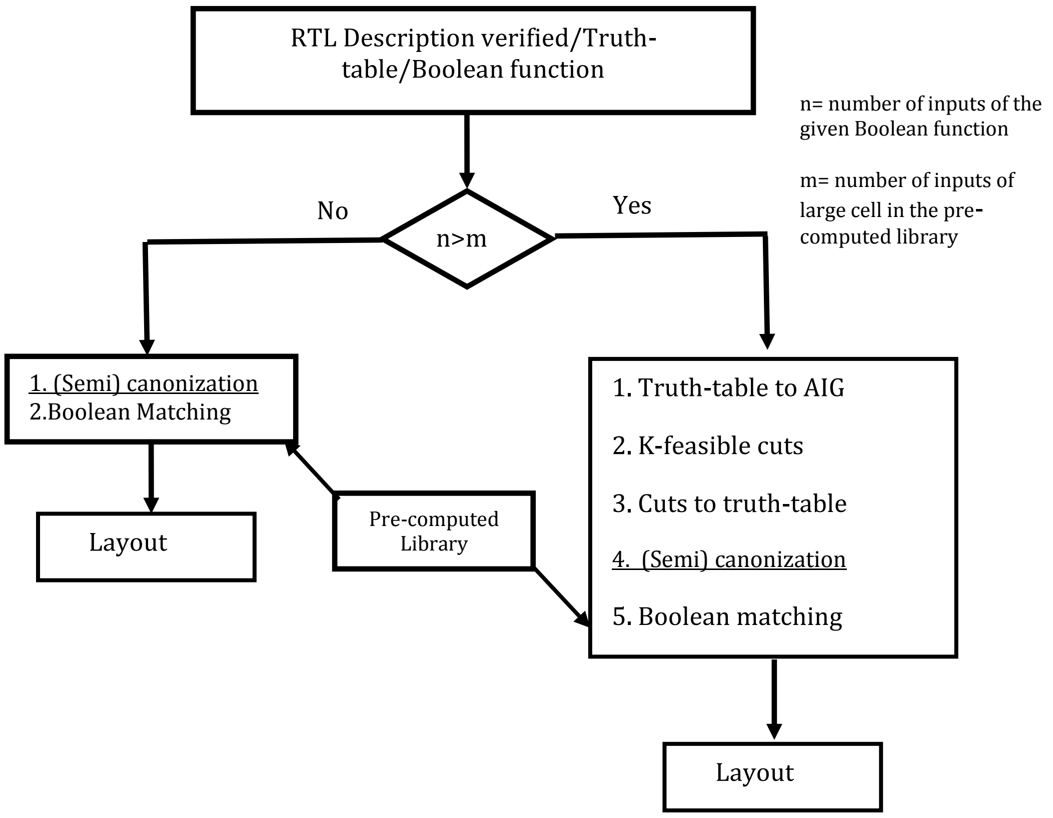


Figure 5.1: Detailed design flow of the proposed cut-based design and automation methodology. The underlined text represents two different representations (semi-canonical and canonical)

5.2 Boolean Matching

Boolean matching is a powerful technique that has been used in technology mapping to overcome the limitations of structural pattern matching [28]. Let g be the Boolean function computed by some element of a technology library. Let f be another Boolean function, called the target function. Boolean matching is the problem of determining whether g can be used to implement f through negation/permutation of the inputs and negation of output of g. Concepts of Boolean matching and symmetry are closely related. Since the functional symmetry was already discussed in chapter 4, we will not present the algorithm for Boolean matching in this chapter.

5.3 Mapping Without Considering the Functional Symmetry

The basic design steps in the proposed design & automation methodology [12] is as follows

1. If the input Boolean function has same or less number of inputs than the pre-computed library, then

   a. Algorithm 5.1 generates NPN semi-canonical form from the input Boolean expression/truth-table
   b. Boolean matching algorithm maps it to the pre-computed library cell. In this case, there is need of doing placement and routing as the cells are pre-placed and routed.

2. Otherwise,

   a. Input truth-table/Boolean expression is converted to AIG graph
   b. K-feasible cuts are enumerated for each node in the AIG
   c. K-feasible cuts are converted to truth table
   d. Each cut is matched to the pre-computed library cells.

In this case there is need of doing the placement and routing of the mapped cells.

Each step in the proposed methodology is explained below.

Case 1*: Input Boolean function/truth-table with less than or equal to number of input variables of pre-computed library cells*

The input truth-table/Boolean expression is converted to its corresponding NPN semi-canonical form using the algorithm 4.1 (refer to chapter 4 for details). The NPN semi-canonical form of the input is matched to the corresponding cell in the pre-computed library and the design is then modified by adding appropriate inverters to generate the complete design of the input Boolean expression.

Case 2*: Input Boolean function/truth-table with more number of input variables than pre-computed library cells*

In this case the input expression cannot be covered directly by the circuits in the library. Rather a combination of circuits from the library will be able to generate the circuit diagram for the input expression. The input needs to be broken into smaller components which the library can match. We have used a graph based approach for that. Following steps explains the case 2.

