EFFICIENT SCHEMES TO SIZE TRANSISTORS FOR OPTIMAL DELAY BY SOLVING FANOUT BRANCHES WITH BALANCING ALGORITHM

By

Mehedi Sarwar

Bachelor of Science in Electrical and Electronics Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka, Bangladesh
2005

Master of Science in Electrical Engineering
University of Texas at Dallas
Richardson, TX 75080 USA
2008

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
MAY, 2015

EFFICIENT SCHEMES TO SIZE TRANSISTORS FOR OPTIMAL DELAY BY SOLVING FANOUT BRANCHES WITH BALANCING ALGORITHM

Dissertation Approved:

Dr. James E. Stine Jr.

Dissertation Adviser

Dr. Louis G. Johnson

Dr. Chriswell Hutchens

Dr. Blayne E. Mayfield

ACKNOWLEDGMENTS

I would like express my sincere appreciation and gratitude to my PhD adviser Dr. James E. Stine for his motivation, continuous support for my PhD and research works. Specially, his immense knowledge on VLSI had helped me a lot in generating new ideas and implementing them. He also should get special thanks for providing very good lab facilities , state of the art licensed software and high-performance server machines which helped not only me but our entire research group.

I will also want to express my humble gratitude to my PhD review committee members – Dr. Louis G. Johnson, Dr. Chriswell Hutchens and Dr. Blayne E. Mayfield for their encouragement , insightful comments and questions at my PhD qualifiers that helped me to do research in many new angles.

I want to thank my close friend and University of Texas at Dallas alumni Altaf Rahman for his generosity , knowledge sharing and inspirations.

My sincere thanks goes to my fellow lab-mates at the VLSI Computer Architecture Group (VCAG) in Oklahoma State University : Dr. Masoud Sadeghian, Dr. Son Bui, Samira Ataei, Manju Kiran Subbarayappa, Robert Elliot, Rabin Thapa, Tuan Nguyen for their since support , encouragement , ideas and discussion.

Lastly, I want to thank my family for their patience and encouragement.

Acknowledgements reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: Mehedi Sarwar

Date of Degree: MAY, 2015

Title of Study: **EFFICIENT SCHEMES TO SIZE TRANSISTORS FOR OPTIMAL DELAY BY SOLVING FANOUT BRANCHES WITH BALANCING ALGORITHM**

Major Field: Electrical Engineering

High performance digital system requires minimal logic and properly sized transistor to operate in all PVT corners. Specifically, high-speed data-path design is mostly about optimizing the system for better timing. In this work, the author proposed a better timing model to analyze parallel data-paths better for performance comparison. Moreover, a novel transistor sizing technique is also proposed as part of the work to minimize delay in parallel data-path circuits in the presence of practical wire capacitance. With this technique it is easier to calculate the optimal capacitance distribution in a fanout branch path that equalizes the delays in all branches as well as minimizes the overall delay starting from the primary inputs to the primary outputs of a circuit. The problem is widely termed as the "Load distribution problem at branch". A collection of fast algorithms were designed to accurately solve the load distribution problem for branch in digital circuits for optimal delay. The author used prior work on Unified Logical Effort[1] as a tool for delay estimation and transistor sizing. This research work also shows the impact of branching on critical path. Experiments are run on industry standard circuits using different types of tools developed to model the circuit. The new developed theories are tested on the circuit models , that are also included in this work.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

Introduction

In modern days, electronics and computer-based products are shaping the way of human life in this planet. In the last 30 years, digital circuit technology has advanced by giant leaps compared to other technologies largely because of scaling of transistors as well as the emergence of different computer aided design tools and algorithms.

Digital circuits are built with logic elements. Logic elements are built with tiny MOSFET (Metal Oxide Semiconductor Field Effect Transistor). Transistors were first invented in 1947 and the MOSFET was invented in 1952. The MOSFET was the first electronic device consisting of three node where the voltage between two nodes can be controlled by the voltage at the third node. The MOSFETs works as a switching device and it is an ideal device to implement switching binary logic.

1.1 Circuit and system design

1.1.1 Design flow

Digital circuit design consists of several stages. The primary few stages deals with optimization and conversion between several different logic level abstraction. These steps are performed in the Front End of the design flow.

In the later stages the actual physical system is implemented on chip by following physical design and verification steps which are collectively called the Back End of design flow.

Figure 1.1 shows the basic front End and back End design flow. Digital circuit design starts with the HDL code that describes the logic or netlist of connection. The

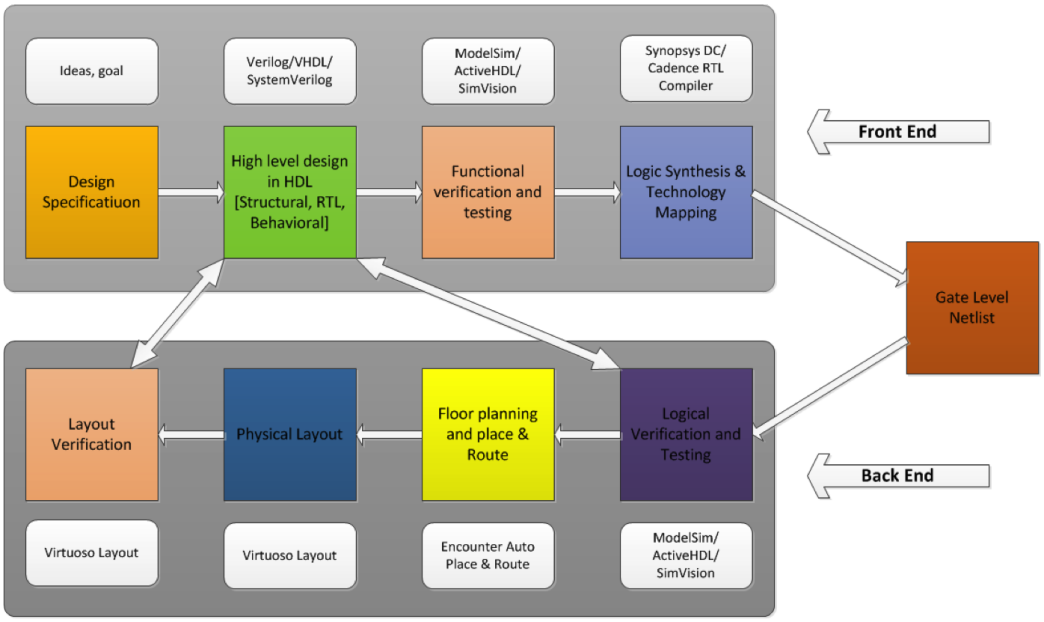


Figure 1.1: ASIC design flow

primary HDL code is then optimized with a logic optimizer. In the next stage, the optimized logic is mapped to the standard cells. The output is called the technology-mapped netlist or netlist in short. Netlist is the last abstraction level before building the actual System on Chip(SOC).

In the back end flow , designers do necessary timing analysis and re-adjust timing performance to ensure desired performance. In the next steps, the netlist is placed and routed and large blocks, analog circuit, memory, power rail and IOs are placed with floor-planning. Eventually, full-chip semiconductor layout is produced and sent to fab to manufacture.

1.1.2 Electronic design automation (EDA)

EDA (Electronic Design Automation) tools are a type of software that is used in electronic design. These design tasks are mostly computationally expensive and hence efficient algorithm and data structures are necessary to efficiently model circuit in many different kind of abstraction level. In the old days before CAD tools, integrated

circuits were designed by hand, and manually laid out. In some advance situations layout was generated using geometric software.

CAD algorithms and tools began to emerge during mid 1970s. During this era the Design Automation Conference started its journey and the ground-breaking text on VLSI system was published in 1980 called as "Introduction to VLSI Systems" by Carver Mead and Lynn Conway [3]. This text advocated for design automation using programming language and also showed that chip design pattern consists of a lot of repetitive modular design routines especially in the layout generation part. The logic optimization algorithm evolved thoroughly over the next decades and open more doors for automation.

Historians marks 1981 as the year when commercial EDA tools started their journey. Semiconductor companies like IBM, Hewlett Packard , Intel were the pioneers and later companies like Cadence, Synopsys , Mentor Graphics dedicated themselves only to EDA development.

1.1.3 Motivation for EDA tool design in academia

- VLSI CAD or EDA tools are around since 1970s.
- Many CAD ideas were generated in academia. Modern days well-known EDA developer like Synopsys and Cadence also started their journey in academia.
- To solve many design problems without actually building the physical device.
- With the change in transistor sizes many new design challenges emerge. Need proper modeling of new process technologies and integrate with existing tools.
- To facilitate developing new algorithm techniques to solve new problems.
- To automate design process of physical device used for other research.

1.1.4 Overview of path optimization using gate sizing

Modern digital circuit design usually starts with a circuit model written in a HDL (Hardware Description Language). Then, the circuit model goes through logical optimization and converted to a structural level logic gate network model that is semantically equivalent to the initial behavioral model through technology mapping process. Logical equivalency does not ensure that the circuit will function properly, because each logic element used in the network consists of practical transistors and wires which are immune to several variables such as Process, Voltage, Temperature (PVT) conditions etc. In order to maintain the correct logic levels at each logic gates output and ensure timely signal propagation, designers usually use gate or transistor sizing techniques.

In this dissertation, the author presented some novel techniques to size paths for optimal delay by taking accurate branching effort into consideration. These techniques solves branching in the context of proper load distribution and minimizes delay by balancing each path from input to output. Chapter 5 and Chapter 6 have all the details of these techniques.

1.1.5 Orientation of the thesis

This thesis consists of two major type of work. The first type is implementation of modeling tools and algorithms that model the behavior of the circuit and helps analyze the performance of the circuit under test. The second type of work are the algorithms for path optimization. The rest of dissertation is organized in the following order – Chapter 2 introduces OkCad tools collection. Then in the following chapters the theories and algorithms behind the tool are explained (in Chapter 3 , 4). In Chapter 5 the theories of Unified Logical Effort (ULE) is revisited and ULE is used as an aiding tool for path optimization. Chapter 6 , 7 give the details of path optimization techniques developed for the work followed by experimental results

(Chapter 9) . Finally , Chapter 9 concludes the dissertation.

CHAPTER 2

Ok CAD tools

2.1 Ok CAD tools

OkCAD tools is a collection of various synthesis, logic optimization and technology mapping tools designed in JAVA. The collection has rich sets of algorithm to carry out the tasks. Moreover, the tools are designed with a object oriented reusable software architecture which provides a lot of flexibility for the end user to explore the existing algorithms and contribute his/her own algorithm and integrate into the system quite easily. In the subsequent sections each tools are described briefly.

2.1.1 Ok CAD tool flow

OkCad tool collection consists of several basic tools for modeling and analyzing circuits as shown in figure 2.1. In the heart of OkCad tool is LogicGraph, a package written in Java. LogicGraph is basically a Directed Acyclic Graph (DAG) to represent the circuit schematic in program domain. The designs are imported from Verilog RTL text. An ANTLR based parser is used to parse a subset of Verilog 2001 and convert the HDL text to LogicGraphs.

The logic optimizer and equivalence checker is based on Binary Decision Diagram (BDD) . A separate independent package was developed for the BDD package. The technology mapping tool is based on structural matching of AIG or Nand-Inv of the subject graph of the circuit and pattern graph from the standard library cell. Figure 2.2 shows a sample ROBDD derived from Verilog HDL and Figure 2.3 shows the technology mapped LogicGraph of the same Verilog.

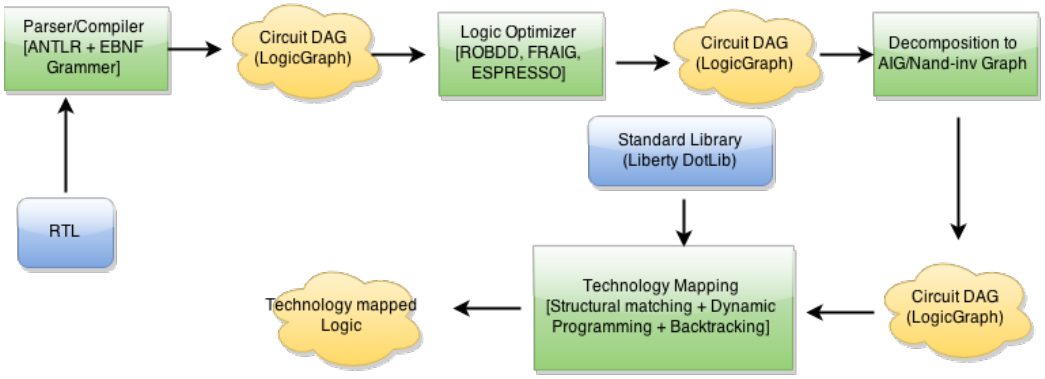


Figure 2.1: OK CAD tool flow

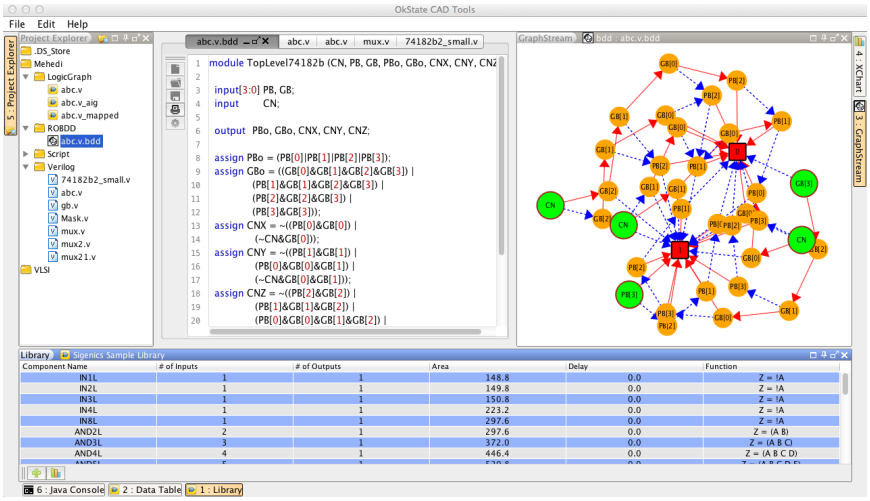


Figure 2.2: OkCAD tool with generated sample ROBDD

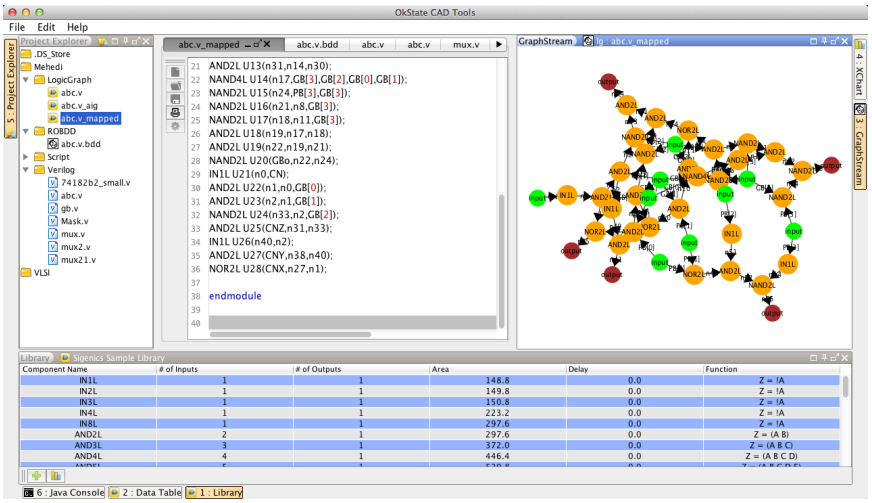


Figure 2.3: OkCAD tools with generated sample LogicGraph

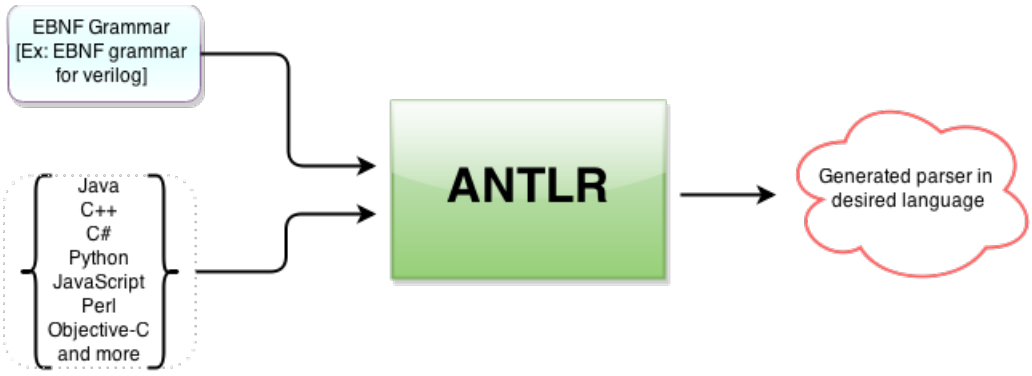


Figure 2.4: ANTLR flow

2.1.2 ANTLR parser generator

ANTLR (ANother Tool for Language Recognition) is a efficient modern parser generator written in Java [4]. The tool takes EBNF (Extended Backus-Naur Form) grammar of a language and generates parser for it in several different target language such as Java, C++, Python, JavaScript , Perl, Objective-C etc.

ANTLR can generate lexer, parser, tree parser, and combined lexer-parser. The generated parser can automatically generate abstract syntax trees which can be further processed with tree parser. Unlike other parser generator ANTLR generates the lexer, parser and tree-parser automatically and generates event-listener classes. This classes are open and can be extended by the client to implement interpreter . The role of the interpreter is to execute some tasks when certain language pattern in detected.

2.1.3 Verilog subset compiler

The compiler is an important and essential tool to build the primary data structure from text form. In past , many kind of text form has been used such as PLA, BLIF, BLIF MV etc. Although those formats are easier to understand they are not sufficient to model real life complex circuits.

The compiler included in OkCAD is a Verilog subset compiler designed using state of the art ANTLR 4 parser generator [4]. ANTLR 4 is powered by event driven and

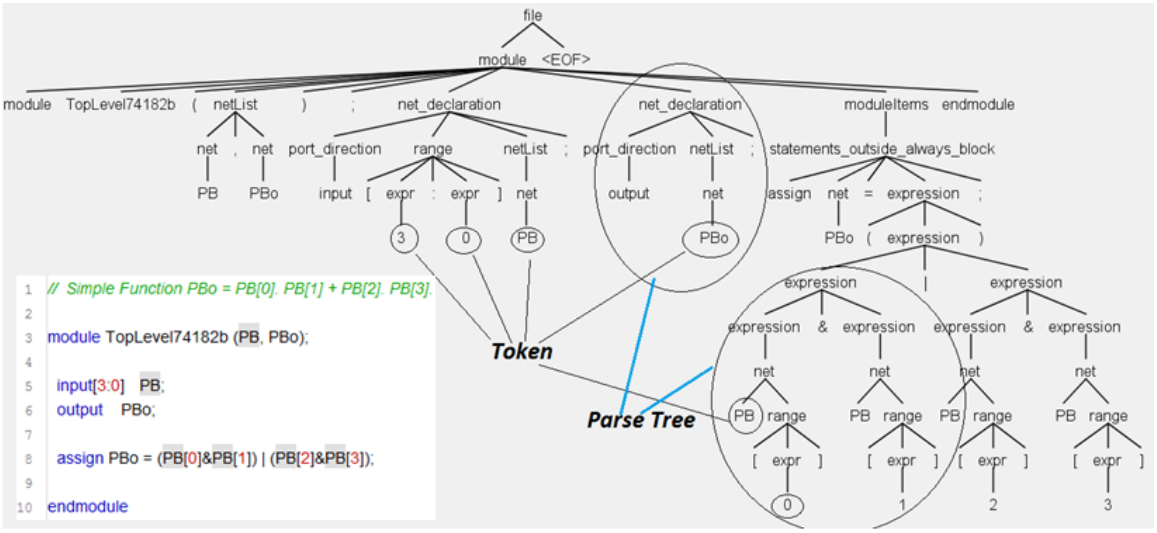


Figure 2.5: Verilog parse tree generated by ANTLR 4 based Verilog compiler

listener based architecture which makes it easier to put the target language action code separately. The compiler produces Logic Graphs (discussed later) as output.

Figure 2.5 shows the abstract syntax tree of a small sample of Verilog. Lexing and parsing is performed from left to right and the parse tree is built bottom-up. ANTLR also generates parse tree walker class and also provides listener classes.

Figure 2.5 shows a sample Verilog module and corresponding parse tree produced by the OkCAD Verilog compiler. The major features of the compiler are listed below

- Full RTL / Structural Verilog.
- Partial support for behavioral (if-else , Case etc).
- Implemented Event driven pattern.
- Understands all kinds of Verilog number notations. Example : 4b1100, 16hffff etc.
- Supports array indexing, mathematical expressions.
- Supports all RTL expression maintaining proper precedence.
- Supports module instantiation.

2.1.4 LogicGraph : digital circuit expressed as directed graph

The logic networks are represented using a graph data structure called "Logic Graph" which is basically a directed graph. The vertices represents different logic elements like logic gates, flip-flop and edges represents connecting wires. Both vertices and edges have the expected functional behavior of different logic gates and connecting wires respectively.

The most useful features of Logic Graphs are described below

A Features in LogicGraph

- Modeled all primitive logic gates for any input.
- Event-driven simulation model.
- Circuit simulation with event based signal propagation through edges
- Static Timing Analysis (STA)
- Basic placement with simulated annealing (Metropolis algorithm)

B Graph algorithms integrated in LogicGraph

- Breadth-first traversal
- Depth-first traversal
- Topological order traversal
- Shortest-path computation
- Longest-path computation
- Cycle detection

2.2 Summary

OK CAD tools is a collection of basic data structures and algorithms to aid in modeling and analyzing circuits. It contains an ANTLR based Verilog compiler to convert Verilog text to graph based data structure. There are separate packages for logic compression, technology mapping and gate placement features. Basically, it is a platform to implement and test new ideas and suitable for academic use. The collection is written entirely in object oriented Java. Hence, it is efficient and platform independent. It is extensively used to conduct most of the research work described in this dissertation.

CHAPTER 3

Binary decision diagram (BDD)

3.1 Logic optimization with reduced ordered binary decision diagram (ROBDD) techniques

OkCAD is equipped with a logic optimizer based on Reduced Ordered Binary Decision Diagram (ROBBD), a data structure that ensures a canonical representation of any logic function for a particular variable order. ROBDD maintains a variable order and hash-based data-structure named unique table to keep track of all the unique sub-functions. Hash-based implementation also ensures fast amortized time for ROBDD operation. in the subsequent the fundamentals of BDD theory and logic optimization is discussed briefly. These theories and ideas are already implemented in the OkCAD BDD package.

3.1.1 History and background

The original idea of Binary Decision Diagram came from Lee (1959) [5] and Akers (1978) [6]. Later in 1986, Randal E Bryant at Carnegie Melon University extended traditional BDD [7]. Bryant represented BDD with a defined variable order and shaped BDD to a strong canonical form. This ordered BDD is called OBDD in short. In the same paper Bryant also showed that by sharing redundant isomorphic sub-graph the logic representation can be compressed. His proposed data structure is now called a Reduced Ordered Binary Decision Diagram (ROBDD) [8], [9], a well-defined data structure with strong canonical form. It is proven that two functions are equivalent if and only if , the ROBDD's for each function are isomorphic [9].

In 1992, Brace, Rudell and Bryant [8] improved the notion of ROBDD with complemented edge BDD. A complement edge indicates that the connected formula is to be interpreted as the complement of the ordinary formula. In this way, complemented functions share the same BDD and thus reduces the size of the overall ROBDD.

The Problem with ROBDD is that some function has exponential size ROBDD for one variable order and linear size ROBDD for another variable order. Bolling and Wegener showed that finding optimal variable order is NP-Complete [10]. Variable order can be computed statically [11],[12],[13]. Static variable ordering works well for many combinational functions that come from circuits we actually build but it usually works does not do well for unstructured problems.

