Tsutomu Sasao

Memory-Based Logic Synthesis

Springer

Memory-Based Logic Synthesis

Tsutomu Sasao

Memory-Based Logic Synthesis

Springer

Tsutomu Sasao
Kyushu Institute of Technology
Department of Computer Science and Electronic
Iizuka, Japan
sasao@cse.kyutech.ac.jp

ISBN 978-1-4419-8103-5 e-ISBN 978-1-4419-8104-2
DOI 10.1007/978-1-4419-8104-2
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011922264

Preface

This book describes the realization of logic functions using memories. The proposed methods can be used to impalement designs in field programmable gate arrays (FPGAs) that contain both small-scale memories, called look-up tables (LUTs), and medium-scale memories, called embedded memories.

The basis for memory-based design is functional decomposition, which replaces a large memory with smaller ones. An LUT cascade is introduced as a new architecture for logic synthesis. This book introduces the C-measure, which specifies the complexity of Boolean functions. Functions with a suitably small C-measure can be efficiently realized by LUT cascades.

This book also shows logic design methods for index generation functions. An index generation function is a mathematical model for an address table which can be used to store internet addresses. Such a table must be updated frequently, and the operation must be performed as fast as possible. In addition, this book introduces hash-based design methods, which efficiently realize index generation functions by pairs of smaller memories. Main applications include: IP address table lookup, packet filtering, terminal access controllers, memory patch circuits, virus scan circuits, fault map of memory, and pattern matching.

This book is suitable for both FPGA system designers and CAD tool developers. To read the book, a basic knowledge of logic design and discrete mathematics is required. Each chapter contains examples and exercises. Solutions for the exercises are also provided.

Tsutomu Sasao

Acknowledgements

This research is supported in part by the Grants in Aid for Scientific Research of JSPS, the grants of MEXT knowledge Cluster Project, and Regional Innovation Cluster Program. Many people were involved in this project: Jon T. Butler, Masayuki Chiba, Bogdan Falkowski, Yukihiro Iguchi, Kazunari Inoue, Atsumu Iseno, Yoshifumi Kawamura, Hisashi Kajihara, Munehiro Matsuura, Alan Mishchenko, Hiroki Nakahara, Kazuyuki Nakamura, Shinobu Nagayama, Marek Perkowski, Hui Qin, Marc Riedel, Takahiro Suzuki, Akira Usui, and Yuji Yano.

Most materials in this book have been presented at various conferences: IWLS, DSD, ISMVL, ICCAD, ICCD, DAC, ASPDAC, FPL, ARC, and SASIMI, as well as journals: IEEETCAD, IEICE and IJE. In many cases, reviewers comments considerably improved the quality of the materials.

Preliminary versions of this book were used as a textbook for seminars in our group. Numerous improvements were proposed by the students of Kyushu Institute of Technology: Taisuke Fukuyama, Takuya Nakashima, Takamichi Torikura, Satoshi Yamaguchi, Takuya Eguchi, Yoji Tanaka, Yosuke Higuchi, Takahiro Yoshida, and Meiji University: Atsushi Ishida, Kensuke Kitami, Koki Shirakawa, Kengo Chihara, Shotaro Hara, Akira Yoda.

Prof. Jon T. Butler read through the entire manuscript repeatedly and made important corrections and improvements.

Dr. Alan Mishchenko's comments on the final manuscript were also appreciated.

Contents

Chapter 1
Introduction

1.1 Motivation

Two of the most crucial problems in VLSI design are their high design cost and long design time. A solution to these problems is to use programmable architectures. Programmable LSIs reduce the hardware development cost and time drastically, since one LSI can be used for various applications. This book considers realizations of logic functions by programmable architectures. Various methods exist to realize multiple-output logic functions by programmable architectures. Among them, memories and programmable logic arrays (PLAs) directly realize logic functions. However, when the number of input variables n is large, the necessary hardware becomes too large. Thus, field programmable gate arrays (FPGAs) are widely used. Unfortunately, FPGAs require layout and routing in addition to logic design. Thus, quick reconfiguration is not so easy.

A look-up table (LUT) cascade is a series connection of memories. It efficiently realizes various classes of logic functions. Since the architecture is simple, LUT cascades are suitable for the applications where frequent update is necessary. This book explores the design and application of LUT cascades. In addition, it shows memory-based methods to realize index generation functions, which are useful for pattern matching and communication circuits.

1.2 Organization of the Book

This book consists of 13 chapters. Figure 1.1 shows the relation between the chapters, where the arrows show the order to read the chapters. For example, Chaps. 6 and 7 can be read after reading Chap. 5. Chapter 2 reviews the basic elements used in this book: memory, PLA, content addressable memory (CAM), and FPGA.

Chapter 3 reviews the definitions and basic properties of logic functions. Functional decomposition is the most important theory covered.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_1,

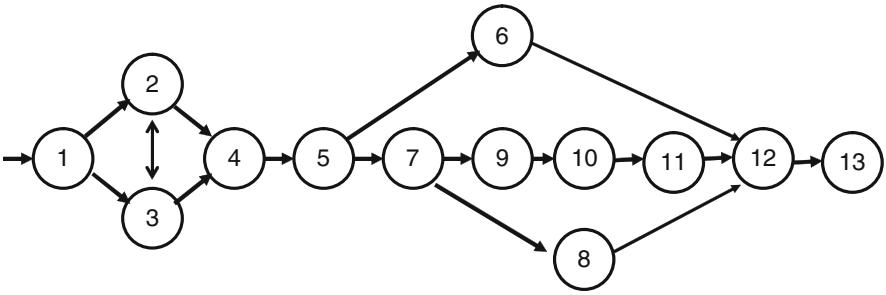


Fig. 1.1 Relation among chapters

Chapter 4 shows multiplexer-based synthesis of logic functions. This method is universal, i.e., it can realize any logic function. However, the number of LUTs required to realize the function tends to increase exponentially with n, the number of input variables.

Chapter 5 shows a method to realize logic functions using LUT cascades. This method is not universal, i.e., it can be applied to only a limited class of logic functions. However, the number of LUTs to realize the function tends to increase linearly with n, the number of input variables. The C-measure is used to estimate the circuit size.

Chapter 6 shows a method to reduce the number of LUTs by considering the encoding of the functional decomposition.

Chapter 7 shows a class of functions whose C-measures are small. Such functions are efficiently realized by LUT cascades.

Chapter 8 considers the C-measure of the functions whose number of minterms is limited. It shows that when the number of minterms is small, the C-measure is also small.

Chapter 9 introduces index generation functions, which are useful to design IP address tables, terminal access controllers for local area network, etc.

Chapter 10 introduces a hash-based synthesis of index generation functions. The method is similar to a hash table for software. Similar to the case of software realization, collisions of data may occur. However, in hardware, we can avoid collisions by using various circuits that work concurrently.

Chapter 11 shows a reduction method for the number of variables for the hash circuit. It introduces incompletely specified functions and shows an algorithm to reduce the number of variables.

Chapter 12 shows various methods to realize index generation functions.

Chapter 13 summarizes the book.

Chapter 2
Basic Elements

This chapter reviews memory, programmable logic array (PLA), content addressable memory (CAM), and field programmable gate array (FPGA).

2.1 Memory

Semiconductor memories have a long history of research, and various types of memories have been developed.

A dynamic random access memory (**DRAM**) uses only a single transistor to store one bit. However, the peripheral circuit is rather complex, and periodical refreshing is necessary. Thus, DRAM is suitable for large-scale memory, but not for a small-scale memory.

A static random access memory (**SRAM**) uses six transistors to store one bit. The peripheral circuit is not so complex as in DRAM, and refreshing is unnecessary. Thus, SRAM is suitable for small-scale memory, but not for large-scale memory. Both DRAM and SRAM are **volatile**, i.e., if the power supply is tuned off, the data are lost.

Read only memories (**ROM**s), however are **nonvolatile** and are used for storing fixed data. For rewritable ROMs, fabrication often requires a dedicated process and such ROMs are expensive.

To reduce the amount of hardware, memories are often implemented as a two-dimensional structure that is shown in Fig. 2.1. The row address decoder selects one row. And the sense amplifiers read the data for the selected row. Finally, the column address decoder selects one bit from the word. In many cases, the memory has a clock. Such a memory is **synchronous**. Many FPGAs contain memories, and many of them are synchronous. However, **asynchronous** memories (i.e., memories operating without clock pulses) are also available.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_2,

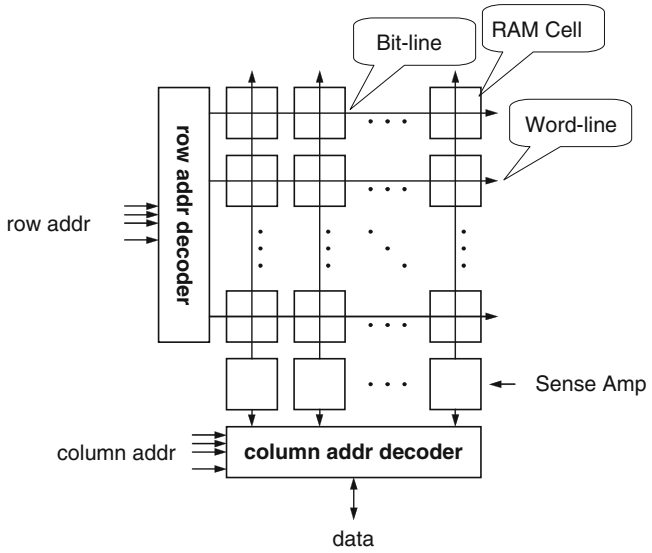


Fig. 2.1 Memory architecture

2.2 Programmable Logic Array

PLAs were often used for implementing controllers of microprocessors [94]. A PLA consists of the AND array and the OR array as shown in Fig. 2.2. For example, a 3-input 2-output function can be implemented as shown in Fig. 2.3. In the AND array, a cross-point denotes the AND connection, while in the OR array, a cross-point denotes the OR connection. Both **static** and **dynamic PLAs** exist. In the dynamic PLA, a clock pulse is used to read out the data. For large-scale PLAs, dynamic realizations are often used, since static CMOS realization is too large and the NMOS realization dissipates much static power. In a **rewritable PLA**, each cross-point consists of a switch and a memory element [3]. Let n be the number of inputs, W be the number of columns, and m be the number of outputs. Then, the size of a PLA is approximated by $W(2n + m)$. The number of columns W can be reduced by minimizing sum-of-products (SOPs) expressions. Thus, the logic design is relatively easy.

2.3 Content Addressable Memory

An ordinary memory such as that introduced in Sect. 2.1 uses a memory address to access data and return the contents of the memory. On the contrary, a **CAM** searches using the contents of the memory, and returns the address where the supplied data were found.

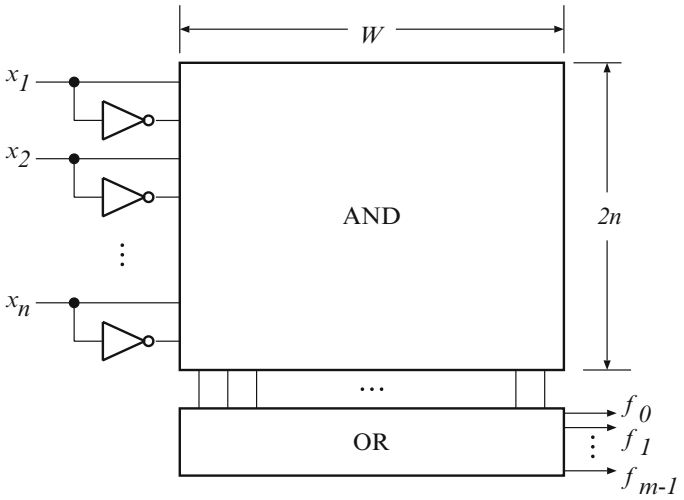
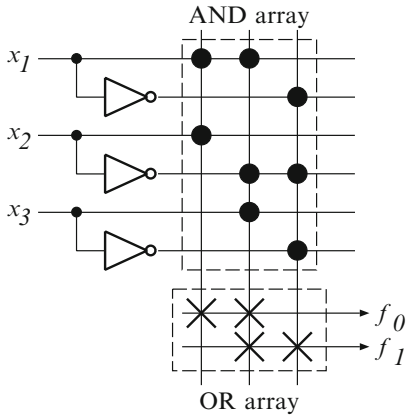


Fig. 2.2 Programmable logic array (PLA)

Fig. 2.3 An example of PLA



CAMs are often used in pattern matching [5]. Two types of CAMs exist: a binary CAM (**BCAM**) and a ternary CAM (**TCAM**). In a BCAM, each cell can take one of two states: 0 and 1. BCAMs are used for exact match. In a TCAM, each cell can take one of three states: 0, 1, and X (*don't care* that matches both 0 and 1). A TCAM can be used for implementing the longest prefix match to be explained in Chap. 7, and range match. To store one bit of information, an SRAM requires 6 transistors, while a TCAM requires 16 transistors [90]. Thus, a TCAM is more expensive than SRAM. Figure 2.4 shows an example of a CAM. In the CAM, for the input (i.e., search data) 01101, the second and the third rows match. However, the **priority encoder** selects the lowest address, i.e., 01. Thus, the output of the CAM is 01, the address of the second line. When the input is 11111, the CAM shows that there is no match.

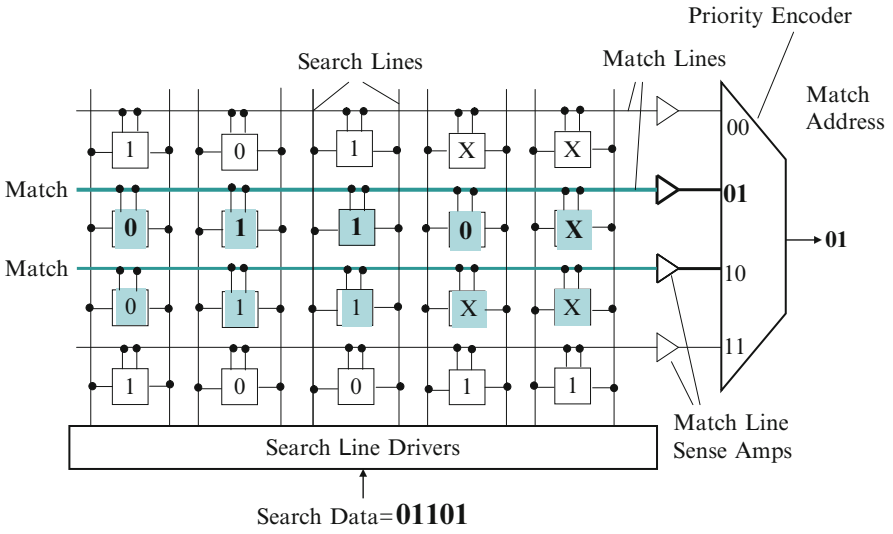


Fig. 2.4 CAM architecture

Fig. 2.5 Comparison of SRAM and CAM

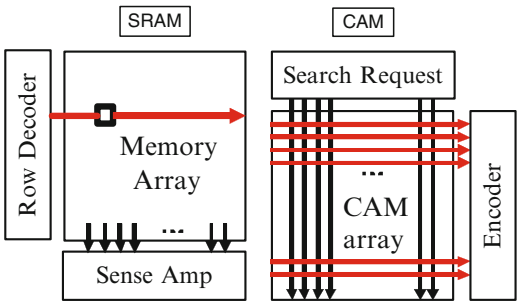


Figure 2.5 compares an SRAM with a CAM. Note that in an SRAM, only one row is activated at a time, while in a CAM, all the rows are activated at the same time. In a CAM, all the rows are charged and discharged in every search operation. Thus, a CAM dissipates more power than an SRAM with the same number of bits.

2.4 Field Programmable Gate Array

Most **FPGA**s have an **island-style** architecture as shown in Fig. 2.6. In this architecture, logic elements are surrounded by interconnection elements such as switch blocks, connection blocks, and wiring. Logic elements are, in most cases, look-up tables (**LUT**s). A K**-LUT** is a module that realizes an arbitrary K-variable function. Thus, logically it is a K-input memory. However, unlike an ordinary

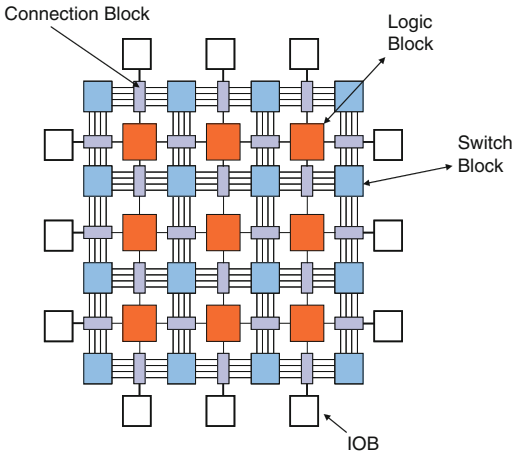


Fig. 2.6 Island-style FPGA

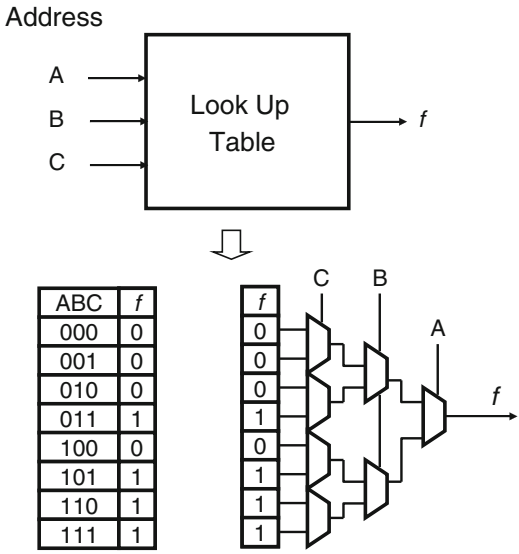


Fig. 2.7 LUT for FPGA

(large-scale) memory presented before, it is often implemented by a register and selector as shown Fig. 2.7. No clock pulse is necessary to read out the data; therefore, it is asynchronous. In modern FPGAs, more than 90% of the chip area is for interconnections. Thus, layout design is as important as logic design. Also, layout design often requires more CPU time than logic design.

Most FPGAs use SRAMs to store configuration data. At power-up, FPGAs are blank, and an external nonvolatile memory (EEPROM, flash RAM, MRAM, or FeRAM) is used to restore the configuration data into SRAM. FPGAs need to be configured from the external nonvolatile memory at every system power-up time. The configuration time is tens to hundreds of milliseconds. Also, the

SRAM for configuration must be kept on even when the system is not used. As programmable circuits become larger, the numbers of LUTs and embedded memories increase, and the power dissipation of SRAMs due to leakage current becomes more serious. In the past, FPGAs with 4 or 5 input LUTs were believed to be the most efficient [99, 100]; however, in the current technology, FPGAs with 6 inputs are the standard [4, 7, 158]. Thus, we have the following:

Problem: Given an n-variable (multiple-output) function f, find the minimum number of 6-LUTs needed to realize f, where $n \leq 20$.

Unfortunately, this is a very hard problem. So, we try to obtain an upper bound by using properties or measures of the function. A property of the function should be easy to detect, such as symmetry, and measures should be efficient to calculate. Such measures include:

- The number of variables (i.e., the support size)
- The weight of the function [133]
- The C-measure of a function [133]
- The number of products in the SOP [78]
- The number of literals in the factored expression [78]
- The number of nodes in the decision diagram [107]

Chapters 4 and 5 consider the number of K-LUTs to realize a given function.

2.5 Remarks

This chapter briefly introduced memory, CAM, and FPGA. As for FPGAs, various architectures are shown in Refs. [15, 100]. As for logic synthesis of FPGAs, see Refs. [20, 78]. In recent FPGAs, the architecture is quite complicated, and a single FPGA can implement a multiprocessor system.

Many commercial FPGAs contain memories in addition to LUTs. They are called **embedded memories** [4] or **embedded block RAMs** [156].

Problems

2.1. Design the CAM function shown in Fig. 2.4 by using a PLA.

2.2. Explain why asynchronous memories are used for LUTs in FPGAs instead of synchronous ones.

2.3. In the past, 4-LUTs or 5-LUTs were mainly used in FPGAs. However, nowadays, 6-LUTs are popular in FPGAs. Discuss the reason why 6-LUTs are preferred.

2.4. Explain why dynamic CMOS is used instead of static CMOS in PLAs.

2.5. Suppose that a TCAM with n inputs and W words is given. Show a method to implement the same function as the TCAM by using a PLA.

2.6. Suppose that a PLA with n inputs, m outputs, and W products is given. Show a method to implement the same function as the PLA by using a TCAM and a memory.

Chapter 3
Definitions and Basic Properties

This chapter first introduces logical expressions and functional decomposition. Then, it introduces some properties of symmetric functions.

3.1 Functions

Definition 3.1.1. A mapping $F : B^n \rightarrow \{0, 1, \ldots, k-1\}$, where $B = \{0, 1\}$ is a **binary-input integer valued function**. A mapping $F : B^n \rightarrow B$, where $B = \{0, 1\}$ is a **logic function**. If $F(\vec{a}_i) \neq 0$ $(i = 1, 2, \ldots, k)$ for k different input vectors, and $F = 0$ for other $(2^n - k)$ input vectors, then the **weight** of the function is k.

3.2 Logical Expression

Definition 3.2.1. Binary variables are represented by x_i $(i = 1, 2, \ldots, n)$. A **literal** of a variable x_i is either x_i, $\bar{x}_i$, or the constant 1. An AND of literals is a **product**, and an OR of products is a **sum-of-products (SOP) expression**.

Definition 3.2.2. Let $B = \{0, 1\}$, $X = (x_1, x_2, \ldots, x_t)$, and x_i can take a value in B. Then, we can consider that X takes its value from $P = \{0, 1, \ldots, 2^t - 1\}$. Let S be a subset $(S \subseteq P)$ of P. Then, X^S is a **literal** of X. When $X \in S$, $X^S = 1$, and when $X \notin S$, $X^S = 0$. Let $S_i \subseteq P_i$ $(i = 1, 2, \ldots, n)$, then ${X_1}^{S_1} {X_2}^{S_2} \cdots {X_n}^{S_n}$ is a **logical product**. $\bigvee_{(S_1, S_2, \ldots, S_n)} {X_1}^{S_1} {X_2}^{S_2} \cdots {X_n}^{S_n}$ is a **SOP** expression. If $S_i = P_i$, then ${X_i}^{S_i} = 1$ and the logical product is independent of X_i. In this case, the literal ${X_i}^{P_i}$ is redundant and can be omitted. A logical product is also called a **term** or a **product term**.

Example 3.2.1. When $t = 2$, we have $P = \{0, 1, 2, 3\}$, and $X = (x_1, x_2)$ takes four values. In this case, $X^{\{1\}}$, $X^{\{1,2\}}$, and $X^{\{0,1,2,3\}}$ are literals. Suppose that $P_1 = P_2 = P_3 = \{0, 1, 2, 3\}$. Then,

$$F = {X_1}^{\{0,1,2,3\}} {X_2}^{\{0\}} {X_3}^{\{0\}} \vee {X_1}^{\{0,1\}} {X_2}^{\{0,1,2,3\}} {X_3}^{\{1,3\}} \vee {X_1}^{\{0,1\}} {X_2}^{\{2\}} {X_3}^{\{0,1,2,3\}}$$

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_3,

is a SOP. Note that $X^{\{0,1,2,3\}} = 1$. In this case, $X^{\{0,1,2,3\}}$ is redundant and can be omitted. By removing redundant literals, this expression can be simplified to

$$F = {X_2}^{\{0\}} {X_3}^{\{0\}} \vee {X_1}^{\{0,1\}} {X_3}^{\{1,3\}} \vee {X_1}^{\{0,1\}} {X_2}^{\{2\}}.$$

■

3.3 Functional Decomposition

Definition 3.3.1. [8] Let $f(X)$ be a logic function, and (X_1, X_2) be a partition of the input variables, where $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$. The **decomposition chart** for f is a two-dimensional matrix with 2^k columns and 2^{n-k} rows, where each column and row is labeled by a unique binary code, and each element corresponds to the truth value of f. The function represented by a column is a **column function** and is dependent on X_2. Variables in X_1 are **bound variables**, while variables in X_2 are **free variables**. In the decomposition chart, the **column multiplicity** denoted by μ_k is the number of different column patterns.

Example 3.3.1. Figure 3.1 shows a decomposition chart of a 4-variable function. $\{x_1, x_2\}$ are the bound variables, and $\{x_3, x_4\}$ are the free variables. Since all the column patterns are different and there are four of them, the column multiplicity is $\mu_2 = 4$. ■

Theorem 3.3.1. *[25] For a given function f, let X_1 be the bound variables, let X_2 be the free variables, and let μ_k be the column multiplicity of the decomposition chart. Then, the function f can be realized with the network shown in Fig. 3.2. In this case, the number of signal lines connecting blocks H and G is $\lceil \log_2 \mu_k \rceil$.*

When the number of signal lines connecting two blocks is smaller than the number of input variables in X_1, we can often reduce the total amount of memory by the realization in Fig. 3.2 [51]. When $\mu_k = 2$, it is an **Ashenhurst decomposition** [8] or

		0 0 1 1	x_1
		0 1 0 1	x_2
0	0	0 0 0 1	
0	1	1 1 0 0	
1	0	0 1 0 0	
1	1	0 0 0 0	
x_3	x_4		

Fig. 3.1 Decomposition chart of a logic function

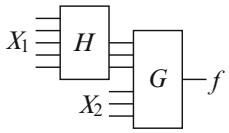


Fig. 3.2 Realization of a logic function by decomposition

a **simple disjoint decomposition**. When $\mu_k > 2$, it is a **Curtis decomposition** [25], also called **Roth–Karp decomposition** [101] or a **generalized decomposition**. The number of functions with Curtis decompositions is much larger than those with Ashenhurst decompositions.

A function with an Ashenhurst decomposition can be written as

$$f(X_1, X_2) = g(h_1(X_1), X_2).$$

A function with a Curtis decomposition can be written as

$$f(X_1, X_2) = g(h_1(X_1), h_2(X_1), \ldots, h_m(X_1), X_2),$$

where $m = \lceil \log_2 \mu \rceil$.

Lemma 3.3.1. *Consider a decomposition chart for $f(X_1, X_2)$, where $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$. Then, the column multiplicity does not change under the permutation of variables within X_1 and X_2. Thus,*

$$\mu(f(X_1, X_2)) = \mu(f(\tilde{X_1}, \tilde{X_2})),$$

where $\tilde{X_1} = (x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(k)})$, $\tilde{X_2} = (x_{\lambda(k+1)}, x_{\lambda(k+2)}, \ldots, x_{\lambda(n)})$, and, π and λ denote the permutation on $\{1, 2, \ldots, k\}$ and $\{k+1, k+2, \ldots, n\}$, respectively.

Lemma 3.3.2. *An arbitrary function of n variables can be decomposed as*

1. $f(X_1, X_2) = g(h_1(X_1), h_2(X_1), X_2)$, *where* $X_1 = (x_1, x_2, \ldots, x_{n-1})$ *and* $X_2 = (x_n)$.
2. $f(X_1, X_2) = g(h_1(X_1), h_2(X_1), h_3(X_1), h_4(X_1), X_2))$, *where* $X_1 = (x_1, x_2, \ldots, x_{n-2})$ *and* $X_2 = (x_{n-1}, x_n)$.

Figures 3.3 and 3.4 show the circuits for the above decompositions.

A functional decomposition can be of two types: A **disjoint decomposition**, where the bound set and the free set are disjoint, as shown in Fig. 3.5. A **nondisjoint decomposition**, where the bound set and free set have at least one common element. Figure 3.6 shows the case where one variable is shared by the bound set and the free set.

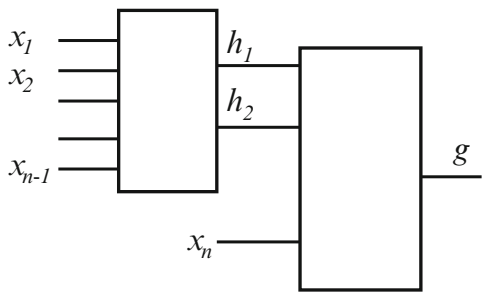


Fig. 3.3 Decomposition of an arbitrary function, where $X_2 = (x_n)$

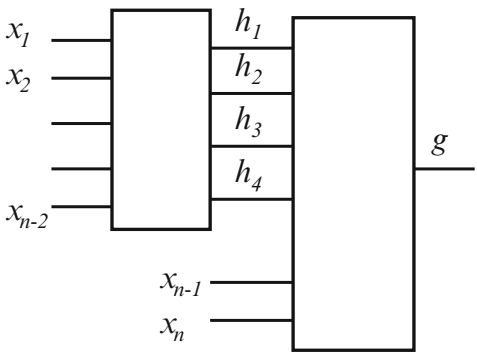


Fig. 3.4 Decomposition of an arbitrary function, where $X_2 = (x_{n-1}, x_n)$

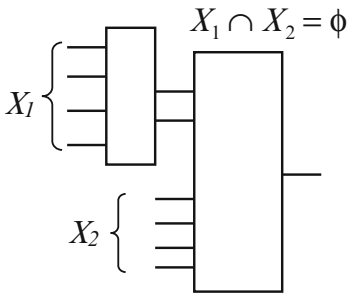


Fig. 3.5 Disjoint decomposition

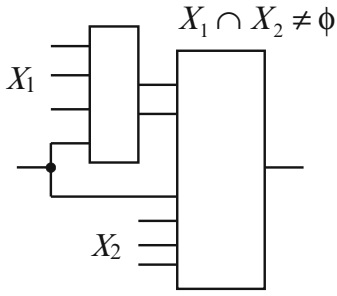


Fig. 3.6 Nondisjoint decomposition

3.4 Binary Decision Diagram

A **binary decision diagram** (**BDD**) [12] is a graphical representation of a logic function. It often has a more compact representation than other methods. Thus, BDDs are widely used in the computer-aided design of logic networks [65].

Definition 3.4.1. A BDD is a directed acyclic graph (DAG) with two terminal nodes: the 0-terminal node and the 1-terminal node. Each nonterminal node is labeled by an index of an input variable of the Boolean function, and has two outgoing edges: the 0-edge and the 1-edge. An **ordered BDD** (**OBDD**) is a BDD such that the input variables appear in a fixed order in all the paths of the graph, and each variable appears at most once in a path.

Fig. 3.7 Node elimination

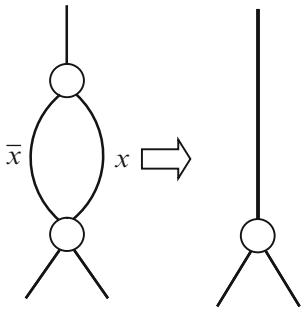
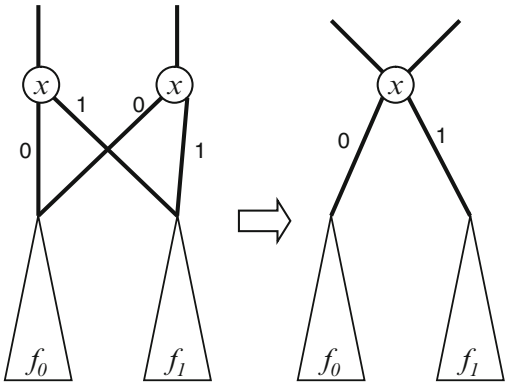


Fig. 3.8 Node sharing



Definition 3.4.2. A **reduced ordered BDD** (**ROBDD**) is obtained from an OBDD by applying the following two reduction rules:

1. Eliminate all the redundant nodes whose two edges point to the same node (Fig. 3.7).
2. Share all the equivalent nodes (Fig. 3.8).

A **quasi-reduced ordered BDD** (**QROBDD**) is obtained from an OBDD by applying the second reduction rule only.

Example 3.4.1. Figure 3.9 is a ROBDD for

$$f(x_1, x_2, x_3) = (\bar{x}_1\bar{x}_2 \vee x_1x_2)x_3.$$

On the other hand, Fig. 3.10 is a QROBDD for the same function. In Fig. 3.10, the left node for x_3 is redundant, and eliminated in the ROBDD. Note that in an QROBDD, all the paths from the root node to a terminal node encounter all the variables. ■

Theorem 3.4.1. *Let $X = (X_1, X_2)$ be a partition of X. Suppose that the QROBDD for $f(X)$ is partitioned into two blocks as shown in Fig. 3.11. Let k be the number of*

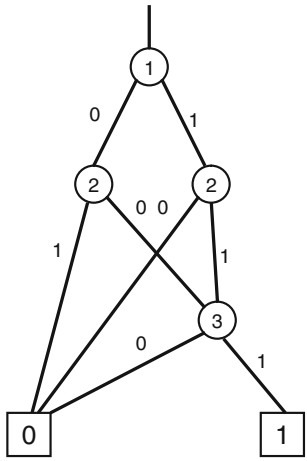


Fig. 3.9 Reduced ordered BDD

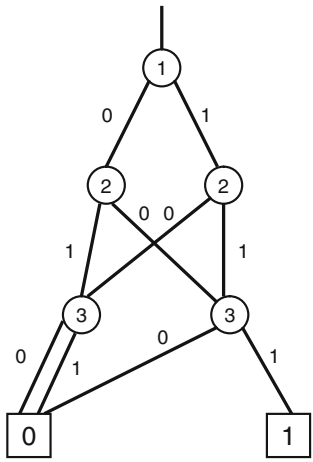


Fig. 3.10 Quasi-reduced OBDD

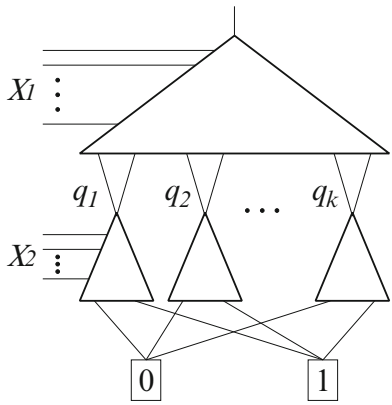


Fig. 3.11 Computation of column multiplicity for functional decomposition $f(X_1, X_2) = g(h(X_1), X_2)$

the nodes in the lower block that is adjacent to the upper block, and μ be the column multiplicity of the decomposition chart for $f = g(h(X_1), X_2)$. Then, $k = \mu$.

Example 3.4.2. Consider the function represented by the BDD in Figs. 3.9 and 3.10. Let $X_1 = (x_1, x_2)$ and $X_2 = (x_3)$. Then, the number of the nodes in the lower block that are adjacent to the upper block is two. The function can be decomposed as $f(X_1, X_2) = g(h(X_1), X_2)$, where $h(X_1) = \bar{x}_1\bar{x}_2 \vee x_1x_2$, and $g(h, X_2) = hx_3$. Figure 3.12 shows the decomposition chart. Note that the column multiplicity is two. In Fig. 3.10, the left node for x_3 represents the constant 0 function, while the right node for x_3 represents the x_3 function. It is clear that the number of different column patterns in the decomposition chart is equal to the number of nodes for x_3 in the QROBDD [106]. ■

Definition 3.4.3. A **multiterminal BDD** (**MTBDD**) is an extension of a BDD with multiple terminal nodes, each of which has an integer value.

An MTBDD can be used to represent a multiple-output function.

Example 3.4.3. Figure 3.13 is an MTBDD for 3-valued output function. Note that it is a reduced ordered MTBDD (ROMTBDD), but is not a quasi-reduced ordered MTBDD (QROMTBDD). ■

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

Fig. 3.12 Decomposition table of a logic function

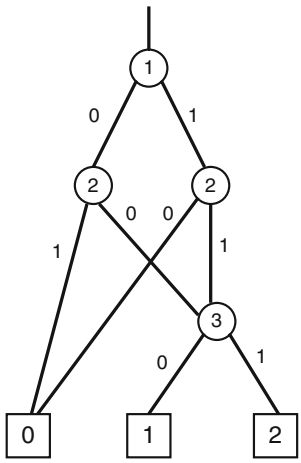


Fig. 3.13 Multiterminal BDD

3.5 Symmetric Functions

Functions that appear in arithmetic circuits, such as adders, often have symmetries. When logic functions have certain symmetries, they are often realized using fewer elements.

Definition 3.5.1. A function f is **totally symmetric** if any permutation of the variables in f does not change the function. A totally symmetric function is also called a **symmetric function**.

Definition 3.5.2. In a function $f(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n)$, if the function $f(x_1, \ldots, x_j, \ldots, x_i, \ldots, x_n)$ that is obtained by interchanging variables x_i with x_j is equal to the original function, then f is **symmetric with respect to** x_i **and** x_j. If any permutation of subset S of the variables does not change the function f, then f is **partially symmetric**.

Definition 3.5.3. The **elementary symmetric functions** of n variables are

$$
\begin{aligned}
S_0^n &= \bar{x}_1\bar{x}_2\cdots\bar{x}_n,\\
S_1^n &= x_1\bar{x}_2\bar{x}_3\cdots\bar{x}_{n-1}\bar{x}_n \vee \bar{x}_1x_2\bar{x}_3\cdots\bar{x}_{n-1}\bar{x}_n \vee \cdots \vee \bar{x}_1\bar{x}_2\bar{x}_3\cdots\bar{x}_{n-1}x_n,\\
&\cdots\cdots\cdots\cdots\cdots\\
S_n^n &= x_1x_2\cdots x_n.
\end{aligned}
$$

$S_i^n = 1$ iff exactly i out of n inputs are equal to 1. Let $A \subseteq \{0, 1, \ldots, n\}$. A symmetric function S_A^n is defined as follows:

$$S_A^n = \bigvee_{i \in A} S_i^n.$$

Example 3.5.1. $f(x_1, x_2, x_3) = x_1x_2x_3 \vee x_1\bar{x}_2\bar{x}_3 \vee \bar{x}_1x_2\bar{x}_3 \vee \bar{x}_1\bar{x}_2x_3$ is a totally symmetric function. $f = 1$ when all the variables are 1, or when only one variable is 1. Thus, f can be written as $S_1^3 \vee S_3^3 = S_{\{1,3\}}^3$. ■

Theorem 3.5.1. *An arbitrary n-variable symmetric function f is uniquely represented by elementary symmetric functions $S_0^n, S_1^n, \ldots, S_n^n$ as follows:*

$$f = \bigvee_{i \in A} S_i^n = S_A^n, \text{ where } A \subseteq \{0, 1, \ldots, n\}.$$

Lemma 3.5.1. *There are 2^{n+1} symmetric functions of n variables.*

Definition 3.5.4. Let $SB(n, k)$ be the n-variable symmetric function represented by the EXOR sum of all the products consisting of k positive literals:

$$
\begin{aligned}
SB(n,0) &= 1,\\
SB(n,1) &= \bigoplus_{} x_i,\\
SB(n,2) &= \bigoplus_{(i<j)} x_i x_j,\\
SB(n,3) &= \bigoplus_{(i<j<k)} x_i x_j x_k,\\
&\cdots\cdots\cdots\\
SB(n,n) &= x_1 x_2 \cdots x_n.
\end{aligned}
$$

Example 3.5.2.

$$
\begin{aligned}
SB(4,1) &= x_1 \oplus x_2 \oplus x_3 \oplus x_4.\\
SB(4,2) &= x_1x_2 \oplus x_1x_3 \oplus x_1x_4 \oplus x_2x_3 \oplus x_2x_4 \oplus x_3x_4.
\end{aligned}
$$

■

$SB(n,k)$ has been used as a benchmark function for an AND-EXOR logic minimizer [104]. The following two lemmas were derived by Komamiya [50] and reformulated by the author [110].

Lemma 3.5.2. *Let $x_1, x_2, \ldots, x_n$ be binary variables and r be an integer defined by $r = x_1 + x_2 + \cdots + x_n$, where $+$ is an ordinary integer addition. Let the binary representation of r be*

$$(y_k, y_{k-1}, \ldots, y_1, y_0)_2,\ y_j \in \{0,1\}\ (j = 0,1,\ldots,k).$$

In other words,

$$x_1 + x_2 + \cdots + x_n = 2^k y_k + 2^{k-1} y_{k-1} + \cdots + 2y_1 + y_0.$$

Then,

$$y_i = SB(n, 2^i).$$

Lemma 3.5.3. *Let $0 \le k_1 < k_2 < \cdots < k_s$, and $2^{k_1} + 2^{k_2} + \cdots + 2^{k_s} \le n$. Then,*

$$\bigwedge_{i=1}^{s} SB(n, 2^{k_i}) = SB\left(n, \sum_{i=1}^{s} 2^{k_i}\right).$$

Example 3.5.3.

$$
\begin{aligned}
SB(7,1)SB(7,2)SB(7,4) &= SB(7,7),\\
SB(4,1)SB(4,2) &= SB(4,3),\\
SB(6,2)SB(6,4) &= SB(6,6).
\end{aligned}
$$

■

Definition 3.5.5. $WGTn$ is an n-input $\lceil \log_2(n+1) \rceil$-output function. It counts the number of 1's in the inputs and represents it as a binary number.

Fig. 3.14 WGT7

By Lemma 3.5.2, WGTn produces $SB(n, 2^i)$, $(i = 0, 1, 2, \ldots, \lceil \log_2(n+1) \rceil - 1)$, where $\lceil a \rceil$ denotes the smallest integer greater than or equal to a.

Example 3.5.4. WGT7 has $x_1, x_2, \ldots, x_7$ as inputs and y_2, y_1, y_0 as outputs (Fig. 3.14). By Lemma 3.5.2, we have

$$y_2 = SB(7,4) = \bigoplus_{i<j<k<l} x_i x_j x_k x_l$$

$$y_1 = SB(7,2) = \bigoplus_{i<j} x_i x_j$$

$$y_0 = SB(7,1) = \bigoplus_{i=1}^{7} x_i$$

WGT 7 is also called as rd73. ∎

The following is an expansion method for symmetric functions using $SB(n,k)$ functions:

Theorem 3.5.2. *An arbitrary n-variable symmetric function f is represented by $y_i = SB(n, 2^i)$, $(i = 0, 1, 2, .., t)$ as follows:*

$$f = \bigoplus_{(a_0, a_1, \ldots, a_t)} g(a_0, a_1, ..., a_t) y_0^{a_0} y_1^{a_1} ... y_t^{a_t},$$

where $g(a_0, a_1, ..., a_t)$ is 0 or 1, and $t = \lceil \log_2(n+1) \rceil - 1$.

Proof. A symmetric function f depends only on the number of 1's in the inputs. Since WGTn counts the number of 1's in the input, we can represent f as a function of $y_0, y_1, ..., y_t$. □

Example 3.5.5. A 7-variable symmetric function S_0^7 can be represented as

$$\begin{aligned} S_0^7 &= \bar{y}_2 \bar{y}_1 \bar{y}_0 \\ &= \overline{SB(7,4)} \cdot \overline{SB(7,2)} \cdot \overline{SB(7,1)}. \end{aligned}$$

Note that the binary representation of 0 is $(0, 0, 0)$. Similarly, S_3^7 is represented as

$$\begin{aligned} S_3^7 &= \bar{y}_2 y_1 y_0 \\ &= \overline{SB(7,4)} \cdot SB(7,2) \cdot SB(7,1). \end{aligned}$$

Thus, $S^7_{\{0,3\}}$ is represented as

$$\begin{aligned} S^7_{\{0,3\}} &= S^7_0 \oplus S^7_3 = \bar{y}_2\bar{y}_1\bar{y}_0 \oplus \bar{y}_2 y_1 y_0 \\ &= \bar{y}_2(\bar{y}_1\bar{y}_0 \oplus y_1 y_0) = \bar{y}_2(y_1 \oplus \bar{y}_0) \\ &= \overline{SB(7,4)} \cdot (SB(7,2) \oplus \overline{SB(7,1)}). \end{aligned}$$

■

3.6 Technology Mapping

Logic synthesis using embedded memories can be considered as a special case of FPGA design, where the number of LUT inputs is large. Most existing methods use LUT-based technology mapping [33]. That is, given a combinational logic circuit, they partition it into sub-circuits depending on at most K variables [22,53,153]. This method was originally used for the LUT-based FPGA synthesis. Here, we introduce the basic idea.

Definition 3.6.1. [23, 57, 71] A combinational network can be converted into a DAG, where each node represents a logic gate (LUT), a **primary input** (PI), or a **primary output** (PO). When the output of the gate i is an input of gate j, a directed edge (i, j) exists. $input(v)$ denotes the set of nodes which are fanins of gate v. $output(v)$ denotes the set of nodes which are fanouts of gate v. A **cone** at v, denoted as C_v, is a subgraph consisting of v and its non-PI predecessors such that any path connecting a node in C_v and v lies entirely in C_v. Node v is the **root** of the cone. The **fanin size** of a cone is the number of input edges. A cone with K input edge is **K-feasible** and can be implemented with a K-LUT.

Example 3.6.1. In Fig. 3.15, a, b, c, d, e are PIs, and f denotes the PO. The cone of v consists of internal nodes v, t, u, s, p, q, and r. The fanin size of C_v is four. Thus, C_v is 4-feasible and can be implemented by a 4-LUT. ■

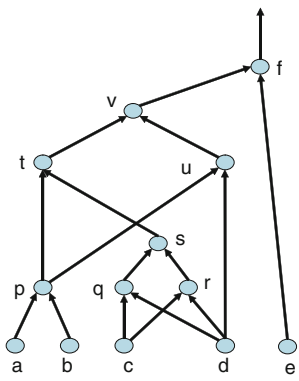


Fig. 3.15 Example of directed acyclic graph

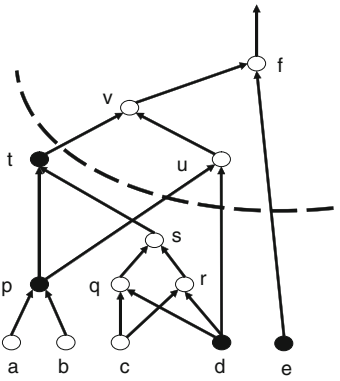


Fig. 3.16 Example of 4-feasible cut-set

Definition 3.6.2. Given a network N with a source s and a sink t, a **cut** $(\mathcal{N}_1, \mathcal{N}_2)$ is a partition of the nodes in the network such that $s \in \mathcal{N}_1$, $t \in \mathcal{N}_2$ and no nodes in $\mathcal{N}_2$ provide input to any node in $\mathcal{N}_1$. A **cut-set** of a cut is the set of all nodes v such that $v \in \mathcal{N}_1$, and v drives a node in $\mathcal{N}_2$. If the size of a cut-set is no more than K, then the cut is K**-feasible**. A **fanout-free cone** (FFC) at v, denoted FFC_v, is a cone of v, with output edge only originating from the root of the cone. A **maximum fanout free cone** (MFFC) is an FFC that maximizes the number of nodes contained in the FFC.

Example 3.6.2. In Fig. 3.16, let a be a source, and f be a sink. The set of nodes is partitioned into $\mathcal{N}_1 = \{a, b, c, d, e, p, q, r, s, t\}$ and $\mathcal{N}_2 = \{u, v, f\}$. The cut-set of the cut $(\mathcal{N}_1, \mathcal{N}_2)$ is $\{t, p, d, e\}$. It is 4-feasible. The MFFC of f is $\{p, q, r, s, t, u, v, f\}$. This is because a cone includes only non-PI nodes, by Definition 3.6.2. If the output of t is connected to an other output, then MMFC of f would be $\{u, v, f\}$. ■

In a technology mapping algorithm, usually, a given circuit is converted into an equivalent two-input network. If the DAG is represented as a set of trees, then the area minimization problem can be solved optimally using dynamic programming [47]. Unfortunately, most circuits have non-tree structure: There exist many fanouts and reconvergence. If the circuit is decomposed into a set of MFFCs, then it can also be solved optimally. In the approach of [22], the circuit is first mapped into LUTs using the best available algorithm. Then, it extracts large single-output and multiple-output fanout-free logic blocks and covers them entirely or partially by embedded memories. Since these design methods start from existing circuits, the quality of the solutions are not so good [57]. Improvements for technology mapping are shown in [71]. Especially, an efficient method to enumerate all the cuts up to $K = 6$ [71] and a method to compute useful cuts for any number of inputs [72]. In this book, the major tool for the memory-based design is a functional decomposition, so we will not go into the detail of the method.

Table 3.1 Approximation error for $1 - x$

x	$1 - x$	e^{-x}	Error(x)
0.001	0.999	0.99900050	0.00000050
0.010	0.990	0.99004983	0.00005034
0.100	0.900	0.90483742	0.00537491
0.200	0.800	0.81873075	0.02341344

3.7 The Mathematical Constant e and Its Property

Definition 3.7.1. The mathematical constant e is defined as

$$e = \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n.$$

Lemma 3.7.1. *When $0 < x << 1$, $1 - x$ can be approximated by e^{-x}.*

Proof. The Taylor expansion of a function $f(x)$ is

$$f(x) = f(0) + \frac{x}{1!} f^{(1)}(0) + \frac{x^2}{2!} f^{(2)}(0) + \frac{x^3}{3!} f^{(3)}(0) + \cdots + \frac{x^k}{k!} f^{(k)}(0) + \cdots.$$

Thus, e^x can be expanded as

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots.$$

When x is small, we need only to consider up to the second term, and we have

$$e^{-x} \simeq 1 - x.$$

□

Consider the approximation error of Lemma 3.7.1:

$$\text{Error}(x) = \frac{e^{-x} - (1 - x)}{1 - x}.$$

Table 3.1 shows that when $0 < x < 0.1$, the approximation error is quite small. Lemma 3.7.1 will be extensively used for the calculation of approximate probability in Chaps. 9, 10, and 11.

3.8 Remarks

Functional decomposition is the key technique in the memory-based logic synthesis. It efficiently represents a given Boolean function with reduced total amount of memory. Most Boolean functions do not have any functional decomposition [109]. However, practical functions often have functional decompositions. Thus, an

attempt to find functional decompositions is, in many cases, rewarding. Ashenhurst decompositions can be efficiently found by BDDs [10, 60]. However, as for Curtis decomposition, no efficient methods are known. Some heuristic methods are known [59, 147]. Excellent surveys on FPGA logic synthesis include [20, 78, 103].

Problems

3.1. Using the definition of $SB(n, k)$, verify the following equation:

$$SB(4, 1) \cdot SB(4, 2) = SB(4, 3).$$

3.2. Show that most $2^{2^k+2^{n-k+1}}$ functions have Ashenhurst decompositions of the form

$$f(X_1, X_2) = g(h(X_1), X_2),$$

where $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$.

3.3. Show that most $2^{m2^k+2^{n-k+m}}$ functions have Curtis decompositions of the form

$$f(X_1, X_2) = g(h_1(X_1), h_2(X_1), \ldots, h_m(X_1), X_2),$$

where $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$.

3.4. Let $f(X)$ be the function that counts the number of 1's in the inputs.

$$\{0, 1\}^5 \rightarrow \{0, 1, 2, 3, 4, 5\}.$$

That is, $f(\vec{a})$ denotes the number of 1's in $\vec{a}$. Write the decomposition chart and obtain the column multiplicity of the decomposition (X_1, X_2), where $X_1 = (x_1, x_2, x_3)$ and $X_2 = (x_4, x_5)$.

3.5. How many functions of n variables with weight k exist?

3.6. Represent the symmetric function of 9 variables: $S^9_{\{3,4,5,6\}}$ by

$$\begin{aligned} y_3 &= SB(9, 8), \\ y_2 &= SB(9, 4), \\ y_1 &= SB(9, 2), \text{and} \\ y_0 &= SB(9, 1). \end{aligned}$$

3.7. *SYM12* is a symmetric function of 12 variables that is 1 iff the number of 1's in the inputs is between 4 and 8. Consider the decomposition of the function $SYM12 = f(X_1, X_2)$, where $X_1 = (x_1, x_2, \ldots, x_9)$ and $X_2 = (x_{10}, x_{11}, x_{12})$. Show that the column multiplicity of the function with respect to (X_1, X_2) is $\mu_9 = 8$.

Chapter 4
MUX-Based Synthesis

This chapter shows a universal method to realize an n-variable function using multiplexers (MUXs) and look up tables (LUTs). It also derives upper bounds on the number of LUTs to realize an n-variable function. Such bounds are useful to estimate the number of LUTs needed to realize a given function when we only know the number of the input variables n.

4.1 Fundamentals of MUX

Definition 4.1.1. A **multiplexer** with a single control input (1-MUX) is the selection circuit shown in Fig. 4.1. It performs the logical operation

$$g(x, y_0, y_1) = \bar{x}y_0 \vee xy_1.$$

A ***t*-MUX** is shown in Fig. 4.2. It is a multiplexer with t control inputs $(x_1, \ldots, x_t)$ and 2^t data inputs $(y_0, y_1, \ldots, y_{2^t-1})$. Let $g(x_1, \ldots, x_t, y_0, y_1, \ldots, y_{2^t-1})$ be the output function. Then, $g = y_a$ when the decimal representation of the control input $(x_1, \ldots, x_t)$ is a. That is, when the control input is $(0, 0, \ldots, 0)$, the top data input y_0 drives the output. When the control input is $(0, 0, \ldots, 1)$, the second data input y_1 drives the output. Also, when the control input is $(1, 1, \ldots, 1)$, the last data input y_{2^t-1} drives the output.

An t-MUX can be realized using 1-MUXs.

