GENERALIZED BUFFERING OF PASS TRANSISTOR LOGIC (PTL) STAGES USING BOOLEAN DIVISION AND DON'T CARES

A Thesis

by

RAJESH GARG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2006

Major Subject: Computer Engineering

GENERALIZED BUFFERING OF PASS TRANSISTOR LOGIC (PTL) STAGES

USING BOOLEAN DIVISION AND DON'T CARES

A Thesis

by

RAJESH GARG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Sunil P. Khatri
Committee Members,	Gwan Choi
	Hong Liang
Head of Department,	C. Georghiades

May 2006

Major Subject: Computer Engineering

ABSTRACT

Generalized Buffering of Pass Transistor Logic (PTL) Stages Using Boolean Division and Don't Cares. (May 2006)

Rajesh Garg, B.Tech., Indian Institute of Technology-Delhi

Chair of Advisory Committee: Dr. Sunil P. Khatri

Pass Transistor Logic (PTL) is a well known approach for implementing digital circuits. In order to handle larger designs and also to ensure that the total number of series devices in the resulting circuit is bounded, partitioned Reduced Ordered Binary Decision Diagrams (ROBDDs) can be used to generate the PTL circuit. The output signals of each partitioned block typically needs to be buffered. In this thesis, a new methodology is presented to perform *generalized* buffering of the outputs of PTL blocks. By performing the Boolean division of each PTL block using different gates in a library, we select the gate that results in the largest reduction in the height of the PTL block. In this manner, these gates serve the function of buffering the outputs of the PTL blocks, while also reducing the height and delay of the PTL block.

PTL synthesis with generalized buffering was implemented in two different ways. In the first approach, Boolean division was used to perform generalized buffering. In the second approach, compatible observability don't cares (CODCs) were utilized in tandem with Boolean division to simplify the ROBDDs and to reduce the logic in PTL structure. Also CODCs were computed in two different manners: one using $full_simplify$ to compute complete CODCs and another using, approximate CODCs (ACODCs).

Over a number of examples, on an average, generalized buffering without CODCs results in a 24% reduction in delay, and a 3% improvement in circuit area, compared to a traditional buffered PTL implementation. When ACODCs were used, the delay was

reduced by 29%, and the total area was reduced by 5% compared to traditional buffering. With complete CODCs, the delay and area reduction compared to traditional buffering was 28% and 6% respectively. Therefore, results show that generalized buffering provides better implementation of the circuits than the traditional buffering method.

To my parents, brother, sister and my nephew

ACKNOWLEDGMENTS

I am very grateful to my advisor Dr. Sunil P. Khatri, for giving me this opportunity to work under him. Without his constant guidance, suggestions and encouragement, this work would not have been possible. I owe him gratitude for showing me this way of research. He has supported and encouraged me whenever I needed him and answered all my questions very openly. The informal group meetings organized by him have been a constant source of knowledge and inspiration. I also want to thank him for all the facilities and support he has given to me. Thanks a lot Dr. Khatri for everything.

I would also like to express my sincere acknowledgment to Nikhil Jayakumar and Kanupriya Gulati. Their constant support, valuable comments and guidance have helped me throughout the course of my Master's study. They have also helped me in learning new things, given time to discuss the problems and has been a source of inspiration all along.

I would also like to thank my parents, brother and sister, who taught me the value of hard work by their own example. I would also like to share my moment of happiness with them. Without their encouragement and confidence in me, I would have never been able to pursue and complete my Master's study.

Finally, I would like to thank all my friends, who directly and indirectly supported and helped me in completing this thesis.

TABLE OF CONTENTS

CHAPTER		Page
I	INTRODUCTION	1
	A. Previous Work	4
	B. Motivation	8
	C. Thesis Organization	10
II	BACKGROUND	11
	A. Pass Transistors Logic (PTL)	11
	B. Reduced Ordered Binary Decision Diagrams (ROBDD)	13
	C. Compatible Observability Don't Cares (CODCs)	17
	D. Conclusion	21
III	PARTITIONED ROBDDS	22
	A. Introduction	22
	B. Algorithm	23
	C. Conclusion	24
IV	PTL WITH GENERALIZED BUFFERING	26
	A. Introduction	26
	B. Boolean Division	26
	C. Generalized Buffering without CODCs	27
	1. ROBDD Division without CODCs	30
	D. Generalized Buffering with CODCs	34
	1. ROBDD Division with CODCs	36
	E. Conclusion	38
V	EXPERIMENTAL RESULTS	39
	A. Implementation Details	39
	B. Gate Library	40
	C. Delay	40
	D. Area	43
	E. Runtime	45
	F. Library Gates Utilized	45
	G. Conclusion	47

CHAPTER Page

VI CONCLUSIONS AND FUTURE WORK 53

A. Conclusions 53
B. Future Work 54

REFERENCES 56

VITA 60

LIST OF TABLES

TABLE Page

I Delay and active area of gate library 41

II Delay Comparison of Generalized and Traditional Buffered PTL 42

III Area Comparison of Generalized and Traditional Buffered PTL 44

IV Run-time for PTL synthesis with generalized buffering 46

V Number of MUXes and INV Utilized during Traditional and Generalized Buffering 48

VI Number of Library Gates Utilized during Generalized Buffering without CODCs 49

VII Number of Library Gates Utilized during Generalized Buffering with ACODCs 50

VIII Number of Library Gates Utilized during Generalized Buffering with CODCs 51

LIST OF FIGURES

FIGURE Page

1 ROBDD Node and its MUX based Implementation 2

2 ROBDD and its PTL Implementation 3

3 Static and PTL based implementation of AND Gate 12

4 Shannon Cofactoring tree of logic function $(x_1 + x_2) \cdot x_3$ 14

5 OBDD of logic function $(x_1 + x_2) \cdot x_3$ 15

6 ROBDD for logic function $(x_1 + x_2) \cdot x_3$ 16

7 Node y_k and its fanins 19

8 Partitioned ROBDD of a Circuit (traditional buffering) 24

9 Back-tracking during Division 28

10 Dividing a Generalized Buffer into a PTL Structure 31

11 Partitioned ROBDD with generalized buffering 33

CHAPTER I

INTRODUCTION

Static complementary metal oxide semiconductor (CMOS) has long been the circuit design style of choice for digital Integrated Circuit (IC) designers. Some the reasons for this are that static CMOS enables the design of reliable, scalable circuits and also due to the availability of a well-developed synthesis methodology for static CMOS. However, the switching capacitances in a static CMOS circuit can be fairly large. With the increasing demand for higher speed and low power, it has become necessary to look for alternative design styles which can offer better performance and power than static CMOS. The shrinking process feature sizes and increasing transistor counts on a chip further emphasize this need. Many alternate circuit design styles have been proposed over the years, such as dynamic CMOS logic [1, 2], pass-transistor logic (PTL) [2], differential cascode voltage switch logic (DCVSL) [2], programmable logic arrays (PLAs) [2], etc.

Among these alternate circuit design styles, pass transistor logic (PTL) offers great promise. In contrast to dynamic logic, PTL is less susceptible to cross talk problems, which is a major issue in deep sub-micron technologies [3]. Several case studies [4, 5] have shown that PTL can implement most functions with fewer transistors than static CMOS or other styles. PTL circuits also have a physically dense structure and hence occupy less area. This reduces the overall capacitance, resulting in faster switching times and lower power.

PTL is a circuit implementation style that has typically been used for many specific circuit blocks like barrel shifters. Although PTL offers great benefits for such designs, there has been *no widely accepted PTL design methodology* that could be used to make PTL more broadly acceptable. There have however been several efforts at the research

---

The journal model is *IEEE Transactions on Automatic Control.*

level, to explore the promise of PTL.

Synthesis approaches for PTL structures typically leverage the fact that there is a direct mapping between the Reduced Ordered Binary Decision Diagram (ROBDD) [4, 6] and the PTL implementation of a circuit. In fact, each ROBDD node can be mapped to a MUX (which can be implemented using NMOS or CMOS devices). Figure 1 illustrates the mapping between an an ROBDD node and a MUX whereas Figure 2 shows the mapping of ROBDD into PTL. In Figure 2 the solid line (dashed line) represents the positive (negative) cofactor of an ROBDD node with respect to the variable of that node. For the PTL implementation of any function, there is an isomorphism between the connectivity of the ROBDD nodes of the function and the MUXes in the PTL implementation. This elegant and easy mapping between ROBDDs and PTL structures is not without attendant problems, however:

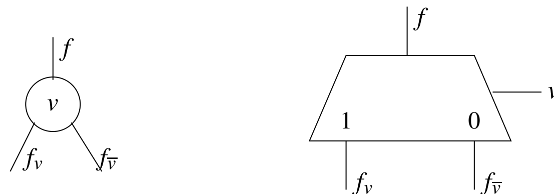

Figure 1. ROBDD Node and its MUX based Implementation

- For one, in a bulk MOS implementation of any circuit, it is not practical to connect more than 4-5 devices in series, due to the phenomenon called *body effect*, which effectively increases the threshold voltage V_T of most of the series-connected MOSFETs. This results in a slower design. Body effect is governed by the equation

$$V_T = V_T^0 + \gamma\sqrt{V_{sb}} \tag{1.1}$$

Here V_T is the threshold voltage of the device, V_T^0 is the threshold voltage of the

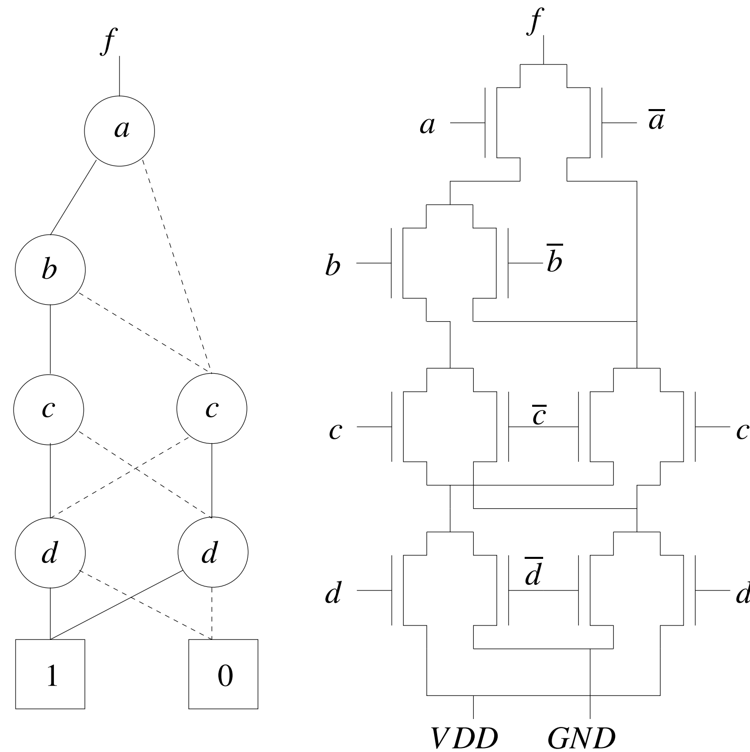

Figure 2. ROBDD and its PTL Implementation

device at zero body bias, γ is the body effect coefficient, and V_{sb} is the source to bulk voltage of the MOSFET. When several MOSFETs are connected in series in a PTL circuit, all but one of them is affected by body effect. For this reason, in practice, VLSI designers do not stack more than 4-5 devices in series.