Step1**:** Given the truth-table/Boolean expression, construct an And-Invert-Graph. There are many ways to construct an AIG from a truth-table/Boolean expression and we have used co-factors based. The AIG is constructed recursively as we go from the first input variable to the last with Shannon co-factors as shown in algorithm 5.1. This algorithm reduces the size of the graph on-the-fly for better results and reducing the required memory size, by handling the nodes which have function “always true (1)” or “always false (0)”. By using Boolean properties, these nodes can be removed to give a more compact graph.

Step2**:** After the graph is constructed, it is portioned into k-exhaustive cuts, where the value of k is ‘L’ defined in Library Creation. The graph is cut using a graph-k-cut algorithm [8, 9], pseudo code for which is mentioned in algorithm 5.2.Using these cuts, the function AIG graph is broken into smaller sub-graphs with maximum ‘k’ inputs which are individually covered by the library elements. Out of all the possible cuts for a node, the best cut is chosen heuristically. The cut is chosen in such a way that it is as much closer to the inputs as possible and contains most number of common nodes with already calculated nodes.

Algorithm 5.1 Pseudo code for an AIG-graph construction

```
Aigmakegraph( truth table F,  it input_variables)
{
//Calculate the positive cofactor (F1) and negative cofactor (F2) of F with respect to each
//input_variable.
aig1=makegraph(F1, next input variable);
aig2= makegraph(F2, next input variable);
aig= input_variable.F1 + input_variable'.F0;
}
```

***Step3*:** All the sub-graphs generated by the previous step are converted into individual truth-tables. Pseudo code for converting cut to truth-table is presented in algorithm 5.3.

Algorithm 5.2 Pseudo code for k-feasible cuts

```
Void NetworkKFeasibleCuts( Graph g, int k )
 {
for each primary output node n of g
NodeKFeasibleCuts( n, k )
}
Cutset NodeKFeasibleCuts( Node n, int k )
{
if ( n is primary input )
return { { n } }
if ( n is visited )
return NodeReadCutSet( n ) ;
//mark n as visited
cutset Set1 = NodeKFeasibleCuts( NodeReadChild1( n ), k);
cutset Set2 = NodeKFeasibleCuts( NodeReadChild2( n ), k) ;
cutset Result = MergeCutSets( Set1, Set2, k ) ∪ { n } ;
NodeWriteCutSet( n, Result );
return Result;
}
cutsetMergeCutSets ( cutset Set1, cutset Set2, int k )
 {
cutset Result = { }
for each cut Cut1 in Set1
for each cut Cut2 in Set2
if ( | Cut1 ∪ Cut2 | ≤ k ) then Result = Result∪ { Cut1∪ Cut2 }
```

Step4**:** All the individual truth-tables are then converted into corresponding circuit designs, by mapping to the pre-computed library cells (Case 1 mentioned above). These designs are then routed together to generate the complete circuit design for the input expression.

Algorithm 5.3 Pseudo code for graph to truth-table conversion

```
truthkcuttable(aig_graph G,cut C, node n)
//enumerate all the interconnecting nodes between node //‘n’ and cut C
Nodeset={n} U enumerate(n)
//generate the entire truth-table of node ‘n’ by calculating //the truth-table of all nodes in
“Nodeset” by
//taking the nodes in cut ‘C’ as primary inputs
set enumerate(node m)
{
 if(m is not in cut ‘C’)
Nodest=enumerate(m-child1);
Nodest=enumerate(m-child2);
}
```

5.4 Results and Discussion

To validate the effectiveness and advantages of our proposed methodology, number of experiments has been carried out using the benchmark circuits from revlib.org [30]. The methodology is implemented in MATLAB running on a Xeon processor (3.4GHZ, 4GB RAM) operating in Linux-based environment. Table 5.1 shows the runtime for different functions with different number of inputs, where column 2, column 3, column 4 and column 5 represent the number of inputs of the input Boolean function, the name of standard benchmark circuit, runtime for deriving physical design of the input function and represents number of circuits taken from the pre-computed technology library for matching the input Boolean function respectively.

As can be seen, the input Boolean equations are matched directly to the pre-computed physical library circuits. The proposed algorithm for converting truth table to AIG graph reduces the graph size on-the-fly by finding the nodes with constant ones or zeros. Due to limitations of used software platform, existing CAD tool are proprietary, encrypted, while presenting our preliminary research results we are unable to compare proposed methodology with that obtained from proprietary CAD tools quantitatively. However we provide the qualitative analysis of the proposed methodology in comparison with conventional digital IC design and automation methodology below. For conventional digital IC design and automation methodology, the runtime can be represented as,

$$Trt = Tbe + Tfe$$

$$Tfe = Trt + To + Titm + Ttd$$

$$Tbe = Tfp + Tpl + Tr$$

$$Tr = Tintra + Tinter$$

Where, Tbe, Tfe, To, Tfp, Tpl , Tr, Tintra, Tinter, Titm, Ttd are runtimes for backend design, frontend design, optimization, floorplanning, placement, routing, intra block routing, inter block routing, technology independent and technology dependent mapping.

