

010017173

TK
7874
P85
D008

AREA-DELAY DRIVEN LIBRARY-FREE SYNTHESIS

by

MATTHEW PULLERITS

B.A.Sc., Queen's University, 2005

A thesis
presented to Ryerson University
in partial fulfillment of the
requirement for the degree of
Master of Applied Science
in the Program of
Electrical and Computer Engineering.

Toronto, Ontario, Canada, 2008

© Matthew Pullerits, 2008

PROPERTY OF
RYERSON UNIVERSITY LIBRARY

Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Matthew Pullerits

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Matthew Pullerits

Abstract

AREA-DELAY DRIVEN LIBRARY-FREE SYNTHESIS

Matthew Pullerits

M.A.Sc. Ryerson University, 2008

Current logic synthesis tools rely on pre-defined cell libraries to assemble an arbitrary circuit to perform a needed function. The efficiency of the synthesized circuit relies on the quality and size of the library used in terms of circuit area and critical path delay. It has been shown that in a process supporting five serial NMOS and PMOS transistors, 425803 unique logic gates may be constructed. Clearly this is beyond what is currently available in standard cell libraries.

A richer cell library allows the technology mapper more freedom to better select matches to reduce area, delay and power consumption. This thesis proposes novel algorithms for mapping an input netlist to a library of virtual cells by minimizing logical effort delay, and gate input capacitance to select an architecture which minimizes the design area-delay. An average 69.43% reduction in transistor count, 53.33% reduction in circuit area, with a 3.76% increase in delay has been realized compared to results obtained from Synopsys Design Compiler with high map effort for delay minimization.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Adnan Kabbani for his help and support during my research. His direction and guidance was always appreciated.

I would like to thank my parents, sister, and grandfather for always looking out for my best interest. They supported me, and were patient during my struggles and mistakes.

Lastly, my friends Raymond, Dennis, Adrian, and Edwina who understood when I needed to focus on studying, and were always there to help me.

Table of Contents

List of Figures	vii
1. INTRODUCTION.....	1
1.1 Integrated Circuit Design Techniques	1
1.2 Logic Synthesis.....	2
1.3 Objectives	3
1.4 Main Contributions.....	3
1.5 Thesis Organization	4
2. LIBRARY-BASED LOGIC SYNTHESIS.....	5
2.1 Decomposition	5
2.2 Partitioning.....	5
2.3 Matching	7
2.3.1 Structural Matching	7
2.3.1.1 Simple Tree-Based Matching	8
2.3.1.2 String Matching with Multiple Base Functions.....	9
2.3.2 Boolean Matching.....	12
2.3.2.1 Input Permutation.....	14
2.3.2.2 Input Negation	14
2.3.2.3 Output Negation.....	14
2.3.2.4 Boolean Signatures	14
2.3.2.5 Canonical Representations of Boolean Functions	16
2.4 Covering.....	20
2.5 Summary	21
3. LIBRARY-FREE TECHNOLOGY MAPPING.....	22
3.1 Previous Studies in Library-Free Mapping.....	23
3.2 Summary	27
4. PROPOSED SYNTHESIS ALGORITHMS	28
4.1 Logical Effort.....	28
4.1.1 Estimating Gate Area with Input Capacitance.....	30
4.2 Fast Boolean Matching of Complex Gates	31
4.3 Complex Gate Generation and Indexing.....	32
4.3.1 Complex Gate Library Population.....	33
4.3.2 Permutations of Order and Complementation of Inputs.....	40
4.3.2.1 Input Tracking in P-Canonical Generation	40
4.3.2.2 Input Tracking in NP-Canonical Generation	40
4.3.3 CMOS Transistor Logic Solver	43
4.3.4 CMOS Gate Skewing.....	44
4.3.5 NMOS Network Generation	48
4.3.5.1 NMOS Generation – Dependency on Library Size	51

4.4 Proposed Boolean Tree Matching.....	52
4.4.1 Sub-tree Extraction	52
4.4.2 Matching Algorithm.....	54
4.4.3 Boolean Logic Solver	55
4.5 Proposed Covering Algorithm	56
4.5.1 Minimizing Logical Effort Delay	56
4.5.2 Minimizing Circuit Area.....	61
4.6. Summary	63
5. IMPLEMENTATION	64
5.1 Synthesis Process Overview	64
5.1.1 Input Parsing and DAG Storage	64
5.1.2 Partitioning.....	66
5.1.3 Memory Management.....	67
5.1.4 Timing Restrictions during Sub-tree Extraction	68
5.1.5 User Interface and Result Presentation.....	69
5.1.6 Automated Verification of Results	73
5.2 Summary	74
6. RESULTS	75
6.1 Design Compiler Area Minimization.....	77
6.1.1 DC vs. Developed Tool Area Minimization (H = 4)	78
6.1.2 DC vs. Developed Tool Delay Minimization (H = 4)	79
6.1.3 DC vs. Developed Tool Area Minimization (H = 100)	80
6.1.4 DC vs. Developed Tool Delay Minimization (H = 100)	81
6.1.5 Comparison of Synthesis Algorithms	82
6.2 Design Compiler Delay Minimization.....	83
6.2.1 DC vs. Developed Tool Delay Minimization (H = 2)	84
6.2.2 DC vs. Developed Tool Area Minimization (H = 2)	85
6.2.3 DC vs. Developed Tool Delay Minimization (H = 4)	86
6.2.4 DC vs. Developed Tool Area Minimization (H = 4)	88
6.2.5 DC vs. Developed Tool Delay Minimization (H = 16)	89
6.2.6 DC vs. Developed Tool Area Minimization (H = 16)	90
6.2.7 DC vs. Developed Tool Delay Minimization (H = 100)	91
6.2.8 DC vs. Developed Tool Area Minimization (H = 100)	92
6.2.9 Comparison of Synthesis Algorithms	93
2.3 Summary	96
7. CONCLUSIONS AND FUTURE WORK	97
8. REFERENCES.....	99

List of Figures

Figure 2-1. Single-Fanout Partitioning	6
Figure 2-2. Gate representation of function $f=(ab + c)'$	7
Figure 2-3. Tree decomposition of function $f=(ab + c)'$ into NAND & Inverter gates	8
Figure 2-4. MATCH Algorithm.....	9
Figure 2-5. AND function.....	9
Figure 2-6. AND pattern tree. Inverter = White, NAND = Black, Input = Grey)	9
Figure 2-7. Pattern trees, strings, tree identifiers for 3 common library cells	11
Figure 2-8. Automation for sample library given in Figure 2-7	12
Figure 2-9. Comparisons of two Boolean functions	13
Figure 2-10. Truth table permutations	17
Figure 2-11. Transformations due to reordering of input variables.....	17
Figure 2-12. Transformations due to reordering and inversion of input variables	17
Figure 2-13. Number of NP-equivalents vs. number of input variables.....	18
Figure 2-14. Complete Link Table for 3-input function	18
Figure 2-15. Sample Covering.....	20
Figure 2-16. Alternative Covering.....	20
Figure 2-17. Alternative Covering.....	21
Figure 3-1. Gate combinations possible with series NMOS & PMOS transistors.	23
Figure 3-2. Possible Cover.....	24
Figure 3-3. Alternate Cover	25
Figure 3-4. Simple Circuit for OTR Example	25
Figure 3-5. Transistor-level representation of Figure 3-4.....	26
Figure 3-6. Intermediate transformation of OTR method.....	26
Figure 3-7. Final transformation of OTR method: Resultant complex gate	27
Figure 4-1. Logical Effort of Simple Gates	30
Figure 4-2. Example of Transistor Size Variation in Gates.....	31
Figure 4-3. Library Cell Data Structure	33
Figure 4-4. Library Data Structure.....	33
Figure 4-5. 3x3 Generation Matrix	35
Figure 4-6. 2x2 Generation Matrix	36
Figure 4-7. Gates Extracted from 2x2 Generation Matrix.....	37
Figure 4-8. Transistor Container Data Structure.....	38
Figure 4-9. Transistor Data Structure	38
Figure 4-10. Transistor Node Data Structure.....	39
Figure 4-11. Hierarchy of the Transistor classes.	39
Figure 4-12. Reference Truth Table for NP-Canonical Generation	41
Figure 4-13. Input Order Permutation Function	42
Figure 4-14. Input Inversion Algorithm.....	43
Figure 4-15. Sample Input Permutation & Minterm Positions.....	43
Figure 4-16. Sizing for equal rise & fall times in CMOS gates.....	44
Figure 4-17. Gate generated from 4x4 Generation Matrix	46
Figure 4-18. Gate Generated from 4x4 Matrix with 6 Series Transistors	47
Figure 4-19. Generating NMOS Network from PMOS Structure	48

Figure 4-20. Truth Table for NOR Gate	49
Figure 4-21. Truth Table for NAND Gate	49
Figure 4-22. PMOS Network $f=!(A+B)C$	50
Figure 4-23. PMOS Network $f=!(XZ+Y)$	50
Figure 4-24. Generated Truth Tables & P-canonical form of Figure 38	51
Figure 4-25. Generated Truth Tables & P-canonical form of Figure 39	51
Figure 4-26. Extracted Sub-trees from Sample Graph.....	52
Figure 4-27. Proposed sub-tree extraction method.....	54
Figure 4-28. Proposed Boolean tree matching algorithm	55
Figure 4-29. Recursive Boolean logic solver pseudo code.....	56
Figure 4-30. Matches Explored in Covering Algorithm.....	58
Figure 4-31. Gate Transformations through Boolean Matching.....	59
Figure 4-32. Proposed logical effort delay minimizing covering algorithm	60
Figure 4-33. Proposed area minimizing covering algorithm	61
Figure 5-1. Simple Example of BENCH Code.....	65
Figure 5-2. Boolean Representation of BENCH Code in Figure 5-1	65
Figure 5-3. Node Object	66
Figure 5-4. Partition Object	67
Figure 5-5. LibraryMatch Object.....	68
Figure 5-6. Configuration Tab of Synthesis Tool.....	69
Figure 5-7. Status of Synthesis Process	70
Figure 5-8. Sample Report Generated by Synthesis Tool.....	71
Figure 5-9. Browsing Partitions.....	72
Figure 5-10. Display of Matching Library Cell	72
Figure 5-11. Display of Critical Path.....	73
Figure 5-12. Result of Automated Verification	74
Figure 6-1. ISCAS'85 Benchmark Circuits	75
Figure 6-2. DC Area Minimization vs. Developed Tool Area Minimization – H=4	78
Figure 6-3. Design Compiler vs. Developed Tool Δ Area-Delay Product	78
Figure 6-4. DC Area Minimization vs. Developed Tool Delay Minimization – H=4.....	79
Figure 6-5. DC vs. Developed Tool Δ Area-Delay Product (Area vs. Delay)	79
Figure 6-6. DC Area Minimization vs. Developed Tool Area Minimization – H=100 ...	80
Figure 6-7. DC vs. Developed Tool Δ Area-Delay Product (Area vs. Area)	80
Figure 6-8. DC Area Minimization vs. Developed Tool Delay Minimization – H=100..	81
Figure 6-9. DC vs. Developed Tool Δ Area-Delay Product (Area vs. Delay)	81
Figure 6-10. DC Area Minimization (H=4).....	82
Figure 6-11. DC Area Minimization (H=100).....	83
Figure 6-12. DC Delay Minimization vs. Tool Delay Minimization – H=2	85
Figure 6-13. DC vs. Developed Tool Δ Area-Delay Product (Delay vs. Delay).....	85
Figure 6-14. DC Delay Minimization vs. Tool Area Minimization – H=2	86
Figure 6-15. DC vs. Developed Tool Δ Area-Delay Product (Delay vs. Area)	86
Figure 6-16. DC Delay Minimization vs. Tool Delay Minimization – H=4	87
Figure 6-17. DC vs. Developed Tool Δ Area-Delay Product (Delay vs. Delay).....	87
Figure 6-18. DC Delay Minimization vs. Tool Area Minimization – H=4	88
Figure 6-19. DC vs. Developed Tool Δ Area-Delay Product (Delay vs. Area)	88
Figure 6-20. DC Delay Minimization vs. Developed Tool Delay Minimization – H=16	89

Figure 6-21. DC vs. Developed Tool Δ Area-Delay Product (Delay vs. Delay).....	89
Figure 6-22. DC Delay Minimization vs. Tool Area Minimization – H=16	90
Figure 6-23. DC vs. Developed Tool Δ Area-Delay Product (Delay vs. Area)	90
Figure 6-24. DC Delay Minimization vs. Tool Delay Minimization – H=100	91
Figure 6-25. DC vs. Developed Tool Δ Area-Delay Product (Delay vs. Delay).....	91
Figure 6-26. DC Delay Minimization vs. Developed Tool Area Minimization – H=100	92
Figure 6-27. DC vs. Developed Tool Δ Area-Delay Product (Delay vs. Area)	92
Figure 6-28. DC Delay Minimization (H=2)	93
Figure 6-29. DC Delay Minimization (H=4)	94
Figure 6-30. DC Delay Minimization (H=16)	95
Figure 6-31. DC Delay Minimization (H=100)	96

1. Introduction

1.1 Integrated Circuit Design Techniques

Since the inception of the integrated circuit in April of 1961 by Robert Noyce [1], the technology has rapidly evolved to a level where every aspect of our lives is driven by computer chips – from health care to national defense.

Application Specific Integrated Circuits (ASICs) in production today are composed of a few, to a few hundred million transistors, depending on the intended task of the chip. Transistors are composed of interconnected layers of silicon, metal (most commonly used metals are aluminum or copper), and insulator (most commonly used is SiO_2 – glass). These ASICs may be manufactured with many interconnecting layers of metal, giving a total metal layer count in the chip of six in a $0.06\mu\text{m}$ process [2]. Individual layers are separated, and “mask layouts” are created for each layer to be produced. These mask layouts are the final step in ASIC design, and are directly used in production of the chip.

ASICs are currently produced in one of three ways: Full Custom, Fully Automatic, and Semi-Custom. In a full custom process, design engineers draw the final mask layout of the circuit to be produced using computer aided design (CAD) software. This allows the engineer to customize every aspect of the chip in regards to transistor sizing for optimal power consumption, delay, or area specifications. As modern integrated circuits may contain around 1-billion transistors, a full custom design although would result in the most optimal product with respect to speed or power consumption, is not always possible or practical due to time or budget constraints.

In a fully automatic process, circuits to be produced are defined in a high level hardware description language (HDL) such as Verilog, VHDL, or SystemC. Using these languages, a design engineer may describe their circuit in a manner similar to that of a computer

program, making use of programming techniques such as `while` loops, `for` loops, and custom data structures. Once a circuit is defined in a HDL, it may be synthesized to a production-ready mask layout easily with currently available synthesis tools, such as Synopsys Design Compiler, or the Berkeley-made SIS. Relative to a fully custom design, usage of HDLs may allow for time-savings of many orders of magnitude. This is a great economic savings for the design firm with respect to engineering time required to develop a production-ready circuit.

Although synthesis allows for a great savings of time to produce a functional ASIC, current CAD tools are not able to produce circuits which are optimal with respect to area and delay. On occasion it is necessary to produce a semi custom design, in which a layout obtained from HDL synthesis is modified to conform to required design specifications.

1.2 Logic Synthesis

Logic synthesis is the process of converting a circuit described in a hardware definition language into a netlist of gates, which may be made to a layout ready for fabrication. Synthesis tools today rely on libraries of pre-defined cells which are used to construct the circuit. These cells act as building blocks which can be connected to produce a desired function.

Cell libraries have played an important role in logic synthesis for the past three decades. They have allowed engineers to quickly utilize these pre-designed building blocks to assemble and fabricate an arbitrary circuit to perform a needed function. While convenient, these library files require significant investment in engineering time to develop and maintain for every generation of technology process.

The efficiency of the synthesized circuit relies heavily on the quality and size of the library used [3]. However, creating rich cell libraries to facilitate the continued demand for faster, smaller, and more complex ASICs is not economically permissible. As such, many of the advances in deep sub-micron fabrication are not being fully utilized by implementing functions in complex gates [4]. It has been shown that in a technology

process capable of supporting five serial NMOS and PMOS transistors, 425803 unique logic gates may be constructed [5]. Clearly this is beyond what is currently available in standard cell libraries. A synthesis tool which is able to dynamically generate library cells during the synthesis process, rather than relying on a pre-constructed cell library, is able to much better utilize resources inside the ASIC by generating a complex gate where previously many smaller library cells would have to be used. These resources being minimized may include transistor count, silicon area, power consumption, or critical path delay.

1.3 Objectives

- To develop library-free synthesis algorithms which:
 - Has a complexity similar to current state-of-the-art synthesis tools
 - Minimizes the power consumption and delay. Since it is well known power dissipation is proportional to the design area, this objective can be achieved by minimizing the area-delay product.
 - Guarantees the consistency of the quality of the synthesized design by ensuring **a)** independence from the initial decomposition of the circuit, and **b)** absolute logic minimization
- To develop a CAD tool which implements the proposed algorithms in a high level language such as C or C++

1.4 Main Contributions

This thesis proposes novel synthesis algorithms for use in library-free synthesis. Using this technique allows the direct production of a transistor netlist of the design, as opposed to the cell-based synthesis method. The proposed technique is independent of the initial decomposition which improves the overall area-delay product. Finally, the developed synthesis algorithms will generate circuits which are not tied to any particular technology process, allowing inexpensive portability between processes.

1.5 Thesis Organization

The remainder of this document is organized as follows. An overview of library-based logic synthesis is presented in chapter 2. Chapter 3 reviews the theory of library-free logic synthesis and previous works in this area. Chapter 4 presents the proposed library-free logic synthesis algorithms. Implementation details are outlined in chapter 5. Experimental results obtained from executing the developed tool with benchmark circuits, and comparisons against industry standard synthesis tools is presented in chapter 6. Finally, conclusions and room for future work is discussed in chapter 7.

2. Library-Based Logic Synthesis

A cell library contains a set of pre-constructed mask layouts of logic functions which are ready for fabrication on a specific technology process. These logic functions range from simple gates such as Inverters, NANDs or NORs, to complex adders and registers. Accurate area and timing information is available for these cells, as well as various sizes of these gates in order to drive different loads.

Logic synthesis is a conversion process which translates a gate level representation of an HDL into a layout ready for further processing such as power planning for production on a particular technology using pre-characterized cells from the cell library. This process can be broken down into four unique steps: decomposition, partitioning, matching, and covering.

2.1 Decomposition

During the technology independent stage of logic minimization, functions are decomposed in to a network of base functions. These base functions could be a set of two input NAND, NOR and Inverter gates. Decomposition allows for a complex circuit to be broken down into smaller gates which are easier to analyze and synthesize [6]. While the structure of the circuit is modified during decomposition, the functionality is maintained. At minimum, the target technology cell library must contain the set of base functions decomposed to, in order to guarantee a complete covering.

2.2 Partitioning

Boolean networks may be represented as directed acyclic graphs (DAGs) with vertices representing Boolean functions, and edges representing connections, or wires. It has been shown [7] that network partitioning into trees is an essential heuristic step in the technology mapping process, as there is currently no algorithm to optimally cover a DAG

in polynomial time¹. Many studies have been conducted on the complexity of covering a DAG, resulting in the conclusion that the problem is NP-hard [8]. As complex Boolean networks may contain thousands or potentially millions of functions, each with at least one, and in most cases, more than one covering, a solution is beyond the capabilities of covering tools. In order to solve the covering problem in a practical amount of time, heuristics must be applied to simplify the problem. One such heuristic is partitioning the DAG into singly-rooted trees [7] [9]. Partitioning serves two purposes: first it simplifies the covering problem so that each network being covered is smaller. Second, by this simplification, partitioning enables the covering problem to become practical. It has been shown that although the covering problem is intractable, computation time is reasonable for problems of practical size [10].

It should be noted that partitioning of a Boolean network serves as a heuristic step, and although simplifies the computation of an optimal covering, will hinder the quality of the overall result.

Singe-rooted trees may be partitioned by traversing the network from primary outputs to primary inputs, creating a new partition for every primary output, or for any multiple fanout traversed. Figure 2-1 illustrates the partitions obtained by traversing from outputs to inputs, with X's marking partition locations.

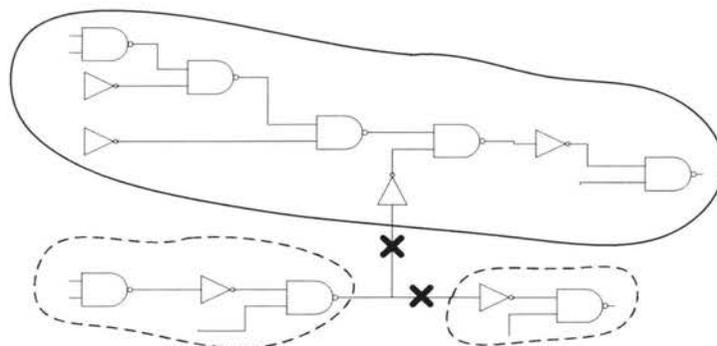


Figure 2-1. Single-Fanout Partitioning

¹ Although numerous algorithms have been proposed to cover a DAG, all implement some sort of heuristic [8] [39] [24] [40] to execute within an acceptable time frame.

2.3 Matching

Matching can be broken down into two flavors: structural matching and Boolean matching [11]. Structural matching analyzes a decomposed, partitioned circuit as a graph, and searches for matching patterns in the cell library. Boolean matching looks at the logic *function* being accomplished by the circuit, and seeks similar functions or permutations of functions in the cell library by comparing the Boolean truth tables of the functions.

2.3.1 Structural Matching

After partitioning, the original DAG will have been broken into a forest of trees. Each individual tree is called a *subject graph* [7], with edges representing wires, or connections between gates, and vertices representing logic functions. A root node is defined as the output of the function, or the node which all other nodes lead to. A child node is defined as a node which stems from another node at a higher level. Leaf nodes are always inputs to the subject graph, and are the lowest-level child nodes. These subject graphs contain multiple inputs, but only a single output.

For example, consider the Boolean network illustrated in Figure 2-2, representing the function $f = \overline{ab + c}$.

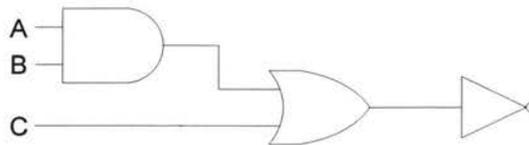


Figure 2-2. Gate representation of function $f = \overline{ab + c}$

Given the base functions: NAND and NOT, the network in Figure 2-2 may be transformed into the subject graph seen in Figure 2-3. NAND gates are represented by the character N, and NOT gates are represented by the character I. Lower case letters represent inputs.

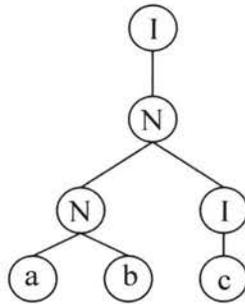


Figure 2-3. Tree decomposition of function $f=(ab + c)'$ into NAND & Inverter gates

Structural matching algorithms function by checking the isomorphism between two rooted trees. A subject graph, or a portion of a subject graph is compared with stored pattern graphs of functions within a cell library. If the two are found to be isomorphic, it can be said that the two match.

2.3.1.1 Simple Tree-Based Matching

One of the simplest forms of tree-based matching considers the case when there is only one base function which the original Boolean network is decomposed to. For the purposes of this Section, we will assume that the two-input NAND gate is used. Similarly, a NOR gate could be used instead.

As an Inverter may be generated by tying two inputs of a NAND together, only one type of non-terminal vertex is required; Differentiation between a NAND and Inverter gate may be determined by the number of children beneath the vertex.

MATCH, a simple tree-based matching algorithm operates by comparing subgraphs of the subject tree to every pattern tree in the library, checking for isomorphism [12]. The algorithm is outlined in Figure 2-4.

```

MATCH (subject, pattern) {
  If (subject is a leaf) return (TRUE);
  Else {
    If (pattern is a leaf) return (FALSE);
    If (degree(pattern) != degree(subject)) return(FALSE);
    If (degree(pattern) == 1) {
      Uc = child of subject; Vc = child of pattern;
      Return (match(Uc, Vc))
    } else {
      Ul = left-child of subject; Ur = right-child of subject;
      Vl = left-child of pattern; Vr = right-child of pattern;
      Return (match(Ul, Vl) . match(Ur, Vr) + match(Ur, Vl) . match(Ul, Vr));
    }
  }
}

```

Figure 2-4. MATCH Algorithm

2.3.1.2 String Matching with Multiple Base Functions

It is not always the case that Boolean networks will be decomposed to one base function, thus an alternate method is required to match and cover networks with multiple types of base function. The remainder of this section describes structural matching based on string encodings of trees and string recognition.

Tree String Encoding

Trees may be stored in memory as an encoding of strings, each representing one path from the root to a leaf [12]. Consider the AND function, decomposed into a NAND and Inverter, seen below in Figure 2-5.



Figure 2-5. AND function

The tree representation of this decomposed AND function may be observed in Figure 2-6.

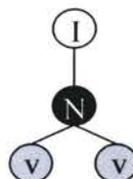


Figure 2-6. AND pattern tree. Inverter = White, NAND = Black, Input = Grey)

Pattern strings for the AND gate may now be generated from the pattern tree illustrated in Figure 2-6, which provide a textual description of every path from the root to each input-leaf. Each unique base function is assigned a unique character representation, such as I for Inverter, or N for NAND. The format of a pattern string is as follows: the string is a pattern of character identifiers, followed by a number, representing the next child node to traverse in order to reach the target leaf.

Looking at Figure 2-6, one can obtain the pattern strings "I1N1v" and "I1N2v". Breaking down both of these strings, first looking at "I1N1v", we can see that the root node is an Inverter, and it's first (and only) child is connected to the output of a NAND gate, whose first input is connected to a leaf node, v.

The second string, "I1N2v" may be broken down as follows: The root node is an Inverter, and it's first (and only) child is connected to the output of a NAND gate, whose second input is connected to a leaf node, v.

Tree Sting Matching using Automation String Detection

A partitioned, decomposed subject tree may be covered using pattern-string detection algorithms by matching substrings of a subject tree string with pattern strings in a cell library. One such method of recognizing pattern strings within a subject tree is to build an automation for the entire cell library [12], which will compare input subject strings one character at a time against the library automation. Such an automation is built incrementally, one pattern string at a time. Initially the automation is set in a reset state, until an appropriate matching input is received. Once an initial input is received, the automation begins operating as a finite-state machine, until reaching a terminating state, when it will return to the reset status.