Friedman and Supowit found that permuting any top part of the variable order has no effect on the nodes labeled by variables in the bottom part and vice verse [14]. Their work lead to researcher to investigate dynamic variable ordering methods. The most popular method is a heuristic called sifting [15].

Many researcher used BDD for technology mapping for VLSI circuit as general. Among them, some of the techniques were pruned and optimized properly to support FPGA. In this regard the work of Mailhot and De Micheli [16],[17] is worth mentionable for their early work in this field. Later, Lehman et al.[18] used BDD for logic optimization and then converted BDD to AIG network with choice node. Their method of technology mapping takes all the optimization choices into consideration through choice nodes.

3.1.2 Basic concepts

Although the underlying model of the decision diagram already studied by Lee (1959) [5] and Akers (1978) [6], this kind of representation of switching function was not used seriously until 1986. In 1986, by introducing some ingenious ordering restrictions to BDD and providing sophisticated reduction mechanism, R. Bryant [7] substantially

improved the model. Since then ROBDD became a useful data structure to represent switching function with strong canonical form and became useful for robust model checking, validation and technology mapping. These facts have been established by the following variable properties

- ROBDD provides a strong canonical representation of switching function.
- ROBDD can be manipulated efficiently.
- For many practical switching function ROBDD representation are quite small.
- A ROBDD can represent an exponential number of paths with a linear number of nodes.
- On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, the actual operations are performed directly on that compressed representation, i.e. without decompression.

Binary decision Diagram (BDD) is directed acyclic graph, in which each vertices or node represents a Boolean function. Each vertex has one associated variable v. Depending on Boolean value (TRUE or FALSE), the associated function is further decomposed into two sub-functions which are independent of variable v and these sub-functions becomes associated with child nodes of the current node. When v is TRUE the true-child node is chosen and its associated function represents the resultant sub-function. In the same way false child node is chosen when v is false. Thus the variable v gives us a decision. Some ROBDD of basic functions are show in figure 3.1.

3.1.3 Shanon's co-factor

The Shannon expansion theorem is an important idea in Boolean algebra. It paved the way for Binary decision diagrams, Satisfiability solvers, and many other techniques

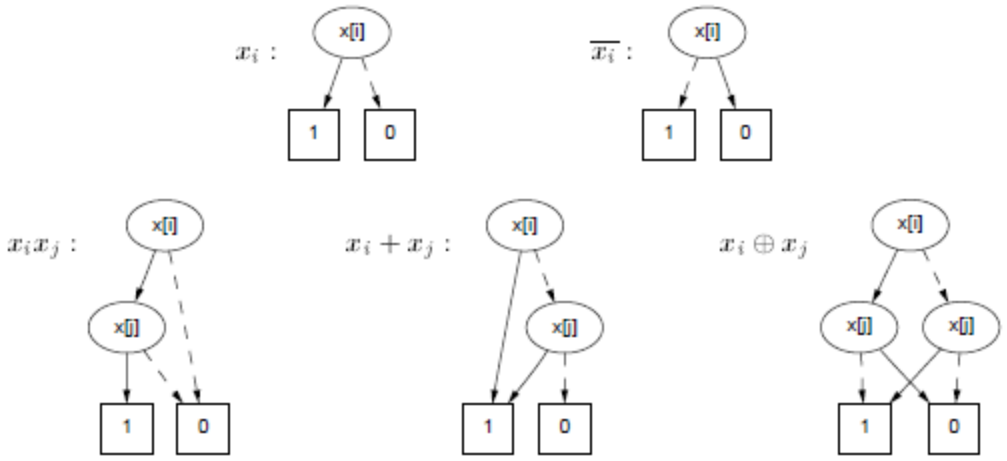


Figure 3.1: BDD of basic functions

relevant for computer engineering and formal verification of digital circuits. Shanons expansion is the fundamental mathematics of OBDD. Shanon co-factors are used in ITE algorithm to build ROBDD. When OBDD nodes are spitted to child nodes, the Boolean function associated with the node is decomposed and represented as sum of sub-functions of the original. Shanon's co-factors are also important in tautology checking in formal verification. Shanons expansion is expressed in equation 3.1

$$f = x \cdot f_{\bar{x}} + \bar{x} \cdot f_x \tag{3.1}$$

where f_x and $f_{\bar{x}}$ are Shanon's positive and negative co-factor respectively. The **positive co-factor** of f with respect to variable x is the sub-function $f_{x_i=1} = f(x_1, ..., x_{i-1}, 1, x_{i+1}, ...x_n)$. The **negative co-factor** of f with respect to variable x is the sub-function $f_{x_i=1} = f(x_1, ..., x_{i-1}, 0, x_{i+1}, ...x_n)$. Shanon's expansion is physically expressed in OBDD as in figure 3.2, where $f_1 = f(x_1, ..., x_{i-1}, 1, x_{i+1}, ...x_n)$ and $f_0 = f(x_1, ..., x_{i-1}, 0, x_{i+1}, ...x_n)$

3.1.4 ROBDD output function inversion

When a logic function is inverted in logic domain, logic 1 and logic 0 nodes swap in the ROBDD domain. In figure 3.3 , ROBDD of AND function and NAND function

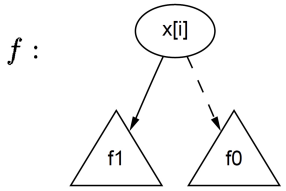


Figure 3.2: Shanon's expansion in OBDD

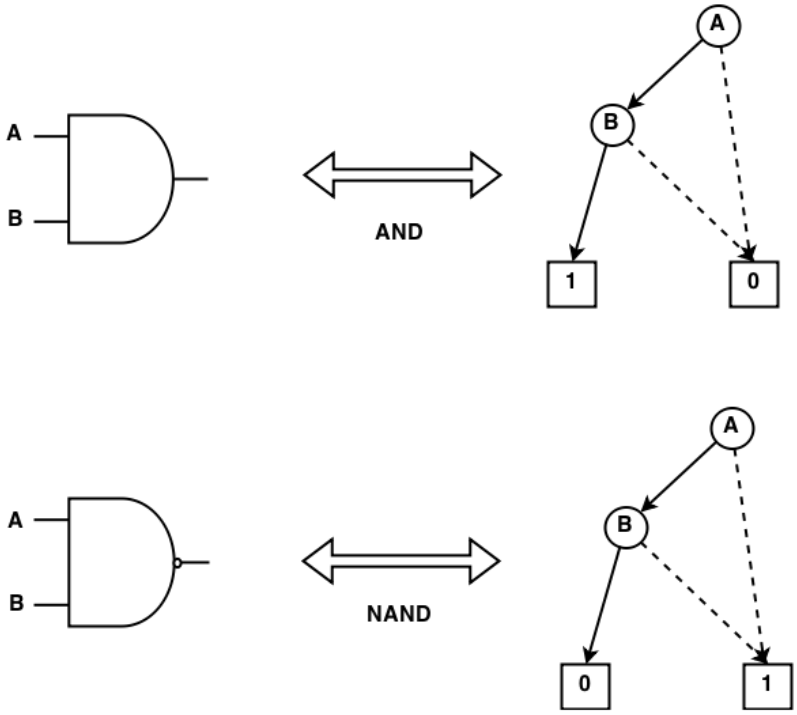


Figure 3.3: ROBDD function inversion

are shown. The only change in the ROBDDs is the logic "1" and logic "0" nodes swapped their places. In other words, all incoming edges towards the logic "1" node is now directed to logic "0" node and vice verse. With this manipulation property it is easy to invert function in ROBDD domain by swapping constant logic nodes.

3.1.5 ROBDD input variable inversion

Figure 3.4 shows ROBDD of different flavors of NAND3 functions with inverted and non-inverted inputs a, b and c. It is noticeable that when a input is inverted a transition edges coming out of that corresponding variable's BDD node is also gets

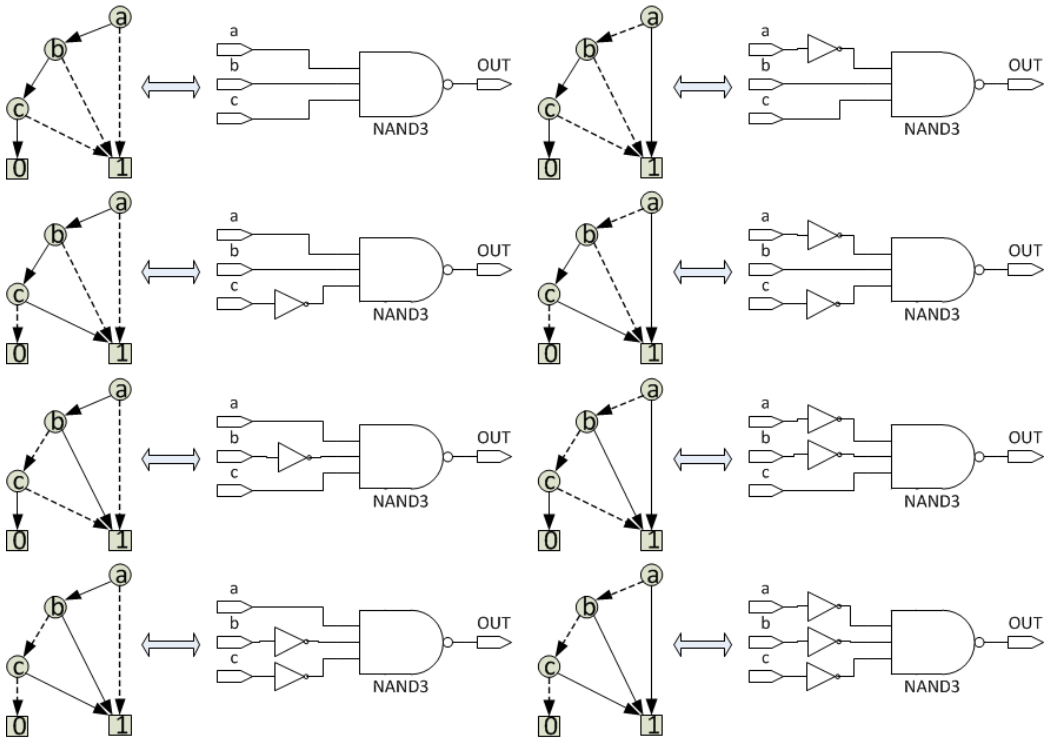


Figure 3.4: ROBDD of different flavors of NAND3function

inverted. For example, in Figure 3.4 , the top left ROBDD denotes NAND3 of a,b,c (all of them non-inverted) and in the next ROBDD (2nd row, left) only input c gets inverted. The difference between these two BDD is only in the "c" nodes. In the first ROBDD, c node's true edge goes to logic 1 and false edge goes to logic 0. In the second ROBDD c nodes true edge goes to logic 0 and false edge goes to logic 1. Thus, when inputs gets inverted in logic domain, edges gets inverted in ROBDD domain.

3.1.6 Ordering rules

A BDD with a specific variable order in all path provides strong canonical representation of logic [7] , [9]. With a specific order, it is ensured that decision on one particular variable will take place in the same level of nodes. In other words, Any particular level of nodes is functionally independent of their parent level nodes. This property allows all the nodes with same variable, same true-child and same false-child

to be shared. During tautology checking the variables of the two function need to be ordered. Theoretically, two BBD are tautological if and only if they have same variable order and encounter same node functions in any chosen path from root to leaves. The BDD ordering rules are

- No variable appears more than once along a path.
- In all paths variable appear in the same order.

Figure 3.5 shows the ordering rule.

3.1.7 Reduction rules

In 1986, Bryant [7] showed that, when BDD is ordered and reduced it offers a strong canonical representation of logic. This form of BDD is called ROBDD which can represent exponential number of paths in linear number of nodes and hence it is useful for synthesis. There are three reduction rules which are discussed below.

A Reduction rule #1 : elimination of redundant leaves

At first, Boolean function is represented by Truth Table or Binary Decision Tree. The first reduction step is to convert the Binary Decision Tree to OBDD by eliminating all the redundant leaves. OBDD should have only two leaves, a constant 1 and a constant 0. According to the first reduction rule, only one copy of the leaves are kept. All the edges that went to redundant leaves are redirected to the surviving leaves. Figure 3.6(a) shows rule #1.

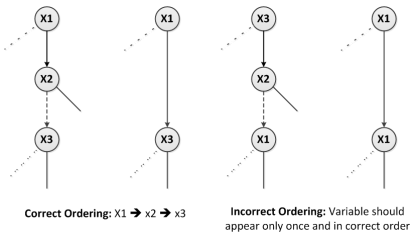
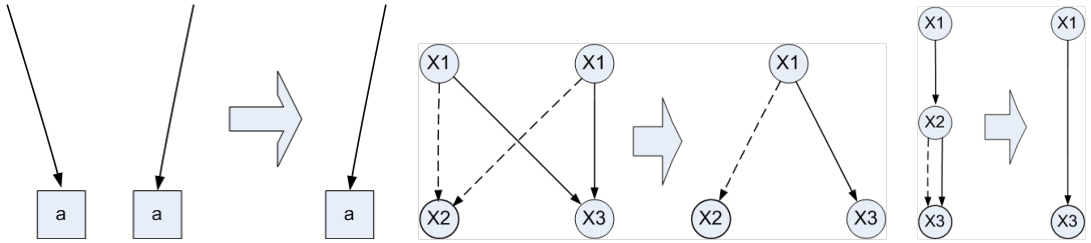


Figure 3.5: Ordering rule



(a) Reduction rule 1 : binary tree to binary decision diagram

(b) Reduction rule 2 : merging rule

(c) Reduction rule 3 :elimination rule

Figure 3.6: OBDD reduction rules

B Reduction rule #2 : merging rule

If two internal nodes are labeled by the same variable, their true-edges lead to the same node and their false-edges lead to the same node, then one of the two nodes is eliminated and all the incoming edges of the eliminated node is redirected to the remaining one. Figure 3.6(b) shows rule #2, where there were two nodes labeled "X1" pointing to the same X3 nodes through true and false edges. One of the node is eliminated.

C Reduction rule #3 : elimination rule

If the true-edge and false-edge of a node v points to the same node u then v is eliminated and all the incoming edges of v is redirected to u. Figure 3.6(c) shows rule # 3, where the true edge and false edge of the X2 node is pointing to the same X3 node. Hence, the X2 node is eliminated.

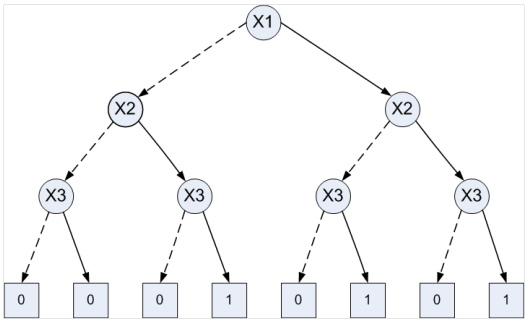
3.1.8 BDD reduction example

Table 3.1 shows the truth table of a Boolean function and figure 3.7(a) shows the Binary Decision Tree for the same function with a variable order $X_1 > X_2 > X_3$. BDD reduction rules are always applied bottom-up. Figure 3.7(b), 3.7(c) and 3.7(d)

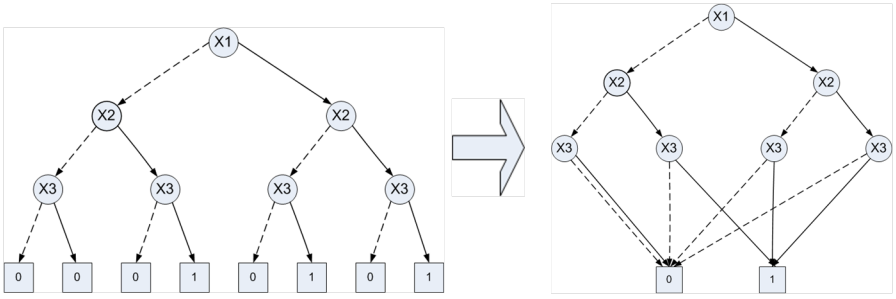
x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 3.1: Truth table representation of a Boolean function

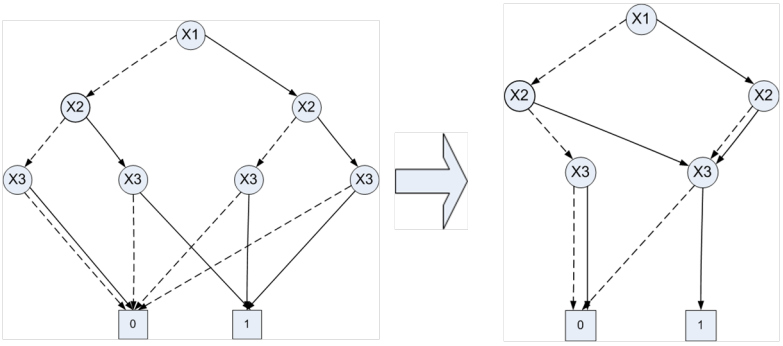
shows the three reduction steps mentioned above to form ROBDD.



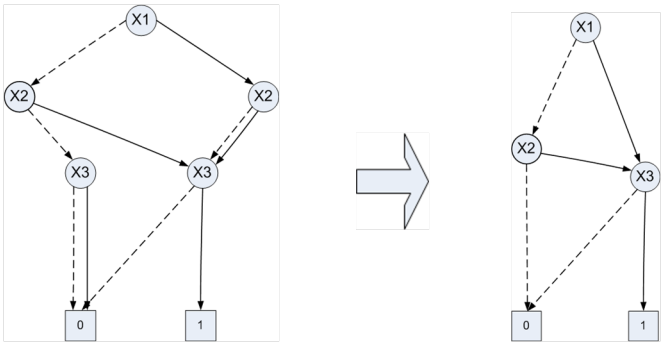
(a) Begin : binary tree of the function of table 3.1



(b) Step 1 : binary tree to binary decision diagram

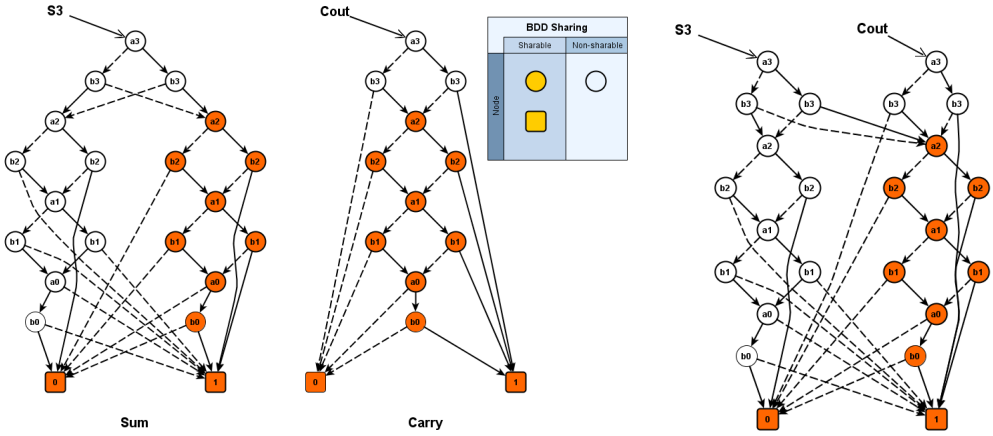


(c) Step 2 : merging rule



(d) Step 3 : elimination rule

Figure 3.7: OBDD reduction example



(a) Two seperate BDD for SUM and CARRY (b) Multi-rooted shared ROBDD

Figure 3.8: ROBDD sharing example

3.1.9 ROBDD manipulation

In traditional commercial Electronic Design Automation (EDA) tools, BDDs are built correctly on the fly. It is better to build it on the fly than to build a bad non-canonical BDD and try to fix it later. The main concept is to create vertices or nodes selectively. With the clever use of a hash table the already created unique nodes are stored and a new node is created if it is not found in the hash table. The hash table is also called a unique table.

In ROBDD if a node is found in the unique table, instead of creating new node the old one is shared. As a result, same node can be shared by multiple parent nodes under same root or multiple roots. Node sharing in ROBBD is the strongest feature of ROBDD. It helps reducing the size of the BDD and in the long run produces optimized circuits with heavily shared resources. For example , in Figure 3.8(a) and Figure 3.8(b) it is shown how BDDs are shared and multi-rooted BDDs are formed.

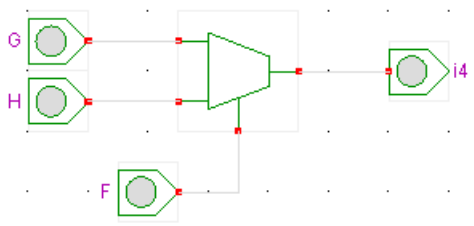


Figure 3.9: ITE analogy with 2:1 MUX in hardware

3.1.10 ITE algorithm

ITE algorithm is the core algorithm behind ROBDD manipulation. It is inconvenient and inefficient to build a BDD and then order and reduce it. This algorithm helps to incrementally build the ROBDD with correct order and reduction on the fly.

- ITE stands for IF THEN ELSE. In Boolean algebra, $ITE(F, G, H) = F \cdot G + \bar{F} \cdot H$
- In hardware ITE can be viewed as 2:1 MUX as shown in figure 3.9. In MUX, signal G is selected when F is logic 1 and signal H is selected when F is logic 0. In comparison, the node variable in ROBDD is analogous to selector signal F, true child is analogous to signal G and false child is analogous to signal H.
- ITE applies Shannon's expansion theorem on the fly when building ROBDD.
- ITE takes three BDD node as I,T and E and performs **If Then Else** operation on the nodes and ultimately returns another BDD node.
- If the required node is found in unique table. then the found node from unique table is sent. In this way, same node is shared.
- If terminal cases are met, trivial results are sent, otherwise Shannon co-factors are computed and a new node is created in the unique table and that node is sent.

- Very useful, as it can be used to implement many other common logic such as AND, OR, NOT etc. Table 3.2 shows a list of common logic function represented in term of *ite* operator.

algorithm 1 shows the ITE algorithm and **algorithm** 2 shows the "Find or Create" algorithm used in Unique table.

Subset	**Expression**	**Equivalent** *ite* **form**
0	0	0
1	1	1
f	f	f
g	g	g
$NOT(f)$	$\bar{f}$	$ite(f, 0, 1)$
$NOT(g)$	$\bar{g}$	$ite(g, 0, 1)$
$AND(f, g)$	$f \cdot g$	$ite(f, g, 0)$
$f > g$	$f \cdot \bar{g}$	$ite(f, \bar{g}, 0)$
$f < g$	$\bar{f} \cdot g$	$ite(f, 0, g)$
$NOR(f, g)$	$\bar{f} \cdot \bar{g}$	$ite(f, 0, \bar{g})$
$OR(f, g)$	$f + g$	$ite(f, 1, g)$
$f \geq g$	$f + \bar{g}$	$ite(f, 1, \bar{g})$
$f \leq g$	$\bar{f} + g$	$ite(f, g, 1)$
$NAND(f, g)$	$\bar{f} + \bar{g}$	$ite(f, \bar{g}, 1)$
$XOR(f, g)$	$f \oplus g$	$ite(f, \bar{g}, g)$
$XNOR(f, g)$	$f \bar{\oplus} g$	$ite(f, g, \bar{g})$

Table 3.2: Boolean functions of two arguments and equivalent representation in terms of the ITE operator

Algorithm 1: ITE

input : Associated variable v, node loson, node hison

output: If the node already exists in the unique table then the pointer to the existing node is returned otherwise a new node is created and stored in unique table and a pointer to the new node is returned

ITE(node I, node T, node E)

begin

if (*terminal case applies to I,T,E*) **then**

return (immediately computed result for terminal conditions)

end

else if (*Computation table has entry (I,T,E)*) **then**

return (result node from computation table)

end

else

pick minimum variable x among roots of I,T,E

PosFactor = `ITE` (I_x, T_x, E_x)

NegFactor = `ITE` $(I_{x'}, T_{x'}, E_{x'})$

R = `FindOrCreate`$(x, PosFactor, NegFactor)$

`InsertToComputationTable`$(hashfunction(I, T, E), address)$

return (R)

end

end

3.1.11 Variable ordering

The size of the ROBDD depends on the variable order. And finding optimal variable order is NP-Complete [10]. Some exact variable ordering algorithm gives excellent result but expensive in time. Sometimes complexity becomes exponential, specially for functions that have few symmetric variable. FYI, symmetric variable are those set

Algorithm 2: Find or create

input : Associated variable v, node loson, node hison

output: If the node already exists in the unique table then the pointer to the existing node is returned otherwise a new node is created and stored in unique table and a pointer to the new node is returned

```
FindORCreateNode(var v, node loson, node hison)

begin
    if (v is actually a constant) then
        if (if this constant does not already exist in unique table) then
            put this constant in the unique table
            return (pointer to constant)
        end
    end
    else if (loson == hison) then
        return (loson)
    end
    else if ((v,loson,hison) node already exists in unique table) then
        return (lpointer to (v,loson,hison) node from unique table)
    end
    else
        create new node = (v,loson, hison)
        put this node in unique table
        return (pointer to this node)
    end
end
```

of variable that does not effect other variables or the ROBDD as a whole when they change places among themselves. Figure 3.10 shows a example of variable ordering complexity for good and bad variable order for the function $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$

Variable ordering algorithms are divided into two classes a) Static and b) Dynamic. In static method, variable order is computed up-front based on the problem structure. It works well for many combinational function that comes from circuits designers build but does not work well for unstructured problems. Thats why, dynamic variable ordering is preferable.

