LOGIC SYNTHESIS FOR FIELD-PROGRAMMABLE GATE ARRAYS

by

Rajeev Murgai

Memorandum No. UCB/ERL M93/98

15 December 1993

LOGIC SYNTHESIS FOR FIELD-PROGRAMMABLE GATE ARRAYS

by

Rajeev Murgai

Memorandum No. UCB/ERL M93/98

15 December 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Logic Synthesis for Field-Programmable Gate Arrays

by

Rajeev Murgai

Doctor of Philosophy in
Electrical Engineering and Computer Sciences
University of California at Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

Short turnaround time has become critical in the design of electronic systems. Software-programmable components such as microprocessors and digital signal processors have been used extensively in these systems, since they provide for quick design revisions. However, the inherent performance limitations of software-programmable systems make them inadequate for high-performance designs. As a result, designers have turned to a mask-programmable hardware solution, namely gate arrays. Recently, user-programmable gate arrays, called field-programmable gate arrays (FPGAs), have emerged and are changing the way electronic systems are designed and implemented. FPGA chips are prefabricated as arrays of identical programmable logic blocks with routing resources, and are configured by the user into the desired circuit functionality. The most popular FPGA architectures use either a look-up table (LUT) or a multiplexor-configuration as the basic building block.

With the growing complexity of the logic circuits that can be packed on an FPGA chip, it becomes important to have automatic synthesis tools that implement logic functions on these architectures. Conventional synthesis approaches fail to produce satisfactory solutions for FPGAs, since the constraints imposed by FPGA architectures are quite different. In this thesis, we explore the problem of logic synthesis for both LUT- and multiplexor-based architectures. The thesis is divided into two parts corresponding to the two classes of architectures.

In the first part, we propose algorithms to synthesize combinational logic with a minimum number of m-input LUTs, where each m-input LUT can realize any Boolean function of up to m inputs. We use the widely-accepted two-phase paradigm for logic synthesis consisting of technology-independent optimization followed by technology mapping. Technology-independent

optimization derives a minimal representation (with respect to a cost function), which is then implemented by the mapping phase on the target technology, in our case LUTs. We present LUT-specific mapping techniques for implementing a function that has more than m inputs and for combining functions with less than m inputs into the fewest possible LUTs. We use the proposed algorithms for mapping sequential logic on to a commercial LUT-based architecture containing sequential elements. We also investigate issues in logic optimization for LUTs. In particular, we establish the inadequacy of the standard cost function and propose a new one. The new cost function suggests that for high quality implementations, optimization should not be technology-independent, but rather should be tightly integrated with mapping.

In the first part we also address the theoretical complexity issues regarding the minimum number of LUTs needed for a function. We derive complexity upper bounds and demonstrate that they can be used to quickly and quite accurately predict the LUT-count without doing any mapping. Finally, algorithms for performance optimization are presented. Because of the constraints imposed by the architecture and programming methodology, the wiring delays can be a significant fraction of the total path delay. Lacking placement information, the logic-level delay models cannot handle wiring delays. Our contribution is to solve the problem by coupling logic-level optimization with timing-driven placement of the LUTs.

In the second part, our main contribution is to demonstrate that for obtaining high quality solutions on multiplexor-based architectures, the mapping algorithm should use a multiplexor-based representation for the functions instead of the conventional NAND-based one. Efficient architecture-specific algorithms to construct and map the representation are given.

In both parts of the thesis, theoretical results regarding the optimality of various algorithms are presented. These algorithms have been implemented in a system called mis-fpga, and are compared with those developed by other researchers. On average, 10-30% improvement in the solution quality is obtained, establishing the effectiveness of our techniques.

Prof. Alberto Sangiovanni-Vincentelli
Thesis Committee Chairman

Contents

List of Figures

List of Tables

List of Theorems and Procedures

Acknowledgements

Taking a slight detour from the norm, I start with a story.[1]

Scene: It's a fine sunny day in the forest, and a rabbit is sitting outside his burrow, tippy-tapping on his typewriter. Along comes a fox, out for a walk.

Fox: "What are you working on?"
Rabbit: "My thesis."
Fox: "Hmm. What is it about?"
Rabbit: "Oh, I'm writing about how rabbits eat foxes."

(Incredulous pause)

Fox: "That's ridiculous! Any fool knows that rabbits don't eat foxes!"
Rabbit: "Sure they do, and I can prove it. Come with me!"

They both disappear into the rabbit's burrow. After a few minutes, the rabbit returns, alone, to his typewriter and resumes typing. Soon, a wolf comes along and stops to watch the hardworking rabbit.

Wolf: "What's that you're writing?"
Rabbit: "I'm doing a thesis on how rabbits eat wolves."

(Loud guffaws)

Wolf: "You don't expect to get such rubbish published, do you?"
Rabbit: "No problem. Do you want to see why?"

[1] lifted, without permission, from a computer newsgroup

The rabbit and the wolf go into the burrow, and again the rabbit returns by himself, after a few minutes, and goes back to typing.

Scene: Inside the rabbit's burrow. In one corner, there is a pile of fox bones. In another corner, a pile of wolf bones. On the other side of the room a huge lion is belching and picking his teeth.

THE END

MORAL OF THE STORY:

- It doesn't matter what you choose for a thesis subject.
- It doesn't matter what you use for data.
- What does matter is who you have for a thesis advisor.

And I was very fortunate to have not one but two wonderful advisors in Bob Brayton and Alberto Sangiovanni-Vincentelli. The flexible and open stimulating research environment they have created in 550 Cory is a matter of pride for those of us in 550 and a thing to envy for those without. *Grazie mille*, Alberto, for providing me with an opportunity to work in the famous CAD group. His foresightedness, technical acumen, inexhaustible enthusiasm, and constant interest have inspired me and kept me going. I am especially indebted to him for teaching me, very patiently, how to give presentations and write papers (once he turned a theorem I had proved into a definition!). Bob was kind enough to take me under his umbrella. My interactions with him were always stimulating and fruitful. I have always marveled at his thoroughness ("*This should be* $abc + a'd$ *and not* $a'bc + ad$."), mathematical precision, and modesty. He is an invaluable resource for the entire group. Thanks a million, Bob.

I am also very grateful to Prof. Shmuel Oren for agreeing to serve on my qualifying exam and thesis committees at a short notice. I also thank Prof. Bob Brodersen for agreeing to be the chair of my qualifying exam committee.

I have always admired Prof. Richard Newton for his mastery of communication skills. The annual CAD group meetings conducted by him are so lively. I thank Prof. Richard Karp for teaching me wonderful courses on complexity theory and algorithms. My M.S. thesis at CMU was under the supervision of Prof. Don Thomas, and I thank him. Prof. Jonathan Rose from University

of Toronto, Steve Trimberger from Xilinx, and Ewald Detjens from Exemplar Logic have been very supportive. I have benefited tremendously from the numerous discussions I had with them on FPGAs. I thank Prof. Gabrielle Saucier for inviting me to Grenoble and Paris for Euro-Asic 1992.

Several people have contributed to the research in this thesis. The research began as a class project of the Logic Design course (EE290LS), in the spring of 1989. Yoshihito Nishizaki, Narendra Shenoy, and I started investigating the combinational logic synthesis problem for look-up table FPGAs under the guidance of Alberto and Bob. Yoshihito formulated the *covering* and *merging* problems of Chapter 3, whereas Narendra developed the first version of the *partition* heuristic. The work on performance optimization in Chapter 7 was done jointly with Narendra, and so was the initial, BDD-based work on the multiplexor-based synthesis of Chapter 8. Needless to say, Bob and Alberto guided me all along, transforming my rumblings to meaningful ideas. Special thanks go to Bob, Alberto, Huey-Yih, and Tiziano for their efforts in reading and correcting the thesis. Countless others helped shape the abstract of the thesis, the most difficult part for me to write, and I thank them all.

The funding for my research was provided by DARPA, NSF, and industrial grants from AT&T, IBM, Intel, Micro, Xilinx, and Actel, and I gratefully acknowledge it. Digital Equipment Corporation ensured that I get a fancy workstation, recently with a CD player whose real purpose, I am told, is not playing CDs!

Brad Krebs and Mike Kiernan provided an excellent computing environment. Heather Brown, Genevieve Thiebaut, and Carol Lynn Stewart in the EECS graduate office made amply sure that my lack of planning did not hinder my progress in the graduate school. A special thanks goes to Heather for being so cordial and warm. Chere, Tito, and the ERL staff deserve a special thanks for their assistance.

The *short turnaround* time for completing a design project in the CAD group is due to FPGA-use.. oops!, well, due to a superb programming environment developed by Rick Spickelmier & Dave Harrison (the OCTTOOLS gurus), and Rick Rudell & Albert Wang (the designers of the wonderful misII framework and interface).

550 Cory has been my second home in Berkeley (first home if that's where you spend the nights). For making this stay comfortable, nay wonderful, I owe infinite gratitude to the following: Abdul Malik (for being unassuming. A close friend. Volunteered to drive us to the South Bay when no one would rent us a car), Abhijit Ghosh - also known as Abhijit Babu, Adnan Aziz (the complexity guru), Albert Wang (for writing an inspiring thesis), Alex Saldanha (a prolific researcher) and Avril, Anantha Chandrakasan (the low-power guru), Andre Nieuwland (the coffee

master), Andrea Cassotto (for his sweet, helpful nature), Arlindo Oliveira (the chess master), Brian Lee, Chris Lennard (for his hilarious e-mails and infectious laughters. I vividly remember the CAD group annual meeting in which the fresh grads were asked to identify themselves and their advisors, if any. Chris identified himself thus: "My name is Chris Lennard. I am from Australia and therefore my advisor is Richard Newton."), Chuck Kring, Cormac Conroy (for making me realize that simple questions like "When are you going back?", when asked in 550 at 4:57 a.m., can have complicated answers like "Hmm - well - spiritually I'm always going back - I'm in a permanent state of regression..."), Daniel Engels (the coconut guy, tough-looking from without and soft from within), Desmond Kirkpatrick, Dev Chen (for being a great friend), Ed Liu (for fun-filled discussions on unimportant issues such as the goal of life and the optimum life-partner), Edoardo Charbon and Tokiko (for wonderful pasta and frequent parties), Elise Mills (for making sure that my bank balance stayed out of trouble and for always being cheerful), Ellen Sentovich (for organic dinners and breakfast chats, and, of course, for being an understanding, tolerant, methodical sis-manager), Eric Felt, Eric Tomacruz, Felice Balarin, Flora "*CAD-group Mom*" Oviedo (for being ever-so-helpful, warm, funny, affectionate, hardworking - words fail me here. A big thank you Flora), Gary York, Gitanjali Swamy (for teasing me by distorting my last name and thus creating an illusion that I am in high school - well, I guess the last remark makes sense to Hindi-speaking folks only) and Sanjay, Gregg Whitcomb, Hamid Savoj (the don't care guy), Harry Hsieh, Henry Chang, Henry Sheng, Hérvé Touati (a software virtuoso), Huey-Yih Wang (for being a true comrade and well-wisher. A wise man from the Orient who taught me that the shortest route from A to B is not always a straight line (no, he was not giving me a lesson in non-Euclidean geometries; it was about ways of the world and matters of the heart)! Without his continued persistence, I would have never applied for the credit card and driver's license - it is altogether a different matter that I still do not have them), Jagesh Sanghavi (for being a great, readily-helpful roommate and friend for last one and a half years. I have benefited immensely from his "the way I look at it"-vision. He has all signs of a dashing, successful entrepreneur), Jaijeet Roychowdhury - also known as Jaijeet Babu in the Indian circle - (for teaching me that being cynical can be fun, and if one repeats "life stinks" many times, it does not any more), Kia Cooper (for spreading the sunshine and for making sure that I stayed out of trouble), K. J. Singh (for his eagerness to help - misll-environment can be a horrendous maze for the novice. KJ, the expert, guided me through it umpteen times. Ever since he left, the graveyard shifts have not been the same), Luciano Lavagno and Paola (whom I should actually sue for the anguish they caused me by leaving Berkeley for Italy. It was an education to be around Luciano - a complete master), Macs Chiodo and Susan (for showing me, by personal example, that

room-mates are made in heaven), Mani Srivastava (the DSP guru), Mark Beardslee (for being full of cheer and warmth), Massoud Pedram (for being a nice, helpful friend), Miodrag Potkonjak (for his sense of humor), Narasimha Bhat and Shashikala (for the appetizing dinners and wonderful times spent together), Narendra Shenoy - popularly known as Nari - (for religiously stopping by my cubicle every day and chatting, a very close friend), Paolo Giusto (for introducing me to Cafe Strada this year! A very popular and charming guy; I envy him. He deserves the credit for adding the new dimension of dancing to my life), Paul Gutwin, Paul Stephan and Shirley (for fun-filled chats), Pranav "the trivial" Ashar and Darshana (for being true caring friends), Praveen Murthy, Rajeev Ranjan - *the other* Rajeev - (uses `rajeev` as his login name, causing much mayhem. Since so many of my e-mails go to him, he knows too much about my private life. So he leaves me with no choice but to ...), Ramin Hojati (for making me believe - at least once - that I can be a millionaire), Renu Mehra (for being a cheerful friend), Rick McGeer (for teaching me that the best way of implementing $\lceil \log_2 a \rceil$ is not $\lceil \log_{10} a / \log_{10} 2 \rceil$), S. Sriram, Shankar Narayanaswamy (for his honest opinion about everything under the sun), Sharad Malik (for being a true mentor, friend, and motivator), Shuvra Bhattacharya and Arundhati (for delicious Bengali food), Sriram Krishnan (for being a great friend and confidant, wonderful cubicle-mate, and tough English teacher), Sovarong Leang (for inspiring me with his dancing prowess), Stephan Edwards (for being so witty), Sunil Khatri, Szu-Tsung Cheng, Tim Kam (for being so nice and modest) and Kate, Tom Shiple (a close buddy I could turn to for honest opinion and sage advice) and Suzanne (for wonderful Mexican dinners), Tiziano Villa (for introducing me to Cal Performances and Western classical and folk music. I could always depend on him for any assistance. A scholar and master not only of logic synthesis, but also of linguistics, music, and literature) and Maria (for delicious pasta) and Marta (for her charming innocence), Umakanta Choudhury, Vigyan Singhal (for amazing me with his versatility), Wendell Baker and Mary (for the best cookies in town), William Lam (for inspiring me with his innovative work), Yosinori Watanabe and Mika (for being wonderful friends), Yoshihito Nishizaki (for being my EE290LS project partner in Spring 89), and Zeina Daoud (for teaching me the basic steps of East Coast Swing after a miserable class).

I thank my neighbors over the years in 550 for putting up with my not-so-soft voiced discussions and for taking phone messages in the unearthly day hours: S.T., Abdul, Tom, Alex, Charmine, Sharad, Nari, Arlindo, Sriram, Huey-Yih, Ramin, Adnan, Hamid, Mitch, ...

Life outside Cory Hall would have been vacuous if it were not for close friends. I thank them all: Sushil Verma (for being one of the closest, true friends from the wonder-years) and Diane (for her cheerful and sweet nature), Ashwini and Uma (for being so warm, affectionate, funny,

helpful, and for wonderful South Indian delicacies), DG, Shomik (my only Indian male friend with a pony-tail. He has attained wisdom at such a young age. Also has a complete mastery over the English language. He shot to fame after being rescued by a chopper at Point Reyes), Vinod (my studious ex room-mate) and Asha, Munnu & Savita & Usha, Madhu (the party animal of the *New York Times* fame), Khedkar (the football freak and the living encyclopaedia), Sandeep Pandey, Mahalingam, Anil, Neerav, Logie, Amit Marathe, and Kumud Sanwal.

All work and no play makes Jack a dull boy. My numerous table-tennis partners (e.g., Mehdi, Chin-Woo, and Dave) at the university and the Berkeley Table-Tennis Club saw to it that not many dull moments crossed my way. Soccer pals at Albany and my tennis buddies (Alberto, Pranav, KJ) took care of the rest. The Bears football team made my stay at Berkeley by trouncing Stanfurd 46-17 in this year's BIG GAME, thus making sure that I graduate this year (last year after our loss, I had pledged that I won't graduate until we won). Also thanks to the Bears basketball team for making it to the last 16 and doing us proud.

I am honored to have been a part of the rich tradition of *Cal.* Berkeley has been a great learning experience for me. I express my heartfelt gratefulness to - the ASUC, the Sproul Hall administration, the International House, the RSF staff, the foreign student office, Roberto - my music teacher, Mrs. Hamida Chopra for the Urdu *baithak*, my dance teachers, the street jugglers, and the list goes on...

I also thank the CMU gang for long-distance fun: Alok (the CAD guru), Ajai (the CS-theory guru), Haresh (the generic guru), Adito (the artificial-gene guru), Aarti (the CS guru), Sharbari (for introducing to the *Rabindra sangeet*) and collectively Rajen, Anthony, Ed, and John (the high-level gurus). My association with Pawan, who was my room-mate at Berkeley but was lured by the whiter pastures of MIT, has been very close and inspiring. A special thanks to him.

I am grateful to Mrs. Ruth Brayton for inviting me over for wonderful lunch and dinner parties. And thanks especially for the flower-plant you sent me on the commencement.

I thank my Bay-area cousins Neena and Poonam, and also Sunil for their constant encouragement and support. We had great times together.

I thank my maternal grandparents, late paternal grandparents, relatives, and friends in India and around the globe for being so loving and helpful.

A couple of nonsensical thanks. First, thanks to God for not existing! And then, thanks to my future life-partner for not meeting me so far (how can I be so sure of that?); otherwise I may not have been able to complete the thesis (and I leave it to the reader to decide if that would have been a better deal for him/her).

Finally, I thank my parents - Mummy and Daddy, elder sister Sangeeta (I call her Didi), younger brother Puneet (he would not want me to mention his nick-name here), and our old sweet little dog Shibu. I know (not about Shibu though) that the last six and a half years of separation from me have been very trying for them, especially for my mother. Their sacrifices have brought me where I am now. I do not know how to thank them for their immense patience, selfless love, and unbounded affection. I respectfully and lovingly dedicate this thesis to all of them.

Chapter 1

Introduction

1.1 Motivation

When faced with the task of designing the next generation processor, the designers of company A first come up with a system description of the processor. It includes detailed description of the instruction set, the interface with the external world, design objectives and constraints, etc. Then, using years of expertise in integrated-circuit design, they produce an implementation that meets the design objectives. In order to verify that the implementation is functionally correct (for example, on fetching and executing an ADD instruction, the correct sum is produced), sequences of input values are applied, and it is checked if the desired outputs are generated. Very likely, the processor is a huge and complex design, and so cannot be tested exhaustively. After achieving a reasonable degree of confidence in the correctness, the designers send the design for fabrication. In due time, say a month, the chip comes back from the foundry and is tested again to verify that it works as expected. This time it is much faster to simulate the same set of test vectors, so many more can be used, and more functionality can be tested for. If the chip fails, it is due to either a manufacturing defect, in which case the chip is discarded, or the non-exhaustive testing done earlier on. If the latter, the faulty part of the circuit is identified and fixed, and the modified design is resent for fabrication.

Consider another scenario in which the chip passes all the tests, but during the fabrication, it is decided that one new instruction should be added to the instruction set. The design-fabrication-test cycle has to be repeated here as well, as shown in Figure 1.1 (A).

After some iterations, the processor chip is finally ready to be shipped - however, the entire cycle may have taken a year or two. If a rival company B is also working on a similar

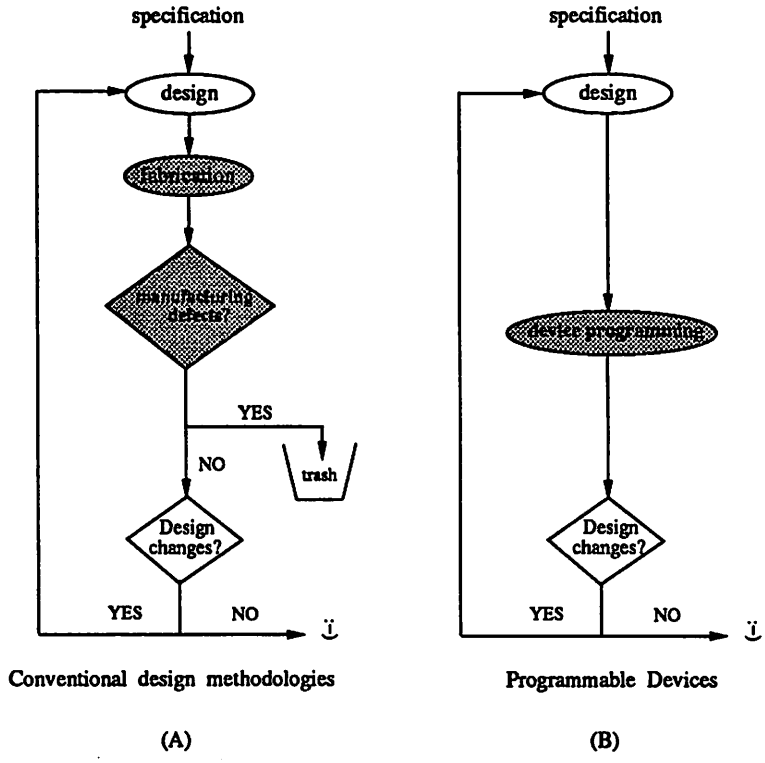


Figure 1.1: Conventional vs. programmable design methodology

processor, it is crucial for A to have its processor hit the market first. This is in accordance with a cardinal principle of the 20^{th} century economics, namely, whoever enters the market first (with the right product) captures it for the first few years at least, when most of the profit is to be made. It becomes critical then to minimize the design-manufacture-test cycle time. One way of doing so is to use programmable hardware. The components of this hardware lie uncommitted on an already fabricated chip, and can be programmed by the user to implement any kind of digital circuit. This methodology eliminates the dependence of the manufacturing/mask process from the design process. In fact, chip fabrication is removed from the cycle, reducing the cycle time from months to hours. This alternate methodology is shown in Figure 1.1 (B). Regardless of whether the final implementation is done on programmable hardware, the entire design process is sped up. Moreover, if the hardware is reprogrammable, the design changes can be made at no added expense.

Although programmability offers significant benefits, it introduces some disadvantages. The current programming technology (i.e., the method by which connections are formed) requires much larger area than the metal lines, causing lower logic densities. In addition, it introduces

series resistance and parasitic capacitance, degrading the overall device performance. Given these inherent limitations, it is not feasible (at least today) to implement a complex, high speed processor using the programmable technology. However, functional prototyping and design modifications can be carried out using the programmable hardware before the final design is sent for fabrication.

Now that the manufacturing step has been bypassed effectively (at least in the first few iterations), the design process itself, which traditionally has been a manual process, can become a bottleneck. With the growing complexity of the digital circuits, a complete manual design is a cumbersome and slow process, and is out of question. Therefore, automatic synthesis tools that start with a specification of the design and produce a satisfactory implementation on the programmable device are required. This thesis addresses issues in designing such tools. To put this work in a proper perspective, we first survey the design process, and then the programmable devices.

1.2 Designing VLSI Circuits

The design of digital systems, especially very large scale integrated (VLSI) systems, is a complex process, and for convenience's sake, is divided into the following steps, as shown in Figure 1.2.

Design Specification: The desired behavior of the system is specified at some level of abstraction. In Figure 1.2, a two-bit comparator that compares $a = (a_1, a_2)$ and $b = (b_1, b_2)$, and generates $out = 1$ when $a > b$, is described by its behavior.

High Level Design: This stage transforms the design specification into a description that uses memories, adders, register files, controllers, etc. This description is called the **register-transfer level**, or RTL, description. If the design is too big, it is partitioned into smaller pieces to reduce the overall complexity. Depending on the design objectives and constraints, this step determines how many functional units (e.g., adders, multipliers, multiplexors (MUXes)) and registers should be used, at what time steps these elements should be exercised (e.g., memory reads and writes, selecting the 0 data input of the MUX, etc.). For the comparator, this step corresponds to generating the Boolean equation specifying the dependence of *out* on inputs a and b.

Logic Design: The RTL description is first optimized for an objective function, such as minimum chip area, meeting the performance constraints, low power, etc. This step is called **optimization**. The optimized representation is then **mapped** to some primitive cells present in a

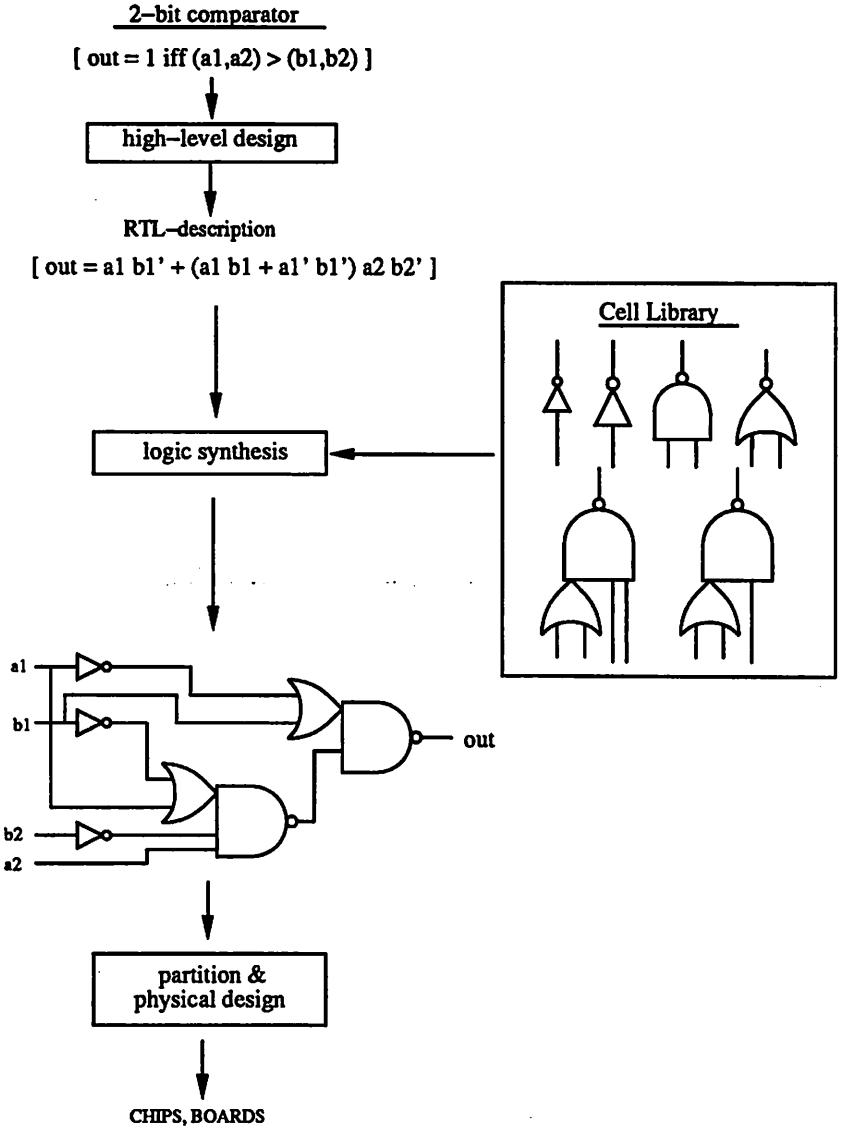


Figure 1.2: The design process

library. This final implementation is in terms of interconnections of gates, functional units, and registers. For the comparator, a simple interconnection of gates of the library is obtained.

Physical Design: The locations of various modules on the chip are determined (**placement**), and the interconnections of the circuit are **routed** between or through the placed modules. Also, the pad locations for inputs and outputs are determined in this step. The final layout is sent for fabrication.

Some of these steps may have to be iterated on if the final implementation does not meet the design objectives.

With the growing complexity of the integrated circuits, it becomes essential to use automatic tools for these steps. These tools are not only faster, but can also explore larger design space as compared to a human designer, potentially generating better designs. As of today, computer-aided design (CAD) tools exist for high level, logic, and physical design.

The subject matter of this thesis pertains to automation of the logic design step, also called **logic synthesis**. Logic synthesis takes the circuit description at the register-transfer level and generates an optimal implementation in terms of an interconnection of logic gates. Typically synthesis is done for an objective function, such as minimizing the cost of the design (which may be measured by the area occupied by the logic gates and interconnect), minimizing the cycle time, minimizing the power consumed, or making the implementation fully testable.

1.3 Programmable Devices: Field-Programmable Gate Arrays

Short turnaround time has become critical in the design of electronic systems. Software-programmable components such as microprocessors and digital signal processors have been used extensively in these systems, since they provide for quick design revisions. However, the inherent performance limitations of software-programmable systems makes them inadequate for high-performance designs. As a result, designers turned to a mask-programmable hardware solution, namely gate arrays. However, they do not offer the flexibility of user-programmability, and the manufacturing time is still a bottleneck.

The user-programmable hardware devices are prefabricated as arrays of identical programmable logic blocks with routing resources, and are configured by the user into the desired circuit functionality. Consequently, turnaround time is much smaller. This makes them attractive for rapid system prototyping. A subclass of these devices is the reprogrammable devices - those

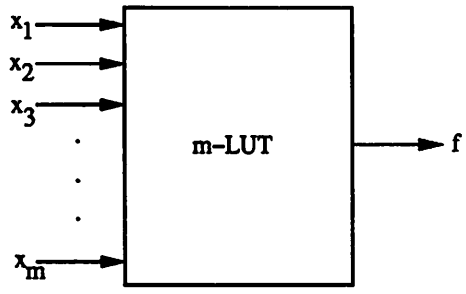


Figure 1.3: An m-input LUT

that can be programmed any number of times. Reprogrammability reduces the overhead for making design changes. Broadly speaking, the user-programmable devices can be broadly classified into two categories:

1. Programmable logic devices (PLDs), and
2. Field-programmable gate arrays (FPGAs)

PLDs are typically interconnections of programmable logic arrays (PLAs) [11]. Commonly used PLD architectures are those offered by A.M.D., Altera, and Plus Logic. FPGAs, on the other hand, have fine-grain logic blocks or gates (gate-array). Examples of such architectures are the Xilinx [88] and Actel architectures [29].

The basic FPGA architectures share a common feature: repeated arrays of identical logic blocks. A **logic block** (or **basic block** or **logic module**) is a versatile configuration of logic elements that can be programmed by the user.

1.3.1 Block Structures

There are two popular categories of FPGA block structures, namely **Look-Up Table-based** (LUT) and **multiplexor-based**; the resulting architectures are called LUT-based and MUX-based architectures respectively.

Look-Up Table-based Architectures

The basic block of an LUT architecture is a look-up table that can implement *any* Boolean function of up to m inputs, $m \geq 2$. For a given LUT architecture, m is a fixed number. In commercial architectures, m is typically between 3 and 6. Figure 1.3 shows an m-input LUT, also

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

Figure 1.4: An SRAM with 3 address lines implements a 3-input function $f(x_1, x_2, x_3)$

written as m-LUT. An m-LUT is typically implemented by static random access memory (SRAM) that has m address lines and 1 data line. The following example illustrates how an m-LUT can implement any Boolean function of up to m inputs.

Example 1.3.1 *Let m be 3. Consider* $f(x_1, x_2, x_3) = x_1'x_2'x_3' + x_1x_2'x_3 + x_1x_2x_3$. *Consider an* SRAM *that is 1 bit wide and has 3 address lines. To implement f, first tie the address lines of the* SRAM *to* x_1, x_2, *and* x_3, *and its single-bit output data line to f. The entries in the* SRAM *are stored as follows. f evaluates to 1 for* $x_1 = 0, x_2 = 0$, *and* $x_3 = 0$. *This corresponds to storing a 1 at the address* $(x_1, x_2, x_3) = (0,0,0)$. *This is shown in Figure 1.4. For other input combinations, 0s or 1s can be stored appropriately. So, the data line of the* SRAM *gives the value of f corresponding to the input combination present at the address lines.*

In commercial LUT-based architectures, each basic block has one or more LUTs, along possibly with other logic elements (such as flip-flops, fast carry logic, etc.). For example, Figure 1.5 shows the basic block of the Xilinx 3090 architecture, also called a **configurable logic block** (CLB). It has 6 external inputs a, b, c, d, e, and DIN, and has two outputs X and Y. The heart of the CLB is the LUT section, which consists of two 4-input LUTs with outputs F and G. Since there are, in all, seven inputs to the LUT section, the two LUTs have some common inputs. This imposes constraints on the possible function pairs realizable by the LUT section. For designing sequential circuits, the CLB has two flip-flops QX and QY, whose outputs are fed back to the LUT section. The outputs X and Y of the CLB can be either F or G (i.e., the outputs are *unlatched*), or QX or QY (i.e., the outputs are *latched*).

Multiplexor-based Architectures

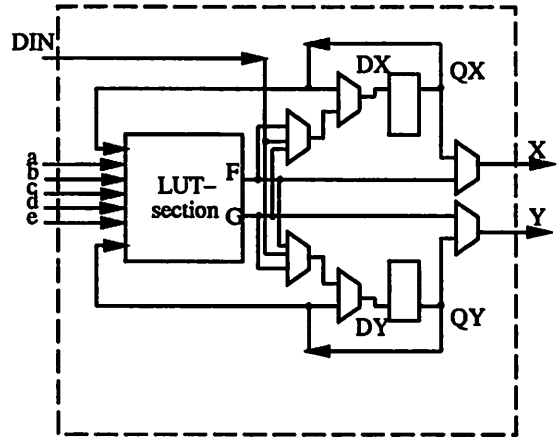


Figure 1.5: Xilinx 3090 CLB

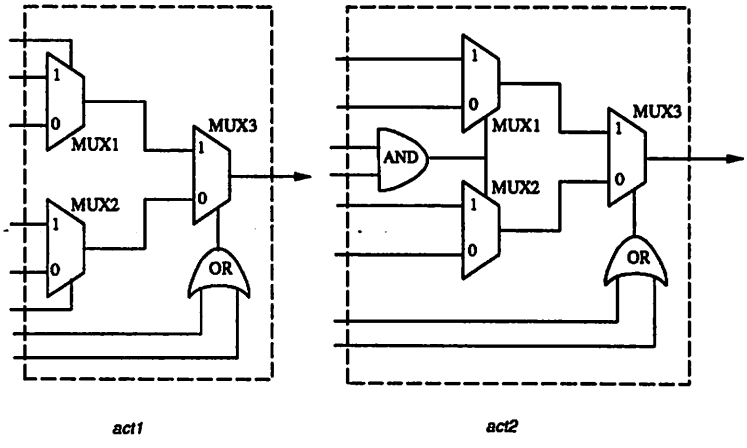


Figure 1.6: Two logic modules from Actel: *act1* and *act2*

In the MUX-based architectures, the basic block is a configuration of multiplexors [29]. Figure 1.6 shows the basic blocks of two architectures, *act1* and *act2*, from Actel. *act1* has three 2-to-1 MUXes, configured in a balanced tree, with an OR gate feeding the select line of MUX3. *act2* is similar, except that MUX1 and MUX2 share their select lines, which is the AND of two of the module inputs.

1.3.2 Realizing Interconnections

The interconnections between the blocks have to be programmed in order to realize the desired circuit connectivity. Interconnect can be either reprogrammable or one-time programmable.

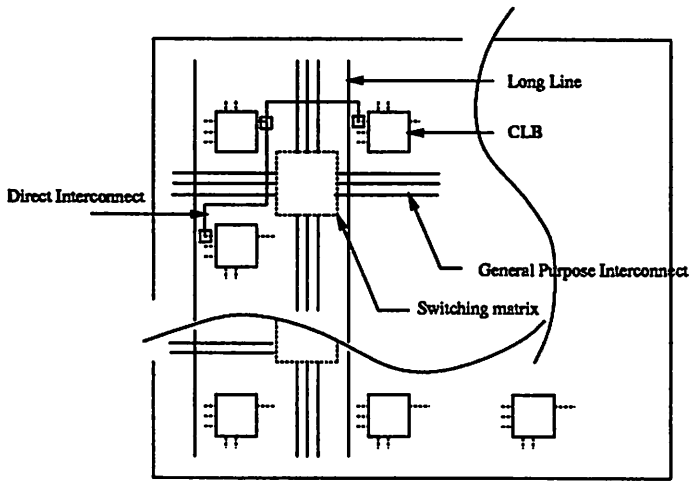


Figure 1.7: Interconnection structure in Xilinx 3090 architecture

Figure 1.7 shows the three kinds of interconnects present in Xilinx 3090:

1. *Direct interconnect*: connects the output of a CLB to an input of the adjacent CLB.
2. *General purpose interconnect*: realizes arbitrary connections using metal segments joined by reprogrammable pass transistors (switching matrix).
3. *Long lines*: run across the chip; mainly used for clocks and global signals.

In the Actel architectures, there are rows of logic modules, which are separated by routing channels, as shown in Figure 1.8. The routing channels contain metal segments, which can be connected by one-time programmable anti-fuses.

1.3.3 Logic Synthesis for Field-Programmable Gate Arrays

The problem of synthesis for PLDs is similar to the PLA-optimization problem, which is well-understood and for which good quality software tools exist (e.g. ESPRESSO [11]). Since FPGA devices are relatively new, the synthesis problem for them has not been studied until very recently. The main constraints in synthesizing circuits onto these architectures are:

1. a limited number of blocks on a chip (e.g., the Xilinx 3090 chip has 320 CLBs),
2. the functionality of the block, i.e., what functions can be put on a block, and
3. limited wiring resources.

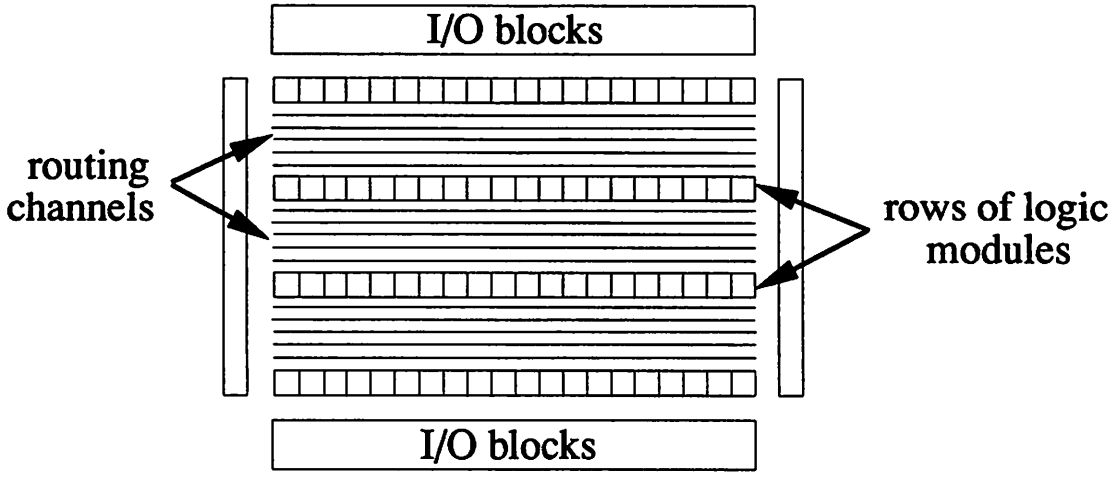


Figure 1.8: Actel architecture

Given a circuit description, say in terms of Boolean equations, the problem is to realize it using basic blocks of the target FPGA architecture, meeting some design objectives. It is this problem that this thesis addresses.

1.4 Thesis Overview

The thesis is in two parts. The first one addresses the synthesis problem for LUT-based architectures, and the second the synthesis problem for the MUX-based architectures. Specifically,

- Chapter 2 first introduces the basic terms used in the thesis, the problem of logic synthesis, and then motivates this research.
- Chapter 3 describes mapping techniques for combinational logic for the smallest design for LUT-based architectures. A small design is approximated as the one that uses the minimum number of basic blocks.[1] Although most of these techniques target the m-LUT of Figure 1.3, which is the simplest LUT architecture, we also show how to use them as a core in the techniques for some of the commercial architectures, e.g., Xilinx 3090.
- Chapter 4 describes techniques for optimization for combinational logic for minimum number of basic blocks for LUT architectures. This work is still in its infancy, and a lot more needs to be done.

[1]This view ignores routing considerations and pin limitations.

- Chapter 5 deals with the problem of determining the minimum number of LUTs needed for a circuit (called its complexity). Computing the complexity is useful because then the absolute quality of the FPGA synthesis tools can be ascertained. Unfortunately, this is a difficult problem. The next best alternative is to determine lower and upper bounds on the complexity. If these bounds are reasonably good, they can be used to predict the table-count without doing any technology mapping.

- Chapter 6 addresses the problem of sequential synthesis for LUT-based architectures; in particular, we describe a few mapping algorithms. Although the algorithms are quite general, the current implementation is for a specific family, namely Xilinx 3090.

- Chapter 7 presents performance-oriented synthesis methods for LUT architectures. A two-phase approach is followed. In the first phase, timing driven transformations are applied at the logic level. An approximate delay model is used. In the second, timing driven placement and local resynthesis are performed. The delay information is obtained from a more accurate delay model that takes the wiring delays and fanout loading into account.

- Chapter 8 describes techniques for combinational mapping for minimum number of basic blocks for MUX-based architectures. Although we primarily deal with Actel's *act1* architecture, same techniques can be used for other architectures, e.g., *act2*.

- Chapter 9 summarizes the contributions of this work and presents directions for future work.

- Finally, Appendix A briefly describes the commands used in the software system we have implemented based on the algorithms described in the thesis.

Chapter 2

Background

2.1 Definitions

First, we define some basic terms pertaining to **combinational circuits**, namely those circuits whose outputs do not depend on the past history, but just on the current values of the inputs. Then, we present definitions for **sequential circuits**, namely those circuits whose outputs depend on the past as well as current inputs. Sequential circuits need memory elements to remember the history. They also have a combinational part to compute the output functions based on the current inputs and the past history.

2.1.1 Logic Functions

Definition 2.1.1 *Let* $B = \{0, 1\}$. *An n-input,* **completely specified logic function** f *is a mapping* $f : B^n \rightarrow B$. *Each element in the domain* B^n *is called a* **minterm** *of* f. $f^{-1}(1) = \{v \in B^n : f(v) = 1\}$ *is the* **on-set** *of* f, *and* $f^{-1}(0) = \{v \in B^n : f(v) = 0\}$ *the* **off-set** *of* f.

If all the minterms of f are in its on-set, i.e., $f(v) = 1$ for all $v \in B^n$, f is a **tautology** (or identically 1), and is also written $f = \mathbf{1}$ or $f \equiv 1$. Similarly, if $f(v) = 0$ for all $v \in B^n$, $f = \mathbf{0}$, or f is identically 0, or $f \equiv 0$.

Definition 2.1.2 *An n-input,* **incompletely specified logic function** f *is a mapping* $f : B^n \rightarrow \{0, 1, -\}$. $f^{-1}(-) = \{v \in B^n : f(v) = -\}$ *is the* **don't care set** *(or* **dc-set***) of* f. *It contains minterms for which the function value is unspecified (i.e., allowed to be either 0 or 1).*

In this thesis, the term "function" means a logic function.

Definition 2.1.3 *The* **complement of a logic function** f, *denoted* f', *is a logic function obtained by exchanging the on-set and off-set of* f.

Definition 2.1.4 *A* **literal** *is a variable or its complement. A* **cube** *or a* **product term** c *is a product or conjunction of one or more literals, such that if* x *appears in the product,* x' *does not, and vice versa.*

A literal a (a') represents the set of all minterms for which the variable a takes on the value 1 (0). A cube represents the intersection of the sets of minterms represented by all the literals in it. If some variable and its complement are present, the cube becomes identically 0. Also, if a variable appears complemented in a cube, it is said to be in the complemented or negative phase. Otherwise, if it is present uncomplemented, it appears in the uncomplemented or positive phase.
For example, consider two cubes ab and $a'b'c'$ in the variable space $\{a, b, c\}$. The cube ab has 2 literals, namely a and b, whereas the cube $a'b'c'$ has 3 literals a', b', and c'. The cube ab contains two minterms: abc and abc', whereas $a'b'c'$ contains just one minterm, namely $a'b'c'$.

Definition 2.1.5 *A cube* c_1 **contains** *another cube* c_2 ($c_2 \subseteq c_1$) *if the set of minterms represented by* c_2 *is a subset of the set of minterms represented by* c_1.

Definition 2.1.6 *An* **implicant** *of a function is a product term that does not contain any minterm of the off-set of the function. An implicant is* **prime** *if it is not contained by any other implicant of the function.*

Definition 2.1.7 *A* **sum-of-products** (SOP) *is a Boolean sum or disjunction of cubes.*

An SOP represents the union of sets of minterms represented by the cubes in it. For example, $ab' + a'bc'$ is an SOP with 2 cubes and 5 literals. It contains three minterms in the variable space $\{a, b, c\}$, namely, $ab'c, ab'c'$, and $a'bc'$.

Definition 2.1.8 *A* **cover** C *of a function* f *is an* SOP *which contains all the minterms of the on-set of* f, *but none from its off-set. A cover* C *is a* **prime cover** *of* f *if it consists only of primes.*

A logic function can have many covers.

Definition 2.1.9 *A cube* c *of a cover* C *of a function* f *is a* **redundant cube** *if* $C - \{c\}$ *is still a cover of* f, *i.e., if* c *covers only those vertices that are either covered by other cubes of* C, *or belong to the dc-set. A cover* C *is a* **redundant cover** *if some cube in it is redundant, otherwise it is* **irredundant**.

Definition 2.1.10 *A* **sum term** *(also an* **OR term***) is a Boolean sum or disjunction of literals. A* **product-of-sums** *(*POS*) is a product of sum terms.*

For example, $a + b + c'$ and $a' + b' + c$ are sum terms, and $(a + b + c')(a' + b' + c)$ is a POS.

Definition 2.1.11 *The* **cofactor** *of a function $f(x_1, x_2, \ldots, x_n)$ with respect to a variable x_1 is $f_{x_1}(x_2, \ldots, x_n) = f(1, x_2, \ldots, x_n)$, i.e., f when x_1 is tied to 1. Similarly, $f_{x_1'}(x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n)$. The* **Shannon expansion** *of $f(x_1, x_2, \ldots, x_n)$ with respect to x_i is*

$$f = x_i f_{x_i} + x_i' f_{x_i'} \tag{2.1}$$

Definition 2.1.12 *A function f is* **monotone increasing** *in a variable x_i if $f_{x_i'}(\beta) = 1$ implies $f_{x_i}(\beta) = 1$ for all $\beta \in B^{n-1}$, i.e., if increasing the value of the variable x_i from 0 to 1 never decreases the value of f from 1 to 0. Similarly, a function f is* **monotone decreasing** *in a variable x_i if $f_{x_i}(\beta) = 0$ implies $f_{x_i'}(\beta) = 0$ for all $\beta \in B^{n-1}$. The function f is* **unate** *in variable x_i if it is either monotone increasing or monotone decreasing in x_i. Otherwise, f is* **binate** *in x_i. The function f is unate if it is unate in all its variables. A* **cover** *C is* **unate in a variable** *x_i if the variable x_i appears in only one phase, either positive or negative, but not both, in C. Otherwise, C is* **binate** *in x_i. A* **cover** *C is* **unate** *if it is unate in all the variables.*

As shown in [11], a function that has a unate cover is unate. However, a unate function can have a binate cover. For example, $f = ab + c$ is unate, but its cover $abc + abc' + c$ is binate, since c occurs in it in both positive and negative phases.

Definition 2.1.13 *A Boolean function $f(x_1, x_2, \ldots, x_n)$ is called* **symmetric** *(or* **totally symmetric***) if it is invariant under any permutation of its variables. It is called* **partially symmetric** *in the variables x_i, x_j, $1 \leq i, j \leq n$, if the interchange of the variables x_i, x_j leaves the function unchanged.*

For example, $f_1(x, y, z) = x'yz + xy'z + xyz'$ is symmetric. $f_2(x, y, z) = xyz + x'y'z$ is partially symmetric in the variables x and y, since $xyz + x'y'z = yxz + y'x'z$. However, f_2 is not partially symmetric in the variables x and z, because $xyz + x'y'z \neq zyx + z'y'x$.

Definition 2.1.14 *A* **factored form** *is defined recursively as follows:*

- *a literal is a factored form,*

- *the sum of two factored forms is a factored form, and*
- *the product of two factored forms is a factored form.*

A factored form is a generalization of SOP that allows arbitrary nesting of operations. For example, $(a + b) + c$ and $((a + b)c') + (de')$ are two factored forms with 3 and 5 literals respectively. In Chapter 5, we will introduce a more general notion of factored form.

So far, we have been talking about a single logic function. Usually, a circuit has more than one outputs. This leads us to the notion of multiple-output functions.

Definition 2.1.15 *An* n**-input,** k**-output function** f *is a mapping* $f : B^n \rightarrow B^k$.

Definition 2.1.16 *A multiple-output function is represented as a Boolean network [12]. A* **Boolean network** η *is a directed acyclic graph (*DAG*), with some primary inputs* $PI(\eta)$*, primary outputs* $PO(\eta)$*, and internal (intermediate) nodes* $IN(\eta)$*.* **Primary inputs** *have no arcs coming into them, and* **primary outputs** *have no arcs going out of them. Associated with each* **internal node** i *of the network is a variable* y_i*, and representation of a logic function* f_i*. The logic at each node is stored typically as a sum-of-products form. There is a (directed) arc from node* i *to node* j *in the network if* j *uses* y_i *or* y_i' *explicitly in the representation of* f_j*. In that case,* i *is called a* **fanin** *of* j*, and* j *a* **fanout** *of* i*. The set of fanins of a node* i *is denoted as* $FI(i)$ *and the set of fanouts as* $FO(i)$*. If there exists a path from node* i *to node* j*, then* i *is said to be a* **transitive fanin** *of* j*, and* j *a* **transitive fanout** *of* i*. The set of transitive fanins of a node* i *is denoted as* $TFI(i)$*, whereas its transitive fanout set is denoted as* $TFO(i)$*. The* **net** *driven by node* i *is the set of edges of the type* (i, f^o)*,* $f^o \in FO(i)$.

Figure 2.1 shows a network with four primary inputs a, b, c, and d, one primary output z, and three internal nodes y, w, and z. The primary inputs and output nodes are drawn as squares, and the internal nodes as circles. a and b are fanins of y, and z is the fanout of y. The function associated with y is f_y (also written y) $= ab$. $TFI(z) = \{a, b, c, d, w, y\}$. $TFO(b) = \{w, y, z\}$.

Definition 2.1.17 *The* **binary decision diagram** (BDD) *of a function* $f(x_1, x_2, \ldots, x_n)$ *is a rooted directed acyclic graph (*DAG*) with vertex set* V *containing two types of vertices. A* **non-terminal vertex** v *has as attributes an argument* **index**$(v) \in \{1, 2, \ldots, n\}$*, and two children,* **low**(v) *and* **high**$(v) \in V$*. A* **terminal vertex** v *has a value,* **value**$(v) \in \{0, 1\}$.

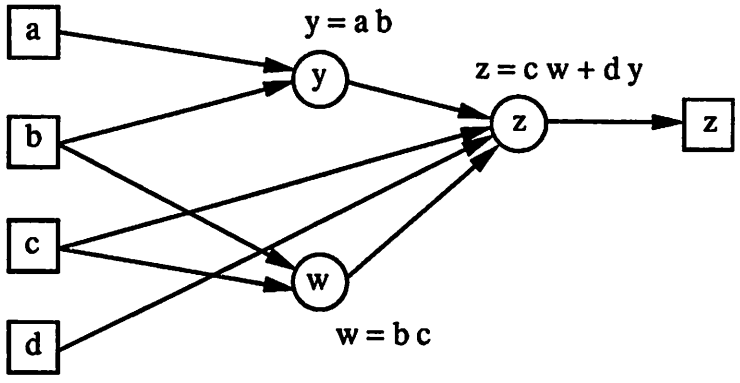


Figure 2.1: A Boolean network

Each vertex of a BDD has a function associated with it. The root vertex corresponds to the function f, the terminal vertex with value 0 to the function **0**, and the terminal vertex with value 1 to the function **1**. If the function at a non-terminal vertex v with index j is g, the function at $low(v)$ is $g_{x_j'}$, and at $high(v)$ is g_{x_j}. The edge connecting a non-terminal vertex v with index j to $low(v)$ carries the label 0, indicating that $low(v)$ is obtained from v by setting x_j to 0. Similarly, the edge connecting v to $high(v)$ carries the label 1.

A vertex v of a BDD is a **leaf vertex** (or a **leaf**) if either v is a terminal vertex, or $low(v)$ and $high(v)$ are terminal vertices with values 0 and 1 respectively (i.e., the function associated with v is simply some input variable). All other vertices are **non-leaf** vertices.

Definition 2.1.18 *A* BDD *is* **ordered** *if the indices of the vertices in all root-to-terminal vertex paths follow a fixed order. A* BDD *is* **reduced** *if there is no vertex u with* low(u) = high(u), *and there are no two distinct vertices v and w such that the sub-graphs rooted at v and w are isomorphic. A reduced ordered* BDD *is called an* ROBDD.

Figure 2.2 (A) shows an unordered BDD for the function $f(a,b,c,d) = ac + a'bd + bc'd'$. Figure 2.2 (B) shows an ordered BDD for f with the order c, a, d, and b, the root vertex being indexed by c. This ordered BDD can be reduced by noting that all vertices with index b represent the same function, namely b. Merging them all in one node, we get the ROBDD in (C).

Given an ordering of the variables, the ROBDD representation for a function is canonical (unique). This fact was first proved by Bryant in his seminal work [14]. This feature makes ROBDDs attractive for tautology checking (i.e., is a given function identically 1?) and hence functional equivalence. Although in the worst case, the size of an ROBDD can be exponential in

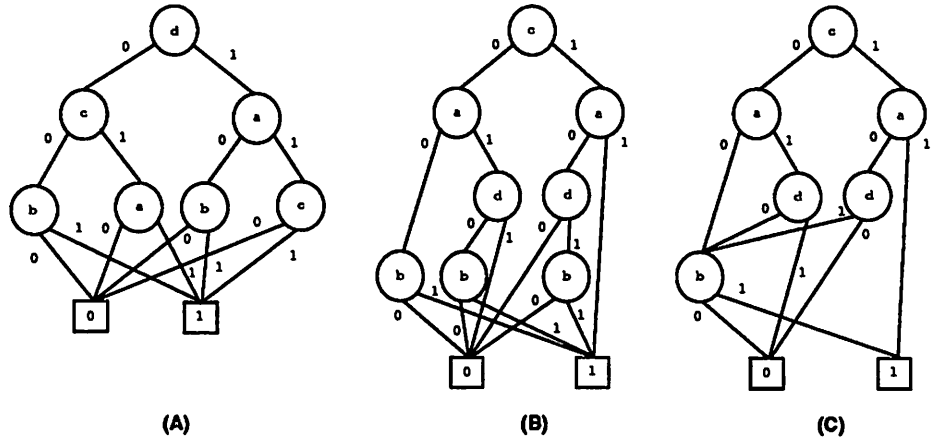


Figure 2.2: Example of a BDD, an ordered BDD, and a ROBDD

the number of variables, this representation is often compact. The size depends strongly on the ordering selected.

Definition 2.1.19 *The* **if-then-else** DAG *(*ITE*) for a function f is a* DAG *with two* **terminal vertices** *with values 0 and 1, and terminal vertices corresponding to inputs. Each* **non-terminal vertex** v *represents a 2-to-1 multiplexor, and has three children:* **if**(v), **then**(v), *and* **else**(v). *The interpretation is that the* if *child is connected to the control input of the multiplexor, and the* then *and* else *children are connected to the data (signal) inputs 1 and 0 of the multiplexor.*

The *if, then,* and *else* edges are denoted by I, T, and E respectively. Note that in a BDD also, a non-terminal vertex can be regarded as a 2-1 multiplexor whose control input is connected to the variable associated with the vertex. In the multiplexor corresponding to an ITE vertex, the control input can be any function. Thus an ITE is more general than a BDD and consequently can be more compact.

Example 2.1.1 *Consider function $f = ab + a'c + de$. As shown in Figure 2.3, a is selected as the top variable in the* BDD. *As a result, de gets replicated in both 0 and 1 branches. This can be avoided in the* ITE *by factoring out de before branching on a.*

Definition 2.1.20 *The* **support** $\sigma(f)$ *of a function-representation f, which is either an* SOP *or a factored form, is the set of variables appearing in the representation. $|\sigma(f)|$ represents the cardinality of $\sigma(f)$. The support of a set of function-representations can be similarly defined as the*

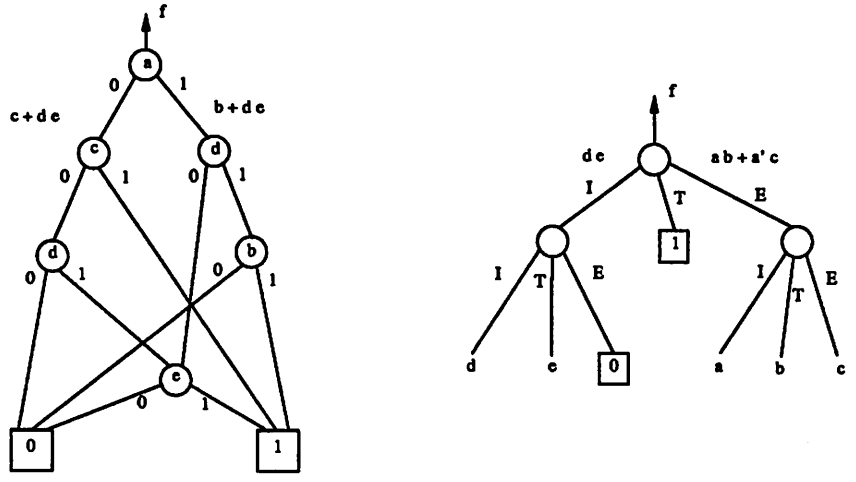


Figure 2.3: BDD vs. ITE

union of the supports of the individual function-representations. The support of a product term is the set of variables appearing in it. In particular, the support of a prime is called **prime-support**.

For example, if a representation $f = abc + ab'd$, $\sigma(f) = \{a, b, c, d\}$, and $|\sigma(f)| = 4$. Note that *the support depends on the function representation used.* For example, the last function can also be written as $f_1 = abc(e + e') + ab'd(e + e')$, in which case, $\sigma(f_1) = \{a, b, c, d, e\}$. However, each completely specified function has a unique minimum support, which is called its true support.

Definition 2.1.21 $x \in \sigma_T(f)$, *the* **true support** *of a function f, if $f_x \neq f_{x'}$. Then, f is said to* **essentially depend** *on x.*

If it is known that any representation of the function f handed to us is using only the true support variables, we can use the term *support of a function f* (or $\sigma(f)$) to mean the true support of f. As we will shortly see in Section 2.2.1 (in *simplification*), that is indeed the case if simplification is applied on the function representation during *optimization.*

In the context of Boolean networks, it is useful to consider the following two notions of supports. The **local support** of the (completely specified) function f at a node n is the set of fanins of n, whereas the **global support** of f is in terms of the primary inputs on which f depends topologically, i.e., the primary inputs that are in the transitive fanin set of n. For example, in Figure 2.1, the local support of z is $\{c, d, w, y\}$, and its global support is $\{a, b, c, d\}$.

The following notation is used:

$\sigma(f)$	local support of f
$\sigma_G(f)$	global support of f
$\sigma_T(f)$	true local support of f
$\sigma_{TG}(f)$	true global support of f

Definition 2.1.22 *For a given basic block, a function f is* **feasible** *if it can be realized with one block. A Boolean network η is feasible for the block if the function at each internal node of η is feasible.*

A feasible network can be directly implemented on the target FPGA architecture simply by implementing each internal node with a block. The final goal of synthesis is then to obtain a feasible network with fewest nodes or minimum delay, depending on the objective.

Definition 2.1.23 *A function f is m-***feasible** *if $|\sigma_T(f)| \leq m$, otherwise it is m-***infeasible**. *A Boolean network η is m-feasible if the function at each internal node of η is m-feasible.*

The motivation behind this definition is that an m-feasible function can be realized with one m-LUT. Note that the notion of m-feasibility has been defined in terms of the true support. In an optimized network, each function f is represented by a prime cover. It is well-known that a prime cover essentially depends on each variable appearing in the cover, and so is already on the minimum local support. In this case, instead of the true support, the support of the representation of f can be used in checking if a function is m-feasible. In many applications, when a prime cover is not available, the true support may be difficult to compute. Then, we approximate the m-feasibility test by checking if the support of the representation has at most m variables. This takes time linear in the representation of the function. As we show next, computing the true support of a function given an SOP is difficult, in fact NP-hard. The reader is referred to the book by Garey and Johnson [30] for a comprehensive coverage of NP-completeness and the well-known NP-complete and NP-hard problems. Define TRUE SUPPORT as the following decision problem:

INSTANCE: Given a cover of a function f, and $k \geq 0$.
QUESTION: Is $|\sigma_T(f)| \leq k$, i.e., does the true support of f have at most k variables?

To show that TRUE SUPPORT is NP-hard, we use an auxiliary problem TRUE SUPPORT ZERO:
INSTANCE: Given a cover of a function f.

QUESTION: Is $|\sigma_T(f)| = 0$, i.e., is f identically 0 or identically 1?

Proposition 2.1.1 *TRUE SUPPORT ZERO is NP-hard.*

Proof It suffices to Turing-reduce [30] an NP-hard problem to TRUE SUPPORT. For this reduction, we use the NP-hard TAUTOLOGY problem [30], which is as follows:

INSTANCE: Given a cover of a function f.
QUESTION: Is f a tautology, i.e., identically 1?

The definition of Turing reduction permits a polynomial number of invocations of an oracle (subroutine) that solves TRUE SUPPORT ZERO (i.e., returns YES if $|\sigma_T(f)| = 0$, NO otherwise) in order to solve TAUTOLOGY.

Note that $|\sigma_T(f)| = 0$ if and only if f is either identically 0 or identically 1. Given an SOP for f, f is identically 0 if and only if the SOP is simply 0 (i.e., has no cubes). To answer if f is a tautology, proceed as follows. Call the oracle for TRUE SUPPORT ZERO.

1. If it returns NO, f is not a tautology.
2. Otherwise, $|\sigma_T(f)| = 0$. Then there are two cases: either f is a tautology or it is identically 0. To differentiate between the two, simply check if the cover of f is 0 (i.e., has no cubes). If it is, f is not a tautology. Otherwise, f is a tautology.

■

Proposition 2.1.2 *TRUE SUPPORT is NP-hard.*

Proof Setting $k = 0$ makes TRUE SUPPORT equivalent to TRUE SUPPORT ZERO, which, from Proposition 2.1.1, is NP-hard. So TRUE SUPPORT is also NP-hard. ■

Definition 2.1.24 *An m-feasible Boolean network η is* **m-optimum** *if η has k internal nodes, and there exists no m-feasible network that realizes all the primary output functions ($PO(\eta)$) in fewer than k internal nodes.*

2.1.2 Representing a Logic Function

A completely specified logic function may be represented in many ways, some of which are as follows:

1. *Truth table*: It is always exponential in the support n of the function, as there are 2^n vertices in B^n.

2. *Minterms in the on-set (or off-set)*: Since a completely specified function partitions the input space into on-set and off-set, it is enough to explicitly give one set; the other one is uniquely determined. For an incompletely specified function, any two out of the on-set, off-set, and dc-set need to be provided. Although typically smaller than the truth table, this representation can be exponential in n.

3. SOP: It is typically more compact than the previous two representations, but in the worst case, it can be exponential. For example, for the EX-OR function

$$f(x_1, x_2, \ldots, x_n) = x_1 \oplus x_2 \oplus \ldots \oplus x_n,$$

the smallest SOP is exponential in n. The problem of obtaining a minimum SOP of a Boolean function, or, in general, a minimum-cost SOP where each product term has a non-negative cost, is referred to as the **two-level minimization problem**. The corresponding implementation is called a **two-level implementation**. The most popular form of two-level implementation is a **programmable logic array** (PLA). A PLA has two *planes* - an AND plane and an OR plane. The AND plane implements the product terms and the OR plane realizes their OR. In general, a PLA can have more than one output.

4. *Factored form*: It is typically smaller than the SOP, but can be exponential in n. The factored form suffers from the fact that it may not be possible in a factored form to share two instances of the same function, or of a function and its complement. This is because no signals except input variables can have multiple fanouts. For example, $f = (ab + cd)p + (ab + cd)'q$ (which is not a factored form since $ab + cd)'$ is not allowed) is represented in factored form as $f = (ab + cd)p + (a' + b')(c' + d')q$, which has 10 literals. A smaller representation is obtained by introducing an intermediate variable $x = ab + cd$. Then $f = xp + x'q$. The total number of literals is then 8, four each for x and f.

5. *Boolean network*: A Boolean network is the most general representation of a Boolean function, single- or multiple-output, in that there is a one-to-one correspondence between a

circuit realization and the Boolean network. It is more general than the factored form, since it removes the restriction of the internal nodes having single fanouts. Also, since a signal may go through many levels of logic, a Boolean network is an example of a **multi-level representation.**

6. BDD

7. ITE

2.1.3 Finite State Machines

Definition 2.1.25 *A* **completely specified Mealy Finite State Machine** FSM *is a six-tuple* $(S, I, O, \delta, \lambda, R)$*, where*

- S *is a finite set of* **states** *of the* FSM,
- I *is a finite set of* **inputs** *to the* FSM,
- O *is a finite set of* **outputs** *of the* FSM,
- δ *is a mapping from* $I \times S$ *to* S*, and is called the* **transition function,**
- λ *is a mapping from* $I \times S$ *to* O*, and is called the* **output function,** *and*
- R *is the* **initial** *(or* **reset***) state.*

In this thesis, since we will deal only with Mealy FSMs, we will call them FSMs. An FSM can also be represented as a directed graph, called the **state transition graph** where:

- each vertex is associated with a state, and
- each edge is labeled with an input/output pair, and is directed from the *present state* vertex to the *next state* vertex.

Example 2.1.2 *Figure 2.4 shows an* FSM *implementing a mod-3 counter. It has 3 states:* $S = \{R, s, t\}$*, one input* a*, and two outputs* b *and* c*. The machine starts from the reset state* R*, with the outputs* b *and* c *both set to 0. Irrespective of which state the machine is in, if the input* a *is 1, the machine counts up 1 (modulo 3) and makes a transition to another state. If* a *is 0, the outputs remain the same and machine stays in the same state.*

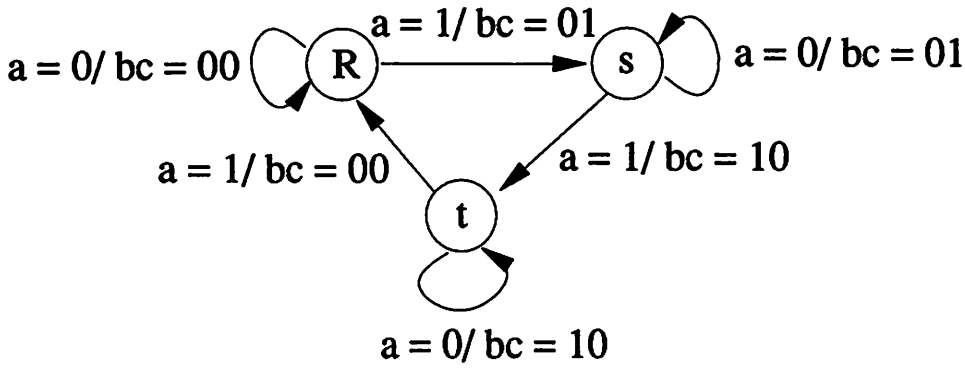


Figure 2.4: A finite state machine

Example 2.1.3 *As another example, consider a controller of a microprocessor, with states* $S = \{S_1, S_2, \ldots, S_k\}$. *Assume that the controller is in state* S_1 *when it fetches the instruction "ADD R1 R2" from the memory. After executing the instruction, the controller moves over to state* S_2. *In order to execute the instruction, the controller has to fetch the two operands from the registers R1 and R2, send a control signal to the adder to compute the sum, and enable the load signal of R1 to store the result in R1. In other words, the controller takes the present state (*S_1*) and external inputs (the instruction ADD and the names of the registers R1 and R2), and generates control signals (READ signal to R1 and R2, transferring their contents on the bus(ses), ADD signal to the adder, and finally LOAD signal to R1) and computes the next state (*S_2*).*

The Encoding Problem

Many descriptions of the logic systems include variables that, instead of being 0 or 1, take values from a finite set. In Example 2.1.2, the FSM has three *symbolic* states: R, s, and t. To obtain a digital circuit from the FSM, the states have to be assigned binary codes, since a signal in a digital circuit can only take values 0 and 1. The size of the circuit depends strongly on the codes assigned to the states. This leads to the problem of assigning binary codes to the states of the FSM such that the final gate implementation after encoding and a subsequent optimization is small. It is called the **state-encoding** (or **state-assignment**) problem. Note that it entails encoding of both symbolic inputs (present state variables) and symbolic outputs (next state variables). In other words, it is an **input-output encoding** problem. The optimization after encoding may be two-level if we are interested in a two-level implementation, or multi-level, otherwise. This gives rise to state-assignment techniques for two-level [20, 58, 89, 84] and for multi-level implementations

[19, 49, 33, 46] respectively.

One parameter in most state-assignment algorithms is the number of bits used to encode the set of states of the FSM. If there are k states, the extreme values of this parameter are:

1. $\lceil \log_2(k) \rceil$: It corresponds to the **minimum-code length scheme.** Since each encoding bit corresponds to a flip-flop, this scheme uses the minimum number of flip-flops, and is, therefore, attractive. The way in which codes are assigned to the states affects the size of the combinational logic needed to compute the outputs and the next state functions.

2. k: The most well-known representative of this class is the **one-hot encoding scheme,** which uses one variable per state. This variable is 1 if and only if the machine is in that state. The number of flip-flops used is k, many more than the minimum-length scheme. The combinational logic resulting from the one-hot scheme is independent of the variables assigned to the states.

Before proceeding any further, we define the concept of a multi-valued function.

Definition 2.1.26 *A* **multi-valued function** *with n inputs is a mapping $\mathcal{F} : P_1 \times P_2 \times \cdots \times P_n \rightarrow B$, where $P_i = \{0, 1, \ldots, p_i - 1\}$, p_i being the number of values that i^{th} (multi-valued) variable may take on.*

An example of a multi-valued variable is S, the set of states of a controller. Analogous to the Boolean case, we can define the notion of a multi-valued product term and cover. Correspondingly we have the problem of determining a minimum-cost cover of a multi-valued function. This problem is referred to as **multi-valued minimization problem.**

A problem that is simpler than state-encoding is the one where just the inputs are symbolic. For example, assigning op-codes to the instructions of a processor so that the decoding logic is small, falls in this domain. This is known as the **input encoding** problem. If the objective is to minimize the number of product terms in a two-level implementation, the algorithm first given by De Micheli *et al.* [58] can be used. It views encoding as a two-phase process. In the first phase, a multi-valued minimized representation is obtained, along with a set of constraints on the codes of the values of the symbolic variables. In the second, an encoding that satisfies the constraints is determined. If satisfied, the constraints are guaranteed to produce an encoded binary representation of the same cardinality as the multiple-valued minimized representation. Details of the two phases are as follows:

1. *Constraint generation*: The symbolic description is translated into a multi-valued description using positional cube notation. For example, let S be a symbolic input variable that takes values in the set $\{S_1, S_2, \ldots, S_k\}$. Let x be a binary input, and y the only (binary) output. In positional cube notation (also called 1-hot notation), a column is introduced for each S_i. Let us assume that a possible *behavior* of the system is: if S takes value S_1 or S_2 and x is 1, then y is 1. This behavior can be written as:

x	S_1	S_2	S_3	...	S_{k-1}	S_k	y
1	1	0	0	...	0	0	1
1	0	1	0	...	0	0	1

A multi-valued logic minimization is applied on the resulting multi-valued description so that the number of product terms is minimized. The effect of multi-valued logic minimization is to group together symbols that are mapped by some input to the same output. The number of product terms in the minimized description is the same as the minimum number of product terms in any encoded final implementation, provided that the symbols in each product term in this minimized cover are assigned to one *face* (or *subcube*) of a binary cube, and no other symbol is on that face. These constraints are called the **face** or **input constraints**. For example, for the behavior just described,

x	S_1	S_2	S_3	...	S_{k-1}	S_k	y
1	1	1	0	...	0	0	1

is a product term in the minimum cover. This corresponds to a face constraint that says there should be a face with only S_1 and S_2. This face constraint can also be written as a set of **dichotomies** [89]: $(S_1S_2; S_3), \ldots, (S_1S_2; S_i), \ldots, (S_1S_2; S_k)$, which says that an encoding bit b_i must distinguish S_1 and S_2 from S_i for $3 \leq i \leq k$.

Also, each symbol should be assigned a different code. These are known as the **uniqueness constraints**, and are handled by adding extra dichotomies. For example, to ensure that the code of S_1 is distinct from other symbols, dichotomies $(S_1; S_2), (S_1; S_3), \ldots, (S_1; S_k)$ are added.

2. *Constraint satisfaction*: An encoding is determined that satisfies all the face and uniqueness constraints. De Micheli *et al.* proposed a satisfaction method based on the constraint matrix (which relates the face constraints to the symbolic values). Yang and Ciesielski [89] proposed an alternate scheme based on dichotomies and graph coloring for solving the constraints. It was later improved by Saldanha *et al.* [69].

2.2 Background

2.2.1 Logic Synthesis

Logic synthesis takes the circuit specification at the Boolean level and generates an implementation in terms of an interconnection of logic gates. Typically synthesis is done for an objective function, such as minimizing the cost of the design, satisfying the performance constraints, minimizing the power consumed, or making the implementation more testable. The cost of the design may be measured by the area occupied by the logic gates and the interconnect.

Since synthesis is a difficult process, it is typically separated into two phases: **technology-independent optimization phase** (also called logic optimization), followed by a **technology mapping phase** [12]. The optimization phase attempts to generate an optimum abstract representation of the circuit. For example, for area minimization, the most commonly used measure is the number of literals of the network in some factored form, which is the sum over all the internal nodes of the network of the number of factored form literals of each node. This cost measure has been found to have a good correlation with the cost of an implementation of the network in various technologies, e.g., standard cells or *CMOS* gate matrix. In the second phase, this optimized representation is mapped onto a pre-defined library of gates. misII [12] is a multi-level logic synthesis system that incorporates this two-phase approach.

Technology-Independent Optimization

The techniques used for optimization of Boolean networks are classified into two categories: restructuring and node minimization.

Restructuring operations massage the network and generate a structure that uses smaller number of literals (for area minimization), or has better delay characteristics (for performance optimization). The main idea in restructuring for area minimization is *to generate sub-functions that can be shared by many functions in the network,* thereby reducing the size of the network. To generate and use these sub-functions, the notion of **division** is a key one, and we review it next.

Definition 2.2.1 *An* **algebraic expression** *is a sum-of-products representation of a logic function that is minimal with respect to single cube containment (i.e., no cube contains another).*

For example, $ab + abc + cd$ is not an algebraic expression (since ab contains abc), but $ab + cd$ is.

Definition 2.2.2 *The* **product** *of two expressions f and g, fg, is a $\sum_{i,j} c_i d_j$ where $f = \{c_i\}, g = \{d_j\}$, made irredundant using containment operation (e.g., $ab + a = a$). The product is* **algebraic** *when f and g have disjoint supports. Otherwise, it is a* **Boolean product.**

For example, $(a+b)(c+d) = ac+ad+bc+bd$ is an algebraic product, whereas $(a+b)(a+c) = aa+ac+ab+bc = a+bc$ is a Boolean product.

Definition 2.2.3 *An operation OP is called* **division** *if, given two expressions f and p, it generates q (***quotient***) and r (***remainder***) such that $f = pq + r$, where p is called the* **divisor**. *If pq is an algebraic product, OP is called an* **algebraic division**. *Otherwise, pq is a Boolean product, and OP is called a* **Boolean division.**

Although Boolean division is more powerful, most of the logic optimization tools use algebraic division. The reasons are as follows.

1. The number of Boolean divisors is typically too many and it is computationally difficult to exploit them in optimization. It is much easier to choose divisors from the restricted algebraic domain.

2. Fast and efficient algorithms are known for algebraic manipulation [10], primarily because logic functions can then be treated as polynomials.

3. Although optimality is not guaranteed, the results obtained using algebraic techniques are encouraging [85, 12].

Weak division is a specific example of algebraic division that yields unique quotient and remainder.

Definition 2.2.4 *Given two algebraic expressions f and p, a division is called* **weak division** *if*

1. *it generates q and r such that pq is an algebraic product,*

2. *r has as few cubes as possible, and*

3. *$pq + r$ and f are the same expression (i.e., have the same set of cubes).*

f/p denotes the quotient q of weak dividing f by p. Given the expressions f and p, it can be shown that q and r generated by weak division are unique.

For example, if $f = abc + ade + kl$, weak-dividing f by a gives quotient $q = f/a = bc + de$, and remainder $r = kl$. Similarly, weak-dividing f by ab gives $f/(ab) = c$ and remainder $r = ade + kl$.

If n is the total number of product terms in f and p, an $O(n \log n)$ algorithm proposed by Brayton and McMullen [10] can be used to find the q and r for weak division. The next question is how to find good candidate divisors p, which serve as sub-expressions common to two or more expressions. The notion of *kernels of an algebraic expression* was introduced in [10] to address this question.

Definition 2.2.5 *An expression is* **cube-free** *if no cube divides the expression evenly.*

For example, $ab + c$ is cube-free, but $ab + ac$ is not cube-free since the cube a divides $ab + ac$ evenly. Since any cube divides itself evenly, a cube-free expression must have at least two cubes. So abc also is not cube-free.

Definition 2.2.6 *The* **primary divisors of an expression** f *are the set of expressions*

$$\mathcal{D}(f) = \{f/c \mid c \text{ is a cube}\}.$$

Definition 2.2.7 *The* **kernels of an expression** f *are the set of expressions*

$$\mathcal{K}(f) = \{g \mid g \in \mathcal{D}(f) \text{ and } g \text{ is cube-free}\}.$$

In other words, the kernels of an expression f are the cube-free primary divisors of f. Note that the division used here is weak division.

Definition 2.2.8 *A cube c used to obtain the kernel $k = f/c$ is called a* **co-kernel** *of k.*

Example 2.2.1

$$\begin{aligned} f &= adh + aeh + bdh + beh + cdh + ceh + g \\ &= (a + b + c)(d + e)h + g \end{aligned}$$

All the kernels and corresponding co-kernels of f as expressed above are shown below.

kernel	co-kernel
$a + b + c$	dh, eh
$d + e$	ah, bh, ch
$(a + b + c)(d + e)h + g$	1

Division is used in most of the restructuring operations. The restructuring operations include decomposition, extraction, substitution, and elimination.

1. **Decomposition** is the process of expressing a given logic function in terms of new, hopefully simpler functions. For example, if

$$f = abc + abd + a'c'd' + b'c'd',$$

one way to decompose f is as follows.

$$\begin{aligned} f &= xy + x'y' \\ x &= ab \\ y &= c + d. \end{aligned}$$

An alternate way of decomposing f is:

$$\begin{aligned} f &= w + x + y + z \\ w &= abc \\ x &= abd \\ y &= a'c'd' \\ z &= b'c'd \end{aligned}$$

2. **Extraction** is an operation closely related to decomposition and is applied to many functions. It is the process of identifying and creating some intermediate functions and variables, and re-expressing the original functions in terms of the original as well as the intermediate variables. This operation identifies common sub-expressions among different logic functions forming a network. New nodes are created, and each of the logic functions in the original network is simplified as a result of the introduction of the new nodes. The optimization problem then is to find a set of intermediate functions such that the resulting network is optimum in an appropriate sense. For example, extraction applied to the following functions

$$\begin{aligned} f &= (a + b)cd + e \\ g &= (a + b)e \\ h &= cde \end{aligned}$$

gives

$$\begin{aligned} f &= xy + e \\ g &= xe \\ h &= ye \\ x &= a + b \\ y &= cd. \end{aligned}$$

Note that new multiple-fanout nodes x and y have been created.

3. **Substitution** of a function g into a function f is the process of re-expressing f in terms of g. For example, if

$$\begin{aligned} g &= a + b \\ f &= a + bc, \end{aligned}$$

substitution of g into f gives

$$f = g(a + c).$$

Substitution can be looked at as a division operation, where we are dividing f by g.

4. **Collapsing** (also called **elimination**) a function g into f is the process of re-expressing f without explicitly using g. For example, if

$$\begin{aligned} f &= ga + g'b \\ g &= c + d, \end{aligned}$$

after collapsing, we get

$$f = ac + ad + c'd'b.$$

Collapsing is analogous to multiplication of polynomials, except that Boolean identities have to be used (e.g., $aa = a$ and not a^2). Note that collapsing is the inverse process of decomposition. Typically, collapsing is applied in two ways. First, nodes which do not save any literals in the network are collapsed. Second, to get away from a locally optimum structure, a node is collapsed even if it was saving a few literals.

In decomposition and extraction, kernels are used as divisors. Given a set of expressions corresponding to the node functions of the network, kernels are extracted for each function. If extracting a kernel k helps in reducing the cost (say, number of literals), a new node corresponding to k is added to the network, and is substituted into all the nodes that had k as a kernel.

Simplification (also called **node minimization**) attempts to reduce the complexity of a given network by using two-level minimization techniques on its node functions. However, if the node functions are treated as independent of each other, much optimization is potentially lost. In fact, the inputs of the Boolean function at a node n are related to each other by the nodes of the network that precede n and hence are not free to take any combination of values. In addition, for some values of the primary inputs of the network, the output of a node may not be observable at the primary outputs of the network (i.e., each primary output remains unchanged if the node value is toggled from 0 to 1 and vice versa). In both cases, the values of the inputs that can never occur at the input of the node function and the values of the primary inputs for which the outputs of the nodes are not observable at the primary outputs are **don't cares** for the two-level minimization of the node. The first kind of don't cares is called the **satisfiability don't care** (SDC) set, while the second is called the **observability don't care** (ODC) set.

An example of SDCs is as follows. If the node n of a network has associated with it the Boolean function $f(x, y)$ where $x = a + b$, $y = ab + c$, and a, b, c are the primary inputs of the network, then $x \neq (a + b) = x(a + b)' + x'(a + b)$ and $y \neq ab + c = y(ab + c)' + y'(ab + c)$ are SDCs. In other words, the SDCs represent combinations of variables of the Boolean network that can never occur because of the structure of the network.

Unfortunately, the SDCs and the ODCs may be very large and it may be impossible to compute them. In that case, a suitably chosen subset of SDCs and ODCs is used to optimize the two-level representation at the node [12]. Simplification has been proven to be effective for a wide variety of cases [72].

We must mention that simplification returns a prime cover, which is known to use the minimum local support. Since this is the cover handed to technology mapping, checking for m-feasibility of the function (which was defined in terms of the true support) in LUT mapping reduces to checking if the support of the prime cover is at most m, a much simpler problem.

The restructuring and simplification operations are applied on an unoptimized, raw network in some order until the cost function does not improve.

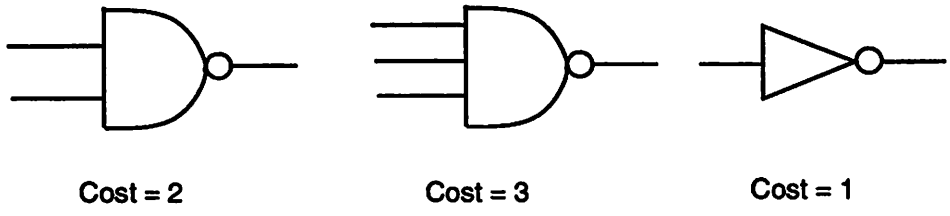


Figure 2.5: An example cell library

Technology Mapping

The network obtained after the optimization phase is implemented using a set of gates that form a **library**. Each gate has a cost that represents its area or delay. First, a set of **base functions** is chosen such as a two-input NAND gate and inverter. The optimized network is converted into a graph where each vertex is restricted to one of the base functions. This graph is called the **subject graph**, and this decomposition is called **technology decomposition**. The logic function for each library gate is likewise represented using the base functions. This generates **pattern graphs**. There may be more than one way to represent the gate function and so more than one pattern graph may result from a gate. A **cover of the subject graph** (not to be confused with the cover of a function) is a collection of pattern graphs such that every node of the subject graph is contained in one (or more) of the pattern graphs. The cover is further constrained so that each primary output is an output of some pattern graph, and each input required by a pattern graph is either a primary input or an output of some other pattern graph. For minimum area, the cost of a cover is the sum of area costs of the gates in the cover. The technology mapping problem may be viewed as the optimization problem of finding a minimum cost cover of the subject graph by choosing from the collection of pattern graphs for all gates in the library. This problem is hard - in fact NP-hard, though efficient heuristics exist. A commonly used heuristic divides the subject-graph into trees and covers the trees optimally by tree patterns in polynomial time using a dynamic programming approach. Typical examples are DAGON [41] and misII [18].

The only requirement imposed on the library is that it be **complete**, i.e., an arbitrary logic function should be realizable in terms of the gates in this set. Although two-input NAND gates and inverters form a complete set, it is desirable to put more gates in the library so as to get better results.

Example 2.2.2 *Figure 2.5 shows a simple cell (gate) library with three gates: a 2-input NAND gate with a cost of 2 units, a 3-input NAND gate with a cost of 3 units, and an inverter with a cost*

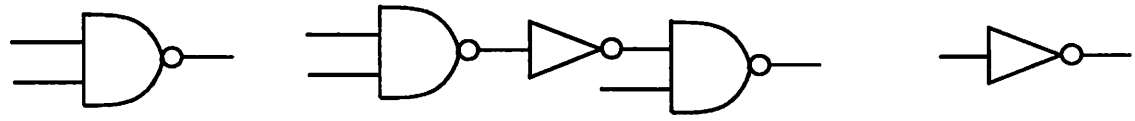


Figure 2.6: Pattern graphs for the gates in the cell library

of 1 unit. The pattern graphs for these gates using 2-input NAND gates and inverter are shown in Figure 2.6. Note that the 3-input NAND gate has a single pattern. Let us say we are interested in finding the minimum cost mapping of a 4-input NAND gate, shown in Figure 2.7. First, we derive its subject graph in terms of 2-input NAND gates and inverters. Many such subject graphs are possible, and we choose the one shown in Figure 2.7. We wish to find a minimum cost cover of this subject graph with the pattern graphs of Figure 2.6. Figure 2.8 shows two covers, (A) and (B), of the subject graph. The chosen patterns are shown as dotted rectangles. Note that (A) has a cost of 8 units: it uses three 2-input NAND gates and two inverters, whereas (B) has a cost of 6: it uses a 2-input NAND gate, a 3-input NAND gate, and an inverter, and is the best possible cover. To see that this indeed is the best cover, consider the root 2-input NAND gate of the subject graph. Two patterns can be rooted at it:

1. *A 2-input NAND gate: the best cover of the subject graph in this case is this 2-input NAND gate, along with the least-cost covers of each of the sub-trees rooted at the two inputs of the NAND gate. Considering all possible patterns rooted at these inputs recursively leads to the cover (A), which has a cost of 8 units.*

2. *A 3-input NAND gate: the best cover of the subject graph then corresponds to this 3-input NAND gate, along with the best possible covers of each of the sub-trees rooted at the three inputs of this NAND gate. Two of the inputs are primary inputs. So we recursively carry out the algorithm on the third NAND gate input. This finally leads to the cover (B) with a cost of 6.*

The cover with the minimum cost is picked, i.e., (B). This, in brief, is how the dynamic programming algorithm works on trees.

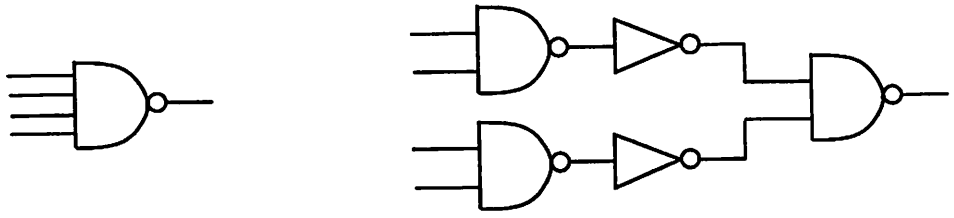
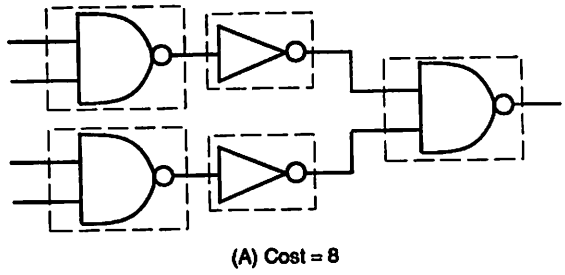


Figure 2.7: A 4-input NAND gate and one of its subject graphs



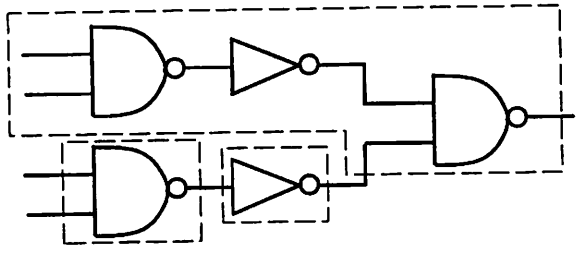


Figure 2.8: Technology mapping: two covers

2.3 Logic Synthesis for Field-Programmable Gate Arrays

Suppose we are interested in minimizing the number of FPGA blocks needed for a combinational circuit. Let us first ask if this problem is any different from the conventional logic synthesis problem, which first minimizes the number of literals (optimization) and then maps the resulting optimized network on to a library of cells (technology mapping). First let us consider logic optimization.

Example 2.3.1 *Assume that the target architecture is based on 5-LUTs. Consider two functions f_1 and f_2:*

$$f_1 = abcdeg,$$
$$f_2 = abc + b'de + a'e' + c'd'.$$

The representation of f_1 has 6 literals and that of f_2 10 literals. Both these representations are optimal, in that the misII *optimization script* (script.rugged) *[73] does not further improve the literal counts. Thus, f_1 is "simpler" than f_2. However, f_1 requires two 5-LUTs, whereas f_2 requires only one. This example shows that number of literals is not the best cost function for optimization for* LUT *architectures.*

Now consider technology mapping. Since traditional mappers use a library of gates, it is natural to ask if we can use a library for FPGA synthesis, and if so, how. For the LUT architectures, we can view an LUT as a collection of those gates that can be realized by it by possibly tying some of its input pins to constants 0 or 1. These gates can be put into the library and assigned a unit cost, indicating that the gate can be realized with one LUT. But the resultant library has 2^{2^m} gates, which is the total number of Boolean functions on m inputs. Even for $m = 4$, the library has over 64K gates. Currently, mappers cannot handle such a big library. If we allow renaming of the inputs (input permutations), many functions become equivalent to each other (called **P-equivalent**). For instance, $f_1(a,b,c) = a'b + ac$ and $f_2(a,b,c) = b'a + bc$ are P-equivalent, since $f_1(a,b,c) = f_2(b,a,c)$. 3984 non-P-equivalent functions are possible for $m = 4$ [24]. Only one function out of all the P-equivalent functions needs to be put in the library, thus reducing the size of the library considerably. However, the library is still large. But its size can be further reduced by noting that along with input permutations, an LUT also allows inversions at its inputs and outputs, i.e., if an m-LUT realizes $f(x_1, x_2, \ldots, x_m)$, it can also realize $\widetilde{f}(\widetilde{x_1}, \widetilde{x_2}, \ldots, \widetilde{x_m})$, where $\widetilde{x_i}$ denotes either x_i or x_i' (same for $\widetilde{f}$). This reduces the number of possibilities to 232 for $m = 4$ [24] (this notion of equivalence, which captures input permutations, input inversions, and output inversion, is called **NPN-equivalence**). However, the size of the library still grows as a double exponential, and for $m \geq 5$, this number is very large. Moreover, since each library function is represented in all possible tree configurations in terms of the base functions, the total number of configurations (or patterns) is much more. The complexity of the tree-based mapping is proportional to the total number of patterns. Since we are interested in techniques for general m-LUT architectures, a library-based approach is not viable.

Of course, one way to reduce the size of the library is to select a reasonable subset of the set of all m-feasible functions. However, a possible match may be missed, resulting in an

optimization loss. This necessitates exploring new methods for technology mapping that are not library-based, but instead, directly map onto the basic block.

For MUX-based architectures, e.g., *act1* and *act2*, the numbers of non-P-equivalent functions are 702 and 766. The corresponding numbers of pattern graphs are huge, so the library-based approach becomes impractical.

Library-based methods suffer from another problem. Most of these methods [41, 12] work on subject graphs and pattern graphs that are trees. If a library gate is complex (which can happen since, for instance, an m-LUT can implement any function of up to m inputs) and is represented only with trees, significant optimization may be lost.

This motivates a fresh look at the synthesis problem for FPGAs. First, we convince ourselves that this problem is indeed difficult. We say that a function f is realizable by k m-LUTs if there exists a single-output Boolean network η that is functionally equivalent to f and has at most k internal nodes, each having at most m fanins. Given a function f, we want to know the minimum number of m-LUTs needed to realize f. We prove that this is an NP-hard problem [30], given that we start from a sum-of-products representation. This implies that no polynomial time algorithm to solve this problem is known, justifying the heuristic approaches we will use in the rest of the thesis. The MINIMUM LUTS problem, stated as a decision problem, is as follows:

INSTANCE: Given a cover of a function f of n variables having c cubes, $m \geq 2$, and $k \geq 0$.
QUESTION: Is f realizable with k m-LUTs?

We show that MINIMUM LUTS is NP-hard. We first show that the following problem, ZERO LUTS, is NP-hard.

INSTANCE: Given a cover of a function f of n variables with c cubes, and $m \geq 2$.
QUESTION: Is f realizable with zero m-LUTs?

Proposition 2.3.1 *ZERO* LUT*S is NP-hard.*

Proof It suffices to Turing-reduce TAUTOLOGY to ZERO LUTS. The definition of Turing reduction permits a polynomial number of invocations of an oracle (subroutine) that solves ZERO LUTS (i.e., returns YES if f is realizable with zero LUTs, NO otherwise) in order to solve TAUTOLOGY.

First, note that a function f can be realized with zero LUTs if and only if it is identically 0, or identically 1, or identically equal to some input. Let $x_1, x_2, \ldots, x_n$ be the set of inputs of f. To answer if f is a tautology, proceed as follows.

1. If the SOP is just 0 (i.e., has no cubes), f is not a tautology.

2. Otherwise, call the oracle for ZERO LUTS. If it returns NO, f is not a tautology. Otherwise, there are two cases: either f is a tautology or it is identically equal to one of its inputs. To differentiate between the two possibilities, evaluate f on the input vector $(x_1, x_2, \ldots x_n) = (0, 0, \ldots, 0)$. If f evaluates to 0, it is not a tautology, otherwise it is. This is because if f were equal to one of the inputs, it cannot evaluate to 1 on the vector $(0, 0, \ldots, 0)$. So if f evaluates to 1 on this vector, it must be a tautology. Note that f can be evaluated on an input vector in time that is polynomial in n and c.