An automation built from the sample library illustrated in Figure 2-7 is illustrated in Figure 2-8.

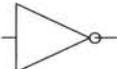
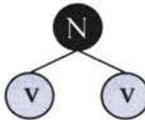
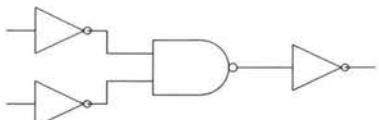
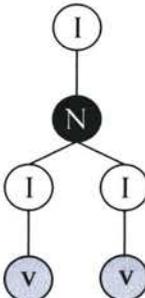
Library Function	Decomposition into NAND/INV base functions	Pattern Tree	Pattern String	Tree Identifier
INV			I1v	t1.1
NAND2			N1v N2v	t2.1 t2.2
NOR2			I1N1I1v I1N2I1v	t3.1 t3.2

Figure 2-7. Sample pattern trees, pattern strings, pattern tree identifiers for 3 common library cells

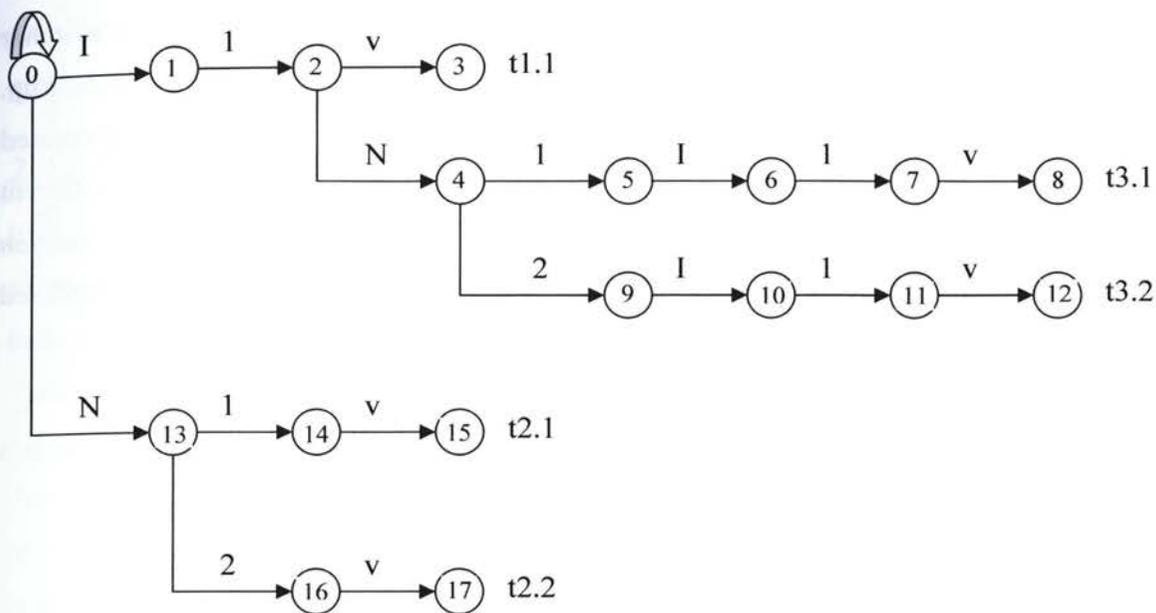


Figure 2-8. Automation for sample library given in Figure 2-7

Example. We will now use the simple automation illustrated in Figure 2-8 to determine if the subject tree in Figure 2-3 may be covered by this library. The following subject strings define the subject tree illustrated in Figure 2-3: $\{I1N1N1v, I1N1N2v, I1N2I1v\}$

By inputting these strings into the automation, one can see that only $I1N2I1v$ has a perfect match with tree string $t3.2$; However since $t3.1$ does not have a match, this cell cannot be used in the cover. After pruning the subject tree, we *can* find a match $I1v$, by trees $t1.1$. Similarly, one can find a match for the strings $N1v$, and $N2v$, covered by tree strings $t2.1$, and $t2.2$, respectfully.

2.3.2 Boolean Matching

As described in the previous Section, structural matching will find a matching library cell in order to generate a complete network cover; however, for complex functions, there may be many different decompositions of the same function, and the quality of the resulting mapped circuit depends heavily on the initial decomposition.

In order to combat this dependency on the initial decomposition, a structural matcher could pre-calculate every possible decomposition and store this in the library for lookup. This lookup procedure becomes computationally hard as complex functions are generated with up to and exceeding 8 inputs, with potentially many thousands of different decompositions. Boolean matching allows us to take a step back, and analyze the *function* being calculated, regardless of the ordering of variables, or structure of decomposed network.

One can say that two functions, $f(x)$ and $g(y)$, are Boolean-equivalent if $f(x) \oplus g(y)$ is a tautology – that their truth tables are equal.

Consider the following two functions:

$$f = xy + x'y' + y'z$$

$$g = xy + x'y' + xz$$

The two are very different structurally, but examining their truth tables in Figure 2-9, one can see that they are a Boolean match. In this case, $f(x) \oplus g(y)$, as for every matching input combination, the same output is obtained.

X	Y	Z	F	X	Y	Z	G
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	1	0	0	1	1	0
1	0	0	0	1	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Figure 2-9. Comparisons of two Boolean functions

It is uncommon for a subject function to exactly match any library function without some sort of permutation to it. There are three unique permutations which may be performed on a subject function in order to match it to a library gate. First, the order of the inputs may be changed – this is called *P*-equivalence. Second, the inputs may be complimented in any combination, as well with permutation of the input variables, which is referred to

as *NP*-equivalence. Finally the output may be complimented with combination of the previous two equivalences, which is referred to as *NPN*-equivalence.

2.3.2.1 Input Permutation

It is not always the case that $f(x)$ is directly equivalent to $g(y)$. In most cases, it is necessary to explore the possible permutations of input variables that yield equivalent behavior. It can be said that f and g are *P-equivalent* if there exists an ordering of input variables i of y such that $f(x) \oplus g(i)$ is a tautology.

For example, the functions $f = ab + c$, and $g = cb + a$ are *P-equivalent*, as a rearrangement of their input variables will result in the same function being computed. This equivalency is not as evident initially to the eye as the number of variables in a function exceeds three. In an n -variable function, $n!$ unique input permutations exist.

2.3.2.2 Input Negation

As Inverters are inexpensive to construct, exploring the possibility of inverting the inputs of a function opens up many new options to more efficiently match a function to a library cell. An n -input function requires 2^n equivalency checks in order to explore all combinations of input permutations, and *NP*-equivalence requires $n!2^n$ computations.

2.3.2.3 Output Negation

By simply allowing the output of a gate to be complimented by inserting an additional Inverter, many new cost saving matches may be explored. This added Inverter doubles the required equivalency checks, with *NPN*-equivalence costing $n!2^{n+1}$ computations for an n -input function.

2.3.2.4 Boolean Signatures

Boolean signatures may be used to reduce the number of tautology checks required to determine if a matching function has been found. A signature is something which identifies certain properties of a Boolean function, however does not guarantee a match. Two Boolean functions cannot be equivalent and have different signatures; However, two

Boolean functions sharing the same signature are not necessarily equivalent. Identifying characteristics of a Boolean function which may be used for a signature could be symmetries, unateness, size of cofactors, etc.

If one were to generate a signature of a Boolean function based on symmetry, it could be defined as follows: A *symmetry set* [13] is a set of variables that are pairwise interchangeable without affecting the logic functionality. A *symmetry class* is an ensemble of symmetry sets with the same cardinality.

Example. Consider the function $f = x_1x_2 + x_3x_4x_5 + x_6x_7$

The support variables of $f(x)$ can be partitioned into three symmetry sets: $\{x_1x_2\}$, $\{x_3x_4x_5\}$, $\{x_6x_7\}$. There are two non-void symmetry classes: $C_2 = \{\{x_1x_2\}, \{x_6x_7\}\}$ and $C_3 = \{x_3x_4x_5\}$. This gives a signature of $[0,2,1,0,0,0,0]$, with each dimension of the signature representing the number of occurrences of one degree of cardinality.

Other signatures may be obtained by examining other aspects of the function in question, for example the *satisfy count*, which is the number of minterms in the function. Regardless of the characteristics used to create the signature, once generated they can be an excellent tool to determine if two functions *do not* match. In practice, signatures may be pre-computed for all available library functions and stored in a hash table. The signature for a function in question may quickly be generated and checked against the table to obtain a list of potential matches.

Example. Consider the following pattern function:

$$f = x_1x_2a + x_1x_2'b + x_1'x_3c + x_1'x_3'd$$

The function has 7 variables, 4 of which are unate, and 3 which are binate. Using a simple Boolean signature to identify this function by its unateness, one can simplify the number of tautology checks required. Assuming no input or output negations are required, with no signature one would need to check $7! = 5040$ different variable orderings in order to locate a matching function. If the information that there are 4 unate and 3 binate variables in the function is available, only $4!3! = 144$ tautology checks is required, a 3500% increase in efficiency.

2.3.2.5 Canonical Representations of Boolean Functions

While Boolean signatures will certainly aid in the matching process, it can be seen that if strictly tautology checks are employed to determine if two functions are a Boolean match, that for each N -input function being checked, $N!2^{N+1}$ tautology checks are required per function. If every subject gate being covered must be matched against a library containing over 400,000 unique functions, the matching process quickly becomes impractical.

Utilizing the canonical form of a Boolean function, this matching process can be simplified to a single integer or string comparison. The canonical form of a Boolean function acts as a unique signature, which differentiates it from all other functions with the same number of inputs, under certain conditions. There exist three equivalency classes which functions may be grouped by: P-equivalent, NP-equivalent, and NPN-equivalent. Any function falling under one of these groups may be interchanged with another in the same group with the appropriate permutations.

Many canonical representations for equivalency checking have been developed over the history of automated circuit design and synthesis which perform differently with respect to CPU time and memory requirements. Two Boolean canonical form generation algorithms were implemented in this synthesis tool, to be used under different circumstances given their performance and abilities.

NPN-equivalent matching

Debnath and Saso proposed an efficient canonical form for Boolean matching with permutation and input/output negation for use in large libraries [14]. Their method proposed that functions which are NP-equivalent belong to the same NP-equivalency class. In an *NP-equivalence class*, the function which has the smallest binary number representation is the NP-representative (NPR) of that class, and is used in matching. If two functions, f and g share the same NPR, they are *NP-equivalent*, and one is able to

represent the other with a permutation and complementation of its input variables. Output negation is checked by comparing the NPR values of f with g and $!g$.

The function in Figure 2-10(b) is obtained by permuting the input variable order of that in Figure 2-10(a). It should be noted the transformation in position of the minterms is irrelevant of the value of the minterm. These transformations may be pre-computed and stored for an arbitrary number of inputs. Figure 2-11 illustrates these transformations for any three-input function.

<table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="border-right: 1px solid black; padding: 5px;">x_1</th> <th style="border-right: 1px solid black; padding: 5px;">x_2</th> <th style="border-right: 1px solid black; padding: 5px;">x_3</th> <th style="padding: 5px;">$f(x_1, x_2, x_3)$</th> </tr> </thead> <tbody> <tr><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="padding: 5px;">c_0</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="padding: 5px;">c_1</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="padding: 5px;">c_2</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="padding: 5px;">c_3</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="padding: 5px;">c_4</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="padding: 5px;">c_5</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="padding: 5px;">c_6</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="padding: 5px;">c_7</td></tr> </tbody> </table>	x_1	x_2	x_3	$f(x_1, x_2, x_3)$	0	0	0	c_0	0	0	1	c_1	0	1	0	c_2	0	1	1	c_3	1	0	0	c_4	1	0	1	c_5	1	1	0	c_6	1	1	1	c_7	<table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="border-right: 1px solid black; padding: 5px;">x_1</th> <th style="border-right: 1px solid black; padding: 5px;">x_2</th> <th style="border-right: 1px solid black; padding: 5px;">x_3</th> <th style="padding: 5px;">$f(x_3, x_2, x_1)$</th> </tr> </thead> <tbody> <tr><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="padding: 5px;">c_0</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="padding: 5px;">c_4</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="padding: 5px;">c_2</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="padding: 5px;">c_6</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="padding: 5px;">c_1</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="padding: 5px;">c_5</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="padding: 5px;">c_3</td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="padding: 5px;">c_7</td></tr> </tbody> </table>	x_1	x_2	x_3	$f(x_3, x_2, x_1)$	0	0	0	c_0	0	0	1	c_4	0	1	0	c_2	0	1	1	c_6	1	0	0	c_1	1	0	1	c_5	1	1	0	c_3	1	1	1	c_7
x_1	x_2	x_3	$f(x_1, x_2, x_3)$																																																																						
0	0	0	c_0																																																																						
0	0	1	c_1																																																																						
0	1	0	c_2																																																																						
0	1	1	c_3																																																																						
1	0	0	c_4																																																																						
1	0	1	c_5																																																																						
1	1	0	c_6																																																																						
1	1	1	c_7																																																																						
x_1	x_2	x_3	$f(x_3, x_2, x_1)$																																																																						
0	0	0	c_0																																																																						
0	0	1	c_4																																																																						
0	1	0	c_2																																																																						
0	1	1	c_6																																																																						
1	0	0	c_1																																																																						
1	0	1	c_5																																																																						
1	1	0	c_3																																																																						
1	1	1	c_7																																																																						
(a)	(b)																																																																								

Figure 2-10. Truth table permutations

$f(x_1, x_2, x_3)$	$f(x_1, x_3, x_2)$	$f(x_2, x_1, x_3)$	$f(x_2, x_3, x_1)$	$f(x_3, x_1, x_2)$	$f(x_3, x_2, x_1)$
c_0	c_0	c_0	c_0	c_0	c_0
c_1	c_2	c_1	c_4	c_2	c_4
c_2	c_1	c_4	c_1	c_4	c_2
c_3	c_3	c_5	c_5	c_6	c_6
c_4	c_4	c_2	c_2	c_1	c_1
c_5	c_6	c_3	c_6	c_3	c_5
c_6	c_5	c_6	c_3	c_5	c_3
c_7	c_7	c_7	c_7	c_7	c_7

Figure 2-11. Minterm transformations due to reordering of input variables

$f(x_1, x_2, x_3)$	$f(x_1, !x_3, x_2)$	$f(!x_3, x_2, !x_1)$	$f(!x_2, !x_1, x_3)$
c_0	c_2	c_5	c_6
c_1	c_0	c_1	c_7
c_2	c_3	c_7	c_2
c_3	c_1	c_3	c_3
c_4	c_6	c_4	c_4
c_5	c_4	c_0	c_5
c_6	c_7	c_6	c_0
c_7	c_5	c_2	c_1

Figure 2-12. Minterm transformations due to reordering and complementation of input variables

Similar to Figure 2-11, Figure 2-12 illustrates four of the NP-equivalents of a three-variable function. All NP-equivalents of a three-variable function are visible in Figure 2-14. In order to match two Boolean functions, both of their NPR values must be available in order to determine if they are NP-equivalents. Calculating the NPR value of a function gets increasingly difficult as the number of inputs to the function increases. Figure 2-13 illustrates the geometrical growth in the number of computations required versus the number of input variables to a function. The algorithm proposed by Debnath and Saso will allow for very fast NPN-equivalence matching, however for inputs greater than seven, becomes impractical to calculate the match. As such, an alternate canonical form is required for matching functions with more than seven inputs.

Maximum number of variables	Maximum number of NP-equivalents
3	48
4	384
5	3840
6	46080
7	645120
8	10321920

Figure 2-13. Number of NP-equivalents vs. number of input variables

Link Tables are pre-generated up for up to 7-input functions to allow for quick generation of the NP-equivalent set of functions for a given truth table. An example Link Table for a three-input function can be seen in Figure 2-14. The leftmost column of this table (highlighted in grey) is the reference column, where the truth table for the function in question is entered.

0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4	4	5	5	5	5	5	6	6	6	6	6	6	7	7	7	7	7	7			
1	1	2	2	4	4	0	0	3	3	5	5	0	0	3	3	6	6	1	1	2	2	7	7	0	0	5	5	6	6	1	1	4	4	7	7	2	2	4	4	7	7	3	3	5	5	6	6
2	4	1	4	1	2	3	5	0	5	0	3	3	6	0	6	0	3	2	7	1	7	1	2	5	6	0	6	0	5	4	7	1	7	1	4	4	7	2	2	4	5	6	3	6	3	5	
3	5	3	6	5	6	2	4	2	7	4	7	1	4	1	7	4	7	0	5	0	6	5	6	1	2	1	7	2	7	0	3	0	6	3	6	0	3	0	5	3	5	1	2	1	4	2	4
4	2	4	1	2	1	5	3	5	0	3	0	6	3	6	0	3	0	7	2	7	1	2	1	6	5	6	0	5	0	7	4	7	1	4	1	7	4	7	2	4	2	6	5	6	3	5	3
5	3	6	3	6	5	4	2	7	2	7	4	4	1	7	4	5	0	6	0	6	5	2	1	7	1	7	2	3	0	6	0	6	3	3	0	5	0	5	3	2	1	4	1	4	2		
6	6	5	5	3	7	7	4	4	2	2	7	4	4	1	1	6	6	5	5	0	7	7	2	2	1	1	6	3	3	0	0	5	5	3	3	0	0	4	4	2	2	1	1				
7	7	7	7	7	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	4	4	4	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	0	0	0	0	0		

Figure 2-14. Complete Link Table for 3-input function

The remaining cells in the table contain memory pointers to the appropriate cell in the reference column. As such, once the reference column contains the truth table of the input

function, the remaining cells will instantly automatically contain the complete *NP-equivalent* set. The NP-representative of this set may quickly be found by comparing the cell values row-by-row, and eliminating columns which will not give a minimal column summation. This is accomplished by comparing every cell value in one row; If there exists any cell containing a zero value in the row, the column of any cell containing a one value is discarded from future comparisons. Once every row is processed, the remaining column(s) is the NP-representative of that set.

PN-equivalent matching

Ciric and Sechen proposed a P-equivalent canonical form capable of efficiently handling functions with up to 25 inputs, which is the maximum which would be required by a CMOS technology process capable of supporting five serial PMOS and NMOS devices in a single complex gate [15]. Negation of the output may be achieved by generating two *P*-equivalent canonical forms per library cell, one with output negation and one without.

The canonical form is generated from a function's truth table, firstly eliminating any row which does not equate to one, leaving only the minterms. Secondly, the function output column is pruned, leaving only a table of inputs which produced a one output. This table may be quite large, with the number of rows depending on the number of minterms in the function. In order to reduce this table before canonical form generation, logic minimization may be applied to generate a table of Boolean values and *don't cares*. The developed synthesis tool initially implemented the Quine-McCluskey [16] Boolean minimization algorithm, however it was not able to efficiently handle large tables. The minimization algorithm as proposed in [17] was eventually implemented as it allowed for much faster runtimes. If the resulting table contains ones and *don't cares*, the don't cares are translated to zeros. If the resulting table contains zeros and *don't cares*, the zeros are translated to ones, and the don't cares to zero. A decreasing weight is applied to every cell in the table, decreasing in the direction from left to right, top to bottom. The columns and rows of the table are to be permuted until a minimum total table weight is reached. The Boolean values in this minimum weight truth table is the canonical form. They may

be extracted into a string, or converted to integer form for using as the index in a library hash table.

2.4 Covering

The matching process identifies many possible matches for every node in every subject graph. The goal of covering is to determine the set of matches which a) completely covers the subject graph with library cells, and b) optimally does so with respect to a set of costs such as area or delay.

The first step in the covering process is covering the subject graph with the base functions available in the library. Figure 2-15 illustrates the simplest network covering, with each base function covered independently, whereas Figure 2-16 and Figure 2-17 show alternative coverings using available three-input gates.

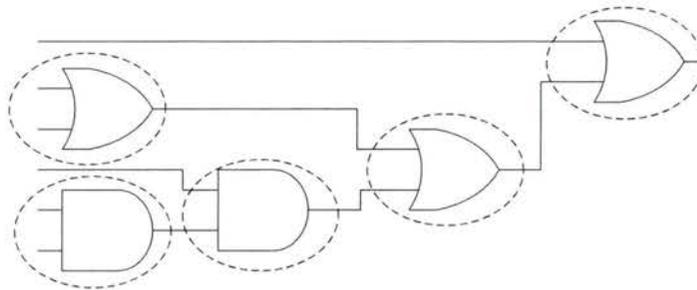


Figure 2-15. Sample Covering

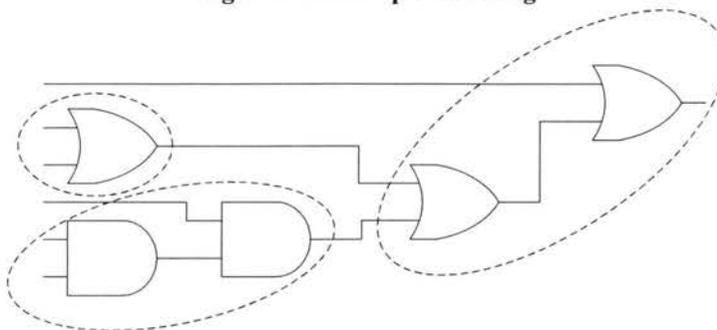


Figure 2-16. Alternative Covering

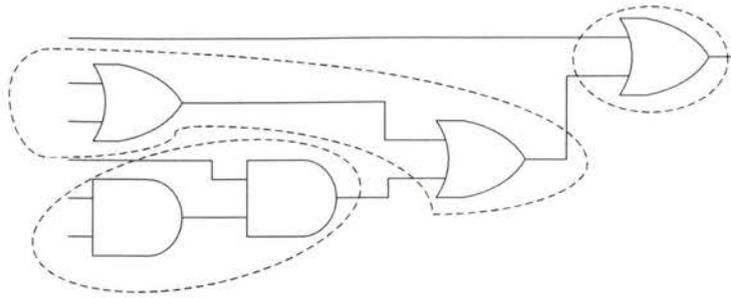


Figure 2-17. Alternative Covering

2.5 Summary

This chapter has overviewed the four main steps in current library-based synthesis techniques, which are decomposition, partitioning, matching and covering. Structural and Boolean matching algorithms were outlined and compared. Boolean signatures were discussed and their drawbacks were outlined. The Boolean canonical form generation algorithms described in section 2.3.2.5 were selected for implementation in the developed synthesis tool. The following chapter will discuss a library-free approach to logic synthesis, and its advantages over a cell-library.

3. Library-Free Technology Mapping

Libraries have played an important role in logic synthesis for the past three decades. They have allowed designers to quickly utilize pre-designed building blocks to assemble and fabricate an arbitrary circuit to perform a needed function. Cell libraries also allow accurate predictions of silicon wafer area utilization and path delay, as individual cells are pre-characterized with precise area, loading, and timing information. These library files require significant investment in engineering time to develop and maintain for every generation of technology process.

As the demand for faster, smaller, and more complex ASICs is increasing, the rate at which technology processes change is not economically permissible of adequately equipped cell libraries to be developed. As such, many of the advances in deep sub-micron (DSM) fabrication are not being fully utilized by implementing functions in complex gates [4]. It has been shown that in a technology capable of supporting five serial NMOS and PMOS transistors, 425803 unique logic gates may be constructed [5].

Aggressive scaling has advanced the state of optical lithography to resolutions of 22nm and below, and electron beam lithography has demonstrated capability to produce minimum features that are less than 10nm wide. These incredible advances in fabrication technology are not only pushing the physical limits of fabrication as we approach atomic sizes, but also come at a huge cost of over \$1 billion to bring a laboratory demonstration to a manufacturable technology [2]. Clearly there is the need to improve upon current design techniques in order to better utilize the resources, i.e. transistors, which are placed within these integrated circuits. Optimization of the circuit architecture to reduce delay and area will allow for improvement on a technology process without reducing the minimum feature size.

The efficiency of the generated circuit relies heavily on the quality of the library used. The larger the available library, the better the mapping tool is able to optimize a given circuit on silicon as more precisely matching cells will be available for placement. The maximum number of unique gates in a library is defined as a function of the number of series NMOS and PMOS transistors $s(n,p)$. It has been shown [18] that a circuit's total number of transistors may be reduced by up to 35% when an $s(4,4)$ library-free approach (containing 3503 unique gates) is used in matching as opposed to using a simple gate library containing three cells: Inverter, 2-input NAND, 2-input NOR.

It can be seen in Figure 3-1 that the task of creating and managing this cell library quickly becomes enormous as the number of allowed series transistors increases.

		Number of Serial PMOS Transistors				
		1	2	3	4	5
Number of Serial NMOS Transistors	1	1	2	3	4	5
	2	2	7	18	42	90
	3	3	18	87	396	1677
	4	4	42	396	3503	28435
	5	5	90	167	28435	425803

Figure 3-1. Gate combinations possible with series NMOS & PMOS transistors.

A tool which could synthesize from logic directly to a transistor netlist would eliminate the need to redesign a cell library every time the production technology process is updated. This would result in greater efficiency in silicon usage as we are no longer placing cells and are directly placing transistors on silicon with complex gates, reducing fabrication costs, engineering costs, and significantly decreasing time-to-market of the product.

3.1 Previous Studies in Library-Free Mapping

The concept of library-free technology mapping has been discussed and debated for nearly a decade in many research papers [5], [19–23]. Numerous approaches to solving the problem have been proposed, which will be discussed and evaluated in this Section.

In [5] a straightforward approach to complex gate generation from Boolean equations was proposed, in which every time a set of literals of the equation was a logic AND, the NMOS networks were associated in series, and PMOS in parallel, with the opposite for a logic OR. An Inverter is then inserted after the output of the complex gate to compensate for the natural negation of CMOS circuits.

Reis proposed an algorithm to cover trees in [19] by dynamically collapsing NAND/NOR trees of gates from the root-downwards so long as the newly generated gate does not violate a globally set maximum number of serial NMOS and PMOS transistors.