Example 4.1.1. A 3-MUX is realized with $2^3 - 1 = 7$ modules of 1-MUXs, as shown in Fig. 4.3. ■

Lemma 4.1.1. *A t-MUX is realized by using $2^t - 1$ modules of 1-MUXs.*

Proof. This can easily be done by mathematical induction. □

When $K \geq 4$, a 2-MUX can be realized with one or two K-LUTs.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_4,

Fig. 4.1 1-MUX

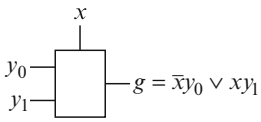


Fig. 4.2 t-MUX

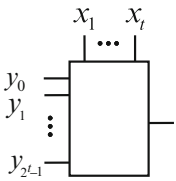
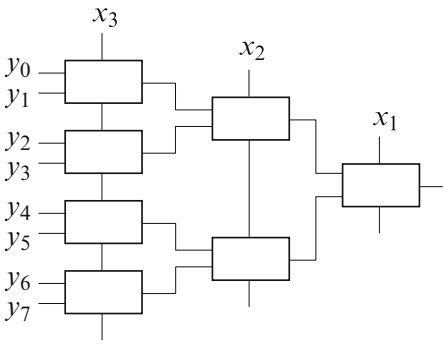


Fig. 4.3 3-MUX realized by 1-MUXs



Lemma 4.1.2. *[57, 73] A 2-MUX can be realized by two 4-LUTs as shown in Fig. 4.4, where*

$$g = (y_0\bar{x}_2 \vee y_1x_2)\bar{x}_1 \vee x_1x_2$$

and

$$h = g\bar{x}_1 \vee (y_2\bar{g} \vee y_3g)x_1.$$

Proof. When $x_1 = 0$, we have $g = y_0\bar{x}_2 \vee y_1x_2$ and $h = g$. Thus, $h = y_0\bar{x}_2 \vee y_1x_2$.
When $x_1 = 1$, we have $g = x_2$ and $h = y_2\bar{g} \vee y_3g$. Thus, $h = y_2\bar{x}_2 \vee y_3x_2$.
Therefore, the circuit realizes the 2-MUX function. □

Figure 4.4 shows a nondisjoint decomposition. A method to derive this decomposition is considered in Chap. 6.

When $K = 5$, a 2-MUX can be realized by two 5-LUTs as shown in Fig. 4.5.

When $K = 6$, a 2-MUX can be realized by a single 6-LUT instead of using three 1-MUXs as shown in Fig. 4.6.

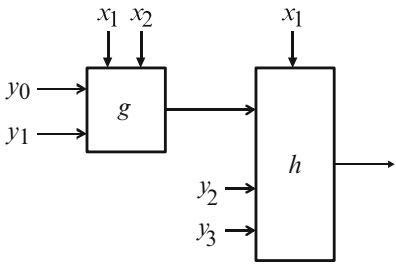


Fig. 4.4 2-MUX realized by 4-LUTs

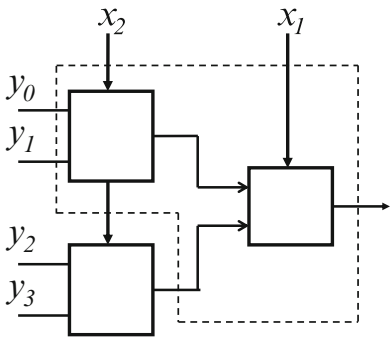


Fig. 4.5 2-MUX realized by 5-LUTs

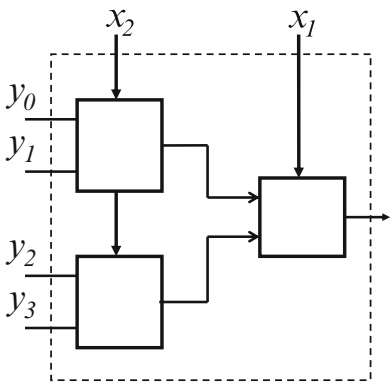


Fig. 4.6 2-MUX realized by a 6-LUT

4.2 MUX-Based Realization

Theorem 4.2.1. *An arbitrary n-variable function can be represented as follows:*

$$f(X_1, X_2) = \bigvee_{i \in P} g_i(X_1) X_2^i,$$

where $X_1 = (x_1, x_2, \ldots, x_k)$ *and* $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$, $P = \{0, 1, \ldots, 2^{n-k} - 1\}$, *and the OR is performed with respect to* 2^{n-k} *elements.*

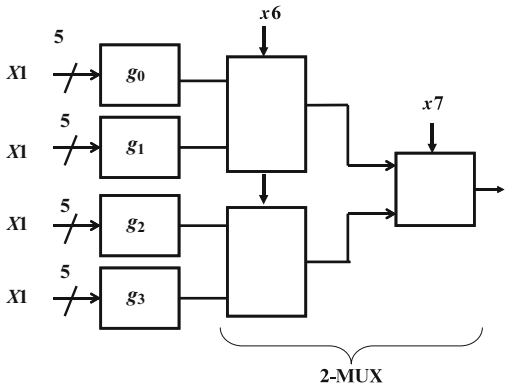


Fig. 4.7 Realization of an arbitrary 7-variable function using 5-LUTs

Example 4.2.1. Consider the realization of a 7-variable function $f(X_1, X_2)$, where $X_1 = (x_1, x_2, \ldots, x_5)$, and $X_2 = (x_6, x_7)$. f is expanded into a sum of four products:

$$f(X_1, X_2) = \bigvee_{i=0}^{3} g_i(X_1) X_2^i$$
$$= g_0(X_1)X_2^0 \vee g_1(X_1)X_2^1 \vee g_2(X_1)X_2^2 \vee g_3(X_1)X_2^3.$$

As shown in Fig. 4.7, a 2-MUX can be realized by using three 1-MUXs. The top LUT in the left most column realizes g_0, which is selected when $(x_6, x_7) = (0, 0)$. The second LUT in the leftmost column realizes g_1, which is selected when $(x_6, x_7) = (0, 1)$. Other LUTs are derived similarly. ■

Theorem 4.2.2. *When $3 \leq K \leq n$, an arbitrary n-variable function is realized by using at most $2^{n-K} - 1$ modules of 1-MUXs and 2^{n-K} modules of K-LUTs.*

Proof. Consider the expansion of Theorem 4.2.1. First, realize an $(n - K)$-MUX by using 1-MUXs. By Lemma 4.1.1, we need $2^{n-K} - 1$ modules of 1-MUXs. Next, by connecting $g_i(X_1)$ to the data inputs of the $(n - K)$-MUX, realize an arbitrary n-variable function. To realize $g_i(X_1)$ $(i = 0, 1, \ldots, 2^{n-K} - 1)$, we use 2^{n-k} modules of K-LUTs. □

Next, consider several special cases.

Lemma 4.2.1. *An arbitrary function of $n = K + 1$ variables can be realized with at most three K-LUTs, where $K \geq 3$.*

Proof. Let $X_1 = (x_1, x_2, \ldots, x_K)$ and $X_2 = (x_{K+1})$. Then, the function can be represented as

$$f(X_1, X_2) = \bar{x}_{K+1} f(X_1, 0) \vee x_{K+1} f(X_1, 1).$$

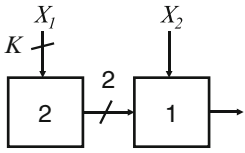


Fig. 4.8 Realization of a k+1 variable function

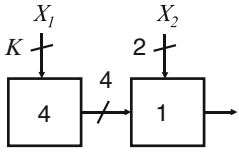


Fig. 4.9 Realization of a k+2 variable function

Thus, f can be realized by three K-LUTs as shown in Fig. 4.8. Note that the left cell realizes $f(X_1, 0)$ and $f(X_1, 1)$, while the right cell works as a selector (1-MUX) to realize f. The integer in a cell denotes the number of LUTs to realize the cell. This is an LUT cascade, which will be explained in the next chapter. □

Lemma 4.2.2. *An arbitrary function of $n = K + 2$ variables can be realized with at most five K-LUTs, where $K \geq 6$.*

Proof. Let $X_1 = (x_1, x_2, \ldots, x_K)$ and $X_2 = (x_{K+1}, x_{K+2})$. Then, the function can be represented as

$$f(X_1, X_2) = \bar{x}_{k+1}\bar{x}_{K+2} f(X_1, 0, 0) \vee \bar{x}_{K+1} x_{K+2} f(X_1, 0, 1) \vee x_{K+1}\bar{x}_{K+2} f(X_1, 1, 0) \vee x_{K+1} x_{K+2} f(X_1, 1, 1).$$

Thus, f can be realized by five K-LUTs, as shown in Fig. 4.9. Note that the left cell generates $f(X_1, 0, 0)$, $f(X_1, 0, 1)$, $f(X_1, 1, 0)$, and $f(X_1, 1, 1)$, while the right cell serves as a selector (2-MUX) to realize f. Figure 4.9 can be considered as a simplified representation of Fig. 4.7. □

Theorem 4.2.3. *[110] The number of 6-LUTs to realize an arbitrary n-variable function ($n \geq 6$) f is:*

- $(2^{n-4} - 1)/3$ *or less, when n is even.*
- $(2^{n-4} + 1)/3$ *or less, when n is odd.*

Proof. **Case 1: n is even** ($n = 2r$):
We realize the function f in the form of Theorem 4.2.1, where $K = 6$. First, realize a $(n - 6)$-MUX by using 2-MUXs. This requires

$$1 + 4 + \cdots + 4^{\frac{n-6}{2}-1} = 1 + 4 + \cdots + 4^{r-4} = \frac{4^{r-3} - 1}{4 - 1}$$

6-LUTs. Next, realize $g_i(X_1)$ $(i = 0, 1, \ldots, 2^{n-k} - 1)$. This requires 4^{r-3} modules of 6-LUTs. So, the total number of 6-LUTs is

$$\frac{4^{r-3} - 1}{3} + 4^{r-3} = \frac{4^{r-2} - 1}{3} = \frac{2^{n-4} - 1}{3}.$$

Case 2: n is odd $(n = 2r + 1)$:
The function f can be expanded into the form

$$f(X_1, x_n) = \bar{x}_n g_0(X_1) \vee x_n g_1(X_1), \tag{4.1}$$

where $X_1 = (x_1, x_2, \ldots, x_{n-1})$. Since $g_i(X_1)$ $(i = 0, 1)$ are functions with $2r$ variables, they can be realized by $(4^{r-2} - 1)/3$ modules of 6-LUTs. To realize the expansion (4.1), we use a 1-MUX. Thus, the total number of 6-LUTs to realize f is

$$2 \cdot \frac{4^{r-2} - 1}{3} + 1 = \frac{2 \cdot 4^{r-2} + 1}{3} = \frac{2^{n-4} + 1}{3}.$$

□

Example 4.2.2. The number of 6-LUTs to realize an n-variable function is:

- 5 or less, when $n = 8$. In this case, $g_i(X_1)$ $(i = 0, 1, 2, 3)$ are realized by four modules of 6-LUTs, while the 2-MUX is realized by a single 6-LUT, as shown in Fig. 4.10. In the figure, the numbers in the squares denote the numbers of necessary LUTs.
- 11 or less, when $n = 9$. In this case, $g_i(X_1)$ $(i = 0, 1, 2, \ldots, 7)$ are realized by 8 modules of 6-LUTs, while the 3-MUX is realized by three 6-LUTs as shown in Fig. 4.11.
- 21 or less, when $n = 10$. In this case, $g_i(X_1)$ $(i = 0, 1, 2, \ldots, 15)$ are realized by 16 modules of 6-LUTs, while the 4-MUX is realized by using five 6-LUTs as shown in Fig. 4.12. ■

Theorem 4.2.4. *Consider the function $f(X_1, X_2)$, where $X_1 = (x_1, x_2, x_3, x_4)$ and $X_2 = (x_5, x_6, \ldots, x_{K+3}, x_{K+4})$. Let μ be the column multiplicity of the decomposition $f(X_1, X_2)$, where X_1 denotes the bound variables. Then, f can be realized with at most $\mu + 5$ modules of K-LUTs, where $K \geq 6$.*

Proof. The function $f(X_1, X_2)$ can be expanded as

$$f(X_1, X_2) = g_0(\vec{a}_0, X_2) \vee g_1(\vec{a}_1, X_2) \vee g_2(\vec{a}_2, X_2) \vee \cdots \vee g_{15}(\vec{a}_{15}, X_2),$$

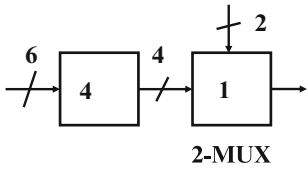


Fig. 4.10 Realization of an arbitrary 8-variable function using 6-LUTs

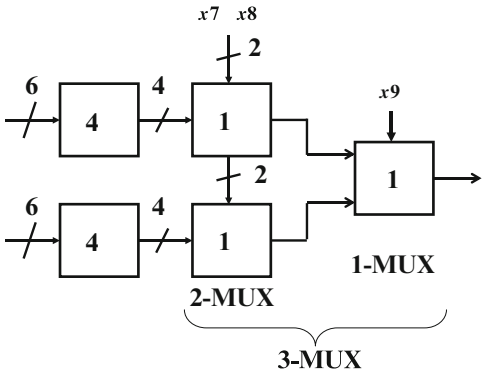


Fig. 4.11 Realization of an arbitrary 9-variable function using 6-LUTs

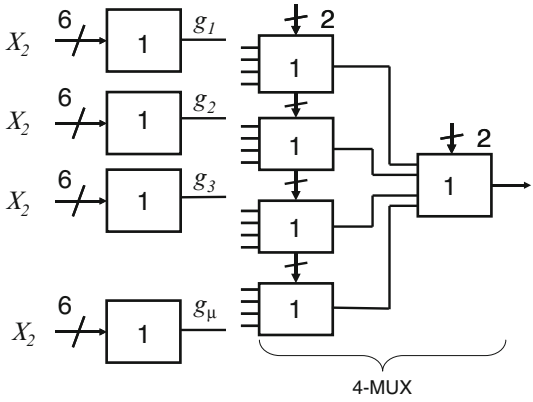


Fig. 4.12 Realization of an arbitrary 10-variable function using 6-LUTs

where $\vec{a}_0 = (0, 0, 0, 0)$, $\vec{a}_1 = (0, 0, 0, 1)$, $\vec{a}_2 = (0, 0, 1, 0)$,..., and $\vec{a}_{15} = (1, 1, 1, 1)$. Thus, $f(X_1, X_2)$ can be realized as the circuit shown in Fig. 4.12. Since the column multiplicity is μ, the number of different column functions $g_i(\vec{a}_i, X_2)$ is μ. So, in Fig. 4.12, the LUTs producing the same functions can be shared, and only μ LUTs are sufficient to produce $g_i(\vec{a}_i, X_2)$. □

4.3 Remarks

This chapter derived the number of 6-LUTs to realize an n-variable logic function. The number of required LUTs increases exponentially with n. So, the MUX-based design is only practical for the functions with a small number of inputs. However, the MUX-based method sometimes produces circuits with fewer LUTs [134] than existing methods [73, 76], in particular, for random logic functions. In this chapter, we represented the function by Shannon expansion. However, if we use

pseudo-Kronecker expansion [110], we can reduce the number of LUTs by 23% [107]. In the pseudo-Kronecker expansion, we can select one from 840 possible expansions to reduce the number of LUTs.

Problems

4.1. Consider a 10-variable function $f(X)$. Let (X_1, X_2) be a partition of the variables X, where $X_1 = (x_1, x_2, x_3, x_4, x_5, x_6)$ and $X_2 = (x_7, x_8, x_9, x_{10})$. Assume that $\mu_6 = 16$. That is, the column multiplicity of the decomposition chart is 16. Show that f can be realized with at most nine 6-LUTs.

4.2. Let (X_1, X_2) be the partition of the variables X, where $X_1 = (x_1, x_2, x_3, x_4)$ and $X_2 = (x_5, x_6, x_7, x_8, x_9, x_{10})$. Suppose that the column multiplicity is 10, i.e., $\mu_4 = 10$. Then, show that f can be realized with at most 15 copies of 6-LUTs.

4.3. Show that an arbitrary logic function can be represented as

$$f(x_1, x_2, Y) = g_0(Y) \oplus x_1 g_1(Y) \oplus x_2 g_2(Y) \oplus x_1 x_2 g_3(Y). \tag{4.2}$$

This is the **Reed–Muller expansion**. Consider the Shannon expansion:

$$f(x_1, x_2, Y) = \bar{x}_1 \bar{x}_2 f_0(Y) \oplus \bar{x}_1 x_2 f_1(Y) \oplus x_1 \bar{x}_2 f_2(Y) \oplus x_1 x_2 g_3(Y). \tag{4.3}$$

Represent $g_0(Y)$, $g_1(Y)$, $g_2(Y)$, and $g_3(Y)$ by $f_0(Y)$, $f_1(Y)$, $f_2(Y)$, and $f_3(Y)$.

4.4. Consider the function $f(X_1, X_2)$, where $X_1 = (x_1, x_2, \ldots, x_{2k})$ and $X_2 = (x_{2k+1}, x_{2k+2}, \ldots, x_{2k+6})$. Let μ be the column multiplicity of the decomposition $f(X_1, X_2)$, where X_1 denotes the bound variables. Then, show that f can be realized with at most

$$\mu + \frac{4^k - 1}{3}$$

6-LUTs.

Chapter 5
Cascade-Based Synthesis

The previous chapter presented a multiplexer (MUX)-based realization. Although such a method is applicable to any n-variable function f, the number of LUTs necessary to realize f increases as $O(2^n)$. This chapter considers a cascade-based logic synthesis. A cascade-based realization is applicable to only a limited class of functions. However, functions with a small C-measure can be realized by cascade-based realizations with $O(n)$ LUTs.

5.1 Functional Decomposition and LUT Cascade

Before considering the general case, we review special cases.

Lemma 5.1.1. *An arbitrary function of $n = K + 1$ variables can be realized with at most three K-LUTs, where $K \geq 3$.*

Proof. This is the same as Lemma 4.2.1. □

Lemma 5.1.2. *An arbitrary function of $n = K + 2$ variables can be realized with at most five K-LUTs, where $K \geq 6$.*

Proof. This is the same as Lemma 4.2.2. □

From the definition of a decomposition chart, we have the following:

Theorem 5.1.1. *[8] Let $\mu_k(n)$ be the column multiplicity of a decomposition chart of an n-variable logic function with k bound variables. Then,*

$$\mu_k(n) \leq \min\left\{2^k, 2^{2^{n-k}}\right\}.$$

When circuits are designed by LUTs, functions with smaller column multiplicities tend to have smaller realizations.

Definition 5.1.1. Let $f(x_1, x_2, \ldots, x_n)$ be a logic function. The **profile** of the function f is the vector $(\mu_1, \mu_2, \ldots, \mu_n)$, where μ_k denotes the column multiplicity of the decomposition chart for $f(X_1, X_2)$, $X_1 = (x_1, x_2, \ldots, x_k)$ and

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_5,

$X_2 = (x_{k+1}, \ldots, x_n)$, assuming that the order of variables $(x_1, x_2, \ldots, x_n)$ is fixed. The **C-measure** of the function f is $\max(\mu_1, \mu_2, \ldots, \mu_n)$ and is denoted by $\mu(f)$.

Note that the order of the variables will affect the C-measure, but we choose the **natural order** $(x_1, x_2, \ldots, x_n)$ of the input variables.

Lemma 5.1.3. *Let f be an arbitrary n-variable function. Then,*

$$\mu(f) \leq \max_{k=1}^{n} \min \left\{ 2^k, 2^{2^{n-k}} \right\}.$$

For any partition (X_1, X_2) of X, we have the decomposition shown in Fig. 5.1. By repeatedly applying functional decompositions to a given function $f(X) = f(X_1, X_2, \ldots, X_s)$, we have an **LUT cascade** [113] shown in Fig. 5.2. An LUT cascade consists of **cells**. The signal lines connecting adjacent cell are **rails**. A logic function with a small C-measure can be realized by a compact LUT cascade.

Lemma 5.1.4. *[113] An arbitrary logic function f can be realized by an LUT cascade, whose cells have at most $\lceil \log_2 \mu(f) \rceil + 1$ inputs, and at most $\lceil \log_2 \mu(f) \rceil$ outputs, where $\mu(f)$ is the C-measure of f.*

Lemma 5.1.5. *[127] In an LUT cascade that realizes an n-variable function f, let s be the number of cells; $w = \lceil \log_2 \mu(f) \rceil$ be the maximum number of rails; K be the number of inputs to a cell; $n \geq K + 1$; and $K \geq \lceil \log_2 \mu(f) \rceil + 1$. Then, an LUT cascade satisfying the following condition exists:*

$$s = \left\lceil \frac{n - w}{K - w} \right\rceil. \tag{5.1}$$

Proof. From the design method of the LUT cascade, we have

$$K + (K - w)(s - 1) \geq n.$$

Here, K on the left-hand side of the equality denotes the number of inputs of the leftmost LUT, and $(K-w)(s-1)$ denotes the sum of inputs for the remaining $(s-1)$

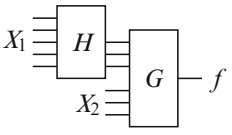


Fig. 5.1 Realization of a logic function by decomposition

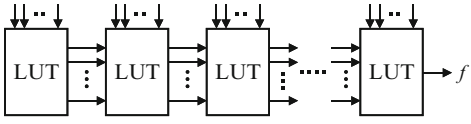


Fig. 5.2 LUT cascade

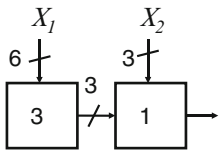


Fig. 5.3 Realization of a 9-variable function with C-measure not exceeding 8

LUTs. When the actual number of rails is smaller than w, we append dummy rails to make the number of rails w. From this, we have

$$s - 1 \geq \frac{n-K}{K-w}, \text{ and } s \geq \frac{n-w}{K-w}.$$

Since s is an integer, we have (5.1). When this is the case, we can realize an LUT cascade for f having s cells with at most K inputs. □

Example 5.1.1. Let $f(X_1, X_2)$ be a 9-variable function, where $X_1 = (x_1, x_2, \ldots, x_6)$ and $X_2 = (x_7, x_8, x_9)$. Let μ_6 be the column multiplicity of the decomposition of f with respect to (X_1, X_2). If $\mu_6 \leq 8$, then $f(X_1, X_2)$ can be realized with four 6-LUTs, as shown in Fig. 5.3. Note that the number of rails between two cells is $\lceil \log_2 8 \rceil = 3$, by Theorem 3.3.1. In many cases, the natural ordering of the input variables does *not* yield the smallest circuit. To check if $\mu_6 \leq 8$ for all arrangements of variables, we need to compute the column multiplicities for $\binom{9}{6} = 84$ combinations of variables. ■

Lemma 5.1.6. *Consider a cascade consisting of K-LUTs.*

1. *When the number of the external input variables to the output LUT is one, the number of the rail inputs to the LUT is at most two.*
2. *When the number of the external input variables to the output LUT is two, the number of the rail inputs to the LUT is at most four.*

Proof. We prove the second case only. The proof for the first case is similar. Let x_{n-1} and x_n be external input variables for the output LUT. Consider the decomposition chart, where the rail inputs $X_1 = (x_1, x_2, \ldots, x_{n-2})$ denotes the set of bound variables and the external input $X_2 = (x_{n-1}, x_n)$ denotes the set of free variables. In this case, the column multiplicity is at most 16, since there exist at most $2^{2^2} = 16$ different column functions by Theorem 5.1.1. Also by Theorem 3.3.1, the number of the rail inputs to the output LUT is at most four. □

5.2 Number of LUTs to Realize General Functions

As for realizations by 6-LUTs, we have the following:

Theorem 5.2.1. *The number of 6-LUTs needed to realize an arbitrary n-variable function with $\mu(f) \leq 32$ is $5n - 35$ or less, where $n \geq 8$.*

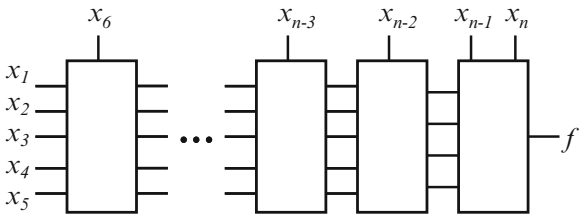


Fig. 5.4 Realization with 6-LUTs

Proof. From Lemma 5.1.4, an arbitrary function with $\mu(f) \leq 32$ can be realized by an LUT cascade, whose cells have at most $\lceil \log_2 \mu \rceil + 1 = \log_2(32) + 1 = 6$ inputs, and $w = \log_2(32) = 5$ outputs. Let $K = 6$. Then, from Lemma 5.1.5, the number of cells is at most $\lceil \frac{n-w}{K-w} \rceil = \frac{n-5}{6-5} = n - 5$. Note that each cell except for the rightmost cell has at most 5 outputs. From Lemma 5.1.6, the second cell from the right has at most four outputs as shown in Fig. 5.4. So, the total number of cells is at most $n - 6$. Note that the second cell from the right has 4 outputs, while the rightmost cell has just one output. Therefore, the total number of LUTs is at most $5(n - 8) + 4 + 1 = 5n - 35$. □

Theorem 5.2.2. *The number of 6-LUTs needed to realize an arbitrary n-variable function with $\mu(f) \leq 16$ is $2n - 11$ or less, where $n \geq 8$.*

Proof. Let $w = \log_2(16) = 4$ and $K = 6$. From Lemma 5.1.5, we have $s \leq \lceil \frac{n-w}{K-w} \rceil = \frac{n-4}{6-4} = \lceil \frac{n-4}{2} \rceil$.

When $n = 2r$, each cell except for the rightmost cell has at most 4 outputs. So, the total number of LUTs is at most $4 \times \left(\lceil \frac{n-4}{2} \rceil - 1 \right) + 1 = 2(n-4) - 4 + 1 = 2n - 11$.

When $n = 2r + 1$, by Lemma 5.1.6, the rightmost cell has one external input and at most two rail inputs, and the second cell from the right has at most two outputs. So, the total number of LUTs is at most $4 \times \left(\lceil \frac{n-5}{2} \rceil - 1 \right) + 2 + 1 = 2n - 11$. □

Theorem 5.2.3. *The number of 6-LUTs needed to realize an arbitrary n-variable function with $\mu(f) \leq 8$ is $n - 5$ or less when $n = 3r$, and $n - 4$ or less when $n \neq 3r$, where $n \geq 8$.*

Proof. Let $w = \log_2 8 = 3$ and $K = 6$. From Lemma 5.1.5, we have $s \leq \lceil \frac{n-w}{K-w} \rceil = \lceil \frac{n-3}{3} \rceil$.

When $n = 3r$, each cell except for the rightmost cell has at most 3 outputs. So, the total number of LUTs is at most $3 \times \left(\lceil \frac{n-3}{3} \rceil - 1 \right) + 1 = (n-3) - 3 + 1 = n - 5$.

When $n = 3r + 1$, the rightmost cell has one external input and at most two rail inputs, and the second cell from the rightmost one has at most 2 outputs. So, the total number of LUTs is at most $3 \times \left(\lceil \frac{n-4}{3} \rceil - 1 \right) + 2 + 1 = (n-4) - 3 + 3 = n - 4$.

When $n = 3r + 2$, the rightmost cell has two external inputs. So, the total number of LUTs is at most $3 \times \left(\lceil \frac{n-5}{3} \rceil - 1 \right) + 3 + 1 = (n-5) - 3 + 4 = n - 4$. □

Lemma 5.2.1. *The number of K-LUTs needed to realize a $(K + 1)$-variable k-output function F with $\mu(F) \leq 2^{K-1} + 2^{K-3}$ is $3K$ or less.*

Proof. Consider the circuit shown in Fig. 5.5. Since the column multiplicity is at most $2^{K-1} + 2^{K-3}$, each pattern can be uniquely represented by a K-bit code. By using the leftmost cell D, generate K-bit codes that correspond to the column patterns. For the first 2^{K-1} patterns, assign K-bit codes with the form $(0, *, *, \ldots, *)$, where * denotes either 0 or 1. For the remaining 2^{K-3} patterns, assign K-bit codes with the form $(1, 0, 0, *, *, \ldots, *)$. Cell A in Fig. 5.5 implements the function for the codes $(0, *, *, \ldots, *)$, while cell B implements the function for the codes $(1, 0, 0, *, \ldots, *)$. Cell C is used for a selector, which is controlled by the most significant bit of the outputs of cell D. In this way, an arbitrary K output function can be realized by the circuit shown in Fig. 5.5. Note that cell B has $K - 2$ inputs, and cell C has three inputs. Cells B and C can be merged and realized by a K-input LUT. Thus, the total number of LUTs to implement this circuit is at most $3K$. □

Theorem 5.2.4. *Let $K \geq 6$ and $n \geq K + 3$. The number of K-LUTs to realize an n-variable function f with $\mu(f) \leq 2^{K-1} + 2^{K-3}$ is $2K(n - K) - 5K + 9$ or less.*

Proof. Consider the cascade shown in Fig. 5.6. Since $\lceil \log_2 \mu \rceil \leq K$, the function can be realized by a cascade with at most K-rails. Also, by Lemma 5.2.1, each of the intermediate cells can be realized with at most $2K$ LUTs. By Lemma 5.1.6, the rightmost cell has at most four rail inputs and two external inputs. Also, note that the second cell from the right can be implemented with at most 8 LUTs. Note that

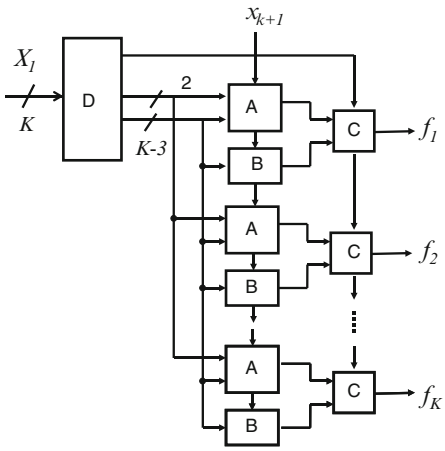


Fig. 5.5 Realization of $(K + 1)$-variable K-output function

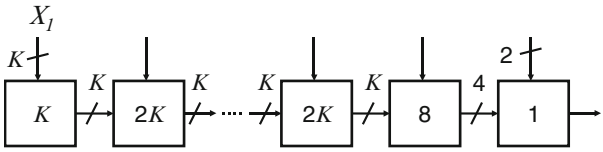


Fig. 5.6 Cascade realization in the proof

the number of outputs is at most four, and the number of inputs is $K + 1$. We can prove that 8 LUTs is sufficient in a similar way to the proof of Lemma 5.2.1. The leftmost cell can be implemented with K LUTs. Thus, the total number of LUTs is $K + 2K(n - K - 3) + 9 = 2K(n - K) - 5K + 9$. □

Theorem 5.2.5.

- *The number of 6-LUTs needed to realize an arbitrary n-variable function with* $\mu(f) \leq 40$ *is* $12n - 93$ *or less, where* $n \geq 9$.
- *The number of 7-LUTs needed to realize an arbitrary n-variable function with* $\mu(f) \leq 80$ *is* $14n - 124$ *or less, where* $n \geq 10$.
- *The number of 8-LUTs needed to realize an arbitrary n-variable function with* $\mu(f) \leq 160$ *is* $16n - 159$ *or less, where* $n \geq 11$.

5.3 Number of LUTs to Realize Symmetric Functions

When the given function is symmetric, it can be realized more efficiently than a general function [106,112]. Efficient algorithms to detect symmetric functions exist, e.g., [70,91].

Lemma 5.3.1. *Let* f *be a symmetric function of n-variables. Then,* $\mu(f) \leq n + 1$.

Proof. Consider the partition of variables (X_1, X_2), where $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$. Let μ_k be the column multiplicity of the decomposition. Since f is symmetric, the column labels with the same weights have the same column patterns in the decomposition chart. Thus, the column multiplicity is at most $k + 1$. From this, we have the lemma. □

Example 5.3.1. Consider a symmetric function $f(X_1, X_2)$, where $X_1 = (x_1, x_2, x_3)$ and $X_2 = (x_4, x_5, x_6)$. This is an example for $k = 3$. In this case, the number of columns is $2^3 = 8$. The column label with weight 0 is $(0, 0, 0)$. The column labels with weight 1 are $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. They have the same column functions. The column labels with weight 2 are $(1, 1, 0)$, $(1, 0, 1)$, and $(0, 1, 1)$. They have the same column functions. And the column label with weight 3 is $(1, 1, 1)$. Thus, the number of different column patterns is at most four. ■

In Chap. 7, tighter bounds are derived.

Theorem 5.3.1. *The number of K-LUTs needed to realize an n-variable symmetric function is:*

- *4 or less, when* $n = 9$ *and* $K = 6$. *Figure 5.3 shows the realization.*
- *7 or less, when* $n = 12$ *and* $K = 6$. *Figure 5.7 shows the realization.*
- *13 or less, when* $n = 15$ *and* $K = 6$. *Figure 5.8 shows the realization.*

Example 5.3.2. Consider *SYM12* [110], a symmetric function of 12 variables. *SYM12* is 1 iff the number of 1's in the inputs is between 4 and 8.

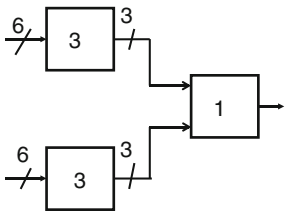


Fig. 5.7 Realization of a symmetric function of 12 variables by 6-LUTs

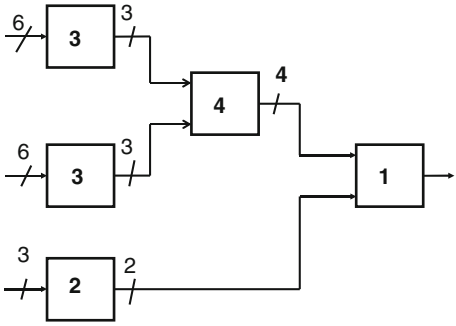


Fig. 5.8 Realization of a symmetric function of 15 variables by 6-LUTs

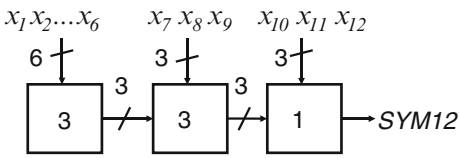


Fig. 5.9 SYM12 realized by 6-LUTs

1. The column multiplicity of the function with respect to the bound set composed of $K = 6$ variables is $\mu_6 = 7$. Thus, the number of rails between two blocks is $\lceil \log_2 7 \rceil = 3$.
2. We decompose the function with respect to the bound set: $x_1, x_2, \ldots, x_6$. Realize the first cell using 6-LUTs, which corresponds to the leftmost cell in Fig. 5.9.
3. The remaining function has $12 - 6 + 3 = 9$ variables. The bound variables for the second decomposition are three outputs of the leftmost cell, and x_7, x_8, x_9. In this case, the column multiplicity is $\mu_9 = 8$ (Problem 3.8). Thus, the number of rails between two blocks is $\lceil log_2 8 \rceil = 3$.
4. We decompose the remaining function with the bound set: three outputs of the leftmost cell, and x_7, x_8, x_9. Realize the second cell using 6-LUTs, which corresponds to the middle cell in Fig. 5.9.
5. The remaining function has $9 - 6 + 3 = 6$ variables. Since the number of remaining variables is equal to $K = 6$, realize the function by a 6-LUT, which corresponds to the rightmost cell in Fig. 5.9.
6. In this way, *SYM12* is realized by $3 + 3 + 1 = 7$ LUTs of 6-inputs. Note that this realization is different from that shown in Fig. 5.7. ■

5.4 Remarks

In this chapter, we showed a method to realize a given function by using a cascade of LUTs. This method is only applicable to the functions whose C-measures are small.

The C-measure of a logic function f is related to the size of its BDD. Sizes of BDDs for various classes of functions are considered in [154]. Classes of functions having small C-measures are considered in Chap. 7.

LSIs for LUT cascades have been fabricated [86–88]. This chapter is based on [134].

Problems

5.1. Compare the tree-type realization in Fig. 5.7 with the cascade realization in Fig. 5.9. Discuss their advantages and disadvantages.

5.2. Consider the 4-bit adder, where x_3, x_2, x_1, x_0 and y_3, y_2, y_1, y_0 denote the inputs, and z_4, z_3, z_2, z_1, z_0 denote the outputs. Design the adder using an LUT cascade. Use 6-LUTs. Show the expression of the output function for each LUT.

5.3. Design a 12-input 4-output circuit that counts the number of 1's in the inputs and represents this by a binary number (i.e., WGT12) by 6-LUTs.

5.4. Consider a set of three functions f_i with 7 variables. Assume that $\mu(f_i) \leq 40$ for $i = 1, 2, 3$. Realize these functions by 6-LUTs, using the design method shown in the proof of Lemma 5.2.1.

5.5. Let $\mu_k(n)$ be the column multiplicity of a decomposition chart of an n-variable function with k bound variables. Show the following relations:

$$\mu_{k+1}(n) \leq 2\mu_k(n)$$

$$\mu_{k-1}(n) \leq \mu_k^2(n)$$

5.6. Enumerate the 8-variable functions whose C-measures are 32.

Chapter 6
Encoding Method

This chapter shows a method to reduce the number of LUTs needed to realize logic functions with nonstandard encodings. In these encodings, intermediate variables in functional decomposition are represented with fewer variables. This method offers a way to find a nondisjoint decomposition.

6.1 Decomposition and Equivalence Class

Definition 6.1.1. Let $f(X_1, X_2)$ be a logic function and (X_1, X_2) be a partition of X. $|X_1|$ denotes the number of variables in X_1. Let $B = \{0, 1\}$. When $n_1 = |X_1|$ and $n_2 = |X_2|$, an equivalence relation $\sim$ on B^{n_1} is defined as follows: $\vec{a} \sim \vec{b} \iff f(\vec{a}, X_2) = f(\vec{b}, X_2)$, where $\vec{a}, \vec{b} \in B^{n_1}$. Let the equivalence classes of B^{n_1} be $\Psi_0, \Psi_1, \ldots, \Psi_{\mu-1}$. In this case, μ is equal to the column multiplicity in the decomposition chart of f with the partition (X_1, X_2). Ψ_i is also used to represent the corresponding logic function.

Example 6.1.1. Consider the function

$$f(X_1, X_2) = y_0\bar{x}_1\bar{x}_2 \vee y_1\bar{x}_1x_2 \vee y_2x_1\bar{x}_2 \vee y_3x_1x_2,$$

where $X_1 = (x_1, x_2, y_0, y_1)$ and $X_2 = (y_2, y_3)$. The decomposition chart is shown in Fig. 6.1. The logic functions for the various equivalence classes are

$$\begin{aligned}
\Psi_0 &= \bar{x}_1\bar{x}_2\bar{y}_0 \vee \bar{x}_1x_2\bar{y}_1 = \bar{x}_1(\bar{x}_2\bar{y}_0 \vee x_2\bar{y}_1),\\
\Psi_1 &= x_1x_2,\\
\Psi_2 &= x_1\bar{x}_2, \text{ and}\\
\Psi_3 &= \bar{x}_1\bar{x}_2y_0 \vee \bar{x}_1x_2y_1 = \bar{x}_1(\bar{x}_2y_0 \vee x_2y_1).
\end{aligned}$$

Note that Ψ_0 denotes the logic function for the equivalence class of the column vector $(0, 0, 0, 0)^t$, where the symbol t denotes the transpose of the vector. Similarly, Ψ_1 corresponds to $(0, 1, 0, 1)^t$, Ψ_2 corresponds to $(0, 0, 1, 1)^t$, and Ψ_3 corresponds to $(1, 1, 1, 1)^t$. ■

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_6,

		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_1
		0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_2
		0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	y_0
		0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	y_1
0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	
0	1	0	0	1	1	0	1	0	1	0	0	0	0	1	1	1	1	
1	0	0	0	1	1	0	1	0	1	1	1	1	1	0	0	0	0	
1	1	0	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1	
y_2	y_3																	

Fig. 6.1 Example of a decomposition chart

6.2 Disjoint Encoding

Definition 6.2.1. In a functional decomposition, the **minimum length encoding** uses $\lceil \log_2 \mu \rceil$ bits to encode equivalence classes: $\Psi_0, \Psi_1, \ldots, \Psi_{\mu-1}$, where $\lceil a \rceil$ denotes the minimum integer greater than a.

Since Ψ_i $(i = 0, 1, \ldots, \mu - 1)$ represents equivalence classes, Ψ_i has the following properties: $\Psi_i \cdot \Psi_j = 0$ $(i \neq j)$ and $\vee_{i=0}^{\mu-1} \Psi_i = 1$.

Let the intermediate variables be $h_1, h_2, \ldots, h_u$, where $u = \lceil \log_2 \mu \rceil$. Suppose that $(h_u, h_{u-1}, \ldots, h_2, h_1)$ denotes the index showing the equivalence class. In this encoding, all the vectors in an equivalence class are assigned the same codes. Such an encoding is a **disjoint encoding**.[1]

Example 6.2.1. Consider the function in Example 6.1.1. Note that $\mu = 4$. When we use disjoint encoding:

Ψ_0 is coded by 00,
Ψ_1 is coded by 01,
Ψ_2 is coded by 10, and
Ψ_3 is coded by 11.

In this case, we have

$$h_1 = \Psi_1 \vee \Psi_3 = x_1 x_2 \vee \bar{x}_1(\bar{x}_2 y_0 \vee x_2 y_1), \text{ and}$$
$$h_2 = \Psi_2 \vee \Psi_3 = x_1 x_2 \vee \bar{x}_1(\bar{x}_2 y_0 \vee x_2 y_1).$$

Note that

$$\bar{h}_2 \bar{h}_1 = \Psi_0,$$
$$\bar{h}_2 h_1 = \Psi_1,$$
$$h_2 \bar{h}_1 = \Psi_2, \text{ and}$$
$$h_2 h_1 = \Psi_3.$$

[1] In the previous publications, disjoint encoding was called strict encoding [44, 112].

However, even if x_1 is realized instead of the function $h_2 = \Psi_2 \vee \Psi_3$, we can still represent the equivalence class as follows:

$$\bar{x}_1\bar{h}_1 = \bar{x}_1(\bar{x}_2\bar{y}_0 \vee x_2\bar{y}_1) = \Psi_0,$$
$$\bar{x}_1 h_1 = \bar{x}_1(\bar{x}_2 y_0 \vee x_2 y_1) = \Psi_3,$$
$$x_1\bar{h}_1 = x_1\bar{x}_2 = \Psi_2, \text{ and}$$
$$x_1 h_1 = x_1 x_2 = \Psi_1.$$

■

The above example shows that an appropriate encoding can simplify intermediate variables. The next section shows a systematic method to simplify intermediate variables. Here, we assume that a function is simpler if it can be represented with fewer variables.

6.3 Nondisjoint Encoding

In a disjoint encoding, all the vectors in an equivalence class are assigned to the same code. However, in general, we can use the code where the vectors in the same equivalence class may be assigned to different codes, as long as the vectors in the different classes are assigned to different codes. Such an encoding is a **nondisjoint encoding**. This often simplifies intermediate variables.

Various methods exist to encode equivalence classes: $\Psi_0, \Psi_1, \ldots,$ and $\Psi_{\mu-1}$. In this chapter, we use the encoding that simplifies the intermediate variable h_u. If we can design an encoding such that $h_u(X_1) = x_i$, then the LUT for h_u is not needed, since x_i is available as an input variable.

Example 6.3.1. Consider a 7-variable function $f(X_1, X_2)$, where $X_1 = (x_1, x_2, x_3, x_4)$ and $X_2 = (x_5, x_6, x_7)$. Assume that f is partially symmetric with respect to X_1. In this case, the column multiplicity μ of the decomposition chart for f (X_1, X_2) is at most 5, since it is sufficient to identify if $0, 1, 2, 3,$ and 4 of $x_1, x_2, x_3,$ and x_4 are 1. Since $\mu \leq 5$ and $\lceil \log_2 \mu \rceil \leq 3$, f can be realized as shown in Fig. 6.2.

Assume $\mu = 5$. In this case, the circuit requires three intermediate variables: h_1, h_2, and h_3.

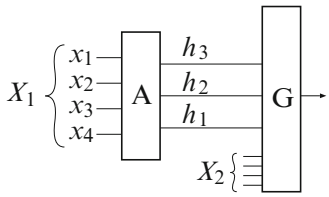


Fig. 6.2 Realization using disjoint encoding

Table 6.1 Disjoint encoding

h_3	h_2	h_1	Number of 1's in X_1
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4

Disjoint Encoding:

Table 6.1 shows the encoding for h_1, h_2, and h_3. Note that (h_3, h_2, h_1) shows the number of 1's in the inputs. In this case, we have

$$\begin{aligned} h_3 &= SB(4,4) = x_1x_2x_3x_4, \\ h_2 &= SB(4,2) = x_1x_2 \oplus x_1x_3 \oplus x_1x_4 \oplus x_2x_3 \oplus x_2x_4 \oplus x_3x_4, \text{ and} \\ h_1 &= SB(4,1) = x_1 \oplus x_2 \oplus x_3 \oplus x_4. \end{aligned}$$

Note that Table 6.1 represents WGT4, which is realized by the block A in Fig. 6.2. Since Table 6.1 uses five code words in disjoint encoding, the network for A requires three 4-LUTs.

The Encoding that Simplifies an Intermediate Variable:

For h_3, we realize x_1 instead of $SB(4,4)$:

$$\begin{aligned} h_3 &= x_1, \\ h_2 &= SB(4,2), \text{ and} \\ h_1 &= SB(4,1). \end{aligned}$$

In this case, (h_3, h_2, h_1) shows the number of 1's in X_1, where

$$\begin{aligned} (h_3, h_2, h_1) &= (0,0,0) \;\; X_1 \text{ has no 1}, \\ (h_3, h_2, h_1) &= (-,0,1) \;\; X_1 \text{ has one 1}, \\ (h_3, h_2, h_1) &= (-,1,0) \;\; X_1 \text{ has two 1's}, \\ (h_3, h_2, h_1) &= (-,1,1) \;\; X_1 \text{ has three 1's, and} \\ (h_3, h_2, h_1) &= (1,0,0) \;\; X_1 \text{ has four 1's}. \end{aligned}$$

Since the function $h_3 = x_1$ is available as an input variable, no LUT is necessary for h_3, as shown in Fig. 6.3. The network for A' realizes a 4-input 2-output function and can be implemented by only two 4-LUTs. This encoding uses eight different code words. ■

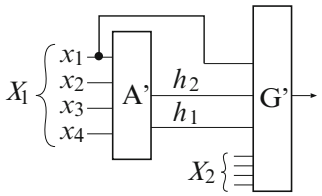


Fig. 6.3 Realization using encoding that simplifies h

Note that the nondisjoint encoding shown in the above example converted a disjoint decomposition (Fig. 6.2) into a **nondisjoint decomposition** (Fig. 6.3). Thus, a disjoint encoding corresponds to a disjoint decomposition, while a nondisjoint encoding corresponds to a nondisjoint decomposition. However, not every nondisjoint encoding leads to a nondisjoint decomposition.

Theorem 6.3.1. *[58] Consider the decomposition*

$$f(X_1, X_2) = g(h_1(X_1), h_2(X_1), \ldots, h_u(X_1), X_2).$$

Let $\Psi_i(X_1), (i = 0, 1, \ldots, \mu - 1)$ *be the equivalence classes of the decomposition. Let* $x_j \in X_1$. *If the number of different nonzero functions* $\bar{x}_j \Psi_i$, $(i = 0, 1, 2, \ldots, \mu - 1)$ *is equal to or less than* 2^{u-1}, *and the number of different nonzero functions* $x_j \Psi_i$, $(i = 0, 1, 2, \ldots, \mu - 1)$ *is equal to or less than* 2^{u-1}, *then* $h_u(X_1)$ *can be represented as* $h_u(X_1) = x_j$.

Proof. Algorithm 6.3.1 shows the method to simplify intermediate variables. □

Algorithm 6.3.1. *(Simplification of an Intermediate Variable)*

1. *Let* Ψ_i $(i = 0, 1, \ldots, \mu - 1)$ *and* $x_j \in X_1$ *satisfy the condition of Theorem 6.3.1. To* $\bar{x}_j \Psi_i \neq 0$, *assign code v from 0 to* $2^{u-1} - 1$.
2. *To* $x_j \Psi_i \neq 0$, *if the function* $\bar{x}_j \Psi_i$ *is already assigned a code v in the previous step, assign a code* $v + 2^{u-1}$.
3. *If there exists a function* $x_j \Psi_i$ *which has not been assigned a code yet, assign an unused code t, where* $2^{u-1} \leq t < 2^u$.

We use examples to show the algorithm.

Example 6.3.2. Consider the decomposition $f(X_1, X_2)$, where $X_1 = (x_1, x_2, x_3)$. Let the equivalence classes for the decomposition be

$$\Psi_0 = \bar{x}_1 \bar{x}_2 \bar{x}_3,$$

$$\Psi_1 = \bar{x}_1 x_2 \vee \bar{x}_2 x_3 \vee \bar{x}_3 x_1, \text{and}$$

$$\Psi_2 = x_1 x_2 x_3.$$

Disjoint Encoding:

Since $\mu = 3$, a disjoint encoding requires $u = \lceil \log_2 3 \rceil = 2$ intermediate variables:

$$h_1 = \Psi_1 = \bar{x}_1 x_2 \vee \bar{x}_2 x_3 \vee \bar{x}_3 x_1, \text{and}$$
$$h_2 = \Psi_2 = x_1 x_2 x_3.$$

Encoding that Simplifies an Intermediate Variable:

Next, let us use Algorithm 6.3.1 to simplify an intermediate variable. First, confirm that Ψ_i, $(i = 0, 1, 2)$ satisfies the conditions of Theorem 6.3.1:

$$\bar{x}_1 \Psi_0 = \bar{x}_1 \bar{x}_2 \bar{x}_3,$$
$$\bar{x}_1 \Psi_1 = \bar{x}_1 (x_2 \vee \bar{x}_2 x_3),$$
$$\bar{x}_1 \Psi_2 = 0,$$

and

$$x_1 \Psi_0 = 0,$$
$$x_1 \Psi_1 = x_1 (\bar{x}_2 x_3 \vee \bar{x}_3), \text{and}$$
$$x_1 \Psi_2 = x_1 x_2 x_3.$$

Since the number of nonzero functions is $2 \leq 2^{u-1}$, where $u = \lceil \log_2 3 \rceil = 2$, the conditions are satisfied. Next, assign codes to the columns:

To $\bar{x}_1 \Psi_0$ assign 00,
to $\bar{x}_1 \Psi_1$ assign 01,
to $x_1 \Psi_1$ assign 11, and
to $x_1 \Psi_2$ assign 10.

Thus, we have the following intermediate variables:

$$h_1 = \bar{x}_1 \Psi_1 \vee x_1 \Psi_1 = \bar{x}_1 (x_2 \vee \bar{x}_2 x_3) \vee x_1 (\bar{x}_2 x_3 \vee \bar{x}_3) = \Psi_1$$
$$h_2 = x_1 \Psi_1 \vee x_1 \Psi_2 = x_1$$

Note that

$$\bar{h}_2 \bar{h}_1 = \bar{x}_1 \bar{x}_2 \bar{x}_3 = \Psi_0$$
$$h_2 \bar{h}_1 = x_1 x_2 x_3 = \Psi_2.$$

Therefore, Ψ_i $(i = 0, 1, 2)$ can be represented by (h_2, h_1). In this encoding, only the LUTs for h_1 are needed. No LUT is necessary to implement $h_2 = x_1$, since x_1 is available as an input. ■

Example 6.3.3. Consider a function $f(X_1, X_2)$, where f is partially symmetric with respect to $X_1 = (x_1, x_2, x_3, x_4)$. Also, assume that the column multiplicity for the decomposition $f(X_1, X_2)$ is five. In this case, the equivalence classes for the decomposition for f with the partition (X_1, X_2) are

$$\begin{aligned}
\Psi_0 &= S_0^4(x_1, x_2, x_3, x_4) = \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_3, \\
\Psi_1 &= S_1^4(x_1, x_2, x_3, x_4), \\
\Psi_2 &= S_2^4(x_1, x_2, x_3, x_4), \\
\Psi_3 &= S_3^4(x_1, x_2, x_3, x_4), \\
\Psi_4 &= S_4^4(x_1, x_2, x_3, x_4) = x_1x_2x_3x_4.
\end{aligned}$$

Since $\mu = 5$, e variables. Note that

$$\begin{aligned}
\bar{x}_1\Psi_4 &= 0, \text{ and} \\
x_1\Psi_0 &= 0.
\end{aligned}$$

In this case $\Psi_i, (i = 0, 1, \ldots, 4)$ satisfies the conditions of Theorem 6.3.1. Thus, an intermediate variable h_3 can be simplified to x_1. Next, assign codes to columns as follows:

$$\begin{aligned}
&\text{To } \bar{x}_1\Psi_0 \text{ assign } 000, \\
&\text{to } \bar{x}_1\Psi_1 \text{ assign } 001, \\
&\text{to } \bar{x}_1\Psi_2 \text{ assign } 010, \\
&\text{to } \bar{x}_1\Psi_3 \text{ assign } 011, \\
&\text{to } x_1\Psi_1 \text{ assign } 101, \\
&\text{to } x_1\Psi_2 \text{ assign } 110. \\
&\text{to } x_1\Psi_3 \text{ assign } 111, \text{ and} \\
&\text{to } x_1\Psi_4 \text{ assign } 100.
\end{aligned}$$

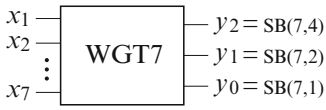
Using these codes, we derive the following expressions for the intermediate variables:

$$\begin{aligned}
h_1 &= \bar{x}_1\Psi_1 \vee \bar{x}_1\Psi_3 \vee x_1\Psi_1 \vee x_1\Psi_3 = \Psi_1 \vee \Psi_3 \\
h_2 &= \bar{x}_1\Psi_2 \vee \bar{x}_1\Psi_3 \vee x_1\Psi_2 \vee x_1\Psi_3 = \Psi_2 \vee \Psi_3 \\
h_3 &= x_1\Psi_1 \vee x_1\Psi_2 \vee x_1\Psi_3 \vee x_1\Psi_4 = x_1
\end{aligned}$$

Note that in this case,

$$\begin{aligned}
\bar{h}_3\bar{h}_2\bar{h}_1 &= \Psi_0, \\
\bar{h}_2h_1 &= \Psi_1, \\
h_2\bar{h}_1 &= \Psi_2, \\
h_2h_1 &= \Psi_3, \text{and} \\
h_3\bar{h}_2\bar{h}_1 &= \Psi_4.
\end{aligned}$$

Fig. 6.4 WGT7



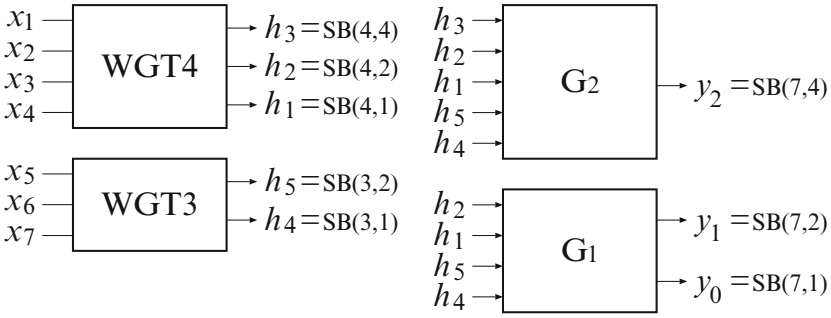


Fig. 6.5 Realization of WGT7 with 8 LUTs

In this encoding, only the LUTs for h_1 and h_2 are needed. No LUT is necessary to implement $h_3 = x_1$, since x_1 is available as an input. ■

Example 6.3.4. Realize WGT7 using LUTs with up to 5 inputs.