■

Corollary 2.3.2 *MINIMUM* LUT*S is NP-hard.*

Proof Restricting k to 0, we get an instance of ZERO LUTS, which is NP-hard, as proved in Proposition 2.3.1. ■

Note that the above result is valid for any m-LUT ($m \geq 2$). The above proof does not really use the fact that the basic block to be used is an LUT. Consequently, the same proof works for any basic block, e.g., a MUX-based block. In fact, it also works if f were to be mapped on to a library of gates, each gate having a positive cost, and the objective were to minimize the cost of the mapped solution.

Since we now know that the synthesis problem for FPGAs is difficult, our hope is to come up with techniques that do well in practice. At the same time, wherever possible, we should try to prove optimality of these techniques for special classes of functions (given the intractability of the general case). The next chapter describes mapping techniques for LUT architectures, and the one after that addresses the logic optimization problem.

Part I

Look-Up Table (LUT) Architectures

Chapter 3

Mapping Combinational Logic

3.1 Introduction

An important problem in synthesis is to minimize the cost of a design, where the cost is measured by the number of chips needed. This includes the routing considerations within and pin constraints of a chip. Since it is difficult to incorporate all these factors during synthesis, and only limited success has been achieved so far, for instance, in combining synthesis and routability [65], we approximate this cost by the number of blocks needed. Minimizing the number of blocks may be an overkill. However, leaving as many blocks unused as possible enables the designer to use the unused logic for improving the properties of the design. In addition, as we will show in Chapter 7, minimizing the number of blocks helps in reducing the circuit delay in a placed and routed implementation of the circuit. This is because the blocks can be placed close to each other, reducing the wiring delays considerably. However, as we saw in the last chapter, minimizing the number of blocks is a difficult problem.

Recall that an m-LUT can implement any Boolean function of up to m inputs.

Example 3.1.1 *A 5-*LUT *can implement, among so many other functions, f_1 or f_2 or f_3, where*

$$\begin{aligned} f_1 &= abcde, \\ f_2 &= abcde + a'b'c'd'e', \\ f_3 &= ab' + a'b. \end{aligned}$$

Since an LUT is an essential component of all the LUT-based architectures (e.g., Xilinx 3090), first we will target the synthesis algorithms for an m-LUT. In some sense, this is the easiest problem. Then we will extend the algorithms for the commercial LUT-based architectures, e.g., Xilinx 3090.

We assume that the Boolean network has already been optimized appropriately. The problem is *to map the optimized network on to the target* LUT *architecture (consisting of* m-LUT*s) such that the final implementation uses the minimum number of* LUT*s*. In order to solve this problem, first, m-infeasible functions should be made m-feasible. Each node function in the resulting network can then be realized by one m-LUT. However, it may be possible to reduce the number of m-LUTs if the nodes in the resulting network are *small*, i.e., more than one can be implemented by the same m-LUT. This is called **block count minimization** (BCM).

This chapter is organized as follows. To put everything in perspective, the history of LUT mapping is summarized in Section 3.2. Section 3.3 discusses techniques for converting an m-infeasible function into a set of m-feasible functions. This corresponds to the technology decomposition step of the conventional mappers. Section 3.4 describes BCM, which corresponds to the covering step of the conventional mappers. It turns out that for efficiency reasons, the two steps should not follow one another, but should be interleaved. Section 3.5 gives the details. Experimental results are presented in Section 3.6. All these techniques target an m-input, single output LUT. The commercial LUT-based architectures are more complex, and typically have two or more outputs. In Section 3.7, we apply our techniques to complex architectures. Finally, Section 3.8 evaluates the overall approach from different angles.

3.2 History

In 1989, when we first started looking at the synthesis problem for these architectures, no work had been reported in the literature.

3.2.1 Library-based Technology Mapping

As discussed in Section 2.3, a library-based approach is not viable for LUT-based architectures, simply because the size of the library and, therefore, the number of pattern graphs become huge.

3.2.2 mis-fpga

In 1990, we proposed mis-fpga [62], which is embedded in misII [12]. Specifically, it is the part of misII that pertains to the FPGA architectures - both LUT and MUX-based. This

includes algorithms, their implementation, and the commands.[1] For LUT architectures, mis-fpga has two phases: decomposition and BCM. In the first, classical Roth-Karp decomposition, kernel-extraction, and simple AND-OR decomposition are used to break each function into a set of feasible functions. In the BCM phase, the notion of an m-feasible supernode was introduced. All m-feasible supernodes of the network are generated using maximum flow technique repeatedly. A *binate covering* formulation is then used to solve for minimum number of supernodes that realize the network. A heuristic *partition* was also used to do greedy covering. It differed from the tree covering in that it can optimize across multiple fanout points. For Xilinx 3090, the problem of obtaining a maximum number of pairs of functions of a feasible network that can be placed on the same CLB was formulated as a maximum matching problem.

3.2.3 chortle

At the same time, Francis *et al.* proposed chortle [25], which, like conventional mappers, uses a dynamic programming paradigm. It applies a 2-input AND-OR decomposition, breaks the network into a forest of trees, and then covers each of them optimally by a set of patterns. These patterns are different from the ones used in standard technology mapping in that they just depend on the number of inputs of the function, and not the function itself - each node in the pattern is a 2-input *generic gate.* chortle suffered from a lack of optimization across tree boundaries, and also did not consider the possibility of architecture specific decomposition.

3.2.4 chortle-crf

In 1991, chortle-crf, an improved version of chortle, was proposed [26]. It introduced an important decomposition technique based on **bin-packing** [30]. It also used optimization across tree boundaries.

3.2.5 Xmap

In 1991, Karplus proposed Xmap [39], which builds an ITE for a function using cofactoring. Each non-terminal vertex in an ITE has at most three non-trivial inputs. So cofactoring can be looked at as a decomposition method. This ITE is covered greedily.

[1] misII is a tool for combinational logic synthesis, and is subsumed by sis [77, 78], which is a tool supporting logic synthesis of sequential circuits.

3.2.6 HYDRA

All the approaches proposed so far targeted an m-LUT, and targeted two-output blocks only in the post-processing phase. HYDRA [23] is a program specifically targeted to Xilinx 3090. Decomposition and BCM are performed keeping in mind the structure of the two-output block.

3.2.7 VISMAP

The approach used in VISMAP [87] is similar to mis-fpga. The main difference is that in the BCM phase, VISMAP does not generate all the supernodes, but a subset, and guarantees that no optimality is lost.

3.2.8 ASYL

In 1990-1991, Sicard *et al.* incorporated technology mapping for LUT architectures into ASYL synthesis system [80, 1]. A lexicographical factorization based optimization generates an ordering of variables, which is used to insert cut-points in the lexicographical trees. These cut-points determine the m-feasible solutions.

3.2.9 mis-fpga (new)

In 1991, we proposed mis-fpga (new) [63]. It had the following new features:

1. It used *cube-packing* - a decomposition technique first proposed in chortle-crf (but was called bin-packing), cofactoring, and Roth-Karp decomposition. It was shown that no single decomposition technique suffices.

2. It was found beneficial to apply decomposition and BCM on each node, and then use *partial collapse* to exploit the structure of the network.

3. It proposed the idea of making optimization specific to these architectures. To this end, kernel extraction was modified.

4. An exact BCM algorithm for the Xilinx 3090 architecture was given.

In the rest of the thesis, the term mis-fpga will be used to refer to the latest version of the system.

3.2.10 TechMap

In 1992, Sawkar and Thomas [74] proposed a mapping approach based on clique partitioning. Both area and delay optimizations were targeted.

3.3 Making an Infeasible Function Feasible

An m-infeasible node function f can be made m-feasible either by breaking it up into a set of m-feasible functions (this is called **decomposition**), or by exploiting its relationship with the rest of the network. First, we examine how various decomposition techniques, many of which had been already proposed for logic synthesis, can be applied to the LUT decomposition problem. These include functional decomposition (Section 3.3.1), cube-packing (Section 3.3.2), cofactoring (Section 3.3.3), kernel extraction (Section 3.3.4), and technology decomposition (Section 3.3.5). Then, in Section 3.3.6 we describe how to exploit the structure and functionality of the network to make f m-feasible.

3.3.1 Functional Decomposition

The first systematic study on decomposition was done by Ashenhurst [3]. He characterized the existence of a simple disjoint decomposition of a function. While being seminal, this work could not be used for functions with more than 10-15 inputs, since it required the construction of a *decomposition chart*, a modified form of the truth table for a function. Few years later, Roth and Karp proposed a technique [36] that does not require building a decomposition chart; instead, it uses a sum-of-products representation, which is, in general, more compact than a truth table. They also extended Ashenhurst's work by characterizing non-simple (or general) decompositions and used this characterization to determine the minimum-cost Boolean network using a library of primitive gates, each having some cost.

We first summarize the main ideas of these two studies, and then show how to apply them to the decomposition problem for LUT architectures.

Ashenhurst Decomposition

Ashenhurst [3] gave necessary and sufficient condition for the existence of a simple disjoint decomposition of a completely specified function f of n variables. A **simple disjoint**

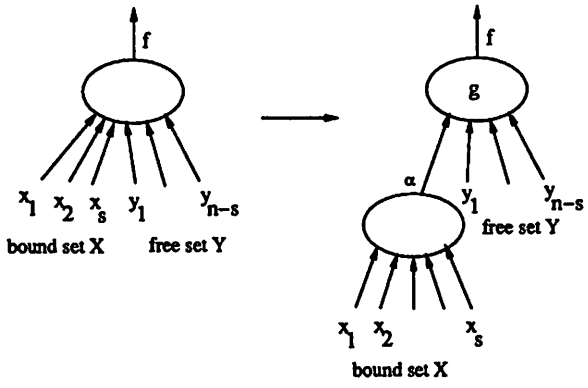


Figure 3.1: A simple disjoint decomposition

decomposition of f is of the form:

$$f(x_1, x_2, \ldots, x_s, y_1, \ldots, y_{n-s}) = g(\alpha(x_1, x_2, \ldots, x_s), y_1, \ldots, y_{n-s}) \tag{3.1}$$

where α is a single function, and $\{x_1, \ldots, x_s\} \cap \{y_1, \ldots, y_{n-s}\} = \phi$. In general, α could be a vector of functions, in which case the decomposition is **non-simple** (or **general**).

Let $X = \{x_1, x_2, \ldots, x_s\}$ and $Y = \{y_1, \ldots, y_{n-s}\}$. Then (3.1) can be rewritten as

$$f(X, Y) = g(\alpha(X), Y) \tag{3.2}$$

The representation (3.2) is called a **decomposition** of f; g is called the **image** of the decomposition. The set $X = \{x_1, x_2, \ldots, x_s\}$ is called the **bound set** and $Y = \{y_1, \ldots, y_{n-s}\}$ the **free set** (Figure 3.1). The necessary and sufficient condition for the existence of such a decomposition was given in terms of the **decomposition chart**[2] $D(X|Y)$ for f for the partition $X|Y$ (also written $\frac{X}{Y}$ or (X, Y)). A decomposition chart is a truth-table of f where vertices of $B^n = \{0,1\}^n$ are arranged in a matrix. The columns of the matrix correspond to the vertices of B^s, and its rows to the vertices of B^{n-s}. The entries in $D(X|Y)$ chart are the values that f takes for all possible combinations.

Example 3.3.1 *Let $f(a, b, c) = abc' + a'c + b'c$. The decomposition chart for f for the partition $ab|c$ is*

$\frac{ab}{c}$	00	01	10	11
0	0	0	0	1
1	1	1	1	0

[2]Ashenhurst called it **partition matrix.**

Note that if $s = 0, 1$, or n, a decomposition always exists. These cases correspond to **trivial decompositions**. All others, for which $1 < s < n$, are called **non-trivial**.

Ashenhurst proved the following fundamental result, which relates the existence of a decomposition to the number of distinct columns in the decomposition chart:

Theorem 3.3.1 (Ashenhurst's Fundamental Theorem of Decomposition) *The simple disjoint decomposition (3.2) exists if and only if the corresponding decomposition chart has at most two distinct column patterns.*

Stated differently, the decomposition (3.2) exists if and only if the **column multiplicity** (i.e., the number of distinct column patterns) of $D(X|Y)$ is at most 2.

We say that two vertices in B^s (i.e., $B^{|X|}$) are **compatible** (written $x_1 \sim x_2$) if they have the same column patterns in $D(X|Y)$, i.e., $f(x_1, y) = f(x_2, y)$ for all $y \in B^{|Y|}$. For an incompletely specified function, a don't care entry '$-$' cannot cause two columns to be incompatible. In other words, two columns c_i and c_j are compatible if for each row k, either $c_i(k) =$ '$-$', or $c_j(k) =$ '$-$', or $c_i(k) = c_j(k)$. For a completely specified function f, compatibility is an equivalence relation on the columns (i.e., $x_1 \sim x_1, x_1 \sim x_2 \Rightarrow x_2 \sim x_1$, and $x_1 \sim x_2 \wedge x_2 \sim x_3 \Rightarrow x_1 \sim x_3$ for all $x_1, x_2, x_3 \in B^{|X|}$), and the set of vertices that are mutually compatible (or equivalent) form an **equivalence class**. Hence *the column multiplicity of the decomposition chart is the number of equivalence classes*. In this subsection, we will consider only a completely specified function, and so use compatibility and equivalence interchangeably.

Given that the column multiplicity of $D(X|Y)$ is at most 2, how do we determine α and g? Since there are at most 2 equivalence classes, and a single α function for a simple decomposition, the vertices of one class are placed in the off-set of α, and of the other class in the on-set. g can then be determined by looking at each minterm in the on-set of f and replacing its bound-part (i.e., the literals corresponding to the variables in the bound set X) by either α or α', depending on whether the bound-part is in the class that was mapped to the on-set of α or the off-set. We illustrate the decomposition technique for the function f of Example 3.3.1.

Example 3.3.2 *$f = abc' + a'c + b'c$, and partition $(X|Y) = ab|c$. $D(ab|c)$ has two distinct column patterns, resulting in the equivalence classes $C_1(a,b) = \{00, 01, 10\}$ and $C_2(a,b) = \{11\}$. Let us assign C_1 to the off-set of α and C_2 to its on-set. Then $\alpha(a,b) = ab$. Since $f = abc' + a'c + b'c$, $g(\alpha, c) = \alpha c' + \alpha' c + \alpha' c = \alpha \oplus c$. The bound part of the first minterm abc' of f is ab, which yields $\alpha = 1$. So this minterm abc' generates $\alpha c'$ in g. Note that if C_1 was assigned to the on-set of α and C_2 to the off-set, the new $\tilde{\alpha}$ would be simply α', and the new $\tilde{g}(\alpha, c)$, $g(\alpha', c)$, which has same*

number of product terms as g. So irrespective of how we encode C_1 and C_2, the resulting g functions have the same complexity. However, things are different if the decomposition is not simple.

Roth-Karp Decomposition

Not every function has a non-trivial simple disjoint decomposition.

Example 3.3.3 *Consider $f(a,b,c) = a'bc + ab'c + abc'$. For a non-trivial decomposition, only $|X| = 2$ needs to be considered. For the input partition $ab|c$, the decomposition chart is*

$\frac{ab}{c}$	00	01	10	11
0	0	0	0	1
1	0	1	1	0

It has 3 distinct column patterns and so a simple disjoint decomposition does not exist for this partition. Since f is totally symmetric, it does not have a non-trivial simple disjoint decomposition.

Roth and Karp [36] extended the decomposition theory of Ashenhurst by characterizing a general (non-simple) disjoint decomposition, which is of the following form:

$$f(X,Y) = g(\alpha_1(X), \alpha_2(X), \ldots, \alpha_t(X), Y) = g(\vec{\alpha}(X), Y), \tag{3.3}$$

where $\vec{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_t)$. The theory of Roth and Karp applies for an incompletely specified function f. We present a summary of their formulation. Let $\hat{X}, \hat{Y}, \hat{Z}$, and $\widehat{W}$ be arbitrary finite sets, and $\hat{E}$ be a subset of $\hat{X} \times \hat{Y}$. Given a function $f : \hat{E} \to \hat{Z}$, we examine the following:

(i) Given $\alpha : \hat{X} \to \widehat{W}$, does there exist a function $g : \widehat{W} \times \hat{Y} \to \hat{Z}$, such that for all $(x,y) \in \hat{E}$,

$$f(x,y) = g(\alpha(x), y)? \tag{3.4}$$

(ii) Under what conditions do there exist functions $\alpha : \hat{X} \to \widehat{W}$, and $g : \widehat{W} \times \hat{Y} \to \hat{Z}$, such that (3.4) holds?

The answer to (i) may be formulated in terms of a relation of compatibility between elements of $\hat{X}$. Let $x_1, x_2 \in \hat{X}$. Then x_1 and x_2 are **compatible** with respect to f (denoted by $x_1 \sim x_2$) if, for all $y \in \hat{Y}$ such that $(x_1,y),(x_2,y) \in \hat{E}$, $f(x_1,y) = f(x_2,y)$; otherwise, x_1 and x_2 are **incompatible** (denoted by $x_1 \not\sim x_2$). The following proposition from [36] answers (i).

Proposition 3.3.2 (Roth and Karp) *Given f and α, there exists g such that (3.4) holds if and only if, for all $x_1, x_2 \in \hat{X}, \alpha(x_1) = \alpha(x_2) \Rightarrow x_1 \sim x_2$, or equivalently, $x_1 \not\sim x_2 \Rightarrow \alpha(x_1) \neq \alpha(x_2)$.*

An answer to (ii) may be given now. The crucial consideration is the number of elements in $\widehat{W}$; for if $\widehat{W}$ has too few elements, it may not be possible to produce a function α such that all elements of $\hat{X}$ mapping into the same element of $\widehat{W}$ are compatible. These considerations are made precise in the next proposition, which in fact follows from Proposition 3.3.2.

Proposition 3.3.3 (Roth and Karp) *If k is the least integer such that $\hat{X}$ may be partitioned into k classes of mutually compatible elements, then there exist α and g such that (3.4) holds if and only if $\widehat{W}$ has at least k elements.*

In order to apply Proposition 3.3.3, the only missing link is the number k. First consider the case when f is completely specified (i.e., is defined at all points in $\hat{X} \times \hat{Y}$). Then $x_1 \sim x_2$ if and only if for all $y \in \hat{Y}$, $f(x_1, y) = f(x_2, y)$. Compatibility is then an equivalence relation, and k is simply the number of equivalence classes. If f is incompletely specified, i.e., it is undefined for some elements of $\hat{X} \times \hat{Y}$, compatibility is no longer an equivalence relation, and the determination of a minimum cover of $\hat{X}$ by sets of mutually compatible elements is nontrivial.

Note that the formulation of Roth and Karp is in terms of arbitrary sets $\hat{X}, \hat{Y}, \hat{Z}$, etc., and functions on these sets. It can be restricted to the Boolean domain by substituting $\hat{X} = B^{|X|}$, $\hat{Y} = B^{|Y|}$, $\hat{Z} = B$, etc. The rest of the section uses Boolean domain, and all references to Propositions 3.3.2 and 3.3.3 should be suitably interpreted.

Let the given Boolean function f be represented by on-set cover $C_1 = \{l_1, l_2, \ldots, l_p\}$ and off-set cover $C_0 = \{m_1, m_2, \ldots, m_q\}$, where $l_1, l_2, \ldots, l_p, m_1, m_2, \ldots, m_q$ are cubes. Let X be the bound set and Y the free set. If Propositions 3.3.2 and 3.3.3 are to be used for the detection of decompositions, it is necessary to determine the specifications of X that are compatible. If X and Y are disjoint, any cube of C_1 or C_0 can be divided into an "X-part" and a "Y-part." For example, consider the cube β

a	b	c	d
1	0	1	2

With $X = \{a, c\}$ and $Y = \{b, d\}$, the X-part of β is 1 1 and the Y-part is 0 2.

The covers C_1 and C_0 can then be written as $C_1 = \{(l_X, l_Y)\}$ and $C_0 = \{(m_X, m_Y)\}$. Then the following lemma [36] holds:

Lemma 3.3.4 (Roth and Karp) *Given $v_1 \in B^{|X|}$ and $v_0 \in B^{|X|}$, $v_0 \not\sim v_1$ if and only if there are cubes $(l_X, l_Y) \in C_1$ and $(m_X, m_Y) \in C_0$ such that l_Y intersects m_Y, and either l_X covers v_1 and m_X covers v_0, or l_X covers v_0 and m_X covers v_1.*

In other words, v_0 and v_1 are incompatible if and only if for some $y \in B^{|Y|}$, the minterms (v_0, y) and (v_1, y) belong to different sets - on and off. This lemma enables the use of the covers of the on-set and the off-set for determining the compatibilities instead of using the truth table. In general, Lemma 3.3.4 can be applied in two ways:

1. In conjunction with Proposition 3.3.2, it can be used to determine, given f, X, Y, and $\{\alpha_1, \alpha_2, \ldots, \alpha_t\}$, whether there is a decomposition of the form (3.3). This is done simply by determining which incompatibilities exist (using Lemma 3.3.4) and ascertaining whether any of them violate the conditions of Proposition 3.3.2.

2. In conjunction with Proposition 3.3.3, it can be used to determine, given f, X, Y, and t, whether there exist functions $\alpha_1, \alpha_2, \ldots, \alpha_t$ such that (3.3) is satisfied. In the language of Proposition 3.3.3, $\widehat{W}$, the range of $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_t)$, has at most 2^t elements, and a decomposition exists if and only if $k \leq 2^t$, where k is the minimum number of classes of mutually compatible elements into which the domain of α can be partitioned.

Determining $\vec{\alpha}$ and g: an encoding problem

Roth and Karp give conditions for the existence of $\vec{\alpha}$ functions, but do not give a method for computing them.[3] This is because they assume that a *library $\mathcal{L}$ of primitive elements* is available, from which $\vec{\alpha}$ functions are chosen. Given a choice of $\vec{\alpha}$ functions, Proposition 3.3.2 may be used to determine if a valid decomposition exists. If it does not exist, then this particular choice $\vec{\alpha}$ of primitive elements is discarded, and the next one is tried. Otherwise, a valid decomposition exists, and then g is determined as follows. Each minterm (x, y) in the on-set of f, where x is the bound part and y is the free-part, maps into a minterm $(\widehat{\alpha_1}\widehat{\alpha_2}\ldots\widehat{\alpha_t}, y)$ in the on-set of g, where

$$\widehat{\alpha_j} = \begin{cases} \alpha_j & \text{if } \alpha_j(x) = 1 \\ \alpha_j' & \text{if } \alpha_j(x) = 0. \end{cases} \tag{3.5}$$

The entire procedure is repeated on g until it becomes equal to some primitive element.

In general, $\vec{\alpha}$ functions are not known *a priori*. For instance, this is the case when decomposition is performed during the technology-independent optimization phase, because the

[3] We believe that they knew how to find these functions, but not how to find "simple" $\vec{\alpha}$ functions.

technology library of primitive elements is not considered. In fact, there are many possible choices for $\vec{\alpha}$ functions that correspond to a valid decomposition.

Example 3.3.4 *Consider the function f of Example 3.3.3.*

$$f = a'bc + ab'c + abc'$$

As it was shown earlier, the decomposition chart for f for the partition $ab|c$ has 3 distinct column patterns (or equivalence classes). This means that $t \geq \lceil \log_2(3) \rceil = 2$. Let us choose $t = 2$. Then there are many choices of $\vec{\alpha} = (\alpha_1, \alpha_2)$, and two of them are shown here.

$$\begin{aligned} 1. \quad \alpha_1(a,b) &= ab \\ \alpha_2(a,b) &= a'b + ab' \\ g(\alpha_1,\alpha_2,c) &= \alpha_1\alpha_2{}'c' + \alpha_1{}'\alpha_2 c \end{aligned}$$

$$\begin{aligned} 2. \quad \alpha_1(a,b) &= a' + b' \\ \alpha_2(a,b) &= a'b' \\ g(\alpha_1,\alpha_2,c) &= \alpha_1{}'\alpha_2{}'c' + \alpha_1\alpha_2{}'c \\ &= \alpha_2{}'(\alpha_1{}'c' + \alpha_1 c) \end{aligned}$$

The second choice leads to a simpler g function and fewer overall literals.

Given that $\hat{X}$ may be partitioned into k classes of mutually compatible elements, and that $t \geq \lceil \log_2(k) \rceil$, each of the k compatibility classes may be assigned a unique binary code of length t, and there are many ways of doing this. Each such assignment leads to different $\vec{\alpha}$ functions. We wish to obtain that set of $\vec{\alpha}$ functions that is *simple* and makes the resulting function g simple as well. The measure of simplicity is the size of the functions using an appropriate cost function. For instance, in the two-level synthesis paradigm, a good cost function is the number of product terms, whereas in the multi-level paradigm, it is the number of literals in the factored form. The general problem can then be stated as follows:

Problem 3.3.1 *Given a function $f(X,Y)$, determine sub-functions $\vec{\alpha}(X)$ and $g(\vec{\alpha},Y)$ satisfying (3.3) such that an objective function on the sizes of $\vec{\alpha}$ and g is minimized.*

This problem has not been addressed in the past to the best of our knowledge. We present an encoding-based formulation for solving this problem exactly given a standard objective function.

It seems intuitive to extend Ashenhurst's method for obtaining the $\vec{\alpha}$ functions. Ashenhurst placed the minterms of one equivalence class in the on-set of α and of the other in the off-set. In other words, one equivalence class gets the code $\alpha = 1$ and the other, $\alpha = 0$. For more than two equivalence classes, we can do likewise, i.e., assign unique $\vec{\alpha}$-codes to equivalence classes. This leads to the following algorithm:

1. Obtain a minimum cardinality partition $\mathcal{P}$ of the space $B^{|X|}$ into k compatible classes. This means that no two classes C_i and C_j of $\mathcal{P}$ can be combined into a single class $C_i \cup C_j$ such that all minterms of $C_i \cup C_j$ are mutually compatible. This means that given any two classes C_i and C_j in $\mathcal{P}$, there exist $v_i \in C_i$ and $v_j \in C_j$ such that $v_i \not\sim v_j$.

2. Then assign codes to the compatibility classes of $\mathcal{P}$. Since there is at least one pair of incompatible minterms for each pair of classes, it follows from Proposition 3.3.2 that each compatibility class must be assigned a unique code. This implies that all the minterms in a compatibility class are assigned the same code. We will discuss shortly how to assign codes to obtain simple $\vec{\alpha}$ and g functions.

Example 3.3.5 *For $f(a,b,c) = a'bc + ab'c + abc'$, the decomposition chart for the partition $ab|c$ was shown in Example 3.3.3. It has 3 distinct column patterns, i.e., $k = 3$. Let us choose $t = 2$. Suppose we assign the following codes:*

class	$\alpha_1\alpha_2$
$C_1 = \{a'b'\}$	00
$C_2 = \{a'b, ab'\}$	01
$C_3 = \{ab\}$	10

This results in

$$\begin{aligned} \alpha_1(a,b) &= ab \\ \alpha_2(a,b) &= a'b + ab' \\ g(\alpha_1,\alpha_2,c) &= \alpha_1\alpha_2{}'c' + \alpha_1{}'\alpha_2 c \end{aligned}$$

This is the approach taken in every work (we are aware of) that uses functional decomposition, e.g., [62, 43]. However, this is not the most general formulation of the problem. To see why, let us re-examine Proposition 3.3.2, which gives necessary and sufficient conditions for the existence of the decomposition. It only constrains two minterms (in $B^{|X|}$ space) that are in different equivalence classes to have different values of $\vec{\alpha}$ functions. It says nothing about the minterms in the same

equivalence class. In fact, there is no restriction on the $\vec{\alpha}$ values that these minterms may take: $\vec{\alpha}$ may evaluate same or differently on these minterms.

To obtain the general formulation, let us examine the problem from a slightly different angle. In Figure 3.2 is shown a function $f(X, Y)$ that is to be decomposed with the bound set X and the free set Y. After decomposition, the vertices in $B^{|X|}$ are *mapped into* vertices in B^t - the space corresponding to the $\vec{\alpha}$ functions. This is shown in Figure 3.3. This mapping can be thought of as an encoding. Assume a symbolic variable $\mathcal{X}$. Imagine that each vertex x in $B^{|X|}$ corresponds to a symbolic value of $\mathcal{X}$, and is to be assigned an $\vec{\alpha}$-code in B^t. This assignment must satisfy the following constraint: if $x_1, x_2 \in B^{|X|}$ and $x_1 \not\sim x_2$, they must be assigned different $\vec{\alpha}$-codes - this follows from Proposition 3.3.2. Otherwise, we have freedom in assigning them different or same codes. Hence, instead of assigning codes to classes, the most general formulation assigns codes to the minterms in the $B^{|X|}$ space.

The problem of determining simple $\vec{\alpha}$ and g can be represented as an input-output encoding (or state-encoding) problem. Intuitively, this is because the $\vec{\alpha}$ functions created after encoding are both inputs and outputs: they are inputs to g and outputs of the square block of Figure 3.3. Minimizing the objective for $\vec{\alpha}$ functions imposes output constraints, whereas minimizing it for g imposes input constraints.

There is, however, one main difference between the standard input-output encoding problem and the encoding problem that we have. Typically input-output encoding requires that each symbolic value be assigned a *distinct* code (e.g., in state-encoding), whereas in our encoding problem some symbols of $\mathcal{X}$ may be assigned the same code. This can be handled by a simple modification to the encoding algorithm. Recall from Section 2.1.3 that an encoding algorithm, in particular the one based on dichotomies, ensures that the the codes are distinct by explicitly adding a dichotomy $(S_i; S_j)$ for each symbol-pair $\{S_i, S_j\}$. This guarantees that the code of S_i is different from that of S_j in at least one bit. In our problem, let x_i and x_j be two symbolic values of $\mathcal{X}$. If $x_i \not\sim x_j$, add a dichotomy $(x_i; x_j)$. Otherwise, no such dichotomy is added. This provides additional flexibility to the encoding algorithm: it may assign the same code to two or more compatible symbols if the resulting $\vec{\alpha}$ and g are simpler.

The encoding algorithm has to encode all the $2^{|X|}$ symbolic values of $\mathcal{X}$. If $|X|$ is large, the problem becomes computationally difficult. We can then use the approximate method of assigning codes to equivalence classes, as described earlier.

Note that t is determined by the encoding algorithm. It is the number of bits used by the algorithm to encode the vertices in $B^{|X|}$, or the equivalence classes if the approximate method is

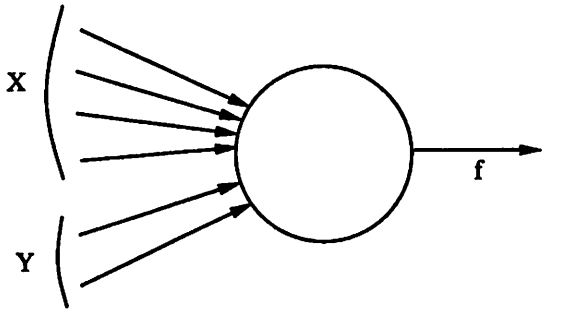


Figure 3.2: Function f to be decomposed with the bound set X and free set Y

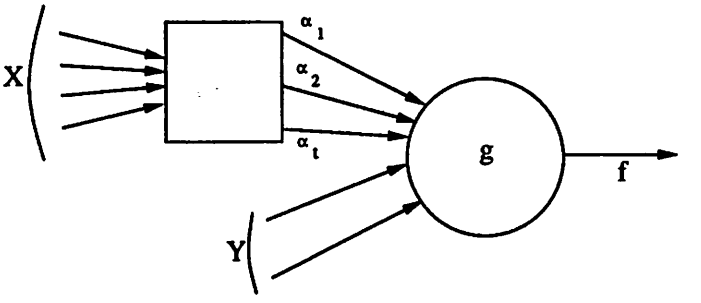


Figure 3.3: A general decomposition of f

being used. Once the codes are known, the $\vec{\alpha}$ functions can be easily computed. Then g can be determined using the procedure described in the last section. The unused codes can be used as don't cares to simplify g.

Application to LUT architectures We have shown that for a given partition, the general decomposition problem is an input-output encoding problem. However, for LUT architectures, we are interested in a particular kind of decomposition: namely, where the bound set X is restricted to have at most m variables, i.e., $|X| \leq m$. Since an LUT can implement any function of up to m inputs, and $\vec{\alpha}$ functions are functions of X, we do not care how large the representation of the functions $\vec{\alpha}$ is. The only concern from the output encoding part is the number of bits used to encode the classes,

since bit b_i corresponds to the function α_i. Then each extra bit implies an extra LUT, we would like to minimize the number of bits. So we use the minimum number of bits, i.e., $t = \lceil log_2 k \rceil$. Then, the contribution by the $\vec{\alpha}$ functions to the objective function disappears. This removes the output encoding part of the formulation, thereby reducing the problem simply to one of input encoding.

Since LUTs impose input constraints, it is tempting to consider minimizing the support of the function g as the objective function in the encoding formulation. However, if the code-length is always chosen to be the minimum possible, the support of g is already determined, and the encoding of $\vec{\alpha}$ functions do not make any difference. Hence, this objective function is not meaningful.

Applying functional decomposition to LUT architectures

It is now straightforward to translate the above discussion into an algorithm for decomposition for LUT architectures. This is shown in Figure 3.4. The approximate algorithm, which encodes classes, is shown in Figure 3.5. Given an m-infeasible function f, a partition (X, Y) of the support of f is chosen such that $|X| \leq m$. This guarantees that the corresponding α functions are m-feasible. Lemma 3.3.4 is used to determine incompatibilities between minterms in X. Then k, the minimum number of mutually compatible classes, is determined. If $k > 2^{m-1}$, the partition is rejected because of the following reason. This partition will result in $t \geq m$. Then g will have at least as many inputs as f. If the algorithm is to terminate, it should create a function g with strictly fewer number of inputs than f. Otherwise (i.e., if $k \leq 2^{m-1}$), an encoding step is performed to determine $\vec{\alpha}$. Subsequently, g is determined. If g is m-infeasible, it is recursively decomposed.

We illustrate the approximate procedure with the following example.

Example 3.3.6

$$f(a,b,c,d,e) = ab' + ac' + ad + ae + a'e'$$

Let $m = 4$. Let us fix the bound set X to $\{a,b,c,d\}$. Then $Y = \{e\}$. Although we do not show the decomposition chart (since it is big), it has three equivalence classes C_0, C_1, and C_2. Let the corresponding symbolic representation for the on-set of g be:

e	*class*	g
1	C_0	1
1	C_1	1
0	C_2	1
0	C_0	1

```
/* η is a network */
/* m is the number of inputs to the LUT */
functional_decomposition_for_LUT(η, m)
{
     while (nodes with support > m exist in η) do {
          n = get_an_m-infeasible_node(η);
          (X,Y) = get_input_partition(n);
          classes = determine_compatibility_classes(n, X, Y);
          if (# (classes) > 2^(m-1)) {
               call an alternate decomposition(n);
               continue;
          };
          codes = encode(n, X);
          α⃗ = determine_α⃗(codes);
          g = compute_g(n, codes);
          g = simplify_g_using_DC(g, α⃗, codes);
          add α⃗ nodes to η;
          replace n by g
     }
}
```

Figure 3.4: Functional decomposition for LUT architectures

```
/* η is a network */
/* m is the number of inputs to the LUT */
approximate_functional_decomposition_for_LUT(η, m)
{
     while (nodes with support > m exist in η) do {
          n = get_an_m-infeasible_node(η);
          (X,Y) = get_input_partition(n);
          classes = determine_compatibility_classes(n, X, Y);
          if (# (classes) > 2^(m-1)) {
               call an alternate decomposition(n);
               continue;
          };
          codes = encode(n, classes);
          α⃗ = determine_α⃗(classes, codes, X);
          g = compute_g(n, classes, codes, X);
          g = simplify_g_using_DC(g, classes, codes);
          add α⃗ nodes to η;
          replace n by g
     }
}
```

Figure 3.5: Approximate method for decomposition for LUT architectures

Let us assume that we are minimizing the number of product terms in g. Then after a multi-valued minimization [11], we get the following cover:

e	C_0	C_1	C_2	g
1	1	1	0	1
0	1	0	1	1

This corresponds to the following face constraints:

C_0	C_1	C_2
1	1	0
1	0	1

To these, uniqueness constraints are added. These constraints are handed over to the constraint satisfier [69]. The following codes are generated:

class	$\alpha_1\alpha_2$
C_0	00
C_1	10
C_2	01

Note that C_0 and C_1 are on a face, namely $\alpha_2 = 0$. Similarly, C_0 and C_2 are on the face $\alpha_1 = 0$. Let α_1 and α_2 be the encoding variables used. Then it can be seen from the minimized multi-valued cover that

$$\begin{aligned} g &= e'(C_0 + C_2) + e(C_0 + C_1) \\ \Rightarrow g &= e'\alpha_1{}' + e\alpha_2{}' \end{aligned}$$

Also, it turns out that C_0, C_1 and C_2 are such that

$$\begin{aligned} \alpha_1 &= abcd' \\ \alpha_2 &= a' \end{aligned}$$

This simplifies to

$$\begin{aligned} g &= e'\alpha_1{}' + ea \\ \alpha_1 &= abcd' \end{aligned}$$

Had we done a dumb *encoding of the equivalence classes, as is the case in [62], we would have obtained the following decomposition,*

$$\begin{aligned} g &= \alpha_1\alpha_2{}'e + \alpha_1{}'\alpha_2 + \alpha_1{}'e' \\ \alpha_1 &= abcd' \\ \alpha_2 &= ab' + ac' + ad, \end{aligned}$$

which uses one more function and many more literals than the previous one. This shows that the choice of encoding does make a difference in the resultant implementation.

Note that unless an alternate decomposition is used, this procedure is not complete, i.e., it does not always guarantee that finally an m-feasible network will result. This happens when for the chosen partition, $k > 2^{m-1}$.

Example 3.3.7 *Consider the function of Example 3.3.3, $f(a,b,c) = a'bc + ab'c + abc'$. Let m be 2. To generate non-trivial decompositions, it suffices to consider bound sets with two elements. As shown earlier, the decomposition chart for f for the partition $(ab|c)$ has a column multiplicity of 3, which is greater than $2^{m-1} = 2^{2-1} = 2$. So our procedure will throw away this partition. Since the function is totally symmetric, any partition with the bound set having two elements will result in the same decomposition chart and hence would have column multiplicity of 3. So our procedure will fail to generate a valid disjoint decomposition for f.*

In such a case, an alternate decomposition strategy is used. For example, a non-disjoint decomposition always exists, and, therefore, an m-feasible implementation can always be obtained for any m-infeasible function. Two such techniques, cube-packing and cofactoring, will be described in Sections 3.3.2 and 3.3.3 respectively.

Choosing a partition To obtain the best possible decomposition, all partitions (X, Y) should be explored. This may not be computationally viable, since there are exponentially many choices. Our current implementation includes two options:

1. Pick an arbitrary bound set X. Although fast, this strategy may miss good partitions, as the following example shows.

   Example 3.3.8 *Consider the function*

   $$f = abc + degh + ij$$

   Let $m = 5$. To make this function 5-feasible, the best partition is $(abcij|degh)$, resulting in the following decomposition:

   $$\begin{aligned} f &= p + degh \\ p &= abc + ij \end{aligned}$$

However, if we pick a partition, say $(abcde|ghij)$*, we get the following:*

$$\begin{aligned} f &= qr + q'r's + rs' \\ p &= abc \\ q &= a'de + b'de + c'de \\ r &= p'gh + p'ij \\ s &= p + ghi' + ghj' \end{aligned}$$

This decomposition, with 5 sub-functions, is far off from the best one.

2. Go through all the partitions and pick the best one. Ideally, the best partition is the one that results in the fewest feasible functions. This is not easy to find if g is infeasible and has to be recursively decomposed: all partitions have to be tried for g as well. So, we approximate the best partition to be the one that gives the minimum number of compatible classes.

   If $|X \cup Y|$ is large, generating all partitions and computing the compatibility classes becomes computationally infeasible. We present a simple result to prune the search. It holds for completely specified functions. The idea behind it is that the number of equivalence classes for the partition $(X|aY)$ (which is the same as $(X|\{a\} \cup Y)$) is related to that for $(aX|Y)$. If the number of equivalence classes for the partition $(aX|Y)$ is known, it may be possible to tell, without computing the equivalence classes for $(X|aY)$, whether $(X|aY)$ will generate fewer equivalence classes than the minimum seen thus far.

   Let $v \in B^{|X|}$, and av and $a'v$ be the corresponding vertices in the extended space $B^{|X|+1}$ with $a = 1$ and $a = 0$ respectively.

Proposition 3.3.5 *Let* $\rho(aX|Y)$ *and* $\rho(X|aY)$ *be the column multiplicities of the decomposition charts* $D(aX|Y)$ *and* $D(X|aY)$ *respectively. Then*

$$\rho(aX|Y)/2 \leq \rho(X|aY) \leq \rho(aX|Y)^2. \tag{3.6}$$

Moreover, these bounds are tight, i.e., there exist decomposition charts on which these bounds hold as equalities.

Proof We prove the inequalities one by one.

- $\rho(aX|Y)/2 \leq \rho(X|aY)$: Consider the decomposition chart $D(X|aY)$. Without loss of generality, each column v of the chart can be divided into two halves, with the top half corresponding to the half-space $a = 1$, and the bottom half corresponding to $a = 0$. When a is moved from the free set to the bound set X, the column v of $D(X|aY)$ splits into two columns, av and $a'v$, where av is the top-half of v and $a'v$ the bottom-half. If there are $\rho(X|aY)$ distinct column patterns in $D(X|aY)$, there cannot be more than $2\rho(X|aY)$ distinct column patterns in $D(aX|Y)$.
- $\rho(X|aY) \leq \rho(aX|Y)^2$: Consider $D(aX|Y)$. Let $c_1, c_2, \ldots, c_{\rho(aX|Y)}$ be its distinct column patterns. When a is moved from the bound set aX and added to the free set Y, the column $a'v$ of $D(aX|Y)$ aligns itself below the column av and results in the column v in the new chart $D(X|aY)$. A column of $D(X|aY)$ is then of the form c_i concatenated with c_j where c_i and c_j are columns of $D(aX|Y)$. The worst case is when each distinct column pattern of $D(aX|Y)$ gets aligned with any other column pattern, thus resulting in $\rho(aX|Y)^2$ distinct column patterns in $D(X|aY)$.

It is easy to construct examples of the decomposition charts where the bounds hold as equalities. ■

How can this result be applied? Assume we are generating partitions one by one, computing the number of equivalence classes, and saving the best partition seen so far. Let there be 5 equivalence classes in the best partition seen so far. Let $D(aX|Y)$ have 20 equivalence classes. There is no need to generate the chart for $D(X|aY)$, since it has at least 10 equivalence classes (using Proposition 3.3.5). Similarly, if $D(\tilde{X}|b\tilde{Y})$ has 30 equivalence classes, $D(b\tilde{X}|\tilde{Y})$ has at least $\lceil\sqrt{30}\rceil = 6$ classes, and need not be generated.

To conclude, it remains an open problem to find a good partition quickly. Hwang *et al.* [35] did some work on this using a partitioning algorithm similar to the one proposed by Kernighan and Lin [40]. The cost function is the number of patterns corresponding to this partition. They first generate an input partition randomly, and then move the input variables across partitions and recompute the cost function. The strategy of accepting a partition is the same as in the Kernighan-Lin algorithm. Hwang *et al.* showed that this technique generates good results. Out of 14 benchmarks, it computes optimum partitions for 11. But this conclusion has to be taken with a pinch of salt. Their benchmark set consists mainly of symmetric and arithmetic circuits. For symmetric circuits, any bound set of a given cardinality is the same, and arithmetic circuits exhibit group-symmetry. For a definitive answer, general benchmark circuits should be chosen and studied.

3.3.2 Cube-packing

Before describing cube-packing, it is useful to define the following two notions.

Definition 3.3.1 *A* **supercube** *of a cube c is a cube that contains c.*

For example, $ab'd$, ade, $ab'd$ and d are some supercubes of the cube $ab'de$.

Definition 3.3.2 *A* **sub-function** *of a function $f = c_1 + c_2 + \ldots + c_n$ is a function whose cover is a subset of the set of cubes $\{c_1, c_2, \ldots, c_n\}$.*

For example, if $f = abc + deg'h + kl'$, then abc, $abc + deg'h$, $abc + kl'$ are some of the sub-functions of f. Note that in the rest of the thesis, the term sub-function of a function f is used somewhat loosely to mean a function that is derived from f in some way. For this subsection, a sub-function is as defined in the last definition.

Cube-packing as a method of decomposition for LUT architectures was first suggested in chortle-crf [26]. The basic idea is to approximate the problem of decomposing a function as that of decomposing an SOP of the function. This is unlike Roth-Karp decomposition, which being a functional technique, is independent of the function representation. Cube-packing uses **bin-packing**, a well studied problem [30]. We are given items with weights and bins of fixed capacity. The objective is to pack all the items using a minimum number of bins without violating the capacity of any bin. Here, the used capacity of a bin is the sum of the weights of the items in the bin.

If each cube in the SOP of the function f is treated as an item with weight equal to its literal count, and an LUT as a bin with capacity m, the problem of decomposing the SOP of f into a minimum number of m-LUTs can be seen as a bin-packing problem, although the two are not exactly the same (as explained later). We call this formulation a **cube-packing** problem.

The decision version of bin-packing is NP-complete, [30], and same is true of cube-packing, as shown later in the section. However, there exist efficient heuristics for bin-packing, which can be modified for cube-packing. One such heuristic is the best fit decreasing (BFD). It is modified for cube-packing as follows:

1. Extract m-input AND gates from each cube c until it has at most m literals. Two methods to select the AND gates are studied:

   (a) *regular*: Order the inputs of the function (arbitrarily). A literal whose corresponding input is earlier in the order is extracted earlier from the cube.

(b) *smart*: Order the inputs in the increasing order of occurrence in the sum-of-products expression. Again, a literal earlier in the order is extracted earlier from a cube. The idea is that a literal in the final expression (after all the cubes have become m-feasible), will be in many cubes. This gives more opportunity for sharing of the supports and hence for placing more cubes in each LUT.

The experimental results for these methods are shown in Section 3.6.

2. Order the cubes in non-increasing order of weights.

3. Pick the largest unplaced cube c and try placing it in the bins that have been already used. If it fits in more than one, choose the best bin. We experimented with two definitions of the best bin for a cube c:

   (a) *minimum support*: The support of a partially filled bin is defined as the union of the supports of the cubes placed in it. According to this definition, the best bin for c is the one that has the least support of all bins after c has been placed.

   (b) *minimum increment in the support*: According to this criterion, the bin whose support increases by the least amount when c is placed in it is the best bin for c.

   If c does not fit in any of the partially filled bins, generate a new bin and place c in it. When all cubes have been placed, the bin *most full* is "closed." That is, it is removed from the list of partially filled bins, generating a single literal cube (equivalently, an item of weight 1), which is added to the list of unpacked items. Repeat this step until only a single item of weight 1 is left.

Note that an LUT with more than one cube in it realizes their OR.

We illustrate this approach for $m = 5$ using *minimum increment in support* as the criterion for defining the best bin.

Example 3.3.9 *Let*

$$f = abcd + a'b' + k\ell r + r'p \tag{3.7}$$

Let

$$c_1 = abcd,$$

$$c_2 = a'b',$$
$$c_3 = k\ell r,$$
$$c_4 = r'p$$

be the four cubes. Since there is no cube with more than 5 literals, step 1 is skipped. Next, we sort the cubes in non-increasing order of number of literals. This results in the order c_1, c_3, c_2, c_4. *The cube* c_1 *is placed in bin 1. The cube* c_3 *cannot fit in bin 1 as* c_1 *and* c_3 *together have 7 inputs. So* c_3 *is placed in a new bin 2. Cube* c_2 *is next. It can fit in both bins 1 and 2. If put in bin 1 with* c_1, *it does not use any leftover capacity (inputs) of bin 1, whereas if put in bin 2, it uses two additional units of capacity. Therefore it is placed in bin 1. Finally,* c_4 *can be placed only in bin 2. The resulting configuration is shown in Figure 3.6. We close the bin that is most full. Both the bins are using a capacity of 4; so we arbitrarily close bin 1. This generates a new single literal cube* x, *which is then put in bin 2, since bin 2 had an unassigned input pin.*

Note the following features:

1. The cubes can share inputs, i.e., the sum of the weights of two cubes may be greater than the weight of the cubes merged together. For instance, the weight of c_1 is 4 and the weight of c_2 is 2, but that of $c_1 + c_2$ is 4. This is in contrast with the standard bin-packing problem, where the items do not share weights.

2. Every bin, except the final one (the one realizing f), generates an item of weight 1. This is because each bin realizes a sub-function of f, which later has to combine with other sub-functions to generate f. To handle this, we generate a single literal cube as soon as a bin is closed and make this new cube a new item.

3. BFD is a polynomial time heuristic.

A property of cube-packing

The cube-packing algorithm as described above has an interesting property. We have shown that *for* $m \leq 5$, *it generates an optimum m-feasible tree network for a function consisting of cubes with disjoint supports.*[4] This is useful since many functions in an optimized multi-level network satisfy the disjoint support property.

[4]We proved this result using an explicit ordering of the cubes, not recognizing that we were in fact using the BFD method for bin-packing. In other words, we came up with an algorithm to generate the best feasible tree implementation for such a function. Later, when chortle-crf [26] was published, we recognized that our algorithm was the same as the BFD heuristic. [26] also independently proved this result.

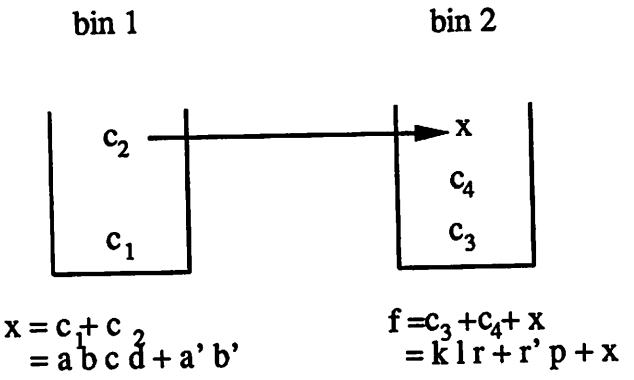


Figure 3.6: An example of cube-packing

Let the function $f = c_1 + c_2 + \ldots + c_n$, where c_i and c_j have mutually disjoint supports (i.e., $\sigma(c_i) \cap \sigma(c_j) = \phi$) for all $1 \leq i, j \leq n, i \neq j$. Let $C = \{c_1, c_2, \ldots, c_n\}$. Let T be any *tree* realization of f using look-up tables. It is convenient in what follows to treat the literals in C as the primary inputs and refer to the support of a cube c_i in terms of its literals instead of variables (in other words, we are assuming, without loss of generality, that all variables appear in the positive phase). This implies that for an internal node t of the tree T, $\sigma_{TG}(t)$ is a subset of the literals in C. We assume that

1. there are no constant inputs to any of the LUTs. If there were, we can always propagate them to the output, and
2. each immediate fanin of an LUT s in T belongs to the true support of the function implemented by s (otherwise, it can be deleted).

The proof of optimal tree realization is in two steps.

1. We first determine T's structure. It is shown in Proposition 3.3.9 that each node of T either implements a supercube of some cube c_i, or the complement of a supercube, or a sub-function of f, or the complement of a sub-function.
2. Using the structure of the tree determined in the first step, we then show in Theorem 3.3.10 that the algorithm generates an optimum tree.

The proof of Proposition 3.3.9 rests on Lemma 3.3.6, Lemma 3.3.7, and Proposition 3.3.8. Lemma 3.3.6 is just a restatement of the fact that if all the children of each LUT are in its true support, then the primary inputs in the TFI of each LUT are also in its global true support.

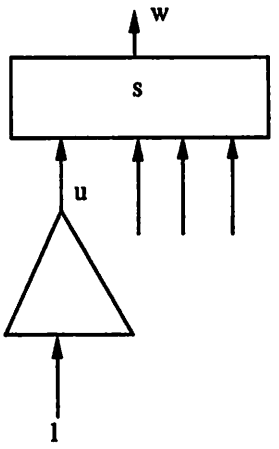


Figure 3.7: $l \in \sigma_{TG}(u)$ & $u \in \sigma_T(w) \Rightarrow l \in \sigma_{TG}(w)$

Lemma 3.3.6 *Let T have an* LUT *s with output w (Figure 3.7) such that $u \in \sigma_T(w)$. Then $l \in \sigma_{TG}(u) \Rightarrow l \in \sigma_{TG}(w)$.*

Proof Using Shannon expansion of w w.r.t. u,

$$w = w_u u + w_{u'} u' \tag{3.8}$$

$$= xu + yu', \text{ where } x = w_u, y = w_{u'}. \tag{3.9}$$

$$\Rightarrow \quad w_l = xu_l + yu'_l \tag{3.10}$$

$$\text{and} \quad w_{l'} = xu_{l'} + yu'_{l'} \tag{3.11}$$

Note that (3.10) and (3.11) hold because x and y are independent of l ($x = w_u$ and therefore depends on the inputs of s that are to the right of u in Figure 3.7. Since T is a tree, these inputs cannot have l in their support. Same argument works for y.). For the sake of contradiction, assume that the global function of w is independent of l, i.e.,

$$w_l = w_{l'} \tag{3.12}$$

Substituting (3.10) and (3.11) in (3.12), we get,

$$xu_l + yu'_l = xu_{l'} + yu'_{l'} \tag{3.13}$$

$$(xu_l + yu'_l) \oplus (xu_{l'} + yu'_{l'}) = 0 \tag{3.14}$$

Multiplying and reorganizing terms,

$$(u_l \oplus u_{l'})(x \oplus y) = \mathbf{0} \tag{3.15}$$

For some point $v \in B^{|\sigma(w)-\{u\}|}$, $x(v) \neq y(v)$ (since $u \in \sigma_T(w)$). Then $x(v) \oplus y(v) = 1$. Since T is a tree, u_l and $u_{l'}$ do not share supports with either x or y. Hence v is independent of the assignment of input values to u_l and $u_{l'}$. So (3.15) becomes

$$\begin{aligned}(u_l \oplus u_{l'}) &= \mathbf{0} \\ \Rightarrow u_l &= u_{l'}\end{aligned}$$

This implies u is independent of l. A contradiction. ■

Next, we state a lemma that is the key to deriving T's structure. Recall that for a set S of functions, $\sigma(S)$ denotes the union of supports of the functions in S. Also, n is the number of cubes in the SOP of f.

Lemma 3.3.7 *For a cube c_i, $1 \leq i \leq n$, there exists an* LUT *block t in T such that it is possible to partition the inputs of t into two sets I_t and II_t, $II_t \neq \phi$ if $n \geq 2$, such that $\sigma_{TG}(II_t) \cap \sigma_{TG}(c_i) = \phi$ and $\sigma_{TG}(I_t) = \sigma_{TG}(c_i)$.*

Proof If there is only one cube in the SOP of f, the output LUT of T is the desired t (with $II_t = \phi$) and we are done. Now, assume there are at least two cubes, i.e., $n \geq 2$. To get the desired t, traverse the tree T from its root towards the inputs. Say during the traversal, we are at an LUT s whose output is r. The LUT s satisfies two invariants:

$$\begin{aligned}\sigma_{TG}(r) \cap \sigma_{TG}(c_i) &\neq \phi, \text{ and} \\ \sigma_{TG}(r) \cap \sigma_{TG}(f - c_i) &\neq \phi,\end{aligned}$$

where $f - c_i = c_1 + c_2 + \ldots + c_{i-1} + c_{i+1} + \ldots + c_n$. Note that the root of T satisfies these invariants. If s has a child v such that

$$\sigma_{TG}(v) \cap \sigma_{TG}(c_i) \neq \phi, \text{ and } \sigma_{TG}(v) \cap \sigma_{TG}(f - c_i) \neq \phi, \tag{3.16}$$

set s to the LUT that generates v (v is not a primary input if it satisfies (3.16)). Clearly, no leaf LUT of T (a leaf LUT is one that has only primary inputs as its children) has such an input v. So we will eventually reach an LUT t realizing function w such that all of the following are satisfied:

$$\begin{aligned}\sigma_{TG}(w) \cap \sigma_{TG}(c_i) &\neq \phi, \\ \sigma_{TG}(w) \cap \sigma_{TG}(f - c_i) &\neq \phi, \\ \sigma_{TG}(v) \cap \sigma_{TG}(c_i) = \phi, \text{ or } \sigma_{TG}(v) \cap \sigma_{TG}(f - c_i) &= \phi, \ \forall v \in \sigma_T(w)\end{aligned}$$

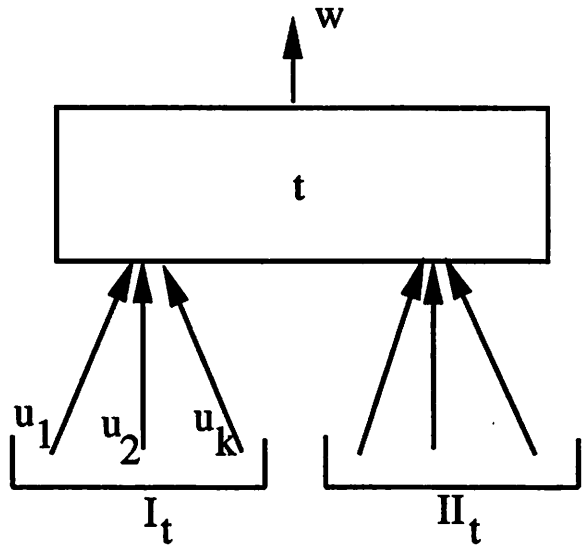


Figure 3.8: Structure of the tree T: $\sigma_{TG}(I_t) = \sigma(c_i)$ and $\sigma_{TG}(II_t) \cap \sigma(c_i) = \phi$

Form the set II_t by putting in it all the inputs v of t that satisfy $\sigma_{TG}(v) \cap \sigma_{TG}(c_i) = \phi$ (so $\sigma_{TG}(v) \cap \sigma_{TG}(f - c_i) \neq \phi$). The rest of the inputs v of t satisfy $\sigma_{TG}(v) \cap \sigma_{TG}(f - c_i) = \phi$ and constitute I_t. II_t is non-empty, since $\sigma_{TG}(w) \cap \sigma_{TG}(f - c_i) \neq \phi$. Similarly, I_t is non-empty. Clearly $\sigma_{TG}(II_t) \cap \sigma_{TG}(c_i) = \phi$ and $\sigma_{TG}(I_t) \subseteq \sigma_{TG}(c_i)$.

Suppose $\sigma_{TG}(I_t)$ is a proper subset of $\sigma_{TG}(c_i)$. Make $c_i = 0$ by setting some variable in $\sigma_{TG}(I_t)$ to an appropriate value (i.e., 0 or 1). As a result, some local functions may become constants (0s or 1s). Propagate these constants as far as possible towards the root of T. This propagation stops at or before t, i.e., the new local function implemented by t, say $\tilde{w}$, is non-trivial (i.e., non-**0**, non-**1**). This is because of the following reason. Now T implements a new global function, $\tilde{f} = f - c_i$. Therefore $\tilde{f}$ depends on $\sigma_{TG}(II_t)$. Since $II_t \neq \phi$, the only way that is possible is if $\tilde{w}$ is non-trivial. Let $\hat{T} = T - \{t\} - \mathsf{TFI}(t)$, and $S = \sigma_{TG}(c_i) - \sigma_{TG}(I_t)$. The last observation then implies that the local function at each LUT in $\hat{T}$ remains unchanged. Now, consider S. By assumption, $S \neq \phi$. Also, the inputs in S fan out only to LUTs in $\hat{T}$. It then follows from Lemma 3.3.6 that $S \subseteq \sigma_{TG}(\tilde{f})$. This leads to a contradiction, since $\tilde{f} = f - c_i$ does not depend on c_i. Hence, $\sigma_{TG}(I_t) = \sigma_{TG}(c_i)$. ■

Figure 3.8 shows the resulting structure.

Proposition 3.3.8 *If an* LUT *s of* T *implements the function* r*, either*

(1) $\sigma_G(r) \subseteq \sigma(c_i)$ *for some cube* $c_i \in C$, *or*

(2) $\sigma_G(r) = \cup_{c_i \in \tilde{C}} \sigma(c_i)$ *for* $\tilde{C} \subseteq C$.

Proof There are two possibilities. Either r is a function of variables from just one cube, say c_i, which gives (1), or it is a function of variables from at least two cubes c_i and c_j of C. Then $\sigma(c_j) \cap \sigma_G(r) \neq \phi$. For contradiction, assume that $\sigma(c_i) \cap \sigma_G(r) = S$, where S is a non-empty proper subset of $\sigma(c_i)$. Now consider the LUT t for the cube c_i, as given by Lemma 3.3.7. Since $\sigma(c_i) \subseteq \sigma_G(w)$ (w is the function implemented by t), and T is a tree, t is in the TFO of s. Let u be the input to t such that s is in the TFI of u. Since $\sigma_G(u)$ contains variables from c_i as well as c_j, partition (I_t, II_t) of inputs of t with the property $\sigma_G(I_t) = \sigma_G(c_i)$ cannot be formed. This contradicts Lemma 3.3.7. Hence, $\sigma(c_i) \subseteq \sigma_G(r)$. Arguing over all the cubes whose supports intersect $\sigma_G(r)$, we get (2). ■

The next question is regarding the global functions implemented by an LUT $s \in T$. The following proposition answers it.

Proposition 3.3.9 *If an* LUT s *of* T *implements the function* r, *either*

1. r *is a supercube of some cube* c_i, *or the complement of a supercube, or*
2. r *is a sub-function of* f, *or the complement of a sub-function of* f.

Proof First we note that the tree T corresponds to a series of simple disjoint decompositions on the function f. In particular, to determine r, we consider a partition of the form $(\sigma_G(r), \sigma_G(f) - \sigma_G(r))$. This is shown in Figure 3.9. By Proposition 3.3.8, two possibilities arise:

1. $\sigma_G(r) \subseteq \sigma(c_i)$ for some cube $c_i \in C$: then we form a decomposition chart for f for the partition $(\sigma_G(r), \sigma(f) - \sigma_G(r))$. We illustrate the proof technique with an example; the general proof is based on exactly the same arguments and is omitted. Let $f = abc + de$. Let $\sigma_G(r) = \{a, b\}$. Then the decomposition chart of Figure 3.10 is formed. It is a superimposition of two charts, as shown in Figures 3.11 and 3.12. The first chart (Figure 3.11) shows the values that f takes on the cube abc. Note that the bound set is a subset of the support of abc. So the entries are 1 only when $f_1 = abc = 1$, i.e., $a = b = c = 1$. This corresponds to 1s appearing in a single column, say v. In the example, v corresponds to $a = 1, b = 1$. The second chart shows the rest of the function $f_2 = f - f_1 = de$. It has the property that 1s occur in it only as complete rows. We superimpose the two charts to obtain

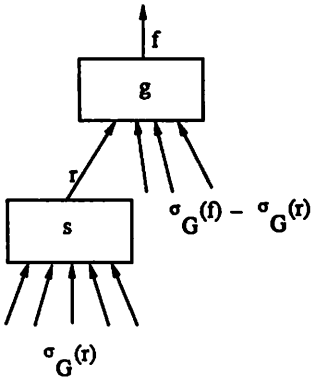


Figure 3.9: Determining r using the theory of simple disjoint decomposition

$\frac{ab}{cde}$	00	01	10	11
000	0	0	0	0
001	0	0	0	0
010	0	0	0	0
011	1	1	1	1
100	0	0	0	1
101	0	0	0	1
110	0	0	0	1
111	1	1	1	1

Figure 3.10: Decomposition chart for $f = abc + de$ for the partition (ab, cde)

$\frac{ab}{cde}$	00	01	10	11
000	0	0	0	0
001	0	0	0	0
010	0	0	0	0
011	0	0	0	0
100	0	0	0	1
101	0	0	0	1
110	0	0	0	1
111	0	0	0	1

Figure 3.11: Decomposition chart for $f_1 = abc$ for the partition (ab, cde)

$\frac{ab}{cde}$	00	01	10	11
000	0	0	0	0
001	0	0	0	0
010	0	0	0	0
011	1	1	1	1
100	0	0	0	0
101	0	0	0	0
110	0	0	0	0
111	1	1	1	1

Figure 3.12: Decomposition chart for $f_2 = de$ for the partition (ab, cde)

the chart for f (shown in Figure 3.10), i.e., if for an entry, one chart has a 1, and the other has a 0, the chart for f has a 1 for that entry. This is because $f = f_1 + f_2$. There are exactly two distinct columns in the chart for f. This can be seen if the rows of Figure 3.10 are rearranged such that the rows of 1s corresponding to f_2's chart appear at the top. The reason is as follows. There is at least one entry that has a 1 in f_1's chart and a 0 in f_2's (since $f_1 - f_2 \neq \mathbf{0}$). One of the two distinct columns corresponds to v and the other, to any other column of the chart. To determine r, we use Ashenhurst's technique described in Section 3.3.1. We put v in the on-set of r and get $r = ab$. Had we put v in the off-set of r, $r = (ab)'$. So r is either a supercube or the complement of a supercube. Note that if f had only one cube, $f = f_1$ and $f_2 = \mathbf{0}$. It is easy to see that f has exactly two distinct column patterns here too.

2. $\sigma_G(r) = \cup_{c_i \in \tilde{C}} \sigma(c_i)$ for $\tilde{C} \subseteq C$. This case is similar to the previous one, except that here

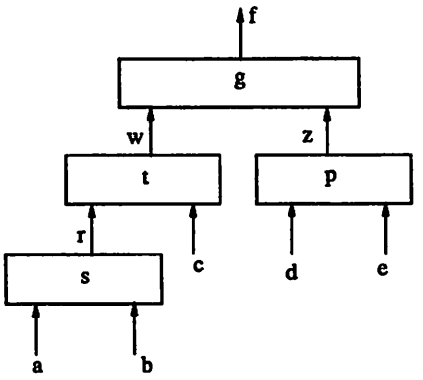


Figure 3.13: Determining $w(r, c)$ using the theory of simple disjoint decomposition

the chart for f_1 corresponds to the subcover $\tilde{C}$, and has 1s occurring as complete columns. If these columns are put in the on-set of r, $r = \sum_{c_i \in \tilde{C}} c_i$, a sub-function of f. On the other hand, if these columns are put in the off-set of r, $r = (\sum_{c_i \in \tilde{C}} c_i)'$, which is the complement of a sub-function.

■

Having determined the possibilities for the global functions implemented by each LUT, the next step is to determine the corresponding local functions. This will complete our understanding of T's structure. The local function r implemented by an LUT s in T is in terms of its immediate fanins $\{u_i\}$. Given $\{u_i\}$ along with their global functions and the global function r, the local function r is unique. This follows from Corollary 5.3.5, since neither u_i nor u_i' is $\mathbf{0}$ for all i. We illustrate this using the previous example. Figure 3.13 shows a tree implementation T for $f = abc + de$. We are interested in the local function implemented by the LUT t. The immediate fanins of t are $\{r, c\}$. Let $r = (ab)'$. First, determine the global function $w(a, b, c)$ implemented by t. From Proposition 3.3.9, either $w = abc$ or $w = (abc)'$. Let $w = abc$. Since $r = (ab)'$, from Ashenhurst's theory, we get $w(r, c) = r'c$. Moreover, since neither r nor r' is $\mathbf{0}$, from Corollary 5.3.5, $w(r, c)$ is unique. Similarly, if $w(a, b, c) = (abc)'$, $w(r, c) = r + c'$. In general, a series of decomposition steps are needed to determine the local function. For instance, to determine the local function $f(w, z)$ in Figure 3.13, first $f(w, d, e)$ is determined (since the global functions $f(a, b, c, d, e)$ and $w(a, b, c)$ are known). Then $z(d, e)$ is determined using Proposition 3.3.9, and finally $f(w, z)$ is determined.

This completes the discussion on determining the structure of T. This result is an extension of the *AND-OR boundary lemma* proved by Wang [85]. The AND-OR boundary lemma is about the structure of a tree decomposition of f using 2-input NAND gates and inverters, where f consists of

cubes with disjoint supports. It says that for each cube c_i, there is some NAND gate in the tree that realizes the complement of c_i. However, derivation of the lemma is simpler than that of Proposition 3.3.9, because

- the relationship between inputs and the output of the gate is known - it is a NAND function, and
- each gate has only 2 inputs.

When we first proved Proposition 3.3.9, we used arguments similar to, but more complicated than, the ones used in the proof of AND-OR boundary lemma. The proof based on Ashenhurst't theory presented here is much simpler.

This result is in conformity with the natural way of generating a tree realization for f - building it cube by cube. We will use the structure of T in proving that the BFD algorithm generates an optimum feasible tree structure for f. We restrict ourselves to the case $m = 5$. The proofs for the cases when $m < 5$ can be similarly derived. First we make a few remarks.

1. If T is an optimum tree implementation of f, then we can get an alternate optimum tree implementation $\tilde{T}$ of f such that all the logic blocks in $\tilde{T}$ realize either a supercube of some cube c_j or a sub-function of f. This is done by pushing inverters and appropriately changing the local functions. From now on, we restrict ourselves to such optimum tree implementations only. Note that for such a tree T, the local and the global functions at each LUT are unique. The only significant thing is the structure of T, i.e., which inputs go to an LUT.

2. Without loss of generality, we assume that no cube of f has more than 4 literals. If it had, we can repeatedly extract 5 literals until this condition is satisfied. It follows from Proposition 3.3.9 that any feasible tree implementation of f must have such a structure. Moreover, since all the inputs of a cube are *similar* (i.e., have same phase), we do not lose any optimality this way.[5]

   We can now show that an LUT cannot be saved by splitting a 5-feasible cube, i.e., partitioning its inputs into two or more sets, each set fanning out to a separate LUT. An example suffices. If there is an optimum implementation T with a split-cube, say $c_i = abcd$, then an alternate optimum implementation $\tilde{T}$ can be obtained with c_i not split - just move all the inputs to fan out to the first LUT encountered in a depth-first search (DFS) of T. This is shown in Figure

[5]No optimality is lost even when some inputs are in negative phase, since inverters are essentially free in LUT realizations.

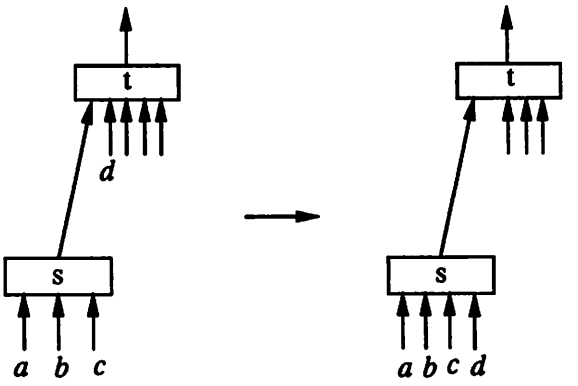


Figure 3.14: Splitting a 4-feasible cube does not help for a 5-feasible implementation

3.14. The movement of inputs can be performed without changing the global function. In Figure 3.14 the LUT t realizes a sub-function of f, and after the input movement, the local functions of s and t change to ensure that t realizes the same global function as before. It follows that there exists an optimum implementation $\tilde{T}$ of f such that all the logic blocks in $\tilde{T}$ realize a sub-function of f. We can get rid of those LUTs of T that realized proper supercube functions.

Let f be a function whose cubes have mutually disjoint supports and none of them has more than 4 literals. It is convenient to use the terms items, weights, and bins, instead of cubes, number of literals, and LUTs respectively. First we classify all the items (cubes) by weight (number of literals). We explicitly list first few steps of the cube-packing algorithm for $m = 5$.

1. Pair items of weight 3 with items of weight 2, until one of them runs out. Each such pair is put into a bin (LUT), whose output is treated as an item of weight 1.

2. Pair items of weight 4 with items of weight 1. Each such pair is put into a bin, whose output is treated as an item of weight 1. This, in turn, can be used in a pair involving an item of weight 4. Repeat the mergings until one of them runs out.

3. We are now left with one of the following weight combinations: 1s, or 2s, or 3s, or 4s, or 1s and 2s, or 1s and 3s, or 2s and 4s, or 3s and 4s. We apply the BFD algorithm on each of these combinations. For instance, for the first combination, repeatedly place five 1s in each bin until no unplaced items remain. For the case when the only items are of weight 2, place two items in one bin, close it, and then place two items of weight 2 and the item of weight 1 generated when last bin was closed. Repeat this process.

Theorem 3.3.10 *The above algorithm generates an optimum feasible tree implementation for a function f whose cubes have mutually disjoint supports.*

Sketch of Proof Let the tree generated by the algorithm be S. Let T be an optimum tree implementation whose each LUT realizes a sub-function of f. We will transform T by changing its edge connections such that finally we obtain S, without increasing the number of LUTs. We do so by *simulating* the steps of the above algorithm on T. Consider step 1. If it merged cubes $c_i = l_1 l_2$ and $c_j = l_3 l_4 l_5$, we locate these cubes on T. Let $\{l_1, l_2\}$ and $\{l_3, l_4, l_5\}$ be inputs to LUTs s and t respectively. If $s = t$, nothing needs to be done. Otherwise, three cases arise:

1. LUTs s and t do not occur in the same path to the root (Figure 3.15): swap cube c_j with other inputs of s, such that c_j is now with c_i. This swap is always possible, since there were at most 3 other inputs to s, namely o, p and q. They can be moved to the three vacant input spots created by l_3, l_4 and l_5 at t. Note that this can always be done as each LUT of T realizes a sub-function of f (Remarks 1 and 2).

2. t is on the path from s to the root (Figure 3.16): swap c_j with the other inputs (o, p and q) of s.

3. s is on the path from t to the root: swap c_i with the other inputs of t.

We repeat this process for all the cubes merged in step 1 of the algorithm. In the end, we have generated T_1, which has as many blocks as T and matches with S on inputs to LUTs that involve cubes of step 1. It is important to note that the matched LUTs (i.e., to which swaps were made in the simulation, e.g., s in the cases 1 and 2, and t in 3) correspond to the bins closed in this step of the simulation, and their inputs correspond to items that have been placed in the bins. These LUTs will not be involved in any swapping in the simulation process to follow.

We now simulate step 2 on T_1 and generate T_2. Carrying on the simulation, we finally generate T_z that matches S on all the LUTs of S. So S is a sub-tree of T_z. Since we never added LUTs in the entire process, the number of LUTs in T_z is no more than that in T. Since T is optimum, T_z is optimum. This implies that S is optimum as well. That is, $T_z = S$. ∎

We have been somewhat sketchy in the last proof. For instance, consider a case that can arise during simulation, but is not mentioned in the proof. Assume that we are about to simulate the step 3 on a tree, say T_k, and that we only have items of weight 1 and 2. Also assume that there are two items c_1 and c_2 of weight 2. The BFD algorithm will place c_1 and c_2 along with an item,

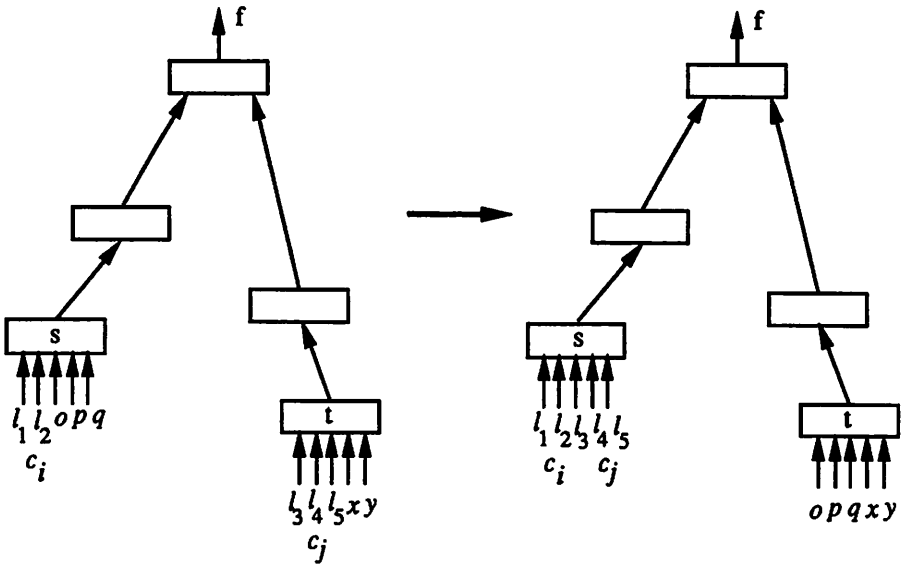


Figure 3.15: Tree before and after the swap - case 1

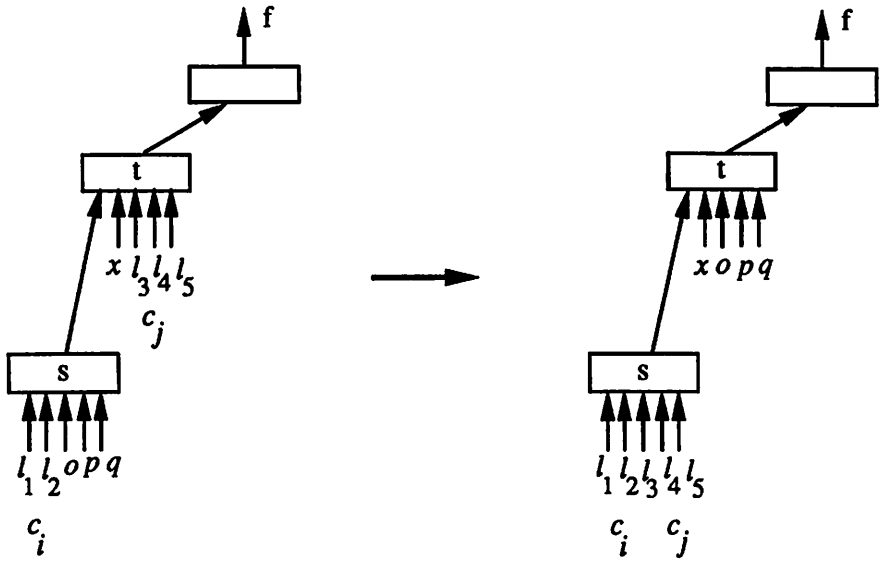


Figure 3.16: Tree before and after the swap - case 2

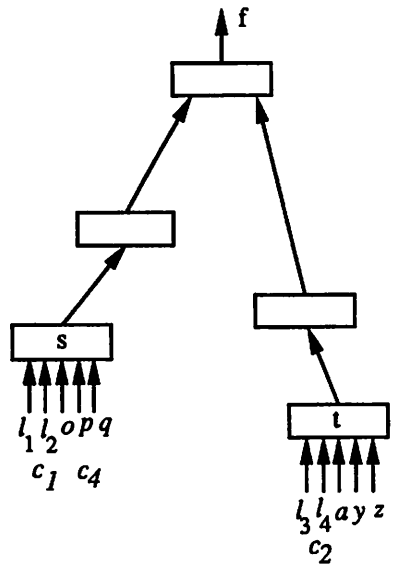


Figure 3.17: c_1 and c_4 cannot be inputs to the same LUT

say c_3, of weight 1. Let c_1 and c_2 be inputs to LUTs s and t respectively in T_k, $s \neq t$ (Figure 3.17). Simulating the step 3, c_1, c_2 and c_3 will be made inputs of the same LUT, say s. Then c_2 will need to be moved to s. But what if s has as input another cube c_4 with 3 literals o, p and q? Cube c_4 needs to be moved to t. However, t may not be able to accommodate c_4; it may have the other three input spots occupied (Figure 3.17). Fortunately, this situation cannot happen. The LUT s cannot have c_4, a 3-literal cube, as its input. By now, all 3-literal cubes have been taken care of in step 1 - they are secure in bins that have been closed. In fact, in this case step 1 terminated exactly when we ran out of 3-literal cubes. So the order in which the items are placed in the bins is important.

The same arguments can be applied for $m < 5$.

A similar result holds if f has a POS representation consisting of sum-terms with disjoint supports. We now pack sum-terms in the bins.

Theorem 3.3.11 *For $m \leq 5$, the* BFD *algorithm operating on sum-terms generates an optimum m-feasible tree implementation for a function with a* POS *consisting of sum-terms with disjoint supports.*

Proof Follows from Theorem 3.3.10 using duality, i.e., replace AND with OR and OR with AND everywhere. ■

Complexity of cube-packing

We have seen that for $m \leq 5$, the BFD algorithm generates an optimum m-feasible tree implementation for a function consisting of cubes with disjoint supports. Note that BFD is a polynomial time algorithm. It is natural to ask the following questions.

1. Is the BFD algorithm optimum for a function f consisting of cubes with disjoint supports for arbitrary m? If not, what is the complexity of the cube-packing problem, i.e., of minimizing the LUT count for f using the cubes as items?

2. What can we say about an arbitrary function?

We may assume that all the cubes have at most m literals.[6] Let **CUBE-PACKING** be the decision version of the corresponding cube-packing problem for a general function. As mentioned earlier, it is similar to the bin-packing problem. Since bin-packing is NP-complete [30], we are inclined to believe that CUBE-PACKING is also NP-complete. That is indeed the case. To see this, we restrict the function in the CUBE-PACKING problem to one consisting of cubes with mutually disjoint supports. Let us call this problem **DISJOINT SUPPORT CUBE-PACKING**. Also note that DISJOINT SUPPORT CUBE-PACKING is the same as obtaining a feasible tree realization of such a function with a minimum number of LUTs. Stated as a decision problem, it becomes:

INSTANCE: *Finite set U of items, weight $w(u) \in \mathcal{Z}^+$, the set of positive integers, for each $u \in U$, bin capacity $m \in \mathcal{Z}^+$, and a positive integer K. Also, each bin (except the "last" one) generates an additional item of weight 1.*
QUESTION: *Can all the items be "packed" in at most K bins, where the sum of the weights of items in each bin is m or less?*

We were not precise in defining the problem; the phrase *each bin (except the "last" one) generates an additional item of weight 1* was loosely used. A precise way would be to annotate the final packing with a directed graph G. The vertices of G are the bins used in the packing, and there is an edge from a vertex u to vertex v if the single-literal item generated from the bin u is placed in v. A packing is valid only if G is acyclic with single root. In fact, for a *minimal* packing (i.e., each new item is placed in exactly one bin), G is a rooted tree. We have preferred to sacrifice preciseness for simplicity.

[6] If we show a restricted version of some problem to be NP-complete, the general problem is also NP-complete.

We show that DISJOINT SUPPORT CUBE-PACKING is NP-complete using a transformation from PARTITION-1, which is:

INSTANCE: *Finite set A and a weight $w(a) \in \mathcal{Z}^+$ for each $a \in A$.*
QUESTION: *Is there a subset $\tilde{A} \subseteq A$ such that*

$$\sum_{a \in \tilde{A}} w(a) = (\sum_{a \in A - \tilde{A}} w(a)) + 1 \tag{3.17}$$

First we show that PARTITION-1 is NP-complete.

Lemma 3.3.12 PARTITION-1 *is NP-complete.*

Proof Given a solution, i.e., subset $\tilde{A}$, it is easy to check that (3.17) is satisfied in polynomial time. This shows that the problem is in NP. We transform PARTITION, which is known to be NP-complete [30], into PARTITION-1. The PARTITION problem is:

INSTANCE: *Finite set B and a weight $s(b) \in \mathcal{Z}^+$ for each $b \in B$.*
QUESTION: *Is there a subset $\tilde{B} \subseteq B$ such that*

$$\sum_{b \in \tilde{B}} s(b) = \sum_{b \in B - \tilde{B}} s(b) \tag{3.18}$$

From B, we construct the set A as $A = B \cup \{\tilde{a}\}$, where $\tilde{a}$ is a new item with $w(\tilde{a}) = 1$. For each $b \in B$, let $w(b) = 3s(b)$. Suppose there exists $\tilde{B} \subseteq B$ that satisfies (3.18). Form $\tilde{A} = \tilde{B} \cup \{\tilde{a}\}$. It is easy to see that $\tilde{A}$ satisfies (3.17). Conversely, assume that an $\tilde{A}$ exists satisfying (3.17). Then $\tilde{a} \in \tilde{A}$. This is because if $\tilde{a} \in A - \tilde{A}$, then the weight of the set $\tilde{A}$ (which is the sum of the weights of the items in $\tilde{A}$) is two more than the weight of the set $A - \tilde{A} - \{\tilde{a}\}$. This is not possible, since all the items in these two sets have weights that are multiples of 3. Then it is easy to see that $\tilde{B} = \tilde{A} - \{\tilde{a}\}$. $\tilde{B}$ satisfies (3.18). ∎

Theorem 3.3.13 DISJOINT SUPPORT CUBE-PACKING *is NP-complete.*

Proof That it is in NP is easy to see. We transform PARTITION-1 to DISJOINT SUPPORT CUBE-PACKING. The set U is the same as set A, and the items have the same weight in set U as in set A. Set $m = \frac{[\sum_{a \in A} w(a)]+1}{2}$ and $K = 2$. If there exists a subset $\tilde{A}$ of A that satisfies (3.17), then put the items of $\tilde{A}$ in the first bin. This generates an item $\tilde{a}$ of weight 1. Put the elements of

$(A - \tilde{A}) \cup \{\tilde{a}\}$ in the second bin. Note that the total weight of items in each of the two bins is m (unless there is just one item, with weight 1, in which case $m = 1$, and one bin suffices). Conversely, given that the items can be packed in two bins, there is a bin B_1 that generates an additional item of weight 1. All the items in B_1 form set $\tilde{A}$. It is easy to check that $\tilde{A}$ satisfies (3.17). Note that the items can be packed in one bin if and only if U has only one item, and that has a weight of 1. $\tilde{A}$ then has just that item, and (3.17) is still satisfied. ■

Corollary 3.3.14 CUBE-PACKING *is NP-complete.*

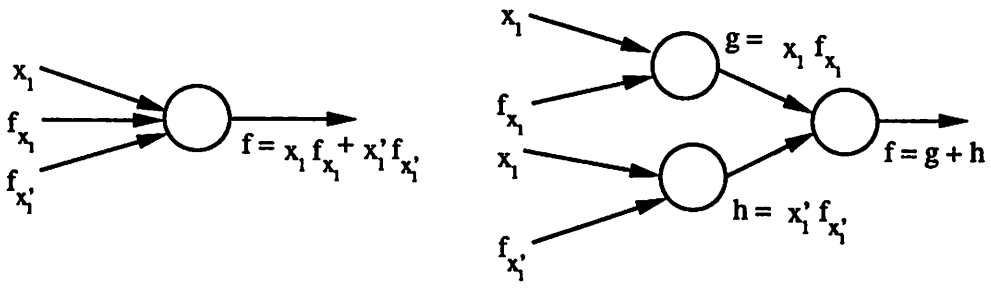
Proof That CUBE-PACKING is in NP is easy to see. NP-completeness follows by noting that each instance of DISJOINT SUPPORT CUBE-PACKING is also an instance of CUBE-PACKING. Thus DISJOINT SUPPORT CUBE-PACKING is just a restricted version of CUBE-PACKING. ■

From the above discussion, we may suspect that the BFD algorithm presented earlier, which takes polynomial time, is not optimum for some m for a function consisting of cubes with mutually disjoint supports. This is indeed the case. For simplicity, consider $m = 4000$. Let there be 6 cubes each with 2002, 1004, and 1002 literals and let there be 12 cubes with 995 literals. The BFD algorithm will pair cubes of sizes 2002 and 1004, using six bins to pack these. Then, it will pack 3 1002-literal cubes in a bin, using two more bins. Finally, it will pack 4 995-literal cubes in a bin, needing three bins for the 12 items. Note that enough space is left over in the bins to take care of the single-literal cubes generated on closing the bins. The total number of bins used is 11. The optimal decomposition (packing) is as follows. Place one 2002, one 1002, and one 995-literal cube in a bin. Six bins are needed for all such items. Then pack two 1004 and two 995-literal cubes in a bin, using three more bins. Again, each bin has enough leftover capacity to accommodate the additional single-literal cubes. The number of bins needed is 9. This counterexample is a slight modification of an example presented in [30] to show that BFD is not always optimum for bin-packing.

3.3.3 Cofactoring

Although cube-packing works reasonably well for functions with cubes having disjoint supports, it may perform poorly if cubes share many variables.

Example 3.3.10 *Consider the function* $f(a,b,c,d) = abc' + ab'd + a'cd + bcd'$. *Let* m *be 3. The* BFD *procedure would pack one cube in each* LUT *and that gives an* LUT *count of 6. However, a cofactoring procedure gives an* LUT *count of 3. Performing Shannon cofactoring of* f *with respect*



(A) m > 2 (B) m = 2

Figure 3.18: Using cofactoring for decomposition

to a, we get

$$\begin{aligned} f_a &= bc' + b'd + bcd', \\ f_{a'} &= cd + bcd', \\ f &= af_a + a'f_{a'}. \end{aligned}$$

*All the three sub-functions shown above are 3-feasible, and hence generate a realization of f using three 3-*LUT*s.*

We construct a cofactor-tree for $f(x_1, x_2, \ldots, x_n)$ by decomposing it as follows:

$$f = x_1 f_{x_1} + x_1' f_{x_1'} \tag{3.19}$$

Both f_{x_1} and $f_{x_1'}$ are functions of at most $n-1$ variables. If $f_{x_1}(f_{x_1'})$ is m-feasible, we stop, otherwise we recursively decompose it. For $m > 2$, we need one LUT to realize f as in (3.19) (Figure 3.18 (A)), whereas for $m = 2$, we need 3 LUTs (Figure 3.18 (B)).

Cofactoring is a special case of disjoint decomposition, which was described in Section 3.3.1. Cofactoring a function $f(x_1, x_2, \ldots, x_n)$ with respect to x_1 can also be done using a disjoint decomposition on f with the input partition $(\sigma(f) - \{x_1\}, \{x_1\}) = (\{x_2, x_3, \ldots, x_n\}, \{x_1\})$. The column multiplicity of the corresponding decomposition chart is at most 4, since the chart has just two rows, in which case, the only column patterns possible are 00, 01, 10, and 11. This is shown in Figure 3.19, where C_i $(0 \leq i \leq 3)$ denotes the equivalence class of those columns whose pattern is a two-bit binary representation of i. These four classes can be encoded using two encoding variables: α_1 and α_2. Assigning the codes to the classes by the scheme of Figure 3.20, we see that $f_{x_1'} = \alpha_1$ and $f_{x_1} = \alpha_2$.

$\frac{x_2x_3\ldots x_n}{x_1}$	C_0	C_1	C_2	C_3
0	0	0	1	1
1	0	1	0	1

Figure 3.19: Cofactoring as a special case of disjoint decomposition: possible equivalence classes

equivalence class	α_1	α_2
C_0	0	0
C_1	0	1
C_2	1	0
C_3	1	1

Figure 3.20: Cofactoring as a special case of disjoint decomposition: assigning codes

Since each cofactoring step generates functions with supports strictly smaller than the original function, cofactoring is used to derive upper bounds on the number of LUTs needed to implement a function. The complete details are in Chapter 5.

3.3.4 Kernel Extraction

Kernels of an infeasible node function f are enumerated, and the best kernel k is extracted. A new node corresponding to k is created and substituted into f. The process is repeated on the new f and k recursively. It may happen that f is infeasible and has no non-trivial kernels. For example, let m be 5. Then $f = abc + deg$ cannot be made 5-feasible by kernel extraction. In this case, we resort to either cube-packing or technology decomposition (to be described next). Both these techniques are guaranteed to return a feasible representation.

3.3.5 Technology Decomposition

Technology decomposition breaks each node function into two-input AND and OR gates, thereby generating an m-feasible network for any $m \geq 2$.

Example 3.3.11 *Consider $f = abc + deg$. After applying technology decomposition into two-input AND and OR gates, we obtain*

$$x_1 = ab$$

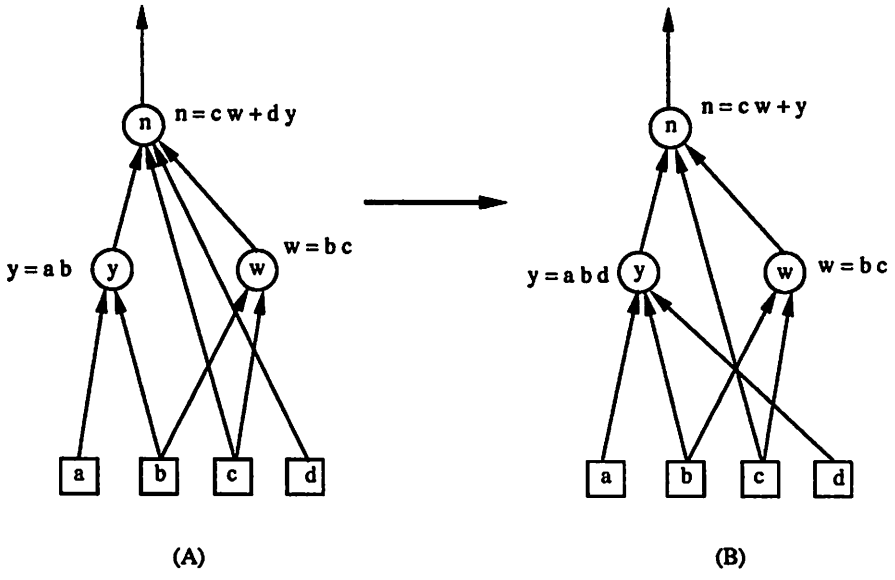


Figure 3.21: Using support reduction to obtain feasibility

$$\begin{aligned} x_2 &= x_1 c \\ y_1 &= de \\ y_2 &= y_1 g \\ f &= x_2 + y_2. \end{aligned}$$

To generate a good implementation for m-LUTs, $m > 2$, one has to rely on BCM, which follows decomposition. One disadvantage of this technique is that the resulting network can have too many nodes, and running BCM in exact mode may not be possible. Even the BCM heuristics may not produce good quality solutions.

3.3.6 Using Support Reduction to Achieve Feasibility

The techniques described in Sections 3.3.1 through 3.3.5 lie in the realm of decomposition, in that they break an infeasible function into a set of feasible functions. We now describe a technique that tries to achieve feasibility by reducing support of the function using the structure and functionality of the network. The following example explains the idea.

Example 3.3.12 *Consider the network η shown in Figure 3.21 (A). Let m be 3. Since w and y are 3-feasible and n is not, the decomposition step only breaks n. The local function at n needs*

*two 3-*LUT*s (one way is to first realize $x = cw$ and then $n = x + dy$). So, the total number of* LUT*s needed for η is four. However, if we move the input d of n over to y, n becomes 3-feasible, without destroying 3-feasibility of y. The resulting network, shown in Figure 3.21 (B), is functionally equivalent to η, and uses three* LUT*s.*

The idea is to try to reduce the support of an infeasible function repeatedly until it becomes feasible. During the process, no feasible function of the network is made infeasible. To see if the support of an infeasible node n can be reduced, the following algorithm is used:

1. If n has a fanin G that is not a primary input and fans out only to n, collapse G into n to get $\tilde{n}$.

2. Redecompose $\tilde{n}$ using one of the decomposition techniques described earlier. If redecomposition results in two m-feasible functions, report success. Otherwise, find another G that satisfies the above properties.

This procedure is shown in action for the network of Example 3.3.12 in Figure 3.22. Note that $G = y$.