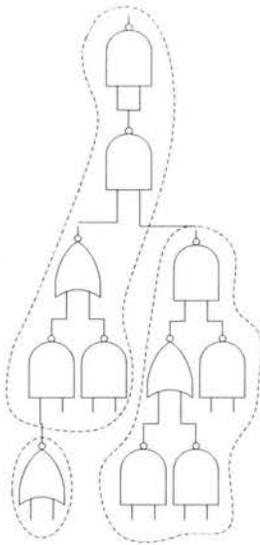


Figure 3-2. Possible Cover

As this covering algorithm began collapsing gates at the root, the method will promote complex gates to form near the root of the tree, which is a charged node, as illustrated in Figure 3-2. The output node of a tree may be sized larger than its child nodes which enlarges any complex gate at this node, leaving smaller gates clustered near the input nodes. Correia and Ries suggested an alternative method to collapse base functions into complex gates [21]. This method takes a bottom-up approach at gate collapsing, which encourages complex gates to be formed near the inputs of the tree rather than the output, as illustrated in Figure 3-3. While this approach is an improvement, the quality of the resulting mapped circuit still highly depends on the initial decomposition.

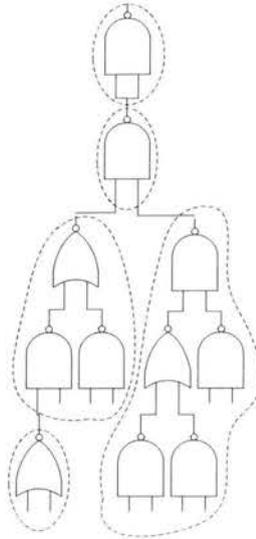


Figure 3-3. Alternate Cover

Jiang et al. proposed the odd-transistor-replacement (OTR) method to dynamically generate complex gates from a decomposed logic network [20]. It is a two-step process, collapsing every three consecutive gates into one complex gate. The OTR method works by selecting three levels of logic and combining them into one complex gate. This process is outlined in the example below.

Example – Odd-Transistor Replacement Method

Figure 3-4 illustrates a sample logic structure which will be collapsed into one complex gate using the OTR method. The transistor structure of this simple circuit before transformation is visible in Figure 3-5.

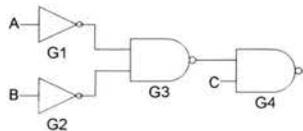


Figure 3-4. Simple Circuit for OTR Example

The first step in the OTR gate collapsing method is to replace NMOS (PMOS) transistor structures from the first level (second level) with the PMOS (NMOS) transistor structures from the second level (first level). Figure 3-6 illustrates this transformation between gates G1, G2 and G3, into G3’.

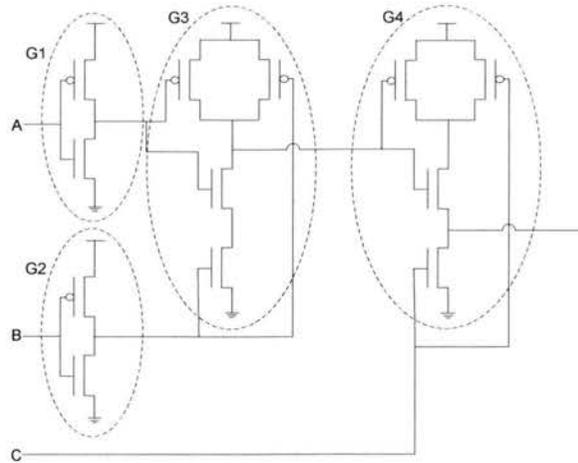


Figure 3-5. Transistor-level representation of Figure 3-4

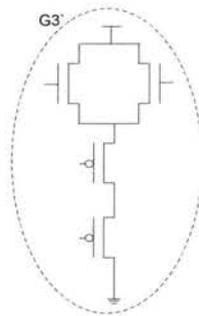


Figure 3-6. Intermediate transformation of OTR method

The second step in the OTR method is to perform the exact same procedure, replacing NMOS and PMOS structures between the newly generated intermediate gate, G3' with those in the last stage. The resulting complex gate can be seen in Figure 3-7.

While these varying techniques will serve to dynamically generate complex gates out of a directed acyclic graph (DAG) partitioned into trees, they may not necessarily produce the best covering which optimizes area, power consumption and delay as they simply extract the complex gates from the structure of the circuit. Marques et. al proposed a modified wavefront [24] covering algorithm which dynamically generates complex gates within the width of the wave, which propagates from inputs to outputs across the entire DAG without partitioning in to trees [22]. This method has been shown to be effective at

reducing delay in a circuit by duplicating logic between fanout-free regions, with comparison to traditional partitioned tree-based mapping. Although this duplication in logic reduces delay, it comes at an increase in area, and power consumption.

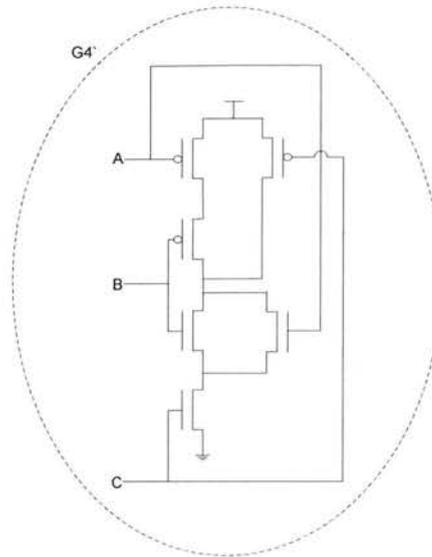


Figure 3-7. Final transformation of OTR method: Resultant complex gate

3.2 Summary

As the limit of how small we can fabricate transistors is quickly being reached, alternative methods of design must be explored to enable continued performance improvements on existing technology processes. Library-free synthesis attempts to improve the architecture of a circuit by covering it with complex gates which are typically not found in current standard cell libraries. This chapter has overviewed a number of current library-free covering algorithms, the majority of which are highly dependent on the structure of the initial decomposition.

The following chapter will propose new matching and covering algorithms for improved library free synthesis which are not tied to the initial decomposition of the circuit.

4. Proposed Synthesis Algorithms

A combination of structural tree-traversing and Boolean matching was employed to optimally match and cover the partitioned subject graphs. The use of Boolean matching enables the resulting mapped circuit to be independent on the structure of the initial decomposition, as complex gates are not directly extracted from the decomposed tree. Size-less gate-transistor structures are pre-generated, defined by the maximum allowed serial PMOS and NMOS transistors, and stored in a hash table in RAM. A transistor logic solver was developed and used to determine the truth table values for these gates being generated. Canonical representations of the generated gates are then created to check for *NPN*-equivalence for functions with seven inputs or less, and *P*-equivalence for functions with 25 or less inputs. The two canonical forms are then saved as a string of bits and indexed in two hash tables in the virtual library.

A Boolean logic solver was developed and used to determine the truth table for the partitioned subject graphs. These truth tables are used to generate canonical representations of the subject graph to be searched against the library hash table for a match.

As the generated library will be large (425,803 gates with 5 series transistors), speed and practicality of the matching and covering algorithms was of paramount concern.

4.1 Logical Effort

The method of logical effort, first publicized by Ivan Sutherland, Robert Sproull, and David Harris in [25] is a simple, quick method to estimate the delay in a CMOS circuit. Logical effort attempts to model delays in a circuit as being caused by capacitive load that the gate drives, and by its own topology. Logical effort gives the minimum delay (D) of a given circuit path as in (4-1). The delay in (4-1) is normalized to τ , where τ is the delay of a unit Inverter driving another Inverter without parasitic effects,

$$D = N(GHB)^{1/N} + P \quad (4-1)$$

where G is the path logical effort, H is the electrical effort, B is branch effort, N is the number of stages on the path, and P is parasitic delay. The electrical effort, H , is determined by the loading of the path in question, and is controlled by the environment surrounding the circuit. The logical effort, G , is controlled by the architecture of the covering gates, and is simple to modify – minimizing G minimizes delay. Logical effort of a gate is obtained by summing the PMOS and NMOS transistor input capacitances (C_{in}) for a given input, and dividing that by the input capacitance for a unit sized Inverter with identical rise and fall characteristics. The third component of delay, P , is introduced by internal parasitic capacitances within the covering gate, controlled by the number, and sizes of transistors with their drain connected to the output node of the gate. The branching effort, B , is the ratio of total capacitance being driven by the gate to the capacitance on the path of interest.

Logical effort does not yield precise values for delay, and is not intended to replace traditional simulation tools. It provides a method to compare two circuits to determine which architecture will yield the lowest delay. The delay in a multistage network can be approximated using (4-2) by summing the logical effort delays of each stage.

$$D = \sum f_i + \sum p_i = \sum g_i h_i + \sum p_i \quad (4-2)$$

One can observe that in order to minimize delay in a given circuit, it is necessary to minimize the sum of the product of electrical effort (H) and logical effort (G), and the parasitic delays (P). Minimizing G and P is simple, and is controlled solely by gate selection. As an example, compare the Inverter, NAND and NOR gates below, which are all sized for equal rise and fall times, given the assumption that the electron mobility in the PMOS transistor is half that of the mobility in the NMOS transistor.

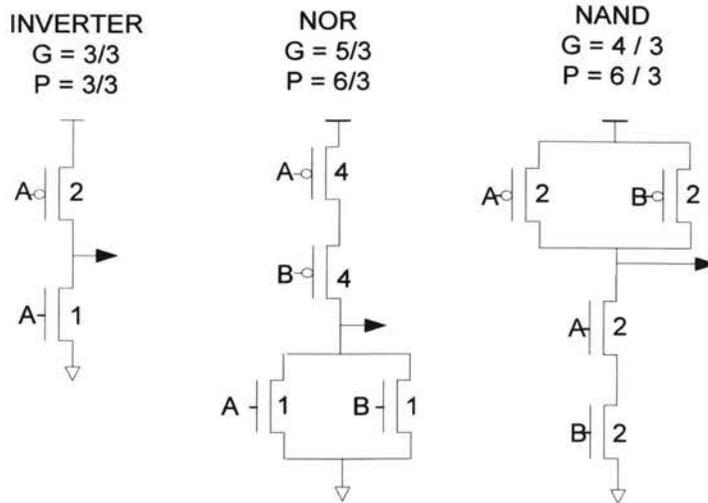


Figure 4-1. Logical Effort of Simple Gates

The Inverter is the simplest CMOS logic gate, and is used as a basis for comparison for every other gate. The logical effort of the Inverter is 1. As the NAND and NOR functions are symmetric, both inputs to these functions will share the same logical effort value. Looking at the NAND gate, the sum of the PMOS and NMOS input capacitances is 4 thus the logical effort for this gate is $4/3$. Similarly, the logical effort of the NOR gate may be found to be $5/3$. If one were to design a circuit, and have the choice to implement the circuit with NAND gates or NOR gates, by logical effort, the implementation utilizing NAND gates will have a lower delay than that using NORs.

This chapter will propose a novel method to obtain the most optimal circuit covering using logical effort as the cost being minimized.

4.1.1 Estimating Gate Area with Input Capacitance

Transistor count alone is not an effective measure of gate area. The area of a gate depends on the number of transistors in the gate, and the size of those transistors. The size of a transistor depends on the driving strength, and complexity of the gate. In logical effort, the complexity of a gate is modeled by G .

Consider an Inverter and a 3-input NOR, as illustrated in Figure 4-2. Also assume the electron mobility of an NMOS device is twice that of PMOS. The average size of a transistor in the NOR gate is 2.33 times larger than that of a transistor in an Inverter of equal driving strength. Thus, transistor count alone is not a valid means to estimate total circuit area. Rather, total circuit area may be estimated by adding the transistor widths of the template gates for every gate in the circuit. Using this method, in Figure 4-2 the area of the Inverter can be seen to be 3, and the NOR to be 21.

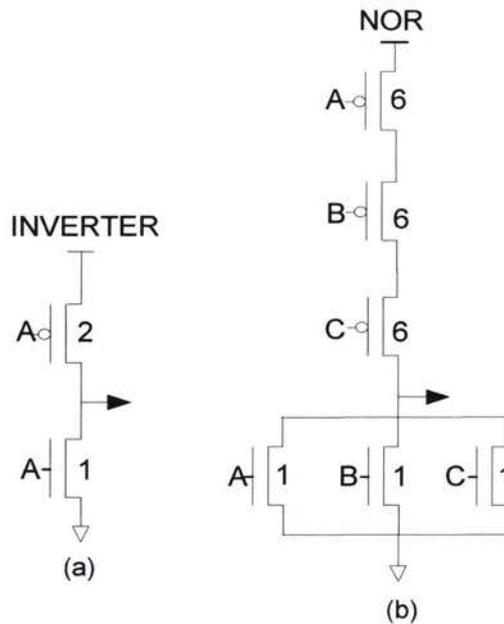


Figure 4-2. Example of Transistor Size Variation in Gates
(a) Inverter; (b) NOR

4.2 Fast Boolean Matching of Complex Gates

As the matching algorithm will be executed many times, it is important to be able to generate the canonical form of a given function as fast as possible. Every generated gate will be indexed in the library by two canonical forms, as outlined in Sections 2.4.5.1 and 2.4.5.2 of this thesis. The canonical form proposed by Ciric and Sechen, as overviewed in Section 2.4.5.2 will be generated from the subject graph first, and a *PN*-equivalence test will first be made for functions up to 25 inputs.

If a matching library cell is not found initially, and the subject graph being matched has seven or fewer inputs, the more computationally expensive *NPN*-equivalence test as proposed by Debnath and Saso, and outlined in Section 2.4.5.1 of this thesis will be used. If a match is not found at this point, there does not exist a cell in the library which can cover the given subject graph in its entirety.

4.3 Complex Gate Generation and Indexing

A technology-independent library of complex gate structures is pre-generated for matching against a subject graph prior to synthesizing the input netlist. Using the inputs (S_p, S_n), which denote the number of serial PMOS and NMOS transistors, the library generation tool will generate all possible combinations of gates. A CMOS transistor gate logic solver was developed to obtain the truth table for these newly created complex gates. Once a truth table is available, the *PN*- and *NPN*-representatives of the function are derived by the method described in Sections 2.4.5.1 and 2.4.5.2 of this thesis. These representatives will be stored as a string of bytes, along with the order of the inputs, relative to the original function inputs, input or output negations, and the transistor structure, as described by the library cell data structure in Figure 4-3.

As gates are generated, they will be subsequently added to the library, as defined in Figure 4-4. After gate generation, the P canonical representation is generated, and checked for existence in the Library's PCanonicalIndex. The PCanonicalIndex is a hash table with the string value of the canonical representation as the key. If no match is found in the table, the NP canonical representation is generated, and both will be indexed, along with the generated gate structure. The NP canonical representation is considerably more expensive than the P canonical representation, in terms of computation time, which is the reason why it is only generated once it has been verified that the cell in question is not equivalent to an existing cell in the library.

```

class LibraryCell
{
    public List<Transistor> Inputs; // transistors in PMOS network
    public double[] LE; // logical effort per input
    public string NPCanonical = "";
    public string PCanonical = "";
    public int[] POrder; // input ordering of P-canonical
    public int[] NPOrder; // input ordering of NP-canonical
    public bool[] TruthTable;
    public bool[] NMOSTruthTable;
    public string ID; // unique id in memory
    public double P; // parasitic delay of gate
    public List<Transistor> VDD; // root which transistors are
                                // connected to

    public List<Transistor> OUTPUT_NMOS; // root of NMOS network
    public TransistorNode OUTPUT; // which all DRAIN nodes lead to
    public TransistorNode GROUND; // which SOURCE nodes connect to
                                // in NMOS network

    public InputChain inputchain; // used to solve truth table
                                // of gate

    public bool NMOSfound = false;
    public LibraryCell NMOSNetwork; // matching NMOS network
    public bool InvertedOutput = false;
}

```

Figure 4-3. Library Cell Data Structure

```

class Library
{
    public List<LibraryCell> Cells;
    public Hashtable NPCanonicalIndex;
    public Hashtable[] PCanonicalIndex;
    public Hashtable TTIndex;
}

```

Figure 4-4. Library Data Structure

4.3.1 Complex Gate Library Population

In order to determine the most optimal covering for a given subject graph, a fully populated library of complex gates must be available to match against. The library generation module takes two integer inputs, the maximum number of allowed serial PMOS and NMOS transistors and uses these values to build an indexed library of complex gates.

The first step in the process is to generate a two-dimensional matrix with the given dimensions. In each cell of the matrix, a TransistorContainer object is instantiated. This object is used to traverse the matrix, and extract many combinations of complex gates available. Each TransistorContainer contains a Transistor object, and a source and drain which contain virtual links to other TransistorContainer elements in the matrix.

Directional links are assigned between the source and drain nodes of these TransistorContainer objects, following the pattern illustrated in Figure 4-5. These links do not represent actual electrical connections between transistors in a gate, but rather *possible* connections to explore which would result in a unique gate being created by following a set of rules.

Complex gates are generated recursively, beginning from the root Transistor Container located at position (0,0) in the matrix illustrated in Figure 4-5. Links are explored one at a time, beginning with those from the drain of the TransistorContainer, taking “skip” wrap around links first (the links visible on the left of certain transistors in Figure 4-5). For every link followed, the generation function is recursively called, with the *root* reference pointing to the current TransistorContainer in scope. After following a link, unless the next skip link is to be taken, the new root TransistorContainer is “enabled”. Once all outgoing links from the TransistorContainer’s Drain node have been explored, the TransistorContainer will be disabled, and the recursion will return the function which called it. Only vertical and skip links may visit a disabled TransistorContainer – all others may only visit a node if it has been previously enabled. When a link is explored which points to the output of the function, the transistor structure up to this point is extracted, and indexed in the library. After indexing, the root is set to the next transistor with its source connected to VDD, and the function is called recursively again.

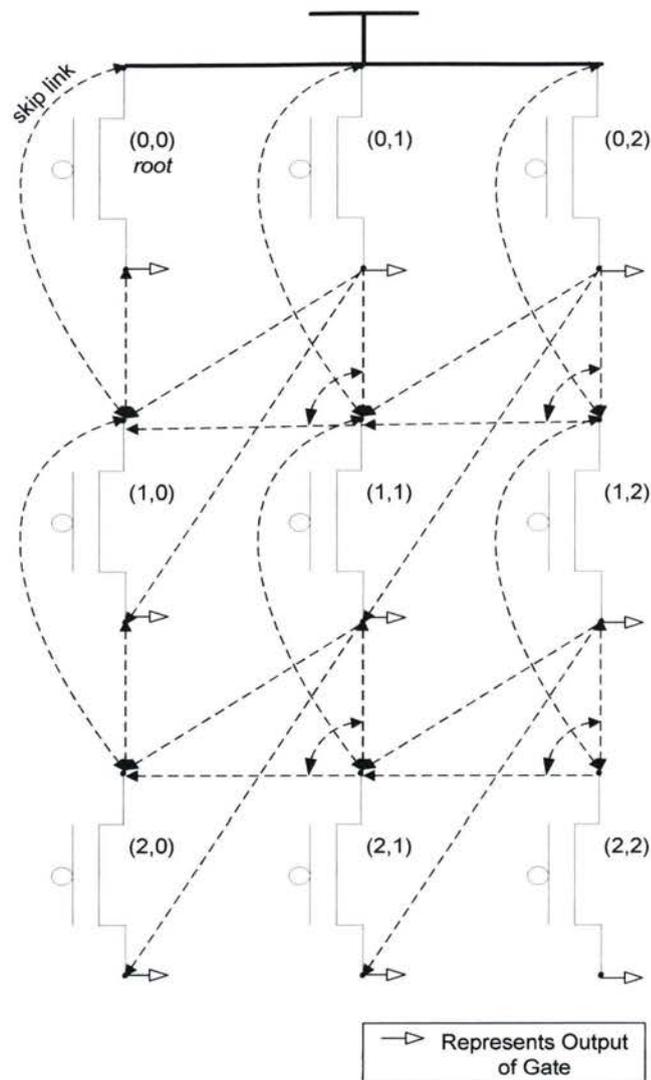


Figure 4-5. 3x3 Generation Matrix

Example – Generation of complex gates with two serial PMOS, NMOS transistors.

The generation function will begin with the Source of the TransistorContainer (TC) located at (0,0) in Figure 4-6. The first link to be explored is the skip link, bringing us to the Source of the TC located at (1,0). As we are visiting a disabled TC, it will now be enabled. There is one outbound link from the Drain of this TC to the output. As we have reached an output node, the transistor structure up to this point will be extracted, yielding a gate with one PMOS transistor, the Inverter. After extraction, we move to the next TC connected to VDD, located at (0,1).

At the Source of the TC at (0,1), the skip link is explored first, bringing us to the Source of the TC at (1,1). As we are visiting a disabled TC, it will now be enabled. There is one outbound link from the Drain of this TC to the output. As we have reached an output node, the transistor structure up to this point will be extracted, yielding a gate with two parallel PMOS transistors, the NAND gate. At this point, as there are no more transistors attached to VDD to explore, the TC at (1,1) is disabled, and the recursion steps back to the Source of the TC at (0,1). This TC is now enabled, and the downwards vertical link from its Drain is explored, bringing us to the TC at (1,1), which is then enabled. There is one outbound link from the Drain of this TC to the output. As we have reached an output node, the transistor structure up to this point will be extracted, yielding a gate with the function $f = \overline{A + BC}$. This process of following links and extraction at outputs follows until all possibilities have been exhausted, and the function returns to the original *root* TC at (0,0).

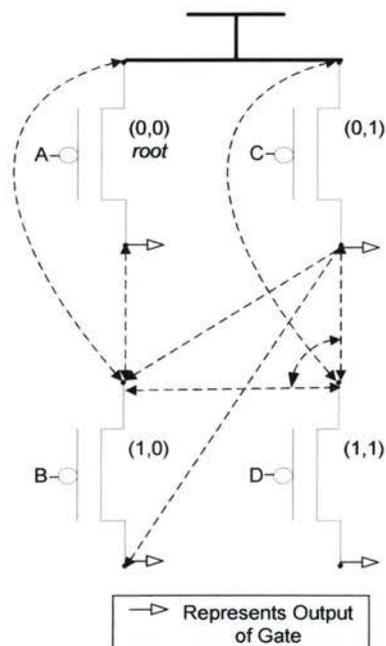


Figure 4-6. 2x2 Generation Matrix

Figure 4-7 illustrates the gates which will be generated, and the order of their generation (A – N) with the input parameters (2,2) which represent the number of serial PMOS and NMOS transistors which are allowed. It should be noted that gates B, C, and G are equivalent. Gates C, and G will not be added to the library once their canonical values are

calculated and it is found that an equivalent, B, is already in the library. Similarly, D, E, I, J and N are equivalents; only D will be indexed in the library.

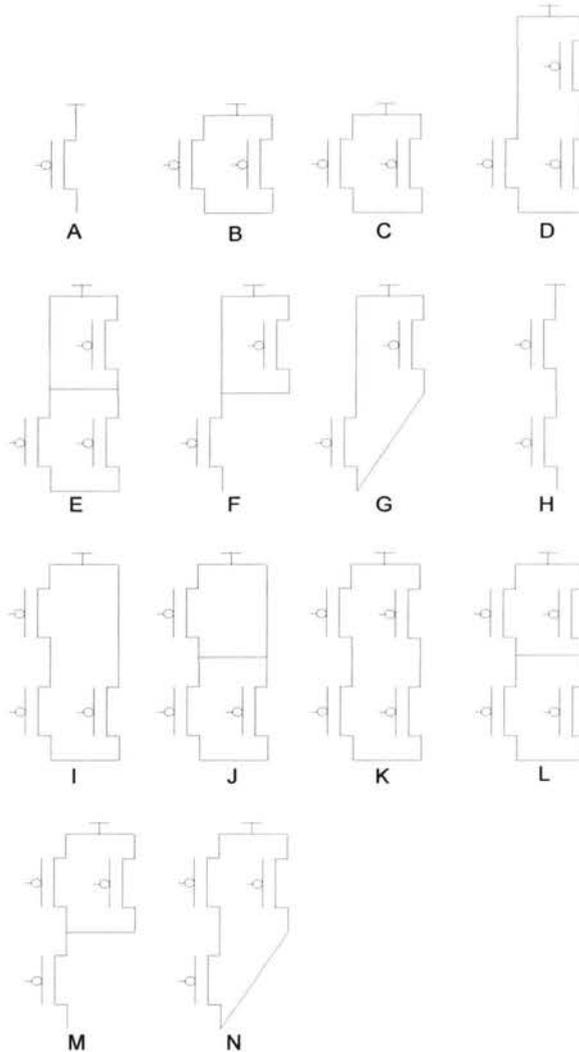


Figure 4-7. Gates Extracted from 2x2 Generation Matrix

The class descriptions for the TransistorContainer, Transistor, and TransistorNode classes can be seen in Figure 4-8, Figure 4-9 and Figure 4-10, respectfully. The TransistorContainer contains one Transistor, an enabled flag, and “links” which are represented as memory pointers to other TransistorContainer objects. The Transistor class contains a Boolean gate variable which is used for truth table generation, type definition which is either PMOS or NMOS, and node variables for the source and drain. These

nodes are used to represent an electrical link to another transistor or the output of the gate, and are managed by the TransistorNode class. The hierarchy of these classes is illustrated in Figure 4-11.

```
public class TransistorContainer
{
    public Transistor mos;
    public string ID;

    public bool enabled = false;
    public bool isOUTPUT = false;
    public bool isVDD = false;

    public List<TransistorContainer> drain;
    public TransistorContainer drainLeftLink;
    public TransistorContainer drainLeftTop;
    public TransistorContainer drainLeftBottom;
    public TransistorContainer drainBottom;
    public TransistorContainer drainBottomAround;

    public List<TransistorContainer> source;

    public TransistorContainer nextTranonVDD;
}
```

Figure 4-8. Transistor Container Data Structure

```
public class Transistor
{
    public TransistorType type;
    public TransistorNode source;
    public TransistorNode drain;
    public string ID;
    public bool gate;
    public bool INVERTED_INPUT = false;
    public bool ON_OUTPUT = false;
    public float skewwidth = 0;
    public bool skip = false;
    public int XPosition; //used for drawing
    public int YPosition; //used for drawing
}
```

Figure 4-9. Transistor Data Structure

```

public class TransistorNode
{
    public Transistor parentTransistor;
    public TransistorNodeType type;
    public List<TransistorNode> externalLink;
    public List<TransistorNode> externalLinkBACK;
    public bool value;
    public string ID;
    public bool recursiveVisit = false;
}

```

Figure 4-10. Transistor Node Data Structure

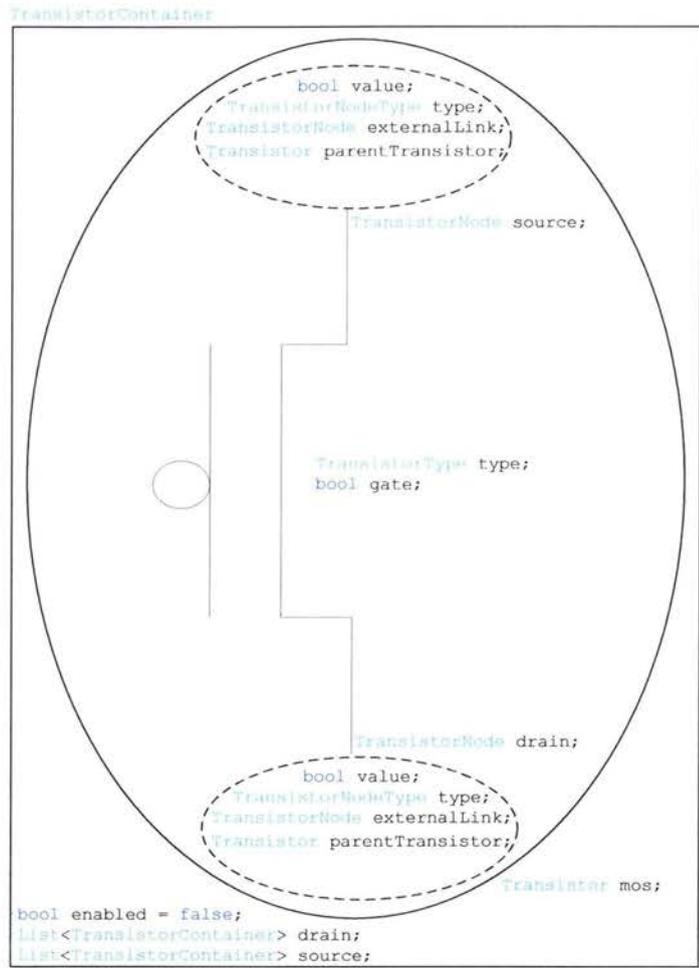


Figure 4-11. Hierarchy of the Transistor classes.
Dashed lines represent class encapsulation borders

4.3.2 Permutations of Order and Complementation of Inputs

As the canonical generation algorithms which are being employed permute the order and complementation of the inputs, it is necessary to map the inputs of the newly generated gate with those in the P and NP representation. For every gate generated a permutation array will be constructed, with the length equal to the number of inputs in the gate.