Table 5.1 Runtime for Different Functions with Different Number of Inputs

S. No	No. of inputs	Circuit name	Runtime	No .of functions used from library
1	4	4gt5	0.028	1
2	4	4gt4	0.035	1
3	4	4gt13	0.017	1
4	4	4gt12	0.016	1
5	4	4gt11	0.016	1
6	4	4gt10	0.015	1
7	4	4mod5	0.017	1
8	4	sf	0.046	1
9	5	alu	0.212	4
10	5	majority	0.261	4
11	5	xor5	0.18	2
12	5	ex3	0.253	4
13	5	ex2	0.255	4
14	5	ex1	0.258	3
15	5	2of5	0.273	4
16	5	5mod5	0.285	4
17	6	sym6	0.773	11
18	7	7bitevenparity	2.762	2
19	6	mux4	0.504	4
20	8	8mod5	17.27	37
21	9	max4	156.5	69

Whereas for the proposed methodology (Fig. 5.1) the run time can be computed for two different cases as follows, If the input Boolean expression has number of inputs less than or equal to pre-computed library functions inputs, the runtime is given by

$$Trt = Tm$$

Where, Tm is the runtime for finding a match. On the other hand, if the input Boolean expression has number of inputs more than or equal to pre-computed library functions input, the runtime can be computed as

$$Trt = Tm + Tfp + Tpl + Tinter$$

Where Tm, Tfp, Tpl and Tinter are same as defined before.

5.5 Mapping with Functional Symmetry

The NPN class representative of an n- input Boolean function/truth-table is semi-canonical, as it has possibility to apply to apply some other transformations. The number of transformations to be carried to convert an n-input variable function to an NPN class function is $2^{n+1}.n!$, and the number of symmetric functions is 2^{n+1}. Detection of the symmetric variables of the semi-canonical NPN class functions and modifying their phase make the semi-canonical NPN representations to canonical [28, 34]. The canonical representation of any Boolean function can be computed using the algorithms 4. 1 & 4.2 (see chapter 4 for details). The advantages of the application of functional symmetry in the proposed methodology are demonstrated in the following experimental reseuls.

5.6 Results and Discussion

The algorithm was implemented in MATLAB running on a Xeon processor (3.4GHZ, 4GB RAM) operating in Linux-based environment, then integrated with the aforementioned algorithms to demonstrate the advantages of the functional symmetry. Table 5.2 (column 1 to 3) shows the average runtime (all in seconds) to compute the canonical form and finding its equivalent circuit from the library (column 2) and number of circuits required to map the each input function (column 3) with and without considering the functional symmetry. It shows that the run time and number of circuits required to map, increase with the number of variables of the input function. This is because of the constraint imposed by the library size. Since the considered input variables' size of the pre-computed library circuits is four, all 4-input functions can be mapped to their corresponding circuits directly. But as the input size increases, it has to construct the graph and enumerate the cuts, and then map the each cut, which will increase the runtime. Table 5.2 (column 4 and 5) shows the effect of functional symmetry on the library size for different variable sizes and runtime. Due to the symmetry the number of distinctive NPN equivalent functions to be stored in the pre-computed library will be less. As a result of small library size, the memory size will be reduced, and the number of comparisons required to find the match will also be cut down. It reduced the number of required pre-computed circuits in our experiments from 1031 to 222 (464.4% reduction in the required memory size for pre-computed library). Even though the symmetry is computationally expensive, the runtime will not be affected shown in column 5, where the ratio of runtimes of 'without symmetry detection' and 'with symmetry' is almost unity. This is because of the time saved in finding a match has been spent in computing the symmetric variables' phase.

Table 5.2 Average Runtime Taken to Find a Match Without Symmetry and With Symmetry and Effect of Symmetry on Library Size and Runtime

No. of inputs(no. of experiments)	**Average runtime: with symmetry(w/o symmetry)**	**No. of pre-computed circuits required**	**Library size**		**Runtime ratio ***
			With symmetry	**W/o symmetry**	
4 (30)	**0.04 (0.039)**	**1**	**222**	**1031**	**0.999**
5 (30)	**0.24 (0.239)**	**2-4**	**-**	**-**	**0.996**
6 (16)	**0.71 (0.708)**	**4-11**	**2103**	**34225**	**0.998**
7 (16)	**2.34 (2.32)**	**5-13**	**-**	**-**	**0.995**
8 (17)	**16.3 (16.25)**	**10-40**	**13932**	**271646**	**0.997**
9 (15)	**150 (149.2)**	**50-74**	**-**	**-**	**0.995**

*runtime= Time to find a match when symmetry is not considered/Time taken to find a match when symmetry is considered

A comparative study of conventional and proposed IC designs on NRE costs contributors is presented in Table 5.3. It shows that the backend design is needed for the functions with more number of variables than the precomputed library functions. To check the proposed methodology performance in terms of average runtime we have taken random functions with different inputs. As we can see from Fig. 5. 2 (a) and (b), when the number of inputs increases average runtime also increases. For an example, when the number of inputs increases from 5 to 6, the runtime increases by 158%. This is because, the number of circuits required for mapping the input functions increases with the number of inputs. However, increased runtime for higher number of input functions can be overcome by increasing the library size. The correctness of the final design has been verified by comparing the truth-tables of the RTL description and the corresponding final layout

Table 5.3 Comparisons of NRE Cost Contributors between the Conventional and Proposed Design Automation Methodology

Methodology	**Frontend RTL simulation**	**Logic Synthesis**	**Backend design**
Conventional	**Yes**	**Yes**	**Yes**
Proposed	**Yes**	**No**	**No (for *)/Yes (for **)**