(Solution) WGT7 has seven inputs and three outputs as shown in Fig. 6.4. It counts the number 1 of 1's in the input, and represents it by a binary number (y_2, y_1, y_0), where $y_2 = SB(7,4)$, $y_1 = SB(7,2)$, and $y_0 = SB(7,1)$. Let X be partitioned as (X_1, X_2), where $X_1 = (x_1, x_2, x_3, x_4)$ and $X_2 = (x_5, x_6, x_7)$. Note that the functions are symmetric with respect to X_1 and X_2. The column multiplicity of the decomposition chart (X_1, X_2) is five. So, the straightforward realization produces the network shown in Fig. 6.5, where WGT4 is a 4-input bit-counting circuit and produces three functions:

$$
\begin{aligned}
h_3 &= SB(4,4) = x_1x_2x_3x_4, \\
h_2 &= SB(4,2) = x_1(x_2 \oplus x_3 \oplus x_4) \oplus x_2(x_3 \oplus x_4) \oplus x_3x_4, \text{ and} \\
h_1 &= SB(4,1) = x_1 \oplus x_2 \oplus x_3 \oplus x_4.
\end{aligned}
$$

Also, WGT3 is a 3-input bit-counting circuit (i.e., a full adder) and produces two functions:

$$
\begin{aligned}
h_5 &= SB(3,2) = x_5x_6 \oplus x_6x_7 \oplus x_7x_5, \text{ and} \\
h_4 &= SB(3,1) = x_5 \oplus x_6 \oplus x_7.
\end{aligned}
$$

G_1 adds two 2-bit numbers (h_2, h_1) and (h_5, h_4), producing the two least significant bits of the sum. And G_2 adds a 2-bit number (h_5, h_4) and a 3-bit number (h_3, h_2, h_1), producing the most significant bit of the sum. In Fig. 6.5, WGT4 has three outputs and requires three LUTs. Note that

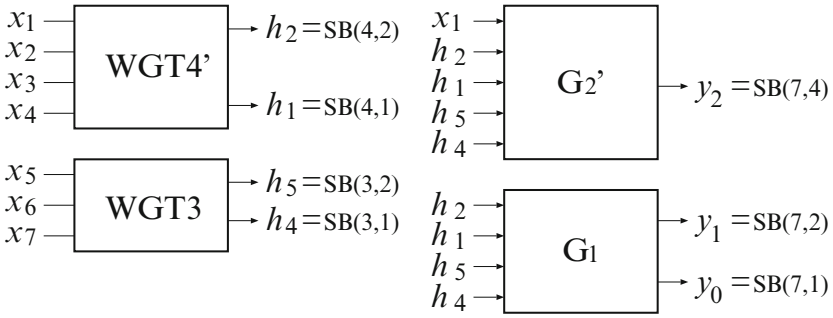


Fig. 6.6 Realization of WGT7 with 7 LUTs

$$y_2 = h_3 \oplus h_4 h_5 \oplus h_1 h_4 (h_2 \oplus h_5),$$
$$y_1 = h_2 \oplus h_5 \oplus h_1 h_4, \text{ and}$$
$$y_0 = h_1 \oplus h_4.$$

Since each output of G_1 and G_2 in Fig. 6.5 requires one LUT, we need 8 LUTs in total. However, if $h_3 = SB(4,4)$ is replaced by x_1 as shown in Fig. 6.6, we need only 7 LUTs. In this case, we use the relation $h_3 = x_1 \bar{h}_1 \bar{h}_2$, and

$$y_2 = x_1 \bar{h}_1 \bar{h}_2 \oplus h_2 h_5 \oplus h_1 h_4 (h_2 \oplus h_5).$$

■

Example 6.3.5. Realize WGT8 by LUTs with up to 5 inputs.

(Solution) WGT8 realizes the four functions SB(8,8), SB(8,4), SB(8,2), and SB(8,1). Let X be partitioned as $X = (X_1, X_2, X_3)$, where $X_1 = (x_1, x_2, x_3, x_4)$, $X_2 = (x_5, x_6)$, and $X_3 = (x_7, x_8)$.

First, realize

$$SB(4,4) = x_1 x_2 x_3 x_4,$$
$$SB(4,2) = x_1(x_2 \oplus x_3 \oplus x_4) \oplus x_2(x_3 \oplus x_4) \oplus x_3 x_4, \text{ and}$$
$$SB(4,1) = x_1 \oplus x_2 \oplus x_3 \oplus x_4.$$

Then, realize

$$SB(6,4) = SB(4,4) \oplus SB(4,3)(x_5 \oplus x_6) \oplus SB(4,2) x_5 x_6,$$
$$SB(6,2) = SB(4,2) \oplus SB(4,1)(x_5 \oplus x_6) \oplus x_5 x_6, \text{ and}$$
$$SB(6,1) = SB(4,1) \oplus x_5 \oplus x_6.$$

Note that we use the relation in Lemma 3.5.3:

$$SB(4,3) = SB(4,2) SB(4,1).$$

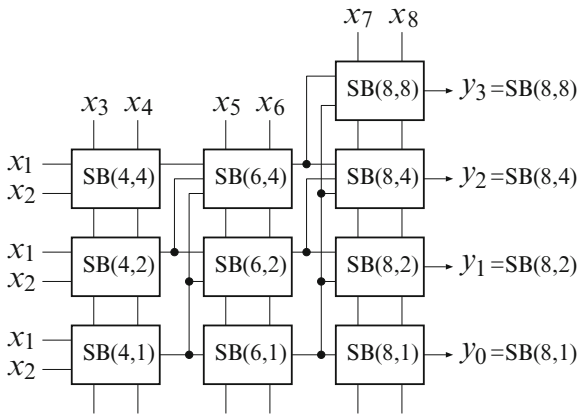


Fig. 6.7 Realization of WGT8

Finally, realize

$$
\begin{aligned}
SB(8,8) &= SB(6,6)x_7x_8,\\
SB(8,4) &= SB(6,4) \oplus SB(6,3)(x_7 \oplus x_8) \oplus SB(6,2)x_7x_8,\\
SB(8,2) &= SB(6,2) \oplus SB(6,1)(x_7 \oplus x_8) \oplus x_7x_8, \text{and}\\
SB(8,1) &= SB(6,1) \oplus x_7 \oplus x_8.
\end{aligned}
$$

In this case, we use the relation in Lemma 3.5.3:

$$
\begin{aligned}
SB(6,6) &= SB(6,4)SB(6,2), \text{and}\\
SB(6,3) &= SB(6,2)SB(6,1).
\end{aligned}
$$

Thus, WGT8 is realized as Fig. 6.7. However, if we use the relation

$$SB(4,4) = x_1x_2x_3x_4 = x_1\overline{SB(4,2)}\,\overline{SB(4,1)},$$

the LUT for $SB(4,4)$ can be replaced by a variable x_1. Thus, WGT8 requires only 9 LUTs. ■

Example 6.3.6. Realize the 9-input symmetric function *SYM9* using LUTs with up to 5 inputs.

(Solution) SYM9 is represented as

$$f = S^9_{\{3,4,5,6\}}(x_1, x_2, \ldots, x_9).$$

$f = 1$ if and only if the number of 1's in the input is 3, 4, 5, or 6. Suppose that the function is decomposed as $f(X_1, X_2, X_3)$, where $X_1 = (x_1, x_2, x_3, x_4, x_5)$, $X_2 = (x_6, x_7)$, and $X_3 = (x_8, x_9)$. When X_1 is the set of bound variables, the equivalence classes are

$$\Psi_0 = S_0^5(x_1, x_2, x_3, x_4, x_5) = \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5,$$
$$\Psi_1 = S_1^5(x_1, x_2, x_3, x_4, x_5),$$
$$\Psi_2 = S_2^5(x_1, x_2, x_3, x_4, x_5),$$
$$\Psi_3 = S_3^5(x_1, x_2, x_3, x_4, x_5),$$
$$\Psi_4 = S_4^5(x_1, x_2, x_3, x_4, x_5), \text{ and}$$
$$\Psi_5 = S_5^5(x_1, x_2, x_3, x_4, x_5) = x_1x_2x_3x_4x_5.$$

Suppose that the first cell realizes *WGT5*. Next, consider the decomposition where X_1 and X_2 are bound variables. Suppose that the second cell realizes WGT7 as shown in Fig. 6.8, where

$$y_2 = SB(7, 4),$$
$$y_1 = SB(7, 2), \text{ and}$$
$$y_0 = SB(7, 1).$$

From the definition of SYM9, we have Table 6.2 showing the function of the rightmost cell. As shown in Fig. 6.8, the network for Table 6.2 requires only one LUT. Thus, SYM9 is realized by seven 5-LUTs.

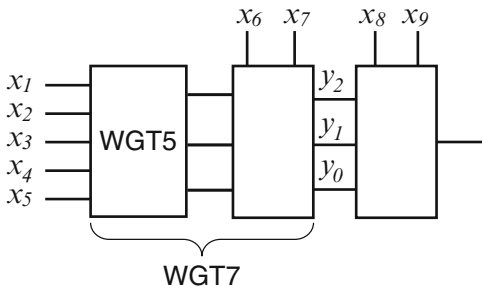


Fig. 6.8 Realization for SYM9

Table 6.2 Truth table for the rightmost cell

y_2	y_1	y_0	x_8	x_9	f
0	0	0	–	–	0
0	0	1	1	1	1
0	1	0	1	–	1
0	1	0	–	1	1
0	1	1	–	–	1
1	0	0	–	–	1
1	0	1	0	–	1
1	0	1	–	0	1
1	1	0	0	0	1
1	1	1	–	–	0

Next, consider whether any output function of WGT7 can be simplified. Table 6.2 shows that when WGT7 produces (0,1,1) and (1,0,0), the value of the function is 1, independent of the values of x_8 and x_9. Thus, the equivalence classes are

$$\begin{aligned}
\Psi_0 &= S_0^7(x_1, x_2, \ldots, x_7) = \bar{x}_1\bar{x}_2 \ldots \bar{x}_6\bar{x}_7,\\
\Psi_1 &= S_1^7(x_1, x_2, \ldots, x_7),\\
\Psi_2 &= S_2^7(x_1, x_2, \ldots, x_7),\\
\Psi_3 &= S_{\{3,4\}}^7(x_1, x_2, \ldots, x_7),\\
\Psi_4 &= S_5^7(x_1, x_2, \ldots, x_7),\\
\Psi_5 &= S_6^7(x_1, x_2, \ldots, x_7), \text{and}\\
\Psi_6 &= S_7^7(x_1, x_2, \ldots, x_7) = x_1x_2 \ldots x_6x_7.
\end{aligned}$$

In this case, we cannot simplify any output of WGT7 by using Theorem 6.3.1, since the number of nonzero functions in $x_1\Psi_i$ and $\bar{x}_1\Psi_i$, where ($i = 0, 1, 2, 3, 4, 5, 6$), is both six. ■

Example 6.3.7. Design a 2-MUX using 4-LUTs, where a 2-MUX realizes the function

$$f(y_0, y_1, y_2, y_3, x_1, x_2) = y_0\bar{x}_1\bar{x}_2 \vee y_1\bar{x}_1x_2 \vee y_2x_1\bar{x}_2 \vee y_3x_1x_2.$$

(Solution) Let (X_1, X_2) be a partition of the input variables, where $X_1 = (y_0, y_1, x_1, x_2)$ and $X_2 = (y_2, y_3)$. The equivalence classes for the decomposition are

$$\begin{aligned}
\Psi_0 &= \bar{x}_1(\bar{x}_2\bar{y}_0 \vee x_2\bar{y}_1),\\
\Psi_1 &= \bar{x}_1(\bar{x}_2y_0 \vee x_2y_1),\\
\Psi_2 &= x_1\bar{x}_2, \text{and}\\
\Psi_3 &= x_1x_2.
\end{aligned}$$

Since $\mu = 4$, disjoint encoding requires two intermediate variables h_1 and h_2. Note that

$$\begin{aligned}
x_1\Psi_0 &= 0\\
x_1\Psi_1 &= 0
\end{aligned}$$

and

$$\begin{aligned}
\bar{x}_1\Psi_2 &= 0\\
\bar{x}_1\Psi_3 &= 0.
\end{aligned}$$

Thus, Ψ_i $(i = 0, 1, 2, 3)$ satisfies the conditions of Theorem 6.3.1. Next, assign codes (h_2, h_1) to columns as follows:

$$\begin{aligned}
&\text{To } \bar{x}_1\Psi_0 = \Psi_0 \text{ assign a code 00,}\\
&\text{to } \bar{x}_1\Psi_1 = \Psi_1 \text{ assign a code 01,}\\
&\text{to } x_1\Psi_2 = \Psi_2 \text{ assign a code 10, and}\\
&\text{to } x_1\Psi_3 = \Psi_3 \text{ assign a code 11.}
\end{aligned}$$

In this case, we have the following intermediate variables:

$$\begin{aligned}
h_2 &= x_1\Psi_2 \vee x_1\Psi_3 = x_1,\\
h_1 &= \bar{x}_1\Psi_1 \vee x_1\Psi_3 = \bar{x}_1(\bar{x}_2 y_0 \vee x_2 y_1) \vee x_1 x_2 = g.
\end{aligned}$$

Note that f can be represented as

$$\begin{aligned}
f(y_0, y_1, y_2, y_3, x_1, y_2) &= \Psi_1 \vee \Psi_2 y_2 \vee \Psi_3 y_3,\\
&= \bar{h}_2 h_1 \vee h_2 \bar{h}_1 y_2 \vee h_2 h_1 y_3,\\
&= \bar{x}_1 g \vee x_1(\bar{g} y_2 \vee g y_3).
\end{aligned}$$

In this way, a 2-MUX can be realized by two 4-LUTs as shown in Fig. 4.4. This is a method to find a nondisjoint decomposition of Lemma 4.1.2. ■

6.4 Remarks

In this chapter, nondisjoint encoding was introduced. It reduces the number of LUTs in a cascade realization by deriving a nondisjoint decomposition. Nondisjoint encoding for symmetric functions was considered in [112]. Experimental results show that with nondisjoint encodings, the number of LUTs can be reduced by 10–30% [35, 68, 69], when multiple-output functions were implemented as encoded characteristic function for nonzero outputs (ECFNs). This chapter is based on [112]. Encoding method in the decomposition that minimized the support is considered in [21, 58, 155]

Problems

6.1. Consider the decomposition $f(X_1, X_2)$, where $X_1 = (x_1, x_2, x_3, x_4)$ and $X_2 = (x_5, x_6, x_7, x_8)$. Let the equivalence classes of the decomposition be

$$\begin{aligned}
\Psi_0 &= \bar{x}_1\bar{x}_2\bar{x}_3,\\
\Psi_1 &= \bar{x}_1 x_2 \bar{x}_3,
\end{aligned}$$

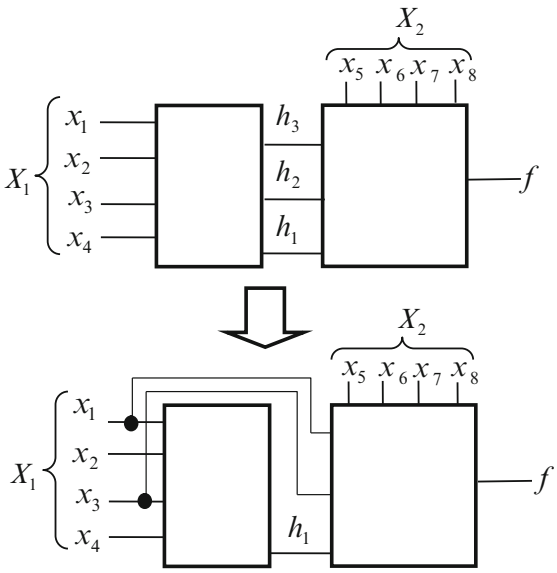


Fig. 6.9 Simplification of intermediate variables

$$\Psi_2 = x_1\bar{x}_3x_4,$$
$$\Psi_3 = x_1\bar{x}_3\bar{x}_4,$$
$$\Psi_4 = x_3.$$

Simplify an intermediate variable if possible.

6.2. Realize the 8-variable symmetric function $f = S^8_{\{0,8\}}$ using 5-LUTs. Note that $f = 1$ iff $\sum_{i=1}^{8} x_i = 0$ or 8.

6.3. By extending Theorem 6.3.1, obtain the condition that the intermediate variables h_u can be represented by x_j, and h_{u-1} can be represented by x_k, where $j \neq k$. For example, in Fig. 6.9, two intermediate variables h_3 and h_2 are replaced by input variables x_1 and x_3, respectively.

Chapter 7
Functions with Small C-Measures

Recall that the C-measure of a function is the maximum column multiplicity among a set of functional decompositions $f(X_1, X_2)$, where $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$. The C-measure tends to increase exponentially with the number of input variables, n. However, many practical functions have small C-measures. This chapter considers classes of functions whose C-measures are small. Such functions can be efficiently realized by LUT cascades.

7.1 C-Measure and BDDs

The column multiplicity of a decomposition chart is equal to the width of the quasi-reduced multi-terminal binary decision diagram (QRMTBDD). So, the C-measure of a logic function is equal to the maximum width of the MTBDD for the given ordering of the input variables.

Example 7.1.1. Consider two functions:

$$f_1(x_1, x_2, x_3, x_4, x_5, x_6) = x_1x_2 \vee x_3x_4 \vee x_5x_6$$

and

$$f_2(x_1, x_2, x_3, x_4, x_5, x_6) = x_1x_4 \vee x_2x_5 \vee x_3x_6.$$

Figure 7.1 shows the BDDs for f_1 and f_2. In this case, the C-measures of f_1 and f_2 are 3 and 8, respectively. Note that f_2 can be obtained from f_1 by permuting the input variables. We consider these functions to be different. ■

Lemma 7.1.1. *Consider a pair of* $2n$ *variable functions:*

$$f_1(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) = x_1x_2 \vee x_3x_4 \vee \cdots \vee x_{2n-1}x_{2n}$$

and

$$f_2(x_1, x_2, \ldots x_{2n-1}, x_{2n}) = x_1x_{n+1} \vee x_2x_{n+2} \vee \cdots \vee x_nx_{2n}.$$

Then, $\mu(f_1) = 3$ *and* $\mu(f_2) = 2^n$.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_7,

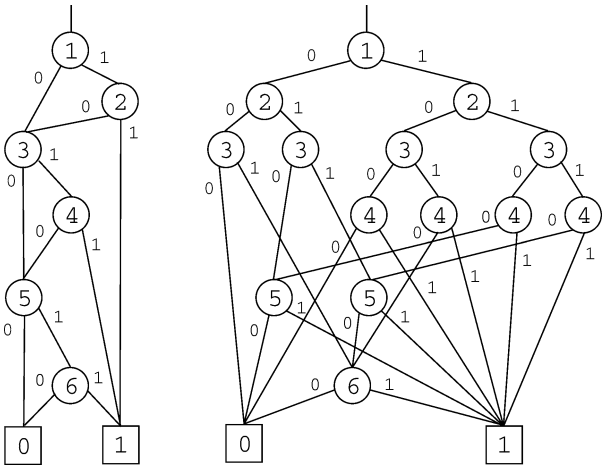


Fig. 7.1 BDD for equivalent functions

For certain function, the C-measure is large, but when variables are permuted, the C-measure becomes smaller. For the other function, the C-measure is large for any permutation of the input variables. For a given logic function and variable ordering, the C-measure is easy to obtain and is uniquely defined.

7.2 Symmetric Functions

Functions that appear in arithmetic circuits often have symmetries. When logic functions have some symmetries, they are often realized using fewer elements.

Definition 7.2.1. A function f is a **totally symmetric function** if any permutation of the variables in f does not change the function. A totally symmetric function is also called a **symmetric function**.

Lemma 7.2.1. *Let f be a symmetric function of n variables. Then,*

$$\mu(f) \leq \max_{k=1}^{n} \min\{k+1, 2^{n-k+1}\}.$$

Proof. Consider the decomposition chart of f, where $X_1 = (x_1, x_2, \ldots, x_k)$ denotes the bound variables and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$ denotes the free variables. The number of different column functions is at most $k+1$, since the column pattern depends only on the number of 1's in the bound variables. Also, by Lemma 3.5.1, the number of column functions is at most 2^{n-k+1}, since the column functions are symmetric functions of $n-k$ variables. □

Theorem 7.2.1. *Let f be a symmetric function f of n variables. Then,*

$$\mu(f) \leq n, \qquad (n \geq 4).$$
$$\mu(f) \leq n-1, (n \geq 9).$$
$$\mu(f) \leq n-2, (n \geq 18).$$
$$\mu(f) \leq n-3, (n \geq 35).$$

Proof. From Lemma 7.2.1, $\mu_k = \max_{k=1}^{n} \min\{k+1, 2^{n-k+1}\}$. Consider the cases, where $k = n, n-1, n-2, n-3$ and $n-4$. In these cases, we have

$$\begin{aligned}
\mu_n &= \min\{n+1, 2^1\} = 2, (n \geq 1),\\
\mu_{n-1} &= \min\{n+0, 2^2\} = 4, (n \geq 4),\\
\mu_{n-2} &= \min\{n-1, 2^3\} = 8, (n \geq 9),\\
\mu_{n-3} &= \min\{n-2, 2^4\} = 16, (n \geq 18), \text{ and}\\
\mu_{n-4} &= \min\{n-3, 2^5\} = 32, (n \geq 35).
\end{aligned}$$

From the above, we have the theorem. □

7.3 Sparse Functions

An integer function whose number of nonzero output values is much smaller than the total number of input combinations is called **sparse**. Sparse functions can be efficiently realized by an LUT cascade.

Theorem 7.3.1. *Let f be an integer valued function or a logic function with weight k. Then, $\mu(f) \leq k+1$.*

Proof. Consider a decomposition chart with the maximum column multiplicity. In this case, each nonzero element corresponds to a unique column pattern. Also, there can be a column with all zero elements. Thus, there is no decomposition chart with greater column multiplicity. Hence, we have the theorem. □

Sparse functions are considered in Chap. 8.

7.4 LPM Functions

The **longest prefix match** (**LPM**) problem is to determine the output port address from a list of prefix vectors stored in memory based on the longest match. It is solved by the internet routers to forward packets of data.

Definition 7.4.1. [125] The **LPM table** stores distinct ternary vectors of the form $VEC_1 \cdot VEC_2$, where VEC_1 is a string of 0's and 1's, and VEC_2 is a string of *'s.

To assure that the longest prefix address is produced, LPM entries are stored in descending prefix length. The first match determines the LPM tables output. The corresponding **LPM function** is a logic function $\vec{f} : B^n \rightarrow B^m$, where $\vec{f}(\vec{x})$ is the smallest index of an entry that is identical to $\vec{x}$ except possibly for don't care values. If no such entry exists, $\vec{f}(\vec{x}) = 0^m$. A circuit that realizes the LPM function is an **LPM index generator**.

Example 7.4.1. Consider the LPM table shown in Table 7.1. In the third row $VEC_1 = 01$ and $VEC_2 = **$, while in the last row $VEC_1 = 0$ and $VEC_2 = ***$. Table 7.2 shows the corresponding LPM function. The output is the index corresponding to the index of the longest prefix that matches the input. ■

Example 7.4.2. In **Internet Protocol version 4** (**IPv4**), an IP address is represented by 32 bits or 4 bytes. An IP address is often represented by four decimal numbers, each representing a byte. For example, 66.249.122.7 corresponds to the 32-bit binary number

01000010.11111001.01111010.00000111.

Table 7.1 LPM table

Vector				
x_1	x_2	x_3	x_4	Index
1	0	0	0	1
0	1	0	*	2
0	1	*	*	3
1	*	*	*	4
0	*	*	*	5

Table 7.2 LPM function truth table

x_1	x_2	x_3	x_4	f_2	f_1	f_0
0	0	0	0	1	0	1
0	0	0	1	1	0	1
0	0	1	0	1	0	1
0	0	1	1	1	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	0	0	1
1	0	0	1	1	0	0
1	0	1	0	1	0	0
1	0	1	1	1	0	0
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	1	0	0

That is, 66 in the decimal number represents the binary number 01000010; 249 in the decimal number represents the binary number 11111001; 122 in the decimal number represents the binary number 01111010; and 7 in the decimal number represents the binary number 00000111.

Consider the IP forwarding table shown in Fig. 7.2. It finds the longest prefix that matches the incoming destination address, and produces the corresponding next hop address. For example, when the input address is 66.249.122.7, the LPM is the first entry. Note that in this case, both the first and the second entries match. The first entry matches three bytes (66,249, and 122), while the second entry matches only two bytes (66 and 249). Thus, we select the first element, and its specified next hop address 161.4.2.22. ■

Ternary content addressable memory, (TCAM) explained in Chap. 2, directly realizes an LPM function.

Example 7.4.3. Figure 7.2 is an example of a forwarding table, and Fig. 7.3 shows the TCAM for the table. In the TCAM, the entries are sorted in the order of decreasing prefix length. When the incoming address is 66.249.122.7, both the first and the second prefixes match. The priority encoder selects the least index. ■

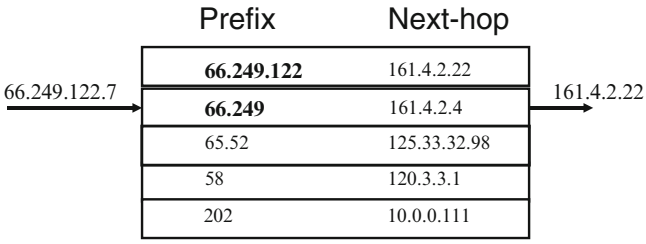


Fig. 7.2 Example of IP look-up

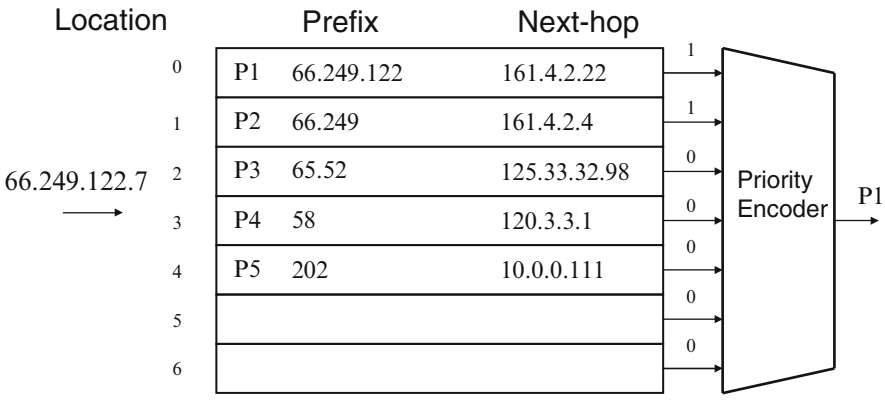


Fig. 7.3 Longest prefix matching by TCAM

Theorem 7.4.1. *[125] Let f be an LPM function with k vectors. Then, $\mu(f) \leq k + 1$.*

Proof. Consider a decomposition chart where $X_1 = (x_1, x_2, \ldots, x_t)$ denotes the bound variables and $X_2 = (x_{t+1}, x_{t+2}, \ldots, x_n)$ denotes the free variables. We prove the theorem for the case where all k entries in the LPM table map to distinct output values. This is the worst case, since forcing certain entries to have the same output value can only reduce the column multiplicity.

We prove the theorem by counting the number of distinct columns in the decomposition chart as LPM vectors are added to the LPM table. Reorder the LPM vectors so that those vectors with the most * entries are first and those with the fewest are last. An *empty* decomposition chart has a unique column pattern (all 0's). Let the first vector be $\vec{\alpha} = (a_1, a_2, ..., a_m, *, *, ..., *)$, where $a_j \in B = \{0, 1\}$. If $m > t$, then the first vector changes only a proper subset of elements in one column. If $m = t$ ($m < t$), then the new vector changes all elements in one (or more) complete column(s) to the vector's output value in the LPM table. In either case, at most one distinct column pattern is added to the decomposition chart.

Because the second vector has no more * entries than the first vector, adding it will change columns only among a subset of the two distinct columns so far in the decomposition chart. Let the new vector be $\vec{\beta} = (b_1, b_2, ..., b_{m'}, *, *, ..., *)$, where $b_j \in B$. If $b_i = a_i$, for all $1 \leq i \leq m'$, then a subset of the columns created by adding $\vec{\alpha}$ to the empty decomposition chart are changed. Otherwise, a subset of the columns containing all 0's are changed. In either case, at most one additional column pattern is added. This process continues until all vectors are exhausted. In all, at most $k + 1$ column patterns are created. The theorem follows. □

7.5 Segment Index Encoder Function

Definition 7.5.1. [116] A **Segment Index Encoder (SIE) function** $g(X)$ is a mapping: $g : I \rightarrow I$, where I is a set of non-negative integers, and $a \leq b$ implies $g(a) \leq g(b)$. **Segment index logic function** f is the SIE function represented by binary variables. We assume that the integer represented by $X = (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$ is $\sum_{i=0}^{n-1} 2^i x_i$.

Example 7.5.1. Table 7.3 shows an example of a SIE function. Note that it is a monotone-increasing function with respect to $X = \sum_{i=0}^{3} 2^i x_i$. ■

Theorem 7.5.1. *[116] Let k be the number of segments in an segment index function g. Let f be the corresponding segment index logic function. Then, $\mu(f) \leq k$. We assume that the integer represented by $X = (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$ is $\sum_{i=0}^{n-1} 2^i x_i$.*

Proof. Consider the decomposition chart of a segment index logic function $f(X_1, X_2)$. Let $X_1 = (x_{n-p-1}, x_{n-p-2}, \ldots, x_0)$ and $X_2 = (x_{n-1}, x_{n-2}, \ldots, x_{n-p})$.

Table 7.3 Example of segment index encoder function

x_3	x_2	x_1	x_0	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	2
0	1	0	1	2
0	1	1	0	2
0	1	1	1	2
1	0	0	0	2
1	0	0	1	3
1	0	1	0	3
1	0	1	1	3
1	1	0	0	4
1	1	0	1	4
1	1	1	0	4
1	1	1	1	5

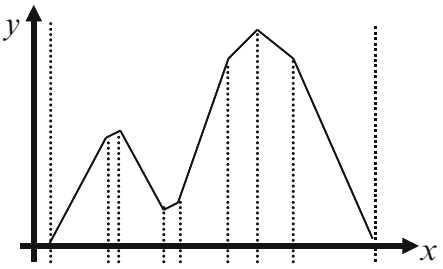


Fig. 7.4 Partition of the domain

Assume, without loss of generality, that both the columns and rows are labeled in ascending order of the values of X_1 and X_2, respectively. Because $g(X)$ is a monotone-increasing function, in scanning left-to-right and then top-to-bottom, the values of $g(X)$ will never decrease. An increase causes two columns to be distinct. Conversely, if no increase occurs anywhere across two adjacent columns, they are identical. In a monotone-increasing k-valued output function, there are $k-1$ dividing lines among 2^n output values. Dividing lines among values divide columns in the decomposition chart. Thus, there can be at most k distinct columns. □

The SIE functions are used in numerical function generators (NFGs).

We developed an architecture and a synthesis method for programmable NFGs for elementary functions such as trigonometric, logarithmic, square root, and reciprocal functions [116, 122, 130]. As shown in Fig. 7.4, a given domain of the function is partitioned into nonuniform segments. For each segment, the function is approximated by a linear function using the architecture shown in Fig. 7.5. By Theorem 7.5.1, when the number of segments is small, the SIE has a small C-measure, and can be efficiently realized by an LUT cascade. In this way, we can implement fast and compact NFGs for a wide range of functions.

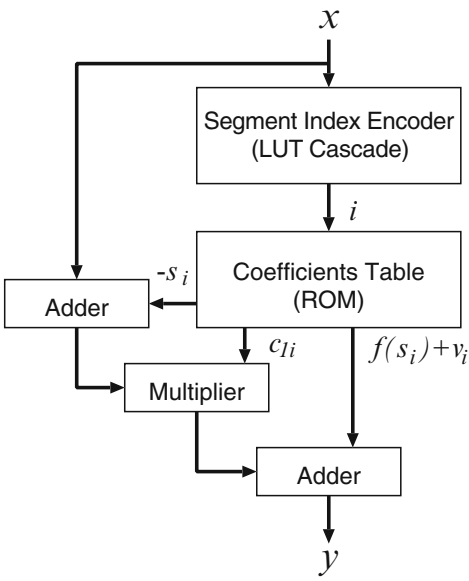


Fig. 7.5 Numerical function generator using first-order approximation

7.6 WS Functions

A weighted-sum function (WS function) [120, 127] is a mathematical model of bit-counting circuits [110], radix converters [43, 119], and distributed arithmetic [121].

Definition 7.6.1. An n-input **WS function** $F(X)$ computes

$$WS(X) = \sum_{i=1}^{n} w_i \cdot x_i,$$

where $X = (x_1, x_2, \ldots, x_n)$ is the **input vector**, $W = (w_1, w_2, \ldots, w_n)$ is the **weight vector**, and w_i $(i = 1, 2, \ldots, n)$ is a positive or negative integer. Let $F = (f_{q-1}, f_{q-2}, \ldots, f_0)$ be the binary representation of the WS function. Then, we have

$$WS(X) = \sum_{i=0}^{q-1} f_i(X) \cdot 2^i.$$

WS functions are used in radix converters and digital filters.

Theorem 7.6.1. *Let $F(X)$ be a WS function with the weight vector $W = (w_1, w_2, \ldots, w_n)$. Let (X_1, X_2) be a partition of $X = (x_1, x_2, \ldots, x_n)$, where $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$. Consider the decomposition chart of F, where X_1 denotes the bound variables and X_2 denotes the free variables. Then, the column multiplicity of the decomposition chart is at most $UB1 = 1 + \sum_{j=1}^{k} |w_j|$.*

Table 7.4 Decomposition chart for a WS function

$X_1 = (x_1, x_2, x_3)$

	000	001	010	011	100	101	110	111
00	0	w_3	w_2	$w_2 + w_3$	w_1	$w_1 + w_3$	$w_1 + w_2$	$w_1 + w_2 + w_3$
01	w_5	w_3+ w_5	w_2+ w_5	$w_2 + w_3+$ w_5	w_1+ w_5	$w_1 + w_3+$ w_5	$w_1 + w_2+$ w_5	$w_1 + w_2 + w_3+$ w_5
10	w_4	w_3+ w_4	w_2+ w_4	$w_2 + w_3+$ w_4	w_1+ w_4	$w_1 + w_3+$ w_4	$w_1 + w_2+$ w_4	$w_1 + w_2 + w_3+$ w_4
11	$w_4 + w_5$	w_3+ $w_4 + w_5$	w_2+ $w_4 + w_5$	$w_2 + w_3+$ $w_4 + w_5$	w_1+ $w_4 + w_5$	$w_1 + w_3+$ $w_4 + w_5$	$w_1 + w_2+$ $w_4 + w_5$	$w_1 + w_2 + w_3+$ $w_4 + w_5$

Table 7.5 A decomposition chart of a WS function (integer representation)

		0	0	0	0	1	1	1	1	x_1
		0	0	1	1	0	0	1	1	x_2
		0	1	0	1	0	1	0	1	x_3
0	0	0	3	2	5	1	4	3	6	
0	1	5	8	7	10	6	9	8	11	
1	0	4	7	6	9	5	8	7	10	
1	1	9	12	11	14	10	13	12	15	
x_4	x_5									

Proof. Consider the decomposition chart for $WS(X_1, X_2)$. In the first row, $X_2 = (0, 0, \ldots, 0)$. Note that the column multiplicity is equal to the number of different values in the first row. For example, Table 7.4 shows the case of $n = 5$ and $k = 3$. Consider the case where all the weights are positive. In this case, the number of different values is at most $UB1$, since WS takes values from 0 to $\sum_{j=1}^{k} w_j$.

Consider the case where some of the weights are negative. Assume that $w_1, w_2, \ldots, w_t$ are negative, and $w_{t+1}, w_{t+2}, \ldots, w_k$ are positive. Then, the WS takes values from $\sum_{j=1}^{t} w_j$ to $\sum_{j=t+1}^{k} w_j$. In this case, the number of different values is at most $1 + \sum_{j=1}^{t} |w_j| + \sum_{j=t+1}^{k} w_j = 1 + \sum_{j=1}^{k} |w_j|$. From these, we can conclude that the column multiplicity of the decomposition chart is at most $UB1$. □

Example 7.6.1. Consider the WS function with $n = 5$ and $W = (w_1, w_2, w_3, w_4, w_5) = (1, 2, 3, 4, 5)$. Let $X_1 = (x_1, x_2, x_3)$ and $X_2 = (x_4, x_5)$. In this case, $UB1 = 1 + w_1 + w_2 + w_3 = 1 + 1 + 2 + 3 = 7$. Table 7.5 shows the decomposition chart of the function. Note that the column multiplicity of the decomposition chart is 7. So, the bound $UB1$ is tight. ■

Definition 7.6.2. A **threshold function** $f(x_0, x_1, \ldots, x_{n-1})$ satisfies the relation: $f = 1$ if $\sum_{i=1}^{n} w_i x_i \geq T$, and $f = 0$ otherwise, where $(w_0, w_1, \ldots, w_{n-1})$ is the **weight vector** and T is the **threshold**.

Although, a threshold function is not a WS function, we can estimate the column multiplicity of a threshold function from the theory of WS functions.

Theorem 7.6.2. *The column multiplicity of a decomposition chart of the threshold function with the weight vector* $(w_0, w_1, \ldots, w_{n-1})$ *is at most*

$$UB = 1 + \sum_{i=0}^{n-1} |w_i|.$$

Proof. The column multiplicity of a decomposition chart for f is no greater than that of the WS function having the same weight vector. By Theorem 7.6.1, the column multiplicity of the WS function is at most UB. Hence, we have the theorem. □

When the sum of the weights is large, a monolithic cascade realization of a WS function can be large. In this case, we can partition the outputs into groups, and realize each group separately.

Theorem 7.6.3. *[120, 127] Let* $F_{\mathrm{LSB}}(X)$ *be the logic function that represents the least significant* q *bits of a WS function. Then,* $F_{\mathrm{LSB}}(X)$ *can be realized by an LUT cascade consisting of cells with* $q + 1$ *inputs and* q *outputs.*

A $2q$-output WS function can be decomposed into a pair of WS functions as follows: Let, w_i be a weight of a $2q$-output WS function. Then, w_i can be written as

$$w_i = 2^q w_{Ai} + w_{Bi},$$

where w_{Ai} denotes most significant q bits, and w_{Bi} denotes least significant q bits. In this case, we can realize the $2q$-output WS function using a pair of WS functions and an adder, as shown in Fig. 7.6.

This is an **arithmetic decomposition** of a WS function. With this method, we can efficiently realize a WS function with cascades and adders [120, 127].

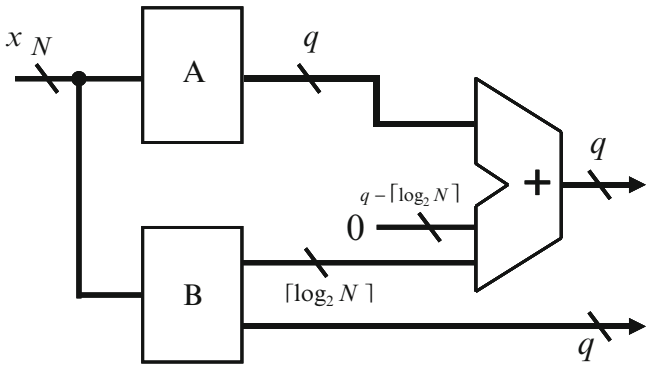


Fig. 7.6 Arithmetic decomposition of $2q$-output WS function

7.7 Modulo Function

Definition 7.7.1. Let $X = (x_{n-1}, x_{n-2} \ldots, x_1, x_0)$ be the input variables. A **modulo m function** is a mapping $B^n \rightarrow M$, where $M = \{0, 1, \ldots, m-1\}$. It computes

$$f(X) = \left(\sum_{i=0}^{n-1} 2^i x_i \right) \pmod{m}.$$

Theorem 7.7.1. *The C-measure of the modulo m function is m.*

The modulo m function can be realized by an LUT cascade with $\lceil \log_2 m \rceil$ rails. The depth of the circuit is $O(n)$.

Next, consider the decomposition of the modulo m function:

$$f(X_1, X_2) = g(h_1(X_1), h_2(X_2)),$$

where $X_1 = (x_{n-1}, x_{n-2} \ldots, x_{n-k})$, and $X_2 = (x_{n-k-1}, x_{n-k-2} \ldots, x_0)$. In this case, $h_1(X_1)$ computes

$$h_1(x_{n-1}, x_{n-2} \ldots, x_{n-k}) = \left(\sum_{i=n-k}^{n-1} 2^i x_i \right) \pmod{m},$$

while $h_2(X_2)$ computes

$$h_2(x_{n-k-1}, x_{n-k-2} \ldots, x_0) = \left(\sum_{i=0}^{n-k-1} 2^i x_i \right) \pmod{m}.$$

Also,

$$g(h_1, h_2) = h_1 + h_2 (\text{mod } m).$$

Note that the C-measures of $h_1(X_1)$ and $h_2(X_2)$ are also m. By decomposing recursively, we have a tree-type modulo circuit with depth $O(\log_2 n)$.

Lemma 7.7.1. *Let a, b, c, and d be integers satisfying the relation $a = bc + d$, then*

$$a \ (mod\ m) = [b \ (mod\ m)] \times [c \ (mod\ m)] + [d \ (mod\ m)].$$

Example 7.7.1. Let $m = 19$.

$$\begin{aligned} 1001 &= 20 \times 50 + 1 \\ 20 \pmod{19} &= 1 \\ 50 \pmod{19} &= 20 \times 2 + 10 \pmod{19} \\ &= 2 + 10 = 12 \end{aligned}$$

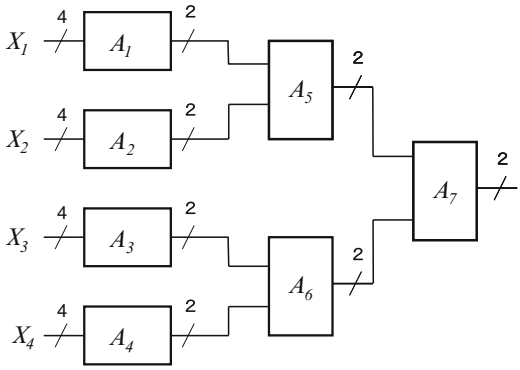


Fig. 7.7 Mod 3 circuit with LUTs

$$1001 \pmod{19} = 1 \times 12 + 1 = 13$$

■

Example 7.7.2. Realize the modulo function by k-LUTs, where $m = 3$, $n = 16$, and $k = 4$.

(Solution): The input variables are partitioned into $X_1 = (x_{15}, x_{14}, x_{13}, x_{12})$, $X_2 = (x_{11}, x_{10}, x_9, x_8)$, $X_3 = (x_7, x_6, x_5, x_4)$, and $X_4 = (x_3, x_2, x_1, x_0)$.

Figure 7.7 shows the circuit to compute the modulo m. The circuit A1 computes:

$$\begin{aligned} h_1(x_{15}, x_{14}, x_{13}, x_{12}) &= \left(\sum_{i=12}^{15} 2^i x_i \right) \pmod 3 \\ &= \left(2^{12} \sum_{i=0}^{3} 2^i x_{i+12} \right) \pmod 3 \\ &= \left(\sum_{i=0}^{3} 2^i x_{i+12} \right) \pmod 3. \end{aligned}$$

The circuit A2 computes:

$$\begin{aligned} h_2(x_{11}, x_{10}, x_9, x_8) &= \left(\sum_{i=8}^{11} 2^i x_i \right) \pmod 3 \\ &= \left(2^8 \sum_{i=0}^{3} 2^i x_{i+8} \right) \pmod 3 \\ &= \left(\sum_{i=0}^{3} 2^i x_{i+8} \right) \pmod 3. \end{aligned}$$

The circuit A3 computes:

$$\begin{aligned} h_3(x_7, x_6, x_5, x_4) &= \left(\sum_{i=4}^{7} 2^i x_i \right) \pmod 3 \\ &= \left(2^4 \sum_{i=0}^{3} 2^i x_{i+4} \right) \pmod 3 \\ &= \left(\sum_{i=0}^{3} 2^i x_{i+4} \right) \pmod 3. \end{aligned}$$

The circuit A4 computes:

$$h_4(x_3, x_2, x_1, x_0) = \left(\sum_{i=0}^{3} 2^i x_i \right) \pmod 3.$$

The circuits A5 and A6 compute:

$$\begin{aligned} h_5 &= (h_1 + h_2) \pmod 3 \\ h_6 &= (h_3 + h_4) \pmod 3. \end{aligned}$$

And, finally, A7 computes:

$$(h_5 + h_6) \pmod 3.$$

Note that in the above equations, we used the relations:

$$2^{12} \equiv 2^8 \equiv 2^4 \equiv 1 \pmod 3.$$

■

7.8 Remarks

Functions with small C-measures have efficient LUT cascade realizations. Practical functions often have small C-measures, although the fractions of functions with small C-measures approaches zero as the number of variables increases. An efficient method to obtain the C-measure of a logic function is to construct the BDD of the function. Functions that have small BDD representations are considered in [154].

Problems

7.1. Show the profile of a 9-variable symmetric function.

7.2. Let $f(X)$ be

$$f = \left\lceil \frac{\sum_{i=0}^{4} 2^i x_i}{3} \right\rceil,$$

and let (X_1, X_2) be a partition of $X = (x_4, x_3, x_2, x_1, x_0)$. Write the decomposition chart when $X_1 = (x_2, x_1, x_0)$ and $X_2 = (x_4, x_3)$, and obtain the column multiplicity. Also, do the same thing for $X_1 = (x_3, x_2, x_1, x_0)$ and $X_2 = (x_4)$.

7.3. Let $X = (x_{n-1}, x_{n-2} \ldots, x_1, x_0)$ be the input variables. An **interval function** $IN(X : A, B)$ is defined as

$$IN(X : A, B) = \begin{cases} 1 & (\text{If } A \leq X \leq B) \\ 0 & (\text{Otherwise}). \end{cases}$$

Here, X is considered as an integer:

$$X = \sum_{i=0}^{n-1} 2^i x_i .$$

Show that the C-measure of the function is at most three.

7.4. Suppose that the given function f is represented as a sum of k products having two literals. Obtain an upper bound on $\mu(f)$.

7.5. Consider the function $f(X_1, X_2, X_3, X_4)$, where $X_1 = (x_1, x_2, x_3)$, $X_2 = (x_4, x_5)$, $X_3 = (x_6)$, and $X_4 = (x_7, x_8)$. Let f be partially symmetric with respect to X_i, where $i = 1, 2, 3, 4$.

1. Let (X_1, X_2) be the bound variables and (X_3, X_4) be the free variables. Then, obtain an upper bound on the column multiplicity of the decomposition.
2. Let (X_1, X_2, X_3) be the bound variables and (X_4) be the free variables. Then, obtain an upper bound on the column multiplicity of the decomposition.

7.6. Let (X_1, X_2, X_3) be a partition of the input variables X. Let μ_i be the column multiplicity of the decomposition of the function $f(X)$, where X_i is the set of bound variables $(i = 1, 2)$. Then, show that the column multiplicity of the decomposition of f, where (X_1, X_2) is the set of bound variables, is at most $\mu_1 \mu_2$.

7.7. A **priority encoder function** has n inputs $(x_1, x_2, \ldots, x_n)$ and $\lceil \log_2(n+1) \rceil$ outputs. If $x_1 = 1$, then the output is 1. Otherwise, if $x_i = 1$ and $x_j = 0$ for all j such that $(1 \leq j < i)$, then the output is i. If all the inputs are 0, then the output is 0. Show that the C-measure of the priority encoder function is $n + 1$.

7.8. Let $X = (x_0, x_1, x_2, x_3, x_4)$ and $W = (1, 2, 3, 4, 5)$. Consider the threshold function $f(X)$, where W is the weight vector and $T = 6$ is the threshold.

1. Write the decomposition chart, where $X_1 = (x_2, x_1, x_0)$ denotes the bound variables and $X_2 = (x_4, x_3)$ denotes the free variables.
2. Obtain the column multiplicity.

7.9. Realize the n-input modulo m function with k-LUT, where $n = 32$, $m = 17$, and $k = 10$.

7.10. Show that

$$2^{nk} \equiv (-1)^n \pmod{2^k + 1}$$
$$2^{nk} \equiv 1 \pmod{2^k - 1}.$$

7.11. Let $f(X)$ be a threshold function with the weight vector $(w_1, w_2, \ldots, w_n)$, where $w_i > 0$. Let (X_1, X_2) be a partition of X, where $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$. Then, show that the column multiplicity of f with respect to the decomposition (X_1, X_2) is at most $1 + \sum_{i=1}^{k} w_i$.

Chapter 8
C-Measure of Sparse Functions

Let f be a function of n variables. Then, the C-measure of f tends to increase exponentially with n. However, for the functions with a fixed weight u, ($u \ll 2^{n-1}$), C-measure increase as $O(n)$. Thus, functions with small weights have small C-measures. This chapter considers the C-measure of functions with small weights.

8.1 Logic Functions with Specified Weights

In this section, we derive upper bounds on C-measures for functions with small weights. Then, we derive the number of LUTs to realize functions with small weights.

Definition 8.1.1. The **weight** of a logic function f, denoted by u, is the number of the binary vectors $\vec{a}$ such that $f(\vec{a}) = 1$.

Let u be the weight of an n-variable function. If u is much smaller than 2^n, then f is a **sparse function**.

Theorem 8.1.1. *[8] Let $\mu_k(f)$ be the column multiplicity of a decomposition chart of an n-variable logic function f with k bound variables. Then,*

$$\mu_k(f) \leq \min\{2^k, 2^{2^{n-k}}\}.$$

Example 8.1.1. Table 8.1 shows the profiles

$$(\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6, \mu_7, \mu_8, \mu_9, \mu_{10})$$

of 10 randomly generated functions with $n = 10$ and $u = 512$. Since one-half of the truth table entries are 1, such functions are not considered to be sparse. The last row (AVG) denotes the average of the column multiplicities. The upper bound profile obtained by Theorem 8.1.1 is

$$(2, 4, 8, 16, 32, 64, 128, 16, 4, 2).$$

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_8,

Table 8.1 Profiles of 10-variable random functions with weight 512

NF	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}
f_0	2	4	8	16	32	64	98	16	4	2
f_1	2	4	8	16	32	64	106	16	4	2
f_2	2	4	8	16	32	64	99	16	4	2
f_3	2	4	8	16	32	64	100	16	4	2
f_4	2	4	8	16	32	64	97	16	4	2
f_5	2	4	8	16	32	64	102	16	4	2
f_6	2	4	8	16	32	64	97	16	4	2
f_7	2	4	8	16	32	64	95	16	4	2
f_8	2	4	8	16	32	64	98	16	4	2
f_9	2	4	8	16	32	64	104	16	4	2
AVG	2.0	4.0	8.0	16.0	32.0	64.0	99.6	16.0	4.0	2.0

Table 8.2 Profiles of 10-variable random functions with weight 64

NF	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}
f_0	2	4	8	16	26	31	21	12	4	2
f_1	2	4	8	16	28	27	22	10	4	2
f_2	2	4	8	16	26	30	20	10	4	2
f_3	2	4	8	16	27	30	19	11	4	2
f_4	2	4	8	16	27	28	21	9	4	2
f_5	2	4	8	16	27	30	19	8	4	2
f_6	2	4	8	16	29	30	19	8	3	2
f_7	2	4	8	16	29	28	19	10	4	2
f_8	2	4	8	16	28	29	20	9	4	2
f_9	2	4	8	16	28	31	18	9	4	2
AVG	2.0	4.0	8.0	16.0	27.5	29.4	19.8	9.6	3.9	2.0

Except for μ_7, the values of profiles in Table 8.1 are equal to the upper bounds given by Theorem 8.1.1. Note that

$$\mu(f_1) = \max_{k=1}^{10}\{\mu_k(f_1)\} = 106,$$

while

$$\mu(f_7) = \max_{k=1}^{10}\{\mu_k(f_7)\} = 95.$$

■

Example 8.1.2. Table 8.2 shows the profiles of 10 randomly generated functions with $n = 10$ and $u = 64$. Since the fraction of 1's in the truth table is 6.25%, such functions are considered to be sparse. In this case, the bound given by Theorems 8.1.1 is not tight for μ_5, μ_6, μ_7, and μ_8. ■

In general, when the weight u of a function is much smaller than 2^n, the bounds on the column multiplicity given by Theorem 8.1.1 are not tight and are not so useful. However, if the weight u of the function is given, then we can derive tighter bounds. From here, we derive a tighter bound using a combinatorial argument.