This results in a problem for the traditional PTL implementations, which attempt to build monolithic ROBDDs. To solve this problem, we need to build ROBDDs in a partitioned [7] manner, such that if an intermediate ROBDD has a depth greater than 4 or 5, a new variable is created. In circuit terms, a buffer is implemented at the output of the new variable, ensuring that the circuit drive capability is regenerated every few levels in the final PTL design.

- The second problem is that ROBDDs, if build monolithically, can exhibit unpredictable memory explosion. This precludes the applicability of the PTL methodology for large designs, *as long as the ROBDDs are build monolithically* (since it is very hard to build monolithic ROBDDs of large functions). Again, the use of partitioned ROBDDs averts this problem.

In summary, partitioned ROBDD construction helps tackle both the above problems associated with PTL synthesis. In such a partitioned PTL design, the outputs of each PTL structure is buffered, to regenerate the electrical drive capability after every 4 or 5 levels.

The main goal of this thesis work is to implement a new PTL synthesis approach to perform efficient partitioned ROBDD based PTL synthesis. In principle, it still employs partitioned ROBDDs, however, the buffering between PTL stages is done using *generalized buffers*. These could be arbitrary gates in the cell library. This is achieved by casting the problem of generalized buffering as an instance of Boolean division. In traditional partitioned ROBDD based PTL synthesis, buffering is required after 4-5 levels as mentioned earlier. By using functionally complex library gates for the buffering logic, it is possible to significantly reduce the logic in the PTL structure which then results in a reduction of total circuit delay and area. A second goal of this thesis is to utilize Compatible Observability Don't Cares (CODCs) during Boolean Division, to further reduce the logic in the PTL structures. The PTL synthesis approach with generalized buffering is implemented for both cases: one using CODCs and the other without CODCs. Also, to augment the applicability of the approach to industrial designs, approximate CODCs (ACODCs) are used as well.

A. Previous Work

There has been a significant amount of work in the area of pass transistor logic synthesis. In addition, there have been several reported design variations on the PTL circuit concept.

A good reference source for published work in this area is [8].The paper gives an overview of the state of the art in PTL technology. This paper covers PTL based circuit technologies, design and synthesis methodologies, applications and commercial use, etc. Given the general nature of the synthesis methodology which is presented in this thesis, the work of this thesis is orthogonal to the circuit issues and ideas that are discussed in the papers referred to in [8].

In [5], Yano et. al. present a scheme which takes a logic function described in hardware description language (HDL) format, and synthesize the corresponding PTL based circuit using a library of PTL cells, macrocells, etc. The scheme which was named as Lean Integration with Pass-Transistors (LEAP) uses a synthesis tool called "circuit inventor" which express the required logic function in ROBDD. Then the ROBDD is partitioned into smaller trees by inserting buffers at the nodes such that the height of the partitioned tree is less than the maximum height of the pass-transistor cell in the library. The library cells used [5] have a maximum height of two variables. After inserting the buffers, the partitioned tree is mapped using the PTL cells from the library. As monolithic ROBDDs are constructed (these can result in memory explosion), this tool can only be used for small circuits. This method uses inverters to buffer the signals between PTL cells.

The philosophy behind the work of Macchiarulo et. al. [9] is that while PTL based synthesis is well researched, layout and back-end tools for PTL are not found as readily. They describe a layout generator to help develop more complete tools for library free PTL design. The approach of Radhakrishnan et. al. [10] is motivated by the need to develop high density, low power, high performance circuits using PTL. The authors presents two synthesis approaches – a modified Karnaugh map [11] approach for small circuits, and a Quine-McCluskey [12, 13] based approach for larger designs. In [14], Cheung et. al. study regenerative pass transistor logic (RPL), a dual rail PTL family. Their interest stems from the compact area of these circuits. They implement an algorithm which tries to use

2-input RPL XOR logic gates and 2-input multiplexers as much as possible, to minimize the total number of transistors in the circuit. The algorithm also tries to minimize the serial connection of the pass-transistors to reduce *body* effect.

In [15], Hsiao et. al. present a layout and logic synthesis approach, with the input being a logic function, while the output is a PTL net-list, containing MUXes and inverters. After synthesizing the logic, the methodology adds inverters or buffers, in order to ensure that the longest series path of MUXes has a bounded depth. The work of Buch et.al. [3] builds ROBDDs [4, 6] in a partitioned manner [7], thereby avoiding the memory blow-up that often occurs while using ROBDDs. The resulting (small) ROBDDs are directly mapped to PTL structures, resulting in an efficient synthesis methodology. The downside of this approach is the absence of generalized buffering. The authors allude to performing PTL synthesis in a manner that maps some nodes to CMOS gates while others are mapped into PTL blocks. However, this leaves the designer little control of the depth of these sections. In contrast, the generalized buffering approach presented in this thesis guarantees a fixed bound on the depth of each PTL block, and also ensures just one level of generalized buffers between PTL blocks.

In [16], an approach using multilevel gates along with PTL and transmission gates is reported by Neves et. al., producing a regular and dense layout. The algorithm implemented by them first minimizes the logic function using a multi-level logic minimizer. Then the multi-level format is translated into a Directed Acyclic Graph (DAG). The DAG is then mapped into network of gates using the multi-level gates implemented in their library. Their approach also permits buffer inclusion. It can be viewed as an approach that combines synthesis and layout in the PTL design flow, and is orthogonal to the work presented in this thesis. The work of Pedron et. al. in the paper [17] reports synthesis algorithms for PTL. Minimization of circuit area is performed by using incomplete transmission gates. The results of synthesis using their tool PAVOS demonstrate good quality results when

compared with manually tuned PTL circuits.

In [18], Markovic et. al. present a PTL synthesis approach for Complementary Pass Transistor Logic (CPL), Dual Pass Transistor Logic (DPL) and Dual Voltage Logic (DVL). The approach for the logic synthesis is based on Karnaugh map coverage and circuit transformations. Finally, the work of Shelar et. al. [19] reports on a novel method to minimize power in a PTL structure. The authors transform the power minimization problem into a ROBDD decomposition problem and solve the latter using a max-flow min-cut approach. The results of the proposed algorithm in their paper achieves good power reduction as compared to previous low power PTL synthesis algorithms.

In [20], a mixed PTL-CMOS approach was presented by Lai et. al. The decomposition process simply extracted unate variables from a sum-of-products (SOP) representation of a function and recursively cofactored these variables (creating MUXes in the process) until unate leaves were reached. Unate leaves were realized using library gates alone. The weakness of this approach is that it starts with a SOP representation, limiting its effectiveness for large designs. Further, if a function is unate, then the entire circuit realization is purely standard-cell based. The work presented in this thesis works equally effectively for unate designs because it utilizes the partitioned ROBDD construction as the first step.

Another mixed PTL-CMOS approach was presented in [21]. The approach adopted by the authors is to construct the ROBDD of a logic function and then replace part of it with CMOS gates. The ROBDD nodes with one input connected to 0 or 1 terminal node are favored for replacement with CMOS gates. A cost function is used to finally decide upon the node replacement. The authors also control the ratio of the pass-transistors and CMOS gates by modifying a parameter of the cost function, and hence are able to do area-oriented, delay-oriented and power-oriented PTL-CMOS synthesis. However, the algorithm does not provide the user any control over the number of pass-transistors between two CMOS gates. Also, monolithic BDDs are used as a starting point for the approach, hence the algorithm

can only be applied to the smaller designs.

Most of the previous PTL logic synthesis approaches utilize monolithic ROBDDs which makes them inapplicable for larger designs. Although some of them have used partitioned ROBDDs, they only use inverters for buffering signals between partitioned blocks. However these approaches employ ad-hoc heuristics, yielding PTL structures which do not have control over the number of pass-transistors between two CMOS gates [21]. Other approaches [20], do not allow any control over the ratio of PTL blocks to standard cells. Our Boolean division based approach explores *all possible ways* to use arbitrary library cells. It also guarantees that *static CMOS gates are used only in the interface between PTL blocks*, and that PTL blocks have bounded depth. Neither [21] nor [20] satisfy any of these properties. Also the methodology [20, 21] can only be applied to smaller designs because of their use of monolithic ROBDDs. Due to these problems associated with the previously proposed methods, there lies a significant scope for work in PTL based logic synthesis using CMOS gates. This thesis explores these opportunities.

B. Motivation

The shrinking feature size in recent VLSI designs and increasing demand for high speed and lower power makes it necessary to look for the circuit design styles other than Static CMOS, which have been predominantly used to implement VLSI circuits. Several different circuit design styles have been proposed such as dynamic CMOS logic, pass-transistor logic (PTL), differential cascode voltage switch logic (DCVSL), programmable logic arrays (PLAs), etc. Among these alternate circuit design styles, PTL circuit style possesses great potential. PTL can implement the logic function with fewer transistors than Static CMOS circuits. This reduction in transistors results in lower overall capacitance which increases design speed and decreases power consumption.

PTL approaches have been extensively researched earlier. However, there has been no widely accepted PTL design methodology. This has resulted in the application of PTL to specific circuit implementations. Many authors have tried to use PTL for general VLSI logic circuits but their approaches suffer from the problems as mentioned in the previous section. The key contribution of this thesis is to develop and implement a new PTL synthesis approach to perform efficient partitioned ROBDD based PTL synthesis. In principle, it still employs partitioned ROBDDs, however, the buffering between PTL stages is done using *generalized buffers*. These could be arbitrary gates in a cell library. In traditional partitioned ROBDD based PTL synthesis, buffering is required after 4-5 levels as mentioned earlier. By using functionally complex library gates for the buffering logic, it is possible to significantly reduce the logic in the PTL structure, which then results in a reduction of total circuit delay and area. The generalized buffering problem is solved using Boolean division. A second goal of this thesis is to utilize Compatible Observability Don't Cares (CODCs) during Boolean division, to further reduce the logic in the PTL structures. The PTL synthesis approach with generalized buffering is implemented for both cases: one using CODCs and the other without CODCs. Finally, this thesis also uses approximate CODCs (ACODCs) during Boolean division, in order to extend the applicability of generalized buffering to indvidual designs.

The new PTL synthesis algorithm implemented in this thesis can synthesize combinational logic circuits of arbitrary size. The approach of performing partitioned ROBDD construction using generalized buffers (library gates) to regenerate signal strengths across PTL structures is a novel contribution in the PTL synthesis world. The approach used for generalized buffering (using Boolean division), and the extension of this approach using CODCs and ACODCs is another novel contribution of this thesis.

C. Thesis Organization

The rest of this thesis is organized as follows: Chapter II provide some background information which will be helpful in understanding the PTL based logic synthesis approach presented in this thesis. In Chapter III partitioned ROBDD construction is described. Chapter IV describes the new PTL synthesis algorithms proposed in this thesis. In Chapter V, experimental results are provided, comparing the new PTL synthesis approach (using generalized buffering) with traditional partitioned ROBDD-based PTL design. Conclusions and future work are discussed in Chapter VI.