An approximation of the above idea is to *move the fanins*. Here we explore the possibility of moving a fanin F of function n over to the fanin G of n, without changing the functionality of the network. This is illustrated in Figure 3.23. Further, after F moves over to G, G should remain feasible. The same procedure as described above may be used with an additional constraint that F is finally moved over to G. This is ensured by using functional decomposition, with F in the bound set. If there are at most two equivalence classes, $\tilde{n}$ can be redecomposed into $\tilde{G}$ and $\hat{n}$.

In literature, some support reduction methods have been proposed. Two such are by Halatsis and Gaitanis [34], and Savoj *et al.* [71]. These methods use the don't care information to express the node function at n on a minimum support; functions at other nodes of the network are not changed. However, in our method, besides n, functions at other nodes change as well.

If a function is infeasible but can be made feasible by repeated applications of the above technique, it is better than decomposing the function. This is because the number of functions in the network does not increase using support reduction, whereas decomposition introduces new nodes in the network. However, it may not always be possible to make a function feasible using support reduction techniques.

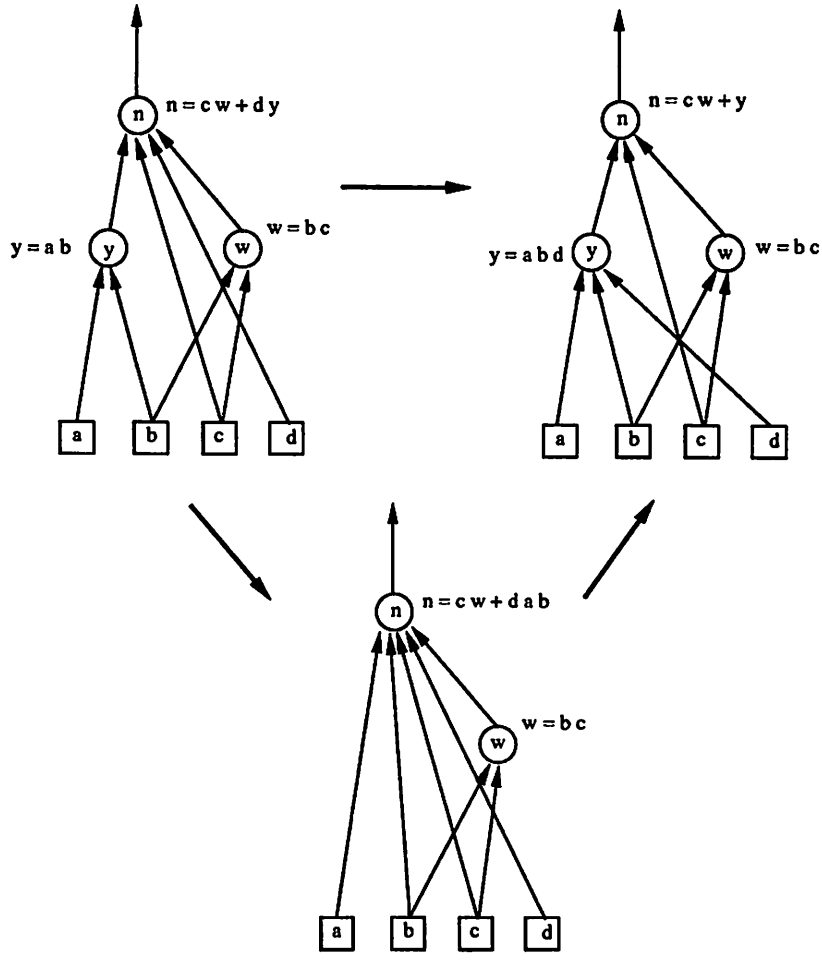


Figure 3.22: Collapse y into n and redecompose n

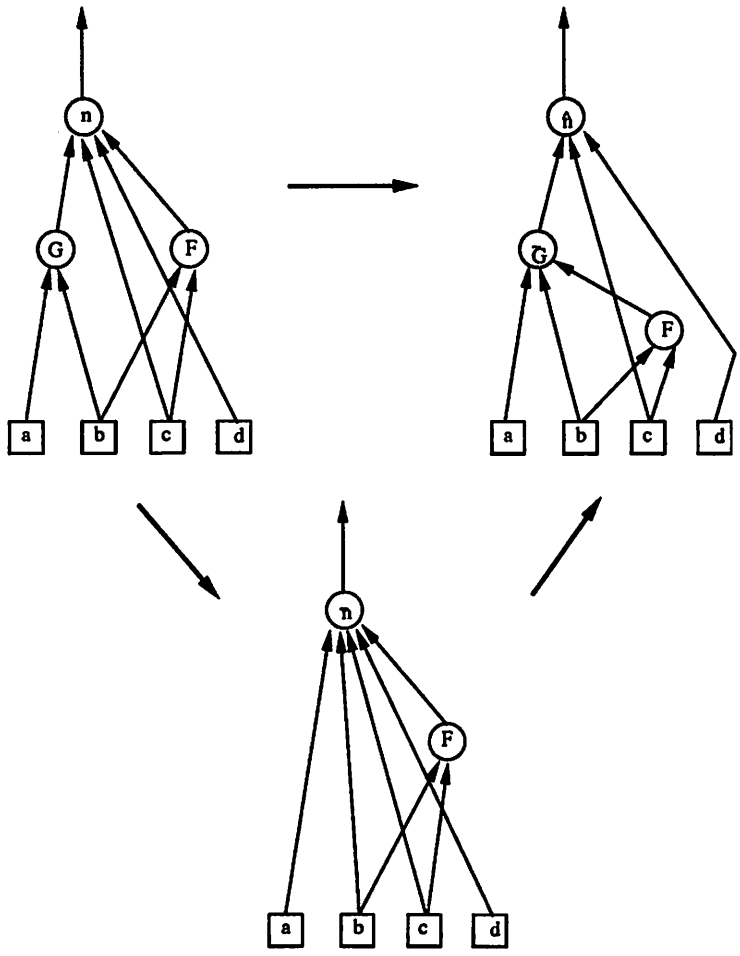


Figure 3.23: Collapse G into n and redecompose n with F in the bound set

3.3.7 Summary

We studied various decomposition techniques - functional, cube-packing, cofactoring, kernel extraction, and simple AND-OR decomposition. A question to ask is: *"When should we apply a particular technique?"* Unfortunately, we do not have a complete understanding of the general case yet. However, for some special classes of functions, it is possible to predict the technique that gives best results. For instance, cube-packing generates optimum tree implementations for functions having cubes with disjoint supports for $m \leq 5$. Similarly, functional decomposition works well for symmetric functions, since finding a good input partition is easy for such functions. However, for an arbitrary function, it is not known *a priori* which method would work well. By choosing functions appropriately, we have shown that no single method works in all cases. In Section 3.6, we will see that applying cube-packing on both SOP and factored form, and picking the better of the two gives reasonably good results.

3.4 Block Count Minimization

After decomposition/support reduction, an m-feasible network is obtained, which can be implemented straightaway on the LUTs by mapping each node to an LUT. This strategy, however, yields sub-optimal results.

Example 3.4.1 *Consider the following optimized network η, with one primary output f, five primary inputs $a, b, c, d,$ and e, and three internal nodes x, y and f:*

$$\begin{aligned} f &= abx' + a'b'x; \\ x &= cy + c'y'; \\ y &= d' + e'; \end{aligned}$$

Let m be 5. Now map η onto the target LUT *architecture. Since each function in η is 5-feasible, decomposition and support reduction have no effect on η. So we need 3* LUT*s. However, if y is collapsed into x, and then x is collapsed into f, we get*

$$f = abcde + abc'd' + abc'e' + a'b'cd' + a'b'ce' + a'b'c'de,$$

which is 5-feasible. So, one LUT *is required.*

We study two transformations that reduce the number of feasible nodes.

1. Collapse nodes into their fanouts while maintaining feasibility. This is called **covering.**[7]
2. Move the fanins of the nodes to create opportunities for collapsing. This is called **support reduction.**

3.4.1 Covering

The covering problem can be stated as:

Problem 3.4.1 *Given an m-feasible Boolean network η, iteratively collapse nodes into their fanouts such that the resulting network $\tilde{\eta}$ is m-feasible and the number of nodes in $\tilde{\eta}$ is minimum.*

By *iteratively*, we mean that a node can be collapsed into its fanouts, which, in turn, can be collapsed into their fanouts, and so on. Also, a node may be collapsed into some or all of its fanouts. We first present an exact method and then some heuristics.

Exact formulation of the covering problem

We first introduce the notion of a supernode.

Definition 3.4.1 *Given a graph $G = (V, E)$ and $U \subseteq V$, the* **induced subgraph** *of G on U is (U, E_1), where $E_1 = \{(u_1, u_2) | u_1, u_2 \in U$ and $(u_1, u_2) \in E\}$. In other words, (U, E_1) is G restricted to U.*

Definition 3.4.2 *A* **supernode** *corresponding to a node n of the network η is an induced directed subgraph S of η, with a single root n such that S does not contain any primary input node of η.*

Let us now define the support of a supernode by considering its inputs. Recall that a node v is an input to a DAG $G = (V, E)$ if $v \notin V$ and there exists a $u \in V$ such that $(v, u) \in E$. In other words, v is an input to G if it is outside G and has an edge incident on G. The set of all inputs to G forms the support σ of G. This also defines the support of a supernode, since a supernode is also a graph.

One interpretation of the support of a supernode corresponding to a node n in the context of a network η can be stated as follows. Add a dummy node s to η. Add edges from s to all the primary inputs. Then the nodes in σ form an (s, n) **node cut-set**, i.e., any path from s to n goes through some node in σ.

[7]This usage does not further overload *cover*. As shown in Section 3.4.1, the collapsing problem is the same as that of deriving a minimum cost *cover* of the subject graph in the standard technology mapping.

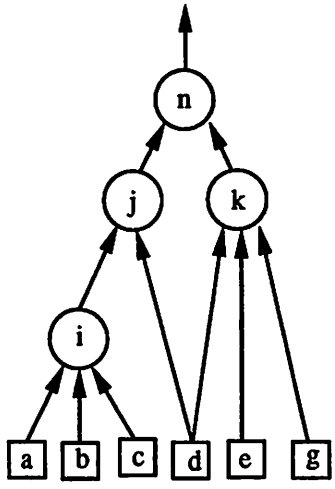


Figure 3.24: A node (e.g., n) can have more than one supernode

Definition 3.4.3 *A supernode $\mathcal{S}$ is* m**-feasible** *if its support cardinality is at most m, i.e., $|\sigma(\mathcal{S})| \leq m$.*

A node may have more than one m-feasible supernode. For example, the node n of Figure 3.24 has the following 5-feasible supernodes:

supernode	support
$\{n\}$	$\{j,k\}$
$\{n,j\}$	$\{i,d,k\}$
$\{n,k\}$	$\{d,e,g,j\}$
$\{n,j,k\}$	$\{i,d,e,g\}$
$\{n,j,i\}$	$\{a,b,c,d,k\}$

Note that $\{n,j,k,i\}$ is not a 5-feasible supernode, since its support has 6 members: a, b, c, d, e, and g. Also, $\{n,i,k\}$ is not a supernode, since it has two roots - n and i.

Being an induced subgraph of η, $\mathcal{S}$ is completely determined by its set of nodes, and we will use these two terms interchangeably. Also, given $\mathcal{S}$, its support is uniquely determined - it is simply the set of inputs to $\mathcal{S}$. As the following proposition shows, the converse is also true, i.e., a supernode is completely determined by its root and its support.

Proposition 3.4.1 *Given two supernodes $\mathcal{S}_1$ and $\mathcal{S}_2$, both rooted at n and having support sets σ_1 and σ_2 respectively, $\sigma_1 = \sigma_2 \Rightarrow \mathcal{S}_1 = \mathcal{S}_2$.*

Proof Suppose $\mathcal{S}_1 \neq \mathcal{S}_2$. Without loss of generality, there is a node $p \in \mathcal{S}_1 - \mathcal{S}_2$. Consider a directed path P from p to n that lies within $\mathcal{S}_1$ (such a path exists since n is the only root of $\mathcal{S}$). Let

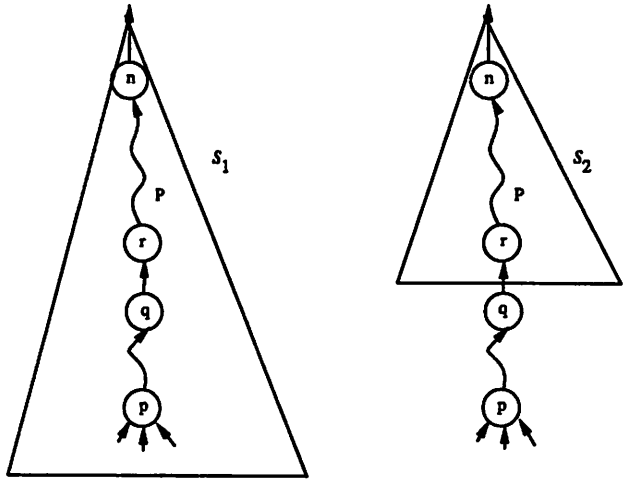


Figure 3.25: Given the root n and support of the supernode S, S is unique

q be the node on P that is closest to n and is in S_1 but not in S_2. Since $q \neq n$, let r be the fanout of q on this path (Figure 3.25). r is in both S_1 and S_2. Then $q \in \sigma_2 - \sigma_1$, resulting in a contradiction. ■

Given the root n and the support σ of a supernode S, the procedure of Figure 3.26 determines the set of nodes in S. It traverses the transitive fanin of n until it hits a support node, or a primary input not in the support, in which case no supernode is possible. From Proposition 3.4.1, this set of nodes is unique.

One way of solving Problem 3.4.1 is the following.

1. Enumerate all possible m-feasible supernodes.
2. Select a minimum subset of these supernodes that covers the network.

Enumerating m-feasible supernodes One way is to first generate, for each node n, all sets σ having at most m nodes, all from the TFI of n. Then, using the algorithm of Figure 3.26, check for each set σ whether it corresponds to some supernode rooted at n. If so, the algorithm returns the supernode S. Note that even if the algorithm of Figure 3.26 returns a supernode S, σ may not be a supernode support. σ may have a node i that is not needed for S, i.e., i does not have a path to n. The *correct* subset of σ can be found by first determining S and then finding the support of S. This formulation requires generating all possible sets with at most m nodes and then checking their validity for supernode support.

```
/* Given the root n ∈ η and the support σ of S, determine S */
/* initialize S = {n} */
/* assume all the nodes are unmarked in the beginning */
determine_supernode_from_support(n, σ)
{
     FI(n) = set of fanins(n);
     for each F ∈ FI(n)
     {
          if (F ∈ σ) continue;
          if (F ∈ PI(η)) return ''no supernode possible'';
          if F is marked continue;
          S = S ∪ {F};
          mark F;
          determine_supernode_from_support(F, σ);
     }
}
```

Figure 3.26: Determining a supernode from its root and support

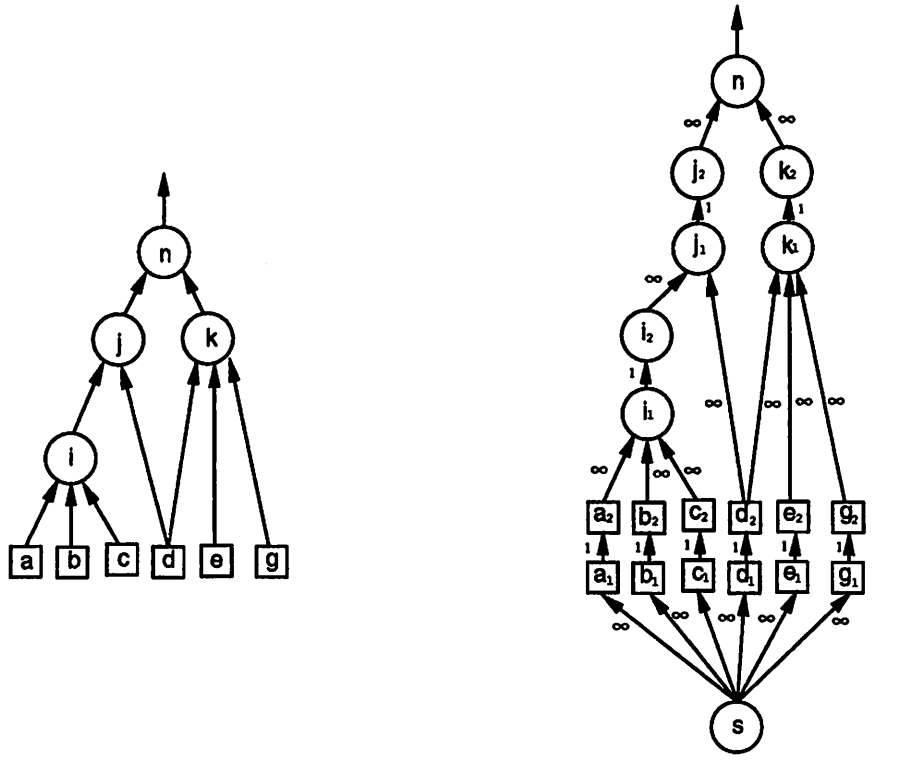


Figure 3.27: Constructing the flow network (B) for a Boolean network (A)

An alternate solution is based on network flows and generates only the valid supernode supports. Let $\eta(n)$ represent η restricted to the transitive fanin of n. We obtain a flow network $\mathcal{F}(n)$ by modifying $\eta(n)$ as follows. Let $n = n_1 = n_2$. We add a new node s, called the **source**. Each node j ($j \neq s, j \neq n$) is split into nodes j_1 and j_2. An edge (k, j) in $\eta(n)$ is replaced by (k_2, j_1) and is assigned a capacity of ∞. An edge (j_1, j_2) is added with a capacity of 1. For each primary input i, an edge (s, i_1) is added with a capacity of ∞. The node n, also called t, is designated as the **sink** in the flow network. Let the capacity of any edge $e = (u, v)$ be denoted as $c(e) = c(u, v)$. For the network of Figure 3.24, which is reproduced in Figure 3.27 (A), the flow network $\mathcal{F}$ is shown in Figure 3.27 (B).

A **cut** in a flow network is a partition of the nodes $(X, \overline{X})$ such that $s \in X, t \in \overline{X}$. The

capacity of a cut $(X, \overline{X})$ is defined as

$$C(X, \overline{X}) = \sum_{u \in X, v \in \overline{X}} c(u, v)$$

Define the support set σ of a node n to be a set of nodes in the transitive fanin of n such that n can be expressed in terms of the nodes in σ. A support set σ need not be minimal, i.e., it may have a proper subset $\tilde{\sigma}$ that is also a support set of n. We now show that there is a one-to-one correspondence between the support sets of a node n of a network η and the finite-capacity cuts in the corresponding flow network $\mathcal{F}(n)$.

Proposition 3.4.2 *There is a one-to-one correspondence between the support sets of a node n of a network η and the finite-capacity cuts in the corresponding flow network $\mathcal{F}(n)$.*

Proof Given a support set σ for n, it is straightforward to construct a finite-capacity cut. First we construct the supernode S in η using the procedure of Figure 3.26. Let $\tilde{\sigma}$ be the support of S (note that $\tilde{\sigma} \subseteq \sigma$, where the equality holds if and only if σ is minimal). Then let

$$\begin{aligned} \overline{X} &= \{n\} \cup \{i_1 | i \neq n, i \in S\} \cup \{i_2 | i \neq n, i \in S\} \cup \{j_2 | j \in \tilde{\sigma}\} \\ X &= \mathcal{F}(n) - \overline{X} \end{aligned}$$

This is a finite-capacity cut, since the only edges going from X to $\overline{X}$ are of the form $(j_1, j_2), j \in \tilde{\sigma}$, and are of a capacity 1. In fact, $C(X, \overline{X}) = |\tilde{\sigma}|$.

Conversely, given a finite-capacity cut $(X, \overline{X})$, we can generate a

$$\sigma = \{j | j_1 \in X, j_2 \in \overline{X}\}$$

First note that since $(X, \overline{X})$ has finite capacity, and the only edges with finite capacity in $\mathcal{F}(n)$ are of the form (i_1, i_2), it makes sense to define σ. The function at n can be represented using just the variables in σ, since any path from s to n passes through some j in σ (if there were a path P from s to n that did not pass through any j in σ, it can be shown using finiteness of the capacity of the cut that either $s \in \overline{X}$ or $n \in X$). ■

In fact, there is a one-to-one correspondence between the minimal support sets of n and the minimal finite-capacity cuts in the flow network (a finite-capacity cut is minimal if the corresponding σ is minimal). The minimal support sets of n of the Boolean network η of Figure 3.27 (A) and the corresponding finite-capacity cuts of $\mathcal{F}(n)$ of Figure 3.27 (B) are as follows:

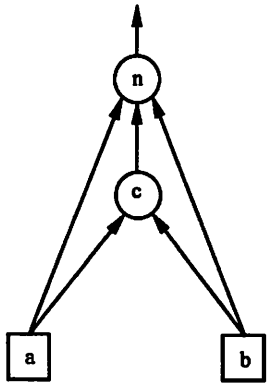


Figure 3.28: Minimal support sets of n do not generate all supernodes

minimal support set	$\overline{X}$ of the corresponding minimal finite-capacity cut $(X, \overline{X})$
$\{j, k\}$	$\{n, j_2, k_2\}$
$\{j, d, e, g\}$	$\{n, j_2, k_1, k_2, d_2, e_2, g_2\}$
$\{i, d, k\}$	$\{n, j_1, j_2, i_2, d_2, k_2\}$
$\{i, d, e, g\}$	$\{n, j_1, j_2, k_1, k_2, i_2, d_2, e_2, g_2\}$
$\{a, b, c, d, e, g\}$	$\{n, j_1, j_2, k_1, k_2, i_1, i_2, a_2, b_2, c_2, d_2, e_2, g_2\}$

We will use an algorithm to generate finite-capacity cuts that are minimal. This means that only minimal support sets are considered. Since our goal is to generate all feasible supernodes, we can ask if all minimal support sets of n generate all supernodes rooted at n. The answer is no. For example, consider the network of Figure 3.28. The only minimal support set of n is $\{a, b\}$, and it corresponds to the supernode $\{n, c\}$. The supernode $\{n\}$ is not generated, since its support is $\{a, b, c\}$, which is not minimal. It turns out that the optimality of the solution is unaffected if only the minimal support sets of n are considered. The reason is explained shortly.

We repeatedly invoke a *maxflow-mincut algorithm* [47] to generate different minimal cuts and minimal support sets therefrom. This is done by the algorithm of Figure 3.29. First, a flow network $\mathcal{F}$ is generated for the entire network. For each node n, the flow network $\mathcal{F}(n)$ is obtained by designating n_1 as the sink node in $\mathcal{F}$ and therefore only considering the TFI of n_1 in $\mathcal{F}$. The maxflow-mincut algorithm is invoked on $\mathcal{F}(n)$. If the value of the maximum flow f is at most m, the corresponding support set σ is derived from the minimum cut using the method in the proof of Proposition 3.4.2 and added to Γ, the set of support sets. Note that σ is a minimal support set, since the cut generated by the flow algorithm is a minimum cut. To generate other support sets, a forward edge (going from X to $\overline{X}$) is suppressed, i.e., its capacity is set to ∞. This makes sure that the edge is not included in the cut generated in the next invocation of the maxflow-mincut

procedure, and so the new cut is different. If the value of the maximum flow is higher than m, all the edge capacities in $\mathcal{F}(n)$ are re-initialized to the values they had when $\mathcal{F}(n)$ was formed. A suppress-matrix $SM = (s_{ij})$ is formed. It represents the relationship between the support sets (generated so far) and the nodes of the Boolean network. Each row corresponds to a support set and each column to a node of the Boolean network. Each entry is defined as follows:

$$s_{ij} = \begin{cases} 1 & \text{if support set } i \text{ includes node } j \\ 0 & \text{otherwise} \end{cases} \tag{3.20}$$

At any point in the algorithm, we do not want to generate a support set that has already been generated. To this end, a column cover C of SM is derived. It represents a set of nodes j such that suppressing the edges (j_1, j_2) ensures that each support set in Γ is suppressed, i.e., no support set in Γ is generated in the next invocation of the maxflow-mincut procedure.

The complexity of the maxflow algorithm is $O(|V||E| \log \frac{|V|^2}{|E|})$ using Goldberg and Tarjan method [32], where $|V|$ and $|E|$ are the numbers of nodes and edges in the flow network respectively.

The next proposition ensures that all m-feasible, minimal support sets are generated.

Proposition 3.4.3 *The procedure* `generate_all_feasible_minimal_support-sets` *(η, n, m) returns in Γ all the m-feasible, minimal support sets for the node n.*

Proof We observe the following.

1. Each invocation of the maxflow-mincut procedure generates a minimum cut in $\mathcal{F}(n)$, with some edges possibly suppressed, i.e., it generates the minimum number of forward edges from X to $\overline{X}$ given the suppressed edges. This implies that the corresponding support set is minimal.

2. Just before `generate_all_feasible_minimal_support-sets` returns Γ, it goes through all the column covers of SM, each generating a cut of capacity greater than m.

Assume, for the sake of contradiction, that at the end of the procedure, some m-feasible, minimal support set $\tilde{\sigma}$ is not generated. A support set $\sigma \in \Gamma$ can be suppressed without including any node of $\tilde{\sigma}$, because otherwise, σ does not have any nodes different from $\tilde{\sigma}$, which implies that $\tilde{\sigma}$ is not minimal, contradicting the first observation. Repeating this for all support sets in Γ, we get a column cover C of SM that does not include any node of $\tilde{\sigma}$. The invocation of maxflow-mincut on $\mathcal{F}$ (with edges corresponding to C suppressed) cannot return a minimum cut with capacity greater

```
generate_all_feasible_minimal_support-sets(η, n, m)
{
    /* Γ is the set of feasible support sets */
    Γ = φ
    F = derive_flow_network(η, n)
    while (TRUE) {
        (f, cut) = maxflow-mincut(F)
        if (|f| > m) {
            reinitialize_edge_capacities(F)
            SM = form_suppress-matrix(Γ)
            flag = 0
            foreach_column-cover_C_of_SM {
                suppress_edges(C, F)
                (f, cut) = maxflow-mincut(F)
                if (|f| ≤ m) {
                    flag = 1
                    break
                }
                unsuppress_edges(C, F)
            }
            if (flag == 0) {
                /* no more feasible support sets */
                return Γ
            }
        }
        σ = generate_support_set(cut)
        Γ = Γ ∪ {σ}
        suppress_a_forward-edge_in_cut(cut)
    }
    /* notreached */
}
```

Figure 3.29: Generating all m-feasible support sets for n

than m, since at least one cut with capacity at most m exists in $\mathcal{F}$. This contradicts the second observation. ■

From Γ, we generate m-feasible supernodes. Given that there are p nodes in the network (including primary inputs and primary outputs), the maximum number of m-feasible supernodes for one node is $C(p, m)$ (which is the notation for the number of all possible ways of choosing m objects out of p). The total number of m-feasible supernodes is bounded by $pC(p, m)$. Since m is a constant, this number is a polynomial in the size of the network. Although our supernode generation algorithm is not one of the most efficient, we found it to be quite fast even on large networks.

Selecting a minimum subset Having generated the set of all feasible supernodes (more precisely, the subset corresponding to the minimal support sets), our task is to select a minimum subset $\mathcal{M}$ of supernodes that satisfies the following constraints:

1. **output constraints:** for each primary output p^o, at least one supernode with p^o as the root must be chosen.

2. **implication constraints:** if a supernode $\mathcal{S}$ is in $\mathcal{M}$, each input of $\mathcal{S}$ should be either a primary input or an output of some supernode in $\mathcal{M}$.

Note that the cost of each supernode is one, since it can be implemented by one LUT. That is why we are interested in selecting a minimum subset of the supernodes. This problem is known as the **binate covering problem.** This formulation is similar to the one used by Rudell for technology mapping in his thesis [68].[8] There is one difference however. In addition to the two types of constraints, Rudell's formulation requires a **covering constraint** to be satisfied for each internal node of the network. The covering constraint for an internal node n requires that n must belong to some supernode in $\mathcal{M}$, i.e., some supernode in $\mathcal{M}$ must cover n. It is easy to see that the covering constraints are not necessary.

Proposition 3.4.4 *Satisfaction of the output and implication constraints guarantees satisfaction of the covering constraints.*

Proof Let $\mathcal{M}$ satisfy the output and implication constraints for a network η. Consider an internal node n of the Boolean network η. If n fans out to a primary output node p^o, its covering is ensured

[8]For the general problem of technology mapping, the term **match** is used instead of supernode; we will use the two interchangeably.

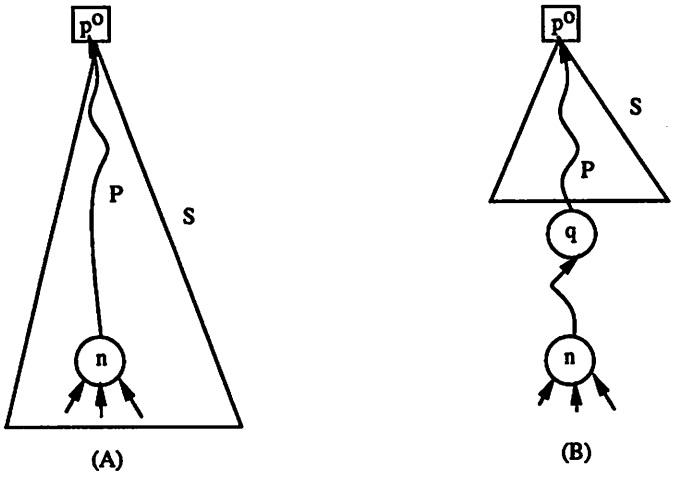


Figure 3.30: The covering constraints are not needed

by the output constraint for p^o. Otherwise, there is at least one path P from n to some output p^o of η (if none exists, n can be deleted from η). Since $\mathcal{M}$ satisfies the output constraints, there exists a supernode $\mathcal{S} \in \mathcal{M}$ with root p^o. Either $\mathcal{S}$ includes n, as shown in Figure 3.30 (A) - in which case the covering constraint for n is satisfied, or there exists a node q lying on P that belongs to the support of $\mathcal{S}$. Since $q \in TFO(n)$, it is not a primary input (Figure 3.30 (B)). The implication constraints require that some supernode with root q be in $\mathcal{M}$. Repeating this argument, eventually n will be covered by some supernode in $\mathcal{M}$. This means that $\mathcal{M}$ satisfies the covering constraints. ■

However, experiments show that adding the covering constraints helps the heuristics in obtaining somewhat better approximate solutions. Of course, their presence does not affect the exact solution. Therefore, in the following discussion, the covering constraints are treated along with the output and implication constraints.

We can now explain why a non-minimal support set of n can be ignored without affecting the quality of the covering solution. Let $\sigma = \{a, b, c, d\}$ be a non-minimal support for n, and $\mathcal{S}$, the corresponding supernode. Let $\sigma_1 = \{a, b, c\}$ be a minimal support subset of σ, and $\mathcal{S}_1$, the corresponding supernode. We are better off selecting $\mathcal{S}_1$ as compared to $\mathcal{S}$ because the selection of $\mathcal{S}$ in the final cover imposes the following constraint:. if d is not a primary input, one of the supernodes that has d as the root must be selected. Clearly, selection of $\mathcal{S}_1$ does not impose this constraint.

Formulation of binate covering The constraints are represented in a 0-1 matrix $\mathcal{B}$. The columns of $\mathcal{B}$ correspond to the supernodes (or matches), and the rows to the nodes of the network. Let n be a node of the network and $\mathcal{S}$ a supernode. Let i be the row of $\mathcal{B}$ corresponding to n, and j be the column corresponding to $\mathcal{S}$. $\mathcal{B}(i,j) = 1$ if and only if $\mathcal{S}$ covers n. We then say that the column j covers the row i. We do likewise for each node n of the network. This takes care of the covering constraints. This part of the matrix can be deleted if the covering constraints are not considered explicitly.

To handle the implication constraints, additional rows and columns are added to $\mathcal{B}$. For each supernode $\mathcal{S}$, consider all its inputs. For each input n, an additional row $\hat{\imath}$ is added. $\mathcal{B}(\hat{\imath},\hat{\jmath})$ is set to 1 for all the columns (supernodes) $\hat{\jmath}$ that have n as the root. The implication constraint imposes that whenever $\mathcal{S}$ is selected in $\mathcal{M}$, $\hat{\imath}$ has to be covered by some column in $\mathcal{B}$. However, the covering problem is formulated in such a way that all the rows of $\mathcal{B}$ have to be covered by some column *in any case.* It suffices to ensure that $\hat{\imath}$ is covered automatically if $\mathcal{S}$ is not selected. This is done by introducing an extra column $\hat{k}$ (called an **anti-supernode** or **anti-match**) for the match $\mathcal{S}$, which has a cost of 0. We set $\mathcal{B}(\hat{\imath},\hat{k}) = 1$. Note that the cost of the match $\mathcal{S}$ is 1.

Finally, the output constraints are handled by adding a row for each primary output. Let i be the row for the output p^o. Then $\mathcal{B}(i,j) = 1$ if the supernode corresponding to the column j is rooted at p^o. Otherwise, $\mathcal{B}(i,j) = 0$.

The problem then is to find a minimum cost cover of $\mathcal{B}$ by the columns such that out of a column and its anti-match, exactly one is selected. One way of deriving the exact cover is to use a branch and bound technique. Lower and upper bounds on the optimum cover are derived. If a supernode $\mathcal{S}$ is selected, the rows it covers are deleted, and the problem is solved recursively on the resulting sub-matrix. This solution is compared with the best one obtained if $\mathcal{S}$ is not selected and the better one is picked.

The binate covering problem can also be formulated as that of obtaining a minimum-cost implicant of a satisfiability problem [68], where an uncomplemented literal has a cost of 1 and a complemented literal has a cost of 0. For each supernode $\mathcal{S}_i$, a Boolean variable x_i is introduced. The constraints generate clauses (sum terms) of a Boolean function as follows:

- *Covering constraints*: If a node is covered by supernodes $\mathcal{S}_2$, $\mathcal{S}_5$, and $\mathcal{S}_9$, write a clause $(x_2 + x_5 + x_9)$. Repeat this for each node of the network.
- *Output constraints*: Given a primary output n_i, let $\mathcal{S}_{i1}, \mathcal{S}_{i2}, \ldots, \mathcal{S}_{ij}$ be the supernodes rooted at n_i. Then the output constraint for n_i can be expressed by $(x_{i1} + x_{i2} + \cdots + x_{ij})$.

- *Implication constraints*: Let supernode S_i have nodes $n_{i_1}, \ldots, n_{i_p}$ as p inputs. If S_i is chosen, one of the supernodes that realizes n_{i_j} must also be chosen for each input j that is not a primary input. Let N_{i_j} be the disjunctive expression in the variables x_k giving the possible supernodes that realize n_{i_j} as an output node. Selecting supernode S_i implies satisfying each of the expressions N_{i_j} for $j = 1, 2, \ldots, p$. This can be written as

$$\begin{aligned} & x_i \Rightarrow (N_{i_1} N_{i_2} \cdots N_{i_p}) \\ \Leftrightarrow \quad & x_i' + (N_{i_1} N_{i_2} \cdots N_{i_p}) \\ \Leftrightarrow \quad & (x_i' + N_{i_1})(x_i' + N_{i_2}) \cdots (x_i' + N_{i_p}). \end{aligned}$$

  The clauses are generated for each supernode likewise.

Take the product of all the clauses generated above to form a product-of-sums expression. Any satisfying assignment to this expression is a solution to the binate covering problem. Finding a satisfying assignment with the least total cost is the same as finding a least cost prime. Mathony has proposed a branch and bound technique [54] for generating all the primes of a function. It uses efficient pruning of the search tree. This technique can be used to find a prime with the minimum cost. This is the technique implemented in SIS by Lavagno [45], and used by us. Recently, Lin and Somenzi [50] showed that the least cost prime of a function f can be determined by constructing a ROBDD for f (with any input ordering) and then finding the shortest path (one with the minimum number of uncomplemented edges) from its root to the terminal 1 vertex.

Heuristic methods Binate covering is an NP-complete problem. The exact formulations presented above are time-intensive even on moderate-size networks. Enumerating all feasible supernodes is fast; the hard part is selecting the minimum subset that satisfies all the constraints. So we resort to heuristic methods.

1. The most straightforward heuristic is a greedy one. It selects a supernode that covers the maximum number of rows of the matrix B and deletes the rows covered by the supernode and the columns corresponding to the supernode and the corresponding anti-supernode. The procedure is repeated on the reduced matrix.

2. This heuristic is a simple variation of the previous one. It has two phases.

   - *Phase 1*: satisfaction of the covering and the output constraints.

- *Phase 2*: satisfaction of the implication constraints.

Phase 1: We satisfy the covering and the output constraints in the best possible way, i.e., using the minimum number of supernodes (matches). This is the well-known **unate covering problem.** It has been studied extensively in the context of solving the two-level minimization problem. Although it is NP-complete, rather efficient ways to solve it are known [31, 68]. Let the solution to the unate covering be the set of matches $\mathcal{M}$.

Phase 2: $\mathcal{M}$ generates a set of implication constraints, constraining the union of the inputs of the supernodes in $\mathcal{M}$ to be the outputs (of a supernode). If already some of these constraints are satisfied by supernodes in $\mathcal{M}$, they are deleted. If no more constraints are left, the procedure terminates. Otherwise, a score λ for each supernode $\mathcal{S} \notin \mathcal{M}$ is calculated as follows. Let $sel(\mathcal{S})$ be the number of remaining constraints that are satisfied if $\mathcal{S}$ is selected (these constraints correspond to the outputs of $\mathcal{S}$), and $not\_sel(\mathcal{S})$ the number of constraints that are satisfied if $\mathcal{S}$ is not selected (these constraints correspond to the inputs of $\mathcal{S}$). Then

$$\lambda(\mathcal{S}) = sel(\mathcal{S}) - not\_sel(\mathcal{S})$$

The supernode with the maximum score is selected and added in $\mathcal{M}$. We update the constraint set, and repeat this process until all the implication constraints are satisfied. This heuristic gave better results on most of the benchmarks as compared to the greedy heuristic.

3. **Partition** The heuristics presented above generate all feasible supernodes, and make fast and possibly non-optimal selections for the supernodes. *Partition* generates only a subset of feasible supernodes - those formed by collapsing a node into its immediate fanout(s). Hence, it looks only one level ahead.

Definition 3.4.4 *A collapse (n, f^o), f^o a fanout of n, is called an* m**-feasible collapse** *if f^o is m-feasible after n is collapsed into it.*

Some of the criteria that *partition* uses while collapsing nodes are as follows.

(a) A node n is collapsed into all of its immediate fanouts, provided all the collapses are feasible.

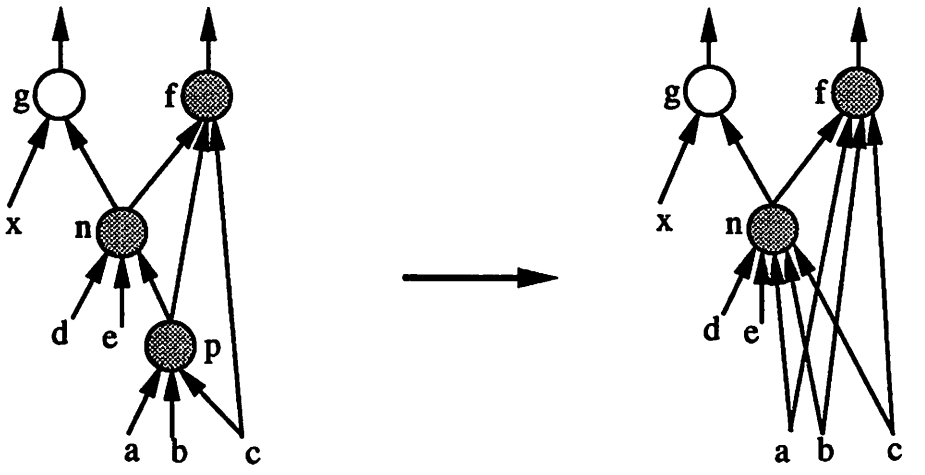


Figure 3.31: Creation of new edges on collapsing

(b) This option is targeted for routing. Let n be a node and f^o be one of its fanouts. If this collapse is feasible, a pair (n, f^o) is formed. All pairs for the entire network are formed likewise. A cost is associated with each pair. It is the number of new edges created after the collapse. This is the same as the fanins of n which are not fanins of f^o. Figure 3.31 shows the creation of new edges when n is collapsed into f and g. At each step of the algorithm, the pair with the least cost is selected and collapsed. This may render a collapse that was feasible earlier infeasible. So the costs are revised after each collapse. Note that this option does not really target the minimization of the number of blocks.

From now on in this chapter, we will use the term *cover* or *covering* to mean the binate covering formulation.

3.4.2 Support Reduction

The covering problem described in the last section is *structural*, in that it ignores the logic function present at a node. The only information it uses is the number of fanins of each node and the node connectivity. Support reduction is fundamentally different, in that it is both structural and functional. It also makes use of the function present at a node. It is based on the same idea as that of Section 3.3.6. Consider a node n that cannot be collapsed into its fanouts during *partition*, as some of its fanouts become infeasible. If the local support of n can be reduced such that n collapses feasibly into its fanouts, n can be deleted from the network, thereby reducing the number of feasible

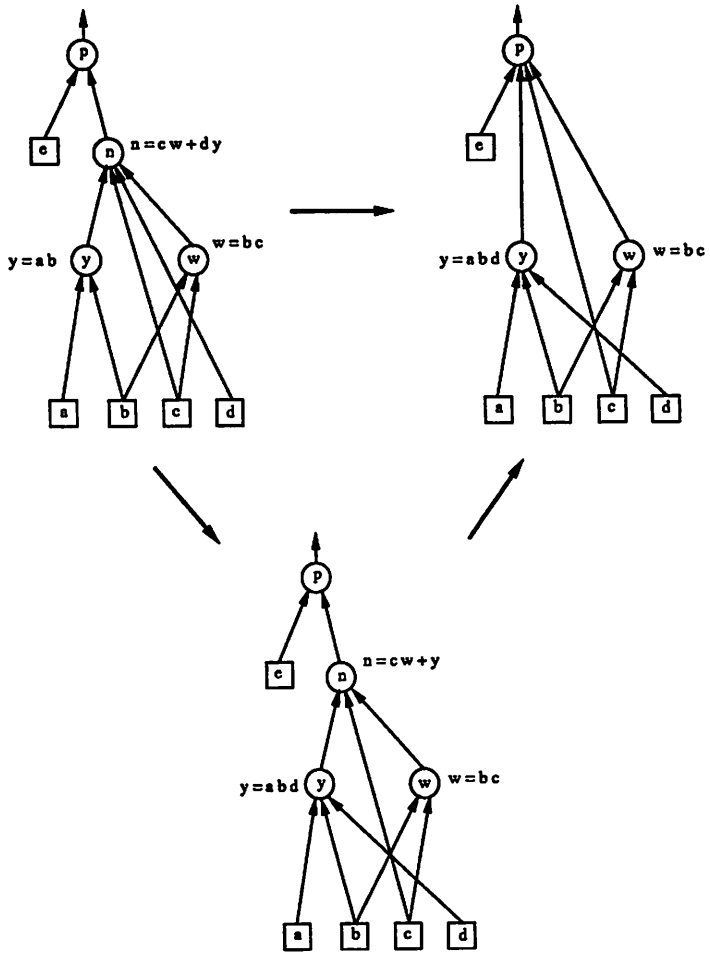


Figure 3.32: Support reduction by fanin movement to achieve collapsing

nodes.

Example 3.4.2 *Consider the network of Figure 3.32. It is a slight variation of the example of Figure 3.21 - an extra node p is present. Let m be 4. The network is 4-feasible. Consider node n - it cannot be feasibly collapsed into p. However, if the support of n is reduced, as was done in Example 3.3.12, we get* $n = cw + y$. *Now, n can be collapsed into p, thus reducing the number of nodes by 1.*

This method has been integrated with *partition*, such that if a node n cannot be feasibly collapsed into all its fanouts, an attempt is made to reduce its support, so that it can be collapsed.

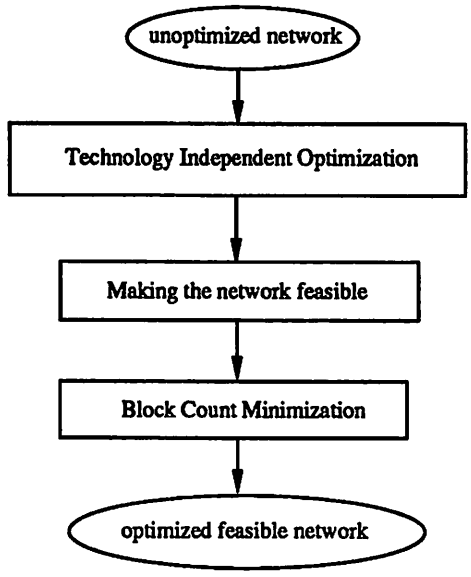


Figure 3.33: A straightforward mapping approach

3.5 The Overall Algorithm

Having presented the two main components of the mapping algorithm, we address the issue of how to use them in a complete synthesis framework. As shown in Figure 3.33, one straightforward way is to first make all infeasible nodes feasible (say, by decomposing them), and then apply block count minimization on the resulting network. However, this approach suffers from the following problems:

1. If the network is large, in the BCM step exact covering methods cannot be applied to the entire network. Even the heuristic methods meet with limited success. Lacking a global view of the network, they make greedy, non-optimal decisions.

2. The covering technique is limited in its applicability: it works only on feasible networks. It will be better if somehow it could be extended to an infeasible network.

It turns out that an **interleaving approach**, in which decomposition and block count minimization are applied node-by-node, which is then followed by **partial collapse**, gives better results than the approach of Figure 3.33. This is shown in Figure 3.34. Block count minimization on the sub-network resulting from decomposition of a node can be often applied in the exact mode, since the sub-network is generally small. However, the node-by-node mapping paradigm alone does

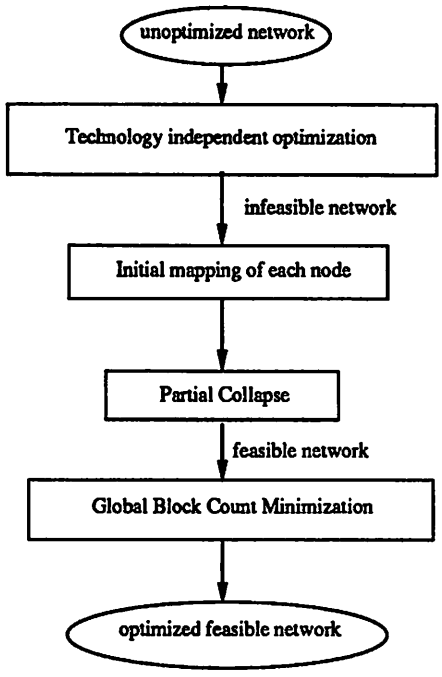


Figure 3.34: The overall algorithm

not exploit the structural relationship *between* the nodes of the network. *Partial collapse* achieves exactly that by collapsing each node into its fanouts, remapping the fanouts (i.e., decomposing them and minimizing the block count), and computing the gain from this collapse.

Example 3.5.1 *An example of partial collapsing is given in Figure 3.35. Let m be 5. Node d is collapsed into its fanout nodes i and j. Let the cost of node i before the collapse be 3* LUT*s. After the collapse, let its cost remain unchanged (after collapsing, i has 6 inputs and can be realized with 3 5-*LUT*s using cofactoring). Nodes d and j have a cost of 1 each before collapsing, as they are 5-feasible. Also, j remains feasible after the collapse. The total gain from this collapse is* $(1+3+1)-(3+1) = 1$. *Note that* partition *would not have accepted the collapse of node d into i, since i remains infeasible after the collapse.*

How can we formulate the *partial collapse* problem? First, consider the problem in its full generality:

Problem 3.5.1 *Given a possibly infeasible network η and a procedure* LUT_cost(f, m) *that computes the cost of a function f in terms of m-*LUT*s needed to implement it, collapse nodes such that the cost of the resulting network is minimum.*

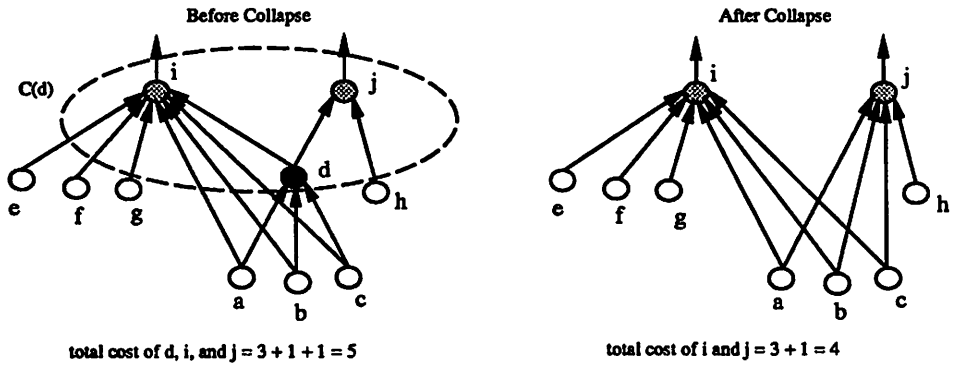


Figure 3.35: *Partial collapse* and forming clusters

This problem is a generalization of the covering problem, since it operates on infeasible networks as well. Note that it is more difficult than *covering*. The basic unit in *covering* is an m-feasible supernode, characterized by its support σ, which should have at most m elements. However, determining if it is beneficial in an infeasible network to treat a subgraph rooted at a node as one (infeasible) supernode is not an easy task - we have to evaluate the cost of the supernode by mapping it.

An exact way of solving Problem 3.5.1 is as follows.

1. Enumerate all m-feasible and m-infeasible supernodes of the network η. For each supernode S, do the following. If f is the global function corresponding to S, i.e., f is expressed using the support of S, determine the cost of S using LUT_cost(f, m).

2. Solve a binate covering problem, similar to the one in Section 3.4, except that instead of selecting a minimum subset of supernodes, select a subset with the minimum cost.

Although a network has only a polynomial number of m-feasible supernodes, it can have an exponential number of supernodes. For example, a complete binary tree with root r and p nodes has $O(2^p)$ supernodes rooted at r.[9] Solving a binate covering problem of this size is hard. So we look for approximations. For example, we may consider only a proper subset of supernodes. One possibility is to consider only supernodes with a bounded depth. The simplest case is a bound of two, which means a node and all its fanins. An interesting formulation is obtained using a simple variation: collapse a node into its fanouts. Let $\omega(i)$ be the LUT cost of an internal node i. For each

[9]Enumerating all the supernodes in a network is the same as finding all the (singly) rooted subgraphs of a directed acyclic graph.

fanout j of i that is not a primary output, we introduce a 0-1 variable x_{ij} defined as follows:

$$x_{ij} = \begin{cases} 1 & \text{if } i \text{ is selected for collapsing into } j, \\ 0 & \text{otherwise.} \end{cases} \tag{3.21}$$

Let $\tilde{\omega}(i,j)$ denote the cost of the node j after i has been collapsed into it. Let

$$\delta(i,j) = \omega(j) - \tilde{\omega}(i,j) \tag{3.22}$$

$\delta(i,j)$ denotes the reduction in the cost of j after i is collapsed into it. Note that $\tilde{\omega}(i,j)$, and hence $\delta(i,j)$, depend on the logic functions at j and i. Our goal is *to select node-fanout pairs* (i,j) *that can be simultaneously collapsed such that the resulting gain is maximized.* However, arbitrary collapses are not allowed - the following constraints need to be satisfied:

1. If we decide to collapse i into j, we should not choose to collapse a node $k \in FI(i)$ into i. This is because both the collapses involve node i. After k is collapsed into i, the logic function at i changes, and so does the cost of i: $\omega(i)$ becomes $\tilde{\omega}(k,i)$. As a result, $\delta(i,j)$ changes. This is not taken into account in our *simultaneous-collapse*-based formulation.

2. For the same reason, simultaneous collapses of the kind (i,j) and $(j,k), k \in FO(j)$, are prohibited.

3. Simultaneous collapses of the kind (k,i) and (l,i), where $k,l \in FI(i)$, are prohibited.

Note that if a node i is collapsed into all its fanouts, it can be deleted from the network, resulting in an additional gain of $\omega(i)$. The aim then is to maximize the gain by selecting nodes for simultaneous *partial collapse* subject to the above constraints. This leads to the following non-linear programming formulation:

$$\textbf{maximize} \sum_{i \in IN(\eta)} [(\sum_{j \in FO(i) \cap IN(\eta)} \delta(i,j) x_{ij}) + (\prod_{j \in FO(i) \cap IN(\eta)} x_{ij}) \, \omega(i)]$$

subject to

$$\begin{aligned} x_{ki} + x_{ij} &\leq 1, \forall\, i,j,k \\ \sum_j x_{ji} &\leq 1, \forall\, i \\ x_{ij} &\in \{0,1\} \end{aligned}$$

This is a 0-1 integer program with a non-linear objective function. It may be converted into a non-linear program by replacing the 0-1 constraint by an equivalent quadratic constraint $x_{ij}^2 = x_{ij}$. The modified program is:

$$\textbf{maximize} \sum_{i \in IN(\eta)} [(\sum_{j \in FO(i) \cap IN(\eta)} \delta(i,j) x_{ij}) + (\prod_{j \in FO(i) \cap IN(\eta)} x_{ij}) \, \omega(i)]$$

subject to

$$\begin{aligned} x_{ki} + x_{ij} &\leq 1, \quad \forall\, i,j,k \\ \sum_j x_{ji} &\leq 1, \quad \forall\, i \\ x_{ij}^2 &= x_{ij}, \quad \forall\, i,j \end{aligned}$$

A linear integer programming formulation is obtained if instead of forming (node, fanout) pairs, we group a node n and all its non-primary-output fanouts in one cluster $C(n)$. We say that $C(n)$ corresponds to n. Either n is collapsed into all its fanouts (in which case, we say that $C(n)$ has been collapsed), or into none. The cost of a cluster $C(n)$, $\omega(C(n)) = \sum_{i \in C(n)} \omega(i)$. After n is collapsed into its fanouts, the saving in the cost is

$$\delta(C(n)) = \omega(C(n)) - \sum_{j \in FO(n) \cap IN(\eta)} \tilde{\omega}(n,j) \tag{3.23}$$

If $\delta(C(n)) > 0$, $C(n)$ is called a **good cluster** and n is a candidate for collapsing. We compute the cost savings for every cluster and then retain only the good clusters.

Let x_i be the 0-1 variable corresponding to a good cluster $C(i)$.

$$x_i = \begin{cases} 1 & \text{if } C(i) \text{ is selected for } \textit{partial\_collapse}, \\ 0 & \text{otherwise.} \end{cases} \tag{3.24}$$

Let $A = (a_{ij})$ be the node-cluster incidence matrix, i.e.,

$$a_{ij} = \begin{cases} 1 & \text{if node } i \text{ belongs to good cluster } C(j), \\ 0 & \text{otherwise.} \end{cases}$$

The problem can then be formulated as

$$\textbf{maximize} \sum_{i \text{ s.t.} C(i) \text{ is good}} \delta(C(i)) \; x_i$$

subject to

$$\sum_j a_{ij} x_j \leq 1 \quad \forall i$$

There are $|IN(\eta)|$ clusters, one cluster corresponding to each internal node. So there are at most $|IN(\eta)|$ good clusters, and hence variables. Since at most one constraint is generated for each internal node of η, the linear integer program has at most $|IN(\eta)|$ variables and as many constraints. The solution of this program generates clusters that should be *partial_collapsed* for maximum gain. In the solution, if an x_i is 1, the node n_i to which the cluster C_i corresponds, is *partially_collapsed* into its non-primary-output fanouts. After collapsing all such clusters, we get a new network $\tilde{\eta}$ and can apply *partial collapse* on $\tilde{\eta}$. The process can be repeated until no more gain is possible.

The formulations presented so far are either non-linear or linear integer programs. In general, both are intractable problems [30]. So we resort to simple heuristics, one of which is to select good clusters in some order, say topological - from inputs to outputs. If the cluster corresponding to the node being visited is good, it is collapsed. After finishing one pass over the network, the procedure is repeated until no good clusters remain. For the sake of efficiency, we do not visit each node of the network in the next iteration. We keep track of the potential nodes for future collapsing. For example, consider a node n that does not have a good cluster in some iteration, but one of its fanouts gets collapsed. In the next iteration, it is possible that n has a good cluster by virtue of the modified fanout set. It is easily seen that when n is collapsed into its fanouts, the following nodes are affected as far as the chances of further collapses are concerned: the fanins of n, the fanouts of n, and the fanins of the fanouts of n. This is called the **affected_set(n)**. In Figure 3.35, affected_set(d) = $\{a, b, c, i, j, e, f, g, h\}$. Say, after one pass, S is the set of nodes collapsed. Then only the nodes in $\cup_{n \in S}$ affected_set(n) are considered for collapse in the next iteration.

A slight improvement of the above greedy strategy is the following. Instead of collapsing a node i either into all the fanouts or into none, we allow for the possibility of a *finer granularity* collapse. Given a node i, we partition the set of its fanouts into two subsets:

1. the good set G consisting of all the fanouts j of i such that $\delta(i, j) > 0$, and
2. the bad set B consisting of fanouts j such that $\delta(i, j) \leq 0$.

It makes sense to collapse i into j when $j \in G$. In general, i should not be collapsed into $j, j \in B$. However, if i is collapsed into each fanout in B, i can be deleted from the network (assuming i did

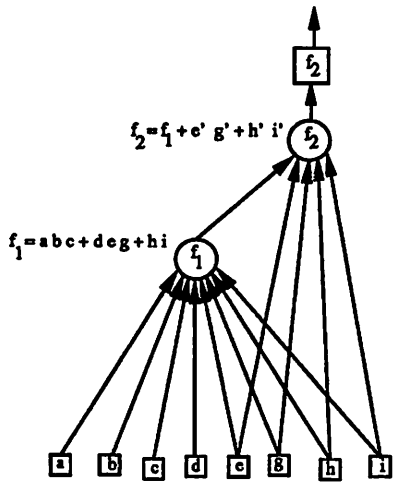


Figure 3.36: A simple network to illustrate LUT mapping

not fan out to a primary output), resulting in a saving of $w(i)$. Note that i is collapsed into each good fanout in any case. In other words, we should collapse i into B if and only if i does not fan out to a primary output and

$$w(i) + \sum_{j \in B} \delta(i,j) > 0.$$

We make the following remarks.

1. *partial collapse* explores more alternatives than *partition*, since it also explores collapses into fanout nodes that become infeasible after the collapse. This routine may be thought of as an LUT analogue of `eliminate -1` in misII.

2. We discovered that in all the benchmark examples we used, only the collapsing of feasible nodes contributed to the gain. This reduces the number of nodes that need to be considered for *partial collapse*. The run time is cut down without significantly degrading the quality of the results.

3. We found that a greedy heuristic like the one proposed above is much faster than an integer-programming based method and does not sacrifice quality of results in general.

3.5.1 An Example

We illustrate some of the techniques described in this chapter with the help of a simple example network η of Figure 3.36. The network η has eight primary inputs - a, b, c, d, e, g, h, i, one

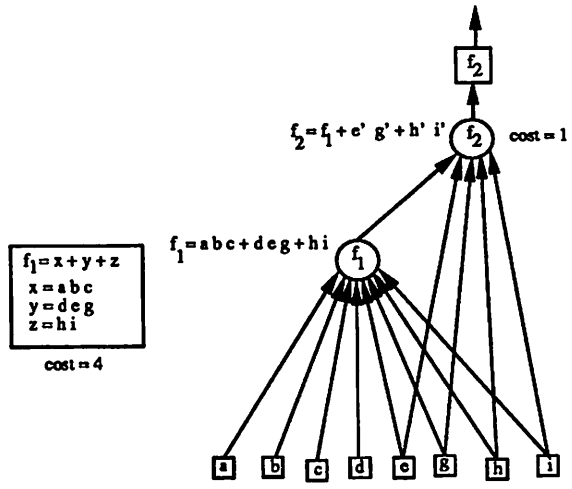


Figure 3.37: After initial decomposition

primary output f_2, and two internal nodes - f_1 and f_2. Let m be 5. Assume that the representations shown for f_1 and f_2 are optimized ones. In the mapping phase, we carry out the following steps:

1. *Map each node*

   (a) *Make each function 5-feasible*: Nothing needs to be done to f_2, since it is already 5-feasible. However, $|\sigma(f_1)| = 8$, so f_1 needs to be made 5-feasible. If we were to apply cube-packing on it, we obtain the optimum tree realization

$$x = deg + hi$$
$$f_1 = abc + x.$$

   For the purpose of illustration, suppose we did not apply cube-packing, but chose a simple AND-OR decomposition, which creates a sub-function for each cube and then ORs the sub-functions. The following decomposition is generated:

$$x = abc$$
$$y = deg$$
$$z = hi$$
$$f_1 = x + y + z.$$

   This sub-network, call it η_1, is attached with f_1. The resulting configuration is shown in Figure 3.37. The cost of η_1 is 4 LUTs, and that of η is 5.

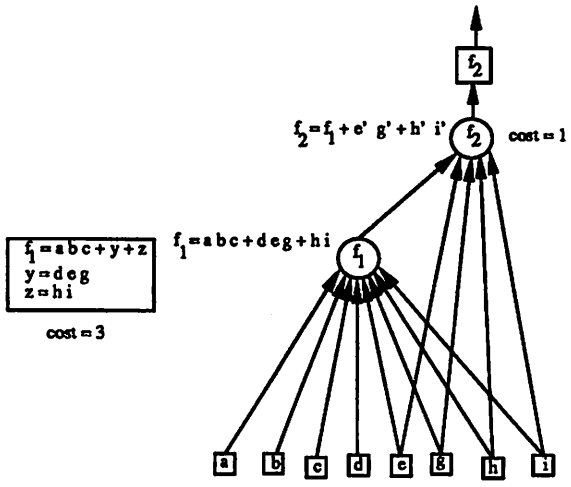


Figure 3.38: After block count minimization (covering)

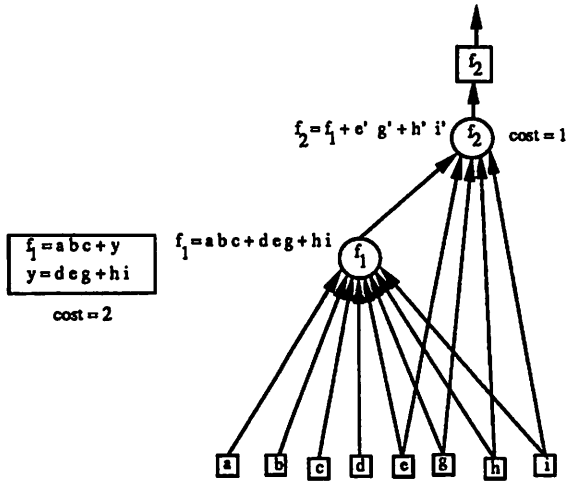


Figure 3.39: After block count minimization (support reduction and collapsing)

(b) *Block count minimization*: On η_1, we first apply covering. This causes x to collapse into f_1 resulting in η_2 (Figure 3.38):

$$\begin{aligned} y &= deg \\ z &= hi \\ f_1 &= abc + y + z. \end{aligned}$$

No more collapses are possible. So we try to reconfigure η_2 by applying the support reduction techniques. Without going through all the intermediate steps (which are similar to those of Figure 3.32), it is seen that the connection from z to f_1 can be removed and replaced by a connection from z to y. As a result, the functions at f_1 and y change:

$$\begin{aligned} y &= deg + z \\ z &= hi \\ f_1 &= abc + y. \end{aligned}$$

Now, z can be collapsed into y, yielding the configuration of Figure 3.39:

$$\begin{aligned} y &= deg + hi \\ f_1 &= abc + y. \end{aligned}$$

Note that for this example the resulting decomposition is the same as generated by cube-packing on f_1.

2. *Partial collapse*: To exploit the relationship between the nodes of η, f_1 is collapsed into f_2. The resulting configuration is shown in Figure 3.40. Carrying out a similar mapping step on the new f_2, we find out that it can be realized in two 5-LUTs. Since the cost of the network has improved as a result, the collapse is accepted. Note that this is an optimum solution for this example network, since $|\sigma_{GT}(f_2)| > 5$.

3.6 Experimental Results

3.6.1 Description of Benchmarks

For validating various ideas and algorithms presented in this and other chapters, we use a benchmark set. The circuits in this set come from the 1991 MCNC logic synthesis benchmark

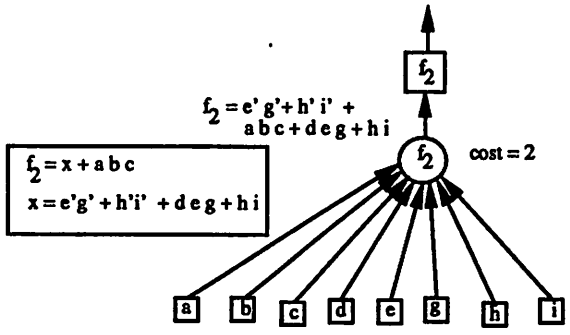


Figure 3.40: After *partial collapse*: the final network

set [90]. Table 3.1 provides some information about these benchmarks. If the functionality of a benchmark is not known, the word "Logic" is used. The last three columns in the table refer to numbers of nodes, edges, and literals in the factored form in the area-optimized networks. The optimized networks are obtained by running twice the multi-level optimization script *script.rugged* [73] provided with SIS [78], with a timeout limit of 1 hour for each run. Unless mentioned otherwise, these optimized benchmarks are the ones used in this thesis for combinational mapping.

We first show results for decomposition, then for decomposition followed by block count minimization, and finally for *partial collapse*.

3.6.2 Decomposition

Cube-packing

We ran cube-packing on the optimized benchmarks using different options mentioned in Section 3.3.2. A 5-LUT was chosen as the target. The results are shown in Table 3.2. No significant difference in the quality of results is seen between all the options, although using the *minimum increment in the support* definition for the best bin, along with the *smart* literal extraction gives slightly better results than the rest of the options.

3.6.3 Decomposition and Block Count Minimization

Here, the results obtained after decomposition and BCM are presented.

example	origin	function	pi	po	nodes	edges	lits (fac)
5xp1	MCNC-tl	Logic	7	10	18	77	114
9sym	MCNC-tl	Count Ones	9	1	12	46	145
C1355	MCNC-ml	Error Correcting	41	32	162	344	552
C1908	MCNC-ml	Error Correcting	33	25	146	383	535
C2670	MCNC-ml	ALU and Control	233	140	152	528	748
C3540	MCNC-ml	ALU and Control	50	22	225	1076	1264
C432	MCNC-ml	Priority Decoder	36	7	52	209	219
C5315	MCNC-ml	ALU and Selector	178	123	374	1335	1763
C6288	MCNC-ml	16-bit Multiplier	32	32	113	2829	3367
C7552	MCNC-ml	ALU and Control	207	108	499	1490	2288
alu2	MCNC-ml	ALU	10	6	54	273	347
alu4	MCNC-ml	ALU	14	8	59	495	893
apex2	MCNC-tl	Logic	39	3	43	219	268
apex3	MCNC-tl	Logic	54	50	200	1361	1567
apex7	MCNC-ml	Logic	49	37	61	218	243
b9	MCNC-ml	Logic	41	21	30	115	124
bw	MCNC-tl	Logic	5	28	35	155	160
clip	MCNC-tl	Logic	9	5	16	82	117
cordic	MCNC-ml	Logic	23	2	11	43	64
dalu	MCNC-ml	Dedicated ALU	75	16	120	778	881
des	MCNC-ml	Data Encription	256	245	508	2661	3319
duke2	MCNC-tl	Logic	22	29	81	392	428
e64	MCNC-tl	Logic	65	65	116	253	253
ex4	MCNC-tl	Logic	128	28	51	241	456
f51m	MCNC-ml	Arithmetic	8	8	16	50	80
k2	MCNC-ml	Logic	45	45	134	1254	1343
misex2	MCNC-tl	Logic	25	18	25	103	104
rd84	MCNC-tl	Logic	8	4	18	71	148
rot	MCNC-ml	Logic	135	107	167	597	664
sao2	MCNC-tl	Logic	10	4	18	93	131
spla	MCNC-tl	Logic	16	46	130	570	598
t481	MCNC-tl	Logic	16	1	11	26	36
vg2	MCNC-tl	Logic	25	8	10	65	88
z4ml	MCNC-ml	2-bit Add	7	4	9	32	43

Table 3.1: Description of example circuits

origin source of the example (tl = two-level, ml = multi-level)
function description of the circuit function (when available)
pi number of primary inputs
po number of primary outputs
nodes number of internal nodes in the optimized network
edges number of edges in the optimized network
lits (fac) number of literals in factored form in the optimized network

example	regular-order		smart-order	
	sup	*incr*	*sup*	*incr*
5xp1	31	31	31	31
9sym	46	46	46	46
C1355	164	164	164	164
C1908	154	154	153	153
C2670	290	289	282	281
C3540	477	475	477	475
C432	93	93	92	92
C5315	477	477	477	477
C6288	1161	1161	1161	1161
C7552	705	705	705	705
alu2	102	102	101	101
alu4	286	285	280	280
apex2	101	100	98	98
apex3	500	500	500	500
apex7	68	68	68	68
b9	47	47	47	47
bw	46	46	46	46
clip	31	31	31	31
cordic	17	17	17	17
dalu	281	281	281	281
des	951	949	950	949
duke2	138	137	138	137
e64	116	116	116	116
ex4	216	216	224	224
f51m	19	19	19	19
k2	402	401	397	397
misex2	33	33	33	33
rd84	43	43	43	42
rot	223	223	224	224
sao2	47	47	46	46
spla	202	202	202	202
t481	11	11	11	11
vg2	24	24	24	24
z4ml	12	12	12	12
total	7514	7505	7496	7490

Table 3.2: Issues in implementing cube-packing

regular-order order inputs arbitrarily
smart-order order inputs based on frequency of occurrence
sup best bin is the one with *minimum support*
incr best bin is the one with *minimum increment in the support*
total sum of 5-LUT counts over all examples

Roth-Karp decomposition and partition

On the optimized networks, the mapping script used is Roth-Karp decomposition, followed by *partition* with support reduction. The following two implementations of Roth-Karp decomposition are compared:

- *serial encoding*: The equivalence classes are encoded serially, that is, the equivalence class C_j is assigned the code corresponding to the binary representation of j.
- *good encoding*: An input encoding algorithm [69] is used to encode the classes. The algorithm is run with a cost function of minimizing the number of literals. Unused codes are used as don't cares to simplify the image of the decomposition, g.

Roth-Karp decomposition is invoked on any function with greater than 5 inputs. It chooses the first partition (X, Y) such that $|X| \leq 5$. If a disjoint decomposition is not found, the implementation switches to another decomposition method that guarantees feasibility.

Table 3.3 shows the comparison. The good encoding scheme gives 8.4% overall better results. Also, there are benchmarks where it completes, and the serial one does not. Note that most of the improvement is on larger benchmarks. This is because in the smaller benchmarks, most functions are simple, with small number of inputs. So the function g obtained after decomposition is mostly m-feasible, and encoding scheme does not make much difference. On the other hand, in the larger benchmarks, typically functions have more inputs, so g is infeasible. Then doing a good encoding does help when g is further decomposed.

Although not reported, the number of literals is also minimized in roughly the same proportion as the number of LUTs using the encoding formulation.

Cube-packing and partition

The mapping script was cube-packing followed by *partition* with support reduction. Three experiments were performed.

- *sop*: Applied the mapping script on the optimized networks.
- *fac: td*: Applied the mapping script after applying *decomp -g; tech-decomp -a 2 -o 2* on the optimized networks. *decomp -g* (i.e., good decomposition) produces a factored form of each node, and *tech-decomp -a 2 -o 2* (technology decomposition) breaks up each node into

example	good enc	serial enc
5xp1	34	41
9sym	41	41
C1355	92	92
C1908	103	103
C2670	298	353
C3540	518	641
C432	119	124
C5315	597	642
C6288	536	542
C7552	540	562
alu2	136	129
alu4	318	-
apex2	89	98
apex3	-	-
apex7	58	60
b9	52	57
bw	60	58
clip	71	87
cordic	14	14
dalu	249	249
des	1059	1127
duke2	256	304
e64	81	81
ex4	243	224
f51m	27	25
k2	700	-
misex2	33	34
rd84	39	51
rot	269	285
sao2	75	90
spla	318	450
t481	5	5
vg2	28	29
z4ml	14	14
total	7072	6612
subtotal	6054	6612

Table 3.3: Encoding schemes for Roth-Karp decomposition

good enc Roth-Karp decomp. with good encoding, then *partition*
serial enc Roth-Karp decomp. with serial encoding, then *partition*
- could not finish
subtotal sum of 5-LUT counts for examples where both complete

2-input AND and OR gates. Cube-packing does nothing on these networks, since they are already 5-feasible.

- *fac: no td*: Applied the mapping script after applying *decomp -g* on the optimized networks. This, in essence, means that cube-packing is applied on the factored form of each node.

Table 3.4 shows the 5-LUT counts after mapping. Clearly, factoring helps. This is as expected, since cube-packing operates on an SOP, which could be considerably larger than the factored form. Also, as seen from *fac:td*, it is not a good idea to break down the network into two-input gates. As already pointed out in Section 3.3.5, one reason is that there may be too many two-input gates, and BCM algorithms cannot handle large networks. These experiments also show that it is a good idea to use a decomposition method (such as cube-packing) targeted specifically for the LUT architectures.

Improved decomposition and partition

In the last set of experiments, cube-packing and technology decomposition were the decomposition techniques used. If better or more or both decomposition techniques are applied, better quality results may be produced. To test this hypothesis, for each internal node n, first a network $\eta(n)$ is constructed. $\eta(n)$ has one primary output, one internal node corresponding to n, and as many primary inputs as there are fanins of n. Various techniques, including cube packing on the SOP, cube packing on the factored form (using *decomp -g*), and cofactoring are used to make $\eta(n)$ m-feasible. Let the resulting network be $\widetilde{\eta}_t(n)$ for each decomposition technique t. BCM is invoked on $\widetilde{\eta}_t(n)$, yielding $\widehat{\eta}_t(n)$. The best feasible implementation out of all $\widehat{\eta}_t(n)$ is selected (this is the one with the minimum number of m-feasible nodes) and replaces node n. After all the nodes have been processed, BCM is applied on the entire network to exploit the relationship among the nodes of the network.

The results are shown in Table 3.5. The following notation is used:

- *fac cpack*: Use *decomp -g* before using the mapping script, which is cube-packing followed by *partition* with support reduction. Snapshots of the mapper are taken after cube packing (column *decomp*) and then after *partition* (column *partition*). The column *partition* is the same as the column **fac: no td** of Table 3.4.

- *best decomp*: For each node n, use various decomposition techniques on $\eta(n)$. For this set of experiments, cube packing on SOP, cube packing after *decomp -g*, and cofactoring

example	sop	fac	
		td	*no td*
5xp1	29	29	26
9sym	46	39	39
C1355	100	102	76
C1908	102	137	100
C2670	237	180	149
C3540	422	449	305
C432	90	63	64
C5315	371	544	366
C6288	467	766	510
C7552	553	685	409
alu2	101	124	99
alu4	279	309	210
apex2	95	102	79
apex3	496	608	483
apex7	54	60	54
b9	45	37	36
bw	37	37	37
clip	30	30	25
cordic	13	11	10
dalu	262	358	232
des	865	1478	855
duke2	132	148	128
e64	81	82	81
ex4	224	156	143
f51m	14	18	17
k2	390	457	362
misex2	30	32	29
rd84	42	46	39
rot	206	202	177
sao2	44	37	37
spla	188	213	178
t481	5	5	5
vg2	24	23	23
z4ml	8	10	6
total	6082	7578	5389

Table 3.4: Cube packing and partition

sop apply cube-packing and *partition*
fac: *td* apply *decomp -g; tech-decomp -a 2 -o 2* before *partition*
fac: *no td* apply *decomp -g* before cube-packing and *partition*
total sum of 5-LUT counts over all the examples

example	fac cpack		best decomp	
	decomp	*partition*	*decomp*	*partition*
5xp1	34	26	30	28
9sym	43	39	40	39
C1355	168	76	164	100
C1908	165	100	151	99
C2670	225	149	196	147
C3540	416	305	351	299
C432	83	64	73	67
C5315	545	366	469	362
C6288	1391	510	1161	467
C7552	697	409	568	403
alu2	110	99	100	100
alu4	269	210	211	210
apex2	94	79	85	82
apex3	532	483	489	484
apex7	82	54	68	54
b9	50	36	41	39
bw	48	37	46	37
clip	37	25	31	30
cordic	15	10	14	10
dalu	288	232	254	229
des	980	855	945	861
duke2	148	128	133	128
e64	116	81	116	81
ex4	180	143	149	149
f51m	25	17	19	14
k2	406	362	361	360
misex2	34	29	32	29
rd84	42	39	40	40
rot	243	177	201	180
sao2	45	37	40	37
spla	209	178	191	179
t481	11	5	11	5
vg2	28	23	23	23
z4ml	14	6	11	7
total	7773	5389	6814	5379

Table 3.5: Cube packing on factored form vs. best decomposition

fac cpack:*decomp* snapshot after *decomp -g* and cube-packing
fac cpack:*partition* after *decomp -g*, cube-packing, and *partition*
best decomp:*decomp* snapshot after *best decomp*
best decomp:*partition* after *best decomp* and global *partition*
total sum of 5-LUT counts over all the examples

were used. Snapshots were taken immediately after replacing each node n by the best $\widehat{\eta}(n)$ (column *decomp*) and then after global BCM - in this case *partition* (column *partition*).

As expected, the results for *best decomp* before *partition* is invoked are much better than for *fac cpack* before *partition* - the total LUT count of 6814 as compared to 7773. Surprisingly, after *partition*, this advantage is lost, and the results are almost identical.

3.6.4 Combining Everything: Using *partial collapse*

Finally, to couple decomposition and BCM more tightly, we use *partial collapse*. The following algorithm is used:

1. Do an initial mapping: First a sub-network $\eta(n)$ is formed from each node n. Cube-packing is applied on it - both on the SOP and on the factored form obtained by *decomp -g*. Then, *cover* in the exact mode is applied on each resulting sub-network if it has no more than 50 nodes. Otherwise, *partition* with support reduction is applied. The better of the two is selected.

2. Apply *partial collapse*. If each node of the network is considered for collapsing, *partial collapse* becomes time-intensive. So an issue to address is: "*Which nodes should be collapsed?*" We experimented with three options - collapse nodes with a cost of 1 LUT, at most 2 LUTs, and at most 3 LUTs.

3. If the circuit is small, collapse it and carry out Roth-Karp decomposition. This step helps if a function is symmetric, since symmetric functions are not checked for *per se*.

4. Apply global BCM in the exact mode if the network has at most 60 nodes, and in the heuristic mode if it has at most 200 nodes. Otherwise, do nothing.

5. Finally, *partition* equipped with support reduction is applied.

It turns out that in *partial collapse*, when only the nodes with cost one are collapsed, the results are almost identical to the case when nodes with cost at most two or three are also considered for collapsing. Difference in the quality of results is below 0.1%.

In Table 3.6, the LUT counts after *partial collapse* are compared with the best results we have thus far (column *best-decomp (partition)* of Table 3.5). Although *partial collapse* does not finish on some examples, it gives about 5% overall better results on those it finishes. On some benchmarks, e.g., *9sym* and *rd84*, the results obtained are much better. This is because these circuits are symmetric, and Roth-Karp decomposition works well on symmetric circuits.

example	part-coll	best decomp
5xp1	21	28
9sym	7	39
C1355	70	100
C1908	96	99
C2670	138	147
C3540	291	299
C432	61	67
C5315	356	362
C6288	481	467
C7552	372	403
alu2	97	100
alu4	-	210
apex2	79	82
apex3	-	484
apex7	54	54
b9	38	39
bw	28	37
clip	25	30
cordic	10	10
dalu	-	229
des	844	861
duke2	124	128
e64	81	81
ex4	139	149
f51m	11	14
k2	-	360
misex2	28	29
rd84	13	40
rot	177	180
sao2	35	37
spla	173	179
t481	5	5
vg2	22	23
z4ml	6	7
total		5379
subtotal	3882	4096

Table 3.6: Comparing *partial collapse* with *best decomp*

part coll	use *partial collapse* - collapse nodes with cost 1
best decomp	the *best-decomp* column of Table 3.5
-	*partial collapse* ran out of memory
total	sum of 5-LUT counts over all the examples
subtotal	sum of 5-LUT counts over examples where part coll finishes

3.6.5 Relating Factored Form Literals and LUTs

To see the relationship between the number of factored form literals and LUT-count, we plot in Figure 3.41 the number of 5-LUTs obtained in the last table using *partial collapse* against the number of literals in the optimized benchmarks, as reported in Table 3.1. Only those benchmarks are considered for which *partial collapse* could finish. It can be seen that the relationship is nearly linear, although many benchmarks lie below the best linear fit. The linear relationship can be explained by the fact that most of the techniques used in mis-fpga (for instance, cube-packing, *partition*, and *covering*) work on an optimized representation of the function. In the optimized representation, each variable appears nearly once on average,[10] and so each literal appears only once. The mapping techniques for LUT then pack m literals in each m-LUT.

To explain the deviant behavior shown by some benchmarks, we observe that out of the larger benchmarks, *C6288*, *C7552*, and *C5315* lie below the linear fit. From their function description in Table 3.1, it is seen that these benchmarks contain circuitry to perform arithmetic, which is efficiently represented using EX-OR gates. Traditional optimization techniques do not work well for such functions, whereas LUT-mapping handles them well, thus resulting in better implementations.

Interestingly, the average number of literals that can be placed in one 5-LUT is 4.8.

3.6.6 Comparing with Other Systems

In Table 3.7, the results from mis-fpga are compared with chortle-crf [26] and Xmap, an improved version of [39]. The starting optimized networks are identical for the systems. On benchmarks where all systems finish, looking at the row *all-finish*, mis-fpga using *partial collapse* is 9.6% better than chortle-crf and 16.8% better than Xmap. If we compute the percentage difference for each example, and then take the average of these differences over all examples, mis-fpga is 14.2% better than chortle-crf and about 16.1% better than Xmap. Figure 3.42 depicts the results graphically.

[10] The average number, 1.01, is obtained by dividing the total number of factored form literals by the total number of edges over all the benchmarks of Table 3.1.

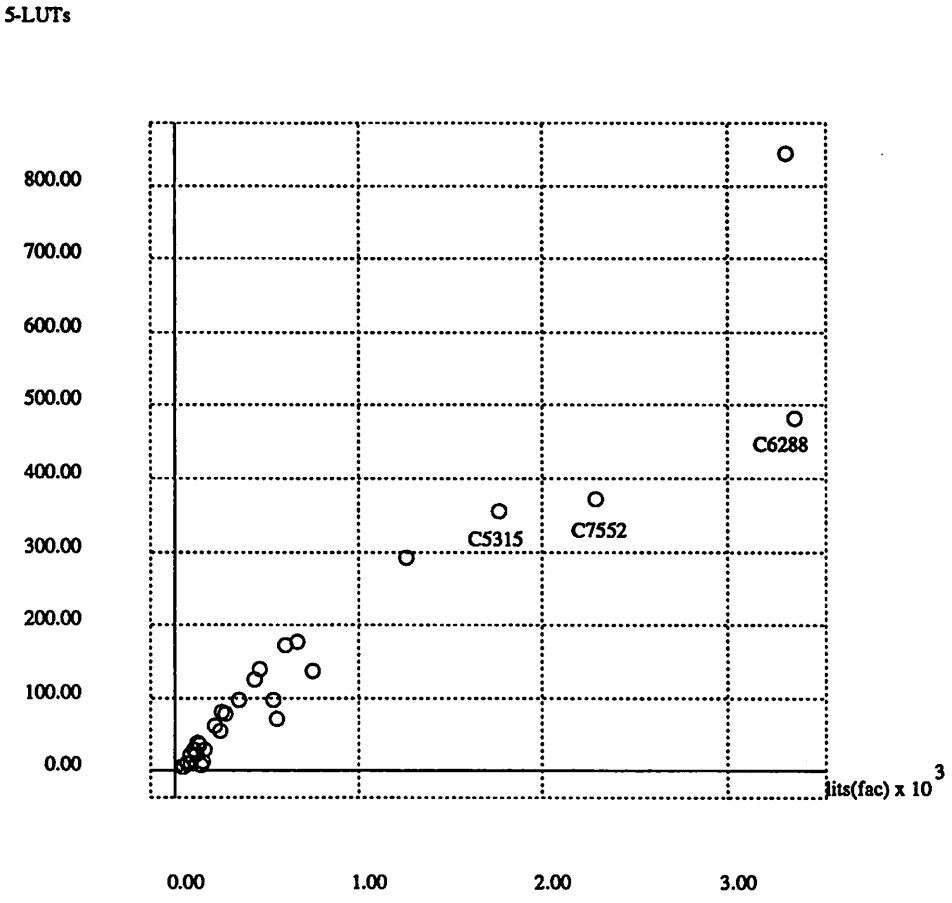


Figure 3.41: Relating factored form literals with 5-LUT-count

example	mis-fpga		chortle-crf	Xmap
	part-coll	*best decomp*		
5xp1	21	28	28	25
9sym	7	39	38	40
C1355	70	100	106	70
C1908	96	99	106	96
C2670	138	147	156	259
C3540	291	299	292	305
C432	61	67	62	62
C5315	356	362	376	427
C6288	481	467	604	658
C7552	372	403	456	488
alu2	97	100	95	103
alu4	-	210	203	214
apex2	79	82	78	83
apex3	-	484	468	527
apex7	54	54	56	64
b9	38	39	36	39
bw	28	37	36	-
clip	25	30	23	25
cordic	10	10	12	10
dalu	-	229	218	257
des	844	861	852	928
duke2	124	128	124	131
e64	81	81	81	83
ex4	139	149	139	150
f51m	11	14	17	19
k2	-	360	356	355
misex2	28	29	29	30
rd84	13	40	38	41
rot	177	180	-	198
sao2	35	37	33	38
spla	173	179	173	-
t481	5	5	11	5
vg2	22	23	23	23
z4ml	6	7	7	9
total	-	5379	-	-
part-coll-subtotal	3882	4096	-	-
chortle-subtotal	-	5199	5332	-
Xmap-subtotal	-	5163	-	5762
all-finish	3504	3700	3878	4211

Table 3.7: Comparing various systems

total sum of 5-LUT counts over all the examples
all-finish sum of 5-LUT counts over examples where every tool completes

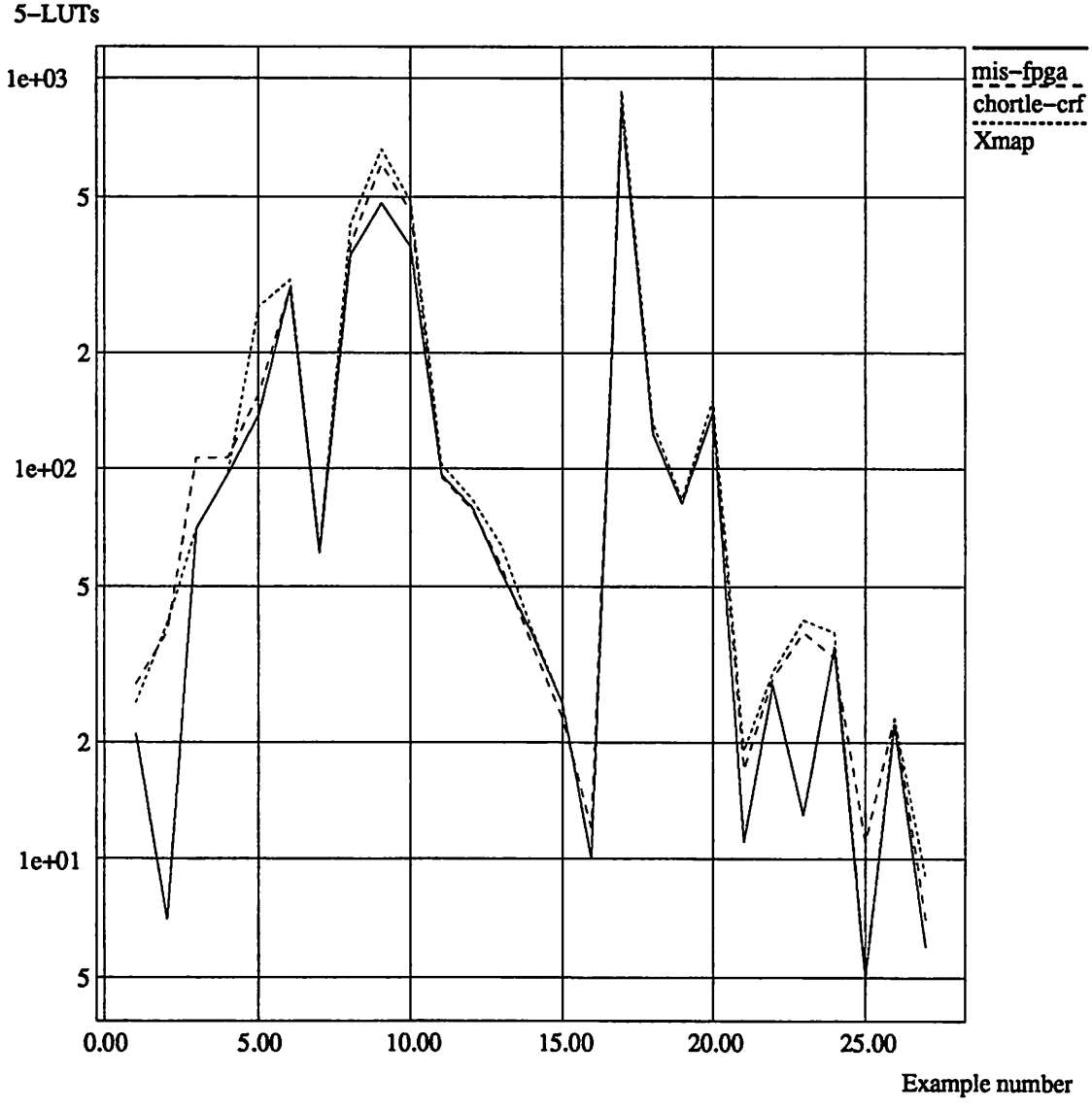


Figure 3.42: Comparing various systems

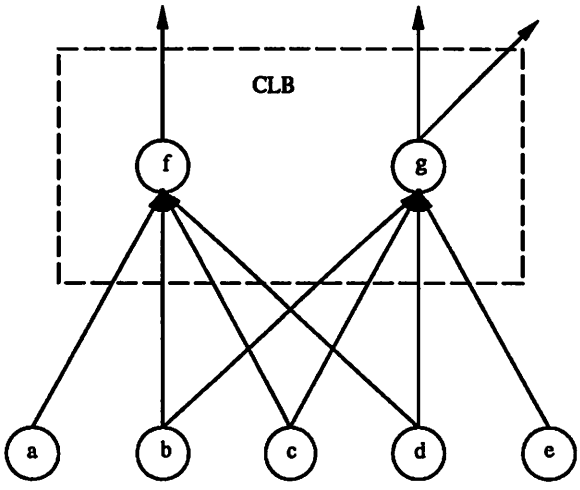


Figure 3.43: Example of mergeable functions

3.7 Targeting Xilinx 3090

The main focus of our work is to target a basic block with a single LUT. But commercial architectures come in slightly different flavors. For example, if we ignore the feedback paths from the flip-flops to the LUT-section (since we are targeting only combinational logic in this chapter) in Figure 1.5, a Xilinx 3090 CLB can implement either:

1. a 5-feasible function f, or
2. two 4-feasible functions f and g provided $|\sigma(f) \cup \sigma(g)| \leq 5$. f and g are then called **mergeable**. These conditions on the supports of the two functions are called the **combinational mergeability conditions** (CMCs). Figure 3.43 shows two mergeable functions.

For area minimization, the objective is to minimize the number of CLBs needed for a network. How do we handle the possibility of placing two functions in a CLB? Our approach is to first obtain a k-feasible network η ($k = 4$ or 5) using the techniques described in Section 3.3, and then apply a modified block count minimization. The BCM problem for Xilinx 3090 architecture is as follows:

Problem 3.7.1 *Given a k-feasible Boolean network η, iteratively collapse nodes or pair mergeable nodes or do both such that the resulting network is m-feasible and the number of 3090* CLB*s needed is minimum.*

To solve it, we modify the binate covering formulation used in Section 3.4.1. We generate all the k-feasible supernodes for each internal node of η as before. Recall that in the binate covering

matrix $\mathcal{B}$ built earlier, there is a column for each feasible supernode s_i and a row for each internal node of η (there are additional rows to take care of the binate covering constraints). There is a 1 in the position $\mathcal{B}(k, i)$ if the node corresponding to row k is contained in the supernode s_i. In the modified procedure, we start with $\mathcal{B}$, and append additional columns to it as follows. Consider a pair of 4-feasible supernodes $\mathcal{S}_{i,1}$ and $\mathcal{S}_{j,2}$ of nodes n_1 and n_2. If the supernodes are mergeable (i.e., the union of their supports has at most 5 inputs), we add a column to $\mathcal{B}$ corresponding to the supernode pair. This column corresponds to a match that covers all the nodes in the union of the two supernodes. We repeat this for all the supernode pairs for nodes n_1 and n_2 and then process all the node pairs likewise. Finally we solve a binate covering problem on the new matrix $\tilde{\mathcal{B}}$ to get an optimum solution.

Note that the new matrix $\tilde{\mathcal{B}}$ may have $\tilde{c} = c^2$ columns in the worst case, where c = number of columns of $\mathcal{B}$. For reasonably sized networks, this exact approach may be computationally expensive. We propose an approximation that speeds up BCM by separating collapsing and pairing steps. On the k-feasible network η, we first apply the BCM procedure of Section 3.4 and get a k-optimal network $\tilde{\eta}$. Then we search for maximum number of pairs of mergeable functions in $\tilde{\eta}$. Each such pair is placed on a CLB. Each unpaired function is assigned a separate CLB. The problem can be formulated as follows:

Problem 3.7.2 *Given a k-feasible network $\tilde{\eta}$, find the largest set of disjoint pairs of mergeable functions.*

Note that we do not lose any optimality by imposing that the pairs be disjoint. From an optimum solution that replicates functions in many pairs, another optimum solution can be constructed that only uses disjoint pairs.

We show that Problem 3.7.2 can be formulated as the problem of **maximum cardinality matching** in a certain graph $G(V, E)$,[11] which is built as follows. Corresponding to each internal node of $\tilde{\eta}$, there is a vertex $v \in V$. Edge $(v, w) \in E$ if and only if in $\tilde{\eta}$ the functions at the nodes corresponding to v and w are mergeable. Then the Problem 3.7.2 can be solved by finding a maximum cardinality matching in G. Each edge (v, w) in the matching means that the nodes corresponding to v and w should be placed on the same CLB. Each node at which no matched edge is incident is placed in a separate CLB. In general, G is a non-bipartite graph, i.e, it can have cycles with an odd number of edges. This makes the problem slightly harder.[12] Even though a

[11] The maximum cardinality matching problem in a graph is to find the largest set of edges no two of which share an end point.