Lemma 8.1.1. *Consider boxes arranged as a rectangle with t rows and many columns. Assume that we distribute u nondistinct balls to these boxes so that each*

box has at most one ball. Let $\lambda(t,u)$ *be the maximum number of distinct column patterns. Then,*

$$\lambda(t,u) = \sum_{i=0}^{\zeta} \binom{t}{i} + r,$$

where ζ *is the integer satisfying the relation:*

$$\sum_{i=1}^{\zeta} i\binom{t}{i} \le u < \sum_{i=1}^{\zeta+1} i\binom{t}{i},$$

and

$$r = \left\lfloor \frac{u - \sum_{i=1}^{\zeta} i\binom{t}{i}}{\zeta + 1} \right\rfloor.$$

Note that $\lambda(t,u)$ is monotone increasing for $0 \le u \le t \cdot 2^{t-1}$ and takes the constant value 2^t when $u \ge t \cdot 2^{t-1}$.

Example 8.1.3. Consider boxes arranged as a rectangle with $t = 4$ rows and many columns. Assume that we distribute $u = 10$ nondistinct balls so that each box has at most one ball. When the balls are distributed as shown in Fig. 8.1, the maximum number of patterns occur. In this case, the first column has no ball; in the second to fifth columns, each column has just one ball; and in the last three columns, each column has two balls. The number of different column patterns can be enumerated as follows. Since,

$$\sum_{i=1}^{1} i\binom{4}{i} = 4 \le 10 < \sum_{i=1}^{2} i\binom{4}{i} = 16,$$

we have $\zeta = 1$, which shows that all the column patterns with weight 0 and weight 1 occur. Also,

$$r = \left\lfloor \frac{10 - \sum_{i=1}^{1} i\binom{4}{i}}{2} \right\rfloor = 3,$$

implies that there are three patterns with weight $\zeta + 1 = 2$. Thus, we have

$$\lambda(4,10) = \sum_{i=0}^{1} \binom{4}{i} + r = 1 + 4 + 3 = 8,$$

which shows the number of different column patterns. ∎

1	2	3	4	5	6	7	8
0	1	0	0	0	1	0	0
0	0	1	0	0	1	1	0
0	0	0	1	0	0	1	1
0	0	0	0	1	0	0	1

Fig. 8.1 Maximum number of column patterns for the function with weight $u = 10$

Table 8.3 Values for $\lambda(t,u)$

		u									
	t	23	47	55	95	111	147	191	223	239	375
μ_{n-2}	4	13	16	16	16	16	16	16	16	16	16
μ_{n-3}	8	**16**	28	**32**	47	52	**64**	79	90	94	**128**
μ_{n-4}	16	20	**32**	36	56	**64**	82	104	120	**128**	176
μ_{n-5}	32	24	40	44	**64**	72	90	112	**128**	136	204
μ_{n-6}	64	24	48	56	80	88	106	**128**	144	152	220

Example 8.1.4. Table 8.3 shows the values of $\lambda(t,u)$ for $t = 4, 8, 16, 32$, and various values of u. ■

Theorem 8.1.2. *Let $\mu_k(n,u)$ be the column multiplicity of a decomposition chart for an n-variable function with weight u and k bound variables. Then,*

$$\mu_k(n,u) \leq \lambda(2^{n-k},u).$$

Note that when $u \leq 2^{n-k}$, $\lambda(t,u) = u+1$, while when $u > 2^{n-k}$, $\lambda(2^{n-k},u) < u+1$. Thus, we have the following:

Corollary 8.1.1. *For any logic function f with weight u, $\mu(f) \leq u+1$.*

For $\mu_k(n,u)$, Theorem 8.1.2 gives better bounds than $2^{2^{n-k}}$ and Corollary 8.1.1 when

$$2^{n-k} < u < 2^{2^{n-k}+(n-k)-1}.$$

From Theorem 8.1.2, we have the following:

Theorem 8.1.3. *The number of 6-LUTs needed to realize an n-variable function with weight u is:*

- *10 or less, when $n = 9$ and $u \leq 55$*
- *15 or less, when $n = 10$ and $u \leq 47$*

Proof. The profile of a 9-variable function given by Theorem 8.1.1 is

$$(\mu_1,\mu_2,\mu_3,\mu_4,\mu_5,\mu_6,\mu_7,\mu_8,\mu_9)$$
$$= (2,4,8,16,32,64,16,4,2).$$

Let $n = 9$ and $u = 55$. By Theorem 8.1.2, we have $\mu_{n-3} = \mu_6 \leq \lambda(2^{n-6},55) = 32$. Thus, $\mu(f) \leq 32$, and by Theorem 5.2.1, f can be realized with at most 10 LUTs.

The profile of a 10-variable function given by Theorem 8.1.1 is

$$(\mu_1,\mu_2,\mu_3,\mu_4,\mu_5,\mu_6,\mu_7,\mu_8,\mu_9,\mu_{10})$$
$$= (2,4,8,16,32,64,128,16,4,2).$$

Let $n = 10$ and $u = 47$. By Theorem 8.1.2, we have $\mu_{n-3} = \mu_7 \leq \lambda(2^{n-7}, 47) = 28$, and $\mu_{n-4} = \mu_6 \leq \lambda(2^{n-6}, 47) = 32$. Thus, $\mu(f) \leq 32$, and by Theorem 5.2.1, f can be realized with at most 15 LUTs. □

Theorem 8.1.4. *The number of 7-LUTs needed to realize an n-variable function with weight u is:*

- *23 or less, when* $n = 12$ *and* $u \leq 95$
- *17 or less, when* $n = 11$ *and* $u \leq 111$
- *11 or less, when* $n = 10$ *and* $u \leq 147$

Proof. From Theorems 8.1.1 and 8.1.2, and Table 8.3, we have the following profiles for 7-LUTs:

When $u \leq 95$, $(2, 4, 8, 16, 32, 64, 64, 56, 47, 16, 4, 2)$.
When $u \leq 111$, $(2, 4, 8, 16, 32, 64, 64, 52, 16, 4, 2)$.
When $u \leq 147$, $(2, 4, 8, 16, 32, 64, 64, 16, 4, 2)$.

In a similar way to the proof of Theorem 8.1.3, we have the numbers of LUTs. □

Theorem 8.1.5. *The number of 8-LUTs needed to realize an n-variable function with weight u is:*

- *33 or less, when* $n = 14$ *and* $u \leq 191$
- *26 or less, when* $n = 13$ *and* $u \leq 223$
- *19 or less, when* $n = 12$ *and* $u \leq 239$
- *12 or less, when* $n = 11$ *and* $u \leq 375$

Proof. For 8-LUTs, we have the following profiles:

When $u \leq 119$, $(2, 4, 8, 16, 32, 64, 128, 128, 112, 104, 79, 16, 4, 2)$.
When $u \leq 223$, $(2, 4, 8, 16, 32, 64, 128, 128, 120, 90, 16, 4, 2)$.
When $u \leq 239$, $(2, 4, 8, 16, 32, 64, 128, 128, 94, 16, 4, 2)$.
When $u \leq 375$, $(2, 4, 8, 16, 32, 64, 128, 128, 16, 4, 2)$.

In a similar way to the proof of Theorem 8.1.3, we have the numbers of LUTs. □

Here, we compare the quality of bounds derived in this section with previous ones.

Theorem 8.1.6. *[78,110] The number of K-LUTs to realize an arbitrary n-variable function is at most*

$$2^{n-K+1} - 1.$$

When n is even, and $K = 6$, we have a better bound as follows:

Theorem 8.1.7. *[110] Let n be even. The number of 6-LUTs to realize an arbitrary n-variable function is at most*

$$\frac{2^{n-4} - 1}{3}.$$

Proof. This is the same as Theorem 4.2.3. □

Theorem 8.1.8. *[78] Let f be represented by a sum-of-products expression with m literals and p products. The number of K-LUTs to realize f is at most*

$$\left\lfloor \frac{m + p(K-3)}{K-1} \right\rfloor + \left\lceil \frac{p-1}{K-1} \right\rceil.$$

Example 8.1.5. Consider the case of $K = 6, n = 10$, and $u = 47$. The upper bound given by Theorem 8.1.7 is

$$\frac{2^{n-4} - 1}{3} = \frac{63}{3} = 21.$$

On the other hand, the bound given by Theorem 8.1.3 is 15.

When the function is random, we can assume that $p = u = 47$. Because the function is sparse and random, the minterms tend to occur as isolated minterms. Thus, p tends to be the same as $u = 47$ [105]. Thus, we have $m = pn = 47 \times 10$. In this case, the upper bound given by Theorem 8.1.8 is

$$\left\lfloor \frac{m + p(K-3)}{K-1} \right\rfloor + \left\lceil \frac{p-1}{K-1} \right\rceil = 122 + 10 = 132.$$

This example shows that the upper bound given by Theorem 8.1.8 is useless for random functions. ■

8.2 Uniformly Distributed Functions

In the previous section, we considered upper bounds on column multiplicities. In this section, however, we consider upper bounds on **the average column multiplicities**. These bounds are only valid when n and u are sufficiently large. It is assumed that 1's in the truth table occur randomly.

Definition 8.2.1. A set of functions is **uniformly distributed**, if the probability of occurrence of any function is the same as any other function.

For example, there are $\binom{16}{4} = 1820$ different 4-variable functions with 4 true minterms. If the functions are uniformly distributed, the probability of the occurrence of any one of them is $\frac{1}{1820}$.

Theorem 8.2.1. *Consider a decomposition chart with k bound variables that realizes a set of uniformly distributed functions of n-variables with weight u. The average number of different column functions with weight i in the decomposition chart is at most*

$$\min\{1, N\alpha^i \beta^{M-i}\} \times \binom{M}{i},$$

where $N = 2^k, M = 2^{n-k}, \alpha = u/2^n$, and $\beta = 1 - \alpha$.

Proof. Consider a column of the decomposition chart. Suppose that the upper i elements take value 1, while the lower $M-i$ elements take value 0. The probability of such a column is given by

$$\alpha^i \beta^{M-i}.$$

Since there exist $\binom{M}{i}$ different ways to choose the rows with 1, the probability that a column function with weight i occurs is

$$\binom{M}{i} \alpha^i \beta^{M-i}.$$

Also, the number of different column functions with weight i is at most $\binom{M}{i}$. Thus, we have the theorem. □

Note that Theorem 8.2.1 gives upper bounds on the **average** column multiplicity μ_k. Thus, there may exist functions whose column multiplicities are greater than the bounds given by Theorem 8.2.1. However, the fraction of such functions approaches to zero as n increases.

8.3 Experimental Results

We developed a program to derive the bounds on the column multiplicities given by Theorems 8.1.1 and 8.2.1. Also, we have obtained statistical data for functions with $n = 10$ and $n = 16$.

8.3.1 Benchmark Functions

An interesting problem is whether the bounds obtained in Sect. 8.1 are applicable to benchmark functions. The answer is yes when the weights of benchmark functions are in a range. The fraction of such functions is not so large, but we did find some. For selected benchmark functions, we counted the number of variables n, and the number of true minterms u. For multiple output functions, each output is examined separately. f_i denotes the i-th output, where the index starts from 0. The results are as follows:

Theorem 8.1.3 is applicable to the following benchmark functions: apex4, f_1 ($n = 9, u = 55$); pdc, f_{23} ($n = 9, u = 43$); spla, f_{43} ($n = 10, u = 28$); spla, f_{44} ($n = 10, u = 44$); amd, f_{14} ($n = 10, u = 44$).

Theorem 8.1.4 is applicable to the following benchmark functions: pdc, f_9 ($n = 10, u = 95$); signet, f_3 ($n = 10, u = 95$); pdf, f_{31}, ($n = 10, u = 120$).

Theorem 8.1.5 is applicable to the following benchmark functions: pdc, f_8 ($n = 11, u = 333$); pdc, f_{33} ($n = 11, u = 340$); signet, f_4 ($n = 11, u = 132$); in2, f_3 ($n = 12, u = 236$); ti, f_5 ($n = 13, u = 160$).

8.3.2 Randomly Generated Functions

8.3.2.1 10-Variable Random Functions

First, we randomly generated 100 functions with $u = 512$ and $n = 10$. Table 8.4 shows the average column multiplicities (AVG), the maximum column multiplicities (MAX), the upper bound derived by Theorem 8.2.1, and the upper bound derived by Theorem 8.1.1 for each μ_k. In this case, except for μ_6 and μ_7, Theorem 8.1.1 gives tight bounds. Also, Theorems 8.1.1 and 8.2.1 give identical bounds.

Second, we randomly generated 10,000 functions with $u = 64$ and $n = 10$. Table 8.5 shows the average column multiplicities (AVG), the maximum column multiplicities (MAX), the upper bounds derived by Theorem 8.2.1, and the upper bounds derived by Theorem 8.1.1 for each μ_k. In this case, for μ_k, where $k = 5, 6, 7, 8$, Theorem 8.2.1 gives better bounds than Theorem 8.1.1.

Table 8.5 also shows that the average C-measure of 10-variable functions with weight 64, is less than 30. The maximum column multiplicity occurs when $k = 6$ and $\mu_6 = 37$. In these experimental results, the order of the input variables is fixed. If we optimize the order of the variables, then we can reduce the column multiplicity. We also optimized the order of the variables and confirmed that, in all 10,000 cases, the column multiplicities are at most 32. Thus, we have the following:

Conjecture 8.3.1. For most 10-variable functions with weight $u \leq 64$, the C-measure is 32 or less, if we optimize the ordering of the input variables.

From Theorem 5.2.1, we have

Conjecture 8.3.2. The number of 6-LUTs needed to realize most of 10-variable functions with weight $u \leq 64$ is 15 or less.

As shown in Theorem 8.1.3, the number of 6-LUTs needed to realize an arbitrary 10-variable function f with weight $u \leq 47$ is 15 or less. Thus, Conjecture 8.3.2 is true for $u \leq 47$.

Table 8.4 Average and maximum profiles of 10-variable functions with weight 512

	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}
AVG	2	4	8	16.00	32.00	63.96	100.86	16	4	2
MAX	2	4	8	16	32	64	111	16	4	2
Theorem 8.2.1	2	4	8	16	32	64	128	16	4	2
Theorem 8.1.1	2	4	8	16	32	64	128	16	4	2

Table 8.5 Average and maximum profiles of 10-variable functions with weight 64

	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}
AVG	2	4	8	15.97	27.97	29.89	18.36	8.76	3.88	2
MAX	2	4	8	16	32	37	25	13	4	2
Theorem 8.2.1	2	4	8	16	27.44	33.75	19.88	10.51	4	2
Theorem 8.1.1	2	4	8	16	32	64	128	16	4	2

Table 8.6 Profiles for 16-variable functions

u	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}	μ_{11}	μ_{12}	μ_{13}	μ_{14}	μ_{15}
A16	7.73	11.30	13.73	15.24	15.95	16.20	16.05	15.42	13.90	11.23	8.11	4.97	3
C16	7.50	11.06	13.56	15.16	16.04	16.51	16.75	16.88	16.94	16.97	9.01	5.01	3
A32	8	14.79	21.52	26.37	28.94	29.64	28.83	26.28	21.46	15.06	8.93	5.02	3.00
C32	8	13.99	21.11	26.18	29.32	31.09	32.03	32.51	32.76	17.12	9.05	5.02	3.01
A64	8	15.97	28.69	41.48	50.10	53.10	50.52	42.05	29.29	17.19	9.23	5.07	3.00
C64	8	16	26.99	41.23	51.36	57.65	61.19	63.07	33.95	17.48	9.22	5.09	3.03
A128	8	16	31.84	56.42	80.18	91.11	83.70	60.99	35.79	18.69	9.57	5.23	3.05
C128	8	16	32	53.00	81.49	101.76	114.35	72.27	36.73	18.84	9.87	5.37	3.13
A256	8	16	32	65.53	110.64	147.29	136.88	88.38	46.95	23.92	12.27	6.32	3.44
C256	8	16	32	63.84	105.05	162.09	174.95	91.84	47.34	24.23	12.45	6.49	3.50
A512	8	16	32	64	126.57	214.49	235.14	155.10	83.05	41.66	19.59	8.69	3.85
C512	8	16	32	64	126.69	209.33	262.51	156.60	86.08	44.89	22.57	10.94	4
A1K	8	16	32	64	128	251.86	391.99	327.91	192.81	89.33	33.93	11.07	4
C1K	8	16	32	64	128	252.46	407.99	332.19	214.99	120.75	38.65	11.25	4
A2K	8	16	32	64	128	256	499.80	658.92	455.91	176.87	48.40	12.72	4
C2K	8	16	32	64	128	256	504.20	626.67	568.03	188.19	49.43	12.95	4
A4K	8	16	32	64	128	256	512	975.44	1019.90	393.73	86.27	15.10	4
C4K	8	16	32	64	128	256	512	1008.54	1095.05	431.43	99.76	15.25	4
A8K	8	16	32	64	128	256	512	1023.92	1834.20	1055.60	150.52	15.98	4
C8K	8	16	32	64	128	256	512	1024	2020.45	1063.58	163.00	16	4
A16K	8	16	32	64	128	256	512	1024	2047.48	2738.27	238.61	16	4
C16K	8	16	32	64	128	256	512	1024	2048	4055.95	256	16	4
A32K	8	16	32	64	128	256	512	1024	2048	3971.75	256	16	4
C32K	8	16	32	64	128	256	512	1024	2048	4096	256	16	4

8.3.2.2 16-Variable Random Functions

In the case of $n = 16$, for each weight $u = 2^i$, where $i = 4$–15, we generated 100 functions and obtained the average of profiles.

Table 8.6 compares the profiles of random functions and the average profiles derived by Theorem 8.2.1. For example, the row for $A16$ denotes the average profile of randomly generated functions with weight 16, while the row for C16 denotes the calculated profile using Theorem 8.2.1. To save space, some fractional numbers are denoted by integers: 2.00 is denoted by 2. Also, the values for μ_1, μ_2 and μ_{16} are omitted from the table, since $\mu_1 = 2$, $\mu_2 = 4$, and $\mu_{16} = 2$, for all cases.

Table 8.6, shows that when u is small (say $u = 16$), Theorem 8.2.1 gives better bounds than Theorem 8.1.1, while when u is large (say $u = 32,768$), Theorem 8.1.1 gives better bounds. In Table 8.6, calculated bounds denoted by integers were obtained by Theorem 8.1.1, while calculated bounds denoted by fractional numbers were obtained by Theorem 8.2.1.

8.4 Remarks

In this chapter, we considered sparse functions, where u, the weight of the function, is much smaller than 2^{n-1}. However, when 0 and 1 are interchanged, the theory also holds. Thus, these bounds are also useful for the functions where the fraction of 0's in the truth table is much smaller than the fraction of 1's. This chapter is based on [133].

Problems

8.1. Obtain the value of $\lambda(8, 20)$.

8.2. Suppose that a function is given as a sum-of-products expression. Show an efficient method to calculate the weigh for f.

8.3. Show a $(2^n + n)$-variable function whose C-measure is 2^{2^n}.

8.4. How many 6-input LUTs are necessary to realize

1. An arbitrary 8-variable function?
2. An arbitrary 9-variable symmetric function?
3. An arbitrary 10-variable function?
4. An arbitrary 10-variable function with weight 47?

Chapter 9
Index Generation Functions

This chapter introduces index generation functions with various applications. Then, it shows a method to realize index generation functions by LUTs.

9.1 Index Generation Functions and Their Realizations

Definition 9.1.1. Consider a set of k different binary vectors of n bits. These vectors are **registered vectors**. For each registered vector, assign a unique integer from 1 to k. A **registered vector table** shows, for each registered vector, its **index**. An **index generation function** produces the corresponding index if the input matches a registered vector, and produces 0 otherwise. k is the **weight** of the index generation function. An index generation function represents a mapping: $B^n \rightarrow \{0, 1, 2, \ldots, k\}$. An **index generator** is a circuit that realizes an index generation function.

Example 9.1.1. Table 9.1 shows a registered vector table with $k = 4$ vectors. The corresponding index generation function is shown in Table 9.2. ■

Here, we assume that k is much smaller than 2^n, the total number of input combinations, i.e., the index generation function is sparse.

Index generators are used in address tables for the internet routers, terminal access controller (TAC) for local area networks, databases, memory patch circuits, electronic dictionaries, password lists, etc.

9.2 Address Table

IP addresses used in the internet are often represented with 32-bit numbers. An **address table** for a router stores IP addresses and corresponding indices. We assume that the number of addresses in the table is at most 40,000. Thus, the number of inputs is 32 and the number of outputs is 16, which can handle 65,536 addresses. Note that the address table must be updated frequently.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_9,

Table 9.1 Registered vector table

Vector				
x_1	x_2	x_3	x_4	Index
1	0	0	1	**1**
1	1	1	1	**2**
0	1	0	1	**3**
1	1	0	0	**4**

Table 9.2 Index generation function

Input				Output		
x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	**0**	**1**	**1**
0	1	0	1	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	0	1	**0**	**0**	**1**
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	**1**	**0**	**0**
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	**0**	**1**	**0**

9.3 Terminal Access Controller

A TAC for a local area network checks whether the requested terminal has permission to access Web addresses outsize the local area network, e-mail, FTP, Telnet, etc.

In Fig. 9.1, eight terminals are connected to the TAC. Some can access all the resources. Others can access only limited resources because of security issue. The TAC checks whether the requested computer has permission to access the Web, e-mail, FTP, Telnet, or not. Each terminal has its unique **MAC address** represented by 48 bits. We assume that the number of terminals in the table is at most 255. To implement the TAC, we use an index generator and a memory. The memory stores the details of the terminals. The number of inputs for the index generator is 48 and the number of outputs is 8. Note that the table for the TAC must be updated frequently.

Example 9.3.1. Figure 9.2 shows an example of the TAC. The first terminal has the MAC address 53:03:74:59:03:02. It is allowed to access everything, including the Web outside the local area network, e-mail, FTP, and Telnet. The second one is allowed to access both the Web and e-mail. The third one is allowed to access only the Web. And, the last one is allowed to access only e-mail. The index generated

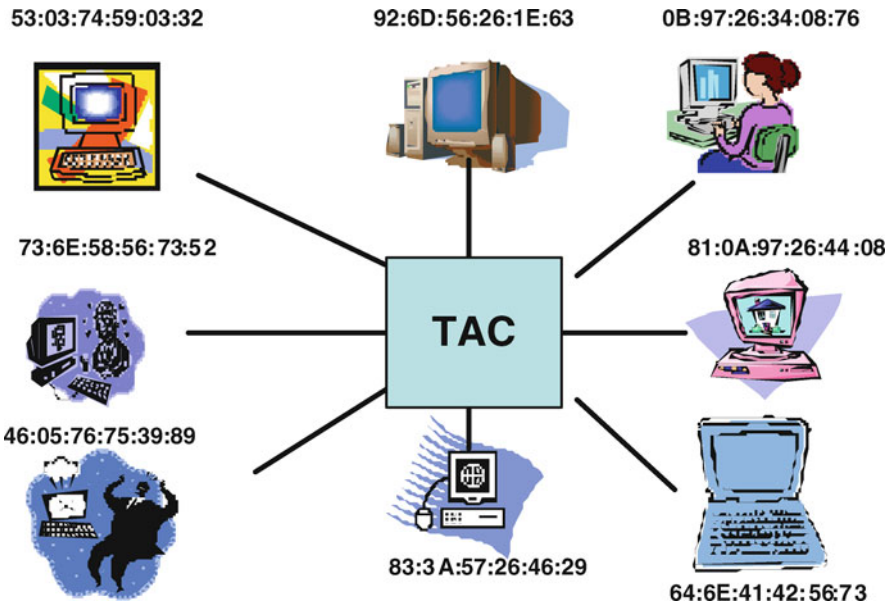


Fig. 9.1 Terminal access controller (TAC)

Index Generator	
MAC Address	Index
53:03:74:59:03:32	1
92:6D:56:26:1E:63	2
0B:97:26:34:08:76	3
73:6E:58:56:73:52	4

48-bit

⇒

Memory				
Address	DATA			
	Web	E-mail	FTP	Telnet
1	1	1	1	1
2	1	1	0	0
3	1	0	0	0
4	0	1	0	0

4-bit

Fig. 9.2 Index generator for TAC

by the index generator is used as an address to read the memory which stores the permissions. If we implement the TAC by a single memory, we need 256 Tera words, since the number of inputs is 48. To reduce the size of the memory, we use an index generator to produce the index, and an additional memory to store the permission data for each internal address. ■

9.4 Memory Patch Circuit

The firmware of an embedded system is usually implemented by Read-Only Memories (ROMs). After shipping the product, it is often necessary to modify a part of the ROM, for example to upgrade to a later version. To convert the address of the ROM

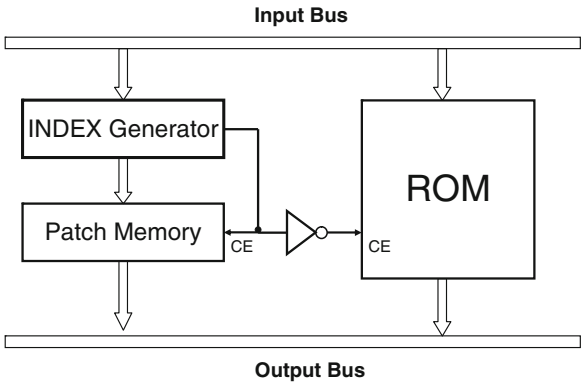


Fig. 9.3 Memory patch circuit

to the address of the patch memory, we use the index generator shown in Fig. 9.3 [28, 29, 74].

The index generator stores addresses (vectors) of the ROM to be updated, and their corresponding indices. The **patch memory** stores the updated data of the ROM. When the address does not match any elements in the index generation function, the output of the ROM is sent to the output bus. In this case, the output the patch memory is disabled. When the address matches to an element in the index generation function, the index generator produces the corresponding index, and the corresponding data of the patch memory is sent to the output bus. In this case, the output of the ROM is disabled. This method can be also used to improve the yield of large-scale memory, which can be "patched" instead of discarded.

9.5 Periodic Table of the Chemical Elements

Consider Table 9.3, which shows a part of periodic table of the chemical elements. Figure 9.4 shows its implementation. This stores, *Atomic Number* (integer), *Chemical Symbol*, *Density*, *Specific Heat*, *State*, and *Category*. We assume that the number of elements in the database is at most 127. The database consists of two circuits:

1. A circuit to produce the *Atomic Number* from a *Chemical Symbol*.
2. A circuit to produce *Density*, *State* and *Category* from an *Atomic Number*.

The first circuit is implemented by an index generator, and the second circuit is implemented by an ordinary memory. *Chemical Symbol* consists of two characters from the 26-letter English alphabet (uppercase and lowercase) and special symbols (blank, etc.). Since each character requires 6 bits, to represent 2 characters, we need $2 \times 6 = 12$ bits. On the other hand, to represent an *Atomic Number*, we need only 7 bits, since the total number of elements in the table is at most 127. In this case,

Table 9.3 Periodic table of the chemical elements

Atomic number	Chemical symbol	English name	Density	Specific heat	State	Category
1	H	Hydrogen	0.082	14.304	Gas	Other nonmetal
2	He	Helium	0.147	5.188	Gas	Noble gas
3	Li	Lithium	0.534	3.489	Solid	Alkali metal
4	Be	Beryllium	1.848	1.824	Solid	Alkaline earth metal
5	B	Boron	2.34	1.025	Solid	Metalloid
6	C	Carbon	3.51	0.513	Solid	Other nonmetal
7	N	Nitrogen	1.2506	1.042	Gas	Other nonmetal
8	O	Oxygen	1.429	0.916	Gas	Other nonmetal
9	F	Fluorine	1.696	0.824	Gas	Halogen
10	Ne	Neon	0.8999	1.029	Gas	Noble gas
11	Na	Sodium	0.971	1.227	Solid	Alkali metal
12	Mg	Magnesium	1.738	1.025	Solid	Alkaline earth metal
13	Al	Aluminum	2.6989	0.902	Solid	Other metal
14	Si	Silicon	2.33	0.712	Solid	Metalloid
15	P	Phosphorus	1.82	0.757	Solid	Other nonmetal
16	S	Sulfur	2.07	0.732	Solid	Other nonmetal
17	Cl	Chlorine	3.214	0.477	Gas	Halogen
18	Ar	Argon	1.848	0.138	Gas	Noble gas
19	K	Potassium	0.862	0.766	Solid	Alkali metal
20	Ca	Calcium	1.55	0.653	Solid	Alkaline earth metal

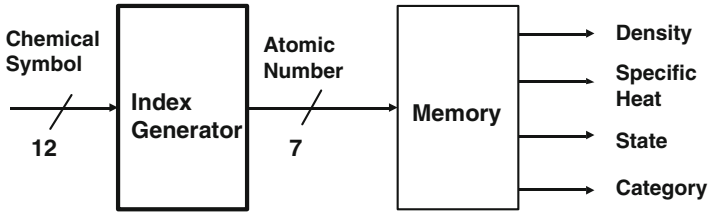


Fig. 9.4 Circuit for the periodic table

the number of possible input combinations is 2^{12}, while the number of the registered vectors is at most 127. Note that each chemical element has its unique atomic number.

9.6 English–Japanese Dictionary

For simple English–Japanese communication, we prepare a dictionary consisting of 1,500 English words. To make a list of 1,500 English words using a single memory or a single circuit is unrealistic. Therefore, we partition the list into three groups, so that each list contains at most 500 words. Let the names of the three lists be *Word list A*, *Word list B*, and *Word list C*. The maximum number of letters in the word

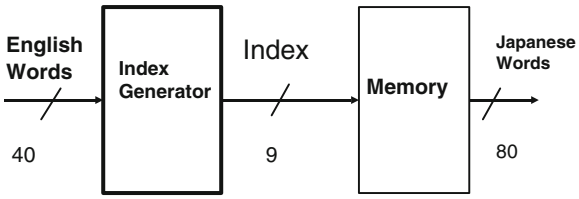


Fig. 9.5 Implementation of English–Japanese dictionary

lists is 13, but we only consider the first 8 letters. For English words consisting of fewer than 8 letters, we append blanks to make the length of words 8. We represent each alphabetic character by 5 bits. So, all the English words are represented by 40 bits. We assume that each group has at most 500 English words, and each word has unique address from 1 to 500. The address is represented by 9 bits.

Figure 9.5 shows the English–Japanese dictionary consisting of the index generator and a memory. In this dictionary, the index generator finds the index for the English word, and the memory produces the Japanese translation. Note that, in Japanese, 80 outputs are needed to represent the Chinese characters and *KANA characters*.

9.7 Properties of Index Generation Functions

The index generators in Sects. 9.2 and 9.3 have common properties:

1. The values of the nonzero outputs are distinct.
2. The number of nonzero output values is much smaller than the total number of the input combinations.
3. High-speed circuits are required.
4. Data must be updated.

The last condition is very important in communication networks. This means that index generators must be programmable.

Example 9.7.1. Consider the decomposition chart in Fig. 9.6. It shows an input index generation function $F(X)$ with weight 7. $X_1 = (x_1, x_2, x_3, x_4)$ denotes the bound variables, and $X_2 = (x_5)$ denotes the free variable. Note that the column multiplicity of this decomposition chart is 7. ■

Lemma 9.7.1. *The C-measure of an index generation function with weight k is at most $k + 1$.*

Proof. Since the number of nonzero outputs is k, the column multiplicity never exceeds $k + 1$. □

	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_1
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_2
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_3
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_4
0	0	1	0	0	3	0	4	5	0	0	0	0	0	0	0	0	
1	0	0	2	0	0	0	0	6	0	0	0	0	7	0	0	0	
x_5																	

Fig. 9.6 Decomposition chart for F

Lemma 9.7.2. *Let F be an index generation function with weight k. Then, there exists a functional decomposition*

$$F(X_1, X_2) = G(H(X_1), X_2),$$

where G and H are index generation functions, and the weight of G is k, and the weight of H is at most k.

Proof. Consider a decomposition chart, in which X_1 denotes the bound variables, and X_2 denotes the free variables. Let $X_1 = (x_1, x_2, \ldots, x_p)$, where $p \geq \lceil \log_2(k+1) \rceil$. Let H be a function where the input variables are X_1, and the output values are defined as follows: Consider the decomposition chart, where assignments of values to X_1 label columns (i.e., bound variables). For the assignments to X_1 corresponding to columns with only zero elements, $H = 0$. For other inputs, the outputs are distinct integers from 1 to w_h, where w_h denotes the number of columns that have nonzero element(s). Since $w_h \leq k$, the weight of H is at most k, and the number of output values of H is at most $k+1$. On the other hand, the function G is obtained from F by reducing some columns that have all zero outputs in the decomposition chart. Thus, the number of nonzero outputs in G is equal to the number of nonzero outputs in F. Thus, G is also an index generation function with weight k. □

Example 9.7.2. Consider the decomposition chart in Fig. 9.6. Let the function $F(X)$ be decomposed as $F(X_1, X_2) = G(H(X_1), X_2)$, where $X_1 = (x_1, x_2, x_3, x_4)$ and $X_2 = (x_5)$. Table 9.4 shows the function H. It is a 4-variable 3-output index generation logic function with weight 6. The decomposition chart for the function G is shown in Fig. 9.7. As shown in this example, the functions obtained by decomposing the index generation function F are also index generation functions, and the weights of F and G are both 7. ■

Table 9.4 Truth table for H

x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	0	0
0	1	0	0	0	1	1
0	1	0	1	0	0	0
0	1	1	0	1	0	0
0	1	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

Fig. 9.7 Decomposition chart for G

	0	0	0	0	1	1	1	1	y_1
	0	0	1	1	0	0	1	1	y_2
	0	1	0	1	0	1	0	1	y_3
0	0	1	0	3	4	5	0	0	
1	0	0	2	0	0	6	7	0	
x_5									

9.8 Realization Using (p, q)-Elements

Definition 9.8.1. A **(p, q)-element** realizes an arbitrary p-input q-output logic function. Its **memory size** is $q2^p$.

Theorem 9.8.1. *An arbitrary two-valued input n-variable index generation function with weight k can be realized as a multilevel network of (p, q)-elements. The number of such elements is at most $\left\lceil \frac{n-q}{p-q} \right\rceil$, where $p > q$ and $q = \lceil \log_2(k+1) \rceil$.*

Proof. An index generation logic function F with weight k can be decomposed as

$$F(X_1, X_2) = G(H(X_1), X_2),$$

where $X_1 = (x_1, x_2, \ldots, x_p)$. In this case, by Lemma 9.7.2, $G(X'_1, X_2)$ is also an index generation logic function with weight k. Note that the number of input variables for G is reduced to $n-(p-q)$, since the number of output variables of H is $q = \lceil \log_2(k+1) \rceil$. By iterating this operation $\left\lceil \frac{n-p}{p-q} \right\rceil$ times, we can reduce the number of variables to p or fewer. Thus, the index generator can be realized by using only (p, q)-elements. The number of elements is at most $\left\lceil \frac{n-p}{p-q} \right\rceil + 1 = \left\lceil \frac{n-q}{p-q} \right\rceil$. □

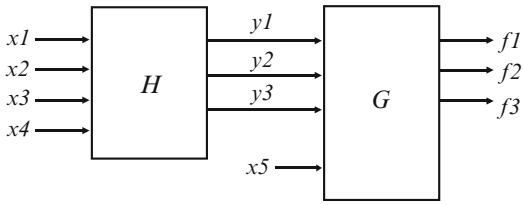


Fig. 9.8 Realization of index generation function F

Example 9.8.1. The number of nonzero outputs in the 5-variable index generation function $F(X)$ shown in Fig. 9.6 is $k = 7$. Since $q = \lceil \log_2(k+1) \rceil = \lceil \log_2(7+1) \rceil = 3$, the index generator can be realized by two $(4, 3)$ elements as shown in Fig. 9.8. ■

When realizing an index generator by (p, q)-elements, increasing p decreases the number of (p, q)-elements, but increases the total amount of memory. On the other hand, decreasing p increases the number of (p, q)-elements, but decreases the total amount of memory. The next theorem shows a strategy to design index generators using (p, q)-elements. It finds a value of p that minimizes the least upper bound on the total amount of memory without increasing the number of elements.

Theorem 9.8.2. *When an index generator is implemented as a multilevel network of (p, q)-elements, the least upper bound on the total amount of memory is minimized when $p - q = 1$ or $p - q = 2$.*

Proof. When an index generation function is decomposed into (p, q)-elements, for each decomposition, we can reduce the number of input variables by $r = p - q$. To reduce n inputs into q, we need $s = \left\lceil \frac{n-q}{r} \right\rceil$ functional decompositions. To realize the index generator, we need s (p, q)-elements. Thus, the total amount of memory necessary to implement the index generator is $MEM = s \cdot 2^p q$. When n is sufficiently large, MEM can be approximated by $\left(\frac{2^r}{r}\right) \cdot (n-q) \cdot 2^q q$. Since n and q are fixed for a given problem, only r can be changed. Note that $\frac{2^r}{r}$ takes its minimum when $r = 1$ or $r = 2$. Hence we have the theorem. □

Since networks with fewer levels are desirable, we often select $r = p - q = 2$ to design the index generator.

Theorem 9.8.1 shows that we can design an index generator as a multilevel network of (p, q)-elements by iterations of functional decompositions.

The next Example 9.8.2 shows that we can generate various multilevel logic networks, including cascades.

Example 9.8.2. Let us design index generators, where the number of inputs is $n = 48$ and the weight is $k = 255$. Since $q = \lceil \log_2(255+1) \rceil = 8$, when $p = 10$, the total amount of memory is minimized, and also the number of levels is minimized. For each (p, q)-element, we can reduce the number of input lines by two. So, by using 20 (p, q)-elements, we can reduce the number of inputs into

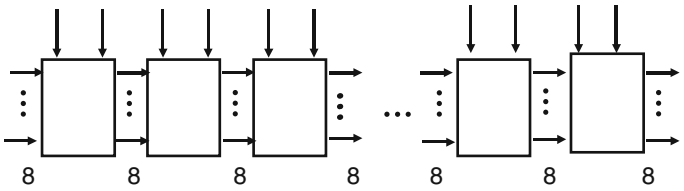


Fig. 9.9 Cascade realization of index generator

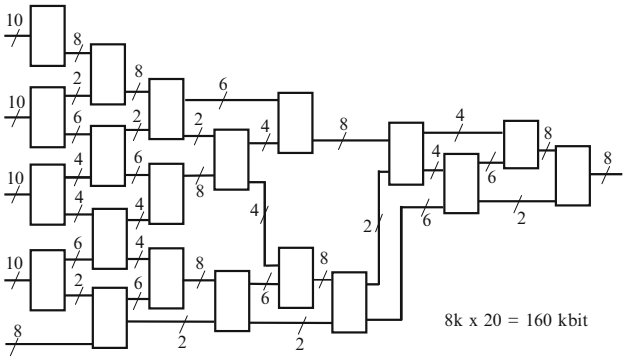


Fig. 9.10 Index generator ($p = 10$)

8. For example, we have the LUT cascade as shown in Fig. 9.9. Or, we have the multilevel logic network as shown in Fig. 9.10, where the number of levels is 10. In this case, the variables are permuted during functional decompositions. Note that both structures require the same amount of memory: 160K bits. We can further reduce the number of levels by using elements with more inputs. Figure 9.11 shows an example with $p = 11$ and $q = 8$. In this case, the number of elements is $(48-8)/(11-8) = 14$, the number of levels is 8, and the total amount of memory is 212 Kibits, where 1 Kibit denotes $2^{10} = 1024$ bits. Figure 9.12 show an example with $p = 12$ and $q = 8$. In this case, the number of elements is $(48-8)/(12-8) = 10$, the number of levels is 5, and the total amount of memory is 320 Kibits. ■

Theorem 9.8.2 shows the strategy for general index generators. It minimizes the least upper bound on the total amount of memory. For a particular index generator, the total amount of memory can be minimum for cases other than $p - q = 2$. The next example illustrates this.

Example 9.8.3. Consider the 6-variable index generation function $F(X)$ shown in Table 9.5. Let the function $F(X)$ be decomposed as $F(X_1, X_2) = G(H(X_1), X_2)$, where $X_1 = (x_1, x_2, x_3, x_4)$ and $X_2 = (x_5, x_6)$. The column multiplicity of the decomposition chart in Table 9.5 is 2. Table 9.6 is the truth table of H, and Table 9.7 is the truth table of G. This index generator can be implemented as Fig. 9.13. In this case, the weight of the function is $k = 7$, but H is realized by a (4,1)-element. ■

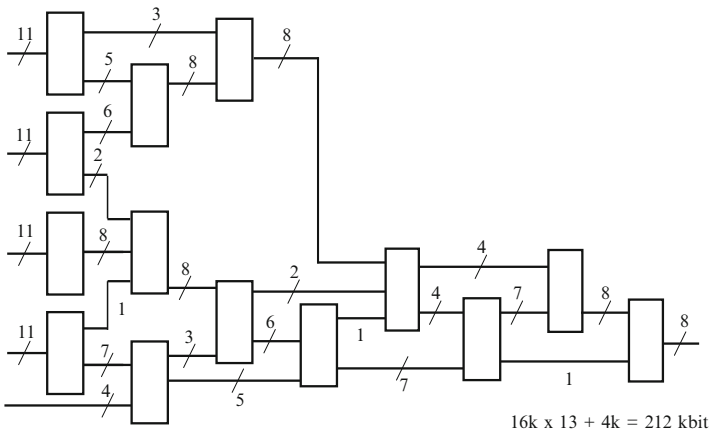


Fig. 9.11 Index generator ($p = 11$)

Fig. 9.12 Index generator ($p = 12$)

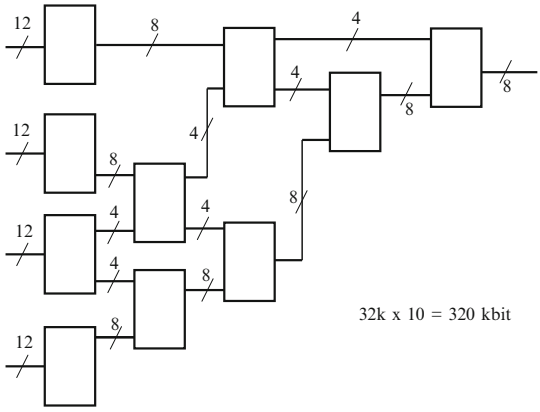


Table 9.5 Index generation function F

x_1	x_2	x_3	x_4	x_5	x_6	f_1	f_2	f_3
0	–	–	–	–	–	0	0	0
–	0	–	–	–	–	0	0	0
–	–	0	–	–	–	0	0	0
–	–	–	0	–	–	0	0	0
1	1	1	1	0	0	0	0	1
1	1	1	1	0	1	0	1	0
1	1	1	1	1	0	0	1	1
1	1	1	1	1	1	1	0	0

9.9 Realization of Logic Functions with Weight *k*

Up to now, we have considered the realization of index generation functions. Next, we consider the realization of general logic functions.

Table 9.6 Truth table for H

x_1	x_2	x_3	x_4	y_1
1	1	1	1	1
–	0	0	0	0
0	–	0	0	0
0	0	–	0	0
0	0	0	–	0

Table 9.7 Truth table for G

y_1	x_5	x_6	f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	1
1	0	1	0	1	0
1	1	0	0	1	1
1	1	1	1	0	0

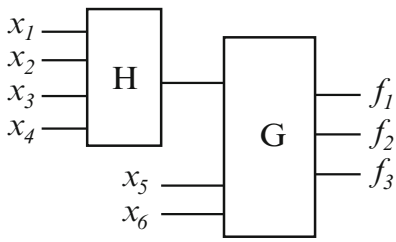


Fig. 9.13 Index generator for Table 9.5

Theorem 9.9.1. *An arbitrary two-valued n-variable u-output function with weight k is realized as a multilevel network of (p, q)-elements. The number of elements needed is at most $\left\lceil \frac{n-q}{p-q} \right\rceil + \left\lceil \frac{u}{q} \right\rceil$, where $p > q$ and $q = \lceil \log_2(k+1) \rceil$.*

Proof. An arbitrary logic function with weight k can be realized as a cascade of an index generator and a decoder, where the index generator produces unique indices for k input combinations, and the decoder converts each index into corresponding outputs. The number of inputs of the decoder is at most $\lceil \log_2(k+1) \rceil$. By Theorem 9.8.1, the index generator can be realized with $s_1 = \left\lceil \frac{n-q}{p-q} \right\rceil$ elements. Also, note that the decoder can be realized by s_2 modules of q-input q-output elements, where s_2 is given by $s_2 = \left\lceil \frac{u}{q} \right\rceil$. Thus, total number of elements is $s_1 + s_2 = \left\lceil \frac{n-q}{p-q} \right\rceil + \left\lceil \frac{u}{q} \right\rceil$. □

Corollary 9.9.1. *An arbitrary two-valued n-variable single-output logic function with weight k is realized as a multilevel network of (p, q)-elements. The number of elements needed is at most $\left\lceil \frac{n-q}{p-q} \right\rceil$, where $p > q$ and $q = \lceil \log_2(k+1) \rceil$.*

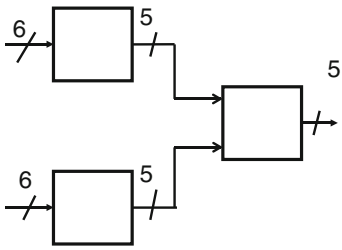


Fig. 9.14 Index generator with weight $k \leq 32$

9.10 Remarks

Index generation functions are multiple-output logic functions useful for pattern matching and communication circuits. Logic synthesis for index generation functions is considered in Chaps. 10 and 11. This chapter is based on [124].

Problems

9.1. Design an index generator where the number of inputs is $n = 32$, and the weight is $k = 63$. Use 8-input 6-output LUTs. Show both the cascade and minimum-delay realizations.

9.2. An index generation function can be directly implemented by a PLA. Discuss the advantage and disadvantage to implement an index generation function using memory instead of a PLA. Consider the case of $n = 48$ and $k = 400$.

9.3. When $k \leq 32$, an arbitrary index generation function of 12 variables with weight k can be realized by the circuit structure shown in Fig. 9.14. Compare this realization with an LUT cascade realization with respect to the size and speed. Assume that we use 6-LUTs.

9.4. Design an index generator for $n = 16$ and $k = 3$ by using $(4, 2)$-elements.

Chapter 10
Hash-Based Synthesis

In Chap. 5, we considered cascade-based realizations of logic functions with small C-measure. When the weight k of the function satisfies the relation $\lceil \log_2(k+1) \rceil < K$, the function can be efficiently realized by a cascade of K-input cells. Also, as shown in Chap. 9, an index generation function can be implemented by a multilevel network of (p, q)-elements, or by a multilevel network with K-LUTs. However, when $\lceil \log_2(k+1) \rceil \geq K$, such methods are not directly applicable.

This chapter presents the **hybrid method**, the **super hybrid method**, and the **parallel sieve method**. These methods efficiently implement index generation functions using memories. They are particularly suitable for FPGA realizations, since most FPGAs have both LUTs and embedded memories inside. These methods use pairs of smaller memories to implement most of the registered vectors.

10.1 Hash Function

Hash functions are often used in software implementations. To show the idea, consider the following:

Example 10.1.1. Assume that one needs to find a name of an employee from his or her 10-digit telephone number, in a company with 5,000 employees. A straightforward method to do this is to build a complete table of 10-digit telephone numbers showing the names of the employees. However, this method is unrealistic, since the table has 10^{10} entries, most of which are empty. To avoid such a problem, a **hash table** can be used. Let x be the telephone number, and consider the **hash function**[1]:

$$\text{hash}(x) = x \pmod{9973}.$$

In this case, the name of the employee can be found from the hash table with 9973 entries, since the value of $hash(x)$ is between 0 and 9972. When two or more

[1] 9,973 is the largest prime number less than 10,000.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_10,

Table 10.1 Registered vector table

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

Table 10.2 Example of an index generation function

x_1	x_2	x_3	x_4	x_5	x_6	f
0	0	0	0	1	0	1
0	1	0	0	1	0	2
0	0	1	0	1	0	3
0	0	1	1	1	0	4
0	0	0	0	0	1	5
1	1	1	0	1	1	6
0	1	0	1	1	1	7

different employees have the same hash value, a **collision** occurs. In such a case, the employees with the same hash value are represented by a **linked list**. Note that using a hash table, the number of digits for table look-up is reduced from 10 to 4. ■

This chapter shows a hash method for hardware implementation. With this method, the number of variables can be reduced and the size of memories can be also reduced.

Besides index generation functions, this design method can implement an n-variable function where the number of nonzero outputs k is much smaller than 2^n.

Example 10.1.2. Table 10.1 shows a registered vector table consisting of 7 vectors. The corresponding index generation logic function shown in Table 10.2 produces a 3-bit number (e.g., 001) of the matched vector. When no entry matches the input vector, the function produces 000. ■

10.2 Index Generation Unit

Figure 10.1 shows the **Index Generation Unit** (IGU). The **programmable hash circuit** has n inputs and p outputs. It is used to rearrange the nonzero elements. We consider two types of programmable hash circuits. The first type is the **double-input hash circuit** shown in Fig. 10.2. It performs a **linear transformation** $y_i = x_i \oplus x_j$ or $y_i = x_i$, where $i \neq j$. It uses a pair of multiplexers for each variable y_i. The upper multiplexers have the inputs $x_1, x_2, \ldots, x_n$. The register with $\lceil \log_2 n \rceil$ bits specifies which variable to select by the multiplexer. The lower multiplexers have the inputs $x_1, x_2, \ldots, x_n$, except for x_i. For the i-th input, the constant input 0 is

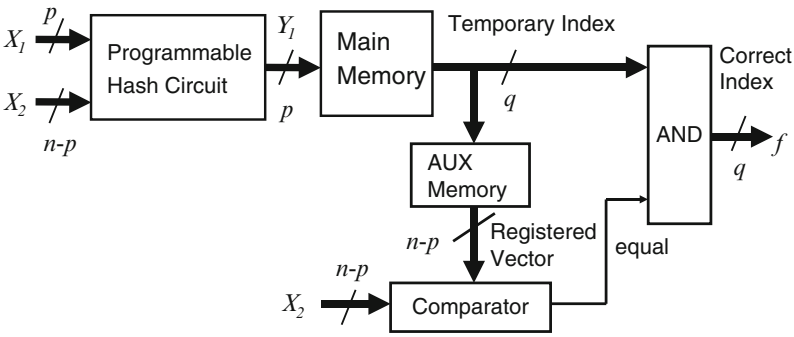


Fig. 10.1 Index generation unit (IGU)

Fig. 10.2 Double-input hash circuit

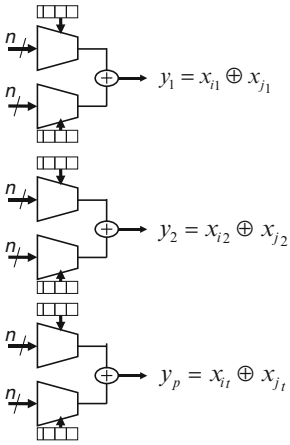
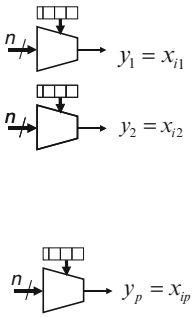


Fig. 10.3 Single-input hash circuit



connected instead of x_i. By setting $y_i = x_i \oplus 0$, we can implement $y_i = x_i$. The second type of a programmable hash circuit is the **single-input hash circuit** shown in Fig. 10.3. It consists of only p multiplexers, and selects p variables from n input variables. Note that both types of hash circuits produce only specific kinds of hash

functions. We have found that these functions are suitable for our application. The **main memory** has p inputs and $\lceil \log_2(k+1) \rceil$ outputs. The main memory produces correct outputs only for registered vectors. However, it may produce incorrect outputs for nonregistered vectors, because the number of input variables is reduced. In an index generation function, if the input vector is nonregistered, then it should produce $00\ldots0$. To check whether the main memory produces the correct output or not, we use the **AUX memory**. The AUX memory has $\lceil \log_2(k+1) \rceil$ inputs and $(n-p)$ outputs: It stores the X_2 part of the registered vectors for each index. The **comparator** checks if the inputs are the same as the registered vector or not. If they are the same, the main memory produces a correct output. Otherwise, the main memory produces a wrong output, and the input vector is nonregistered. Thus, the **output AND gates** produce $00\ldots0$, showing that the input vector is nonregistered. Note that the main memory produces the correct outputs only for the registered vectors.