CHAPTER II

BACKGROUND

PTL possesses many advantages over static CMOS and other circuit styles. Among these properties of pass-transistors, the fact that there is direct mapping between an ROBDD and the PTL implementation of the corresponding function are mainly responsible for the potential that PTL possesses. To explain this further, it is useful to briefly describe pass transistors, ROBBDs, and compatible observability don't cares (CODCs).

A. Pass Transistors Logic (PTL)

Pass transistor logic is a popular and widely used alternative to Static CMOS because of the advantages it possesses in terms of area and power as compared to CMOS, and other circuit styles [10, 3]. The pass transistor is simply an NMOS or PMOS device with inputs driving the gate as well as the source-drain terminals of the MOSFETS. This is in contrast with other circuit styles, which only allow inputs to drive the gate terminal. As an NMOS device is faster than PMOS transistor of the same size therefore, mostly NMOS transistors are used commonly to implement pass transistors in the PTL.

The main advantage of PTL is that it requires fewer transistors to implement a given logic function. Figure 3 shows the PTL and static CMOS implementation of a 2-input AND function. It can be seen that PTL based implementation requires 3 NMOS and 1 PMOS transistors (including the inverter required to invert B) to implement the AND gate whereas the corresponding implementation in static CMOS would require 6 transistors. The reduction in the number of devices further decreases the overall capacitances and thereby increases the switching rate with a decrease in power.

The NMOS (PMOS) pass transistor is effective at passing a $GND(VDD)$, but it is poor at passing a $VDD(GND)$. Therefore, the output of a NMOS or PMOS pass-transistor

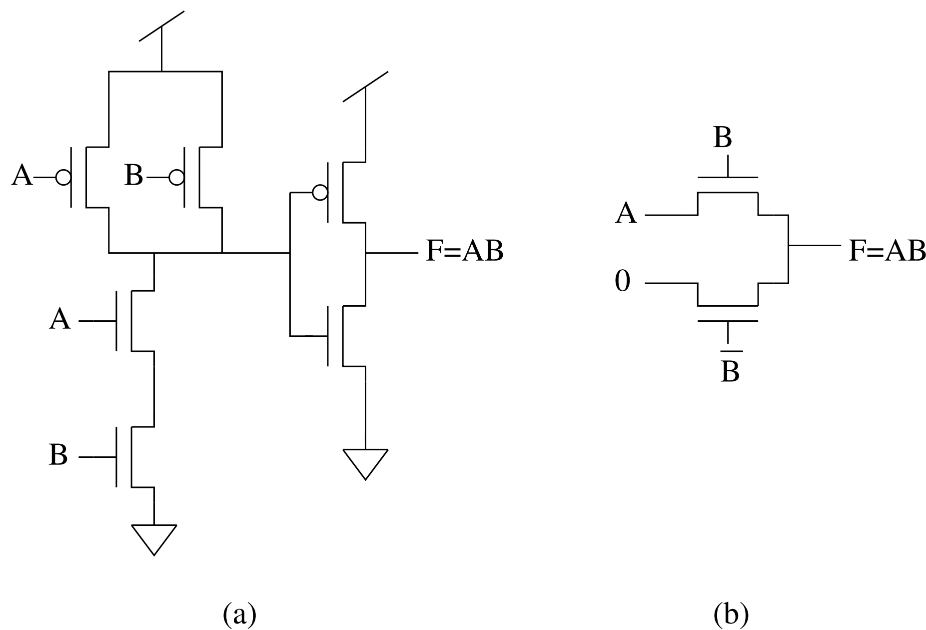

Figure 3. Static and PTL based implementation of AND Gate

is not at the supply voltage. Also in a bulk MOS implementation of a circuit, the bulk terminal of all NMOS (PMOS) transistors are connected to *GND* (*VDD*). Due to this, when the transistors are connected in series, all but one of them experience an increase in its threshold voltage due to *body effect*. The increase in threshold voltage is determined by the equation 1.1. The increase in threshold voltage increases the delay of the pass transistor (slows down the switching speed of the device). The NMOS transistor which is closer to *GND* in the series connected stack experiences a smaller increase in threshold voltage as compared to the transistor further away from *GND*. Therefore, in practice it is not advisable to connect more than 4-5 transistors in series.

As mentioned in the last chapter, that there is a direct mapping between ROBDDs and PTL structures. Therefore, buffers have to be used after every 4-5 ROBDD nodes in series. In other words, partitioned ROBDDs with a depth less than 5 have to be built and

then mapped to PTL structures. ROBDDs are briefly discussed in next section whereas the implementation of the partitioned ROBDDs are described in detail in the next chapter.

B. Reduced Ordered Binary Decision Diagrams (ROBDD)

An ROBDD is a graphical way of representing a Boolean function. It can represent many logic functions compactly as compared with sum-of product (SOP). Moreover, many logic operations like tautology and complementation can be performed on ROBDDs in constant time. For a particular variable ordering , an ROBDD is a canonical form of representing a Boolean function. As the name suggests ROBDDs are a reduced form of binary decision diagram (BDDs) with a particular variable ordering. The reduction rules used are described in the sequel.

A BDD represents a Boolean function as a directed acyclic graph (DAG), with each nonterminal node labeled by a variable of the function. It is also referred to as a Shannon co-factoring tree, where each node performs the Shannon co-factoring of the Boolean function represented by that node, with respect to the variable assigned to it. Figure 4 illustrates the BDD for the function $(x_1 + x_2) \cdot x_3$. Each node has two outgoing edges, corresponding to the positive cofactor of the node function with respect to the node variable (shown as a solid line) or the negative cofactor of the node function with respect to the node variable (shown as a dashed line). The terminal nodes (shown as boxes) are labeled with 0 or 1, corresponding to the possible function values. For any assignment to the function variables, the function value is determined by tracing a path from the root of the BDD to a terminal node following the appropriate positive or negative branch from each node. The number of vertices from the root to the leaves in the BDD is exponential in terms of number of variables in the logic function. Therefore, for functions with a large number of variables, BDDs might not be a good choice for representing the function. In general, the variable

ordering along different paths in the BDD can be different.

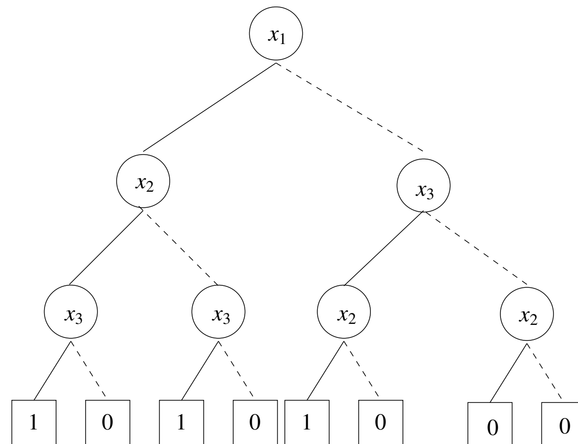

Figure 4. Shannon Cofactoring tree of logic function $(x_1 + x_2) \cdot x_3$

The graph in Figure 4 is transformed into an ordered BDDs (OBDDs) if we use a fixed variable ordering. Consider the variable to be order $x_1 < x_2 < x_3$. That is, every path from the root to a leaf encounters variables in the order $x_1 < x_2 < x_3$. The resulting OBDD is shown in Figure 5. If we further apply a set of reduction rules on the OBDD, we obtain an ROBDD for the function. The set of reduction rules are:

- Any node with two identical children is removed.
- Two nodes with isomorphic BDDs are merged.

ROBDDs are a canonical form of representing a logic function for a particular variable ordering. Figure 6 shows the resulting ROBDD when the above mentioned reduction rules are applied to the graph shown in Figure 5. In this case as well, the number of nodes can be exponential in terms of the number of variables. However, the size of ROBDDs (i.e. number of nodes) depends upon the variable ordering. Therefore, variables must be ordered in a manner that minimizes the size of the ROBDD. The problem of computing an optimum variable ordering is a NP-Complete problem. However there are efficient

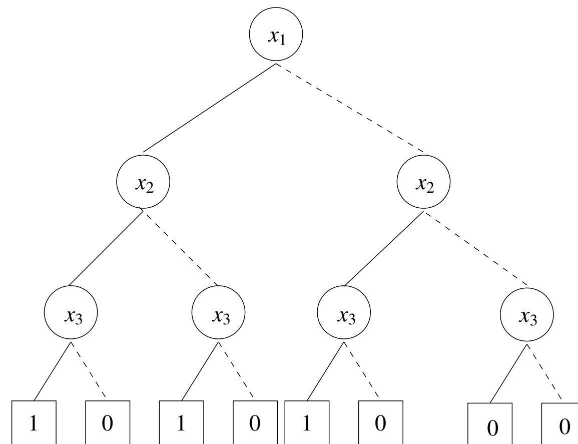

Figure 5. OBDD of logic function $(x_1 + x_2) \cdot x_3$

heuristics available that can choose an appropriate ordering of variables which results in the ROBDD of reasonable size. At the same time, there are functions that have polynomial sized multi-level representations while their ROBDDs are exponential for all input orderings. A multiplexer is an example of such a function.

The following ROBDD operations are used in the work presented in the thesis:

- *bdd_and* : This function returns the ROBDD which is the logical AND of the two ROBDDs passed to the function.
- *bdd_or* : This function returns the ROBDD which is the logical OR of the two ROBDDs passed to the function.
- *bdd_xnor* : This function returns the ROBDD which is the logical XNOR of the two ROBDDs passed to the function as a parameter.
- *bdd_compose*: This function returns a ROBDD F' with a variable v replaced by a ROBDD g in the original ROBDD F.
- *bdd_smooth* : Returns the BDD formula of f existentially quantified with respect to

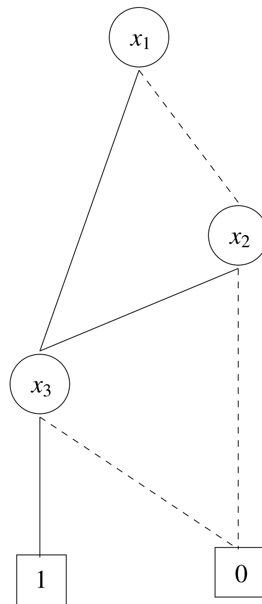

Figure 6. ROBDD for logic function $(x_1 + x_2) \cdot x_3$

the variables in the array “smoothing_vars”. “smoothing_vars” is an array of ROBDDs, which are the single variable BDD formulas to be quantified.

Existential quantification of f with respect to a variable x is expressed as $\exists_x f = f_x + f_{\overline{x}}$. Here f_x represents the function f co-factored with respect to variable x. Smoothing a set of variables is achieved by performing existential quantification, one variable at a time.

- *bdd_between* : Returns a heuristically minimized BDD containing f_{min}, and contained in f_{max}.

- *bdd_depth* : Returns the depth of ROBDD i.e. the maximum number of nodes from the root vertex to any leaf.