[12] For a bipartite graph, a simple network flow-based technique can be used to solve the matching problem.

polynomial-time algorithm (in fact $O(|V|^{2.5})$) exists [57], it is difficult to implement and its running time can be long for large networks. We initially formulated this problem as a 0-1 linear integer program as follows. Define

$$\begin{aligned} x_j &= \text{selection variable for edge } j \text{ of } G, \\ A &= (a_{ij}) = \text{incidence matrix of G} \end{aligned}$$

where

$$x_j = \begin{cases} 1 & \text{if edge } j \text{ is in the matching,} \\ 0 & \text{otherwise} \end{cases}$$

and

$$a_{ij} = \begin{cases} 1 & \text{if in } G \text{ the edge } j \text{ is incident on vertex } v_i, \\ 0 & \text{otherwise} \end{cases}$$

The maximum cardinality matching problem now becomes:

$$\textbf{maximize} \sum_j x_j$$

subject to

$$\sum_j a_{ij} x_j \leq 1 \quad \forall i \tag{3.25}$$

(3.25) constrains every vertex v_i of G to be an end-point of at most one edge of the matching. We solve for x_js. The edges j for which x_j is 1 are in the matching. This is a pure 0-1 integer linear programming problem, which is NP-complete [30]. However, for the size of problems we are dealing with, its solution does not normally require excessive computer time. The basic advantage of using an integer programming formulation is that it is simple and, more important, it provides a wider framework for potential extensions of the merging problem. Among them is the merging of three or more functions into one CLB, which may be a feature in some future architecture. Providing these flexibilities helps to accommodate new architectures if a general tool for FPGAs is to be built.

Currently, we use a simple heuristic that greedily pairs two adjacent vertices of G that have minimum number of edges incident on them. Then it deletes these two vertices from G. These steps are repeated until no more pairs can be found. The idea is that vertices that have many neighbors in the beginning will very likely have some neighbors towards the end and can still be paired at the end. Such greedy matching algorithms do not guarantee to find an optimum solution, but it has been observed empirically that they are not far from one. Moreover, they are fast. This heuristic was first proposed in [39, 87].

3.7.1 Experimental Results

The aforementioned technique has been implemented in a command merge in mis-fpga. First, k-feasible networks are generated for two values of k: 4 and 5. Then, *merge* is applied to obtain maximum number of pairs of mergeable functions. If $k = 4$, it is possible that an unpaired function can be collapsed into its unpaired fanouts without causing any infeasibility. For example, consider two functions f and g: $f = abcd, g = fe$. Although f and g are not mergeable, f can be collapsed into g, yielding a 5-feasible function. *merge* exploits this possibility by invoking a collapsing step on the unpaired functions at the end. We found that some, though not significant, improvement is obtained as a result of this collapsing. In Table 3.8, the results of the experiments are shown. It turns out that the two values of k give comparable results.

In Table 3.9, we compare the performance of mis-fpga (using $k = 5$), Xmap, chortle-crf, and HYDRA. On the subset of examples where all the tools can finish, on average mis-fpga is 9.1% better than Xmap, 9.6% better than chortle-crf, and 11.7% better than HYDRA.

Relating factored form literals and CLBs

To see the relationship between the number of factored form literals and CLB-count, we plot in Figure 3.44 the number of CLBs obtained in the Table 3.8 in the column 5 against the number of literals in the optimized benchmarks, as reported in Table 3.1. Only those benchmarks are considered for which mis-fpga could finish. It can be seen that the relationship is nearly linear, although many benchmarks lie below the best linear fit. The average number of literals that can be placed in one CLB is 6.02.

3.8 Discussion

We proposed algorithms for minimizing the number of m-LUTs needed to realize a circuit. Two key steps - making a network feasible and block count minimization - were identified. One main difference with the conventional mappers is that decomposition is specific to the architecture. This indeed yields good results. Various decomposition techniques - functional, cube-packing, cofactoring, kernel extraction, and simple AND-OR decomposition were studied. Sometimes it is possible to predict the technique that gives best results. For instance, it was proved that cube-packing generates optimum tree implementations for functions having cubes with disjoint supports for $m \leq 5$. Similarly, functional decomposition works well for symmetric functions, since finding

example	4	5
5xp1	17	17
9sym	11	7
C1355	61	67
C1908	77	81
C2670	132	111
C3540	251	262
C432	52	51
C5315	331	304
C6288	287	288
C7552	299	295
alu2	84	83
alu4	-	-
apex2	68	64
apex3	-	-
apex7	38	44
b9	29	28
bw	33	27
clip	19	21
cordic	10	10
dalu	-	-
des	687	686
duke2	109	103
e64	45	54
ex4	104	113
f51m	12	10
k2	-	-
misex2	24	23
rd84	13	12
rot	122	137
sao2	26	28
spla	136	142
t481	5	5
vg2	18	21
z4ml	5	6
subtotal	3105	3100

Table 3.8: Xilinx 3090 CLB counts for mis-fpga

4	do mapping onto 4-LUTs, then apply *merge*
5	do mapping onto 5-LUTs, then apply *merge*
subtotal	sum of CLB counts over examples where completed
-	LUT script did not finish

example	mis-fpga	Xmap	chortle-crf	HYDRA
5xp1	17	19	21	20
9sym	7	31	32	41
C1355	67	54	83	63
C1908	81	78	84	66
C2670	111	162	122	212
C3540	262	251	243	285
C432	51	48	48	62
C5315	304	297	297	278
C6288	288	462	596	474
C7552	295	339	335	370
alu2	83	76	73	74
alu4	-	173	172	188
apex2	64	63	59	68
apex3	-	394	376	384
apex7	44	46	41	40
b9	28	29	27	24
bw	27	-	31	31
clip	21	20	21	23
cordic	10	10	10	13
dalu	-	199	195	199
des	686	692	635	661
duke2	103	98	95	92
e64	54	59	58	43
ex4	113	129	113	159
f51m	10	13	14	10
k2	-	287	301	288
misex2	23	23	21	23
rd84	12	30	30	32
rot	137	135	124	125
sao2	28	30	29	31
spla	142	-	125	127
t481	5	5	7	6
vg2	21	20	20	21
z4ml	6	7	7	5
total	-	-	4445	4538
all-finish	2931	3226	3245	3321

Table 3.9: Xilinx 3090 CLB counts: comparing various systems

mis-fpga apply mis-fpga with partial collapse
chortle-crf apply chortle-crf
Xmap apply Xmap
HYDRA apply HYDRA
total sum of CLB counts over all the examples
all-finish sum of CLB counts over examples where every tool completes

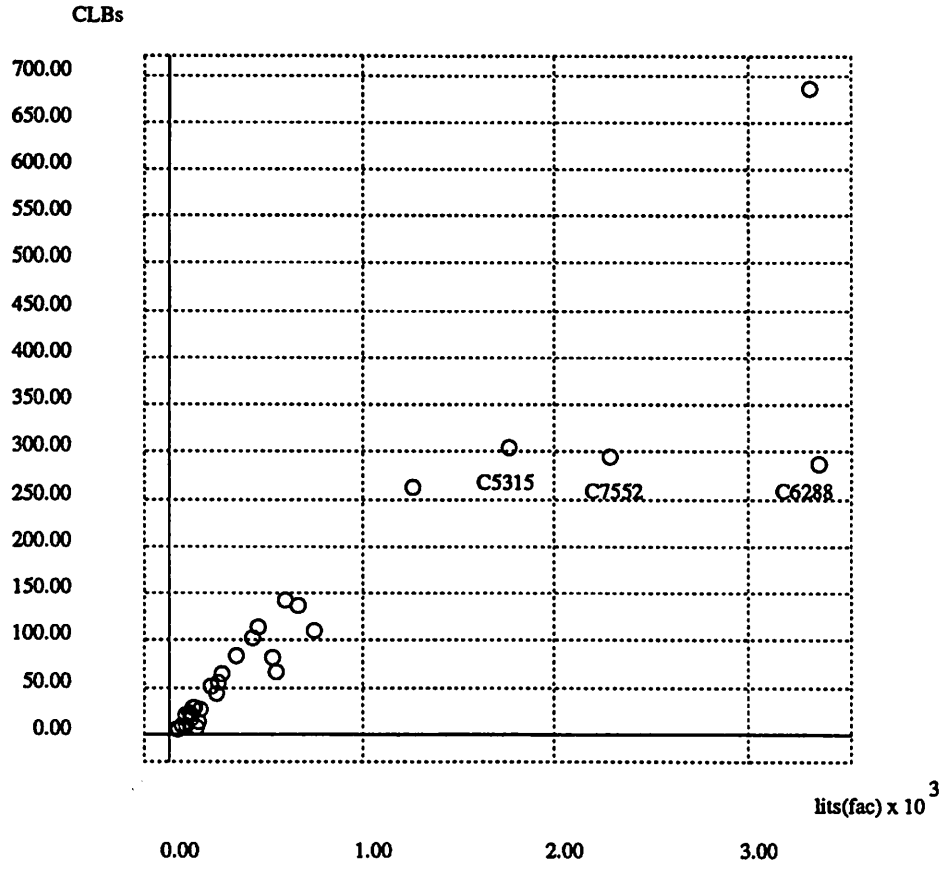


Figure 3.44: Relating factored form literals with CLB-count

a good input partition is easy for such functions. However, for an arbitrary function, it is not known *a priori* which method would work well. No single method works in all cases. Empirically, it was observed that applying cube-packing on both SOP and factored form and picking the better of the two gives reasonably good results. Another method based on support reduction was proposed to make the network feasible. While studying functional decomposition, we proposed an encoding-based solution to the problem of optimally obtaining simple $\vec{\alpha}$ and g functions. This has implications not only for synthesis for LUT architectures, but also for logic synthesis in general. In block count minimization, an exact method for solving the covering problem was presented. The key notion was that of a feasible supernode. The exact method is computationally expensive. So heuristic approaches were proposed. The idea of reducing the support of a node function such that the node can be eliminated from the network by collapsing it into its fanouts was also explored. The two steps of decomposition and BCM were coupled tightly in *partial collapse.*

Some issues have not been addressed explicitly in the techniques described thus far:

1. *Routing considerations*: Almost all the algorithms presented do not consider the routability of the final implementation. Since routing resources of the chip may be scarce, as in Xilinx 3090, it is crucial that routability be given due consideration during synthesis. One measure of routability is the number of edges in the Boolean network. So one way of addressing routability during synthesis is to minimize the number of newly created edges during decomposition and BCM. As described earlier, this is considered in the *partition* step of mis-fpga. Recently, some approaches have been proposed that do routing-driven synthesis. Bhat and Hill [8] have proposed a *placking* algorithm that integrates synthesis, placement, and packing (or mapping) in a tight loop. A similar approach proposed by Schlag, Kong, and Chan [76] considers covering and merging simultaneously, with an objective of minimizing the number of edges in the resulting network.

2. *Mapping a multi-output function*: mis-fpga maps one function at a time. Better results may be obtained if many functions are mapped simultaneously. For example, when making the network feasible, if, instead of decomposing a single function, multiple functions are decomposed collectively, good common divisors or sub-functions can be extracted, thus reducing the overall block count.

3. *Mapping with don't cares*: In the spirit of conventional technology mapping, the algorithms described here assume that the function being mapped is completely specified. Typically there

are don't cares associated with each node function of the network, which can be beneficially used in mapping. Mailhot and De Micheli showed one way of doing it in their seminal work [52]. This led to an improved approach by Savoj *et al.* [71]. These approaches can be applied to FPGA mapping as well.

4. *Determining a good input partition in functional decomposition.*

Chapter 4

Logic Optimization

4.1 Introduction

The goal of synthesis is to convert a design specification into an implementation. Since it is a complex task, it is divided into two phases. The first phase is technology-independent optimization (or logic optimization), in which a *minimal* representation of the design is sought. The minimality criterion may be either the number of product terms (if a PLA-like implementation is desired) or the number of literals in a factored form (if a standard-cell implementation is desired). The second phase is technology mapping, in which the minimal representation is mapped onto the target technology/architecture. The quality of the final implementation depends on the minimal representation generated by the optimization phase.

The last few years have seen a tremendous surge in the design and development of mapping algorithms targeted for LUT architectures [25, 26, 62, 63, 23, 39, 80, 87]. Almost all of these tools start from a representation of the circuit optimized for the number of literals in the factored form. Since the LUT architectures impose constraints on the synthesis process that are different from those imposed by PLAs or standard-cells, it is not clear if such cost functions are good measures of the complexity of a design for LUT architectures. Consider the following two examples.

Example 4.1.1 *Let m be 5. Consider two functions f_1 and f_2:*

$$\begin{aligned} f_1 &= abcdeg, \\ f_2 &= abc + b'de + a'e' + c'd'. \end{aligned}$$

The representation of f_1 has 6 literals, and that of f_2 10 literals. Both these representations are optimal, in that the misII *optimization script* (script.rugged) *does not further improve the literal*

counts. If the complexity (or cost) is measured using literals, f_2 is more complex than f_1. However, f_1 requires two 5-LUTs, whereas f_2 only one. So f_1 is costlier than f_2 for 5-LUT architecture.

Example 4.1.2 *Let m be 2. Consider the following representation of function f.*

$$f = ab' + a'c' + bc.$$

This representation remains unchanged after applying script.rugged. Moreover, this is the SOP with the minimum number of cubes and literals for this function. Now, if cube-packing were applied on this SOP, we get the following decomposition with five 2-LUTs.

$$\begin{aligned} t_1 &= ab' \\ t_2 &= a'c' \\ t_3 &= bc \\ t_4 &= t_1 + t_2 \\ f &= t_3 + t_4. \end{aligned}$$

No improvement is obtained after applying the block count minimization step. Now consider the following alternate representation of f:

$$f = ab' + a'b + bc + b'c'.$$

It uses one more cube and two more literals than the minimum representation. However, cube-packing yields

$$\begin{aligned} t_1 &= ab' + a'b \\ t_2 &= bc + b'c' \\ f &= t_1 + t_2, \end{aligned}$$

which uses two fewer LUTs.

Both of these examples underscore the need for targeting optimization phase for LUT architectures. Let us examine various steps in the optimization phase and see what can be done in each step. There are three main steps in optimization:

1. Kernel-extraction
2. Elimination
3. Simplification

4.2 Kernel-extraction

Certain kinds of sub-functions that are common to many nodes of the network are extracted and implemented once, thereby resulting in a smaller network representation. There may be many choices for kernels to be extracted. To evaluate the *best* kernel, the notion of **value of a kernel** was introduced [12]. It is the number of literals saved in the network if the kernel is used. To target LUT architectures, the value of a kernel is redefined as the number of LUTs saved by the kernel. How do we compute the new value? In the original definition, value-computation was easy - count the number of literals in the factored forms of the kernel and of the affected nodes before and after extracting the kernel, and compute the savings. In the new definition, the LUT-counts corresponding to the kernel and the affected node-functions have to be computed. This means that a mapping step has to be applied on these nodes. Since such computation is repeated for each kernel, the mapping step should be fast. We use cube-packing as a fast mapping technique.

The results of the experiments are not reported here. We modified *gkx*, the original kernel-extraction routine in misII. Later, a faster algorithm *fx* was implemented. The new optimization script used in the thesis, *script.rugged*, does not use *gkx*, but *fx*. We have yet to modify *fx* for LUTs.

4.3 Elimination

Elimination is a process that is the inverse of kernel-extraction. It collapses a node n into its fanouts if the value of the function at n is no more than a specified threshold. For LUT architectures, the value of a node can be redefined, exactly as in kernel-extraction.

4.4 Simplification

In simplification, for each node of the network, an appropriate don't care set is computed and a two-level minimizer is applied, resulting in a smaller representation.

Fujita and Matsunaga [28] modify simplification to target LUT architectures such that the support of each node of the network is minimized. Each node n is simplified as follows. First, candidate nodes that may be used as fanins of n are selected. From these, sets of minimal supports for n are computed using the algorithm of Halatsis and Gaitanis [34]. Finally, an irredundant cover for n is computed using a minimal support. The basic idea in choosing the minimal support as the cost function is that if this support is at most m, a minimum-LUT implementation of the node

function is obtained. However, if this support is greater, the cost function may not be appropriate. Two functions with the same support can have different LUT costs. Moreover, a function with fewer inputs can have a higher cost.

Example 4.4.1 *Let $m = 2$. Consider*

$$\begin{aligned} f_1(a,b,c) &= abc, \\ f_2(a,b,c) &= ac + bc + ab + a'b'c', \\ f_3(a,b,c,d) &= abcd. \end{aligned}$$

*f_1 and f_2 have the same support - $\{a,b,c\}$. The minimum-*LUT *implementation of f_1 has two* LUT*s, whereas that of f_2 has four, as will be proved in Lemma 5.3.11. Also, $|\sigma(f_3)| = 4$, and it can be implemented using three* LUT*s, whereas $|\sigma(f_2)| = 3$, and the cost of f_2 is four.*

To target simplification for LUT architectures, we should examine LUT mapping techniques that work on an SOP. One such technique, which has proved effective, is cube-packing, as described in Section 3.3.2. Cube-packing groups the cubes of an SOP into minimum number of LUTs. The number of LUTs obtained depends on the SOP under consideration. Since there are many possible SOPs for a function, the natural question to ask is: "*How should we obtain an* SOP *or a two-level representation of a function that yields better* LUT *implementation after cube-packing?*" Let us first describe the standard two-level minimization problem and one way to solve it.

4.4.1 Two-level Minimization

The two-level minimization problem, in its simplest form, is that of obtaining an SOP representation of a function f with minimum number of cubes (or literals). The classical solution consists of three steps:

1. Generate all the primes of f.
2. Form the covering (or prime implicant) table C.
3. Derive a minimum cover of C.

The covering table C has a row for each minterm in the on-set of f and a column for each prime. $C(i,j) = 1$ if and only if the prime j covers (contains) minterm i. Then we say that column j covers

abc	$a'c'$	$a'b$	bc	$b'c'$	ac	ab'
000	1	0	0	1	0	0
010	1	1	0	0	0	0
011	0	1	1	0	0	0
111	0	0	1	0	1	0
101	0	0	0	0	1	1
100	0	0	0	1	0	1

Table 4.1: The covering table C

row i, or equivalently, row i is covered by column j. The problem of selecting a minimum-cube cover, i.e., a minimum subset of primes that cover all the minterms in the on-set of f, is thus mapped into the problem of selecting a minimum column cover of C.[1] This is the well-known unate covering problem. The formulation can be modified easily to generate an SOP with minimum number of literals. With each prime, a weight equal to the number of literals is associated. The problem then is one of choosing a minimum-weighted cover of C.

Example 4.4.2 *Consider*

$$f = ab' + a'b + bc + b'c'$$

f has 6 primes: $a'c', a'b, bc, b'c', ac, ab'$. The covering table for f is shown in Table 4.1. The minimum-cube cover is obtained by picking the primes $a'c', bc$, and ab'.

For large PLAs, it is not feasible to build the covering table. Instead, an iterative improvement strategy is used. ESPRESSO [11] is such a heuristic minimizer that produces excellent quality solutions.

We first examine what happens if a standard two-level minimizer is used before cube-packing. In other words, we study if minimum-cube or minimum-literal SOPs are good for cube-packing.

4.4.2 Are Minimum-cube and Minimum-literal SOPs Good for Cube-packing?

Given a function f, the SOP representation used dictates heavily the LUT count obtained after cube-packing. A two-level minimizer typically uses a cost function, which is the number of

[1] A column cover of C is a set of columns of C such that each row of C is covered by some column in this set. A minimum column cover is a column cover with minimum number of columns.

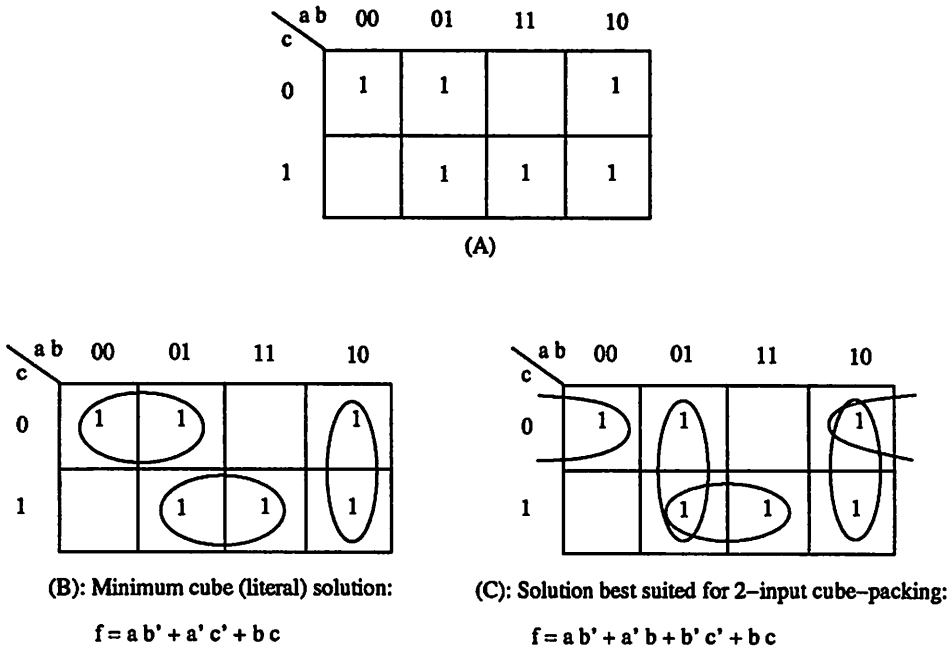


Figure 4.1: Two-level minimization and LUT decomposition

cubes (sometimes it may be the number of literals). A multi-level optimizer minimizes the number of literals in factored form. Since cube-packing operates on an SOP representation, we concentrate on two-level minimization, and therefore on the corresponding cost functions - the number of cubes or literals. With the help of a simple example, we show that these may not be appropriate cost functions.

Example 4.4.3 *Let $m = 2$. Consider the function*

$$f(a,b,c) = a'c' + a'b + bc + b'c' + ac + ab'. \tag{4.1}$$

The Karnaugh map for f is shown in Figure 4.1 (A). Applying cube-packing on this SOP *yields*

$$\begin{aligned} t_1 &= a'c' + ac \\ t_2 &= a'b + ab' \\ t_3 &= bc + b'c' \\ t_4 &= t_1 + t_2 \\ f &= t_3 + t_4. \end{aligned}$$

This decomposition uses five 2-LUT*s.*

A minimum-cube cover of f is shown in Figure 4.1 (B) and is

$$f = ab' + a'c' + bc. \tag{4.2}$$

Note that this is also a minimum literal solution, since all the six primes of f (all are present in (4.1)) have two literals each. Applying cube-packing on this cover, we get the following:

$$\begin{aligned} t_1 &= ab' \\ t_2 &= a'c' \\ t_3 &= bc \\ t_4 &= t_1 + t_2 \\ f &= t_3 + t_4. \end{aligned}$$

It takes five 2-LUTs to realize f - no improvement over the starting configuration. Let us consider another representation of f, shown in Figure 4.1 (C):

$$f = ab' + a'b + bc + b'c'. \tag{4.3}$$

This representation uses one more cube and two more literals than the minimum solution of (4.2). After applying cube-packing, we get:

$$\begin{aligned} t_1 &= ab' + a'b \\ t_2 &= bc + b'c' \\ f &= t_1 + t_2. \end{aligned}$$

This decomposition uses three 2-LUTs, two less than the previous covers. This improvement comes from the relationship between the supports of primes in (4.3). Primes ab' and $a'b$ have the same support; so they can be placed in one 2-LUT. Similarly, bc and $b'c'$ are placed in one 2-LUT. One extra LUT is needed to realize the OR of the resulting sub-functions. In contrast, the minimum-cube cover of (4.2) is such that no two primes can fit in one 2-LUT, implying that three LUTs are needed to realize the three primes, and then two extra LUTs to realize the ORs.

This simple example underscores the need for a different formulation of the two-level minimization problem for the LUT architectures.

abc	$a'c'$	$a'b$	bc	$b'c'$	ac	ab'
000	1	0	0	1	0	0
010	1	1	0	0	0	0
011	0	1	1	0	0	0
111	0	0	1	0	1	0
101	0	0	0	0	1	1
100	0	0	0	1	0	1

Table 4.2: The covering table C

4.4.3 Targeting Two-level Minimization for Cube-packing

Let us examine the covering table C for the function f of (4.1). It is shown in Table 4.2.[2] The minterms of the on-set of f are the rows of the table and the primes are the columns. The problem is *to select a subset of primes such that the resulting* SOP *yields good results after cube-packing.*

If we decide to put prime $a'c'$ in the final cover, we can put ac also in the cover without increasing the number of LUTs generated by cube-packing. The reason is as follows. Assume that cube-packing places $a'c'$ in an LUT T. It can then place ac in T without using any leftover capacity of T. This is because the support of the cube ac is contained in that of $a'c'$ (in fact, the supports are the same). The minterms 111 and 101 contained in ac are thereby covered *free of cost.* It is the case then that the inclusion of a prime p_1 affects the cost of inclusion of another prime p_2 whose support is contained in that of p_1. In general, the total cost of selecting a set of primes can be less than the sum of the costs of the primes selected because of support-sharing. For example, if f is to be implemented using 3-LUTs, the combined cost of all the primes is one LUT, whereas the sum of the costs of the primes in the minimum-cube cover is three. Note that the standard formulation of the covering problem works only if the columns have independent, fixed costs (weights). It does not allow a *dynamic* relationship among the costs of the columns. We next show that it is possible to solve the problem at hand by transforming it into another covering problem in which the costs of the columns are fixed and independent. In the modified formulation, *not all columns are primes; some correspond to sets of primes.* These sets are generated by taking all possible unions of the prime-supports and including a union if it has at most m variables. If a prime p is such that $|\sigma(p)| > m$, p will be excluded from the union process. Of course, it will be included in the

[2] It is the same as Table 4.1.

modified covering table as a singleton set.

The modified formulation consists of the following three steps:

1. Generate all the primes of f. Then, generate a set $\mathcal{U}$ of all possible unions of prime-supports where each union has at most m variables.
2. Using $\mathcal{U}$, build the modified covering table $\tilde{C}$ from the original table C.
3. Solve $\tilde{C}$.

Generation of unions of prime-supports

Let $\mathcal{P}$ denote the set of primes of the function f. Our goal is to generate $\mathcal{U}$, where

$$\mathcal{U} = \{s : s = \cup_{p \in Q} \sigma(p),\ Q \subseteq \mathcal{P}, \text{ such that } |s| \leq m\} \tag{4.4}$$

Depending on the supports of the primes, we divide $\mathcal{P}$ into two sets: $\mathcal{P}_{\leq m}$ and $\mathcal{P}_{>m}$.

$$\begin{aligned} \mathcal{P}_{\leq m} &= \{p \in \mathcal{P} : |\sigma(p)| \leq m\} \\ \mathcal{P}_{>m} &= \mathcal{P} - \mathcal{P}_{\leq m} \end{aligned}$$

To generate $\mathcal{U}$, it suffices to consider only $Q \subseteq \mathcal{P}_{\leq m}$. We systematically take union of supports of primes in $\mathcal{P}_{\leq m}$ as follows. First, we generate the set of supports $S_0 = \{s : s = \sigma(p)$ for some $p \in \mathcal{P}_{\leq m}\}$. With each support s, we maintain a tag of primes $P(s) = \{p \in \mathcal{P}_{\leq m} : \sigma(p) \subseteq s\}$, i.e., $P(s)$ is the set of those primes whose support is contained in s.[3] Let S_1 be a duplicate copy of S_0, and $S_j = \phi$ for $j > 1$. The algorithm proceeds in iterations. In iteration j, for each $s \in S_0$, and $t \in S_j$, it computes $u = s \cup t$. If $|u| > m$, it is discarded. If u has already been generated, $P(u) = P(u) \cup P(s) \cup P(t)$. Otherwise, it is added to S_{j+1} with $P(u) = P(s) \cup P(t)$. This process is repeated until a fixed point is reached. The final set of supports is $\mathcal{U} = S_0 \cup S_2 \cup S_3 \ldots \cup S_{k+1}$, where k is the final iteration. Since S_1 is same as S_0, it is not included.

This algorithm is quite fast. For all the benchmarks we tried, four iterations were sufficient to generate the entire $\mathcal{U}$ for $m = 5$. We illustrate the working of the algorithm with the following example. Let $\mathcal{P}_{\leq m}$ be $\{p_0, p_1, p_2, p_3, p_4\}$. Let the prime-supports be as follows:

[3] Initially, for each $s \in S_0$, $P(s) = \{p \in \mathcal{P}_{\leq m} : \sigma(p) = s\}$. As the algorithm proceeds, primes whose supports are subsets of s get added in $P(s)$.

support	prime
$\{a,b\}$	p_0
$\{a,b\}$	p_1
$\{a,c\}$	p_2
$\{a,b,c\}$	p_3
$\{d,e,g\}$	p_4

Let us also assume that $m = 5$. We generate S_0 and tag each support with the corresponding prime(s).

support	prime tag
$\{a,b\}$	p_0, p_1
$\{a,c\}$	p_2
$\{a,b,c\}$	p_3
$\{d,e,g\}$	p_4

Next, S_1, a copy of S_0, is generated.

set	support	prime tag
S_0	$\{a,b\}$	p_0, p_1
	$\{a,c\}$	p_2
	$\{a,b,c\}$	p_3
	$\{d,e,g\}$	p_4
S_1	$\{a,b\}$	p_0, p_1
	$\{a,c\}$	p_2
	$\{a,b,c\}$	p_3
	$\{d,e,g\}$	p_4

We pick $t = \{a,b\} \in S_1$ and vary s over S_0. First, $s = \{a,b\}$. Since $s = t$, no new support is generated and the prime tags also remain unchanged. Next, $s = \{a,c\}$. Then $u = \{a,b,c\}$, which is already in S_0. We set $P(u) = P(u) \cup P(s) \cup P(t)$, i.e., $P(\{a,b,c\}) = \{p_3, p_0, p_1, p_2\}$. Next, $s = \{a,b,c\}$, for which $u = \{a,b,c\}$, already present in S_0. $P(u)$ remains same. Finally, $s = \{d,e,g\}$, and we generate $u = \{a,b,d,e,g\}$. This is a new support and is added in S_2 with the prime tag $P(u) = \{p_0, p_1, p_4\}$. The current supports and the corresponding tags are as follows:

set	support	prime tag
S_0	$\{a,b\}$	p_0, p_1
	$\{a,c\}$	p_2
	$\{a,b,c\}$	p_3, p_0, p_1, p_2
	$\{d,e,g\}$	p_4
S_1	$\{a,b\}$	p_0, p_1
	$\{a,c\}$	p_2
	$\{a,b,c\}$	p_3, p_0, p_1, p_2
	$\{d,e,g\}$	p_4
S_2	$\{a,b,d,e,g\}$	p_0, p_1, p_4

Next, $t = \{a, c\} \in S_1$. Repeating the above process, we generate a new support $\{a, c, d, e, g\}$ with tag = $\{p_2, p_4\}$.

set	support	prime tags
S_0	$\{a, b\}$	p_0, p_1
	$\{a, c\}$	p_2
	$\{a, b, c\}$	p_3, p_0, p_1, p_2
	$\{d, e, g\}$	p_4
S_1	$\{a, b\}$	p_0, p_1
	$\{a, c\}$	p_2
	$\{a, b, c\}$	p_3, p_0, p_1, p_2
	$\{d, e, g\}$	p_4
S_2	$\{a, b, d, e, g\}$	p_0, p_1, p_4
	$\{a, c, d, e, g\}$	p_2, p_4

The next two elements from S_1 do not generate any new supports of cardinality at most 5. Although a new support $\{a, b, c, d, e, g\}$ is generated, it is discarded because it has 6 variables. The last table then marks the end of the first iteration.

In the second iteration, we pick elements from S_2 and take their unions with the elements of S_0. It can be easily verified that no new supports of cardinality at most 5 are generated. Hence, a fixed point has been reached and the procedure terminates. $\mathcal{U} = S_0 \cup S_2 = \{\{a, b\}, \{a, c\}, \{a, b, c\}, \{d, e, g\}, \{a, b, d, e, g\}, \{a, c, d, e, g\}\}$.

Constructing the modified covering table

Let C be the original covering table for f. The entries of the modified covering table $\tilde{C}$ are determined as follows. The column set of $\tilde{C}$ corresponds to $\mathcal{P}_{>m} \cup \mathcal{U}$, whereas its rows are the minterms of the on-set of f. The column in $\tilde{C}$ corresponding to a prime in $\mathcal{P}_{>m}$ is identical to the corresponding column in C. Next, consider a column, say j, of $\tilde{C}$ corresponding to support $s \in \mathcal{U}$. Then, $\tilde{C}(i, j) = 1$ if and only if some prime $p \in P(s)$ covers the minterm of f corresponding to row i. In other words, column j is the *entry-wise* union of the columns in C corresponding to primes in $P(s)$. Note that each such column j corresponds to a set of primes that can be put on one m-LUT.

Each column of $\tilde{C}$ is assigned a weight equal to the cardinality of the corresponding support. This is because a support s_1 with 2 elements is less expensive than a support s_2 with 5 elements: s_1 can accommodate any other support with at most $(m - 2)$ elements, whereas s_2 can accommodate only a support with at most $(m - 5)$ elements.

abc	$\{a'c', ac\}$	$\{a'b, ab'\}$	$\{bc, b'c'\}$
000	1	0	1
010	1	1	0
011	0	1	1
111	1	0	1
101	1	1	0
100	0	1	1

Table 4.3: The modified covering table $\widetilde{C}$

Solving the covering table

Given a covering table, selecting a subset of columns of minimum-weight that covers all the rows is an NP-hard problem. However, efficient heuristics exist that work well even on reasonably large tables [31, 68]. We use one such heuristic [68, 11].

We now show how the algorithm works on the example of Table 4.2. Recall that $m = 2$ for this example. Applying first two steps, we find that the modified covering table $\widetilde{C}$ has three columns, with prime tags $\{a'c', ac\}$, $\{a'b, ab'\}$, and $\{bc, b'c'\}$. Note that $\mathcal{P}_{>m}$ is empty. Each column of $\widetilde{C}$ is formed by merging the corresponding columns of C. For example, the column $\{a'c', ac\}$ has a 1 in a row whenever either $a'c'$ or ac has a 1 in that row in the table C. The resulting covering table is shown in Table 4.3. Each column is assigned a weight of 2. Any two columns cover all the rows of $\widetilde{C}$. In particular, we can generate the SOP of (4.3) by picking the last two columns.

Interestingly, the algorithm described above has the capability of increasing the number of cubes too. For instance, in the previous example, if the starting representation is assumed to be the minimum-cube cover $f = ab' + a'c' + bc$, the algorithm still comes up with the representation $f = ab' + a'b + bc + b'c'$, which has 4 cubes.

4.4.4 The Overall Algorithm

Given a cover of a function f, the overall algorithm works as follows:

1. If $|\sigma(f)| \leq m$, no minimization is done. This prevents a feasible function from becoming infeasible after simplification, which could happen because of substitution of a function outside the support of f into f through the use of don't cares.

2. The local don't care set d is generated for f using the algorithm by Savoj *et al.* [72]. This is useful in a multi-level framework.

3. All primes of the incompletely specified function (f, d) are generated. The prime-generation algorithm used is the one implemented in ESPRESSO-EXACT [11]; it uses the method of iterative consensus. While generating them, if it is detected that they are too many (i.e., more than a user-specified parameter NUM_PRIME), the generation process stops. Only NUM_PRIME primes are generated. Then, cubes from the original cover are added to ensure that the resulting SOP is a cover of (f, d).

4. The covering table C is built.[4] $\widetilde{C}$ is generated, as described in the step 2 of the algorithm. If $\widetilde{C}$ is not too big, it is solved as described in step 3, thus producing a subset of primes that covers all the rows of $\widetilde{C}$, and hence of C. Otherwise, $\widetilde{C}$ is so big that solving it by the heuristic mentioned for step 3 in the last subsection does not give good results. So, we enter the REDUCE-EXPAND-IRREDUNDANT ESPRESSO loop, appropriately modified. The modification is in the procedure IRREDUNDANT [11], which first generates a covering table C^* and then solves it to produce a minimal cover. Typically, C^* is much smaller than C or $\widetilde{C}$. Our modification is an obvious one: instead of solving C^*, we first generate $\widetilde{C^*}$, and then solve it.

5. The cover obtained from the last step is checked against the original cover. Whichever gives better results after cube-packing is accepted. This step corresponds to a similar one in misII [12], where a simplified function returned from the two-level minimizer is accepted only if it improves the cost (typically number of literals in the factored form).

4.4.5 Experimental Results

The main claim of this work is that it is possible to obtain a representation of a function using two-level minimization that is better suited for cube-packing. The benchmarks should be such that we can perform effective two-level minimization. The MCNC two-level benchmarks (PLAs) provide such a scenario. Since PLAs generally have multiple outputs and cube-packing operates on single output functions, we treat each function independently. After appropriate two-level minimization (either standard or the one targeting LUTs), we run cube-packing on the minimized

[4] In ESPRESSO, the rows of C do not correspond to the minterms of the on-set of f; they, in fact, represent sets of minterms [11], thus yielding a smaller table size.

example	ESP_CUBES	ESP_SOP-LITS	ESP_LUT	ESP_CUBES / ESP_LUT	ESP_SOP-LITS / ESP_LUT
5xp1	24	24	**11**	2.181	2.181
9sym	114	114	**111**	1.027	1.027
Z5xp1	25	25	**14**	1.785	1.785
Z9sym	115	115	**72**	1.597	1.597
alu4	236	236	**226**	1.044	1.044
apex1	1266	1265	**1255**	1.008	1.007
apex2	**1118**	1159	**1118**	1.000	1.037
apex3	889	892	**874**	1.017	1.020
apex4	802	802	**736**	1.089	1.089
apex5	**1116**	**1116**	1117	0.999	0.999
b12	27	27	**25**	1.080	1.080
bw	**28**	**28**	**28**	1.000	1.000
clip	64	57	**35**	1.828	1.628
con1	**3**	**3**	**3**	1.000	1.000
cordic	1127	1127	**1125**	1.001	1.001
cps	1092	1088	**1064**	1.026	1.022
duke2	319	314	**305**	1.045	1.029
e64	**545**	**545**	**545**	1.000	1.000
ex1010	607	607	**574**	1.057	1.057
ex4	**269**	**269**	274	0.981	0.981
ex5	148	148	**143**	1.034	1.034
inc	**19**	**19**	**19**	1.000	1.000
misex1	**9**	**9**	10	0.900	0.900
misex2	41	**39**	**39**	1.051	1.000
misex3	1083	1083	**1040**	1.041	1.041
misex3c	236	236	**227**	1.039	1.039
pdc	167	**166**	**166**	1.006	1.000
rd53	4	11	**3**	1.333	3.666
rd73	61	62	**56**	1.089	1.107
rd84	195	195	**132**	1.477	1.477
sao2	63	54	**42**	1.500	1.285
seq	2100	2111	**1937**	1.084	1.089
spla	**592**	**592**	593	0.998	0.998
squar5	13	13	**8**	1.625	1.625
t481	**295**	**295**	**295**	1.000	1.000
table3	902	902	**874**	1.032	1.032
table5	**832**	**832**	839	0.991	0.991
vg2	**22**	**22**	**22**	1.000	1.000
xor5	**1**	**1**	**1**	1.000	1.000
total	16569	16603	15958	-	-
mean	-	-	-	1.126	1.147

Table 4.4: Number of 5-input LUTs after two-level minimization and cube-packing

ESP_CUBES ESPRESSO loop for minimum cubes, followed by cube-packing
ESP_SOP-LITS ESPRESSO loop for minimum SOP literals, followed by cube-packing
ESP_LUT modified ESPRESSO for LUT architectures, followed by cube-packing

function and evaluate the LUT count. The cost of a benchmark is the sum of the LUT counts of the individually minimized functions of the benchmark. For this set of experiments, $m = 5$. We do two kinds of minimization:

- ESP: Apply the two-level minimizer ESPRESSO [11], i.e., invoke the REDUCE-EXPAND-IRREDUNDANT loop on the function until no further improvement takes place. This is done for two cost functions:
  1. number of cubes (column ESP_CUBES in Table 4.4),
  2. number of literals in the SOP (column ESP_SOP-LITS in Table 4.4).
- ESP_LUT: Run the new minimizer described in Section 4.4.4. The following parameters are set:
  1. NUM_PRIME = 20,000.
  2. Maximum size of $\tilde{C}$ = 100 x 2500, i.e., if $\tilde{C}$ has more than 100 rows and 2500 columns, the LUT-modified REDUCE-EXPAND-IRREDUNDANT loop of ESPRESSO is entered.

The results of these experiments are shown in Table 4.4. The bold entries indicate the minimum LUT count for the benchmark. On most of the benchmarks, ESP_LUT outperforms ESPRESSO in the LUT count. On a per example basis and using geometric means, ESP_CUBES is 12.6% worse than our approach, and ESP_SOP-LITS, 14.7%. On *5xp1*, *Z5xp1*, *Z9sym*, *apex4*, *clip*, *rd84*, *sao2*, *seq*, and *squar5*, the difference is significant, and often as much as 50%. Most of these examples have a lot of primes with at most 5 literals, thereby providing ESP_LUT with plenty of opportunity to generate unions of supports. Though not shown, the numbers of cubes in the covers generated by ESP_LUT are almost always higher than those produced by ESP_CUBES or ESP_SOP-LITS. This is as expected, since ESP_CUBES generates optimum or near-optimum solutions for number of cubes. This validates our hypothesis that neither of the two cost functions (number of literals and number of cubes) is an accurate measure of the design complexity for LUT architectures.

On some benchmarks, we hardly find any improvement. Moreover, on some, ESP_LUT gives worse results. One reason is that sometimes there are not too many unions of prime-supports with at most 5 elements, and the optimization problem to be solved is not different from the standard one. Moreover, in such cases, whenever the covering table is big, the heuristic solution obtained after solving the covering table does not compare well with the solution generated by the iterative improvement loop of ESPRESSO.

With respect to the run times, ESP_LUT is slower than ESP_CUBES and ESP_SOP-LITS, especially on the benchmarks with large number of primes. ESP_LUT keeps on generating primes until it hits the NUM_PRIME limit, at which point it exits prime-generation and very likely switches over to the modified minimization loop. This is because in the experiments, NUM_PRIME = 20,000, whereas the maximum column limit in $\widetilde{C}$ is 2500. On such examples, ESP_LUT is 10-15 times slower.

4.5 Discussion

In our quest of the right cost function in logic optimization for LUTs, we were led to look at the mapping step. We modified kernel-extraction and simplification. In kernel-extraction, the value of a kernel was redefined, whereas in simplification, we targeted a representation suited for cube-packing. In particular, we showed that number of cubes or literals in a sum-of-products representation is not the best cost function for cube-packing. In the new formulation, the main idea is to use the support of a set of primes as the basic object, as opposed to a prime. The proposed technique generates around 13% better solutions after cube-packing as compared to a standard two-level minimizer.

For a benchmark with a large number of primes, it becomes impractical to generate all of them in ESP_LUT. So we switch to a modified ESPRESSO loop, but typically that too does not give significant improvement as compared to the standard ESPRESSO loop. Recently, Mcgeer *et al.* presented a new formulation of two-level minimization [55] that generates a covering table typically smaller than the one generated by conventional two-level minimizers (so it can handle benchmarks with a large number of primes), and guarantees that no optimality is lost in the process. We will like to incorporate our work in this framework.

Let us see how our approach can be applied in a multi-level environment. Typically, two-level minimization with an appropriate don't care set is applied on the SOP at each node of the multi-level network. It is natural to replace all the invocations of two-level minimization with our formulation in a multi-level optimization script. However, we discovered that the modified script, followed by a LUT mapping script, does not improve the quality of the final results for most of the benchmarks. This is somewhat surprising. We attribute it to the following two factors:

1. We have not modeled cube-packing in other optimization routines such as kernel-extraction, elimination, and resubstitution. By targeting minimum-cube and minimum-literal counts,

these routines undo the work of ESP_LUT.

2. We have not modeled decomposition techniques such as Roth-Karp decomposition and co-factoring, which are also used in LUT mapping.

All this underscores the need for a tighter coupling of the optimization and mapping steps for LUT architectures. This is conceptually simpler for LUT architectures as compared to standard-cells, because an LUT is easier to characterize than a standard-cell library. The work described here is a step in this direction, which, we believe, will provide fertile ground for future research.

Chapter 5

Complexity Issues

5.1 Introduction

As discussed earlier, many synthesis tools for LUT architectures have been proposed - XNFMAP [88], chortle [25], chortle-crf [26], mis-fpga [62, 63], HYDRA [23], Xmap [39], VISMAP [87], ASYL [80], flow-map [16] to name a few. We can compare one tool with another and get an idea of the relative quality of these tools, but there is no way of judging the absolute quality of the solutions generated by any of these tools. This is an important concern, as it is directly related to the research effort that should go into improving these tools and the solution quality. One way to answer this question is to compute the minimum number of LUTs needed for the realization of a Boolean function. However, as shown in Corollary 2.3.2, this is a difficult problem - in fact, NP-hard. It is desirable then to at least derive tight lower and upper bounds; with tight bounds one can evaluate with some confidence how far various synthesis tools are from optimality. Also, if good upper bounds can be obtained, one can use them to predict quickly the LUT-count of a circuit without technology mapping.

This chapter presents some results on the complexity of a function measured in terms of the number of m-LUTs required to implement it. Previous work is summarized in Section 5.2. The new results are presented in Section 5.3. The first result is on computing upper bounds on the complexity of an n-input function. Then the exact complexity of a particular class of functions, namely those with $(m + 1)$ inputs, is derived. The next set of results is on proving upper bounds, given a representation of the function. Two representations are considered: sum-of-products and factored form. To check how good they are, these bounds are compared with the synthesized circuits in Section 5.4. Finally, open problems are discussed in Section 5.5.

5.1.1 Definitions

Throughout this chapter, we will use the following generalized notion of factored forms.

Definition 5.1.1 *A* **factored form** *of a logic function is the generalization of a sum-of-products form allowing nested parentheses and arbitrary binary operations.*

For example, $ab'c' + a'bc' + d$ is an SOP with 7 literals, and it can be written in factored form as $((a \oplus b)c') + d$ with 4 literals. Note that the definition presented in Chapter 2 (Definition 2.1.14) allows only AND and OR binary operations.

Definition 5.1.2 *A* **leaf-DAG** *[68] is a rooted directed acyclic graph in which the only multiple fanout nodes are possibly the inputs. The non-input nodes of the leaf-*DAG *are called* **internal**. *If there is an edge from a node i to a node j, i is a* **child** *of j, and j the* **parent** *of i. In a leaf-*DAG, *an internal node all of whose children are inputs is called a* **leaf-node**, *or simply a* **leaf**. *Every other internal node is a* **non-leaf**.

A leaf-DAG is a rooted tree (single fanouts at internal nodes) possibly with multiple fanout points at some inputs. Figure 5.21 shows a leaf-DAG. Note that the notion of a leaf of a leaf-DAG is slightly different from that of a leaf of a BDD, as defined just after Definition 2.1.17. This is because the context in which the term will be used here is different.

Definition 5.1.3 *A Boolean function $f(x_1, x_2, \ldots, x_n)$ is* **trivial** *if it is either identically 0, identically 1, x_i or x_i' for some i, $1 \leq i \leq n$. Otherwise f is* **non-trivial**.

Inversion is considered a trivial function because it is essentially free for LUT architectures.

Recall from Definition 2.1.21 that a function $f(x, \ldots)$ essentially depends on x if $f_x \neq f_{x'}$. For example, $f(x_1, x_2, x_3) = x_1x_2 + x_1'x_2 + x_3$ essentially depends on x_2 and x_3, but not on x_1, since f can be rewritten as $x_2 + x_3$.

Definition 5.1.4 *The minimum number of m-*LUT*s needed to realize a function f is called its* **complexity**, *and is denoted as $C_m(f)$. For a set S of functions,*

$$C_m(S) = \max_{f \in S} C_m(f). \tag{5.1}$$

For example, if $f = x_1x_2 + x_1x_3 = x_1(x_2 + x_3)$, $C_2(f) = 2$. One 2-LUT realizes $z(x_2, x_3) = x_2 + x_3$, and the other $f = x_1z$. Moreover, two LUTs are necessary, since f essentially depends

on all the three input variables. Note that the complexity of a function does not depend on the representation used.

An alternate way of measuring the complexity of a set of functions is to define it as the sum of the complexities of the functions in the set. We will find this notion useful in Corollary 5.3.29, where we are interested in finding upper bounds on the number of LUTs needed to realize an entire network. The network corresponds to a set of local node functions. However, in the majority of the chapter, the notion given in (5.1) will be more appropriate. Also, it is the one we have seen used in the literature [70].

5.2 Previous Work

Two types of results relate to this chapter - LUT-count prediction, which has been entirely empirical, and theory for lower and upper bounds on the complexity.

5.2.1 Prediction of LUT-count

The only prediction work known to us is by Schlag *et al.* [75], who optimize benchmark circuits using misII [12] and map the optimized circuits on to LUT-based architectures. By counting the total number of factored form literals and the number of Xilinx 3090 CLBs needed for the entire benchmark set (recall that each CLB can either implement any function of up to 5 inputs, or two functions with at most 4 inputs each, with no more than 5 inputs in all [88]), they conclude that, on average, roughly 5 literals are packed in a CLB.

Using mis-fpga, results with superior quality are produced. From Section 3.6.5, it is seen that, on average, mis-fpga puts 4.8 literals in a 5-LUT, and 6 literals in a CLB (Section 3.7.1).

Though reasonable, both these approaches are empirical. We would like to provide a theoretical basis for predicting the LUT-count.

5.2.2 Bound Theory

Let $S(n)$ be the set of functions whose true support has at most n variables, i.e., $S(n) = \{f : |\sigma_T(f)| \leq n\}$. Then $S(n) - S(n-1)$ represents the set of n-input functions that essentially depend on each of the n inputs. We first present lower bounds and then upper bounds on $C_m(f)$ for $f \in S(n)$.

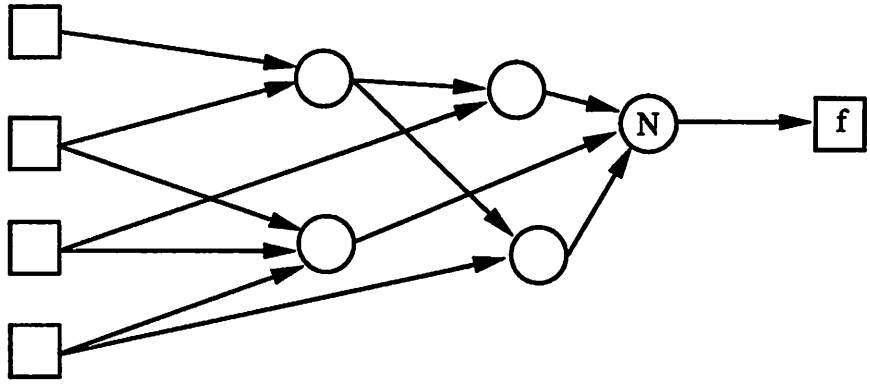


Figure 5.1: A single output, n-input network

Lower Bounds

Proposition 5.2.1 (Savage [70]) *Let* $f \in S(n) - S(n-1)$, $n > 1$. *Then* $C_m(f) \geq \lceil \frac{n-1}{m-1} \rceil$. *Moreover, this bound is tight, i.e., there exists a function* $f \in S(n) - S(n-1)$ *that can be realized in* $\lceil \frac{n-1}{m-1} \rceil$ LUT*s*.

Proof Consider η, an optimum m-feasible realization of f. This is shown in Figure 5.1. It has $k = C_m(f)$ internal nodes, n primary inputs, and 1 primary output. There is one distinguished internal node N that realizes f. The total number of edges in η is counted in two different ways - one, by counting the total number of fanins (i.e., the sum of in-degrees) and two, by counting the total number of fanouts (i.e., the sum of out-degrees). The sum of in-degrees of all the nodes is at most $km + 1$, since there are at most m inputs to any internal node, and, in addition, there is an edge fanning in to the output node from N. To count the out-degrees, observe that each internal node fans out to some other node in η, otherwise it can be deleted, resulting in a smaller realization. This would contradict the fact that η is an optimum realization of f. This sums to at least k. Then, since f essentially depends on each input, there must be at least one outgoing edge from each primary input node. The sum of the out-degrees is then at least $k + n$.

$$\begin{aligned} km + 1 &\geq \text{total in-degree} = \text{total out-degree} \geq k + n \\ \Rightarrow k &\geq \frac{n-1}{m-1} \\ C_m(f) &\geq \frac{n-1}{m-1} \end{aligned}$$

Since $C_m(f)$ is integral, we can put *ceil* on the right side and obtain the desired result.

The bound is tight, and is met by $f(x_1, x_2, \ldots, x_n) = \text{AND}(x_1, x_2, \ldots, x_n)$. ■

Definition 5.2.1 *([86]) The notion "almost all functions f of a class $F(n) \subseteq S(n)$ have property P" stands for the assertion that*

$$\frac{|\{f \in F(n) \mid f \text{ has } P\}|}{|F(n)|} \to 1 \text{ as } n \to \infty.$$

Using simple counting arguments, Shannon [79] proved that optimum circuits (in terms of any two-input gates) for almost all functions have exponential size. The reason is that the number of circuits with small size grows much slower than the number of different Boolean functions. We make this precise in the next proposition.

Proposition 5.2.2 *([86]) Almost all n-variable functions f satisfy $C_2(f) \geq \frac{2^n}{n}$.*

Proof See [86]. ■

Upper Bounds

Proposition 5.2.3

$$C_m(S(n)) \leq \begin{cases} 2^{n-m+1} - 1 & \text{if } m > 2, \\ 2^n - 3 & \text{if } m = 2. \end{cases} \tag{5.2}$$

Proof $f(x_1, x_2, \ldots, x_n) \in S(n)$ is decomposed as

$$f = x_1 f_{x_1} + x_1' f_{x_1'} \tag{5.3}$$

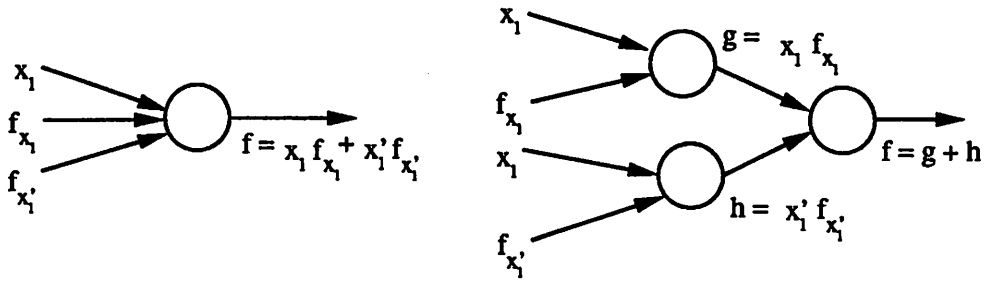
As mentioned in Section 3.3.3, for $m > 2$, we need one LUT to realize f as in (5.3) (Figure 5.2 (A)), whereas for $m = 2$, we need 3 LUTs (Figure 5.2 (B)). We recursively decompose f_{x_1} and $f_{x_1'}$, which are functions of at most $n - 1$ variables. This leads to the following recurrence inequality:

$$C_m(S(n)) \leq \begin{cases} 2C_m(S(n-1)) + 1 & \text{if } m > 2, \\ 2C_m(S(n-1)) + 3 & \text{if } m = 2. \end{cases} \tag{5.4}$$

The boundary condition is $C_m(S(m)) = 1$. Solving the recurrence, we get (5.2). ■

Later we will improve the bound for $m = 2$.

We must mention the following classical result by Lupanov [51]. Here, a **simple gate** is either an *inverter*, a two-input AND gate, or a two-input OR gate.



(A) m > 2 (B) m = 2

Figure 5.2: Using cofactoring for decomposition

Theorem 5.2.4 (Lupanov [51]) *Every function of n variables is realizable with* $\frac{2^n}{cn}$ *simple gates for some c.*

The proof is constructive and, in the light of Proposition 5.2.2, gives an optimum realization (to within a constant factor) for almost all functions.

Miscellaneous

The following result shows the effect on complexity when m is changed to k.

Proposition 5.2.5 (Savage [70]) *Let k and m be two constants such that* $k \leq m$. *Then*

$$C_k(f) \leq AC_m(f), \tag{5.5}$$

where A is a constant.

Proof Let η be an optimum m-feasible realization of f, so it uses $C_m(f)$ m-LUTs. Let $A = C_k(S(m))$. Then, the function implemented by each m-LUT in η can be replaced by a sub-network with no more than A k-LUTs. This results in a k-feasible realization of f using at most $AC_m(f)$ k-LUTs. ■

Thus the complexity remains within a constant when the number of inputs to the LUT is changed.

5.3 New Results

First, the problem of realizing.an arbitrary n-input function using m-LUTs is addressed. Section 5.3.1 derives an upper bound on the complexity of such a function, which is an improvement

over the bound in Proposition 5.2.3. The exact value of $C_m(S(m+1))$ is presented in Section 5.3.2. The next two sections are devoted to deriving upper bounds on the complexity, given a representation of the function. This is useful, since in a synthesis environment, such a representation already exists. Two representations are considered: the SOP and the factored form. Given an SOP, Section 5.3.3 provides an upper bound in terms of the numbers of cubes and literals in the SOP. Finally, Section 5.3.4 provides an upper bound in terms of the number of literals in a given factored form. This result is extended for a multi-level, multi-output Boolean network.

5.3.1 Complexity Bound for an n-input Function

The technique presented in the proof of Proposition 5.2.3 to realize an n-input function f results in a structure in which each non-leaf LUT is 3-feasible. It may be possible to collapse some of the LUTs into their fanout LUTs, while maintaining m-feasibility (for $m \geq 4$). This improves the bound for $C_m(S(n))$. Here, we only describe the result for $m = 5$.

Proposition 5.3.1

$$C_5(S(n)) \leq \begin{cases} 2^{n-4} - 1 - \frac{2^{n-5}-1}{3} & \textit{for odd } n, \\ 2^{n-4} - 1 - \frac{2^{n-5}-2}{3} & \textit{for even } n. \end{cases} \tag{5.6}$$

Proof If we substitute $m = 5$ in (5.2), we get $C_5(S(n)) \leq 2^{n-4} - 1$. This bound corresponds to a cofactor tree that is truncated at the leaves when a 5-feasible function is reached. We assume that barring the truncation at the leaves, the tree is complete, i.e., has exactly $2^{n-4} - 1$ nodes. This means that if on cofactoring some tree node function with respect to x_i, a function g is obtained that is independent of x_{i+1}, g is cofactored with respect to x_{i+1} and replicated twice one level below. First consider the case when n is odd. Let $n = 2k + 1$. The cofactor tree T for this case is shown in Figure 5.3. The maximum possible number of inputs in the global support of a node function, called the *label of the node*, is shown at each level in the tree on the right. At each level, we have chosen to cofactor with respect to the same variable.[1] Since each non-leaf node of T has at most 3 inputs (a leaf node may have as many as 5 inputs), some of the nodes of T can be collapsed into their parents without destroying 5-feasibility. The strategy for collapsing is the following. Given a non-leaf node t with an odd label (so its label is at least 7), collapse one of its two children into it. The resulting node has at most 5 inputs. Each collapsed node is a node saved. The total savings are then the number of non-leaf, odd-labeled nodes. T has 1 node with label $n = 2k + 1$, 2^2 nodes

[1] It is not necessary for this proof, but for $m = 6$ better bound results if this strategy is followed.

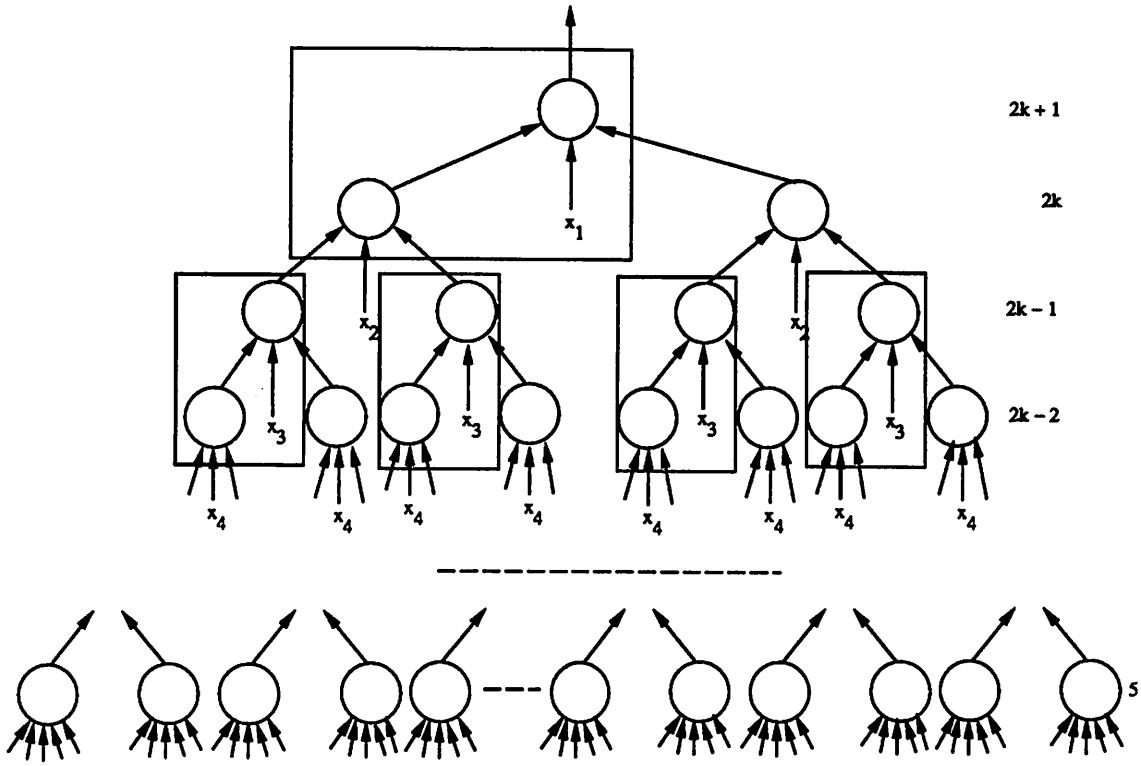


Figure 5.3: Cofactor tree for f for $m = 5$ and $n = 2k + 1$

with label $n - 2 = 2k - 1$, and 2^{n-7} nodes with label 7.

$$\begin{aligned}
\text{Total savings} &= 1 + 2^2 + 2^4 + \ldots + 2^{n-7} \\
&= 1 + 2^2 + 2^4 + \ldots + 2^{2k-6} \\
&= \frac{(2^2)^{k-2} - 1}{3} \\
&= \frac{2^{n-5} - 1}{3}
\end{aligned}$$

Hence the number of nodes in the resulting tree is $2^{n-4} - 1 - \frac{2^{n-5}-1}{3}$.

Similar argument is used for even n. ■

Similar bounds may be derived for different values of m. However, the cofactoring or collapsing strategies may need to be modified in order to derive better bounds. For instance, for $6 \leq m \leq 11$, cofactoring should be done with respect to two input variables.

Asymptotically, Lupanov bound of Theorem 5.2.4 is better than that of Proposition 5.3.1. However, for some small values of n, Proposition 5.3.1 gives a better bound.

5.3.2 Implementing $(m+1)$-input Functions with m-LUTs

Most optimized multi-level networks have simple node functions - with 3 to 10 inputs. The entire set of optimized benchmarks of Chapter 3 has a total of 4694 internal nodes. Out of these node functions, 4131 have at most 7 fanins and 4368 have at most 10. The average number of fanins to each node is just below 4. Note that in commercial LUT architectures, m is small - e.g., 4 or 5. This motivated us to restrict the problem of determining the complexity of arbitrary functions (which is a hard problem) to that of determining the complexity of the class of $(m+k)$-input functions, where k is a small constant, say less than 5. Now we solve for $k = 1$. We show that for $m \geq 3$, three m-LUTs, and, for $m = 2$, four m-LUTs suffice for an $(m+1)$-input function. Moreover, these bounds are tight, i.e., for each $m \geq 2$, there exists a function of $(m+1)$ inputs that cannot be realized with fewer m-LUTs.

Theorem 5.3.2

$$C_m(S(m+1)) = \begin{cases} 3 & \text{if } m \geq 3 \\ 4 & \text{if } m = 2 \end{cases}$$

We use the classical decomposition theory of Ashenhurst [3] to prove this theorem. We stated the main decomposition result, Theorem 3.3.1, in Chapter 3. This result, however, deals only with disjoint decomposition, i.e., when the bound set and the free set are disjoint. To prove our result, we need to consider non-disjoint decomposition as well. We now extend the theory of Ashenhurst to handle such decompositions.

Consider $f(X,Y,Z)$, where $X \cap Y = X \cap Z = Y \cap Z = \phi$ such that the bound set is $X \cup Z$ and the free set $Y \cup Z$. So Z is the set of variables common to the two sets. For the decomposition to be non-trivial, we require that $X, Y \neq \phi$. However, to model disjoint decomposition, Z is allowed to be empty. For example, let $f(a,b,c) = abc + a'b'c$. Let $X = \{b\}, Y = \{c\}$, and $Z = \{a\}$. Then the bound set is $\{a,b\}$, and the free set is $\{a,c\}$. The corresponding decomposition chart is shown in Figure 5.4. The columns are indexed by the bound set variables and the rows by the free set variables. The "-" entries are don't cares, since for each of these entries, a takes on conflicting values in the bound and the free sets. The common variables, Z, of the two sets are written before other variables. Also, we follow the convention that the row and the column numbers start from 0. The assignment of the variables Z and Y corresponding to i^{th} row is obtained from the binary representation of i using $|Z \cup Y|$ binary symbols. The same holds for the variables Z and X for the columns. For example, in the chart of Figure 5.4, 0^{th} row corresponds to the assignment $Z, Y = 00$,

$\frac{ab}{ac}$	00	01	10	11
00	0	0	-	-
01	1	0	-	-
10	-	-	0	0
11	-	-	0	1

Figure 5.4: Decomposition chart for a non-disjoint decomposition

$\frac{ab}{ac}$	00	01
00	0	0
01	1	0

Figure 5.5: $\mathcal{B}_0$

i.e., $a = 0, b = 0$, whereas the first row corresponds to the assignment $Z, Y = 01$, i.e., $a = 0, b = 1$. Given these conventions, it is easily seen that the meaningful entries of the decomposition chart consists of **diagonal blocks** (not necessarily square), the non-diagonal blocks being "-". Each diagonal block corresponds to an assignment to the Z variables. For example, the chart of Figure 5.4 has two diagonal blocks. The block at the top left, $\mathcal{B}_0$, corresponds to $Z = a = 0$, and the second one, $\mathcal{B}_1$, to $Z = a = 1$. This is shown in Figures 5.5 and 5.6 respectively. Note that if $Z = \phi$, the entire chart is a diagonal block. Also note that the columns in a diagonal block correspond to all possible vertices in $B^{|X|}$, and the rows to all possible vertices in $B^{|Y|}$.

Next, we introduce the notion of a trivial row of a decomposition chart or a diagonal block.

Definition 5.3.1 *A row of a decomposition chart (or a diagonal block) is* **trivial** *if all its entries are zeros or if all its entries are ones. Otherwise, it is said to be* **non-trivial**.

$\frac{ab}{ac}$	10	11
10	0	0
11	0	1

Figure 5.6: $\mathcal{B}_1$

It turns out that the characterization of (possibly non-disjoint) decomposition is similar to that of disjoint decomposition, except that it is in terms of the diagonal blocks.

Theorem 5.3.3 *A simple (possibly non-disjoint) decomposition of* $f(X,Y,Z)$ *with* $X \cap Y = X \cap Z = Y \cap Z = \phi$, *and* $X, Y \neq \phi$,

$$f(X,Y,Z) = g(\alpha(X,Z),Y,Z) \tag{5.7}$$

exists if and only if each diagonal block of the corresponding decomposition chart $D(X \cup Z | Y \cup Z)$ *has a column multiplicity of at most 2.*

Proof If $Z = \phi$, the complete decomposition chart is one diagonal block, and the theorem reduces to the disjoint decomposition case, i.e., Theorem 3.3.1. So let $Z \neq \phi$. Our proof is an extension of the proof of Ashenhurst's Fundamental Theorem of Decomposition (Theorem 3.3.1) [3].

($\Rightarrow$): We are given that

$$f(X,Y,Z) = g(\alpha(X,Z),Y,Z),$$

where α is a single-output Boolean function. For the sake of contradiction, assume that a diagonal block $\mathcal{B} = (b_{ij})$ of the decomposition chart D has more than 2 distinct column patterns. Let $\mathcal{B}$ correspond to the assignment $Z = z$. We index $\mathcal{B}$'s columns with X and rows with Y (so the 0^{th} row of $\mathcal{B}$ corresponds to $Y = 0$, and so on). Since $\mathcal{B}$ has more than 2 distinct column patterns, it has two non-trivial rows i and j (corresponding to $Y = i$ and $Y = j$ respectively) that are neither identical nor complementary. Then there exist two columns k and l such that

$$b_{ik} = b_{il} = A, \text{ and} \tag{5.8}$$

$$b_{jk} \neq b_{jl}, \tag{5.9}$$

where A is a constant - either 0 or 1. These conditions can be rewritten as

$$f(k,i,z) = f(l,i,z) = A, \text{ and} \tag{5.10}$$

$$f(k,j,z) \neq f(l,j,z). \tag{5.11}$$

An example is shown in Figure 5.7, with $A = 0$. Two cases arise:

$\frac{X}{Y}$	k	.	.	l
i	0	.	.	0
.	.	.	.	.
.	.	.	.	.
j	0	.	.	1

Figure 5.7: A portion of the diagonal block $\mathcal{B}$

$\frac{ZX}{ZY}$				
	$\mathcal{B}_0$	-	-	-
	-	$\mathcal{B}_1$	-	-
	-	-	$\mathcal{B}_2$	-
	-	-	-	$\mathcal{B}_3$

Figure 5.8: Diagonal blocks in a portion of the decomposition chart D

1. $\alpha(k,z) = \alpha(l,z)$: Then,

$$\begin{aligned} f(k,j,z) &= g(\alpha(k,z),j,z) \\ &= g(\alpha(l,z),j,z) \\ &= f(l,j,z). \end{aligned}$$

   A contradiction to (5.11)!

2. $\alpha(k,z) \neq \alpha(l,z)$ (one is 0 and the other is 1): Consider a column of $\mathcal{B}$ corresponding to $X = p$. Since $\alpha(p,z) = 0$ or 1, either $\alpha(p,z) = \alpha(k,z)$ or $\alpha(p,z) = \alpha(l,z)$. Without loss of generality, assume $\alpha(p,z) = \alpha(l,z)$. Then

$$\begin{aligned} f(p,i,z) &= g(\alpha(p,z),i,z) \\ &= g(\alpha(l,z),i,z) \\ &= f(l,i,z) \\ &= A. \end{aligned}$$

   Since p was chosen arbitrarily, this means that the row i of $\mathcal{B}$ is constant, i.e., trivial. This contradicts the fact that the row i is non-trivial!

($\Leftarrow$): Figure 5.8 shows some diagonal blocks in the decomposition chart $D(ZX|ZY)$ for f. For each $Z = z$, diagonal block $\mathcal{B}_z$ has at most two distinct column patterns. Then, it follows from a lemma proved by Ashenhurst [3] that $\mathcal{B}_z$ has at most four distinct row patterns: 0, 1, α_z, α_z'.[2] The row 0(1) means that all the entries in that row are 0s(1s). Each α_z is a function of the variables X. For the chart of Figure 5.4, if we label the non-zero row of $\mathcal{B}_0$ (Figure 5.5) as α_0 and the non-zero row of $\mathcal{B}_1$ (Figures 5.6) as α_1, it is easily seen that $\alpha_0 = b'$ and $\alpha_1 = b$. Define

$$\alpha(X, Z) = \sum_{z \in B^{|Z|}} (Z^z \alpha_z(X)), \tag{5.12}$$

where Z^z, borrowed from Brown [13], is defined as follows. For a Boolean variable x, and $a \in \{0, 1\}$, define x^a as

$$x^0 = x', \quad x^1 = x. \tag{5.13}$$

The notation is extended to vectors as follows. For $X = (x_1, x_2, \ldots, x_n)$, x_i Boolean variables, and $A = (a_1, a_2, \ldots, a_n) \in \{0, 1\}^n$, define X^A as

$$X^A = {x_1}^{a_1} {x_2}^{a_2} \cdots {x_n}^{a_n}. \tag{5.14}$$

Then, interpreting z as a binary n-tuple, Z^z becomes meaningful in (5.12). Note that when $Z = z$, $\alpha(X, Z) = \alpha_z(X)$, since for $Z = z$, all $Z^{\tilde{z}}, \tilde{z} \neq z$, evaluate to 0, and Z^z evaluates to 1. For the chart of Figure 5.4, $\alpha(X, Z) = \alpha(b, a) = a^0 \alpha_0(b) + a^1 \alpha_1(b) = a'b' + ab$.

If for some $\mathcal{B}_z$, only one distinct column exists, which implies that the only row vectors are 0 and 1, define α_z to be 0.

The final step is to define $g(\alpha, Y, Z)$ such that (5.7) holds. An assignment $Z = z$ corresponds to the diagonal block $\mathcal{B}_z$. The assignment of Y fixes the row in $\mathcal{B}_z$. Depending on whether the row vector is 0, 1, α_z, or α_z', g is defined, as in Figure 5.9. It can be checked that the value of g is the same as that of f for all values of X, Y, and Z. For instance, when the label of the row determined by (Y, Z) is 0, the row has all zeros and f is 0 for the entire row. Then, g is set to 0, irrespective of α. A similar argument works if the label of row (Y, Z) is 1. For the row with label α_z, f is equal to $\alpha_z = \alpha$, and so is g. A similar argument works if the label of the row (Y, Z) is α_z'.

In the example of Figure 5.4, $g(\alpha, Y, Z) = g(\alpha, c, a) = a'c\alpha + ac\alpha = c\alpha$. Clearly, $g(\alpha(b, a), c, a) = c\alpha = c(a'b' + ab) = f(a, b, c)$.

[2]Deciding whether a row pattern is labeled α_z or α_z' is arbitrary.

label of row (Y,Z)	α	$g(\alpha,Y,Z)$
0	0	0
0	1	0
1	0	1
1	1	1
α_z	0	0
α_z	1	1
$\alpha_z{}'$	0	1
$\alpha_z{}'$	1	0

Figure 5.9: Defining g for a valid decomposition

■

The proof is constructive, i.e., it generates functions α and g as a by-product. It also provides information about all different choices for α. These correspond to the choices made in labeling the rows of the diagonal blocks. A question that remains unanswered is that of uniqueness of g, given f and α. Next, we present necessary and sufficient conditions for g to be unique. Since the result also holds for a non-simple decomposition (i.e., when $\alpha = \vec{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_t)$), it is stated in full generality.

Proposition 5.3.4 *Given $f(X,Y,Z)$, $\vec{\alpha}(X,Z) = (\alpha_1(X,Z), \alpha_2(X,Z), \ldots, \alpha_t(X,Z))$, and that a decomposition of f exists as*

$$f(X,Y,Z) = g(\vec{\alpha}(X,Z), Y, Z). \tag{5.15}$$

Then, g is unique if and only if for each $v \in B^t$, there exists an (x,z) pair for each $z \in B^{|Z|}$ such that $\vec{\alpha}(x,z) = v$.

Proof ($\Rightarrow$) : Assume that for some $v \in B^t$, there is a $z \in B^{|Z|}$ such that for all $x \in B^{|X|}$, $\vec{\alpha}(x,z) \neq v$. Then we can define $\tilde{g}$ such that $\tilde{g}(v,y,z) \neq g(v,y,z)$ (if $g(v,y,z) = 0$, $\tilde{g}(v,y,z)$ is set to 1, and vice versa), where y is an arbitrary vertex in $B^{|Y|}$. On all other triples, $\tilde{g}$ is defined to be the same as g. Clearly $\tilde{g} \neq g$. To see that $\tilde{g}$ gives a valid decomposition for f, i.e.,

$$f(X,Y,Z) = \tilde{g}(\vec{\alpha}(X,Z), Y, Z), \tag{5.16}$$

it suffices to check that (5.16) is satisfied at $Y = y, Z = z$ (on all other points, $\tilde{g}$ agrees with g, and g satisfies (5.15)). Pick any $x \in B^{|X|}$. Now, $\vec{\alpha}(x,z) = \tilde{v} \neq v$ (by assumption). Then,

using the definition of $\tilde{g}$,

$$\begin{aligned}\tilde{g}(\vec{\alpha}(x,z),y,z) &= \tilde{g}(\tilde{v},y,z)\\ &= g(\tilde{v},y,z) \text{ [since } v \neq \tilde{v}]\\ &= g(\vec{\alpha}(x,z),y,z)\\ &= f(x,y,z).\end{aligned}$$

Thus, (5.16) is satisfied at $Y = y, Z = z$, and all $x \in B^{|X|}$. This leads to a contradiction, since only a unique g satisfies (5.15).

($\Leftarrow$) : Let g_1, g_2 ($g_1 \neq g_2$) satisfy (5.15). Then for some v, y, z, $g_1(v,y,z) \neq g_2(v,y,z)$. Consider an x such that $\vec{\alpha}(x,z) = v$, the existence of such an x being guaranteed by assumption. Then, $f(x,y,z) = g_1(\vec{\alpha}(x,z),y,z) = g_2(\vec{\alpha}(x,z),y,z)$. This implies that $g_1(v,y,z) = g_2(v,y,z)$. A contradiction!

■

For a disjoint decomposition, $Z = \phi$, and Proposition 5.3.4 reduces to the following.

Corollary 5.3.5 *Given $f(X,Y)$, $\vec{\alpha}(X) = (\alpha_1(X), \alpha_2(X), \ldots \alpha_t(X))$, and that a decomposition of f exists as*

$$f(X,Y) = g(\vec{\alpha}(X),Y). \tag{5.17}$$

Then, g is unique if and only if for each $v \in B^t$, there exists an x such that $\vec{\alpha}(x) = v$.

In other words, if all the minterms in the $\vec{\alpha}$-space are *used*, there are no don't cares associated with g, and so g is unique. If some minterm in the $\vec{\alpha}$-space is not used, it is a don't care for g, and g is no longer unique.

Now we are ready to prove Theorem 5.3.2. We do so by proving Lemmas 5.3.6 and 5.3.11.

Lemma 5.3.6

$$C_m(S(m+1)) \leq \begin{cases} 3 & \text{if } m \geq 3 \\ 4 & \text{if } m = 2 \end{cases}$$

Proof For $m \geq 3$, the desired bound, 3, is obtained by putting $n = m+1$ in (5.2). The case $m = 2$ is more complicated. If we substitute $n = m + 1 = 3$ in (5.2), an upper bound of 5 is obtained, which corresponds to the following decomposition of $f(a, b, c)$:

$$f = g + h, \tag{5.18}$$

$$g = cf_c, \tag{5.19}$$

$$h = c'f_{c'}. \tag{5.20}$$

The functions f_c and $f_{c'}$, being 2-feasible, can be realized with one LUT each. This bound can be lowered to 4 by showing for each 3-input function a realization that uses 4 LUTs. The technique used does not show explicitly a realization for all such functions. It removes many functions from consideration, since the complexity of these functions is easily seen to be either at most 4 or at most the complexity of some function that is not removed from consideration. To prove the bound of 4, consider a 3-input function $f(a, b, c)$. It can be written as

$$f(a, b, c) = cf_c + c'f_{c'}.$$

The functions f_c and $f_{c'}$ depend only on a and b. Since the pair $(f_c, f_{c'})$ determines f uniquely, to generate all 3-input functions, it suffices to consider all ordered pairs of functions that have a and b as inputs. The following proposition removes some functions from consideration if one of its cofactors is trivial.

Proposition 5.3.7 *Given a function* $f(a, b, c)$,

1. $f_c = \mathbf{0}$ *or* $f_c = \mathbf{1} \Rightarrow C_2(f) \leq 2$.
2. $f_c = a$ *or* $f_c = b$ *or* $f_c = a'$ *or* $f_c = b' \Rightarrow C_2(f) \leq 4$.

Proof 1. $f_c = \mathbf{0} \Rightarrow f = c'f_{c'}$, which can be realized with two 2-LUTs. If $f_c = \mathbf{1}$, $f = c + c'f_{c'} = c + f_{c'}$, which is also realizable with two 2-LUTs.

2. If $f_c = a$ or $f_c = b$, f_c does not need an LUT. If $f_c = a'$ or $f_c = b'$, the LUT that realizes f_c is an inverter and can be removed by absorbing the inverter into the fanout LUT.

■

So, out of sixteen possible 2-feasible functions, only the following ten need be considered as the candidates for f_c and $f_{c'}$ (the case numbers 0 through 9 are used for labeling the functions):

0. ab

1. ab'
2. $a'b$
3. $a'b'$
4. $a+b$
5. $a+b'$
6. $a'+b$
7. $a'+b'$
8. $ab+a'b'$
9. $a'b+ab'$.

The following labeling notation is used. Functions have case numbers as their subscripts. If the case number for a function f is ij, $0 \leq i,j \leq 9$, then f_c and $f_{c'}$ correspond to the case numbers i and j respectively in the above list of ten functions. For example, $f_{01}(a,b,c) = cab + c'ab'$ is obtained from the pair $(f_c, f_{c'}) = (ab, ab')$, which are functions numbered 0 and 1 respectively in the list.

At this point, there are 100 candidates for f. If $f_c = f_{c'}$, then f is independent of c, it has only two inputs, and therefore is realizable by one 2-LUT. This eliminates all functions f_{ii} from consideration. Hence for a fixed f_c, we need to consider 9 choices for $f_{c'}$. Now 90 choices remain for $f = (f_c, f_{c'})$. All these choices correspond to non-trivial functions, since each such f essentially depends on c, and so is not identically 1 or identically 0. Moreover, $f \neq c$ and $f \neq c'$, because $f_c \neq 1$ and $f_c \neq 0$. Now, we can use the next proposition to reduce the number of choices by half.

Proposition 5.3.8 *Let $f(a,b,c)$ be a non-trivial function. Let $g(a,b,c) = cf_{c'} + c'f_c$. Then $C_2(g) = C_2(f)$.*

Proof $f = cf_c + c'f_{c'}$. Let $H(a,b) = f_c, I(a,b) = f_{c'}$. Then,

$$\begin{aligned} f(a,b,c) &= cH(a,b) + c'I(a,b), \\ g(a,b,c) &= cI(a,b) + c'H(a,b), \\ \Rightarrow g(a,b,c) &= f(a,b,c'). \end{aligned}$$

Since inverters are free in LUT architectures for a non-trivial function, the result follows. ■

This proposition allows us to consider only the two-element *sets* of two-input functions as the candidates for $(f_c, f_{c'})$ rather than *ordered pairs*.

Further, we can eliminate from consideration functions satisfying the condition of the next proposition.

Proposition 5.3.9 *If $f_{c'} = f_c'$ for a function $f(a,b,c)$, then f can be realized with two 2-LUTs.*

Proof Since $f(a,b,c) = cf_c + c'f_{c'} = cf_c + c'f_c' = \tilde{f}(c, f_c)$. Both $\tilde{f}$ and f_c need at most one LUT each, and so f can be realized with two 2-LUTs. ■

Proposition 5.3.10 *A (generalized) factored form with $\ell(\ell > 1)$ literals can be realized with $(\ell - 1)$ 2-LUTs.*

Proof There are $(\ell - 1)$ binary operations in the factored form, and one 2-LUT suffices for each such operation. ■

Now, we are ready to show 2-feasible realizations of the remaining 3-input functions. In the following analysis, Proposition 5.3.10 will be the default reason for the upper bound on $C_2(f)$.

01. $f = abc + ab'c'$. Since $f_{a'} = \mathbf{0} \Rightarrow C_2(f) \leq 2$ (Proposition 5.3.7)
02. $f = abc + a'bc'$. Since $f_{b'} = \mathbf{0} \Rightarrow C_2(f) \leq 2$ (Proposition 5.3.7)
03. $f = abc + a'b'c' = (a'b' + ab)(a'c' + ac) = (a\overline{\oplus}b)(a\overline{\oplus}c)$[3] $\Rightarrow C_2(f) \leq 3$ (Proposition 5.3.10)
04. $f = abc + (a + b)c' = ab + ac' + bc' = a(b + c') + bc' \Rightarrow C_2(f) \leq 4$
05. $f = abc + (a + b')c' = ab + b'c' \Rightarrow C_2(f) \leq 3$
06. $f = abc + (a' + b)c' = ab + a'c' \Rightarrow C_2(f) \leq 3$
07. $f = abc + (a' + b')c'$. From Proposition 5.3.9, $C_2(f) \leq 2$
08. $f = abc + (ab + a'b')c' = ab + a'b'c' \Rightarrow C_2(f) \leq 4$
09. $f = abc + (ab' + a'b)c' = (ab\overline{\oplus}c)(a + b) \Rightarrow C_2(f) \leq 4$

Cases 12 to 19 can be derived from appropriate cases 01 to 09 by replacing b by b'. For example, $f_{12}(a,b,c) = ab'c + a'bc' = f_{03}(a,b',c)$. Similarly, cases 23 to 29 can be derived by replacing a by a', and 34 to 39 by replacing a and b by a' and b' respectively, in cases 01 to 09.

[3] $\overline{\oplus}$ denotes EX-NOR operation.

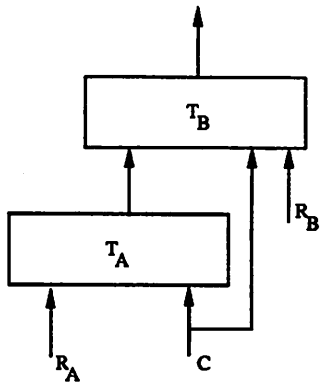


Figure 5.10: The configuration for two LUTs

45. $f = (a+b)c + (a+b')c' = a + (bc + b'c') = a + (b \overline{\oplus} c) \Rightarrow C_2(f) \leq 2$
46. $f = (a+b)c + (a'+b)c' = (ac + a'c') + b = (a \overline{\oplus} c) + b \Rightarrow C_2(f) \leq 2$
47. $f = (a+b)c + (a'+b')c' = (ac + a'c') + (bc + b'c') = (a \overline{\oplus} c) + (b \overline{\oplus} c) \Rightarrow C_2(f) \leq 3$
48. $f = (a+b)c + (ab + a'b')c' = ab + ac + bc + a'b'c' = (a+b) \oplus c' + ab \Rightarrow C_2(f) \leq 4$
49. $f = (a+b)c + (ab' + a'b)c' = ab' + a'b + bc = (a \oplus b) + bc \Rightarrow C_2(f) \leq 3$

Cases 56 through 59, 67 through 69, and 78 through 79 can be derived from 45 through 49 by replacing a or b or both by a', b' appropriately.

89. $f = (ab + a'b')c + (ab' + a'b)c'$. From Proposition 5.3.9, $C_2(f) \leq 2$

This proves Lemma 5.3.6. ■

Lemma 5.3.11

$$C_m(S(m+1)) \geq \begin{cases} 3 & \text{if } m \geq 3, \\ 4 & \text{if } m = 2. \end{cases}$$

Proof We prove the lemma by first proving a few propositions.

Proposition 5.3.12 *Consider a configuration of two* m-LUT*s* T_A *and* T_B *(Figure 5.10), which realizes a function* $f \in S(m+1)$ *that essentially depends on all* $m+1$ *inputs. Then at least two inputs of* f *are only connected to* T_A*, and at least one input only connected to* T_B*.*

$\frac{C\ldots x_1x_2}{C\ldots x_3}$	1..100	1..101	1..110	1..111
1..10	0	0	0	1
1..11	0	1	1	0

Figure 5.11: A portion of the decomposition chart for f

Proof As shown in Figure 5.10, let C be the (possibly empty) set of common inputs to both T_A and T_B, and R_A (R_B) be the rest of the inputs of $T_A(T_B)$. Then,

$$|C| + |R_A| \leq m, \text{(since } T_A \text{ can have only } m \text{ inputs)}, \tag{5.21}$$

$$|C| + |R_B| \leq m - 1, \text{(since one input of } T_B \text{ is the output of } T_A), \tag{5.22}$$

$$|C| + |R_A| + |R_B| = m + 1, \text{total number of inputs of } f \tag{5.23}$$

From (5.21) and (5.23), $|R_B| \geq 1$, and from (5.22) and (5.23), $|R_A| \geq 2$. ■

Proposition 5.3.13 *For each $m \geq 2$, there exists a function of $(m+1)$ inputs that cannot be realized with two m-LUTs.*

Proof Consider

$$f(x_1, x_2, \ldots, x_{m+1}) = x_1{}'x_2 \ldots x_{m+1} + x_1x_2{}' \ldots x_{m+1} + \ldots + x_1x_2 \ldots x_{m+1}{}'$$

Note that f is a totally symmetric function. Also, it essentially depends on all $(m+1)$ inputs and, hence, cannot be realized with one m-LUT. Let f have a realization with two m-LUTs T_A and T_B of Figure 5.10. This corresponds to a simple decomposition of f with the bound set $C \cup R_A$, and the free set $C \cup R_B$. From Proposition 5.3.12, without loss of generality, let x_1 and x_2 be the inputs of f that are connected only to T_A, and x_3 be the input connected only to T_B. The bound set can then be written as $C \ldots x_1x_2$, and the free set as $C \ldots x_3$. Consider the portion of the decomposition chart for f corresponding to the assignment $(x_4, x_5, \ldots, x_{m+1}) = (1, 1, \ldots, 1)$ (i.e., all 1s), and all possible values for x_1, x_2, and x_3 (shown in bold in Figure 5.11). This portion, shown in Figure 5.11, is a part of a diagonal block, and has three distinct column patterns. This leads to a contradiction, since the configuration of Figure 5.10 implies the existence of a simple decomposition. Recall from Theorem 5.3.3 that for a function to have a simple decomposition, each diagonal block must have at most 2 distinct column patterns. Hence f cannot be realized with two m-LUTs. ■

$\frac{ab}{c}$	00	01	10	11
0	1	0	0	1
1	0	1	1	1

Figure 5.12: Checking if f is realizable with two 2-LUTs

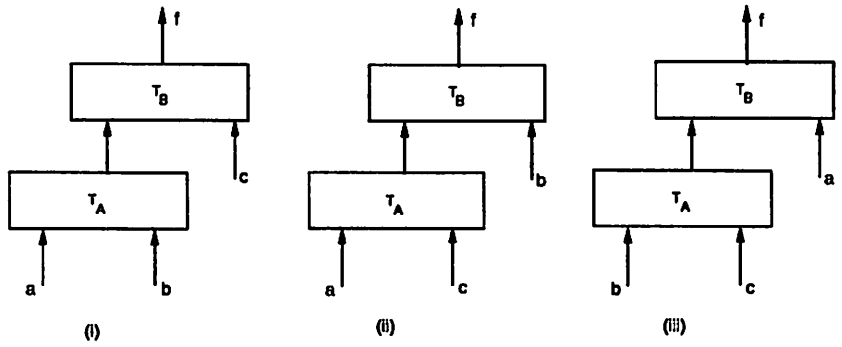


Figure 5.13: Possible cases for two LUTs

This establishes the lemma for $m \geq 3$, since we have shown that $C_m(S(m+1)) \geq 3$. For $m = 2$, we have to show that there exists a 3-input function that cannot be realized with less than four 2-LUTs. Consider the function $f(a, b, c)$:

$$f(a, b, c) = ac + bc + ab + a'b'c'$$

Once again, f is totally symmetric and essentially depends on all the variables. Our proof strategy is to assume that f can be realized with two and then three 2-LUTs and reach a contradiction.

1. *Two* LUT *realizability*: As Figure 5.13 shows, there are three ways in which inputs can be assigned to the pins of the LUTs. However, since f is totally symmetric, only one need be considered. We examine (i). Here, the decomposition chart corresponds to the partition $ab|c$ and is shown in Figure 5.12. There are three distinct column patterns, resulting in a contradiction.

2. *Three* LUT *realizability*: This is a difficult case to analyze, since many configurations are possible. Furthermore, for each configuration, there are many assignments to the pins of the LUTs. We first enumerate all possible configurations for three LUTs, then assign the inputs of f to the pins of the LUTs, and finally check if the conditions for a valid decomposition are satisfied.

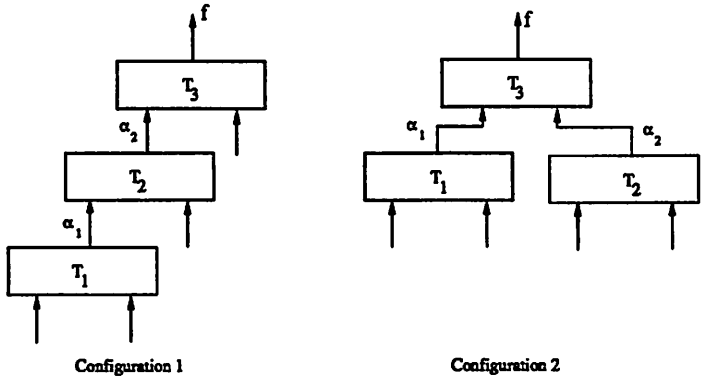


Figure 5.14: Configurations for three LUTs

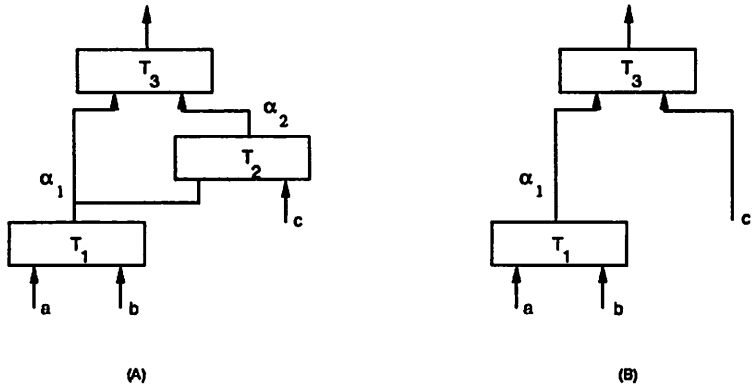


Figure 5.15: T_1 feeding into T_2 is an unnecessary case

Finding all possible configurations: We claim that only two configurations, shown in Figure 5.14, need to be considered. This can be seen as follows. Consider the root node T_3 of the configuration. It either has

(a) one input from another LUT, T_2, and the other one from an input of f. Then necessarily T_2 receives one input from T_1 and the other from f. Clearly, T_1 has both its input pins tied to some inputs of f. This is the configuration 1, or

(b) both inputs from LUTs, T_1 and T_2. If T_1 was providing an input to T_2 as well (Figure 5.15 (A)), f can be realized with two LUTs (collapse T_2 into T_3), as shown in Figure 5.15 (B). Hence T_1 and T_2 receive both their inputs from inputs of f. This results in the configuration 2.