Example 10.2.1. Consider the registered vectors in Table 10.3. The number of variables is four, but only two variables x_1 and x_4 are necessary to distinguish these four registered vectors. Figure 10.4 shows the IGU. In this case, the programmable hash circuit produces $Y_1 = (x_1, x_4)$ from $X = (x_1, x_2, x_3, x_4)$. The main memory stores the indices for $X_1 = Y_1 = (x_1, x_4)$, and the AUX memory stores the values of $X_2 = (x_2, x_3)$ for the corresponding registered vector.

Table 10.3 Index generation function

Inputs				Index
x_1	x_2	x_3	x_4	f
0	0	1	0	1
0	1	1	1	2
1	1	0	0	3
1	1	1	1	4

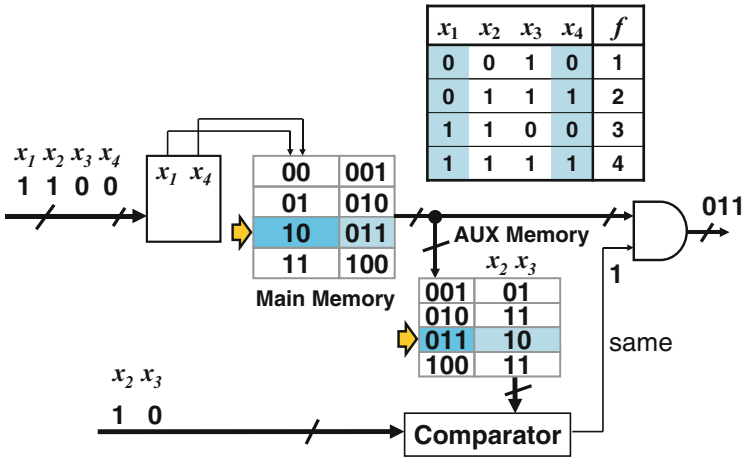


Fig. 10.4 When the input vector is registered

When the input vector is registered

Suppose that a registered vector $(x_1, x_2, x_3, x_4) = (1, 1, 0, 0)$ is applied to the IGU in Fig. 10.4. First, the programmable hash circuit selects two variables, x_1 and x_4, and produces the value $X_1 = (x_1, x_4) = (1, 0)$. Second, the main memory produces the corresponding index $(0, 1, 1)$. Third, the AUX memory produces the values of $X_2 = (x_2, x_3) = (1, 0)$ corresponding registered vector $(1, 1, 0, 0)$. Fourth, the comparator confirms that the values of $X_2 = (x_2, x_3)$ of the input vector is equal to the output of the AUX memory. And, finally, the AND gate produces the index for the input vector.

When the input vector is not registered

Suppose that a nonregistered vector $(x_1, x_2, x_3, x_4) = (1, 0, 1, 0)$ is applied to the IGU in Fig. 10.5. In this case, the main memory also produces the index $(0, 1, 1)$, and the AUX memory produces the values of $X_2 = (x_2, x_3)$ for the corresponding registered vector $(1, 1, 0, 0)$. However, in this case, the comparator indicates that $X_2 = (x_2, x_3) = (0, 1)$ is different from the output $X_2 = (x_2, x_3)$ of the AUX memory. Thus, the AND gate produces zero output, which shows that the input vector is not registered. ■

Unfortunately, not all index generation functions have the nice properties of Example 10.2.1. So, we decompose the given function into two:

1. A function that is implemented by an IGU.
2. The remaining part.

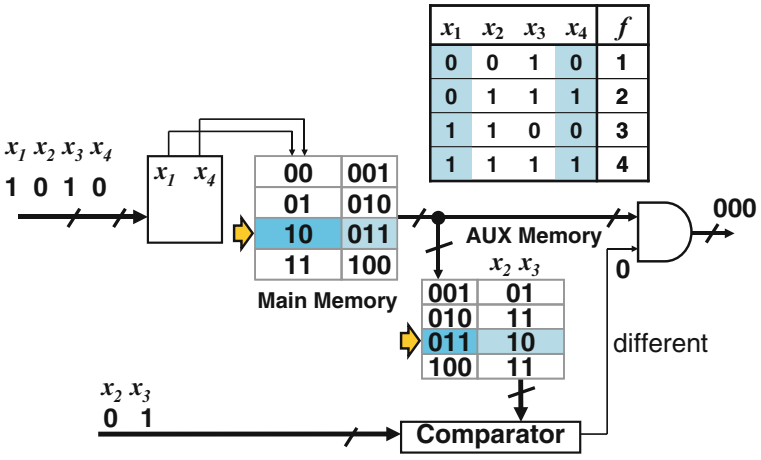


Fig. 10.5 When the input vector is not registered

Given an index generation function $f(X_1, X_2)$, where $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$, we decompose it into two disjoint subfunctions:

$$f(X_1, X_2) = \hat{f}_1(Y_1, X_2) \vee f_2(X_1, X_2),$$

where each column of the decomposition chart for $\hat{f}_1(Y_1, X_2)$ has at most one nonzero element. In this case, $\hat{f}_1(Y_1, X_2)$ can be implemented by an IGU, where the inputs to the main memory is $Y_1 = (y_1, y_2, \ldots, y_p)$. Since $f_2(X_1, X_2)$ has fewer nonzero elements than the original function, it is simpler to implement.

Theorem 10.2.1. *Consider the IGU in Fig. 10.1. Assume that $Y_1 = (y_1, y_2, \ldots, y_p)$, where $y_i = x_i \oplus x_j$ for $j \in \{p+1, p+2, \ldots, n\}$, or $y_i = x_i$, are applied to the input to the main memory. If the main memory of an IGU implements the function $\tilde{g}(Y_1)$, where $\tilde{g}(Y_1)$ produces the nonzero value when the column Y_1 of the decomposition chart for $\hat{f}_1(Y_1, X_2)$ has a nonzero value, and $\tilde{g}(Y_1)$ produces 0 otherwise, then only the values for X_2 must be stored in the AUX memory.*

Proof. Consider the decomposition chart of the function $\hat{f}_1(Y_1, X_2)$. By construction, each column of the decomposition chart has at most one nonzero element. When a registered vector is applied to the IGU, the main memory produces a nonzero output. In this case, the X_2 part of the input vector is equal to the output of the AUX memory, showing that the vector is registered. Thus, the IGU produces the correct nonzero output.

Assume that the input vector is not registered, but the output of the AUX memory is equal to the X_2 part of the input vector. We have two cases:

1. The main memory produces the zero-output.
   In this case, even if the X_2 part of the input vector is equal to the output of the AUX memory, the output of the main memory is zero. Thus, the IGU produces the correct output.
2. The main memory produces a nonzero output.
   Due to the construction of the IGU, the input vector is registered. However, this contradicts the assumption. So, such a case never happens. □

10.3 Reduction by a Linear Transformation

As will be suggested by Conjecture 11.5.2, most incompletely specified index generation functions with weight k can be represented by at most $p = 2\lceil \log_2(k + 1) \rceil - 3$ variables. However, there exist functions that require more variables. Example 10.3.1 shows such a function. In this case, we can often reduce the number of variables by a linear transformation of the input variables.

Example 10.3.1. Consider the incompletely specified index generation function shown in Table 10.4. Note that all the variables are essential in f. Now, replace the variables x_1, x_2, x_3, and x_4 with $y_1 = x_1 \oplus x_4$, $y_2 = x_2 \oplus x_4$, $y_3 = x_3$,

Table 10.4 Original index generation function

Inputs				Index
x_1	x_2	x_3	x_4	f
1	0	0	0	1
0	1	0	0	2
0	0	1	0	3
0	0	0	1	4
0	0	0	0	5

Table 10.5 Transformed index generation function

Inputs				Index
y_1	y_2	x_3	x_4	g
1	0	0	0	1
0	1	0	0	2
0	0	1	0	3
1	1	0	1	4
0	0	0	0	5

Table 10.6 Registered vector table for a 6-variable function

Vector						
x_1	x_2	x_3	x_4	x_5	x_6	Index
1	0	0	0	0	0	1
0	1	0	0	0	0	2
0	0	1	0	0	0	3
0	0	0	1	0	0	4
0	0	0	0	1	0	5
0	0	0	0	0	1	6
0	0	0	0	0	0	7

and $y_4 = x_4$, respectively. Then, f can be represented as the index generation function $g(y_1, y_2, y_3, y_4)$ shown in Table 10.5. Note that g can be represented using only y_1, y_2, and x_3, since they can uniquely specify five different patterns. The programmable hash circuit in Fig. 10.3 performs this linear transformation. ■

Example 10.3.2. In the registered vector table in Table 10.6, the number of 0's is much larger than that of 1's.

1. A single-input hash circuit is used.
   In this case, all the variables are necessary to represent the function, since any change of each variable from $(0, 0, 0, 0, 0, 0)$ will change the value of the function. Thus, the main memory requires 6 variables.
2. A double-input hash circuit is used.
   Consider the transform:

$$y_1 = x_1 \oplus x_5$$
$$y_2 = x_2 \oplus x_5$$
$$y_3 = x_3 \oplus x_6$$
$$y_4 = x_4 \oplus x_6$$

Table 10.7 Registered vector table for IGU with double-input hash circuit

Vector				
y_1	y_2	y_3	y_4	Index
1	0	0	0	1
0	1	0	0	2
0	0	1	0	3
0	0	0	1	4
1	1	0	0	5
0	0	1	1	6
0	0	0	0	7

Table 10.8 Registered vector table for IGU with triple-input hash circuit

Vector			
z_1	z_2	z_3	Index
1	0	0	1
0	1	0	2
0	0	1	3
0	1	1	4
1	0	1	5
1	1	0	6
0	0	0	7

Table 10.7 shows the transformed function. In this case, all the patterns are different. This means that these four variables are sufficient to represent the function. In fact, this is a minimum solution when a double-input hash circuit is used.

3. A triple-input hash circuit is used.
   Consider the transform:

$$z_1 = x_1 \oplus x_5 \oplus x_6$$
$$z_2 = x_2 \oplus x_4 \oplus x_6$$
$$z_3 = x_3 \oplus x_4 \oplus x_5$$

Table 10.8 shows the transformed function. In this case, all the patterns are different. This means that three variables are sufficient to represent the function. In fact, this is a minimum solution when a hash circuit with any number of inputs is used. ■

We have developed a heuristic algorithm [135] to find a linear transformation that reduces the number of variables, when the double-input hash circuit is used. To find a linear transformation, we use the following:

Theorem 10.3.1. *Let* $f(x_1, x_2, \ldots, x_n)$ *be an incompletely specified index generation function. Let* $Y_1 = (y_1, y_2, \ldots, y_p)$, *where* $y_i = x_i \oplus x_j$ *and* $j \in \{p+1, p+2, \ldots, n\}$, *and* $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$. *Consider the transformed function* $g(Y_1, X_2) = f(X_1, X_2)$. *Then,* f *can be represented using only* Y_1, *if each column of the decomposition chart* (Y_1, X_2) *has at most one specified element.*

10.4 Hybrid Method

From here, we consider methods to implement an index generation function using memories. In the index generation function, the number of registered vectors k, is usually much smaller than 2^n, the total number of the input combinations.

Definition 10.4.1. The **hybrid method** is an implementation of an index generation function using the circuit consisting of an IGU as shown in Fig. 10.6. An IGU is used to realize most of the registered vectors, while a rewritable PLA is used to realize remaining registered vectors. The OR gate in the output combines the indices to form a single output. The rewritable PLA can be replaced by another circuit, such as an LUT cascade or a CAM.

In the hybrid method, the main memory has $p = q + 2$ inputs, and realizes 88% of the registered vectors, where $q = \lceil \log_2(k + 1) \rceil$. The rest of the registered vectors are implemented by the rewritable PLA.

Example 10.4.1. For the circuit shown in Fig. 10.7, the values of only $X_2 = (x_4, x_5, x_6)$ are compared with the output of the AUX memory to check if the main memory produces the correct output. ■

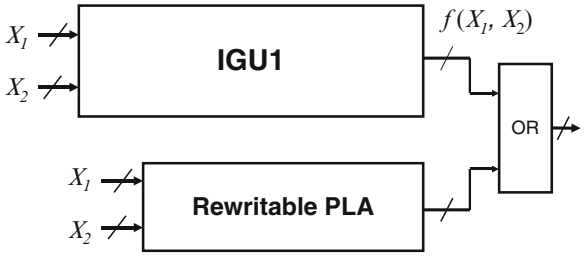


Fig. 10.6 Index generator using hybrid method

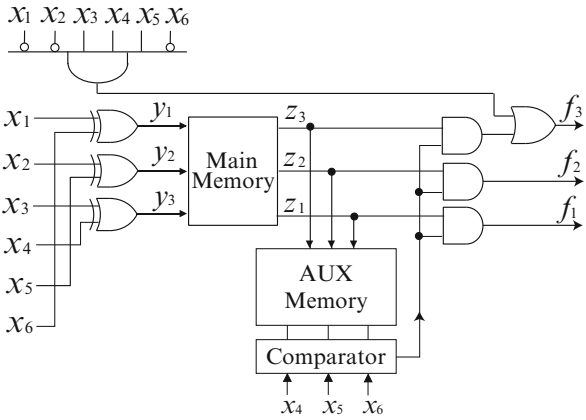


Fig. 10.7 6-variable function implemented by a hybrid method

Table 10.9 Decomposition chart for $f(X_1, X_2)$

			0	0	0	0	1	1	1	1	x_3
			0	0	1	1	0	0	1	1	x_2
			0	1	0	1	0	1	0	1	x_1
0	0	0	0	0	0	0	0	0	0	0	
0	0	1	0	0	0	0	0	0	0	0	
0	1	0	1	0	2	0	3	0	0	0	
0	1	1	0	0	0	0	4	0	0	0	
1	0	0	5	0	0	0	0	0	0	0	
1	0	1	0	0	0	0	0	0	0	0	
1	1	0	0	0	0	0	0	0	0	6	
1	1	1	0	0	7	0	0	0	0	0	
x_6	x_5	x_4									

Table 10.10 Decomposition chart for $\hat{f}(Y_1, X_2)$ (transformed function)

			0	0	0	0	1	1	1	1	y_3
			0	0	1	1	0	0	1	1	y_2
			0	1	0	1	0	1	0	1	y_1
0	0	0	0	0	0	0	0	0	0	0	
0	0	1	0	0	0	0	0	0	0	0	
0	1	0	2	0	1	0	0	0	3	0	
0	1	1	0	0	4	0	0	0	0	0	
1	0	0	0	5	0	0	0	0	0	0	
1	0	1	0	0	0	0	0	0	0	0	
1	1	0	0	0	0	0	6	0	0	0	
1	1	1	0	0	0	0	0	7	0	0	
x_6	x_5	x_4									

Example 10.4.2. Consider the index generation function defined by Table 10.2. Table 10.9 is a decomposition chart of a 6-variable function $f(X_1, X_2)$ with weight $k = 7$. In this function, transform the variables $X_1 = (x_1, x_2, x_3)$ into $Y_1 = (y_1, y_2, y_3) = (x_1 \oplus x_6, x_2 \oplus x_5, x_3 \oplus x_4)$. The decomposition chart of the transformed function $\hat{f}(Y_1, X_2)$ is shown in Table 10.10. In the transformed function, the columns of the original truth tables are permuted. Also, each row has a different permutation. In the original table, three columns for $(x_1, x_2, x_3) = (0, 0, 0), (0, 1, 0), (0, 0, 1)$ have two nonzero elements. On the other hand, in the decomposition chart in Table 10.10, for the transformed function $\hat{h}(Y_1, X_2)$, only one column $(y_1, y_2, y_3) = (0, 1, 0)$ has two nonzero elements. Let $\hat{f}_1(Y_1, X_2)$ be the function where the nonzero element 4 is replaced by 0. The decomposition chart is shown in Table 10.11. Table 10.12 shows the decomposition chart of the function $\hat{f}_2(Y_1, X_2)$ that is realized by the rewritable PLA. In this case, the function has only one nonzero element. $\hat{f}_1(Y_1, X_2)$ is implemented by the main memory shown in Table 10.13 and the AUX memory shown in Table 10.14. The output of the main memory $\hat{f}(Y_1)$ shows the nonzero value of the function $\hat{f}_1$ for the column $Y_1 = (y_1, y_2, y_3)$. The AUX memory shown in Table 10.14 stores the corresponding values of x_4, x_5, and x_6 when $f(X_1, X_2)$ takes nonzero values. Figure 10.8 shows that the pair of the main memory and the AUX memory is sufficient to represent

Table 10.11 Decomposition chart for $\hat{f}_1(Y_1, X_2)$

			0	0	0	0	1	1	1	1	y_3
			0	0	1	1	0	0	1	1	y_2
			0	1	0	1	0	1	0	1	y_1
0	0	0	0	0	0	0	0	0	0	0	
0	0	1	0	0	0	0	0	0	0	0	
0	1	0	2	0	1	0	0	0	3	0	
0	1	1	0	0	0	0	0	0	0	0	
1	0	0	0	5	0	0	0	0	0	0	
1	0	1	0	0	0	0	0	0	0	0	
1	1	0	0	0	0	0	6	0	0	0	
1	1	1	0	0	0	0	0	7	0	0	
x_6	x_5	x_4									

Table 10.12 Decomposition chart for $\hat{f}_2(Y_1, X_2)$

			0	0	0	0	1	1	1	1	y_3
			0	0	1	1	0	0	1	1	y_2
			0	1	0	1	0	1	0	1	y_1
0	0	0	0	0	0	0	0	0	0	0	
0	0	1	0	0	0	0	0	0	0	0	
0	1	0	0	0	0	0	0	0	0	0	
0	1	1	0	0	4	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	
1	0	1	0	0	0	0	0	0	0	0	
1	1	0	0	0	0	0	0	0	0	0	
1	1	1	0	0	0	0	0	0	0	0	
x_6	x_5	x_4									

Table 10.13 Function $\hat{h}(Y_1)$ realized by the main memory

y_3	0	0	0	0	1	1	1	1
y_2	0	0	1	1	0	0	1	1
y_1	0	1	0	1	0	1	0	1
$\hat{h}(Y_1)$	2	5	1	0	6	7	3	0

Table 10.14 Contents of the AUX memory

z_3	z_2	z_1	x_4	x_5	x_6
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	0	0	1
1	1	0	0	1	1
1	1	1	1	1	1

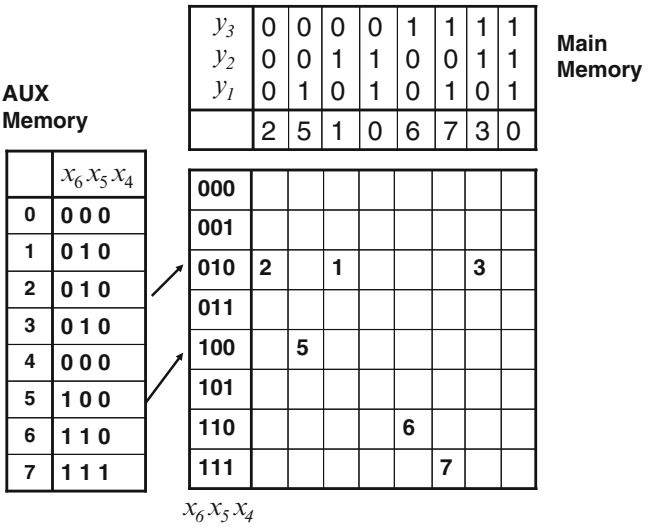


Fig. 10.8 Index generator implemented using a sparse matrix technique

the information of Table 10.11. Table 10.13 is implemented by the main memory. The function that is implemented by the rewritable PLA has nonzero output 4. The corresponding input values are $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 0, 1, 1, 1, 0)$. The nonzero output is 4, and its binary representation is $(1, 0, 0)$. This is implemented by ORing the most significant bit of the AND gates. Figure 10.7 shows the whole circuit for function f. The AUX memory and comparator check if (x_4, x_5, x_6) is the input that produces the nonzero output. ■

10.5 Registered Vectors Realized by Main Memory

In this part, we assume that the nonzero elements in the index generation function are uniformly distributed in the decomposition chart. In this case, we can estimate the fraction of registered vectors realized by the main memory.

Lemma 10.5.1. *Let $f(X)$ be a uniformly distributed index function of n variables with weight k. Consider a decomposition chart, and let p be the number of bound variables. Then, the probability that a column of the decomposition chart has all zero elements is approximately* $e^{-\xi}$, *where* $\xi = \frac{k}{2^p}$.

Proof. The probability that a function takes a specified value is $\alpha = \frac{k}{2^n}$. The probability that a function takes a zero value is $\beta = 1 - \alpha$. Since the decomposition chart has 2^{n-p} rows, the probability that a column of the chart has all zero elements is

$$\beta^{2^{n-p}} = (1 - \alpha)^{2^{n-p}}.$$

Since $\alpha = \frac{k}{2^n}$ is sufficiently small, by Lemma 3.7.1, $\beta = 1-\alpha$ can be approximated by $e^{-\alpha}$. Thus, we have

$$\beta^{2^{n-p}} \simeq e^{-\alpha 2^{n-p}} = e^{-\xi},$$

where $\xi = \frac{k}{2^p}$, □

Theorem 10.5.1. *Consider a set of uniformly distributed index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k. Consider an IGU whose inputs to the main memory are $x_1, x_2, \ldots,$ and x_p. Then, the expected number of registered vectors of f that can be realized by the IGU is $2^p(1 - e^{-\xi})$, where $\xi = \frac{k}{2^p}$.*

Proof. Let (X_1, X_2) be a partition of the input variables X, where $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$. Consider the decomposition chart for $f(X_1, X_2)$, where X_1 labels the column variables and X_2 labels the row variables. If a column has at least one *care* element, then the IGU can realize an element of the column. From Lemma 10.5.1, the probability that each column has at least one nonzero element is $1 - e^{-\xi}$, where $\xi = \frac{k}{2^p}$. Since there are 2^p columns, the expected number of registered vectors realized by the IGU is $2^p(1 - e^{-\xi})$. □

Example 10.5.1. Table 10.9 is the decomposition chart for a 6-variable index generation function with weight $k = 7$. Note that $X_1 = (x_1, x_2, x_3)$ denotes the bound variables, and $X_2 = (x_4, x_5, x_6)$ denotes the free variables. In this case, three columns $(x_1, x_2, x_3) = (0, 0, 1)$, $(0, 1, 1)$, $(1, 0, 1)$, and $(1, 1, 0)$ have all zero elements. In the other words, the fraction of columns that have all zero elements is $\frac{4}{8} = 0.5$. In Lemma 10.5.1, we have $n = 6$, $p = 3$, and $\xi = \frac{k}{2^p} = 0.875$. It shows that the probability that a column has all zero elements is $e^{-\xi} = 0.4169$. In Theorem 10.5.1, the expected number of vectors realized by the IGU is

$$2^p(1 - e^{-\xi}) = 8 \times 0.583 = 4.665.$$

In Table 10.9, four vectors for 1, 2, 3, 6 can be realized by an IGU. The remaining vectors should be realized by other parts of the circuit. ■

Corollary 10.5.1. *Consider a set of uniformly distributed incompletely specified index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k. Consider an IGU whose inputs to the main memory are $x_1, x_2, \ldots,$ and x_p. Then, the fraction of registered vectors of f that can be realized by the IGU is*

$$\delta = \frac{1 - e^{-\xi}}{\xi},$$

where $\xi = \frac{k}{2^p}$.

For example, when $\frac{k}{2^p} = \frac{1}{4}$, we have $\delta \simeq 0.8848$, when $\frac{k}{2^p} = \frac{1}{2}$, we have $\delta \simeq 0.7869$, and when $\frac{k}{2^p} = 1$, we have $\delta \simeq 0.63212$.

Example 10.5.2. Consider the case of $n = 40$ and $k = 1730$. Let us compare two realizations: LUT cascade and hash-based. Since $q = \lceil \log_2(k+1) \rceil = \lceil \log_2(1730+1) \rceil = 11$, the number of bound variables is $p = 13$.

1. **Realization with an LUT cascade alone**
   Let $p = 13$ be the number of inputs for cells. Then, from Lemma 5.1.5, the number of levels s of the cascade is given by

$$s = \left\lceil \frac{n-q}{p-q} \right\rceil = \left\lceil \frac{40-11}{13-11} \right\rceil = \left\lceil \frac{29}{2} \right\rceil = 15.$$

   For each cell, the size of the memory is $2^p \times q = 2^{13} \times 11$ bits. Thus, the total amount of memory is $2^{13} \times 11 \times 15 = 1351680$ bits.
2. **Realization with the hybrid method**
   From the Corollary 10.5.1, the fraction of registered vectors of f that can be realized by the IGU is

$$\delta \simeq \frac{1 - \mathrm{e}^{-\xi}}{\xi} = 0.9015.$$

   The main memory has $p = 13$ inputs and $q = 11$ outputs. The AUX memory has $q = 11$ inputs and $r = n - p = 27$ outputs. The LUT cascade realizes the index generation function with weight $1730 \times (1 - 0.901) \simeq 171$. In this case, each cell in the cascade has $\lceil \log_2(171 + 1) \rceil = 8$ outputs. Let the number of inputs of cells be 10. Then, the number of levels in the LUT cascade is

$$\left\lceil \frac{n-q}{p-q} \right\rceil = \left\lceil \frac{40-8}{10-8} \right\rceil = \left\lceil \frac{32}{2} \right\rceil = 16.$$

   Note that the size of a cell except for the last stage is $2^{10} \times 8$ bits. The size of the cell in the last stage is $2^{10} \times 11$ bits. Thus, the total amount of memory for the cascade is $2^{10} \times 8 \times 15 + 2^{10} \times 11 = 134{,}144$ bits. The size of the main memory is $2^{13} \times 11 = 90{,}112$ bits. The size of the AUX memory is $2^{11} \times 27 = 55{,}296$ bits. Thus, the total amount of memory is 279,552 bits, which is 20.7% of the total memory for the LUT cascade-only realization.

In this example, the hybrid method requires a smaller amount of memory than the LUT cascade alone. ■

10.6 Super Hybrid Method

In the hybrid method, about 88% of the registered vectors are implemented by an IGU, and the remaining 12% are implemented by the PLA. When we use two IGUs, about 96% of the registered vectors are implemented by IGUs and the remaining 4% are implemented by the PLA.

Definition 10.6.1. The **super hybrid method** is an implementation of an index generation function using a circuit consisting of two IGUs, as shown in Fig. 10.9.

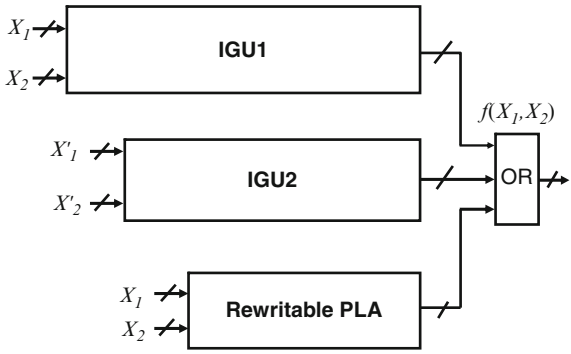


Fig. 10.9 Index generator implemented by super hybrid method

IGU_1 is used to realize most of the registered vectors, IGU_2 is used to realize the registered vectors not realized by IGU_1, and the rewritable PLA is used to realize registered vectors not realized by either IGU. The OR gate in the output combines the indices to form a single output. The rewritable PLA can be replaced by another circuit, such as an LUT cascade, a CAM or an IGU.

The super hybrid method shown in Fig. 10.9 is more complicated than the hybrid method, but requires smaller memory overall. In this method, the main memory in IGU_1 has $p = q + 1$ inputs, and realizes 79% of the registered vectors. The main memory in IGU_2 has $p = q$ inputs, and realizes 16.6% of registered vectors. The rest of the registered vectors are implemented by the rewritable PLA.

Hybrid Method. In a typical hybrid method, the main memory has $p = q+2$ inputs and $q = \lceil \log_2(k+1) \rceil$ outputs, while the AUX memory has q inputs and $n-q-2$ outputs. Therefore, the total amount of memory is

$$M_1 = q \cdot 2^{q+2} + (n-q-2) \cdot 2^q = (4n + 12q - 8) \cdot 2^{q-2}.$$

Super Hybrid Method. In a typical super hybrid method, in IGU_1, the main memory has $p_1 = q + 1$ inputs and q outputs, while the AUX memory has q inputs and $n-q-1$ outputs. Also, in IGU_2, the main memory has $p_2 = q-1$ inputs and $q-2$ outputs, while the AUX memory has $q-2$ inputs and $n-q+1$ outputs. Therefore, the total amount of memory is

$$M_2 = q \cdot 2^{q+1} + (n-q-1) \cdot 2^q + (q-2) \cdot 2^{q-1} + (n-q+1) \cdot 2^{q-2} = (5n+5q-7) \cdot 2^{q-2}.$$

This implies that, when $n \leq 7 \log_2(k+1) - 1$, the super hybrid method requires a smaller amount of memory.

Theorem 10.6.1. *1. In a typical hybrid method, about 88% of the registered vectors can be realized by an IGU.*

2. *In a typical super hybrid method, about 96% of the registered vectors can be realized by two IGUs.*

Proof. **Hybrid Method**
In this case, the number of inputs to the main memory is $p = \lceil \log_2(k+1) \rceil + 2$. Thus, $2^p \geq 4(k+1)$ and $\xi = \frac{k}{2^p} \leq \frac{k}{4(k+1)} \simeq \frac{1}{4}$. From Corollary 10.5.1, the fraction of registered vectors realized by the IGU is

$$\delta = \frac{1 - \mathrm{e}^{-\xi}}{\xi}.$$

When $\xi = \frac{1}{4}$, we have $\delta = 4(1 - \mathrm{e}^{-0.25}) = 0.8848$.

Super Hybrid Method

In this case, $p_1 = \lceil \log_2(k+1) \rceil + 1$. Thus, $2^{p_1} \geq 2(k+1)$ and $\xi_1 \leq \frac{k}{2(k+1)} \simeq \frac{1}{2}$. When $\xi_1 = \frac{1}{2}$, we have $\delta_1 = 4(1-\mathrm{e}^{-0.5}) = 0.7869$. Thus, the fraction of remaining vectors is 0.213.

Note that $p_2 = \lceil \log_2(k+1) \rceil - 1$. Thus, $2^{p_2} \geq \frac{1}{2}(k+1)$, and $\xi_2 \leq \frac{k_2}{2^{p_2}} \simeq \frac{0.213k}{0.5(k+1)} \simeq 0.426$. When, $\xi_2 = 0.426$, we have $\delta_2 = \frac{1-\mathrm{e}^{-0.426}}{0.426} = 0.8142$.

Thus, the total number of registered vectors realized by IGU_1 and IGU_2 is

$$0.7896\,\mathrm{k} + 0.8142 \times 0.2134\,\mathrm{k} = 0.9603\,\mathrm{k}.$$

Thus, we have the theorem. □

Example 10.6.1. Consider the index generation function with $n = 40$ and $k = 1730$. In this case, $q_1 = \lceil \log_2(k+1) \rceil = \lceil \log_2(1730+1) \rceil = 11$.

Rewritable PLA

The number of vectors realized by the rewritable PLA is 1730.

Hybrid Method

The main memory has $p = q_1 + 2 = 13$ inputs and $q_1 = 11$ outputs. The AUX memory has $q_1 = 11$ inputs and $r = n - p = 27$ outputs. Since, $\xi = \frac{1730}{2^{13}}$, from Corollary 10.5.1, the fraction of registered vectors of f that can be realized by the IGU is

$$\delta \simeq \frac{1 - \mathrm{e}^{-\xi}}{\xi} = 0.901.$$

Thus, the number of vectors realized by the IGU is $1730 \times 0.901 = 1599$, and the number of remaining vectors to be realized by the rewritable PLA is 171.

The size of the main memory is $2^{13} \times 11 = 90{,}112$ bits. The size of the AUX memory is $2^{11} \times 27 = 55{,}296$ bits. Thus, the total amount of memory is 145,408 bits.

Super Hybrid Method

The first main memory has $p_1 = q_1 + 1 = 12$ inputs and $q_1 = 11$ outputs. The first AUX memory has $q_1 = 11$ inputs and $r_1 = n - p_1 = 27$ outputs.

Since, $\xi_1 = \frac{1730}{2^{12}} = 0.422$, from Corollary 10.5.1, the fraction of registered vectors of f that can be realized by IGU_1 is

$$\delta_1 \simeq \frac{1 - \mathrm{e}^{-\xi_1}}{\xi_1} = 0.8156.$$

Thus, the number of vectors realized by IGU_1 is $1730 \times 0.8156 = 1411$, and the number of the remaining vectors is $1730 - 1411 = 319$.

The second main memory has $p_2 = q_1 - 1 = 10$ inputs and $q_2 = 9$ outputs. The second AUX memory has $q_2 = 9$ inputs and $r_2 = n - p_2 = 30$ outputs. Since $\xi_2 = \frac{319}{2^{p_2}} = 0.3115$, the fraction of registered vectors of f that can be realized by IGU_2 is

$$\delta_2 \simeq \frac{1 - \mathrm{e}^{-\xi_2}}{\xi_2} = 0.859223.$$

Thus, the number of vectors realized by $IGU2$ is $319 \times 0.859223 = 274$, and the number of the remaining vectors is 45.

The size of the first main memory is $2^{12} \times 11 = 45{,}056$ bits. The size of the first AUX memory is $2^{11} \times 28 = 57{,}344$ bits. The size of the second main memory is $2^{10} \times 9 = 9{,}216$ bits. The size of the second AUX memory is $2^9 \times 30 = 15{,}360$ bits. Thus, the total amount of memory is 126,976 bits. The number of vectors realized by the rewritable PLA is 45.

Thus, for this problem, the super hybrid method requires a smaller amount of memory than the hybrid method. ■

A problem with the super hybrid method is that the second main memory has only $q - 2$ outputs. Thus, the indices of the registered vectors in the second main memory should be smaller than or equal to $2^{q-2} - 1$. The first main memory stores registered vectors whose indices are greater than 2^{q-2}.

10.7 Parallel Sieve Method

The hybrid method uses only one IGU, while the super hybrid method uses two IGUs. By increasing the number of IGU's, we have the parallel sieve method. The parallel sieve method is especially useful when the number of the registered vectors is very large [85] (Fig. 10.11).

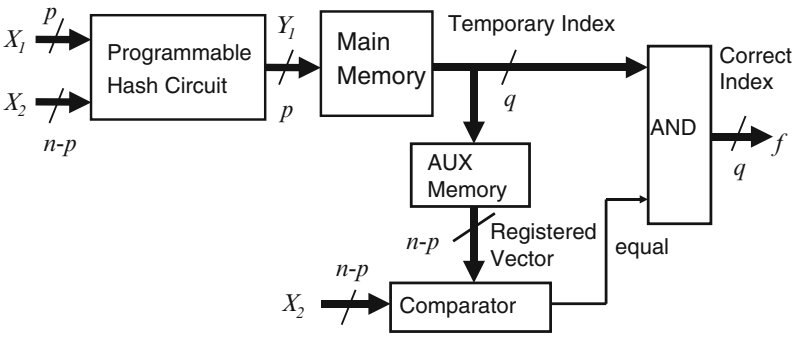


Fig. 10.10 Index generator unit (IGU)

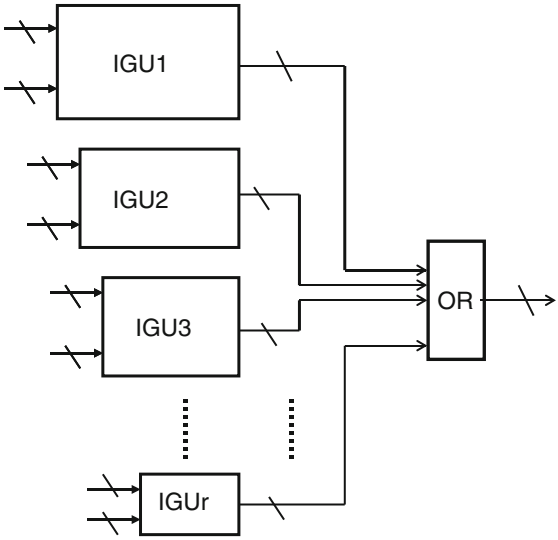


Fig. 10.11 Index generator implemented by parallel sieve method

Definition 10.7.1. The **parallel sieve method** is an implementation of an index generation function using the circuit consisting of multiple IGUs as shown in Fig. 10.11. IGU_{i+1} is used to realize a part of the registered vectors not realized by IGU_1, IGU_2, ..., or IGU_i. The OR gate in the output combines the indices to form a single output. In the **standard parallel sieve method**, the number of inputs to the main memory is selected as

$$p_i = \lceil \log_2(k_i + 1) \rceil.$$

Example 10.7.1. By using the standard parallel sieve method, realize an index generation function with $n = 40$ and $k_1 = 10,000$. Note that $q_1 = \lceil \log_2(k_1 + 1) \rceil = 14$. Consider Fig. 10.11.

1. In IGU_1, the number of inputs for the main memory is $p_1 = q_1 = 14$. By Theorem 10.5.1, the number of the vectors realized by IGU_1 is $2^{p_1}(1 - e^{-\xi_1})$ where $\xi_1 = \frac{k_1}{2^{p_1}}$, which is $16384 \times (1 - 0.5432) = 7484$. The number of remaining vectors is $k_2 = k_1 - 7484 = 2516$.
2. In IGU_2, since $q_2 = \lceil \log_2(2516 + 1) \rceil = 12$, the number of the inputs for the main memory is $p_2 = q_2 = 12$. The number of the vectors realized by IGU_2 is $2^{p_2}(1 - e^{-\xi_2})$ where $\xi_2 = \frac{k_2}{2^{p_2}}$, which is $4096 \times 0.4589 = 1879$. The number of remaining vectors is $k_3 = k_2 - 1879 = 637$.
3. In IGU_3, since $q_3 = \lceil \log_2(637 + 1) \rceil = 10$, the number of inputs for the main memory is $p_3 = q_3 = 10$. The number of vectors realized by IGU_3 is $2^{p_3}(1 - e^{-\xi_3})$ where $\xi_3 = \frac{k_3}{2^{p_3}}$, which is $1024 \times 0.46317 = 474$. The number of remaining vectors is $k_4 = k_3 - 474 = 163$.
4. In IGU_4, since $q_4 = \lceil \log_2(163 + 1) \rceil = 8$, the number of inputs for the main memory is $p_4 = q_4 = 8$. The number of vectors realized by IGU_4 is $2^{p_4}(1 - e^{-\xi_4})$ where $\xi_4 = \frac{k_4}{2^{p_4}}$, which is $256 \times 0.46317 = 120$. The number of remaining vectors is $k_4 = k_3 - 120 = 43$.
5. In IGU_5, since $q_5 = \lceil \log_2(43 + 1) \rceil = 6$, the number of inputs for the main memory is $p_5 = q_5 = 6$. The number of vectors realized by IGU_5 is $2^{p_5}(1 - e^{-\xi_5})$ where $\xi_5 = \frac{k_5}{2^{p_5}}$, which is $64 \times 0.48925 = 31$. The number of remaining vectors is $k_5 = k_4 - 31 = 12$.
6. In IGU_6, since the number of the remaining vectors is only $k_5 = 12$, they can be implemented by an IGU [132], or rewritable PLA or an LUT cascade.

Note that, for each IGU_i, the main memory has p_i inputs and p_i outputs, while the AUX memory has p_i inputs and $(n - p_i)$ outputs. Thus, the total amount of memory for IGU_i is

$$p_i 2^{p_i} + (n - p_i)2^{p_i} = n2^{p_i}.$$

The amount of memory for each IGU_i is:

$$IGU_1 : 40 \times 2^{14} = 640 \times 2^{10}.$$
$$IGU_2 : 40 \times 2^{12} = 160 \times 2^{10}.$$
$$IGU_3 : 40 \times 2^{10}.$$
$$IGU_4 : 40 \times 2^{8} = 10 \times 2^{10}.$$
$$IGU_5 : 40 \times 2^{6} = 2.5 \times 2^{10}.$$

The total amount of memory for the standard parallel sieve method is

$$\sum_{i=1}^{5} n2^{n_i} = 640 + 160 + 40 + 10 + 2.5 = 852.5$$

Kibits. ■

10.8 Experimental Results

10.8.1 List of English Words

To demonstrate the usefulness of the design method, first we realized lists of frequently used English words by the hybrid method shown in Fig. 10.6 and the super hybrid method shown in Fig. 10.9. Here, we use three kinds of English word lists: List 1, List 2, and List 3. The numbers of letters in the word lists are at most 13, but we only consider the first 8 letters. For the English words consisting of fewer than 8 letters, we append blanks to the end of words to make them 8-letter words. Each English alphabet letter is represented by 5 bits. Thus, each English word is represented by 40 bits. The numbers of words in the lists are 1,730, 3,366, and 4,705, respectively. Within each word list, each English word has a unique index, an integer from 1 to k, where $k = 1{,}730$ or 3,360 or 4,705. The numbers of bits for the indices are 11, 12, and 13, respectively.

The number of inputs for the main memory is $\lceil \log_2(k+1) \rceil + 2$. List 1 consists of $k = 1730$ words. The number of bits for the index is $q = \lceil \log_2(1+k) \rceil = \lceil \log_2(1+1730) \rceil = 11$. The number of bound variables is $p = q + 2 = 13$. The number of columns in the decomposition chart is $2^p = 2^{13} = 8,192$. The number of columns that has only one nonzero element is 1,389. The number of columns that has two or more nonzero elements is 165. The number of registered vectors that are not realized by the main memory is 176. In other words, about 90% of the registered vectors are realized by the main memory, and the remaining 10% of the registered vectors are realized by the rewritable PLA. Table 10.15 shows the design results for three English word lists by the hybrid method shown in Fig. 10.6.

Table 10.15 compares the amount of hardware for the hybrid method, and the super hybrid method. In the super hybrid method, the number of vectors realized by the rewritable PLA is smaller than 4% of the registered vectors. This is because we optimized the hash functions.

Table 10.15 Realization of English word lists by hybrid method

	List 1	List 2	List 3
# of words: k	1,730	3,366	4,705
# of inputs: n	40	40	40
# of outputs: q	11	12	13
# of inputs for the main memory :	p 13	14	15
# of columns with only one nonzero element	1389	2752	3980
# of columns with two or more nonzero elements	165	293	351
# of registered vectors not realized by main memory	176	321	374

10.8.2 Randomly Generated Functions

Next, we generated index generation functions with the same sizes by pseudo-random numbers. We did the similar experiments for List 2 and List 3. The experimental results using randomly generated functions and English word lists are close to the theoretical results obtained in Sect. 10.5. This shows that the hash function generated by the hash network effectively scatters the nonzero elements in the decomposition charts.

10.8.3 IP Address Table

To verify the effectiveness of the method, we also used IP addresses of computers that accessed our web site in a certain period. List 1 contains 1,730 addresses, List 2 contains 3,366 addresses, and List 3 contains 4,588 addresses. The number of inputs are all 32, but the number of outputs for Lists 1–3 are 11,12, and 13, respectively. Also, in this case, results produced by the real address tables, the data obtained from the random address tables, and the data obtained by analytical results in Sect. 10.5 were similar (Table 10.16). (Experimental results are omitted.)

Table 10.16 Amount of hardware for English word lists

Size of Lists		List 1	List 2	List 3
# of inputs	n	40	40	40
# of outputs	q	11	12	13
# of vectors	k	1,730	3,366	4,705
Hybrid Method		**List 1**	**List 2**	**List 3**
# of inputs for main memory	p	13	14	15
Size of main memory	$q2^p$	90,112	196,608	425,984
Size of AUX memory	$r2^q$	55,296	106,496	204,800
Total amount of memory		145,408	303,104	630,784
# of remaining vectors		176	321	374
Super Hybrid Method		**List 1**	**List 2**	**List 3**
# of inputs for main memory 1	p_1	12	13	14
# of inputs for main memory 2	p_2	10	11	12
Size of main memory 1	$q_1 2^{p_1}$	45,056	98,304	212,992
Size of AUX memory 1	$r_1 2^{q_1}$	57,344	110,592	212,992
Size of main memory 2	$q_2 2^{p_2}$	9,216	20,480	40,960
Size of AUX memory 2	$r_2 2^{q_2}$	15,360	2,969	28,672
Total amount of memory		126,976	232,345	495,616
# of remaining vectors		30	61	42

10.9 Remarks

This chapter presented the hybrid method, the super hybrid method, and the parallel sieve method to realize index generation functions. In these methods, an index generation function f is decomposed into nonoverlapping index generation functions, and each function is realized by an IGU or a rewritable PLA. In this chapter, a rewritable PLA is used to realize the remaining vectors. However, other methods can also be used: a CAM, an LUT cascade, or an IGU implemented by the method shown in Chap. 11. This chapter is based on [126, 128, 136].

Problems

10.1. In Example 10.1.1, suppose that, in a company with 5,000 employees, each person has a unique employee number between 1 and 5,000 inclusive. Suppose that $\text{hash}(x) = x (\text{mod } 9973)$ is used to find the employee number from his or her 10-digit telephone number. Calculate the expected number of collisions in the hash table. Do the same calculation when the number of the employees is 2,000, instead of 5,000. Assume that the hash function produces a uniform distribution.

10.2. Let $f(x)$ be the index generation function, where $f(x) = i$ when x is the i-th prime number, and $f(x) = 0$ otherwise. Let $\pi(x)$ be the **prime-counting function** that gives the number of primes less than or equal to x, for any integer number x. For example, $\pi(8) = 4$ because there are four prime numbers (2, 3, 5, and 7) less than or equal to 8. It is known that $\pi(100,000) = 9,592$.

Design the circuit of $f(x)$ that works for $x \leq 100,000$, by the standard parallel sieve method. Estimate the size of the circuit, and compare it with the single-memory realization.

10.3. Design an 8-digit **ternary-to-binary converter** [119]. Use the binary-coded-ternary code to represent a ternary digit. That is, 0 is represented by (00); 1 is represented by (01); and 2 is represented by (10). (11) is an unused code. Let $\vec{y} = (y_{m-1}, y_{m-2}, \ldots, y_0)$ be the outputs of the converter, where $y_i \in \{0, 1\}$. Then, in general, y_i depends on all the inputs $x_i (i = 0, 1, \ldots, n-1)$. When this converter is implemented by a two-valued logic circuit, unused combinations occur. So, we have an incompletely specified function. For example, the truth table of the 2-digit ternary to 4-bit binary converter is shown in Table 10.17. In the case of binary-coded-ternary representation, (11) is an undefined input, and the corresponding output is a *don't care*. In Table 10.17, the binary-coded-ternary representation is denoted by $\vec{w} = (w_3, w_2, w_1, w_0)$, the ternary representation is denoted by $\vec{x} = (x_1, x_0)$, and the binary representation is denoted by $\vec{y} = (y_3, y_2, y_1, y_0)$. Design a converter for an 8-digit ternary number to a 13-digit binary number by the standard parallel sieve method. Compare the memory size with that of the single-memory realization.

Table 10.17 Truth table for a ternary-to-binary converter

Binary-coded Ternary				Ternary		Binary				
w_3	w_2	w_1	w_0	x_1	x_0	y_3	y_2	y_1	y_0	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	1	1
0	0	1	0	0	2	0	0	1	0	2
0	1	0	0	1	0	0	0	1	1	3
0	1	0	1	1	1	0	1	0	0	4
0	1	1	0	1	2	0	1	0	1	5
1	0	0	0	2	0	0	1	1	0	6
1	0	0	1	2	1	0	1	1	1	7
1	0	1	0	2	2	1	0	0	0	8

Table 10.18 1-out-of-15 to binary converter

1-out-of-15 code															Index
x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	3
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	4
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	5
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	6
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	7
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	8
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	9
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	10
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	11
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	12
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	13
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	14
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15

10.4. A linear transformation of the input variables, in general, changes a function into an another function. However, it does not change the weight of the function. Prove this.

10.5. In the proof of Lemma 10.5.1, $\beta^{2^{n-p}}$ is approximated by $e^{-\xi}$, where $\xi = \frac{k}{2^p}$, $\beta = 1 - \alpha$, and $\alpha = \frac{k}{2^n}$. When $k = 2^p$, compute the approximation error: $ERROR = e^{-\xi} - \beta^{2^{n-p}}$. Make a table similar to Table 3.1 for $\alpha = 2^{-1}, 2^{-2}, 2^{-3}, \ldots, 2^{-14}$, and 2^{-15}.

10.6. Consider the 15-variable incompletely specified index generation function $f(X)$ shown in Table 10.18. Show that at least 14 variables are necessary to represent the function. Next, consider the linear transformation:

$$
\begin{aligned}
y_1 &= x_1 \oplus x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11} \oplus x_{13} \oplus x_{15},\\
y_2 &= x_2 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11} \oplus x_{14} \oplus x_{15},\\
y_3 &= x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_{12} \oplus x_{13} \oplus x_{14} \oplus x_{15},\\
y_4 &= x_8 \oplus x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12} \oplus x_{13} \oplus x_{14} \oplus x_{15}.
\end{aligned}
$$

Show that f can be represented with y_1, y_2, y_3, and y_4.

Chapter 11
Reduction of the Number of Variables

This chapter considers a method to reduce the number of variables needed to represent incompletely specified logic functions. When the number of specified minterms is k, most functions can be represented with at most $2\lceil \log_2(k+1) \rceil - 2$ variables.

11.1 Optimization for Incompletely Specified Functions

For completely specified logic functions, logic minimization is a process of reducing the number of products to represent the given function. However, for incompletely specified functions (i.e., functions with *don't cares*), at least two problems exist [46]. The first is to reduce the number of the products to represent the function, and the second is to reduce the number of variables. The first problem is useful for sum-of-products expression (SOP)-based realizations [13], while the second problem is useful for memory-based realizations, since reducing the number of variables reduces the memory size.

Example 11.1.1. Consider the 4-variable function shown in Fig. 11.1, where the blank cells denote *don't cares*. The SOP with the minimum number of products is $\mathcal{F}_1 = x_1x_4 \vee x_2\bar{x}_3$, while the SOP with the minimum number of variables is $\mathcal{F}_2 = x_1x_2 \vee x_1x_4 \vee x_2x_4$. Note that $\mathcal{F}_1$ shown in Fig. 11.2 has two products and depends on four variables, while $\mathcal{F}_2$ shown in Fig. 11.3 has three products and depends on only 3 variables. x_3 is a **nonessential** variable, since $\mathcal{F}_2$ does not include it. ■

As shown in this example, the expression corresponding to the minimal number of products is different from the expression corresponding to the minimal number of variables. This chapter considers the minimization of the number of variables in incompletely specified functions. Indeed, it is shown that many variables can be eliminated when the fraction of don't cares is large.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_11,

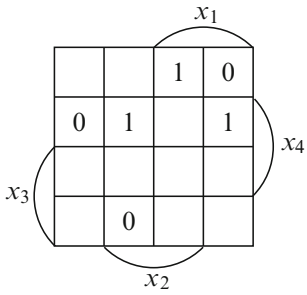


Fig. 11.1 4-variable incompletely specified logic function

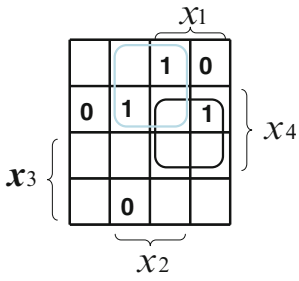


Fig. 11.2 Expression with the fewest products

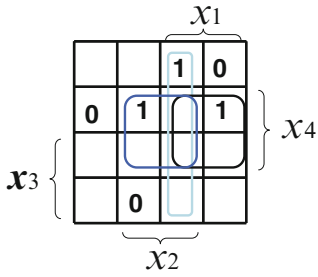


Fig. 11.3 Expression with the fewest variables

11.2 Definitions and Basic Properties

Definition 11.2.1. An **incompletely specified logic function** f is a mapping $D \to B$, where $D \subset B^n$, $B = \{0, 1\}$.

Definition 11.2.2. An incompletely specified logic function is represented by a pair of **characteristic functions** F_0 and F_1, where $F_0(\vec{a}) = 1$ iff $f(\vec{a}) = 0$, and $F_1(\vec{a}) = 1$ iff $f(\vec{a}) = 1$. Note that $F_0 F_1 = 0$. If $\vec{a} \in D$, then the value of $f(\vec{a})$ is specified, and is called *care* value. Otherwise, the value of $f(\vec{a})$ is unspecified, and is called *don't care.*

Table 11.1 Function for Fig. 11.1

x_1	x_2	x_3	x_4	f
0	0	0	1	0
0	1	1	0	0
1	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	0	0	1

Example 11.2.1. Consider the function in Fig. 11.1. In this case $n = 4$. Table 11.1 also shows this function. The characteristic functions are

$$F_0 = \bar{x}_1\bar{x}_2\bar{x}_3x_4 \vee \bar{x}_1x_2x_3\bar{x}_4 \vee x_1\bar{x}_2\bar{x}_3\bar{x}_4 \text{ and}$$
$$F_1 = \bar{x}_1x_2\bar{x}_3x_4 \vee x_1\bar{x}_2\bar{x}_3x_4 \vee x_1x_2\bar{x}_3\bar{x}_4.$$

In this case, the function is specified for only 6 out of 16 possible minterms. ■

Definition 11.2.3. f **depends on** x_i if there exists a pair of vectors

$$\vec{a} = (a_1, a_2, \ldots, a_i, \ldots, a_n) \text{ and}$$
$$\vec{b} = (a_1, a_2, \ldots, b_i, \ldots, a_n),$$

such that both $f(\vec{a})$ and $f(\vec{b})$ are specified, and $f(\vec{a}) \neq f(\vec{b})$.