C. Compatible Observability Don't Cares (CODCs)

Technology independent logic optimization of a multi-level network is an important part of logic synthesis. During such optimization, one of the operations involves the computation of multi-level don't cares of the circuit. These don't cares can take the form of Satisfiability Don't Cares (SDCs), Observability Don't Cares (ODCs) or External Don't Cares (XDCs).

Of the different kinds of don't cares, XDCs are specified with the network whereas SDCs and ODCs are computed. Consider a multi-level boolean network (η) in which every node has a Boolean function f_i associated with it. Also, f_i has a corresponding Boolean variable y_i associated with it, such that $y_i \equiv f_i$. Suppose that the Boolean network η has n primary inputs and m nodes. Then the SDCs are defined by the equation 2.1. SDCs simply convey that for each node in the network, the value of the node output (y_j) cannot differ from the value obtained after evaluating its boolean function (f_j).

$$SDC = \Sigma_{j=1}^{m}(y_j\overline{f_j} + \overline{y_j}f_j) \tag{2.1}$$

The Observability Don't Care (ODC) of node y_j (in a multi-level Boolean network) with respect to output z_k is

$$ODC_{jk} = \{x \in B^n \ s.t. \ z_k(x)|_{y_j=0} = z_k(x)|_{y_j=1}\} \tag{2.2}$$

In other words ODC_{jk} is the set of minterms on the primary inputs for which the value of y_j is *not observable* at z_k [22]. This can also be denoted as

$$ODC_{jk} = (\overline{\frac{\partial z_k}{\partial y_j}})$$

where

$$\frac{\partial z_k}{\partial y_j} = z_k(x)|_{y_j=0} \oplus z_k(x)|_{y_j=1} \tag{2.3}$$

$\frac{\partial z_k}{\partial y_j}$ is also known as the Boolean Difference of z_k with respect to y_j.

Once a node function is changed by minimizing it against its ODCs (using Espresso [23]), the ODCs of the other nodes must be recomputed. To avoid re-computation of ODCs during optimization, Compatible Observability Don't Cares (CODCs) [22] were developed. The CODC of a node is a subset of the ODC for that node. Unlike ODCs, CODCs have a property that one can *simultaneously* change the function of all nodes in the network as long as each of the modified functions are contained in their respective CODCs. Compatibility is achieved by ordering the don't care computation in a manner that if node m_i precedes node m_j in the order, then node m_i gets the most don't cares. Also, while computing the don't cares of m_j, compatibility is maintained with the don't cares already assigned to node m_i.

The computation of CODCs [22] in SIS [24] is performed in two phases.

- The first phase computes the CODC of a node using node operations. The resulting CODC is a function of arbitrary nodes in the network. However, we desire the support of the CODC to be the support of the node itself. For this reason, we need a second phase in the computation to achieve this.

- In the second phase, image computation is performed to map the CODC points to the local fanin space of each node. This image computation uses BDDs.

In the first phase, the CODC computation for the network η starts from the primary outputs and proceeds towards primary inputs in a reverse topological order. The CODC at each primary output is initialized to the external don't care (XDC) at that node.

The CODC at node y_i (denoted as $CODC_i^{\eta}$) is found by using the CODC for each fanout edge e_{ik} of y_i. This compatible don't care of edge e_{ik} is denoted by $CODC_{ik}^{\eta}$. The CODC for y_i is obtained by intersecting the CODCs computed for its fanout edges.

Suppose, as shown in Figure 7, that we have a node with function $y_k = f_k$, and ordered fanins $y_1 < y_2 < \cdots < y_i$. Given $CODC_k^{\eta}$, we can compute the CODCs of the fanin edges

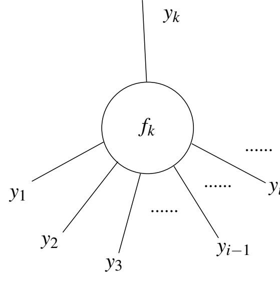

Figure 7. Node y_k and its fanins

of y_k as follows.

$$CODC^{\eta}_{ik} \equiv (\frac{\partial f_k}{\partial y_1} + C_{y_1})\ldots(\frac{\partial f_k}{\partial y_{i-1}} + C_{y_{i-1}})\overline{\frac{\partial f_k}{\partial y_i}} + CODC^{\eta}_{k} \tag{2.4}$$

Note that in the above computation, we assume that $CODC^{\eta}_{k}$ have already been computed. Also, C_{y_j} is the consensus[1] operator. For the first input in the ordered list of fanins, we have $CODC^{\eta}_{1k} = \overline{(\frac{\partial f_k}{\partial y_1})} + CODC^{\eta}_{k}$, indicating that this node obtains maximum flexibility. The intuition behind the correctness of the computation of Equation 2.4 in general is that the new edge e_{ik} should have its don't care condition as the conjunction of $\overline{\frac{\partial f_k}{\partial y_i}}$ with the condition that other inputs $j < i$ are not insensitive to the input y_j ($\frac{\partial f_k}{\partial y_j}$), or are independent of y_j (C_{y_j}) indicating that the node i is free to use such terms independently of how y_j was simplified. Finally, the CODCs of the fanout node y_k are also CODCs of the edge e_{ik}. As a result, the computation of $CODC^{\eta}_{ik}$ using the formula above, performed in the specified order, results in the Compatible set of ODCs of the edges e_{ik}. From $CODC^{\eta}_{ik}$, we compute the CODC for node i ($CODC^{\eta}_{i}$) as follows:

[1]Consensus operation on f with respect to a variable y is expressed as $C_y f = f_y f_{\overline{y}}$. Here f_y represents the function f co-factored with respect to variable y.

$$CODC_i^{\eta} \equiv \prod_{k \in FO_i} CODC_{ik}^{\eta} \tag{2.5}$$

The intuition of the above method of computing the CODC of y_i ($CODC_i^{\eta}$) based on its edge CODCs ($CODC_{ik}^{\eta}$) is that $CODC_i^{\eta}$ must not be greater than any $CODC_{ik}^{\eta}$. Note also that all terms in $CODC_{ik}^{\eta}$ from Equation 2.4 except $CODC_k^{\eta}$ have a support which is the support of y_k. However, $CODC_i^{\eta}$ has a support which includes its fanout's fanins, and in general, we have to do an image computation to convert this CODC to a function which is on the support of y_i.

In the second phase, we perform this image computation. Using the ordering heuristics of [25], global BDDs at each node of the Boolean network are computed in terms of the primary inputs. BDDs are also built for each primary output in the external don't care network using this same ordering. We next compute the CODC at y_i in terms of primary inputs, using a BDD based computation. This is done by substituting each literal of the CODC of y_i by its global BDD function (which was computed earlier). From this we find all the points that are reachable in the local space of y_i by a BDD based image computation. The functions used for image computation are the global functions at the fanins of y_i. In most cases the number of primary outputs is much less than the number of primary inputs; therefore the recursive image computation method [26] is used to do the computation. However, this computation is highly memory intensive, since it is typically done using ROBDDs. As it is already mentioned earlier, ROBDDs can exhibit highly irregular memory requirements, with unexpected blowup. As a result, the CODC computation, though very elegant in its conception, is typically not feasible for large circuits. In general, the computation is not robust for medium-sized circuits either. The work presented in this thesis is applicable for large circuits also therefore, an alternative way of computing

CODCs has to be employed.

In [27], the authors have presented a way of computing approximate compatible don't cares (ACODCs). These don't cares are approximation of CODCs. The ACODCs can be computed for larger designs and the time required and memory utilization for the ACODC computation is much less (both typically 30X lower) than the corresponding values for full CODC computation. At the same time, the literal count reduction obtained by ACODCs is typically about 80% of that obtained by full CODCs. To compute ACODCs for any node n, a smaller sub-network is extracted from the original network. The smaller network is a network rooted at the node of interest, up to a certain topological depth in the original network. Then the CODCs are computed for the smaller network. This can be done quite efficiently. The authors show that the don't cares computed in this manner yields a literal count reduction which is about 80% of that obtained by complete CODCs. Hence, ACODCs will be used in this thesis work.

D. Conclusion

This chapter briefly dicuss the background information which will be required to understand this thesis. The next chapter will descibe the algorithm for create of partitioned ROBDDs.

CHAPTER III

PARTITIONED ROBDDS

A. Introduction

Though PTL circuits possess many advantages over other circuit styles there are a couple of issues that need to be addressed. Firstly, due to *body effect* in a practical PTL approach it is not advisable to connect more than 4-5 pass transistors in series. Secondly, since the PTL synthesis method uses ROBDDs, memory explosion can occur if ROBDDs are built monolithically. The first problem can be solved by inserting buffers after every 4-5 ROBDD nodes (pass-transistors). However, if monolithic ROBDD are used as the starting point for PTL synthesis (monolithic ROBDDs can have a size which is exponential in number of the inputs) then the advantage of PTL over other circuit styles will be lost in terms of area, power and delay. Therefore to avoid memory explosion, partitioned ROBDDs should be employed to construct the ROBDDs of the design. The use of partitioned ROBDDs also tackle the first problem, by buffering the output of every PTL structure.

Partitioned ROBDDs avoid the memory explosion which is possible when using monolithic ROBDD. The intuition behind this savings in ROBDD size is as follows: in general, when constructing the ROBDD of function $F = G_1 < op > G_2$, the size of F is $|F|$, $F = O(|G_1||G_2|)$ [4], where $|G_1|$ and $|G_2|$ are the sizes of the ROBDDs of G_1 and G_2 respectively. If the partitioned ROBDDs are built for F, then the aggregate size of ROBDDs will be $O(|G_1|+|G_2|)$ [3]. Thus partitioned ROBDDs certainly reduce the aggregate size of the ROBDDs of the circuit. For this reason, partitioned ROBDDs allow us to construct the ROBDDs of large circuits when monolithic ROBDDs fail. The price to pay is that unlike monolithic ROBDD, partitioned ROBDDs are not canonical for a given variable ordering. However, this does not pose a problem in this thesis work, since this thesis deals with gen-

erating PTL circuits and not in manipulating ROBDD as a data structure. The algorithm for constructing partitioned ROBDDs is briefly discussed in the next section.

B. Algorithm

Consider a Boolean network η. First the network η is optimized and decomposed using 2-input gates and inverters only. Let the modified network be represented by η^*. Now η^* is sorted in a depth-first manner. The resulting array of nodes is sorted in *levelization*[1] order, and placed into an array *L*. Thus the nodes of η^* are stored in *A* in topological order from the inputs to the outputs.

Now a node *n* is fetched from array *A* in index order. The ROBDD *f* of *n* is built using the *ntbdd_node_to_bdd* function. Let *m* be the level of node *n*. Then the depth of the resulting node ROBDD *f* is compared with the maximum allowable depth of 5 (which maximum number of ROBDD nodes between the root and leaves). If the depth of *f* is less than 5 then we continue to create the ROBDD of other nodes in array *A*. However, if the depth of *f* is equal to 5 then the node *n* is made a new variable. If the depth is greater than 5 then one of the fanins of *n* is made a new variable. The fanin which is made a new variable is one whose topological level is one less than that of *n*. Recall that any node can have at most two fanins. If both fanins have the same topological level, then the fanin node with maximum ROBDD depth is made as a new variable.