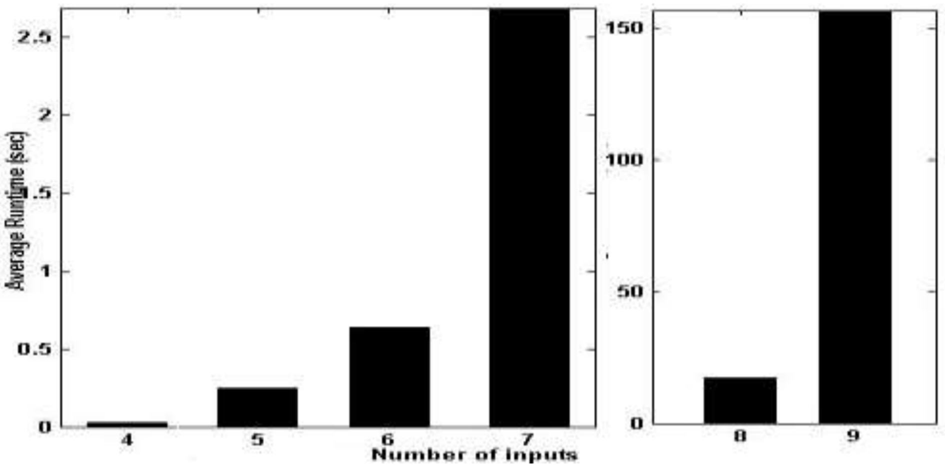


Figure. 5.2: (a) and (b) show the variation of average runtime with number of input variables of 4-7 and 8-9 for different functions respectively.

The inherent advantage of the symmetry is that the number of distinctive circuits required to map the input function will be reduced. There by it will maintain the regularity in the design, which will help the engineering change order (ECO) to get the better quality of results. Proposed symmetry detection algorithm finds the probability that the function has symmetric variables or not beforehand based on the number of 1's in their cofactors, thereby it reduces complexity and runtime. Table 5.4 shows the advantages of the proposed methodology over the conventional methodology in terms of NRE costs, time-to-market, regularity and routing congestion. It maintains the regularity independent of the design strategy whereas in the conventional methodology, it depends on the design strategy.

Table 5.4 Advantages of the Proposed Methodology

Methodology	**Tool cost as a part of NRE costs**	**TTM**	**Regularity in the design**	**Intra-block routing congestion Probability**
Conventional	**High**	**High**	**Designer dependent**	**High**
Proposed	**less**	**less**	**Inherent property**	**less**

Due to the small design cycle (mapping directly to physical designs) and elimination of frontend design tool, the TTM and has to be done to connect different blocks (cuts). The following theoretical analysis explains the advantage of the proposed methodology in reducing the routing congestion.

Assuming the NRE costs will be cut down drastically. Once the input function is mapped to its physical design(s), only the inter-block routing intra-block routing congestion probability is Pinbr and inter-block routing congestion is Pibr,

Then, in the conventional methodology:

Total congestion probability = Pinbr + Pibr

Where Pinbr and Pibr include both global and detailed routing congestion probability.

Similarly, in the proposed methodology:

Total congestion probability = Pibr (∵Pinbr = 0)

Since the each block is already placed and routed, the probability of intra-block routing congestions is reduced (~0) and it in turn reduces the number of iterative cycles required to get proper placement and routing.

Chapter 6

Proposed Cut-less Digital IC Design and Automation methodology

The methodologies proposed in [12-15], to minimize the NRE costs and TTM, have been proved as a potential candidates for minimizing the number of incremental and iterative design steps. The aforementioned methodologies are cut-based methodologies, which require cut-enumeration, storing of cuts, computation of truth-tables and canonical form for mapping. The cut enumeration is computationally complex and requires more memory for storing the cuts. If there are n number of nodes in the subject graph and k is the cut size, then the possible number of cuts is O(n^k), which is exponentially growing with the cut size and number of nodes. Due to this exponential complexity, the runtime and required memory grows exponentially and finally affects the overall design cycle. Therefore, as a solution here we propose a cut-less design & automation methodology.

*Lemma1: Considering the maximum input size of the cells in the pre-computed library is 'L' and the input Boolean function has 'M' (M>2L) variables, the minimum and maximum number of primary inputs that a node receives in the SFG will be L (bottom nodes) and M-L+1 (primary output node) respectiv*ely.
Proof: Assuming that there are all L-input cells in the pre-computed library. Using the Boolean matching techniques these gates can be used to map all input Boolean functions, whose size is not more than L. Therefore, the SFG constructed to a level, whose nodes at the bottom will have at most L inputs, is enough. Initially the primary output node will have M-inputs, but after decomposing it to a level L, its level becomes M-L+1. The bottom nodes can be mapped directly to their pre-computed library cells by using their truth-table or decimal value. In this way, the size of the SFG and building time can be minimized by maintaining a set of large input size cells in the pre-computed library.