If f depends on x_i, then x_i is **essential** in f and x_i must appear in every expression for f.

Definition 11.2.4. Two functions f and g are **compatible** when the following condition holds: For any $\vec{a} \in B^n$, if both $f(\vec{a})$ and $g(\vec{a})$ are specified, then $f(\vec{a}) = g(\vec{a})$.

Lemma 11.2.1. *Let* $f_0 = f(|x_i = 0)$ *and* $f_1 = f(|x_i = 1)$. *Then,* x_i *is* **nonessential** *in* f *iff* f_0 *and* f_1 *are compatible.*

If x_i is nonessential in f, then f can be represented by an expression without x_i.

Example 11.2.2. Consider the function f in Fig. 11.4. It is easy to verify that all the variables are nonessential. Note that f can be represented as $\mathcal{F}_1 = \bar{x}_2 \vee x_3$ or $\mathcal{F}_2 = x_1 \oplus \bar{x}_4$. ■

Essential variables must appear in every expression for f, while nonessential variables may appear in some expressions and not in others. Algorithms to represent a given function by using the minimum number of variables have been considered [14, 32, 38, 46, 67].

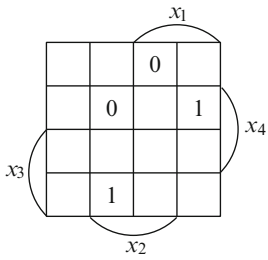


Fig. 11.4 4-variable function without essential variables

11.3 Algorithm to Minimize the Number of Variables

This section describes an algorithm to represent an incompletely specified index generation function $f : D \to \{1, 2, \ldots, k\}$, where $D \subset B^n$, using the minimum number of variables. To show the idea of the method, we use the following:

Example 11.3.1. Let us minimize the number of variables to represent the index generation function shown in Fig. 11.5.

1. Let the four vectors be $\vec{a}_1 = (1, 0, 0, 1)$, $\vec{a}_2 = (1, 1, 1, 1)$, $\vec{a}_3 = (0, 1, 0, 1)$, and $\vec{a}_4 = (1, 1, 0, 0)$.
2. To distinguish $\vec{a}_1$ and $\vec{a}_2$, either x_2 or x_3 is necessary. Thus, we have the condition $x_2 \vee x_3 = 1$, where $x_1 = 1$ denotes that x_1 must appear in the expression. Thus, $x_2 \vee x_3 = 1$ denotes either x_2 or x_3 must appear in the expression. In the same way, to distinguish $\vec{a}_1$ and $\vec{a}_3$, we have the condition $x_1 \vee x_2 = 1$; to distinguish $\vec{a}_1$ and $\vec{a}_4$, we have the condition $x_2 \vee x_4 = 1$; to distinguish $\vec{a}_2$ and $\vec{a}_3$, we have the condition $x_1 \vee x_3 = 1$; to distinguish $\vec{a}_2$ and $\vec{a}_4$, we have the condition $x_3 \vee x_4 = 1$; and to distinguish $\vec{a}_3$ and $\vec{a}_4$, we have the condition $x_1 \vee x_4 = 1$.
3. To distinguish all the vectors, all the conditions must hold at the same time. This is expressed by the condition $R = 1$, where

$$R = (x_2 \vee x_3)(x_1 \vee x_2)(x_2 \vee x_4)(x_1 \vee x_3)(x_3 \vee x_4)(x_1 \vee x_4).$$

4. By the distributive law, and the absorption law, we have

$$R = x_1x_2x_4 \vee x_1x_2x_3 \vee x_2x_3x_4 \vee x_1x_3x_4.$$

5. Since every product has three literals, each corresponds to a minimum solution. Thus, f can be represented by 3 variables. Since no variable appears in all product terms, no variable is essential. ■

In principle, the above method produces the minimum number of variables to represent an incompletely specified index generation function. However, the straightforward application is quite inefficient. Also, we have an efficient minimization algorithm for SOPs, but do not have one for product-of-sums expressions. Thus, instead of obtaining R directly, first we obtain $\bar{R}$, the complement of R, and perform simplification, and then convert $\bar{R}$ into the SOP for R as follows:

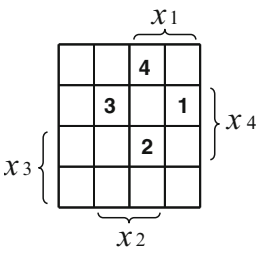


Fig. 11.5 Index generation function with 4 variables

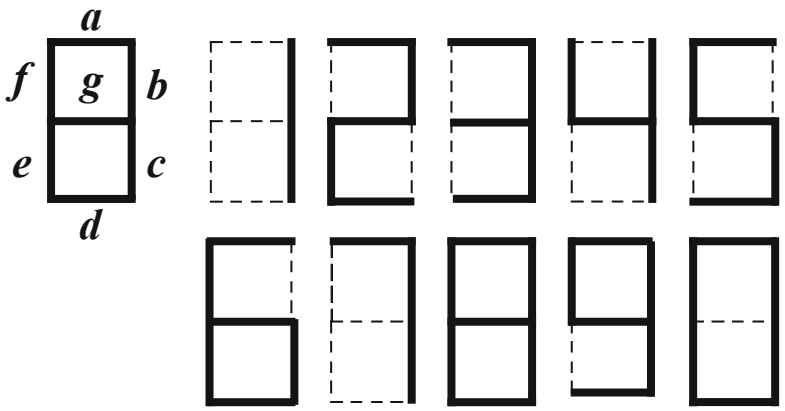


Fig. 11.6 7-segment display

Algorithm 11.3.1. *(Algebraic Method)*

1. *Let A be the set of vectors $\vec{a}_i$, such that $f(\vec{a}_i) = i$, where $i = 1, 2, \ldots, k$.*
2. *For each pair of vectors $\vec{a}_i = (a_1, a_2, \ldots, a_n) \in A$ and $\vec{b}_j = (b_1, b_2, \ldots, b_n) \in A$, associate a product defined by $s(i, j) = \bigwedge_{r=1}^{n} y_r$, where $y_r = 1$ if $a_r = b_r$ and $y_r = \bar{x}_r$ if $a_r \neq b_r$, where $r = 1, 2, \ldots, n$. Note that there are $k(k-1)/2$ pairs.*
3. *Define a covering function $\bar{R} = \bigvee_{i<j} s(i, j)$.*
4. *Represent $\bar{R}$ by the a minimum SOP.*
5. *Represent R, the complement of $\bar{R}$ by a minimum SOP.*
6. *The product with the fewest literals corresponds to the minimum solution.*

In Algorithm 11.3.1, Steps 4, 5, and 6 compute a minimum covering. Since $\bar{R}$ has only complemented literals, we generate only products with complemented literals. Applying absorption law yields the minimum SOP for $\bar{R}$.

By first detecting the essential variables, we can reduce the computational effort to derive the covering function. The next example illustrates this.

Example 11.3.2. The 7-segment display shown in Fig. 11.6 displays a decimal number by using 7 segments: a, b, c, d, e, f, and g.

Table 11.2 shows the correspondence between segment data and the binary number. Consider a logic circuit that converts 7-segment data into the corresponding

Table 11.2 7-segment to BCD converter

7-segment							BCD code			
a	b	c	d	e	f	g	8	4	2	1
0	1	1	0	0	0	0	0	0	0	1
1	1	0	1	1	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	1	1
0	1	1	0	0	1	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	1
1	0	1	1	1	1	1	0	1	1	0
1	1	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	0	1	1	1	0	0	1
1	1	1	1	1	1	0	1	0	1	0

Binary Coded Decimal (**BCD**) representation of a digit. The straightforward circuit requires 7 inputs. However, only 5 inputs are necessary to distinguish the decimal numbers. This means that only 5 segments are needed to distinguish between the 10 digits.

1. Let the vectors be
   $\vec{a}_1 = (0,1,1,0,0,0,0), \vec{a}_2 = (1,1,0,1,1,0,1), \vec{a}_3 = (1,1,1,1,0,0,1),$
   $\vec{a}_4 = (0,1,1,0,0,1,1), \vec{a}_5 = (1,0,1,1,0,1,1), \vec{a}_6 = (1,0,1,1,1,1,1),$
   $\vec{a}_7 = (1,1,1,0,0,0,0), \vec{a}_8 = (1,1,1,1,1,1,1), \vec{a}_9 = (1,1,1,1,0,1,1),$
   and
   $\vec{a}_{10} = (1,1,1,1,1,1,0).$
2. First, find the essential variables.
   From $\vec{a}_6$ and $\vec{a}_8$, we can see that b is essential. From $\vec{a}_8$ and $\vec{a}_9$, we can see that e is essential. From $\vec{a}_3$ and $\vec{a}_9$, we can see that f is essential. From $\vec{a}_8$ and $\vec{a}_{10}$, we can see that g is essential.
3. Next, we derive $\bar{R}$. Since b, e, f, and g are essential, we can ignore the pairs, where the essential variables are inconsistent. For example, from the pair $(\vec{a}_1, \vec{a}_2)$, we have the product $\bar{a}\bar{c}\bar{d}\bar{e}\bar{g}$. Note that, in this case, two vectors are inconsistent with the essential variable g. Since the essential variable g is always included in the solution, we know that $g = 1$. Thus, we need not generate it. From $\vec{a}_4$ and $\vec{a}_9$, we have $\bar{a}\bar{d}$. Thus, $\bar{R} = \bar{b} \vee \bar{e} \vee \bar{g} \vee \bar{f} \vee \bar{a}\bar{d}$.
4. By using De Morgan's law, and the distributive law, we have
   $R = befg(a \vee d) = abefg \vee bdefg.$
5. Since each product has five literals, each corresponds to a minimum solution. Thus, the binary numbers can be represented by 5 variables.

Thus, we can eliminate either segments c and d, or a and c, and still determine which digit is being represented. ■

11.4 Analysis for Single-Output Logic Functions

This section derives the number of variables to represent single-output incompletely specified logic functions. In the analysis that follows, we consider a set of functions (e.g., all incompletely specified functions) restricted by conditions (e.g. the number of *care* values is $2u$).

Definition 11.4.1. A set of functions is **uniformly distributed**, if the probability of occurrence of any function is the same as any other function.

For example, the set of 4-variable incompletely specified functions with 1 *care* value consists of 32 members, 16 having a single 1 and 16 having a single 0. If the functions are uniformly distributed, the probability of the occurrence of any one of them is $\frac{1}{32}$.

Theorem 11.4.1. *Consider a set of uniformly distributed incompletely specified functions, where u combinations are mapped to 0, u combinations mapped to 1, and the other* $2^n - 2u$ *combinations are mapped to don't cares. Let* η *be the probability that* $f(x_1, x_2, \ldots, x_n)$ *can be represented by using only* $x_1, x_2, \ldots, x_{p-1}$, *and* x_p, *where* $p < n$. *Then,* $\eta \geq (1 - \tilde{\alpha})^u$, *where* $\tilde{\alpha} = \frac{u}{2^p}$.

Proof. Let $f(X_1, X_2)$ be an incompletely specified function, where $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$. Consider the decomposition chart of $f(X_1, X_2)$, where X_1 labels the columns, and X_2 labels the rows. If no column has both 0 and 1, a completely specified function can be formed by setting all column entries to the same value, yielding a function independent of X_2. From here, we obtain the probability η.
Assume that u 0's are already distributed to the decomposition chart. Thus, at most u columns have 0's. Next, we distribute u 1's to the decomposition chart. The probability of distributing a single 1 to a column not containing 0's is at least $\frac{2^p - u}{2^p} = 1 - \tilde{\alpha}$. Thus, the probability of distributing u 1's to the columns without 0's is larger than or equal to $(1 - \tilde{\alpha})^u$. Hence, we have the relation:

$$\eta \geq (1 - \tilde{\alpha})^u.$$

□

Theorem 11.4.1 considers the probability for one partition: $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$. However, in practice, we can select a minimum set of variables to represent the function. The following theorem considers such a case:

Theorem 11.4.2. *Consider a set of uniformly distributed incompletely specified functions, where u combinations are mapped to 0, u combinations mapped to 1, and the other* $2^n - 2u$ *combinations are mapped to don't cares. Then, the probability that* $f(x_1, x_2, \ldots, x_n)$ *can be represented by using only p variables is at least*

$$1 - \sigma^{\binom{n}{p}},$$

where $\sigma = 1 - \eta$, and η is the probability that a function can be represented by using only $x_1, x_2, \ldots, x_{p-1}$, and x_p.

Proof. The probability that a function cannot be represented by using $x_1, x_2, \ldots, x_{p-1}$, and x_p is $\sigma = 1 - \eta$. Since there are $\binom{n}{p}$ ways to choose p variables out of n variables, the probability that a function cannot be represented by using any combinations of p variables is $\sigma^{\binom{n}{p}}$. The probability that a function can be presented by using at least one combination of p variables is

$$1 - \sigma^{\binom{n}{p}}.$$

□

From Theorem 11.4.2, we have the following:

Conjecture 11.4.1. Consider a set of uniformly distributed functions of n variables, where u combinations are mapped to 0, u combinations are mapped to 1, and the other $2^n - 2u$ combinations are mapped to *don't cares.* If

$$p \geq 2 \log_2 u - 2,$$

then more than 95% of the functions can be represented with p variables.

(Explanation supporting the Conjecture) Since $\sigma = 1 - \eta < 1.0$, $1 - \sigma^{\binom{n}{p}}$ approaches 1.0, as n increases. When $p < n$, $\binom{n}{p} \geq n(n-1)/2$. Assume that $n \geq 20$. The condition that $\sigma^{\binom{n}{p}} \leq 0.05$ is $\sigma < 0.984$. Thus, if $\eta \geq 0.0156$, then at least 95% of the functions can be realized with p products. When $\tilde{\alpha}$ is sufficiently small, $1 - \tilde{\alpha}$ is approximated by $\mathrm{e}^{-\tilde{\alpha}}$. Thus,

$$\eta \geq (1 - \tilde{\alpha})^u \simeq \mathrm{e}^{-\tilde{\alpha} u} = \mathrm{e}^{-\frac{u^2}{2^p}}.$$

When $p \geq 2 \log_2 u - 2$, we have $\eta > \mathrm{e}^{-4} = 0.0183$. (End of explanation)
From experimental results in Sect. 11.6 , we have the following:

Conjecture 11.4.2. Consider a set of uniformly distributed functions of n variables, where u combinations are mapped to 0, u combinations are mapped to 1, and the other $2^n - 2u$ combinations are mapped to *don't cares*. Then, the fraction of the functions represented with $p = 2\lceil \log_2 u \rceil - 2$ variables approaches 1.0, as n increases.

11.5 Extension to Multiple-Output Functions

In practical applications, many functions have multiple outputs, and the outputs values are different for different inputs. So, we now consider such a class of functions. First, we consider the class of index generation functions, which are special case of multiple-output functions. Then, we extend the theory to general multiple-output functions.

11.5.1 Number of Variables to Represent Index Generation Functions

This section derives the number of variables to represent an incompletely specified index generation function with k registered vectors. The basic idea is as follows: a function $f(X_1, X_2)$ is represented by a decomposition chart, where X_1 labels the columns and X_2 labels the rows. If each column has at most one *care* element, then the function can be represented by using only variables in X_1. The next example illustrates this.

Example 11.5.1. Table 11.3 shows a registered vector table consisting of 6 vectors. When no entry matches the input vector, the function produces 0. Consider the decomposition chart shown in Table 11.4. In Table 11.4, x_1, x_2, and x_3 specify the columns, and x_4 and x_5 specify the rows, and blank elements denote *don't cares*. Note that, in Table 11.4, each column has at most one *care* element. Thus, the function can be represented by only the column variables: x_1,x_2, and x_3. ■

From here, we obtain the probability of such a condition by a statistical analysis.

Theorem 11.5.1. *Consider a set of uniformly distributed incompletely specified index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k, where $2 \leq k < 2^{n-2}$. Let $\eta(k)$ be the probability that f can be represented with $x_1, x_2, \ldots,$ and x_p, where $p < n$. Then,*

$$\eta(k) \simeq \exp\left(-\frac{k^2}{2^{p+1}}\right). \tag{11.1}$$

Proof. Let (X_1, X_2) be a partition of the input variables X, where $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$. Consider the decomposition chart for

Table 11.3 Registered vector table

x_1	x_2	x_3	x_4	x_5	f
0	0	1	0	0	1
0	1	0	0	1	2
0	1	1	1	0	3
1	0	0	1	1	4
1	0	0	1	1	5
1	1	1	1	0	6

Table 11.4 Decomposition chart for $f(X_1, X_2)$

		0	0	0	0	1	1	1	1	x_3
		0	0	1	1	0	0	1	1	x_2
		0	1	0	1	0	1	0	1	x_1
0	0		1							
0	1			2			5			
1	0				3			6		
1	1					4				
x_5	x_4									

$f(X_1, X_2)$, where X_1 labels the column variables and X_2 labels the row variables. If each column has at most one *care* element, then f can be represented by using only X_1. Assume that k *care* elements are distributed in the decomposition chart. Then, the probability that each column has at most one *care* element is

$$\begin{aligned}\eta(k) &= \frac{2^p}{2^p} \cdot \frac{2^p - 1}{2^p} \cdot \frac{2^p - 2}{2^p} \cdot \cdots \cdot \frac{2^p - (k-1)}{2^p} \\ &= 1 \cdot \left(1 - \frac{1}{2^p}\right) \cdot \left(1 - \frac{2}{2^p}\right) \cdot \cdots \cdot \left(1 - \frac{k-1}{2^p}\right) \\ &= \prod_{i=0}^{k-1} \left(1 - \frac{i}{2^p}\right).\end{aligned}$$

That is, in such a distribution, '1' can be placed in any column, '2' can be placed in any column except that for '1', etc.

Next, $\eta(k)$ can be approximated as follows:

$$\begin{aligned}\eta(k) &\simeq \prod_{i=0}^{k-1} \exp\left(-\frac{i}{2^p}\right) = \exp\left(-\sum_{i=1}^{k-1} \frac{i}{2^p}\right) \\ &= \exp\left(-\frac{k(k-1)}{2 \times 2^p}\right) \simeq \exp\left(-\frac{k^2}{2^{p+1}}\right)\end{aligned}$$

□

The above theorem shows the case when the input variables are removed without considering the property of the function. In practice, we can remove the maximum number of nonessential variables by an optimization program.

Theorem 11.5.2. *Consider a set of uniformly distributed incompletely specified index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k, where $2 \leq k < 2^{n-2}$. The probability that f can be represented with $p < n$ variables is greater than*

$$1 - \sigma^{\binom{n}{p}},$$

where $\sigma = 1 - \eta(k)$, and $\eta(k)$ is the probability that f can be represented with $x_1, x_2, \ldots, x_{p-1}$, and x_p.

The proof of Theorem 11.5.2 is similar to that of Theorem 11.4.2. From Theorem 11.5.2, we have the following:

Conjecture 11.5.1. Consider a set of uniformly distributed incompletely specified index generation functions with weight k. If $p \geq 2\lceil \log_2(k+1) \rceil - 3$, then more than 95% of the functions can be represented with p variables.

From experimental results, we have the following:

Conjecture 11.5.2. Consider a set of uniformly distributed incompletely specified index generation functions with weight k. Then, the fraction of the functions represented with $p = 2\lceil \log_2(k+1) \rceil - 3$ variables approaches 1.0, as n increases.

Note that there exist functions that require more than $p = 2\lceil \log_2(k+1) \rceil - 3$ variables, as shown below. However, the fraction of such functions approaches 0.0, as n increases.

Example 11.5.2. Consider the n-variable incompletely specified index generation function f with weight $k = n + 1$:

$$\begin{aligned}
f(1,0,0,\ldots,0,0) &= 1\\
f(0,1,0,\ldots,0,0) &= 2\\
f(0,0,1,\ldots,0,0) &= 3\\
\vdots \qquad\qquad &\quad \vdots\\
f(0,0,0,\ldots,1,0) &= n-1\\
f(0,0,0,\ldots,0,1) &= n\\
f(0,0,0,\ldots,0,0) &= n+1\\
f(a_1,a_2,a_3,\ldots,a_{n-1},a_n) &= d \quad \text{(for other combinations).}
\end{aligned}$$

In this function, all the variables are essential, and no variable can be removed. ■

Theorem 11.5.3. *To represent an incompletely specified index generation function with weight k, at least $\lceil \log_2(k+1) \rceil$ variables are necessary.*

Proof. To distinguish $k+1$ outputs, at least $\lceil \log_2(k+1) \rceil$ variables are necessary. Note that one output is used to show that there is no matched vector. Nonzero outputs denote registered vectors, while zero outputs denote nonregistered vectors. □

11.5.2 Number of Variables to Represent General Multiple-Output Functions

Theorem 11.5.4. *Let F be an arbitrary n variable m output function, and let D be a set of k randomly selected vectors in B^n. Let $\hat{F}$ be an incompletely specified function defined on only D. The probability that $\hat{F}$ can be represented with $x_1, x_2, \ldots,$ and x_p, for $p < n$, is $\eta(k)$, where $\eta(k)$ is defined in (11.1).*

Proof. For each vector in D, assign a unique index in $\{1, 2, \ldots, k\}$. From D, we can define an incompletely specified index generation function: $D \to \{1, 2, \ldots, k\}$. Next, for each vector in D, obtain the output value of F, and make a truth table showing the function $\{1, 2, \ldots, k\} \to B^m$. Note that this function can be

implemented by memory with $\lceil \log_2(k+1) \rceil$ inputs. Thus, the incompletely specified function $\hat{F}$ can be realized as the cascade connection of the index generation circuit and memory.

By Theorem 11.5.1, the index generation function can be represented with at most p variables. Thus, the function $\hat{F}$ can be also represented with at most p variables. □

Similarly to Conjecture 11.5.2, we have the following:

Conjecture 11.5.3. When k minterms are selected randomly, the fraction of multiple-output functions with only k minterms that can be represented by at most $p = 2\lceil \log_2(k+1) \rceil - 3$ variables approaches 1.0, as n increases.

11.6 Experimental Results

11.6.1 Random Single-Output Functions

For different n, we randomly generated 1,000 functions, where u combinations are mapped to 0, u combinations are mapped to 1, and the other $2^n - 2u$ combinations are mapped to *don't cares*. We minimized the number of variables by an exact optimization algorithm, which is similar to Algorithm 11.3.1 shown in Sect. 11.3.

Table 11.5 shows the average numbers of variables to represent the single-output functions, where the set of variables are selected by the optimization algorithm. For example, 16-variable functions where 15 minterms are mapped to zeros, 15 minterms are mapped to ones, and the other minterms are mapped to *don't cares*, require, on the average, only 5.157 variables to represent the functions.

Table 11.5 shows that the necessary number of variables to represent the functions mainly depends on u. The last column of the table shows the number of variables to represent incompletely specified functions by Conjecture 11.4.2. For

Table 11.5 Average numbers of variables to represent single-output logic functions with u 1's and u 0's

u	$n = 16$	$n = 20$	$n = 24$	$2\lceil \log_2(u+1) \rceil - 2$
7	3.334	3.145	3.017	4
15	5.157	4.981	4.940	6
31	7.126	6.980	6.003	8
63	9.179	8.972	8.861	10
127	11.362	10.971	10.776	12
255	13.754	12.990	12.725	14
511	15.739	15.098	14.805	16
1023	16.000	17.508	16.918	18
2047	16.000	19.705	18.996	20
4095	16.000	20.000	21.394	22
8191	16.000	20.000	23.630	24

Table 11.6 Average number of variables to represent incompletely specified index generation function

k	$n = 16$	$n = 20$	$n = 24$	$2\lceil \log_2(k+1) \rceil - 3$
7	3.052	3.018	3.003	3
15	4.980	4.947	4.878	5
31	6.447	6.115	6.003	7
63	8.257	8.007	8.000	9
127	10.304	10.000	9.963	11
255	12.589	11.996	11.896	13
511	14.890	14.019	13.787	15
1023	15.991	16.293	15.874	17
2047	16.000	18.758	17.965	19
4095	16.000	19.992	20.093	21

example, when $u = 15$, to represent a uniformly distributed function, Conjecture 11.4.2 shows that 6 variables are sufficient. On the other hand, experimental results show that only 4, 5, or, 6 variables are necessary to represent the functions. We note that the variance is very small.

11.6.2 Random Index Generation Functions

We generated uniformly distributed index generation functions. Table 11.6 shows the average numbers of variables to represent n-variables index generation functions with k registered vectors. For the other $2^n - k$ combinations, the outputs are set to *don't cares*. The values are the average of 1,000 randomly generated functions. Table 11.6 shows that the necessary number of variables to represent the functions strongly depends on k.

The last column of Table 11.6 shows the number of variables to represent incompletely specified index generation functions with weight k given by Conjecture 11.5.2. For example, when $k = 31$, to represent a uniformly distributed function, Conjecture 11.5.2 shows that 9 variables are sufficient. On the other hand, experimental results show that only 6 or 7 variables are necessary to represent the functions. Again, the variance is very small.

11.6.3 IP Address Table

To verify the effectiveness of the method in a practical application, we used distinct IP addresses of computers that accessed our web site over a period of a month. We considered four lists of different sizes: List 1, List 2, List 3, and List 4. Table 11.7 shows the results. The first row shows the number of registered vectors: k. The second row shows the number of inputs: n. The third row shows the number of outputs: $q = \lceil \log_2(k+1) \rceil$. The fourth row shows the number of variables sufficient to represent the functions given by Conjecture 11.5.2, i.e., $2\lceil \log_2(k+1) \rceil - 3$. The fifth

Table 11.7 Realization of IP address tables

	List 1	List 2	List 3	List 4
# of vectors: k	1,670	3,288	4,591	7,903
# of inputs: n	32	32	32	32
# of outputs: q	11	12	13	13
$2\lceil \log_2(k+1) \rceil - 3$	19	21	23	23
# of variables using Single-input hash: n_s	18	20	21	23
# of variables using Double-input hash: n_d	17	19	20	21
Single-memory realization ($\times 10^{10}$ bits)	4.72	5.15	5.58	5.58
Realization using Single-input hash ($\times 10^6$ bits)	2.95	12.7	27.5	109.3
Realization using Double-input hash ($\times 10^6$ bits)	1.51	6.3	13.9	27.5

row shows the number of variables to represent the function, where the number of variables was minimized by Algorithm 11.3.1. In this case, selected input variables are connected to the main memory through the single-input hash circuit shown Fig. 10.3. The sixth row shows the number of variables to represent the function, where a linear transformation is used to reduce the number of variables. In this case, the double-input hash circuit shown in Fig. 10.2 is used. The seventh row shows the number of bits to represent the function by a single memory: $q2^n$. The eighth row shows the total number of bits to represent the function by using the single-input hash circuit shown in Fig. 10.3: $q2^{n_s} + n2^q$, where the first term denotes the size of the main memory, while the second term denotes the size of the AUX memory. The last row shows the total number of bits needed to represent the function by using the double-input hash circuit shown in Fig. 10.2: $q2^{n_d} + n2^q$. As shown in Table 11.7, the total amount of memory can be drastically reduced.

11.6.4 Benchmark Multiple-Output Functions

We reduced the number of variables for selected PLA benchmark functions [159]. Table 11.8 shows the numbers of variables to represent benchmark functions (*bc0, chkn, in2, in7, intb*, and *vg2*) for different values of *care* minterms k. In the table, n denotes the number of original input variables, q denotes the number of outputs, and W denotes the number of products in the PLA. The rightmost column shows the upper bound derived by Conjecture 11.5.3: $2\lceil \log_2(k+1) \rceil - 3$. Out of 2^n combinations, we randomly selected k different combinations as *care* minterms, and set other $2^n - k$ minterms to *don't cares*. Then, we minimized the number of variables. From the table, we observe that the number of variables strongly depends on k, but is virtually independent of n, q, W, and the function name. Again, for these benchmark functions, the upper bounds on the number of products given by the Conjecture 11.5.3 are valid.

Table 11.8 Number of variables needed to represent incompletely specified multiple-output PLA benchmark functions

k	*bc*0	*chkn*	*in*2	*in*7	*intb*	*vg*2	Conj. 11.5.3
	$n = 26$	$n = 29$	$n = 19$	$n = 27$	$n = 15$	$n = 25$	
	$q = 11$	$q = 7$	$q = 7$	$q = 10$	$q = 7$	$q = 8$	Upper
	$W = 179$	$W = 142$	$W = 135$	$W = 55$	$W = 631$	$W = 110$	bound
15	4	4	4	4	5	4	5
31	6	5	6	6	7	6	7
63	8	7	7	8	8	7	9
127	9	9	8	8	10	9	11
255	11	10	9	10	12	11	13
511	12	12	10	12	14	13	15
1023	14	14	13	13	15	14	17

11.7 Remarks

For incompletely specified index generation functions, reduction of the number of variables is quite effective. Combined with the hash method presented in the previous chapter, this method drastically reduces the amount of memory to implement the function. An extension to multi-valued input functions is considered in [137]. This chapter is based on [111, 131, 132, 137].

Problems

11.1. Minimize the number of variables for the incompletely specified logic function whose characteristic functions are:

$$F_0 = \bar{x}_5\{\bar{x}_1 x_2(x_3 \vee \bar{x}_4) \vee x_1 x_2(x_3 \oplus x_4)\} \vee x_5\{x_2\bar{x}_3 x_4 \vee \bar{x}_1\bar{x}_3\bar{x}_4 \vee \bar{x}_1 x_2\bar{x}_3\}.$$

$$F_1 = \bar{x}_5(\bar{x}_1\bar{x}_2 x_3 \vee x_1\bar{x}_3\bar{x}_4 \vee x_1\bar{x}_2 x_4) \vee x_5(x_1\bar{x}_2\bar{x}_3 \vee x_1\bar{x}_2\bar{x}_4 \vee x_1 x_2 x_3 x_4 \vee \bar{x}_1\bar{x}_2 x_3 x_4).$$

11.2. Consider the incompletely specified function of 8 variables in Table 11.9. Minimize the number of variables.

11.3. Consider a binary matrix of 8 columns and 7 rows, where 0's and 1's are distributed uniformly.

1. Calculate the probability that all the rows are distinct.
2. Remove the first column. Calculate the probability that all the rows are distinct.

11.4. Let $n = 10$ and $k = 31$. Consider a decomposition chart of an index generation function $f(X_1, X_2)$ with weight k, where $X_1 = (x_1, x_2, \ldots, x_9)$ and $X_2 = (x_{10})$. Calculate the probability that each column has at most one nonzero element. Also, calculate the probability, among the 10 partitions, where $X_1 = X - x_i$,

Table 11.9 Incompletely specified function of 8 variables

	a	b	c	d	e	f	g	h	F
v_1	0	0	1	0	0	0	0	0	0
v_2	0	1	0	0	0	0	1	0	0
v_3	0	0	1	0	0	1	1	0	0
v_4	0	0	0	1	0	0	1	0	1
v_5	0	1	0	0	1	1	0	0	1
v_6	1	0	0	1	0	1	0	0	1

Table 11.10 4-Valued input index generation function

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	f
A	A	G	A	G	C	T	A	1
A	A	G	C	A	C	G	C	2
G	A	A	G	A	T	C	A	3
C	T	G	G	A	G	G	G	4
T	A	G	G	G	A	T	A	5
T	A	T	G	C	C	A	G	6
T	G	A	C	C	G	C	G	7

$X_2 = (x_i)$, and $X = (x_1, x_2, \ldots, x_{10})$ for $(i = 1, 2, \ldots, 10)$, there exist at least one partition, in which each column has at most one nonzero element. It is suggested that you use a computer or a calculator to obtain this value.

11.5. Consider a set of uniformly distributed incompletely specified index generation functions of n variables with weight k. Derive the probability that all the variables are essential, for $n = 2r$ and $k = 2^r$.

11.6. Show a 4-variable incompletely specified index generation function satisfying the following conditions:

1. Four combinations are mapped to 1.
2. Four combinations are mapped to 0.
3. All other 8 combinations are mapped to *don't cares*.
4. All the variables are essential.

11.7. The four bases found in deoxyribonucleic acid (DNA) are adenine (abbreviated A), cytosine (C), guanine (G), and thymine (T). Consider the DNA patterns shown in Table 11.10. Find the minimum set of variables to distinguish these patterns.

11.8. Consider a set of uniformly distributed, incompletely specified index generation functions $f : D \to I$, where $B = \{0, 1\}$, $D \subset B^n$ and $I = \{1, \ldots, k\}$. Then, the probability that f can be represented with only $x_1, x_2, \ldots, x_{p-1}$ and x_p, where $p < n$ is $\delta_{n-p} = \gamma_{n-p}^{M}$, where $\gamma_{n-p} = \beta^{2^{n-p}} + 2^{n-p}\alpha\beta^{2^{n-p}-1}$, $\alpha = \frac{k}{2^n}$, $\beta = 1 - \alpha$, and $M = 2^p$. Prove this.

11.9. Consider a set of uniformly distritbuted index generation functions $f : B^n \to I$, where $B = \{0, 1\}$ and $I = \{0, 1, \ldots, k\}$. Let PR be the probability that $f(x_1, x_2, \ldots, x_n)$ can be represented by using only p variables. Then

$$PR = 1 - (1 - \delta_{n-p})^{\binom{n}{p}},$$

where δ_{n-p} is the probability that $f(x_1, x_2, \ldots, x_n)$ can be represented by using only $x_1, x_2, \ldots, x_{p-1}$, and x_p. Compute the numerical values of PR when $n = 20$ and $k = 2047$, for $p = 17, 18$, and 19. You can use the results of the previous problem.

11.10. Consider a group of 64 people. Obtain the probability that the birthdays of all the people are distinct. Assume that the probabilities of the birth are the same for all 365 days in a year. Calculate the expected number of distinct birthdays in the group of 64 people.

11.11. Assume that the probabilities of birth are the same for all 365 days in a year. Calculate the expected number of distinct birthdays in a room with 365 people.

Chapter 12
Various Realizations

This chapter shows various realizations of index generation functions.

12.1 Realization Using Registers, Gates, and An Encoder

An index generator can be directly implemented using a Programmable Logic Array (PLA) or a Content Addressable Memory (CAM). An index generator can also be implemented using ordinary logic elements. Figure 12.1 [98] shows an index generator for an LPM implemented by registers, gates, and a priority encoder. A register pair (Reg. 1 and Reg. 0) is used to store each digit of a ternary vector. For example, if the digit is * (*don't care*), the register pair stores (1,1). Thus, for n bit data, we need a $2n$-bit register. The comparison circuit consists of an n-input AND gate and n comparison circuits, each of which produces a 1 if and only if the input bit matches the stored bit or the stored bit is *don't care* (* or 11).

For each prefix vector of an n-input LPM index generator, we need a $2n$-bit register, n comparison circuits, and an n-input AND gate. For an n-input index generator with k registered prefix vectors, we need k registers of $2n$ bits, nk comparison circuits, and k AND gates with n inputs. In addition, we need a priority encoder with k inputs and $\lceil \log_2(k+1) \rceil$ outputs to generate the LPM address.

The demerit of this circuit is that it becomes complex as the number of registered vectors increases.

12.2 LUT Cascade Emulator

In an LUT cascade, once the number of inputs and outputs for each cell are fixed, only a limited class of functions can be realized. Thus, the **LUT cascade emulator**[1] having the architecture shown in Fig. 12.2 has been developed [96]. It consists

[1] In some publications [113], the emulator was called an LUT cascade. However, later the sequential circuit that emulates an LUT cascade is called an LUT cascade emulator.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_12,

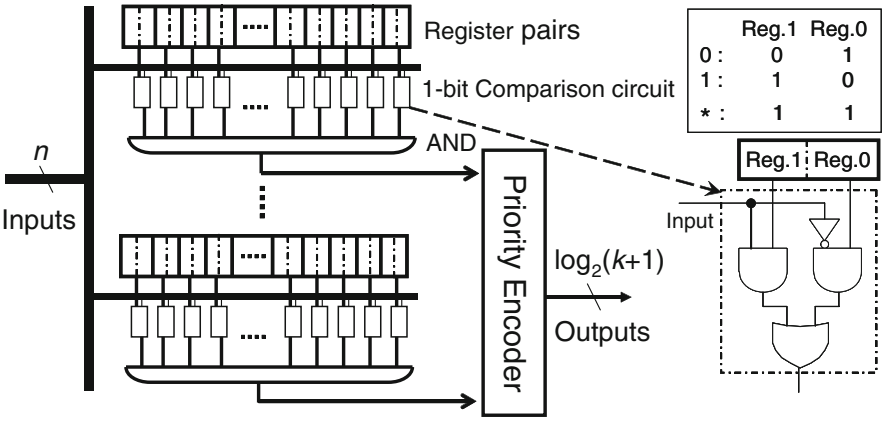


Fig. 12.1 Index generator implemented using registers, gates, and a priority encoder

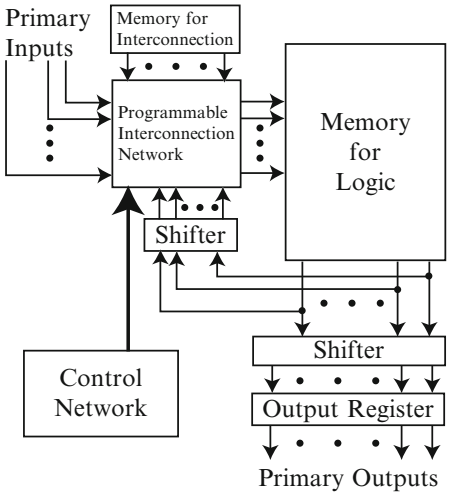


Fig. 12.2 LUT cascade emulator

of a large memory that stores the data for cells, a programmable interconnection network, and a control circuit. It emulates an LUT cascade sequentially. That is, the outputs of the various LUTs are produced in sequence starting with the leftmost LUT. Although the emulator is slower than the LUT cascade, it is more flexible than the LUT cascade.

A shifter that drives the programmable interconnection network is used for memory packing [118], while a shifter that drives the output register is used to accumulate the outputs.

Example 12.2.1. Let us emulate the LUT cascade with four cells shown in Fig. 12.3 by the emulator [123].

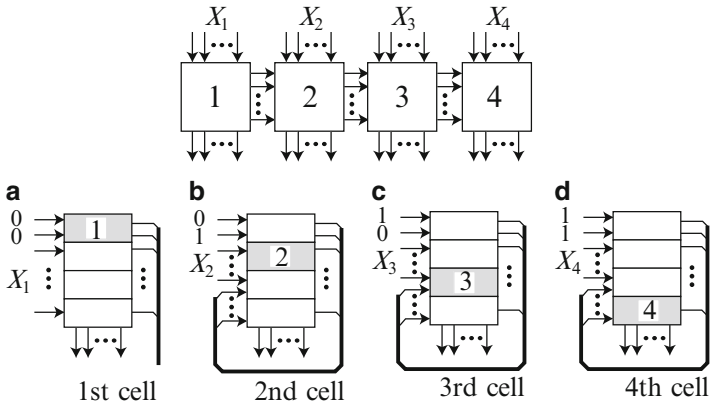


Fig. 12.3 Operation of an LUT cascade emulator

Step 1 To emulate $Cell_1$, the two most significant bits of the address for memory for logic are set to $(0, 0)$ to specify the first page of the memory for logic, which corresponds to the first cell. Also, the values of X_1 are set to the lower address bits of the memory for logic, as shown in Fig. 12.3a. This is done through the programmable interconnection network in Fig. 12.2. By reading the content of the first page, we obtain the outputs of $Cell_1$.

Step 2 To emulate $Cell_2$, the two most significant bits of address are set to $(0, 1)$ to specify the second page. Also, the values of X_2 are set to the middle address bits, and the outputs of $Cell_1$ are connected to the least significant bits through the programmable interconnection network, as shown in Fig. 12.3b. By reading the content of the second page, we obtain the outputs of $Cell_2$.

Step 3 To emulate $Cell_3$, the two most significant bits of address are set to $(1, 0)$ to specify the 3rd page. Also, the values of X_3 are set to the middle address bits, and the outputs of $Cell_2$ are connected to the least significant bits through the programmable interconnection network, as shown in Fig. 12.3c. By reading the content of the 3rd page, we obtain the outputs of $Cell_3$.

Step 4 To emulate $Cell_4$, the two most significant bits of address are set to $(1, 1)$ to specify the last page. Also, the values of X_4 are set to the middle address bits, and the outputs of $Cell_3$ are connected to the least significant bits through the programmable interconnection network, as shown in Fig. 12.3d. By reading the content of the 4th page, we obtain the outputs of $Cell_4$. At this point, the outputs of all LUTs have been obtained, and the complete circuit's output is specified. ■

12.3 Realization Using Cascade and AUX Memory

Here, we show a method to reduce hardware by using an auxiliary memory and a comparator. Figure 12.4 illustrates the idea of the method.

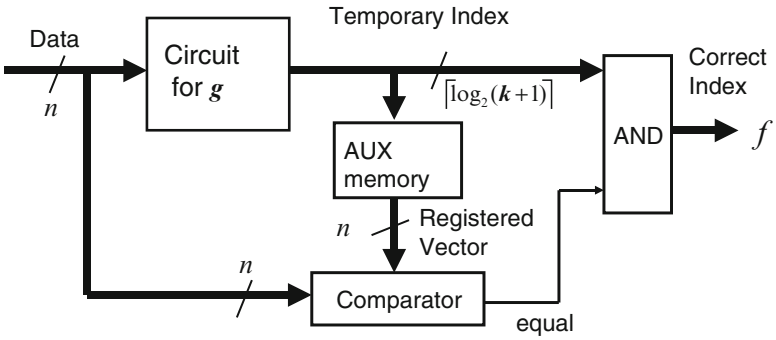


Fig. 12.4 Index generator using auxiliary memory

Algorithm 12.3.1. *(Simplification of Index Generators).*

1. *Let f be an index generation function. Let g be the function where the output values for the nonregistered inputs in f are replaced by don't cares.*
2. *Reduce the number of variables to represent g. Produce the circuit for g. In general, the circuit for g is simpler than the circuit for f.*
3. *When the query data matches a registered vector, the circuit g produces correct outputs. When the query data does not match any registered vector, the circuit for g may produce wrong output values.*
4. *To detect the correct outputs, we use an* **auxiliary (AUX) memory** *with $q = \lceil \log_2(k+1) \rceil$ inputs and n outputs. The AUX memory stores corresponding registered vector for each address.*
5. *Apply the output address of the circuit for g to the AUX memory, and read out the registered vector in the AUX memory. If the output vector of the AUX memory equals to the input vector, then the circuit for g produces the correct output value. If the output vector of the AUX memory is different from the input vector, then the input vector is not registered. In this case, the comparator sends 0 to the AND gate. Thus, the circuit produces 0.*

An ordinary logic circuit can be simplified by *don't cares* [110]. The present method has the following features:

- The number of nonzero outputs (k) of the index generation function is much smaller than the total number of input combinations 2^n. In g, the outputs for the nonregistered inputs are set to don't cares.
- To verify the correctness of the output of the circuit for g, we use the AUX memory.

The total amount of hardware is smaller than the direct implementation of f. In logic synthesis using memories, reduction of support variables is important. In the index generation functions, the fraction of *don't cares* is very large, and we can often reduce the number of support variables. Details are shown in Chap. 11.

Example 12.3.1. (LUT Cascade and LUT Memory) Let us design the index generator for the registered vector table shown in Table 12.1. Since the number of the

Table 12.1 Registered vector table

Index	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
1	0	0	1	0	0	1	0	1	1	0	0
2	0	0	1	1	1	1	1	0	1	0	1
3	0	0	1	1	1	1	1	0	1	1	0
4	0	1	0	0	1	1	0	0	1	0	1
5	0	1	0	0	1	1	0	1	1	1	1
6	1	0	0	0	0	1	0	0	1	0	0
7	1	0	0	0	1	1	0	0	1	0	1
8	1	0	0	0	1	1	0	1	0	0	1
9	1	0	0	0	1	1	0	1	1	1	1
10	1	0	1	0	0	0	0	1	0	0	1
11	1	0	1	0	0	1	0	0	1	0	0
12	1	1	0	0	0	1	1	0	1	0	1
13	1	1	0	0	1	0	0	1	1	1	1
14	1	1	0	1	0	0	0	0	0	0	0
15	1	1	0	1	0	0	0	1	0	0	1

Table 12.2 Reduced registered vector table

Index	x_1	x_2	x_3	x_8	x_{10}	x_{11}
1	0	0	1	1	0	0
2	0	0	1	0	0	1
3	0	0	1	0	1	0
4	0	1	0	0	0	1
5	0	1	0	1	1	1
6	1	0	0	0	0	0
7	1	0	0	0	0	1
8	1	0	0	1	0	1
9	1	0	0	1	1	1
10	1	0	1	1	0	1
11	1	0	1	0	0	0
12	1	1	0	0	0	1
13	1	1	0	1	1	1
14	1	1	0	0	0	0
15	1	1	0	1	0	1

registered vectors is 15, the index generator has 4 outputs. It has 11 inputs, as well. Let g be a function, where the output values for the nonregistered input vectors are replaced by *don't cares*. By reducing the number of variables, g can be represented with only six variables $\{x_1, x_2, x_3, x_8, x_{10}, x_{11}\}$. This can be obtained by Algorithm 11.3.1 with the help of a computer program. Sufficiency can be verified by Table 12.2, where all the reduced vectors are different. Thus, six bits are sufficient to distinguish the indices. Also, the set is minimal. That is, deletion of any variable makes at least one pair of indices indistinguishable. Next, realize the reduced function g by an LUT cascade. Figure 12.5 shows an index generator using the AUX memory. Note that, in Fig. 12.5, only six variables $\{x_1, x_2, x_3, x_8, x_{10}, x_{11}\}$ are used as the inputs for the LUT cascade. We designed the cascade with 5-LUTs. Table 12.3 shows the content of the AUX memory, where the values for $\{x_4, x_5, x_6, x_7, x_9\}$ are stored.

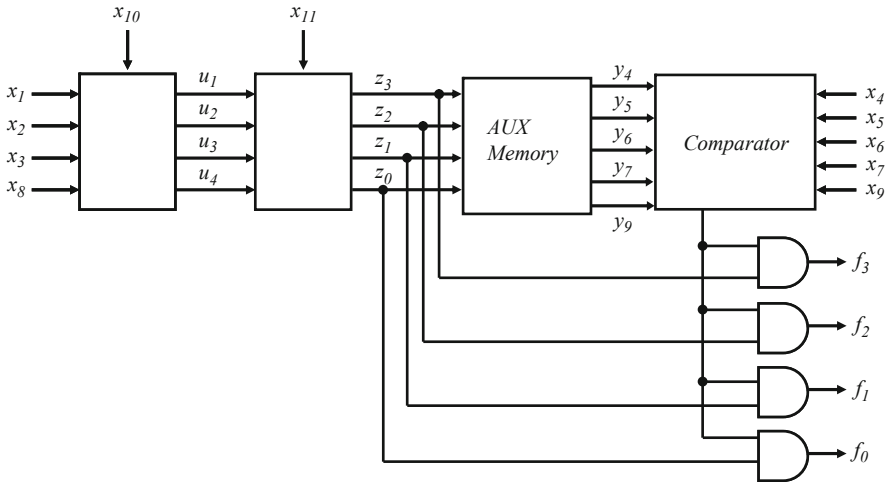


Fig. 12.5 Index generator using auxiliary memory

Table 12.3 Truth table for auxiliary memory

z_3	z_2	z_1	z_0	y_4	y_5	y_6	y_7	y_9
0	0	0	1	0	0	1	0	1
0	0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1	1
0	1	1	0	0	1	1	0	1
0	1	0	1	0	1	1	0	1
0	1	1	0	0	0	1	0	1
0	1	1	1	0	1	1	0	1
1	0	0	0	0	1	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	0	0	0	0	0	0
1	0	1	1	0	0	1	0	1
1	1	0	0	0	0	1	1	1
1	1	0	1	0	1	0	0	1
1	1	1	0	1	0	0	0	0
1	1	1	1	1	0	0	0	0

The second cell realizes the **temporary index** (z_3, z_2, z_1, z_0). This index is used to read the AUX memory. The output of the AND memory $(y_4, y_5, y_6, y_7, y_9)$ is compared with the input values $(x_4, x_5, x_6, x_7, x_9)$. If they agree, the temporary index is correct, then $(f_3, f_2, f_1, f_0) = (z_3, z_2, z_1, z_0)$ is the output. Otherwise, the input query data is not in the AUX memory, and $(f_3, f_2, f_1, f_0) = (0, 0, 0, 0)$ is produced at the output.

Figure 12.5 shows the circuit. The total memory is

$$32 \times 4 \times 2 + 16 \times 5 = 256 + 80 = 336$$

bits. Tables 12.4 and 12.5 are truth tables for the cells in the cascade. ■

Table 12.4 Truth table for the first cell

Index	x_1	x_2	x_3	x_8	x_{10}	u_1	u_2	u_3	u_4
1	0	0	1	1	0	0	0	0	1
2	0	0	1	0	0	0	0	1	0
3	0	0	1	0	1	0	0	1	1
4	0	1	0	0	0	0	1	0	0
5	0	1	0	1	1	0	1	0	1
6	1	0	0	0	0	0	1	1	0
7	1	0	0	0	0	0	1	1	0
8	1	0	0	1	0	1	0	0	0
9	1	0	0	1	1	1	0	0	1
10	1	0	1	1	0	1	0	1	0
11	1	0	1	0	0	1	0	1	1
12	1	1	0	0	0	1	1	0	0
13	1	1	0	1	1	1	1	0	1
14	1	1	0	0	0	1	1	0	0
15	1	1	0	1	0	1	1	1	1

Table 12.5 Truth table for the second cell

Index	u_1	u_2	u_3	u_4	x_{11}	z_3	z_2	z_1	z_0
1	0	0	0	1	0	0	0	0	1
2	0	0	1	0	1	0	0	1	0
3	0	0	1	1	0	0	0	1	1
4	0	1	0	0	1	0	1	0	0
5	0	1	0	1	1	0	1	0	1
6	0	1	1	0	0	0	1	1	0
7	0	1	1	0	1	0	1	1	0
8	1	0	0	0	1	1	0	0	0
9	1	0	0	1	1	1	0	0	1
10	1	0	1	0	1	1	0	1	0
11	1	0	1	1	0	1	0	1	1
12	1	1	0	0	1	1	1	0	0
13	1	1	0	1	1	1	1	0	1
14	1	1	0	0	0	1	1	0	0
15	1	1	1	1	1	1	1	1	1

12.4 Comparison of Various Methods

Example 12.4.1. Estimate the amount of hardware to implement index generation functions for

1. $n = 10$ and $k = 500$. Use a single LUT.
2. $n = 10$ and $k = 15$. Use an LUT cascade.
3. $n = 48$ and $k = 100$. Use the hybrid method.

4. $n = 48$ and $k = 1{,}000$. Use the super hybrid method.
5. $n = 32$ and $k = 500{,}000$. Use the standard parallel sieve method.

(Solution)

1. **Single LUT.**
   When $n = 10$ and $k = 500$. The number of inputs for the memory is $n = 10$, and the number of outputs is $q = \lceil \log_2(500 + 1) \rceil = 9$. Thus, the size of the memory is $2^{10} \times 9 = 9$ Kibits, where 1 Kibit denotes 2^{10} bits.
2. **LUT Cascade.**
   When $n = 10$ and $k = 15$. Consider an LUT cascade with $(K = 6)$-input LUTs. The number of inputs is $n = 10$, the number of rails is $w = \lceil \log_2(15 + 1) \rceil = 4$, and the number of outputs is $m = w = 4$. The number of cells is

$$s = \left\lceil \frac{n - w}{K - w} \right\rceil = \left\lceil \frac{10 - 4}{6 - 4} \right\rceil = \frac{6}{2} = 3.$$

   The total amount of memory is

$$2^6 \times 4 \times 3 = 3 \times 2^8 = 0.75 \times 2^{10},$$

   or, 0.75 Kibits.
3. **Hybrid Method.**
   When $n = 48$, $k_1 = 100$. $q = \lceil \log_2(100 + 1) \rceil = 7$. In this case, an LUT cascade would be too large, and so we use the hybrid method. Let the number of inputs to the main memory be $p = q + 2 = 9$. In this case, by Corollary 10.5.1, the fraction of remaining registered vectors is

$$\gamma_1 = 1 - \delta_1 = \frac{\xi - 1 + \mathrm{e}^{-\xi}}{\xi}.$$

   Since $p = 9$ and $k_1 = 100$, we have $\xi = 0.1953$, and

$$\gamma_1 = 1 - 0.9084 = 0.0916.$$

   Thus, the number of remaining vectors is $\gamma_1 k_1 \simeq 9$, which can be implemented by an LUT cascade or a rewritable PLA.
   The sizes of memories are as follows:

   Main memory: 9-input, 7-outputs: $2^9 \times 7 = 3.5 \times 2^{10} = 3.5$ Kibits.
   AUX memory: 7-input, 41-outputs: $2^7 \times 41 = 5.1 \times 2^{10} = 5.1$ Kibits.