In this way, we can guarantee that no PTL structure will have a depth more than the maximum allowable depth of 5. Algorithm 1 summarizes the methodology of partitioned ROBDD construction. Figure 8 shows the resulting partitioned PTL implementation of a circuit with one primary output and 10 primary inputs. The triangles in the figure represent the partitioned ROBDDs whereas the top vertex of each triangle is a variable created during

[1] Primary inputs are assigned a level 0, and other nodes are assigned a level which is one larger than the maximum level among all their fanins

partitioned ROBDD construction. The top vertex of each triangle is also buffered as shown by the buffers (two back to back inverters) connected to each triangle's vertex.

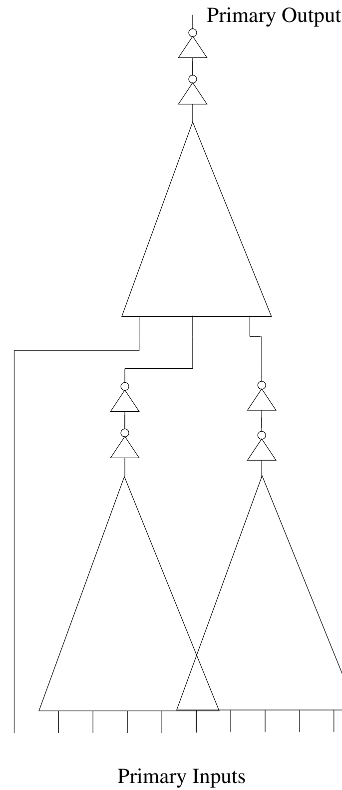

Figure 8. Partitioned ROBDD of a Circuit (traditional buffering)

C. Conclusion

Partitioned ROBDDs are very effective in the ROBDD construction of large circuits as compared with monolithic ROBDDs. They also help in PTL synthesis by allowing the resultant PTL implementation to have a bounded depth (thereby avoiding a delay increase due to body effect). The use of partitioned ROBDDs in PTL synthesis also avoids the memory explosions that is possible if monolithic ROBDDs are used. This avoids an area increase that would result if monolithic ROBDDs are used.

Algorithm 1 Partitioned ROBDD Construction

```
η = optimize_network(η)
η* = decompose_network(η, 2)
A = dfs_and_levelize_nodes(η*)
i = 1
while i ≤ size(A) do
    n = array_fetch(A,i)
    f = ntbdd_node_to_bdd(node)
    if bdd_depth(f) > 5 then
        fanin = node_fanin(n,level(n-1))
        bdd_create_variable(fanin)
    else
        if bdd_depth(f) == 5 then
            bdd_create_variable(f)
        else
            continue
        end if
    end if
end while
```

CHAPTER IV

PTL WITH GENERALIZED BUFFERING

A. Introduction

Partitioned ROBBD (described in the last chapter) enable us to implement a PTL based circuit efficiently. However, this approached uses only inverters for buffering the signals between the partitioned PTL blocks and hence does not explore the advantage which generalized buffering can offer. This thesis formulates a novel approach to do generalized buffering, by casting the problem as an instance of Boolean division. The Boolean division of a partitioned PTL block with complex library gates can significantly simplify the logic function of that PTL block. This results in the reduction of total circuit delay and area.

This chapter presents the key contribution of this thesis i.e. a new PTL synthesis methodology which uses generalized buffering between partitioned PTL blocks. The new PTL synthesis method is discussed for two cases: one with CODCs and the other without CODCs. In order to extend the applicability of the technique, the experiments were conducted using ACODCs as well. The next section introduces Boolean division. The following two sections will present a PTL synthesis methodology for generalized buffering without using CODCs, and with CODCs.

B. Boolean Division

Boolean Division is extensively used in many logic optimization procedures like factoring, resubstitution, extraction, etc. Consider a completely specified Boolean function f which we want to divide by another completely specified Boolean function g. Boolean division can be defined as:

Definition 1 *g is a* **Boolean divisor** *of f if h and r exist such that $f = gh + r$, $gh \neq \emptyset$.*

g is said to be a **Boolean factor** of f if, g is a Boolean divisor of f, and, in addition $r = \emptyset$, i.e., $f = gh$.

In this case, h is called the **quotient** and r is called the **remainder**. Note that h and r are not unique.

Theorem B.1 *If $fg \neq \emptyset$, then g is a Boolean divisor of f.*

Proof: If $fg \neq \emptyset$, we can write

$f = fg + f\overline{g}$

$f = g(f + x) + f\overline{g}$ where $x \subseteq \overline{g}$

which is of the form $f = gh + r$, where $h = f + x$ and $r = f\overline{g}$. ■

C. Generalized Buffering without CODCs

The new PTL synthesis method described in this section does not use CODCs. Lets define the *depth* of an ROBDD g as the maximum number of nodes in any traversal of g to its terminal nodes. Our method creates partitioned PTL structures with a maximum depth of 5 (this number can be arbitrary, though). This ensures that there are no more than 5 transistors in series, in any PTL block.

In order to implement generalized buffering, Boolean division will be used. Algorithm 2 describes the new PTL synthesis methodology which uses generalized buffers. Consider a boolean network η. First the network η is optimized and decomposed using 2-input gates and inverters only. Let the new network be referred to as η^*. Now η^* is sorted in depth-first manner. The resulting array of nodes is sorted in *levelization* order, and placed into an array L.

In this approach, we first build ROBDDs of the nodes of η, topologically from the inputs to the outputs by fetching the nodes from the array A in index order. Suppose we encounter a new node n for which we want to construct a ROBDD f. The ROBDD of each

of its fanins must have a depth of at most 4 ROBDD variables (assuming that the fanins are not partitioned ROBDD variables). When constructing the ROBDD f of n, it can initially have at most 8 ROBDD variables (since any node in the original network can have at most 2 fanins). If the depth of f is less than 5 then we continue with creating the ROBDD of the other nodes in array A. However, if the depth of f is greater than or equal to 5 then we attempt to divide f with the gates in a library by calling the *test_division* function. If the division is successful, then the depth of the ROBDD f will reduce . The detailed description of the *test_division* function is provided after the synthesis flow.

When we perform ROBDD construction, the maximum depth of the ROBDD of a node n before division can be 8, as discussed earlier. The division routines systematically reduce this depth to below 5, by using one or more generalized buffers. If an ROBDD cannot be divided (and the resulting depth after division is greater than 5), then we *back-track*, and make one of the fanins of n a new variable. The fanin which is made a new variable is the one whose topological level is one less than that of n. The reason for this is illustrated in Figure 9.

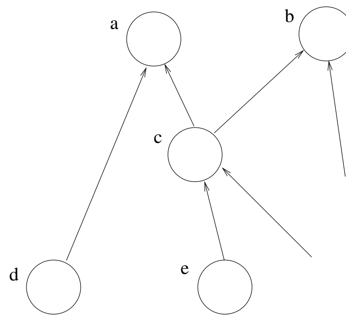

Figure 9. Back-tracking during Division

Consider node a, at topological level n. Suppose it has two fanins c and d, with levels $n-1$ and $n-2$ respectively. Since partitioned ROBDD construction occurs in topological

Algorithm 2 Pseudo Code for PTL synthesis using generalized buffers and without CODCs

```
η = optimize_network(η)
η* = decompose_network(η, p)
A = dfs_and_levelize_nodes(η*)
i = 1
while i ≤ size(A) do
    n = array_fetch(A,i)
    f = ntbdd_node_to_bdd(n)
    if bdd_depth(f) ≥ 5 then
        for g ≡ G ∈ Gate Library do
            f = test_division(f,g,G)
        end for
        if bdd_depth(f) > 5 then
            back-track
        else
            bdd_create_variable(n)
            continue
        end if
    else
        continue
    end if
end while
```

order from inputs to outputs, it is possible that node b (at level n) has already been processed and divided. Its ROBDD therefore has a depth of less than 5 nodes. Now suppose the ROBDD depths of d and c are 4 each, and the ROBDD of a has depth 8. Suppose all divisions for node a fail. In that case, we need to back-track and make either c or d a new variable. This would guarantee that a has a new depth 5. We select c (with level $n-1$) as the new variable since it is more likely that d (which is at a lower topological level) has more fanouts of level n or $n-1$. When c is made a new variable, all its fanouts are checked. If any of them has already been processed (such fanouts must have level n), then their ROBDDs are re-computed, and division is re-done. In this way, the PTL network is kept as small as possible. If this re-computation was not performed, then the logic of the node c would be implemented twice as a PTL structure (once for the node b and then again for the variable corresponding to node c itself).

If after a back-track the depth of the node n is less than 5 then we continue to grow the corresponding ROBDD further up to a depth 8.

After attempting division (regardless of whether the division succeeded or failed), the depth of n is guaranteed to be less than or equal to 5, allowing for an elegant exit in the case division fails. In general, however, this division strategy yields a number of good generalized buffers, so this occurrence is rare. The *test_division* routine is based on Boolean division. The next section describes the *test_division* algorithm, for the cases where CDOCs are not used.

1. ROBDD Division without CODCs

The key idea behind ROBDD division is that we take the ROBDD of a node n and attempt to divide it with the gates in a library. If such a Boolean division is possible *and* it is strictly ROBDD depth-reducing, we select it. The test for whether a library gate g, with an associated variable G, divides the ROBDD f of n is described next. A pictorial view of the

process is shown in Figure 10.

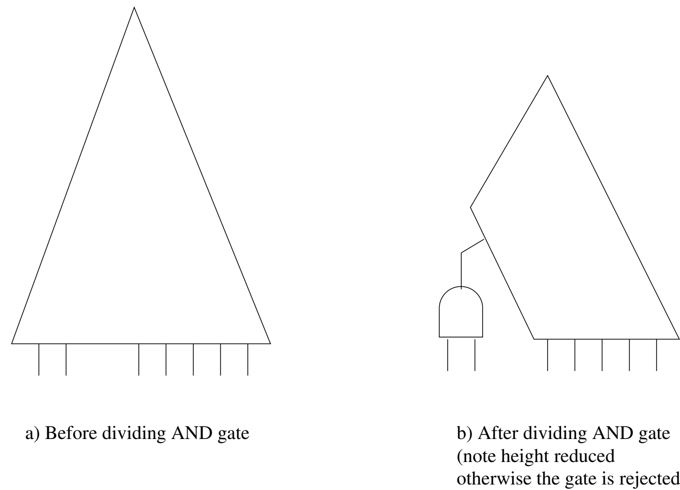

Figure 10. Dividing a Generalized Buffer into a PTL Structure

The Boolean division based test for whether a library gate g divides a ROBDD f can be represented by following logic equations. Here, f is considered to be Completely Specified Function (CSF).

$f = G(f + \overline{x}) + (f)\overline{g}$, where $g \equiv G$ and $x \subseteq \overline{g}$.