According to the Friedman theorem [14], permuting any top part of the variable order has no effect on the nodes labeled by variables in the bottom part. Permuting any bottom part of the variable order has no effect on the nodes labeled by variables in the top part. In figure 3.11, two adjacent variable layer exchange using Friedman's theorem is shown.

Friedman's theorem is the basis for the sift algorithm , a popular dynamic variable ordering scheme first proposed by Rudell [15]. Later the idea was extended by taking variable symmetry into consideration. This form of dynamic variable sifting is called symmetric sifting and it is investigated by many researchers till today [19], [20], [21],

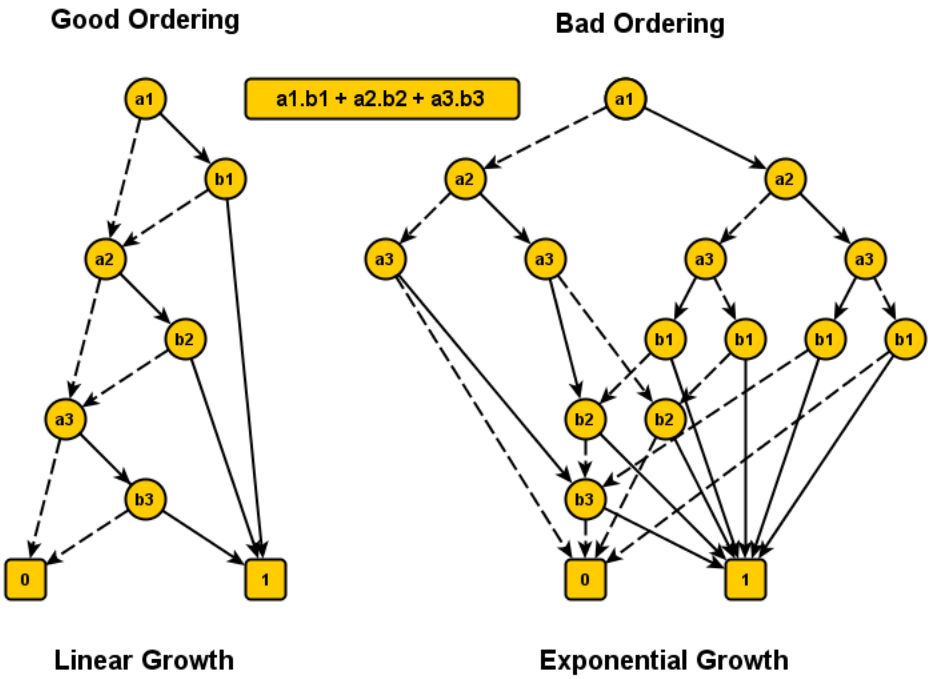


Figure 3.10: Complexity of variable ordering

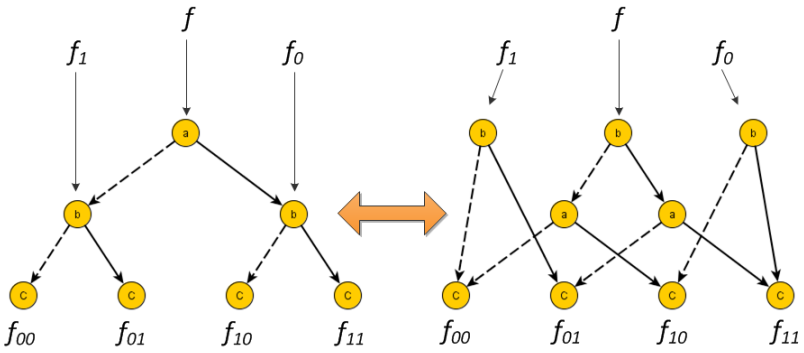


Figure 3.11: Adjacent variable exchange in ROBDD

[22].

3.2 OkBDD features

3.2.1 ROBDD data structure

OkBDD uses pointer based BDD nodes. The node is only 16 bit w.r.t 32-bit OS for minimal basic function. There are 4 more Boolean flags (another 4 byte) for graph processing and visualization tasks. There is a BDD manager class called only "BDD" in the core BDD package. All the BDD formation and variable ordering algorithms are associated with the BDD manager to keep the Node data structure lightweight. OkBDD support dynamic variable ordering. The BDD variable can be declared any time during runtime and ordering algorithm can be invoked anytime selectively. There is a general variable ordering algorithm interface declared and user can use their own implementation of algorithm. The OkBDD also implements separate unique tables for each variable to make variable ordering efficient. The variable ordering problem is nothing but a combinatorial optimization problem and the ordering interface provides the facilities to use any other general optimization algorithm framework with minimal effort. That means, user can use 3rd party optimization framework without knowing any details about the ROBDD data structure.

3.2.2 GraphStream visualization

OkBDD visualization is based on GraphStream [23], [24] - an open source dynamic graph library that provides an easy way to represent dynamic evolving graph in memory, in GUI and save it as popular graph format files. GraphStream provides a framework to handle the evolution of graphs, that is the changes on values stored on edges and nodes of a graph during time and also the "topology" changes of the graph that is the addition, removal and modification of nodes and edges during time. As GraphStream can handle graph evolution, graphs are defined as a "flow of graph events", instead of only a set of nodes, edges and eventually sets of values associated to node and edges. Dynamic graphs events tells when a node, edge or associated value appears, changes and disappears. GraphStream provides the ideal model, view , controller paradigm (MVC). Here , the model is the ROBDD data structure, the view is the visualization which runs on a separate thread and follows the evolving model accurately. Controllers are separate Java classes that initiates the communication between the model and view. Apart from visualization, GraphStream also provides the facilities to save the ROBDD graphs as popular graph formats such as Dot, GraphML, GML, TLP, NET, GEXF etc which allows the graphs to be analyzed by other Graph processing and analysis tools.

3.3 Comparison and analysis

ROBDD formation and manipulation of ROBDD is memory and computation intensive task. ROBDD operations create large amount of intermediate data, performs hashing function frequently. Specially, memory management and garbage collection is vital for processing ROBDDs in machine with limited memory. Most of the BDD packages came out in 1995 to 2000 and at that time memory size was a huge concern. Cache misses and page faults were frequent and degraded the performance and decreased the usefulness of BDD packages.In modern days, memory size requirement

is not a great issue but better memory access and utilization of spatial and temporal locality will still boost-up performance. In summary, raw performance depends on memory size, memory hierarchy and even OS. Also, many packages are not exhaustive in their implementation and thus they remain lightweight.On the other hand, packages like CUDD[25], CMU[26] offers more advanced functionality. In short, comparing different packages only by raw performance and completion time measurement is misleading.

In 2003, Janssen from IBM TJ Watson Research center published a through report on 13 BDD packages at that time [2]. In his own words, he investigated underlying algorithmic and data structure principles that are at the core of each package's implementation and tried to relate the strength of various BDD packages with respect to their usefulness in serious applications. Table 3.4 is adapted from Janssen's paper which shows how the packages faired with respect to the different criteria. We entered our BDD package as the 14th entry in the table and compared according to the same measures that were used for the previous 13 packages. Table 3.3 elaborates all the measurement keys used in Table 3.4. It is worth to mention that, BDD node size can be squeezed to as little as 8 byte but it will decrease the capacity of handling nodes and variables as shown in the case of ABCD and StaticBDD. As OkBDD node uses pointer-based referencing, the node size is expected to increase in 64 bit machines. The authors wanted to extend the functionality of the package in future with bi-decomposition algorithms and functionalities, cube manipulation and logic synthesis oriented architecture.

3.4 Summary

Binary decision diagram is a powerful data structure to manipulate Boolean logic. BDDs are frequently used for logic minimization and equivalence checking. In industry, BDDs are used in logic minimization , formal verification and test pattern

Key	Description
Package	Name of the package
Lang.	Programming language used
T	Integer indices (I) or Pointer (P) based parent-sibling relationship
R	Depth-first traversal(D) / Breadth-first traversal(B)
M	Presence of BDD manager
N_{max}	Max nodes that can be allocated assuming infinite memory
Size	A single BDD node size for basic function
64	Indicates whether the node size will increase if ported to 64 bit system
V_{max}	Max number of variables disregarding machine resources
P	Indicates whether nodes are kept in separate unique tables per variable
GC	Garbage collection scheme
CT	Single (S) / Multiple (M) compute table
DVO	Dynamic variable algorithm supported or not
Year	Year of first release

Table 3.3: Measures for Table 3.4

generation. Its unique ability to represent logic in canonical form and representing exponential amount of information in linear space made it a popular choice for both industry and academia.

Package	Lang.	T	R	M	N_{max}	Size	64	V_{max}	P	GC	CT	DVO	Year
ABCD 0.3	C	I	D	Y	2^{25}	8	-	2^{10}	N	MS	S	N	1997
Alliance 5.0	C	P	D	Y	2^{28}	16	Y	2^{16}	Y	MS/RC16	S	Y	1996
BuDDy 1.9	C/C++	I	D	N	2^{32}	20	N	2^{21}	N	MS	M	Y	1996
CAL (VIS)	C	P	B	Y	2^{28}	16	Y	2^{16}	Y	RC8	S	Y	1994
CMU (VIS)	C	P	D	Y	2^{29}	16	Y	2^{16}	Y	RC8	S	Y	1993
CUDD 2.31	C/C++	P	D	Y	2^{28}	16	Y	2^{16}	N	RC16	S	Y	1995
EST led	C	P	D	N	2^{32}	36	Y	2^{32}	N	RC32	M	N	2000
IBM	C	I	D	Y	2^{30}	16	N	2^{24}	N	MS	S	Y	1999
MONA	C	I	D	Y	2^{24}	16	N	2^{16}	N	MS	M	N	1997
PDT	C	I	D	N	2^{31}	24	N	2^{16}	N	MS	M	N	1993
StaticBDD 1.0	C++	I	D	N	2^{23}	8	N	2^{9}	N	MS	S	N	1999
TiGeR 3.0	C	P	D	Y	2^{30}	16	Y	2^{16}	Y	RC16	M	Y	1994
TUDD	C	I	D	Y	2^{30}	20	N	2^{16}	Y	RC16	S	Y	1998
OkBDD	JAVA	P	D	Y	2^{31}	16	Y	2^{31}	Y	MS	S	Y	2012

Table 3.4: Comparison of BDD packages adapted from Janssen's paper [2]

CHAPTER 4

Technology mapping

4.1 The technology mapping problem

The technology mapping problem is in general a combinatorial optimization problem with the goal of optimally transform a technology independent Boolean logic into physical logic elements available through one particular technology.

In general the problem is defined as : given a set of gates L, called the library, and a Boolean network G, let M be the set of Boolean networks constructed using gates from L that are functionally equivalent to G. In this scenario, M is called the mapped networks. The goal of mapping is to find a optimal mapped network that minimizes some objective such as area. delay etc subject to certain constraints such as timing.

The general theory is applicable to both standard cells and look-up table (LUT) based FPGAs. (A look-up table with k inputs, called a k-LUT is a configurable gate that can implement any Boolean function of k variables.)

4.2 History and background

In the general terms, the mapping problem is intractable since it is hard to enumerate either implicitly or explicitly the elements of M that optimizes the desired goal. The first most significant algorithm was proposed by Keutzer [27]. He proposed a significant simplification of the mapping problem by restricting the set of mapped networks considered during mapping to be those networks that are structurally similar to [27]. As this approach depends on structural matching of network , this approach is defined

as structural matching approach.

The main idea behind the structural mapping algorithm is a simple one. It is assumed that the original Boolean network G already has a good structure. Mapping is then done by a process of local re-writing. In this process, a single output sub-network N of G is identified that is functionally equivalent to a gate g in L and replace N by g.

The problem is, there are many ways to do this re-writing and finding the optimal re-writing in polynomial time is tricky. Keutzer proved that for certain classes of networks (trees) and for certain cost functions (delay in the constant-delay timing model and area) it is possible to compute the optimally mapped network by a dynamic programming algorithm.

For this to work, it is assumed that G has a good structure for the final network. This is ensured by applying technology independent logic synthesis algorithms before the technology mapping steps to the initial Boolean network entered by the circuit designer to obtain G. Structural mapping is not expected to significantly change the network structure of G, but merely to convert it in to a similar network built from gates in L.

In Keutzers original work [27], the subject graph is first partitioned in to a forest of trees. Each library gate is decomposed in to a tree of two input and gates and inverters. These trees are called pattern graphs. A multi-input gate may lead to multiple pattern graphs since the decomposition is not unique. A pattern graph (and thereby an equivalent library gate) is matched at a node n by checking for structural isomorphism using a tree matching algorithm.

Richard Rudell extended tree matching algorithm to the case where the pattern graphs could be leaf DAGs [28]. This allowed non-tree gates such as multiplexers and xors to be matched. He also observed that by replacing every wire in the subject graph by a pair of inverters, and by adding a wire gate to the library (whose pattern

consists of a pair of inverters in series), the set of matches is larger.

A radically new method to reduce structural bias was proposed by Lehman et al. [18]. They observed that there are many different ways in which a subject graph can be derived from a Boolean network. For example, a multi-input AND can be decomposed in to a tree of 2-input AND gates in a variety of ways. However, in the mapping algorithm described above, a decision has to be made a priori as to which decomposition is to be used to generate the subject graph. Thus, certain matches that would have been detected with a different decomposition are no longer detected.

A different approach was inspired by a solution in the FPGA domain. Kukimoto extended the set of matches explored at a node by using a general DAG matching procedure instead of tree matching [29]. Thus both the subject graph and the pattern graphs are allowed to be general DAGs in Kukimotos extension. This significantly increases the number of matches found at a node, especially when partitioning the subject graph into a forest of trees would lead to many small trees which eventually will result in many local optima but will be globally sub-optimal.

Mailhot and De Micheli initiated a different line of research with their proposal for Boolean matching [30]. Keutzers work [27] (and later in Lehman et al. [18] and Kukimoto [29]) the actual process of matching a gate (or more precisely, its pattern graph) with a sub-graph H rooted at a node n is through graph isomorphism. Since a library gate, especially complex gates, have many possible decompositions into pattern graphs, only a subset of all possible decompositions is used in practice. Now if the sub-graph of the subject graph is not structurally identical to any one of the decompositions of the library gate, a match may not be found, even though replacing the H by the gate is valid. One can think of this as being a more local structural bias as opposed to the global bias that we were so far concerned with.

Boolean matching addresses local structural bias by directly matching a gate with the sub-graph H by comparing their Boolean functions. Thus, with Boolean matching

there is no need for pattern graphs.

Kravets and Sakallah advanced their research in the line of constructive synthesis algorithm proposed by [31] and subsequently improved by Mishchenko et al. [32]. In these algorithms, the structure of the subject graph is not used at all instead the subject graph is used to construct a representation of the Boolean functions (using BDDs or truth-tables), and then a decomposition algorithm is applied to the functions to obtain the mapped network. At first, this may seem like the answer to our original technology mapping formulation. However, the chief conceptual drawback of these algorithms is that they are committed to a specific decomposition scheme. Thus they are not able to explore the full space of mapped solutions, i.e. M. Therefore, they just introduce a structural bias of a different kind.

In FPGA mapping, local structural bias is not a problem since a k-LUT can implement any function of k variables or less. The main challenge is to enumerate the possible different sub-networks rooted at a node n in the subject graph that is implementable by a k-LUT.

Cong and Ding presented a network flow based algorithm Flowmap that can identify a single sub-network rooted at a node n that minimizes depth [33].The limitation of Flowmap is, it produces only one cut that minimizes depth. However, the network flow based method that it uses to find the cut cannot be extended easily to handle other cost functions. Cong and Ding explored further techniques to enumerate all cuts [34]. This work was later improved by Pan and Lin who presented an elegant algorithm to enumerate all cuts [35]. It is worth mentionable that Chen and Cong adapted the algorithm proposed by Lehman et al [18] for FPGA mapping [36].

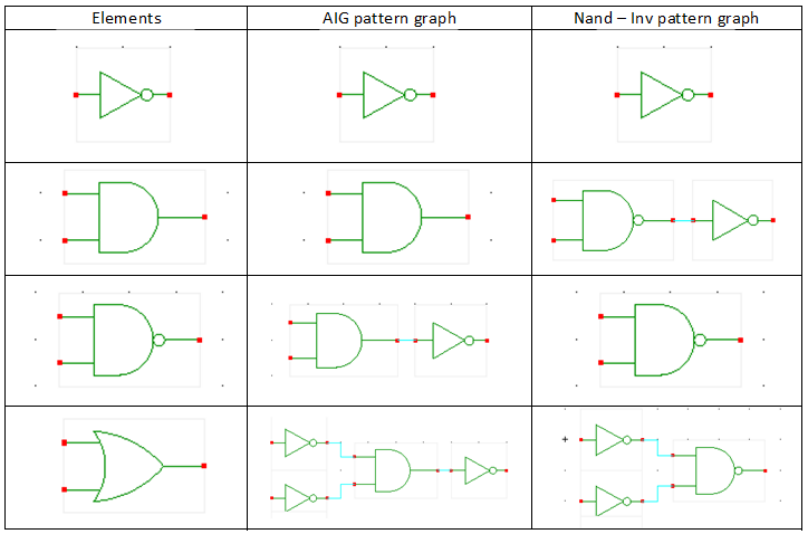


Figure 4.1: Pattern graph of different logic elements of a typical standard library

4.3 Technology mapping using structural hashing and dynamic programming

Technology mapping is the process of mapping logic independent circuit to technology dependent circuit using resources available in the standard library. The idea is to represent each logic elements in standard cell library with base function which will produce circuit graph with base elements. This graphs are called pattern graphs. The technology independent circuit is also represented by the same base function and the resultant graph is called subject graph. In the last crucial stage, pattern graphs are structurally matched with sub-graphs of subject graph. Subsequently, the matched portion of subject graph is replaced with corresponding standard cell of the matched pattern graph. Generally, there are multiple choices for matches and the best overall match is found by dynamic programming based algorithm and appropriate cost model.The cost model can be function of one or combination of more than one factor such as delay, area, power etc .

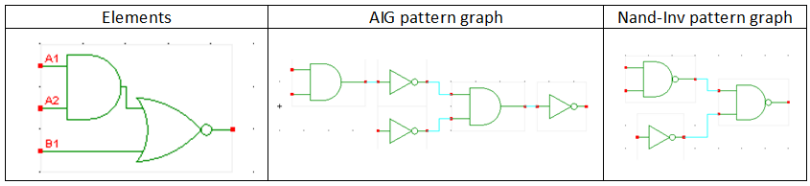


Figure 4.2: Asymmetric pattern graph example

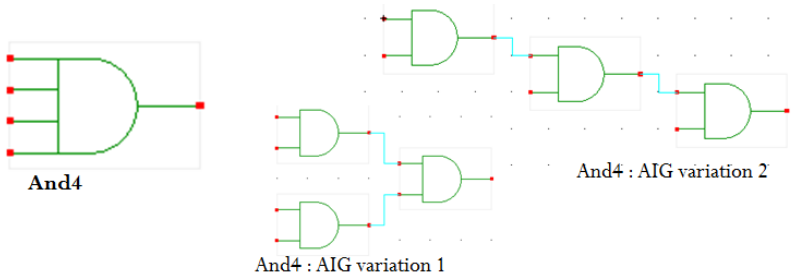


Figure 4.3: AIG pattern variation of 4-input AND gate

4.3.1 Base function and pattern graph

The base function is the smallest subsets to represent circuit. Two types of base functions are most frequently used namely 1) Inverter and 2-input AND 2) Inverter and 2-input NAND. They produce AIG (AND-Invert Graph) and NAND2-Invert pattern graphs respectively. Figure 4.1 shows an example of a typical standard cell library and pattern graphs. The pattern graphs of Figure 4.1 are symmetric ie, any path from primary input to primary output traverse through the same kind of elements. Figure 4.2 shows an example of asymmetric pattern graph. Asymmetric pattern graph has different structure when input orders are changed and hence all different pattern should be tried in the matching algorithm. The logic gates with more than 3-input usually have different pattern graph structures. For example, 4-input AND gate can have two different structure as shown in figure 4.3.

Algorithm 3: Pattern matching

input : Subject Graph Node *subject, Pattern Graph Node *pattern

output: Subject Graph Node *matchedNode

Match(node subject, node pattern)

begin

if (*NodeType(pattern) == INPUT*) **then**
return matchedNode

end

else if (*subject == Leaf of Subject Graph*) **then**
return null

end

else if (*NodeType(pattern) != NodeType(subject)*) **then**
return null

end

else if (*NodeType(pattern) != NOT*) **then**
return `Match`((*child*(*subject*), *child*(*pattern*))

end

else
return `Match`(*left*(*subject*), *left*(*pattern*)) && `Match`
(right(subject),right(pattern)) ||
`Match`
(left(subject),right(pattern))&&`Match`(*right*(*subject*), *left*(*pattern*))

end

end

4.3.2 Structural hashing and pattern matching

Algorithm 3 shows the pattern matching algorithm implemented in OkCAD. The matching criteria is described below

- Pattern graph's leaf (Input) matches anything in subject graph
- Pattern graph's Inverter matches inverter in subject graph
- Pattern graph's And2/Nand2 matches And2/Nand2 in subject graph

4.3.3 Optimal covering with dynamic programming

Optimal graph covering is basically a form of the Knapsack optimization problem where minimum cost is better solution. The Knapsack problem is a combinatorial optimization problem where, given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. This kind of optimization problem can be efficiently solved by Dynamic Programming. The DP based technology mapper in OkCAD does the following steps,

- Start dynamic programming from primary inputs to primary outputs
- Cost at the primary inputs are zero
- Traverse the DAG in topological order from inputs toward the outputs
- At each node V in the subject graph, try all the pattern graph in library, find the best match (Lowest cost) and save the result for later use
- Start backtracking from primary outputs to primary inputs
- For each node pick the best match and re-factor out the matched pattern tree out of the subject graph and replace it with the matched element from standard library.
- Repeat the same on the remaining portion of the subject graph.

4.3.4 Technology mapping features in OK CAD tools

OK CAD tools has a basic technology mapping tool. Standard library contains all the pattern graph in AIG form. The library can contain different gates of same logic. The key features of the technology mapping tool are

- Algorithm is an efficient dynamic programming algorithm (O(n)).
- It visits each node maximum once and the matching routine is a depth-first search which goes max depth is equal to the max depth in the library pattern graphs.
- Library supports AIG, Nand-Inv base function
- Library creates all possible combination of pattern graphs for each element in the library.
- User can create new functions and add to the library (for research purposes).