4.3.2.1 Input Tracking in P-Canonical Generation

The P-canonical generation algorithm as described in Section 2.3.2.5 permutes the order of the inputs by rotating columns in order to achieve the lowest weight minterm table. A one-dimensional rotation array is instantiated with the length equal to the number of inputs in the function. The table is initially filled with the integers 0 – (n-1), where n is the number of inputs, which can be directly used as the index to the array Inputs of the library cell as in Figure 4-3. As columns are rotated, the indices of the input order array are also switched. Once the P-canonical form is generated, the rotation array is copied to the POrder variable of the library cell, as in Figure 4-3.

4.3.2.2 Input Tracking in NP-Canonical Generation

The NP-canonical generation algorithm as described in Section 2.3.2.5 permutes the order of the inputs as well as complements these inputs in order to find the NP-canonical value. The canonical value is obtained through the use of pre-calculated *link tables*, which are generated in the initialization process of the synthesis tool for functions with up to 7 inputs.

Link Table Generation

In order to be able to quickly obtain the NP-canonical form of a function, look-up tables are pre-calculated in the initialization phase of the synthesis tool. These tables are created by calculating minterm positions based on a given permutation of the input variables. A reference truth table is generated, and each row is assigned a unique identifier, as in Figure 4-12. The order of the inputs to the function are then permuted by recursively calling the permutation function `GenerateInputSequence` in Figure 4-13. The available inputs are initially stored in an array of integer IDs, with each ID representing

an index to the `Inputs` list in Figure 4-3. Each ID in the array is visited, and added to a linked list of inputs. As an ID is added to the linked list, it is removed from the array of available inputs, and the recursive function is called again, being passed the linked list and the updated array. This process continues until the array of available inputs is empty at which time the linked list contains the selected input ordering. Similarly, the combinations of input inversions possible with the given input ordering is generated by recursively calling the `invertInputs` function in Figure 4-14, which inverts an input by negating the integer input ID value. The linked list is traversed, calling the function again with the child of the input node as the input. This process continues until the end of the linked list is reached, and the list is extracted to the link table through the `addInputColumn` function. At this point, every combination of complementation of the inputs is generated for the given input ordering.

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	c_0
0	0	1	c_1
0	1	0	c_2
0	1	1	c_3
1	0	0	c_4
1	0	1	c_5
1	1	0	c_6
1	1	1	c_7

Figure 4-12. Reference Truth Table for NP-Canonical Generation

Given an input ordering, minterm positions can be calculated using the reference truth table by calculating the position a minterm would appear in this table. For example, the minterm positions for the input ordering 3,2,1 are illustrated in Figure 4-15. They can be calculated as follows: for the inputs (0,0,0), regardless of the ordering, this minterm can be found at position c_0 in the reference table. The inputs (0,0,1) are translated to (1,0,0) using the input ordering (3,2,1). This minterm can be found at position c_4 in the reference table. Similarly, the inputs (0,1,0) are mapped to c_2 , and (0,1,1) is mapped to c_6 in the reference table. This process continues until all of the minterm positions have been calculated for a given permutation of the input variables. The function `addInputColumn` takes the input linked list, and writes the calculated minterm positions to the link table for NP-canonical generation.

```

for (int inputCounter = 0; inputCounter < numInputs; inputCounter++)
{
    int chosenStart = availInputs[inputCounter];
    int newCounter = 0;
    for (int innerloop = 0; innerloop < numInputs; innerloop++)
    {
        if (availInputs[innerloop] != chosenStart)
        {
            updatedInputs[newCounter] = availInputs[innerloop];
            newCounter++;
        }
    }

    inputNode newNode = new inputNode();
    newNode.id = chosenStart;
    inputNode root = newNode;
    GenerateInputSequence(availInputs, newNode, root);
}

GenerateInputSequence(int[] availInputs, inputNode node, inputNode root)
{
    if (availInputs.Length == 0)
    {
        InvertInputs(root)
    }
    else
    {
        inputNode newNode = new inputNode();
        node.child = newNode;
        foreach (int input_id in availInputs)
        {
            newNode.id = input_id;
            int[] newAvail = new int[availInputs.Length];
            availInputs.CopyTo(newAvail, 0);
            int[] UpdatedAvail = new int[availInputs.Length - 1];
            int counter = 0;
            foreach (int to_remove in availInputs)
            {
                if (to_remove != input_id)
                {
                    UpdatedAvail[counter] = to_remove;
                    counter++;
                }
            }

            GenerateInputSequence(UpdatedAvail, newNode, root);
        }
        node.child = null;
    }
}
}

```

Figure 4-13. Input Order Permutation Function

```

private void invertInputs(inputNode n) {
    if (n.child != null)
    {
        invertInputs(n.child);
        n.id = n.id * -1;
        invertInputs(n.child);
    }
    else
    {
        addInputColumn(n);
        n.id = n.id * -1;
        addInputColumn(n);
    }
}

```

Figure 4-14. Input Inversion Algorithm

x_1	x_2	x_3	$f(x_3, x_2, x_1)$
0	0	0	c_0
0	0	1	c_4
0	1	0	c_2
0	1	1	c_6
1	0	0	c_1
1	0	1	c_5
1	1	0	c_3
1	1	1	c_7

Figure 4-15. Sample Input Permutation & Minterm Positions

4.3.3 CMOS Transistor Logic Solver

In order to calculate the canonical representation of a newly generated library cell, it is necessary to derive the truth table of the function performed by the gate. As is shown in Figure 4-3, a library cell contains a list of all transistors whose source node is attached to VDD. The cell also contains an output node, which the bottom-most transistors' drains are connected to, and is also where we will be sampling output values of the function. Transistors are aligned in a linear linked list, and are recursively traversed, exciting every possible gate input combination – 2^n in all, where n is the number of inputs (transistors) to the function (gate). Once a particular input pattern has been assigned to the gates of the transistors in the complex gate, the output is solved recursively, from VDD to output.

Initially, the source values of every node attached to VDD are set to TRUE. Following each node recursively downwards through the drain links, the drain values are set to TRUE only if the value of the gate is FALSE (as we are working with PMOS transistors). The evaluation method works on the basic function:

$$\text{Drain} = \text{Drain OR Source AND NOT}(\text{Gate})$$

Once every source node attached to VDD is explored, the Boolean value of the output node may be read, and inserted into a truth table row corresponding to the input sequence tested.

4.3.4 CMOS Gate Skewing

It is often desired for the output rise and fall times of a CMOS gate to be equal; as such, the resistance between VDD and the output must be the same as that between the output and VSS in order to allow equal charge and discharge currents to and from the output load being driven. Figure 4-16 illustrates this requirement, representing the entire PMOS and NMOS networks with their equivalent resistance, R_{PMOS} and R_{NMOS} , respectively. In order for the rise and fall currents, I_{rise} and I_{fall} to be equal, the transistors in the PMOS and NMOS networks must be sized such that $R_{\text{PMOS}} = R_{\text{NMOS}}$.

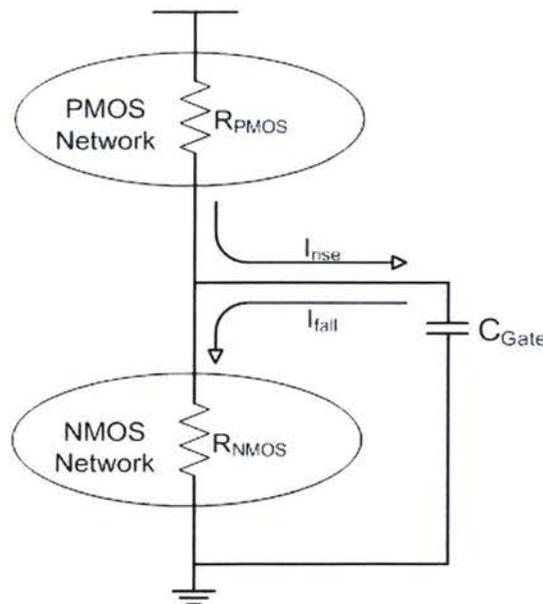


Figure 4-16. Sizing for equal rise & fall times in CMOS gates

In order to size a gate for equal rise and fall times, we analyze the PMOS and NMOS networks independently. In the worst case, only one path in each network will be active at a time. As such, all paths in each network must be sized so that they have the same resistance. Taking the PMOS network first, each path from VDD to output is traversed, and added to a list of paths. This list is sorted descending in the order of the number of transistors on the path.

Taking each path at a time, if there are no previously sized transistors on the path, every transistor width is equal to the number of serial transistors on the path. If there are previously sized transistors on the path, transistor widths are assigned according to Equation 4-3,

$$W = \left(\frac{Weq}{Weq - 1} \right) S \quad (4-3)$$

where S is the number of serial transistors on the path without a defined size, and Weq is the equivalent width of the sized transistors on the path, as defined by Equation 4-4,

$$Weq = \frac{1}{\sum_n \frac{1}{w_n}} \quad (4-4)$$

The W value from Equation 4-3 is then assigned as the width for every unsized transistor in the current path.

Example – Sizing of complex gate for equal path resistance from VDD to output

The gate generation algorithm as described in Section 4.3.1 will produce a transistor structure similar to that seen in Figure 4-17, with vertical and horizontal directional links. Paths are traced beginning from VDD to output, from the left most transistor connected to VDD, to the rightmost. Tracing the paths in Figure 4-17, we retrieve the set of paths {ABCD, ABG, EFCD, EFG, HIJ, HL, KIJ, KL}. This list of paths is then sorted in descending order by the number of transistors on the path, yielding a new set of paths: {ABCD, EFCD, ABC, EFG, HIJ, KIJ, HL, KL}

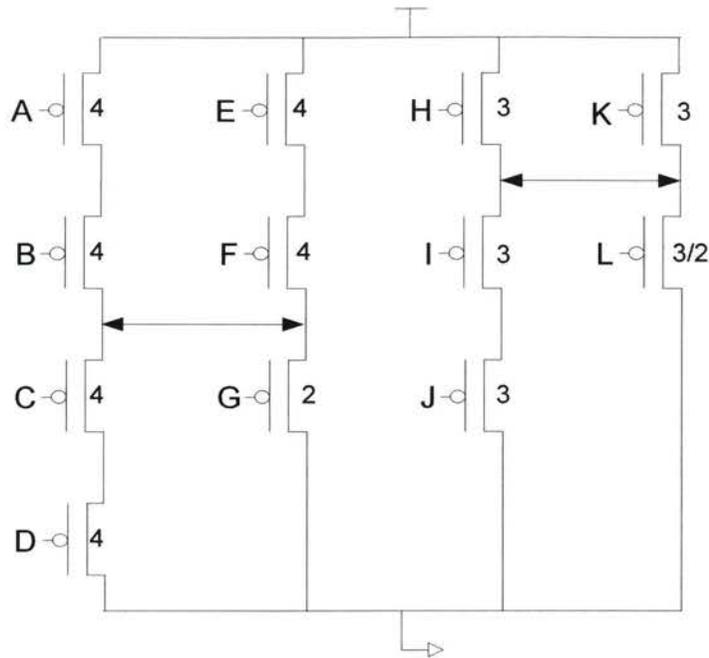


Figure 4-17. Gate generated from 4x4 Generation Matrix

First we traverse the path ABCD. As none of these transistors have been previously sized, their widths are set to 4, the number of transistors on the path. The path EFCD is analyzed next. By (4-4), the equivalent resistance of the sized transistors is calculated:

$$W_{eq} = \frac{1}{1/4 + 1/4} = \frac{4}{2}. \text{ Using (4-3), the widths of transistors E and F, the only unsized transistors along the path EFCD to be } W = 2\left(\frac{4}{2}\right) = 4. \text{ Solving the third path, ABG, equates } W_{eq} \text{ to 2. As there is one unsized transistor on this path, } S = 1. \text{ Thus, the width of G is calculated to be } W = 1(2) = 2.$$

Following in this manner, all of the remaining transistors in the gate may be ratioed such that every path shares the same resistance.

It is common knowledge in CMOS digital logic design that PMOS and NMOS transistors often do not share the same electron mobility (μ) and as such will not have the same resistance for the same width of transistor. The electron mobility of the PMOS transistor (μ_p), depending on technology process, is often quite different than that of its NMOS (μ_n)

counterpart. For this work, we consider $\mu_n = 2\mu_p$ which is a very common assumption in CMOS digital design. Thus in order to size for equal rise and fall times, the gate ratios are then multiplied by a skewing factor, which will double the widths of the PMOS transistors in order to counterbalance the lower electron mobility.

The generation algorithm as presented in Section 4.3.1 is not perfect, and at times will generate a gate which does not meet the required maximum number of allowed series transistors. Figure 4-18 illustrates a gate which will be generated by a 4x4 generation matrix. By studying the interconnections, it can be seen that a path HIFBCD from VDD to output exists which contains six series transistors. Clearly this is beyond the allowed maximum of four. As such, the skewing algorithm also serves as a filter – if a path is found which exceeds the maximum allowed series transistors, the gate is discarded and not indexed in the library.

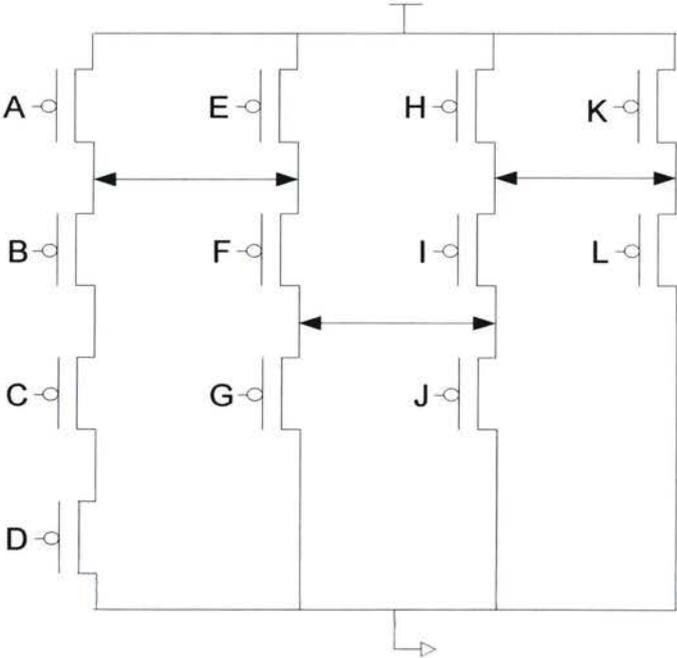


Figure 4-18. Gate Generated from 4x4 Matrix with 6 Series Transistors

4.3.5 NMOS Network Generation

The complex gate generation algorithm described in Section 4.3.1 only creates and solves the PMOS network, from VDD to the output. In order to synthesize an input HDL, the NMOS network for gates in the library must be available in order to obtain the logical effort value for every input. The gate generation matrix presented in Figure 4-5 will generate every possible gate given the restrictions on the maximum allowed series transistors. Although the algorithm described in Section 4.3.1 generates the PMOS network for a logic gate, if the transistors are replaced to be NMOS, the generated structure may be an exact match to a previously generated PMOS network. Combining these two will create a complete CMOS logic gate.

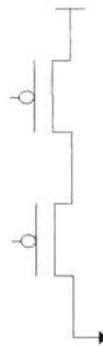


Figure 4-19. Generating NMOS Network from PMOS Structure

In order to match PMOS and NMOS networks to form a complete gate, Boolean truth tables must be independently generated for each network, and P-canonical matching is used to identify pairs. When a PMOS network is generated, the CMOS transistor logic solving algorithm as in Section 4.3.3 is employed to generate the truth table for the PMOS structure. From this truth table, the canonical forms are generated and the cell is added to the library. At the same time, the truth table for the network is generated as if the transistors were NMOS. This is accomplished by solving the truth table for the PMOS structure again, however inverting the inputs to the gate. Figure 4-19 illustrates a simple NOR gate extracted from the gate generation matrix in Figure 4-5. The truth table for this PMOS structure can be seen in Figure 4-20. In order to obtain the truth table for the NMOS network, the inputs are inverted before applying the transistor logic solving

algorithm from Section 4.3.3, while keeping the rows identical in the truth table. Once the function output is obtained via the logic solving algorithm, it is inverted and added to the truth table. Figure 4-21 illustrates the truth table obtained by inverting the inputs to the gates of the transistors in Figure 4-19 and inverting the output value. This can be verified simply by looking at the last row of Figure 4-21. Inverting both inputs, and applying a logic zero to both gates of the transistors in Figure 35, one can quickly see that the output will be a logic one.

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Figure 4-20. Truth Table for NOR Gate

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

Figure 4-21. Truth Table for NAND Gate

Once all library cells have been indexed, it is necessary to pair PMOS networks with NMOS networks in order to complete each CMOS gate in the library. Each library cell only containing a PMOS network is visited. The P-canonical representation of the calculated NMOS truth table is generated, and searched for in the library's P-canonical hash table. The matching cell in the P-canonical hash table is copied, and assigned as the NMOS network to the cell. At this point, it is necessary to re-arrange the order of the inputs in the NMOS network so that they align with those in the PMOS network, allowing the logical effort value of each input to the gate to be calculated.

When the P-canonical representation of a function is calculated, the canonical representation may have a different permutation of input variables to that of the truth table which it was calculated from. This ordering is saved in memory as the POrder, and is represented as a 1-dimensional array of integers, with length equal to the number of

inputs there are to the truth table. Each element in the array stores one integer which represents the index of the inputs to the original function. For example, the POrder {0,1,3,2} implies the first and second inputs remain fixed, and the third and fourth are switched.

As both the PMOS and NMOS networks share the same P-canonical representation, their POrders may be used to match their inputs together, using the P-canonical form as a common reference.

Example – Input matching between PMOS and NMOS networks

Consider the PMOS networks presented in Figure 4-22 and Figure 4-23. The network in Figure 4-23 is the NMOS counterpart of that in Figure 4-22, but with PMOS transistors, and vice versa. The truth table for the PMOS network in Figure 4-22 is available in Figure 4-24(a). The NMOS truth table is calculated by solving the network again, instead inverting its inputs, and the output, and is available in Figure 4-24(b). From this table, the P-canonical representation of the NMOS truth table is calculated, and is available in Figure 4-24(c). Figure 4-24(d) illustrates the calculated POrder for this canonical form, showing that no permutation is required to obtain the canonical form.

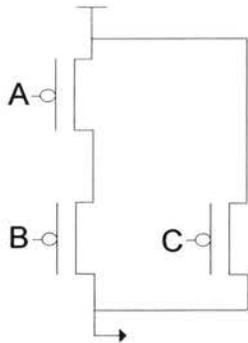


Figure 4-22. PMOS Network $f = \overline{(A+B)C}$

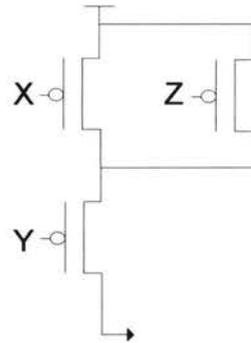


Figure 4-23. PMOS Network $f = \overline{(XZ+Y)}$

The truth table for the PMOS network in Figure 4-22 is illustrated in Figure 4-25(a). The P-canonical representation is calculated from this table, and is visible in Figure 4-25(b). It can be seen that Figure 4-24(c) and Figure 4-25(b) are identical, showing the two functions are a match. The POrders of each gate are used to match inputs. The POrder is iterated through, and inputs are assigned as follows:

GATE1.Inputs[POrder1[index]] = GATE2.Inputs[POrder2[index]]
 The index variable is iterated from 1 to N, where N is the number of inputs to the gate. In this case, POrder1[1] = 1; POrder2[1] = 1; Thus, A=X. POrder1[2] = 2; POrder2[2] = 3; Thus, B=Z. POrder1[3] = 3; POrder2[3] = 2; Thus, C=Y.

A	B	C	F	A	B	C	f_{NMOS}	A	B	C	{1, 2, 3}
0	0	0	1	0	0	0	1	0	1	1	
0	0	1	1	0	0	1	0	1	0	1	
0	1	0	1	0	1	0	1	(c)			
0	1	1	0	0	1	1	0				
1	0	0	1	1	0	0	1				
1	0	1	0	1	0	1	0				
1	1	0	1	1	1	0	0				
1	1	1	0	1	1	1	0				
(a)				(b)							

Figure 4-24. Generated Truth Tables & P-canonical form of Figure 38
 (a) truth table of Figure 38; (b) calculated NMOS network truth table; (c) NMOS P-canonical; (d) NMOS POrder

X	Y	Z	F	X	Z	Y	{1, 3, 2}
0	0	0	1	0	1	1	
0	0	1	1	1	0	1	
0	1	0	0	(b)			
0	1	1	0				
1	0	0	1				
1	0	1	0				
1	1	0	0				
1	1	1	0				
(a)							

Figure 4-25. Generated Truth Tables & P-canonical form of Figure 39
 (a) truth table of Figure 39; (b) PMOS P-canonical; (c) PMOS POrder

4.3.5.1 NMOS Generation – Dependency on Library Size

The gate-generation matrix in Figure 4-5 will generate all combinations of PMOS structures possible given the input library dimensions. If the library is not square, certain NMOS networks will not be generated. For example, if a library is generated with a maximum allowed 3 series NMOS and 2 series PMOS, NMOS networks will not be

found for PMOS networks which are more than 2 transistors wide. As such, for non-square libraries, a complimentary library with opposite dimensions is generated for searching for NMOS networks. Using the same example, a library with the dimensions of 2 series NMOS and 3 series PMOS is generated to locate appropriate NMOS networks.

4.4 Proposed Boolean Tree Matching

It will be necessary to determine the most optimal match for every node in the subject graph in order to obtain the best cover with respect to logical effort delay or area. The subject tree will be traversed recursively from root to leaf nodes, from left child to right child to find matching gates.

4.4.1 Sub-tree Extraction

At each node in the traversal, it is necessary to explore all possible matches from this point to its primary inputs, and all combinations of inputs leading up to the current node.

During the matching process, it is necessary to extract all sub-trees at a particular node in order to explore all possible covers. From the simple tree below, five unique sub-trees can be extracted, as illustrated in Figure 4-26.

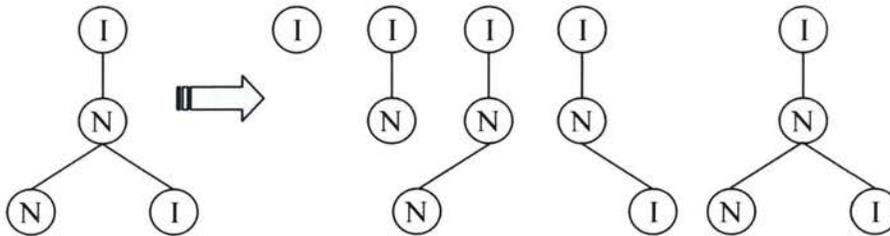
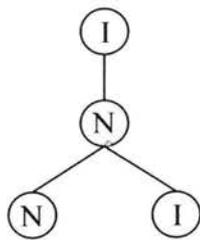


Figure 4-26. Extracted Sub-trees from Sample Graph

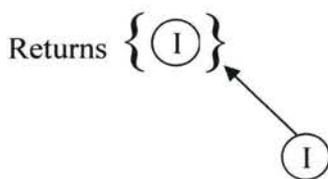
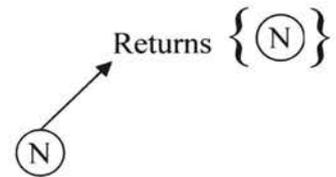
An algorithm has been developed which will perform this extraction, taking the root node of the extraction as its input, and its output being a set of all combinations of sub-trees. The extraction process works by recursively traversing the tree downwards, from left child to right child, and returning upwards the set of nodes { self; self + left child; self + right child; self + [left child X right child] }. This process is illustrated in the example below.