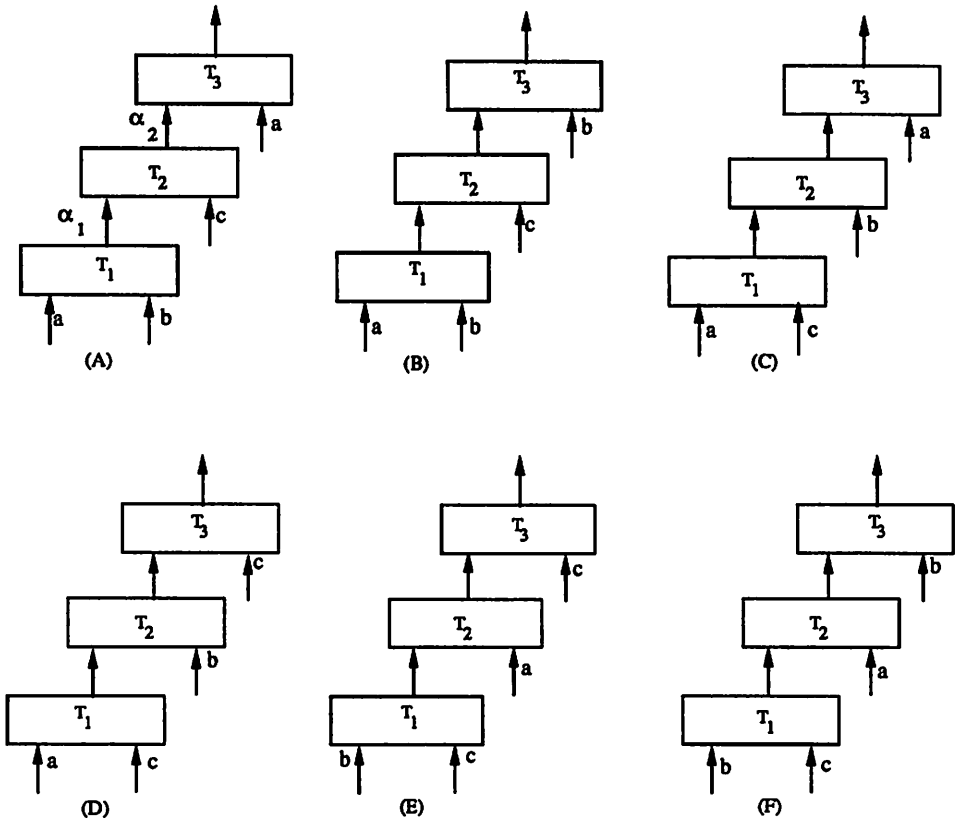


Figure 5.16: Possible cases for Configuration 1

Comment: No LUT may have just a single input, otherwise f can be realized with fewer LUTs.

We now consider each configuration, and assign the inputs of f to the pins of the LUTs.

Configuration 1: All possible cases are shown in Figure 5.16. We do not need to consider cases where the same input feeds two adjacent LUTs (say, T_1 and T_2), because f could be realized with fewer LUTs (remove T_1 by collapsing it into T_2). We just consider (A). The rest follow from symmetry. Let α_1 be the output of T_1. Figure 5.16 (A) implies that $f = ab + ac + bc + a'b'c'$ has a decomposition of the form $f(a,b,c) = g(\alpha_1(a,b),a,c)$. This corresponds to the bound set $\{a,b\}$ and the free set $\{a,c\}$. The decomposition chart for f with these sets is shown in Figure 5.17. We first derive all possible candidates for α_1. Since the maximum number of distinct columns in both diagonal blocks is 2, α_1 exists. There are four choices for α_1, two of them being trivial, namely $\alpha_1 = b$ and $\alpha_1 = b'$. These can be ignored, since they correspond to T_1 using only one input, in which case T_1 can be collapsed

$\frac{ab}{ac}$	00	01	10	11
00	1	0	-	-
01	0	1	-	-
10	-	-	0	1
11	-	-	1	1

Figure 5.17: Checking if f can be realized by configuration 1 (A)

$\frac{\alpha_1 c}{a}$	00	01	10	11
0	0	1	1	0
1	0	1	1	1

Figure 5.18: Decomposition chart for g

into T_2, yielding a smaller realization of f. The non-trivial choices are

(a) $\alpha_1 = a'b' + ab$.

(b) $\alpha_1 = ab' + a'b$.

Since these two are related to each other by an inversion, we need to consider only one, say $\alpha_1 = a'b' + ab$. Then, using the rules of Figure 5.9, $g(\alpha_1, a, c) = a'c'\alpha_1 + a'c\alpha_1' + a\alpha_1 + ac\alpha_1' = c'\alpha_1 + c\alpha_1' + a\alpha_1$. It can be checked that α_1 satisfies the conditions of Proposition 5.3.4 and hence, g is unique. Figure 5.16 (A) corresponds to the decomposition of g of the form

$$g(\alpha_1, a, c) = h(\alpha_2(\alpha_1, c), a) \tag{5.24}$$

The decomposition chart of g is shown in Figure 5.18. 3 distinct column patterns indicate that this decomposition is not possible. A contradiction! So f cannot be realized with three 2-LUTs using configuration 1.

Configuration 2: As shown in Figure 5.19, there are three possible ways to assign a, b, and c to the pins of the LUTs. Once again, using symmetry, we only consider (A). Let $g(\alpha_1, \alpha_2)$ be the function realized by T_3. Then $f = g(\alpha_1(a, b), \alpha_2(a, c))$. The functions α_1 and g are the same as those for the configuration 1 (A).

$$\alpha_1 = a'b' + ab,$$

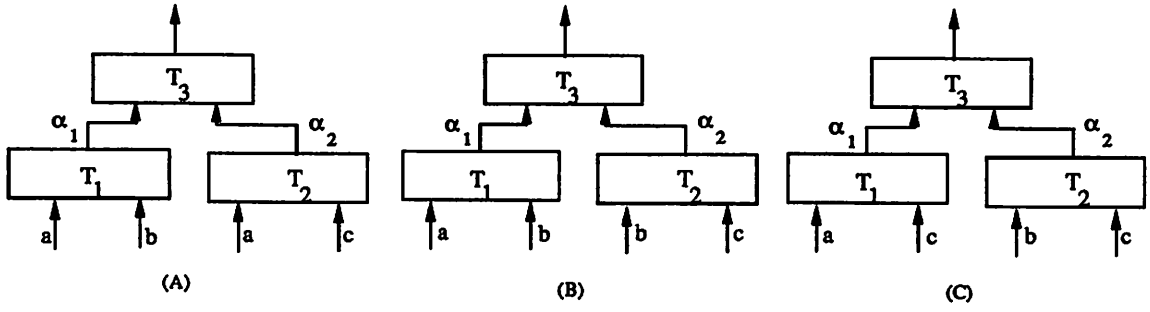


Figure 5.19: Possible cases for Configuration 2

$\frac{ac}{\alpha_1}$	00	01	10	11
0	0	1	0	1
1	1	0	1	1

Figure 5.20: Checking if f can be realized by configuration 2 (A)

$$g(\alpha_1, a, c) = c'\alpha_1 + c\alpha_1' + a\alpha_1.$$

Figure 5.19 (A) corresponds to a decomposition of g of the form

$$g(\alpha_1, a, c) = h(\alpha_2(a, c), \alpha_1)$$

The corresponding decomposition chart for g is shown in Figure 5.20. 3 distinct column patterns indicate that this decomposition is not possible. A contradiction! So f cannot be realized with three 2-LUTs using configuration 2.

Hence, f cannot be realized with three 2-LUTs.

This completes the proof of Lemma 5.3.11. ■

Comment: This lemma also shows that there exists a 3-input function that has a minimum (generalized) factored form of more than 4 literals.

Theorem 5.3.2 follows immediately from Lemmas 5.3.6 and 5.3.11.

Two Applications

One application is to use the theorem to improve the upper bound on $C_2(S(n))$ as given in Proposition 5.2.3.

Corollary 5.3.14

$$C_2(S(n)) \leq 2^{n-1} + 2^{n-2} + 2^{n-3} - 3 \tag{5.25}$$

Proof Use the recurrence inequality (5.4) for $m = 2$.

$$C_2(S(n)) \leq 2C_2(S(n-1)) + 3$$

While solving the recurrence, instead of terminating at a 2-input function with $C_2(S(2)) = 1$, use Theorem 5.3.2 to now terminate at a 3-input function with $C_2(S(3)) = 4$. This yields (5.25). ■

This bound is better than that of Proposition 5.2.3 (for $m = 2$) by 2^{n-3}. However, for large n, Lupanov's bound (Theorem 5.2.4) is better.

The second application is to find a lower bound on $C_3(f)$ in terms of $C_2(f)$.

Corollary 5.3.15

$$\frac{C_2(f)}{4} \leq C_3(f) \leq C_2(f)$$

Proof From Theorem 5.3.2, we know that $C_2(S(3)) = 4$. The result follows from Proposition 5.2.5 by substituting $A = C_2(S(3)) = 4, k = 2, m = 3$. ■

As will be shown in Corollary 5.3.22, the upper bound can be tightened for a special case.

5.3.3 Complexity Bound Given an SOP Representation

We are normally given a sum-of-products representation of the function. The following proposition derives a simple upper bound on the complexity of the function based on the numbers of literals and cubes in the SOP.

Proposition 5.3.16 *For a function f with $\ell > 1$ literals and c cubes in a sum-of-products representation,*

$$C_m(f) \leq \lfloor \frac{\ell - 1 + (c+1)(m-2)}{m-1} \rfloor. \tag{5.26}$$

Proof Realize f by realizing all cube functions and then ORing them together. If a cube i has ℓ_i literals, it can be realized with $\lceil \frac{\ell_i - 1}{m-1} \rceil$ m-LUTs (from Proposition 5.2.1). Then, all the cubes can be realized with $\sum_{i=1}^{c} \lceil \frac{\ell_i - 1}{m-1} \rceil$ m-LUTs. If a cube j has just one literal (i.e., $\ell_j = 1$), it is left as such, consistent with its 0 contribution to the sum. To OR these c cube functions together, additional $\lceil \frac{c-1}{m-1} \rceil$ m-LUTs are needed. This gives

$$\begin{aligned} C_m(f) &\leq \sum_{i=1}^{c} \lceil \frac{\ell_i - 1}{m-1} \rceil + \lceil \frac{c-1}{m-1} \rceil \\ &\leq \sum_{i=1}^{c} (\frac{\ell_i - 1}{m-1} + \frac{m-2}{m-1}) + \frac{c-1}{m-1} + \frac{m-2}{m-1} \\ &\leq \frac{\sum_{i=1}^{c} (\ell_i - 1)}{m-1} + (c+1)\frac{m-2}{m-1} + \frac{c-1}{m-1} \\ &\leq \frac{\ell - c}{m-1} + (c+1)\frac{m-2}{m-1} + \frac{c-1}{m-1} \\ &\leq \frac{\ell - 1 + (c+1)(m-2)}{m-1} \end{aligned}$$

Since $C_m(f)$ is an integer, we can take the *floor* of the right hand side, and obtain the desired result. ■

Note that the bound also holds for $\ell = 1$ when $m > 2$. However, it fails for $m = 2$ and $\ell = 1$. This bound uses one term. A slight improvement is possible if two terms are used.

Proposition 5.3.17 *For a function f with $\ell > 1$ literals and c cubes in a sum-of-products representation,*

$$C_m(f) \leq \lfloor \frac{\ell + c(m-3)}{m-1} \rfloor + \lceil \frac{c-1}{m-1} \rceil \tag{5.27}$$

Proof In the last proof, do not touch $\lceil \frac{c-1}{m-1} \rceil$; just manipulate the first term. ■

This bound does not hold for $\ell = 1$ - when the function is a simple inversion.

5.3.4 Complexity Bound Given a Factored Form

Since a factored form representation is generally smaller than a sum-of-products representation and is more closely related to the area of the circuit implemented (e.g., in a standard-cell methodology), it is natural to ask if we can find an upper bound on the complexity in terms of literals in the factored form.[4] For $m = 2$, it is easy. For an ℓ-literal factored form, $(\ell - 1)$ 2-LUTs suffice (Proposition 5.3.10). So we focus on $m > 2$. Our main result is the following:

[4]Recall that we are using a generalized notion of a factored form, in which the binary operations are not restricted to just AND and OR. All ten binary operations including NAND, NOR, EX-OR, EX-NOR, etc., are allowed. This allows potentially smaller factored forms. However, if only AND and OR are allowed, all the results presented here continue to hold.

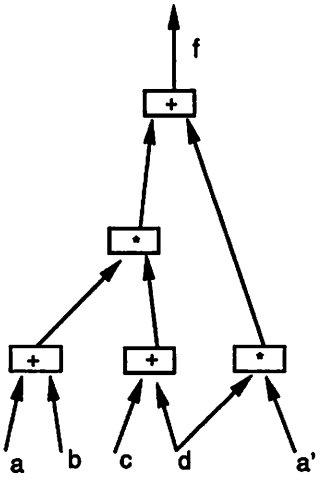


Figure 5.21: Converting a factored form to a 2-feasible leaf-DAG $\mathcal{R}$

Theorem 5.3.18 *For a function f with ℓ literals in a factored form,*

$$C_3(f) \leq \lfloor \frac{2\ell - 1}{3} \rfloor (\text{for } \ell \geq 2)$$
$$C_4(f) \leq \lfloor \frac{\ell - 1}{2} \rfloor (\text{for } \ell \geq 3)$$
$$C_5(f) \leq \lfloor \frac{2\ell - 1}{5} \rfloor (\text{for } \ell \geq 4)$$
$$C_6(f) \leq \lfloor \frac{\ell - 1}{3} \rfloor (\text{for } \ell \geq 4)$$

This theorem states, for example, that *any* factored form having 10 literals can be realized by six 3-LUTs, or four 4-LUTs, or three 5-LUTs, or three 6-LUTs. *Note that computation of the bounds does not require the factored form to be known, but just the number of literals in it.*

The proof has three main steps:

1. Given an ℓ-literal factored form ($\ell > 1$), obtain a 2-feasible leaf-DAG $\mathcal{R}$ with $T = (\ell - 1)$ internal nodes. This derivation was described in Proposition 5.3.10. Note that an m-feasible leaf-DAG implementation of f is a leaf-DAG whose internal nodes are m-LUTs (and so have at most m children) realizing m-feasible functions. In addition, the root node realizes f.

Example 5.3.1 *Consider*

$$f = ((a + b)(c + d)) + (da') \tag{5.28}$$

The 2-feasible leaf-DAG corresponding to f is shown in Figure 5.21.

2. Convert $\mathcal{R}$ into an m-feasible (leaf-DAG) implementation $\tilde{\mathcal{R}}$. The basic strategy here is *to place the* LUT *nodes of* $\mathcal{R}$ *in rooted, disjoint, connected groups, such that each group is realizable by one* m-LUT. We restrict ourselves to disjoint groups, since for a leaf-DAG, there always exists an optimum grouping (cover) that is disjoint. The algorithm used to group the nodes is key in deriving good bounds.

3. Derive bounds on the size of $\tilde{\mathcal{R}}$ in terms of that of $\mathcal{R}$.

We assume that no internal node (LUT) of $\mathcal{R}$ has only one child (input). If there is such an LUT, it is either a buffer or an inverter, which can be suitably absorbed in the fanout (parent).

If $\mathcal{R}$ is a leaf-DAG or a tree, dynamic programming based exact algorithms that minimize the number of nodes in the resulting m-feasible implementation $\tilde{\mathcal{R}}$ are known [25, 41]. These algorithms, however, do not provide any bounds on the size of $\tilde{\mathcal{R}}$. Therefore different techniques have to be resorted to. First, we solve for $m = 3, 4, 5$, and 6. Then, we present a technique for general m. Since step 1 will be identical for all m, in the following arguments, we start from the second step with a 2-feasible leaf-DAG $\mathcal{R}$ generated from the first step. In the following, r will refer to the root of the leaf-DAG $\mathcal{R}$, T ($\tilde{T}$) to the number of LUTs (internal nodes) in the leaf-DAG $\mathcal{R}$ ($\tilde{\mathcal{R}}$), and $\Delta = T - \tilde{T}$, to the reduction in the LUT count after converting $\mathcal{R}$ to $\tilde{\mathcal{R}}$. Also, unless stated otherwise, "visiting or grouping a node of a DAG" will mean visiting or grouping an LUT or an internal node.

2-feasibility to 3-feasibility

A 2-feasible leaf-DAG $\mathcal{R}$ is converted into a 3-feasible leaf-DAG $\tilde{\mathcal{R}}$ by forming parent-child pairs. Each such pair has no more than 3 inputs and hence can be realized with a 3-LUT. First, we examine a simple top-down traversal, which pairs nodes starting from the root of $\mathcal{R}$. Then, we show how to obtain an improved bound by traversing the DAG bottom-up.

Proposition 5.3.19 *Given a 2-feasible leaf-*DAG *realization* $\mathcal{R}$ *of a function* f *with* T *internal nodes, it is possible to obtain a 3-feasible leaf-*DAG *realization* $\tilde{\mathcal{R}}$ *of* f *that has at most* $(3T + 1)/4$ *internal nodes.*

Proof We give a simple algorithm to pair the LUTs of $\mathcal{R}$. Traverse $\mathcal{R}$ topologically from top (root) to bottom (leaf LUTs), i.e., visit a parent before any of its children. If an unpaired non-leaf LUT vertex is encountered, pair it with one of its LUT children; ties are broken by pairing the vertex with

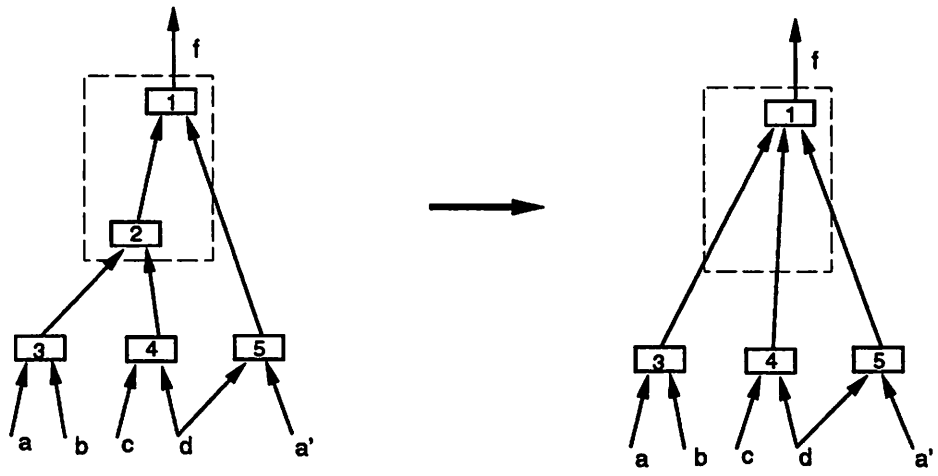


Figure 5.22: Converting $\mathcal{R}$ into $\tilde{\mathcal{R}}$ using the top-down pairing algorithm

its left child. The working of the algorithm is illustrated for the leaf-DAG of Figure 5.21 in Figure 5.22. The number inside an LUT denotes the order in which the LUT is visited; so LUT 1 is visited first and LUT 5 last. LUT 1 is paired with 2, and then no more pairings are possible. This gives a reduction of one LUT in the resulting 3-feasible DAG, i.e., $\Delta = 1$.

Analysis of the algorithm: It is easy to see that after the algorithm is applied, all non-leaf LUTs get paired. Since each pair can be implemented with one 3-LUT, a 3-feasible implementation $\tilde{\mathcal{R}}$ is obtained after collapsing the pairs appropriately.

Let L be the number of leaf LUTs in $\mathcal{R}$. The number of non-leaf LUTs is then $T - L$. Since all the non-leaf LUTs are paired, the number of pairs (i.e., Δ) is at least $(T - L)/2$. Since the number of non-leaf nodes in a 2-feasible leaf-DAG is at least $L - 1$, $L \leq (T + 1)/2$. So

$$\begin{aligned}
\tilde{T} &= T - \Delta \\
&\leq T - (T - L)/2 \\
&= (T + L)/2 \\
&\leq (T + (T + 1)/2)/2 \\
&= (3T + 1)/4
\end{aligned}$$

Since $\tilde{T}$ is integral, $\tilde{T} \leq \lfloor (3T + 1)/4 \rfloor$. If T is even, $\tilde{T} \leq \lfloor (3T)/4 \rfloor$. ■

This bound is tight *given that we use the top-down pairing algorithm described above.* Tightness means that there is a leaf-DAG on which the algorithm cannot do any better than the bound. Consider the leaf-DAG $\mathcal{R}$ of Figure 5.23. $\mathcal{R}$ contains p copies of the sub-structure $\mathcal{S}$. So

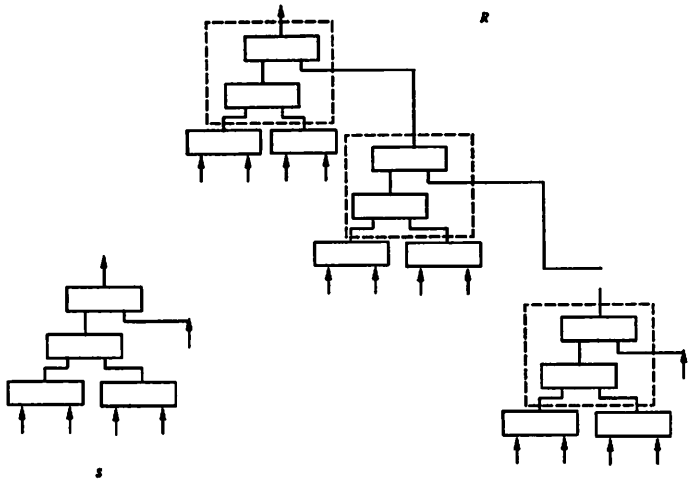


Figure 5.23: Tightness of bound *given* the top-down pairing algorithm

$T = 4p$. On applying the pairing algorithm, the root node of $\mathcal{S}$ gets paired with its left child (as shown). None of the leaves of $\mathcal{S}$ can be unpaired. This holds for all the p copies. So all $2p$ leaves of $\mathcal{R}$ remain unpaired. The number of pairs is p, and hence $\tilde{T} = 3p = 3T/4$. This argument, however, does not discount the possibility of another strategy resulting in an improved bound. In fact, as we show next, a bottom-up traversal of $\mathcal{R}$ yields a saving of at least $(T-1)/3$.

Proposition 5.3.20 *Given a 2-feasible leaf-*DAG *realization $\mathcal{R}$ of f with T internal nodes, it is possible to obtain a 3-feasible leaf-*DAG *realization $\tilde{\mathcal{R}}$ of f that has at most $(2T+1)/3$ internal nodes.*

Proof Traverse $\mathcal{R}$ bottom-up, i.e., visit an LUT before its parent. Initially, all LUT nodes are ungrouped. When visiting a vertex v, the following possibilities arise:

1. v is already grouped: do nothing.
2. v is ungrouped: if $v = r$, do nothing. Otherwise, let the parent of v be w. If w is ungrouped, group v with w, as shown in Figure 5.24 (A). If w is grouped already (the only way that can happen is that w is paired with its other child x (Figure 5.24 (B)), do nothing.

Let us apply this algorithm on the leaf-DAG of Figure 5.21. The pairings are shown in Figure 5.25. Note that $T = 5$. LUT 1 is visited first and the root r, the last. Two pairs are formed - first $\{1, 3\}$ and then $\{4, 5\}$. This means that for each such pair, $\tilde{\mathcal{R}}$ will have one LUT node. So $\tilde{T} = 3$. Note that this is better than what is obtained by a top-down traversal.

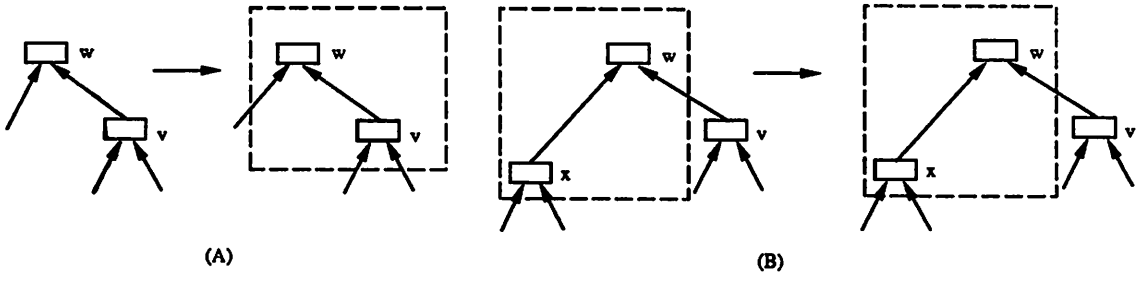


Figure 5.24: Converting a 2-feasible leaf-DAG into a 3-feasible leaf-DAG

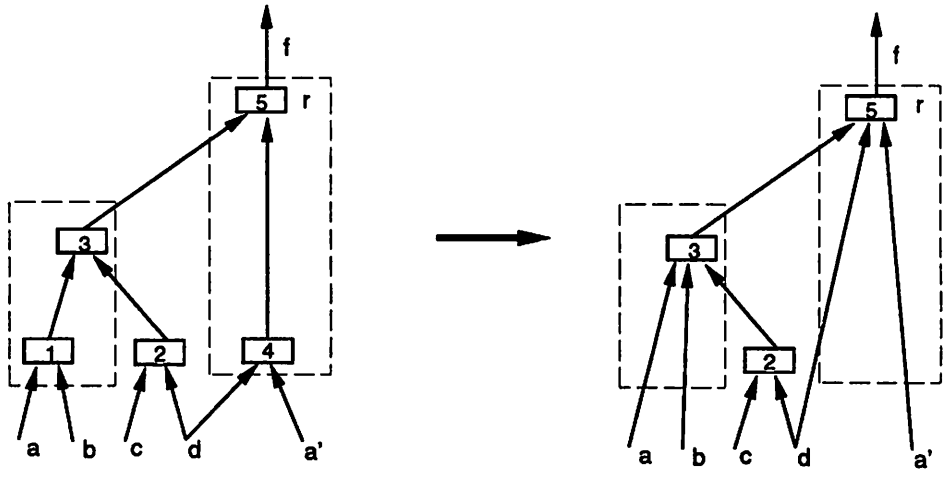


Figure 5.25: Example: converting a 2-feasible leaf-DAG into a 3-feasible leaf-DAG

Analysis of the algorithm: Assume that k nodes remain unpaired after the algorithm. The number of paired nodes is then $(T-k)$, so Δ is $(T-k)/2$. At least $(k-1)$ unpaired nodes are non-root nodes and have parents. Consider such a node v. The only reason it was not paired was that at the time it was visited, its parent w was already paired (this pair has root w). We call this pair *problem pair(v)*. For instance, in Figure 5.25, problem pair(2) = $\{1, 3\}$. It is easy to see that if u and v are two distinct, non-root, unpaired nodes, their problem pairs are disjoint. So there are at least $2(k-1)$ paired nodes.

$$\begin{aligned}
(T-k) &\geq 2(k-1) \\
\Rightarrow k &\leq (T+2)/3 \\
\Delta &= (T-k)/2 \\
&\geq (T-(T+2)/3)/2
\end{aligned}$$

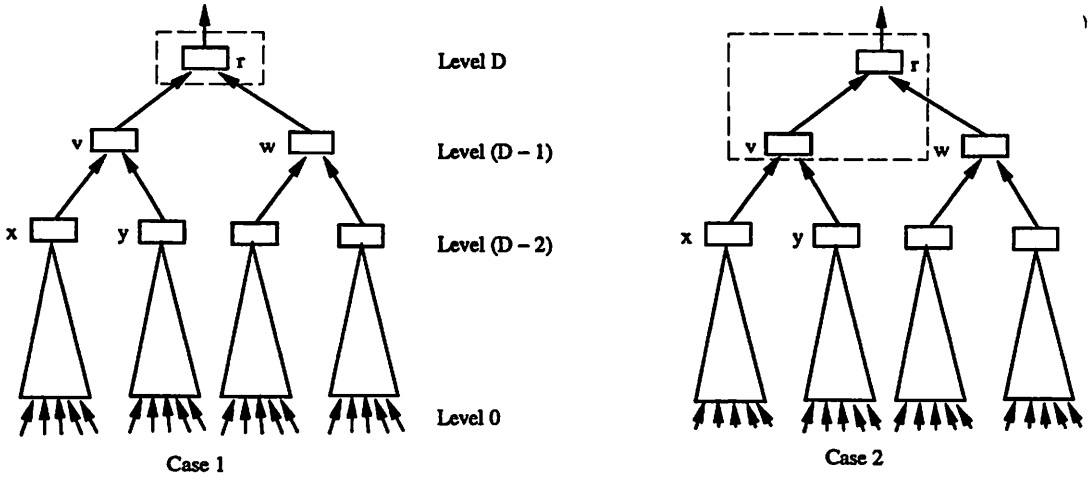


Figure 5.26: Covering a balanced tree

$$= (T-1)/3$$

Therefore, $\tilde{T} = T - \Delta \leq (2T+1)/3$. ■

As the following proposition shows, this bound is tight, i.e., there exists a leaf-DAG on which no more than one-third size-reduction is possible, allowing any pairings.

Proposition 5.3.21 *Let $\mathcal{R}$ be a balanced 2-feasible tree with T internal nodes, and $\tilde{\mathcal{R}}$ a 3-feasible tree with $\tilde{T}$ internal nodes obtained by covering $\mathcal{R}$ optimally. Then, $\Delta = T - \tilde{T} \leq \lceil (T-1)/3 \rceil$.*

Proof Let $\mathcal{R}$ have D levels. Then $T = 2^D - 1$. Let $\Delta_{\mathcal{R}} = T - \tilde{T}$. We show by induction on D that $\Delta_{\mathcal{R}} \leq \lceil \frac{2^D-2}{3} \rceil$.

Basis: Since $D = 1$ is trivially satisfied, consider $D = 2$ as the basis. By collapsing either of the two leaf nodes into the root r, an optimum $\tilde{\mathcal{R}}$ is obtained. Then, $\Delta_{\mathcal{R}} = 1 \leq \lceil (3-1)/3 \rceil$

Induction hypothesis: For all balanced trees $\mathcal{R}$ with depth $d < D$, the proposition is true.

Induction step: Consider a balanced tree $\mathcal{R}$ with $D > 2$ levels. Let r be the root of $\mathcal{R}$, v and w, the two children of r, and $\mathcal{V}$ and $\mathcal{W}$, the sub-trees rooted at v and w respectively. $\mathcal{V}$ and $\mathcal{W}$ have $(D-1)$ levels each. The root r can be covered in one of the following two ways (Figure 5.26):

1. *r remains a singleton*: Then the reduction is obtained only in $\mathcal{V}$ and $\mathcal{W}$.

$$\begin{aligned} \text{Total reduction} &= \Delta_{\mathcal{V}} + \Delta_{\mathcal{W}} \\ &\leq 2\lceil \frac{(2^{D-1}-1)-1}{3} \rceil \text{[Induction hypothesis]} \\ &= 2\lceil \frac{2^{D-1}-2}{3} \rceil \end{aligned}$$

There are two sub-cases: either $2^{D-1}-2$ is a multiple of 3, or it is of the form $(3k-1)$.[5] In the first case,

$$\begin{aligned}\text{Total reduction} &\leq \frac{2^D-4}{3} \\ &= \lceil\frac{2^D-4}{3}\rceil[2^D-4 \text{ is a multiple of 3}] \\ &< \lceil\frac{2^D-2}{3}\rceil\end{aligned}$$

In the second case, $2^{D-1}-2$ is of the form $(3k-1)$. Then,

$$\begin{aligned}\text{Total reduction} &\leq 2\lceil\frac{2^{D-1}-2}{3}\rceil \\ &= 2(\frac{2^{D-1}-2}{3}+\frac{1}{3}) \\ &= \frac{2^D-4}{3}+\frac{2}{3} \\ &= \frac{2^D-2}{3} \\ &= \lceil\frac{2^D-2}{3}\rceil[2^D-2 \text{ is a multiple of 3}]\end{aligned}$$

2. *r is paired with v* (the other case when r is paired with w is symmetric): Let x and y be the two children of v, and $\mathcal{X}$ and $\mathcal{Y}$, the trees rooted at them respectively. $\mathcal{X}$ and $\mathcal{Y}$ have $D-2(\geq 1)$ levels. Then, the total reduction is one more than the sum of the maximum reductions in $\mathcal{X}, \mathcal{Y}$ and $\mathcal{W}$ (the additional reduction of one being obtained from pairing r and v):

$$\begin{aligned}\text{Total reduction} &= \Delta_{\mathcal{X}}+\Delta_{\mathcal{Y}}+\Delta_{\mathcal{W}}+1 \\ &\leq 2\lceil\frac{2^{D-2}-2}{3}\rceil+\lceil\frac{2^{D-1}-2}{3}\rceil+1[\text{Induction hypothesis}]\end{aligned}$$

Once again, consider two sub-cases. When $(2^{D-2}-2)$ is a multiple of 3, we get

$$\begin{aligned}\text{Total reduction} &\leq 2\lceil\frac{2^{D-2}-2}{3}\rceil+\lceil\frac{2^{D-1}-2}{3}\rceil+1 \\ &= 2(\frac{2^{D-2}-2}{3})+\frac{2^{D-1}-2}{3}+\frac{1}{3}+1[2^{D-1}-2 \text{ is of the form } (3k+2)]\end{aligned}$$

[5] It is easy to see that it is never of the form $(3k-2)$.

$$
\begin{aligned}
&= \frac{2^{D-1}-4}{3} + \frac{2^{D-1}+2}{3} \\
&= \frac{2^D-2}{3} \\
&= \lceil \frac{2^D-2}{3} \rceil [2^D - 2 \text{ is a multiple of 3}]
\end{aligned}
$$

When $(2^{D-2}-2)$ is of the form $(3k-1)$, we get

$$
\begin{aligned}
\text{Total reduction} &\leq 2\lceil \frac{2^{D-2}-2}{3} \rceil + \lceil \frac{2^{D-1}-2}{3} \rceil + 1 \\
&= 2(\frac{2^{D-2}-2}{3} + \frac{1}{3}) + \frac{2^{D-1}-2}{3} + 1[2^{D-1}-2 \text{ is a multiple of 3}] \\
&= \frac{2^{D-1}-2}{3} + \frac{2^{D-1}-2}{3} + 1 \\
&= \frac{2^D-1}{3} \\
&= \lceil \frac{2^D-2}{3} \rceil [2^D - 1 \text{ is a multiple of 3}]
\end{aligned}
$$

$\Delta_{\mathcal{R}}$ is the maximum of the total reductions obtained in all the cases. Hence $\Delta_{\mathcal{R}} \leq \lceil \frac{2^D-2}{3} \rceil$. This completes the induction step. ■

In fact, for a balanced tree, the reduction $\lceil \frac{T-1}{3} \rceil$ is achievable. This should be clear from Propositions 5.3.20 and 5.3.21. An alternative is to use the dynamic programming argument of the proof of Proposition 5.3.21 itself.

We use Proposition 5.3.20 to tighten the upper bound of Corollary 5.3.15 for a special case.

Corollary 5.3.22 *If there exists an optimum 2-feasible implementation of f that is a leaf-*DAG*, then*

$$\frac{C_2(f)}{4} \leq C_3(f) \leq \frac{2}{3}C_2(f) + \frac{1}{3} \tag{5.29}$$

Proof Immediate from Corollary 5.3.15 and Proposition 5.3.20. ■

2-feasibility to 4-feasibility

Proposition 5.3.23 *Given a 2-feasible leaf-*DAG *realization $\mathcal{R}$ of f with $T \geq 2$ internal nodes, it is possible to obtain a 4-feasible leaf-*DAG *realization $\tilde{\mathcal{R}}$ of f that has at most $T/2$ internal nodes.*

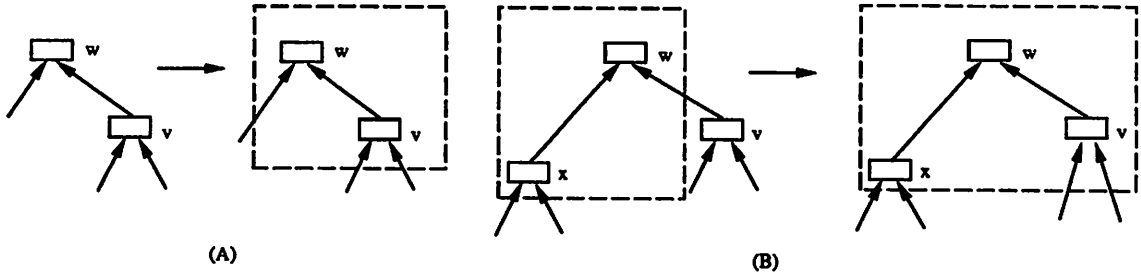


Figure 5.27: Converting a 2-feasible leaf-DAG into a 4-feasible leaf-DAG

Proof At most 3 nodes are allowed in a group. Once again, traverse $\mathcal{R}$ bottom-up. Initially, all LUT nodes are ungrouped. When visiting a vertex v, the following cases arise:

1. v is already grouped: do nothing.
2. v is ungrouped: there are two sub-cases:
   i. $v = r$: if some child of v is involved in a pair, merge v in this pair to get a triple. Otherwise, do nothing.
   ii. $v \neq r$: group v with its parent w. If w was initially ungrouped, the grouping looks as in Figure 5.27 (A). If w was grouped already, the only way that can happen is that w is paired with its other child x (Figure 5.27 (B)). Place v in the group.

Analysis of the algorithm: After all the nodes have been visited, all LUT nodes of $\mathcal{R}$, except possibly r, are grouped. Let there be p triples (i.e., groups with three nodes, as in Figure 5.27 (B), the group with x, v, and w) and q doubles (or pairs). Each triple gives a saving of two LUTs, and double, a saving of one LUT. There are two possibilities:

1. The root r is grouped: then

$$\begin{aligned} T &= 3p + 2q \\ \Delta &= 2p + q \\ &\geq 1.5p + q \\ &= T/2 \end{aligned}$$

2. The root r is ungrouped: then

$$\begin{aligned} T &= 3p + 2q + 1 \\ \Delta &= 2p + q \end{aligned}$$

If $T = 2$, clearly $\Delta = 1 = T/2$. For $T > 2$, some child x of r must be involved in a triple, otherwise r would have been grouped with x in the step 2 i). This implies $p \geq 1$.

$$\begin{aligned} \Delta &= 2p + q \\ &\geq (1.5p + 1/2) + q \\ &= T/2 \end{aligned}$$

Hence, the 4-feasible $\tilde{\mathcal{R}}$ has at most $T/2$ nodes. ■

Next, we prove that this bound is tight.

Proposition 5.3.24 *Let $\mathcal{R}$ be the 2-feasible tree shown in Figure 5.28 with $T = 4p$, and $\tilde{\mathcal{R}}$ be the 4-feasible tree with $\tilde{T}$ LUT nodes obtained by covering $\mathcal{R}$ optimally. Then $\Delta = T - \tilde{T} \leq 2p = T/2$.*

Proof $\mathcal{R}$ has $2p$ copies of the basic sub-structure $\mathcal{S}$. Assign labels to the nodes of $\mathcal{R}$ as in Figure 5.28. Consider all possible types of groups (or matches) rooted at a given node v. If v is an even-numbered node, the only possibility is that the group is a singleton, and, hence, no savings are obtained. Let v be an odd-numbered node, say 1. In all, six matches are possible, as shown in Figure 5.28. It is easy to see that for each match, we cannot obtain a reduction of more than 50% on the portion *affected* by the match. For example, if the match 5 is selected, node 4 has to remain a singleton, and this eliminates two out of the four affected nodes (i.e., nodes 2 and 3 out of 1, 2, 3 and 4). Applying this argument to an optimum (disjoint) cover of $\mathcal{R}$ by these six types of matches, we get the desired result. ■

2-feasibility to 5-feasibility

Using similar but more complicated arguments, we show that a 60% reduction in T can result when we go from a 2-feasible implementation to a 5-feasible implementation.

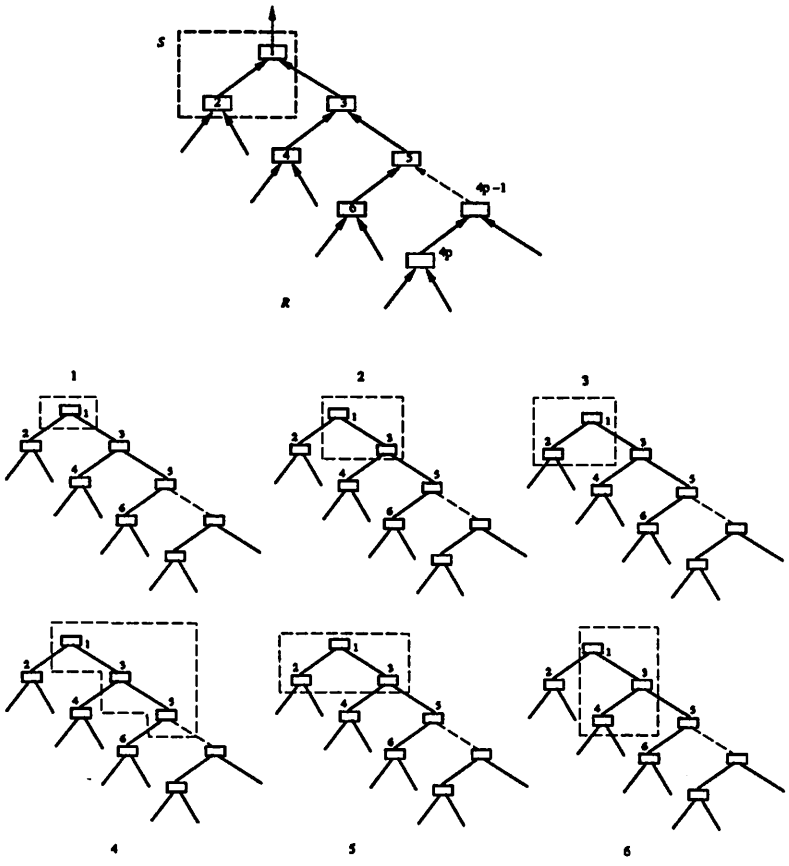


Figure 5.28: 2-feasibility to 4-feasibility: proving tightness

Proposition 5.3.25 *Given a 2-feasible leaf-*DAG *realization $\mathcal{R}$ of f with T internal nodes, $T \geq 3$, it is possible to obtain a 5-feasible leaf-*DAG *realization $\widetilde{\mathcal{R}}$ of f that has at most $(2T+1)/5$ internal nodes.*

Proof We put at most 4 nodes in a group. As in the earlier proofs, we visit the LUT nodes of $\mathcal{R}$ bottom-up and consider them for covering. Initially, all the LUT nodes are ungrouped. When visiting a node v, there are two cases:

1. v is ungrouped: consider the three sub-cases:

   i. v is a leaf-node (i.e., both its children are inputs): if $v \neq r$, group v with its parent p. If p is already in a group, put v in the same group. As we shall show in Claim 2, this grouping with the parent results in either a double or a triple. Note that $v = r$ cannot happen, since $T \geq 3$.

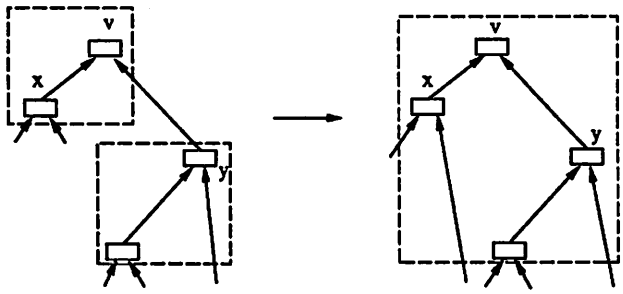


Figure 5.29: Case v already grouped: v is the root of a double and so is y

ii. v is a non-leaf node, and its one child, x, is in a double: group v with x to get a triple.

iii. v is a non-leaf node, and none of its LUT children is in a double: if $v \neq r$, group v with its parent. This generates either a double or a triple. Otherwise, do nothing.

2. v is already grouped: then v is root of either a triple, or a double. It cannot be part of a quartet, as will be shown in Claim 1 momentarily.

   i. v is the root of a triple: do nothing.

   ii. v is the root of a double: let x be the child of v with which v forms a double. If the other child y of v is an LUT node and is in a double, merge the two doubles to form a quartet as shown in Figure 5.29. Otherwise, do nothing.

Analysis of the algorithm: We first make a few claims:

Claim 1: If a vertex w was already in a group G when it was visited, G can only contain w and some of its children.

Proof: w can be grouped before being visited only in cases 1 i) and 1 iii), and, hence, only with its children (when they were singletons). In any case, either a double or a triple is generated, and w is the root of the group in either case.

Claim 2: While v is being visited in case 1, whenever v is grouped with its parent p (i.e., sub-cases i and iii), either a double or a triple is generated.

Proof: If p is ungrouped at the time of visiting v, a double containing v and p is generated. The case that p is already grouped can happen only when the other child of p, say w, was single at the time w was visited and had to be grouped with p. When v is added into this double, a triple containing v, w and p is created.

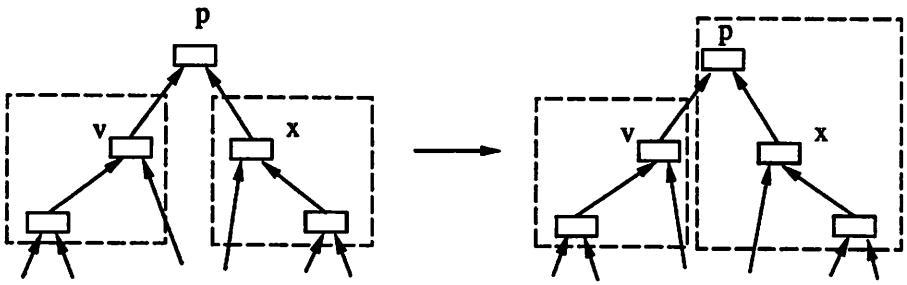


Figure 5.30: p gets merged with a double to form a triple

Claim 3: After vertex v is visited ($v \neq r$), it is grouped.

Proof: If v is not already grouped, the case 1 of the algorithm will group it.

It follows that after r has been visited, all the LUT nodes of $\mathcal{R}$, except possibly r, are grouped.

Claim 4: After r has been visited, for each double (except the one containing r (if it exists)), we can associate a unique triple.

Proof: Let v be the root of such a double. By assumption, $v \neq r$; so let p be the parent of v. Consider the two possible cases:

a. p was ungrouped when visited: Since one of the children v of p is in a double, case 1 tells us that this can happen only when there is another child x of p that was also in a double at that time, but on visiting p, p merged in this double generating a triple (shown in Figure 5.30). Since no triple is modified by the algorithm, this triple will be in the final cover, and the one we will associate with the double rooted at v.

b. p was grouped before it was visited: by Claim 1, at the time of visiting, p can be either in a double or in a triple. If p were in a double (which is necessarily different from that of v), then by case 2 ii), the two doubles (rooted at p and v) will be merged to result in a quartet. This is not possible, since the double at v exists in the final cover. Also, p can be involved in a triple only with its two children (Claim 1). Since v is in a double with its child, this case is not possible either.

We have thus associated a triple for each double (that does not involve r). Since this triple is rooted at the parent of the double's root, it is unique. This proves Claim 4.

We now compute Δ. Let there be p triples, q doubles and s quartets in the final cover of $\mathcal{R}$. It is convenient to analyze the following cases:

a. r is in a double: then from Claim 3, and the disjointedness of the groups, we get $T = 4s + 3p + 2q$. Also, from Claim 4, $p \geq q - 1$. Since a quartet gives a saving of 3, a triple of 2, and a double of 1, we get

$$\begin{aligned}
\Delta &= 3s + 2p + q \\
&= \frac{3}{5}[5s + 10p/3 + 5q/3] \\
&= \frac{3}{5}[5s + 3p + p/3 + 5q/3] \\
&\geq \frac{3}{5}[5s + 3p + 2q - 1/3] (\text{since } p \geq q - 1) \\
&= \frac{3}{5}[4s + 3p + 2q + s] \\
&= \frac{3}{5}[T + s - 1/3] \\
&\geq \frac{3}{5}T - \frac{1}{5}
\end{aligned}$$

b. r is in a triple, or in a quartet: then, $T = 4s + 3p + 2q$, $p \geq q$. Then

$$\begin{aligned}
\Delta &= 3s + 2p + q \\
&= \frac{3}{5}[5s + 10p/3 + 5q/3] \\
&= \frac{3}{5}[5s + 3p + p/3 + 5q/3] \\
&\geq \frac{3}{5}[5s + 3p + 2q] (\text{since } p \geq q) \\
&= \frac{3}{5}[4s + 3p + 2q + s] \\
&= \frac{3}{5}[T + s] \\
&\geq \frac{3}{5}T
\end{aligned}$$

c. r is a singleton. Then $T = 4s + 3p + 2q + 1$, and $p \geq q$. We analyze all possible cases:

i. Some child of r is in a quartet: Then $s \geq 1$.

$$\begin{aligned}
\Delta &= 3s + 2p + q \\
&= \frac{3}{5}[5s + 10p/3 + 5q/3] \\
&= \frac{3}{5}[5s + 3p + p/3 + 5q/3] \\
&\geq \frac{3}{5}[5s + 3p + 2q] (\text{since } p \geq q)
\end{aligned}$$

$$= \frac{3}{5}[4s + 3p + 2q + s]$$
$$\geq \frac{3}{5}[4s + 3p + 2q + 1](\text{since } s \geq 1)$$
$$= \frac{3}{5}T$$

ii. No child of r is in a quartet, but a child, x, is in a triple: then, as a post-processing step, we can merge r in this triple to get a quartet. Since we are reducing the number of triples in the final cover, $p \geq q - 1$, and $s \geq 1$ where p, q and s are the values obtained after post-processing. Also, $T = 4s + 3p + 2q$. This case then is same as a.).

iii. No child of r is in a quartet, or a triple: then r's child(ren) is (are) part of a double. But then, 1 i) of the algorithm would have combined r in this double. So this case is not possible.

Hence the number of nodes in the resulting 5-feasible DAG is at most $\frac{2}{5}T + \frac{1}{5}$. ■

We can also get a 5-feasible implementation by converting first $\mathcal{R}$ to a 3-feasible leaf-DAG $\tilde{\mathcal{R}}'$, and then $\tilde{\mathcal{R}}'$ to a 5-feasible leaf-DAG $\tilde{\mathcal{R}}$. Then, the size of $\tilde{\mathcal{R}}$ is bounded above by 1/2 the size of $\mathcal{R}$ - a weaker bound as compared to that of Proposition 5.3.25.

We now show that like previous propositions, this bound is also tight.

Proposition 5.3.26 *Let $\mathcal{R}$ be the 2-feasible tree shown in Figure 5.31 with $T = 5p$. Let $\tilde{\mathcal{R}}$ be a 5-feasible tree with $\tilde{T}$ internal nodes obtained by covering $\mathcal{R}$ optimally. Then $\Delta = T - \tilde{T} \leq 3p = (3T)/5$.*

Proof There are p copies of $\mathcal{S}$ in $\mathcal{R}$. We assign a label k to each - the copy with the nodes $\{1,2,3,4,5\}$ is assigned $k = 1$, and the root copy is assigned $k = p$. Let $\mathcal{R}(i)$ be the sub-tree rooted at node i, and $\Delta_{\mathcal{R}(i)}$ the optimum reduction in the size for $\mathcal{R}(i)$ when it is converted to a 5-feasible $\tilde{\mathcal{R}}(i)$. There are 7 possible patterns for converting $\mathcal{R}$ into $\tilde{\mathcal{R}}$, as shown in Figure 5.32. In this figure, no distinction is made between the left and the right children. For instance, the pattern 2 does not imply that the *left* child of the root of the pattern is an LUT, just that *one* child of the root is an LUT. Each of these patterns can be implemented as a 5-LUT.

Define $Q(k)$: $\Delta_{\mathcal{R}(5k-4)} \leq 3k - 2, \Delta_{\mathcal{R}(5k-2)} \leq 3k$.

It suffices to prove $Q(p)$, since $Q(p)$ implies $\Delta_{\mathcal{R}(5p-2)} \leq 3p$. We prove $Q(k)$ by induction on k.

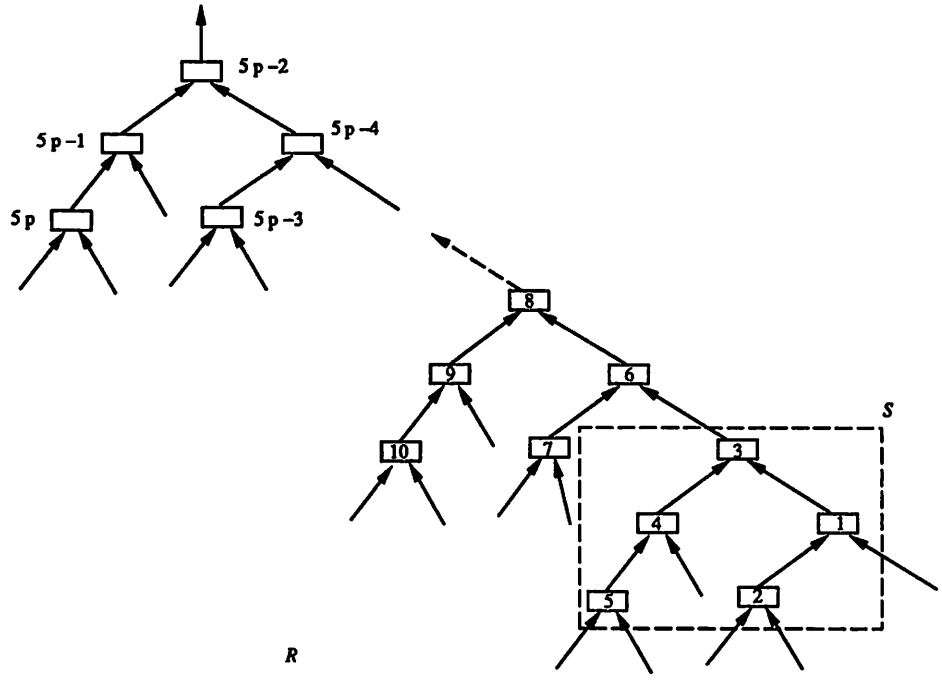


Figure 5.31: 2-feasibility to 5-feasibility: proving tightness

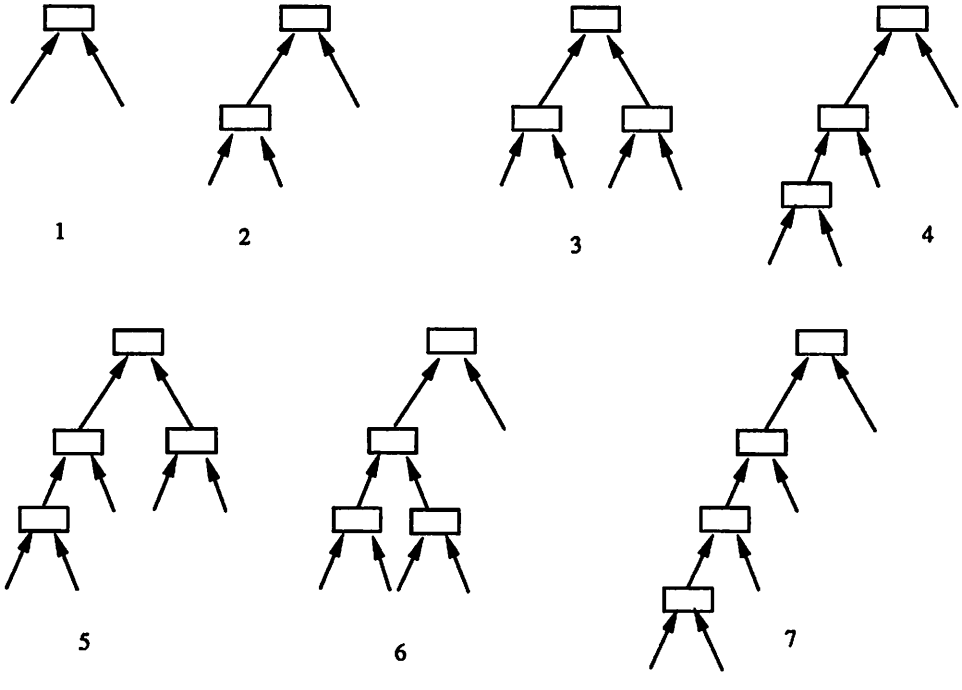


Figure 5.32: All possible patterns for getting a 5-feasible leaf-DAG

Basis $Q(1)$: It is easy to see that $\Delta_{\mathcal{R}(1)} \leq 1$. Also, since no match with 5 nodes (which results in a reduction by 4) is possible, a saving of at most 3 is possible (e.g., by covering nodes 1 and 2 with pattern 2, and 3, 4 and 5 with pattern 4). This proves that $\Delta_{\mathcal{R}(3)} \leq 3$.

Induction hypothesis: $Q(k')$ is true for all $k' \leq k$.

Induction step: We have to prove that $Q(k+1)$ is true. We prove $\Delta_{\mathcal{R}(5(k+1)-4)} \leq 3(k+1)-2$. The proof of $\Delta_{\mathcal{R}(5(k+1)-2)} \leq 3(k+1)$ is similar. Figure 5.33 shows all matches $i.j$ rooted at node $5(k+1)-4 = 5k+1$, where i is the corresponding pattern number from Figure 5.32, and j is an index for the match. For example, corresponding to the pattern 1, there is only one match, 1.1, whereas corresponding to the pattern 2, there are two matches, namely 2.1 and 2.2. Given a match $i.j$ rooted at $5k+1$, we compute the best possible savings for the inputs to the match, and then sum them up along with the saving due to the match to obtain the total saving $\Delta_{\mathcal{R}(5k+1)}(i.j)$ for the sub-tree $\mathcal{R}(5k+1)$. $\Delta_{\mathcal{R}(5k+1)}$ is the maximum over all i, j of $\Delta_{\mathcal{R}(5k+1)}(i.j)$. For each match, we also show the best way to cover the left children fanning in to the match, and are left children of some node in the match. The best cover for the right children is obtained from the induction hypothesis. This asymmetry arises because the subtree rooted at the left child of a node is simple, and its optimum cover can be obtained from inspection. From Figure 5.33 and the induction hypothesis, we have

$$
\begin{aligned}
\Delta_{\mathcal{R}(5k+1)}(1.1) &= 0+0+\Delta_{\mathcal{R}(5k-2)} \\
&\leq 3k \\
\Delta_{\mathcal{R}(5k+1)}(2.1) &= 1+\Delta_{\mathcal{R}(5k-2)} \\
&\leq 3k+1 \\
\Delta_{\mathcal{R}(5k+1)}(2.2) &= 1+0+1+\Delta_{\mathcal{R}(5k-4)} \\
&\leq 2+(3k-2) \\
&= 3k \\
\Delta_{\mathcal{R}(5k+1)}(3.1) &= 2+1+\Delta_{\mathcal{R}(5k-4)} \\
&\leq 3+(3k-2) \\
&= 3k+1 \\
\Delta_{\mathcal{R}(5k+1)}(4.1) &= 2+0+0+\Delta_{\mathcal{R}(5k-4)} \\
&\leq 2+(3k-2) \\
&= 3k
\end{aligned}
$$

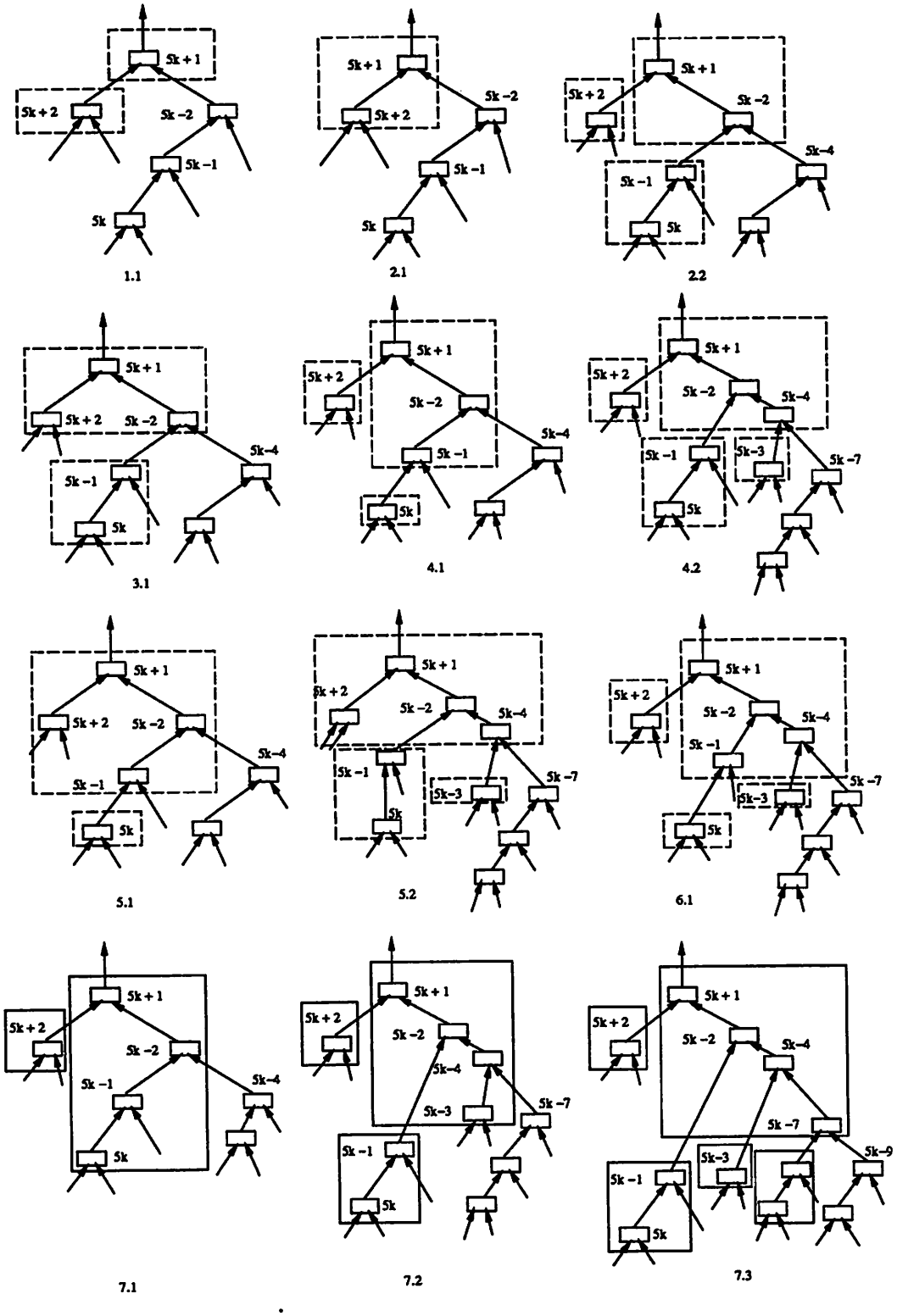


Figure 5.33: 2-feasibility to 5-feasibility: proving tightness

$$
\begin{aligned}
\Delta_{\mathcal{R}(5k+1)}(4.2) &= 2+0+1+0+\Delta_{\mathcal{R}(5k-7)} \\
&\leq 3+3(k-1) \\
&= 3k \\
\Delta_{\mathcal{R}(5k+1)}(5.1) &= 3+0+\Delta_{\mathcal{R}(5k-4)} \\
&\leq 3+(3k-2) \\
&= 3k+1 \\
\Delta_{\mathcal{R}(5k+1)}(5.2) &= 3+1+0+\Delta_{\mathcal{R}(5k-7)} \\
&\leq 4+3(k-1)) \\
&= 3k+1 \\
\Delta_{\mathcal{R}(5k+1)}(6.1) &= 3+0+0+0+\Delta_{\mathcal{R}(5k-7)} \\
&\leq 3+3(k-1)) \\
&= 3k \\
\Delta_{\mathcal{R}(5k+1)}(7.1) &= 3+0+\Delta_{\mathcal{R}(5k-4)} \\
&\leq 3+(3k-2) \\
&= 3k+1 \\
\Delta_{\mathcal{R}(5k+1)}(7.2) &= 3+1+0+\Delta_{\mathcal{R}(5k-7)} \\
&\leq 4+3(k-1)) \\
&= 3k+1 \\
\Delta_{\mathcal{R}(5k+1)}(7.3) &= 3+1+0+1+0+\Delta_{\mathcal{R}(5k-9)} \\
&\leq 5+(3(k-1)-2) \\
&= 3k
\end{aligned}
$$

Then

$$
\begin{aligned}
\Delta_{\mathcal{R}(5k+1)} &= \max_{i,j}\{\Delta_{\mathcal{R}(5k+1)}(i.j)\} \\
&\leq 3k+1 \\
&= 3(k+1)-2
\end{aligned}
$$

In Figure 5.34, the matches rooted at the node $5(k+1)-2 = 5k+3$ are shown. Using similar arguments, it can be proved that $\Delta_{\mathcal{R}(5k+3)} \leq 3(k+1)$ (in fact, the proof uses $\Delta_{\mathcal{R}(5k+1)} \leq 3k+1$, which we just proved). ■

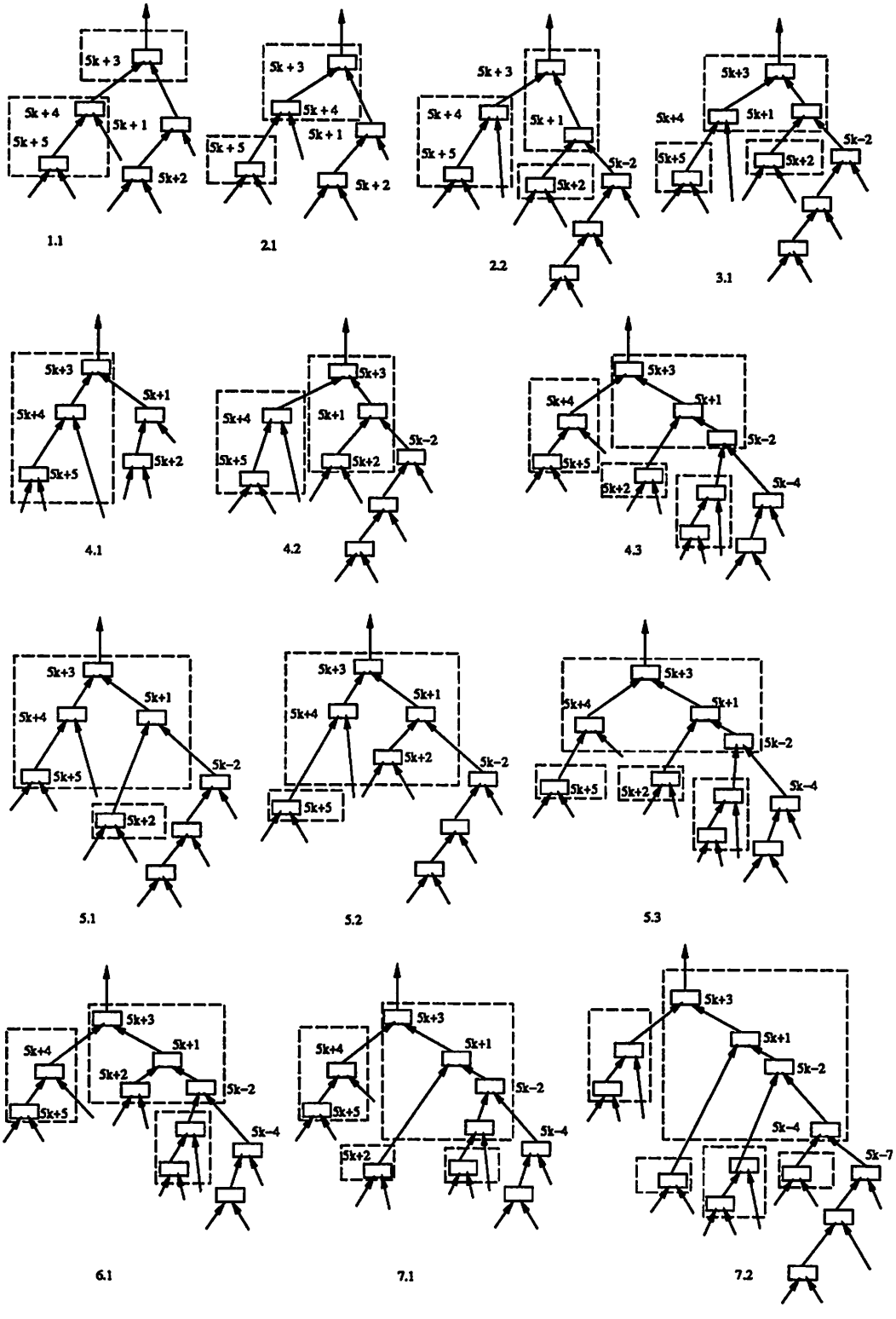


Figure 5.34: 2-feasibility to 5-feasibility: proving tightness

2-feasibility to 6-feasibility

Proposition 5.3.27 *Given a 2-feasible leaf-*DAG *realization $\mathcal{R}$ of f with T internal nodes, $T \geq 3$, it is possible to obtain a 6-feasible leaf-*DAG *realization $\tilde{\mathcal{R}}$ of f that has at most $T/3$ internal nodes.*

Proof We use the same algorithm that was used to convert a 2-feasible leaf-DAG to a 5-feasible one, except for a slight modification - we add a post-processing step at the end. It was shown in the proof of Claim 4 in Proposition 5.3.25 that if there is a double rooted at v ($v \neq r$), there exists a triple rooted at the parent of v. Since quintets are allowed while converting a 2-feasible DAG to a 6-feasible dag, we merge the corresponding doubles and triples to get quintets. Note that this merging is legal, in that it results in a connected group. As a result, there are only triples, quartets and quintets in the final grouping, except possibly for a singleton at r, or a double rooted at r. A triple is the least area-saver per node of $\mathcal{R}$ as compared to a quartet or a quintet: it gives a saving of 2 nodes for every 3. But that suffices our purpose. A more careful analysis (along with a simple post-processing step) of the cases when r is a singleton, or is in a double, shows that $\mathcal{R}'$ still has at most $T/3$ internal nodes. ■

Unlike for $3 \leq m \leq 5$, we do not know if the bound of Proposition 5.3.27 is tight. We conjecture that it is. The conjecture is based on the forms of the bounds: all are of the form $\frac{2}{m}T$.

A Unified Technique

While deriving the upper bounds on $\tilde{T}$, we used different algorithms for different values of m. Also, it becomes difficult to develop algorithms for $m > 6$. A natural question to ask is the following: *"Is it possible to develop a single generic technique that works for all m?"* In this subsection, we attempt to answer this question by presenting a uniform technique of converting a 2-feasible leaf-DAG $\mathcal{R}$ into an m-feasible leaf-DAG for any $m > 2$. This technique divides $\mathcal{R}$ into sets of levels and covers each set in a top-down fashion. Although it does not improve the bounds in the general case,[6] it does yield better bounds for leaf-DAGs with special structures.

First we define the *level* of an LUT in $\mathcal{R}$. The level of an input is 0. The level of an LUT is 1 + max{levels of its children}. Let k_i be the number of LUTs at level i. In particular, k_1 is the number of leaf-LUTs.

Theorem 5.3.28 *Given a 2-feasible leaf-*DAG *realization $\mathcal{R}$ of f with T internal nodes such that the number of leaf-*LUT*s, $k_1 \leq T/a$, it is possible to obtain an m-feasible leaf-*DAG *realization $\tilde{\mathcal{R}}$*

[6] We know from the tightness of the bounds that for $m = 3$, 4, and 5, the bounds cannot be improved.

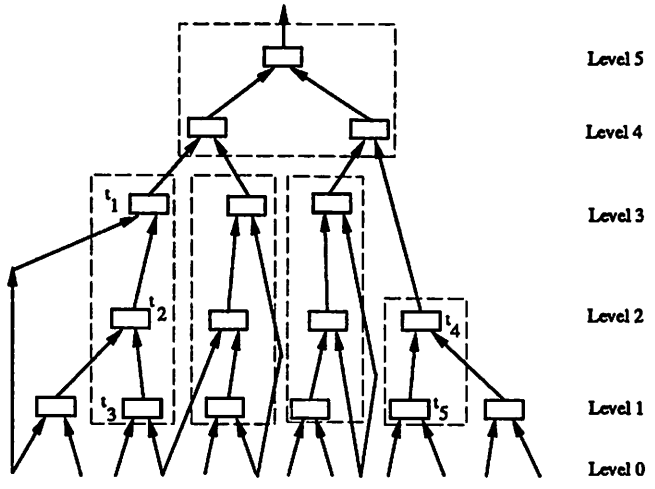


Figure 5.35: Converting a 2-feasible leaf-DAG to a 4-feasible leaf-DAG using leveling

of f with at most $[(a+m-2)/(am-a)]T$ internal nodes (plus a small constant).

Proof The argument is based on leveling $\mathcal{R}$, and is explained using an example with $m = 4$ in Figure 5.35. Grouping three LUTs t_1, t_2, and t_3 corresponds to a 4-LUT and saves two LUTs. A pair, say t_4 and t_5, is valid too and saves one LUT. The strategy then is to group an LUT t_1 at level $3i$ with an LUT t_2 at $(3i-1)$ and t_3 at $(3i-2)$, where t_2 is a child of t_1 and t_3 is a child of t_2. This group is a triple and so saves 2 LUTs. Repeating this process for all nodes at level $3i$, then varying i over the entire DAG ($i \geq 1$), and summing up gives total savings of $2(k_3 + k_6 + k_9 + \ldots)$. Note that this is possible since $\mathcal{R}$ is a leaf-DAG, and, therefore, $k_{j+1} \leq k_j$ for all $j \geq 1$. Next, we look at all the ungrouped nodes at level $(3i-1)$; there are $(k_{3i-1} - k_{3i})$ of them. We pair each such node with its child at level $(3i-2)$. Note that none of its children had been grouped. Varying i over $\mathcal{R}$ and summing up gives additional savings of $(k_2 - k_3) + (k_5 - k_6) + \ldots$. So

$$\begin{aligned} \Delta &= 2(k_3 + k_6 + \ldots) + (k_2 - k_3 + k_5 - k_6 + \ldots) \\ &= (k_2 + k_3) + (k_5 + k_6) + \ldots \end{aligned}$$

The case of general m is similar. For simplicity, we assume that the number of levels D in $\mathcal{R}$ is of the form $D = b(m-1) + 1$ (otherwise, we get an extra constant in the formula below). We separate $\mathcal{R}$ into sets of levels: 1 to $(m-1)$, m to $2(m-1)$, and so on. Finally, we group nodes within each set, as we did for $m = 4$. We get

$$\Delta = (k_2 + k_3 + \ldots + k_{m-1}) + (k_{m+1} + \ldots + k_{2(m-1)}) + \ldots$$

a	$m=3$	$m=4$	$m=5$
2	3/4	2/3	5/8
3	2/3	5/9	1/2
4	5/8	1/2	7/16
5	3/5	7/15	2/5

Table 5.1: $(a+m-2)/(am-a)$ as a function of a and m

$$\begin{aligned}
&\geq \frac{m-2}{m-1}[(k_2+k_3+\ldots+k_{m-1}+k_m)+(k_{m+1}+\ldots+k_{2(m-1)}+k_{2m-1})+\ldots] \\
&\qquad\qquad \text{(since } k_i \geq k_{i+1}\text{)} \\
&= \frac{m-2}{m-1}[T-k_1] \\
&\geq \frac{m-2}{m-1}[T-T/a]
\end{aligned}$$

Let the resulting m-feasible leaf-DAG $\tilde{\mathcal{R}}$ have $\tilde{T}$ LUTs. Then,

$$\begin{aligned}
\tilde{T} &= T-\Delta \\
&\leq T-\frac{m-2}{m-1}(T-T/a) \\
&= T\,\frac{a+m-2}{a(m-1)}
\end{aligned}$$

■

The fraction $(a+m-2)/(am-a)$ denotes the best bounds the unified technique can achieve. To compare it with the previous bounds, we evaluate this fraction for various values of a and m in Table 5.1. $a=2$ denotes the most general case, since $k_1 \leq (T+1)/2$ for any leaf-dag. The unified technique does not fare well for $a=2$ as compared to the previous bounds. For instance, it can never give a (worst-case) reduction (i.e., Δ) of more than 50% for any m, whereas the previous techniques could achieve this reduction for $m \geq 4$. However, for higher values of a, we can guarantee more reduction as compared to the previous bounds. For example, for $m=3$, $a>3$ gives better bounds than Proposition 5.3.20. Similarly, for $m=4$, $a>4$ gives better bounds than Proposition 5.3.23.

It is instructive to note that higher values of a correspond to the case when $\mathcal{R}$ assumes a more chain-like structure. In the limit, the bound of the theorem gives us the best possible size of $\tilde{\mathcal{R}}$, i.e., $T/(m-1)$. This size is possible when $\mathcal{R}$ is simply a chain.

Relating Factored Form and m-feasibility

We are now ready to prove the main result of this section, Theorem 5.3.18.

Proof (of Theorem 5.3.18) For $\ell > 1$, it is possible to construct a 2-feasible leaf-DAG for f with $T = (\ell - 1)$ 2-LUTs. Substituting $T = (\ell - 1)$ in Propositions 5.3.20, 5.3.23, 5.3.25, and 5.3.27, we get the desired result, since $C_m(f) \leq \widetilde{T}$ for $3 \leq m \leq 6$ and $\widetilde{\mathcal{R}}$ is an m-feasible realization of f. ■

Upper Bound on Complexity of a Network

Theorem 5.3.18 states the upper bounds on complexity of a single function. Since a general circuit has multiple outputs, which can share logic, it is desirable to extend the bounds to a multi-output, multi-level network η. The number of factored form literals in η is the sum over all the internal nodes of the factored form literals of the function at each node. One way of generating the complexity upper bound for η is to sum the bounds obtained using the factored form of each node function. However, to get a closed form expression, one technicality has to be dealt with: Theorem 5.3.18 is stated only for the case when the number of literals in the factored form is greater than a small constant. However, it is valid if the function is m-infeasible. This implies that we can apply it to η if we are told that η has k m-feasible nodes and that the infeasible portion of η (i.e. all the m-infeasible nodes) has ℓ literals in factored form.

Corollary 5.3.29 *Given a network η, with k m-feasible nodes and ℓ factored form literals in the m-infeasible portion,*

$$\sum_{n \in IN(\eta)} C_m(n) \leq \begin{cases} k + \lfloor \frac{2\ell}{3} \rfloor & (m = 3) \\ k + \lfloor \frac{\ell}{2} \rfloor & (m = 4) \\ k + \lfloor \frac{2\ell}{5} \rfloor & (m = 5) \\ k + \lfloor \frac{\ell}{3} \rfloor & (m = 6) \end{cases}$$

Proof Each m-feasible function of η can be realized in one m-LUT. To each m-infeasible function, apply Theorem 5.3.18 and ignore the additive constants. ■

5.4 Experiments

m	N	$\sum S$	$\sum F$	$\sum G$	$\sum Q$
3	1187	12014	8091	6234	6774
4	881	9242	5268	3918	4178
5	727	7800	4017	2821	3068
6	527	6451	2855	2004	2158

Table 5.2: Experimental data

m number of inputs to the LUT
N number of m-infeasible functions over all benchmarks
$\sum S$ sum over N fns. of the SOP bound (Proposition 5.3.16)
$\sum F$ sum over N fns. of the factored form bound (Thm. 5.3.18)
$\sum G$ sum over N fns. of the mis-fpga results in *good* mode
$\sum Q$ sum over N fns. of the mis-fpga results in *quick* mode

m	$\frac{\sum S-\sum G}{\sum S}100\%$	$\frac{\sum S-\sum Q}{\sum S}100\%$	$\frac{\sum F-\sum G}{\sum F}100\%$	$\frac{\sum F-\sum Q}{\sum F}100\%$	$\frac{\sum\lvert F-G\rvert}{N}$	$\frac{\sum\lvert F-Q\rvert}{N}$
3	48.1	43.6	22.9	16.3	1.6	1.1
4	57.6	54.8	25.6	20.7	1.5	1.2
5	63.8	60.7	29.8	23.6	1.6	1.3
6	68.9	66.5	29.8	24.4	1.6	1.4

Table 5.3: Bounds vs. mis-fpga

We conducted experiments to compare the bounds with the synthesis results generated by mis-fpga. A suite of MCNC benchmarks optimized for literal count provided the starting point, as described in Section 3.6.1. For each node function f in a benchmark, the following is done:

1. The numbers of cubes and literals in the given SOP representation for f are substituted in Proposition 5.3.16 to obtain the S bound on the complexity of f for different values of m. Also, a factored form is computed using the quick decomposition of misII [12]. This factored form uses only the AND, OR, and NOT operations.[7] The number of literals in this factored form is used to derive the complexity bounds F for f using Theorem 5.3.18.

2. mis-fpga is run in two modes: *good* and *quick*. In the *good* mode, mis-fpga is run so as to obtain the best possible results. Different decomposition methods are used. These generate

[7]Theorem 5.3.18 is independent of the binary operations used in the factored form. It is better to use a generalized factored form (with all binary operations), as we did throughout the chapter, because potentially a smaller factored form is possible. However, misII creates a factored form only with AND, OR, and NOT operations.

various feasible realizations of f, on which block count minimization techniques including *covering* and support reduction are applied. The best realization is finally chosen. The number of LUTs in this realization is denoted by G (for *good*). In the *quick* mode, a factored form of f is obtained using the quick decomposition. This form is decomposed into two-input AND and OR gates using *tech-decomp* in misII. Finally, *covering* is performed. The number of LUTs obtained is denoted by Q (for *quick*). The idea is that the bound F was derived using a factored form and then applying a simple covering (recall the proof strategy of Theorem 5.3.18). Neither alternate decompositions nor support reduction were used. Since the *quick* mode uses a similar method, it provides a fairer comparison for the bounds as compared to the *good* mode.

Table 5.2 summarizes the statistics of the block counts obtained from these experiments. Values of m from 3 to 6 are considered. For each m, the number of m-infeasible functions encountered over all benchmarks is given in column N. An m-feasible node function is rejected from consideration, since there is no uncertainty about its complexity. Column $\sum S$ gives the sum over all N functions of the bounds obtained from the SOP using Proposition 5.3.16. Similarly, column $\sum F$ gives the sum of the bounds obtained from Theorem 5.3.18. $\sum G$ and $\sum Q$ list the sums of the LUT counts for the mis-fpga synthesized results for the N functions in *good* and *quick* modes respectively.

Table 5.3 shows the percentage differences between the SOP bounds $\sum S$ and the synthesized results $\sum G$ and $\sum Q$ in columns 2 and 3 respectively. The corresponding differences for the factored form bounds $\sum F$ are shown in columns 4 and 5. Note that columns 2 through 5 give only the percentage differences between the bounds and the synthesized results taken over all the examples. To obtain better information about each individual function, we compute in columns 6 and 7 the average sum of *absolute differences* between F and G, and F and Q respectively. This guards against the following cancelling effect present in the other columns. If for one function, F is higher than Q, but for another, it is lower, then *simple* differences may cancel out over the two functions, giving an impression that on average, the two quantities are closer to each other.

It can be seen from Tables 5.2 and 5.3 that the bound S from the SOP representation is weak. The mis-fpga results - G and Q values - are about 40-70% less. This was expected, since an SOP is not a good indicator of the number of gates or LUTs in multi-level synthesis. However, G and Q are closer to F. The Q bound is about 16-25% less than the F bound (column 5 in Table 5.3). Moreover, the difference grows with m. To understand the reason for this behavior, we study

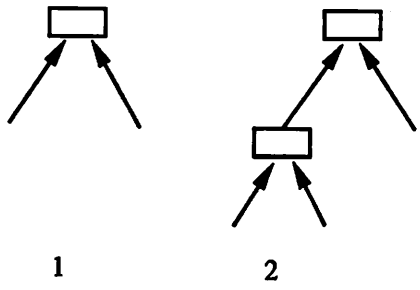


Figure 5.36: All possible patterns for getting a 3-feasible leaf-DAG

the kinds of groups (or patterns) used in the algorithms of Propositions 5.3.20, 5.3.23, 5.3.25, and 5.3.27. Figure 5.36 shows all possible patterns for converting a 2-feasible leaf-DAG to a 3-feasible leaf-DAG. The algorithm of Proposition 5.3.20 uses both of them. Figure 5.37 shows the patterns used to obtain a 4-feasible leaf-DAG. The algorithm of Proposition 5.3.23 uses all of them except the pattern 4. The usage of patterns worsens for $m = 5$ and $m = 6$. These patterns are shown in Figures 5.32 and 5.38 respectively. Only the first 5 patterns of Figure 5.32 are used in Proposition 5.3.25, whereas only the patterns 1, 3, 4, 5, and 9 of Figure 5.38 are used in Proposition 5.3.27. So for $m = 6$, only 5 out of 13 patterns are used. This may be why the factored form bounds for higher values of m move further away from the synthesized results, which have the freedom of using all the patterns. Another possible explanation is as follows. There are many factored forms that have ℓ literals. These result in different 2-feasible leaf-DAG structures. The bounds produced using the covering algorithms presented in the previous propositions hold even for the worst case. It may be that for a given leaf-DAG $\mathcal{R}$, much more reduction in the block count can be obtained when it is converted to an m-feasible leaf-DAG. The difference increases with m, partially because of the reason given earlier.

From the column 7 of Table 5.3, we see that, on average, the F bound differs from the Q bound by about one LUT. This is indeed encouraging, since F was derived using only the information about the literal count; the structure of the corresponding leaf-DAG was not taken into account.

To get an idea of the speed of the prediction (i.e., bound computation), we noted the total times taken for all benchmarks on a DEC 5900 workstation. While mis-fpga took 217 seconds to synthesize the functions in *quick* mode and 576 seconds in *good* mode, only 1.3 seconds were needed to compute the bounds from the SOP, and 1.6 seconds to compute first the factored forms and then the factored form bounds for all the functions. This fast prediction capability can be

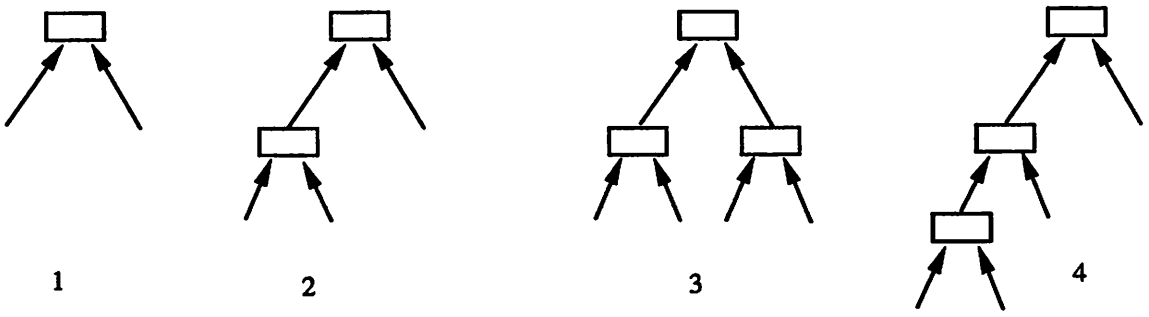


Figure 5.37: All possible patterns for getting a 4-feasible leaf-DAG

employed also to estimate whether a circuit will fit on an FPGA chip, without performing any technology mapping. Corollary 5.3.29 may be used to derive the corresponding bounds for the circuit.

5.5 Discussion

At the beginning of the chapter, we set out to evaluate the quality of a synthesis tool for LUT architectures. Since it is a hard problem, we addressed the next most promising problem: that of determining lower and upper bounds on the complexity of functions. Since the lower bound theory is weak, the problem of finding tight lower bounds was abandoned, although we did compute the exact complexity of the set $S(m+1)$.

Since in a synthesis environment, a representation of the function is already present, we addressed the problem of determining upper complexity bounds for a given representation. Two representations - sum-of-products and factored form, were considered. For $m \leq 5$, we proved that the factored form bounds we derive are tight under reasonable assumptions. The bounds were compared with the results produced by mis-fpga *(new)*. For all the values of m tried, the bounds were close enough to the synthesized results. This means that the bounds can be used for a fast prediction of the number of LUTs needed to implement a function.

The following problems remain unsolved:

1. Determining exact values of the complexity of the set of $(m+k)$-input functions using m-LUTs for $k > 1$. Extending the technique for $k = 1$ to higher values of k becomes tedious. This is because $C_m(S(m+k))$ is a non-decreasing function of k. To prove tightness, it has to be demonstrated that for all $p < C_m(S(m+k))$, p m-LUTs cannot implement some fixed

$f \in S(m+k)$. For each p, all possible configurations of p LUTs have to be enumerated. The number of configurations increases with p, making the task formidable.

2. Proving or disproving the tightness of the bound of Proposition 5.3.27.

3. Finding a technique for deriving upper bounds for a factored form for *all* m. So far, we have proposed techniques for $m \leq 6$. It is desirable that the bounds be tight under the assumptions already discussed. We conjecture that $\frac{2\ell}{m}$ is a tight upper bound for any m. It is based on the form of the bounds in Theorem 5.3.18.

4. Deriving these bounds for the commercial LUT-based architectures. For example, the LUT-section of a Xilinx 3090 CLB can realize two combinational functions, but there are some restrictions on their inputs.

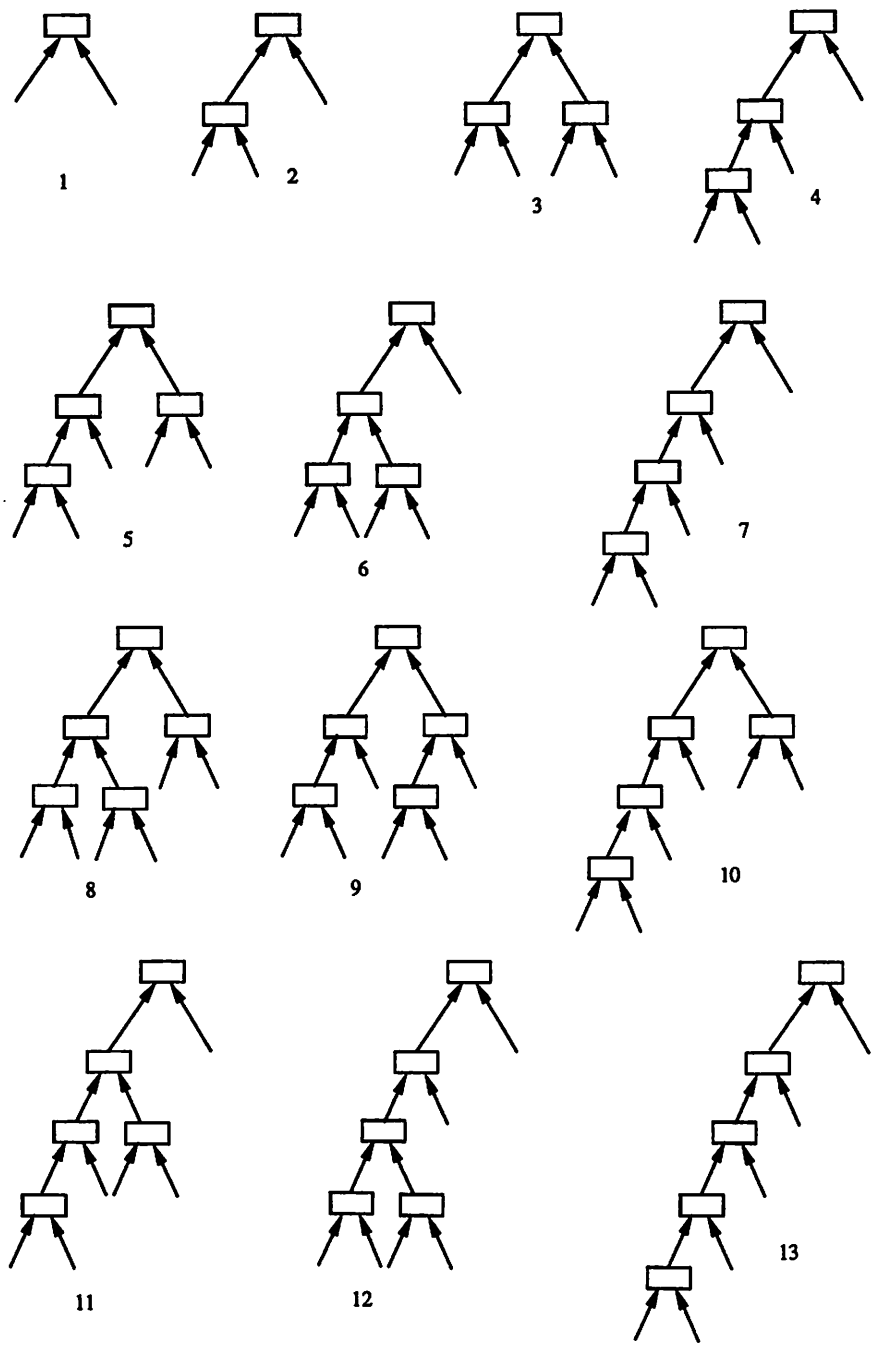


Figure 5.38: All possible patterns for getting a 6-feasible leaf-DAG

Chapter 6

Mapping Sequential Logic

6.1 Introduction

Thus far, we discussed synthesis techniques for combinational circuits. In this chapter, we address how to synthesize sequential circuits on to LUT-based architectures. The problem is interesting and important for many reasons:

1. Most of the circuits that are designed are sequential in nature.

2. Commercially available architectures, like Xilinx 3090, have sequential elements or flip-flops (which we had ignored for the combinational case) inside the basic block. These flip-flops are connected to the LUT-section of the CLB in such a manner that many different ways of mapping portions of circuit on to the CLB are possible. Finding all possible ways and then using them in mapping becomes a key problem.

3. The combinational and sequential resources on a chip are fixed. For instance, a Xilinx 3090 chip contains 320 CLBs, each of which has one combinational LUT-section that can implement two functions, and two flip-flops. Given a digital circuit C with some number of combinational and sequential elements that does not fit on a chip, it may be possible to transform it into another equivalent circuit $\tilde{C}$ with a different number of combinational and sequential elements such that $\tilde{C}$ fits on the chip. Finding these transformations is a challenge in itself.

A digital circuit may be described at an abstract level by a finite-state machine (FSM), which is a directed graph with labels on the edges. The vertices represent states of the machine, and

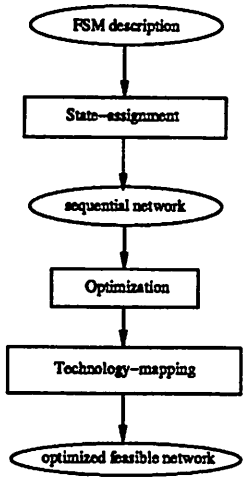


Figure 6.1: Three steps in sequential synthesis

the edges the transitions between the states. The label(s) on an edge (u, v) has two parts - an input part and an output part. The input part describes (input) conditions under which a transition occurs from state u to state v. The output part specifies values that the outputs of the machine take under this transition.

Given an FSM description, the aim of synthesis is to translate it to the target LUT architecture while optimizing some cost function. This may be the number of CLBs, delay through the circuit, or a combination. One way of achieving this is to divide the process into three steps. The first step assigns unique codes to the states of the FSM, the second optimizes the circuit obtained from the first step, and the third maps the result to the target architecture. This is shown in Figure 6.1.