   Thus, the total memory size is 8.6 Kibits.
4. **Super Hybrid Method.**
   When $n = 48$, $k_1 = 1{,}000$. $q_1 = \lceil \log_2(1000 + 1) \rceil = 10$. In the hybrid method, the remaining vector is 10% of the original vectors. That is, 100, which is fairly large. Thus, we use the super hybrid method. In the super hybrid method, we use

the first main memory with $p_1 = q_1 + 1 = 11$ inputs, and $q_1 = 10$ outputs. The fraction of vectors not realized by the first IGU is

$$\gamma_1 = 1 - \delta_1 = \frac{\xi_1 - 1 + e^{-\xi_1}}{\xi_1}.$$

When, $k_1 = 1{,}000$ and $p_1 = 11$, we have $\xi_1 = 0.48828$ and $\gamma_1 = 0.2088$. The number of remaining vectors is $k_2 = k_1\gamma_1 \simeq 209$. $q_2 = \lceil \log_2(209 + 1) \rceil = 8$.

The second main memory has $p_2 = q_2 + 1 = 8 + 1 = 9$ inputs and $q_2 = 8$ outputs. The fraction of vectors not realized by the second IGU is

$$\gamma_2 = 1 - \delta_2 = \frac{\xi_2 - 1 + e^{-\xi_2}}{\xi_2}.$$

When, $k_2 = 209$ and $p_2 = 9$, we have $\xi_2 = 0.398437$, and $\gamma_2 \simeq 0.1752$. Thus, the number of remaining vectors is $k_3 = k_2\gamma_2 \simeq 36$, which can be implemented by an LUT cascade or a rewritable PLA.
The sizes of memories are as follows:

first main memory: 11-input, 10-outputs: $2^{11} \times 10 = 20$ Kibits.
first AUX memory: 10-input, 37-outputs: $2^{10} \times 37 = 37$ Kibits.
second main memory: 9-input, 8-outputs: $2^9 \times 8 = 4 \times 2^{10} = 4$ Kibits.
second AUX memory: 8-input, 39-outputs: $2^8 \times 39 = 9.75$ Kibits.

Thus, the total memory size is 70.75 Kibits.

5. **Standard Parallel Sieve Method.**
When $k_1 = 500{,}000$, we have $q_1 = \lceil \log_2(500,000 + 1) \rceil = 19$. In the super hybrid method, the remaining vector is 4% of the original vectors. That is, 20000, which is very large. Thus, we use the standard parallel sieve method. In the standard parallel sieve method, the first main memory has $p_1 = q_1 = 19$ inputs and $q_1 = 19$ outputs. The fraction of vectors not realized by the first IGU is

$$\gamma_1 = 1 - \delta_1 = \frac{\xi_1 - 1 + e^{-\xi_1}}{\xi_1}$$

When $k_1 = 500{,}000$ and $p_1 = 19$, we have $\xi_1 = 0.953674$ and $\gamma_1 = 0.35546$. Thus, the number of remaining vectors is $k_2 = k_1\gamma_1 \simeq 177,733.$ and $q_2 = \lceil \log_2(177733 + 1) \rceil = 18$.

The second main memory has $p_2 = q_2 = 18$ inputs and $q_2 = 18$ outputs. The fraction of vectors not realized by the second IGU is

$$\gamma_2 = 1 - \delta_2 = \frac{\xi_2 - 1 + e^{-\xi_2}}{\xi_2}.$$

When, $k_2 = 177,733$ and $p_2 = 18$, we have $\xi_2 = 0.6779976$, and the number of remaining vectors is $k_3 = k_2\gamma_2 \simeq 48662$.

In the similar way, we have

$$\begin{aligned} p_3 &= 16,\ k_4 \simeq 14,316. \\ p_4 &= 14,\ k_5 \simeq 4,771. \\ p_5 &= 13,\ k_6 \simeq 1,155. \\ p_6 &= 11,\ k_7 \simeq 273. \\ p_7 &= 9,\quad k_8 \simeq 62. \end{aligned}$$

Thus, the number of remaining vectors is $k_8 \simeq 62$, which can be implemented by an LUT cascade or a rewritable PLA.

Note that for each IGU_i, the main memory has p_i inputs and p_i outputs, while the AUX memory has p_i inputs and $(n - p_i)$ outputs. Thus, the total amount of memory is

$$p_i 2^{p_i} + (n - p_i)2^{p_i} = n2^{p_i}.$$

So, the total amount of memory for the parallel sieve method is

$$\sum_{i=1}^{r} n2^{p_i} = 32 \cdot (2^{19} + 2^{18} + 2^{16} + 2^{14} + 2^{13} + 2^{11} + 2^9).$$

It is about 24-Mibit, where 1 Mibit denotes 2^{20} bits. ■

Example 12.4.2. Consider a system that detects computer viruses. A complete system using only hardware is too complex, so we use two-stage method: In the first stage, suspicious patterns are detected by hardware, and in the second stage, a complete match is performed by software only for the patterns detected in the first stage. Here, we consider the hardware part in the first stage. Assume that we check the text using a window of four characters, and the number of suspicious patterns is $k = 500{,}000$. Fig. 12.6 shows the circuit to detect the suspicious patterns. Eight 4-stage shift registers are used to store four characters. These registers work as a window. Note that the number of inputs to the memory is $4 \times 8 = 32$, and the number of outputs is $\lceil \log_2(k+1) \rceil = 19$.

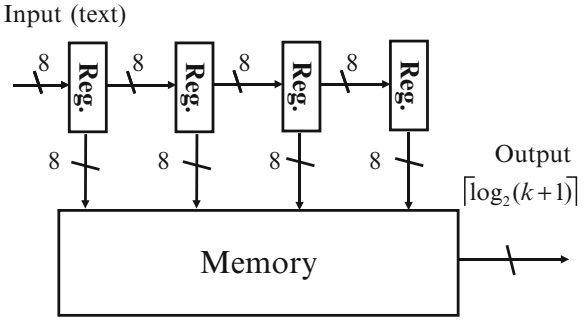


Fig. 12.6 Virus scanning circuit

A straightforward implementation requires a memory with impractical size: $\lceil \log_2(k+1) \rceil 2^{32} = 76$ Gibits, where 1 Gibit denotes 2^{30} bits. If we use the standard parallel sieve method shown in Example 12.4.1, we need only 24 Mibits. ■

12.5 Code Converter

In this part, we consider a class of code converters that can be treated as index generation functions.

Definition 12.5.1. An **m-out-of-n code** consists of $\binom{n}{m}$ binary code words whose weights are m.

Definition 12.5.2. An **m-out-of-n to binary converter** realizes an index generation function with $\binom{n}{m}$ nonzero elements. It has n inputs and $\lceil \log_2 \binom{n}{m} + 1 \rceil$ outputs. When the number of 1's in the inputs is not m, the converter produces the all 0 code. The m-out-of-n code is produced in ascending lexicographical order. That is, the smallest number is denoted by $(0, 0, \ldots, 0, 1, 1, \ldots, 1)$, while the largest number is denoted by $(1, 1, \ldots 1, 0, 0, \ldots, 0)$.

Example 12.5.1. When $n = 6$ and $m = 3$, we have the function shown in Table 12.6.

Table 12.6 Registered vector table for 3-out-of-6 to binary converter

3-out-of-6 code						
x_1	x_2	x_3	x_4	x_5	x_6	Index
0	0	0	1	1	1	1
0	0	1	0	1	1	2
0	0	1	1	0	1	3
0	0	1	1	1	0	4
0	1	0	0	1	1	5
0	1	0	1	0	1	6
0	1	0	1	1	0	7
0	1	1	0	0	1	8
0	1	1	0	1	0	9
0	1	1	1	0	0	10
1	0	0	0	1	1	11
1	0	0	1	0	1	12
1	0	0	1	1	0	13
1	0	1	0	0	1	14
1	0	1	0	1	0	15
1	0	1	1	0	0	16
1	1	0	0	0	1	17
1	1	0	0	1	0	18
1	1	0	1	0	0	19
1	1	1	0	0	0	20

Theorem 12.5.1. *Let $f(x_1, x_2, \ldots, x_n)$ be a m-out-of-n to binary converter. Let $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$ be a partition of $X = (x_1, x_2, \ldots, x_n)$. Then, the column multiplicity μ_p of the decomposition $f(X_1, X_2)$ is*

$$\mu_p = 2^p \quad (\textit{when } 1 \le p \le m)$$

$$\mu_p = 1 + \sum_{i=0}^{m} \binom{p}{i} \quad (\textit{when } m < p < n-1)$$

$$\mu_p = \binom{n}{m} + 1 \quad (\textit{when } p = n-1, \textit{ or } n)$$

Example 12.5.2. Consider the case of $m = 2$ and $n = 9$. The profile of the 2-out-of-9 to binary converter is

$$(\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6, \mu_7, \mu_8, \mu_9) = (2, 4, 8, 12, 17, 23, 30, 37, 37)$$

Thus, it can be realized by an LUT cascade as shown in Fig. 12.7. Note that the rightmost LUT has 8 inputs. Lemma 4.2.2 shows that an 8-LUT can be realized with 5 modules of 6-LUTs. Thus, the total number of 6-LUTs to implemented the function is $6 + 5 + 5 \times 6 = 41$. ■

Example 12.5.3. Consider the case of $m = 2$ and $n = 20$. The profile of the 2-out-of-20 converter is

$$(2, 4, 8, 12, 17, 23, 30, 38, 47, 57, 68, 80, 93, 107, 122, 138, 155, 173, 191, 191)$$

In this case, the LUT cascade realization is not attractive. Thus, we consider a tree-type realization. Partition the inputs into $X_1 = (x_1, x_2, \ldots, x_{10})$ and $X_2 = (x_{11}, x_{12}, \ldots, x_{20})$. The column multiplicity of the decomposition with respect to (X_1, X_2) and (X_2, X_1) are the same and are both 57. Thus, it can be realized by the circuit shown in Fig. 12.8.

Another implementation is IGU shown in Fig. 10.1. If we can use the linear transformation with many EXOR inputs, then the number of inputs to the main memory is reduced to 9. The derivation of such a linear transformation is beyond the scope of the book. ■

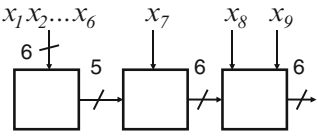


Fig. 12.7 LUT cascade realization of 2-out-of-9 to binary converter

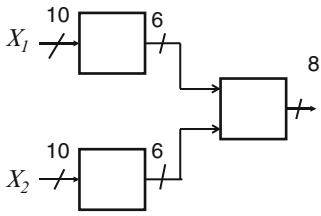


Fig. 12.8 Tree-type realization of 2-out-of-20 to binary converter

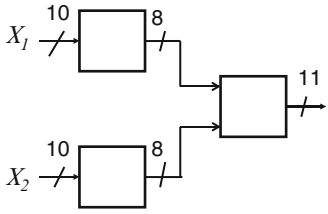


Fig. 12.9 Tree-type realization of 3-out-of-20 to binary converter

Example 12.5.4. Consider the case of $m = 3$ and $n = 20$. The profile of the 3-out-of-20 converter is

$$(2, 4, 8, 16, 27, 43, 65, 94, 131, 177, 233, 300, 379, 471, 577, 698, 835, 988, 1141, 1141)$$

In this case, the LUT cascade realization is impractical. To realize a tree-type circuit, we partition the inputs into $X_1 = (x_1, x_2, \ldots, x_{10})$ and $X_2 = (x_{11}, x_{12}, \ldots, x_{20})$. The column multiplicity of the decomposition with respect to (X_1, X_2) and (X_2, X_1) are the same and equal to 177. Thus, it can be realized by the tree-type circuit shown in Fig. 12.9. Unfortunately, the output LUT has 16 inputs and 11 outputs, and is rather large. Since the function is an index generation function, it can be realized as shown in Chap. 10.

Another implementation is IGU shown in Fig. 10.1. If we can use the linear transformation with many EXOR inputs, then the number of inputs to the main memory is reduced to 11. Again, the derivation of such a linear transformation is beyond the scope of the book. ■

12.6 Remarks

This chapter presented various methods to implement index generation functions. Given the number of variables n, the number of registered vectors k, and available devices, we can select the best design method among various methods: single-memory, LUT cascade, LUT cascade emulator, the hybrid method, the super hybrid method, and the standard parallel sieve method. This chapter is based on [85, 97, 98, 123].

Problems

12.1. Realize the index generation function shown in Table 12.1. Use an LUT cascade, where each LUT has at most 6 inputs. Compare the amount of memory to implement the function by the method shown in Example12.3.1 and the LUT cascade.

12.2. Realize the index generation function shown in Table 12.1. Use an LUT cascade emulator, where each LUT has at most 6 inputs. Explain the operation of the emulator. Compare the size of memory with the LUT cascade realization.

12.3. The standard parallel sieve method presented in Chapter 10 uses IGUs with different sizes. Consider the parallel sieve method that uses IGUs with the same sizes. Discuss the advantage and disadvantage of this approach [136].

12.4. Realize the 2-out-of-12 to binary converter.

Chapter 13
Conclusions

This book showed various methods to realize logic functions with LUTs. These methods can be used to design FPGAs as well as custom integrated circuits. Main applications are communication circuits and pattern matching circuits, where frequent reconfiguration is needed. Major results are as follows:

1. Chapter 4 showed a general method to realize logic functions by LUTs. The number of 6-LUTs to realize an n-variable function is $(2^{n-4}-3)/3$ or less, when $n = 2r$.
2. Chapter 5 derived the number of LUTs to realize logic functions with small C-measure $\mu(f)$. The number of 6-LUTs to realize an n-variable function ($n \geq 8$) is:

   (a) $5n - 35$ or less, when $\mu(f) \leq 32$.
   (b) $2n - 11$ or less, when $\mu(f) \leq 16$.
   (c) $n - 5$ or less, when $\mu(f) \leq 8$ ($n = 3r$).
   (d) $n - 4$ or less, when $\mu(f) \leq 8$ ($n \neq 3r$).

3. Chapter 6 showed a method to reduce the number of LUTs using nonstrict encoding in a functional decomposition. This method is a way to find a nondisjoint decomposition.
4. Chapter 7 showed various functions with small C-measures. These include symmetric functions, sparse functions, LPM functions, segment index encoder functions, and WS functions. Thus, these functions can be efficiently realized by a cascade-based method.
5. Chapter 8 derived upper bounds on the column multiplicity of a decomposition chart for logic functions with weight u. The number of 6-LUTs needed to realize an n variable function is:

   (a) 10 or less, when $n = 9$ and $u \leq 55$.
   (b) 15 or less, when $n = 10$ and $u \leq 47$.
   (c) 15 or less, when $n = 10$ and $u \leq 64$, for most functions.

6. Chapter 9 introduced index generation functions, and showed their applications. An index generation function with weight k can be realized by (p, q)-elements, where a (p, q)-element is an LUT with $p = \lceil \log_2(k + 1) \rceil + 1$ inputs and q outputs, where $p > q$.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2_13,

7. Chapter 10 showed three methods to realize index generation functions: the hybrid method, the super hybrid method, and the parallel sieve method. In these methods:
   (a) The output values for nonregistered inputs are set to *don't cares*, and an incompletely specified logic function is realized by a main memory.
   (b) The output of the main memory is verified by an auxiliary memory. If the output is correct, the circuit produces it as is; otherwise the outputs are set to $00\ldots0$.
   (c) A hash circuit is used to randomize the distribution of registered vectors.
   (d) A main memory is used to realize most of the registered vectors.
   (e) To realize the remaining vectors, the same method is recursively used until the final stage is sufficiently small.
8. Chapter 11 showed a reduction method of the number of variables to represent incompletely specified functions. It was shown that most index generation functions with weight k can be represented with $p = 2\lceil \log_2(k+1) \rceil - 1$ or fewer variables. A method was introduced to reduce the number of variables by using a linear transformation of the input variables.
9. Chapter 12 reviewed various methods to realize index generation functions.

Solutions

Problems of Chapter 2

2.1 In the CAM, if two or more rows match, then the priority encoder selects the first one. In the PLA, the product for the first row is not necessary to implement, since the outputs are (0,0). Also, note that the pattern of the top row is disjoint from other rows, and it can be ignored. The second pattern (row) and the third pattern (row) intersect. When $(x_1, x_2, x_3, x_4) = (0, 1, 1, 0)$, both the second and the third patterns match. However, the second pattern has the priority. So, we have to modify the third pattern to make these patterns disjoint. We have to implement the PLA shown in Table 13.1.

2.2 A synchronous memory requires one clock pulse to read out the data. So, it is unsuitable for combinational circuits. On the other hand, asynchronous memory requires no clock pulse to read out the data. The data can be read out immediately. So, it can be used as an ordinary logic gate.

2.3 With the miniaturization of transistors and interconnections in LSIs, the delay time of the transistors reduces, while that of the interconnection increases. Also, the number of logic devices that can be included in an FPGA increases. Thus, the reduction of the delay time is more important than the reduction of the logic elements. To reduce the delay time, increasing the number of inputs to an LUT is the most effective.

2.4 To implement NOR or NAND in CMOS, transistors must be connected in series. A series connection of up to four transistors is permitted and it still maintains the performance. Thus, when the number of inputs to a PLA is large, a static CMOS circuit is hard to implement. Dynamic CMOS is a method to implement a logic function that is small and dissipates relatively less power.

2.5

1. Form a blank truth table of n inputs and m outputs, where $m = \lceil \log_2 W \rceil$.
2. From the top word to the bottom word, fill in the truth table entries.
3. From the truth table, obtain a simplified sum-of-products expression.
4. Program the PLA.

T. Sasao, *Memory-Based Logic Synthesis*, DOI 10.1007/978-1-4419-8104-2,

Table 13.1 Pattern for PLA

x_1	x_2	x_3	x_4	x_5	f_1	f_0
0	1	1	0	–	0	1
0	1	1	1	–	1	0
1	0	0	1	1	1	1

2.6

1. Form the truth table from the PLA.
2. Form a disjoint sum-of-products expression. Reduce the number of products, if possible.
3. For each product, form a CAM word.

Problems of Chapter 3

3.1 The left-hand-side is

$$\begin{aligned} &SB(4,1)\cdot SB(4,2)\\ &\quad = (x_1 \oplus x_2 \oplus x_3 \oplus x_4)\cdot(x_1x_2 \oplus x_1x_3 \oplus x_1x_4 \oplus x_2x_3 \oplus x_2x_4 \oplus x_3x_4). \end{aligned}$$

The right-hand-side is

$$SB(4,3) = x_1x_2x_3 \oplus x_1x_2x_4 \oplus x_1x_3x_4 \oplus x_2x_3x_4.$$

The straightforward expansion of the left-hand-side will produce a complicated expression. So, we use the Shannon's expansion to prove the equality.
When, $x_1 = 0$: The left-hand-side is

$$\begin{aligned} &(x_2 \oplus x_3 \oplus x_4)\cdot(x_2x_3 \oplus x_2x_4 \oplus x_3x_4)\\ &\quad = \{(x_2 \oplus x_3) \oplus x_4\}\{x_2x_3 \oplus (x_2 \oplus x_3)x_4\}\\ &\quad = x_2x_3x_4 \oplus (x_2 \oplus x_3)x_4 \oplus (x_2 \oplus x_3)x_4\\ &\quad = x_2x_3x_4. \end{aligned}$$

The right-hand-side is $x_2x_3x_4$.
When, $x_1 = 1$: The left-hand-side is

$$\begin{aligned} &(1 \oplus x_2 \oplus x_3 \oplus x_4)\cdot(x_2 \oplus x_3 \oplus x_4 \oplus x_2x_3 \oplus x_2x_4 \oplus x_3x_4)\\ &\quad = \{(x_2 \oplus x_3) \oplus \bar{x}_4\}\{(x_2 \oplus x_3) \oplus x_2x_3 \oplus (x_2 \oplus x_3)x_4 \oplus x_4\}\\ &\quad = [(x_2 \oplus x_3) \oplus (x_2 \oplus x_3)x_4 \oplus (x_2 \oplus x_3)x_4] \oplus [(x_2 \oplus x_3)\bar{x}_4 \oplus x_2x_3\bar{x}_4]\\ &\quad = (x_2 \oplus x_3)x_4 \oplus x_2x_3\bar{x}_4 \end{aligned}$$

The right-hand-side is

$$\begin{aligned} x_2x_3 \oplus x_2x_4 \oplus x_3x_4 \oplus x_2x_3x_4 &= (x_2 \oplus x_3)x_4 \oplus x_2x_3(1 \oplus x_4) \\ &= (x_2 \oplus x_3)x_4 \oplus x_2x_3\bar{x}_4 \end{aligned}$$

Therefore, the equality holds.

3.2 For h, the number of the functions is at most 2^{2^k}. For g, the number of the functions is at most $2^{2^{n-k+1}}$. Thus, the total number of functions is at most

$$2^{2^k} \cdot 2^{2^{n-k+1}} = 2^{2^k+2^{n-k+1}}.$$

3.3 For h, the number of functions is at most $(2^{2^k})^m = 2^{m2^k}$. For g, the number of functions is at most $2^{2^{n-k+m}}$. Thus, the total number of functions is at most

$$2^{m2^k} \cdot 2^{2^{n-k+m}} = 2^{m2^k+2^{n-k+m}}.$$

3.4 When $X_1 = (x_1, x_2, x_3)$, as shown in Fig. 13.1, the column multiplicity is four.

3.5 In an n variable function, the total number of possible input combination is 2^n. Thus, the number of functions with weight k is

$$\binom{2^n}{k}.$$

3.6 Note that

$$\begin{aligned} S_6^9 &= \bar{y}_3y_2y_1\bar{y}_0, \\ S_5^9 &= \bar{y}_3y_2\bar{y}_1y_0, \\ S_4^9 &= \bar{y}_3y_2\bar{y}_1\bar{y}_0, and \\ S_3^9 &= \bar{y}_3\bar{y}_2y_1y_0. \end{aligned}$$

		0	0	0	0	1	1	1	1	x_3
		0	0	1	1	0	0	1	1	x_2
		0	1	0	1	0	1	0	1	x_1
0	0	0	1	1	2	1	2	2	3	
0	1	1	2	2	3	2	3	3	4	
1	0	1	2	2	3	2	3	3	4	
1	1	2	3	3	4	3	4	4	5	
x_5	x_4									

Fig. 13.1 Decomposition chart for $WGT5$

Table 13.2 Condensed decomposition chart for *SYM*12

	0	1	2	3	4	5	6	7	8	9	X_1
0	0	0	0	0	1	1	1	1	1	0	
1	0	0	0	1	1	1	1	1	0	0	
2	0	0	1	1	1	1	1	0	0	0	
3	0	1	1	1	1	1	0	0	0	0	
X_2											

Thus, we have

$$\begin{aligned} S^9_{\{3,4,5,6\}} &= \bar{y}_3 y_2 y_1 \bar{y}_0 \vee \bar{y}_3 y_2 \bar{y}_1 y_0 \vee \bar{y}_3 y_2 \bar{y}_1 \bar{y}_0 \vee \bar{y}_3 \bar{y}_2 y_1 y_0 \\ &= \bar{y}_3 (y_2 (\bar{y}_1 \vee \bar{y}_0) \vee \bar{y}_2 y_1 y_0) \\ &= \bar{y}_3 (y_2 \oplus y_1 y_0). \end{aligned}$$

3.7 Consider the decomposition chart of *SYM*12. Let $X_1 = (x_1, x_2, \ldots, x_8)$ and $X_2 = (x_9, x_{10}, x_{11}, x_{12})$. Since it is a symmetric function, we can write a **condensed decomposition chart** shown in Table 13.2 instead of the full table. Note that the labels show the weights of X_1 or X_2. Thus, for example, the column 4 represents $\binom{9}{4} = 126$ columns of weight 4. It is clear that the number of different column patterns is 8.

Problems of Chapter 4

4.1 Since $\mu_6 = 16$, f can be realized by a circuit shown in Fig. 13.2. Note that the first block has 6 inputs and $\lceil \log_2(16) \rceil = 4$ outputs. Thus, the second block has $4 + 4 = 8$ inputs. Also, note that an arbitrary 8-variable function can be realized with at most five 6-LUTs. Thus, f can be realized with $4 + 5 = 9$ LUTs.

4.2 Consider the expansion of the function

$$f(X_1, X_2) = \bigvee_{\vec{a}_i \in B^4} f(\vec{a}_i, X_2) X_1^{\vec{a}_i}.$$

Among 16 subfunctions $f(\vec{a}_i, X_2)$, $\vec{a}_i \in B^4$, only 10 functions are distinct. So, f can be realized as Fig. 13.3. Note that 10 LUTs are used to realize these 10 different subfunctions, while 5 LUTs are used to realize a 4-MUX.

4.3 From (4.2) and (4.3), we have the following relations:

$$\begin{aligned} g_0(Y) &= f_0(Y) \\ g_1(Y) &= f_0(Y) \oplus f_2(Y) \end{aligned}$$

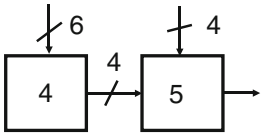


Fig. 13.2 Realization of 10-variable function

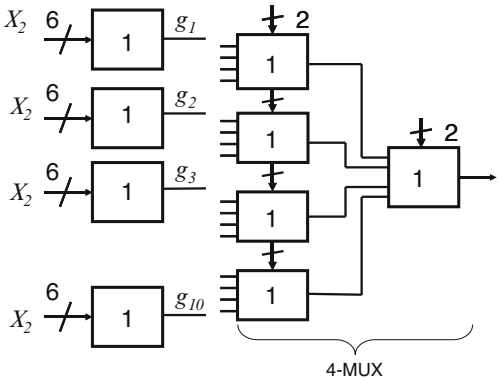


Fig. 13.3 Realization of 10-variable function with 6-LUTs

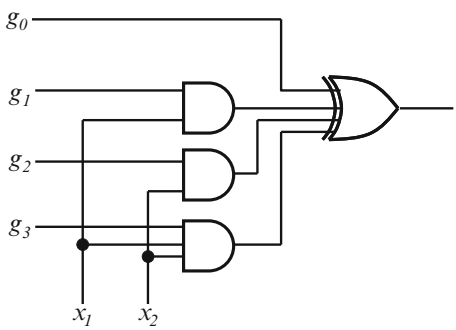


Fig. 13.4 Reed–Muller type module

$$g_2(Y) = f_0(Y) \oplus f_1(Y)$$
$$g_3(Y) = f_0(Y) \oplus f_1(Y) \oplus f_2(Y) \oplus f_3(Y)$$

Thus, instead of the Shannon's expansion, we can also use the Reed–Muller expansion to realize the function. The universal logic element for this expansion is shown in Fig. 13.4, which can be implemented by a 6-LUT.

4.4 The function $f(X_1, X_2)$ can be expanded as

$$f(X_1, X_2) = g_0(\vec{a}_0, X_2) \vee g_1(\vec{a}_1, X_2) \vee \cdots \vee g_{4^k-1}(\vec{a}_{4^k-1}, X_2),$$

where $\vec{a}_0 = (0, 0, \ldots, 0, 0)$, $\vec{a}_1 = (0, 0, \ldots, 0, 1)$, $\ldots$, $\vec{a}_{15} = (1, 1, \ldots, 1, 1)$. Thus, $f(X_1, X_2)$ can be realized as the circuit similar to Fig. 4.12. Since the column

multiplicity is μ, the number of different column functions $g_i(\vec{a}_i, X_2)$ is μ. So, in Fig. 4.12, the LUTs producing the same functions can be shared, and only μ LUTs are sufficient to produce $g_i(\vec{a}_i, X_2)$. The number of 6-LUTs to realize $(2k)$-MUX is $\frac{4^k-1}{3}$. Thus, we need at most $\mu + \frac{4^k-1}{3}$ LUTs.

Problems of Chapter 5

5.1 Tree-type Realization. Advantage: fast. Disadvantage : layout is complex.

Cascade realization. Advantage: layout is simple. Disadvantage : slow.

5.2 Figure 13.5 shows a realization of a 4-bit adder by 6-LUTs. The expressions for the outputs are:

$$
\begin{aligned}
z_0 &= x_0 \oplus y_0 \\
z_1 &= (x_1 \oplus y_1) \oplus (x_0 y_0) \\
z_2 &= (x_2 \oplus y_2) \oplus (x_1 y_1) \oplus (x_1 \oplus y_1)(x_0 y_0) \\
c_2 &= (x_2 y_2) \oplus (x_2 \oplus y_2)(x_1 y_1) \oplus (x_2 \oplus y_2)(x_1 \oplus y_1)(x_0 y_0) \\
z_3 &= (x_3 \oplus y_3) \oplus c_2 \\
z_4 &= (x_3 y_3) \oplus c_2 (x_3 \oplus y_3)
\end{aligned}
$$

In total, it uses 6 LUTs.

5.3 The outputs of the circuits are $y_3 = SB(12, 8)$, $y_2 = SB(12, 4)$, $y_1 = SB(12, 2)$, and $y_0 = SB(12, 1)$. Each function y_i is a symmetric function of 12 variables.

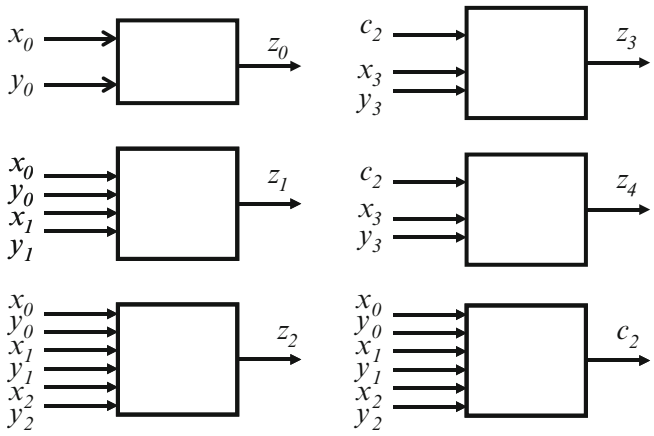


Fig. 13.5 Realization of 4-bit adder by 6-LUTs

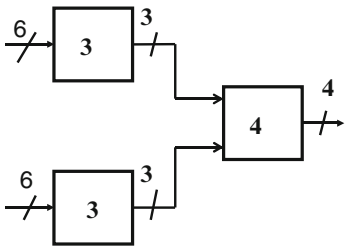


Fig. 13.6 Tree-type realization of WGT12 using 6-LUTs

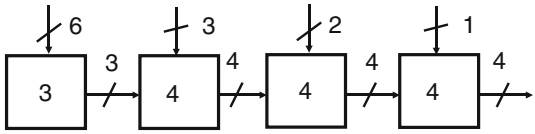


Fig. 13.7 Cascade realization of WGT12 using 6-LUTs

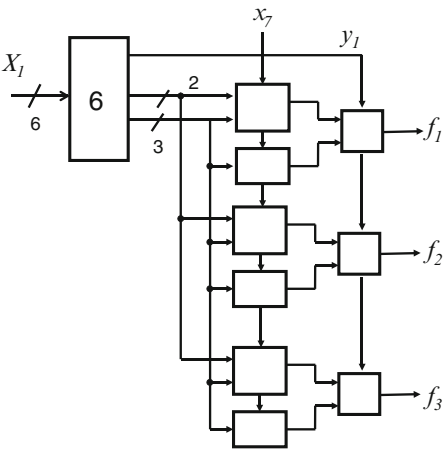


Fig. 13.8 Cascade realization of 7-variable functions by 6-LUTs

Tree-type realization. The input variables $X=(x_1, x_2, \ldots, x_{12})$ are partitioned into $X_1 = (x_1, x_2, \ldots, x_6)$ and $X_2 = (x_7, x_8, \ldots, x_{12})$. Each output can be implemented by seven 6-LUTs as shown in Fig. 5.9. Thus, WGT12 can be implemented by $7 \times 4 = 28$ LUTs. However, when the function is decomposed as $g(h_1(X_1), h_2(X_2))$ as shown in Fig. 13.6, the number of LUTs is reduced to $3 + 3 + 4 = 10$.

Cascade realization. The input variables $X = (x_1, x_2, \ldots, x_{12})$ are partitioned into $X_1 = (x_1, x_2, \ldots, x_6)$, $X_2 = (x_7, x_8, x_9)$, $X_3 = (x_{10}, x_{11})$, and $X_4 = (x_{12})$.

When X_1 denotes the bound variables, $\mu_6 = 7$. When (X_1, X_2) denotes the bound variables, $\mu_9 = 10$. When (X_1, X_2, X_3) denotes the bound variables, $\mu_{11} = 12$. Figure 13.7 shows the realization. In this case, the total number of LUTs is $3 + 4 + 4 + 4 = 15$.

5.4 Figure 13.8 shows the cascade realization of 7-variable functions. First, each function is decomposed as $f(X_1, x_7) = g(h(X_1), x_7)$, where $X_1 = (x_1, x_2, \ldots, x_6)$. Note that, the column multiplicity is at most 40. So, the outputs of h can be encoded

Table 13.3 Encoding for intermediate functions

y_1	y_2	y_3	y_4	y_5	y_6
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
.	.	.	.	.	.
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	0	1	1
1	0	0	0	0	0
1	0	0	1	0	1
1	0	0	1	1	0
1	0	0	1	1	1

by 6 bits as shown in Table 13.3. Note that in the first 32 codes, $y_1 = 0$, while in the remaining 8 codes, $y_1 = 1$. Let the function with $y_1 = 0$ be $f_0(X_1, x_7)$, and let the function with $y_1 = 1$ be $f_1(X_1, x_7)$. Then, f can be represented by $f = \bar{y_1} f_0(X_1, x_7) \vee y_1 f_1(X_1, x_7)$, where y_1 denotes the first bit. The LUTs in the middle column realize $f_0(X_1, x_7)$ and $f_1(X_1, x_7)$. The LUTs in the output column realize $f(X_1, x_7)$. Note that this realization requires 12 LUTs. It is possible to realize each function independently by using Lemma 4.2.1, which requires only $3 \times 3 = 9$ LUTs.

5.5 Let (X_1, X_2) be a partition of the input variables, where $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$. Consider the decomposition chart where X_1 denotes the bound variables. The number of different column patterns is $\mu_k(n)$. Next, consider the decomposition chart with the partition $(\hat{X_1}, \hat{X_2})$, where $\hat{X_1} = (x_1, x_2, \ldots, x_k, x_{k+1})$ and $\hat{X_2} = (x_{k+2}, x_{k+3}, \ldots, x_n)$. In this chart, the number of the columns is halved, while the number of the rows is doubled. The number of different column patterns is at most $2\mu_k(n)$, since the column functions of the original decomposition chart are split into two parts, where the upper and the lower parts have at most $\mu_k(n)$ patterns. Thus, we have the first relation.

Next, consider the decomposition chart with the partition $(\check{X_1}, \check{X_2})$, where $\check{X_1} = (x_1, x_2, \ldots, x_{k-2}, x_{k-1})$ and $\check{X_2} = (x_k, x_{k+1}, \ldots, x_n)$. In this case, the number of the columns is reduced by half, while the number of the rows is increased by two. The number of different column patterns is at most $\mu_k^2(n)$, since the upper (lower parts) of the column functions of the new decomposition chart have at most $\mu_k(n)$ patterns. Thus, we have the second relation.

5.6 Consider the decomposition chart for $f(X_1, X_2)$, where $X_1 = (x_1, x_2, x_3, x_4, x_5)$ and $X_2 = (x_6, x_7, x_8)$. The number of columns is $2^5 = 32$, and the number of rows is $2^3 = 8$. Consider a function whose column multiplicity is 32. Since the

column functions are 3-variable functions, there are $2^{2^3} = 256$ candidate functions. The number of ways to choose 32 different column functions is

$$\binom{256}{32}.$$

Next, the number of ways to order 32 columns is 32! So, the total number of functions is

$$\binom{256}{32} \cdot (32!) = (256!)/(224!).$$

Problems of Chapter 6

6.1 Consider the decomposition $f(X_1, X_2) = g(h(X_1), X_2)$. The column multiplicity is $\mu_4 = 5$. So, we need three intermediate variables (h_3, h_2, h_1). Note that $x_1\Psi_0 = x_1\Psi_1 = 0$, $\bar{x}_1\Psi_2 = \bar{x}_1\Psi_3 = 0$, $x_3\Psi_0 = x_3\Psi_1 = x_3\Psi_2 = x_3\Psi_3 = 0$, $\bar{x}_3\Psi_4 = 0$.

To $\bar{x}_1\Psi_0$ assign the code 000,
to $\bar{x}_1\Psi_1$ assign the code 001,
to $\bar{x}_1\Psi_4$ assign the code 010,
to $x_1\Psi_4$ assign the code 110,
to $x_1\Psi_2$ assign the code 100,
to $x_1\Psi_3$ assign the code 101.

From these, we have intermediate variables:

$$h_3 = x_1(\Psi_2 \vee \Psi_3 \vee \Psi_4) = x_1$$
$$h_2 = \bar{x}_1\Psi_4 \vee x_1\Psi_4 = \Psi_4 = x_3$$
$$h_1 = \bar{x}_1\Psi_1 \vee x_1\Psi_3 = \bar{x}_1x_2\bar{x}_3 \vee x_1\bar{x}_3\bar{x}_4.$$

Note that, in this case, two variables can be simplified.

6.2 First, consider the decomposition $f(X) = g(h(X_1), X_2)$, where $X_1 = (x_1, x_2, x_3, x_4, x_5)$, and $X_2 = (x_6, x_7, x_8)$. The equivalence classes of the decompositions are

$$\Psi_0 = \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5,$$
$$\Psi_1 = x_1x_2x_3x_4x_5,$$
$$\Psi_2 = \overline{\Psi_0 \vee \Psi_1}.$$

Since $\mu = 3$, a standard encoding requires $u = \lceil \log_2 3 \rceil = 2$ intermediate variables. Ψ_i and x_1 satisfy the condition of Theorem 6.3.1: $\bar{x}_1 \Psi_1 = 0$, and $x_1 \Psi_0 = 0$.

To $\bar{x}_1 \Psi_0$, assign the code 00.
To $\bar{x}_1 \Psi_2$, assign the code 01.
To $x_1 \Psi_2$, assign the code 11.
To $x_1 \Psi_1$, assign the code 10.

Thus, we have the following intermediate variables:

$$h_1 = \bar{x}_1 \Psi_2 \vee x_1 \Psi_2 = \Psi_2,$$
$$h_2 = x_1 \Psi_1 \vee x_1 \Psi_2 = x_1.$$

Note that

$$\bar{h}_2 \bar{h}_1 = \Psi_0,$$
$$h_2 \bar{h}_1 = \Psi_1.$$

Therefore, Ψ_i $(i = 0, 1, 2)$ can be represented by (h_2, h_1). Thus, f can be represented as

$$\begin{aligned} f(X) &= \Psi_0 \bar{x}_6 \bar{x}_7 \bar{x}_8 \vee \Psi_1 x_6 x_7 x_8 \\ &= \bar{x}_1 \bar{h}_1 \bar{x}_6 \bar{x}_7 \bar{x}_8 \vee x_1 \bar{h}_1 x_6 x_7 x_8 \end{aligned}$$

Figure 13.9 shows the realization using 5-LUTs.

6.3 Consider the decomposition

$$f(X_1, X_2) = g(h_1(X_1), h_2(X_1), \ldots, h_u(X_1), X_2).$$

Let $\Psi_i(X_1)$,$(i = 0, 1, \ldots, \mu - 1)$ be the equivalence classes of the decomposition. Let $x_j, x_k, \in X_1$. If the number of different nonzero functions in each of

$$x_j x_k \Psi_i,$$
$$x_j \bar{x}_k \Psi_i,$$

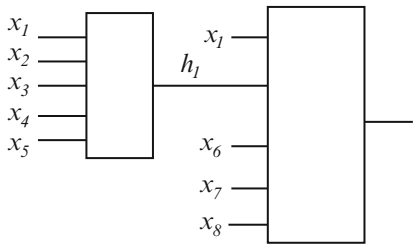


Fig. 13.9 Realization of symmetric function by 5-LUTs

$$\bar{x}_j x_k \Psi_i,$$
$$\bar{x}_j \bar{x}_k \Psi_i,$$

$(i = 1, 2, \ldots, \mu)$ is equal to or less than 2^{u-2}, then $h_u(X_1)$ and $h_{u-1}(X_1)$ can be represented as follows: $h_u(X_1) = x_j$ and $h_{u-1}(X_1) = x_k$.

Problems of Chapter 7

7.1 From Lemma 7.2.1, we have

$$(\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6, \mu_7, \mu_8, \mu_9) = (2, 3, 4, 5, 6, 7, 8, 4, 2).$$

7.2 When $X_1 = (x_2, x_1, x_0)$, as shown in Fig. 13.10, the column multiplicity is 8. When $X_1 = (x_3, x_2, x_1, x_0)$, as shown in Fig. 13.11, the column multiplicity is 11.

7.3 Suppose that X is partitioned as (X_2, X_1), where $X_1 = (x_{n-p-1}, x_{n-p-2}, \ldots, x_0)$ and $X_2 = (x_{n-1}, x_{n-2}, \ldots, x_{n-p})$. We assume that the integer represented by $X = (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$ is $\sum_{i=0}^{n-1} 2^i x_i$. Consider the decomposition chart. Assume that both the columns and rows are labeled in ascending order of the values of X_1 and X_2, respectively. In scanning left-to-right and then top-to-bottom, the

		0	0	0	0	1	1	1	1	x_2
		0	0	1	1	0	0	1	1	x_1
		0	1	0	1	0	1	0	1	x_0
0	0	0	1	1	1	2	2	2	3	
0	1	3	3	4	4	4	5	5	5	
1	0	6	6	6	7	7	7	8	8	
1	1	8	9	9	9	10	10	10	11	
x_4	x_3									

Fig. 13.10 Decomposition chart for $f(X_1, X_2)$

	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_3
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_2
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_1
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	x_0
0	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	
1	6	6	6	7	7	7	8	8	8	9	9	9	10	10	10	11	
x_4																	

Fig. 13.11 Decomposition chart for $f(X_1, X_2)$

values of $g(X)$ will increase from 0 to 1, and then decrease from 1 to 0. In the chart, remove the rows with all 0's and rows with all 1's. Then, the column patterns of the decomposition chart are $(0, 1)$, $(1, 1)$, and $(1, 0)$. Thus, the column multiplicity is three.

7.4 When the function is represented as an SOP with k products, the C-measure increases exponentially with k. For example, when $n = 2k$, consider the function

$$f = x_1x_k \vee x_2x_{k+1} \vee x_3x_{k+3} \vee \cdots \vee x_kx_{2k}.$$

Let $X_1 = (x_1, x_2, \ldots, x_k)$ and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_{2k})$. In this case, the column multiplicity with respect to the decomposition (X_1, X_2) is 2^k. Thus, $\mu(f) = 2^k$.

7.5 Let n_i be the number of variables in X_i, where $i = 1, 2, 3, 4$.

1. Since f is symmetric with respect to X_1,X_2, and X_3, the number of different column patterns is at most

$$(n_1 + 1) \times (n_2 + 1) = 12.$$

2. Since f is symmetric with respect to X_4, the column functions are also symmetric. The number of symmetric functions of n_4 variables is $2^{n_4+1} = 8$. Thus, the number of different column patterns is at most 8.

7.6 When X_1 is the set of bound variables, the column functions have μ_1 different patterns. When (X_1, X_2) is the set of bound variables, the number of different column patters is at most $\mu_1 \cdot \mu_2$.

7.7 Consider the case of $n = 8$. Table 13.4 shows the function of the priority encoder. Note that this is an LPM function. Thus, the C-measure is at most $8 + 1 = 9$. In general, we can show that the C-measure of the priority encoder function of n variables is $n + 1$.

7.8 Table 13.5 shows the decomposition chart of the WS function, while Table 13.6 shows the decomposition chart of the threshold function. The column multiplicity is four.

Table 13.4 Priority encoder ($n = 8$)

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	output
1	–	–	–	–	–	–	–	0001
0	1	–	–	–	–	–	–	0010
0	0	1	–	–	–	–	–	0011
0	0	0	1	–	–	–	–	0100
0	0	0	0	1	–	–	–	0101
0	0	0	0	0	1	–	–	0110
0	0	0	0	0	0	1	–	0111
0	0	0	0	0	0	0	1	1000

Table 13.5 A decomposition chart of a WS function

		0	0	0	0	1	1	1	1	x_2
		0	0	1	1	0	0	1	1	x_2
		0	1	0	1	0	1	0	1	x_0
0	0	0	1	2	3	3	4	5	6	
0	1	4	5	6	7	7	8	9	10	
1	0	5	6	7	8	8	9	10	11	
1	1	9	10	11	12	12	13	14	15	
x_4	x_3									

Table 13.6 A decomposition chart of a threshold function

		0	0	0	0	1	1	1	1	x_2
		0	0	1	1	0	0	1	1	x_2
		0	1	0	1	0	1	0	1	x_0
0	0	0	0	0	0	0	0	0	1	
0	1	0	0	1	1	1	1	1	1	
1	0	0	1	1	1	1	1	1	1	
1	1	1	1	1	1	1	1	1	1	
x_4	x_3									

7.9 Let the input variables $X = (x_{31}, x_{30}, \ldots, x_1, x_0)$ be partitioned into

$$\begin{aligned} X_1 &= (x_{31}, x_{30}, \ldots, x_{25}, x_{24}), \\ X_2 &= (x_{23}, x_{22}, \ldots, x_{17}, x_{16}), \\ X_3 &= (x_{15}, x_{14}, \ldots, x_9, x_8), \text{and} \\ X_4 &= (x_7, x_6, \ldots, x_1, x_0). \end{aligned}$$

Then,

$$f(X) = \left(\sum_{i=0}^{31} 2^i x_i \right) \pmod{17}$$

is implemented as

$$f(X) = (h_1(X_1) + h_2(X_2) + h_3(X_3) + h_4(X_4)) \pmod{17}.$$

Figure 13.12 shows the circuit to compute the modulo. The circuit A1 computes

$$\begin{aligned} h_1(X_1) &= \left(\sum_{i=24}^{31} 2^i x_i \right) \pmod{17} \\ &= \left(2^{24} \sum_{i=0}^{7} 2^i x_{i+24} \right) \pmod{17} \\ &= \left(\sum_{i=0}^{7} 2^i x_{i+24} \right) \pmod{17}. \end{aligned}$$

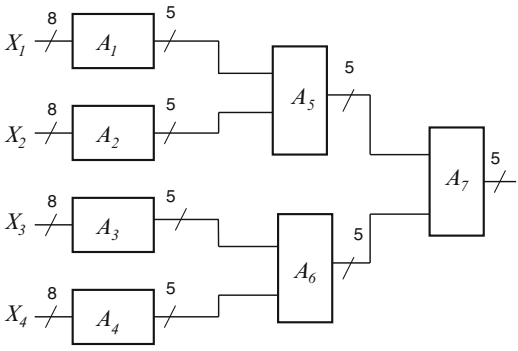


Fig. 13.12 Mod 17 circuit with LUTs

The circuit A2 computes

$$\begin{aligned} h_2(X_2) &= \left(\sum_{i=16}^{23} 2^i x_i\right) \pmod{17} \\ &= \left(2^{16} \sum_{i=0}^{7} 2^i x_{i+16}\right) \pmod{17} \\ &= \left(\sum_{i=0}^{7} 2^i x_{i+16}\right) \pmod{17}. \end{aligned}$$

The circuit A3 computes

$$\begin{aligned} h_3(X_3) &= \left(\sum_{i=8}^{15} 2^i x_i\right) \pmod{17} \\ &= \left(2^8 \sum_{i=0}^{7} 2^i x_{i+8}\right) \pmod{17} \\ &= \left(\sum_{i=0}^{7} 2^i x_{i+8}\right) \pmod{17}. \end{aligned}$$

The circuit A4 computes

$$h_4(X_4) = \left(\sum_{i=0}^{7} 2^i x_i\right) \pmod{17}.$$

The circuits A5, A6, and A7 compute

$$\begin{aligned} &(h_1 + h_2) \pmod{17}, \\ &(h_3 + h_4) \pmod{17}, \end{aligned}$$

and

$$f(X),$$

respectively. Note that in the above equations, we used the relations:

$$2^{24} \equiv 2^{16} \equiv 2^8 \equiv 1 \pmod{17}.$$

7.10 The proofs are easily derived from the relations:

$$2^k \equiv -1 \pmod{2^k + 1}$$
$$2^k \equiv 1 \pmod{2^k - 1}.$$

7.11 Consider the decomposition table (X_1, X_2), where $X_1 = (x_1, x_2, \ldots, x_k)$, and $X_2 = (x_{k+1}, x_{k+2}, \ldots, x_n)$. Consider two columns: $(a_1, a_2, \ldots, a_k)$ and $(b_1, b_2, \ldots, b_k)$. If

$$\sum_{i=1}^{k} a_i w_i = \sum_{i=1}^{k} b_i w_i,$$

then, two columns have the same patterns due to the definition of the threshold function. Thus, the column multiplicity is at most $1 + \sum_{i=1}^{k} w_i$.

Problems of Chapter 8

8.1 From Table 8.3, we can conjecture that $\lambda(8, 20) \leq 16$, since $\lambda(8, 23) = 16$. Consider the case when the column multiplicity is maximum.
8 columns have weight 1, 6 columns have weight 2, and all other columns have weight 0.
Thus, the column multiplicity is $8+6+1 = 15$. In this way, we have $\lambda(8, 20) = 15$.

8.2 First, convert the SOP into a disjoint SOP. Then, obtain the sum of volumes for the cubes.

8.3 Let $g_i(y_1, y_2, \ldots, y_n)$, where $(i = 0, 1, \ldots, 2^{2^n} - 1)$, be 2^{2^n} different functions of n variables. Let the function $f(x_1, x_2, \ldots, x_{2^n}, y_1, y_2, \ldots, y_n)$ represent $g_i(y_1, y_2, \ldots, y_n)$, when $(x_1, x_2, \ldots, x_{2^n})$ denotes the integer i, where $0 \leq i \leq 2^{2^n} - 1$. Consider the decomposition chart of $f(X, Y)$, where $X = (x_1, x_2, \ldots, x_{2^n})$, and $Y = (y_1, y_2, \ldots, y_n)$. In this case, all the columns represent 2^{2^n} distinct functions g_i. Thus, the column multiplicity is 2^{2^n}.

8.4

1. 5 LUTs (Fig. 4.9).
2. 4 LUTs (Fig. 5.3).
3. 21 LUTs (Fig. 4.12).
4. 15 LUTs. (Fig. 13.13).

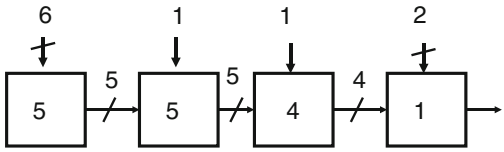


Fig. 13.13 Realization of 10-variable function with weight 47

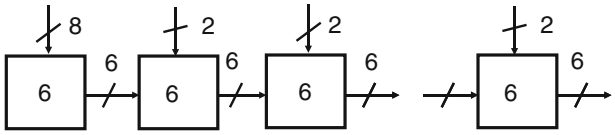


Fig. 13.14 Cascade realization of index generation function with 32 inputs

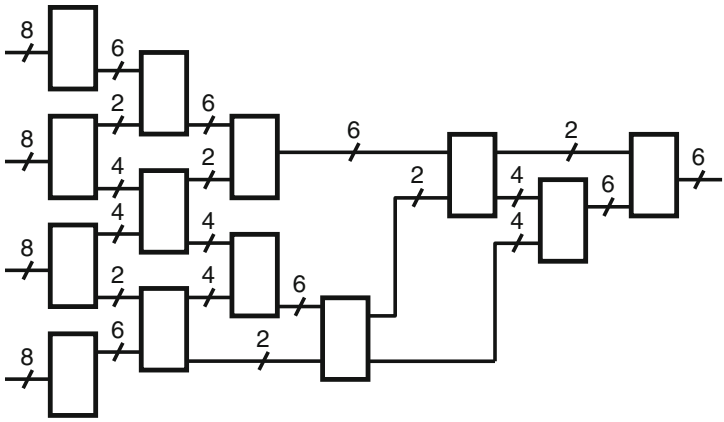


Fig. 13.15 Multilevel realization of index generation function with 32 inputs

Problems of Chapter 9

9.1 Cascade Realization.
Consider Lemma 5.1.5. Let s be the number of cells in the cascade. Note that $n = 32, K = 8, w = \lceil \log_2(k+1) \rceil = 6$. We have:

$$s \leq \left\lceil \frac{n-w}{K-w} \right\rceil = \frac{32-6}{8-6} = 13.$$

Figure 13.14 shows the cascade with 13 cells. Minimum-delay Realization:
Fig. 13.15 shows the multilevel realization.