6.1 Cut-less Technology Mapping

Input to the proposed cut-less technology mapping is SFG. Here we consider a pre-computed library which is having at most 3-input cells, so the SFG is constructed to a level where the bottom nodes will at most receive three primary inputs (see lemma 1). First, the bottom nodes are mapped using the 3- input cells. The correct match for each bottom node is found by comparing the decimal value/Boolean expression of the cube cofactor and its level with the pre-computed library cells. Then the nodes above the bottom nodes will be selected for mapping. The topology of SFG, which has inherent regularity in its structure, helps in cut-less technology mapping. This feature of SFG facilitates the mapping of a node to its technology dependent cells without computing the node’s local functionality. Therefore, IDs of the nodes are deleted after SFG is constructed to free up the memory used for storing IDs.

Multiplexers are used for mapping the nodes other than the bottom nodes. The multiplexers were optimized against the area by considering the nature of the nodes [13]. The multiplexer cells and their size will be selected from the pre-computed library depending upon the number of levels in SFG. The size of Multiplexer is used for selecting the nodes of SFG for mapping. For an example, if the size of multiplexer is 4X1, then a node which is at two levels above than previously mapped node is selected for mapping. The process of selection of nodes and mapping them to the pre-computed library cells continues till all the nodes in the SFG are covered.

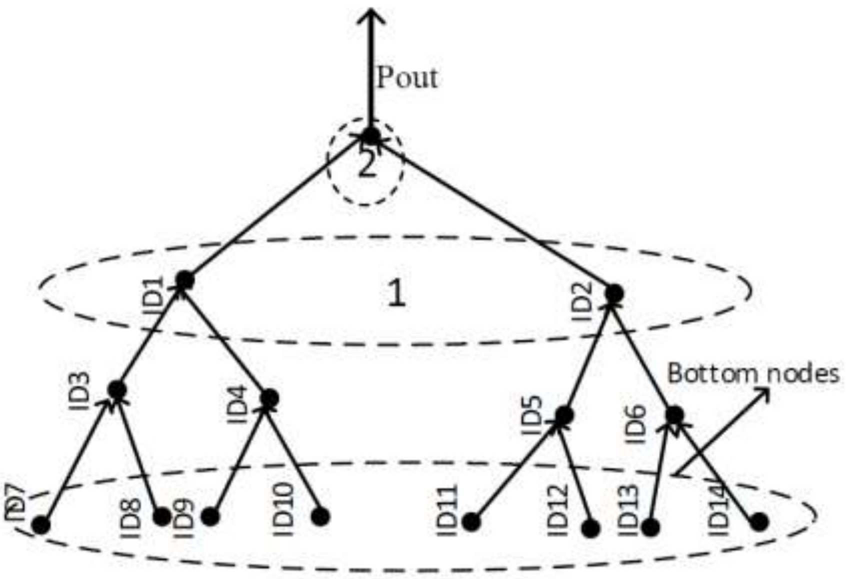


Figure 6.1: Illustration of proposed cut-less technology mapping technique.

The Fig. 6.1 describes the mapping of SFG of a 6-input Boolean function. Here we considered pre-computed library with all 3-input cells and multiplexers of size 4X1 and 2X1. Since the pre-computed library cells can be used to map the nodes, which receive 3 primary inputs, the 6-input function is decomposed till the bottom most nodes receive three primary inputs. Once the bottom nodes are mapped, nodes having a level 2 higher than the bottom nodes are selected for mapping. The selection lines for multiplexer are the hidden variables of the nodes. In Fig. 1, nodes labeled with 1 are selected and mapped with the 4x1 Multiplexers. The primary output, labeled 2, is mapped with the 2x1 multiplexer.