State assignment: Most state-assignment tools minimize the number of literals in the resulting circuit, while not using too many flip-flops. This is achieved using a minimum- or near-minimum-length encoding. A typical LUT architecture provides many flip-flops on a chip (e.g., a Xilinx 3090 chip has twice as many flip-flops as CLBs). Hence the state-assignment problem for LUT architectures is different, in that the cost of a flip-flop is almost negligible. Some empirical work on the state-assignment problem for LUT architectures done by Schlag *et al.* [75] concluded that one-hot state-assignment gives minimum CLB count. We will have more to say on this in Section 6.3.1.

Optimization and mapping: A sequential circuit obtained from state-assignment must be optimized and mapped to the target FPGA architecture. More work needs to be done to optimize

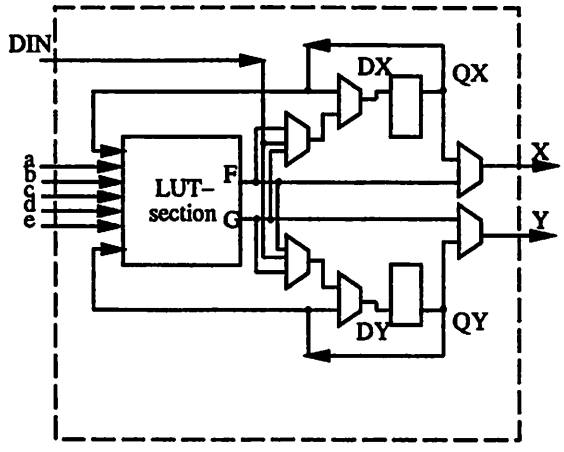


Figure 6.2: Xilinx 3090 CLB

the circuit for the LUT architectures. For lack of such algorithms, we will use standard optimization algorithms, which minimize the number of literals. Our focus will be on the problem of mapping an optimized sequential circuit onto LUT architectures. We know of one technology mapping program called XNFOPT [88], which supports sequential circuits.[1] However, it is a proprietary program, and details of its algorithms are not known to us. In Section 6.3, we compare our results with XNFOPT.

The chapter is organized as follows. We give an overview of the Xilinx 3090 architecture in Section 6.1.1, and precisely state the problem in Section 6.1.2. The proposed algorithms for sequential mapping are described in Section 6.2. Finally, we present results on a number of benchmark examples in Section 6.3.

6.1.1 Overview of the Architecture

In Figure 6.2, a CLB of the Xilinx 3090 architecture [88] is shown. The main features of the CLB are:

1) A combinational section (LUT-section): This sub-block has 7 inputs a, b, c, d, e, QX, and QY, and two outputs F and G. The inputs a, b, c, d, and e are called logic inputs. The LUT-section can be configured to implement any one of the following.

   a) Any single function F (or G) of up to five inputs. These inputs can be any five out of the seven inputs.

[1] XNFOPT is no longer in the Xilinx toolkit.

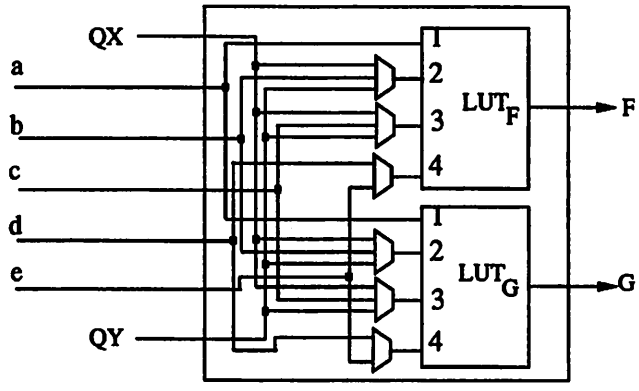


Figure 6.3: Internal structure of the LUT-section

b) Two functions F and G, which satisfy certain conditions, namely, each should have at most four inputs, one variable a should be common to both functions, the second variable (to both functions) can be any choice of b, QX, and QY, the third variable - any choice of c, QX, and QY, and the fourth variable - any choice of d or e. These conditions are called the **sequential mergeability conditions** (SMCs) and are shown in Figure 6.3.

c) Some function F of six or seven inputs. Here input e selects between two functions of four variables: both functions have common inputs a and d and any choice of b, c, QX, and QY for the remaining two variables.

2) Two D-type flip-flops with outputs QX and QY. Data input for each flip-flop is supplied from either F or G or direct data input DIN.

3) The CLB has two outputs X and Y. The output X can be either QX, in which case we say that X is *latched*, or F, in which case X is *unlatched*.[2] A similar statement holds for Y.

The flip-flop outputs QX and QY may be used as inputs to the LUT-section, thereby providing feedback paths that can be used to implement, within a CLB, the feedback loops of a sequential circuit.

We are interested in finding all possible choices for the CLB outputs X and Y. Let f^L denote the latched value of the signal f. From Figure 6.2, $DX, DY \in \{F, G, DIN\}$, so

[2]We will use the terms flip-flop and latch to mean a flip-flop.

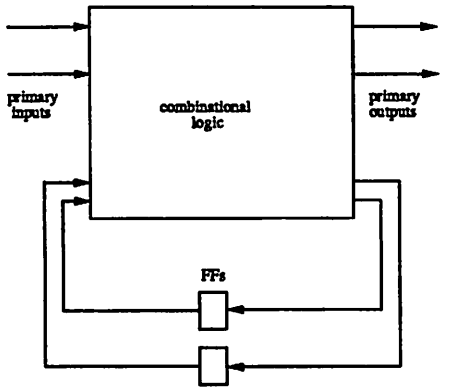


Figure 6.4: A synchronous sequential network

$QX, QY \in \{F^L, G^L, DIN^L\}$. Since $X = F$ or QX, and $Y = G$ or QY, we get

$$X \in \{F, F^L, G^L, DIN^L\} \tag{6.1}$$

$$Y \in \{G, F^L, G^L, DIN^L\} \tag{6.2}$$

As will be explained later, we do not make use of option 1 (c) in this work.

6.1.2 Problem Statement

"Given a sequential circuit, represented as blocks of combinational logic and flip-flops, and described in terms of Boolean equations, realize it using the minimum number of CLBs *of the target* LUT*-based architecture."*

We address this problem for the Xilinx 3090 architecture. The algorithms to be described here were published in an abridged form in [61]. We are not aware of any other published work for the synthesis of sequential circuits for LUT-based architectures.

6.2 Proposed Mapping Algorithms

Figure 6.4 shows the structure of a general synchronous sequential network. We view it as combinational logic plus sequential elements (flip-flops). Hence, we can use combinational techniques for the sequential synthesis problem.

We start with an optimized sequential network η and map it to the target LUT architecture. We assume that η does not have more than one flip-flop with the same input. If it has, all such flip-flops except one can be removed and the appropriate connections to the remaining flip-flop can

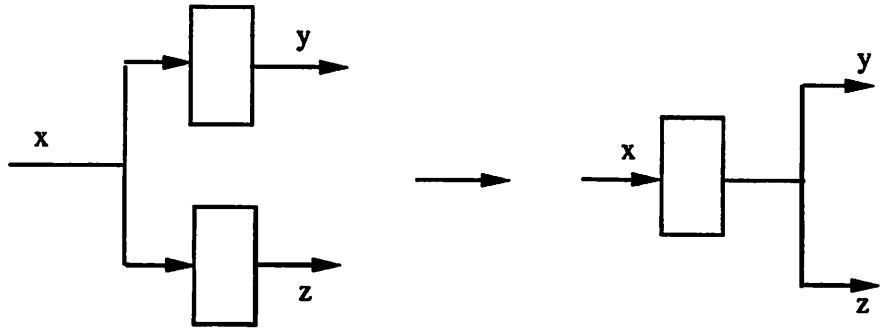


Figure 6.5: Deleting unnecessary flip-flops

be made. This is illustrated in Figure 6.5. No optimality is lost by using this transformation if a mapping algorithm that allows logic replication is used. The mapping algorithm has two main steps.

1. Obtain a k-optimal network $\tilde{\eta}$ from η, $k \leq m$ (for Xilinx 3090, $m = 5$) using the combinational techniques of Chapter 3; we experiment with $k = 4$ and $k = 5$.

2. Cluster (map) the elements of $\tilde{\eta}$ subject to the constraints imposed by the structure of the CLB; these constraints were described in Section 6.1.1. Each cluster is a CLB. So we are looking for a clustering that has the minimum number of clusters. We study two approaches to clustering - *map_together* and *map_separate*. These map combinational logic and flip-flops together or separately respectively.[3]

6.2.1 *map_together*

This approach simultaneously maps the combinational logic and flip-flops of the sequential network. It is an instance of the standard technology problem (as described in Sections 2.2.1 and 3.4.1). It has three main steps:

1. We first derive a set of pattern graphs, which represent possible ways in which a set of combinational and sequential elements can be placed together on one CLB. Since this step depends only on the structure of the CLB, it is performed just once and is not repeated for each network.

[3]These approaches do not use option 1 (c) of Section 6.1.1, because starting with a k-feasible network ($k \leq 5$), these just group nodes of the network, and hence cannot generate a function with greater than 5 inputs. However, option 1 (b) is used in *map_together*, which means that a pair of functions with a total of more than 5 inputs can be placed on a CLB.

2. From $\tilde{\eta}$, we generate all possible candidate matches. A match is a set of combinational and sequential elements of $\tilde{\eta}$ that can be placed on a CLB. Since patterns correspond to all the ways in which a CLB can be configured, each match has a pattern type (number) associated with it.

3. Finally, we select a minimum subset of these matches that will realize the network.

These steps are detailed next.

Pattern Graphs

The pattern graphs are derived from the CLB by considering a subset of the features of the CLB. Informally, a **pattern graph** is a configuration in which a CLB can be used. In our definition, a pattern is determined by the following:

1. The number and the names of the outputs used. For example, a pattern may use either X or Y or both - each belonging to the corresponding sets from (6.1) and (6.2).

2. The number of flip-flops used and their inputs. For instance, if one flip-flop is used, is its input DIN, F, or G?

For example, pattern 1 in Figure 6.6 is the configuration corresponding to the Xilinx 3090 CLB realizing just one combinational function with at most 5 inputs. Note that different ways of programming an LUT-section do not generate new patterns (there is one exception to this, and will be pointed out shortly). For example, the same pattern 1 corresponds to the output function of the CLB being abc or $abcd + a'b'c'd'e$. A pattern is determined by the connections within a CLB, which, in turn, are determined by how the multiplexors are programmed. If the outputs X and Y of the CLB are simply F and G respectively (so the outputs are unlatched), and DIN is being used, the corresponding patterns are 15 and 16. Then, if the second flip-flop is also being used, the pattern is 16; otherwise, it is 15.

Figure 6.6 shows 19 patterns corresponding to the Xilinx 3090 CLB.[4] A pattern is enclosed in a dotted rectangle. The square within each pattern represents the LUT-section and the two, small rectangles represent the flip-flops. A line going across the LUT-section indicates a buffer (patterns 27, 28, and 32). For convenience, we do not explicitly show the logic inputs a, b, c, d, and e; they are assumed to feed the LUT-section. Moreover, we do not show the multiplexors that realize options

[4]The unusual pattern numbering is because of the way the patterns were derived; see proof of Proposition 6.2.1.

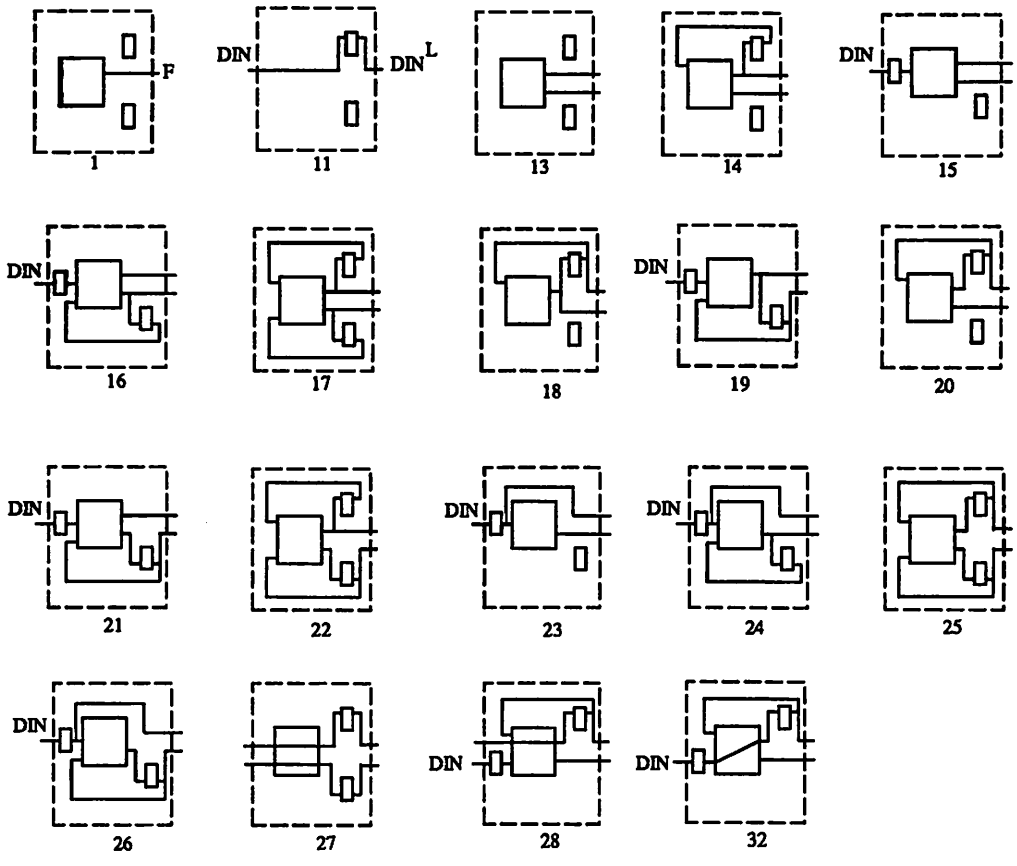


Figure 6.6: A complete set of pattern graphs

mentioned in Section 6.1.1, because patterns themselves are a result of choosing these options. However, we do show DIN whenever it is used. Also, whenever an output of a CLB is latched, the corresponding feedback connection within the CLB may not be actually used by the LUT functions F and G, i.e., the latched signal may not be an input to either F or G. But, if the output is unlatched, and the corresponding flip-flop is being used, the feedback connection must be used by either F or G. Finally, if a pattern can be realized in more than one way, we list it just once. For example, pattern 11 can also be realized without using DIN. We now motivate the choice of this set of pattern graphs.

Definition 6.2.1 *A set S of pattern graphs is* **complete** *for an* FPGA *architecture if for any sequential network η, there exists an optimum realization of η in terms of the basic block of the architecture, each block configured as some pattern of S.*

Proposition 6.2.1 *The set of pattern graphs shown in Figure 6.6 is complete for the Xilinx 3090 architecture.*

Proof First, we systematically derive an exhaustive and, therefore, complete set of patterns from the 3090 CLB. Then, we reduce the number of patterns without destroying completeness. In the following, f^L denotes the latched value of the signal f.

Recall that a pattern is determined by the following.

1. The number and names of the outputs used, and
2. The number of flip-flops used, and their inputs.

We enumerate the complete set of patterns by selecting X and Y from the sets in (6.1) and (6.2). First, we consider patterns with a single output, say X, and then, patterns with two outputs X and Y. Although there are 16 possibilities for the two-output case, they can be reduced because of symmetry. For each possibility, all sub-cases based on the number of flip-flops are enumerated.

1. $X = F$ (Figure 6.7): Pattern 1 corresponds to the case when no flip-flops are used. Patterns 2 through 4 use exactly one flip-flop and correspond to the three choices for the flip-flop input - F, DIN, and G. Patterns 5 through 7 use two flip-flops. Note that since the network does not have two flip-flops with the same input, patterns using two flip-flops with the same input are not considered.
2. $X = F^L$ (Figure 6.8): Since X uses one flip-flop, the case of using no flip-flop does not arise.
3. $X = G^L$: This case is symmetric to $X = F^L$. Since pattern graphs (as we have shown them) do not carry labels at the outputs of the CLB and the LUT-section, patterns of Figure 6.8 can be used.
4. $X = DIN^L$ (Figure 6.9).
5. $X = F, Y = G$ (Figure 6.10).
6. $X = F, Y = F^L$ (Figure 6.11).
7. $X = F, Y = G^L$ (Figure 6.12).
8. $X = F, Y = DIN^L$ (Figure 6.13).

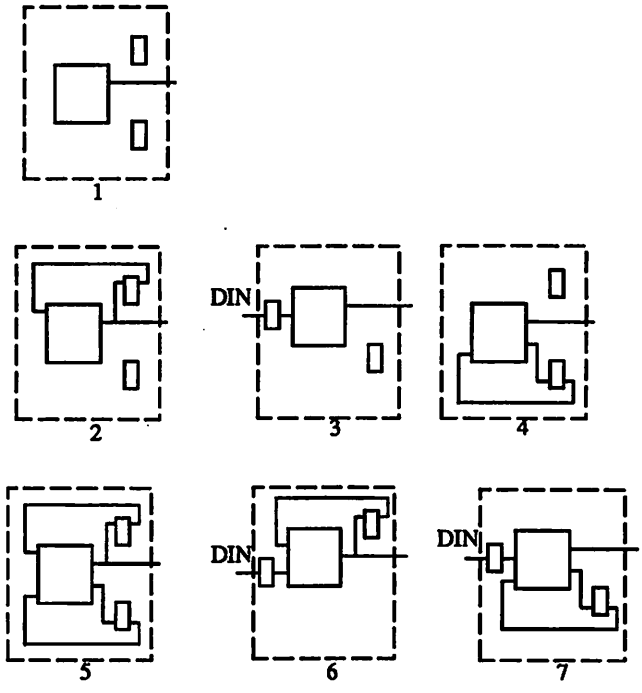


Figure 6.7: $X = F$

9. $X = F^L, Y = F^L$: since no two flip-flops in the network have the same input, this case is identical to $X = F^L$.

10. $X = F^L, Y = G^L$ (Figure 6.14).

11. $X = F^L, Y = DIN^L$ (Figure 6.15).

12. The other cases are symmetric to the ones already discussed, and since in pattern graphs, we do not label the outputs of the CLB and LUT-section (see the figures), same patterns can be used in these cases. For example, $Y = G$ is symmetric to $X = F$; therefore, patterns of

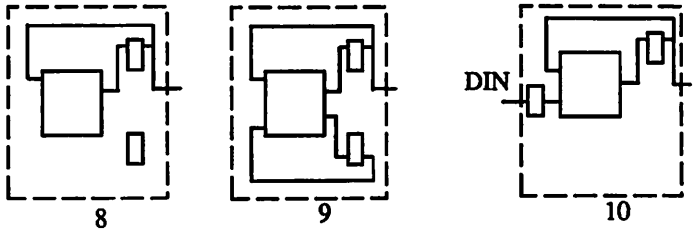


Figure 6.8: $X = F^L$

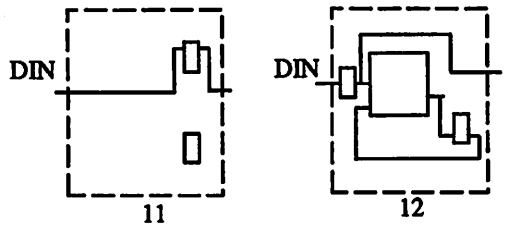


Figure 6.9: $X = DIN^L$

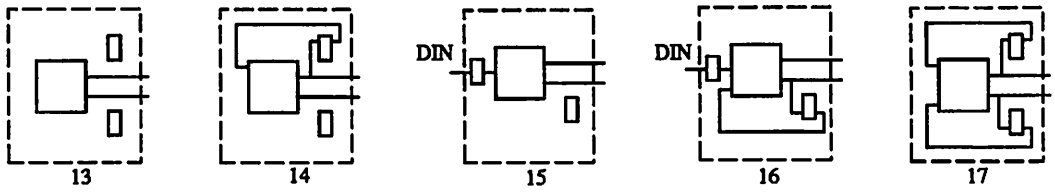
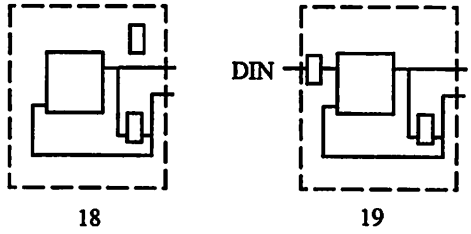


Figure 6.10: $X = F, Y = G$

Figure 6.7 can be used.

It is easily checked that for each case, all possibilities are generated. And since we explicitly generated all the cases, this set of pattern graphs is complete. However, there is an incompatibility between the way patterns are derived and the way matches are generated. A pattern assumes that F and G are *any* functions, including the trivial buffer function. As we will see soon, matches are generated in our mapping algorithms by picking at most two combinational elements f and g, and two flip-flops Q_1 and Q_2 *from the sequential network*, and checking if their interconnection structure is identical to that of some pattern. The possibility of F or G being buffers is not considered, because in the sequential network buffers are not normally present (except for fanout optimization). We then have two options:



. Figure 6.11: $X = F, Y = F^L$

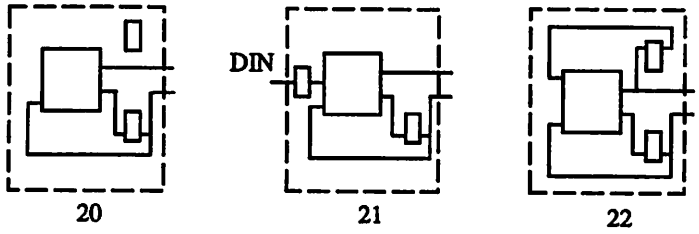


Figure 6.12: $X = F, Y = G^L$

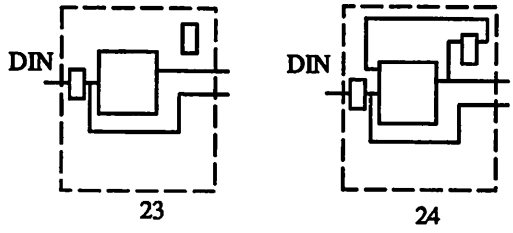


Figure 6.13: $X = F, Y = DIN^L$

1. While generating matches, use patterns 1 through 26, and for each pattern, consider the possibility that F or G could be buffers, or
2. Explicitly generate patterns corresponding to buffers.

We choose the second option. Doing a complete analysis for the cases - when both F and G are buffers, and when exactly one, say F, is a buffer, yields nine more patterns, shown in Figure 6.16. This analysis is more cumbersome than what we just did, because there are 3 choices for a buffer input: QX, QY, or a logic input (a, b, c, d, e). Details of enumeration are omitted for brevity.

Next, we state an observation that reduces the number of patterns needed (with a binate covering algorithm) without destroying completeness. Let us examine the patterns more closely. For example, the only difference between pattern 4 and pattern 14 is that 14 uses both outputs,

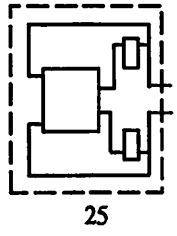


Figure 6.14: $X = F^L, Y = G^L$

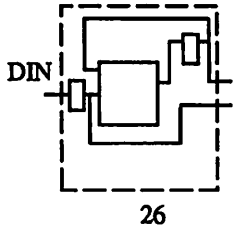


Figure 6.15: $X = F^L, Y = DIN^L$

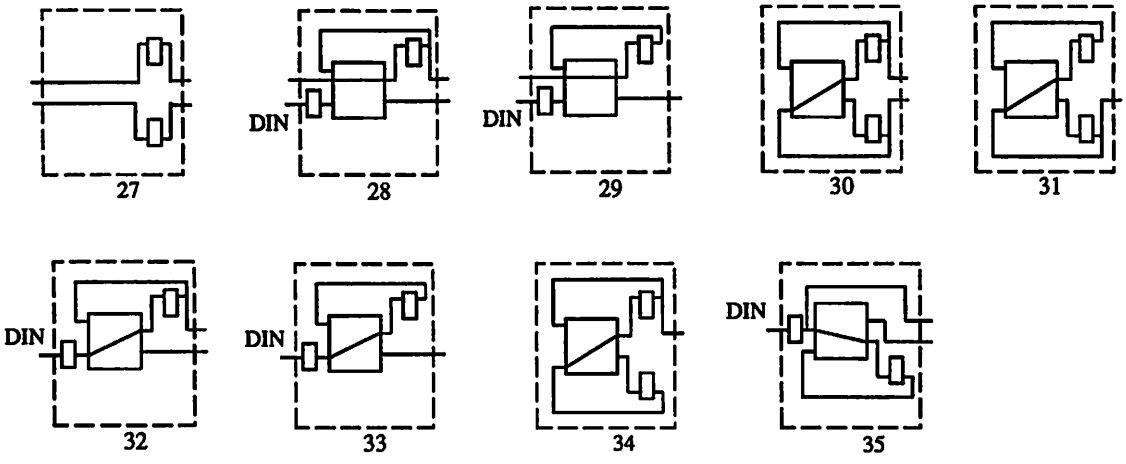


Figure 6.16: Patterns with buffers

whereas 4 uses just one. Both use same number of combinational functions and flip-flops, and have identical interconnection structures. So pattern 4 can be deleted from the pattern set. This leads to the notion of a pattern covering another pattern.

Definition 6.2.2 *Pattern i* **covers** *pattern j if i and j use the same number of combinational functions and flip-flops, have identical interconnection structures, except that i has more outputs than j.*

So pattern 14 covers pattern 4. Note that pattern 1 is not covered by pattern 13, because 13 uses both combinational functions of the LUT-section, whereas 1 uses just one.

Proposition 6.2.2 *Let i and j be two patterns in a complete set S of patterns such that i covers j. Then, S - $\{j\}$ is also complete.*

Proof Consider a match M_j with the pattern type j. Let (f, g, Q_1, Q_2) be the information attached with M_j, i.e., f and g are the combinational functions of the sequential network chosen to map onto

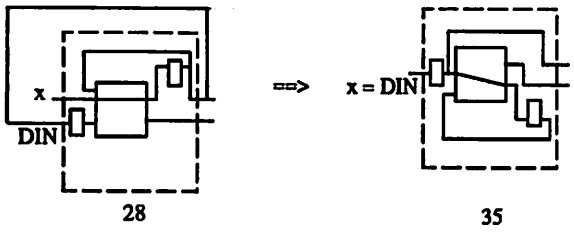


Figure 6.17: Pattern 35 can be implemented by pattern 28

F and G, and Q_1 and Q_2 are the flip-flops chosen to map onto QX and QY of the CLB. Some of f, g, Q_1 or Q_2 may be NIL. Then there exists a match M_i with the pattern type i and with the same information (f, g, Q_1, Q_2). This is because i and j use the same number of combinational functions and flip-flops, and have identical interconnection structures, except that i has more output(s). Let there be an optimum mapping solution in which M_j is present. Then, M_j may be replaced by M_i - we can ignore the unused output(s) of M_i. Note that M_i has exactly the same inputs as M_j, and each input is either a primary input or an output of some match. Since the number of matches remains the same, the new solution is also optimum.

This process is repeated for all matches in the optimum solution with pattern type j, resulting in an optimum solution without matches of type j. Thus pattern j can be deleted from S without destroying the completeness of S. ■

Scanning various patterns, it is seen that pattern 2 is covered by 18, 3 by 23, 4 by 14, 5 by 17, 6 by 24, 7 by 16, 8 by 18, 9 by 25, 10 by 26, 12 by 26, 29 by 28, 31 by 30, 33 by 32, and 34 by 30. So patterns 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 29, 31, 33, and 34 can be deleted.

As shown in Figure 6.17, pattern 35 can be implemented by 28 if we route the buffered, latched output of 28 to the DIN input using routing resources external to the CLB. So pattern 35 can be deleted. Similarly, pattern 30 can be implemented by 26 by routing the output F^L of 26 to the DIN input. So 30 can also be deleted.

This gives us a complete set of 19 patterns, as shown in Figure 6.6. ■

Generating Matches

The starting network $\tilde{\eta}$ is k-optimal, $k \leq 5$. We systematically generate all the matches of $\tilde{\eta}$. A match is a set of combinational nodes and flip-flops *of the sequential network* $\tilde{\eta}$ that can be mapped onto a CLB. Since patterns correspond to all the ways in which a CLB can be configured, each match has a pattern type (number) associated with it. For example, if a match has just one

```
/* one or both of Q1,Q2 may be NIL */
foreach set of candidate flip-flops Q1,Q2
     foreach pattern p in pattern set
          if (f,g,Q1,Q2) mappable to p &&
          (mergeability satisfied(f,g,Q1,Q2,p))
               create match with type p
```

Figure 6.18: Match generation for (f, g)

flip-flop (and no combinational function), its pattern type is 11 (Figure 6.6). However, if it also has one combinational logic function of the network, its type can be either 18 or 23.

For the Xilinx 3090 CLB, a match cannot contain more than two combinational nodes and two flip-flops from $\tilde{\eta}$. These combinational nodes will be denoted by f and g, and the flip-flops by Q_1 and Q_2. The combinational nodes f and g map to F and G, and the flip-flops Q_1 and Q_2 to QX and QY. Therefore, the information attached to a match for (f, g, Q_1, Q_2) consists of the following:

1. for F and G, candidate combinational functions f and g,
2. for QX and QY, candidate flip-flops Q_1 and Q_2,
3. correspondence of the inputs of f and g to the input pins of the LUT-section, LUT_F, and LUT_G (Figure 6.3). This is generated only if the match is finally selected and the final netlist is desired,
4. outputs X and Y, and
5. the pattern type.

Note that one or more of f, g, Q_1, and Q_2 may be NIL, implying that fewer elements were chosen from $\tilde{\eta}$ for placement on the CLB and so the corresponding element of the CLB will not be used.

Matches are generated by considering the sequential circuit as a set of combinational nodes and flip-flops. In a CLB, we can place either zero, one, or two combinational nodes (the case of zero combinational nodes corresponds to both f and g being NIL, the case of one combinational node to exactly one of f and g being NIL, and the case of two to none of f and g being NIL), and

either zero, one, or two flip-flops. Cases with zero or one combinational node inside a CLB are simple. So we just describe the case of a pair of combinational functions (nodes) of the network, say f and g, and show the pseudo-code for match generation in Figure 6.18. The details of various steps are as follows.

Generating candidate flip-flop(s) : Given combinational functions f and g of $\tilde{\eta}$, we wish to determine all possible candidate flip-flops Q_1 and Q_2 such that (f, g, Q_1, Q_2) is a candidate match. This is done as follows. f and g can be mapped on functions F and G of a CLB either with or without the use of DIN. If DIN is not used (for instance, patterns 14 and 17), only those flip-flops that receive their data inputs from either f or g are candidates for QX and QY. If DIN is used, flip-flops that fan out to either f or g should also be considered. This is evident from the patterns 15, 16, and 21 in Figure 6.6 - these are the patterns that use DIN and accommodate two combinational functions of $\tilde{\eta}$.

Mapping (f, g, Q_1, Q_2) to a pattern p : Given (f, g, Q_1, Q_2), this step checks for the compatibility of (f, g, Q_1, Q_2) with each pattern p. Since the match will use both f and g, a pattern p that does not use both F and G cannot generate a match corresponding to (f, g, Q_1, Q_2). Also, the number of non-NIL flip-flops in $\{Q_1, Q_2\}$ should be the same as that in p. Additionally, in a CLB whenever a combinational logic node f feeds a flip-flop Q, there are two choices: the corresponding output of the CLB - X or Y - can be either f or Q. So an (f, g, Q_1, Q_2) combination may map to many patterns.

Checking for mergeability conditions : After at most two flip-flops Q_1 and Q_2 have been chosen to be placed inside a CLB, and a compatible pattern p has been found, it cannot yet be concluded that (f, g, Q_1, Q_2) is a match for the CLB with pattern type p. This is because we have not yet checked if (f, g, Q_1, Q_2) satisfy the sequential mergeability conditions, or SMCs, which were described in Section 6.1.1. If we use SMCs as such for this check, we have to find a correspondence from the input pins of the LUT-section to the inputs of f and g. This is because SMCs are in terms of the input pins of the LUT-section. In the worst case, all possible mappings have to be tried, which may be too much work. We now give an alternate characterization of SMCs in terms of the inputs of f, inputs of g, Q_1, and Q_2 that does not require the correspondence. We first assume that the outputs of the flip-flops in a CLB are *available* (i.e., they are connected to the outputs of the CLB) and hence can be connected to the logic inputs $\{a, b, c, d, e\}$ of the same CLB using the routing

resources external to the CLB. Then we characterize the case when the flip-flop outputs are not available at the CLB outputs. Note that *the availability of flip-flop outputs is a pattern-dependent property*. For example, in pattern 20, the flip-flop output is available, whereas in 14, it is not. Given a pattern type, the corresponding characterization should be checked. That is why the mergeability check in Figure 6.18 takes p as an argument.

Proposition 6.2.3 *Given that the flip-flop outputs are available, the following are equivalent:*

A. *Flip-flops Q_1 and Q_2 are assigned to QX and QY respectively, and f and g satisfy sequential mergeability conditions.*

B. 1. $|\sigma(f)| \leq 4, |\sigma(g)| \leq 4$, *and*

   2. *Let N be the number of flip-flops in the* CLB *whose outputs are truly being used by the* LUT-*section.*[5] *Then*

      (a) *if $N = 0$, $|\sigma(f) \cup \sigma(g)| \leq 5$.*

      (b) *if $N = 1$, $|\sigma(f) \cup \sigma(g)| \leq 6$.*

      (c) *if $N = 2$, for $i = 1, 2$, define*
      $FLAG_{f,i} = 1$ *if* $Q_i \in \sigma(f)$, *else* $FLAG_{f,i} = 0$,
      $FLAG_{g,i} = 1$ *if* $Q_i \in \sigma(g)$, *else* $FLAG_{g,i} = 0$.
      Let $SUM = \sum_i FLAG_{f,i} + \sum_i FLAG_{g,i}$.
      Then,

$$|\sigma(f) \cup \sigma(g)| \leq \begin{cases} 7 & \text{if } SUM = 2 \\ 6 & \text{otherwise.} \end{cases}$$

Notes:

- Any of Q_1 and Q_2 may be NIL.
- In 2 (c), since $N = 2$, $SUM \geq 2$ always.

For the proof, we will need the correspondence between the inputs pins of the LUT-section and the inputs of the functions placed on the LUT-section. The following formalism is developed with that aim in mind.

[5] For example, if the outputs of the CLB flip-flops realize functions Q_1 and Q_2, but only Q_1 appears in $\sigma(f) \cup \sigma(g)$, then $N = 1$.

Mapping inputs of LUT_F and LUT_G to inputs of the functions

1. Given a partial mapping ρ from A to B, and $S \subseteq A$,

$$\rho(S) = \{y | y = \rho(x), x \in S \text{ and } \rho(x) \text{ is defined}\}$$

We shall denote $\rho(A)$ as $Im_A(\rho)$, and if it is clear which A we have in mind, we will simply write $Im(\rho)$.

2. Given two (partial) mappings π and γ, their composition is defined as $(\gamma \circ \pi)(x) = \gamma(\pi(x))$, if $\pi(x)$ and $\gamma(\pi(x))$ are defined. Otherwise, it is undefined.

Let $P = \{a, b, c, d, e, QX, QY\}$ denote the set of input pins of the LUT-section, and P_F (P_G) the 4 input pins of LUT_F (LUT_G). To address the problem of placing two Boolean functions on the LUT-section, we define two mappings γ and π.

1. If a Boolean function f is placed on the LUT-section, the partial mapping γ_f is from the set of input pins of the LUT-section to the set of inputs of f, i.e., from P to $\sigma(f)$. Let $x \in P$. If γ_f is defined at x, then pin x of the LUT-section is tied to the input $\gamma_f(x)$ of f. Otherwise, x is not connected to any input of f.

Example 6.2.1 *Let $f = A'BC + D$. Then $\sigma(f) = \{A, B, C, D\}$. If f is to be placed on the* LUT*-section, we can define $\gamma_f(a) = A, \gamma_f(b) = D, \gamma_f(c) = B, \gamma_f(d) = C$. This asserts that pins a, b, c, and d of the* LUT*-section should be tied to the inputs A, D, B, and C of f.*

If two Boolean functions f and g are to be placed on the LUT-section, we can similarly define γ from P to $\sigma(f) \cup \sigma(g)$. This is shown in Figure 6.19.

2. $\pi = (\pi_F, \pi_G)$ is a mapping defined from P_F and P_G, the input pins of LUT_F and LUT_G respectively (i.e. pins 1, 2, 3, 4 in Figure 6.3), to P. This mapping specifies the selection at the multiplexors of Figure 6.3, and is shown in Figure 6.19. We shall display π with a table.

Example 6.2.2 *The table*

function	1	2	3	4
F	a	QX	c	d
G	a	b	QY	e

means that for LUT_F*, pin 1 is tied to pin a of the* LUT*-section, pin 2 to QX, pin 3 to c, and pin 4 to d. Similarly, for* LUT_G*, pin 1 is tied to pin a, pin 2 to b, pin 3 to QY, and pin 4 to e.*

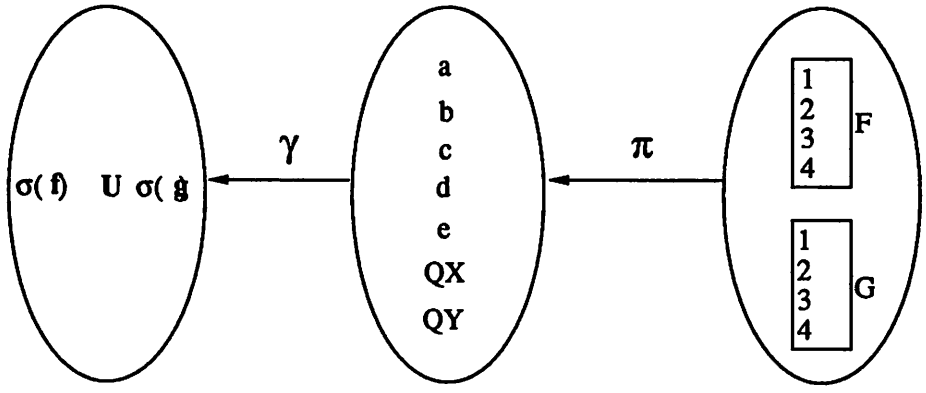


Figure 6.19: Mappings π and γ

From SMCs and Figure 6.3, it is clear that for $\pi = (\pi_F, \pi_G)$ to be valid, necessarily

$$\pi_F(1) = \pi_G(1) = a; \quad \pi_F(2), \pi_G(2) \in \{b, QX, QY\}; \tag{6.3}$$

$$\pi_F(3), \pi_G(3) \in \{c, QX, QY\}; \qquad \pi_F(4), \pi_G(4) \in \{d, e\} \tag{6.4}$$

Without loss of generality, we can assume that $\pi_F(4) = d$, $\pi_G(4) = e$. This is because if pin 4 of LUT_F and pin 4 of LUT_G are tied to different external inputs x and $y \in \sigma(f) \cup \sigma(g)$ respectively, define $\gamma(d) = x$, and $\gamma(e) = y$, and if to the same external input x, connect d and e pins to x. In other words, set $\gamma(d) = \gamma(e) = x$.

In the course of determining if two functions f and g can be mapped to F and G respectively of the LUT-section, we initially assign some inputs of f and g to some pins of the LUT-section, and some pins of the LUT-section to some pins of LUT_F and LUT_G. In other words, the mappings γ and π are partially determined. It is convenient to display this information in a composite $\pi - \gamma$ chart.

Example 6.2.3

function	1	2	3	4
F	$a(A)$	$QX(B)$	c	d
G	$a(A)$	b	QY	$e(C)$

where $A, B, C \in \sigma(f) \cup \sigma(g)$. Then, in addition to what the previous table conveyed, this table says that pin a is tied to A, QX to B, and e to C, i.e., $\gamma(a) = A, \gamma(QX) = B, \gamma(e) = C$. Nothing is said about $\gamma(b), \gamma(c), \gamma(d)$, and $\gamma(QY)$.

We characterize the sequential mergeability conditions in terms of the existence of the mappings π and γ.

Proposition 6.2.4 *The following are equivalent:*

1. *Flip-flops Q_1 and Q_2 are assigned to QX and QY respectively, and f and g satisfy sequential mergeability conditions.*

2. *There exist partial functions π_F, π_G, and γ such that $(\gamma \circ \pi_F)(P_F) \supseteq \sigma(f)$, $(\gamma \circ \pi_G)(P_G) \supseteq \sigma(g)$, where*

   (a) π_F, π_G satisfy (6.3) and (6.4),

   (b) $\gamma(QX) = Q_1$ if Q_1 is not 'NIL' and $Q_1 \in \sigma(f) \cup \sigma(g)$, and

   (c) $\gamma(QY) = Q_2$ if Q_2 is not 'NIL' and $Q_2 \in \sigma(f) \cup \sigma(g)$.

Note the following conventions:

1. If Q_1 is 'NIL', $\gamma(QX)$ is undefined. Same holds for Q_2.

2. If Q_1 is not 'NIL', but $Q_1 \notin \sigma(f) \cup \sigma(g)$, $\gamma(QX)$ is undefined. This is primarily because we assumed that γ is from P to $\sigma(f) \cup \sigma(g)$. Same holds for Q_2.

Sketch of Proof The second statement is a mathematical way of saying that we can connect the set of pins P_F and P_G to the set of inputs of f and g respectively through the pins of the LUT-section such that no pin of LUT_F, LUT_G, and LUT-section is connected to two different pins/inputs. At the same time, the conditions in (6.3) and (6.4) are satisfied. These conditions represent exactly the choices that are allowed inside the LUT-section through the use of the multiplexors, i.e., the SMCs. ■

Proof (of Proposition 6.2.3) (A $\Rightarrow$ B): Because the CLB is symmetric, we can assume that f is mapped to F and g is mapped to G. From Proposition 6.2.4, there exist mappings γ, π_F, π_G satisfying (6.3) and (6.4), where $\gamma(QX) = Q_1, \gamma(QY) = Q_2$ (if Q_1 is NIL, $\gamma(QX)$ is undefined).

From SMCs (or Figure 6.3), it can be seen that condition 1 is met, i.e., $|\sigma(f)| \leq 4$ and $|\sigma(g)| \leq 4$. To see that condition 2 is met, we do a case analysis on N.

(a) $N = 0$: pins QX and QY of the LUT-section are not being truly used by either LUT_F or LUT_G. We can, then, safely assume that $Im(\pi_F) \subseteq \{a, b, c, d, e\}$, and $Im(\pi_G) \subseteq \{a, b, c, d, e\}$. $Im(\pi) = Im(\pi_F) \cup Im(\pi_G) \subseteq \{a, b, c, d, e\}$. Since $\gamma(Im(\pi)) = \sigma(f) \cup \sigma(g)$, and $|\gamma(T)| \leq |T|$ for any $T \subseteq \{a, b, c, d, e, QX, QY\}$, we get $|\sigma(f) \cup \sigma(g)| \leq 5$ (by letting $T = Im(\pi)$), which is exactly the condition 2 (a).

(b) $N = 1$: let Q_1 be the flip-flop output being used in $\sigma(f) \cup \sigma(g)$. This means that Q_2 is not being used in $\sigma(f) \cup \sigma(g)$. Since Q_2 is assigned to QY, we can assume, without loss of generality, that $QY \notin Im(\pi)$. Then, $Im(\pi) \subseteq \{a, b, c, d, e, QX\}$. Using similar arguments as in (a), we get $|\sigma(f) \cup \sigma(g)| \leq 6$.

(c) $N = 2$: We do a case analysis on SUM, where $2 \leq \text{SUM} \leq 4$.

- SUM = 2: It is always true that $Im(\pi) \subseteq \{a, b, c, d, e, QX, QY\}$. It follows that $|\sigma(f) \cup \sigma(g)| \leq 7$.
- SUM = 3: Without loss of generality, f uses both Q_1 and Q_2, and g uses Q_1. If we decide to tie either Q_1 or Q_2, say Q_1, to $y \in \{a, b, c, d, e\}$ (using routing external to the CLB - we can do so since the flip-flop outputs are available), we are done, since $\gamma(y) = \gamma(QX) = Q_1$ (here we used the fact that QX corresponds to Q_1), and hence $|\sigma(f) \cup \sigma(g)| \leq 6$. Otherwise, if neither Q_1 nor Q_2 is tied to P, both $b, c \notin Im(\pi_F)$. This is because f is using both Q_1 and Q_2, which is possible only if $\pi_F(2) \neq b$ and $\pi_F(3) \neq c$. Similarly, either b or $c \notin \pi_G$). Then, either b or $c \notin Im(\pi)$. Hence $|\sigma(f) \cup \sigma(g)| \leq 6$.
- SUM = 4: Same argument as for SUM = 3.

(B $\Rightarrow$ A): Because of symmetry, we assume, without loss of generality, that f is mapped to F, and g is mapped to G. Hereafter, f and F will be used interchangeably, and so will g and G.

The proof is constructive; it provides the mappings γ - from the pins of the LUT-section to $\sigma(f) \cup \sigma(g)$, and $\pi = (\pi_F, \pi_G)$ - from P_F and P_G to the pins of the LUT-section. To prove that f and g satisfy SMCs, it follows from Proposition 6.2.4 that it is enough to show the existence of these mappings.

In the proof, we will repeatedly use the fact that given two sets A and B,

$$|A \cup B| = |A| + |B| - |A \cap B| \tag{6.5}$$

Our strategy to determine γ is as follows.

1. If Q_1 or Q_2 appear in $\sigma(f) \cup \sigma(g)$, we assign them to certain pins of the LUT-section. This partially determines γ.
2. Then, we construct three ordered lists L_f, L_g, and L_C (C for common), each being a list of pin names from $\{a, b, c, d, e\}$. The lists tell the order in which the function inputs are to be assigned to the pins of the LUT-section.

3. Next, we assign the set of common inputs $C = \sigma(f) \cap \sigma(g)$ to the pins of the LUT-section using the order in L_C. If an assigned pin is present in L_f or L_g, it is deleted therefrom.

4. Now, we start with the remaining (unassigned) inputs of f and assign them to the (unassigned) pins in the list L_f. We delete the assigned pins from L_g.

5. Finally, we assign the unassigned inputs of g using L_g in a similar fashion.

A few remarks are in order here.

(a) An entry $\{d, e\}$ appearing in L_C means that both d and e are to be tied together to the same common input of f and g. Similar semantics hold for $\{b, c\}$.

(b) L_g is not really needed; in the last step of our strategy, we can assign the remaining inputs of g in any order as long as we assign these inputs to unassigned pins only. For the sake of clarity, we explicitly give L_g.

The proof is by a case analysis on N.

1. $N = 0$: Then $|\sigma(f) \cup \sigma(g)| \leq 5$. The lists are as follows:

$$\begin{aligned} L_C &= abc\{d, e\} \\ L_f &= dcba \\ L_g &= ecba \end{aligned}$$

We consider sub-cases depending on the value of $|C| = |\sigma(f) \cap \sigma(g)|$.

(a) $|C| = 0$: From (6.5), $|\sigma(f)| + |\sigma(g)| \leq 5$. Since $\sigma(f)$ and $\sigma(g)$ are disjoint, it can be easily checked that for all $|\sigma(f)|$ and $|\sigma(g)|$ satisfying $|\sigma(f)| + |\sigma(g)| \leq 5, |\sigma(f)| \leq 4$, and $|\sigma(g)| \leq 4$, a valid pin to input assignment can be obtained. For example, if $|\sigma(f)| = 2$ and $|\sigma(g)| = 3$, tie the two inputs of f to pins d and c - the first two items on L_f, and then tie the inputs of g to pins e, b, and a. Note that after having assigned pin c to an input of f, we cannot use it for any input of g. Also note that although a is common to both LUT_F and LUT_G, the input connected to a will not appear in the expression realized by F.

Comment: Had we arbitrarily assigned inputs of f to the pins of the LUT-section, a match could have been missed. For example, consider f and g such that $N = 0, \sigma(f) =$

$\{V, W, X\}, \sigma(g) = \{Y, Z\}$, and $|C| = 0$. f and g satisfy SMCs - in fact, we can define γ as: $\gamma(d) = V, \gamma(c) = W, \gamma(b) = X, \gamma(e) = Y, \gamma(a) = Z$. Then we have a natural choice for π: $\pi_F(1) = \pi_G(1) = a, \pi_F(2) = b, \pi_F(3) = c, \pi_F(4) = d, \pi_G(4) = e$; other values can be set arbitrarily. This is a valid assignment. However, had we defined γ as: $\gamma(a) = V, \gamma(b) = W, \gamma(c) = X$, then either $\gamma(d) = Y$ and $\gamma(e) = Z$, or $\gamma(d) = Z$ and $\gamma(e) = Y$. In either case, we will not find a valid match, since pin 4 of LUT_G cannot be connected to both d and e (using map π_G)

(b) $|C| = 1$: We will assign the common input to pin a, the first item in L_C. Then, f and g have at most 3 unassigned inputs each, and a total of at most 4 unassigned inputs. Once again, it is easy to see that in all cases we get valid γ, π_F, and π_G.

(c) The cases $|C| = 2$, $|C| = 3$, and $|C| = 4$ are exactly similar.[6]

2. $N = 1$: Then $|\sigma(f) \cup \sigma(g)| \leq 6$. Let Q_1 be the flip-flop output that appears in $\sigma(f) \cup \sigma(g)$. Two cases are possible:

(a) Q_1 occurs only in one function, say f. We let the internal feedback of the CLB through QX provide Q_1 at pin 2 of LUT_F. In other words, $\gamma(QX) = Q_1, \pi_F(2) = QX$. We can safely eliminate QX and QY from consideration at pins 2 and 3 of LUT_G and pin 3 of LUT_F. Then the $\pi - \gamma$ chart looks like:

function	1	2	3	4
f	a	$QX(Q_1)$	c	d
g	a	b	c	e

The lists are:

$$L_C = ac\{d, e\}$$
$$L_f = dca$$
$$L_g = ebac$$

(b) Q_1 occurs both in f and g. We assign Q_1 to QX, and QX to pin 2 of LUT_F and pin 3 of LUT_G; i.e., $\gamma(QX) = Q_1, \pi_F(2) = \pi_G(3) = QX$. Then the $\pi - \gamma$ chart looks like:

function	1	2	3	4
f	a	$QX(Q_1)$	c	d
g	a	b	$QX(Q_1)$	e

[6] For the rest of the proof, we just give lists L_C, L_f, and L_g, and leave it to the reader to make sure that for all possible values of $|C|$, a valid assignment exists.

The lists are:

$$\begin{aligned} L_C &= a\{d,e\}\{b,c\} \\ L_f &= cda \\ L_g &= bea \end{aligned}$$

3. $N = 2$. Let Q_1, Q_2 be the outputs of the flip-flops. We have three cases:

   (a) SUM = 2: Then $|\sigma(f) \cup \sigma(g)| \leq 7$. There are two sub-cases:

      i. Q_1 and Q_2 are used by different functions. Let us assume, without loss of generality, that Q_1 occurs in $\sigma(f)$, and Q_2 in $\sigma(g)$. We assign Q_1 to pin QX, and QX to pin 2 of LUT_F. Similarly, assign Q_2 to pin QY, and QY to pin 3 of LUT_G; i.e., $\gamma(QX) = Q_1, \gamma(QY) = Q_2, \pi_F(2) = QX, \pi_G(3) = QY$. The $\pi - \gamma$ chart looks like

function	1	2	3	4
f	a	$QX(Q_1)$	c	d
g	a	b	$QY(Q_2)$	e

      The lists are:

$$\begin{aligned} L_C &= a\{d,e\}\{b,c\} \\ L_f &= cda \\ L_g &= bea \end{aligned}$$

      ii. Q_1 and Q_2 are used by the same function, say f. We set $\gamma(QX) = Q_1, \gamma(QY) = Q_2, \pi_F(2) = QX, \pi_F(3) = QY$. The $\pi - \gamma$ chart looks like

function	1	2	3	4
f	a	$QX(Q_1)$	$QY(Q_2)$	d
g	a	b	c	e

      The lists are:

$$\begin{aligned} L_C &= a\{d,e\} \\ L_f &= da \\ L_g &= bcea \end{aligned}$$

   (b) SUM = 3: $|\sigma(f) \cup \sigma(g)| \leq 6$. Without loss of generality, we assume that $Q_1 \in \sigma(f) \cap \sigma(g)$ and $Q_2 \in \sigma(f) - \sigma(g)$. We set $\gamma(QX) = Q_1, \gamma(QY) = Q_2, \pi_F(2) = \pi_G(2) = QX, \pi_F(3) = QY$. We then have the following $\pi - \gamma$ chart:

function	1	2	3	4
f	a	$QX(Q_1)$	$QY(Q_2)$	d
g	a	$QX(Q_1)$	c	e

The lists are:

$$\begin{aligned} L_C &= a\{d,e\} \\ L_f &= da \\ L_g &= eca \end{aligned}$$

(c) SUM = 4: Then $|\sigma(f) \cup \sigma(g)| \leq 6$. For the first time, we make use of the assumption that the flip-flop outputs are available and hence can be tied to logic pins. In fact, we tie Q_1 to pin 1 of both LUT_F and LUT_G. Then we tie Q_2 to pin 2 of LUT_F and pin 3 of LUT_G; i.e., $\gamma(a) = Q_1, \gamma(QY) = Q_2, \pi_F(2) = \pi_G(3) = QY$. We have the following $\pi - \gamma$ chart:

function	1	2	3	4
f	$a(Q_1)$	$QY(Q_2)$	c	d
g	$a(Q_1)$	b	$QY(Q_2)$	e

The lists are:

$$\begin{aligned} L_C &= \{b,c\}\{d,e\} \\ L_f &= dc \\ L_g &= eb \end{aligned}$$

■

Remarks:

1. If neither of the flip-flop outputs are available, only condition 2 (c) of Proposition 6.2.3 needs to be changed. It becomes "if $N = 2$, $|\sigma(f) \cup \sigma(g)| \leq (9 - SUM)$."

2. If one flip-flop output is available and the other is not, condition B of Proposition 6.2.3 remains unchanged.

3. The preceding characterizations use phrases like *"given that the flip-flop outputs are available"*. What if only one flip-flop is to be placed inside the CLB? In that case, $N \leq 1$. Then, only the conditions 2 (a) and 2 (b) apply. From the preceding remarks, these conditions remain same for $N \leq 1$, irrespective of the availability of the flip-flop output.

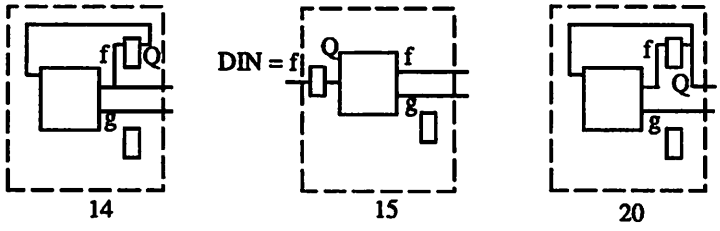


Figure 6.20: An example of match generation

4. After a match has been selected for the final implementation, the correspondence between the LUT pins and the function inputs is needed to determine the netlist. A by-product of the proofs of the above characterizations is a fast algorithm for determining this correspondence.

We identify all the patterns that can be matched against the (f, g, Q_1, Q_2) combination, and create the corresponding matches. If no patterns match, this combination is rejected. We repeat this process for all combinational function pairs f and g in the network $\tilde{\eta}$.

We now illustrate the match generation process.

Example 6.2.4 *Let f and g be two functions in $\tilde{\eta}$, and let f^L denote the latched value of f. Let*

$$\begin{aligned} f &= abcQ \\ g &= c + d + e + Q \\ Q &= f^L \end{aligned}$$

Let us consider (f, g, Q, NIL) as a candidate for the matches. Note that $|\sigma(f)| \leq 4$ and $|\sigma(g)| \leq 4$, so the condition B 1 of Proposition 6.2.3 is satisfied. In Proposition 6.2.3, $N = 1$. Since $|\sigma(f) \cup \sigma(g)| = |\{a, b, c, d, e, Q\}| = 6$, (f, g, Q, NIL) satisfies $2(b)$ and hence the sequential mergeability conditions. From remark 3 it follows that the same characterization holds irrespective of whether the flip-flop output is available. Next, we identify all possible patterns that match against (f, g, Q, NIL). Only the patterns that use F, G, and exactly one flip-flop need be considered. It turns out that three matches are possible, as shown in Figure 6.20. On the other hand, if (f, g, NIL, NIL) is considered as a candidate, no matches are generated. This is because $N = 0$, and from Proposition 6.2.3 it follows that to satisfy the mergeability conditions, $|\sigma(f) \cup \sigma(g)|$ must be at most 5.

Selecting a Minimum Subset of Matches

Having generated all the matches, we are interested in selecting a minimum subset $\mathcal{S}$ of these, because each match corresponds to a CLB. $\mathcal{S}$ should satisfy three types of constraints:

1. *Covering constraints*: Each node of the network, i.e., combinational or sequential (flip-flop) node, should be covered by some match in $\mathcal{S}$. A node is covered by a match if it is mapped to either F, G, QX, or QY of the CLB.

2. *Implication constraints*: If a match is in $\mathcal{S}$, each external input to the match (i.e., the input of a combinational function or a flip-flop of the match that is connected to some pin a, b, c, d, e, and DIN) should be either a primary input or an output of some match in $\mathcal{S}$.

3. *Output constraints*: Each primary output of the network should be an output of some match in $\mathcal{S}$.

As mentioned in Section 3.4.1, the covering constraints are subsumed by the implication and the output constraints, and strictly speaking, are not needed. However, the heuristics tend to work better if these constraints are explicitly added in the formulation.

This is once again a binate covering problem and we use algorithms of Section 3.4.1. If the covering problem is solved exactly, we would solve the mapping problem optimally *for the given k-feasible network* $\tilde{\eta}$ (and not for the functionality represented by $\tilde{\eta}$).

Proposition 6.2.5 *Given a feasible sequential network* $\tilde{\eta}$*, the procedure map_together gives the optimum solution to the technology mapping problem for Xilinx 3090 architecture,*[7] *provided it employs a complete set of patterns (say of Figure 6.6) and an exact algorithm to solve the binate covering formulation.*

6.2.2 *map_separate*

Though in theory the optimum solution can be computed by *map_together*, in practice it is a computationally infeasible exercise. In addition, the number of matches and therefore the size of the binate covering matrix $\mathcal{B}$ is so large (see Table 6.1) that even the heuristics take a long time. One way of speeding things up is by introducing a shift in the paradigm. Instead of taking a global view of the network, in which the combinational and sequential elements are considered simultaneously, we view the network to be composed of combinational and sequential elements.

[7] without the capability of option 1 (c) of Section 6.1.1

First we map the combinational logic nodes onto the CLBs and then attempt the best placement for the flip-flops. This approach is the most natural extension of mis-fpga to sequential mapping.

Given a k-optimal network $\tilde{\eta}$, the algorithm works as follows.

1. It first maps combinational logic onto CLBs. This is done by running *merge* on the combinational subnetwork of $\tilde{\eta}$ (this subnetwork is obtained by ignoring the flip-flops). As explained in Section 3.7.1, this identifies the maximum number of pairs of functions that satisfy the combinational mergeability conditions (CMCs) and assigns each pair to a different CLB. Each unpaired function is assigned to a new CLB.

2. It then assigns the flip-flops of the sequential network to CLBs. An attempt is made to assign maximum number of flip-flops to the CLBs used in the first step, so that minimum number of new CLBs is used. Each flip-flop is assigned to exactly one CLB. Two methods are used to solve the assignment problem. One is based on a network flow formulation and the other is a greedy heuristic. We describe them next.

A Network Flow Formulation

As Figure 6.6 shows, there are many possible ways in which a flip-flop can be used inside a CLB. In this discussion, we will consider only a proper subset of these ways (or patterns). We will show that using this subset, the optimum assignment of flip-flops to CLBs can be obtained in polynomial time using network flow. We impose that a flip-flop is assigned to at most one CLB.

First we define the set of patterns we will allow ourselves. Let C be the set of CLBs in use after assigning combinational functions (i.e., after step 1 of *map_separate*). Any CLB $j \in C$ uses only the combinational function(s) and has both flip-flops unassigned. Given a flip-flop i of the network and a CLB $j \in C$, we consider the following patterns:

- *Type 1*: It exists if and only if j has a combinational block f that is connected to the input of i and f is a single-fanout node. Then, i and j form a pattern as shown in Figure 6.21 (1). Note that g may or may not be present.

- *Type 2*: It exists if and only if j has a combinational block f to which i is fanning in and i fans out to node(s) in j only. Then i and j form a pattern, as shown in Figure 6.21 (2). Once again, g may or may not be present.

- *Type 3*: It exists if and only if j has a single combinational function. Then i and j form a pattern as shown in Figure 6.21 (3).

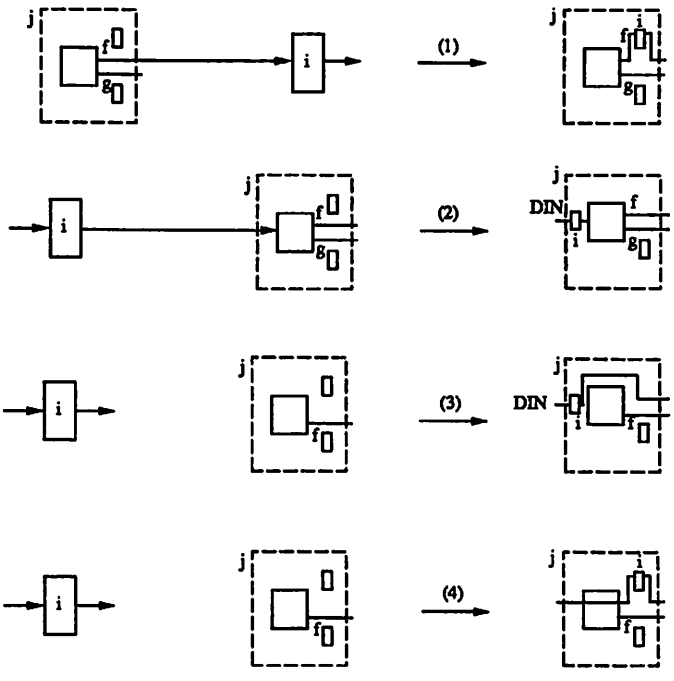


Figure 6.21: Patterns used in the maxflow formulation

- *Type 4*: It exists if and only if j has a single combinational function f such that number of inputs of f is at most 4. Then i and j form a pattern as shown in Figure 6.21 (4). Here i is placed on j using the unused combinational function of the LUT as a buffer. Note that in the type 3 pattern, DIN was used to place i on the CLB.

Definition 6.2.3 *A flip-flop i of the network and a* CLB $j \in C$ *are* **associated** *by pattern k, $1 \leq k \leq 4$, if i can be placed on j using a pattern of type k.*

The above types define how a single flip-flop can be placed on a CLB. However, each CLB can accommodate two flip-flops. This is handled by taking combinations of the above pattern types for the two flip-flops. For example, two flip-flops i_1 and i_2 may be assigned to the same CLB using pattern types 1 and 2 respectively. However, not all combinations are allowed. For example, two flip-flops i_1 and i_2 cannot be assigned to the same CLB using pattern types 2 and 3, since it implies an overloading of DIN. We now give a complete list of the constraints associated with the problem:

1. Each flip-flop is assigned to at most one CLB in C.

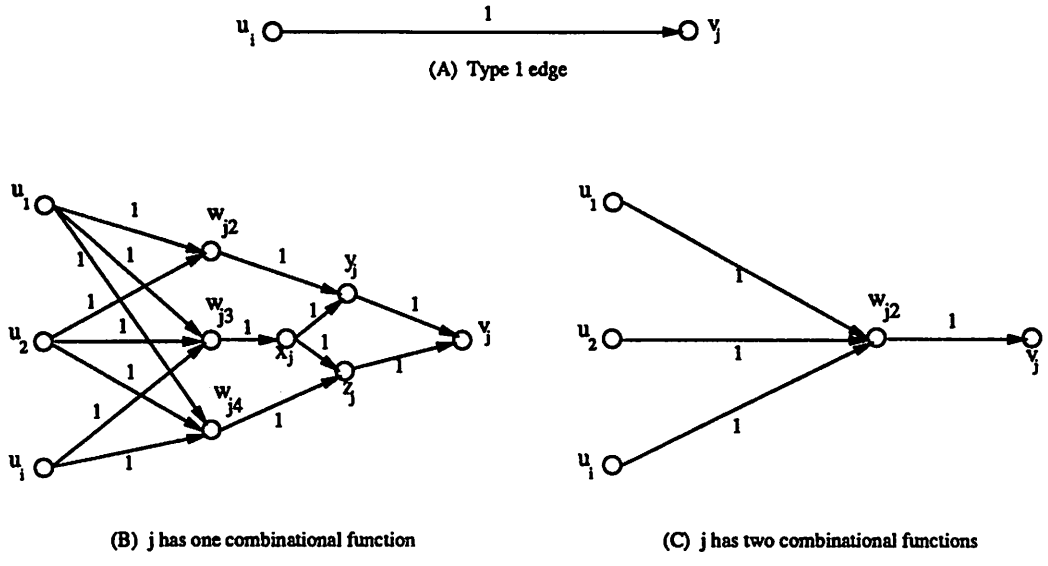


Figure 6.22: Constructing maxflow network

2. Each CLB can accommodate at most 2 flip-flops.

3. For a CLB $j \in C$, there can be at most one assignment of type k for each $k \in \{2,3,4\}$. Two assignments of type 2 or of type 3 imply an overloading of DIN, whereas two assignments of type 4 imply use of three outputs (j was already using one output).

4. A CLB $j \in C$ cannot get assigned to two flip-flops if one assignment is of type 2 and the other of type 3 (otherwise DIN will be overloaded).

5. Similarly, j cannot get assigned to two flip-flops if one assignment is of type 3 and the other of type 4 (otherwise j will use 3 outputs).

The problem is *to assign the maximum number of flip-flops of the sequential network to the* CLB*s in C using only the four pattern types subject to the above constraints.* We show how to solve this problem exactly using maximum flow or *maxflow*. We construct a maxflow network (a directed graph) as follows: there is a vertex u_i for each flip-flop i and a vertex v_j for each CLB $j \in C$. For each i and j that are associated with a match of type 1, there is an edge from u_i to v_j (Figure 6.22 (A)). For a CLB j, let I_{j2}, I_{j3}, and I_{j4} be the sets of flip-flops associated with j corresponding to match types 2, 3, and 4 respectively. Add vertices $w_{j2}, w_{j3}, w_{j4}, x_j, y_j$, and z_j. For each $k \in \{2,3,4\}$, add an edge (u_i, w_{jk}) if flip-flop $i \in I_{jk}$. Add edges $(w_{j2}, y_j), (w_{j3}, x_j), (w_{j4}, z_j), (x_j, y_j), (x_j, z_j), (y_j, v_j)$, and (z_j, v_j). This is shown in Figure 6.22 (B). If j contains two combinational functions, the

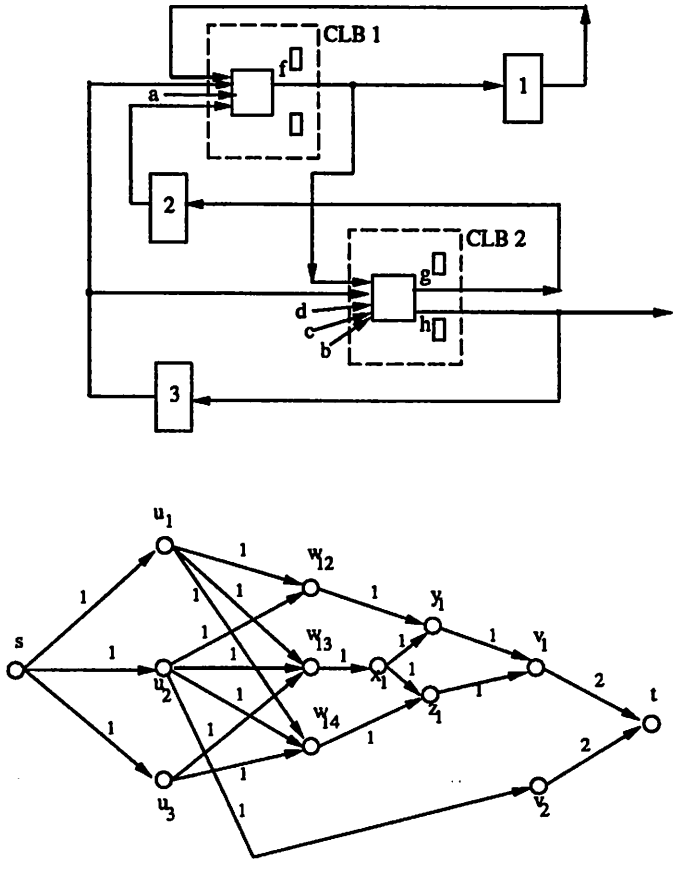


Figure 6.23: Maxflow network for a simple circuit

structure reduces to Figure 6.22 (C). This is because edges of type 3 and 4 exist only if CLB j has one combinational function. Finally, add two distinguished vertices s and t, with an edge from s to each u_i and from each v_j to t. All the edges of the network have a capacity of 1 each, except edges (v_j, t), which have a capacity of 2.

A capacity of 1 on (s, u_i) corresponds to the condition that flip-flop i cannot be assigned to more than one CLB, and a capacity of 2 on (v_j, t) restricts the maximum number of flip-flop assignments to a CLB to two. The structure of Figure 6.22 (B) corresponds exactly to the constraints 3, 4, and 5. Vertices w_{j2}, w_{j3}, and w_{j4} (along with the capacities of 1 on edges fanning out of them) serve to satisfy constraint 3, x_j and y_j - constraint 4, and x_j and z_j - constraint 5. Note that x_j is introduced since type 3 association does not go with either type 2 or type 4.

Example 6.2.5 *We illustrate the above construction for a simple circuit in Figure 6.23. In this example, there are 4 primary inputs $a, b, c,$ and d, and one primary output h.* CLB*s 1 and 2*

implement the three combinational functions f, g, and h. Flip-flops 1, 2, and 3 are to be assigned to the CLBs*. Note that no match of type 1 exists for* CLB *1, since f is a multi-fanout node: it fans out to flip-flop 1 and* CLB *2. Recall from the definition of a type 1 pattern that the combinational function of the* CLB *feeding the flip-flop should be a single-fanout node. Similarly a match of type 2 between flip-flop 3 and* CLB *1 does not exist, since flip-flop 3 fans out to* CLB *2 as well. All other possible matches exist for* CLB *1. For* CLB *2, type 1 match exists with flip-flop 2. Type 2 match cannot exist for fanout reasons. Also, since* CLB *2 has two functions g and h, it cannot have any type 3 or type 4 match. The resulting flow network is shown.*

After running a maxflow algorithm, if there is a unit flow on the edge (s, u_i), flip-flop i has been assigned to a CLB. This CLB is determined by traversing the unique path from u_i to some v_j such that there is a unit flow on each edge of the path. Hence the maximum flow corresponds to the number of assigned flip-flops. The unassigned flip-flops are placed in additional CLBs - two in each CLB.

For the circuit of Figure 6.23, we get a maxflow of 3. The flip-flop 2 gets assigned to CLB 2, and flip-flops 1 and 3 to CLB 1.

The running time of the algorithm is $O(n^3)$ where n is the total number of combinational and sequential elements in the sequential network.

Remarks:

1. Sometimes it may be beneficial to replicate flip-flops, i.e., assign one flip-flop to more than one CLB. We do not know how to solve the problem in polynomial time if the condition of unique assignment of a flip-flop is relaxed and replication of flip-flops is allowed.

2. We do not know how to obtain a network flow formulation with a set of patterns larger than the one being used currently.

3. Although there are just 4 pattern-types corresponding to assigning *one* flip-flop to a CLB, an assignment of two flip-flops to a CLB results in a larger pattern set.

A Greedy Assignment

In this method, flip-flops are assigned to CLBs one by one. For a flip-flop FF, with input D and output Q, we consider two cases:

- If there is no combinational block with output D (i.e., either a primary input or a flip-flop directly feeds FF), search for a CLB with exactly one output unused. If found, map FF

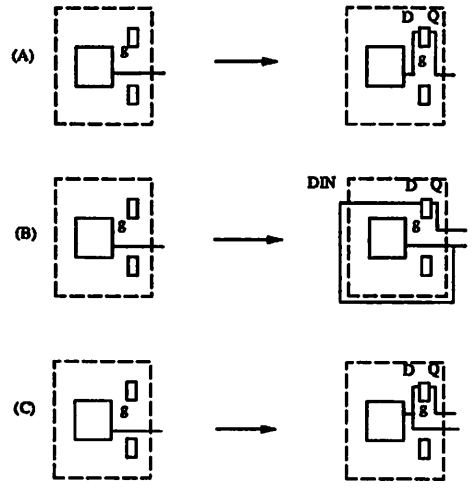


Figure 6.24: Mapping flip-flops in the greedy heuristic for *map_separate*

using the DIN input of the CLB. Otherwise, assign a fresh CLB to FF using one of the LUT-section functions as a buffer.

- Otherwise, there is a combinational block g in the network, feeding D. Let this block be mapped to a CLB C in step 1. We then consider the following two cases:
  - g is not being used elsewhere in the circuit (i.e., g has a single fanout - to FF): at least one flip-flop in the CLB C is unused. Use this flip-flop for FF (Figure 6.24 A).
  - Otherwise, g fans out to other nodes (and hence should remain an output of the CLB C). If only one output of C is being used (for g), tie DIN input of the CLB C to g and map FF to one of the flip-flops (Figure 6.24 B). Another option for this case is to route g as shown in Figure 6.24 C. Otherwise (i.e., if both outputs of C are being used), search for a CLB with one output unused. If found, map FF using the DIN input of the CLB. Otherwise, use a fresh CLB for FF.

6.2.3 Comparing *map_together* and *map_separate*

1. *Pattern set*: *map_together* uses a complete set of patterns whereas the *map_separate* formulations, both the network flow and the greedy, use a smaller set. It may be possible to incorporate an enlarged set of patterns in the greedy heuristic. However, it is not clear how to obtain a network flow formulation with a set larger than the one being used currently.

Each pattern type in the flow formulation is restrictive. For example, the pattern type 1 in Figure 6.21 restricts the number of fanouts of f to 1. The patterns used in *map_together* impose no such restriction.

2. *Feedback inside the* CLB: *map_separate* does not exploit the feedback inside the CLB. It ignores QX and QY inputs to the LUT-section. Consequently it is incapable of realizing a pair of functions with more than 5 inputs with one CLB. *map_together* exploits this feedback.

3. *Logic replication*: *map_together* allows replication of combinational functions and flip-flops, whereas *map_separate* does not allow any replication. Hence the solution space is larger for *map_together*.

4. *Quality*: If the binate covering problem is solved exactly, *map_together* gives an optimum solution. However, the heuristics for the problem are not known to come close to the optimum. Moreover, the heuristics are generic, in the sense that they do not exploit the features of the architecture. On the contrary, the *map_separate* approach decomposes the mapping problem for the sequential network into two sub-problems - combinational pairing and assignment of flip-flops. It does reasonably well on both of them.

6.3 Experimental Results

The algorithms presented above have been incorporated inside sis [77] and are named sis-fpga. The experimental set-up is as follows. We use finite state machine benchmarks in the *kiss* format obtained from MCNC [90]. We apply state-assignment using jedi [49] on these symbolic machines, except on *s349*, *s382*, *s444*, *s526*, *s641*, and *s838*, for which we did not have *kiss* description and had to start with the available encoded networks. The resulting sequential network is then optimized for minimum number of literals by standard methods [77] - we ran *script.rugged* twice with a timeout limit of 1000 seconds for each run. The optimized network is used as the starting point for both sis-fpga and XNFOPT. In jedi, we chose minimum-length encoding over one-hot, since the networks obtained after one-hot encoding and subsequent optimization and mapping give poorer results (see Section 6.3.1).

For XNFOPT, the number of passes is set to 10.[8] For sis-fpga, first k-optimal networks ($k = 4, 5$) are obtained using the algorithm of Section 3.5. The script used had three steps:

[8]The benchmark *keyb* had to be interrupted after 9 passes, since XNFOPT was taking too long.

example	FFs	sis-fpga (4-optimal)				sis-fpga (5-optimal)				XOPT
		comb.	tog.		sep.	comb.	tog.		sep.	
		nodes	*matches*	CLBs	CLBs	nodes	*matches*	CLBs	CLBs	CLBs
bbara	4	18	551	**11**	12	15	235	13	12	**11**
bbsse	4	37	1206	**21**	26	29	344	25	25	25
bbtas	3	7	251	**6**	5	5	60	**4**	**4**	5
beecount	3	12	310	8	7	9	126	8	**7**	**7**
dk14	3	24	447	**16**	**16**	19	198	**16**	17	18
dk15	2	17	209	11	12	7	30	**7**	**7**	9
dk16	5	82	4019	**47**	60	49	438	**47**	48	66
dk17	3	14	254	9	10	6	54	**6**	**6**	8
dk27	3	5	198	5	**3**	5	198	5	**3**	**3**
dk512	4	16	983	10	8	7	90	**7**	**7**	**7**
ex1	5	78	3527	**46**	52	65	1741	49	48	57
ex2	5	42	1908	**24**	29	29	252	30	29	30
ex3	4	18	526	11	12	11	119	**10**	11	**10**
ex4	4	19	780	12	11	16	553	12	11	**10**
ex5	4	16	586	11	10	11	97	11	10	**8**
ex6	3	25	746	15	**14**	21	274	16	15	20
ex7	4	16	520	**10**	**10**	10	131	**10**	**10**	16
keyb	5	68	2635	**40**	46	54	736	47	45	54
kirkman	5	53	2717	**34**	36	43	972	35	**33**	37
lion	2	3	53	**2**	**2**	3	26	**2**	**2**	**2**
lion9	4	5	146	5	**4**	5	87	6	**4**	6
planet	6	186	-	-	136	144	3884	125	**123**	157
s1	5	75	3270	**46**	54	55	534	51	50	73
s1a	5	67	2855	**38**	43	57	1101	49	47	54
s349	15	50	5690	32	**26**	41	2069	35	29	35
s382	21	49	-	-	**27**	37	3923	37	28	31
s386	5	40	1384	**23**	25	30	298	26	26	29
s420	5	26	1572	16	**15**	17	257	16	**15**	**15**
s444	21	44	6196	32	**26**	33	3113	37	27	27
s510	6	109	-	-	74	84	1972	**73**	74	77
s526	21	58	-	-	**32**	43	3826	46	34	35
s641	17	64	-	-	**36**	59	-	-	37	46
s8	3	10	219	**7**	8	8	93	**7**	**7**	8
s820	5	95	5032	**57**	62	77	1494	68	64	80
s832	5	92	4239	67	**59**	74	1724	61	**59**	72
s838	32	86	-	-	49	59	-	-	**48**	55
sand	5	167	-	-	123	133	3081	114	**111**	138
shiftreg	3	3	72	4	**3**	3	72	4	**3**	**3**
styr	5	142	-	-	104	113	2765	98	**92**	110
tbk	6	68	2161	50	49	55	1101	49	**44**	60
train4	2	3	53	2	2	3	47	2	2	2
total					1328				1274	1516
subtotal1					1243			1264	1189	1415
subtotal2				718	747			771	727	867

Table 6.1: Results for Xilinx 3090 architecture

example	FFs	sis-fpga (4-optimal)				sis-fpga (5-optimal)				XOPT
		comb.	tog.		sep.	comb.	tog.		sep.	
		nodes	*matches*	CLBs	CLBs	nodes	*matches*	CLBs	CLBs	CLBs
total					1328				1274	1516
subtotal1					1243			1264	1189	1415
subtotal2				718	747			771	727	867

Table 6.2: Summary of results

FFs	number of flip-flops
tog.	using *map_together*
sep.	using *map_separate*
comb. nodes	the number of feasible functions in the k-optimal network $\tilde{\eta}$
matches	the number of matches generated in *map_together*
CLBs	the number of CLBs in the final implementation
XOPT	using XNFOPT
-	spaceout or timeout
total:	total number of CLBs for all 41 examples
subtotal1:	total number of CLBs for examples where *map_together (5-optimal)* could finish
subtotal2:	total number of CLBs for examples where *map_together (4-optimal)* could finish

1. *Partial collapse*, which includes an initial mapping.
2. If the network is small, collapse it and use Roth-Karp decomposition and cofactoring to search for a better solution.
3. *Partition*, with support reduction embedded.

On the k-optimal network, both *map_together* and *map_separate* algorithms are run. We used the two-phase binate covering heuristic described in Section 3.4.1 to solve the binate covering problem in *map_together*. We used both network flow and the greedy heuristic for *map_separate* (on these benchmarks, both produce the same results).

Table 6.1 reports the results for sis-fpga and XNFOPT. A summary of the results, along with the meaning of the columns, is given in Table 6.2. A "-" indicates that the program ran out of memory or exceeded a time-limit of 1 hour. The minimum CLB count for each example is highlighted. The *map_separate* (4-optimal and 5-optimal) and XNFOPT algorithms finished on all benchmarks. However, *map_together* could not (for both 4-optimal and 5-optimal cases). In particular, the benchmarks on which 4-optimal finished were a subset of those on which 5-optimal finished. The row *total* is for the complete benchmark set and hence gives results only for *map_separate*

(4-optimal), *map_separate (5-optimal)*, and XNFOPT. The row *subtotal1* gives the subtotals for those examples on which *map_together (5-optimal)* finished. It compares *map_separate (4-optimal)*, *map_separate (5-optimal)*, *map_together (5-optimal)* and XNFOPT. Similarly, the row *subtotal2* gives the subtotals for those examples on which *map_together (4-optimal)* finished. It compares *map_separate (4-optimal)*, *map_separate (5-optimal)*, *map_together (4-optimal)*, *map_together (5-optimal)*, and XNFOPT.

On the complete set (row *total*), *map_separate (5-optimal)* gives the best results. It is 16% better than XNFOPT. However, on examples corresponding to the row *subtotal2*, *map_together (4-optimal)* is better than other methods, being slightly better than *map_separate (5-optimal)* and 17% better than XNFOPT. This improvement is partially due to a better combinational synthesis done by mis-fpga and partially due to the techniques specific to the sequential synthesis. For instance, we noticed that on *dk16*, just for the combinational subnetwork (5-optimal case), mis-fpga uses 48 CLBs and XNFOPT 58. When the sequential network is considered as a whole, *map_separate* accommodates all the flip-flops in the 48 CLBs, whereas XNFOPT uses 66 CLBs. Also, in examples like *bbsse*, *ex1*, *ex2*, *dk16*, *keyb*, *s1*, *s1a*, and *s386*, *map_together* achieves lower CLB counts than all other techniques. Therefore, for these examples, mapping the combinational logic and flip-flops at the same time helps.

We see from the table that the number of matches for *map_together* depends heavily on k. The number of matches for 4-optimal case is much higher than 5-optimal (and that is why on many examples, it runs out of memory). It generally implies that the number of CLBs used is less. This is despite the fact that with $k = 4$ option, the full potential of the architecture is not being tapped, since the network does not have any functions with 5 inputs.

Comparing 4-optimal and 5-optimal cases for *map_separate*, the results are mixed. In the 5-optimal case, there are fewer nodes to start with, but the possibilities of pairing are also lower. Overall, starting from 5-optimal networks gives 4% better CLB count than starting from 4-optimal networks.

Finally, comparing the *map_together* and *map_separate*, it is seen that the results obtained from *map_separate (5-optimal)* are better than those from *map_together (5-optimal)*. But results from *map_separate (4-optimal)* are worse than those from *map_together (4-optimal)*. So no one technique is a clear winner. In principle, if we use an exact binate covering method, *map_together* will be at least as good as *map_separate*. This is because there are matches that *map_together* will detect but *map_separate* will not. For example, *map_together* can place some function pairs with a total of 6 or 7 inputs in one CLB. This is beyond the capabilities of *map_separate*. However, since

we are using a generic heuristic to solve the binate covering problem in *map_together*, the results are mixed.

6.3.1 Different Encoding Schemes

We study the effect of two different schemes: *one-hot* and *minimum length*. On the symbolic FSM description, we used jedi to assign codes either in a one-hot mode (using the *-eh* option) or the minimum length mode (default). Then, the experiment was conducted exactly the same way as earlier, except that *map_separate* was used to finally cluster the elements of the k-optimal network, $k = 4, 5$. The results are shown in Table 6.3. Note that many large state machines used in Table 6.1 are missing. This is because we did not have symbolic *kiss* descriptions of the missing machines. It turns out that on the benchmark set of Table 6.3, the minimum length encoding scheme gives better results than one-hot. This is in contrast to the results obtained by Schlag *et al.* [75]. Their conclusion was that a one-hot encoding scheme is better than others for LUT architectures. The anomaly may be due to different encoding techniques - they used MUSTANG [19], different optimization scripts, and different mapping algorithms. In fact, in their table of results, we observed that the encoding scheme that yields minimum number of literals also yields minimum number of CLBs. This is expected since most LUT mapping techniques work on the representation generated by the optimization phase and group the literals or cubes subject to the fanin constraint. In their experiments, it so happens that one-hot gives minimum number of literals in most of the cases. A detailed study needs to be done to nail down the exact cause of the anomaly.

6.4 Discussion

We presented two methods for technology mapping of sequential circuits onto a popular LUT architecture. Both use a k-optimal network as the starting network. In the first, we consider combinational logic and flip-flops together for mapping. We formulate this as a binate covering problem. The second method uses a maximum cardinality matching algorithm to place as many function-pairs together on CLBs as possible; each remaining function is assigned to a fresh CLB. Flip-flops can then be assigned using either a network flow technique or a greedy heuristic. No single method does well on all the cases. However, both are better than XNFOPT, a commercial logic sythesis tool for LUTs. On the benchmark examples tested, we obtained average improvement of about 16% over XNFOPT using *map_separate*, and 17% using *map_together*. This improvement

example	4-optimal		5-optimal	
	one-hot	*min-length*	*one-hot*	*min-length*
bbara	14	12	17	12
bbsse	26	26	27	25
bbtas	5	5	5	4
beecount	14	7	16	7
dk14	26	16	28	17
dk15	19	12	20	7
dk16	55	60	56	48
dk17	16	10	17	6
dk27	6	3	6	3
dk512	16	8	17	7
ex1	51	52	51	48
ex2	30	29	31	29
ex3	13	12	13	11
ex4	9	11	13	11
ex5	14	10	13	10
ex6	17	14	19	15
ex7	11	10	13	10
keyb	41	46	42	45
kirkman	36	36	34	33
lion	3	2	4	2
lion9	8	4	10	4
s1	74	54	74	50
s386	28	25	28	26
s420	18	15	19	15
total	550	479	573	445

Table 6.3: One-hot encoding vs. minimum-length encoding

one-hot	use one-hot encoding in the state-assignment step
min-length	use minimum-length encoding of jedi in the state-assignment step
4-optimal	obtain 4-optimal network, then use *map_separate*
5-optimal	obtain 5-optimal network, then use *map_separate*
total	sum of Xilinx 3090 CLB counts over all examples

is partially due to a better combinational synthesis done by mis-fpga and partially due to the techniques specific to the sequential synthesis we have proposed.

One contribution of this work is a fast method of determining whether two functions, along with some flip-flops, can be placed on a single Xilinx 3090 CLB. The correspondence of the inputs of the functions with the pins of the CLB can also be found out quickly. Another contribution is proving that there exists a complete set of 19 patterns for the Xilinx 3090 CLB. Finally, we presented a polynomial-time algorithm to obtain the best assignment of flip-flops after combinational logic elements had already been placed. However, we restricted ourselves to a subset of patterns and allowed no replication of flip-flops.

Although specific details of the algorithm are worked out for the Xilinx 3090 architecture, the generic algorithms presented here can be tailored to other configurations of LUT architectures. The pattern set will change, and with it the heuristics used in *map_separate*. *map_together* will be the same except for the match generation step, which is pattern dependent. The binate covering formulation will be solved the same way.

An important issue not considered in this work is that of starting the machine from an initial state. This requires a proper setting or resetting of the individual flip-flops in the CLBs. In the Xilinx 3090 CLB, a flip-flop can be reset, but not set. Appropriate inverting logic has to be used for flip-flops that need to be initialized to 1.

This work is a first step towards synthesis of the sequential circuits for LUT architectures. We visualize it as the back-end of a complete synthesis system. The front end is a state-assignment program targeted for LUT architectures. A more thorough study of various state-assignment methods needs to be done. Of great interest is the problem of *balancing* the combinational and sequential components of the circuit. Any imbalance will cause more CLBs to be used than are necessary. We have the following approach in mind to solve this problem.

1. Do the state-assignment and optimization for minimum combinational logic, irrespective of the number of flip-flops used. Let us say that combinational logic uses CLBs distributed over c chips.

2. If the flip-flops can be assigned to the CLBs in the c chips, terminate. Otherwise, the circuit is *more sequential than combinational in nature*. Try to reduce the number of flip-flops. One way of doing so is by retiming the circuit (i.e., moving the latches across combinational logic) for minimum number of flip-flops, as proposed by Leiserson *et al.* [48]. Since retiming does not touch the combinational logic, the mapping of combinational logic is unaffected.

3. If the flip-flops still do not fit, re-encode parts of the FSM such that combinational logic is traded for sequential logic.

4. When all else fails, use extra chip(s) to accommodate the flip-flops.

Another important area is that of performance optimization for sequential circuits. One effort in this direction was by Touati *et al.* [83], who applied retiming on a circuit mapped on to 3090 CLBs to improve the circuit delay. However, due to lack of room for moving latches in a mapped circuit, the technique did not give encouraging results.

Chapter 7

Performance Directed Synthesis

7.1 Introduction

Figure 7.1 shows a typical section of an LUT architecture. The interconnections to realize the circuit are programmed using scarce wiring resources provided on the chip. There are three kinds of interconnect resources:

1. *long lines*: run across the chip; mainly used for clocks and global signals.
2. *direct interconnect*: connects the output of a CLB to an input of the adjacent CLB.
3. *general purpose interconnect*: consists of a grid of horizontal and vertical metal segments running along the edge of each block. Switching matrices join the ends of these segments. This interconnection is programmed using pass transistors. A typical block delay in the Xilinx case is 15 ns. Wiring delay could vary from 2 to 80 ns. The main constraints from the synthesis viewpoint are:
   (a) a maximum number of inputs to a CLB,
   (b) limited wiring resources, and
   (c) a limited number of CLBs on a chip (e.g. a Xilinx chip may have around 320 CLBs).

Most of the work on synthesis for FPGAs mainly focussed on minimizing the number of blocks needed to implement a circuit. In this chapter, we address the problem of delay optimization for FPGAs.

The chapter is organized as follows. The development of performance optimization for LUT architectures is presented in Section 7.2. The terms used in the chapter are defined in Section

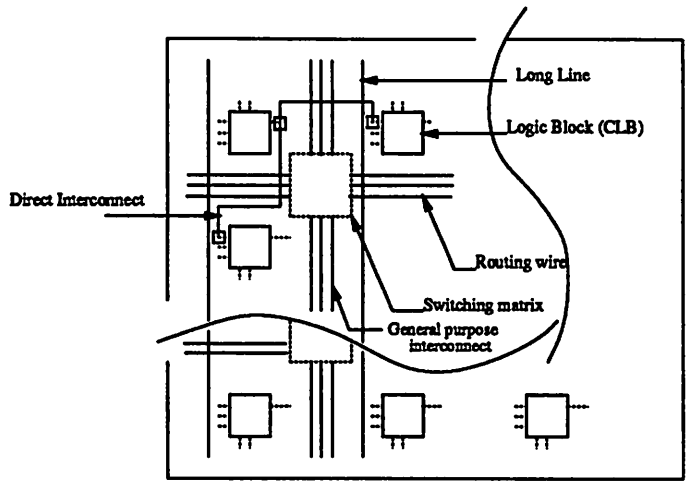


Figure 7.1: Interconnection structure in an LUT architecture

7.3. Section 7.4 describes our two-phase approach. Results and comparisons on benchmarks are presented in Section 7.5. Finally, some conclusions are drawn in Section 7.6.

7.2 History

Most of the existing logic synthesis techniques for delay optimization are geared towards designs without dominant wiring delays ([81, 82]). Consequently, these techniques either roughly approximate or ignore wiring delay. In the case of FPGAs, a significant portion of the delay of a path from an input to an output can be due to wiring delays. Recently, Pedram and Bhat showed one way of taking into account the wiring delays [66] while mapping a design into a predefined library of cells.

7.2.1 chortle-d

In 1991, Francis *et al.* proposed chortle-d [27], which is level-reducing algorithm for the LUT architectures. Like chortle-crf, it is based on the concept of bin-packing. It generates reasonably good solutions in terms of the number of levels of LUTs. But it suffers from two drawbacks. First, the number of levels alone may not be a good objective function for the minimum delay. The wiring delays can be unpredictable and can cause the total delay of the placed and routed implementation to vary significantly, even if the number of levels is kept the same. Second, it achieves the level-reduction at the expense of extra logic. This typically introduces extra routing

complexity, leading to additional wiring delays.

7.2.2 mis-fpga (delay)

At the same time as chortle-d, we proposed our performance optimizer [64]. It does not suffer from the above drawbacks. We solve the problem by a two-phase approach: first, we apply transformations at the logic level using an approximate delay model and then, couple timing-driven placement with resynthesis using a more accurate delay model.

7.2.3 DAG-Map

In 1992, Cong and Ding presented DAG-Map [16], a delay minimization algorithm, which like chortle-d, addressed only the problem of minimizing the number of levels at the logic level, given an m-feasible network. The significance of this algorithm is that it generates the exact minimum number of levels, given the original structure of the m-feasible network, and the only transformation allowed is node collapsing (this is the delay analogue of the covering problem for minimum block count, Problem 3.4.1).

7.2.4 TechMap-L

In 1992, Sawkar and Thomas [74] proposed a mapping approach for delay optimization, in which the delay-critical sections of the area optimized network are remapped using clique partitioning.

7.3 Definitions

Definition 7.3.1 *A* **path** $i \xrightarrow{p} j$ *is a sequence of alternating nodes and edges in the graph (network), starting at the node* i *and ending at the node* j.

We assume that each node of the network has some delay associated with it. Also, **arrival times** a_i at each primary input i and **required times** r_j at each primary output j are given, indicating the times at which signal becomes available at the inputs and by which it is required at the output. These times are derived from the performance constraints on the design. They are then used to compute the arrival time a_n and required time r_n at each node n in the network. If arrival times are not specified for some primary inputs, they are assumed to be zero. Similarly, required times

at primary outputs, if unspecified, are forced to be the maximum required time of an output. If no required times are specified, we set the required times of all the outputs to the maximum arrival time of an output. A **forward trace** (from inputs to outputs) of the network gives the arrival time at each node (i.e., at its output). Similarly, a **backward trace** of the network yields the required time at each node. We can extend the notion of arrival and required times for the edges of the network too as follows. If the delay through an edge (n, f^o), $f^o \in FO(n)$, is d, then the arrival time at the edge (n, f^o) is the $a_n + d$, and the required time at (n, f^o) is the difference between the required time at f^o and the delay through the node f^o. We compute the **slack** s_i at each node (edge) i as the difference between the required time r_i and the arrival time a_i at the node (edge), i.e., $s_i = r_i - a_i$. A node (edge) is **ϵ-critical** if its slack is within ϵ of the most negative slack in the network.