4.4 Summary

Technology mapping process is one of the most complex process in EDA/CAD. Nowadays industrial standard library contains many different cells of the same function with different architecture, area, speed etc. So, there are a lot of options available and technology mapping algorithms should be smart enough to take all the factors into consideration and produce the best mapping results under the given constraints. Structural matching is one of the basic way to compare and map logic. AND-INVERT graphs or NAND-INVERT graphs based matching is fast and produces good results in a reasonable time. In this work, dynamic programming is used for the global optimized technology mapping which runs in linear time and produces the exact optimized solution according to the given cost function.

CHAPTER 5

ULE : Unified logical effort

5.1 Background

In this section, the authors presents a brief overview of Logical Effort (LE) [37] and its advance extension called Unified Logical Effort (ULE) [38]. Both LE and ULE are simple back of envelope method to estimate delay and size logic gates for optimal delay. As, ULE is the basis of this paper, ULE is discussed more elaborately. In addition, it is explained how branch calculation is modeled in ULE and in the last sub-section the load distribution problem is explained.

5.1.1 Logical effort

The method of Logical Effort was first proposed by Sutherland et al [37]. According to logical effort , the delay of a logic gate model is estimated with a linear function of the load being driven as

$$D = f + p = g * h + p = g * \frac{c_l}{c_i} + p \tag{5.1}$$

where g is the logical effort , $h = \frac{c_l}{c_i}$ is the electrical effort, $f = gh$ is the effort delay and p is the parasitic delay of the gate. Minimum delay is estimated as

$$D = NF^{\frac{1}{N}} + P \tag{5.2}$$

where $F = GH$ is referred to as path effort. P is the path parasitic delay and N is the number of gates in the path. G is the path logical effort and expressed as the product of all the gate logical effort. H is the path electrical effort and expressed as

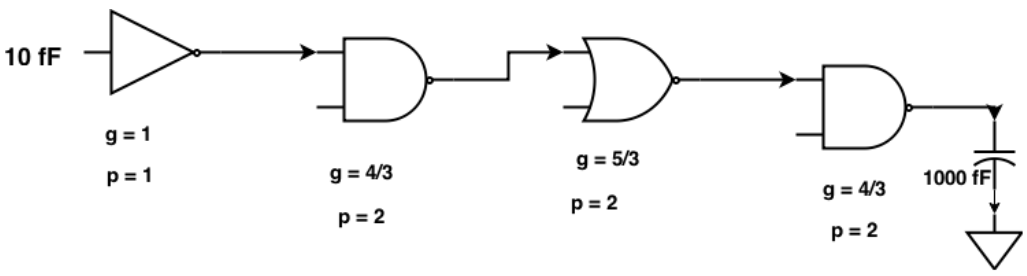


Figure 5.1: Logical Effort (LE) example

the product of all gate electrical effort. Minimum delay is achieved when F is equally distributed to each gate on the path.

For example, in Figure 5.1 there are 4 logic gates – one inverter, two 2-input nand gate and one 2-input nor gate. For this circuit , LE calculations are shown below,

path logical effort , $G = 1 \cdot \frac{4}{3} \cdot \frac{5}{3} \cdot \frac{4}{3} = \frac{80}{27}$

path electrical effort, $H = \frac{1000}{10} = 100$

path parasitic effort, $\mathrm{P} = 1 + 2 + 2 + 2 = 7$

total path effort, $\mathrm{F} = G * H = \frac{80}{27} * 100 = \frac{8000}{27}$

stage effort, $\hat{f} = (\frac{8000}{27})^{\frac{1}{4}}$

optimal delay, $D = 4 * (\frac{8000}{27})^{\frac{1}{4}} + 7$

5.1.2 Unified logical effort

Although, Logical Effort (LE) is an amazing tool to estimate delays it has some drawbacks. The optimization rules of Logical Effort only considers logic gates and does not consider the interconnect wires. As the technology is scaling fast, the interconnect is becoming a important factor to design high-speed low power circuits.The useful LE rule that path delay is minimum when the efforts of each of the stages are equal also becomes invalid in the presence of interconnects, because interconnect have fixed capacitance which do not co-relate well with the capacitances of the logic gates which follows a geometric progression when optimized.

Unified Logical Effort (ULE) proposed by Morgenshtein, Friedman et al [38], is a

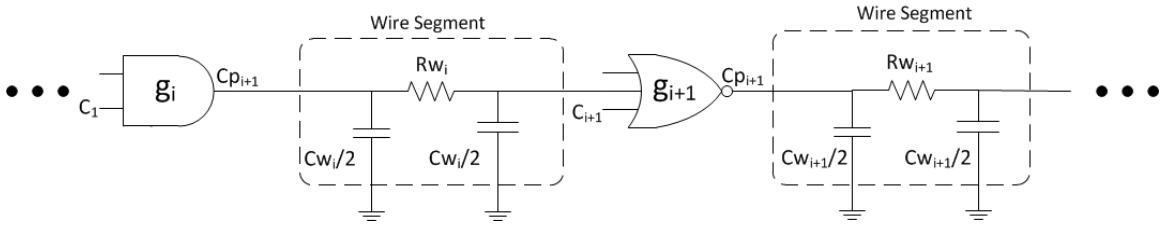


Figure 5.2: Practical logic path with logic gates and RC interconnect.

simple model that overcomes the drawbacks of original Logical Effort method. Unlike LE, ULE can size logic gates for optimal delay in the presence of RC interconnect. ULE treats a broad scope of design problems with a simple single analytic model. combining logic and interconnect delay optimization.

In ULE, a typical practical circuit is represented with logic gates and RC interconnect as shown in figure 5.2. The RC interconnect is represented with a π-model and Elmore delay model is used for delay estimation [39]. According to ULE, the combined delay expression is,

$$D_i = R_i \cdot (C_{pi} + C_{wi} + C_{i+1}) + R_{wi} \cdot (0.5 \cdot C_{wi} + C_{i+1}) \tag{5.3}$$

where R_i is the effective output resistance of the gate i, C_{pi}, is the parasitic output capacitance of gate i, C_{wi} and R_{wi}, are the wire capacitance and resistance of segment i respectively and C_{i+1} is the input capacitance of the gate i.

By introducing the delay of a minimum-sized inverter as a technology constant $\tau = R_0.C_0$, the expression is rewritten as,

$$\begin{aligned} D_i = \tau \cdot d_i = \tau \cdot \left[\frac{R_i}{R_0} \cdot \frac{(C_{wi} + C_{i+1} + C_{pi})}{C_0} \right] \\ + \tau \cdot \left[\frac{R_{wi}}{R_0 \cdot C_0} \cdot (0.5 \cdot C_{wi} + C_{i+1}) \right] \end{aligned} \tag{5.4}$$

The rewritten expression is similar to [40], [41] and [42]. In ULE, the stage delay , normalized with a minimum inverter delay τ, is expressed in LE form as the following expression,

$$d_i = g_i \cdot \left(h_i + \frac{C_{wi}}{C_i} \right) + \frac{R_{wi} \cdot (0.5 \cdot C_{wi} + C_{i+1})}{\tau} + p_i \tag{5.5}$$

where $g_i = \frac{R_i \cdot C_i}{R_0 \cdot C_0}$ is the logical effort related to the gate topology , $h_i = \frac{C_{i+1}}{C_i}$ is the electrical effort of the gate, and $p_i = \frac{R_i \cdot C_{pi}}{R_0 \cdot C_0}$ is the delay factor the parasitic impedance. The scaling factor is denoted as x_i. Expression $C_i = C_0 \cdot g_i \cdot x_i$ and $R_i = R_0/x_i$ relates gate resistance and gate capacitance respectively to the scaling factor.

ULE also defines capacitive interconnect effort as h_w and resistive interconnect effort as p_w. h_w and p_w are expressed as

$$h_{wi} = \frac{C_{wi}}{C_i} \tag{5.6}$$

$$p_{wi} = \frac{R_{wi} \cdot (0.5 \cdot C_{wi} + C_{i+1})}{\tau} \tag{5.7}$$

Using 5.5, 5.6 and 5.7, the final expression of ULE delay for a single stage is

$$d = g \cdot (h + h_w) + (p + p_w) \tag{5.8}$$

The expression for N-stage logic path is,

$$d = \sum_{i=1}^{N} g_i \cdot (h_i + h_{wi}) + (p_i + p_{wi}) \tag{5.9}$$

In case of short wire R_w can be neglected. also, when the wire impedance along the logic path is negligible, the ULE delay expression reduces to original LE expression.

Condition for minimum delay and optimum sizing can be derived by using a 2-stage logic path as shown in figure 5.2 and extending the result for general case. The ULE expression for the total delay for the circuit in figure 5.2 is

$$\begin{aligned} d = {} & g_i \cdot (h_i + h_{wi}) + (p_i + p_{wi}) \\ & + g_{i+1} \cdot \left(h_{i+1} + h_{w_{i+1}} \right) + \left(p_{i+1} + p_{w_{i+1}} \right) \end{aligned} \tag{5.10}$$

Putting $C_{i+1} = h_i \cdot C_i$ into 5.10 the delay can be expressed in terms of h_i

$$\begin{aligned} d = g_i \cdot \left(h_i + \frac{C_{wi}}{C_i} \right) + p_i + \frac{R_{wi} \cdot (0.5 \cdot C_{wi} + h_i \cdot C_i)}{R_0 \cdot C_0} \\ + g_{i+1} \cdot \left(\frac{C_{i+2} + C_{w_{i+1}}}{h_i \cdot C_i} \right) + p_{i+1} + p_{w_{i+1}} \end{aligned} \tag{5.11}$$

The condition for optimal gate sizing is determined by equating the derivative of the delay with respect to the gate size to zero (see [1] for detail derivation)

$$\left(g_i + \frac{R_{wi} \cdot C_i}{R_0 \cdot C_0} \right) \cdot h_i = g_{i+1} \cdot \left(h_{i+1} + h_{w_{i+1}} \right) \tag{5.12}$$

The optimal condition of ULE 5.12 converges to $g_i \cdot h_i = g_{i+1} \cdot h_{i+1}$, the original LE equation for optimal condition, when wires are ignored , ie , when $h_{wi} = 0$ and $R_{wi} = 0$

Multiplying equation 5.12 by $R_0 \cdot C_0$ and using the relationships $h_i = \frac{C_{i+1}}{C_i}, C_i = C_0 \cdot g_i \cdot x_i, R_i = \frac{R_0}{x_i}$, the following simplified expression can be found

$$(R_i + R_{wi}) \cdot C_{i+1} = R_{i+1} \cdot \left(C_{i+2} + C_{w_{i+1}} \right) \tag{5.13}$$

The authors of ULE further developed the optimal condition for any gate i based to the characteristic that the total delay is the sum of the upstream and downstream delay components as shown below,

$$D_{C_i} = \left(R_{i-1} + R_{w_{i-1}} \right) \cdot C_i = \left(R_{i-1} + R_{w_{i-1}} \right) \cdot C_0 \cdot g_i \cdot x_i \tag{5.14}$$

$$D_{R_i} = R_i \cdot (C_{i+1} + C_{w_i}) = \frac{R_0}{x_i} \cdot (C_{i+1} + C_{w_i}) \tag{5.15}$$

$$D_i = D_{C_i} + D_{R_i} + constant \tag{5.16}$$

When the total delay is minimum , the sum of the differential of the delay components with respect to the sizing factor x_i is equaled to 0

$$\frac{\partial D_{C_i}}{\partial x_i} = \left(R_{i-1} + R_{w_{i-1}}\right) \cdot C_0 \cdot g_i \tag{5.17}$$

$$\frac{\partial D_{R_i}}{\partial x_i} = -\frac{R_0}{{x_i}^2} \cdot (C_{i+1} + C_{w_i}) \tag{5.18}$$

$$\frac{\partial D_i}{\partial x_i} = \frac{\partial D_{C_i}}{\partial x_i} + \frac{\partial D_{R_i}}{\partial x_i} = 0 \tag{5.19}$$

After solving equation 5.19 , the optimal sizing factor $x_{i_{opt}}$ is expressed as

$$x_{i_{opt}} = \sqrt{\frac{R_0}{\left(R_{i-1} + R_{w_{i-1}}\right)} \cdot \frac{(C_{i+1} + C_{w_i})}{C_0 \cdot g_i}} \tag{5.20}$$

A general optimum condition is derived by substituting $x_{i_{opt}}$ into equation 5.13.

$$\begin{aligned} \left(R_{i-1} + R_{w_{i-1}}\right) \cdot C_i &= R_i \cdot (C_{i+1} + C_{w_i}) \\ &= \sqrt{[\left(R_{i-1} + R_{w_{i-1}}\right) \cdot C_0 \cdot g_i] \cdot [R_i \cdot (C_{i+1} + C_{w_i})]} \end{aligned} \tag{5.21}$$

From Equation 5.21 it is evident that the minimum delay is achieved when the downstream delay component (due to C_i) and the upstream delay component (due to R_i) of an optimally sized gate are both equal to the geometric mean of the upstream and downstream delays that would be obtained if the gate is arbitrarily sized. In summary,

$$D_{R_{i_{opt}}} = D_{C_{i_{opt}}} = GM[D_{R_i}, D_{C_i}] \tag{5.22}$$

The total delay D_i is the summation of 4 delay components : the constant delays $0.5 \cdot R_{w_{i-1}} C_{w_{i-1}}$ and $0.5 \cdot R_{w_i} C_{w_i}$ and the variable delays $D_{C_i} = \left(R_{i-1} + R_{w_{i-1}}\right) \cdot C_i$ and $D_{R_i} = R_i \cdot (C_{i+1} + C_{w_i})$ that are dependent on sizing factor x_i.

In 1991, Vemuru et al showed that the drive ability of a gate is related to size of the gate and can be represented by a ratio of input capacitance [43]. Equation 5.12

can be rewritten to find an expression for the required input capacitance of each gate as shown below

$$
\begin{aligned}
C_{i_{opt}} &= \sqrt{\frac{g_i}{g_{i-1} + \frac{R_{w_{i-1}} \cdot C_{i-1}}{R_0 \cdot C_0}} \cdot C_{i-1} \cdot (C_{i+1} + C_{w_i})} \\
&= \underbrace{\sqrt{C_{i-1} \cdot C_{i+1}}}_{\textbf{LE}} \cdot \underbrace{\sqrt{\left(1 + \frac{C_{w_i}}{C_{i+1}}\right)}}_{\textbf{wire capacitance}} \cdot \underbrace{\sqrt{\frac{g_i}{g_{i-1} + \frac{R_{w_{i-1}} \cdot C_{i-1}}{R_0 \cdot C_0}}}}_{\textbf{logical effort and wire resistance}}
\end{aligned}
\tag{5.23}
$$

It is worth to note that, the first of the resulting expressions described in Equation 5.23 is similar to the condition that was described in original Logical Effort model for a path of identical gates [37]. The second part expresses the influence of interconnect capacitance. When wire length is zero this part vanishes. The last part is related to the influence due to wire resistance and the difference among the individual logical efforts of neighboring gates. As a whole the expression illustrates the quadratic relationship between the size of the neighboring gates. From, equation 5.23 it is also evident that, if wire is ignored this ULE expression converges to the original logical effort expression.

In order to simplify the solution of equation 5.23 the authors of ULE used a relaxation method. They proposed an iterative calculation along the path while applying the optimum conditions [1]. In this method, each capacitance along the path is iteratively replaced by the capacitance determined from applying the optimum expression to two neighboring logic gates.

Equation 5.23 is the single most important derivation in ULE method. This expression can be used to calculate optimum sizing of logic gates in a logic path in the presence of arbitrary RC interconnect. If the lengths of the wire segments of a logic path is known , the wire capacitance can easily be calculated from the technology data. In consequence, any logic path with RC interconnect can be properly sized for optimal delay by iteratively solving for each input capacitance of the logic gates in

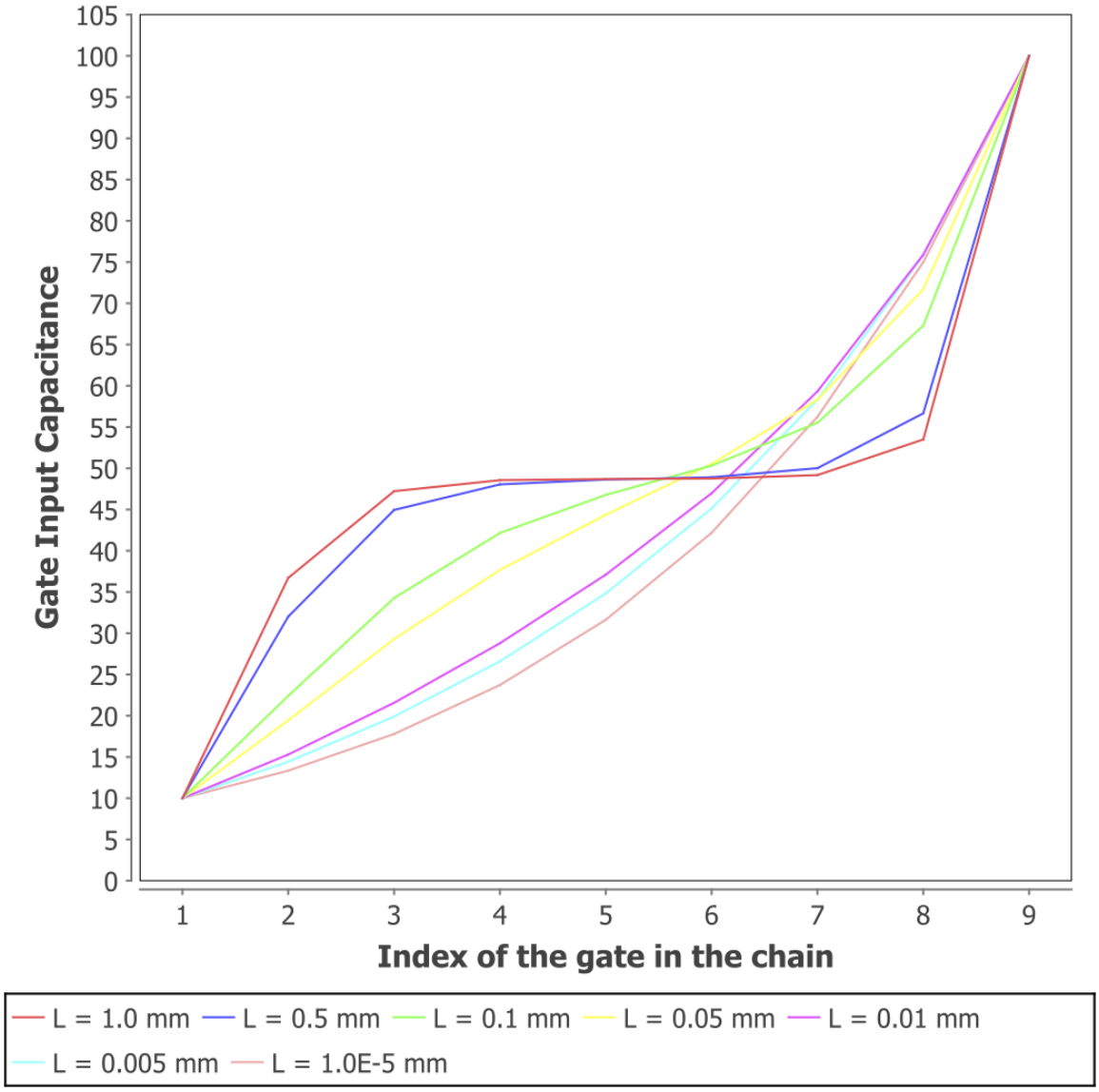


Figure 5.3: Optimization with ULE for a chain of nine NAND gates with equal wire segments in between for a variety of wire length. All the gate sizes are normalized with respect to C_0. Input capacitance of the first and the last gates are $10 \cdot C_0$ and $100 \cdot C_0$ respectively and H = 10.

the path using equation 5.23. The authors of ULE showed that, the result converges to 95 percent of optimal after only 3 iteration.

Figure 5.3 and 5.4 were also included in the original ULE paper [38] and show example logic chain sized by ULE. In the first example, the logic path contains 9

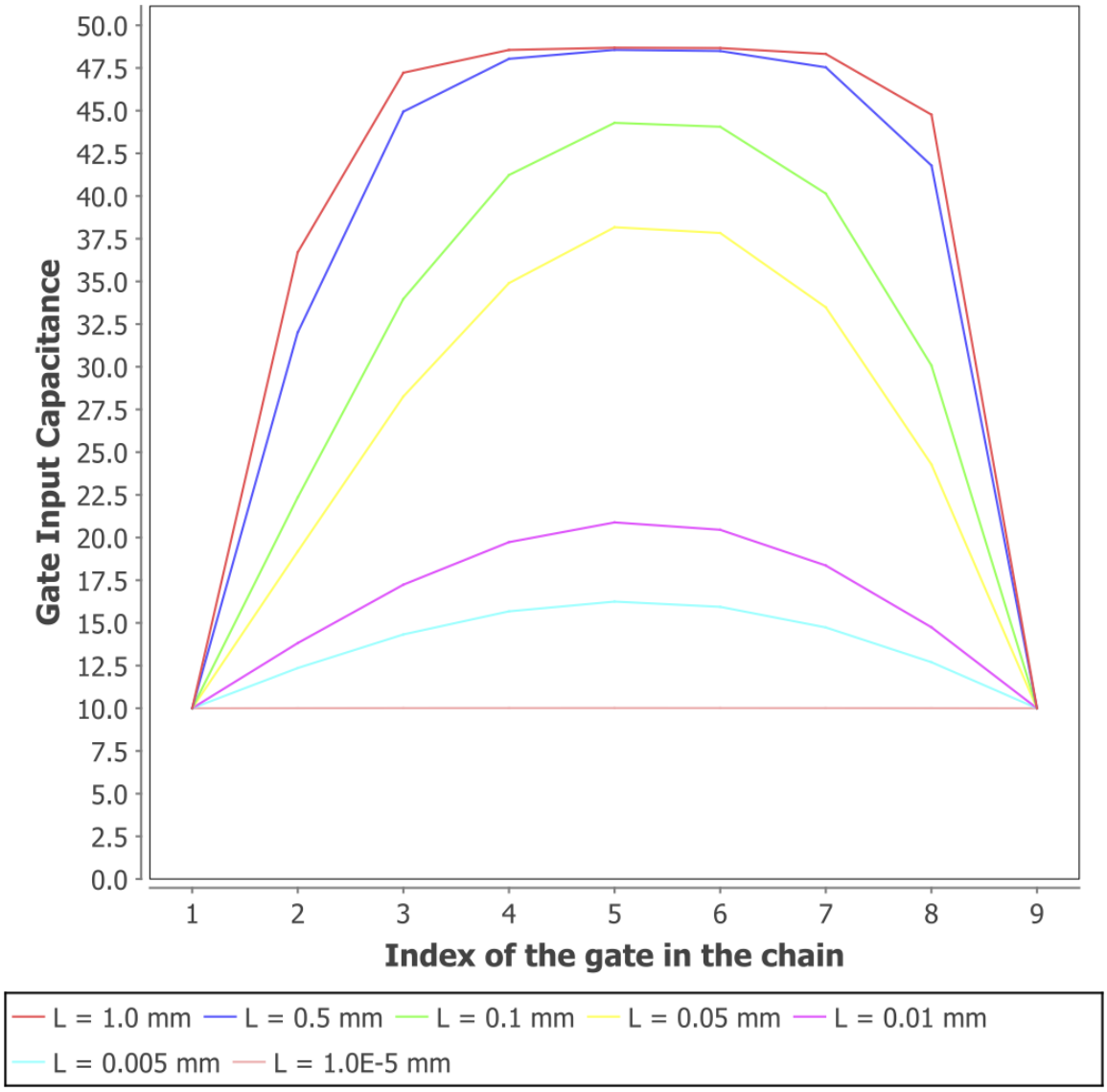


Figure 5.4: The optimization with ULE for a chain of nine NAND gates with equal wire segments in between for a variety of wire length. All the gate sizes are normalized with respect to C_0. Input capacitance of the first and the last gates are $10 \cdot C_0$ and $10 \cdot C_0$ respectively and H = 1.

identical logic gates separated by equal length RC wire segments of length L. A 65 nm technology used in this example. The input capacitance of the first and last logic gates are $10 \cdot C_0$ and $100 \cdot C_0$. Several data-sets were plotted for different L in figure 5.3. Note that, the solutions range between two limits. For near zero wire length the

solution converges to LE optimization and for long wires the gate sizes in the middle stage of the path converges to a fixed value $x_{opt} \simeq 50$ which matches with the repeater insertion methods described in [44] and [45].

