# Self-Restored Current-Mode CMOS Multiple-Valued Logic Design and Synthesis

A Thesis
Submitted to the College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in the
Department of Electrical Engineering
University of Saskatchewan
Saskatoon, Saskatchewan

by



Hsiang-Yung Teng

Spring, 2003

# PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, it is agreed that the Libraries of this University may make it freely available for inspection. Permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professors who supervised this thesis work or, in their absence, by the Head of the Department of Electrical Engineering or the Dean of the College of Graduate Studies and Research at the University of Saskatchewan. Any copying, publication, or use of this thesis, or parts thereof, for financial gain without the written permission of the author is strictly prohibited. Proper recognition shall be given to the author and to the University of Saskatchewan in any scholarly use which may be made of any material in this thesis.

Request for permission to copy or to make any other use of material in this thesis in whole or in part should be addressed to:

Head of the Department of Electrical Engineering
57 Campus Drive
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9

# ACKNOWLEDGEMENTS

There have been many people helped in making this thesis possible. First of all, I must acknowledge my gratitude to my supervisor, Dr. R. J. Bolton, who despite his many obligations made me a priority when it counted most. Words cannot express my appreciation for his constant supervision and constructive criticism throughout the work and the timely encouragement especially near the end of the journey. Without his continued belief in me, this thesis would not be completed.

I am deeply indebted to the members of advisory committee for their guidance and giving me the space to grow during these years. In particular, Dr. C. D. McCrosky helped me view my research from a different perspective and question what I had learned but not completely discount it. His insightful comments on the reporting of my research have been invaluable. I am very grateful to Dr. R. E. Gander for careful reading of the thesis chapters and to Dr. J. E. Salt for sharing his wealth of knowledge and experience so freely. I am also grateful to Dr. T. S. Sidhu, Dr. S. O. Kasap and Prof. D. E. Dodds for their administrative help. Special thanks are due to Dr. D. M. Miller from the University of Victoria for acting as the external examiner and asking questions that made me think in more depth.

I would like to extend my gratitude to the College of Graduate Study and Research and the Department of Electrical Engineering for assisting me in many different ways. Dr. T. B. Wishart, Ms. J. Matthies and Ms. H. Sobry deserve special mention. I would also like to express my appreciation to the University of Saskatchewan and the National Sciences and Engineering Research Council of Canada for providing financial assistance in the form of scholarship, and to Canadian Microelectronics Corporation for providing facilities to undertake this work.

Last but not least, I would like to thank Dr. K. K. Sun and Mrs. Y. C. Sun for treating me as their own son for all these years. It was their shared sense of excitement and confidence that reassured me it was all worthwhile. Warm thoughts go to my Sun sisters and brothers for all the emotional support and caring they provided, sometimes from thousands of miles away, and especially to Vivian who stood by me through all of the inevitable rough periods as well as the many enjoyable moments. Above all, they reminded me that there is more to life than a Ph.D.

# ABSTRACT

This thesis presents a self-restored current-mode CMOS multiple-valued logic (MVL) design architecture which consists of an input block, a control block, and an output block. A $r$-valued logic function to be implemented by this architecture is expressed in a sum of $r-1$ binary logic functions, $j \bullet V_j$ where $j$ is from 0 to $r-1$. That is, each binary function has an unique value for logic high. The binary logic functions are in *sum-of-product* (SOP) form, where a product term (PT) consists of multiple *and* operations on *up-literal* operators and the *sum* is an *or* operation. For a given MVL function, the input block implements *up literal* operations with current mirrors and thresholds. The control block implements the $r-1$ binary functions. The control signals from the control block turn on/off switches in the output block to generate the desired output signals directly from the current sources.

According to this architecture, the *sum* operation of the $r-1$ binary functions is an arithmetic sum and the *min* operation between $j$ and $V_j$ is implemented by properly connecting outputs of the control block to the switches — no extra transistors are required for these operations in a current-mode design. Also the input block functions as a MVL-binary converter and the control block is a voltage-mode binary logic circuit and therefore, they can be used as interfaces to external binary logic circuits without extra MVL-binary converters. The average transistor count of resulting circuits is 1.1 to 2.5 times smaller than that of other operator-based MVL designs without sacrificing speed and power. Variations to the architecture that make use of the arithmetic *sum* and *diff* in the input block and output blocks can further reduce the circuit size.