7.3.1 Problem Statement

Given a circuit, described in terms of Boolean equations, the arrival times at the inputs and the required times at the outputs, obtain an implementation on the target LUT *architecture that meets all the timing constraints with the least block count.*

7.4 Approach

We propose a two-phase approach:

*Placement-independent (*PI*) phase*: It involves transformations at the *logic level*, which are guided by an estimate of the final delay.

*Placement-dependent (*PD*) phase*: It does a synthesis-driven and performance-directed placement. Delay models that explicitly take into account the wiring delays are used.

7.4.1 Placement-Independent (PI) Phase

Since wiring delays are important, we may often prefer a trade-off between the number of levels and the number of nodes and edges.[1] In this phase, a standard delay reduction script is used to obtain a delay-optimized network. The resulting network is in terms of 2-input gates and hence is m-feasible (since $m \geq 2$). The remaining task is to reduce the number of levels in this network while maintaining feasibility and with a simultaneous control on the number of nodes and

[1] We use the number of edges in the network as a measure of routing complexity and hence wiring delay.

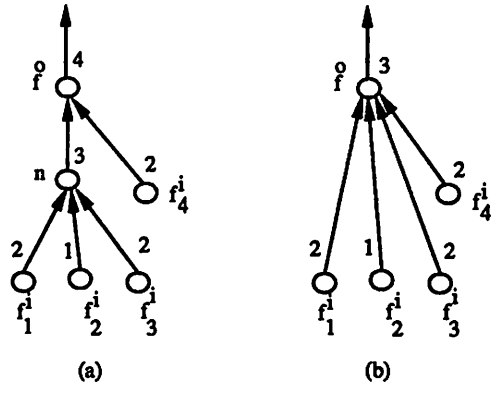


Figure 7.2: Reducing the level by collapsing

edges. As we shall see in Section 7.5, the following algorithm LUT_**reduce_depth** achieves the above objective.

LUT_*reduce_depth* first finds the critical nodes of the network η by a delay trace. The delay trace assigns levels to each node. The level of a primary input is 0, the level of any other node is one plus the maximum level of its fanins. This delay model is also called a **unit delay model**. Then it traverses η from inputs and tries to collapse each critical node n into a subset S of its fanouts. S is the set of those fanouts of n whose level is one higher than the level of n. If a fanout $f^o \in S$ remains feasible after collapse (Figure 7.2), or becomes infeasible but can be redecomposed (e.g., by cofactoring, as shown in Figure 7.3) resulting in reduction of the level of the fanout, then n is collapsed into it. In both these figures, m is assumed to be 5 and the number beside a node is its level. If after this first pass through S, there exists some fanout f^o into which n could not be collapsed, the algorithm tries to move some non-critical fanins of n as inputs of some other non-critical fanins of n. The condition under which this can be done can be derived from functional decomposition theory, e.g., that of Roth and Karp [36]. This fanin movement was described in Section 3.3.6. Similar fanin movement is tried at the fanout node f^o as well. Such transformations increase the likelihood of collapsing node n into f^o. The whole process is repeated until critical nodes can no longer be collapsed. A delay trace using the unit delay model is done after each collapse, so that the algorithm always operates on correct delay values. Let the resulting network be $\tilde{\eta}$.

If $\tilde{\eta}$ has a small number of primary inputs (say up to 10), we try two other decompositions. We first collapse $\tilde{\eta}$ into two levels. We then apply cofactoring and the Roth-Karp decomposition techniques and delay-evaluate the resulting decompositions. If either of these delay evaluations is

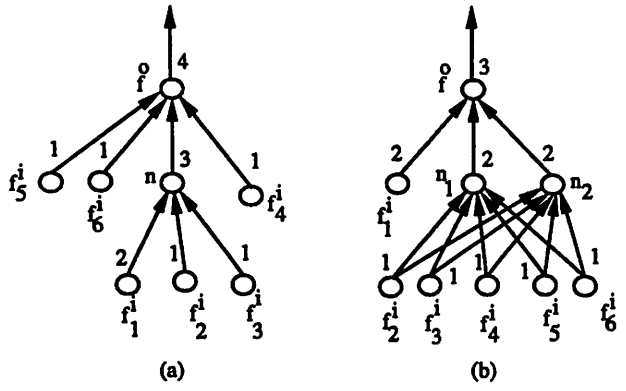


Figure 7.3: Reducing the level by collapsing and redecomposing

less than that of $\tilde{\eta}$, the best decomposition is accepted. This final check helps in reducing delay in many examples.

Since the covering step of the BCM phase (Section 3.4.1) reduces the number of nodes, and often edges, without increasing the number of levels in the network, the *partition* routine is called as the final step. It works by collapsing nodes into their fanouts if they remain feasible after collapsing. This takes care of the collapsing of non-critical nodes.

7.4.2 Placement-Dependent (PD) Phase

The starting network for this phase is the one generated by the PI phase. We combine the techniques of logic synthesis with a placement algorithm. We model the placement as assigning locations to point modules on a k by k grid (in the Xilinx 3000 series, k can take values from 8 to 18). We use a simulated annealing based algorithm for placement. At the end of the iterations at each temperature we identify *critical* sections that are ill-placed. We use logic synthesis and force-directed placement techniques to restructure and reposition these sections. The logic synthesis techniques used are *decomposition* and *partial collapse*. These techniques are local, i.e., they explore only the neighborhood of a critical section for a better solution. The algorithm is summarized by the pseudo-code in Figure 7.4.

We note that this approach can be incorporated in any iterative placement techniques, like the force directed methods, or resistive network methods. We chose simulated annealing because we wanted to model the simulated-annealing based place and route tool of Xilinx.

```
/* a = temp factor (a < 1); T = current temperature;
T_l = starting temperature for logic synthesis;
m = number of moves per temperature; */
{
     T = start_temp;
     while (T > final_temp) {
          j = 0;
          while (j < m) {
               get two random locations for swap;
               evaluate δc, change in cost;
               accept swap with probability e^(-δc/T);
               if swap accepted, do delay trace;
               j++;
          }
          if (T < T_l) do logic resynthesis and
                    replacement for delay;
          T = T * a;
     }
}
```

Figure 7.4: Simulated annealing for placement and resynthesis

Delay Models

The following assumptions are made:

1. There is no capacitance seen into an input pin of a block. This is as per the Xilinx manual [88].

2. We do not consider the non-linearities of the pass transistors used to connect two routing segments.

3. We ignore the output resistance of each block (simply because we do not know it). This is consistent with the way the Xilinx manual suggests the computation of the delay through each edge [88].

4. The delays from a primary input to a node and from a node to a primary output are ignored for want of proper pad placement.

The delay computation is done as follows. The delay through each block is a constant, called the **block delay**. To compute the wiring delay, we use two models: Elmore delay model and Rubinstein-Penfield-Horowitz delay model (RPH) [67].

1. *Elmore delay model*: We consider each edge of a net independently. In other words, there is no affect of other fanouts of n on the delay of the edge (n, f^o), $f^o \in FO(n)$. The model is simplistic but fast to compute. The delay d through the edge (n, f^o) is the time signal takes to reach from zero volts to v_t of its final value at the node f^o. The starting time is the time when the signal becomes available at the driver node n. The delay d is given by

$$d = RC \log \frac{1}{1 - v_t}$$

where R and C are the resistance and capacitance of the edge (n, f^o) respectively. We compute the RC of the edge by first computing the length l of the edge. We then compute RC by $l^2 \times rc\_pu\_len\_squared$, where $rc\_pu\_len\_squared$ is the factor used to convert from unit of length squared to unit of delay.

2. *Rubinstein-Penfield-Horowitz delay model (*RPH*)*: The net driven by the node n is treated as an RC-tree. We construct a minimum length spanning tree connecting node n and all its fanouts. Each edge in the spanning tree is modeled as a lumped RC-line. Two bounds are calculated for each edge (n, f^o): a lower bound and an upper bound. The formulae for

computation may be found in [67]. The upper bound is used as the delay of the edge. This model takes into account the loading effect due to other fanouts on a fanout f^o. A problem with this model is that it sometimes gives a huge difference between the lower and upper bounds for edges in the net, implying that some of the bounds are not tight. So, by taking the upper bound, we may be doing a pessimistic analysis. Also, it is computationally more expensive than the Elmore delay model.

Either model can be used in the delay computations. However, to make computations fast, we use the Elmore model for evaluating the effect of a transformation on delay. The transformations used are:

1. a swap of contents of two locations in the simulated annealing step,
2. logic synthesis operations on the critical sections.

The delay models require the constant $rc\_pu\_len\_squared$, called the *delay per unit length squared*. Section 7.5 describes how this constant is computed experimentally.

Cost Function

The change in cost function δc is computed as a weighted sum of the change in total interconnection length $\delta \ell$ and change in the delay δd through the network.

$$\delta c = (1 - \alpha(T))\, \delta \ell + \alpha(T)\, \delta d \tag{7.1}$$

$\alpha(T)$ is a temperature-dependent weight that increases as the temperature decreases. The reason for using total interconnection length while optimizing for delay is that we would like to get the blocks closer to each other before trying to optimize the delay. In our experience, this approach tends to give better results than when we just optimize for delay alone. The length of a net driven by a node n is estimated as the minimum length rectilinear spanning tree connecting the node n and all its fanouts. Computing $\delta \ell$ is then straightforward; it involves recomputing minimum length rectilinear spanning trees for the nets that are affected by the swap. Computation of δd is more involved and is described next.

Computing δd: We estimate δd, the change in delay of the network, when two nodes n_1 and n_2 are considered for swap as follows. We start the delay trace at the fanins of the node n_1. We assume that the arrival times of the fanins of n_1 remain the same.[2] We recompute the new

[2]This may not be true if n_2 is in the transitive fanin of n_1.

```
estimate_delay_change(n1) {
    change1 = LARGE;
    foreach_fanout(n1, f°) {
        diff = old_slack(n1, f°) - (min_slack(N) + ε);
        diff_arr = old_arr(n1, f°)-new_arr(n1, f°);
        change_delay = diff + diff_arr;
        if (diff > 0) && (change_delay > 0)
            continue; /*not critical*/
        change1 = min (change_delay, change1);
    }
    /*no fanout became critical*/
    if (change1 == LARGE) change1 = 0;
}
```

Figure 7.5: Estimating the delay change

delays through all edges (f^i, n_1) and (n_1, f^o), $f^i \in FI(n_1)$, $f^o \in FO(n_1)$. As stated earlier, we recompute these delays using Elmore delay model. We use the algorithm of Figure 7.5 to estimate the change in delay, `change1`, of the network $\mathcal{N}$ by placing node n_1 at the new position. Note that `diff` is the available delay before n_1 becomes ϵ-critical. We similarly compute `change2` for node n_2. To estimate the delay change δd through the network, we do a worst case analysis. If `change1` and `change2` are both negative, δd is estimated to be `-(change1 + change2)` (this case may happen if n_1 is in the transitive fanin of n_2 or vice-versa); else δd is estimated as `-min(change1, change2)`. By doing a worst case analysis, this approach tries to ensure that if the actual delay increases after swap, our estimate of δd is non-negative. It should be emphasized that we are trying to predict the changes in the delay through the entire network by just examining the neighborhood of the nodes n_1 and n_2.

If the swap is finally accepted, delays on some edges change, thereby changing the arrival times and required times of various nodes in the network. To determine the nets whose delays are affected, we handle the two delay models separately:

1. If the delay model is the Elmore delay model, the delays on all the edges of the type (n_j, f^o) or (f^i, n_j) need to be updated $(j = 1,2)$. Here $f^o \in FO(n_j)$ and $f^i \in FI(n_j)$.

2. If the delay model is RPH, we need to look at *nets* instead of *edges*. The nets affected are the ones that are driven either by node n_j or by a fanin of n_j $(j = 1,2)$.

Next, we need to update the arrival and required times of appropriate nodes. Let S be the set of edges whose delays were updated as a result of swap. Then the arrival times of edges in S and all the nodes and edges in the transitive fanout of an edge in S need to be updated. As for the required times, theoretically, only the nodes and edges in the transitive fanins of the edges in S need to have their required times updated. However, if none of the required times were specified for primary outputs initially, then as per our assumption, the required times of all the outputs were set to maximum arrival time of an output. As a result of updating the arrival times after swap, this maximum arrival time may have changed, changing in turn the required times of all the outputs. So a backward delay trace may need to be done on the entire network.

The analysis is suitably modified if there is no node at a location, i.e., the location is a vacancy.

Logic synthesis

After a set of critical nodes has been identified in the network, we use logic resynthesis to correct the structure of the network. Then we do a force directed placement of the resynthesized region. The logic operations are of two kinds, *decomposition* and *collapsing*. These operations change the structure of the mapped network while preserving the functions at the outputs, and can be interpreted as topological changes that yield a better placement. Decomposition introduces new nodes and new edges in the network. Collapsing may cause the number of edges to increase, decrease, or remain the same.

Decomposition: Since we start the placement with an m-feasible network, each node (block) has at most m inputs. We use the Roth-Karp decomposition technique to find a suitable decomposition [36]. In Roth-Karp decomposition, a bound set X is chosen from the fanins of the node n to be decomposed. The rest of the inputs of n form the free set Y. Then the decomposition of n is of the form: $f(X,Y) = g(\alpha_1(X), \alpha_2(X), \ldots, \alpha_t(X), Y)$, where $g, \alpha_1, \alpha_2, \ldots, \alpha_t$ are Boolean functions. As a result of the decomposition, we get a tree of logic blocks for the single block n that we started with. The new blocks are placed using a force directed technique. This has the effect of reducing routing congestion, ensuring, at the same time, that the signal at the output of

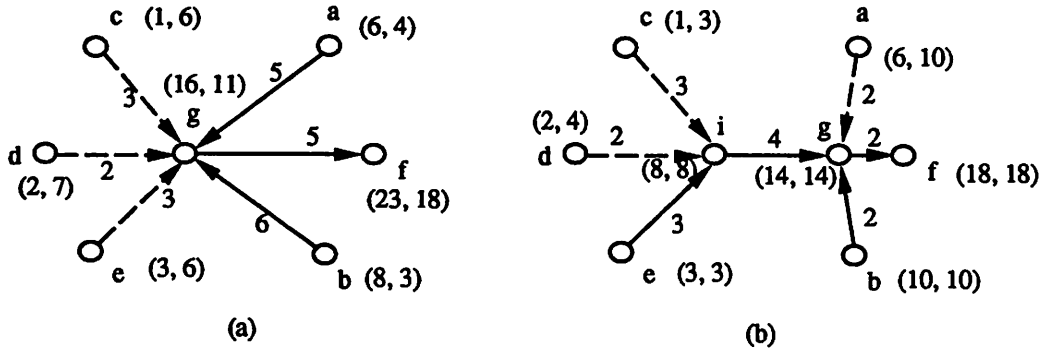


Figure 7.6: Decomposition example

the tree arrives no later than the original arrival time at the output of n. The motivation for this decomposition can be understood by the following simple example. Let $g = (a + b)\ c\ d\ e$ be a node in the feasible network. The initial physical placement is shown in Figure 7.6(a). The (arrival, required) times are shown as the bracketed pair next to each node. The critical paths are shown with bold lines and the remaining with dotted ones. The block delay is assumed to be 2 units. The delay along each edge is shown alongside it. Since the signals a and b arrive late, we should put them close to the output of the decomposition. Let $\{c, d, e\}$ be the bound set X and $\{a, b\}$ be the free set Y. We obtain a decomposition as $i = cde, g = i(a + b)$. Figure (b) shows the placement after decomposition. Note that the decomposition satisfies the constraint that the output of f arrives no later than 18 units of time. A simple move of node g from its initial location to its new location would have increased the lengths of paths $c \rightarrow g, d \rightarrow g, e \rightarrow g$. A swap between the node g and the vacancy at the new location of g could have been possibly considered in the simulated annealing step, but it may have been rejected if the wiring cost was high. Hence this placement is better for delay. At the same time, the routing penalty is kept low.

Placement of decomposition tree: Let N be the node to be decomposed (Figure 7.7). Let η denote the set of nodes replacing N after decomposition. We can look at η as a network with a single output driving the fanouts of N. The primary inputs of η, $PI(\eta)$, are the fanins of node N. We need to determine the locations of the intermediate nodes of η. The locations of the inputs of η and the fanouts of N are known. We start placing nodes of η from the inputs in such a way that when a node n is being placed, the locations of all of its fanins are known. So for n, we can do a force-directed placement with respect to the fanins of n, $FI(n)$, and the fanouts of N, $FO(N)$. For each edge in η, we find the delay that the signal can expend passing through it. This is obtained

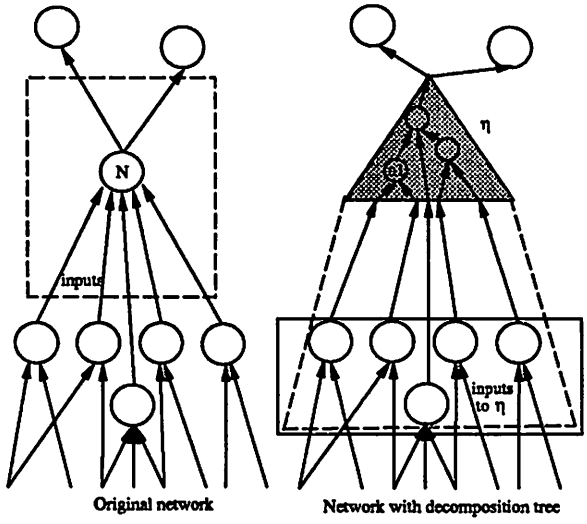


Figure 7.7: Placement of the decomposition tree

from the arrival times $a_i, i \in PI(\eta)$ and required times $r_j, j \in FO(N)$. Then, the node n is placed closer to those fanins f^i or fanouts f^o such that delay to be expended along (f^i, n) and (n, f^o) is small.

Recall that a_i is the arrival time, r_i the required time and s_i the slack at node i. Let p be a path from a primary input $i \in PI(\eta)$ of η to a node $j \in FO(N)$. Then we have $r_j - a_i$ units of time to distribute over all paths from i to j (there could be many such paths). We say an edge-path pair is **valid** if the edge lies on the path. We associate with each valid edge-path pair (e_{kl}, p), a value of $\frac{r_j - a_i}{|p|}$, where $|p|$ denotes the number of edges in the path p. For each edge e_{kl}, define its weight (w_{kl}) to be the minimum value over all paths that contain it.

$$w_{kl} = \min_{\forall p: e_{kl} \in p} \frac{r_j - a_i}{|p|} \tag{7.2}$$

The weight on each edge gives an estimate of the time that a signal can expend passing through it, thus bounding the length of the edge in the placed circuit.

Let $n \in \eta$ be the node whose location is to be determined. Construct an undirected graph $G_n(V_n, E_n)$ as follows. For every $k \in FI(n)$ there is a node $k_n \in V_n$. There is a node corresponding to n, say n_n. For each $j \in FO(N)$ there is a node $j_n \in V_n$. There are edges between n_n and all other vertices, denoted by $e^n_{n_n k_n}, k_n \in V_n, k_n \neq n_n$. Weigh each of the edges as follows. The weight of the edge joining n_n and $k_n (k \in FI(n))$, denoted by W^n_{kn}, is the same as the edge weight w_{kn} described in the above paragraph. The weight of the edge joining n_n and

$j_n (j_n \in FO(N))$ is approximated as,

$$W^n_{nj} = \min_{\forall p: i \xrightarrow{p} j} \left(\frac{|p_{nj}|}{|p|}(r_j - a_i)\right),$$

where p is a path from $i \in FI(n)$ to $j \in FO(N)$ in η, and node n lies on the path. Note that a_i is known since fanins of n have been placed. $|p_{nj}|$ is the number of edges along the path from n to j ($|p_{nj}| = |p| - 1$).

Let the location of each node $l \in G_n (l \neq n_n)$ be given as an ordered pair (x_l, y_l). We define the co-ordinates of n as

$$\begin{aligned} x_n &= \sum_{k \in FI(n)} f(W^n_{kn}) x_k + \sum_{j \in FO(N)} f(W^n_{nj}) x_j \\ y_n &= \sum_{k \in FI(n)} f(W^n_{kn}) y_k + \sum_{j \in FO(N)} f(W^n_{nj}) y_j \end{aligned}$$

where f is a function satisfying

1. $0 \leq f(W^n_{kl}) \leq 1$
2. $\sum_{k \in FI(n)} f(W^n_{kn}) + \sum_{j \in FO(N)} f(W^n_{nj}) = 1$
3. f is a strictly decreasing function.

The function f serves to weigh the co-ordinates appropriately. Intuitively, f should be a decreasing function: lesser the amount of time available to expend along a wire, the closer should the location of the node n to its fanin or fanout. The conditions on f above guarantee that (x_n, y_n) lies in the convex hull defined by the locations of the fanins of n and the locations of the fanouts of N. If this location (or nearest integer location) is empty, we can place n at the position obtained. We allow a tolerance of ± 1 units around the location calculated above. If we cannot find an empty location, we discard the decomposition and continue with the algorithm.

We define the function f as follows: Let

$$\begin{aligned} S_n &= \max\{W^n_{kl} : e^n_{k_n l_n} \in G_n\} \\ F_n &= \sum_{i \in FI(n)} (-W^n_{in} + S_n) + \sum_{j \in FO(N)} (-W^n_{nj} + S_n) \\ f(W^n_{in}) &= \frac{(-W^n_{in} + S_n)}{F_n} \qquad i \in FI(n) \\ f(W^n_{nj}) &= \frac{(-W^n_{nj} + S_n)}{F_n} \qquad j \in FO(N) \end{aligned}$$

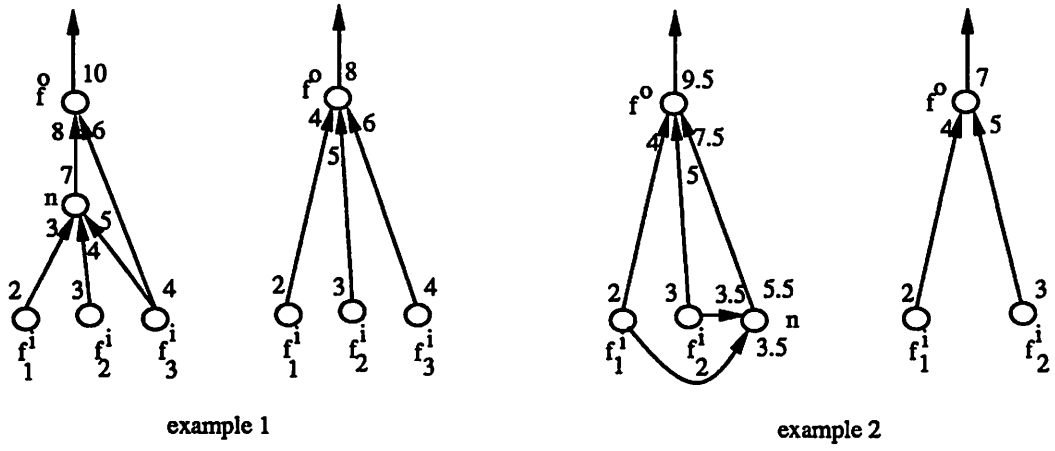


Figure 7.8: Collapsing examples

S_n gives the largest weight in the graph G_n. F_n is the normalizing factor. The f so defined satisfies the conditions described above. Doing so we guarantee that the location of n is closer to a node that is connected to n by an edge with lower weight than other nodes.

Partial Collapse: We motivate the collapsing operation with the help of two examples. In both examples, node n is collapsed into its fanout f^o (Figure 7.8). All the numbers are the arrival times at nodes and edges. The block delay is assumed to be 2 units. In the first example, the benefit gained from the collapse is due to the removal of node n from the path. In the second, additional benefits accrue due to a skewed configuration.

We consider a critical node n for collapsing into one of its fanouts f^o. If collapsing n into f^o is feasible, we estimate the savings in delay because of this transformation. For this, we recompute the arrival time at f^o by first computing the delay through each of the edges (f^i, f^o), where f^i is a fanin of n. The pair (n, f^o) is ranked with a score equal to the difference between the old arrival time and new arrival time at f^o if this difference is nonnegative; else the pair is not considered for collapse. Note that the position of the node f^o remains unchanged. This way we rank all the critical (node, fanout) pairs in the network that result in a feasible collapse. We greedily select the pairs with large scores.

7.5 Experimental Results

MCNC benchmarks were used for all the experiments. First, the benchmarks were optimized for area.[3] Then a delay reduction script, *script.delay*, with *speed_up* at the end was used to obtain delay optimized networks in terms of 2-input NAND gates. In Table 7.1, we report results after the placement-independent phase of mis-fpga and chortle-d, using the same starting networks for both the programs. We set m to 5. We are restricting ourselves to single output CLBs, i.e., m-LUTs. For chortle-d we used the option -K 5 -r -W -e. For each example, we report the number of levels, 5-LUTs, edges and the CPU time (in sec.) on DEC5500 (a 28 mips machine) in the columns *lev*, *5-LUTs*, *edges* and *sec.* respectively. Out of 27 benchmarks, mis-fpga generates fewer levels on 9 and more on 13. On average (computed as the arithmetic mean of the percentage improvements for each example), mis-fpga needed 2.9% more levels. The number of blocks and the number of edges it needs are 58.7% and 66.2% respectively, of chortle-d. As shown later, the number of nodes and edges may play a significant role in determining delay of a network. However, chortle-d is much faster than mis-fpga.[4]

For the placement-dependent phase, the starting networks are the ones obtained from the level reduction algorithms. We conducted three sets of experiments:

1. PI + apr: The network obtained after the placement-independent phase of mis-fpga was placed and routed using apr, the Xilinx place and route system [88].

2. chortle-d + apr: The network obtained after chortle-d was placed and routed using apr.

3. PI + PD + apr: After the placement-independent algorithm, the placement-dependent phase of mis-fpga is applied. The resulting placed network is routed using apr, with its placement option disabled. The routing order is based on the slacks computed for the edges: apr is instructed to route more critical nets first. Logic synthesis is invoked once at each temperature. However, it is started only at a low temperature, the reason being that collapsing and decomposition may increase the number of edges in the network. At higher temperatures, a swap that increases the cost function is accepted with higher probability. A subsequent synthesis phase at that temperature may decrease the cost function, but increase the routing

[3] Regrettably, the area optimization script is not *script.rugged*, which is used in other chapters. The reason is that the experimental set-up in this section uses some proprietary tools, whose license expired some time ago. The results reported here are two years old, taken from [64], which used different area optimization scripts.

[4] One reason is that since mis-fpga is based on misII, a lot of time is spent in allocating (freeing) memory for additional data structures whenever a node is created (freed).

example	mis-fpga - PI				chortle-d			
	lev	LUTs	*edges*	*sec.*	*lev*	LUTs	*edges*	*sec.*
z4ml	2	10	42	2.1	3	20	74	0.1
misex1	2	17	71	1.7	3	25	99	0.1
vg2	4	39	165	1.7	3	54	206	0.1
5xp1	2	21	88	3.5	4	29	115	0.1
count	4	81	336	5.1	3	102	368	0.1
9symml	3	7	35	9.9	4	76	273	0.1
9sym	3	7	35	15.2	5	130	477	0.2
apex7	4	95	383	8.4	4	131	452	0.2
rd84	3	13	61	9.8	4	69	268	0.2
e64	5	212	857	15.7	4	356	1236	0.6
C880	9	259	1070	39.0	7	383	1437	0.9
apex2	6	116	481	9.8	5	165	578	0.2
alu2	6	122	543	42.6	8	316	1189	0.7
duke2	6	164	685	16.4	4	248	863	0.4
C499	8	199	896	58.8	6	436	1736	1.8
rot	7	322	1312	50.0	6	439	1608	1.0
apex6	5	274	1209	60.0	5	361	1360	0.8
alu4	11	155	648	15.4	8	194	710	0.3
sao2	5	45	189	9.5	4	58	220	0.1
rd73	2	8	36	4.4	4	52	183	0.1
misex2	3	37	160	1.4	2	52	188	0.1
f51m	4	23	100	5.9	5	65	237	0.1
clip	4	54	219	3.7	4	83	281	0.1
bw	1	28	138	8.3	1	28	138	2.6
b9	3	47	199	2.3	3	62	225	0.1
des	11	1397	6159	937.8	9	3024	10928	9.2
C5315	10	643	2826	282.2	9	1221	4509	3.6

Table 7.1: Results for level reduction

lev number of levels in the final network
LUTs number of LUTs in the final network
edges number of edges in the final network
sec. time taken in seconds (on a DEC5500) to generate the final network

example	mean value of $(\frac{RC}{length^2})$
5xp1	0.774
9sym	0.682
9symml	0.767
C499	0.842
alu2	0.649
alu4	0.663
apex2	0.738
apex6	0.609
apex7	0.805
b9	0.940
bw	0.851
clip	0.828
count	1.103
duke2	0.615
f51m	1.306
misex1	0.848
rd84	0.933
rot	0.785

Table 7.2: Calculation of delay per unit length squared

complexity and number of levels. Though each such synthesis phase is *good* relative to the current placement state, from an absolute point of view, it may be *bad*. We found that it is better to start synthesis at a temperature when there are no major hill-climbing swaps. Then each synthesis phase results in an improvement. Also, we found it helpful to have the total net length dominate the cost function at higher temperatures. This brings the blocks close to each other. As the annealing proceeds, we increase the contribution of delay change δd in δc by increasing $\alpha(T)$. Finally, at very low temperatures, $\alpha(T) = 1$, i.e., only the delay appears in the cost function. The arrival times of the primary inputs are set to 0 and the required times of the primary outputs to the arrival time of the latest arriving output. We used Elmore delay model in all delay computations.

To compute the delay per unit length squared, apr was run on a set of examples and delay of each edge along with its length was obtained. From this, the $RC/(length^2)$ value for each edge and then the average values for the networks are computed. These are shown in the Table 7.2 for some examples. Then the delay per unit length squared is set to the average value 0.82.

The results of the experiments for placement, resynthesis, and routing are shown in Table 7.3. The table shows the delay through the circuits in nanoseconds after placement and routing. The block delay used is 15 ns. We chose only those benchmarks that could be successfully placed and routed on one chip. The PI phase of mis-fpga gives lower delay than chortle-d on most of the examples. More interestingly, we can study the affect of number of nodes and edges on the delay. For example, though the number of levels in *count* for chortle-d is 3 and for mis-fpga is 4 (Table 7.1), the delay through the circuit for mis-fpga is 3 ns less than chortle-d. In fact, using the PD phase of mis-fpga makes the difference even larger. For *vg2*, *duke2*, *alu4* and *misex2*, the delays for mis-fpga are higher than chortle-d, but the difference in delays is less than $15(difference\ in\ levels)$. This effect will be more pronounced in examples where the number of nodes is greater than the capacity of the chip. Then, extra inter-chip delay will be incurred, which may be an order of magnitude higher than the on-chip delay.

It turns out that logic synthesis in the PD gives mixed results on these networks. This is partially because many networks did not have much room for delay improvement by local resynthesis. We observed that if we start with networks that did not use an aggressive block count minimization during the PI phase, resynthesis improves the delays significantly.

7.6 Discussion

Given an m-feasible network, the goal in the PI phase is to minimize the number of levels. What we presented was a heuristic to solve the problem. The DAG-Map algorithm, proposed recently by Cong and Ding [16] computes a minimum delay solution if only collapsing of nodes is allowed. However, in the PI phase of mis-fpga, covering is integrated with resynthesis (e.g., functional decomposition) when it is found out that collapsing a node into its fanout is not feasible.

Experimental evidence indicates that two circuits having the same number of levels can have widely varying delays after the final placement and routing. So, the number of levels may not be a good cost function.

The delay model used currently at the logic level is weak. This is because it has no idea about the wiring delays, which are a function of the module locations. Better delay models need to be developed also for placement. The Elmore delay model does not consider fanout loading, and the Rubinstein-Penfield-Horowitz model gives two delay numbers, which could differ from each other significantly.

example	mis-fpga PI + apr	chortle-d + apr	mis-fpga PI + PD + apr
z4ml	33.60	56.00	31.00
misex1	33.10	58.00	36.20
vg2	82.90	76.40	76.30
5xp1	33.60	77.40	35.90
count	88.40	91.88	79.02
9symml	54.00	84.10	53.50
9sym	53.70	110.40	53.50
apex7	97.75	108.00	93.90
rd84	50.70	77.80	54.30
apex2	147.43	134.30	142.50
duke2	125.13	114.70	151.83
alu4	256.35	230.68	-[1]
sao2	104.00	82.30	96.00
rd73	33.60	85.00	31.00
misex2	53.80	47.80	53.70
f51m	72.60	107.50	76.60
clip	81.10	84.10	84.60

[1] two nets could not be routed.

Table 7.3: Delays after placement and routing

Part II

Multiplexor-based Architectures

Chapter 8

Mapping Combinational Logic

8.1 Introduction

In this chapter, we study MUX-based architectures, in which the basic block is a combination of multiplexors, with possibly a few additional logic gates such as ANDs and ORs. Interconnections are realized by programmable switches (*anti-fuses*) that may connect the inputs of the block to signals coming from other blocks or to the constants 0 or 1, or may bridge together some of these inputs.

The most popular architectures are *act1* and *act2* introduced by Actel [29], and are shown in Figure 8.1. Each module has eight inputs and one output. While *act1* is a tree configuration of three 2-to-1 multiplexors with an OR gate at the control (select) input of MUX3, *act2* has three multiplexors, an OR gate, and an AND gate. These logic blocks can implement a large number of logic functions. For example, the *act1* module can implement all two-input functions, most three-input functions [38], and several functions with more inputs. However, some of these functions are P-equivalent. In [56], 702 non-P-equivalent functions for *act1* and 766 for *act2* were counted.

As in the case of LUT-based architectures, the number of blocks on a chip, logic functions that these blocks can implement, and the wiring resources are the main constraints. Also, the architecture-specific mapping typically starts with a network that has been optimized by the technology-independent operations. In future, we expect these operations to be driven by the target technology.

This chapter is organized as follows. A brief history of the MUX-based mapping is presented in Section 8.2. BDD-based techniques are described in Section 8.3. Their extensions to ITEs, along with the matching algorithms, form the complete mapping algorithm, which is outlined

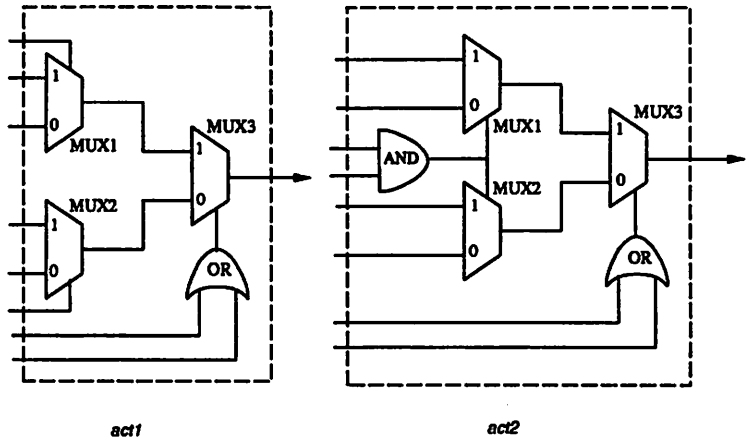


Figure 8.1: Actel architectures: *act1* and *act2*

in Section 8.4. The matching algorithms for *act1* and *act2* are described in Section 8.5, and the ITE-based mapping in Section 8.6. Experimental results using these techniques are described in Section 8.7. The approach is critiqued in Section 8.8.

8.2 History

8.2.1 Library-based

In 1989, when we first started looking at the synthesis problem for MUX-based architectures, we knew of only one mapping approach - the one based on the creation of a library. The library can be created with gates that represent functions obtained from the basic block by connecting the function inputs to some of the input pins and then tying the remaining pins to constants (0 or 1). The network is represented as a subject graph in terms of a set of base functions, typically a 2-input NAND gate and an inverter. All the functions in the library are also represented in terms of the base functions. These are the pattern graphs. The problem then is to cover the subject graph using the minimum number of pattern graphs.

An advantage of this approach is that it is quite insensitive to the basic block architecture. The only change needed is the creation of a new library. If we put in all the functions that can be implemented with the *act1* or *act2* module, the library size will be unmanageable. The size can be reduced by putting only one out of all P-equivalent functions in the library. Efficient algorithms that use BDDs can produce all non-P-equivalent functions implemented by a MUX-based block in

a short time. However, even this number may still be large, although not as large as for the LUT architectures (706 for the *act1* as compared with 9,014 for a 4-input, 1-output LUT). Also, each such function is represented in all possible ways, and so the number of pattern graphs is much larger. In addition, the time to map becomes quite high. One remedy is to select a smaller subset of functions for the library, say by throwing away less frequently used functions. In general, such a reduction can result in a loss of quality of results.

8.2.2 BDD-based

In 1990, we proposed in mis-fpga [62] that for MUX-based architectures, the set of base functions should be changed from a NAND gate and an inverter to a 2-1 multiplexor, since it is a more natural representation for the MUX-based architectures. We defined both the subject graph and the pattern graphs in terms of BDDs. Recall that each non-terminal vertex of a BDD represents a 2-1 multiplexor. We used both ordered and unordered BDDs for the subject graph. A very small set of pattern graphs was needed, another benefit of the new function representation. As in the popular library-based approaches, the underlying mapping algorithm in mis-fpga was based on dynamic programming. To further improve the quality of results, we added an iterative improvement step after initial mapping; it tries to exploit the relationship between nodes of the network. It consists of three main operations: *partial collapse*, decomposition, and quick-phase (which decides the phase - positive or negative, in which a function should be implemented). Significant gains over the library-based approach of misII were obtained in the quality of results.

A similar approach was later used in ASYL [80, 6]. The input ordering for the BDD is obtained by lexicographical factorization, the central theme of ASYL. Both area- and speed-oriented solutions were presented.

8.2.3 ITE-based

In 1991, Karplus proposed Amap in which he extended the idea of using a MUX-based representation. Instead of using BDDs, he used if-then-else dags (ITEs), the most general representation of a function in terms of multiplexors. The selector function at each vertex of an ITE can be a function of inputs, rather than being an input, which is the case for BDDs. As pointed out, one main advantage of ITEs over BDDs is that duplication of cubes can be avoided [38]. Amap does not give results as good as mis-fpga but is much faster.

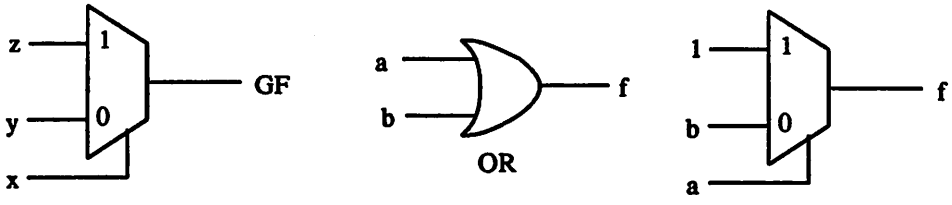


Figure 8.2: Can a 2-1 MUX implement an OR gate?

8.2.4 Boolean Matching-based

Also in 1991, Ercolani *et al.* [21] proposed PROSERPINE, an approach based on Boolean matching, which answers the following fundamental question: *"Can a given function f be realized by a gate-function GF, where some of the gate-inputs may be tied to constants (i.e., 0 or 1), or other inputs?"*

Example 8.2.1 *In Figure 8.2, we show that a 2-input OR function f can be realized by a 2-1 multiplexor if the select and the 0 input of the* MUX *are tied to a and b respectively, and the 1 input to the constant 1.*

PROSERPINE constructs ROBDDs for f and GF and then checks if the ROBDD for f is a subgraph of the ROBDD for GF. In the worst case, all possible variable orderings for the ROBDD for GF have to be considered. The problem is further complicated by the fact that some inputs of GF may have to be tied to either constants or other inputs (**bridging**) to realize the desired function f. The PROSERPINE paper [21] does not report the CPU times, but we have reasons to believe that the matching check is expensive. This matching forms the core of the mapping algorithm, in which sub-functions of the network are extracted and checked for realizability by one block.

In 1992, Burch and Long proposed matching algorithms using BDDs [15]. Their main contribution was an algorithm for matching under input negations that takes time polynomial in the size of the BDDs representing the functions to be matched. This algorithm forms the basis of the algorithms for matching under input permutations, bridging, and constant inputs.

8.2.5 Combining Various Approaches

In 1992, following Karplus, we incorporated ITEs in our approach. In addition, we developed fast matching algorithms for *act1* and *act2*. We combined both techniques along with the

iterative improvement step of mis-fpga in a single framework, and this yielded better results [60]. Later, in 1993, we improved various aspects of the algorithm, including the following:

1. selection of the branching variable at each step of the ITE construction,
2. construction of ITEs for functions having cubes with disjoint supports,
3. coupling between the matching algorithm and the ITE paradigm - it was made tighter,
4. use of the new ROBDD data structure [9], which has complementary edges, and
5. creation of multi-rooted subject graphs.

Note that no approach except the first one uses an explicit library.

8.3 Constructing Subject Graph and Pattern Graphs using BDDs

We consider two BDD representations: ROBDD and unordered BDD (or simply BDD).

8.3.1 ROBDDs

ROBDDs are attractive as a subject graph representation because they are compact and do not have any function implemented more than once. The basic idea is to construct an ROBDD for the function and then cover it by minimum number of pattern graphs. So we will like to have an ROBDD that is small. It is well known that the ROBDD size is sensitive to the ordering of the input variables [14]. Unfortunately, no polynomial-time algorithm is known for finding an ordering that results in the smallest ROBDD. However, if the function has a small number of inputs, an optimum ordering can be determined by trying out all possible orderings and picking the best one. In our case, an optimum ordering is one whose corresponding ROBDD can be covered with fewest patterns. One way of obtaining functions with small number of inputs is by transforming the given network η into a network $\tilde{\eta}$ in which every node has at most N fanins, where N is a small constant. The problem of obtaining $\tilde{\eta}$ from η is same as the synthesis problem for N-LUT architectures, and the techniques of Chapter 3 can be used. For each node, the ROBDDs corresponding to all the input orderings are constructed, their costs are evaluated using the covering algorithm (to be described shortly), and the ordering that yields the minimum cost is picked.

Although for an arbitrary function we do not know an easy way of computing an optimum ordering, for some simple functions we do know how to do it. As the next two propositions show,

two such classes of functions are: those consisting of only single-literal cubes in their SOP, and those with just one cube.

Proposition 8.3.1 *If a function $f = f(x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_l)$ consists of only single literal cubes with input variables $x_1, x_2, \ldots, x_k$ occurring in the positive phase and $y_1, y_2, \ldots, y_l$ in the negative phase, then the* ROBDD *corresponding to the ordering $x_1, y_1, x_2, x_3, y_2, x_4, x_5, y_3, x_6, x_7, \ldots$ of input variables results in the minimum number of act1 blocks after an optimum covering as compared to other orderings, where this ordering starts from the leaves.*

Sketch of Proof For any ordering, the ROBDD for f is a chain, i.e., at least one child of each non-terminal vertex is a terminal vertex. Since the covering method is based on ROBDDs, the select lines of MUX1 and MUX2, and the inputs of the OR gate can only be the inputs of f. Then, to cover the chain ROBDD of f, *act1* can be configured in one of the following two ways.[1]

1. The OR gate inputs are tied to x_i and x_j $(i \neq j)$, which are inputs occurring in positive phase in f. Then, the data input '1' of MUX3 is constant 1, and the input '0' is a sub-function of f. The select line of MUX2 can be either an x input or a y input.

2. Only one input of the OR gate is tied to an input of f, the other input being constant 0. The OR gate input can be x_i or y_j. If it is x_i, the data input '1' of MUX3 is constant 1, and the input '0' is a sub-function of f. The select line of MUX2 can be either an x input or a y input. If it is y_j, the data input '0' of MUX3 is constant 1, and the input '1' is a sub-function. The select line of MUX1 can be either an x input or a y input.

In the first case, the *act1* module can "cover" a maximum of three inputs of f; in the second case, it covers two inputs, the only exception being the *act1* module that is bottommost in the chain. This module can cover an extra input of f if the data input of its MUX1 (or MUX2) is an x input. Then, the problem of minimizing the number of *act1* modules reduces to that of finding an optimum ordering of the input variables. The x inputs should be used as the inputs to the OR gate as much as possible, since two of them can be covered by an OR gate. To save them for the OR gate, the MUX1 (or MUX2) select input should be a y input. These simple rules correspond to an ordering where two x inputs are interleaved with a y input. The exceptional case of the bottommost module is handled by putting a positive phase input (say x_1) first in the ordering. ■

[1] modulo complementation of some intermediate functions, which does not alter the basic argument.

Proposition 8.3.2 *If a function $f = f(x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_l)$ is a single cube with input variables $x_1, x_2, \ldots, x_k$ occurring in the positive phase and $y_1, y_2, y_3, \ldots y_l$ in the negative phase, then the* ROBDD *corresponding to the ordering $x_1, x_2, y_1, y_2, x_3, y_3, y_4, x_4, y_5, y_6, \ldots$ of input variables results in the minimum number of act1 blocks after an optimum covering as compared to other orderings, where this ordering starts from the leaves.*

Proof Similar to that of Proposition 8.3.1. ■

Note the following:

1. The first input in both the orderings is always an input in the positive phase. Starting from the second variable, the orderings follow a repetition rate of 3: either a positive input followed by two negative inputs, or a negative input followed by two positive inputs.

2. When inputs of some phase get over, the remaining inputs (of the other phase) are simply concatenated to complete the ordering.

3. Both orderings are listed so that they start from the terminal vertices and end at the root of the ROBDD.

Example 8.3.1 *Consider functions*

$$\begin{aligned} f_1 &= a'bc'de', \\ f_2 &= a' + b + c' + d + e' + g + h. \end{aligned}$$

From Proposition 8.3.2, an ordering for f_1 that results in the minimum number of act1 *blocks is b, d, a, c, e. In the corresponding* ROBDD*, vertex corresponding to input e would be the root. Also, from Proposition 8.3.1, the ordering for f_2 is b, a, d, g, c, h, e.*

From the perspective of synthesis for MUX-based architectures, the ROBDD representation has the following drawbacks.

1. The input ordering constraint imposed by the ROBDD may be too severe and can result in a higher block-count after mapping. For instance, see Example 8.3.2.

2. Since an ROBDD has only one copy of a function, a vertex v can be pointed at by many vertices - all are parents of v. Since the covering procedure we use is tree-based, the subject graph is clipped at vertices with multiple parents. And if there are several such vertices, the subject graph gets partitioned into many small trees. Although each tree is mapped optimally, more the trees, more is the deviation from a global optimum solution.

This leads us to consider another representation that does not have these two restrictions.

8.3.2 BDDs

The goal is to construct a BDD (without any ordering restrictions) such that the number of vertices in the BDD is small and the number of vertices with multiple parents is small. The method used to construct BDDs is subsumed by the method for constructing ITEs - to be described in Section 8.6. Since ITEs are the representation of choice currently, we do not present the BDD construction algorithm here. It can be easily derived from that for ITEs. In any case, the reader is referred to [62].

A drawback of the BDD representation, or at least the heuristic to construct it, is that some function may be replicated many times in different branches of the BDD.

In general, it is not possible to predict which type of representation, ROBDD or BDD, will give a lower-cost implementation; we have to construct both the types for a node function and select the one with lower cost.

8.3.3 Local versus Global Subject Graphs

Experiments showed that, in general, constructing a subject graph for the entire network in terms of the primary inputs (**global subject graph**) leads to worse results as compared to when a subject graph for the local function at each node of the network (**local subject graph**) is constructed. This can be explained as follows.

1. The global ROBDD and BDD require all vertices to be indexed by primary inputs. This is too restrictive; smaller representations are possible if the vertices can be indexed by internal node functions, precisely what a local subject graph is.

2. The ordering constraint imposed on a global ROBDD can cause it to be huge. It is crucial that the subject graph be small, so that acceptable mapping solutions are generated after covering it with a small pattern-set (which we use). The basic assumption behind having a small set of pattern graphs is that the subject graph is close to optimum. If it is not, then the pattern-set needs to be enlarged.

So, we construct subject graphs for each node of the network separately.

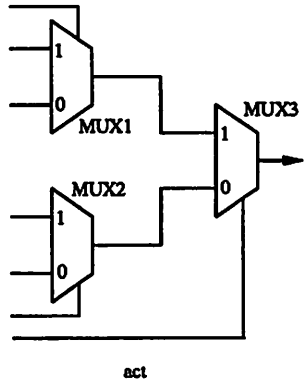


Figure 8.3: *act* - a simplified *actl*

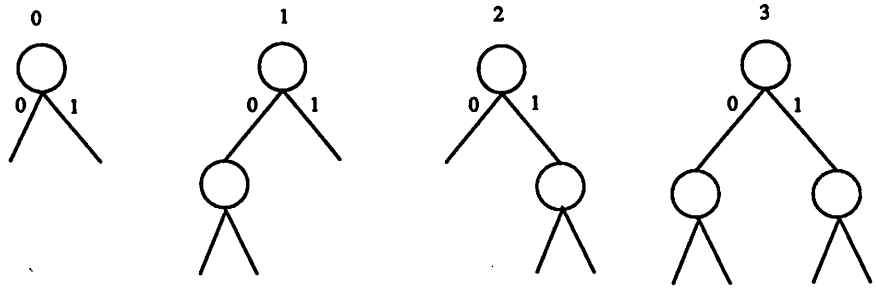


Figure 8.4: Patterns for *act*

8.3.4 Pattern graphs

First, consider a simplified version of *actl* module, *act*, as shown in Figure 8.3. It is obtained by ignoring the OR gate of *actl*. We consider four pattern graphs for the *act* module, as shown in Figure 8.4. A circle in a pattern graph is a 2-1 MUX, whose data input 0 is provided by the *low* or 0 child, and input 1 by the *high* or 1 child. If a function is realizable by one *act* block, it either uses one multiplexor, or two, or all three multiplexors.[2] The pattern graphs are in one-to-one correspondence with these possibilities. This small set of patterns suffices to capture all possible functions realizable by one *act* block. This is formally stated in Proposition 8.3.3. Note that all the pattern graphs are leaf-DAGs, as the *hanging edges* (whose one end-point is not circled) are allowed to terminate at any vertex of the BDD.

In *actl*, introducing the OR gate at the control input of MUX3 increases the number of functions realized as compared to *act* considerably. However, from an algorithmic point of

[2]If a function does not use any multiplexor, it is either **1**, **0**, or identically equal to an input. Then it can be realized without any block.

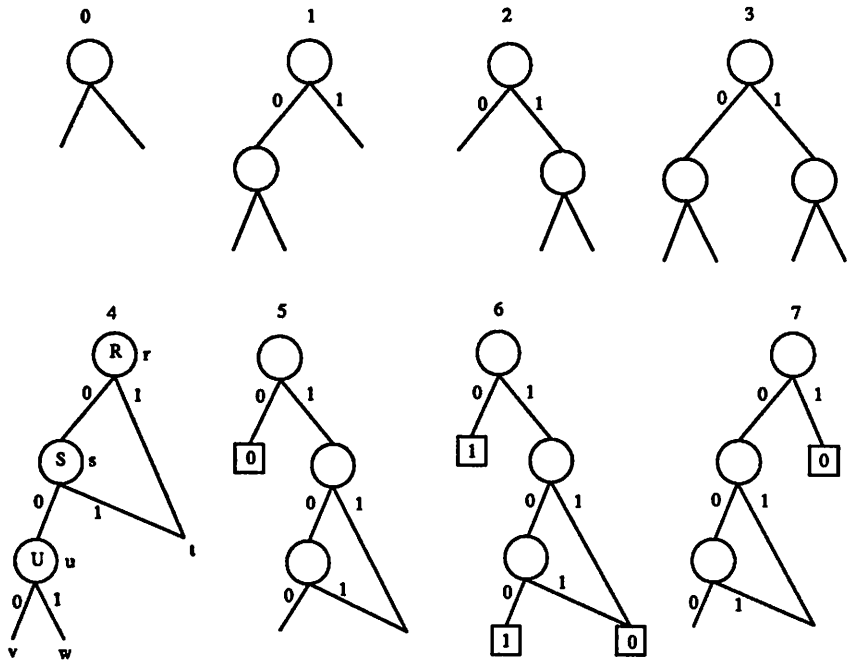


Figure 8.5: Patterns for *act1*

view, it causes some difficulties - number of pattern graphs increases, the correspondence between multiplexor usage and the pattern graphs is destroyed, and some of the pattern graphs are no longer leaf-DAGs. Currently, we have a set of 8 pattern graphs for *act1*. They are shown in Figure 8.5. The patterns 0 through 3 are the same as the ones in Figure 8.4. The patterns 4 through 7 exploit the presence of the OR gate. To see how, consider the pattern 4. Let

r = root vertex of the pattern 4,

R = the variable at r,

$s = low(r)$,

S = the variable at s,

$t = high(r) = high(s)$,

$u = low(s)$,

U = the variable at u,

$v = low(u)$, and

$w = high(u)$.

Then the function g implemented by r in terms of the inputs of the pattern is

$$g = Rt + R'(St + S'(Uw + U'v))$$

$$= Rt + R'St + R'S'(Uw + U'v)$$
$$= (R+S)t + R'S'(Uw + U'v)$$

If R and S are tied to the OR gate inputs of *act1* (see Figure 8.1), MUX1 is configured to realize t, and MUX2 is configured to realize $Uw + U'v$, g can be realized with one *act1* module. Similar derivations can be made for the patterns 5 through 7. Although non-leaf-DAG patterns (i.e., ones with internal fanouts) are possible, we do not include them in our set. This is because the covering algorithm breaks the subject graph into trees and no sub-tree can match against a pattern graph that has internal fanouts.

8.3.5 The Covering Algorithm

We use the tree-covering heuristic proposed in [41]. The only difference is that in our case, the subject graph and the pattern graphs are in terms of 2-to-1 multiplexors. We now justify the use of the set of pattern graphs for *act*, as shown in Figure 8.4, by stating the following proposition.

Proposition 8.3.3 *For a function f realizable by one* act *module, there exists a subject graph S whose non-leaf portion is isomorphic to one of the four pattern graphs of Figure 8.4, say p. The covering algorithm will map S onto p.*[3]

Proof Deferred to Section 8.5.1. ■

Later, in Section 8.5.1, we will present an algorithm to determine if f is realizable by one *act* module without constructing all possible ROBDDs.

Since the pattern-set is small, the covering algorithm is fast.

Example 8.3.2 *Consider $f = d(ac + a'b) + d'(ca + c'b)$. Figure 8.6 (A) shows a* BDD *for f, and (B) an* ROBDD. *After applying the covering algorithm, it is seen that 3* act1 *modules are needed for the* ROBDD, *whereas 1 module suffices for the* BDD. *A dotted rectangle denotes a match, i.e., an instance of* act1 *module, which is configured as one of the patterns of Figure 8.5. Also note that the vertices not covered by any match are the leaf vertices.*

8.4 Proposed Mapping Algorithm

[3]This subject graph S is an ROBDD and can be found by constructing ROBDDs for all possible input orderings for f. Since f is a function of at most 7 inputs, 7! orderings may need to be examined.

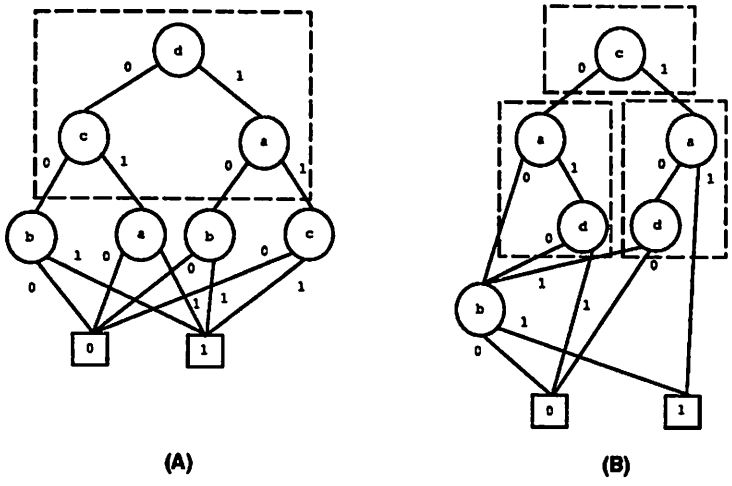


Figure 8.6: BDD representations

An outline of the overall synthesis algorithm for the MUX-based architectures is shown in Figure 8.7. It resembles, at this level of abstraction, the algorithm for the LUT architectures of Figure 3.34. The differences are in the way some of the specific algorithms (e.g., initial mapping) work.

First, the network is optimized (currently, we use the standard technology-independent optimization techniques). The mapping algorithm works as follows on the optimized network.

1. *Initial mapping of each node*: First, we check, using the matching algorithm to be described in Section 8.5, if the local node function f can be implemented with one basic block. If not, we construct an ITE for f (the subject graph) using the procedure of Section 8.6 and map it, i.e., cover it with the pattern graphs corresponding to the basic block by the tree-based, dynamic programming algorithm.

   After this step, each node of the network has a feasible implementation associated with it. However, the interconnection structure of the network remains the same. The cost of a node is the number of basic blocks in its feasible implementation.

2. *Iterative improvement*: The node-by-node mapping paradigm used in the previous step does not exploit the relationship between the nodes. To do so, we use an iterative improvement phase. Two operations, *partial collapse* and decomposition, are tried. *Partial collapse* is the same as that for the LUT architectures (see Section 3.5), except that the cost of a function is now measured in terms of the MUX-based blocks. Decomposition uses *decomp -g* of

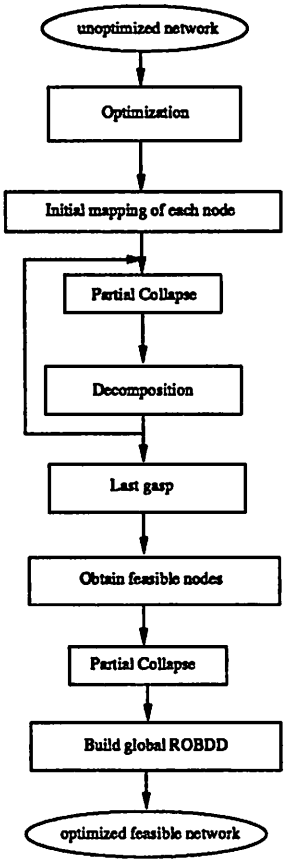


Figure 8.7: Overview of the algorithm

misII [12] to break a node into a set of nodes, which are then mapped. If the cost improves, the original node is replaced by its decomposition. *Partial collapse* and decomposition are repeated for some number of iterations.

3. *Last gasp*: A node may cost more than one *act1* block. In last gasp (a term borrowed from ESPRESSO), we try to reduce this cost using one or both of the following techniques:

   (a) Construct an ROBDD for the local node function and map it. If the node cost improves, save the new mapping. Note that so far in the algorithm, ROBDDs had not been used.

   (b) From each node n, construct a network $\tilde{\eta}(n)$ with one internal node that is a copy of n, one primary output, and as many primary inputs as the fanins of n. Apply a decomposition algorithm on $\tilde{\eta}(n)$, generating many smaller nodes. Then, invoke the steps 1 and 2 of this algorithm on the resulting network and determine its implementation

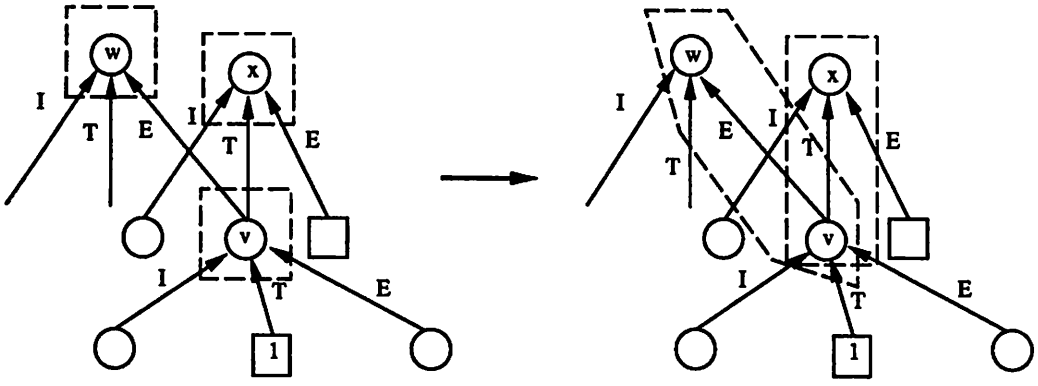


Figure 8.8: Why does the last iteration help?

cost. If this cost is less than that of n, replace n by the new network. This last gasp technique mimics on a node what the first two steps of the algorithm do on a network.

4. *Obtaining feasible nodes*: At this point, each node has attached to it a feasible implementation, which is either the node itself (if the matching algorithm succeeded in step 1), or a mapped ITE, or a mapped BDD. In this step, we replace each node by its feasible implementation, so that the resulting network is feasible with respect to the basic block.

5. *Last iteration*: Here we perform one iteration of *partial collapse*, as it can potentially improve the quality. This can be attributed to two factors.

   (a) *Partial collapse* is now being performed on smaller nodes, each having a cost of one block. It is possible that two such nodes can be absorbed in one block (for instance, when these two nodes belong to the feasible implementations of two nodes of the original network, one of them fanning out to the other).

   (b) As shown in Figure 8.8, there may be a multiple-fanout vertex v within a mapped ITE (just before step 4) that is covered by the pattern 0 of Figure 8.21 (i.e., a single MUX) and that can be absorbed in its fanouts. This is possible, for instance, when all the fanouts of v (in our example, w and x) are roots of some pattern graphs (in our example, 0) in the cover of the ITE. This happens when the fanouts of v are multiple-fanout vertices. So the node corresponding to v in the feasible network obtained after step 4 can be collapsed into all its fanouts, thereby reducing the network cost by 1.

This step was inspired by [6] and the *partition* algorithm for LUT architectures.

6. *Global* ROBDD : Sometimes it is beneficial to construct a global subject graph. Since the algorithm described so far is based on unordered ITEs and local ROBDDs, it is worthwhile to experiment with a global subject graph that is ordered. We found ordered ITEs [37, 44] to be ineffective: creating them takes a long time and the quality of the solution is not redeeming either, and so decided to use ROBDDs. We discovered experimentally that global ROBDDs may improve results when the network does not have too many primary inputs. Also, for symmetric circuits, this step works very well, since an n-input symmetric function f can be realized with $O(n^2)$ multiplexors. Such a realization requires that function-sharing be detected. Since f is symmetric, an ROBDD for f automatically generates this realization irrespective of the input ordering chosen for the ROBDD.

   We used two ROBDD packages:

   - *old* [53]: It does not use *complementary edges* (a complementary edge carries an inverter to complement the function). This conforms to the target architectures *act1* and *act2*, which do not have explicit inverters. However, since different primary outputs are not allowed to share vertices in their ROBDDs, the representation provided in this package is larger than it ought to be.
   - *new* - based on [9]: It uses complementary edges and permits the ROBDDs for different primary outputs to share vertices wherever possible. Due to the presence of the inverters on the complementary edges, the ROBDD is not mapped directly. Instead, it is first converted into a network of MUXes and inverters, which is then mapped using one iteration of *partial collapse*.

The algorithm is flexible, i.e., except for the initial mapping, other steps are optional. Moreover, they can be reordered. For example, right after step 1, step 4 can be applied followed by step 2. The user is free to experiment with different orders, thus generating different designs.

The algorithm is applicable for both *act1* and *act2* architectures. However, some architecture-specific differences are there. The differences arise in the matching algorithm, construction of ITEs, and pattern graphs. Since the matching algorithms for *act1* and *act2* are quite different, we present them for both in Section 8.5. For the rest (i.e., construction of ITEs and corresponding pattern graphs), we focus on *act1* in Section 8.6; these steps can be suitably modified for *act2*.

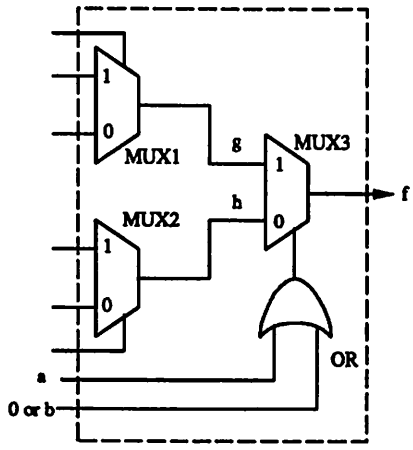


Figure 8.9: An *act1*-realizable function

8.5 The Matching Problem

In this subsection, we address a fundamental problem in mapping: *given a completely specified function f and the basic block of the architecture, is f realizable by a single block?* A more general version of this problem is addressed in [21]: given a function f and a gate-function GF, is f realizable by GF? We restrict ourselves to two special gate-functions - those corresponding to the *act1* and *act2* blocks. By this restriction, we hope to have a much faster algorithm.

8.5.1 The Matching Problem for *act1*

First the matching theory is developed, which is then incorporated in an algorithm.

Assume that a completely specified function f is realizable by an *act1* block, and that a minimum support cover for f is given.[4] Since *act1* has 8 inputs, assume that $1 \leq |\sigma(f)| \leq 8$ (**0** and **1** functions are trivially realizable, so $|\sigma(f)| = 0$ is not considered). Observe that, without loss of generality, one of the inputs of the OR gate can be tied to an input of f. This is because otherwise, the select input of MUX3 is either constant 0 or constant 1. In either case, if f is realizable by an *act1* module, it is realizable by a 2-1 MUX, and hence by MUX3 with a non-constant input (i.e., an input of f) at its select line. Refer to Figure 8.9 for the rest of the discussion. There are two cases:

1. *One OR gate input is $a \in \sigma(f)$, and the other input is 0.* Then, f has a decomposition of the form

$$f = ag + a'h \tag{8.1}$$

[4]The optimization phase, by generating a prime cover, guarantees minimum support.

for some input a of f, where g and h are each realizable by a 2-1 MUX. The problem is to find g and h. We may assume, without loss of generality, that g and h are independent of the input a. This follows from the following proposition:

Proposition 8.5.1 *If f has a decomposition as in (8.1), where g and h are realizable by a 2-1* MUX *each, then it also has a decomposition of the form*

$$f = ag_1 + a'h_1, \tag{8.2}$$

where g_1 and h_1 are realizable by a 2-1 MUX *each and are independent of the input a.*

Proof Since g is realizable by a 2-1 MUX, it can be written as $g = CA + C'B, A, B, C \in \{0,1\} \cup \sigma(f)$. Assume g depends on a. Then, if $C = a$, $g = aA + a'B$. So,

$$\begin{aligned} f &= ag + a'h \\ &= a(aA + a'B) + a'h \\ &= a(A) + a'h. \end{aligned}$$

If $A \neq a$, set $g_1 = A$, otherwise set $g_1 = 1$. In either case, $f = ag_1 + a'h$, g_1 being independent of a and realizable (trivially) by a 2-1 MUX. If $C \neq a$, let $A = a$ (the case $B = a$ is similar). Then, $g = Ca + C'B$. So,

$$\begin{aligned} f &= ag + a'h \\ &= a(Ca + C'B) + a'h \\ &= a(C1 + C'B) + a'h. \end{aligned}$$

If $B \neq a$, set $g_1 = C + C'B = C + B$. Otherwise, set $g_1 = C + C' = 1$. Once again, g_1 is independent of a, and realizable by a 2-1 MUX.

In other words, if g depends on a, to obtain the desired g_1, delete any cube in g having the literal a'. Also replace each literal a by 1. Then, g_1 is independent of a, is realizable by a 2-1 MUX, and satisfies $f = ag_1 + a'h$. Use similar arguments on h to obtain the desired h_1. ■

The proposition tells us that whenever f is realizable by one *act1* module with a single input a at the OR gate, and g or h or both depend on a, an alternate realization is possible, as in (8.2), with g_1 and h_1 independent of a. Since we are interested only in finding *one* way of implementing f rather than *all*, in this case (of single input a at the OR gate) the problem can

be reduced to the simpler problem of finding g_1 and h_1 (or alternately, g and h) which are independent of a. So, without loss of generality, g and h can be assumed to be independent of a. Then, cofactoring (8.1) with respect to a and a' yields

$$g = f_a,\ h = f_{a'}. \tag{8.3}$$

2. *The OR gate inputs are tied to a and b, $a, b \in \sigma(f)$.* Then, f has a decomposition of the form

$$f = (a + b)g + a'b'h, \tag{8.4}$$

where g and h are each realizable by a 2-1 MUX. The problem is once again to find g and h. The following proposition provides a way to find h.

Proposition 8.5.2 *If there exists a decomposition of f as in (8.4), with g and h realizable by a 2-1* MUX *each, then there exists another decomposition of f of the form*

$$f = (a + b)g + a'b'h_1, \tag{8.5}$$

where h_1 is independent of the inputs a and b and is realizable by a 2-1 MUX.

Proof The proof is similar to that for Proposition 8.5.1. We first write h as $h = CA + C'B$ and then observe that any cubes with literals a or b can be removed from h (since h is ANDed with $a'b'$ in (8.4)). Also, if present, the literals a' and b' can be simply deleted from a cube (i.e., replaced by 1). Let the new function be h_1. It satisfies (8.5) and is independent of the inputs a and b. It is easy to see that h_1 is also realizable by a 2-1 MUX. ■

This means that, without loss of generality, we can assume h to be independent of a and b. Then, from (8.4), $h = f_{a'b'}$. The problem now reduces to finding a g that is 2-1 MUX realizable. We divide it into two cases.

(a) *g is independent of a, b.* The following proposition gives necessary and sufficient conditions for this to happen.

Proposition 8.5.3 *A decomposition of f as in (8.4), with g independent of a and b, exists if and only if $f_a = f_b$.*

Proof Note that

$$f_a = f_b \text{ if and only if } f_{ab} = f_{a'b} = f_{ab'} \tag{8.6}$$

The if part follows from considering Shannon expansion of f_a and f_b with respect to b and a respectively. The only if part follows from the fact that f_a and f_b are independent of both a, b (since $f_a = f_b$).

($\Rightarrow$) Assume

$$\begin{aligned} f &= (a+b)g + a'b'h && (8.7)\\ &= abg + a'bg + ab'g + a'b'h && (8.8)\end{aligned}$$

Cofactoring (8.8) with respect to ab, ab', and $a'b$, and using the fact that g is independent of a and b,

$$g = f_{ab} = f_{ab'} = f_{a'b} \tag{8.9}$$

From (8.6), the result follows.

($\Leftarrow$) Doing the Shannon expansion of f with respect to a and then b,

$$f = abf_{ab} + a'bf_{a'b} + ab'f_{ab'} + a'b'f_{a'b'} \tag{8.10}$$

Using (8.6) in (8.10),

$$\begin{aligned} f &= (ab + a'b + ab')f_{ab} + a'b'f_{a'b'}\\ &= (a+b)g + a'b'h\end{aligned}$$

where $g = f_{ab} = f_{a'b} = f_{ab'}$ and $h = f_{a'b'}$. Since f_{ab} is independent of a and b, so is g. ■

Then, from (8.4), it follows that $g = f_a$.

(b) *g depends on a or b or both.* Cofactoring (8.4) with respect to ab, $a'b$, and ab', we get $g_{ab} = f_{ab}$, $g_{a'b} = f_{a'b}$, $g_{ab'} = f_{ab'}$. Then, from the Shannon expansion of g,

$$\begin{aligned} g &= g_{ab}ab + g_{a'b}a'b + g_{ab'}ab' + g_{a'b'}a'b' && (8.11)\\ &= f_{ab}ab + f_{a'b}a'b + f_{ab'}ab' + g_{a'b'}a'b' && (8.12)\\ &= G + Ha'b'. && (8.13)\end{aligned}$$

Here, $G = f_{ab}ab + f_{a'b}a'b + f_{ab'}ab'$ and $H = g_{a'b'}$. Note that

$$\begin{aligned} G_{a'b'} &= 0, && (8.14)\\ H_{a'b'} &= H. && (8.15)\end{aligned}$$

Given f and a choice of a and b, we are interested in finding a g that is MUX-realizable. g will be known if G and H are known. We already know G (since we know f), but we do not know H. To find H, note that H does not affect f, since f is obtained by ANDing $g = G + Ha'b'$ with $(a + b)$. So H is a don't care for f. However, we are interested in finding only those choices of H that make g realizable by a 2-1 MUX. Writing g as a MUX implementation, we get $g = CA + C'B$, where $A, B, C \in \{0, 1\} \cup \sigma(f)$. Since $H = g_{a'b'}$, we get

$$H = (CA + C'B)_{a'b'} = C_{a'b'}A_{a'b'} + (C_{a'b'})'B_{a'b'}. \tag{8.16}$$

If we use this formula as such to compute H, there could be 224 possibilities (when $|\sigma(f)| = 8$). However, we observe that at least one of $A, B, C \in \{a, b\} = V$. This is because, by assumption, g depends on a or b. If $C \in V$, then $H = B_{a'b'}$. If $A \in V$, $H = C'_{a'b'}B_{a'b'}$. If $B \in V$, $H = C_{a'b'}A_{a'b'}$. Thus

$$H \in \{0, 1, c, c', cd, c'd\},\ c, d \in \sigma(f) - \{a, b\},\ c \neq d. \tag{8.17}$$

This reduces the possible choices of H to at most $1 + 1 + 6 + 6 + 15 + 30 = 59$ (when $|\sigma(f)| = 8$). These can be reduced further by examining the cases when $H = cd$ and when $H = c'd$. When $H = cd, g = G + cda'b'$. Since g is MUX-realizable, it has at most 3 inputs. Hence, g does not depend on at least one of a, b, c, d.

i. If g does not depend on c,

$$\begin{aligned} g_c &= g_{c'} \\ G_c + da'b' &= G_{c'} \\ G_{ca'b'} + d &= G_{c'a'b'} \\ d &= 0 \text{ (since } G_{a'b'} = 0 \text{ from (8.14))}, \end{aligned}$$

which is a contradiction. So g depends on c.

ii. Similarly g depends on d.

iii. If g does not depend on a,

$$\begin{aligned} g_a &= g_{a'} \\ G_a &= G_{a'} + cdb' \\ G_{ab'} &= cd \\ f_{ab'} &= cd \text{ (since } G_{ab'} = f_{ab'}) \end{aligned}$$

iv. If g does not depend on b, we can similarly show that

$$f_{a'b} = cd$$

So either $f_{a'b} = cd$ or $f_{ab'} = cd$. Similarly, if $H = c'd$, either $f_{ab'} = c'd$ or $f_{a'b} = c'd$. Hence, for the case of a two-input H, we just examine $f_{ab'}$ and $f_{a'b}$. If any of these are of the form cd or $c'd$, we consider that form for H. This reduces the number of possibilities from 59 to at most 16.

The Matching Algorithm

We first give a subroutine that checks if a function f with a given prime and irredundant cover C, is realizable by a 2-1 multiplexor. Such an f should have at most 3 inputs. Also $|C| \leq 2$. The check is based on $|\sigma(f)|$.

1. If $|\sigma(f)| \leq 1$, the function is realizable.
2. If $|\sigma(f)| = 2$, say x and y are the inputs, then the possible functions are $x + y, x + y', x' + y, xy, xy', x'y$, and these are easy to check for.
3. If $|\sigma(f)| = 3$, then f should be of the form $xy + x'z$, i.e., there should be exactly two cubes in C, each with two literals, with one input x occurring in both positive and negative phases, and the other two inputs y and z occurring exactly once in the positive phase.

We can now describe the matching algorithm for *act1*. Its outline is shown in Figure 8.10.

(0) If $|\sigma(f)| = 0$, return the match for the constant function f. If $|\sigma(f)| > 8$, no match exists; quit.

(i) Compute $T_3(f) = \{a | a \in \sigma(f), |\sigma(f_a)| \leq 3\}$. We can restrict the potential OR gate inputs to $T_3(f)$, i.e., if $a \in \sigma(f)$ is an input to the OR gate for an *act1*-realizable function f, then $a \in T_3(f)$. This follows from (8.1) and (8.4) by noting that $f_a = g_a$ and g, being realizable by a 2-1 MUX, has at most 3 inputs.

(ii) *Single input at the OR gate*: Check if an input $a \in T_3(f)$ may be put along with a 0 at the OR gate. The check is performed by seeing if f_a and $f_{a'}$ are each realizable by a 2-1 MUX each (using (8.3)). If so, report a match and quit. Otherwise, pick another $a \in T_3(f)$ and check likewise. When all the choices are exhausted, go to the next step.

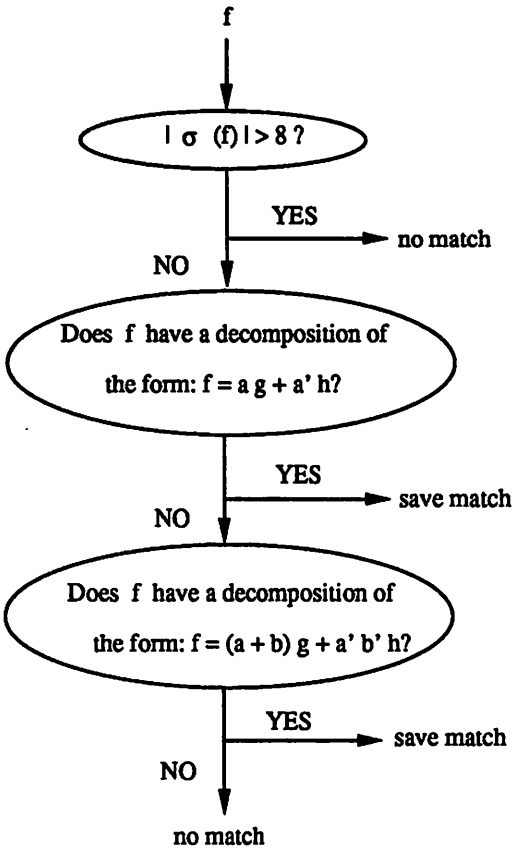


Figure 8.10: Matching algorithm for *act1*

(iii) *Two inputs at the OR gate and g independent of these two inputs*: Select a pair of inputs a and $b, a, b \in T_3(f)$, at the OR gate. From Proposition 8.5.3, f_a should be equal to f_b. This check is fast, since f_a and f_b are functions of at most 3 inputs. If they are equal, and f_a and $f_{a'b'}$ are each realizable by a 2-1 MUX, we know f is realizable by one *act1* block. Otherwise, pick another a, b pair and do the check.

(iv) *Two inputs at the OR gate and g depends on at least one of them*: For each pair $a, b \in T_3(f)$, first check for MUX realizability of $f_{a'b'}$. If successful, look for g that is realizable by a 2-1 MUX and depends on either a or b. For that, go through all the possibilities (at most 16) for H one by one and see if $g = G + Ha'b'$ is MUX realizable.

In steps (ii), (iii) and (iv), we first obtain a prime and irredundant cover of the candidates for g and h before calling the subroutine for MUX-realizability, as it is a pre-condition for our

subroutine. From the discussion, it follows that

Proposition 8.5.4 *If a completely specified function f is expressed on minimum support, then the above procedure is complete for* act1*, i.e., it generates a match if and only if f is realizable by one* act1 *module.*

Note that if f is not expressed on minimum support, the algorithm may not find a match that otherwise exists. A simple example is the tautology function $f = \mathbf{1}$ expressed as a function of greater than 8 variables.

Using just the steps (i), (ii) and (iii), we may miss a match. For example, $f = (a+b)(a'c+ab)+a'b'(xy+x'z)$ is realizable by *act1*, as shown in Figure 8.11. However, (ii) and (iii) fail to find a match. $T_3(f) = \{a, b\}$. Step (ii) fails to generate a match. Since $f_{a'} = bc + b'(xy + x'z)$ is not realizable by a 2-1 MUX, a cannot be the only input to the OR gate. Similarly, $f_{b'} = a'(xy+x'z)$ is not realizable by a 2-1 MUX, and hence b also cannot be the only input to the OR gate. So, we move to step (iii). The only candidates for the OR gate inputs are a and b. However, since $f_a \neq f_b$, step (iii) also fails to generate a match. Thus, f can be matched to *act1* only when the OR gate inputs are a and b, and g depends on a or b or both. This example also demonstrates that a method based on cofactoring cannot guarantee optimality for mapping a function onto minimum number of *act1* modules. This is because after cofactoring f with respect to a and b, the newly-created function g will be independent of a and b. However, for the simplified *act* module, which does not have the OR gate, cofactoring suffices. For, a function f realizable by one *act* module has a decomposition of the form (8.1), and from Proposition 8.5.1, g and h can be assumed independent of a: $g = f_a$ and $h = f_{a'}$. Hence there exists a BDD for f which is isomorphic to one of the patterns of Figure 8.4. We are assuming, of course, that f cannot be realized with 0 modules. This incidentally completes the proof of Proposition 8.3.3.

8.5.2 The Matching Problem for *act2*

The *act2* architecture (Figure 8.1) is slightly more difficult to handle than *act1*, because, unlike *act1*, MUX1 and MUX2 cannot be treated separately - they have a common select line, and the extra AND gate complicates the matching check. This is similar to the complications arising from the presence of the OR gate in the *act1* block.

Definition 8.5.1 *Two functions g and h are* **common-select MUX realizable** *if each can be realized with a 2-1 multiplexor, with the same select line (which may be a constant).*

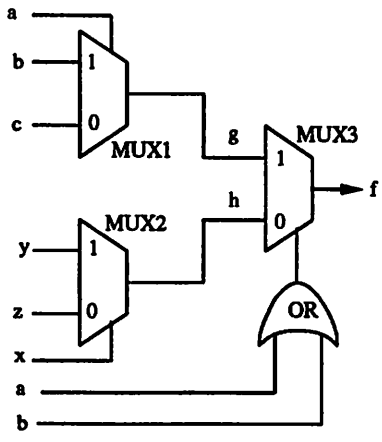


Figure 8.11: An *act1*-realizable function that escapes steps (ii) and (iii) of the matching-algorithm

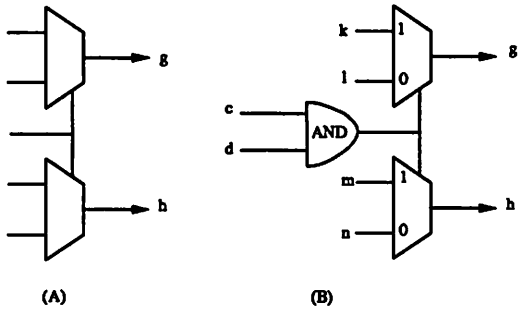


Figure 8.12: Common-select and common_AND-select MUX realizable functions

Such functions have a realization as shown in Figure 8.12 (A).

Definition 8.5.2 *Two functions g and h are* **common_AND-select MUX realizable** *if both g and h can be realized with a 2-1 multiplexor each, such that the select line for both the multiplexors is the same and is equal to AND of some inputs c, d, where* $c, d \in \sigma(g) \cup \sigma(h), c \neq d$ *(so c and d cannot be the constants 0 and 1*[5]*).*

Such functions have a realization as shown in Figure 8.12 (B).

Given two functions g and h, it is natural to ask if they are common-select MUX realizable or common_AND-select MUX realizable. We now give methods that answer these questions.

Problem 8.5.1 *Given two functions g and h, are they common-select* MUX *realizable?*

[5]The cases when the constants 0 and 1, or $c = d$ are allowed are handled by common-select MUX realizable functions.

Solution First find out all possible select line inputs C_g under which g can be realized by a multiplexor. If a function g is MUX-realizable, either

1. there are no restrictions on the selector function, i.e., g is 0, 1 or some input, so tie g to 0 and 1 pins of the MUX, or
2. there is exactly one candidate, x, for the select line, i.e., g is one of the following forms: $x', x'y, x' + y, xy + x'z$, or
3. there are exactly two candidates, x and y, for the select line, i.e., g is of the form $x + y$ or xy.

These candidates form C_g. Similarly find C_h for h. Compute $C_g \cap C_h$. Then $C_g \cap C_h \neq \phi$ if and only if g and h are common-select MUX realizable. ■

Problem 8.5.2 *Given two functions g and h, are they common_AND-select* MUX *realizable?*

Solution Let $\sigma(g)$ and $\sigma(h)$ be the set of inputs of g and h respectively. Let $U = \sigma(g) \cup \sigma(h)$. Assume g and h are common-AND_select MUX realizable (Figure 8.12 (B)). Since $c, d \notin \{0,1\}, |U| \geq 2$. Then,

$$g = cdk + (c' + d')l, \tag{8.18}$$

$$h = cdm + (c' + d')n, \tag{8.19}$$

$$c, d \in U, \quad c \neq d, \text{ and } k, l, m, n \in U \cup \{0,1\} \tag{8.20}$$

We let c, d vary over U. Given g, h, c, and d, we may assume k, m to be independent of c and d, i.e., $k, m \notin \{c, d\}$. If they are not, we can replace them by 1 without changing g, h. We now present necessary and sufficient conditions for g and h to be common_AND-select MUX realizable for a given c, d pair. The corresponding values of k, l, m, and n are also given.

1. $g_{cd} \in \{0,1\} \cup (\sigma(g) - \{c, d\})$ (set $k = g_{cd}$), and
2. $h_{cd} \in \{0,1\} \cup (\sigma(h) - \{c, d\})$ (set $m = h_{cd}$), and
3. Exactly one of the following should hold:
   (i) $g_{c'} = 0$ and $g_{d'} = c$. (Set $l = c$).
   (ii) $g_{c'} = 0$ and $g_{d'} = 0$. (Set $l = 0$).

(iii) $g_{c'} = 1$ and $g_{d'} = 1$. (Set $l = 1$).

(iv) $g_{c'} = d$ and $g_{d'} = 0$. (Set $l = d$).

(v) $g_{c'} = z$ and $g_{d'} = z$. Here $z \in \sigma(g) - \{c, d\}$ (Set $l = z$), and

4. Exactly one of the following should hold:

(i) $h_{c'} = 0$ and $h_{d'} = c$. (Set $n = c$).

(ii) $h_{c'} = 0$ and $h_{d'} = 0$. (Set $n = 0$).

(iii) $h_{c'} = 1$ and $h_{d'} = 1$. (Set $n = 1$).

(iv) $h_{c'} = d$ and $h_{d'} = 0$. (Set $n = d$).

(v) $h_{c'} = z$ and $h_{d'} = z$. Here $z \in \sigma(h) - \{c, d\}$. (Set $n = z$).

We just present a sketch of the proof. For necessity, assume (8.18), (8.19), and (8.20). Cofactoring (8.18) and (8.19) with respect to cd gives the first two conditions. Cofactoring (8.18) with respect to c' and d', and using the fact that $l \in U \cup \{0, 1\}$, we get the third condition. Likewise we get the fourth one from (8.19). For sufficiency, do Shannon expansion of g and h with respect to c and d.

$$\begin{aligned} g &= g_{cd}cd + g_{cd'}cd' + g_{c'd}c'd + g_{c'd'}c'd' \\ h &= h_{cd}cd + h_{cd'}cd' + h_{c'd}c'd + h_{c'd'}c'd' \end{aligned}$$

Using the conditions 1-4 in these two equations, it is easy to see that k, l, m, and n can be appropriately selected, as written parenthetically above, to give the desired implementation of g and h as (8.18) and (8.19) respectively.

These conditions are easy to check. Since $|U| \leq 6$, in the worst case, they have to be checked for all 15 c, d pairs. ■

We are now ready to answer the original question: given f, is f realizable by one *act2* block? The strategy is to assume that f is realizable by one *act2* block and derive all possible function pairs g and h (Figure 8.13). Then depending on the number of proper (i.e., non-constant) inputs at the AND gate, check if g and h are common-select MUX realizable or common_AND-select MUX realizable. Refer to Figure 8.13 for the rest of this subsection. It suffices to consider the following five cases:

1. *One proper input, a, at the OR gate and at most one proper input, c, at the AND gate*: If f is realizable by a single *act2* block, then $f = ag + a'h$, $g = ck + c'l$, $h = cm + c'n$. Note

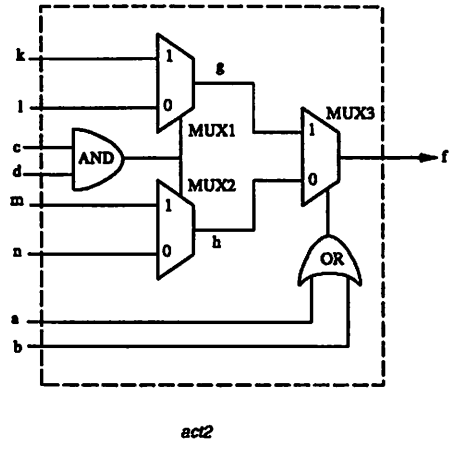


Figure 8.13: An *act2*-realizable function f

that g and h are common-select MUX realizable. It is enough to check for the case when g and h are independent of a For, if either k or $l = a$, replace it/them by 1. If any of $m, n = a$, replace it/them by 0. Then, if $c = a$,

$$\begin{aligned} f &= a(ak + a'l) + a'(am + a'n) \\ &= ak + a'n. \end{aligned}$$

If $g_1 = k$ and $h_1 = n$, then g_1, h_1 are common-select MUX realizable, are independent of a, and can replace g and h without changing f. Then $g = f_a, h = f_{a'}$. Finally check if g and h are common-select MUX realizable.

2. *One proper input, a, at the OR gate and two proper inputs, c and d, at the AND gate, $c \neq d$*: If f is realizable by a single *act2* block, $f = ag + a'h, g = cdk + (c' + d')l, h = cdm + (c' + d')n$. Again, without loss of generality, we can assume that g, h are independent of a. For, if $k, l, m, n = a$, they may be replaced by appropriate constants 0 or 1. Then if $c = a$ (case $d = a$ is handled similarly), we get

$$\begin{aligned} f &= a(adk + (a' + d')l) + a'(adm + (a' + d')n) \\ &= a(dk + d'l) + a'n. \end{aligned}$$

Set $g_1 = dk + d'l, h_1 = n = dn + d'n$. Then replacing g and h by g_1 and h_1 respectively results in a decomposition covered by case 1. So, g, h can be assumed independent of a. Then $g = f_a, h = f_{a'}$. Finally check if g and h are common_AND-select MUX realizable.

3. *Two proper inputs, a and b, at the OR gate, $a \neq b$, and at most one proper input, c, at the AND gate*: If f is realizable by a single *act2* block, $f = (a+b)g+a'b'h$, $g = ck+c'l$, $h = cm+c'n$. h may be assumed independent of a, b. For, if either m or $n \in \{a, b\}$, they may be replaced by 0. Then if $c \in \{a, b\}$, h may be replaced by $h_1 = n$, which is independent of a, b and is common-select MUX realizable with any MUX realizable function. This implies that $h = f_{a'b'}$. For g, we need to consider two cases:

   - *g is independent of a, b*: Then f_a should be equal to f_b, and $g = f_a$ (Proposition 8.5.3). We then check if g and h are common-select MUX realizable.
   - *g depends on either a or b or both*: As shown earlier in (8.13), $g = G + Ha'b'$. Then, as before, for g to be MUX-realizable, we have at most 16 choices for H. For each such choice, we compute the corresponding g and check if g and h are common-select MUX realizable. We quit if we get a match (i.e, a yes answer), otherwise we consider the next choice of H and repeat this step.

4. *Two proper inputs, a and b, at the OR gate, $a \neq b$, and two proper inputs, c and d, at the AND gate, $c \neq d$*: If f is realizable by a single *act2* block,

$$f = (a+b)g + a'b'h \tag{8.21}$$

$$g = cdk + (c'+d')l \tag{8.22}$$

$$h = cdm + (c'+d')n. \tag{8.23}$$

   h may be assumed independent of a, b. For, m, n can be made independent of a, b. Then, if $c = a$,

$$\begin{aligned} f &= (a+b)g + a'b'h \\ &= (a+b)g + a'b'(adm + (a'+d')n) \\ &= (a+b)g + (a'b'n + a'b'd'n) \\ &= (a+b)g + a'b'n. \end{aligned}$$

   Then h may be replaced by $h_1 = n$, which is independent of a, b and is common_AND-select MUX realizable with any g of (8.22).

   This implies that $h = f_{a'b'}$. For g, we need to consider two cases:

   - *g is independent of a, b*: Then f_a should be equal to f_b and $g = f_a$ (Proposition 8.5.3). We check if g and h are common_AND-select MUX realizable.

- *g depends on either a or b*: As done earlier, we see that $g = G + Ha'b'$. Also, from (8.22), $g = cdk + (c' + d')l$. Here, $c, d \in \sigma(f)$ and $k, l \in \{0,1\} \cup \sigma(f)$. Since $G_{a'b'} = 0$, and $H = g_{a'b'}$ is independent of a, b, we get

$$H = c_{a'b'}d_{a'b'}k_{a'b'} + (c'_{a'b'} + d'_{a'b'})l_{a'b'}.$$

Since g depends on $\{a, b\} = V$, we consider the following cases:

(a) If $c \in V, c_{a'b'} = 0$. Then $H = l_{a'b'}$. Similarly handle the case $d \in V$.

(b) Otherwise,

- if $k \in V, H = (c'_{a'b'} + d'_{a'b'})l_{a'b'}$.
- if $l \in V, H = c_{a'b'}d_{a'b'}k_{a'b'}$.

We then get

$$H \in \{0, 1, x, x'y, x' + y', (x' + y')z, xy, xyz\}, \tag{8.24}$$

$$x, y, z \in \sigma(f) - V, \tag{8.25}$$

$$x \neq y, y \neq z, x \neq z \tag{8.26}$$

As in the algorithm for *act1*, we investigate the cases $H = xyz$ and $H = (x' + y')z$. Let $H = xyz$. Then $g = G + xyza'b'$. But g has at most 4 inputs. So, g must be independent of at least one of x, y, z, a, b. We can then show that either $f_{ab'} = xyz$ or $f_{a'b} = xyz$.

(a) If g does not depend on x, we get

$$\begin{aligned} g_x &= g_{x'} \\ G_x + yza'b' &= G_{x'} \\ G_{xa'b'} + yz &= G_{x'a'b'} \\ yz &= 0 \end{aligned}$$

Not possible. Hence g depends on x.

(b) Similarly g depends on y and z.

(c) If g does not depend on a,

$$\begin{aligned} g_a &= g_{a'} \\ G_a &= G_{a'} + xyzb' \\ G_{ab'} &= xyz \\ f_{ab'} &= xyz \end{aligned}$$

Similarly, if g does not depend on b, $f_{a'b} = xyz$.

Similarly, if $H = (x' + y')z$, either $f_{ab'} = (x' + y')z$ or $f_{a'b} = (x' + y')z$. Hence, for a three-input H, we just need to examine $f_{ab'}$ and $f_{a'b}$. Only if any of these are of the form xyz or $(x' + y')z$, we need to consider that form for H. This reduces the possible choices of H to at most 1 + 1 + 6 + 30 + 15 + 1 + 15 + 1 = 70. For each such H, we obtain a g and check if g and h are common_AND-select MUX realizable.

5. *The output of the OR gate is a constant (0 or 1)*: So $f = g$ or $f = h$. Then if f is realizable by a single *act2* block, either $f = x$ ($x \in \sigma(f)$), or $f = ck + c'l$, or $f = cdk + (c' + d')l = c(dk + d'l) + c'l$. All these subcases are covered by case 1.