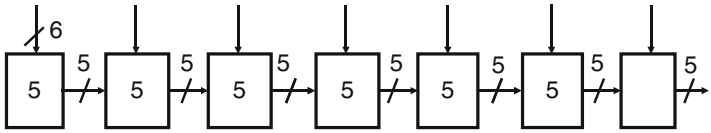


Fig. 13.16 LUT Cascade Realization of the index generation function

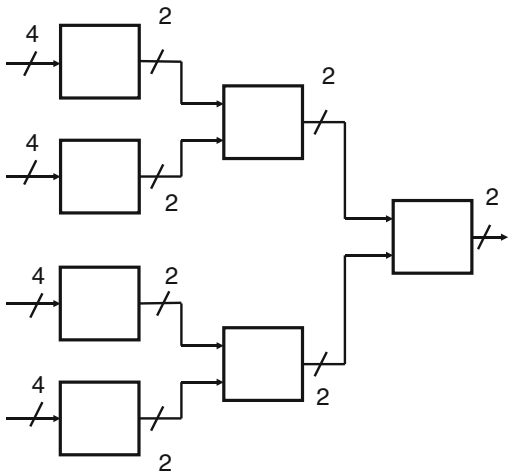


Fig. 13.17 Index generator using $(4, 2)$-elements

9.2 **Advantage:** Memory is cheaper than PLA. Memory dissipates lower power than PLA.

Disadvantage: Single-memory realization is usually too large. Logic design using smaller memories is more complicated than with PLA's.

9.3 The LUT cascade is shown in Fig. 13.16. The total number of bits for the LUTs is

$$2^6 \times 5 \times 7 = 2240.$$

On the other hand, the total number of bits for the realization shown in Fig. 9.14 is

$$2^6 \times 5 \times 2 + 2^{10} \times 5 = 5760.$$

Thus, the realization shown in Fig. 9.14 is faster, but larger than the cascade realization.

9.4 Figure 13.17 shows the circuit.

Problems of Chapter 10

10.1 Suppose that balls are thrown independently and uniformly into v bins. Then, $\alpha = \frac{1}{v}$ is the probability of having a ball in a particular bin. Also, $\beta = 1 - \alpha$ is the probability of not having a ball in a particular bin. When, we throw u balls,

1. The probability that a bin has no ball is β^u.
2. The probability that a bin has at least one ball is $1 - \beta^u$.
3. The probability that a bin has exactly one ball is

$$\binom{u}{1}\alpha \cdot \beta^{u-1} = u \cdot \alpha \cdot \beta^{u-1}.$$

4. The probability that a bin bas more than one ball is

$$\begin{aligned}\eta &= (1 - \beta^u) - u \cdot \alpha \cdot \beta^{u-1} \\ &= 1 - \beta^{u-1} \cdot (\beta + u\alpha) = 1 - \beta^{u-1} \cdot (1 - \alpha + u\alpha) \\ &= 1 - \beta^{u-1} \cdot [1 + (u-1)\alpha] \\ &= 1 - (1-\alpha)^{u-1} \cdot [1 + (u-1)\alpha] \\ &\simeq 1 - \mathrm{e}^{\alpha(u-1)} \cdot [1 + (u-1)\alpha].\end{aligned}$$

Here, we used the approximation $1 - \alpha \simeq e^{-\alpha}$.

Since there are v bins, the expected number of bins with more than one ball is ηv. When $u = 5000$ and $v = 9973$, we have $\eta v = 903$. When $u = 2000$ and $v = 9973$, we have $\eta v = 175$.

10.2 Since the maximum number is $N = 100,000$, x can be represented with $n = 17$ bits. Also, there exist $k_1 = 9592$ prime numbers between 1 and 100,000.

1. In IGU_1, the number of inputs for the main memory is $p_1 = q_1 = \lceil \log_2 \times (k_1 + 1) \rceil = 14$. The number of the vectors realized by IGU_1 is $2^{p_1}(1 - \mathrm{e}^{-\xi_1})$, where $\xi_1 = \frac{k_1}{2^{p_1}}$; that is $2^{14} \times 0.4431444 = 7260$. The number of the remaining vectors is $k_2 = k_1 - 7260 = 2332$.
2. In IGU_2, since $q_2 = \lceil \log_2(2332 + 1) \rceil = 12$, we have $p_2 = q_2 = 12$. The number of the vectors realized by IGU_2 is $2^{p_2}(1 - \mathrm{e}^{-\xi_2})$, where $\xi_2 = \frac{k_2}{2^{p_2}}$; that is $4096 \times 0.4340989 = 1778$. The number of the remaining vectors is $k_3 = k_2 - 1778 = 554$.
3. In IGU_3, since $q_3 = \lceil \log_2(554 + 1) \rceil = 10$, we have $p_3 = q_3 = 10$. The number of vectors realized by IGU_3 is $2^{p_3}(1 - \mathrm{e}^{-\xi_3})$, where $\xi_3 = \frac{k_3}{2^{p_3}}$; that is $1024 \times 0.4178433 = 427$. The number of the remaining vectors is $k_4 = k_3 - 427 = 127$.
4. In IGU_4, since $q_4 = \lceil \log_2(127 + 1) \rceil = 7$, we have $p_4 = q_4 = 7$. The number of vectors realized by IGU_4 is $2^{p_4}(1 - \mathrm{e}^{-\xi_4})$, where $\xi_4 = \frac{k_4}{2^{p_4}}$; that is $128 \times 0.6292353 = 80$. The number of the remaining vectors is $k_5 = k_4 - 80 = 47$.

5. In IGU_5, since $q_5 = \lceil \log_2(47+1) \rceil = 6$, we have $p_5 = q_5 = 6$. The number of vectors realized by IGU_5 is $2^{p_5}(1 - \mathrm{e}^{-\xi_5})$, where $\xi_5 = \frac{k_5}{2^{p_5}}$; that is $64 \times 0.5201948 = 33$. The number of the remaining vectors is $k_6 = k_5 - 33 = 14$.
6. In IGU_6, since the number of the remaining vectors is only $k_6 = 14$, they can be implemented by an IGU [132], or rewritable PLA or an LUT cascade.

Memory for the IGUs is distributed as follows:

$$
\begin{aligned}
&IGU_1 : 17 \times 2^{14} = 272 \times 2^{10}.\\
&IGU_2 : 17 \times 2^{12} = 68 \times 2^{10}.\\
&IGU_3 : 17 \times 2^{10}.\\
&IGU_4 : 17 \times 2^{7} = 2.125 \times 2^{10}.\\
&IGU_5 : 17 \times 2^{6} = 1.0625 \times 2^{10}.
\end{aligned}
$$

The total amount of memory for the parallel sieve method is

$$\sum_{i=1}^{5} n 2^{n_i} = (272 + 68 + 17 + 2.125 + 1.0625) \times 2^{10} \simeq 360 \times 2^{10}$$

bits or 360 Kibits. The single-memory realization requires

$$14 \times 2^{17} = 1.75 \times 2^{20}$$

bits or 1.75 Mibits.

10.3 In this case, $n = 16$ and $k_1 = 3^8 = 6561$. We assume that the probabilities of appearing 0's and 1's are made to equal by a hash circuit.

1. In IGU_1, the number of inputs for the main memory is $p_1 = q_1 = \lceil \log_2 \times (k_1 + 1) \rceil = 13$. The number of the vectors realized by IGU_1 is $2^{p_1}(1 - \mathrm{e}^{-\xi_1})$, where $\xi_1 = \frac{k_1}{2^{p_1}}$; that is $2^{13} \times 0.5510768 = 4514$. The number of the remaining vectors is $k_2 = k_1 - 4514 = 2047$.
2. In IGU_2, since $q_2 = \lceil \log_2(2047+1) \rceil = 11$, we have $p_2 = q_2 = 11$. The number of the vectors realized by IGU_2 is $2^{p_2}(1 - \mathrm{e}^{-\xi_2})$, where $\xi_2 = \frac{k_2}{2^{p_2}}$; that is $2048 \times 0.6319409 = 1294$. The number of the remaining vectors is $k_3 = k_2 - 1294 = 753$.
3. In IGU_3, since $q_3 = \lceil \log_2(753+1) \rceil = 10$, we have $p_3 = q_3 = 10$. The number of vectors realized by IGU_3 is $2^{p_3}(1 - \mathrm{e}^{-\xi_3})$, where $\xi_3 = \frac{k_3}{2^{p_3}}$; that is $1024 \times 0.5206631 = 533$. The number of the remaining vectors is $k_4 = k_3 - 533 = 220$.
4. In IGU_4, since $q_4 = \lceil \log_2(220+1) \rceil = 8$, we have $p_4 = q_4 = 8$. The number of vectors realized by IGU_4 is $2^{p_4}(1 - \mathrm{e}^{-\xi_4})$, where $\xi_4 = \frac{k_4}{2^{p_4}}$; that is $256 \times 0.5765734 = 147$. The number of the remaining vectors is $k_5 = k_4 - 147 = 73$.

5. In IGU_5, since $q_5 = \lceil \log_2(73+1) \rceil = 7$, we have $p_5 = q_5 = 7$. The number of vectors realized by IGU_5 is $2^{p_5}(1 - e^{-\xi_5})$, where $\xi_5 = \frac{k_5}{2^{p_5}}$; that is $128 \times 0.4346513 = 55$. The number of the remaining vectors is $k_6 = k_5 - 55 = 18$.
6. In IGU_6, since the number of the remaining vectors is only $k_6 = 18$, they can be implemented by an IGU [132], or rewritable PLA or an LUT cascade.

Memory for the IGUs is distributed as follows:

$$\begin{aligned} IGU_1 &: 16 \times 2^{13} = 128 \times 2^{10}. \\ IGU_2 &: 16 \times 2^{11} = 32 \times 2^{10}. \\ IGU_3 &: 16 \times 2^{10}. \\ IGU_4 &: 16 \times 2^{8} = 4 \times 2^{10}. \\ IGU_5 &: 16 \times 2^{7} = 2 \times 2^{10}. \end{aligned}$$

The total amount of memory for the parallel sieve method is

$$\sum_{i=1}^{5} n 2^{n_i} = (128 + 32 + 16 + 4 + 2) \times 2^{10} = 182 \times 2^{10}$$

bits or 182 Kibits. The single-memory realization requires

$$13 \times 2^{16} = 0.8125 \times 2^{20}$$

bits or 0.8125 Mibits.

10.4 Suppose a linear transformation changes an input vector $\vec{a}$ into another vector $\vec{b}$. Let A be the transformation matrix. Then, we have the relation:

$$\vec{a}A = \vec{b},$$

where $\vec{a}, \vec{b} \in B^n$, A is a 0-1 matrix, and the addition is *mod* 2 sum. Assume that A is *nonsingular*. Then, the mapping is 1-to-1. This means that the minterms are permuted by the transform. Thus, the number of 1's in the original function is the same as in the transformed one.

10.5 When $k = 2^p$, $\xi = 1$, and $\alpha = \frac{2^p}{2^n} = \frac{1}{2^{n-p}}$. Let $z = \frac{1}{\alpha}$. Then, we have $z = 2^{n-p}$ and $\beta^{2^{n-p}} = (1 - \frac{1}{z})^z$. Note that $e^{-1} \simeq 0.3678794$. Table 13.7 shows the approximation error.

10.6 The function f can be represented by $g(y_1, y_2, y_3, y_4)$ as shown in Table 13.8. In this case, we can assume that only one variable among the inputs $x_1, x_2, \ldots, x_{15}$ takes value 1, and other variables take value 0. Since the function is represented with $p = \lceil \log_2(k+1) \rceil = 4$ variables, it is an optimal linear transformation.

Table 13.7 Approximation error for $\beta^{2^{n-p}}$

α	$\beta^{2^{n-p}}$	*Error*
2^{-1}	0.25000000	0.11787945
2^{-2}	0.31640625	0.05147320
2^{-3}	0.34360892	0.02427053
2^{-4}	0.35607412	0.01180532
2^{-5}	0.36205530	0.00582416
2^{-6}	0.36498654	0.00289293
2^{-7}	0.36643770	0.00144173
2^{-8}	0.36715975	0.00071970
2^{-9}	0.36751989	0.00035956
2^{-10}	0.36769974	0.00017971
2^{-11}	0.36778960	0.00008984
2^{-12}	0.36783454	0.00004492
2^{-13}	0.36785698	0.00002246
2^{-14}	0.36786821	0.00001124
2^{-15}	0.36787382	0.00000562

Table 13.8 Transformed 1-out-of-15 to binary converter

Transformed code				
y_4	y_3	y_2	y_1	Index
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Problems of Chapter 11

11.1 To solve manually, write a Karnaugh map of the function. Only one variable x_5, can be removed.

11.2 From $(\vec{v}_1, \vec{v}_4)$, we have $\bar{c}\,\bar{d}\,\bar{g}$.
From $(\vec{v}_2, \vec{v}_4)$, we have $\bar{b}\bar{d}$.
From $(\vec{v}_3, \vec{v}_4)$, we have $\bar{c}\bar{d}\,\bar{f}$.
From $(\vec{v}_1, \vec{v}_5)$, we have $\bar{b}\bar{c}\bar{e}\,\bar{f}$.
From $(\vec{v}_2, \vec{v}_5)$, we have $\bar{e}\,\bar{f}\,\bar{g}$.

From $(\vec{v}_3, \vec{v}_5)$, we have $\bar{b}\bar{c}\bar{e}\bar{g}$.
From $(\vec{v}_1, \vec{v}_6)$, we have $\bar{a}\bar{c}\bar{d}\bar{f}$.
From $(\vec{v}_2, \vec{v}_6)$, we have $\bar{a}\bar{b}\bar{d}\bar{f}\bar{g}$.
From $(\vec{v}_3, \vec{v}_6)$, we have $\bar{a}\bar{c}\bar{d}\bar{g}$.

$$\begin{aligned}
\bar{R} &= \bar{c}\bar{d}\bar{g} \vee \bar{b}\bar{d} \vee \bar{c}\bar{d}\bar{f} \vee \bar{b}\bar{c}\bar{e}\bar{f} \vee \bar{e}\bar{f}\bar{g} \vee \bar{b}\bar{c}\bar{e}\bar{g} \vee \bar{a}\bar{c}\bar{d}\bar{f} \vee \bar{a}\bar{b}\bar{d}\bar{f}\bar{g} \vee \bar{a}\bar{c}\bar{d}\bar{g} \\
&= \bar{c}\bar{d}\bar{g} \vee \bar{b}\bar{d} \vee \bar{c}\bar{d}\bar{f} \vee \bar{b}\bar{c}\bar{e}\bar{f} \vee \bar{e}\bar{f}\bar{g} \vee \bar{b}\bar{c}\bar{e}\bar{g} \\
R &= (c \vee d \vee g)(b \vee d)(c \vee d \vee f)(b \vee c \vee e \vee f)(e \vee f \vee g)(b \vee c \vee e \vee g) \\
&= (c \vee d \vee fg)(b \vee c \vee e \vee fg)(b \vee d)(e \vee f \vee g) \\
&= [fg \vee c \vee d(b \vee e)](b \vee d)(e \vee f \vee g) \\
&= (fg \vee c)(b \vee d)(e \vee f \vee g) \vee d(b \vee e)(b \vee d)(e \vee f \vee g) \\
&= (fg \vee c)(b \vee d)(e \vee f \vee g) \vee d[e \vee b(f \vee g)]
\end{aligned}$$

The product with the minimum literals is de.

The function can be represented by only two variables, d and e.

11.3

1. Similar to the proof of Theorem 11.5.1, we have

$$\begin{aligned}
PR &= \frac{256}{256} \times \frac{255}{256} \times \frac{254}{256} \times \frac{253}{256} \times \frac{252}{256} \times \frac{251}{256} \times \frac{250}{256} \\
&= \prod_{i=0}^{6} \left(1 - \frac{i}{256}\right) \\
&\simeq e^{-\sum_{i=1}^{6} \frac{i}{256}} = e^{-\frac{21}{256}} = 0.9212.
\end{aligned}$$

2. In a similar way, we have

$$\begin{aligned}
PR &= \frac{128}{128} \times \frac{127}{128} \times \frac{126}{128} \times \frac{125}{128} \times \frac{124}{128} \times \frac{123}{128} \times \frac{122}{128} \\
&= \prod_{i=0}^{6} \left(1 - \frac{i}{128}\right) \\
&\simeq e^{-\sum_{i=1}^{6} \frac{i}{128}} = e^{-\frac{21}{128}} = 0.8487.
\end{aligned}$$

11.4 From the proof of Theorem 11.5.1, we have

$$\eta(k) = \prod_{i=0}^{k-1} \left(1 - \frac{i}{2^p}\right),$$

where $k = 31$ and $p = 9$.

$$\eta(k) = \prod_{i=0}^{30} \left(1 - \frac{i}{512}\right) \simeq \prod_{i=0}^{30} e^{-\frac{i}{512}} = e^{-\sum_{i=1}^{30} \frac{i}{512}} = e^{-\frac{465}{512}} = 0.40$$

There are 10 different partitions. So, the probability that, in all 10 partitions, at least one column has two or more nonzero elements is $(1-\eta(k))^{10}$. Thus, the probability that at least one variable is redundant is

$$1 - (1 - \eta(k))^{10} = 0.994.$$

11.5 First, obtain the probability that the function $f(X)$ does not depend on x_1. Consider the decomposition (X_1, X_2), where $X_1 = (x_2, x_3, \ldots, x_n)$, and $X_2 = (x_1)$.

1. The probability that f takes a specified value is $\alpha = \frac{k}{2^n}$.
2. The probability that a column has at least one don't care is $1 - \alpha^2$.
3. The probability that f does not depend on x_1 is

$$\gamma = (1 - \alpha^2)^{2^{n-1}}.$$

4. The probability that f depends on x_1 is $1 - \gamma$.
5. The probability that f depends on all the variables is $(1 - \gamma)^n$.

When $n = 2r$ and $k = 2^n$, we have $\alpha = \frac{2^r}{2^n} = 2^{-r}$. Thus, $\gamma = (1 - 2^{-n})^{2^{n-1}}$. Since 2^{-n} is sufficiently small, $1 - 2^{-n}$ can be approximated by $e^{-2^{-n}}$. Thus,

$$\gamma \simeq e^{-2^{-n} \cdot 2^{n-1}} = e^{-\frac{1}{2}} \simeq 0.6065.$$

Thus, the probability that all the variables are essential is

$$(1 - \gamma)^n \simeq 0.393^n.$$

11.6 Consider the 4-variable function $f(X)$ such that

$$\begin{aligned}
f(0,1,1,1) &= 1 \\
f(1,0,1,1) &= 1 \\
f(1,1,0,1) &= 1 \\
f(1,1,1,0) &= 1 \\
f(0,1,0,1) &= 0 \\
f(1,0,0,1) &= 0 \\
f(1,0,1,0) &= 0 \\
f(0,1,1,0) &= 0 \\
f(a_1,a_2,a_3,a_4) &= d, \quad \text{(for other combinations)}.
\end{aligned}$$

In this case, all the variables are essential.

11.7 Two variables are sufficient to distinguish all the patterns: (x_1, x_7), (x_2, x_7), (x_4, x_6), (x_4, x_7), (x_5, x_6), or (x_6, x_7).

11.8 Let (X_1, X_2) be a partition of the input variables X, where $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$. Consider the decomposition chart for $f(X_1, X_2)$, where X_1 labels the column variables and X_2 labels the row variables. If each column has at most one *care* element, then f can be represented by using only X_1. Such a condition is true if one of the following is satisfied:

1. A column has only don't cares. The probability of this condition is $\beta^{2^{n-p}}$.
2. A column has only one care element, and the other $2^{n-p} - 1$ elements have don't cares. The probability of this condition is $2^{n-p}\alpha\beta^{2^{n-p}-1}$.

Thus, we have $\gamma_{n-p} = \beta^{2^{n-p}} + 2^{n-p}\alpha\beta^{2^{n-p}-1}$. Since all the $M = 2^p$ columns must satisfy this condition, we have $\delta_{n-p} = \gamma_{n-p}^M$.

11.9

- When $p = 17$, $PR = 0.0011$.
- When $p = 18$, $PR = 0.3817$.
- When $p = 19$, $PR = 0.9461$.

11.10 The probability that the birthdays are all different is the same as the probability that each column of the decomposition chart has at most one care element (See the proof of Theorem 11.5.1). Thus,

$$\eta(k) = \prod_{i=0}^{k-1}\left(1 - \frac{i}{N}\right) = \prod_{i=1}^{k-1}\left(1 - \frac{i}{N}\right),$$

where $k = 64$ and $N = 365$.

$$\eta(k) \simeq 2.81 \times 10^{-3}.$$

Consider the distribution of u distinct balls into v distinct bins, where $u = 64$ and $v = 365$. Assume that any distribution is as likely as any other. We can use the same argument as in the previous exercise. The probability that a bin has at least one ball is $1 - \beta^u$, where $\beta = 1 - \alpha$ and $\alpha = \frac{1}{v}$. Note that

$$1 - \beta^u = 1 - (1 - \alpha)^u \simeq 1 - \mathrm{e}^{-\alpha u}.$$

The expected number of bins with at least one ball is

$$(1 - \beta^u)v \simeq (1 - \mathrm{e}^{-\alpha u})v.$$

By setting $u = 64$ and $v = 365$, we have 58.7, as the expected number of distinct birthdays.

11.11 We can use the same argument as the problem 11.10. The probability that a bin has at least one ball is $1 - \beta^u$, where $\beta = 1 - \alpha$ and $\alpha = \frac{1}{v}$. The expected number of bins with at least one ball is

$$(1 - \beta^u) \cdot v \simeq (1 - \mathrm{e}^{-\alpha u})v.$$

By setting $u = 365$ and $v = 365$, we have

$$(1 - \mathrm{e}^{-\alpha u})v = (1 - e^{-1}) \times 365 = 0.632 \times 365 = 230.724,$$

as the expected number of distinct birthdays.

Problems of Chapter 12

12.1 Figure 13.18 shows the cascade realization. The amount of memory for the cascade is

$$64 \times 4 \times 3 + 32 \times 4 = 896.$$

Note that the method shown in Example 12.3.1 requires 336 bits.

12.2 Figure 13.19 shows the LUT cascade emulator. The counter has two bits.

1. When the value of the counter is 00, set the MSBs to 00. Also, MUX A selects the variables x_1, x_2, x_3, and x_4. Also, MUX B selects x_5 and x_6. In this case, Page 0 is used to emulate $Cell_0$.
2. When the value of the counter is 01, set the MSBs to 01. Also, MUX A selects the outputs from the memory, which corresponds to the rail outputs. Also, MUX B selects x_7 and x_8. In this case, Page 1 is used to emulate $Cell_1$.
3. When the value of the counter is 10, set the MSBs to 10. Also, MUX A selects the outputs from the memory, which corresponds to the rail outputs. Also, MUX B selects x_7 and x_8. In this case, Page 2 is used to emulate $Cell_2$.
4. When the value of the counter is 11, set the MSBs to 11. Also, MUX A selects the outputs from the memory, which corresponds to the rail outputs. Also, MUX B selects x_{11}. In this case, Page 3 is used to emulate $Cell_3$. Since this is the last cell, the output terminal shows the function value.

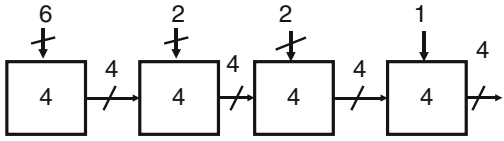


Fig. 13.18 Cascade realization of index generator

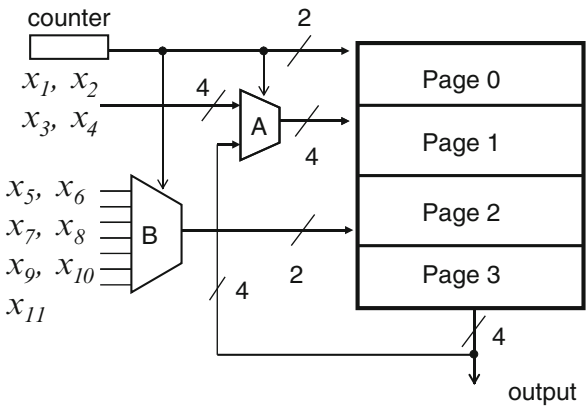


Fig. 13.19 LUT cascade emulator

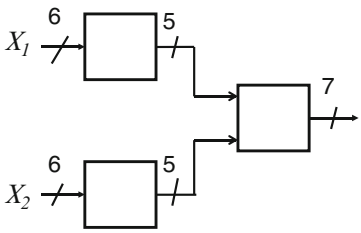


Fig. 13.20 Tree-type realization of 2-out-of-12 to binary converter

12.3 Advantage: Since the sizes of the IGUs are the same, the design of the circuit is easy. Also, the delays of the IGUs are the same. Addition of new registered vectors is easier.

Disadvantage: The approach requires more memory than the method presented in Chap. 10.

12.4 The profile of the 2-out-of-12 to binary converter is

$$(2, 4, 8, 12, 17, 23, 30, 38, 47, 57, 67, 67)$$

In this case, the LUT cascade realization is not attractive. To realize a tree-type circuit, we partition the inputs into $X_1 = (x_1, x_2, \ldots, x_6)$ and $X_2 = (x_7, x_8, \ldots, x_{12})$. The column multiplicity with the decomposition with respect to (X_1, X_2) and (X_2, X_1) are the same and both equal to 23. Thus, it can be realized by the tree-type circuit shown in Fig. 13.20. By using a linear transformation, the number of variables can be reduced to 7, which is the optimum.

Bibliography

1. E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density, " *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, pp.3–12, Feb. 10–11, 2000, Monterey, California.
2. E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 12, No. 3, March 2004, pp.288–298.
3. C. Akrout, et al., "Reprogrammable logic fuse based on a 6-device SRAM cell for logic arrays," US Patent 5063537.
4. http://www.altera.com
5. Altera, "Implementing high-speed search applications with Altera CAM," *Application Note 119*, Altera Corporation, July 2001.
6. Altera, "Stratix IV FPGA Core Fabric Architecture," http://www.altera.com/
7. E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 12, No. 3, March 2004, pp.288–298.
8. R. L. Ashenhurst, "The decomposition of switching functions," *International Symposium on the Theory of Switching*, pp. 74–116, April 1957.
9. Berkeley Logic Synthesis and Verification Group, *ABC: A System for Sequential Synthesis and Verification*, Release 70911. http://www.eecs.berkeley.edu/ alanmi/abc/
10. V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions,"*ICCAD-97*, pp. 78–82, Nov. 1997.
11. T. Bengtsson, A. Martinelli, and E. Dubrova, "A BDD-based fast heuristic algorithm for disjoint decomposition," *ASPDAC* 2003, pp.191–196.
12. R.E. Bryant,"Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.* Vol. C-35, No. 8, pp.677–691, Aug. 1986.
13. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.
14. F. M. Brown, *Boolean Reasoning: The logic of Boolean Equations*, Kluwer Academic Publishers, Boston, 1990.
15. S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field Programmable Gate Arrays*, Kluwer Academic Publishers, Boston (1992).
16. S. Brown , "FPGA architectural research: A Survey," *IEEE Design & Test of Computers*, vol. 13, no. 4, 1996, pp. 9–15.
17. S-C. Chang, M. Marek-Sadowska, and T. Hwang, "Technology mapping for LUT FPGA's based on decomposition of binary decision diagrams," *IEEE Trans. on CAD*, Vol. CAD-15, No. , pp. 1226–1236, Oct. 1996.
18. L. Chisvin and R. J. Duckworth, "Content-addressable and associative memory: Alternatives to the ubiquitous RAM, " *IEEE Computer*, Vol. 22, pp. 51–64, July 1989.

19. G. R. Chiu, D. P. Singh, V. Manohararajah, and S. D. Brown, "Mapping arbitrary logic functions into synchronous embedded memories for area reduction on FPGAs", *Proceedings of the International Conference on Computer-aided design*, (ICCAD-2006), Nov. 2006, San Jose, California, pp.135–142.
20. J. Cong and Y. Ding, "Combinational logic synthesis for LUT based field programmable gate arrays," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol.1, No.2, pp.145–204, April 1996.
21. J. Cong and Y.-Y Hwang, "Partially-dependent functional decomposition with applications in FPGA synthesis and mapping," *Fifth Int. Symp. on Field-Programmable Gate Arrays*, pp. 35–42, Feb. 1997.
22. J. Cong and S. Xu, "Technology mapping for FPGAs with embedded memory blocks,"Feb. 1998, pp.179–188.
23. J. Cong and K. Yan, "Synthesis for FPGAs with embedded memory blocks", In *Proc. of the 2000 ACM/SIGDA 8th International Symposium on Field Programmable Gate Arrays*, pp. 75–82, ACM Press NY, 2000, Monterey, California.
24. J. Cong and K. Minkovich, "Optimality study of logic synthesis for LUT-Based FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, Issue 2, Feb. 2007, pp.230–239.
25. H. A. Curtis, *A New Approach to the Design of Switching Circuits*, D. Van Nostrand Co., Princeton, NJ, 1962.
26. M. Davio, J.-P Deschamps, and A. Thayse, *Digital Systems with Algorithm Implementation*, John Wiley & Sons, New York , 1983, p. 368.
27. D. L. Dietmeyer, *Logic Design of Digital Systems* (Second Edition), Allyn and Bacon Inc., Boston, 1978.
28. C. H. Divine, "Memory patching circuit with increased capability," US Patent 4028679.
29. C. H. Divine and J. C. Moran, "Memory patching circuit with repatching capability," US Patent 4028684.
30. S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," ACM *SIGCOMM'03*, August 25–29, 2003, Karlsruhe, Germany.
31. J. Ditmar, K. Torkelsson, and A. Jantsch, "A reconfigurable FPGA-based content addressable memory for internet protocol characterization," *Proc. FPL2000*, LNCS 1896, Springer, 2000, pp. 19–28.
32. 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," *ICCAD-91*, pp. 560–563, 1991.
33. R. J. Francis, J. Roze, and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," *DAC-1991*, pp. 227–233, June 1991.
34. D. Green, *Modern Logic Design*, Addison-Wesley Publishing Company,1986.
35. H. Gouji, T. Sasao, and M. Matsuura,"On a method to reduce the number of LUTs in LUT cascades," *Technical Report of IEICE*, VLD2001-99, Nov. 2001.
36. S. Guccione, D. Levi, and D. Downs,"A reconfigurable content addressable memory," *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Volume 1800,May 2000. Parallel and Distributed Processing, p.882.
37. P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds, " *Proc. INFOCOM*, IEEE Press, Piscataway, N.J., 1998, pp. 1240–1247.
38. C. Halatsis and N. Gaitanis, "Irredundant normal forms and minimal dependence sets of a Boolean functions," *IEEE Trans. on Computers*, Vol. C-27, No. 11, pp. 1064–1068, Nov. 1978.
39. S. Hassoun and T. Sasao (eds.), *Logic Synthesis and Verification*, Kluwer Academic Publishers, Oct. 2001.
40. J-D. Huang, J-Y. Jou, and W-Z. Shen, "Compatible class encoding in Roth-Karp decomposition for two-output LUT architecture," ICCAD 1995, Nov. 1995, pp.359–363.
41. Ting-Ting Hwang, R. M. Owens, M. J. Irwin, and Kuo Hua Wang,"Logic synthesis for field-programmable gate arrays,"*IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*,Vol. 13, No. 10, pp. 1280–1287, Oct. 1994.

42. Y. Iguchi, T. Sasao, and M. Matsuura, "Realization of multiple-output functions by reconfigurable cascades," *International Conference on Computer Design: VLSI in Computers & Processors (ICCD-2001)*, Austin, TX, Sept. 23–26, 2001. pp. 388–393.
43. Y. Iguchi, T. Sasao, and M. Matsuura, "On designs of radix converters using arithmetic decompositions," *ISMVL-2006*, Singapore, May 17–20, 2006.
44. J.-H. R. Jiang, J.-Y. Jou, and J.-D. Huang, "Compatible class encoding in hyper-function decomposition for FPGA synthesis," *Design Automation Conference*, pp. 712–717, June 1998.
45. T. Kam, T. Villa, R. Brayton, and A. Sangiovanni, " Multi-valued decision diagrams: theory and applications," *Journal of Multiple-Valued Logic*, Vol. 4, No.1–2, 1998, pp. 9–62.
46. Y. Kambayashi, "Logic design of programmable logic arrays," *IEEE Trans. on Computers*, Vol. C-28, No. 9, pp. 609–617, Sept. l979.
47. K. Keutzer, " DAGON: technology binding and local optimization by DAG matching," *24th ACM/IEEE Design Automation Conference* June 28–July 01, 1987.pp.341–347.
48. S. P. Khatri, R. K. Brayton, A. Sangiovanni-Vincentelli, "Cross-talk immune VLSI design using a network of PLAs embedded in a regular layout fabric," *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design*, Nov. 05–09, 2000, San Jose, California.
49. T. Kohonen, *Content-Addressable Memories*, Springer Series in Information Sciences, Vol. 1, Springer Berlin Heidelberg 1987.
50. Y. Komamiya, *Theory of Computing Networks*, Researches of ETL, Sept. 1959.
51. V. Kravets and K. Sakallah, "Constructive library-aware synthesis using symmetries," *Proc. DATE-2000*, pp. 208–213.
52. S. Krishnamoorthy and R. Tessier, "Technology mapping algorithms for hybrid FPGAs containing lookup tables and PLAs", In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 22, No. 5, 2003, pp. 545–559.
53. A. M. Kumar, J. Bobba, and V. Kamakoti, "MemMap: Technology mapping algorithm for area reduction in FPGAs with embedded memory arrays using reconvergence analysis," *Design, Automation, and Test in Europe – DATE-2004*, March 2004, pp.922–929.
54. Y-T. Lai, M. Pedram and S. B. K. Vrudhula, "BDD based decomposition of logic functions with application to FPGA synthesis", *30th ACM/IEEE Design Automation Conference*, June 1993.
55. Y-T. Lai, M. Pedram, and S. B. K. Vrudhula, "EVBDD-based algorithm for integer linear programming, spectral transformation, and functional decomposition," *IEEE Trans. CAD*, Vol. 13, No. 8, pp. 959–975, Aug. 1994.
56. P-F. Lin and J. B. Kuo, "A 1-V 128-kb four-way set-associative CMOS cache memory using wordline-oriented tag-compare (WLOTC) structure with the content-addressable-memory (CAM) 10-transistor tag cell," *IEEE Journal of Solid-State Circuits*, Vol. 36, pp. 666–675, April 2001.
57. A. Ling, D. P. Singh, and S. D. Brown, "FPGA technology mapping:a study of optimality," *42nd Design Automation Conference*, 13–17 June 2005, pp. 427–432.
58. C. Legl, B. Wurth, and K. EcKl, "Computing support-minimal subfunctions during functional decomposition," *IEEE Trans. VLSI*, Vol. 6, No. 3, pp. 354–363, Sept. 1998.
59. A. Martinelli, R. Krenz, and E. Dubrova, "Disjoint-support boolean decomposition combining functional and structural methods," Proceedings of the Asia and South Pacific Design Automation Conference, 2004 (ASP-DAC 2004), 27–30, Jan. 2004. pp.597–599.
60. Y. Matsunaga, "An exact and efficient algorithm for disjunctive decomposition," *SASIMI'98*, pp. 44–50, Oct. 1998.
61. H. J. Mattausch, T. Gyohten, Y. Soda, T. Koide, "Compact associative-memory architecture with fully-parallel search capability for the minimum Hamming distance," *IEEE Journal of Solid-State Circuits*, Vol. 37, No. 2, pp. 218–227, Feb. 2002.
62. K. McLaughlin, N. O'Connor, and S. Sezer, "Exploring CAM design for network processing using FPGA technology," *Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services* (AICT-ICIW'06), p. 84, 19–25, Feb. 2006.

63. C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications*, Springer, 1998.
64. S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation,"*Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 52–57, June 1990.
65. S. Minato, "Graph-based representations of discrete functions," in T. Sasao and M. Fujita (e.d.), *Representations of Discrete Functions*, Kluwer Academic Publishers, 1996.
66. S. Minato and G. De Micheli, "Finding all simple disjunctive decompositions using irredundant sum-of-products forms," *ICCAD-99*, pp. 111–117, Nov. 1998.
67. A. Mishchenko, C. Files, M. Perkowski, B. Steinbach, and Ch. Dorotska, "Implicit algorithms for multi-valued input support manipulation," *Proc. 4th Intl. Workshop on Boolean Problems*, Sept. 2000, Freiberg, Germany.
68. A. Mishchenko and T. Sasao, "Logic synthesis of LUT cascades with limited rails: A direct implementation of multi-output functions,", *IEICE Technical Reports*, VLD2002-99, Nov. 2002.
69. A. Mishchenko and T. Sasao, "Encoding of Boolean functions and its application to LUT cascade synthesis, " *International Workshop on Logic and Synthesis* (IWLS2002), New Orleans, Louisiana, June 4–7, 2002, pp.115–120.
70. A. Mishchenko, "Fast computation of symmetries in Boolean functions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* Vol. 22, Issue 11, pp. 1588–1593, Nov. 2003.
71. A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* Vol. 26, Issue 2,pp. 250–253, Feb. 2007.
72. A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, "Combinational and sequential mapping with priority cuts," *Proc. ICCAD'07*, Nov. 2007, pp.354–361.
73. A. Mishchenko, R. K. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks", *Proc. ICCAD'08*, Nov. 2008, pp.38–45.
74. J. C. Moran, "Memory patching circuit," US Patent 4028678.
75. M. Motomura et al., "A 1.2-million-transistor, 33-MHz, 20-b dictionary search processor (DISP) ULSI with a 160-Kbyte CAM, " *IEEE J. Solid-State Circuits*, Vol. 25, No. 5, Oct. 1990, p. 1158–1165.
76. V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 25, No. 11, November 2006, pp. 2331–2340.
77. R. Murgai and R. K. Brayton, "Optimum functional decomposition using encoding," *Design Automation Conference*. DAC 1994, pp.408–414, June 1994.
78. R. Murgai, R. Brayton, and A. Sangiovanni Vincentelli, *Logic Synthesis for Field-Programmable Gate Arrays*, Springer, July 1995.
79. R. Murgai, F. Hirose, and M. Fujita, "Logic synthesis for a single large look-up table," *Proc. International Conference on Computer Design*,pp. 415–424, Oct. 1995.
80. S. Muroga, *VLSI System Design*, John Wiley & Sons, 1982, pp. 293–306.
81. S. Nagayama, T. Sasao, and J. T. Butler, "Programmable numerical function generators based on quadratic approximation: Architecture and synthesis method," *ASPDAC 2006*, Yokohama Jan. 2006, pp. 378–383.
82. S. Nagayama, T. Sasao, and J. T. Butler "Compact numerical function generators based on quadratic approximation: architecture and synthesis method," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E89-A, No.12, Dec. 2006, pp.3510–3518, Special Section of VLSI Design and CAD Algorithms.
83. H. Nakahara, T. Sasao, and M. Matsuura, "A design algorithm for sequential circuits using LUT rings," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E88-A, No.12, Dec. 2005, pp.3342–3350.
84. H. Nakahara and T. Sasao, "A PC-based logic simulator using a look-up table cascade emulator," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E89-A, No.12, Dec. 2006, pp. 3471–3481.

85. H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, "A Parallel sieve method for a virus scanning engine," *12th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Patras, Greece (DSD 2009).
86. K. Nakamura, T. Sasao, M. Matsuura, K. Tanaka, K. Yoshizumi, H. Qin, and Y. Iguchi, "Programmable logic device with an 8-stage cascade of 64K-bit asynchronous SRAMs," *Cool Chips VIII*, IEEE Symposium on Low-Power and High-Speed Chips, April 20–22, 2005, Yokohama, Japan.
87. K. Nakamura, T. Sasao, M. Matsuura, K. Tanaka, K. Yoshizumi, H. Nakahara, and Y. Iguchi, "A memory-based programmable logic device using a look-up table cascade with synchronous SRAMs," *2005 International Conference on Solid State Devices and Materials (SSDM 2005)*, Kobe, Japan, Sep. 2005.
88. K. Nakamura, T. Sasao, M. Matsuura, K. Tanaka, K. Yoshizumi, H. Nakahara and Y. Iguchi, "A memory-based programmable logic device using look-up table cascade with synchronous static random access memories," *Japanese Journal of Applied Physics*, Vol. 45, No. 4B, 2006, April, 2006, pp. 3295–3300.
89. G. Nilsen, J. Torresen and O. Sorasen, "A variable word-width content addressable memory for fast string matching," *NorChip 2004*, pp. 214–217.
90. K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, March 2006.
91. S. Panda, F. Somenzi, and B. F. Plessier,"Symmetry detection and dynamic variable ordering of decision diagrams," *ICCAD1994*, Nov. 1994, pp.628–631.
92. C. A. Papachristou,"Characteristic measures of switching functions," *Information Science*, Vol.13, No.1, pp.51–75, 1977.
93. G. Papadopoulos and D. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching," in 15th *International Conference on Field Programmable Logic and Applications*, Aug. 2005, pp. 39–44.
94. S. Posluszny, N. Aoki, D. Boerstler, J. Burns, S. Dhong, U. Ghoshal, P. Hofstee, D. LaPotin, K. Lee, D. Meltzer, H. Ngo, K. Nowka, J. Silberman, and O. Takahashi,"Design methodology for a 1.0 GHz microprocessor,"*Proc. ICCD-1998*, (Computer Design: VLSI in Computers and Processors), Oct. 5–7, 1998, pp. 17–23.
95. J. Qiao, M. Ikeda, and K. Asada, " Finding an optimal functional decomposition for LUT-based FPGA synthesis," in *Proceedings of the 2001 conference on Asia South Pacific design automation*, pp. 225–230, Jan. 2001, Yokohama, Japan.
96. H. Qin, T. Sasao, M. Matsuura, K. Nakamura S. Nagayama and Y. Iguchi "A realization of multiple-output functions by a look-up table ring," *IEICE Transactions on Fundamentals of Electronics*, Vol.E87-A, Dec. 2004, pp.3141–3150.
97. H. Qin, T. Sasao, and J. T. Butler, "Implementation of LPM address generator on FPGAs," *International Workshop on Applied Reconfigurable Computing (ARC2006)*, Delft, the Netherlands, March 1–3, 2006, also appeared in *Lecture Notes in Computer Science* 3985, pp.170–181.
98. H. Qin, T. Sasao, and J. T. Butler, "On the design of LPM address generators using multiple LUT Cascades on FPGAs," *International Journal of Electronics*, Vol. 94, Issue 5, May 2007, pp.451–467.
99. J. Rose, R. J. Francis, D. Lewis, and P. Chow, "Architecture of field programmable gate arrays: The effect of logic block functionality on area efficiency,"*IEEE J. Solid State Circ. 25,5*, pp. 1217–1225, Oct. 1990.
100. J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proc. IEEE*, Vol. 81, No. 7, pp.1013–1029, July 1993.
101. J. P. Roth and R. M. Karp, "Minimization over Boolean graphs,"*IBM Journal of Research and Development*, pp. 227–238, April 1962.
102. R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *ICCAD-93*, pp. 42–47, 1993.
103. A. Sangiovanni-Vincentelli, A. El Gamal, J. Rose, " Synthesis method for field programmable gate arrays," *Proceedings of the IEEE*, Vol.81, No.7,July 1993, pp.1057–1083.

104. T. Sasao and P. Besslich, "On the complexity of MOD-2 Sum PLA's," *IEEE Trans. on Comput*. Vol. 39. No. 2, pp. 262–266, Feb. 1990.
105. T. Sasao, "Bounds on the average number of products in the minimum sum-of-products expressions for multiple-valued input two-valued output functions," *IEEE Trans. on Comput.* Vol. 40, No. 5, pp. 645–651, May 1991.
106. T. Sasao, "FPGA design by generalized functional decomposition," In *Logic Synthesis and Optimization*, Kluwer Academic Publisher, pp. 233–258, 1993.
107. T. Sasao and J. T. Butler, "A design method for look-up table type FPGA by pseudo-Kronecker expansion," *IEEE International Symposium on Multiple-Valued Logic*, Boston, May 1994, pp. 97–106.
108. T. Sasao and M. Matsuura, "DECOMPOS: An integrated system for functional decomposition," *1998 International Workshop on Logic Synthesis*,Lake Tahoe, June 1998.
109. T. Sasao, "Totally undecomposable functions: applications to efficient multiple-valued decompositions," *IEEE International Symposium on Multiple-Valued Logic*, Freiburg, Germany, May 20–23, 1999, pp. 59–65.
110. T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.
111. T. Sasao, "On the number of dependent variables for incompletely specified multiple-valued functions," *30th International Symposium on Multiple-Valued Logic*, pp. 91–97, Portland, Oregon, U.S.A., May 23–25, 2000.
112. T. Sasao," A new expansion of symmetric functions and their application to non-disjoint functional decompositions for LUT-type FPGAs," *International Workshop on Logic Synthesis*, Dana Point, California, U.S.A., May 31–June 2, 2000, pp.105–110.
113. T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *International Workshop on Logic and Synthesis (IWLS01)*, Lake Tahoe, CA, June 12–15, 2001, pp. 225–230.
114. T. Sasao, "Design methods for multi-rail cascades," (invited paper) *International Workshop on Boolean Problems (IWBP2002)*, Freiberg, Germany, Sept. 19–20, 2002, pp. 123–132.
115. T. Sasao and M. Matsuura, "A method to decompose multiple-output logic functions," *Design Automation Conference*, pp. 428–433, San Diego, USA, June 2–6, 2004.
116. T. Sasao, J. T. Butler, and M. D. Riedel, "Application of LUT cascades to numerical function generators," *The 12th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI2004)*, Oct. 18–19, 2004, Kanazawa, Japan, pp.422–429.
117. T. Sasao, H. Nakahara, M. Matsuura and Y. Iguchi, "Realization of sequential circuits by look-up table ring," *The 2004 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2004)*, Hiroshima, July 25–28, 2004, pp.I:517-I:520.
118. T. Sasao, M. Kusano, and M. Matsuura, "Optimization methods in look-up table rings," *International Workshop on Logic and Synthesis* (IWLS-2004), June 2–4, Temecula, California, U.S.A. .pp. 431–437.
119. T. Sasao, "Radix converters: Complexity and implementation by LUT cascades," *ISMVL-2005*, pp. 256–263.
120. T. Sasao, "Analysis and synthesis of weighted-sum functions," *International Workshop on Logic and Synthesis (IWLS-2005)*, Lake Arrowhead, CA, USA, June 8–10, 2005,pp.455–462.
121. T. Sasao, Y. Iguchi, T. Suzuki, "On LUT cascade realizations of FIR filters," *DSD2005, 8th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, Porto*, Portugal, Aug. 30–Sept. 3, 2005, pp.467–474.
122. T. Sasao, S. Nagayama, and J. T. Butler, "Programmable numerical function generators: Architectures and synthesis system," *FPL2005*, Tampere, Aug.24–26, 2005, pp.118–123.
123. T. Sasao, Y. Iguchi, and M. Matsuura, "LUT cascades and emulators for realizations of logic functions,"*RM2005*, Tokyo, Japan, Sept.5–Sept.6, 2005, pp.63–70.
124. T. Sasao, "Design methods for multiple-valued input address generators," *ISMVL-2006 (invited paper)*, Singapore, May 17–20, 2006.
125. T. Sasao and J. T. Butler, "Implementation of multiple-valued CAM functions by using LUT cascade, " *International Symposium on Multi-Valued Logic*, May 17–20, 2006, Singapore, May 17–20, 2006.

126. T. Sasao, "A Design method of address generators using hash memories," *IWLS-2006*, pp. 102–109, Vail, Colorado, U.S.A, June 7–9, 2006.
127. T. Sasao, "Analysis and synthesis of weighted-sum functions," *IEEE TCAD, Special issue on International Workshop on Logic and Synthesis*, Vol. 25, No. 5, May 2006, pp. 789–796.
128. T. Sasao and M. Matsuura, "An implementation of an address generator using hash memories," *10th EUROMICRO Conference on Digital System Design*, Architectures, Methods and Tools (DSD-2007), Aug. 27–31, 2007, Lubeck, Germany, pp.69–76.
129. T. Sasao and H. Nakahara, "Implementations of reconfigurable logic arrays on FPGAs," *International Conference on Field-Programmable Technology 2007 (ICFPT'07)*, Dec. 12–14, 2007, Kitakyushu, Japan, pp.217–223.
130. T. Sasao, S. Nagayama and J. T. Butler, "Numerical function generators using LUT cascades," *IEEE Transactions on Computers*, Vol.56, No.6, June 2007, pp.826–838.
131. T. Sasao, "On the number of variables to represent sparse logic functions," *17th International Workshop on Logic & Synthesis (IWLS-2008)*, Lake Tahoe, California, USA, June 4–6, 2008, pp.233–239.
132. T. Sasao, "On the number of variables to represent sparse logic functions," *ICCAD-2008*, San Jose, California, USA, Nov.10–13, 2008, pp. 45–51.
133. T. Sasao, "On the number of LUTs to realize sparse logic functions," *International Workshop on Logic and Synthesis* (IWLS-2009), July 31–Aug.2, 2009.
134. T. Sasao and A. Mishchenko, "LUTMIN: FPGA logic synthesis with MUX-based and cascade realizations," *International Workshop on Logic and Synthesis* (IWLS-2009), July 31–Aug. 2, 2009.
135. T. Sasao, T. Nakamura, and M. Matsuura, "Representation of incompletely specified index generation functions using minimal number of compound variables," *12th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Patras, Greece *DSD-2009* .
136. T. Sasao, M. Matsuura, and H. Nakahara, "A realization of index generation functions using modules of uniform sizes," *International Workshop on Logic and Synthesis* (IWLS-2010), Irvine, California, June 11–13, 2010, pp.201–208.
137. T. Sasao, "On the numbers of variables to represent multi-valued incompletely specified functions," *13th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Lille, France *DSD-2010*, Sept. 2010, pp.pp.420–423.
138. T. Sasao, "Index generation functions: Recent developments," *International Symposium on Multiple-Valued Logic* (ISMVL-2011), Tuusula, Finland, May 23–25, 2011. (invited paper).
139. H. Sawada, T. Suyama, and A. Nagoya, "Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization," *ICCAD*, pp. 353–359, Nov. 1995.
140. C. Scholl and P. Molitor, "Communication based FPGA synthesis for multi-output Boolean functions," *Asia and South Pacific Design Automation Conference*, pp. 279–287, Aug. 1995.
141. C. Scholl, R. Drechsler, and B. Becker, "Functional simulation using binary decision diagrams," *ICCAD'97*, pp. 8–12, Nov. 1997.
142. K. J. Schultz, *CAM-Based Circuits for ATM Switching Networks,* PhD thesis, University of Toronto, 1996.
143. D. A. Simovici, D. Pletea, and R. Vetro, "Information-theoretical mining of determining sets for partially defined functions," *ISMVL-2010*, May 2010, pp.294–299.
144. H. Song, and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," *International Symposium on Field-Programmable Gate Arrays* (FPGA'05), Monterey, CA, Feb 20–22, 2005.
145. I. Sourdis, D. N. Pnevmatikatos, and S. Vassiliadis, "Scalable multigigabit pattern matching for packet inspection," *IEEE Trans. VLSI Syst.* Vol.16, No.2, pp.156–166, 2008.
146. V. Y-S Shen, A. C. Mckellar, and P. Weiner, "A fast algorithm for the disjunctive decomposition of switching functions," *IEEE Trans. on Comput.*, Vol. C-20, No. 3, pp. 304–309, March 1971.
147. W.-Z. Shen, J.-D. Huang, and S.-M. Chao, "Lamda set selection in Roth-Karp decomposition for LUT-based FPGA technology mapping," *32nd Design Automation Conference*, pp. 65–69, June 1995.

148. D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys*, Vol. 37, Issue 3, pp. 238–275, Sept., 2005.
149. M.Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," *ACM SIGCOMM Computer Communication Review*, Vol. 27, No. 4, pp. 25–38, 1997.
150. W. Wan and M. A. Perkowski,"A new approach to the decomposition of incompletely specified functions based on graph-coloring and local transformations and its application to FPGA mapping," *IEEE EURO-DAC'92*, pp. 230–235, Hamburg, Sept. 7–10, 1992.
151. S. J. E. Wilton, "SMAP: Heterogeneous technology mapping for FPGAs with embedded memory arrays," In *Proc. o f the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 171–178, 1998.
152. S. J. E. Wilton, J. Rose, and Z. Vranesic, "The memory/logic interface in FPGA's with large embedded memory arrays," *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, Vol. 7, Issue 1, March 1999, pp.80–91.
153. S. J. E. Wilton, "Heterogeneous technology mapping for area reduction in FPGAs with embedded memory arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* Vol. 19, Issue 1, Jan. 2000, pp.56–68.
154. I. Wegener, *Branching Programs and Binary Decision Diagrams: Theory and Applications*, Society for Industrial Mathematics, Jan. 1987.
155. B. Wurth, K. Eckl, and K. Anterich, "Functional multiple-output decomposition: Theory and implicit algorithm," *Design Automation Conf.*, pp. 54–59, June 1995.
156. http://www.xilinx.com
157. Xilinx,"Designing flexible, fast CAMs with Virtex family FPGAs," *Application Note*, XAPP203, Sept. 23, 1999.
158. Xilinx, "Advantages of the Virtex-5 FPGA, 6-input LUT architecture," WP284 (v1.0), Dec. 19, 2007, http://www.xilinx.com/
159. S. Yang, "Logic synthesis and optimization benchmark user guide, Version 3.0," *MCNC*, Jan. 1991.
160. S. S. Yau and C. K. Tang, "Universal logic modules and their applications," *IEEE Transactions on Computers*, vol. 19, no. 2, pp. 141–149, Feb. 1970.
161. F. Zane, G. Narlikar, and A. Basu, "CoolCAM: Power-efficient TCAMs for forwarding engines", *Proceeding of IEEE INFOCOM '03*, April, 2003.

Index