The second example is shown in Figure 5.4. In this example, the input capacitance of the first and last logic gate is the same $10 \cdot C_0$, ie, H = 1. In this case, no scaling is performed by ULE in the absence wires and converges to LE solution. And for long wires, the gate sizes again converges to a fixed value.

5.1.3 ULE expressions for long wire segments

In the example in the previous sub-section it is already shown that, the gate sizing optimization process converges to the scale factor x_{opt} in case of long wire segments. In the case of equal interconnect segments, the scale factor is independent of wire length.

When long wires are assumed, C_{w_i} and $R_{w_{i-1}}$ dominate the expression in equation 5.23. The authors of ULE paper derived the scale factor of a general gate for the case of long wires

$$x_{i_{opt}} \approx \sqrt{\frac{R_0 \cdot C_{w_i}}{R_{w_{i-1}} \cdot C_0 \cdot g_i}} = \sqrt{\frac{c_w \cdot R_0}{r_w \cdot C_0 \cdot g_i}} \cdot \sqrt{\frac{L_i}{L_{i-1}}} \tag{5.24}$$

using the relationships $C_{w_i} = c_w \cdot L_i$ and $R_{w_i} = r_w \cdot L_i$, where r_w and c_w are the resistance and capacitance of the wire per unit length and L_{i-1} and L_i are the wire lengths before and after the logic gate g_i, respectively. Note that scale factor of gate in the case of long wire segments depends only upon the ratio of the length of the adjacent wire segments.

A general optimum condition is derived for the long wire case similar to equation 5.21

$$R_{w_{i-1}} \cdot C_i = R_i \cdot C_{w_i} = \sqrt{[R_{w_{i-1}} \cdot C_0 \cdot g_i] \cdot [R_0 \cdot C_{w_i}]} \tag{5.25}$$

Equation 5.25 implies that the minimum delay is achieved when the downstream and upstream delay components of an optimally sized gate are both equal to the geometric mean of the upstream and downstream delays that would be obtained for an arbitrary sized gate.

If all the wire segments are equal , () the capacitance and resistance of all the wire segments are equal to C_w and R_w respectively), the scaling factor x_{opt} is independent of the wire length and expression in equation 5.24 reduces to

$$x_{i_{opt}} = \sqrt{\frac{c_w \cdot R_0}{r_w \cdot C_0 \cdot g_i}} \tag{5.26}$$

This expression can be used as an extension of basic repeater sizing equation. The advantage of this expression is that the size for any logic gate can be determined according to original LE. For the special case of Inverter-based repeater insertion the expression reduces further to

$$x_{i_{opt}} = \sqrt{\frac{c_w \cdot R_0}{r_w \cdot C_0}} \tag{5.27}$$

Equation 5.27 agrees with the optimal scaling factor expression found for optimal repeater insertion by Bakoglu et all [46] .

In addition, similar to equation 5.25 , the optimal sizing condition for repeater is given below,

$$R_{rep} \cdot C_w = C_{rep} \cdot R_w \tag{5.28}$$

So, using similar intuition as previously discussed for general case and long wire segment case, the best sizing of a repeater is found when the delay component $R_w \cdot C_{rep}$ due to the repeater capacitance is equal to the delay component $R_{rep} \cdot C_w$ due to the effective resistance of the repeater.

5.1.4 Application of ULE to repeater insertion problems

The authors of ULE paper [38] showed two example of applications.

A Wire layout constraint

Given a wire of total length L consisting of two unequal wire segments of length L_1 and L_2 , the optimal size of the repeater located between the wire segments is

$$x_{rep_{opt}} = \sqrt{\frac{c_w \cdot R_0}{r_w \cdot C_0}} \cdot \sqrt{\frac{L_2}{L_1}} \tag{5.29}$$

B Cell size constraint

Given a repeater of size x_rep diving a wire of total length L into two wire segments, the optimal wire segment lengths $L_{1_{opt}}$ and $L_{2_{opt}} = L - L_{1_{opt}}$ is calculated by

$$\frac{L_{2_{opt}}}{L_{1_{opt}}} = \frac{x_{rep}^2}{\frac{c_w \cdot R_0}{r_w \cdot C_0}} \tag{5.30}$$

5.1.5 ULE method handling branch

ULE optimization expressions can be derived for more practical case where logic gates in logic path contains branches or multiple fan-out. The multiple fan-out scenario can be depicted by figure 5.5 which defines a theoretical framework for delay minimization of circuits with side branches and multiple fan-out paths. The circuit shows the general structure containing a side branch with RC interconnect and/or a fan-out load with arbitrary capacitance.

Using similar method as in Equation 5.11, the ULE expression for total delay containing branches and fan-outs is written as

$$
\begin{aligned}
d =& g_i \cdot \left[h_i + h_{w_i} + \frac{C_{b1_i} + C_{f1_i}}{C_i} + \frac{C_{b2_i} + C_{f2_i}}{C_i} \right] \\
&+ \frac{R_{w_i}}{\tau} \times \left[0.5 \cdot C_{w_i} + h_i \cdot C_i + C_{b2_i} + C_{f2_i} \right] \\
&+ g_{i+1} \times \left[\frac{C_{w_{i+1}} + C_{i+2} + C_{b1_{i+1}} + C_{f1_{i+1}} + C_{b2_{i+1}} + C_{f2_{i+1}}}{h_i \cdot C_i} \right] \\
&+ \frac{R_{w_{i+1}}}{\tau} \times \left[0.5 \cdot C_{w_{i+1}} + C_{i+2} + C_{b2_{i+1}} + C_{f2_{i+1}} \right]
\end{aligned}
\tag{5.31}
$$

where $\tau = R_0 \cdot C_0$ is the minimum sized inverter delay. By the equating the derivative of the delay with respect to the gate size to zero

$$
\left(g_i + \frac{R_{w_i} \cdot C_i}{\tau} \right) \cdot h_i = g_{i+1} \times \left(h_{i+1} + h_{w_{i+1}} \cdot \underbrace{\frac{C_{b1_{i+1}} + C_{f1_{i+1}} + C_{b2_{i+1}} + C_{f2_{i+1}}}{C_{i+1}}}_{\textbf{branches and fanouts}} \right)
\tag{5.32}
$$

It should be noted that the branch wire resistance R_{b_i} is not part of the optimum condition since the resistance is not along the path where Elmore delay is calculated. Also, in case of zero fanouts or branch interconnects this expression converges to equation 5.12 .

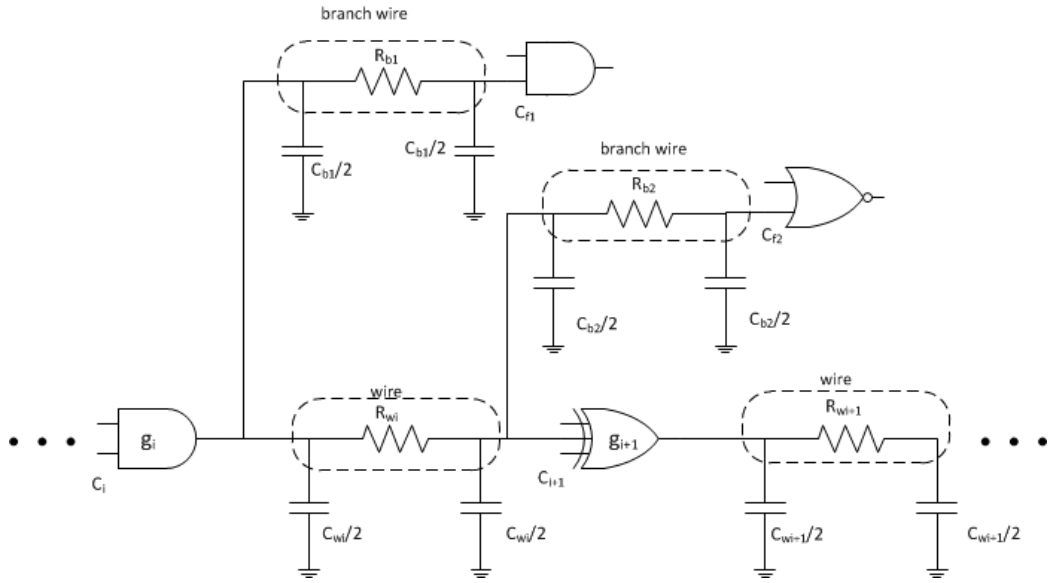


Figure 5.5: ULE delay calculation with branching

By following the similar iterative process of gate size shown previously , equation 5.23 can be re-written with more generalized form

$$C_i = \sqrt{\frac{g_i \cdot C_{i-1} \cdot (C_{w_i} + C_{i+1} + C_{b1_i} + C_{f1_i} + C_{b2_i} + C_{f2_i})}{g_{i-1} + \frac{R_{w_{i-1}} \cdot C_{i-1}}{\tau}}}$$
$$= \sqrt{C_{i-1} \cdot C_{i+1}} \times \sqrt{1 + \frac{C_{w_i}}{C_{i+1}} + \underbrace{\frac{C_{b1_i} + C_{f1_i} + C_{b2_i} + C_{f2_i}}{C_{i+1}}}_{\textbf{branches and fanouts}}} \times \sqrt{\frac{g_i}{g_{i-1} + \frac{R_{w_{i-1}} \cdot C_{i-1}}{\tau}}} \tag{5.33}$$

Using $(g_i \cdot \tau)/C_i = R_i$, optimum condition can be expressed as

$$\left(R_{i-1} + R_{w_{i-1}}\right) \cdot C_i = R_i \cdot \left(C_{w_i} + C_{i+1} \cdot \underbrace{\overbrace{C_{b1_{i+1}} + C_{f1_{i+1}}}^{C_{bf1}} + \overbrace{C_{b2_{i+1}} + C_{f2_{i+1}}}^{C_{bf2}}}_{\textbf{branches and fanouts}} \right) \tag{5.34}$$

The load on the side branches are represented by C_{bf1} and C_{bf2} . Note that side branch resistance R_{bf1} and R_{bf2} do not effect the Elmore delay calculation of the main path.

In summary, optimum expression in ULE can be expresses as,

$$C_{BF} = \sum_{1}^{n} Cb_n + \sum_{1}^{m} Cf_m \tag{5.35}$$

where n and m are the number of branch wires and fanout gates in a path stage, respectively.

The general ULE conditions are summarized below,

$$\left(g_i + \frac{R_{w_i} \cdot C_i}{\tau}\right) \cdot h_i = g_{i+1} \cdot \left(h_{i+1} + h_{w_{i+1}} + \frac{C_{BF_{i+1}}}{C_{i+1}}\right) \tag{5.36}$$

$$C_i = \sqrt{C_{i-1} \cdot C_{i+1}} \cdot \sqrt{1 + \frac{C_{w_i}}{C_{i+1}} + \frac{C_{BF_i}}{C_{i+1}}} \cdot \sqrt{\frac{g_i}{g_{i-1} + \frac{R_{w_{i-1}} \cdot C_{i-1}}{\tau}}} \tag{5.37}$$

$$\left(R_{i-1} + R_{w_{i-1}}\right) \cdot C_i = R_i \cdot (C_{w_i} + C_{i+1} + C_{BF_i}) \tag{5.38}$$

Without the branches and fan-outs these equations converges to equations 5.12 , 5.13 and 5.23 respectively. And eventually, if the wire capacitance is ignored then all these equations will converge to the original logical effort expressions.

5.2 Summary

Unified logical effort (ULE) is the first model to integrate logical effort and Elmore delay for wire. The model uses the π model to represent wires and Elmore delay equations to estimate the delay. Moreover, the wire segment is bundled with the logic that drive that wire and treated as a single stage unlike the LE model where only the logic is considered as a stage and wires are ignored. In the absence of wire, ULE equations converge to LE equation and in the presence of long wires the equations converges to repeater insertion equations. ULE model produces better gate sizing results in the presence of practical wire segments and 100 times faster than industrial standard tool such as Analog Optimizer.

CHAPTER 6

Path optimization using ULE

6.1 Background

In order to maintain the correct logic levels at each logic gates output and ensure timely signal propagation, designers usually use gate or transistor sizing techniques. In many application, performance directly depends on minimizing the critical path delay often at the expense of power consumption and increase in delay in the non-critical paths. In many other applications, the requirement is to balance the delay of multiple paths. For example, decoders and parallel data-paths like parallel adders, multipliers, etc. In these type of circuits, the computation is intended to be parallel and finish at the same time and the performance is bounded by the maximum propagation delay in any of the parallel paths. In the second type of application the goal of the designers would be to ensure the proper balance of the circuit on all branching path as well as minimizing the overall maximum delay that bounds the limit of performance. The work in this paper tries to address this problem as a whole rather than treating them independently.

Balancing logic paths and fan-out optimization is not trivial task. The task has two requirements – 1) Accurate delay estimation methodology that works for all kinds of logic network having practical interconnect wires. 2) A methodology to calculate the appropriate load distribution or branch effort efficiently.

Fortunately, requirement 1 is fulfilled by prior research work in the form of Unified Logical Effort (ULE) [38]. In recent years many extensions of Logical Effort (LE) have been proposed and among them ULE is the most accurate and versatile model that

overcomes many limitations of the original LE algorithm especially in the presence of practical arbitrary wire segments. It should be noted that, both LE and ULE do not address the load distribution or branch balancing problem with practical wire segments. Although the original ULE paper highlights an introduction to branching, it does **not** evaluate branching effort within the different fan-out paths, rather it shows how to calculate delay through a path if the branching effort is already given.

Branching is commonly not utilized in other methods, mainly because it is difficult to calculate efficiently. However, some methods have attempted to compute the LE within a circuit system utilizing a buffer chain that has multiple fan-out nodes [47], [48] . This algorithm called LEOPARD utilizes a fan-out optimization algorithm that finds the optimal number of buffers and their sizes in multiple fan-out. But, their work is more related to buffer chain optimization to drive different loads. More importantly, it does not compute the branching effort accurately.

It should be noted that, authors of [49] and [50] also discussed about the load distribution/branching problem in their paper on technology mapping. But they tried to solve the load distribution problem in the context of technology mapping rather than optimal timing. They used Logical Effort (LE) methods for the delay estimation and used the load distribution information to formulate better technology mapping. The difference between their work and this papers work is 1) this dissertation proposes an extension to the ULE method for delay estimation that works better for practical circuits with RC interconnects and 2) the load distribution problem is solved in the context of optimal gate sizing and optimal equal delay between the fan-out paths in the presence of RC interconnect.

In a prior work, the authors of this paper [51], presented a technique to calculate the capacitance distribution in multiple fan-out branches that will ensure equal propagation delay in each fan-out path. The technique described in this paper is a continuation of the prior work and proposes a technique to design for optimal critical

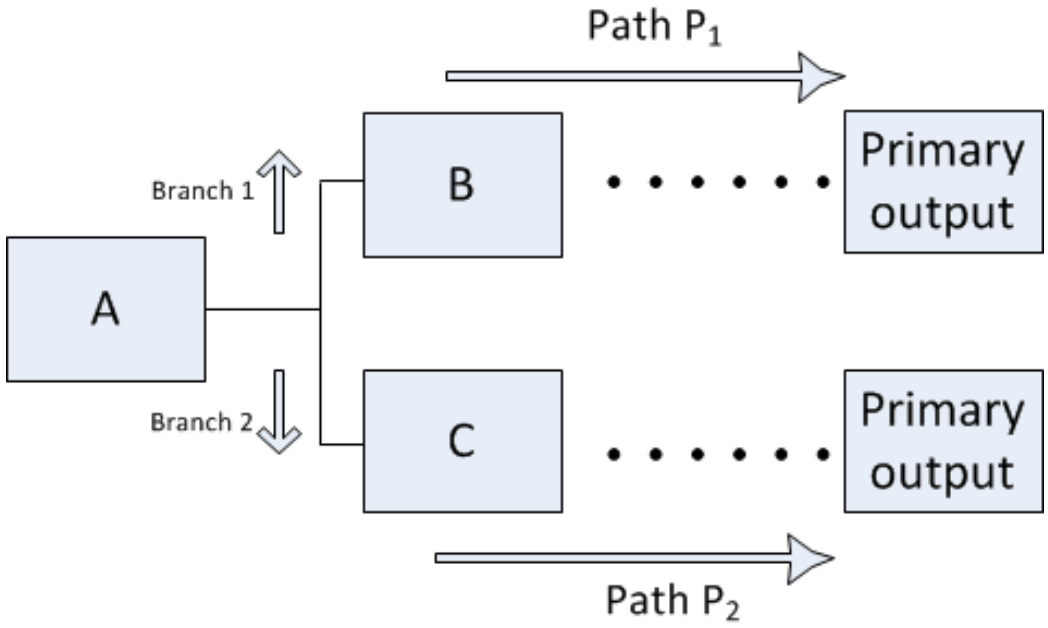


Figure 6.1: Load distribution model

path delay in addition to the prior branch path balancing technique.

6.2 Load distribution and path optimization

6.2.1 Load distribution problem

In order to understand the load distribution problem, first consider the case shown in Figure 6.1. A is some logic gate with its output driving two fan out paths of logic blocks B and C. Eventually, block B and C will be driving the primary outputs. In order to understand the load distribution problem one need to understand the interaction between A , B and C assuming they are fan-out free region.

The optimal delay through logic block A directly depends on the load being driven by A. That is, the load at output of A is nothing more than the input capacitance of the logic blocks B and C. If this load is increased then it will make the logic gate A slower and logic blocks B and C faster, and vice-verse.

In order for both the paths B and C to have the same delay the capacitance at the input of these blocks should be equal if the number gates in the path and logic

type of each gate in both the blocks are same. If the two path differs in number of logic gates or type of logic gates equal capacitance distribution will result in different propagation delay.

Given the optimum load at the output of logic block A, finding the right distribution of load to the input of the logic blocks B and C respectively will not only have equal delay through both the logic paths but also optimal delay through the entire circuit.

6.2.2 Path optimization problem

Path optimization consists of finding the optimal input capacitance of logic gates that makes the circuit run faster without violating certain constraint. Intuitively, it is the load distribution problem in multiple nested branch plus sizing the transistor for optimal delay for individual paths. In most practical cases, it is not necessary to solve for every branch. Rather, it is easier to solve for one critical path at a time and removing the path from the problem after solving.

In Figure 6.2. A is some logic gate with its output driving multiple fan out paths of purely combinational logic blocks. Eventually, the fan out paths will be driving the primary outputs. As mentioned in the previous section, In order to understand the load distribution problem one have to understand the interaction between A and all the fan-out paths assuming they are fan-out free region.

The optimal delay through the common path and logic block A directly depends on the load being driven by A. That is, the load at output of A is nothing more than the input capacitance of all the fan-out paths. For this work, this load or capacitance is termed as "Budget capacitance" which will used in rest of the paper. And gate A is termed as "Branch Driver".

If budget capacitance is increased then it will make the common path slower A slower but will drive the branches faster, and vice-verse. So, the delay in critical path

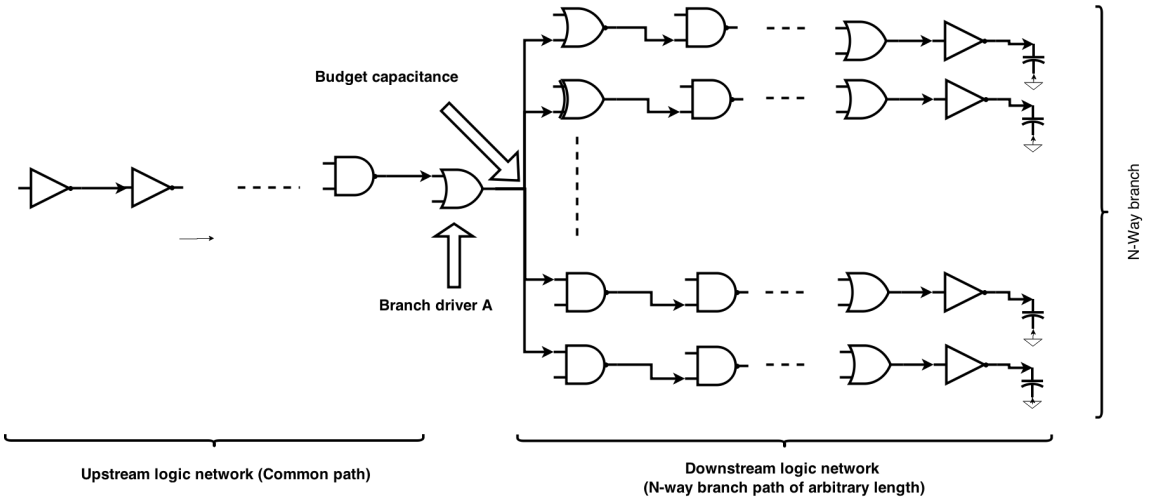


Figure 6.2: Practical logic network with arbitrary number of branch fan-out

can be minimized by making other paths slower and vice verse. So, it is evident that, a circuit as shown in figure 6.2 will have the least delay when all the branches are balanced.

In order for all the branch paths to have the same delay the capacitance at the input of these paths should be equal if the number gates in the path and logic type of each gate in both the blocks are same. If the the paths differs in number of logic gates or type of logic gates equal capacitance distribution will result in different propagation delay, which is the practical scenario in almost all the cases..

Given the optimum budget capacitance at the output of logic gate A, finding the right distribution of load to the input of the branch paths will not only have equal delay through the branch paths but also optimal delay through the entire circuit.

The authors of this paper proposed a balance algorithm 4 that distributes the already known budget capacitance to multiple fanout paths [51]. That is basically solving the 2nd part of the problem first. In this work, the author proposes a technique to whole problem together by finding the the optimal budget capacitance and distributing it to to balance the fan-out paths for overall optimal delay from source to any primary output loads.

6.2.3 Calculating branching effort for equal delay for fan-out of two path

For the proposed algorithm to efficiently compute the branching effort, the problem statement requires that there are two logic path (P1 and P2) having arbitrary number of logic gates, arbitrary wire segments between the gates and at the the logic paths are driving arbitrary output loads. The input capacitance (C_{in}) is given, which needs to be divided between the paths (P_1 and P_2) in such a way that the propagation delay D_1 and D_2 of path P_1 and P_2 respectively are equal and minimized. Algorithm 4 describes the methodology of this approach[1]

At first, the input capacitance is divided into n small pieces of capacitance, where n is defined as the precision index and integer multiple power of 2. The bigger the value of n, the smaller the unit of capacitance C_{unit} which is precisely $\frac{C_{in}}{n}$ or $C_{in} = n \cdot C_{unit}$. This C_{unit} is a significant expression and it expresses the smallest unit of capacitance that the algorithm can distinguish. In other words, parameters n and C_{unit} determines the precision.

At the beginning, half of input capacitance (which is precisely equal to $\frac{n}{2} \cdot C_{unit}$) is assigned to P_1 and the other half is assigned to P_2. That means the distribution factor is $(0.5, 0.5)$. The delay D_1 and D_2 is then measured and if D_1 is greater than D_2 than it is evident that path P_1 needs larger input capacitance to drive the path. Precisely, path P_1 should get capacitance distribution in the range of 0.51 to 1.00. Furthermore, the algorithm subsequently chooses the middle value of the desired range, $\frac{100+50}{2} = 0.75$ for P_1. And the remaining 0.25 is assigned to P_2. In the same way, if $D_1 < D_2$ a distribution of 0.25 and 0.75 is chosen, respectively, for path P_1 and P_2.