The self-restored MVL architecture allows MVL synthesis using a binary logic synthesizer. A computer program was developed to work together with a binary logic synthesizer to generates an area-optimized circuit for a given MVL function according to the self-restored MVL design architecture. An additional computer program was also designed to automatically derive equivalent binary logic circuits for a given MVL function for comparison purposes.

This thesis also proposes a new VHDL library for high-level simulation of multiple-valued Current-Mode CMOS Logic (CMCL) designs supporting faster verification of synthesized results without using a time-consuming circuit simulator such as SPICE or Spectre. The library has basic MVL entities (behavioral), complex MVL entities

(behavioral and structural) as well as standard binary logic gates. A bus resolution function working cooperatively with the basic MVL entities allows MVL logic levels (currents) in individual connections to be displayed. Design examples of a quaternary full adder and bit-slice circuit of a transversal matched filter are presented along with both VHDL and circuit simulation results. The design examples verify that the CMCL library allows the VHDL simulation of current-mode CMOS logic using Leapfrog. Spectre was used to confirm that the VHDL simulations were correct. The circuits were also verified by importing the CMCL library into VSS and performing the simulation.

# Table of Contents

# List of Tables

# List of Figures

# List of Abbreviations

| | |
|---|---|
| ACM | Active Current Mirror |
| ASOP | Arithmetic-Sum-of-Product |
| BCM | Bi-Directional Current Mirror |
| BiCMOS | Bipolar Complementary Metal Oxide Semiconductor |
| CMC | Canadian Microelectronics Corporation |
| CMCL | Current-Mode CMOS Logic |
| CMOS | Complementary Metal-Oxide Semiconductor |
| DCM | Dynamic Current Mirror |
| EDA | Electronic Design Automation |
| HDL | Hardware Description Language |
| IC | Integrated Circuit |
| KCL | Kirchhoff's Current Law |
| MODSUM | Modulus Sum |
| MOP | Max-of-Product |
| MOS | Metal Oxide Semiconductor |
| MLC | Multiple-Levels-per-Cell |
| MVL | Multiple-Valued Logic |
| NCM | N-Type Current Mirror |
| PCM | P-Type Current Mirror |
| PT | Product Term |
| SOP | Sum-of-Products |
| SOT | Sum of Logic Terms |
| TSUM | Truncated Sum |
| VHDL | VHSIC Hardware Description Language |
| VLSI | Very Large Scale Integration |
| VHSIC | Very High-Speed Integrated Circuit |
| VMCL | Voltage-Mode CMOS Logic |

# 1. Introduction

As the result of the advance of integrated circuit (IC) fabrication processes and the development of electronic design automation (EDA) tools, the design complexity of very large scale integrated (VLSI) circuits has increased dramatically over the past decade. Together with the increasing complexity of VLSI circuits many problems arose, such as higher percentage of interconnection area, routing difficulty and packaging. Improper interconnections not only result in larger chip area but also cause timing and cross-talk problems. These problems are more prominent in deep-submicron designs. Partial solutions to these problems include multiple metal layers, flip-chip, etc. A deep submicron process normally has four to six metal layers. For example, LSI Logic G11 0.18$\mu$m technology provides five metal layers.