Example – Sub-tree extraction algorithm in operation



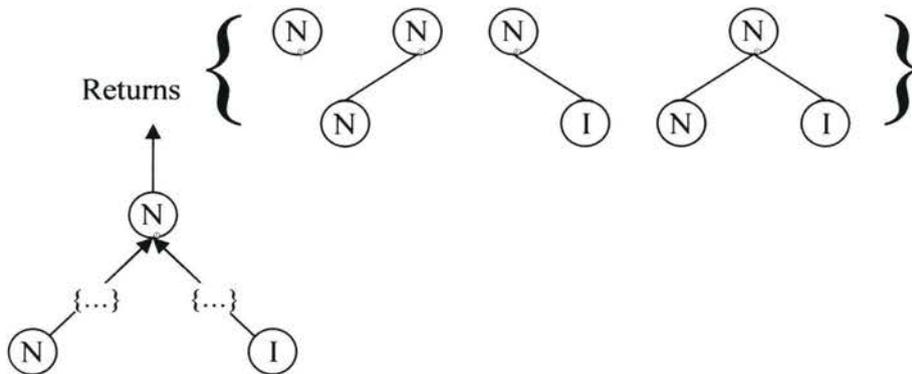
Traversing the tree illustrated to the immediate left (also shown in Figure 4-26), from left child to right, bottom up, returning the possible combinations of nodes at each step, all possible combinations of sub-trees can be obtained.

Starting from the leaf nodes, and passing upwards, the bottom NAND (“N”) node does not have any children, and as such can only pass itself forward.

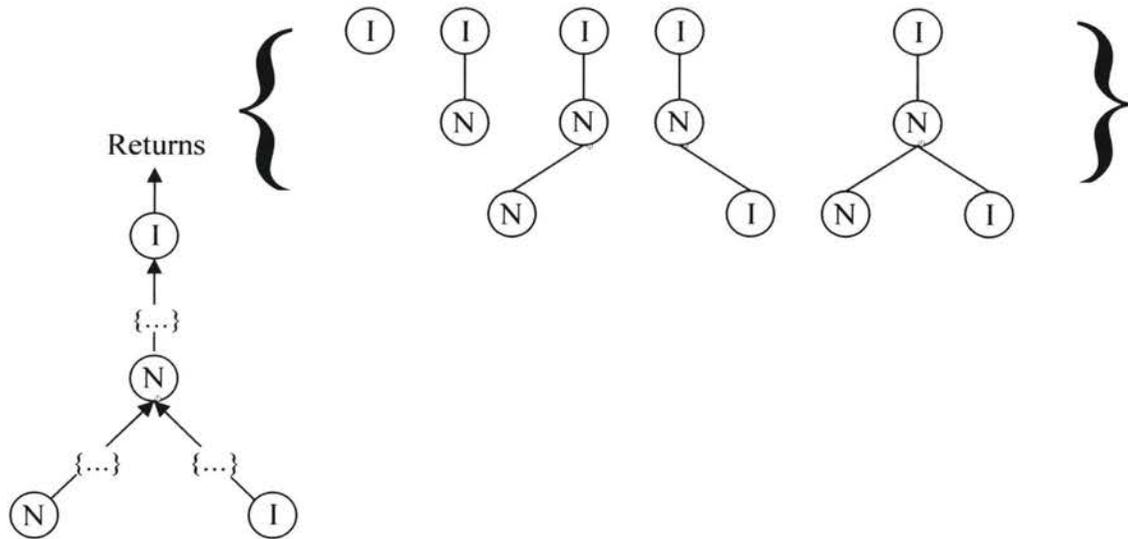


Similarly, the Inverter (“I”) node does not have any children, and passes only itself forward.

The middle-level NAND gate passes upwards the set of itself, itself + it’s left child (in this case the NAND gate), itself + it’s right child (in this case the Inverter), and itself + all combinations of it’s left and right children together. Since both children only returned one result each, the number of possible combinations of arrangements of it’s children is one.



Finally, the top-level Inverter gate passes upwards the set of itself, itself + it’s only child, as illustrated below. At this point, all possible sub-trees have been extracted from the root node of the input tree.



This procedure is detailed in pseudo code in Figure 4-27.

```

subtreeExtract (node n) {
  if n has 1 child
    return "set of" { self, self + subtreeExtract(n.child) }
  if n has 2 children {
    lc = subtreeExtract(n.leftChild)
    rc = subtreeExtract(n.rightChild)
    return "set of " { self, self + lc, self + rc, self + [lc X rc] }
  }
  return null
}

```

Figure 4-27. Proposed sub-tree extraction method

4.4.2 Matching Algorithm

In order to obtain all matches in a given subject tree, it is necessary to traverse the tree from root to leaf nodes, and explore all potential matches along the way. If a match is found, the node in the tree is marked with this match for later analysis in the covering step. This is accomplished by visiting each node in the subject graph, extracting all

subtrees from this node to the inputs of the partition, and iterating through each of these subtrees looking for a Boolean match with a cell in the library. This matching algorithm is detailed in pseudo code in Figure 4-28.

```
findMatch(Node n) {
    extractedTrees = extractSubtrees(n);

    foreach (Node ex in extractedTrees) {
        TT = GenerateTruthTable(ex);
        PNCanonical = GeneratePNCanonical(TT);

        if (Library.PNIndex.Contains(PNCanonical)) {
            ex.Matches.Add(PNCanonical);
        }
        else
            if (ex.numInputs < 8) {
                NPNCCanonical = GenerateNPNCCanonical(TT)
                if Library.NPNIndex.Contains(NPNCCanonical)
                    ex.Matches.Add(NPNCCanonical);
            }
        foreach (Node c in n.children)
            findMatch(c);
    }
}
```

Figure 4-28. Proposed Boolean tree matching algorithm

4.4.3 Boolean Logic Solver

The matching algorithm in Figure 4-28 requires that the truth table of the input extracted subtree be generated in order to calculate the canonical representations used in matching. Inputs to the tree are aligned in a linear linked list, and are recursively traversed; exciting every possible input combination – 2^n in all, where n is the number of inputs to the tree. Once a particular input pattern has been assigned to the inputs of the function, the output value is solved recursively, from the output node to the inputs.

At each node, the output is passed upwards based on the value at its inputs and the type of function. The simplified pseudo code for this function is presented in Figure 4-29.

```

bool booleanSolver (Node t)
{
    if (t.type = INV)
        return NOT(booleanSolver(t.child));
    if (t.type = NAND)
        return NOT(booleanSolver(leftChild) AND booleanSolver(rightChild));
    if (t.type = NOR)
        return NOT(booleanSolver(leftChild) OR booleanSolver(rightChild));
}

```

Figure 4-29. Recursive Boolean logic solver pseudo code

4.5 Proposed Covering Algorithm

Once all matches have been obtained using the method described in Figure 4-28, it is necessary to determine the most optimal covering in order to reduce area-delay consumption. A dynamic programming approach is taken to solve this minimization problem. Similar to the recursive tree-traversal algorithm implemented in Figure 4-28, the most optimal result will flow from the leaf nodes and arrive at the root node upon completion. The algorithm works by beginning at the root node of a subject graph, and iterating through each available match at this node. Each match iterated will be fixed in place at the current node being analyzed; the function will then be called recursively for each child of the match being fixed in place. Upon completion of the recursive call, the function will return the *best* match, and the *cost* of this match for each of its children, given the restriction of the match being fixed in place. This iteration process will continue until finally the leaf nodes of the subject tree are reached, where only one match may be found – the base function. The cost function being minimized may be calculated as logical effort delay, or total circuit area.

4.5.1 Minimizing Logical Effort Delay

The matching phase as described in Section 4.4 of this paper will generate a large set of matches for a given circuit, with an even larger set of possible coverings obtainable using

these matches. It is necessary to determine which set of matches should be employed to obtain the minimum critical path logical effort delay.

The critical input in a subject graph may be identified by the input with the highest path logical effort value, G , observed from the input towards the output of the circuit [26]. As defined by logical effort theory, path logical effort is the product of the logical effort of all inputs seen along the path being analyzed [25].

Example – Logical effort delay minimizing covering algorithm

The covering algorithm outlined above may be applied to the very simple circuit in Figure 4-30(a). The following example assumes that a library is available containing a two-input NAND, a two-input AND (NAND+Inverter), a two-input OR (NOR+Inverter), a three-input OR (NOR+Inverter), and an OR-AND-INVERT (OAI).

The first step of the covering algorithm looks at the root node, and lists the matches available for this node. At node 1, there are three matches, which are illustrated in Figure 4-30(b). These matches are a single NAND, an OAI containing nodes 1 and 2, and an OAI containing nodes 1 and 3. The covering algorithm works by iterating through each of these matches, determining which minimizes logical effort delay. First the NAND match is selected, and fixed in place. First the left input to this gate is visited – gate 2. As there is only one gate at this node, only the OR match is found. It is returned, along with its cost to gate 1. The delay of gate 2 by (4-2) is 5.66, taking account for the Inverter added to the NOR to make it an OR. Next, the right child of gate 1 is visited, gate 3 which has 2 potential matches, as seen in Figure 4-30(c).

First, the OR match is selected and locked into place. As each of the children of the OR match only contain one match themselves, they are visited, and locked in place in a similar manner, passing their delays upwards to node 3. This is visible in Figure 4-30(d). The critical delay of node 3 passes through node 5 to primary inputs, giving a delay of 11.32.

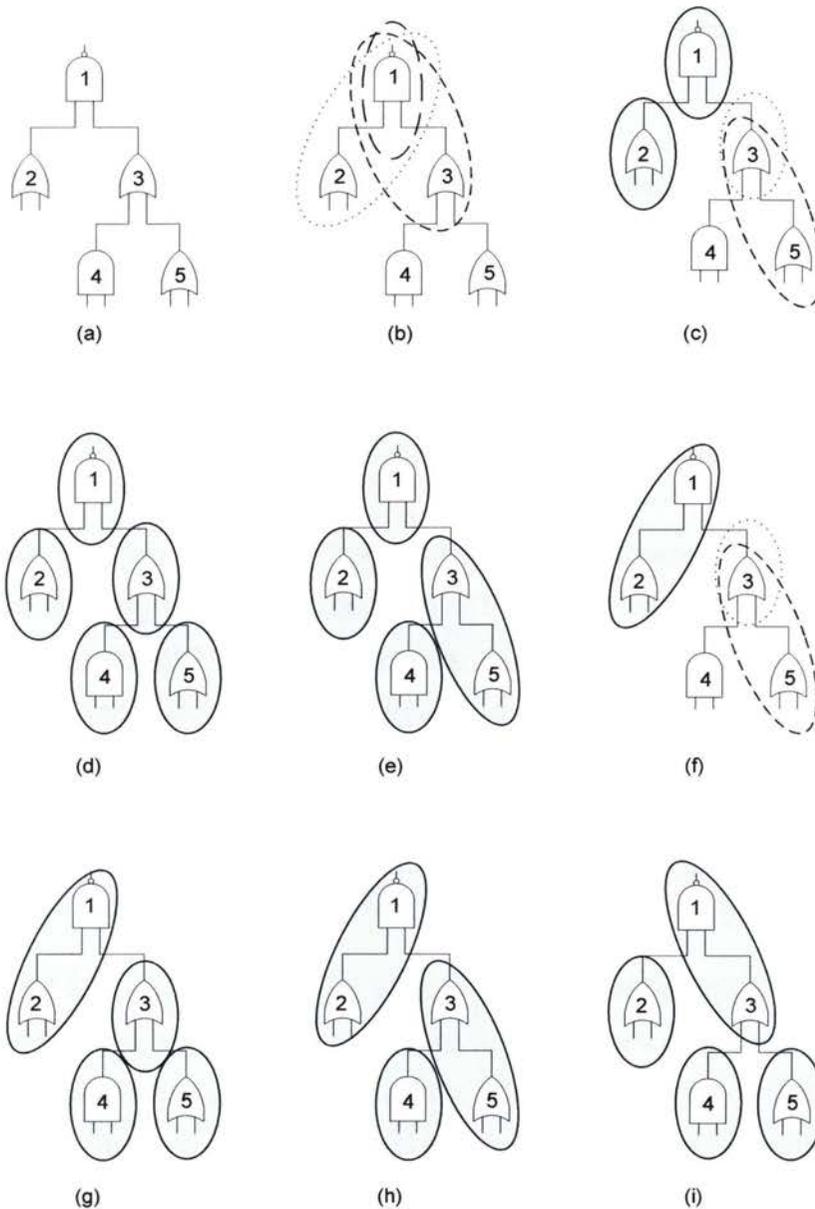


Figure 4-30. Matches Explored in Covering Algorithm

Next, the 3-input OR match is explored, and locked in place. The only child to this match is the 2-input AND at gate 4, and is locked in place as well. This is visible in Figure 4-30(e). The critical delay of the match at gate 3 is 12.66, and flows through the 2-input NAND at gate 4. The delay of the configuration in Figure 4-30(d) is lower than that in Figure 4-30(e); as such, the 2-input OR match is selected as the best match for gate 3, is

passed upwards to gate 1. The critical delay at the output of gate 1 is $3.33 + 11.33 = 14.66$.

Next, the OAI covering gates 1 and 2 is selected and locked into place. There is only one child node of this match, and it is gate 3, which has 2 potential matches to explore, as illustrated in Figure 4-30(f). First, the 2-input OR match is locked into place. As each of its children only contain one match, they are both visited and their costs are returned to gate 3. This covering is visible in Figure 4-30(g), with a delay at the output of gate 3 of 11.33.

Next, the 3-input OR covering is explored at gate 3. As there is only one child match, it is fixed into place as in Figure 4-30(h). The delay at the output of gate 3 is calculated to be 12.66. Clearly this is higher than 11.33; as such, the 2-input OR match is returned to gate 1. The critical delay of the covering in Figure 4-30(g) is calculated to be $4 + 7.33 = 11.33$.

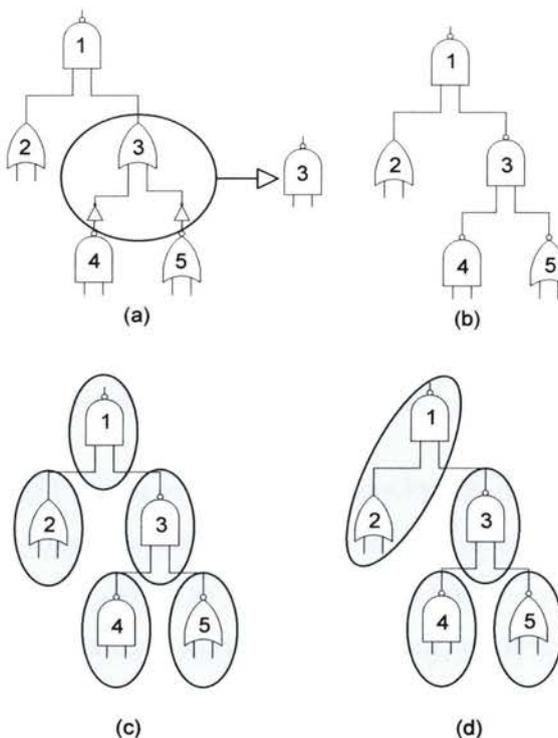


Figure 4-31. Gate Transformations through Boolean Matching

```

Match coverNode(Node n) {
    Match bestMatch = null;
    BestPath_Delay = Float.MaxValue;
    foreach (Match m in n.Matches) {
        m.PathDelay = -1;
        foreach (Node ni in m.Inputs) {
            ni.bestMatch = coverNode(ni);
            m.ChildMatches.Add(ni.bestMatch);
        }

        for (int i = 0; i < m.Inputs.Count; i++) {
            PathDelay = m.Inputs[i].bestMatch.PathDelay +
                m.LibraryCellMatch.LE[i] + m.LibraryCellMatch.P;
            if (PathDelay > m.PathDelay)
                m.PathDelay = PathDelay;
        }

        if (m.PathDelay < BestPath_Delay) {
            BestPath_Delay = m.PathDelay;
            bestMatch = m;
        }
    }
    return bestMatch;
}

```

Figure 4-32. Proposed logical effort delay minimizing covering algorithm

Finally, the OAI covering gates 1 and 3 is fixed into place as in Figure 4-30(i). As each child gate of this match only contains one match itself, they are all locked into place. The critical delay of this covering flows through the 2-input OR at gate 2, through the OAI and to the output. This delay is calculated to be $4.66 + 5.66 = 10.33$. Clearly 10.33 is the lowest critical path delay, and as such, this is the covering which optimizes the logical effort delay from the matches found in Figure 4-30.

Boolean matching allows the discovery of gates which are not included in the original netlist. Figure 4-31(a) illustrates how gate 3 can be transformed into a NAND by utilizing the Inverters associated with the AND, and OR gates from gates 4 and 5 respectfully, transforming them into NAND and NOR gates. Figure 4-31(b) illustrates the new circuit which is logically equivalent to those in Figure 4-30. This transformation allows the exploration of two additional covers which were not included in Figure 4-30, visible in

Figure 4-31(c) and Figure 4-31(d). The delay of Figure 4-31(c) can be calculated to be 10.33, and the delay of Figure 4-31(d) to be 11.33. Figure 4-31(c) is tied for the lowest delay with the cover in Figure 4-30(i), and can be selected as the best cover, with an area of 47 by the area calculation method described in Section 4.1.1.

The pseudo code for this logical effort delay minimizing algorithm is available in Figure 4-32.

```

Match coverNode(Node n) {
    Match bestMatch = null;
    BestArea = Float.MaxValue;
    foreach (Match m in n.Matches) {
        m.TotalArea = 0;
        foreach (Node ni in m.Inputs) {
            ni.bestMatch = coverNode(ni);
            m.ChildMatches.Add(ni.bestMatch);
        }

        for (int i = 0; i < m.Inputs.Count; i++) {
            m.TotalArea += m.Inputs[i].bestMatch.TotalArea +
                m.LibraryCellMatch.LE[i];
        }

        if (m.TotalArea < BestArea) {
            BestArea = m.TotalArea;
            bestMatch = m;
        }
    }
    return bestMatch;
}

```

Figure 4-33. Proposed area minimizing covering algorithm

4.5.2 Minimizing Circuit Area

In order to determine which set of matches will produce a cover with minimal circuit area, the method as described in Section 4.1.1 will be employed to calculate the area of each gate. A similar recursive algorithm was developed to that for delay minimization,

however the cost being minimized is total circuit area. For each match locked in place, the cost returned will be the total area of it, and that of all its children. The pseudo code for this area minimizing algorithm is available in Figure 4-32.

Example – Area minimizing covering algorithm

The area covering algorithm may be applied to the very simple circuit in Figure 4-30(a). The following example assumes that a library is available containing a two-input NAND, a two-input AND (NAND+Inverter), a two-input OR (NOR+Inverter), a three-input OR (NOR+Inverter), and an OR-AND-INVERT (OAI).

The first step of the covering algorithm looks at the root node, and lists the matches available for this node. At node 1, there are three matches, which are illustrated in Figure 4-30(b). These matches are a single NAND, an OAI containing nodes 1 and 2, and an OAI containing nodes 1 and 3. The covering algorithm works by iterating through each of these matches, determining which minimizes total circuit area. First the NAND match is selected, and fixed in place. First the left input to this gate is visited – gate 2. As there is only one gate at this node, only the OR match is found. It is returned, along with its cost to gate 1. The area of gate 2 is 13. Next, the right child of gate 1 is visited, gate 3 which has 2 potential matches, as seen in Figure 4-30(c).

First, the OR match is selected and locked into place. As each of the children of the OR match only contain one match themselves, they are visited, and locked in place in a similar manner, passing their delays upwards to node 3. This is visible in Figure 4-30(d). The area of gate 3 including its two children is 37.

Next, the 3-input OR match is explored, and locked in place. The only child to this match is the 2-input AND at gate 4, and is locked in place as well. This is visible in Figure 4-30(e). The area of the match at gate 3 including its one child is 35.

The area of the configuration in Figure 4-30(e) is lower than that in Figure 4-30(d); as such, the 3-input OR match is selected as the best match for gate 3, is passed upwards to gate 1. The total area of the covering in Figure 4-30(d) is **58**.

Next, the OAI covering gates 1 and 2 is selected and locked into place. There is only one child node of this match, and it is gate 3, which has 2 potential matches to explore, as illustrated in Figure 4-30(f). First, the 2-input OR match is locked into place. As each of its children only contain one match, they are both visited and their costs are returned to gate 3. This covering is visible in Figure 4-30(g), with the area of gate 3 and its children being 37.

Next, the 3-input OR covering is explored at gate 3. As there is only one child match, it is fixed into place as in Figure 4-30(h). The area of gate 3 with this match is 35. Clearly this is lower than 37; as such, the 3-input OR match is returned to gate 1. The area of the covering in Figure 4-30(h) is calculated to be **51**.

Finally, the OAI covering gates 1 and 3 is fixed into place as in Figure 4-30(i). As each child gate of this match only contains one match itself, they are all locked into place. The area of this covering is **53**.

Exploring the coverings in Figure 4-31(c) and Figure 4-31(d), their areas can be calculated to be 47 and 42. It can be seen that the covering which produces the minimum area is Figure 4-31(d) with a delay of 11.33.

4.6. Summary

This chapter has proposed novel matching and covering algorithms for library-free synthesis which minimize the critical path delay or total circuit area. The following chapter will overview implementation of these algorithms for testing against benchmark circuits.

5. Implementation

The proposed matching and covering algorithm was implemented in Microsoft C# .NET as a windows application. This environment was selected for its ease of development and testing. Microsoft .NET allows the developer to easily create many layers of abstraction in code allowing for organization and portability to other platforms. The developed synthesis algorithms and support functions are encapsulated in numerous levels of abstraction to allow for simple insertion into another application such as an Internet-based web service.

5.1 Synthesis Process Overview

The desired output of the synthesis tool is a netlist of complex gates which cover an input HDL, minimizing a given cost function. There are many steps which must be followed in order to accomplish this task, and many obstacles needed to be overcome.

5.1.1 Input Parsing and DAG Storage

The BENCH format is a simple, easy to read textual description of a circuit, which all ISCAS'85 benchmark circuits are provided in. The ISCAS'85 combinational benchmark circuits were selected as a measure for comparison between the developed tool and Synopsys Design Compiler due to their wide acceptance in academic research for CAD optimization. The BENCH format is very simple to parse, and is composed of three main Sections. The first is the definition of inputs to the circuit, defined in the format INPUT(ID). Second, the outputs are defined in the format OUTPUT(ID). Lastly, the gates of the circuit are defined in the format ID=GATE(ID₁, ID₂, .. ID_N), where ID is a unique identifier to that gate, and GATE is the type of gate in the set {AND, OR, NAND, NOR, NOT, XOR, BUF}. The NOT and BUF gates may only have one input, and the XOR gate may only have two inputs; the rest are unlimited. The inputs to the gate are the

unique identifiers as defined by other gates, or inputs. Figure 5-1 is the BENCH equivalent of the circuit in Figure 5-2.

```
# Simple BENCH example. Text on lines with a hash (#) are treated as  
# comments.  
  
INPUT(A)  
INPUT(B)  
INPUT(C)  
INPUT(D)  
INPUT(E)  
INPUT(F)  
OUTPUT(Y)  
  
1 = OR(A, B)  
2 = AND(1, C, D)  
3 = NAND(E, 2)  
Y = OR(F, 3)
```

Figure 5-1. Simple Example of BENCH Code

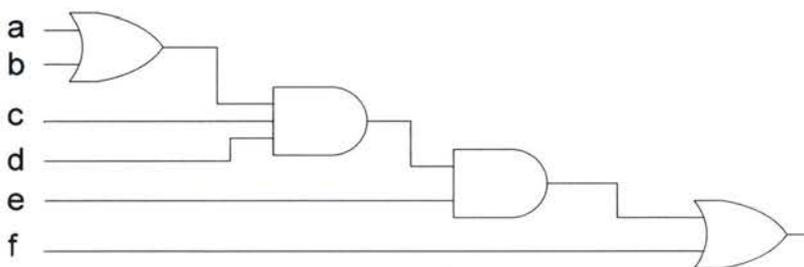


Figure 5-2. Boolean Representation of BENCH Code in Figure 5-1

The Node object is the most basic and most used object in the developed synthesis tool. The class description for the Node object can be seen in Figure 5-3, whose primary components are the Parent and Children linked lists. A network of Nodes is created as the input BENCH file is parsed, creating the DAG which will later be partitioned, matched, and covered. As each line of the input BENCH file is read, a new Node object is instantiated, and assigned the type and ID according to the BENCH file. The DAG is built from input to outputs, in the order gates appear in the BENCH file, with primary outputs being the roots of the DAG, and inputs being the child leaves.

All nodes are indexed in a globally accessible hash table for quick lookup by their unique ID. Similarly, primary inputs and outputs are each indexed in their own hash tables for quick lookup as needed.

```
public class Node
{
    public NODEType type;
    public List<Node> parent;
    public List<Node> children;

    public string ID;
    public string PartitionID = "";

    public Node inputChainLeft; //used for boolean solving
    public Node inputChainright; //used for boolean solving

    public bool value;
    public int LE;
    public int inputs_to_level; //used for subtree extraction

    public List<LibraryMatch> matches;
    public LibraryMatch bestMatch = null;
}
```

Figure 5-3. Node Object

5.1.2 Partitioning

Once the HDL has been parsed, and the DAG is created in memory, partitions are formed by iterating through each Node in the primary output hash table and traversing the DAG to primary inputs. Partitions are marked at each fanout point as described in Section 2.2. As partition locations are found in the DAG, partition objects are instantiated as defined in Figure 5-4. The primary components of the Partition object are a list of child partitions, a unique identifier, and a root node, which is linked to a node in the DAG. The Node object contains a variable PartitionID. The PartitionID of every Node is assigned to that of its containing Partition's ID. Barrier Nodes are inserted into the DAG at the inputs of each Partition; these Nodes are assigned the type IN. Although they do not logically change the DAG, they allow for quick identification of boundaries while traversing the graph.