In the 2nd iteration, the delay D_1 and D_2 is measured again with the input capacitance calculated from the distribution in the first iteration. If still $D_1 > D_2$,

[1]Equation 5.23 is utilized to calculate all the capacitors in each chain and Equation **??** is utilized to calculate the delays on line 10 and 11 of Algorithm 1

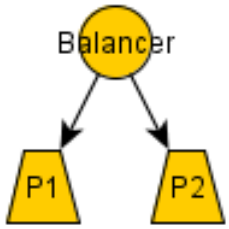


Figure 6.3: Single balancer with two logic path P_1 and P_2

algorithm chooses the upper half of the current range as the next range otherwise it chooses the lower half. This process continues and the selection range is halved after each iteration. As a result , the change in capacitance value is higher in initial phases. but the change becomes smaller with each iteration. The algorithm terminates when the difference between delay D_1 and D_2 becomes less than a threshold value. The algorithm reaches a $\frac{1}{2^t}$th precision in capacitance values after t iterations and the smallest range after t iterations will be $\frac{1}{2^t}.C_{in}$ or $\frac{1}{n}.C_{in}$, where n is the precision index that the algorithm started with.

The runtime of the algorithm is logarithmic in n or $O(log_2(n))$ where n = precision index = number of small unit capacitance (C_{unit}). Obviously, a smaller n will decrease the precision and result in logarithmic decrease in processing time and vice verse. The delays in each iterations are calculated using ULE method [38], which incorporates optimal gate capacitance in the presence of practical RC wire capacitance.

6.2.4 Calculation method for multiple branching paths

Algorithm 4 can be used recursively to solve larger problem consisting more than 2 paths. The original balance algorithm can balance any two arbitrary logic paths. In order to use the algorithm to solve larger problem with multiple path a hierarchical approach should be taken. The authors implemented an object-oriented software implementation with a balancer class and path class. A balancer has an input capacitance budget and deals with precisely two logic path. Therefore, each balancer's

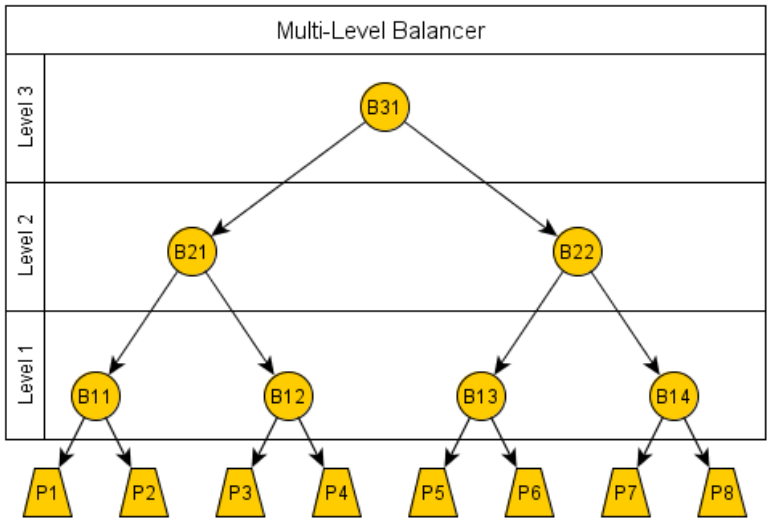


Figure 6.4: Hierarchical setup for solving branch with multiple path

goal is to distribute the budgeted input capacitance between the two paths in such a way that the propagation delays of the two paths becomes the same. In order to achieve its goal, the balancer chooses values of input capacitance C_1 , C_2 of path P_1, P_2 respectively. A balancer object uses algorithm 4 to assign capacitance to the paths in each step of algorithm and notes the average of the delays measured in two paths until the delay difference in the paths becomes lower than the threshold value. The threshold value is defined as a percentage of the average delay measured in each step (for example, 0.001 % of the average delay measured). It should be noted that, the number of iteration step in the algorithm is fixed for one runtime when the algorithm is called for one branch and the number of step is equal to $log_2(n)$ where n is the precision index. The complexity of the algorithm is O($log_2(n)$).

Figure 6.4 shows the hierarchical arrangement for 8 paths. At first, the paths are grouped into groups of 2 and one primary level balancer is assigned for each such group. In figure 6.4 P_1,P_2, ... P_8 are the logic paths and B_{11},B_{12},B_{13},B_{14} are the primary level (level 1) balancer. Second level balancers (B_{21}, B_{22}) distributes capacitance and equalize the group delay between the two primary level balancer.

Algorithm 4: Balance algorithm using ULE

input : Reference Path (P_1), Merging Path (P_2), Precision index (n) ,Input Cap (C_{In}) and Output Caps ($Cout_1$, $Cout_2$)

output: Capacitance distribution (C_1, C_2) that results in same propagation delay in P_1 and P_2

```
/* Number of small cap = precision index = n = 2^t , where t is any
positive integer.  So, n is integer multiple power of 2.            */
```

```
1
2  begin
3      StartPointer = 0
4      EndPointer = n
5      UnitCap = C_in/n
6      for 0 to log_2(n) do
7          Selector = (StartPointer + EndPointer)/ 2
8          C_1 = UnitCap * Selector
9          C_2 = C_In- C_1
10         D_1 = ULEDelay (P_1, C_1, Cout_1)
11         D_2 = ULEDelay (P_2, C_2, Cout_2)
12         if D_1 >D_2 then
13             StartPointer = Selector
14         end
15         else
16             EndPointer = Selector
17         end
18     end
19     return C_1, C_2
20 end
```

The third level balancer does the same thing with second level balancers and so on. The top most level always has only one balancer and it is the root.

In Figure 6.4, B_{31} is the root balancer and its capacitance budget is the entire input capacitance that needs to be distributed among the 8 paths. At the beginning, B_{31} will distribute half of the capacitance to B_{21} and B_{22} in the second level. Again, B_{21} and B_{22} will assign their respective assigned capacitance between the primary level balancers B_{11},B_{12},B_{13} and B_{14}.

In this hierarchical approach, balancer balances every time a new budget capacitance is assigned. So, each time the parent balancer changes capacitance distribution, the children balancer re-evaluates with the new budget just assigned from the parent. In worst case, if the parent runs k step , the children runs $k * k$ steps. So, the runtime for multiple paths solution depends on the height of the balancer tree. It is evident that the height of the tree is $log_2(m)$ where m is the number of paths. Runtime for a single balancer is $O(log_2(n))$ where n is the precision index. Runtime for the hierarchical arrangement is $O((log_2 n)^{log_2 m})$. For fan out of 8 paths , the worst case runtime is $O((log_2 n)^3)$ or cubic. It is one of the limitation of the algorithm that for higher fan-out branches the hierarchical arrangement slowly goes to exponential complexity. For example, the complexity for 2-way branch is $O(log_2(n))$, for 4-way branch the complexity is $O(log_2(n))^2$, for 8-way branch the complexity is $O(log_2(n))^3$ and so on. The algorithm 4 itself has logarithmic complexity but when the hierarchical arrangement is not logarithmic. But on the bright side, it calculates the input capacitances distribution correctly (upto 7 digit decimal precision). Also, branches with more than 8 fan-out is not practical and not good practice while designing complex time sensitive circuit. These kind of branches are usually split into smaller fan-out branches by inserting buffer. So, the algorithm described in this paper still provides a practical tool to size all the gates in presence of interconnect and balance them with great accuracy.

6.3 Summary

In this chapter , load distribution problem is introduced. Load distribution problem has 2 sub problem – 1) to determine the distribution of budget capacitance to the fan-out branches , 2) to determine the optimal budget capacitance for overall minimal delay. In this chapter, an efficient solution for the first sub problem is presented. Algorithm 4 solves the capacitance distribution problem in logarithmic time for fan-out of 2. Later, s hierarchical arrangement is also proposed in this chapter for fan-out of more than 2.

CHAPTER 7

Optimal budget capacitance

In Chapter 6 the term "budget capacitance" has been introduced. Basically the budget capacitance is defined as the total capacitance that must be distributed among the fan-out input capacitance and must be driven by the branch driver. In Figure 6.2 , the gate that drives the branch is denoted as the branch driver A and the capacitance driven by the branch driven is shown as the budget capacitance. Budget capacitance determines how much load the upstream network should drive and how much input capacitance should the downstream network have.

7.1 Calculation of optimal budget capacitance

7.1.1 Upper bound and lower bound of optimal budget capacitance

In section 6.2.3 and 6.2.4 , it is shown how to balance multiple branch fanout paths if the total budget capacitance at branch is already given. But, in general, the budget capacitance at branch is unknown because the branch might be driven by a upstream path just like the scenario shown in figure 6.2 . From figure 6.2 it is evident that optimal overall delay is the sum of common path delay (upstream) and the optimal balanced branch path delay (downstream). In the previous sections , it is discussed how to balance the downstream network with the balance algorithm which is detailed in [51]. But the overall optimal delay depends on the budget capacitance chosen at branch point. This budget capacitance determines how much load the common path should see and how much input capacitance the individual branch paths should get after balancing.

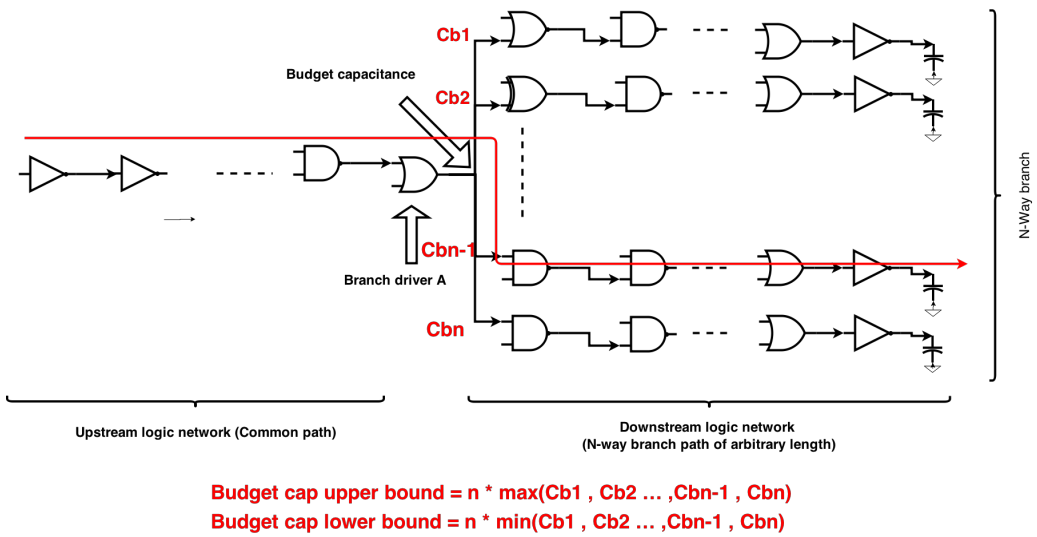


Figure 7.1: Bounds of 5 point algorithm

In order to choose the budget capacitance one must consider finding the practical upper bound and lower bound of the capacitance value. In order to do that , the author proposes to size all possible path from input to output ignoring the other branches. Figure 7.1 one such path is shown in red. In each iteration of sizing, the calculated capacitance at branch point C_{b_i} is recorded. So, for n number of branches there will be n such C_{b_i}. The next step is to determine the maximum and minimum value , $C_{b_{upper}}$ and $C_{b_{lower}}$ respectively. It is evident that, all the sized path have portion of the path common, ie, the part that starts at the source and ends at the branch point. And all the paths are different after the branch point. So it is evident that, the max value of C_{b_i} is found on the slowest path with worst loading and minimum value of C_{b_i} is found on the fastest path.

From this observation it can be concluded that the total budget capacitance at branch point can not be more than $n * C_{b_{slowest}}$ and hence it is the upper bound. With similar arguments it can be showed that, the lower bound is $n * C_{b_{fastest}}$. If the budget capacitance higher than $C_{b_{upper}}$ then the downstream paths will be faster and upstream path will be slower. And if the budget capacitance is lower than $C_{b_{lower}}$ it

Algorithm 5: 5 point algorithm for budget capacitance calculation

input : Common Path (P_c) that starts at source and ends at the branch, Branching paths $P_{b_1}, P_{b_2}, P_{b_3}.....P_{b_n}$, Input capacitance C_{in}, Output capacitance $C_{out_1}, C_{out_2}, C_{out_3}, ...C_{out_n}$, number of iteration t

output: Total optimum capacitance at branch point $C_{b_{opt}}$ and optimum individual capacitances $C_{b_1}, C_{b_2}, C_{b_3}.....C_{b_n}$ that results in same propagation delay through all branch from PI to PO, where $C_{b_{opt}} = C_{b_1} + C_{b_2} + C_{b_3} + ... + C_{b_n}$

begin

  /* First determine the upper limit $C_{b_{upper}}$ and lower limit $C_{b_{lower}}$ of branch capacitance C_b */

  for i <- *1* **to** n) **do**

    $P_i = P_c + P_{b_i}$

    /* Size P_i ignoring branch */

    $ULESize\ (P_i)$

    C_{b_i} <- Capacitance at branch point after sizing

  end

  n = fan out at branch point

  $C_{b_{upper}} = n * \max(C_{b_1}, C_{b_2}, C_{b_3}, ..., C_{b_n})$

  $C_{b_{lower}} = n * \min(C_{b_1}, C_{b_2}, C_{b_3}, ..., C_{b_n})$

  for *1* **to** t **do**

    $C_{b_H} = \frac{C_{b_{upper}} + C_{b_{lower}}}{2}$

    $C_{b_{Q1}} = \frac{C_{b_{lower}} + C_{b_H}}{2}$

    $C_{b_{Q2}} = \frac{C_{b_H} + C_{b_{upper}}}{2}$

    $D_{lower} = ULEDelay(C_{in}, C_{b_{lower}}) + Balance(C_{b_{lower}})$

    $D_{upper} = ULEDelay(C_{in}, C_{b_{upper}}) + Balance(C_{b_{upper}})$

    $D_H = ULEDelay(C_{in}, C_{b_H}) + Balance(C_{b_H})$

    $D_{Q1} = ULEDelay(C_{in}, C_{b_{Q1}}) + Balance(C_{b_{Q1}})$

    $D_{Q2} = ULEDelay(C_{in}, C_{b_{Q2}}) + Balance(C_{b_{Q2}})$

    if $D_{Q1} < D_{lower}$ ***and*** $D_{Q1} < D_H$ **then**

      $C_{b_{upper}} = C_{b_H}$

    end

    else if $D_{Q2} < D_H$ ***and*** $D_{Q2} < D_{upper}$ **then**

      $C_{b_{lower}} = C_{b_H}$

    end

    else

      $C_{b_{upper}} = C_{b_{Q1}}$

      $C_{b_{lower}} = C_{b_{Q2}}$

    end

  end

  $C_{b_{opt}} = \frac{C_{b_{upper}} + C_{b_{lower}}}{2}$

  $D_{opt} = ULEDelay(C_{in}, C_{b_{opt}}) + Balance(C_{b_{opt}})$

  return $C_{b_{opt}}$, D_{opt}

end

will not be good enough to drive the branch. The optimal solution is somewhere in between $C_{b_{upper}}$ and $C_{b_{lower}}$. Algorithm 5 has two major parts. The first part finds the upper bound and lower bound of the optimal budget capacitance.

7.1.2 5-point algorithm

The nature of the budget capacitance Vs total delay is shown in figure 7.2 . When the budget capacitance is low the downstream path delay is large and dominates the overall delay. As the budget capacitance is increased the downstream path delay start to decrease fast and upstream path delay starts ot increase slowly. As a result the overall delay is decreased. This behavior of overall delay continues upto the optimal minimum point and after that the overall delay start to increase again because of faster increase in upstream path delay compared to the downstream path delay. It is evident that, there is only one optimal minimum point for this curve and that point also produces the optimal overall delay.

In section 7.1.1 , it was shown how to determine the upper and lower bound on the budget capacitance of a branch. The bound values gives the search initial space for the optimal solution. The second major part of algorithm 5 describes how to reduce the search space in each step by calculating delays in 5 different points and comparing them. At first, delays are calculated at the upper bound $C_{b_{upper}}$ and lower bound $C_{b_{lower}}$ of the budget capacitance and they are D_{upper} and D_{lower} respectively.

The three more points are 1) half way point, 2) first quarter point and 3) third quarter point between the upper and lower bound, ie,

$$C_{b_H} = \frac{C_{b_{Lower}} + C_{b_{Upper}}}{2} \tag{7.1}$$

$$C_{b_{Q1}} = \frac{C_{b_{Lower}} + C_{b_H}}{2} \tag{7.2}$$

$$C_{b_{Q2}} = \frac{C_{b_H} + C_{b_{Upper}}}{2} \tag{7.3}$$

The corresponding delays for $C_{b_H}, C_{b_{Q1}}$ and $C_{b_{Q2}}$ are D_H, D_{Q1}, D_{Q2} respectively.

In the next steps, these 5 delays are compared to reduce the search space. From

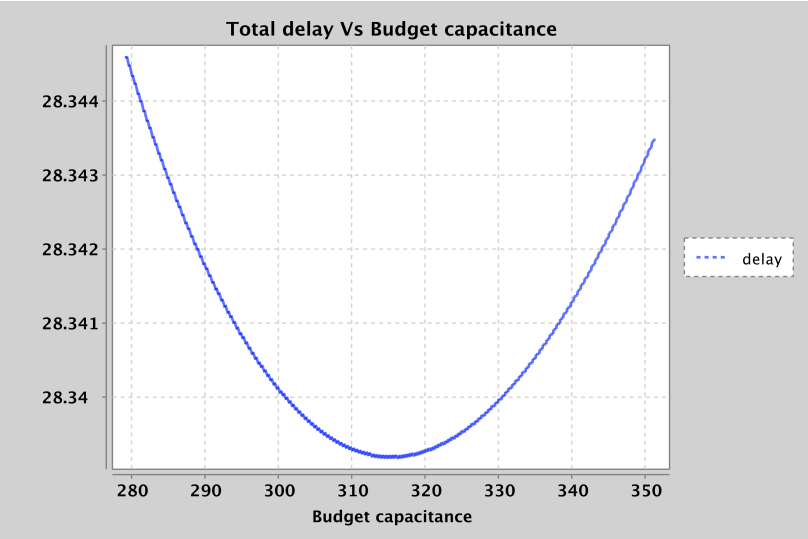


Figure 7.2: Delay vs budget capacitance

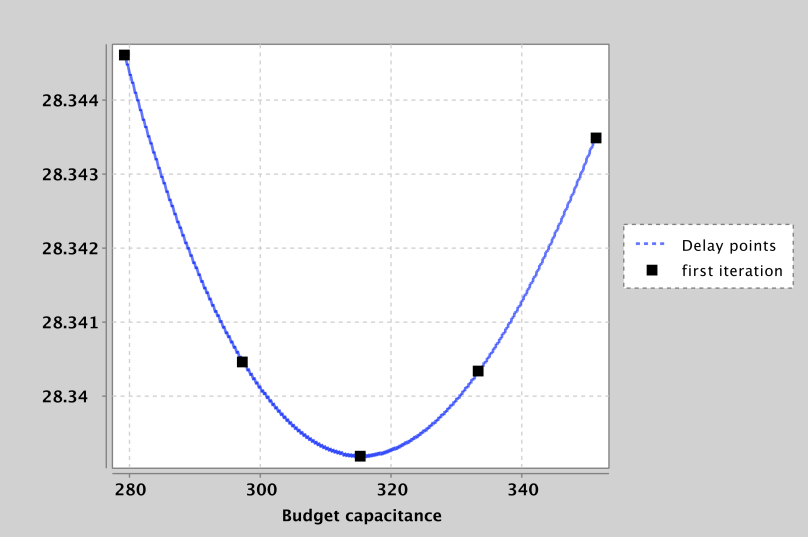


Figure 7.3: Case 1: optimal point lies between point 2, 4

the budget capacitance vs total delay curves (figure 7.2) if any three points C_1 , C_2 , C_3 are taken such that the delays C_1 , C_2 , C_3 corresponding to those point are either ever increasing or ever decreasing then the optimal capacitance C_{opt} can not reside between $C1$ and $C3$. In other words, optimal solution can exist in between $C1$ and $C3$ if and only if delay at the mid point of the range $D2$ is smaller than both $D1$ and $D3$.

There are three possible scenario to consider in order to reduce the search space

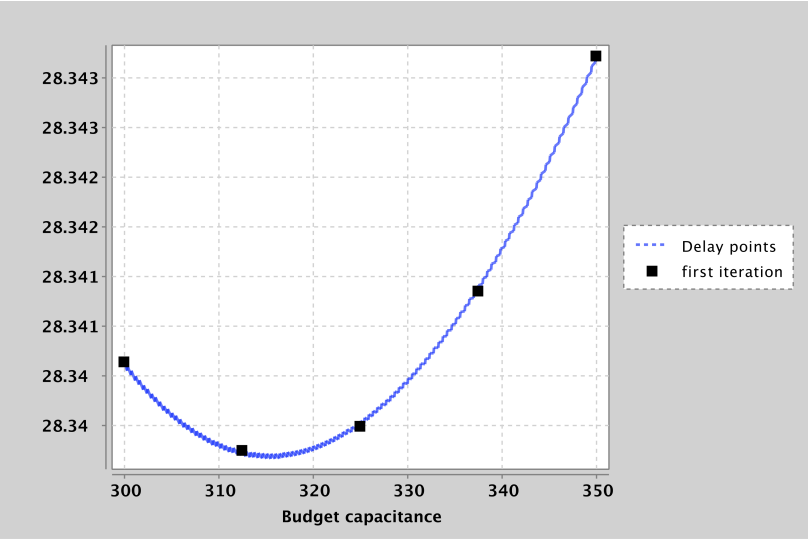


Figure 7.4: Case 2: optimal point lies between point 1, 3

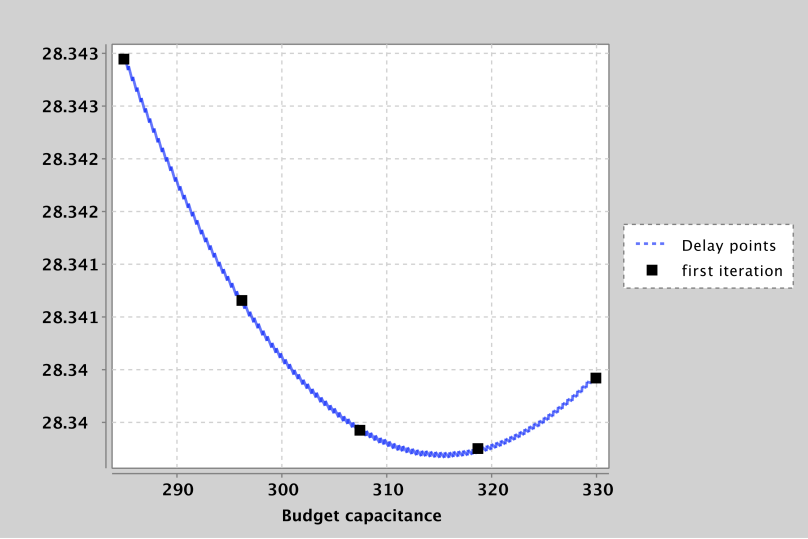


Figure 7.5: Case 3: optimal point lies between point 3, 5

and zeroing on the optimal delay point. Figure **??** shows the three possible cases. In the first case, the optimal point lies between $C_{b_{Q1}}$ and $C_{b_{Q2}}$ points (figure 7.3). In the 2nd case, the optimal point lies between $C_{b_{lower}}$ and C_{b_H} points (figure 7.4). And in the 3rd case, the optimal point lies between C_{b_H} and $C_{b_{upper}}$ points (figure 7.5).