Note that in each of the above cases, we need to vary a, b over $T_4(f) = \{x | x \in \sigma(f), |\sigma(f_x)| \leq 4\}$, and c, d over $\sigma(f)$, wherever appropriate. Also note that as for *act1*, each time we obtain a sub-function of f, we first derive a prime and irredundant cover before performing any checks. From the above discussion, it follows that

Proposition 8.5.5 *If a completely specified function f is expressed on minimum support, then the above procedure is complete for* act2.

8.6 Constructing Subject Graph and Pattern Graphs using ITEs

We construct an unordered ITE for a function keeping in mind that it is to be mapped to *act1*. Say we are given the cover (or SOP) C of a function f. The overall procedure for constructing the ITE for f is as follows:

1. If f is identically 0 or 1, its ITE is just one vertex, which is a terminal vertex with value 0 or 1.

2. If f is a single cube, order the variables as per Proposition 8.3.2. If it is a sum of single-literal cubes, order the variables as per Proposition 8.3.1. In either case, construct the chain ITE.

3. Check if f can be realized by a single *act1* block. This is done by invoking the matching algorithm of Section 8.5.1. If f is realizable, construct its ITE appropriately: each MUX of *act1* is represented by a non-terminal vertex in the ITE. The OR gate of *act1* can also be represented in the ITE, as will be shown in Section 8.6.5.

4. If all the cubes in the cover C of f have mutually disjoint supports, use the procedure of Section 8.6.1 to construct the ITE.

5. If C is unate, construct the ITE using the method of Section 8.6.2.

6. C is binate. First select variables that occur in all the cubes in the same phase. Next, a branching variable x is selected. Compute the three *algebraic cofactors* of f with respect to x and construct their ITEs recursively using this procedure. These ITEs are combined to get the ITE for f. The details are in Section 8.6.3.

We now present details of the steps 4, 5, and 6. The rest of the steps are either straightforward or have been described already.

8.6.1 Creating ITE for a Function with Disjoint Support Cubes

In any tree implementation of f, the cubes have to be realized first and then ORed together. So one way to construct the ITE for f is to first construct an ITE for each cube, and then OR the cube-functions. The ITE for each cube may be constructed using Proposition 8.3.2. The subsequent ORing can then be done using Proposition 8.3.1. This procedure, however, ORs the cube ITEs arbitrarily. Better results are possible if the cube ITEs are combined carefully. The improved procedure is as follows.

1. Construct an ITE for each cube function using Proposition 8.3.2.

2. Determine for each cube whether after mapping its ITE, the output multiplexor, MUX3, of the root *act1* module is used, i.e., some input of the OR gate is a proper input of the cube. If both the inputs to the OR gate are constants, the cube can be realized without the OR gate and MUX3. For instance, as shown in Figure 8.15, MUX3 is unused for the cube ab (the inputs to the OR gate at MUX3 are constant) and is used for the cube cde. Determining if MUX3 will be used is easy and the corresponding procedure is given in Figure 8.14. It is a function of numbers of positive and negative literals in the cube. The procedure mimics ITE mapping. The procedure is initially called with `count = 0`, which means that no module has been used yet. Some positive and negative literals are selected for each *act1* module. The `count` is set to 1 when at least one module has been used. From then on, it remains 1. The case `count = 1` is handled separately, because for a cube function there is a slight difference in the way a leaf *act1* is handled as compared to other *act1* modules.

After this step, the cubes (and hence their ITEs) are partitioned into two sets: `free` - those with the root multiplexor unused, and `used` - those with the root multiplexor used.

3. One ITE from the `free` set and two from the `used` set are picked and ORed together. The ITEs corresponding to the `used` cubes are mapped to the inputs of the OR gate of the *act1* block whose output multiplexor is unused. The `free` ITE is an input to this output multiplexor. The resulting ITE is placed in the `used` set.

   If the required number of `free` or `used` ITEs are not available, suitable modifications are made. For example, if no `free` ITEs are present, four `used` ITEs are picked and ORed. This corresponds to using an extra *act1* module: two ITEs feed the OR gate of the new *act1* module and the other two, MUX2. On the other hand, if no `used` ITEs are present, three `free` ITEs are picked. This corresponds to using the OR gate inputs of the root *act1* module of one of the ITEs. In either case, the resultant ITE is placed in the `used` set.

   This step is repeated until just one ITE is left unpicked.

Example 8.6.1 *Consider* $f = ab + cde + g'h'ij'$. *Let* $c_1 = ab, c_2 = cde, c_3 = g'h'ij'$. *For* $c_1, p = 2, q = 0$. *Then, from Figure 8.14, it can be seen that* c_1 *is in* `free`. *Similarly, for* c_2, $p = 3$ *and* $q = 0$, *so* c_2 *belongs to* `used`. *For* c_3, $p = 1$ *and* $q = 3$, *so* c_3 *is in* `used` *as well. The corresponding realizations for the three cubes are shown in Figure 8.15. The next step is to realize* f *using these realizations. Figure 8.16 shows how the unused MUX3 of* c_1*'s block is used to realize* f *by connecting* c_2 *and* c_3 *to the OR gate inputs of block 1. Note that to realize the desired OR of* $c_2 + c_3$ *with* c_1, c_1 *switches from MUX1 to MUX2.*

8.6.2 Creating ITE for a Unate Cover

In the spirit of the *unate recursive paradigm* of ESPRESSO [11], a unate cover is handled specially. One of the following methods is used to generate an ITE for a unate cover C of the function f.

1. *use factored form*: Generate a factored form of f, say by using *decomp -g* on the SOP C. Unless the SOP consists of cubes with disjoint supports, there will be at least two sub-functions in the resulting decomposition. This process is repeated until each of the resulting sub-covers consists of either a single cube or a set of cubes with disjoint supports. In either case, ITEs are constructed as described earlier. These ITEs are combined to get the ITE for f.

```
/* return 1 if the top mux is unused, else return 0 */
/* p = number of +ve literals,
   q = number of -ve literals in a cube */
int is_top_mux_unused(p, q, count)
  int p, q, count;
{
  int q_3 = q % 3;
  if (p == 0 && q == 0) return 0;
  if (count == 0) {
      if (p == 1 && q == 0) return 0;
      if (q == 0)
          if (p == 2) return 1; else return ((p - 3) % 2);
      if (p == 0)
          if (q_3 == 0 || q_3 == 2) return 0; else return 1;
      if (q == 1)
          if (p == 1) return 1; else return ((p - 2) % 2);
      if (p == 1)
          if (q_3 == 0 || q_3 == 2) return 0; else return 1;
      return is_top_mux_unused(p - 2, q - 2, 1);
  }
  if (q == 0) return (p % 2);
  if (p == 0)
      if (q_3 == 0 || q_3 == 2) return 0; else return 1;
  if (q == 1) return ((p - 1) % 2);
  return is_top_mux_unused(q - 2, p -1 , 1);
}
```

Figure 8.14: For a cube, is the top multiplexor unused?

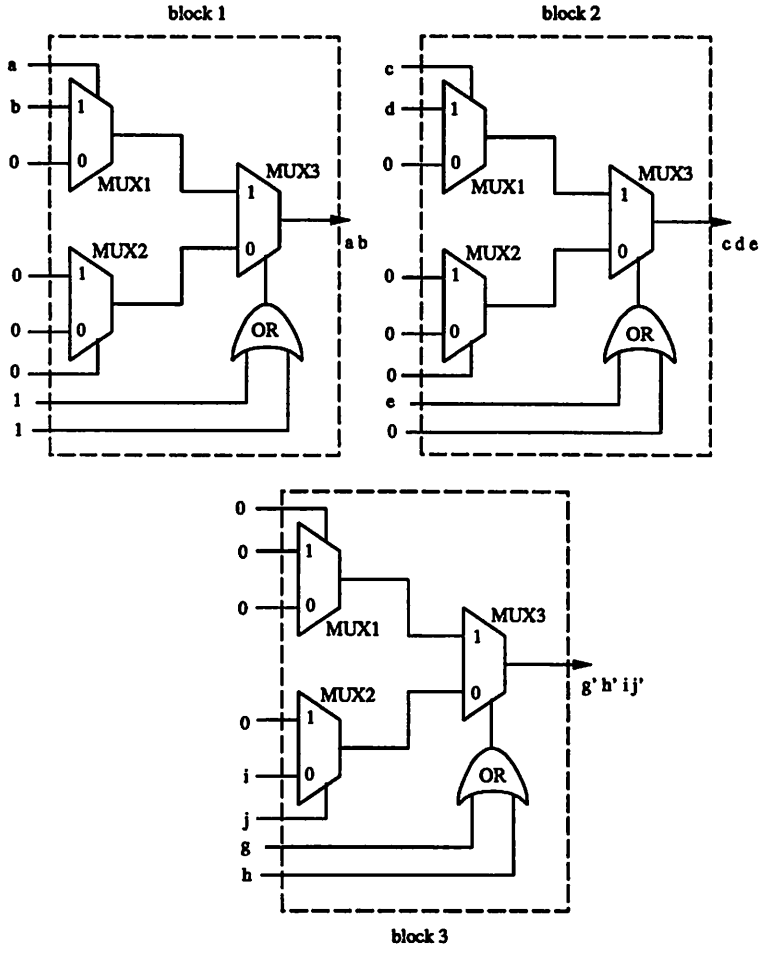


Figure 8.15: Example: realizing the three cubes

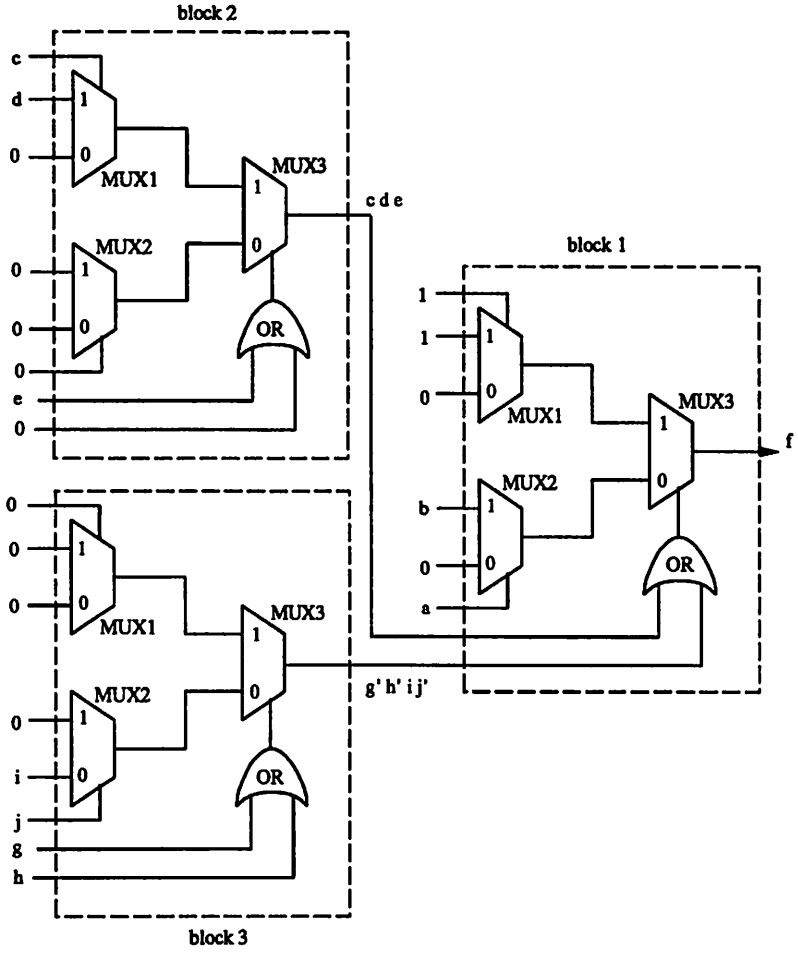


Figure 8.16: Example: using MUX3 of block 1 to realize f

For example, let us assume that the resulting decomposition for f leads to two new functions G and $F(G,\ldots)$. ITEs for G and F are created. Then, in the ITE for F, all the pointers to the variable G are replaced by pointers to the ITE for G, and this yields the ITE for f.

2. *use column cover*: If f is constant - **1** or **0**, return the corresponding ITE constant 1 or 0. If $|C| = 1$, construct ITE for the single cube using the variable ordering of Proposition 8.3.2. If C consists of single-literal cubes, use the variable ordering of Proposition 8.3.1 to get the ITE for f. Otherwise, construct a *0-1* matrix $B = (b_{ij})$ for the unate cover C, where the rows of B correspond to cubes, the columns correspond to the variables, and

$$b_{ij} = \begin{cases} 1 & \text{if the variable } x_j \text{ appears in the } i^{th} \text{ cube of } C, \\ 0 & \text{otherwise.} \end{cases}$$

x_j may be in the positive phase or the negative phase.[6] A minimal/minimum weight column cover CC for B is then obtained [11] (see Section 4.4.1). CC contains some variables $\{x_j\}$ of f. For each variable x_j in CC, a sub-cover SC_j is constructed using cubes of C that depend on x_j. Each cube is put in exactly one such sub-cover (ties are broken arbitrarily). From the sub-cover SC_j, x_j (or x_j') is extracted out to yield a modified sub-cover MSC_j.

$$\begin{aligned} C &= \sum_{x_j \in CC} SC_j \\ &= \Big(\sum_{x_j \in CC,\ x_j \text{ pos. binate}} x_j MSC_j \Big) + \Big(\sum_{x_j \in CC,\ x_j \text{ neg. binate}} x_j' MSC_j \Big) \end{aligned}$$

In MSC_j, x_j is a don't care. This corresponds to creating an ITE vertex v_j for SC_j with the *if* child corresponding to x_j, the *then* child corresponding to MSC_j, and the *else* child corresponding to 0. Then v_j represents an AND operation. This is the case if x_j appears in C in the positive phase. If it appears in the negative phase, just swap the *then* and *else* children of v_j. The column covers for MSC_j are recursively obtained. The ITE of C is then obtained by ORing repeatedly pairs of ITEs corresponding to the sub-covers SC_j.

The weight of a positive unate variable is chosen slightly more than the weight of a negative unate variable. To see why, suppose $f = ab' + ac' + c'd$. If all the variables were given equal weights, we may obtain a column cover $CC_1 = \{a, d\}$. Let $x_1 = a, x_2 = d$. Then $SC_1 = ab' + ac', SC_2 = c'd$. Then we factor f as $f = a(b' + c') + d(c')$. $MSC_1 =$

[6] Recall that since C is a unate cover, each variable x_j appears in it in either the positive phase or the negative phase, but not both.

$b' + c', MSC_2 = c'$. Since MSC_1 and MCS_2 satisfy the *base* cases (they are either a single cube or sum of single-literal cubes), their ITEs are constructed. The ITE for f is constructed from them as shown in Figure 8.17 (A). While ORing SC_1 and SC_2, pointers pointing to the terminal 0 in say SC_2 are changed to point to the ITE for SC_1. After mapping using the patterns to be described in Section 8.6.5, we get the cost as 3 blocks (the dotted rectangle stands for an *act1* block). The reason is that we end up with c', an inverted input, at the leaf of the ITE, and a multiplexor is used up to realize this inversion. However, if the weight of a positive unate variable is slightly more than that of a negative unate variable, we obtain the column cover $CC_2 = \{b', c'\}$. These variables appear as the *if* children of some ITE vertex that implements AND operation and hence their complements are automatically realized. f is then factored as $f = b'(a) + c'(a + d)$. The corresponding ITE is shown in Figure 8.17 (B). Note that d appears in the positive phase towards the bottom of the ITE and does not need a multiplexor. As a result, a cost of 2 is incurred after mapping.[7] By giving lower weights to the negative unate variables, negative variables tend not to be at the leaves of the sub-cover ITEs. We choose the weights such that if the cardinality of a column cover is t, then each cover of size greater than t should have weight greater than t. It is easy to see that weight(negative unate variable) = 1, weight(positive unate variable) = $1 + 1/n$, where f is a function of n variables, satisfy this property.

3. *use the method for binate cover*: In this method, C is treated as if it were binate. A variable is selected at each step, and the algebraic cofactors are computed, for which the ITEs are constructed recursively. The method of selecting the branching variable is described in Section 8.6.3.

8.6.3 Creating ITE for a Binate Cover

Given the binate cover C for f, we first select variables that occur in all the cubes in the same phase. Next, at each step a branching variable x is selected. One of the following procedures is used to select x:

1. *most binate* [11]: Select the variable that occurs most often in the SOP. Priority is given to a variable that occurs in all the cubes in the same phase. Subsequently, ties are broken by

[7] Incidentally, this function can be realized by one block. The match is discovered by the matching algorithm described in Section 8.5.1.

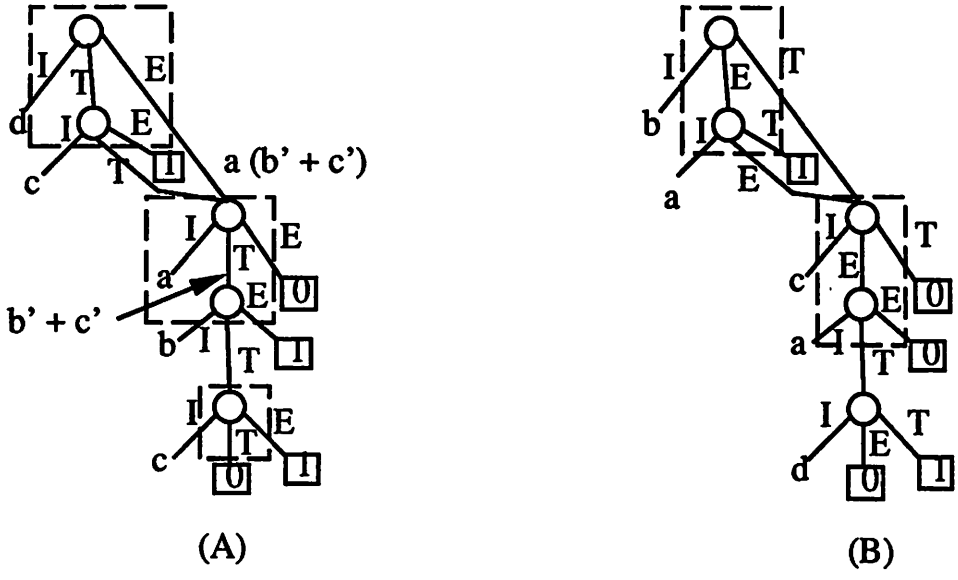


Figure 8.17: Positive unate variables should have higher weights

selecting the variable that occurs nearly equal number of times in the positive and negative phases.

2. *quick map*: For each input variable v, compute the three **algebraic cofactors** C_0, C_1, and C_2 of f with respect to v as follows:

$$f = C_1 v + C_0 v' + C_2 \tag{8.27}$$

Example 8.6.2 *Let* $f = abc + a'de + gh + ac'd + g'k$, *and* $v = a$. *Then,*

$$\begin{aligned} C_1 &= bc + c'd, \\ C_0 &= de, \\ C_2 &= gh + g'k. \end{aligned}$$

Compute the cost of each algebraic cofactor by constructing an ITE for it using the *most binate* variable selection heuristic,[8] and then mapping it without any iterative improvement. Finally, sum up the costs of all the cofactors to obtain the cost corresponding to selecting v as the branching variable. Then x is the minimum cost variable.

[8]One could use *quick map* recursively on the cofactor, but the computation becomes time-intensive.

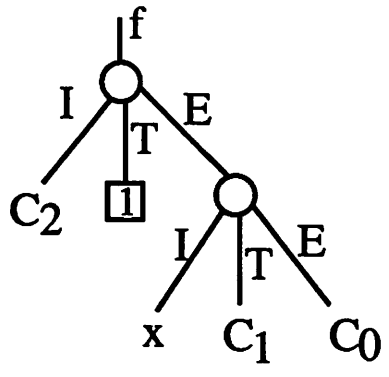


Figure 8.18: Obtaining algebraic cofactors with ITE

After x has been selected, compute the algebraic cofactors of f with respect to x:

$$f = C_1 x + C_0 x' + C_2 \tag{8.28}$$

If we were to construct a BDD for f, f_x and $f_{x'}$ would have to be computed. $f_x = C_1 + C_2$ and $f_{x'} = C_0 + C_2$. So C_2 is replicated in both. While C_2 can be shared between f_x and $f_{x'}$ in theory, in practice it may not always. The currently used heuristics for constructing unordered BDDs do not have an inbuilt mechanism for detection of sharing. As of ROBDDs, a bad input ordering can thwart opportunity for a complete sharing. However, as pointed out by Karplus [38], ITEs avoid the duplication. As shown in Figure 8.18, we can represent f as *if* C_2 *then* **1** *else* (*if* x *then* C_1 *else* C_0). This way we do not have to duplicate any literal of C_2. We illustrated this earlier in Example 2.1.1, but repeat it here for convenience.

Example 8.6.3 *Consider function* $f = ab + a'c + de$. *In Figure 8.19, we show that if* a *is selected as the top variable in the* BDD, $C_2 = de$ *gets replicated in both 0 and 1 branches. This is avoided in the* ITE *by factoring out* de *before branching on* a.

We recursively construct ITEs for C_0, C_1, and C_2 until they become unate, in which case techniques from Section 8.6.2 are used to construct the ITEs. For two special cases, however, we use a slightly different realization. When $C_2 = a'$ or $C_2 = a'b'$, where a, b are inputs of f, we realize f as *if* C_2' *then* (*if* x *then* C_1 *else* C_0) *else* **1**. This is shown in Figure 8.20. Such a realization allows us to save an inverter when $C_2 = a'$, and use the OR gate of *act1* when $C_2 = a'b'$.

Note that this procedure targets *act1* and would have to be modified for other architectures, say *act2*.

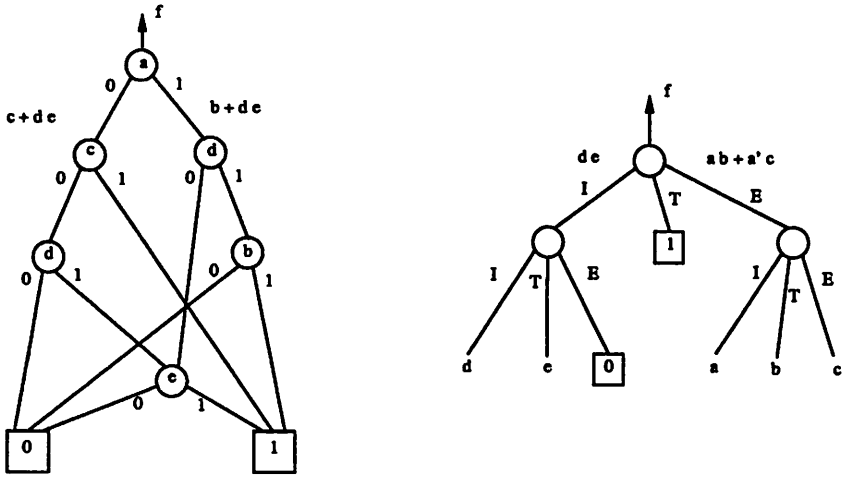


Figure 8.19: BDD vs. ITE

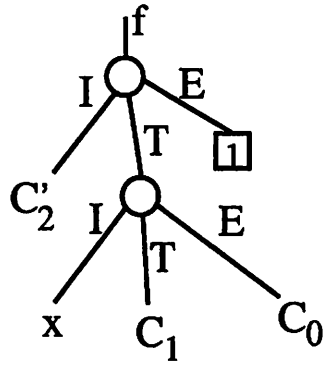


Figure 8.20: Modifying algebraic cofactoring for a special case

8.6.4 Comparing with Karplus's Construction

Our method of constructing the ITE for a function f borrows from Karplus [38] the idea of using algebraic cofactors instead of cofactors. However, we construct the ITEs for special cases differently. We use different techniques to construct the ITE for a single cube, sum of single-literal cubes, cubes having disjoint supports, unate cover, and feasible functions (for which we use the matching algorithm).

8.6.5 Pattern Graphs for ITE-based Mapping

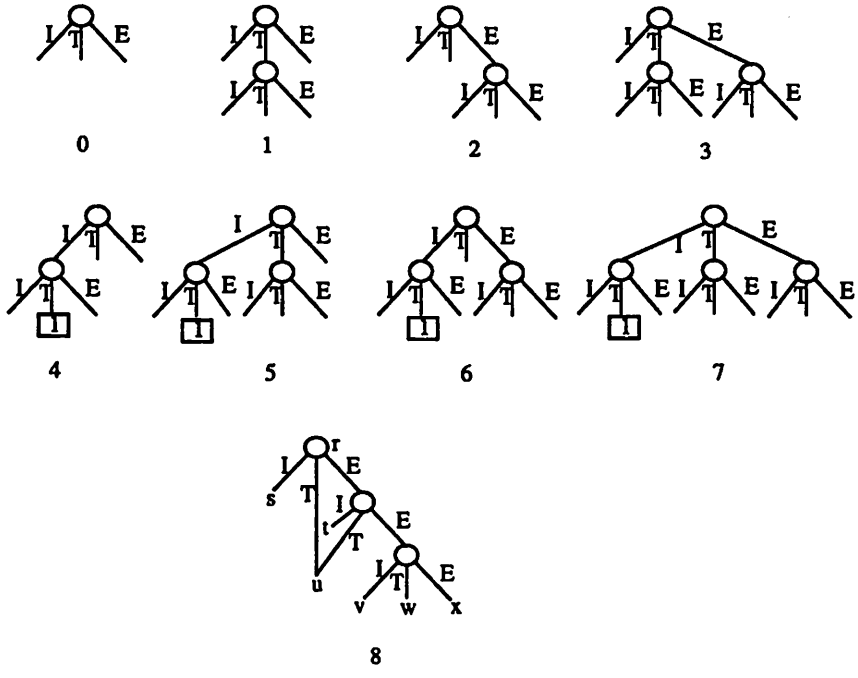


Figure 8.21: Pattern graphs for *act1*

The pattern graphs are also ITEs. As shown in Figure 8.21, we use 9 pattern graphs for obtaining a feasible realization of an infeasible function onto *act1*. The first four pattern graphs (from 0 to 3) correspond to the *act1* without the OR gate and using one, two, or all the three multiplexors. The next four patterns (from 4 to 7) are the same as the first four except that the patterns 4 to 7 use the OR function at the *if* child of the root vertex. The OR function is detected by the presence of the constant 1 at the *then* child of the *if* child of the root vertex. Pattern graph 8 represents a special case derivable from the basic block and uses the OR gate. Note the string of 3 MUXes, which as such cannot be implemented by one *act1* module, since the *act1* module has only two levels of MUXes. But it may be possible to juggle some inputs. Let

r = root vertex of the pattern 8,

$s = if(r)$,

$u = then(r) = then(else(r))$,

$t = if(else(r))$,

$v = if(else(else(r)))$,

$w = then(else(else(r)))$, and

$x = else(else(else(r)))$.

Then the function g implemented by r

$$\begin{aligned} g &= su + s'(tu + t'(vw + v'x)) \\ &= su + s'tu + s't'(vw + v'x) \\ &= su + tu + s't'(vw + v'x) \\ &= (s + t)u + s't'(vw + v'x) \end{aligned}$$

which is realizable by one *act1* module if s and t are mapped to the OR gate inputs.

Note that if we do a technology decomposition either into 2-input AND and OR gates or into 2-1 MUX gates, each function becomes feasible for *act1*, thus obviating the need for pattern graphs. We can then simply use a matching algorithm that is embedded in the covering algorithm. In our paradigm, this is the same as applying one iteration of *partial collapse*. However, as we will show in Section 8.7, much better results are possible if we use the complete algorithm of Section 8.4.

8.7 Experimental Results

First we experiment with simpler options and then apply increasingly sophisticated ones for better quality. The starting networks are the same as those obtained after optimization in Section 3.6.1.

8.7.1 Without Iterative Improvement

This is the simplest option. For each node of the network, an ITE is constructed and then mapped using the pattern graphs. This corresponds to the step 1 of the mapping algorithm of Section 8.4. The results are shown in the column *ite-map* of Table 8.1.

However, it may be possible to collapse some nodes into their fanouts without increasing the fanout costs. This transformation corresponds to global covering and is implemented using the steps 1, 4, and 5 of the mapping algorithm of Section 8.4. In other words, for each node of the network, a feasible implementation is determined. Then each node is replaced by its feasible implementation. Finally, one iteration of *partial collapse* is applied on the resulting network to mimic the global covering of the network. The results are shown in the column *ite-map + cover* and, on comparing with the column *ite-map*, establish that the covering step is useful in reducing the block-count.

example	ite-map	ite-map + cover
5xp1	46	42
9sym	54	53
C1355	166	164
C1908	176	156
C2670	377	340
C3540	552	502
C432	102	92
C5315	658	588
C6288	1226	988
C7552	813	755
alu2	142	134
alu4	352	335
apex2	120	113
apex3	714	697
apex7	94	85
b9	60	56
bw	63	55
clip	45	41
cordic	26	22
dalu	401	366
des	1399	1279
duke2	186	170
e64	116	90
ex4	208	192
f51m	25	23
k2	570	545
misex2	41	37
rd84	56	55
rot	270	246
sao2	60	55
spla	266	246
t481	15	10
vg2	35	33
z4ml	16	13
total	9450	8578

Table 8.1: *act1* count without iterative improvement

ite-map step 1 of the algorithm of Section 8.4
ite-map + cover steps 1, 4, and 5 of the algorithm of Section 8.4
total sum of *act1* counts over all the examples

Initial decomposition

To test the hypothesis that a technology decomposition into two input AND and OR gates is not the best alternative, we performed an experiment using three decomposition techniques.

1. *nand decomp*: Apply *decomp -g* to create a factored representation and then use *tech-decomp -a 2 -o 2* to derive a representation in terms of 2-input AND and OR gates.

2. MUX *decomp*: For each node, construct an ITE and break it up into MUX nodes.

3. LUT *decomp*: It seems a good idea to use the synthesis techniques for LUT architectures to obtain a 3-feasible network. Initially, the motivation was that for functions with a small number of inputs, one could construct ROBDDs for all possible input orderings and pick the best one. We suggested this first in [62]. Karplus, noting that an *act1* module can implement most of the 3-input functions, used the same idea in [38]. Whatever the motivation, the idea is good, as the experiments will prove.

While the first two techniques generate an *act1*-feasible network, the third, LUT *decomp*, may not, since not all 3-input functions can be realized with an *act1* module. So, in this case, an ITE is constructed and mapped, resulting in a feasible implementation.

On the feasible network generated using each technique, an iteration of partial collapse is applied to mimic the covering step. The results obtained thereafter are shown in Table 8.2. About 10% improvement is attained from MUX *decomp* over *nand decomp*. LUT *decomp* yields the best results, attributable to superior LUT mapping techniques. As we demonstrate next, it is possible to further improve the quality by means of the iterative improvement phase.

8.7.2 Iterative Improvement Without Last Gasp

We use steps 1, 2, 4, and 5 of the algorithm. A binate cover is mapped using the *most binate* variable option, and a unate cover, using the *factored form* option. Just one iteration is used in the iterative improvement step, i.e., in step 2. To make the *partial collapse* more effective, a node is considered for *partial collapse* only if it satisfies all of the following conditions:

1. Its cost is at most 3 *act1* modules,

2. it has at most 7 fanins, and

3. after collapsing it into the fanouts, each of its fanouts has at most 50 fanins.

example	nand decomp	MUX decomp	LUT decomp
5xp1	48	43	43
9sym	79	63	19
C1355	216	200	202
C1908	219	195	182
C2670	330	305	238
C3540	618	605	489
C432	123	114	83
C5315	798	766	555
C6288	1458	1264	1079
C7552	1097	905	712
alu2	185	167	152
alu4	475	443	321
apex2	133	129	120
apex3	799	769	-
apex7	104	90	84
b9	56	56	56
bw	65	62	60
clip	59	46	41
cordic	25	23	26
dalu	487	449	385
des	1617	1563	1222
duke2	210	191	184
e64	97	97	90
ex4	233	221	221
f51m	37	31	26
k2	739	660	536
misex2	48	47	45
rd84	78	67	43
rot	287	288	251
sao2	64	61	52
spla	301	291	251
t481	11	10	10
vg2	39	33	34
z4ml	16	18	16
total	11151	10272	-
subtotal	10352	9503	7828

Table 8.2: Initial decomposition

nand decomp apply *decomp -g* and then *tech-decomp -a 2 -o 2*; then cover
MUX decomp decompose into MUX nodes; then cover
LUT decomp get a 3-feasible network; then cover
total sum of *act1* counts over all the examples

Also, in the decomposition step of iterative improvement, only those nodes are considered that have at least 4 fanins. In the rest of the section, all parameters in the mapping algorithm take on same values as just described.

The results are shown in Table 8.3 in the column *ite-map + iter. imp.*, and are compared with the column *ite-map + cover* of Table 8.1. They are, on average, 6% better, confirming the usefulness of iterative improvement.

Constructing ITE for a unate cover

We experiment with all the three techniques of Section 8.6.2 for handling unate covers: *use fac*, *use column cover*, and *use the method for binate cover*, and report the results in Table 8.4. For the binate cover, the *most binate* variable heuristic is applied. The techniques give comparable results, the factored form being slightly better than the other two.

Constructing ITE for a binate cover

We experiment with both branching variable selection methods of Section 8.6.3: *most binate* and *quick map*. The results are shown in Table 8.5. On *dalu* and *des*, *quick map* ran out of memory. On the rest of the examples, it is 0.5% better than *most binate*, but many times slower. This slight improvement is probably not worth the time penalty to be paid.

8.7.3 Iterative Improvement with Last Gasp

So far, last gasp (step 3) was skipped in all the experiments. It is put to use for the first time, and is applied right after the iterative improvement step. The results obtained thus are compared with iterative improvement and no last gasp (column *ite-map + iter. imp.* of Table 8.3). The results, compiled in Table 8.6, show that last gasp is ineffective. We believe this is because the mapping solution generated by iterative improvement without last gasp is close to the best attainable with our techniques.

8.7.4 Using Global ROBDDs

For networks with small number of inputs (for this experiment, at most 15), we construct global ROBDDs, i.e., step 6 of the algorithm, and check if it improves upon the solution generated using iterative improvement (column *ite-map + iter. imp.* of Table 8.3). Last gasp was not used, since it is not beneficial. We used both *old* and *new* ROBDD packages. The results are shown

example	ite-map + iter. imp.	ite-map + cover
5xp1	40	42
9sym	53	53
C1355	164	164
C1908	155	156
C2670	205	340
C3540	472	502
C432	89	92
C5315	519	588
C6288	988	988
C7552	651	755
alu2	130	134
alu4	308	335
apex2	106	113
apex3	697	697
apex7	82	85
b9	56	56
bw	54	55
clip	37	41
cordic	21	22
dalu	361	366
des	1216	1279
duke2	164	170
e64	90	90
ex4	191	192
f51m	23	23
k2	526	545
misex2	37	37
rd84	52	55
rot	236	246
sao2	52	55
spla	242	246
t481	10	10
vg2	32	33
z4ml	14	13
total	8073	8578

Table 8.3: *act1* count with basic iterative improvement

ite-map + iter. imp. steps 1, 2, 4, and 5 of the algorithm of Section 8.4
ite-map + cover steps 1, 4, and 5 of the algorithm of Section 8.4
total sum of *act1* counts over all the examples

example	use fac	column cover	as binate
5xp1	40	40	40
9sym	53	53	53
C1355	164	164	164
C1908	155	156	156
C2670	205	205	205
C3540	472	472	473
C432	89	85	86
C5315	519	520	519
C6288	988	989	988
C7552	651	657	655
alu2	130	129	130
alu4	308	304	305
apex2	106	110	112
apex3	697	692	699
apex7	82	83	82
b9	56	55	55
bw	54	54	54
clip	37	37	37
cordic	21	21	21
dalu	361	363	361
des	1216	1216	1218
duke2	164	168	166
e64	90	90	90
ex4	191	199	197
f51m	23	23	23
k2	526	527	522
misex2	37	36	37
rd84	52	52	52
rot	236	236	238
sao2	52	52	53
spla	242	243	242
t481	10	10	10
vg2	32	32	32
z4ml	14	14	14
total	8073	8087	8089

Table 8.4: Handling unate covers

use fac obtain a factored form for the unate cover
col cover use a column covering procedure to construct ITE for the unate cover
binate handle the unate cover exactly like a binate cover
total sum of *act1* counts over all the examples

example	most binate	quick map
5xp1	40	38
9sym	53	52
C1355	164	164
C1908	155	156
C2670	205	202
C3540	472	463
C432	89	88
C5315	519	503
C6288	988	989
C7552	651	657
alu2	130	131
alu4	308	300
apex2	106	110
apex3	697	692
apex7	82	82
b9	56	52
bw	54	55
clip	37	35
cordic	21	22
dalu	361	-
des	1216	-
duke2	164	163
e64	90	90
ex4	191	203
f51m	23	24
k2	526	522
misex2	37	37
rd84	52	50
rot	236	238
sao2	52	53
spla	242	239
t481	10	10
vg2	32	31
z4ml	14	14
total	8073	-
subtotal	6496	6465

Table 8.5: Variable selection method

most binate select the most binate variable at each step
quick map select the minimum-cost variable: compute cost by mapping algebraic cofactors
total sum of *act1* counts over all the examples
subtotal sum of *act1* counts when *quick map* finishes

example	last gasp	no last gasp
5xp1	40	40
9sym	53	53
C1355	164	164
C1908	154	155
C2670	205	205
C3540	472	472
C432	89	89
C5315	519	519
C6288	988	988
C7552	651	651
alu2	130	130
alu4	308	308
apex2	107	106
apex3	698	697
apex7	82	82
b9	55	56
bw	54	54
clip	37	37
cordic	21	21
dalu	361	361
des	1216	1216
duke2	163	164
e64	90	90
ex4	191	191
f51m	23	23
k2	524	526
misex2	36	37
rd84	52	52
rot	236	236
sao2	52	52
spla	242	242
t481	10	10
vg2	30	32
z4ml	14	14
total	8067	8073

Table 8.6: Last Gasp

last gasp apply last gasp with iterative improvement
no last gasp do not apply last gasp - only iterative improvement
total sum of *act1* counts over all the examples

example	init	old bdd	new bdd
5xp1	40	51	51
9sym	53	17	14
alu2	130	131	111
bw	54	112	86
clip	37	105	96
f51m	23	39	53
rd84	52	36	37
sao2	52	67	77

Table 8.7: Using global ROBDDs

init starting block count
old bdd construct ROBDD using the old package and then map
new bdd construct ROBDD using the new package and then map

in Table 8.7. Some improvement is obtained in examples *9sym*, *alu2*, and *rd84*. These are either completely symmetric or partially symmetric benchmarks. As mentioned earlier, global ROBDDs provide a compact representation for such functions. On the rest of the benchmarks, the results are worse and therefore rejected. Also, the *new* ROBDD package is seen to be slightly better than the *old* one. This is possibly because of the sharing of complement functions in the *new* package.

8.7.5 Treating Node as the Image of the Network

This idea explores the possibility of extending the last gasp paradigm. Recall that last gasp attempts to improve the solution quality towards the end of the mapping algorithm, but fails since iterative improvement produces good-quality results. Here, we embed last gasp at the node-mapping level. For each node n of the network η, instead of constructing an ITE and mapping it, we construct a network $\eta(n)$, which has one primary output, one internal node corresponding to n, and as many primary inputs as the fanins of n. Then, on $\eta(n)$ we apply the algorithm of Section 8.4 (with iterative improvement). This results in a network $\tilde{\eta}(n)$. We replace n in η by its implementation $\tilde{\eta}(n)$. After this step has been performed for each node n, η gets converted into a new network, say $\tilde{\eta}$. Each intermediate node in $\tilde{\eta}$ can be implemented by an *act1* module. Hence, the number of internal nodes in $\tilde{\eta}$ gives the number of *act1* modules needed for η. Finally, we construct a global ROBDD for the network and map the ROBDD on *act1* modules. If we get a better block count than that of $\tilde{\eta}$, we accept the ROBDD decomposition. These results are shown in the column *best-map*

of Table 8.8 and are compared with those obtained after iterative improvement (Table 8.3, column *ite-map + iter. imp.*) and then building global ROBDDs (Table 8.7), as reproduced in the column *prev best*. There is a slight improvement in the result quality. But it is at the expense of enormous run-time.

8.7.6 Comparing Various Systems

Here we compare mis-fpga with the library-based mapper in misII [18] and Amap - actually an improved version of [38]. The starting optimized networks are identical for all the systems. For mis-fpga, we use the results from *prev best* column of Table 8.8, since it completes on all the examples, is just 1% worse than the *best-map* option, and is much faster.

In Table 8.9, we compare Amap with our approach. It can be seen that mis-fpga generates much better results as compared to Amap, on average 18.5% (using the row *subtotal*). Using the average of percentage difference for each example as the measure, we get 20.5% improvement. Although both the tools use ITE-based representation, mis-fpga uses a different way of constructing ITEs. Also, it uses an iterative improvement step, which accounts for better results. Not only that, as can be seen from **ite-map** column of Table 8.1, mis-fpga with no iterative improvement is slightly better than Amap. The results of Table 8.9 are pictorially shown in Figure 8.22.

To compare mis-fpga with the library-based approach of misII, we created two libraries:

1. *complete*: it consists of all 702 non-P-equivalent functions that can be realized with one *act1* module,

2. *actel manual*: it consists of all the cells in the Actel manual [2], which are about 90.

Table 8.10 shows the results. It turns out that using a complete library helps - by 6.7% on average. However, the mapping is slow - in fact, about 7-10 times slower than in the case of partial library. The mis-fpga results as shown in the previous table are about 12.5% better than the complete library's and 18.2% than the partial library's. Using the measure of average of percentage improvement for each example, mis-fpga is 13.7% better than the complete library and 19.7% better than the partial one. It reaffirms the thesis that better quality can be achieved by using architecture-specific algorithms. The results are compared graphically in Figures 8.23 and 8.24.

example	best-map	prev best
5xp1	37	40
9sym	14	14
C1355	164	164
C1908	149	155
C2670	200	205
C3540	439	472
C432	86	89
C5315	503	519
C6288	990	988
C7552	621	651
alu2	111	111
alu4	293	308
apex2	104	106
apex3	696	697
apex7	80	82
b9	54	56
bw	54	54
clip	36	37
cordic	21	21
dalu	340	361
des	-	1216
duke2	166	164
e64	90	90
ex4	187	191
f51m	23	23
k2	527	526
misex2	36	37
rd84	37	37
rot	232	236
sao2	50	52
spla	238	242
t481	11	10
vg2	30	32
z4ml	14	14
total	-	8000
subtotal	6633	6784

Table 8.8: *act1* count: treating node as the image of the network

best-map do the best mapping of each node - use all the steps of the algorithm of Section 8.4 except last gasp
prev-best all the steps of the algorithm of Section 8.4 except last gasp
total sum of *act1* counts over all the examples
- out of memory

example	amap	mis-fpga
5xp1	47	40
9sym	60	14
C1355	142	164
C1908	172	155
C2670	398	205
C3540	541	472
C432	104	89
C5315	631	519
C6288	1224	988
C7552	820	651
alu2	163	111
alu4	356	308
apex2	129	106
apex3	789	697
apex7	98	82
b9	64	56
bw	-	54
clip	45	37
cordic	27	21
dalu	429	361
des	1527	1216
duke2	204	164
e64	120	90
ex4	243	191
f51m	30	23
k2	567	526
misex2	49	37
rd84	63	37
rot	296	236
sao2	63	52
spla	-	242
t481	12	10
vg2	39	32
z4ml	18	14
total	-	8000
subtotal	9470	7704

Table 8.9: mis-fpga vs. Amap

amap using Amap
mis-fpga *prev best* column of Table 8.8
total sum of *act1* counts over all the examples
- segmentation fault

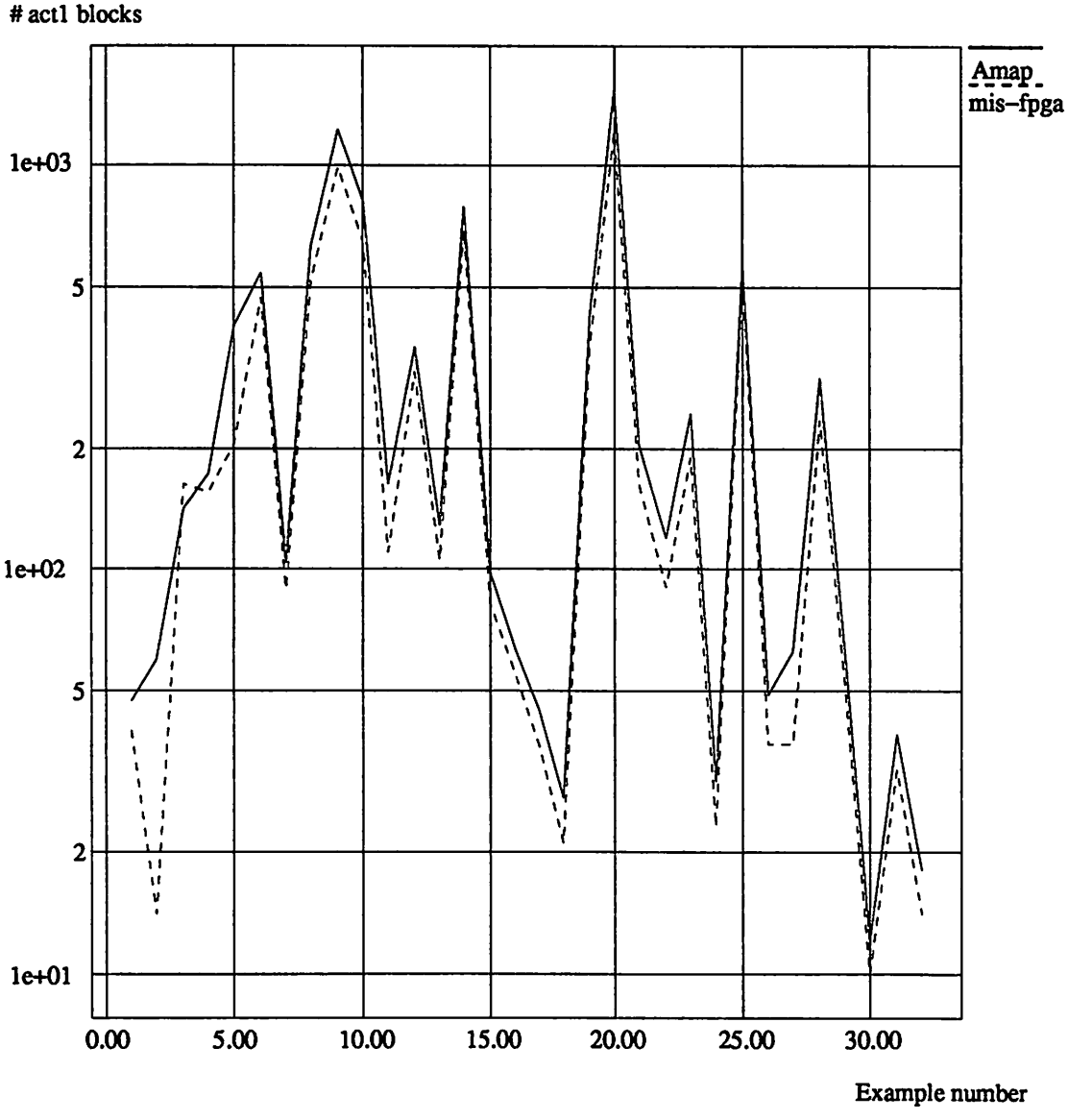


Figure 8.22: Comparing mis-fpga with Amap

example	library-based		mis-fpga
	complete	*actel manual*	
5xp1	45	52	40
9sym	57	61	14
C1355	182	174	164
C1908	181	188	155
C2670	282	297	205
C3540	485	516	472
C432	91	92	89
C5315	621	682	519
C6288	1425	1456	988
C7552	816	855	651
alu2	137	152	111
alu4	329	372	308
apex2	104	114	106
apex3	650	695	697
apex7	89	101	82
b9	53	62	56
bw	68	74	54
clip	49	50	37
cordic	22	23	21
dalu	361	420	361
des	1339	1446	1216
duke2	173	188	164
e64	116	116	90
ex4	197	210	191
f51m	28	33	23
k2	507	546	526
misex2	45	46	37
rd84	58	65	37
rot	276	305	236
sao2	54	59	52
spla	237	257	242
t481	13	13	10
vg2	40	43	32
z4ml	18	18	14
total	9148	9781	8000

Table 8.10: Library-based approach vs. mis-fpga

complete use the complete library
actel manual use the library provided by Actel
mis-fpga *prev best* column of Table 8.8
total sum of *act1* counts over all the examples

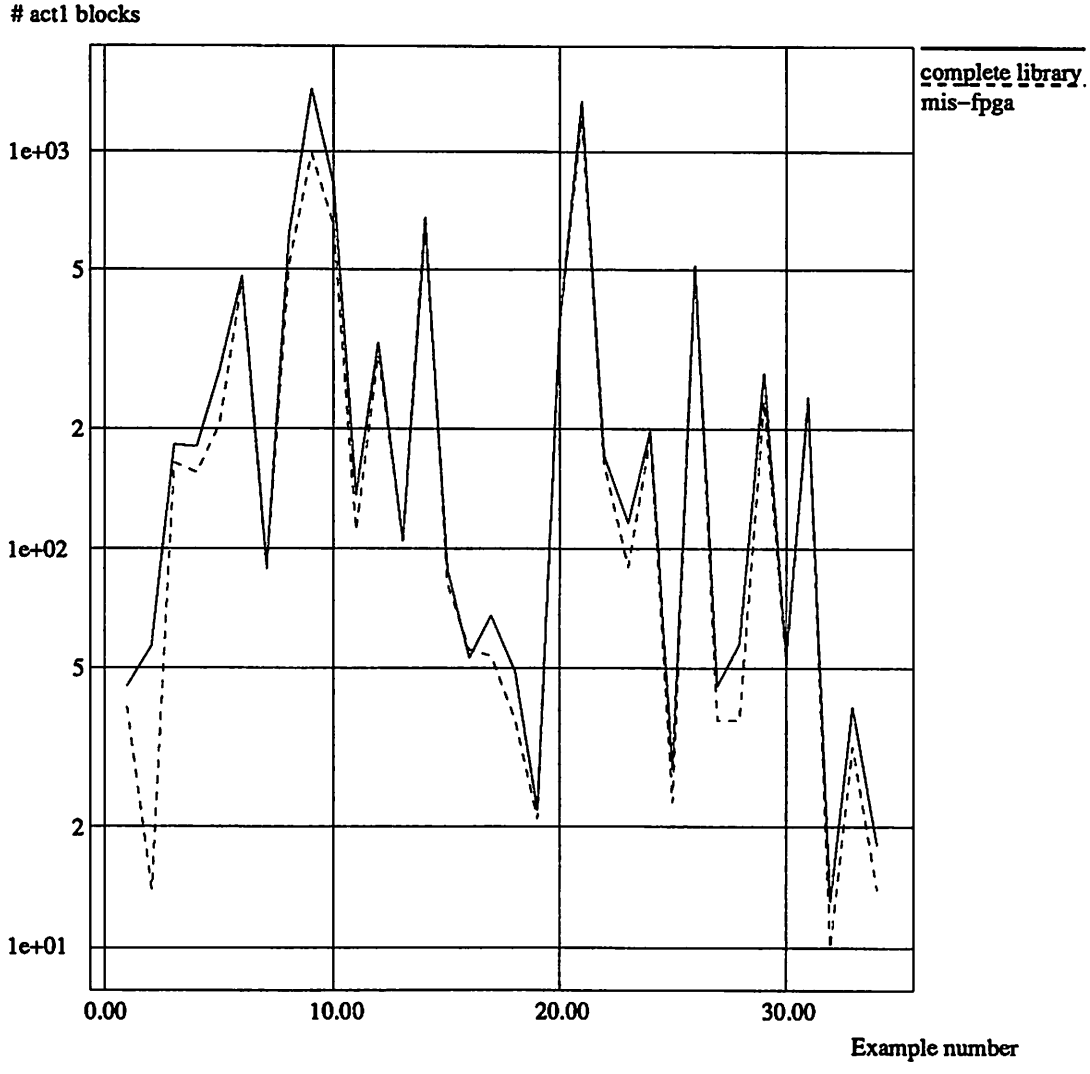


Figure 8.23: Comparing mis-fpga with the complete library

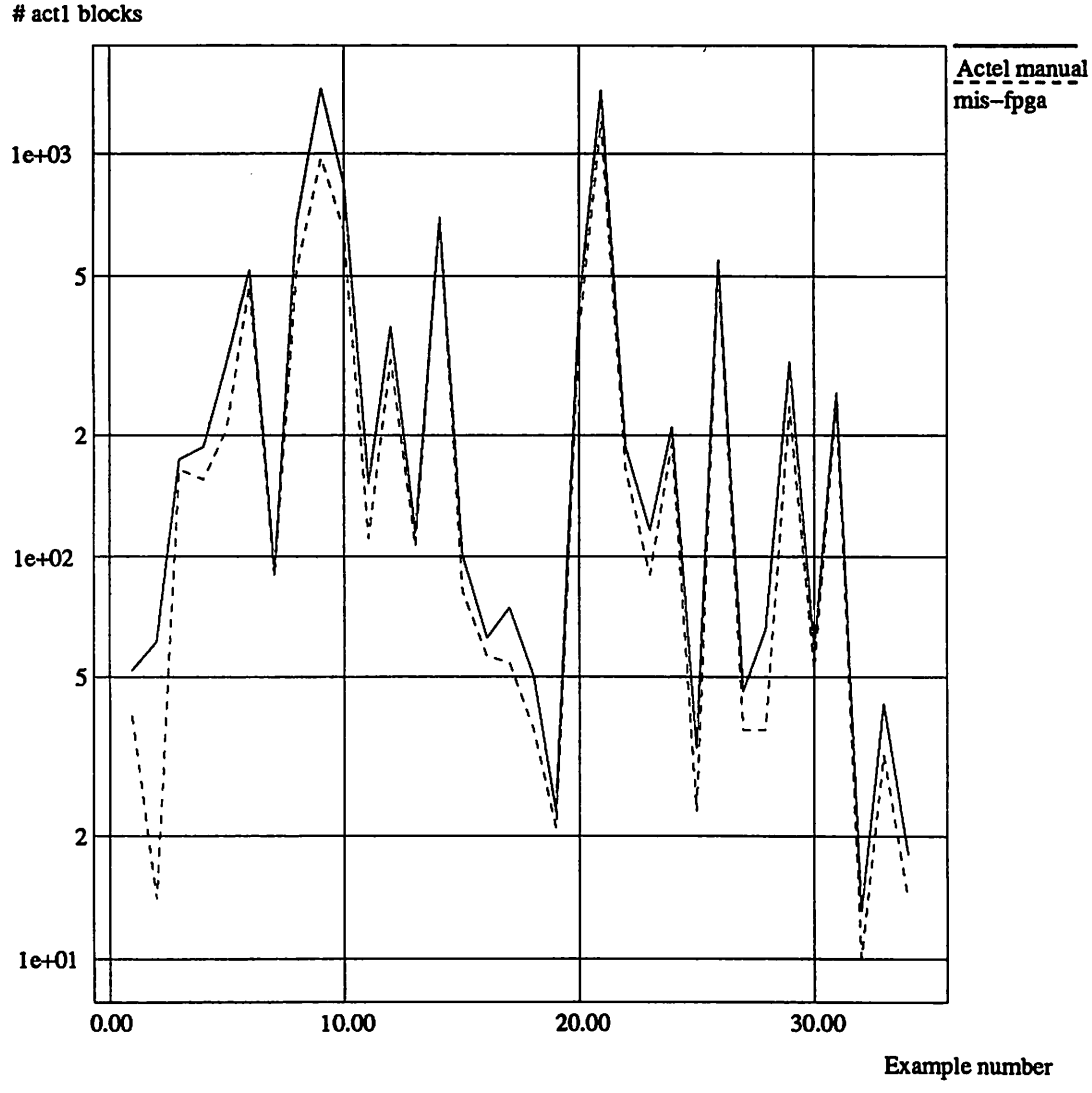


Figure 8.24: Comparing mis-fpga with the partial library provided by Actel

8.7.7 Using Multi-rooted ITEs

The subject graph in our algorithm is either a BDD or an ITE, and is constructed and mapped for each node function separately. It looks promising to explore a subject graph that is global and at the same time does not suffer from the ordering constraint of global ROBDDs. Then, the mapping algorithm would have a larger graph to map, potentially yielding better mapping solutions. A natural choice for such a representation is a multi-rooted unordered ITE. It has as many roots as the primary outputs of the network, and is constructed by composing the ITEs of the individual node functions. To handle the new representation, we made appropriate modifications in our mapping algorithm.

To test the new representation, a multi-rooted ITE is constructed for the entire network and mapped without any iterative improvement. The results are presented in the column *mroot-ite* of Table 8.11, and are compared in the column *ite-map* with those obtained after constructing ITEs for each node and mapping them without any iterative improvement (this column is a copy of the column *ite-map* of Table 8.1). It turns out the results for the two columns are almost identical. We believe this is because of the tree-based nature of the mapper. In an optimized network, each node is saving some literals, i.e., the number of literals in the network will increase if the node were eliminated from the network. So, very likely, the root vertex of the node ITE, after composition of the local ITEs, fans out to more than one ITE vertex in the the multi-rooted ITE. The mapper breaks up this ITE into trees by clipping the ITE at all multiple-fanout vertices, and maps each tree separately. The problem then is no different from that of mapping singly rooted ITEs. However, if an exact mapper (e.g., one based on an exact binate covering solver) were to be used, we expect multi-rooted ITE to be superior.

8.8 Discussion

Our basic premise was that in the mapping step for MUX-based architectures, instead of a NAND-based representation, a MUX-based representation should be used. We started with a BDD-based representation and later switched over to a more general ITE-based representation. Table 8.10 shows that an ITE-based approach does much better than a library-based approach that uses the NAND-gate representation, thus establishing the premise.

Comparing various representations, ROBDD has only one copy of a logic function, but it suffers from unnecessary ordering constraint, and the results depend strongly on the ordering. BDD

example	mroot-ite	ite-map
5xp1	46	46
9sym	54	54
C1355	166	166
C1908	176	176
C2670	381	377
C3540	550	552
C432	101	102
C5315	661	658
C6288	1226	1226
C7552	818	813
alu2	142	142
alu4	352	352
apex2	120	120
apex3	715	714
apex7	94	94
b9	61	60
bw	63	63
clip	45	45
cordic	26	26
dalu	401	401
des	1398	1399
duke2	186	186
e64	116	116
ex4	208	208
f51m	25	25
k2	566	570
misex2	41	41
rd84	56	56
rot	273	270
sao2	60	60
spla	264	266
t481	14	15
vg2	35	35
z4ml	16	16
total	9456	9450

Table 8.11: Multi-rooted ITEs vs. singly rooted ITEs

mroot-ite create multi-rooted ITE for the network and map it
ite-map step 1 of the algorithm of Section 8.4
total sum of *act1* counts over all the examples

does not have any ordering problem, but it can have multiple copies of the same function.

On cofactoring, both these representations can replicate product terms in the two branches. However, as shown in [38], ITEs avoid this replication by using algebraic cofactors.

We believe that the best quality results are obtained if the subject graph is created keeping in mind the target architecture. As we demonstrated, approaches like ours that directly map on to the architecture can create such a subject graph. Since a library has many gate functions, library-based approaches are unable to construct subject graphs that are *good* for the architecture.

Iterative improvement is essential in getting good quality results.

We targeted primarily *act1* module. At a time when there is a surge of new block architectures, it is difficult for synthesis to keep up. So our guiding philosophy was to fix an architecture, strive for the best quality results, and hope that the algorithms developed for this architecture can be appropriately modified for other architectures. For instance, to come up with a good mapper for *act2*, pattern graphs have to be derived. Ideally, the subject graph construction should also be tailored. We proposed a matching algorithm for *act2*. Then, the same core of initial mapping, iterative improvement, etc., can be used.

Chapter 9

Conclusions

9.1 Contributions

This thesis addressed the problem of synthesizing circuits on field-programmable gate array architectures. When we started the work, no synthesis algorithms had been published for the most popular FPGA architectures such as look-up table- and multiplexor-based architectures.

We showed that the synthesis problem for these architectures is different from that solved by the conventional, standard-cell based logic synthesis tools. Both optimization and technology-mapping, the two main steps of logic synthesis paradigm, need to be modified in order to obtain high quality results.

9.1.1 Synthesizing Combinational Logic

For mapping, we considered two objective functions - minimum area, approximated by the minimum number of basic blocks, and minimum delay. Most of our effort was directed towards producing a circuit with the minimum number of blocks. Even computing this number for an arbitrary Boolean function is an NP-hard problem. So good heuristics are needed. We divided mapping into two main steps: breaking infeasible functions into sets of feasible functions, and then minimizing the number of feasible functions.

For LUT architectures, we examined various decomposition strategies, as well as variable support reduction, for the first step. We were the first to apply the classical functional decomposition for these architectures. We showed that the problem of obtaining *small* functions after decomposition can be exactly formulated as an encoding problem - an input-output encoding problem, to be precise.

Previous formulations encoded the equivalence classes. We showed that this is not the most general formulation. We presented the most general formulation, which encodes the vertices in the Boolean space corresponding to the bound set such that vertices in different equivalence classes are assigned different codes. This result is in the context of the general functional decomposition algorithm, and, as such, is architecture-independent. We applied it to LUT architectures and showed that an input encoding formulation suffices. We also studied decomposition using cube-packing. We proved that for look-up tables with at most 5 inputs, cube-packing gives optimum tree implementation for functions consisting of cubes with mutually disjoint supports. We also showed that, in general, finding an optimum cube-packing solution is NP-hard. Other decomposition techniques such as cofactoring and AND-OR decomposition were also studied. For MUX-based architectures, the decomposition step generates a network in terms of 2-to-1 multiplexors, since it is a more natural representation for such architectures. Either BDDs or ITEs may be used as the basic representation. An important problem in mapping is the matching problem, i.e., determining if a function can be implemented by a module. We proposed matching algorithms for the *act1* and *act2* modules. Exploitation of the module-structures makes the algorithms fast. Note that in conventional technology-mapping, just before the covering step, the network is decomposed into two-input NAND gates. In principle, we could apply the covering (block count minimization) step on a network of NANDs. However, decomposition that explicitly targets FPGA architectures yields better results after covering. This is one contribution of our work.

We devised two strategies for minimizing the number of functions in a feasible network. The first is a covering method, similar to that used in the conventional technology-mapping. A binate covering formulation is used. For LUT architectures, we defined the notion of an m-feasible supernode, which is the basic object in the covering algorithm. Optimum and heuristic algorithms were used/developed to solve the problem. The second strategy, specific to LUT architectures, is based on reducing the support of a node function so that the node can be absorbed in other nodes. These two block count minimization strategies were coupled into one algorithm.

For both LUT and MUX-based architectures, making the entire network feasible and then minimizing the block count is not as effective as applying these operations on each node of the network separately and then exploiting the structural relationship between the nodes of the network in a *partial collapse* operation.

We also examined the optimization issues for FPGAs. In particular, we have modified kernel extraction for LUT architectures, and are exploring simplification. We have shown how to modify two-level minimization so as to obtain an SOP that is better suited for cube-packing.

9.1.2 Mapping Sequential Logic

For sequential circuits, we focussed only on LUT architectures. The *act1* and *act2* MUX-based architectures do not have any flip-flops, and so two modules have to be configured appropriately using feedback to realize a flip-flop. Then the problem of mapping presents no new challenges. However, the commercially available LUT architectures have flip-flops. For instance, the Xilinx 3090 CLB is a complex block with two flip-flops and can be configured in many ways. One contribution of this thesis was showing that 19 configurations suffice. Another one was developing a fast algorithm to answer the following question: "*Can a given set of combinational functions and flip-flops of the sequential circuit fit on one* CLB ?" This is crucial since it forms the basis of a mapping technique we proposed, in which combinational functions and flip-flops are mapped simultaneously. An alternate and faster way is to map the combinational functions first and then the flip-flops. A flow-based polynomial time algorithm for the best placement of flip-flops in the already used CLBs was presented. This is optimum under some assumptions. Both techniques build on and use the combinational mapping techniques proposed earlier.

9.1.3 Complexity Issues

Questions like how good various FPGA tools are and how much more they can be improved led us to examine theoretical complexity issues. We were able to determine the exact complexity for a subset of functions for LUT architectures. Unfortunately, the problem of determining the complexity of a function (i.e., the minimum number of blocks needed to realize the function) is a hard one. The next best alternative is to determine bounds on the complexity. The thesis explored how to compute upper bounds for a function, and a network in general, given some representation. Two representations - SOP and factored form - were considered. Most of these bounds were proven tight under some simplifying assumptions. These bounds can be used to predict quickly the block count for a circuit without doing any technology mapping, and we provided some experimental evidence for the accuracy of these estimates.

We did not derive similar bounds for the MUX-based architectures. The main reason is that an LUT is easy to characterize - simply by the number of inputs. However, the basic blocks for other architectures cannot be characterized so easily, making the bound derivation hard.

9.1.4 Performance Directed Synthesis

All of the above approaches were related to obtaining a minimum-area circuit. This thesis also addresses performance optimization for LUT architectures. Because of the constraints imposed by the architecture and programming methodology, the wiring delays can be unpredictable and can be a significant fraction of the total path delay. Lacking placement information, the logic-level delay models cannot handle wiring delays. Our contribution is to solve the problem by a two-phase approach: first, apply transformations at the logic level using an approximate delay model and then, couple timing-driven placement with resynthesis using a more accurate delay model.

All of these techniques, including those for minimum block count and minimum delay, have been implemented in mis-fpga and sis-fpga, which are built on top of sis. They are currently being used in both academic and industrial environments. An average improvement of 10-30% over other systems can be expected, though at the cost of longer run times.

9.2 Future Work

9.2.1 Improving the Implementation

The primary goal of this thesis was to obtain the best quality results. The run-time of the algorithms was a secondary consideration. Consequently, there is reason to believe that the current implementations of mis-fpga and sis-fpga can be speeded up significantly. One way is to first identify the critical sections of the code through profiling and then speed them up. Another is through *memoization*. Right now, the mapping algorithm is applied afresh on each function f under consideration. It may be the case that either f or another function g equivalent to f (NPN-equivalent for LUT and P-equivalent for MUX-based architectures) were encountered earlier and therefore already mapped. If stored in a hash table, these mappings can be used for f, thus avoiding unnecessary computation. One benefit to accrue from this speed up is that for the same CPU time spent, more computation can be performed, potentially translating to better results.

9.2.2 Using Don't Cares

The mapping techniques proposed in this thesis dealt only with completely specified functions. In general, because of the structure of the network, each node function has a don't care set associated with it, which can be used to derive a representation that is better suited for mapping. In a general mapping framework, the use of don't cares was initiated by Mailhot and Micheli [52],

and later extended by Savoj *et al.* [71]. Their ideas can be applied to FPGA mapping techniques. In fact, recently Lai *et al.* [43] have implemented a BDD-based version of Roth-Karp decomposition for incompletely specified functions.

9.2.3 Logic Optimization

We have just started targeting the logic optimization phase for FPGAs, in particular for LUT architectures. We modeled cube-packing in a two-level minimizer and showed how to obtain a sum-of-products representation that is suited for cube-packing. This is yet to be incorporated in a multi-level environment. We believe that other FPGA mapping techniques need to be modeled in various optimization steps and only then can success be attained in this venture.

9.2.4 Delay Optimization and Modeling

The delay model used currently at the logic level is weak. This is because it has no idea about the wiring delays, which are a function of the module locations. Better delay models need to be developed also for placement. The Elmore delay model does not consider fanout loading, and the Rubinstein-Penfield-Horowitz model gives two delay numbers, which could differ from each other significantly.

9.2.5 Area-Delay Trade-offs

During synthesis, we did not take into account the capacity of chips. It may be the case that the circuit fits on one chip, with some blocks unused. By modifying the circuit such that the unused resources are used, we may be able to improve circuit speed. Similarly, the circuit may meet the performance constraints, and it may be possible to recover area without violating the constraints. Recently Cong *et al.* [17] and Bhat [7] have proposed some techniques for LUT architectures. Bhat's approach is particularly attractive, since it modifies an existing library-based mapper and thus makes available the whole wealth of techniques developed already [82].

9.2.6 Synthesis for Routability

In Xilinx 3090 architecture, routing is a bottleneck. A smaller implementation may not necessarily be easier to route. Synthesis algorithms that take routing constraints into account have to be devised. This thesis does not directly address routing issues, except that one of the block

count minimization heuristics approximates routability by the number of edges in the network. When there is a choice of nodes to collapse into fanouts, the one that creates the least extra edges is selected. Although it promises better routability, there are no guarantees. Work by Schlag *et al.* [76] is similar in that they also model routability with the number of edges in the network. They modify the cost function of the covering algorithm for routing and target the 2-output CLB of the Xilinx 3090 architecture directly. Recently, Bhat and Hill [8] proposed an algorithm that couples synthesis, placement, and routing in a tight loop.

9.2.7 Sequential Circuits

Although we addressed the problem of mapping for sequential circuits on to LUT-based architectures, we did not devote much attention to the state-assignment and optimization problems. Given a chip that has some number of combinational and sequential components, the synthesis algorithm is assigned the task of fitting the circuit on a minimum number of chips. Synthesis has to be carried out under the constraint of fixed resources. State-assignment can trade-off combinational and sequential elements, thus producing a design that matches the architectural constraints.

9.2.8 Partitioning Issues

If a design is too large to fit on one chip, it has to be partitioned into many chips. If a minimum cost solution is desired, i.e., using minimum number of chips, the standard partitioning algorithms, e.g., of Kernighan and Lin [40], can be used. If the minimum delay is the primary objective, an algorithm described in [59] may be used. This problem is interesting because every time a signal crosses chip boundaries, an extra delay called **inter-chip delay** is incurred, which could be much more than the delay on a signal that lies completely within a chip. However, this algorithm does not consider pin constraints of the chip. So a few extensions need to be made.

Another interesting partitioning problem arises because of the existence of different LUT architecture-families (e.g., Xilinx 2000, Xilinx 3090, Xilinx 4000). Since the cost, capacity, pins, and performance differ for the families, the following question assumes great importance: *"Given a cost objective and a performance constraint (alternatively, a performance objective and a cost constraint), how many chips of each family should be used, and how should the different parts of the circuit be assigned to these families?"* Recently, Kužnar *et al.* [42] proposed a multi-way partitioning algorithm based on a recursive application of the Fiducia-Mattheyses bipartitioning heuristic [22] to address a restricted problem (without performance issues).

In our current approach, partitioning constraints are ignored during synthesis. Recent work by Beardslee *et al.* [4, 5] addresses logic partitioning subject to pin limitations. The number of wires going across chips is minimized at the cost of extra encoding logic, thus trading off area for pins. This work is in a general synthesis framework and can be applied in the FPGA domain.

9.2.9 Targeting New Architectures

New FPGA architectures are being proposed. For instance, the new Xilinx 4000 CLB has three LUTs - two four-input LUTs feeding a three-input LUT. Our techniques, being architecture-specific, have to be modified to be applicable to a new architecture. Is it realistic to devise separate algorithms for separate architectures? Though better results are possible, it may not be viable from a business viewpoint. We propose an *integrated approach*, which uses a fast matching algorithm for each architecture and provides a common core of synthesis operations and transformations (e.g., the *partial collapse*, decomposition, etc.) for all the architectures. Its motivation comes from the way mis-fpga evolved for LUT and MUX-based architectures. Initially the techniques were quite different for the two architectures, but finally they converged. This approach, however, does not consider the following:

1. Technology-independent optimization: Although the number of literals in a factored form is, in general, a reasonable cost function, it may not be the best one, as was shown in Chapter 4 for LUT architectures. For such cases, should we embed a quick mapper in the optimization steps to determine if the cost has improved? The cost computation should be fast, since it has to be performed many times. Some researchers consider the support of a function as its cost [28]. Though easy to compute, it may not be a good cost function, since functions with the same support can require widely varying number of blocks.

2. Technology-decomposition: Architecture-specific decomposition generally yields better quality results. A representation that is more suitable for the architecture is desired. One solution, of course, is to consider a 2-input decomposition as a starting point irrespective of the architecture.

As long as some quality-hit is acceptable (which is reasonable given the alternative of developing different algorithms for different architectures), this approach is attractive.

9.2.10 What is a Good Architecture?

The last few years have seen a proliferation of FPGA architectures, which vary in the basic block structure, the routing architecture, the programming technology, etc. Given this wide variety, is some architecture better than others? Although no satisfactory answer is available, a few comments can be made.

1. An architecture is only as good as the synthesis algorithm targeting it. An otherwise good architecture may remain underutilized if good synthesis algorithms cannot be developed for it. Non-uniformity of the architectures, such as the Xilinx 4000, makes the development of synthesis algorithms hard. There is a need to have architectures that are *good* from the synthesis perspective.

2. A finer grain block is more efficient in terms of area, but not in terms of the number of levels of blocks needed to implement a function. Moreover, each time a new level is needed, some delay through the programmable switch may be incurred, thus making a very fine-grain architecture unattractive.

Appendix A

mis-fpga

A.1 Introduction

The techniques described in this thesis are implemented in misII, now a part of sis. The term mis-fpga is used to refer to the part of misII that pertains to the FPGA architectures. This includes the algorithms, implementation, and the commands.

In particular, combinational circuits can be synthesized for both LUT and MUX-based architectures. Area minimization for both architectures and delay minimization for LUT architectures are supported. The implementation for sequential synthesis, sis-fpga, is not being distributed currently, but we hope to make it available in the next sis release.

This appendix briefly describes the commands corresponding to the algorithms described in the thesis to map on to LUT and *act1* architectures. Detailed descriptions are provided only for those commands that have user-controlled options. For a complete description of each command and the corresponding options, **help <command-name>** should be used within sis.

A.2 Synthesis for LUT Architectures

Only the mapping algorithms are currently distributed. A standard optimization script such as *script.rugged* [73] should therefore be used before invoking the commands described below.

A.2.1 Making an Infeasible Network Feasible

The following is a summary of various decomposition commands to make m-infeasible networks feasible.

xl_ao	cube-packing on an infeasible network
xl_k_decomp	apply Roth-Karp decomposition
xl_split	modified kernel extraction
tech_decomp	existing technology decomposition
xl_imp	apply different decomposition schemes and pick the best

To get the best possible decomposition in *xl_k_decomp*, all possible choices of input partitions have to be tried. This is made possible with the options `-e` and `-f`. Then, the decomposition with the minimum number of nodes is selected.

xl_split extracts kernels from an m-infeasible node. This procedure is recursively applied on the kernel and the node, until either they become m-feasible or no more kernels can be extracted, in which case a simple AND-OR decomposition is performed.

xl_imp tries to obtain the best possible decomposition for the network. It applies a set of decomposition techniques on each m-infeasible node n of the network. These techniques include cube-packing on the sum-of-products form of n, cube-packing on the factored form of n, Roth-Karp decomposition, and kernel extraction. The best decomposition result - the one which has a minimum number of m-feasible nodes - is selected. There are options to control which techniques to use.

tech_decomp is a command which already existed in misII and is used typically just before technology mapping. It takes two parameters `-a AND-limit` and `-o OR-limit`, and decomposes the SOP at each node into AND and OR nodes, the fanins of each node being limited from above by the AND-limit and OR-limit respectively.

One command which does not necessarily generate an m-feasible network, but reduces the *infeasibility* of a network is **xl_absorb**. This is based on the support reduction technique of Section 3.3.6. Infeasibility of a network is measured as the sum of the number of fanins of the infeasible nodes. The command *xl_absorb* moves the fanins of the infeasible nodes to feasible nodes so as to decrease the infeasibility of the network. Roth-Karp decomposition is used to determine if a fanin of a node could be made to fan in to another node.

A.2.2 Block Count Minimization

The following commands are used.

xl_cover	use binate covering
xl_partition	collapse nodes into immediate fanouts

xl_cover corresponds to the covering technique of Section 3.4.1. Mathony's algorithm [54] is used to solve this formulation exactly. For large networks, this algorithm is computationally

intensive and we have developed several heuristics for fast approximate solutions. The `-h` option provides a means to select the exact or a heuristic method. The options `-e` and `-u` enable the program to automatically switch between the exact and the heuristic based on the number of nodes in the network. If this number is no more than the one specified by the `-e` option, the exact method is applied, if it is more than the one specified by the `-u` option, *xl_cover* does nothing, otherwise, it works as per the `-h` option.

xl_partition corresponds to *partition* heuristic of Section 3.4.1, which tries to reduce the number of nodes by collapsing them into their some or all the immediate fanouts. It can also take into account extra nets created. In the *default* mode, it collapses a node into its fanout only if the resulting fanout is m-feasible. It associates a cost with each (node, fanout) pair which reflects the extra edges generated in the network if node is collapsed into the fanout. It then selects pairs with lowest costs and collapses them. With `-t` option, a node is considered for collapsing into all the fanouts, and is collapsed if all the fanouts remain m-feasible after collapsing. The node is then deleted from the network. Further optimization can be obtained by considering the technique of support reduction, described in Section 3.4.2. This technique is applied as follows. Before considering the collapse of a node n into its fanout(s), we check if any fanin F of n could be moved to G - another fanin of n. This increases the chances of n being collapsed into its fanout(s). Moreover, it may later enable some other fanin of n to be collapsed into n. This technique is invoked using the `-m` option.

A.2.3 The Overall Algorithm

The command **xl_part_coll** corresponds to the *partial collapse* of Section 3.5, except for one difference. It also performs initial mapping for each node. So it invokes decomposition and block count minimization routines. The `-g value` option specifies which representation to use for mapping a node. A value of 0 just maps the SOP, value 1 maps a factored form, and value 2 maps both and picks the better of the two. The `-c` option puts an upper limit on the number of nodes for the exact *cover*. *Partition* is also invoked and brings the `-m` option along with it.

The following script was used to generate the results of Table 3.6.

```
xl_part_coll -m -g 2 -c 50
xl_coll_ck
xl_cover -e 60 -u 200
xl_partition -m
```

xl_coll_ck collapses a feasible network if the number of primary inputs is small (this number can be specified by `-c` option), applies Roth-Karp decomposition and cofactoring schemes, picks the better of the two, and compares it with the original network (before collapsing). If the number of nodes is smaller in the new network, the original network is replaced with the new one. If `-k` option is specified, Roth-Karp decomposition is not applied; only cofactoring is used. Currently, $m = 2$ is not supported in *xl_coll_ck*.

Intermediate solutions in the quality versus run-time trade-off curve can be obtained by suitably choosing the scripts. For example, to get reasonably good results in a short time, the following script may be used:

```
xl_ao
xl_partition -tm
```

In all the commands, the default value of m is 5. It can be changed by specifying the new value using `-n` option for each command.

One useful command not described thus far is **_xl_nodevalue -v support**. It prints nodes that have at least *support* fanins. This command is used to make sure that a feasible network has indeed been obtained. For example, if $m = 5$, *_xl_nodevalue -v 6* prints the 5-infeasible nodes.

A.2.4 Targeting Xilinx 3090

For the Xilinx 3090 CLB, mis-fpga has two special commands:

xl_merge	identify function-pairs to be placed on the same block
xl_decomp_two	cube-packing targeted for two-output CLB's

The approach currently followed is to minimize first the number of single-output blocks using the script(s) described above and then use *xl_merge* as a post-processing step to place a maximum number of mergeable function-pairs in one CLB each. The conditions for mergeability of two functions can be specified by placing upper bounds on the number of inputs to each function, the number of common inputs, and the total number of inputs. This problem can be formulated as the maximum cardinality matching problem. An exact solution can be generated using *lindo*, an integer linear programming package. If *lindo* is not found in the path, *xl_merge* switches to a heuristic to solve the matching problem.

A different approach that sometimes gives better results is the following. First obtain a 4-feasible network (by running any one of the above scripts with `-n 4`) and then use *xl_merge*

without -F option. Since there are no nodes with 5 inputs, all nodes can potentially pair with other nodes. As a result the number of matches is higher. When -F option is not used, *xl_merge* first finds maximum number of mergeable function pairs and then applies block minimization on the subnetwork consisting of unmatched nodes of the network. This sometimes results in further savings. We recommend that the user run the scripts for both 4-feasible and 5-feasible cases, apply *xl_merge*, and pick the network that uses fewer CLBs.

The command *xl_decomp_two* does decomposition of the network targeted for two-output CLBs. It is a modification of the cube-packing approach. However, it does not guarantee a feasible network; other decomposition commands should be run afterwards to ensure feasibility. The details of the algorithm corresponding to *xl_decomp_two* are not described in the thesis.

A.2.5 Performance Optimization

xl_rl performs placement-independent logic optimizations for performance. Given an m-feasible network (preferably generated by *speed_up*, the delay optimization command in SIS that generates a 2-feasible network), *xl_rl* reduces the number of levels of LUTs used in the network. Then, any block count minimization command (e.g., *xl_cover*, *xl_partition* (without -m option)) can be applied to reduce the number of LUTs without increasing the number of levels. The code for placement-dependent phase is not distributed.

A.3 Synthesis for MUX-based Architectures

We have implemented mapping algorithms for Actel's *act1* architecture. The command currently available is called **act_map**. The simpler architecture *act* of Figure 8.3 (that is, *act1* with the OR gate removed) is also supported. No library needs to be read. The algorithm of Section 8.2.2 is used. The user may specify the number of iterations to be used in the iterative improvement phase, limits on the numbers of fanins of nodes for collapsing or decomposition, etc. He also has the option to write out for the final mapped network a net-list file in a format similar to that of *bdnet*. Each node of the mapped network is realizable by one basic block.

Currently, the method based on ITEs is not distributed.

Bibliography

[1] P. Abouzeid, K. Bouchet, K. Sakouti, G. Saucier, and P. Sicard. Lexicographical Expression of Boolean Function for Multilevel Synthesis of High Speed Circuits. In *Proc. SASHIMI 90*, pages 31–39, October 1990.

[2] Actel. *ACT1 Family Gate Arrays: Design Reference Manual.*

[3] R. L. Ashenhurst. The Decomposition of Switching Functions. In *Proceedings of International Symp on Theory of Switching Functions*, 1959.

[4] M. Beardslee, B. Lin, and A. Sangiovanni-Vincentelli. Communication based Logic Partitioning. In *Proceedings of the European Design Automation Conference*, 1992.

[5] M. Beardslee and A. Sangiovanni-Vincentelli. An Algorithm for Improving Partitions of Pin-Limited Multi-Chip Systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 378–385, 1993.

[6] T. Besson, H. Bouzouzou, M. Crastes, and G. Saucier. Synthesis on Multiplexor-based Programmable Devices using (Ordered) Binary Decision Diagrams. In *Euro ASIC*, June 1992.

[7] N. Bhat. Library-based Mapping for LUT FPGAs Revisited. In *Proceedings of the International Workshop on Logic Synthesis*, May 1993.

[8] N. Bhat and D. Hill. Routable Technology Mapping for FPGA's. In *1st International ACM/SIGDA Workshop on FPGAs*, 1992.

[9] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proc. of* 27^{th} *Design Automation Conference*, pages 40–45, June 1990.

[10] R. K. Brayton and C. McMullen. The Decomposition and Facorization of Boolean Expressions. In *Proceedings of the Int'l Symposium on Circuits and Systems*, pages 49–54, Rome, May 1982.

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

[12] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[13] F. M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Kluwer Academic Publishers, 1990.

[14] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[15] J. R. Burch and D. E. Long. Efficient Boolean Function Matching. In *Proceedings of the International Conference on Computer-Aided Design*, pages 408–411, November 1992.

[16] J. Cong and Y. Ding. An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup Table Based FPGA Designs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 48–53, 1992.

[17] J. Cong and Y. Ding. On Area/Depth Trade-Off in LUT-Based FPGA Technology Mapping. In *Proceedings of the Design Automation Conference*, pages 213–218, 1993.

[18] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in MIS. In *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[19] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State Assignment of Finite State Machines Targeting Multi-Level Logic Implementations. In *IEEE Transactions on Computer-Aided Design*, pages 1290–1300, December 1988.

[20] S. Devadas and A. R. Newton. Exact Algorithms for Output Encoding, State Assignment and Four-Level Boolean Minimization. In *Proceedings of the* 23^{rd} *Hawaii Int'l Conference on System Sciences*, January 1990.

[21] S. Ercolani and G. De Micheli. Technology Mapping Electrically Programmable Gate Arrays. In *Proceedings of the Design Automation Conference*, pages 234–239, 1991.

[22] C. Fiduccia and R. Mattheyses. A Linear-time Heuristic for Improving Network Partitions. In *Proceedings of the Design Automation Conference*, pages 175–181, 1982.

[23] D. Filo, J. C. Yang, F. Mailhot, and G. De Micheli. Technology Mapping for a Two-Output RAM-based Field-Programmable Gate Arrays. In *Proceedings of the European Design Automation Conference*, pages 534–538, 1991.

[24] R. J. Francis. A Tutorial on Logic Synthesis for Lookup-Table Based FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 40–47, 1992.

[25] R. J. Francis, J. Rose, and K. Chung. Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays. In *Proceedings of the Design Automation Conference*, pages 613–619, 1990.

[26] R. J. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs. In *Proceedings of the Design Automation Conference*, pages 227–233, 1991.

[27] R. J. Francis, J. Rose, and Z Vranesic. Technology Mapping of Lookup Table-Based FPGAs for Performance. In *Proceedings of the International Conference on Computer-Aided Design*, pages 568–571, 1991.

[28] M. Fujita and Y. Matsunaga. Multi-level Logic Minimization based on Minimal Support and its Application to the Minimization of Look-up Table Type FPGAs . In *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[29] A. Gamal, J. Greene, E. Rogoyski, K. A. El-Ayat, and A. Mohsen. An Architecture for Electrically Configurable Gate Arrays. In *IEEE Journal of Solid State Circuits*, April 1989.

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., NY, 1979.

[31] J. Gimpel. A reduction technique for prime implicant tables. *IRE Transactions on Electronic Computers*, August 1965.

[32] A. V. Goldberg, E. Tardos, and R. E. Tarjan. Network flow algorithms. Technical Report STAN-CS-89-1252, Dept. of Computer Science, Stanford University, March 1989.

[33] G. Hachtel, X. Du, and P. Moceyunas. Algorithms for state assignment based on multi-level representations. In *Proceedings of the 23rd Hawaii Int'l Conference on System Sciences*, January 1990.

[34] C. Halatsis and N. Gaitanis. Irredundant Normal Forms and Minimal Dependence Sets of a Boolean Function. In *IEEE Transactions on Computers*, pages 1064–1068, November 1978.

[35] T. Hwang, R. M. Owens, and M. J. Irwin. Efficiently Computing Communication Complexity for Multilevel Logic Synthesis. In *IEEE Transactions on Computer-Aided Design*, pages 545–554, May 1992.

[36] R. M. Karp and J. P. Roth. Minimization over Boolean Graphs. In *IBM Journal of Research and Development*, April 1962.

[37] K. Karplus. Using If-then-else Dags for Multi-level Logic Minimization. In *UCSC-CRL-88-29, Technical Report, Board of Studies in Computer Engineering, University of California, Santa Cruz*, December 1988.

[38] K. Karplus. Amap: A Technology Mapper for Selector-based Field-Programmable Gate Arrays. In *Proceedings of the Design Automation Conference*, pages 244–247, 1991.

[39] K. Karplus. Xmap: A Technology Mapper for Table-Lookup Field-Programmable Gate Arrays. In *Proceedings of the Design Automation Conference*, pages 240–243, 1991.

[40] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. In *The Bell System Technical Journal*, pages 291–307, 1970.

[41] K. Keutzer. Dagon: Technology Binding and Local Optimization by DAG Matching. In *Proceedings of the Design Automation Conference*, pages 341–347, 1987.

[42] R. Kužnar, F. Brglez, and K. Kozminski. Cost Minimization of Partitions into Multiple Devices. In *Proceedings of the Design Automation Conference*, pages 315–320, Dallas, Texas, June 1993.

[43] Y. T. Lai, M. Pedram, and S. Vrudhula. BDD Based Decomposition of Logic Functions with Application to FPGA Synthesis. In *Proceedings of the Design Automation Conference*, pages 642–647, Dallas, Texas, June 1993.

[44] W. Lam and R. K. Brayton. On Relationship between ITE and BDD. In *Proceedings of the International Conference on Computer Design*, pages 448–451. IEEE, 1992.

[45] L. Lavagno. Exact and heuristic algorithm for binate covering problem. *EE290ls Class Project, EECS Dept., University of California at Berkeley*, May 1989.

[46] L. Lavagno, S. Malik, R.K. Brayton, and A. Sangiovanni-Vincentelli. Symbolic minimization of multilevel logic and the input encoding problem. *IEEE Transactions on Computer-Aided Design*, 11(7):825–843, July 1992.

[47] E. L. Lawler. *Combinational Optimization: Networks and Matroids*. New York: Holt, Rinehart and Winston, 1976.

[48] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In *Proceedings of* 3^{rd} *CalTech Conference on VLSI*, March 1983.

[49] B. Lin and A. R. Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *Proceedings of the International Conference on Very Large Scale Integration*, 1989.

[50] Bill Lin and F. Somenzi. Minimization of Symbolic Relation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 88–91. IEEE, 1990.

[51] O. B. Lupanov. A Method of Circuit Synthesis. In *Izv. V. U. Z. Radiofiz.*, volume Vol. 1, No. 1, pages 120–140, 1958.

[52] F. Mailhot and G. De Micheli. Technology Mapping using Boolean Matching and Don't Care Sets. In *Proceedings of the European Design Automation Conference*, pages 2120–216, 1990.

[53] S. Malik, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 6–9, November 1988.

[54] H. J. Mathony. A universal logic design algorithm and its application to the synthesis of two-level switching circuits. *IEE Proceedings*, 136(3):171–177, May 1989.

[55] P. McGeer, J. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. ESPRESSO-SIGNATURE: A New Exact Minimizer for Logic Functions. In *Proceedings of the Design Automation Conference*, June 1993.

[56] M. Mehendale, C. H. Shaw, and D. Wilmoth. ALFA: Automatic Library Generation for Logic Module Based FPGA's. In *1st International ACM/SIGDA Workshop on FPGAs*, 1992.

[57] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|}.E)$ Algorithm for Finding Maximum Matching in General Graphs. In *Symposium on the Foundations of Computer Science*, 1980.

[58] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal State assignment of Finite State Machines. In *IEEE Transactions on Computer-Aided Design*, pages 269–285, July 1985.

[59] R. Murgai, R. K Brayton, and A. Sangiovanni-Vincentelli. On Clustering for Minimum Delay/Area. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[60] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli. An Improved Synthesis Algorithm for Multiplexor-based PGA's. In *Proceedings of the Design Automation Conference*, pages 380–386, 1992.

[61] R. Murgai, R. K Brayton, and A. Sangiovanni-Vincentelli. Sequential Synthesis for Table Look Up Programmable Gate Arrays. In *Proceedings of the Design Automation Conference*, 1993.

[62] R. Murgai, Y. Nishizaki, N. Shenoy, R. K Brayton, and A. Sangiovanni-Vincentelli. Logic Synthesis for Programmable Gate Arrays. In *Proceedings of the Design Automation Conference*, pages 620–625, 1990.

[63] R. Murgai, N. Shenoy, R. K Brayton, and A. Sangiovanni-Vincentelli. Improved Logic Synthesis Algorithms for Table Look Up Architectures. In *Proceedings of the International Conference on Computer-Aided Design*, pages 564–567, 1991.

[64] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Performance Directed Synthesis for Table Look Up Pragrammable Gate Arrays. In *Proceedings of the International Conference on Computer-Aided Design*, pages 572–575, 1991.

[65] M. Pedram and N. Bhat. Layout Driven Technology Mapping. In *Proceedings of the Design Automation Conference*, pages 99–105, San Francisco, June 1991.

[66] M. Pedram and N. Bhat. Layout Driven Technology Mapping. In *Proceedings of the Design Automation Conference*, pages 99–105, 1991.

[67] J. Rubinstein, P. Penfield, and M. A. Horowitz. Signal Delay in RC Tree Networks. In *IEEE Transactions on CAD*, pages 119–127, July 1983.

[68] R. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, University of California, Berkeley, 1989.

[69] A. Saldanha, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. A framework for satisfying input and output encoding constraints. In *Proceedings of the* 28^{th} *Design Automation Conference*, pages 170–175, June 1991.

[70] J. E. Savage. *The Complexity of Computing.* Wiley Interscience Publication, 1976.

[71] H. Savoj, M. Silva, R. K. Brayton, and A. Sangiovanni-Vincentelli. Boolean Matching in Logic Synthesis. In *Proceedings of the European Design Automation Conference*, pages 168–174, 1992.

[72] H. Savoj, H. Touati, and R. K. Brayton. Extracting Local Don't Cares for Network Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[73] H. Savoj, H. Y. Wang, and R. K. Brayton. Improved Scripts in MIS-II for Logic Minimization of Combinational Circuits. In *Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[74] P. Sawkar and D. Thomas. Area and Delay Mapping for Table-Look Up Based Field Programmable Gate Arrays. In *Proceedings of the Design Automation Conference*, pages 368–373, Los Angeles, June 1992.

[75] M. Schlag, P. K. Chan, and J. Kong. Empirical Evaluation of Multilevel Logic Minimization Tools for an FPGA Technology. In *Oxford 1991 International Workshop on Field Programmable Logic and Applications*, 1991.

[76] M. Schlag, J. Kong, and P. K. Chan. Routability-Driven Technology Mapping for LookUp Table-Based FPGAs. In *UCSC-CRL-92-06, Technical Report, Computer Research Lab., University of California, Santa Cruz*, February 1992.

[77] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[78] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis.* Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[79] C. E. Shannon. The Synthesis of Two-Terminal Switching Circuits. In *The Bell System Technical Journal*, volume Vol. 28, pages 59–98, 1949.

[80] P. Sicard, M. Crastes, K. Sakouti, and G. Saucier. Automatic Synthesis of Boolean Functions on Xilinx and Actel Programmable Devices. In *Euro ASIC*, pages 142–145, May 1991.

[81] K. J. Singh, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing Optimization of Combinational Logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 282–285, 1988.

[82] H. Touati. *Performance-Oriented Technology mapping.* PhD thesis, U.C. Berkeley, 1990.

[83] H. Touati, N. Shenoy, and A. Sangiovanni-Vincentelli. Retiming for Table-Lookup Field Programmable Gate Arrays. In *1st International ACM/SIGDA Workshop on FPGAs*, 1992.

[84] T. Villa and A. Sangiovanni-Vincentelli. Nova: State assignment for optimal two-level logic implementations. In *IEEE Transactions on Computer-Aided Design*, September 1990.

[85] A. Wang. *Algorithms for Multi-Level Logic Optimization.* PhD thesis, University of California, Berkeley, 1989.

[86] I. Wegener. *The Complexity of Boolean Functions.* Wiley-Teubner, 1987.

[87] N-S. Woo. A Heuristic Method for FPGA Technology Mapping Based on Edge Visibility. In *Proceedings of the Design Automation Conference*, pages 248–251. ACM-IEEE, June 1991.

[88] Xilinx Inc., 2069, Hamilton Ave., San Jose, CA-95125. *The Programmable Gate Array Book.*

[89] S. Yang and M. Ciesielski. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):4–12, January 1991.

[90] Saeyang Yang. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. In *Technical report*. MCNC, January 1991.