Each partition may be considered completely independent of the others. The developed synthesis tool matches and covers partitions independently, in the order they are formed. A multi CPU computer may process partitions in parallel.

```
public class Partition
{
    public List<Partition> children;
    public string ID;
    public Node root;
    public Node rootChainInput;
    public int inputs = 0;
    public bool[] TruthTable;
}
```

Figure 5-4. Partition Object

5.1.3 Memory Management

Once partitioning is complete, matching and covering may begin. It can be seen in Figure 5-3 that each Node contains a list of matching library cells. Each match is defined by the LibraryMatch object as in Figure 5-5. Each LibraryMatch contains a list of child matches; creating a second graph composed of LibraryMatch objects. As each Node in the network may have multiple matches, one can see that the complexity of the interconnections between LibraryMatch objects quickly becomes great as the number of Nodes in the network increases. Initially, the synthesis tool was developed to match every partition, and then cover each partition. The amount of RAM required for this quickly exceeded 3GB during the synthesis of simple benchmark circuits due to the immense networks of matches created. In order to reduce this memory usage, partitions are processed one at a time, and are matched and covered in one step. Once an optimal cover is found for a partition, all remaining unused matches are removed from the graph and purged from memory.

```

public class LibraryMatch
{
    public string ID;
    public LibraryCell cell;
    public Node parentNode;
    public bool isBest = false;

    public string PCanonical;
    public string NPCanonical;
    public int[] POrder;
    public int[] NPOrder;

    public ArrayList Inputs; //nodes
    public bool[] InvertedInputs;

    public List<LibraryMatch> ChildMatches = new List<LibraryMatch>();
    public string TT;

    public double CriticalPath_LE;
    public double LE_AREA;
    public double LE_DELAY;
}

```

Figure 5-5. LibraryMatch Object

5.1.4 Timing Restrictions during Sub-tree Extraction

During the matching process, it is necessary to extract all sub-trees at a particular node in order to explore all possible covers. The number of extracted sub-trees strongly depends on the number of inputs to a partition which are below the node being extracted. The sub-tree extraction algorithm will not produce a tree which contains more inputs than the limit imposed by the maximum allowed series NMOS and PMOS transistors. Some partitions however, contain many times this number of inputs; as the number of inputs to a partition rises, so do the number of extracted sub-trees.

The P-canonical form for each extracted sub-tree must be generated, and checked for existence in the library's P-canonical hash table. Should a partition be processed with an excessive number of inputs, in order to not incur significant runtimes, the sub-tree extraction algorithm is timed. Should the main loop of the algorithm take more than a preset amount of time, the maximum number of inputs to be considered is reduced by one. This self-tuning algorithm allows practical runtimes for special case partitions.

5.1.5 User Interface and Result Presentation

In order for the developed synthesis tool to be useful, the results of the synthesis process must be easy for the end user to view, and verify. The tool has a simple tabbed interface, allowing the user to configure the settings of the synthesis process in the first tab, as is visible in Figure 5-6. Once the synthesis settings have been selected and the process has begun, status is updated on-screen in the Status tab, visible in Figure 5-7.

At the completion of the synthesis process, a report is generated similar to that visible in Figure 5-8. This report presents details on the results of the mapped circuit, including transistor count, circuit area, and critical path delay.

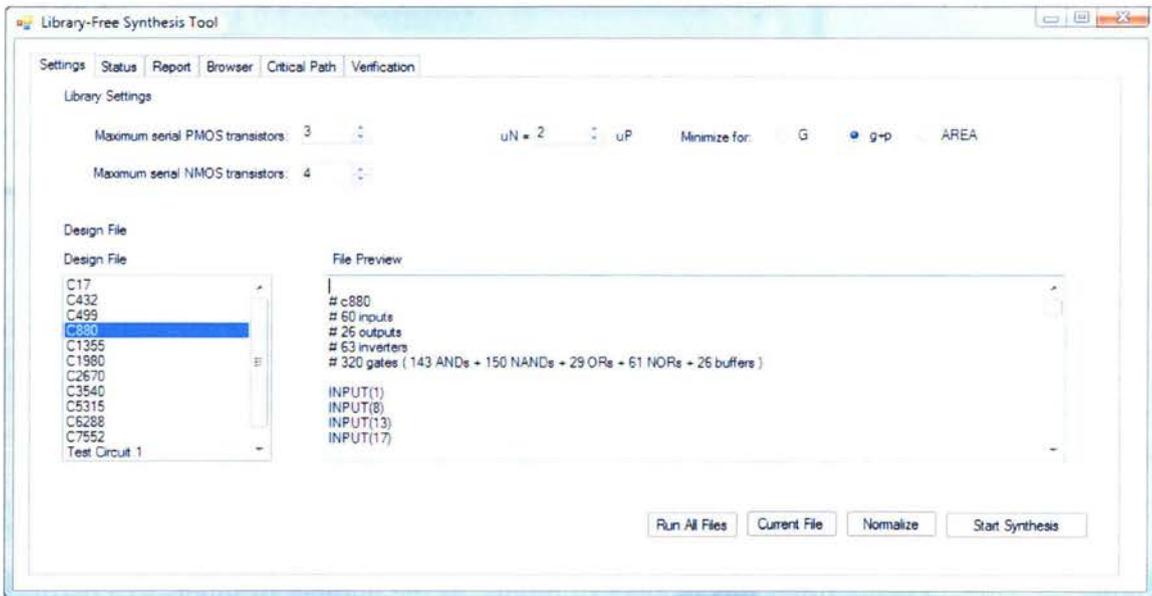


Figure 5-6. Configuration Tab of Synthesis Tool

The structure of the mapped library cells can be obtained by exploring the partitions in the Browser tab of the application, as is seen in Figure 5-9. A list of every partition is available in a drop-down box. Upon selecting a partition, a tree diagram is drawn allowing the user to explore the hierarchy of matches from the root of the partition down to its inputs. Selecting any individual match draws the PMOS transistor structure of the

gate. Transistors highlighted in red have inverted inputs, and were found using the NP-canonical form.

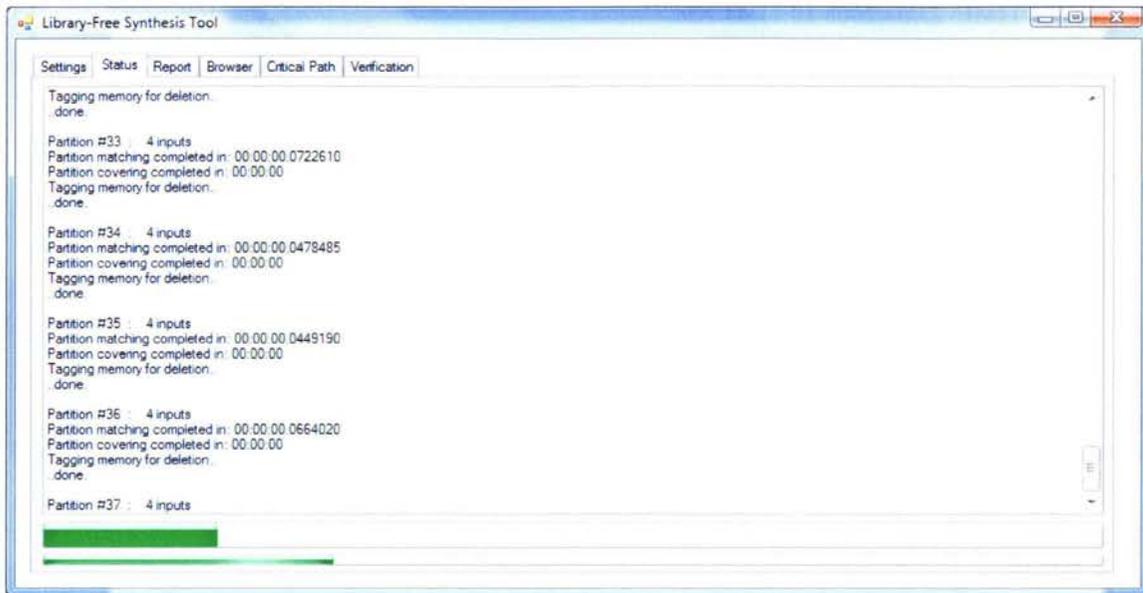


Figure 5-7. Status of Synthesis Process

The critical path may be viewed in the Critical Path tab. This allows the exploration of the gates along the slowest path of the mapped circuit. Figure 5-11 illustrates this feature.

Library-Free Synthesis Report

START TIME: 23/07/2008 10:02:01 AM
END TIME: 23/07/2008 10:11:44 AM
Total time spent: 00:09:30.6167985

Library: 155 cells
--Serial PMOS transistors: 3
--Serial NMOS transistors: 3

TOTAL TRANSISTOR COUNT WITH ADDED INPUT AND OUTPUT INVERTERS: 1540
TOTAL TRANSISTOR AREA WITH ADDED INPUT AND OUTPUT INVERTERS: 3394

Critical Path LE (G): 1517.38386763535
Critical Path # Stages (N): 14
Critical Path Parasitics (P): 44.6666666666667
Critical Path Branch Effort (B): 2000.7

H = 4
Critical Path Delay $D = N(\text{GHB})^{1/N} + P$: 89.556928395978
Area - Delay Product = $D * A$: (89.556928395978) * (3394) = 303956.214975949

H = 10
Critical Path Delay $D = N(\text{GHB})^{1/N} + P$: 91.8354071896211
Area - Delay Product = $D * A$: (91.8354071896211) * (3394) = 311689.372001574

H = 50
Critical Path Delay $D = N(\text{GHB})^{1/N} + P$: 97.5818992008538
Area - Delay Product = $D * A$: (97.5818992008538) * (3394) = 331192.965887698

H = 100
Critical Path Delay $D = N(\text{GHB})^{1/N} + P$: 100.267698537843
Area - Delay Product = $D * A$: (100.267698537843) * (3394) = 340308.568837438

Figure 5-8. Sample Report Generated by Synthesis Tool

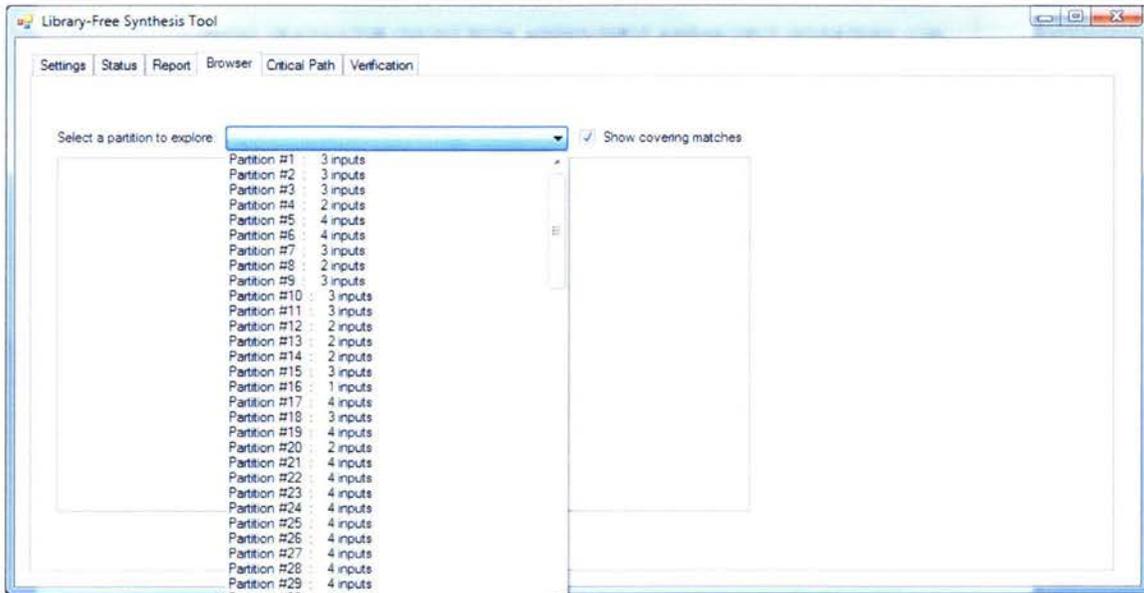


Figure 5-9. Browsing Partitions



Figure 5-10. Display of Matching Library Cell

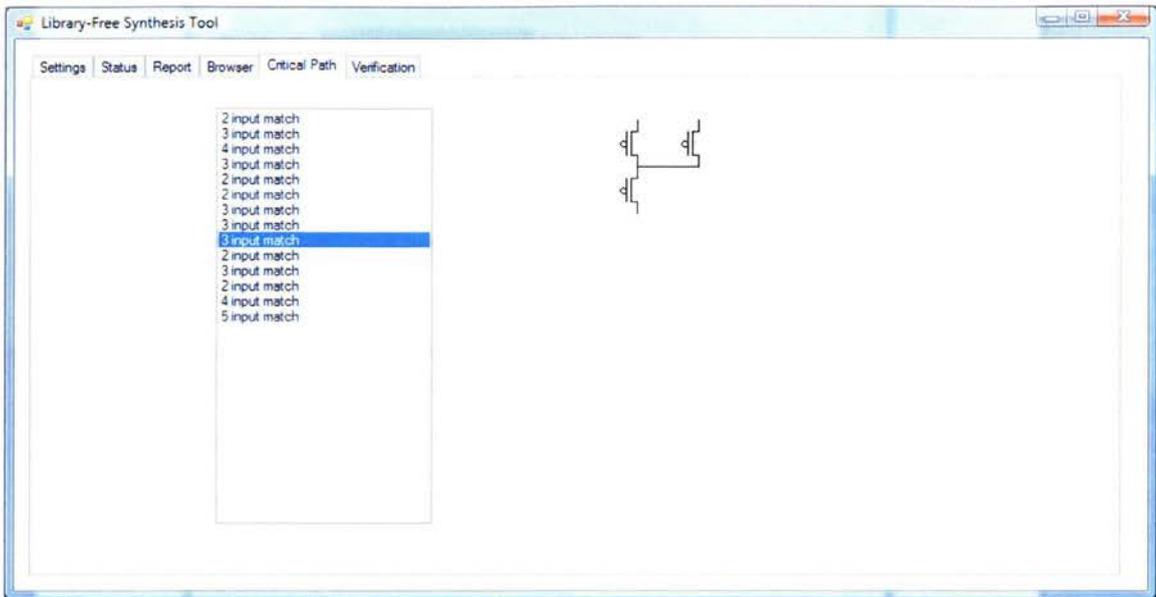


Figure 5-11. Display of Critical Path

5.1.6 Automated Verification of Results

In order to verify the correctness of the synthesized circuit, an algorithm was developed which generates random inputs to each primary input of the circuit. The original DAG is then solved recursively obtaining output values at each primary output given the set of inputs. The DAG is traversed again however the transistor structures of each match are solved recursively until values are available at the primary outputs of the circuit. If these two output values are identical, it can be said that synthesis is correct. The results of this automated verification are visible in Figure 5-12.

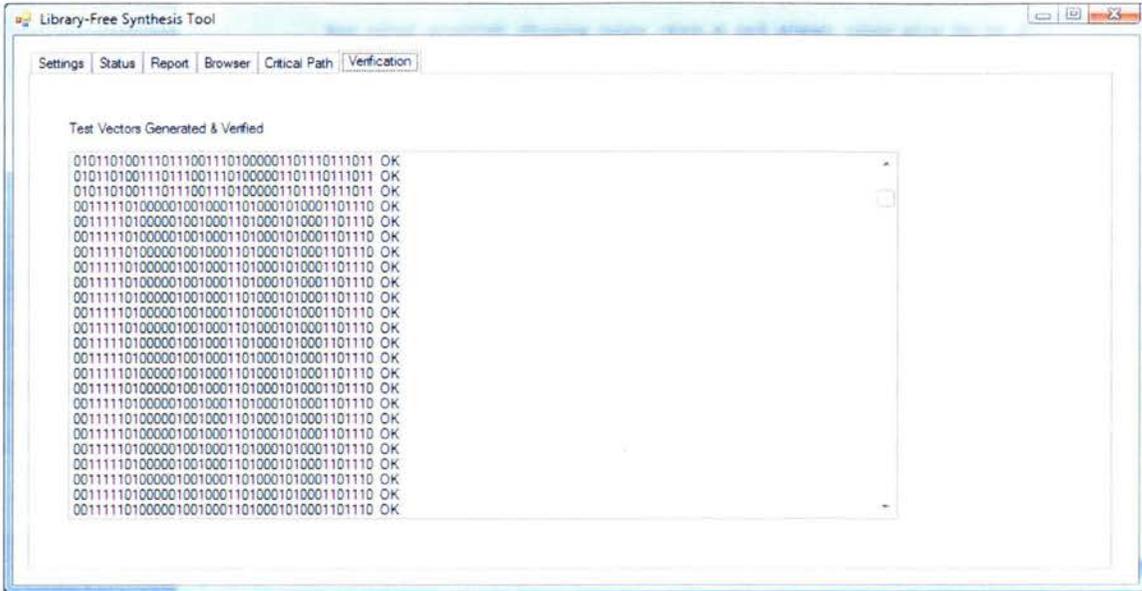


Figure 5-12. Result of Automated Verification

5.2 Summary

This chapter outlined the synthesis process and described implementation details for the proposed synthesis algorithms. The proposed synthesis algorithms were implemented using Microsoft Visual C#.NET and tested on a Windows Vista platform. The following chapter outlines performance results of the proposed algorithms compared with Synopsys Design Compiler 2005.

6. Results

In order to judge the performance of the developed synthesis tool, Synopsys Design Compiler (DC) 2005 was selected as a reference point. Synopsys offers synthesis tools which are widely used in industry for commercial ASIC design. The Synopsys tools are library based, and synthesize an input HDL to a set of pre-defined cells. The technology library selected for comparison is the TSMC 0.18 μ m library, which is available in the Ryerson VLSI lab.

Standard cell libraries contain exact area and timing information for each cell placed during the synthesis process. The developed tool does not have simulated area and timing data for each gate placed; as such, alternative metrics must be considered for comparison with Design Compiler. The area of the mapped circuit as produced by DC and that from the developed tool will be estimated using the method described in Section 4.1.1. The delay of the critical path as identified by DC is calculated using (4-1) in Section 4.1. Similarly, this equation is used to identify and calculate the delay of the critical path in the mapped circuit produced by the developed tool.

<u>Circuit Name</u>	<u>Circuit Function</u>	<u>Total Gates</u>	<u>I/O Pins</u>
C432	Priority Decoder	160	36
C499	ECAT	202	41
C880	ALU and Control	383	60
C1355	ECAT	546	41
C1908	ECAT	880	33
C2670	ALU and Control	1193	233
C3540	ALU and Control	1669	50
C5315	ALU and Selector	2307	178
C6288	16-bit Multiplier	2406	32
C7552	ALU and Control	3512	207

Figure 6-1. ISCAS'85 Benchmark Circuits

In 1985, the International Symposium on Circuits and Systems (ISCAS) released a set of ten combinational logic benchmark circuits for testing CAD algorithms. Since their release two decades ago, these circuits have been used to judge to the performance of

many algorithms in countless research papers. Each circuit, along with its output function, total gates and I/O pin count is available in Figure 6-1. Circuits C499 and C1355 are logically equivalent; All XOR gates in C499 have been expanded to their four NAND gate equivalent in C1355.

Design Compiler synthesized these benchmark circuits to the TSMC 0.18 μ m technology library with high map effort for cost function minimization. The proposed synthesis tool generated mapped circuits with the design constraint allowing a maximum of three serial PMOS, and four serial NMOS devices; creating gates with up to 12 inputs.

The path logical effort (G) and branching effort (B) of the critical path as defined by DC has been calculated for each benchmark circuit. Branching effort is defined as the ratio of the sum of the input capacitances of the gates being driven, to the input capacitance of the gate on the path. As exact capacitance values are not known, it is estimated as the ratio of the sum of the logical effort of the gates being driven to the logical effort of the gate on the path of interest, which is an estimation of the input capacitance without sizing. The branching effort for an individual stage in the critical path, b , is calculated as in (5-1). For simplicity, we will be ignoring the area and delay incurred with interconnects, and will not be concerned with the actual fabrication sizes of the transistors. Critical path delay is estimated using (4-1), which takes account for the logical effort, number of stages, branching effort, electrical effort and the parasitic delays along this path. Typically, the design constraint being minimized during synthesis is critical path delay; however, in certain cases circuit area minimization is desired. Design Compiler was set to independently minimize circuit critical path delay, as well as minimize total circuit area for all testable benchmark circuits with high map effort. In order to observe the performance impact of a load on the primary outputs, cases were studied with a varying load on the primary output equal to a multiple of the input capacitance.

$$b = \frac{g_{off_path} + g_{on_path}}{g_{on_path}} \quad (5-1)$$

The total circuit area and critical path delay has been calculated for the ISCAS'85 benchmark circuits after being synthesized in Synopsys Design Compiler and the developed tool. The developed tool was also run over every benchmark circuit for area, and delay minimization. The 16-bit multiplier in the ISCAS benchmark file C6288 has an extremely high number of paths from input to output pins. It has been shown [27] that there are at least 98943441738294937238 unique paths in this circuit. Due to this high number of paths, it is practically impossible to determine which is the path with the highest delay. The developed tool is able to partition, match, and cover the circuit within a similar amount of time as the other benchmark circuits, minimizing delay or area; however, it is not able to determine what the critical path delay is due to the number of paths to explore. As such, this circuit has been excluded from comparison with Synopsys Design Compiler.

6.1 Design Compiler Area Minimization

In this Section, Synopsys Design Compiler was set to minimize the total circuit area of each benchmark circuit. The resulting total circuit area and critical path delay of each circuit are compared with the developed synthesis tool for both area and delay minimization algorithms. Two cases are studied, one with a low output capacitance equal to $4\times$ the input capacitance, and one with high load equal to $100\times$ the input capacitance. The critical path delays of the mapped circuits obtained from both the designed synthesis tool and DC were calculated using (4-1) with an H value of 4 and 100. Section 6.1.1 compares Synopsys Design Compiler with high map effort for area minimization against the developed tool for area minimization with $C_{out}/C_{in}=4$. Section 6.1.2 compares Design Compiler area minimization against the developed tool for delay minimization with $C_{out}/C_{in}=4$. Section 6.1.3 compares Design Compiler with high map effort for area minimization with the developed tool for delay minimization with $C_{out}/C_{in}=100$. Section 6.1.4 compares Design Compiler area minimization with the developed tool for area minimization with $C_{out}/C_{in}=100$.

6.1.1 DC vs. Developed Tool Area Minimization (H = 4)

The developed synthesis tool was set to minimize area for each benchmark circuit by minimizing the area in each partition as described in Section 4.5.2. Synopsys Design Compiler 2005 was set to minimize the area of each combinational benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-2.

Circuit	<u>Design Compiler</u> <u>Area Minimization</u>			<u>Developed Synthesis Tool</u> <u>Area Minimization</u>						
	Area	Delay	# Transistor	Area	Delay	# Transistor	Δ Area	Δ Tran. Count	Δ Delay	
C432	1547	116.79	612	1677	115	696	7.75%	12.07%	-1.56%	
C499	3605	135.02	1526	4134	80.9	1724	12.80%	11.48%	-66.90%	
C880	3029	134.69	1160	2947	89.28	1212	-2.78%	4.29%	-50.86%	
C1355	3700	121.96	1584	4342	116.41	2140	14.79%	25.98%	-4.77%	
C1980	3683	144.39	1532	5461	115	2314	32.56%	33.79%	-25.56%	
C2670	5879	99.5	2442	7179	136.21	3050	18.11%	19.93%	26.95%	
C3540	8465	178.75	3194	8318	162.38	3548	-1.77%	9.98%	-10.08%	
C5315	12932	114.67	5008	18087	124.56	7584	28.50%	33.97%	7.94%	
C7552	14493	287.16	5854	23934	97.97	10162	39.45%	42.39%	-193.11%	
Avg.Δ - (Design Compiler vs. Developed Tool)							16.60%	21.54%	-35.33%	

Figure 6-2. DC Area Minimization vs. Developed Tool Area Minimization – H=4

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from the Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-3. From this data, it can be seen that in the majority of cases Design Compiler is able to better minimize area than the developed tool, while the developed tool achieves an average delay reduction of 35.33%. Overall, the developed tool produced mapped circuits with an average area-delay product 9.27% less than that from Design Compiler.

<u>Circuit</u>	<u>Δ Area\timesDelay</u>
C432	6.32
C499	-45.54
C880	-55.06
C1355	10.72
C1980	15.32
C2670	40.18
C3540	-12.03
C5315	34.18
C7552	-77.49
Avg.Δ	-9.27 %

Figure 6-3. Design Compiler vs. Developed Tool Δ Area-Delay Product (Area vs. Area)

6.1.2 DC vs. Developed Tool Delay Minimization (H = 4)

The developed synthesis tool was set to minimize delay for each benchmark circuit by minimizing the critical path delay in each partition as described in Section 4.5.1. Synopsys Design Compiler 2005 was set to minimize the area of each combinational benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-4.

Circuit	<u>Design Compiler</u> <u>Area Minimization</u>			<u>Developed Synthesis Tool</u> <u>Delay Minimization</u>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	Δ Area	Δ Tran. Count	Δ Delay
C432	1547	116.79	612	2041	117.39	850	24.20%	0.51%	0.51%
C499	3605	135.02	1526	4182	78.49	1756	13.80%	-72.02%	-72.02%
C880	3029	134.69	1160	3268	86.93	1400	7.31%	-54.94%	-54.94%
C1355	3700	121.96	1584	4390	113.98	2172	15.72%	-7.00%	-7.00%
C1980	3683	144.39	1532	5581	116.97	2362	34.01%	-23.44%	-23.44%
C2670	5879	99.5	2442	7332	131.02	3072	19.82%	24.06%	24.06%
C3540	8465	178.75	3194	8949	167.55	3614	5.41%	-6.68%	-6.68%
C5315	12932	114.67	5008	18909	124.99	7846	31.61%	8.26%	8.26%
C7552	14493	287.16	5854	24779	97.68	10316	41.51%	-193.98%	-193.98%
Avg.Δ - (Design Compiler vs. Developed Tool)							21.49%	-36.14%	-36.14%

Figure 6-4. DC Area Minimization vs. Developed Tool Delay Minimization – H=4

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-5. From this data, it can be seen that in the majority of cases Design Compiler is able to better minimize area than the developed tool, while the developed tool achieves an average delay reduction of 36.14%. Overall, the developed tool produced mapped circuits with an average area-delay product 3.94% less than that from Design Compiler.

<u>Circuit</u>	<u>Δ Area\timesDelay</u>
C432	24.59%
C499	-48.29%
C880	-43.61%
C1355	9.82%
C1980	18.54%
C2670	39.11%
C3540	-0.91%
C5315	37.26%
C7552	-71.95%
Avg.Δ	-3.94 %

Figure 6-5. Design Compiler vs. Developed Tool Δ Area-Delay Product (Area vs. Delay)

6.1.3 DC vs. Developed Tool Area Minimization (H = 100)

The developed synthesis tool was set to minimize area for each benchmark circuit by minimizing the area in each partition as described in Section 4.5.2. Synopsys Design Compiler 2005 was set to minimize the area of each combinational benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-6.