By using this observation , the search space reduction decisions are made. If D_{Q1} is less then both D_{lower} and D_H then the new upper bound of budget capacitance is set to current C_{b_H} and algorithm goes to next iteration. If D_{Q3} is less then both D_H

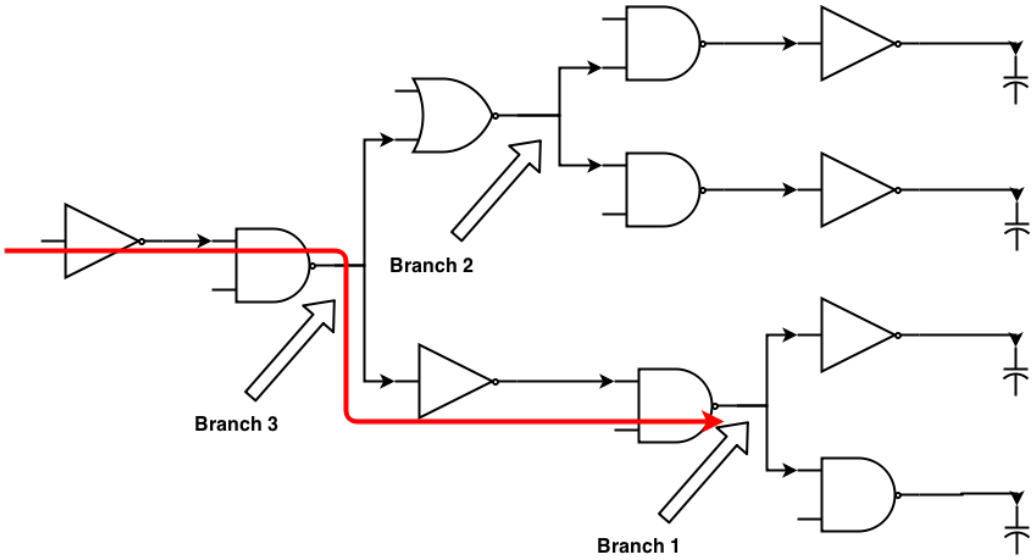


Figure 7.6: Heuristic for solving multi-level branches in practical circuit

and D_{upper} then the new lower bound of budget capacitance is set to current C_{b_H} and algorithm goes to next iteration. If however, neither of the above conditions are met, the optimal point must reside in the middle portion ie, between $C_{b_{Q1}}$ and $C_{b_{Q3}}$. In the 3rd case, C_{bQ1} is set as the new lower bound and $C_{b_{Q3}}$ is set as the new upper bound for the next iteration.

In each step of the algorithm, the search space is halved and the range between upper and lower bound gets narrower. In each iteration , the mid-range point delay is compared with that of the previous iteration. The algorithm stops when the difference between calculated values is lower than a predefined threshold.

7.1.3 Multiple branch points

In practical circuit, there are multiple branch points in general. Solving multiple branches together is complex problem. But fortunately, in most cases it is not necessary to to size and optimize all the paths in the circuit . A good estimated sizing of the critical path is important tough. Because branch calculations are usually dependent on each other and that is why it is difficult to calculate actual branching effort effi-

Algorithm 6: Heuristic for solving multi-level branches in practical circuit

input : Circuit with multi-level branches , Input caps (c_{in_i}) , Output loads (c_{load_i})

output: Branching efforts (b_i) , Gate input capacitance (c_i)

```
1 begin
2     foreach branch b_i of circuit in reverse topological order do
3         CommonPath = longest path from the source to the branch
4         c_budget = FivePointAlgorithm (CommonPath, Out Caps, b_i)
5         Remove b_i from the circuit
6         c_budget becomes the output load at branch point b_i
7     end
8     return b_i, c_i
9 end
```

ciently. In order to avoid complexity of inter dependency between branches, heuristic methods can be applied to reach near optimal solution to speed up computation.

The authors proposed a good heuristic that would take only the critical path and all of its fanout and optimize the critical path applying all the techniques described in the previous sections. In this approach, the circuit will be traversed in reverse topological order from primary outputs towards the primary inputs and budget capacitance will be assigned to branches. For example, in figure 7.6 there are three branches and they will be solved in the reverse topological order (Branch 1, 2, 3) .

Only one branch is processed at a time assuming the common path is the longest path from the source to the branch. At this moment, other fanout branches of the common path is ignored for simplicity. After assigning a budget capacitance , the downstream portion of the branch is removed from the problem and the assigned budget capacitance becomes the load capacitance for the remaining circuit. The

technique is listed in algorithm 6.

For example, in figure 7.6 branch 1 is processed first. The common path for this case is shown in red line starting from the source ending at branch 1 and in this iteration other branches (2 , 3) are ignored . Basically, the common path and branch 1 forms a fork of logic path which is easy to solve with the 5 point algorithm described in algorithm 5 .

7.2 Summary

In this chapter, several algorithms are documented and these algorithms are collectively provide a solution for the load distribution problem in logic branch in the presence of arbitrary wire and arbitrary number of logic gates on each fan-out paths. The balancing algorithm (algorithm 4) distributes a given capacitance to the fan-out branches and the 5-point algorithm (algorithm 5) finds the optimal budget capacitance to minimize the delay of upstream and downstream network of a branch. These two algorithm has logarithmic complexity. And the top level heuristic (algorithm 6) provides a systematic way to solve all the branches in a circuit for only once. This algorithm traverses branches in reverse topological order and uses balance algorithm and 5-point algorithm on the branches to minimize the overall delay. The heuristic has linear complexity.

CHAPTER 8

Experiments and results

In the previous chapters, background and theoretical works was described. And in chapter 5 Unified Logical Effort background was discussed. And chapter 6 contains all the proposed research work regarding transistor sizing in the presence of arbitrary interconnect and accurate branch effort calculation algorithms. In this chapter, all the experimental setup and results are presented. These experiments were conducted with help of Ok CAD tool collections which contains tools to accurately model and analyze industrial standard circuits . These tools are based on Graph based data structures and algorithms which are described in chapters 2 , 3 and 4 .

8.0.1 Experimental setups

Various tools from OkCAD tool collection were most for accurately model the industrial standard circuits. The circuits under tests were classified in two types

- Randomly generated logic chain and branches to tests the accuracy and runtime of algorithms that are used to accurately calculate branching effort and solve the load distribution problem in branches.
- Industry standard data-path circuits to analyze the impact of transistor sizing , accurate branching effort calculated by the higher level algorithms proposed in this dissertation.

For the second type of test circuits few extra automation steps has been followed to accurately generate higher level RTL code for different bit-size and different types

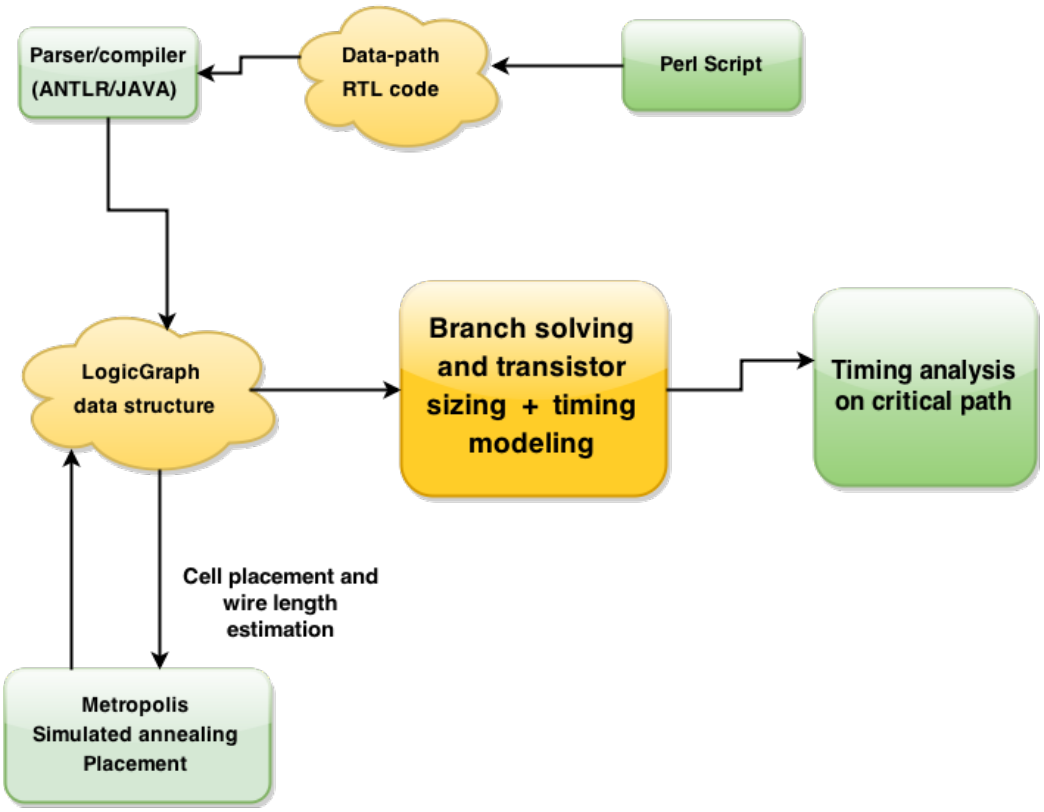


Figure 8.1: Tool/Algorithm flow for the experiments

of adder architectures. Verilog RTL code was generated using Perl scripts and the generated scripts was modeled into LogicGraph data structure using a Verilog subset parser (designed with ANTLR [4]) . For estimating practical wire lengths the gates were placed using Metropolis Simulated Annealing Placement algorithm [52], [53] . After placement, wire lengths were measured and included into the circuit model. Figure 8.1 shows the steps of modeling the circuits in software. Next, the Metropolis algorithm algorithm is briefly described.

8.0.2 Metropolis simulated annealing gate placement algorithm

In 1983 a modified version of Metropolis simulated annealing algorithm for gate placement was proposed by Kirkpatrick et. all [52] and used in IBM. Later further analysis was presented by Rutenbar [53] . This algorithm works very well for small to medium

level circuits of gate counts less than 50K and regarded as one of the best solution for gate placement even today. The maximum gate count for this work is around 3.5K and hence this algorithm is acceptable solution for this work.

The objective of the algorithm is to place gates while minimizing the total wire length. The gates are placed in a 2 dimensional grid where each cell contains one gate and the cell location are represented with X, Y coordinates. Multi-point wire's length is estimated using Half Perimeter Wire Length (HPWL) . The definition of HPWL is

$$HPWL = \Delta X - \Delta Y \tag{8.1}$$

where , ΔX = maximum X coordinates of all connected gates - minimum X coordinates of all connected gates and ΔY = maximum Y coordinates of all connected gates - minimum Y coordinates of all connected gates.

Algorithm 7 describes the placement algorithm using Metropolis Simulated Annealing technique. At the beginning, all the gates are randomly placed in the 2D space and each gate is assigned X,Y coordinates. Then total wire length of the current placement is calculated using HPWL. In the main optimization loop, random gates are swapped and changes in HPWL (ΔL) is calculated. If ΔL is negative ie. if the total wire length L is decreased , the current swap is accepted. Otherwise, the algorithm finds a inferior solution. In simulated annealing solution, some inferior solutions are randomly accepted in order to avoid getting stuck at local minima. If all the inferior solutions are rejected there is a high probably of greedy algorithm getting stuck in local minima. In simulated annealing, the acceptance of inferior solution depends on a temperature cost function. In the initial period of the simulated annealing the temperature parameter is high and as a result more inferior solutions are accepted randomly. In the course of time, the inferior solution acceptance rate decreases as the temperature parameter is decreased and more inferior solution gets

Algorithm 7: Metropolis simulated annealing placement algorithm

```
   input : Netlist with gate and wire connection information
   output: Near optimal cell placement that results in minimum combined wire length
 1 begin
       /* Start with random initial placement                                          */
 2     foreach gate G_i in netlist do
 3         Place G_i in unoccupied random location (x,y) in 2D grid
 4     end
 5     L = 0  /* total wire length                                                     */
       /* Start with random initial placement                                          */
 6     foreach net N_i in netlist do
 7         Place G_i in unoccupied random location (x,y) in 2D grid L = L + HPWL (N_i)
 8     end
 9     T = hot  /* Temperature for simulated annealing                                 */
10     Frozen = false
       /* Optimization loop                                                            */
11     do
           /* M = swaps per gate                                                       */
12         for (i = 0; i < M* Gate count ; i++ ) do
13             Swap random gates G_i and G_j
14             ΔL = Σ HPWL (net) after swap - ΣHPWL (net) before swap
15             if ΔL < 0 then
16                 accept this swap
17             end
18             else
19                 if uniform_random() < e^(-ΔL/T) then
20                     accept this inferior swap
21                 end
22                 else
23                     undo this inferior swap
24                 end
25             end
26         end
27         if ΣHPWL (net) is decreasing for the last few iteration then
               /* Cooling effect in simulated annealing.  Decrease in temperature T results in less acceptance of inferior
               results.                                                                */
28             T = 0.9*T
29         end
30         else
31             Frozen = true  /* get out of the while loop                           */
32         end
33     while Frozen ==false
34     return final placement as best solution
35 end
```

rejected. In case, of rejection the swapped gates returns to their original coordinates.

number of branch solved	runtime (2-way branch) [ms]	runtime (8-way branch [ms])
1	4	66
10	6	395
100	26	3,907
1,000	129	39,100
10,000	1,010	390,700

Table 8.1: Runtime

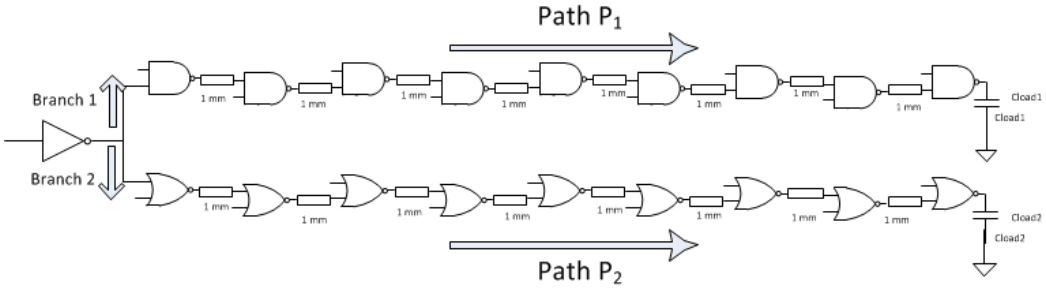


Figure 8.2: Test 1 : Sizing logic chain of equal length but different type

8.1 Results

All the experiment was carried out with a 65 nm CMOS process as used in the original ULE paper [38]. The process parameters are, Supply voltage 1.0 V, $R_0 = 8800\ \Omega$, $C_0 = 0.74$ fF. Intermediate wire $r_w = 1.0\ \Omega/\mu m$ and $c_w = 0.15$ fF$/\mu m$. Global wire $r_w = 0.04\ \Omega/\mu m$ and $c_w = 0.23$ fF$/\mu m$.

8.1.1 Results branching effort calculations

The algorithm described in this paper can equalize branches with at least 7 digit decimal precision. The precision can be increased at the expense of computation time. Another huge advantage is, when the algorithm completes, the network is already sized for optimal delay. Moreover, the algorithm works well with any kind of logic path consisting of arbitrary logic gates, RC wire segments with different

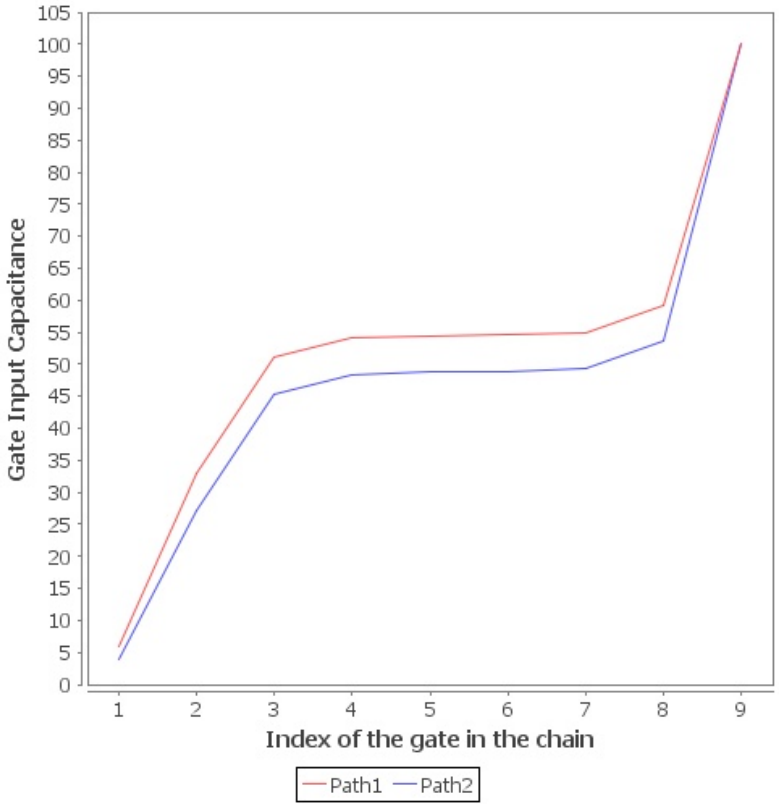


Figure 8.3: Sizing result of test 1

lengths and output loads. Several thousand random branching combinations and path combinations were chosen to test the results and the results were accurately verified. The algorithm was coded in Java and runs in just several milliseconds on a 3GHz iMac.

The algorithm was subsequently coded into a parser aided by ANTLR [**?**] and examples were coded in RTL Verilog to examine the impact branching has on their circuit implementation. Examples from ISCAS-85 benchmarks were used to examine the impact of the algorithm and to allow examination of the branching that goes on within these combinational circuits [**?**]. All benchmarks run efficiently within the code and branching was computed less than 1 minute for all implementations, which range from array multipliers to large decoders. It is important to note that the algorithm works for both equal and unequal paths on multiple-sized branch legs.

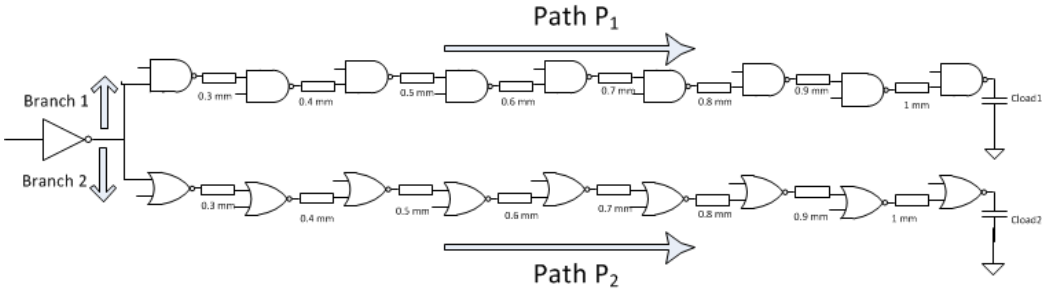


Figure 8.4: Test 2 : Branch consisting logic path of varying wire length

Figure 8.3 shows a specific example where two paths are balanced having the same number of gates and driving the same amount output load for a 65nm process. This example is for an optimal sizing result of two paths is for the circuit shown in Figure 8.2. The loading is $C_{Load1} = C_{Load1} = 100 \cdot C_0$ and and the budgeted input capacitance $C_{in} = 10 \cdot C_0$. Path 1, consists only 2-input nor gate and Path 2 contains only 2-input NAND gates. The same length wire is assumed between each gate (1 mm) for simplicity. From Figure 8.3 it is evident NOR gates are slower and harder to drive (remember: $g = 5/3_{NOR2}$ vs. $g = 4/3_{NAND2}$). However, both the paths produces the same kind of pattern which is a linear rise for the first few gates and then a flat region with a value around $50 \cdot C_0$ when the paths traverse through long wires. Interestingly, Figure 8.3 shows Path 2 getting a bigger reflection and pushing down the capacitance, as theory dictates. The length of wire segment in between each element, $L = 1$ mm and can be easily modified. The estimated delays after the algorithm completes is, $D_1 = 287.36055372931710$ and $D_2 = 287.36055347487420$.

Figure 8.4 shows a example of sizing two branch paths that have same number and same type of gates just like the previous example in figure 8.2 but have different length of wire in between the gates. And figure 8.3 shows the optimal sizing result after load distribution in the same 65nm process. Estimated delays after load distribution , D_1 = 114.23515777265783 and $D_2 = 114.23515773271560$ The circuit under consideration has nine 2-input NAND gates along Path 1 and nine 2-input NOR gates along path

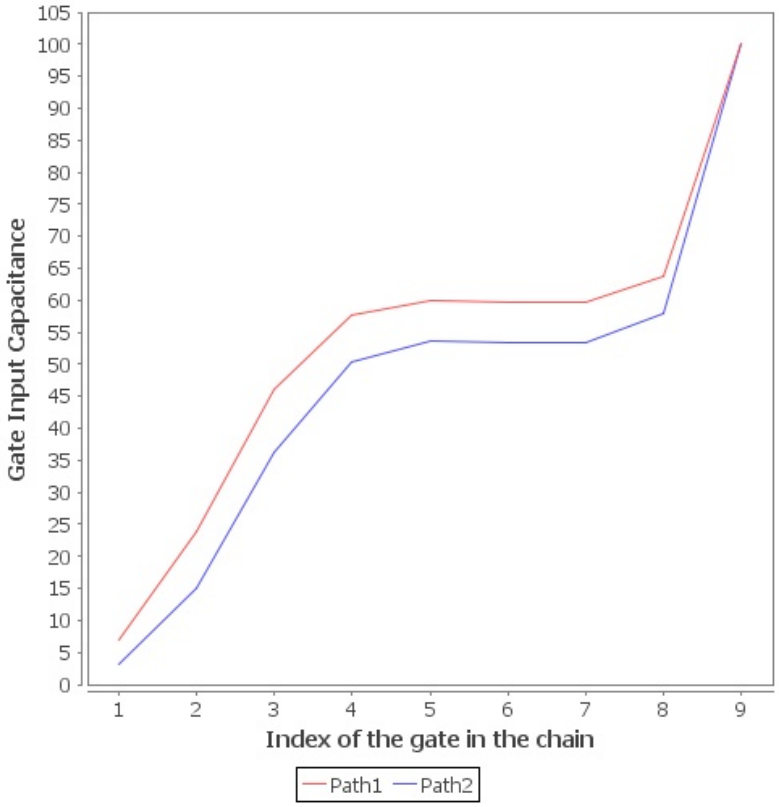


Figure 8.5: Sizing result of test 2

2. The wire length L starts with 0.3 mm and increases up to 1 mm near the load. Also $C_{Load1} = C_{Load1} = 100 \cdot C_0$ and and budgeted input capacitance $C_{in} = 10 \cdot C_0$.

The third example circuit in figure 8.6 consists of two logic branch path consisting of unequal number of gates. And figure 8.7 shows the optimal sizing result for this example for the same 65nm process that was used before. The estimated delays after load distribution , $D_1 = 255.28367622498365$ and $D_2 = 255.28367620247857$. The circuit under consideration have seven 2-input NAND gates along Path 1 and nine 2-input NAND gates along path 2. The wire length $L = 1$ mm between each element. Also $C_{Load1} = C_{Load1} = 100 \cdot C_0$ and and budgeted input capacitance $C_{in} = 10 \cdot C_0$.