Therefore, the upper and lower bounds (U and L) for f are:

$U = G(f + \overline{g}) + (f)\overline{g}$

and

$L = (fG + f\overline{g})$

In this case, with quotient $h = f + x$ and remainder $r = f\overline{g}$, we can represent f as $f = Gh + r$

Algorithm 3 describes the $test_division$ procedure without CODCs. This algorithm represents the test performed during division of ROBDD f by a gate g (having a variable G). 2.

In Algorithm 3, the functions L and U are ANDed with $(g\overline{\oplus}G)$ to express the fact

Algorithm 3 Pseudocode for Division of f by $g \equiv G$

$test_division(f, g, G)\{$

if $fg \neq 0$ **then**

$L = (fG + f\overline{g})(g\overline{\oplus}G)$

$U = (fG + G\overline{g} + f\overline{g})(g\overline{\oplus}G)$

$Z = bdd_between(L, U)$

$Z^{*} = bdd_smooth(Z, gvars)$

$R = bdd_compose(Z^{*}, G, g)$

if $R = f$ and $bdd_depth(Z^{*}) < bdd_depth(f)$ **then**

$return(success, Z^{*})$

end if

else

return fail

end if

$\}$

that g and G are not independent variables, but rather are related as $G \equiv g$. We next find a small ROBDD Z (using the function *bdd_between*, which returns the heuristically smallest ROBDD Z such that $L \subseteq Z \subseteq L+U$). Next, we smooth out[1] the variables of g. If the resulting ROBDD Z^*, with g composed back into G, is identical to f, we return Z^* as the quotient.

Figure 11 shows the resulting synthesized PTL circuit obtained after performing PTL synthesis with generalized buffering as described in Algorithm 2.

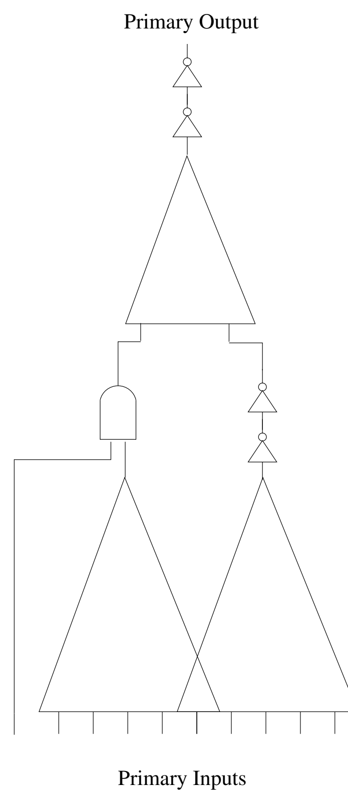

Figure 11. Partitioned ROBDD with generalized buffering

[1]Smoothing is also referred to as Existential Quantification. The existential quantification of f with respect to a variable x is expressed as $\exists_x f = f_x + f_{\overline{x}}$. Smoothing a set of variables is achieved by performing existential quantification, one variable at a time

D. Generalized Buffering with CODCs

The PTL synthesis methodology with generalized buffering described in the last section attempts to divide the ROBDD of a node with the gates in the library. During the division, the method finds an upper U and a lower bound L on the resulting ROBDD and then select a heuristically minimum ROBDD between L and U by using the *bdd_between* procedure. The range of this choice can extended by using CODCs, allowing the final ROBDD size to be be further reduced. In other words, the CODCs can be used to further simplify the logic of a PTL structure. The CODCs can be computed in two manners: one using $full_simplify$ to compute the complete CODCs, and another using approximate CODCs (ACODCs). The ACODCs are computed in a similar manner as mentioned in [27].

When we want to use CODCs (or alternatively ACODCs) during the division process, we first build the ROBDDs of the nodes of η topologically from the inputs to the outputs by fetching the nodes from the array A in increasing order of index. When we encounter a new node n whose ROBDD f has a depth greater than 5, then we compute its CODCs by using the $compute_dc$ function. After CODC computation we try to divide f with the gates in the library by calling the $test_division$ function. The $test_division$ makes use of the CODCs of that node to simplify the logic, and is described in Algorithm 5. If the division is successful and if the depth of ROBDD reduces below 5 then we make n a new ROBDD variable. Otherwise, we backtrack (in the same manner as described in the case when CODCs were not used) and make one of the fanins of n a new variable. The fanin which is made a new variable is the one whose topological level is one less than that of n. Algorithm 4 summaries the synthesis flow for PTL synthesis with generalized buffering and using CODCs.

Algorithm 4 Pseudo Code for PTL synthesis using generalized buffers and with CODCs

```
η = optimize_network(η)
η* = decompose_network(η, p)
A = dfs_and_levelize_nodes(η*)
i = 1
while i ≤ size(A) do
    n = array_fetch(A,i)
    f = ntbdd_node_to_bdd(n)
    if bdd_depth(f) ≥ 5 then
        d = compute_dc(η*,n)
        for g ≡ G ∈ Gate Library do
            f = test_division(f,d,g,G)
        end for
        if bdd_depth(f) > 5 then
            back-track
        else
            bdd_create_variable(n)
            continue
        end if
    else
        continue
    end if
end while
```

1. ROBDD Division with CODCs

In principle, we still attempt to divide the ROBDD of n with the gates in a library. However, CODCs are utilized during division to simplify the logic further. If such a Boolean division is possible *and* it is strictly height-reducing, we select it. The test for whether a library gate g, with an associated variable G, divides the ROBDD f of n is described next.

In this case, the ROBDD f of n is considered to be Incompletely Specified Function (ISF). Let d represents the CODCs computed for the node n. Then the division of f with the logic gates $G \equiv g$ in the library can be represented by the following equations:

$f = G(f + d + \overline{x}) + (f + d)\overline{g}$, where $g \equiv G$ and $x \subseteq \overline{g}$

Therefore, the upper bound U for f is:

$U = G(f + d + \overline{g}) + (f + d)\overline{g}$

The lower bound L for f is still the same as when CODCs are not used.

$L = (fG + f\overline{g})$

The lower bound L can be found by setting $x =$ and $d =$ in the expression for f above. The test we perform when we try to divide a gate g (having a variable G) into the ROBDD f of node n with CODC d is shown in Algorithm 5.

In Algorithm 5, the functions L and U are ANDed with $(g\overline{\oplus}G)$ to express the fact that g and G are not independent variables, but rather are related as $G \equiv g$. We next find a small ROBDD Z (using the function $bdd_between()$, which returns the heuristically smallest ROBDD Z such that $L \subseteq Z \subseteq L + U$), which is a Boolean divisor of f. Next, we smooth out the variables of g. If the resulting ROBDD Z^*, with g composed back into G, lies between f and $f + d$, we return Z^* as the quotient. If we have a valid division then we map the don't care d of the node n onto a new set of variables V so that it can be used during the next iteration. The CODCs are mapped using following equations:

$Q = d(g\overline{\oplus}G)$

Algorithm 5 Pseudocode for Division of f by $g \equiv G$ using CODCs

$test_division(f,d,g,G)\{$

if $fg \neq 0$ **then**

$L = (fG + f\overline{g})(g \overline{\oplus} G)$

$U = (fG + dG + G\overline{g} + f\overline{g} + d\overline{g})(g \overline{\oplus} G)$

$Z = bdd_between(L,U)$

$Z^* = bdd_smooth(Z, gvars)$

$R = bdd_compose(Z^*, G, g)$

if $f \subseteq R \subseteq f + d$ and $bdd_depth(Z^*) < bdd_depth(f)$ **then**

$Q = d(g \overline{\oplus} G)$

$d = \exists_V Q$

$return(success, Z^*)$

end if

else

return fail

end if

$\}$

$d = \exists_V Q$

where V is the set of variables which lie in the support of g but are not present in the support of Z^*. The re-mapped don't cares of n are used for further Boolean division, to explore additional generalized buffering opportunities for the node n.

E. Conclusion

This chapter describes the PTL logic synthesis using generalized buffering for both cases i.e. with and without CODCs. The methods described in this chapter was implemented and the experimental results are provided in the next chapter.

CHAPTER V

EXPERIMENTAL RESULTS

A. Implementation Details

The new PTL synthesis algorithms with generalized buffering were implemented in SIS [24]. The code consisted of reading a circuit (which was decomposed before-hand into 2-input gates and inverters), and then building ROBDDs of its nodes in a topological manner from the inputs to the outputs. When the depth of ROBDD of any node grew beyond 5, division was invoked, as described in last chapter. For PTL synthesis with CODCs, CODCs were computed and used during division. The CODCs computation were done in two manners: one using $full_simplify$ [24] (which will provide complete CODCs) and another using, approximate CODCs (ACODCs) were computed as mentioned in [27]. Results are presented and compared for both styles of CODCs.

The resulting library gates that were utilized for division, and the new ROBDD after division were stored within each node's data structure. Since the ROBDDs in question were small (with a maximum height of 8), division was performed exhaustively. If the resulting depth after division was greater than 5, then a back-track step was invoked making one of the node's fanins a new variable. After this we check the nodes which are in the fanout of the fanin node corresponding to the newly created variable. If any of these fanout nodes was already processed, then their ROBDDs were re-computed, and division was re-done. After attempting division (regardless of whether the division succeeds or fails), the depth of the final ROBDD is guaranteed to be less than or equal to 5, allowing for an elegant exit strategy in case division fails.

The next section provides the information about the library used in the PTL synthesis implementation and the process technology used. Subsequent sections, report the delay

and area results for various circuits which were synthesized using the new PTL approach. These sections also compare these results with the corresponding results for a traditional buffer-based PTL synthesis methodology.

A set of benchmark circuits were synthesized using the new PTL synthesis algorithm with generalized buffering. Experiments show a clear advantage of new PTL synthesis algorithms over traditional PTL synthesis. The traditional method used for comparison was similar to that reported in [3].

B. Gate Library

The gate library used in this implementation consisted of the gates AND2, AND3, AND4, OR2, OR3 and OR4. Since any divided gate becomes a new variable, it is required in both its polarities. Therefore, by DeMorgan's law, we only have non-inverting gates in our library.

The MUX and all gates in the library were characterized for delay in SPICE [28], using a 100nm BPTM [29] process technology. Table B shows the delay and active area of MUX and all gates in the library.

C. Delay

The delay of a synthesized PTL circuit was extracted by finding the longest delay path from any output to any input. Table C compares the delay of the traditionally buffered partitioned ROBDD based implementation, with the generalized buffering methodology (both cases: with and without CODCs). In this table, column 1 lists the example under consideration. Column 2 reports the circuit delay, for the traditional method. Column 3 reports delay for new PTL synthesis algorithm with generalized buffering without CODCs (as a fraction of the delay of the traditional method). Column 4 and 5 report delay number for PTL synthesis

Table I. Delay and active area of gate library

Gate	Delay(ps)	Area(μ^2)
MUX	18	0.08
INV	10.26	0.08
Buffer	20.5	0.16
AND2	30.20	0.28
AND3	37.76	0.44
AND4	47.39	0.64
OR2	38.70	0.36
OR3	46.08	0.68
OR4	68.28	1.12

using ACODCs and CODCs respectively (again as a fraction of the delay of the traditional method). Some entries in the table marked as "-". This indicates that $full_simplify$ was not able to compute the CODCs for that circuit.