6.2 Modified Digital IC Design and Automation Methodology using Cut-less Technology Mapping

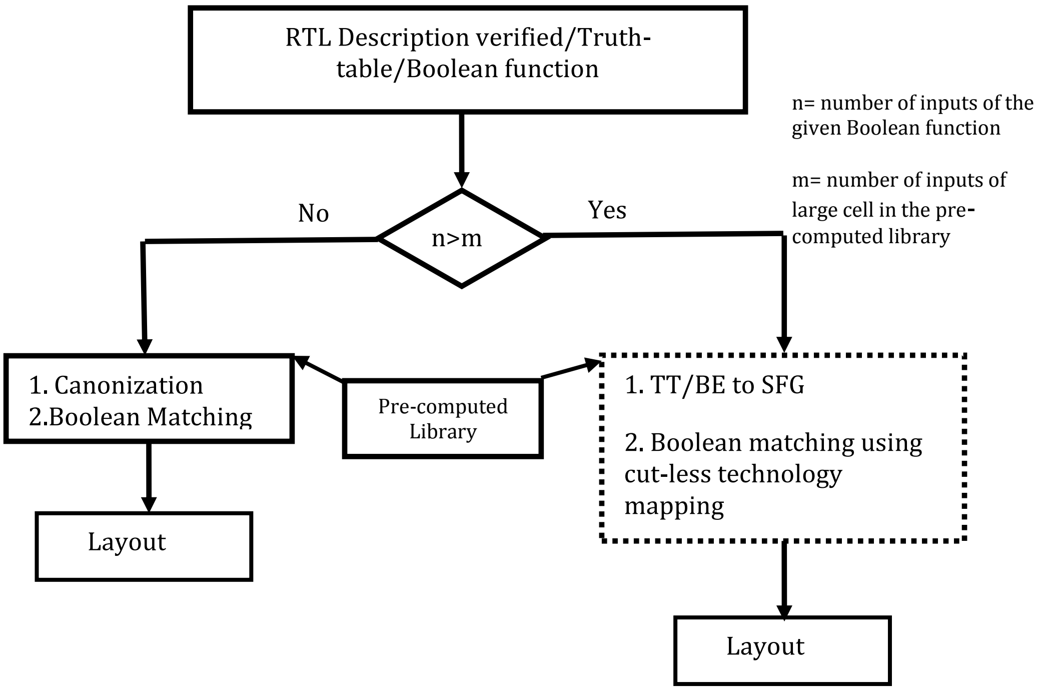

RTL Description verified/Truth-table/Boolean function
n= number of inputs of the given Boolean function
m= number of inputs of large cell in the pre-computed library
No
n>m
Yes
1. Canonization
2.Boolean Matching
Pre-computed Library
1. TT/BE to SFG
2. Boolean matching using cut-less technology mapping
Layout
Layout

Figure 6.2: Modified design and automation methodology. The dotted box represents the proposed modification.

The cut-less design and automation methodology is shown in Fig. 6.2. It has a few steps than the cut-based design and automation methodology proposed in chapter 5. The algorithms used in the aforementioned modification are already explained in the previous sections (see chapter 6.1 & 3 for details).

6.3 Results and Discussion

The proposed cut-less technology mapping technique is implemented in MATLAB running on a Xeon processor (3.4GHZ, 4GB RAM) operating in Linux-based environment. We validated our algorithms extensively with standard benchmark circuits [29, 30]. The circuits given in the Table 6.1 are taken from the revlib [30], which are in PLA format, and they are converted into truth-tables using the SimpleSolver [31]. The circuits given in the Table 6.2 are taken from [29]. First, the Boolean equations of the benchmark circuits given in Table 6.2 are computed using the ABC tool [32], then the truth-tables of each Boolean equations are harvested using a simple program. The combinatorial part of the sequential circuits is harvested by cutting at the register edges.

The conventional technology mapping that was used in [12-15], requires the computation of cuts for all nodes, pruning the cuts, computing the local function of each cut in terms of formal variables assigned to its fan-in cone nodes, converting the local function into canonical form and finally finding a match for each cut using its canonical representation. Whereas the proposed cut-less technology mapping technique directly maps the SFG nodes to pre-computed library cells by using the node IDs and exploiting inherent regularity in the structure. The theoretical analysis given below will explain the advantage of the proposed cut-less technology mapping technique compared to the conventional cut-based technology mapping technique used in [12-15].

Assume that the number of nodes in the graph is n, cut size is k, the pruning is tp, the time required for computing the local function of each node in terms of its cut is tf , the time required for finding the match for each node is tm and time for computing the canonical form is tc. The time required for mapping using the conventional cut-based technology mapping TCon and for the proposed cut-less technology mapping TPro can be given as,

$$\text{TCon} = O(n^K) + tp + tf + tm + tc \qquad (1)$$

$$\text{Pro} = tm \qquad (2)$$

$$\text{Amount of time saved } (t_S) = O(n^K) + tp + tf + tc \qquad (3)$$

Theoretically, it is clear from the equation (3) that the proposed cut-less technology mapping technique will improve the runtime significantly. The experimental results given in Table 6.1 and Table 6.2 will substantiate the aforementioned theoretical analysis.

All the runtime values given in the Table 6.1 and Table 6.2 are in seconds. Table I compares the proposed cut-less technology mapping technique with the cut-based methodology used in [10-13] in terms of runtime. Column 2 gives the runtime taken to map the input Boolean function to the pre-computed library cells using the cut-based technology mapping technique. The runtime includes time taken for building the AIG graph, cut-enumeration, calculating the local function of cut, converting the local function into canonical form and finding the match. Column 3 describes the runtime taken to map the input Boolean function to the pre-computed library cells using the proposed cut-less technology mapping technique. Column 4 represents the runtime ratio of cut-based to proposed cut-less technology mapping technique. The proposed mapping technique is at an average ~62X faster than the previous techniques [12-15].

Table 6.1 Runtime Comparison of Cut-based Technology Mapping Technique used in [12-15] with the Proposed Cut-less technology Mapping Technique for Pre-computed Library with all 3-input cell and 4X1 Multiplexers

Circuit name	**Cut-based(r1)**	**Cut-less(r2)**	**Runtime Ratio(r1/r2)**
gt5	**0.028**	**0.0039**	**7.17**
gt4	**0.035**	**0.004**	**8.75**
gt13	**0.017**	**0.0041**	**4.146**
gt12	**0.016**	**0.004**	**4**
gt11	**0.016**	**0.0039**	**4.1**
gt10	**0.015**	**0.004**	**3.75**
mod5	**0.017**	**0.004**	**4.25**
alu	**0.046**	**0.005**	**9.2**
majority	**0.212**	**0.006**	**35.3**
xor5	**0.261**	**0.006**	**43.5**
ex3	**0.18**	**0.004**	**45**
ex2	**0.253**	**0.0062**	**40.8**
ex1	**0.258**	**0.006**	**43**
2of5	**0.273**	**0.005**	**54.6**
mod5	**0.285**	**0.0061**	**46.7**
sym6	**0.773**	**0.01**	**77.3**
7-bitevenparity	**2.762**	**0.08**	**34.5**
mux4	**0.504**	**0.0097**	**51.9**
Average	**5.9515**	**0.0955**	**(5.9515/0.0955)=62.3**