Circuit	<u>Design Compiler</u> <u>Area Minimization</u>			<u>Developed Synthesis Tool</u> <u>Area Minimization</u>						
	Area	Delay	# Transistor	Area	Delay	# Transistor	ΔArea	ΔTran. Count	ΔDelay	
C432	1575	128.86	622	1677	123.95	696	6.08%	10.63%	-3.96%	
C499	3689	126.87	1582	4134	89.12	1724	10.76%	8.24%	-42.36%	
C880	3102	143.64	1206	2947	100.32	1212	-5.26%	0.50%	-43.18%	
C1355	3796	136.07	1648	4342	124.48	2140	12.57%	22.99%	-9.31%	
C1980	3614	146.45	1533	5461	122.04	2314	33.82%	33.75%	-20.00%	
C2670	5786	172.75	2456	7179	144.18	3050	19.40%	19.48%	-19.82%	
C3540	8480	218.63	3208	8318	171.64	3548	-1.95%	9.58%	-27.38%	
C5315	12925	174.82	5054	18087	131.47	7584	28.54%	33.36%	-32.97%	
C7552	14593	296.74	5953	23934	105.42	10162	39.03%	41.42%	-181.48%	
Avg.Δ - (Design Compiler vs. Developed Tool)							15.89%	19.99%	-42.27%	

Figure 6-6. DC Area Minimization vs. Developed Tool Area Minimization – H=100

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-7. From this data, it can be seen that in the majority of cases, the Synopsys tools are able to better minimize area than the developed tool, while the developed tool achieves an average delay reduction of 42.27%. Overall, the developed tool produced mapped circuits with an average area-delay product 15.94% less than that from Design Compiler.

<u>Circuit</u>	<u>Δ Area×Delay</u>
C432	2.36
C499	-27.03
C880	-50.71
C1355	4.43
C1980	20.58
C2670	3.43
C3540	-29.86
C5315	4.98
C7552	-71.63
Avg.Δ	-15.94 %

Figure 6-7. Design Compiler vs. Developed Tool Δ Area-Delay Product (Area vs. Area)

6.1.4 DC vs. Developed Tool Delay Minimization (H = 100)

The developed synthesis tool was set to minimize delay for each benchmark circuit by minimizing the critical path delay in each partition as described in Section 4.5.1. Synopsys Design Compiler 2005 was set to minimize the area of each combinational benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-8.

Circuit	<i>Design Compiler Area Minimization</i>			<i>Developed Synthesis Tool Delay Minimization</i>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	Δ Area	Δ Tran. Count	Δ Delay
C432	1575	128.86	622	2041	127.64	850	22.83%	26.82%	-0.96%
C499	3689	126.87	1582	4182	86.6	1756	11.79%	9.91%	-46.50%
C880	3102	143.64	1206	3268	96.6	1400	5.08%	13.86%	-48.70%
C1355	3796	136.07	1648	4390	122	2172	13.53%	24.13%	-11.53%
C1980	3614	146.45	1533	5581	125.5	2362	35.24%	35.10%	-16.69%
C2670	5786	172.75	2456	7332	138.81	3072	21.09%	20.05%	-24.45%
C3540	8480	218.63	3208	8949	176.92	3614	5.24%	11.23%	-23.58%
C5315	12925	174.82	5054	18909	131.75	7846	31.65%	35.59%	-32.69%
C7552	14593	296.74	5953	24779	104.89	10316	41.11%	42.29%	-182.91%
Avg. Δ - (Design Compiler vs. Developed Tool)							20.84%	24.33%	-43.11%

Figure 6-8. DC Area Minimization vs. Developed Tool Delay Minimization – H=100

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-9. From this data, it can be seen that in the majority of cases, the Synopsys tools are able to better minimize area than the developed tool, while the developed tool achieves an average delay reduction of 43.11%. Overall, the developed tool produced mapped circuits with an average area-delay product 10.32% less than that from Design Compiler.

<u>Circuit</u>	<u>Δ Area\timesDelay</u>
C432	22.09%
C499	-29.23%
C880	-41.14%
C1355	3.56%
C1980	24.43%
C2670	1.79%
C3540	-17.1%
C5315	9.3%
C7552	-66.61%
Avg. Δ	-10.32 %

Figure 6-9. Design Compiler vs. Developed Tool Δ Area-Delay Product (Area vs. Delay)

6.1.5 Comparison of Synthesis Algorithms

In all cases outlined above for both low and high output loads, the developed synthesis tool outperformed Synopsys Design Compiler with respect to the average area-critical path delay product for the set of ISCAS benchmark circuits.

Figure 6-10 illustrates the area-delay product difference between Synopsys Design Compiler for area minimization and the developed tool for both area and delay minimization for each circuit with an output load $4\times$ the input capacitance. In nearly all cases, the area minimization algorithm as described in Section 4.5.2 outperforms the delay minimization algorithm of Section 4.5.1.

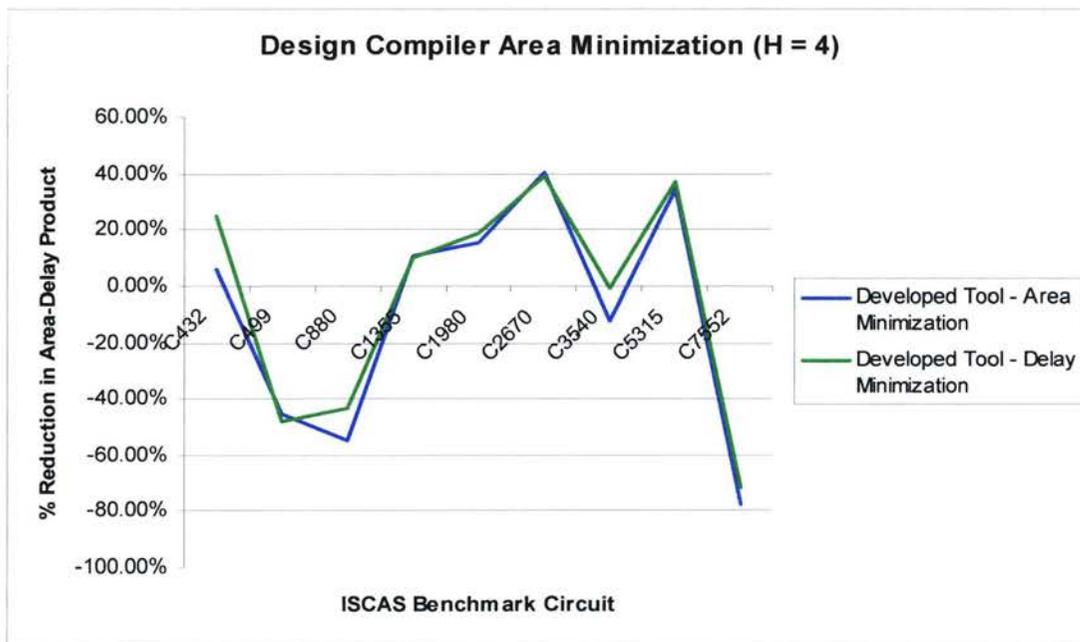


Figure 6-10. DC Area Minimization (H=4)

Figure 6-11 illustrates the area-delay product difference between Synopsys Design Compiler for area minimization and the developed tool for both area and delay minimization for each circuit with an output load $100\times$ the input capacitance. In nearly all cases, the area minimization algorithm as described in Section 4.5.2 outperforms the delay minimization algorithm of Section 4.5.1.

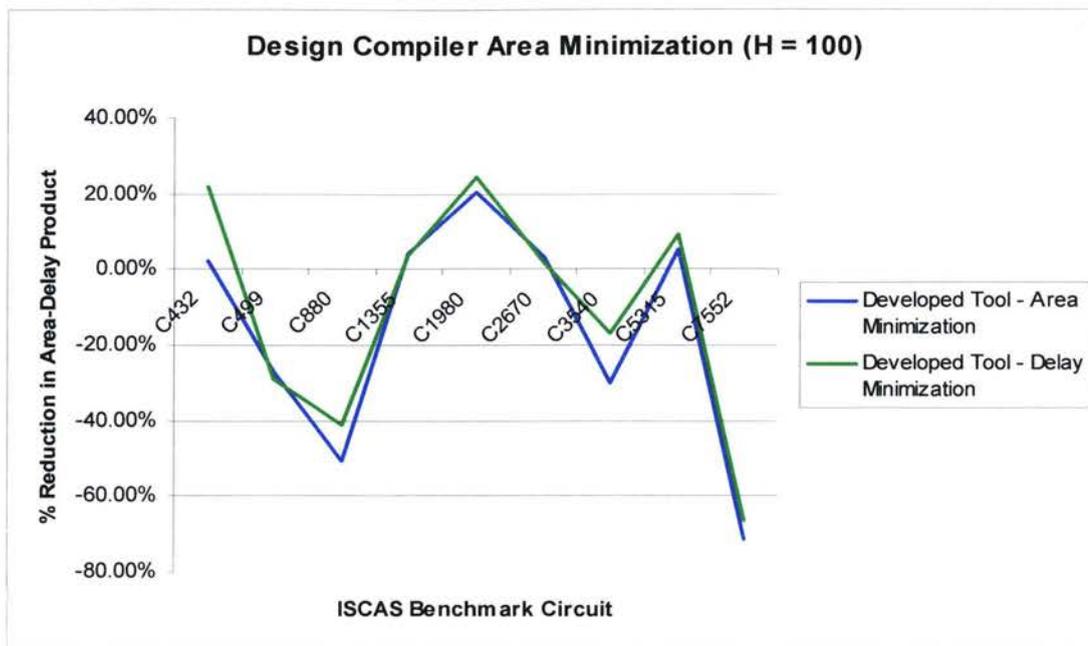


Figure 6-11. DC Area Minimization (H=100)

6.2 Design Compiler Delay Minimization

In this Section, Synopsys Design Compiler was set to minimize the critical path delay of each benchmark circuit, the result of which will be compared with the developed synthesis tool for both area and delay minimization. As delay minimization is typically preferred over area minimization, more cases will be studied here to observe the effects of various output loads than in Section 6.1.

Four cases are studied, with output capacitances equal to 2×, 4×, 16× and 100× the input capacitance. A high output load requires higher current to charge the load in the same time than a lower load does. As the source voltage is not modified if the load is increased, the resistance of the paths from VDD to load, and load to VSS must be lowered in order to support this higher current. As such, the widths of the transistors in the gate directly driving the output load will typically be higher than those of gates elsewhere along the path. These larger transistors themselves become more difficult to drive by earlier stages as they are now larger. Typically a design with a high output load requires more stages

along the path to gradually “upsized” the current in order to drive the large load in a timely manner. This dependence on the number of stages and the output load size are both represented by the variables N and H in (4-1) when calculating the logical effort delay of the critical path.

The critical path delays of the mapped circuits obtained from both the designed synthesis tool and DC were calculated using (4-1) with an H value of 2, 4, 16 and 100. Sections 6.2.1 and 6.2.2 compare Synopsys Design Compiler with high map effort for delay minimization against the developed tool for delay, and area minimization, respectively with $C_{out}/C_{in}=2$. Sections 6.2.3 and 6.2.4 compare Design Compiler with high map effort for delay minimization against the developed tool for delay, and area minimization, respectively with $C_{out}/C_{in}=4$. Sections 6.2.5 and 6.2.6 compare Design Compiler with high map effort for delay minimization against the developed tool for delay, and area minimization, respectively with $C_{out}/C_{in}=16$. Sections 6.2.7 and 6.2.8 compare Design Compiler with high map effort for delay minimization against the developed tool for delay, and area minimization, respectively with $C_{out}/C_{in}=100$.

6.2.1 DC vs. Developed Tool Delay Minimization ($H = 2$)

The developed synthesis tool was set to minimize delay for each benchmark circuit by minimizing the delay in each partition as described in Section 4.5.1. Similarly, Synopsys Design Compiler was set to minimize critical path delay in each benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-12. From this data, it can be seen that the developed tool produces a circuit with a critical path delay only marginally higher than that from Design Compiler, while achieving a 41.97% reduction in area, and a 53.72% reduction in transistor count.

Circuit	<i>Design Compiler Delay Minimization</i>			<i>Developed Synthesis Tool Delay Minimization</i>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	Δ Area	Δ Tran. Count	Δ Delay
C432	2573	119	1308	2041	115.23	850	-26.07%	-53.88%	-3.27%
C499	6985	91.63	3218	4182	76.82	1756	-67.03%	-83.26%	-19.28%
C880	7234	63.6	3164	3268	84.94	1400	-121.36%	-126.00%	25.12%
C1355	9446	94.6	4470	4390	112.26	2172	-115.17%	-105.80%	15.73%
C1980	8677	79.79	4012	5581	115.14	2362	-55.47%	-69.86%	30.70%
C2670	6874	124.68	3734	7332	129.33	3072	6.25%	-21.55%	3.60%
C3540	12049	161.78	5406	8949	165.51	3614	-34.64%	-49.58%	2.25%
C5315	16351	98.98	7056	18909	123.53	7846	13.53%	10.07%	19.87%
C7552	19271	126.4	8630	24779	96.17	10316	22.23%	16.34%	-31.43%
Avg. Δ - (Design Compiler vs. Developed Tool)							-41.97%	-53.72%	4.81%

Figure 6-12. DC Delay Minimization vs. Developed Tool Delay Minimization – H=2

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-13. From this figure, it can be seen that the developed tool is able to reduce the overall area-delay product by 30.86% with comparison to Design Compiler.

<u>Circuit</u>	<u>Δ Area\timesDelay</u>
C432	-30.19%
C499	-99.23%
C880	-65.75%
C1355	-81.32%
C1980	-7.74%
C2670	9.62%
C3540	-31.61%
C5315	30.71%
C7552	-2.22%
Avg. Δ	-30.86 %

Figure 6-13. Design Compiler vs. Developed Tool Δ Area-Delay Product (Delay vs. Delay)

6.2.2 DC vs. Developed Tool Area Minimization (H = 2)

The developed synthesis tool was set to minimize area for each benchmark circuit by minimizing the area in each partition as described in Section 4.5.2. Similarly, Synopsys Design Compiler was set to minimize critical path delay in each benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-14. From this data, it can be seen that the developed tool produces a circuit with a critical path

delay 5.35% higher than that from Synopsys, while achieving a 50.65% reduction in area, and a 63.41% reduction in transistor count.

Circuit	<i>Design Compiler Delay Minimization</i>			<i>Developed Synthesis Tool Area Minimization</i>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	Δ Area	Δ Tran. Count	Δ Delay
C432	2573	119	1308	1677	113.18	696	-53.43%	-87.93%	-5.14%
C499	6985	91.63	3218	4134	79.21	1724	-68.96%	-86.66%	-15.68%
C880	7234	63.6	3164	2947	87.04	1212	-145.47%	-161.06%	26.93%
C1355	9446	94.6	4470	4342	114.67	2140	-117.55%	-108.88%	17.50%
C1980	8677	79.79	4012	5461	113.48	2314	-58.89%	-73.38%	29.69%
C2670	6874	124.68	3734	7179	134.49	3050	4.25%	-22.43%	7.29%
C3540	12049	161.78	5406	8318	160.37	3548	-44.85%	-52.37%	-0.88%
C5315	16351	98.98	7056	18087	123.07	7584	9.60%	6.96%	19.57%
C7552	19271	126.4	8630	23934	96.41	10162	19.48%	15.08%	-31.11%
Avg.Δ - (Design Compiler vs. Developed Tool)							-50.65%	-63.41%	5.35%

Figure 6-14. DC Delay Minimization vs. Developed Tool Area Minimization – H=2

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-15. From this figure, it can be seen that the developed tool is able to reduce the overall area-delay product by 37.83% with comparison to Design Compiler.

<u>Circuit</u>	<u>Δ Area\timesDelay</u>
C432	-61.32%
C499	-95.46%
C880	-79.36%
C1355	-79.47%
C1980	-11.72%
C2670	11.23%
C3540	-46.13%
C5315	27.29%
C7552	-5.56%
Avg.Δ	-37.83 %

Figure 6-15. Design Compiler vs. Developed Tool Δ Area-Delay Product (Delay vs. Area)

6.2.3 DC vs. Developed Tool Delay Minimization (H = 4)

The developed synthesis tool was set to minimize delay for each benchmark circuit by minimizing the delay in each partition as described in Section 4.5.1. Similarly, Synopsys Design Compiler was set to minimize critical path delay in each benchmark circuit with

high map effort. The results of these synthesis tools are presented in Figure 6-16. From this data, it can be seen that the developed tool produces a circuit with a critical path delay only marginally higher than that from Design Compiler, while achieving a 44.75% reduction in area, and a 59.95% reduction in transistor count.

Circuit	<u>Design Compiler Delay Minimization</u>			<u>Developed Synthesis Tool Delay Minimization</u>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	Δ Area	Δ Tran. Count	Δ Delay
C432	2527	129.77	1274	2041	117.39	850	-23.81%	-49.88%	-10.55%
C499	8285	95.83	3992	4182	78.49	1756	-98.11%	-127.33%	-22.09%
C880	6785	79.37	3063	3268	86.93	1400	-107.62%	-118.79%	8.70%
C1355	9662	94.36	4838	4390	113.98	2172	-120.09%	-122.74%	17.21%
C1980	7604	127.09	3604	5581	116.97	2362	-36.25%	-52.58%	-8.65%
C2670	7880	84.4	3748	7332	131.02	3072	-7.47%	-22.01%	35.58%
C3540	13152	147.83	6048	8949	167.55	3614	-46.97%	-67.35%	11.77%
C5315	15863	115.34	7224	18909	124.99	7846	16.11%	7.93%	7.72%
C7552	19462	106.6	8954	24779	97.68	10316	21.46%	13.20%	-9.13%
Avg.Δ - (Design Compiler vs. Developed Tool)							-44.75%	-59.95%	3.40%

Figure 6-16. DC Delay Minimization vs. Developed Tool Delay Minimization – H=4

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-17. From this figure, it can be seen that the developed tool is able to reduce the overall area-delay product by 40.06% with comparison to Design Compiler.

<u>Circuit</u>	<u>Δ Area\timesDelay</u>
C432	-36.87%
C499	-141.88%
C880	-89.56%
C1355	-82.21%
C1980	-48.04%
C2670	30.77%
C3540	-29.67%
C5315	22.59%
C7552	14.29%
Avg.Δ	-40.06 %

Figure 6-17. Design Compiler vs. Developed Tool Δ Area-Delay Product (Delay vs. Delay)

6.2.4 DC vs. Developed Tool Area Minimization (H = 4)

The developed synthesis tool was set to minimize area for each benchmark circuit by minimizing the area in each partition as described in Section 4.5.2. Similarly, Synopsys Design Compiler was set to minimize critical path delay in each benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-18. From this data, it can be seen that the developed tool produces a circuit with a critical path delay 3.76% higher than that from Synopsys, while achieving a 53.33% reduction in area, and a 69.54% reduction in transistor count.

Circuit	<u>Design Compiler</u> <u>Delay Minimization</u>			<u>Developed Synthesis Tool</u> <u>Area Minimization</u>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	ΔArea	ΔTran. Count	ΔDelay
C432	2527	129.77	1274	1677	115	696	-50.69%	-83.05%	-12.84%
C499	8285	95.83	3992	4134	80.9	1724	-100.41%	-131.55%	-18.45%
C880	6785	79.37	3063	2947	89.28	1212	-130.23%	-152.72%	11.10%
C1355	9662	94.36	4838	4342	116.41	2140	-122.52%	-126.07%	18.94%
C1980	7604	127.09	3604	5461	115	2314	-39.24%	-55.75%	-10.51%
C2670	7880	84.4	3748	7179	136.21	3050	-9.76%	-22.89%	38.04%
C3540	13152	147.83	6048	8318	162.38	3548	-58.11%	-70.46%	8.96%
C5315	15863	115.34	7224	18087	124.56	7584	12.30%	4.75%	7.40%
C7552	19462	106.6	8954	23934	97.97	10162	18.68%	11.89%	-8.81%
Avg.Δ - (Design Compiler vs. Developed Tool)							-53.33%	-69.54%	3.76%

Figure 6-18. DC Delay Minimization vs. Developed Tool Area Minimization – H=4

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-19. From this figure, it can be seen that the developed tool is able to reduce the overall area-delay product by 47.56% with comparison to Design Compiler.

<u>Circuit</u>	<u>Δ Area×Delay</u>
C432	-70.04%
C499	-137.4%
C880	-104.68%
C1355	-80.37%
C1980	-53.88%
C2670	31.99%
C3540	-43.95%
C5315	18.79%
C7552	11.52%
Avg.Δ	-47.56 %

Figure 6-19. Design Compiler vs. Developed Tool Δ Area-Delay Product (Delay vs. Area)

6.2.5 DC vs. Developed Tool Delay Minimization (H = 16)

The developed synthesis tool was set to minimize delay for each benchmark circuit by minimizing the delay in each partition as described in Section 4.5.1. Similarly, Synopsys Design Compiler was set to minimize critical path delay in each benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-16. From this data, it can be seen that the developed tool produces a circuit with a critical path delay only marginally higher than that from Design Compiler, while achieving a 52.29% reduction in area, and a 65.17% reduction in transistor count.

Circuit	<i>Design Compiler Delay Minimization</i>			<i>Developed Synthesis Tool Delay Minimization</i>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	ΔArea	ΔTran. Count	ΔDelay
C432	3106	129.74	1622	2041	121.94	850	-52.18%	-90.82%	-6.40%
C499	8422	90.39	3966	4182	82.06	1756	-101.39%	-125.85%	-10.15%
C880	7390	76.39	3338	3268	91.18	1400	-126.13%	-138.43%	16.22%
C1355	9625	93.71	4506	4390	117.57	2172	-119.25%	-107.46%	20.29%
C1980	8422	127.91	3926	5581	120.78	2362	-50.90%	-66.22%	-5.90%
C2670	8013	99.19	3716	7332	134.52	3072	-9.29%	-20.96%	26.26%
C3540	12605	152.72	5546	8949	171.76	3614	-40.85%	-53.46%	11.09%
C5315	16637	116.1	7386	18909	128.01	7846	12.02%	5.86%	9.30%
C7552	20471	111.54	9198	24779	100.88	10316	17.39%	10.84%	-10.57%
Avg.Δ - (Design Compiler vs. Developed Tool)							-52.29%	-65.17%	5.57%

Figure 6-20. DC Delay Minimization vs. Developed Tool Delay Minimization – H=16

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-21. From this figure, it can be seen that the developed tool is able to reduce the overall area-delay product by 42.75% with comparison to Design Compiler.

<u>Circuit</u>	<u>Δ Area×Delay</u>
C432	-61.91%
C499	-121.83%
C880	-89.45%
C1355	-74.75%
C1980	-59.81%
C2670	19.42%
C3540	-25.24%
C5315	20.2%
C7552	8.66%
Avg.Δ	-42.75%

Figure 6-21. Design Compiler vs. Developed Tool Δ Area-Delay Product (Delay vs. Delay)

6.2.6 DC vs. Developed Tool Area Minimization (H = 16)

The developed synthesis tool was set to minimize area for each benchmark circuit by minimizing the area in each partition as described in Section 4.5.2. Similarly, Synopsys Design Compiler was set to minimize critical path delay in each benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-22. From this data, it can be seen that the developed tool produces a circuit with a critical path delay 3.76% higher than that from Synopsys, while achieving a 61.81% reduction in area, and a 76.09% reduction in transistor count.

Circuit	<u>Design Compiler</u> <u>Delay Minimization</u>			<u>Developed Synthesis Tool</u> <u>Area Minimization</u>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	ΔArea	ΔTran. Count	ΔDelay
C432	3106	129.74	1622	1677	119.04	696	-85.21%	-133.05%	-8.99%
C499	8422	90.39	3966	4134	84.51	1724	-103.73%	-130.05%	-6.96%
C880	7390	76.39	3338	2947	94.01	1212	-150.76%	-175.41%	18.74%
C1355	9625	93.71	4506	4342	120.02	2140	-121.67%	-110.56%	21.92%
C1980	8422	127.91	3926	5461	118.16	2314	-54.22%	-69.66%	-8.25%
C2670	8013	99.19	3716	7179	139.8	3050	-11.62%	-21.84%	29.05%
C3540	12605	152.72	5546	8318	166.54	3548	-51.54%	-56.31%	8.30%
C5315	16637	116.1	7386	18087	127.65	7584	8.02%	2.61%	9.05%
C7552	20471	111.54	9198	23934	101.26	10162	14.47%	9.49%	-10.15%
Avg.Δ - (Design Compiler vs. Developed Tool)							-61.81%	-76.09%	5.86%

Figure 6-22. DC Delay Minimization vs. Developed Tool Area Minimization – H=16

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-23. From this figure, it can be seen that the developed tool is able to reduce the overall area-delay product by 51.06% with comparison to Design Compiler.

<u>Circuit</u>	<u>Δ Area×Delay</u>
C432	-101.86%
C499	-117.9%
C880	-103.76%
C1355	-73.08%
C1980	-66.95%
C2670	20.81%
C3540	-38.96%
C5315	16.34%
C7552	5.79%
Avg.Δ	-51.06 %

Figure 6-23. Design Compiler vs. Developed Tool Δ Area-Delay Product (Delay vs. Area)

6.2.7 DC vs. Developed Tool Delay Minimization (H = 100)

The developed synthesis tool was set to minimize delay for each benchmark circuit by minimizing the delay in each partition as described in Section 4.5.1. Similarly, Synopsys Design Compiler was set to minimize critical path delay in each benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-24. From this data, it can be seen that the developed tool produces a circuit with a critical path delay only marginally higher than that from Design Compiler, while achieving a 43.09% reduction in area, and a 55.44% reduction in transistor count.

Circuit	<i>Design Compiler Delay Minimization</i>			<i>Developed Synthesis Tool Delay Minimization</i>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	ΔArea	ΔTran. Count	ΔDelay
C432	2779	136.64	1392	2041	127.64	850	-36.16%	-63.76%	-7.05%
C499	7946	93.8	3722	4182	86.6	1756	-90.00%	-111.96%	-8.31%
C880	6206	78.29	2772	3268	96.6	1400	-89.90%	-98.00%	18.95%
C1355	9313	106.16	4452	4390	122	2172	-112.14%	-104.97%	12.98%
C1980	7029	141	3268	5581	125.5	2362	-25.95%	-38.36%	-12.35%
C2670	8982	100.8	4318	7332	138.81	3072	-22.50%	-40.56%	27.38%
C3540	12395	170.11	5452	8949	176.92	3614	-38.51%	-50.86%	3.85%
C5315	16859	121.65	7766	18909	131.75	7846	10.84%	1.02%	7.67%
C7552	20688	97.62	9436	24779	104.89	10316	16.51%	8.53%	6.93%
Avg.Δ - (Design Compiler vs. Developed Tool)							-43.09%	-55.44%	5.56%

Figure 6-24. DC Delay Minimization vs. Developed Tool Delay Minimization – H=100

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-25. From this figure, it can be seen that the developed tool is able to reduce the overall area-delay product by 34.86% with comparison to Design Compiler.