We observe that new PTL synthesis approach with generalized buffering results in a speed-up of about 24% on average, compared to the traditional method (when CODCs were not used). When ACODCs were utilized for generalized buffering speed-up over the traditional method improves to about 29%. We also observe that it is better to use ACODCs than CODCs because of the following reasons:

- $full_simplify$ is not able to compute CODCs for many large circuits whereas ACODCs can be computed robustly for arbitrary sized circuits [27].

- Second, using CODCs instead of ACODCs does not appreciably improve the delay of the PTL circuits. On average the additional improvement is less than 1% (calculated

Table II. Delay Comparison of Generalized and Traditional Buffered PTL

	Traditional Buffering	Generalized Buffering		
		Without CODC's	With ACODCs	With CODCs
Ckt	Delay (ps)	Delay	Delay	Delay
alu2	5125.32	0.53	0.55	0.54
alu4	18849.06	0.45	0.43	-
apex6	1519.02	0.73	0.75	0.75
C432	3505.68	0.86	0.79	0.76
C499	1033.20	1.01	1.01	1.01
C880	2264.40	0.70	0.53	0.54
C1908	4017.96	0.74	0.67	0.68
C3540	21056.04	0.71	0.53	-
C5315	4949.28	1.04	1.03	1.04
x3	1045.98	0.82	0.82	0.82
i8	1732.32	0.69	0.62	0.6
x1	860.94	0.67	0.67	0.67
pair	2197.44	0.75	0.70	0.70
rot	1870.74	0.82	0.80	0.80
C6288	36024.12	0.94	0.86	-
des	2547.54	0.89	0.79	-
too_large	3130.56	0.52	0.54	0.54
AVERAGE		0.756	0.710	-

over the examples for which the CODC method completed).

- The time taken by $full_simplify$ to compute CODCs is much larger than the time required to compute ACODCs. The run time for each PTL circuit synthesis approach is mentioned in Table E.

D. Area

The area calculation was performed by determining the active area of the MUXes and all the library cells. The area numbers for various gates are mentioned in Table B. The number of MUXes is simply the sum of the sizes of each of the partitioned ROBDDs in the design.

Table D compares the active area of the traditionally buffered partitioned ROBDD based implementation, with that of the generalized buffering methodology (with and without CODCs). In this table, Column 1 lists the example under consideration. Column 2 reports the circuit active area for the traditional method. Columns 3, 4 and 5 report the area for the new PTL synthesis algorithm with generalized buffering without CODCs, with ACODCs and with CODCs respectively. The area in Columns 3, 4 and 5 are expressed as a fraction of the area of the traditional method.

The generalized buffers occupy a greater area than the traditional buffers. Due to this the area improvement of the new PTL synthesis approach is not as high i.e. 3% on average when CODCs were not used and 5% when an ACODCs were utilized. On average (computed over the examples for which CODCs could be computed), the CODCs based method exhibits a 1% area overhead over ACODC based method. Again we observe that using CODCs does not result in area reduction over the case where ACODCs are used. This is an additional reason to use ACODCs rather than

Table III. Area Comparison of Generalized and Traditional Buffered PTL

	Traditional Buffering	Generalized Buffering		
		Without CODC's	With ACODCs	With CODCs
Ckt	Area (μ^2)	Area	Area	Area
alu2	164.32	0.87	0.86	0.83
alu4	963.68	1.12	1.12	-
apex6	305.52	0.95	0.93	0.93
C432	87.12	1.02	0.93	1.10
C499	106.24	0.92	0.92	0.92
C880	136.16	0.80	0.79	0.79
C1908	152.24	0.97	0.93	0.93
C3540	532.88	1.01	0.98	-
C5315	540.16	1.12	1.11	1.12
x3	315.76	0.96	0.96	0.96
i8	520.48	0.75	0.72	0.70
x1	120.88	0.96	0.95	0.95
pair	640.32	1.08	1.07	1.06
rot	229.20	0.96	0.96	0.96
C6288	1220.48	1.10	1.06	-
des	1808.88	0.94	0.91	-
too_large	125.68	0.98	0.97	0.97
AVERAGE		0.970	0.950	-

CODCs. In the case of the circuit C432, when CODCs were used the area increases by 17% compared to the area when ACODCs were used. This increase in area is due to the use of OR4 and OR3 gates during Boolean division (when CODCs were used). When ACODCs were used, these gates were not utilized.

E. Runtime

Table E reports the run-time for the different PTL synthesis algorithms with generalized buffering. In this table, Column 1 lists the circuit under consideration while Column 2 reports the run-time for generalized buffering without CODCs. Columns 3 and 4 report the run-time for generalized buffering using ACODCs and CODCs respectively.

We observe from the Table E that the run-time taken generalized buffering with ACODCs is 8x more than the run-time for generalized buffering without CODCs. The comparison of Columns 3 and 4 also shows that the run-time in the case of the full CODCs based computation is much larger than the run-time when ACODCs are used. Averaged over the examples for which the CODCs could be computed, the runtime for the ACODC based method was 76X better than that of the CODC based method. It has already been mentioned in the previous section that the advantage offered by using CODCs over ACODCs is minimal. The run-time comparison further underscore this conclusion.

F. Library Gates Utilized

Tables F reports the number of MUXes and inverters used for PTL synthesis using traditional and generalized buffering. Column 1 lists the example under considera-

Table IV. Run-time for PTL synthesis with generalized buffering

	Generalized Buffering		
	Without CODC's	With ACODCs	With CODCs
Ckt	Time(s)	Time(s)	Time(s)
alu2	0.380	11.48	435.32
alu4	8.990	54.52	-
apex6	1.490	4.6	117.23
C432	0.460	13.82	236.820
C499	0.360	11.67	86.78
C880	0.240	2.41	70.0
C1908	0.790	8.08	362.12
C3540	5.010	41.56	-
C5315	30.840	29.2	9543.8
x3	1.400	4.12	104.27
i8	1.630	28.73	2651.48
x1	0.240	1.08	16.2
pair	3.720	25.85	7110.52
rot	0.850	4.59	332.39
C6288	10.790	151.49	-
des	14.730	140.71	-
too_large	0.310	2.89	78.33
AVERAGE	4.837	33.55	-

tion. Columns 2 and 3 report the number of MUXes and inverters required for the traditional method. Columns 4 and 5 reports the number of MUXes and inverters used by generalized buffering without CODCs (as a fraction of the corresponding numbers for the traditional method). Columns 6 and 7 reports the number of MUXes and inverters used for generalized buffering using ACODCs (again as a fraction of the corresponding numbers for the traditional method). Finally, Columns 8 and 9 report the number of MUXes and inverters for generalized buffering using CODCs. We observe that PTL synthesis with generalized buffering and without CODCs utilizes 23% fewer MUXes and 36% fewer inverters compared to the traditional method. Whereas in case of generalized buffering using ACODCs the number of MUXes was used reduced by 27% and the number of inverters used was reduced by 40% as compared to traditional buffering.

Tables F, F and F report the number of library gates utilized by the generalized buffering approaches (without CODCs, with ACODCs and with CODCs). We observe that in each case healthy number of divisions was performed.

The general trend in these tables shows that more generalized buffers were found by the CODC method over the ACODC method. Both flavors of CODCs found more generalized buffers compared to the case when don't cares were not used.

G. Conclusion

We observe that the new PTL synthesis approach with generalized buffering results in a speed-up of about 24% on average, compared to the traditional method when CODCs were not utilized. With the use of ACODCs, the generalized buffering results in a speed-up of up to 29%. Also, the new PTL synthesis without CODCs utilizes about 23% fewer MUXes and 36% less inverters. In case when ACODCs were used,

Table V. Number of MUXes and INV Utilized during Traditional and Generalized Buffering

	Traditional		Generalized Buffering					
	Buffering		Without CODC's		ACODCs		CODCs	
Ckt	MUX	INV	MUX	INV	MUX	INV	MUX	INV
alu2	718	2054	0.600	0.429	0.577	0.403	0.532	0.36
alu4	4188	12046	0.704	0.498	0.689	0.480	-	-
apex6	1337	3819	0.737	0.604	0.719	0.586	0.718	0.586
C432	404	1089	0.995	0.928	0.795	0.691	0.834	0.704
C499	404	1328	0.762	0.699	0.762	0.699	0.762	0.698
C880	594	1702	0.731	0.634	0.702	0.605	0.712	0.613
C1908	660	1903	0.858	0.782	0.806	0.723	0.792	0.704
C3540	2362	6661	0.732	0.573	0.672	0.501	-	-
C5315	2366	6752	1.068	1.015	1.037	0.976	1.03	0.972
x3	1393	3947	0.797	0.680	0.789	0.672	0.791	0.673
i8	2418	6506	0.483	0.331	0.449	0.298	0.439	0.275
x1	527	1511	0.666	0.514	0.657	0.506	0.652	0.499
pair	2925	8004	0.790	0.661	0.726	0.592	0.718	0.582
rot	999	2865	0.803	0.688	0.796	0.675	0.79	0.671
C6288	5393	15256	0.917	0.741	0.834	0.646	-	-
des	7854	22611	0.796	0.652	0.740	0.596	-	-
too_large	552	1571	0.654	0.488	0.643	0.474	0.644	0.476
AVERAGE			0.770	0.642	0.729	0.595	-	-

Table VI. Number of Library Gates Utilized during Generalized Buffering without CODCs

Ckt	AND2	AND3	AND4	OR2	OR3	OR4
alu2	64	20	6	62	22	4
alu4	353	162	82	374	145	131
apex6	116	6	4	111	31	5
C432	17	0	0	10	0	0
C499	0	16	16	16	0	0
C880	47	9	9	0	0	0
C1908	18	13	1	44	2	0
C3540	157	68	98	155	45	8
C5315	52	13	6	62	12	0
x3	64	12	2	120	26	3
i8	298	117	0	166	29	3
x1	49	8	2	27	15	14
pair	208	28	14	193	72	61
rot	76	19	3	49	16	2
C6288	744	30	12	424	48	23
des	499	181	132	435	71	4
too_large	54	3	0	48	18	14

Table VII. Number of Library Gates Utilized during Generalized Buffering with ACODCs

Ckt	AND2	AND3	AND4	OR2	OR3	OR4
alu2	65	22	6	61	23	5
alu4	368	169	84	374	152	133
apex6	133	6	4	93	32	6
C432	30	13	0	19	0	0
C499	0	16	16	16	0	0
C880	49	10	9	2	0	0
C1908	23	18	1	44	2	0
C3540	192	80	97	152	48	17
C5315	64	19	7	70	13	5
x3	66	11	2	119	27	4
i8	342	115	4	120	28	6
x1	51	9	2	26	14	14
pair	229	40	18	197	74	79
rot	79	19	3	49	16	3
C6288	857	50	12	409	69	37
des	561	196	132	429	83	29
too_large	55	3	1	50	18	13

Table VIII. Number of Library Gates Utilized during Generalized Buffering with CODCs

Ckt	AND2	AND3	AND4	OR2	OR3	OR4
alu2	68	27	6	55	22	7
apex6	133	6	4	93	32	6
C432	28	15	13	19	2	3
C499	0	16	16	16	0	0
C880	48	10	9	2	0	0
C1908	24	18	1	44	2	1
C5315	66	19	8	73	15	8
x3	65	11	2	119	27	4
i8	344	116	5	137	28	6
x1	51	9	2	27	14	14
pair	235	42	19	191	70	82
rot	79	19	4	48	16	3
C1355	23	0	0	36	4	5
too_large	55	3	1	49	18	13

the number of MUXes utilized were reduced by 27% whereas the number of inverters were reduced by 40%.