Table 6.2 describes the performance of the proposed cut-less technology mapping technology on standard benchmark circuits, variation of runtime with multiplexers' size and reduction in memory size compared to the cut-based methodology [12-15]. Column 2, 3 and 4 represent the time taken to map the input Boolean function when the size of the multiplexer is 2x1, 4x1 and 6x1. As the size of the multiplexer cell increases, more number of nodes can be covered and cells in the pre-computed library, we can reduce the runtime significantly. The other advantage of the proposed cut-less technology mapping technique is

reduction in memory size (Column 5). The average reduction in the required memory size is ~400 due to the elimination of cut-enumeration and storing of cuts.

Table 6.2 Validation of the Proposed Cut-less Technology Mapping Technique Using Standard Benchmark Circuits, Variation of Runtime with the Size of the Multiplexer and Reduction in the Memory size Compared Cut-based technology mapping

Circuit name	2x1 Mux	4x1 Mux	6x1 Mux	Reduction in memory size
cm138	**0.04**	**0.034**	**0.03**	**647**
cmb	**12.64**	**11.12**	**10.2**	**253**
cm163a	**15.94**	**13.43**	**12.01**	**146**
cm162	**3.86**	**3.04**	**2.84**	**134**
cm162a	**0.19**	**0.13**	**0.089**	**85**
alu2	**0.33**	**0.3**	**0.24**	**475**
cm151a	**0.41**	**0.37**	**0.31**	**769**
ex4	**0.37**	**0.29**	**0.26**	**848**
ex1	**15.36**	**13.03**	**11.32**	**349**
Total	**49.2**	**41.8**	**37.3**	

Chapter 7

Conclusion and Future Scope of Work

7.1 Conclusion

A novel and unified digital IC design and automation methodology [12-15] is proposed. Our proposed methodology is compact and simplified compared to the conventional digital IC design and automation methodology. Logic synthesis step, which is a major task in IC design, is eliminated by merging logic synthesis step with backend design step. Therefore, the input RTL description is directly mapped to its physical design by retrieving the cells from the pre-computed library, which contains the already placed and routed circuits. Due to elimination of design steps with CAD tools, our methodology shows significantly reduced NRE costs. Moreover, due to shortened design time, the time-to-market is drastically reduced. We have also exploited the functional symmetry of the Boolean functions which has major impact on the library size, the number of comparisons and it helps the ECO by maintain the regularity in the design.

The proposed cut-less technology mapping technique benefits the digital IC design and automation methodology proposed in [12-15] in terms of runtime and memory, which improve the TTM and the NRE costs. Since the computational complexity of the proposed cut-less technology mapping technique is significantly less compared to the cut-based technology mapping techniques, inherently the scalability will be improved, which will help in handling the today's highly complex designs. The modified SFG construction algorithm can take input as truth-table (for faster computation of Shannon cube cofactors for smaller function) or Boolean expression (to handle large functions), depending upon the number of variables. We validated our methodology using a number of different benchmark

7.2 Future Scope of Work.

We presented here the preliminary research results. Creating a pre-computed library with placed and routed circuits, optimization and validation using larger designs form part of the future research.

Besides this, area and delay efficient mapping techniques, and scalable SFG data structure can be considered for the future research to improve the results further.

References

[1] Robert A. Walker and Donald E. Thomas “A Model of Design Representation and Synthesis”. IEEE DAC’1985, pp.453-459.

[2] International Technology Roadmap for Semiconductors, http://public.itrs.net

[3] Aniel D. Gajski, University of Illinois Robert H. Kuhn, “Guest editors’ introduction to New VLSI tools”, Gould Research Center,IEEE computer’1983, vol.10,pp.11-14.

[4] Jie-Hong Roland Jiang, SrinivasDevadas, “Logic Synthesis in a Nutshell”, November 2009.

[5] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, “Multilevel logic synthesis”, Proc. IEEE, Vol. 78, Feb.1990

[6] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, “Reducing structural bias in technology mapping”, IEEE TCAD’06, Vol. 25(12), pp. 2894-2903.

[7] L. Wang, and A.E.A. Almaini, “Multilevel logic simplification based on containment recursive paradigm”, IEE Proceedings Computers and Digital Techniques, 2003, Vol.150, No.4, pp, 218-226.

[8] Wenlong Yang, Lingli Wang, Alan Mishchenko “Lazy Man’s Logic Synthesis”, IEEE/ACM ICCAD’2012, pp.597-604.

[9] A. Mishchenko, S. Chatterjee, R. Brayton, X. Wang, and T. Kam, “Technology mapping with Boolean matching, supergates and choices”, ERLTechnical Reports, EECS Dept.,UC Berkely, EECS Dept., UC Berkeley, March 2005.

[10] Wenlong Yang, Lingli Wang, Alan Mishchenko “Lazy Man’s Logic Synthesis”, IEEE/ACM ICCAD’2012, pp.597-604.

[11] V.P. Correia, A. Reis, “Classifying n-Input Boolean Functions”, in Proc. IWS 2001.