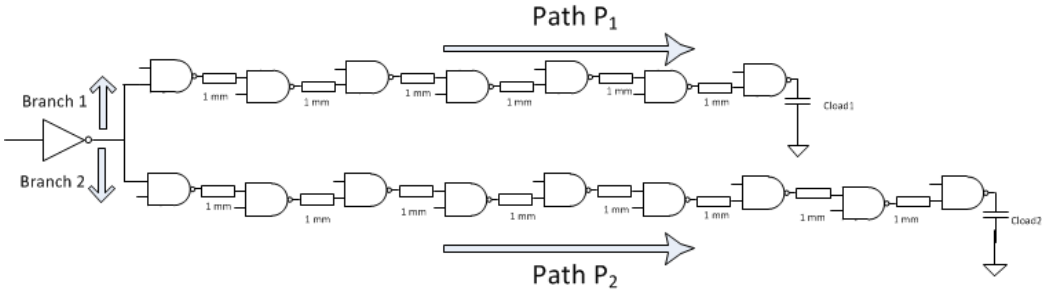


Figure 8.6: Test 3 : Branch consisting logic path of unequal number of gates

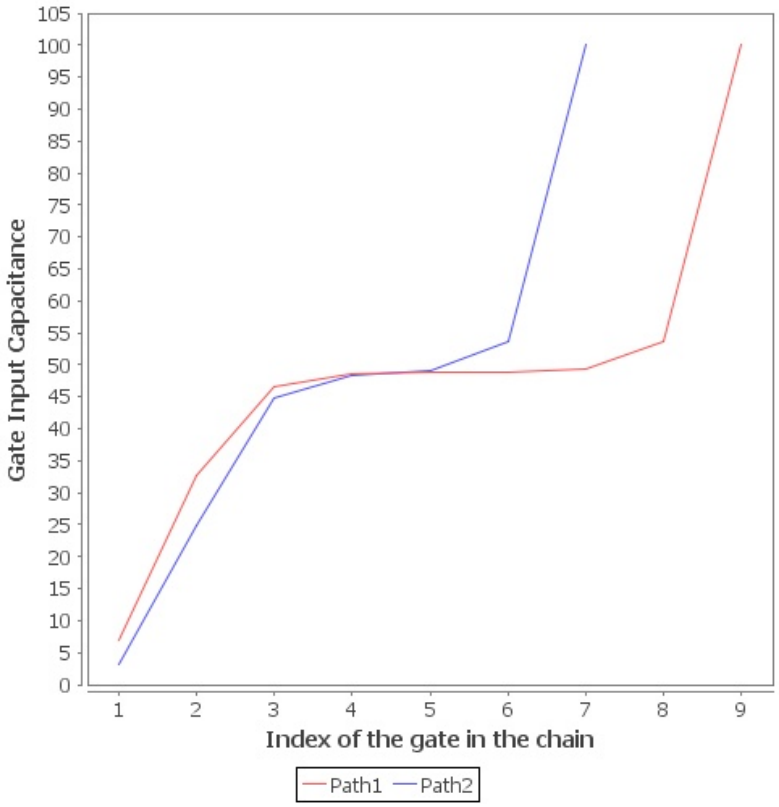


Figure 8.7: Sizing result of test 3

8.1.2 Data-paths under test

To analyze the performance and measure the quality of the results produced by the algorithms of this dissertation , high speed parallel prefix tree adders circuits were used. There are 3 basic architecture for parallel prefix tree adders – 1) Brent-Kung, 2) Skalansky and 3) Kogge-Stone as described by David Harris in his paper of parallel prefix network [54],[68], [69], [70]. Although , the three architecture process in parallel,

they differ in area, critical path length, maximum fan-out count and branching. The Brent-Kung [68] architecture has the smallest area but a longer critical path because of almost twice the logic level compared to the other two architecture. Skalansky architecture [69] on the other hand has slightly larger area but it has a shorter critical path. The drawback of this architecture is it has a large number of fan-out especially near the output logic levels. Kogee-Stone architecture [70] has the same critical path length but the maximum fan-out is limited to 2. But KS architecture usually have very complex branching and wiring requirements.

8.1.3 Results of delay optimization with accurate branching for large data-paths

In order to apply the ideas in this dissertation, industrial standard high-speed prefix adders are applied to the algorithm, as shown in Table 8.2. Table 8.2 shows the balancing and sizing result of parallel prefix tree adders of different bit sizes.

Each adder is generated in Verilog RTL from hand-made Perl scripts that can generate different prefix adders given specific input arguments. Then, the Verilog RTL text is converted to a DAG-like data structure (LogicGraph) where the vertices represent gates and edges represents wires. The conversion is done using ANTLR [4] based Verilog parser written in Java.

In order to estimate practical wire lengths it necessary to retrieve post placement wire length. For this work, a Metropolis Simulated Annealing Placement Algorithm is used to place the gates in a 2-dimensional grid and wire lengths are estimated from the grid distance. The wire length information is also saved on the same circuit graph model (DAG model) used to represent the circuit.

Then, the proposed algorithms were tested on the circuit graph model. Table 8.2 shows the balancing and sizing result of different parallel prefix tree adders of different bit widths for two different electrical efforts. The adder architectures analyzed and

	Kogge-Stone	Brent-Kung	Skalansky	Kogge-Stone	Brent-Kung	Skalansky
8 bit prefix adders		H = 16			H = 64	
delay in ps (LE only)	84.09	99.56	80.44	159.07	177.69	130.55
delay in ps (5-point + balance)	92.62	112.44	87.11	174.09	202.23	151.57
delay in ps (LE + RC)	112.02	130.28	108.37	186.99	208.41	158.47
gate count	70	52	55	70	52	55
branches solved	8	5	5	8	5	5
gates on critical path	10	11	10	10	11	10
algorithm runtime (ms)	351	451	434	331	538	481
16 bit prefix adders		H = 16			H = 64	
delay in ps (LE only)	121.37	139.48	98.72	190.97	232.39	161.15
delay in ps (5-point + balance)	127.24	156.07	120.90	216.41	267.77	184.56
delay in ps (LE + RC)	157.88	181.37	132.23	224.48	274.28	194.66
gate count	182	113	131	182	113	131
branches solved	14	9	9	14	9	9
gates on critical path	12	15	12	12	15	12
algorithm runtime (ms)	949	5,269	4,352	868	6,079	3,385
32 bit prefix adders		H = 16			H = 64	
delay in ps (LE only)	131.27	175.11	136.23	215.21	286.47	180.57
delay in ps (5-point + balance)	146.45	199.67	166.65	243.73	331.72	209.90
delay in ps (LE + RC)	168.37	228.18	175.33	254.31	339.53	219.67
gate count	454	238	307	454	238	307
branches solved	24	17	17	24	17	17
gates on critical path	14	19	14	14	19	14
algorithm runtime (ms)	4,836	11,010	8,867	7,093	9,521	8,885
64 bit prefix adders		H = 16			H = 64	
delay in ps (LE only)	157.20	217.61	175.28	266.53	355.39	221.87
delay in ps (5-point + balance)	179.79	249.90	211.53	300.46	407.41	248.24
delay in ps (LE + RC)	201.88	281.84	219.96	311.21	419.62	266.55
gate count	1,094	492	707	1,094	492	707
branches solved	42	33	33	42	33	33
gates on critical path	16	23	16	16	23	16
algorithm runtime (ms)	15,200	23,358	18,420	18,143	40,704	20,007
128 bit prefix adders		H = 16			H = 64	
delay in ps (LE only)	186.12	259.83	246.91	326.16	424.14	249.14
delay in ps (5-point + balance)	212.42	299.97	282.40	363.16	481.54	287.98
delay in ps (LE + RC)	236.39	335.24	297.18	376.43	499.55	299.41
gate count	2,566	1,000	1,603	2,566	1,000	1,603
branches solved	76	65	65	76	65	65
gates on critical path	18	27	18	18	27	18
algorithm runtime (ms)	37,485	58,243	37,349	43,458	44,709	37,600

Table 8.2: Algorithm simulation results for high-performance adders.

compared in Table 8.2 are Kogge-Stone, Brent-Kung and Skalansky architectures. Brent-Kung adders are very area-efficient, but its critical path contains double logic levels compared to other architectures. The Skalansky architecture has the same logic level as the Kogge-Stone architecture, but its branch fanout is the worst. Finally, the Kogge-Stone architecture has minimal logic level and a max fanout of 2 , but the wiring is complex. It should be noted that, Kogge-Stone adders are practically the best adder architecture for operating on over 64 bit numbers. However, this adder architecture usually has complex wiring and branching. So, for an efficient design it is important to size all the gates considering practical wire lengths and accurate branching effort.

From the analysis result shown in Table 8.2, each adder architecture was sized using LE and the RC wire delay is calculated from the wire length using a π model and technology data (wire capacitance per unit length). Using the traditional method, gates are sized separately using LE and the RC wire delay is added to the gate delay to get the total delay. Sizing is also shown for LE only (i.e., without RC delay) to illustrate that LE is not accurate with wire delays. For comparison, each adder architecture was balanced and sized using the 5-point algorithm (algorithm 5) according to the heuristic described in this paper and delays on critical path were calculated. In Table 8.2 , these delays obtained through these two methods (LE + RC and ULE) are compared.

From the result, it is evident that, sizing only with Logical Effort (LE) always overestimates the delay. The authors of ULE paper also reported the same trend during their result comparison [1] . In the traditional approach, gates were sized using LE and wire RC delay was estimated separately. Obviously, this results in more delay, because when the gates are sized with LE techniques the wires are not considered. On the other hand, the 5-point algorithm and the heuristic approach uses ULE as a helping tool and produces better sizing result and delay is minimized in

all cases. The results also confirms the practical trends – 1) Brent-Kung adders have more delay (due to logic level) , 2) Skalansky adders perform really good when the size of the operand is small and 3) Kogge-Stone adders are best for large operands.

8.2 Summary

In this chapter, the author proposed a novel technique to solve the load distribution problem in logic fanout branch for overall optimal delay from the primary input to the primary outputs. The 5-point algorithm proposed in this paper has logarithmic complexity and it determines the optimal budget capacitance very efficiently. Along with that, a heuristic algorithm is also proposed to use the 5-point algorithm to solve for multiple branches in reverse topological order. The algorithms were tested on high-speed parallel prefix tree adders and results were compared with the traditional LE based method. The results confirms the fact that LE based method often over estimates the delay. ULE based method in conjunction with balance algorithm [51] and 5-point algorithm achieves better delays through critical path.

CHAPTER 9

Conclusion

This research work conducted in the area of VLSI CAD and VLSI circuit optimization. The specific goal of this work was to develop better transistor sizing and accurate branch effort calculation techniques and using all these techniques build improved model to analyze and optimize high-speed parallel data-paths such as prefix adders. At the end of this work, most of the objective is met. Also, this work opened the door for many future improvement ideas.

9.0.1 Research accomplished

A Algorithms

A few novel algorithms were proposed as part of this dissertation. The algorithms are summarized below

Algorithm	Runtime complexity	Description
Branch Balance Algorithm	O(log(n))	Solves branches with arbitrary number of gates and arbitrary wire capacitance.
5-point Algorithm	O(log(n))	Solves complex fork branch and solves load distribution problem from branch.
Top level heuristic	O(n)	Solves multi-level branches in reverse topological order.

Table 9.1: Summary of algorithms

B Timing model

One of the objective of the work was to develop a better timing model to analyze high-speed parallel data-paths. Using the LogicGraph package in OK CAD tools and the sizing algorithms described above , a better timing model is proposed in this

research work. The proposed model has the following benefits over the model that is traditionally used

- The proposed model incorporates wire capacitance impacts
- The proposed model Incorporates accurate branching efforts and its impact
- Delays are more accurately measured in proposed model using ULE.

C Tools

Several robust software tools were developed for this work which are included in the OK CAD tools collections. The main tools were developed in Java programming language. The important tools are listed below,

- Compiler (EBNF Grammar, ANTLR parser)
- Logic Graph data structure collection
- General graph traversal algorithms.
- Transistor sizing (LE and ULE based)
- Load balancing (LE and ULE)

9.0.2 Future work

The proposed work is based on preliminary algorithms to size transistors, calculate branching effort. The current software implementation is easy to scale and improve. The current algorithms does not incorporate input ramping and velocity saturation effects in the delay estimation techniques and can be included in the tools in future. Another improvement will be a detail non-linear delay model for estimating node to node delay in the graphs. The current timing model works well for parallel prefix adders and in future it can be extended for other kind of data-paths and even for general circuits.

REFERENCES

[1] Arkadiy Morgenshtein, Eby G. Friedman, Ran Ginosar, and Avinoam Kolodny. Unified logical effort- a method for delay evaluation and minimization in logic paths with rc interconnect, 2007.

[2] Geert Janssen. A consumer report on bdd packages. In *Proceedings of the 16th Annual Symposium on Integrated Circuits and Systems Design, SBCCI 2003, Sao Paulo, Brazil, September 8-11, 2003*. IEEE Computer Society, 2003.

[3] Carver Mead and Lynn Conway. *Introduction to VLSI systems*, volume 802. Addison-Wesley Reading, MA, 1980.

[4] Terence Parr. ANTLR: ANother Tool for Language Recognition. `http://www.antlr.org/`, 1989.

[5] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[6] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6): 509–516, 1978.

[7] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[8] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *DAC*, pages 40–45, 1990.

[9] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[10] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[11] Adnan Aziz, Serdar Tasiran, and Robert K. Brayton. Bdd variable ordering for interacting finite state machines. In *DAC*, pages 283–288, 1994.

[12] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Mince: A static global variable-ordering heuristic for sat search and bdd manipulation. *J. UCS*, 10(12): 1562–1596, 2004.

[13] Yuan Lu, Jawahar Jain, Edmund M. Clarke, and Masahiro Fujita. Efficient variable ordering using abdd based sampling. In *DAC*, pages 687–692, 2000.

[14] Steven J. Friedman and Kenneth J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Computers*, 39(5):710–713, 1990.

[15] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, pages 42–47, 1993.

[16] Frederic Mailhot and Giovanni De Micheli. Algorithms for technology mapping based on binary decision diagrams and on boolean operations. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 12(5):599–620, 1993.

[17] Frédéric Mailhot. *Technology Mapping for VLSI Circuits Exploiting Boolean Properties and Operations*. PhD thesis, to the Department of Electrical Engineering.Stanford University, 1991.

[18] Eric Lehman, Yosinatori Watanabe, Joel Grodstein, and Heather Harkness. Logic decomposition during technology mapping. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(8):813–834, 1997.

[19] Rolf Drechsler, Nicole Drechsler, and Wolfgang Günther. Fast exact minimization of bdds. In *DAC*, pages 200–205, 1998.

[20] Christoph Scholl, Dirk Möller, Paul Molitor, and Rolf Drechsler. Bdd minimization using symmetries. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(2):81–100, 1999.

[21] Shipra Panda and Fabio Somenzi. Who are the variables in your neighborhood. In *ICCAD*, pages 74–77, 1995.

[22] Shipra Panda, Fabio Somenzi, and Bernard Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *ICCAD*, pages 628–631, 1994.

[23] Yoann Pigné, Antoine Dutot, Frédéric Guinand, and Damien Olivier. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. *CoRR*, abs/0803.2093, 2008.

[24] Yoann Pigné, Antoine Dutot, Frédéric Guinand, and Damien Olivier. GraphStream – A Dynamic Graph Library. `http://graphstream-project.org/`, 2008.

[25] Fabio Somenzi. CUDD: CU Decision Diagram Package. `http://vlsi.colorado.edu/ fabio/CUDD/`, 1995.

[26] David Long. CMU BDD. `http://www.cs.cmu.edu/ modelcheck/bdd.html/`, 1993.

[27] Kurt Keutzer. Dagon: technology binding and local optimization by dag matching. In *Papers on Twenty-five years of electronic design automation*, pages 617–624. ACM, 1988.

[28] Richard L. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, EECS Department, University of California, Berkeley, 1989. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/1223.html`.

[29] Yuji Kukimoto, Robert K Brayton, and Prashant Sawkar. Delay-optimal technology mapping by dag covering. In *Design Automation Conference, 1998. Proceedings*, pages 348–351. IEEE, 1998.

[30] Frédéric Mailhot and Giovanni De Micheli. Technology mapping using boolean matching and don't care sets. In *Design Automation Conference, 1990. EDAC. Proceedings of the European*, pages 212–216. IEEE, 1990.

[31] Victor N Kravets and Karem A Sakallah. Constructive library-aware synthesis using symmetries. In *Proceedings of the conference on Design, automation and test in Europe*, pages 208–215. ACM, 2000.

[32] Alan Mishchenko, Xinning Wang, and Timothy Kam. A new-enhanced constructive decomposition and mapping algorithm. In *Design Automation Conference, 2003. Proceedings*, pages 143–148. IEEE, 2003.

[33] Jason Cong and Yuzheng Ding. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(1): 1–12, 1994.

[34] Jason Cong and Yuzheng Ding. On area/depth trade-off in lut-based fpga technology mapping. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(2):137–148, 1994.

[35] Peichen Pan and Chih-Chang Lin. A new retiming-based technology mapping algorithm for lut-based fpgas. In *Proceedings of the 1998 ACM/SIGDA sixth*

international symposium on Field programmable gate arrays, pages 35–42. ACM, 1998.

[36] Gang Chen and Jason Cong. Simultaneous logic decomposition with technology mapping in fpga designs. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 48–55. ACM, 2001.

[37] I. E. Sutherland and B. F. Sproull. Logical effort: Designing for speed on the back of an envelope. *Advanced Research in VLSI*, 1991.

[38] Arkadiy Morgenshtein, Eby G. Friedman, Ran Ginosar, and Avinoam Kolodny. Unified logical effort - a method for delay evaluation and minimization in logic paths with rc interconnect. *IEEE Trans. VLSI Syst.*, 18(5):689–696, 2010.

[39] W.C.Elmore. The transient response of damped linear networks with particular regard to wideband amplifiers. *J Appl Phys*, 19(1):55–63, 1948.

[40] Kumar Venkat. Generalized delay optimization of resistive interconnections through an extension of logical effort. In *ISCAS*, pages 2106–2109, 1993.

[41] Michael Moreinis, Arkadiy Morgenshtein, Israel A. Wagner, and Avinoam Kolodny. Logic gates as repeaters (lgr) for area-efficient timing optimization. *IEEE Trans. VLSI Syst.*, 14(11):1276–1281, 2006.

[42] Aiqun Cao, Ruibing Lu, and Cheng-Kok Koh. Post-layout logic duplication for synthesis of domino circuits with complex gates. In *ASP-DAC*, pages 260–265, 2005.

[43] Srinivasa R Vemuru and Arthur R Thorbjornsen. Variable-taper cmos buffers. *Solid-State Circuits, IEEE Journal of*, 26(9):1265–1269, 1991.

[44] Michael Moreinis, Arkadiy Morgenshtein, Israel A. Wagner, and Avinoam Kolodny. Logic gates as repeaters (lgr) for area-efficient timing optimization. *IEEE Trans. VLSI Syst.*, 14(11):1276–1281, 2006.

[45] H. B. Bakoglu and J.D. Meindl. Optimal interconnection circuits for vlsi. *Electron Devices, IEEE Transactions on*, 32(5):903–909, 1985. ISSN 0018-9383. doi: 10.1109/T-ED.1985.22046.

[46] H Bakoglu. Circuit, interconnections, and packaging for vlsi, addision-wesley. *Reading, Mass*, 1990.

[47] Peyman Rezvani, Amir H. Ajami, Massoud Pedram, and Hamid Savoj. Leopard: a logical effort-based fanout optimizer for area and delay. In *ICCAD*, pages 516–519, 1999.

[48] Peyman Rezvani and Massoud Pedram. A fanout optimization algorithm based on the effort delay model. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(12):1671–1678, 2003.

[49] Shrirang K. Karandikar and Sachin S. Sapatnekar. Logical effort based technology mapping. In *ICCAD*, pages 419–422, 2004.

[50] Shrirang K. Karandikar and Sachin S. Sapatnekar. Technology mapping using logical effort for solving the load-distribution problem. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(1):45–58, 2008.

[51] Mehedi Sarwar and James E Stine. Enhancing the unified logical effort algorithm for branching and load distribution. In *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, pages 173–176. IEEE, 2014.

[52] Scott Kirkpatrick, MP Vecchi, et al. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

[53] Rob A Rutenbar. Simulated annealing algorithms: An overview. *Circuits and Devices Magazine, IEEE*, 5(1):19–26, 1989.

[54] David Harris. A taxonomy of parallel prefix networks. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*, volume 2, pages 2213–2217. IEEE, 2003.

[55] C. J. Hawthorn, K. P. Weber, and R. E. Scholten. Littrow configuration tunable external cavity diode laser with fixed direction output beam. *Review of Scientific Instruments*, 72(12):4477–4479, December 2001. URL `http://link.aip.org/link/?RSI/72/4477/1`.

[56] A. S. Arnold, J. S. Wilson, and M. G. Boshier. A simple extended-cavity diode laser. *Review of Scientific Instruments*, 69(3):1236–1239, March 1998. URL `http://link.aip.org/link/?RSI/69/1236/1`.

[57] Carl E. Wieman and Leo Hollberg. Using diode lasers for atomic physics. *Review of Scientific Instruments*, 62(1):1–20, January 1991. URL `http://link.aip.org/link/?RSI/62/1/1`.

[58] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *DAC*, pages 532–535, 2006.

[59] Satrajit Chatterjee. *On Algorithms for Technology Mapping*. PhD thesis, UNIVERSITY OF CALIFORNIA, 2007.

[60] Mark Alan Horowitz. *Timing models for MOS circuits*. PhD thesis, Stanford University, 1983.

[61] Jrn Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. `http://buddy.sourceforge.net/`, 1996.

[62] Arkadiy Morgenshtein, Eby G Friedman, Ran Ginosar, and Avinoam Kolodny. Timing optimization in logic with interconnect. In *Proceedings of the 2008 international workshop on System level interconnect prediction*, pages 19–26. ACM, 2008.

[63] Jorge Rubinstein, Paul Penfield Jr., and Mark A. Horowitz. Signal delay in rc tree networks. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2(3): 202–211, 1983.

[64] Rohini Gupta, Bogdan Tutuianu, and Lawrence T Pileggi. The elmore delay as a bound for rc trees with generalized input signals. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(1):95–104, 1997.

[65] Kenneth D Boese, Andrew B Kahng, Bernard A McCoy, and Gabriel Robins. Fidelity and near-optimality of elmore-based routing constructions. In *Computer Design: VLSI in Computers and Processors, 1993. ICCD'93. Proceedings., 1993 IEEE International Conference on*, pages 81–84. IEEE, 1993.

[66] Jason Cong and Lei He. Optimal wiresizing for interconnects with multiple sources. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1(4):478–511, 1996.

[67] L. Chan, B. Geuskens, R. Mangaser, and K. Rose. Beol yield predictions for sia roadmap. In *Advanced Semiconductor Manufacturing Conference and Workshop, 1997. IEEE/SEMI*, pages 87–90, 1997. doi: 10.1109/ASMC.1997.630712.

[68] Richard P Brent and H-T_ Kung. A regular layout for parallel adders. 1979.

[69] J Sklansky. Conditional-sum addition logic. *Electronic Computers, IRE Transactions on*, (2):226–231, 1960.

[70] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on*, 100(8):786–793, 1973.

VITA

MEHEDI SARWAR

Candidate for the Degree of

Doctor of Philosophy

Dissertation: **EFFICIENT SCHEMES TO SIZE TRANSISTORS FOR OPTIMAL DELAY BY SOLVING FANOUT BRANCHES WITH BALANCING ALGORITHM**

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Dhaka, Bangladesh on July 4, 1981.

Education:
Received the B.S. degree from Bangladesh Universoty of Engineering and Technology (BUET),
Dhaka, Bangladesh, 2005, in Electrical and Electronics Engineering
Received the M.S. degree from University of Texas at Dallas(UTD),
Richardson, TX, USA, 2002, in Electrical Engineering
Completed the requirements for the degree of Doctor of Philosophy with a major in Electrical Engineering Oklahoma State University in May, 2015.

Experience:
Electrical Engineering EDA Co-op, May 2011-December 2011,
IBM Corporation , Essex Junction, VT, USA
SOC Design Intern, May 2012-August 2012,
Lattice Semiconductor Corporation , San Jose, CA, USA
CCDO Graduate Intern (PhD level), May 2014-August 2014,
Intel Corporation , Hillsboro, OR, USA