Since the generalized buffers occupy greater area than the traditional buffers, the overall area improvement of the new PTL synthesis approaches is not as high (3% on average when CODCs were not used and 5% when ACODCs were used). The usage of complete CODCs using $full_simplify$ does yield a small improvement in delay in comparison with ACODCs. However, the run-time for complete CODCs was on average 76X that the run-time for ACODCs. Also complete CODCs cannot be computed for larger circuits. However, ACODCs can be computed for arbitrary sized circuits. Therefore, the use of complete CODCs does not provide any practical benefits over ACODCs.

The results in this thesis indicate that generalized buffering yields significantly faster designs, with a modest area benefit as well, compared to traditionally buffered PTL approaches. The run-time of the partitioning algorithm is slightly less than 15 seconds for the largest example in our benchmark suite when CODCs were not used. In case of ACODCs, the run time was less than 152 seconds. Also, a healthy number of library gates are utilized for each example during division, validating the benefits of the generalized buffering concept.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

A. Conclusions

Pass transistor logic (PTL) is a viable alternative to Static CMOS because of the advantages it possesses in terms of area and power as compared to static CMOS as well as other circuit styles. Although PTL offers great benefits for specialized circuits such as barrel shifters, there has been no widely accepted PTL design methodology which is applicable for random logic circuits. Many attempts have been made in the past to use PTL for general VLSI logic circuits, but the proposed methodologies suffer from scalability problems. Some PTL approaches used simple buffers to buffer the outputs of PTL stages. However a scaleable method to perform generalized buffering of the outputs of PTL stages has not been available to date. Therefore, there lies a significant scope for work in PTL based logic synthesis using general CMOS gates to buffer the outputs of PTL stages. This thesis explores these opportunities.

This thesis presents a new PTL circuit synthesis scheme. In order to handle larger designs, and also to ensure that the total number of series devices in the resulting circuit is bounded, partitioned Reduced Ordered Binary Decision Diagrams (ROBDDs) have been used to generate the PTL circuit.

The approach utilizes partitioned ROBDDs to construct the PTL circuit, with the interfaces between these PTL structures being buffered using library gates. The problem of *generalized* buffering is cast as an instance of Boolean division of the PTL block, using different gates in a library as divisions. In this way, we select the gate that results in the largest reduction in the height of the PTL block. In this manner,

these gates serve the function of buffering the outputs of PTL blocks, and also perform circuit computations at the same time.

PTL synthesis with generalized buffering was implemented in two different ways. In the first approach, compatible observability don't cares (CODCs) were not used. In the second approach, CODCs were utilized to further simplify the ROBDDs and to further reduce the logic in PTL structure. CODCs were computed in two different ways: one using $full_simplify$ to compute the complete CODCs and another using approximate CODCs.

Over a number of examples, on average, generalized buffering without CODCs results in a 24% reduction in delay, and a 3% improvement in circuit area, compared to a traditionally buffered PTL implementation. However, when ACODCs were used, the delay further reduced by 5% resulting in a total delay reduction of 29%. The total area reduced by 5% when ACODCs were used, compared to traditional buffering.

The use of complete CODCs provides minimal benefits over ACODCs. At the same time, the complete CODC computation took longer than the ACODCs on average. Additionally complete CODCs can only be computed for circuits with up to a few thousand gates, while ACODCs can be computed for arbitrary sized circuits. Therefore, it is better to use ACODCs over complete CODCs. Results show that, ACODCs provide enough don't cares to yield significantly better PTL structures in terms of area and delay.

B. Future Work

The effectiveness of our method can be augmented by invoking dynamic variable re-ordering while performing the ROBDD construction. This will allow a further re-

duction in circuit size for both the traditional and the generalized buffering methodologies.

The algorithm implemented in this thesis can be used for future technologies like quantum electronics, spintronics, etc. For example, in spintronics, the displacement of electrons is equivalent to the current (I_{ds}) through the MOSFET. The excitation provided for electron displacement is equivalent to drain-source voltage (V_{ds}) in MOSFET. Using these representation, if we plot the displacement versus excitation then the curve will look same as the I_{ds} vs V_{ds} curve for MOSFET. When the signal propagates through MOSFETs the signal strength reduces. Similarly, when the electron will spin around, it will collide with other electrons and this will propagate the signal. But the strength of this signal will also reduce. Therefore, we can use the technique proposed in this thesis to restore the signal strength. Similarly, the idea of thesis work can also be applied to quantum electronics also.

REFERENCES

[1] J Rabaey, *Digital Integrated Circuits: A Design Perspective*, Prentice Hall Electronics and VLSI Series. Prentice Hall, 1996.

[2] Kamran Eshraghian Neil H. E. Weste, *Principles of CMOS VLSI Design*, Pearson Education, 1994.

[3] P Buch, A Narayan, A Newton, and A Sangiovanni-Vincetelli, "Logic synthesis for large pass transistor circuits," in *Proceedings, IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, Nov 1997, pp. 663–670.

[4] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Aug. 1986, vol. C-35, pp. 677–691.

[5] K Yano, Y Sasaki, K Rikino, and K Seki, "Top-down pass-transistor logic design," *IEEE Journal of Solid-State Circuits*, June 1996, vol. 31, pp. 792–803.

[6] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," in *Proc. of the Design Automation Conf.*, San Jose, CA, June 1990, pp. 40–45.

[7] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, et al., "VIS: A system for verification and synthesis," in *International Computer-Aided Verification Conference*, Rajeev Alur and Thomas A. Henzinger, Eds., New Brunswick, NJ, Springer-Verlag, July–August 1996, vol. 1102, pp. 428–432.

[8] K Taki, "A survey for pass-transistor logic technologies-recent researches and developments and future prospects," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, Yokohama, Japan, Feb 1998, pp. 223–226.

[9] L Macchiarulo, L Benini, and E Macii, "On-the-fly layout generation for ptl macrocells," in *Proceedings, Design, Automation and Test in Europe Conference*, Munich, Germany, March 2001, pp. 546–551.

[10] D Radhakrishnan, S Whitaker, and G Maki, "Formal design procedures for pass transistor switching circuits," *IEEE Journal of Solid-State Circuits*, April 1985, vol. 20, pp. 531–536.

[11] E McCluskey, *Logic design principles : with emphasis on testable semicustom circuits*, Prentice-Hall, 1986.

[12] W. Quine, "The problem of simplifying truth functions," *American Math. Monthly*, 1952, vol. 59, pp. 521–531.

[13] E. McCluskey, "Minimization of Boolean functions," in *The Bell System Technical Journal*, Nov 1956, vol. 35, pp. 1417–1444.

[14] T Cheung, K Asada, and H Wong, "Design automation algorithms for regenerative pass-transistor logic," in *Proceedings of 1997 IEEE International Symposium on Circuits and Systems (ISCAS)*, Hong Kong, June 1997, vol. 3, pp. 1540–1543.

[15] S-F Hsiao, J-S Yeh, and D-Y Chen, "High-performance multiplexer-based logic synthesis using pass-transistor logic," in *Proceedings of 2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, Geneva, May 2000, vol. 2, pp. 325–328.

[16] J Neves and A Albicki, "A pass transistor regular structure for implementing multi-level combinational circuits," in *Proceedings, Seventh Annual IEEE International ASIC Conference and Exhibit*, Austin, TX, Sept 1994, pp. 88–91.

[17] C Pedron and A Stauffer, "Analysis and synthesis of combinational pass transistor circuits," *IEEE Transactions on Computer-Aided Design of Integrated

Circuits and Systems, July 1988, vol. 7, pp. 775–786.

[18] D Markovic, B Nikolic, and V Oklobdzija, "General method in synthesis of pass-transistor circuits," in *Proceedings. 22nd International Conference on Microelectronics*, Serbia, May 2000, vol. 2, pp. 695–698.

[19] S Shelar and S Sapatnekar, "An efficient algorithm for low power pass transistor logic synthesis," in *Proceedings, Asia and South Pacific Design Automation Conference and the 15th International Conference on VLSI Design*, Yokahama, Japan, Jan 2002, pp. 87–92.

[20] Y-T Lai, Y-C Jiang, and H-M Chu, "BDD decomposition for mixed CMOS/PTL logic circuit synthesis," in *Proceedings, IEEE International Symposium on Circuits and Systems*, Kobe, Japan, May 2005, pp. 5649–5652.

[21] S Yamashita, K Yano, Y Sasaki, Y Akita, H Chikata, K Rikino, and K Seki, "Pass-transistor/CMOS collaborated logic: The best of both worlds," in *Digest of Technical Papers, Symposium on VLSI Circuits*, June 1997, pp. 31–32.

[22] H. Savoj and R. K. Brayton, "The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks," in *Proc. of the Design Automation Conf.*, Orlando, FL, June 1990, pp. 297–301.

[23] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[24] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA, May 1992.

[25] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," in *Proc. of the Intl. Conf. on Computer-Aided Design*, Santa Clara, CA, Nov. 1988, pp. 6–9.

[26] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's," in *Proc. of the Intl. Conf. on Computer-Aided Design*, Santa Clara, CA, Nov. 1990, pp. 130–133.

[27] N Saluja and S Khatri, "A robust algorithm for approximate compatible observability don't care (codc) computation," in *Proceedings of the Design Automation Conference*, San Jose, CA, June 2004.

[28] L Nagel, "Spice: A computer program to simulate computer circuits," in *University of California, Berkeley UCB/ERL Memo M520*, May 1995.

[29] Y Cao, T Sato, D Sylvester, M Orshansky, and C Hu, "New paradigm of predictive MOSFET and interconnect modeling for early circuit design," in *Proc. of IEEE Custom Integrated Circuit Conference*, Jun 2000, pp. 201–204, Available at http://www-device.eecs.berkeley.edu/ ptm.

VITA

Rajesh Garg received his Bachelor of Technology Degree in Electrical Engineering (Power) from the Indian Institute of Technology-Delhi, New Delhi, India in August 2004. His undergraduate research focussed on switch mode power supplies and DSP based control of drives. He has worked for Verizon Data Services India, Chennai, India and Tejas Networks, Bangalore, India. In January 2005, he joined Texas A&M University to pursue his Master's degree in Computer Engineering. His research at Texas A&M is focused on PTL based logic synthesis and radiation-tolerant circuit design. He received his M.S. in Computer Engineering in May 2006.

Permanent Address:
Rajesh Garg
332-A WERC,
Texas A&M University,
College Station, TX-77843
E-mail: rajesh.garg@yahoo.com

The typist for this thesis was RajeshGarg.