It is well known that multiple-valued logic (MVL) circuits are suitable for reducing the interconnecting complexity [1, 2]. Unlike binary logic, MVL allows more than two values to exist in a logic system. Aside from the philosophical meaning of these logic values, an immediate benefit is improved overall information efficiency. Each $r$-valued signal can carry $\log_2 r$ times the information available in a binary signal. Wiring area is reduced on a logarithmic scale with the increase of $r$. Referring to Figure 1.1, the wiring area of a 4-valued logic design could be $\log_2 4$ or $(\log_2 4)^2$ times smaller. Moreover, for a $r$-valued $n$-variable function, $f(x_0, x_1, \cdots, x_{n-1})$, where $x_i \in \mathbf{R} = \{0, 1, 2, \cdots, r-1\}$

$$f : \mathbf{R}^n \longrightarrow \mathbf{R}$$

there are $r^{r^n}$ different possible combinations of outputs. For example, if $r = 4$, $f(x_0, x_1)$ can implement $4^{4^2}$ possible functions. It has been therefore believed that

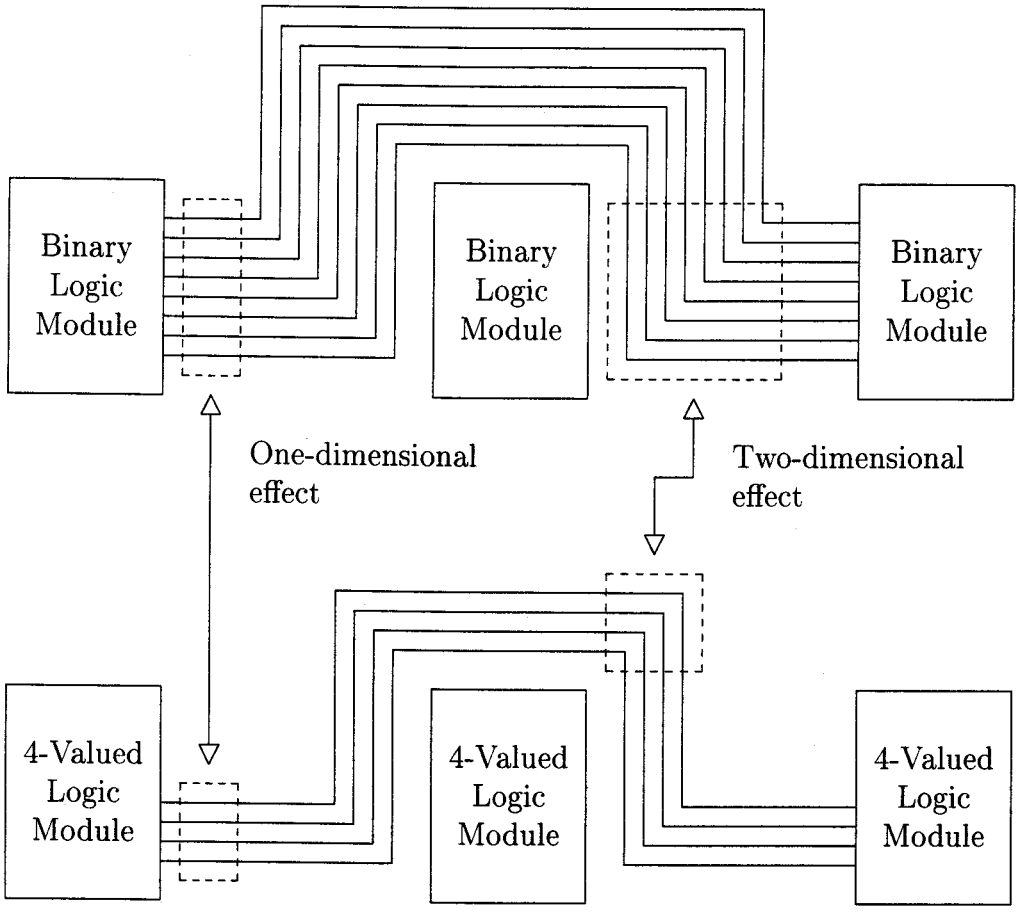



**Figure 1.1** Reduction of wiring area by using MVL.

multiple-valued logic is advantageous in reducing the number of interconnections and attaining higher functional density.

In fact, the optimal logic radix in terms of implementation cost were studied by several researchers before.

- According to G. Abraham [3], optimal logic radix is equal to the Euler constant, $e \approx 2.718$.
- According to S. L. Hurst [4], the circuit implementation cost is decreasing with increasing logic radix.
- According to C. M. Allen and D. D. Givone [5], the optimal radix is greater than $e$.