[12] B. Karunakar Reddy, S. Sabbavarapu, K. Gupta, R. Prabhat, A. Acharyya, R. A. Shafik and J. Mathew, “A Novel and Unified Digital IC Design and Automation Methodology with Reduced NRE Cost and Time-to-Market”, IEEE International Symposium on Electronic System Design, Singapore, 12-13 December, 2013,pp. 36-40

[13] S. Sabbavarapu, B. Karunakar Reddy, R. Prabhat, K. Gupta, A. Acharyya, R. A. Shafik and J. Mathew, “A Novel Physical Synthesis Methodology in the VLSI Design Automation by Introducing Dynamic Library Concept”, IEEE International Symposium on Electronic System Design, Singapore, 12-13 December, 2013, pp. 103-107.

[14] S. Sabbavarapu, B. Karunakar Reddy, Srinivasulu. N, Amit Acharyya, Jimson Mathew, “A novel IC design methodology using dynamic library concept with reduced NRE cost and time-to-market”, Journal of Low Power Electronics. 10, 429-442, 2014.

[15] B. Karunakar Reddy, S. Sabbavarapu, Amit Acharyya, “Effect of Constant One and Zero, Shared and Non-decomposed Nodes on Runtime and Graph Size of the Shannon Factor Graph (SFG)”, IEEE International Symposium on Electronic System Design, NITK, 2014, pp. 135-139.

[16] Andreas Kuehlmann and Florian Krohm., “Equivalence checking using cuts and heaps”, In DAC ’97: Proceedings of the 34th annual conference on Design automation, pages 263–268, New York, NY, USA, 1997. ACM

[17] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, “FRAIGs: A unifying representation for logic synthesis and verification”. ERL Technical Report, EECS Dept., UC Berkeley, March 2005.

[18] R. K. Brayton and C. McMullen, “The decomposition and factorization of Boolean expressions,” Proc. ISCAS ‘82, pp. 29-54.

[19] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, “Cutless FPGA mapping”, ERL Technical Report, EECS Dept., UC Berkeley, 2007.

[20] J. Cong and Y. Ding, “FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs”, IEEE Trans. CAD, vol. 13(1), Jan. 1994, pp. 1-12.

[21] Claude Shannon, “The Synthesis of Two-Terminal Switching Circuits”, Bell System Technical Journal, 28: 1. January 1949 pp 59-98.

[22] S. B. Akers, “Functional testing with binary decision diagrams”, in Eighth Annual Conf. Fault-Tolerant Computing, 1978, pp. 75-82.

[23] R.E.Bryant. “Graph-based algorithms for Boolean function manipulation”, IEEE Transactions on Computers, vol. C-35, n° 8, pp. 677-691, August 1986.

[24] K.S.Brace, R.L.Ruddel, R.E.Bryant. “Efficient implementation of a BDD package”. Proc. of 27th DAC, pp. 272-277, 1990.

[25] Steven J. Friedman and Kenneth J. Supowit, “Finding the Optimal Variable Ordering for Binary Decision Diagrams”, IEEE Transactions on Computers, Vol. 39, No. 5,pp. 710-713, May 1990.

[26] Heh-Tyan Liaw and Chen-Shang Lin, “On the OBDD-Representation of General Boolean Functions”, IEEE Transactions on Computers, Vol. 41, No. 6, pp. 661-664, June 1992.

[27] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification”, IEEE Trans. CAD, Vol. 21(12), 2002, pp. 1377-1394.

[28] D. Chai,A. Kuehlman,“Building a better Boolean matcher and symmetry detector”, proc.of DATE2006,vol.1, pp.1-6.

[29] www.eecs.berkeley.edu/ alanmi/benchmarks/

[30] RevLib, “http://www.revlib.org/”

[31] home.roadrunner.com/ ssolver/

[32] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification.http://wwwcad.eecs.berkeley.edu/ alanmi/abc.

[33] E. Detjens, G.Gannot, R.Rudell, A.L.Sangiovanni-Vinccentelli, A.Wang. “Technology mapping in MIS” ICCAD, 1987, pp. 116-119.

[34] Z. Huang, et al,”Fast Boolean Matching for small Practical Functions”, proc.of IWLS’13.

[35] B. Karunakar Reddy, S. Sabbavarapu, Amit Acharyya, “A New VLSI IC Design Automation Methodology with Reduced NRE Costs and Time-to-Market using the NPN class Representation and Functional Symmetry”, IEEE International Symposium on Circuits and Systems (ISCAS), Australia, 2014, pp. 177-180.

[36] A. Mishchenko, S. Chatterjee, R. K. Brayton, “Improvements to Technology Mapping for LUT-Based FPGAs”, IEEE Trans. CAD, Vol. 26(2), 2007, pp. 240-253.

[37] O. Martinello, F. Marques, R. Ribas, A. Reis, “KL-Cuts:A New Approach for Logic Synthesis Targeting Multiple Output Blocks”, DATE’10, pp. 777-782.

[38] A. Mishchenko, Sungmin Cho, S. Chatterjee, R. Brayton, “Combinational and sequential mapping with priority cuts”, IEEE/ACM ICCAD’07, pp. 354-361.

[39] S. Chatterjee, A. Mishchenko and R. K. Brayton, “Factor Cuts”, IEEE ICCAD06, pp.143-150