<u>Circuit</u>	<u>Δ Area×Delay</u>
C432	-45.76%
C499	-105.8%
C880	-53.91%
C1355	-84.6%
C1980	-41.5%
C2670	11.04%
C3540	-33.18%
C5315	17.68%
C7552	22.3%
Avg.Δ	-34.86 %

Figure 6-25. Design Compiler vs. Developed Tool Δ Area-Delay Product (Delay vs. Delay)

6.2.8 DC vs. Developed Tool Area Minimization (H = 100)

The developed synthesis tool was set to minimize area for each benchmark circuit by minimizing the area in each partition as described in Section 4.5.2. Similarly, Synopsys Design Compiler was set to minimize critical path delay in each benchmark circuit with high map effort. The results of these synthesis tools are presented in Figure 6-26. From this data, it can be seen that the developed tool produces a circuit with a critical path delay 5.72% higher than that from Design Compiler, while achieving a 51.72% reduction in area, and a 64.93% reduction in transistor count.

Circuit	<i>Design Compiler Delay Minimization</i>			<i>Developed Synthesis Tool Area Minimization</i>					
	Area	Delay	# Transistor	Area	Delay	# Transistor	ΔArea	ΔTran. Count	ΔDelay
C432	2779	136.64	1392	1677	123.95	696	-65.71%	-100.00%	-10.24%
C499	7946	93.8	3722	4134	89.12	1724	-92.21%	-115.89%	-5.25%
C880	6206	78.29	2772	2947	100.32	1212	-110.59%	-128.71%	21.96%
C1355	9313	106.16	4452	4342	124.48	2140	-114.49%	-108.04%	14.72%
C1980	7029	141	3268	5461	122.04	2314	-28.71%	-41.23%	-15.54%
C2670	8982	100.8	4318	7179	144.18	3050	-25.11%	-41.57%	30.09%
C3540	12395	170.11	5452	8318	171.64	3548	-49.01%	-53.66%	0.89%
C5315	16859	121.65	7766	18087	131.47	7584	6.79%	-2.40%	7.47%
C7552	20688	97.62	9436	23934	105.42	10162	13.56%	7.14%	7.40%
Avg.Δ - (Design Compiler vs. Developed Tool)							-51.72%	-64.93%	5.72%

Figure 6-26. DC Delay Minimization vs. Developed Tool Area Minimization – H=100

The area-critical path delay product was calculated for each benchmark circuit for the results obtained from Design Compiler and the developed synthesis tool. The difference between these two is visible in Figure 6-27. From this figure, it can be seen that the developed tool is able to reduce the overall area-delay product by 38.2% with comparison to Design Compiler.

<u>Circuit</u>	<u>Δ Area×Delay</u>
C432	-82.68%
C499	-102.3%
C880	-64.34%
C1355	-82.92%
C1980	-48.71%
C2670	12.53%
C3540	-47.69%
C5315	13.75%
C7552	19.96%
Avg.Δ	-42.49 %

Figure 6-27. Design Compiler vs. Developed Tool Δ Area-Delay Product (Delay vs. Area)

6.2.9 Comparison of Synthesis Algorithms

In all cases outlined above for all output loads, the developed synthesis tool outperformed Synopsys Design Compiler with respect to the average area-critical path delay product for the set of ISCAS benchmark circuits.

Figure 6-28 illustrates the area-delay product difference between Synopsys Design Compiler for delay minimization and the developed tool for both area and delay minimization for each circuit with an output load $2\times$ the input capacitance. In nearly all cases, the area minimization algorithm as described in Section 4.5.2 outperforms the delay minimization algorithm of Section 4.5.1.

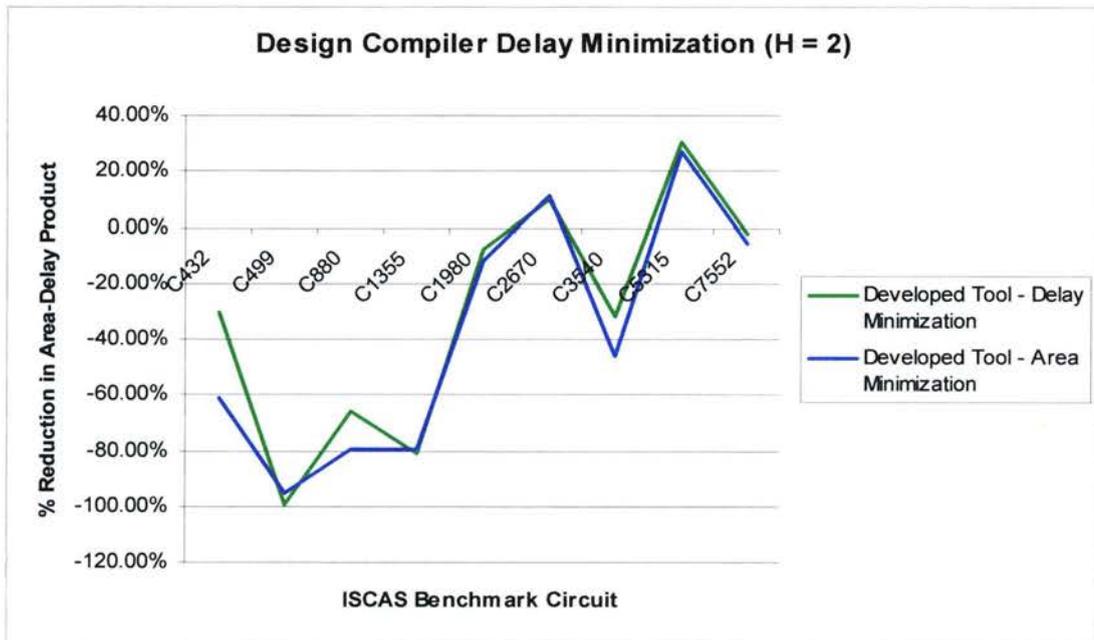


Figure 6-28. DC Delay Minimization (H=2)

Figure 6-29 illustrates the area-delay product difference between Synopsys Design Compiler for delay minimization and the developed tool for both area and delay minimization for each circuit with an output load $4\times$ the input capacitance. In nearly all cases, the area minimization algorithm as described in Section 4.5.2 outperforms the delay minimization algorithm of Section 4.5.1.

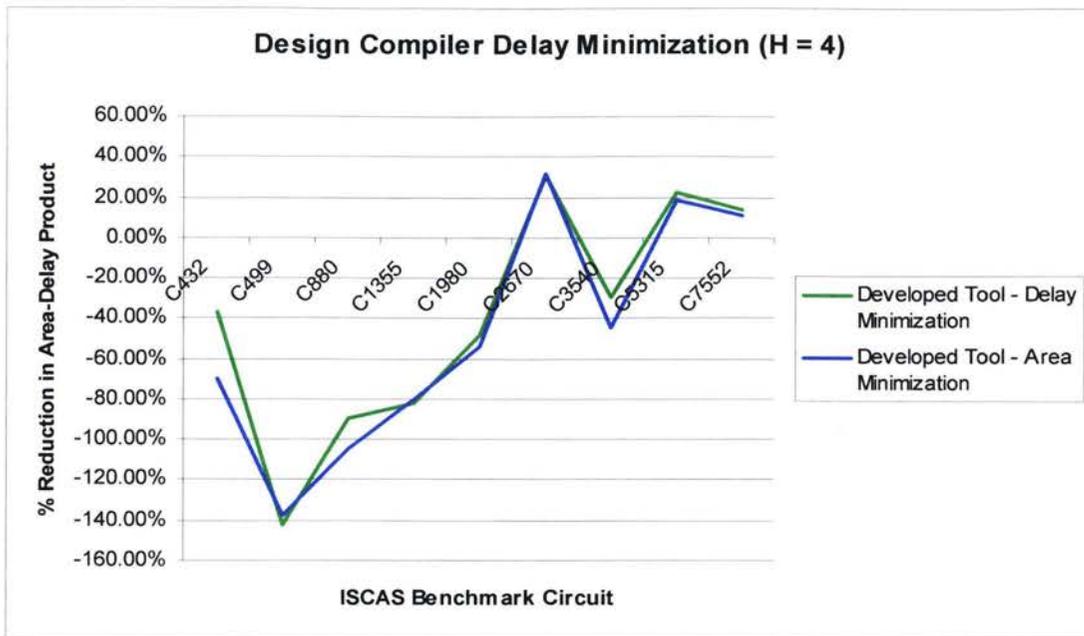


Figure 6-29. DC Delay Minimization (H=4)

Figure 6-30 illustrates the area-delay product difference between Synopsys Design Compiler for delay minimization and the developed tool for both area and delay minimization for each circuit with an output load $4\times$ the input capacitance. In nearly all cases, the area minimization algorithm as described in Section 4.5.2 outperforms the delay minimization algorithm of Section 4.5.1.

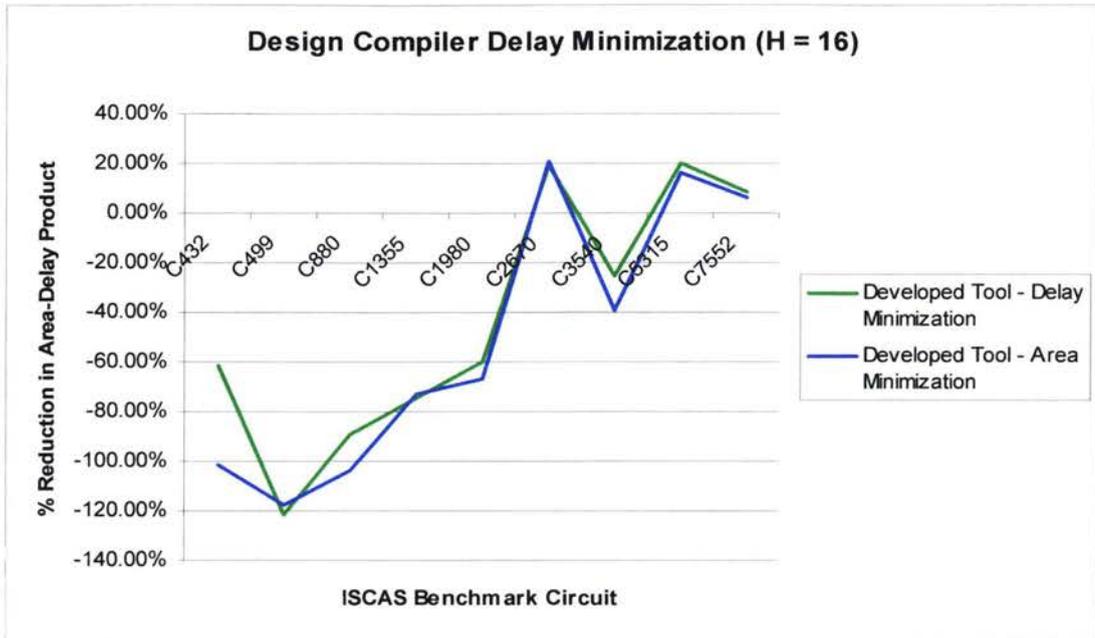


Figure 6-30. DC Delay Minimization (H=16)

Figure 6-31 illustrates the area-delay product difference between Synopsys Design Compiler for delay minimization and the developed tool for both area and delay minimization for each circuit with an output load $4\times$ the input capacitance. In nearly all cases, the area minimization algorithm as described in Section 4.5.2 outperforms the delay minimization algorithm of Section 4.5.1.

From the results presented above, it can be seen that the proposed area minimization algorithm as described in Section 4.5.2 outperforms the delay minimization algorithm of Section 4.5.1. There are certain circuits however, which neither algorithm is able to outperform the results obtained from Synopsys Design Compiler. These benchmark circuits are C2670 and C5315. This is due to the fact that Design Compiler is able to map XOR gates to the input HDL, rather than many smaller functions. The XOR gate is not a standard CMOS logic gate as the compliment of each input must be available – a two input XOR actually receives four inputs: input A, input B, and the compliments of both. Due to DAG partitioning, the developed tool will never be able to recognize these XOR gates. Similarly, the multiplexor (MUX) is not a standard CMOS logic gate due to the

fact that many of the drains of the transistors in the gate are not directly connected to the output, but rather to the gates of other internal transistors to the logic gate. Design Compiler is able to use MUX and XOR gates from its cell library – as such, benchmark circuit 2670 has 30% by area utilization XOR circuits. Similarly, C5315 has 43% by area MUX and XOR circuits. This is much higher than the other benchmark circuits such as C3540 which has 17% area XOR gates which the developed tool outperforms.

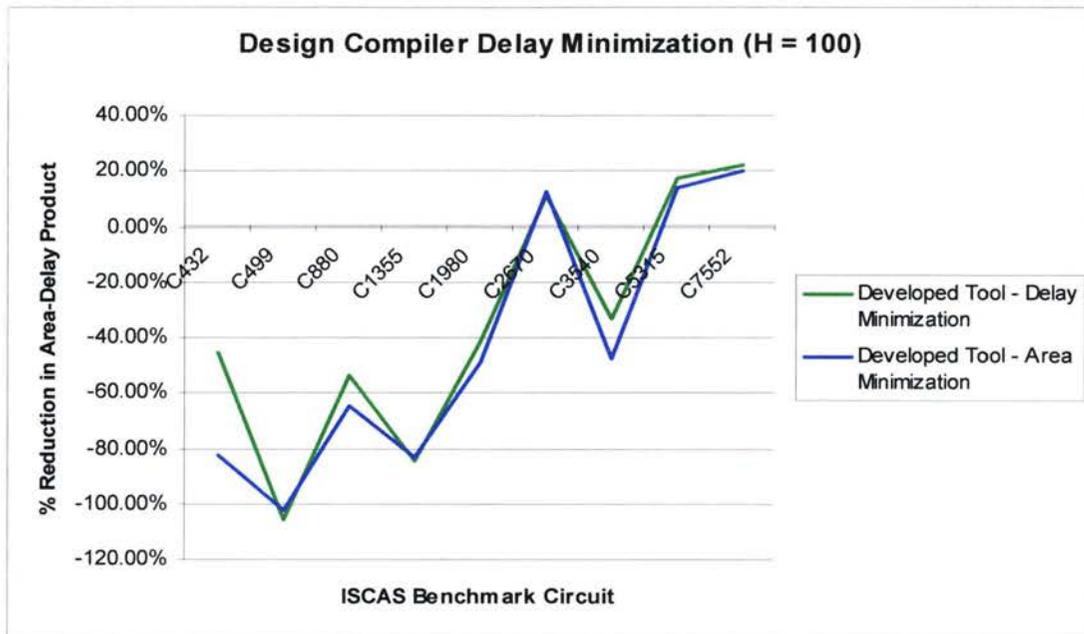


Figure 6-31. DC Delay Minimization (H=100)

2.3 Summary

This chapter outlined the results obtained from executing the developed synthesis tool over a set of benchmark circuits. Two problematic benchmark circuits were identified, and reasons for the lack of improvement were suggested. The following chapter will discuss conclusions and future work.

7. Conclusions and Future Work

This thesis has proposed new matching and covering algorithms for use in library-free synthesis which produce a cover minimizing path delay as in logical effort theory or overall circuit area by minimizing input capacitance to gates. The results of these algorithms are independent of the initial decomposition of the circuit through the use of Boolean matching. Due to runtime constraints, the tool is currently limited to matching functions with up to 12 inputs. This 12 input limitation will allow for covering with gates with up to 24 transistors. With this constraint, most benchmark circuits were able to complete in a few minutes. Due to the exponential nature of the problem being solved, this time quickly grows beyond 12 inputs. In the future, given advances in processing capabilities, improved minimization, and matching algorithms, this value will allow matching functions with higher number of inputs.

The most significant improvement in results with comparison to Synopsys Design Compiler comes when the developed tool is set to minimize overall circuit area, and Design Compiler is set to minimize critical path delay. From Figure 6-22, it can be seen that for a marginal delay increase of 5.86%, an area reduction of 61.81% can be realized, yielding an average area-delay product improvement of 51.06%.

The area of the benchmark circuits is calculated by summing the input capacitances, C_{in} , to each transistor of each gate in the mapped circuit. The dynamic power consumed by a circuit results from the charging and discharging of input capacitances to gates as they switch [28]. Dynamic power can be calculated by (7-1), where α represents the switching activity, C is the sum of all input capacitances to the circuit, V_{DD} is the supply voltage, and f is the frequency of the switching. The switching activity for general circuits is in the range of 0.4 – 0.5 [29].

$$P_{dynamic} = \alpha C V_{DD}^2 f \quad (7-1)$$

A reduction in input capacitance of 61.81% implies a proportional reduction in dynamic power by the same amount. As such, the area minimization algorithm on average will reduce circuit area and dynamic power consumption by 61.81% while suffering a 5.86% increase in critical path delay with respect to Synopsys Design Compiler with high map effort for delay minimization. This reduction in area and power may be very useful in the development of small, low power mobile devices such as in cellular phones or wireless sensors. Other work has been proposed [30] which attempts to synthesize to a transistor netlist optimizing delay by minimizing the input capacitances of the switching transistors.

There is much potential in the area of library-free synthesis to improve the efficiency of circuits produced on existing technology processes, and to ease the transition to new processes. A virtual cell library is not tied to any particular technology; as such, circuits may be synthesized using a virtual library to any technology process without the need to recreate the individual cells. The developed tool may be improved by expanding the capability of the matching algorithm to match functions with more than 12 inputs, allowing for much more complex gates to be used in the final cover.

8. References

- [1] Noyce, R.N., *U.S. Patent No. 2,981,877 USA*, 1961.
- [2] S. Thompson, M. Alavi, M. Hussein, P. Jacob, C. Kenyon, P. Moon, M. Prince, S. Sivakumar, S. Tyagi, M. Bohr., "130nm Logic Technology Featuring 60nm Transistors, Low-K Dielectrics, and Cu Interconnects." *Intel Technology Journal*, 2002, Issue 02, Vol. 06. 1535766X.
- [3] B. Guan, C. Sechen., "Large standard cell libraries and their impact on layout area and circuit performance." Austin : IEEE, 1996. *International Conference Computer Design, VLSI in Computers and Processors*. pp. 378-383. 0-8186-7554-3 .
- [4] T. Sakurai, A.R. Newton., "Delay analysis of series-connected MOSFET circuits." *IEEE Journal of Solid-State Circuits*, February 1991, Issue 2, Vol. 26, pp. 122-131. 0018-9200.
- [5] J. Togni, F.R. Schneider, V.P. Correia, R.P. Ribas, A.I. Reis., "Automatic generation of digital cell libraries." 2002. *15th Symposium on Integrated Circuits and Systems Design*. pp. 265-270. 0-7695-1807-9.
- [6] M. Rawski, Z. Jachna, I. Brzozowski, R. Rzechowski., "Practical Aspects of Logic Synthesis Based on Functional Decomposition." Warsaw : s.n., 2001. *Euromicro Symposium on Digital Systems Design*. pp. 38-45. 0-7695-1239-9.
- [7] Keutzer, K., "DAGON: Technology Binding and Local Optimization by Dag Matching." 1987. *24th Conference on Design Automation*. pp. 341-347. 0-8186-0781-5.
- [8] M.J. Chung, K. Sangchul., "A path-oriented algorithm for the cell selection problem." *Computer-Aided Design of Integrated Circuits and Systems*, IEEE Transactions, 1995, Issue 3, Vol. 14, pp. 296-307. 0278-0070.
- [9] Mailhot, F., *Technology Mapping For VLSI Circuits Exploiting Boolean Properties and Operations*. Stanford : Stanford University, 1994.
- [10] Detjens, E., "Technology Mapping in MIS." Los Alamitos : IEEE Computer Society Press, 1987. ICCAD. pp. 116-119.
- [11] L. Lavagno, G. Martin, L. Scheffer., *Electronic Design Automation for Integrated Circuits Handbook*. 2006. 0849330963.
- [12] Micheli, G., *Synthesis and Optimization of Digital Circuits*. s.l. : McGraw-Hill Science, 1994. 0-07-016333-2.

- [13] F. Mailhot, G. Micheli., "Technology mapping using boolean matching and don't care sets." Glasgow : s.n., 1990. *European Design Automation Conference*. 0-8186-2024-2.
- [14] D. Debnath, T. Sasao., "Efficient Computation of Canonical Form under Variable Permutation and Negation for Boolean Matching in Large Libraries." *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, s.l. : Oxford University Press, December 2006, Issue 12, Vols. E89-A. 0916-8508.
- [15] J. Ciric, C. Sechen., "Efficient canonical form for Boolean matching of complex functions in large libraries." San Jose : s.n., 2001. pp. 610-617. 0-7803-7247-6.
- [16] Biswas, N.N., "Minimization of Boolean Functions." *IEEE Transactions on Computers*, August 1971, Issue 8, Vols. C-20, pp. 925- 929. 0018-9340.
- [17] Kahramanli, S. Gunes, S. Sahan, S. Basciftci, F., "A New Method Based on Cube Algebra for the Simplification of Logic Functions." *Arabian Journal for Science and Engineering*, Konya,Turkey : King Fahd University of Petroleum & Minerals, April 2007, Issue 1B, Vol. 32, pp. 101-114. 0377-9211.
- [18] A. Reis, M. Robert, R. Reis., "Topological Parameters for Library Free Technology Mapping." Rio de Janeiro : s.n., 1998. *Brazilian Symposium on Integrated Circuit Design*. 0-8186-8704-5.
- [19] Reis, A., "Covering strategies for library free technology mapping." Natal : s.n., 1999. *Symposium on Integrated Circuits and System Design*. 0-7695-0387-X.
- [20] Y. Jiang, S. Sapatnekar, C. Bamji., "Technology mapping for high performance static CMOS and pass transistor logic designs." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, s.l. : IEEE Educational Activities Department, 2001, Issue 5, Vol. 9. 1063-8210.
- [21] V.Correia, A. Reis., "Advanced technology mapping for standard-cell generators." Pernambuco : s.n., 2004. *17th symposium on Integrated circuits and system design*. pp. 254-259. 1-58113-947-0.
- [22] F. Marques, L. Rosa, R. Ribas, S. Sapatnekar, A. Reis., "DAG based library-free technology mapping." Stresa-Lago Maggiore : ACM, 2007. *17th ACM Great Lakes symposium on VLSI*. pp. 293-298. 978-1-59593-605-9.
- [23] S. Gavrilov, A. Glebov, S. Pullela, S.C. Moorehoudhury, R. Panda, G. Vijayan, D.T. Blaauw A. Dhar., "Library-less synthesis for static CMOS combinational logic circuits." San Jose : IEEE Computer Society, 1997. *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*. pp. 658 - 662. 0-8186-8200-0 .

- [24] L. Stok, M. Iyer, A. Sullivan., "Wavefront technology mapping." Munich : s.n., 1999. *Design, Automation, and Test in Europe*. 1-58113-121-6 .
- [25] I. Sutherland, B. Sproull, D. Harris., *Logical effort: designing fast CMOS circuits*. s.l. : Morgan Kaufmann Publishers Inc., 1999. 1-55860-557-6.
- [26] S. Karandikar, S. Sapatnekar., "Logical Effort Based Technology Mapping." s.l. : IEEE Computer Society, 2004. *2004 IEEE/ACM International conference on Computer-aided design*. pp. 419 - 422. 0-7803-8702-3 .
- [27] O. Gjermundnes, E.J. Aas., *Design of a path delay fault simulator for evaluation of abist generated stimuli*. 2005. 0-7803-9345-7.
- [28] N. Weste, D. Harris., *CMOS VLSI Design*. s.l. : Addison-Wesley, 2005. 0-321-14901-7.
- [29] J.M. Rabaey, M. Pedram (Eds.), *Low Power Design Methodologies*. s.l. : Kluwer Academic Publishers, 1995. 978-0-7923-9630-7.
- [30] D. Kagaris, T. Haniotakis., "Transistor-Level Synthesis for Low-Power Applications." Carbondale : s.n., 2007. *8th International Symposium on Quality Electronic Design*. pp. 607-612. 0-7695-2795-7.
- [31] R.F. Pease, S.Y. Chou., "Lithography and Other Patterning Techniques for Future Electronics." *Proceedings of the IEEE*. 2008, Vol. 96, 2.
- [32] Kabbani, A. Al-Khalili, D. Al-Khalili, A.J., "Logical path delay distribution and transistor sizing." Toronto : s.n., 2005. *The 3rd International NEWCAS Conference*. pp. 391- 394. 0-7803-8934-4.
- [33] Kabbani, A., Al-Khalili, D. and Al-Khalili, A.J., "Delay analysis of CMOS gates using modified logical effort model." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005, Issue 6, Vol. 24, pp. 937 - 947.
- [34] Xue, Jingyue, Al-Khalili, D. and Rozon, C.N., "Tree-based transistor topology extraction algorithm for library-free logic synthesis." Kingston : s.n., 2004. *IEEE International Conference on Semiconductor Electronics*. 0-7803-8658-2.
- [35] Jain, Alok and Bryant, R.E., "Switch-Level Technology Mapping and Modeling." 1991. *Design Automation Conference*. pp. 219 - 222.
- [36] Sechen, C., "Libraries: LifeJacket or Straitjacket." 2003. *Design Automation Conference*. pp. 642- 643. 1-58113-688-9.

- [37] L. Rung-Bin, I. Chou, T. Chi-Ming., "Benchmark Circuits Improve the Quality of a Standard Cell Library." Wanchai : s.n., 1999. *Design Automation Conference*. pp. 173-176. 0-7803-5012-X.
- [38] Scott, K. and Keutzer, K., "Improving cell libraries for synthesis." 1994. pp. 128 - 131.
- [39] T.H. Kim, Y.H. Kim., "DEMI: A Delay Minimization Algorithm for Cell-Based Digital VLSI Design." *IEICE Trans Fundam Electron Commun Comput Sci*, 1999, Issue 3, Vols. E82-A, pp. 504-511. 0916-8508.
- [40] S. Lin, M. Marek-Sadowska, E. S. Kuh., "Delay and area optimization in standard-cell design." Orlando : ACM, 1991. *Annual ACM IEEE Design Automation Conference*. pp. 349-352. 0-89791-363-9 .