Their conclusions are quite different since they are based on different assumptions, but they all agree that the optimal logic radix is greater than 2. In practice the value of $r$ must be an integer. This implies that the radix should be at least 3, i.e., ternary logic. Ternary logic circuits can be interfaced with binary logic circuits by using two binary bits to represent a ternary value. For example, for ternary logic values {0,1,2}, $00_2$ represents $0_3$, $01_2$ represent $1_3$, and $10_2$ represents $2_3$; for ternary logic values {-1,0,1}, $00_2$ again represents $0_3$, $01_2$ represent $1_3$, and $10_2$ represents $-1_3$. Using these schemes (two of many), one combination of two binary bits, i.e., $11_2$ in this case, is not assigned to any logic value. In order to allow full utilization of the available functions, it is preferable to have radix, $r$, be a power of 2. Higher radix enables more information per bit in MVL designs, but the noise tolerance limits the value of $r$. No matter whether a current-mode design approach or a voltage-mode design approach is used, the most practical choice of the radix is $r = 4$. The next most interesting radix is $r = 8$.

As far as CMOS dominates IC technologies, current-mode CMOS logic (CMCL) appears to be the natural choice for MVL designs since the radix is less limited by the combination of device and supply properties [1]. CMCL is able to implement the arithmetic sum and subtraction by simply connecting wires together by Kirchhoff's Current Law (KCL). That is, no additional transistors are required for implementing the arithmetic operations. This feature of CMCL is very useful for further reducing the chip area of arithmetic processing chips [6, 7].

Voltage-mode CMOS logic (VMCL) is good for MVL memory design [8]. In 1997, Intel unveiled the first two bit/cell StrataFlash$^{TM}$ memory device based on their multiple-levels-per-cell (MLC) technology [9, 10]. It is reported that the 64 Mbit two bit/cell Intel StrataFlash memory is just 5% larger than the 32 Mbit one bit/cell device on the same standard 0.4$\mu$ ETOX$^{TM}$ flash memory process at the same yields, delivering on the promise of 2x the bits in 1x the space and setting a new cost paradigm for flash memory devices. The current StrataFlash series memory are manufactured

on Intel's $0.18\mu$ and $0.25\mu$ process technologies with $2.7V$ to $3.6V$ $V_{CC}$ operation.

Summing up the above, the major reason for using MVL circuits in VLSI is that MVL can make better use of the chip area through increased functional density and reduced number of interconnections. The reduced number of interconnections also implies fewer pins needed on IC packages and reduced routing complexity. Considering the increasing need of arithmetic processing chips, there should be substantial room for MVL circuits to compete with binary circuits.

## 1.1 Research Motivation

Despite the potential advantages of MVL design and recognition of the optimal radix, today's digital designs are still dominated by binary logic. Investigation into the current MVL designs shows there exist problems hindering MVL from growth, such as deteriorating signal integrity and low noise immunity. It has been therefore difficult to convince IC designers to explore MVL alternatives as long as CMOS dominates the IC industry. These drawbacks must be overcome.

The most significant reason for slow progress in MVL VLSI is tighter device tolerances and reduced noise immunity. Unlike the currently dominant voltage-mode CMOS binary logic, MVL circuits are not self-restoring. It is necessary to insert a logic-level restorer after several stages to recover signals. The number of stages depends on various factors such as fabrication processes, design architecture and operators. There has not been a general rule to determine when to use a logic-level restorer. The extra level-restorers increase the implementation cost of CMCL MVL significantly and neutralize the benefit of MVL design. This is against the MVL researcher's recognition of optimal logic radix. Therefore, they should be able to reduce the circuit size of MVL designs. One way to accomplish it is to remove the level-restorer.

CMCL MVL signals are not self-restored due to the limitation of the range of

the operating current and the sloped $I$-$V$ characteristic of the basic elements. The first factor can be eliminated by keeping the total input current to a node below the maximum current, but the second factor exists as a property of the basic elements. The overall performance of a CMCL design can be greatly improved if the problem can be resolved.

The problem of self-restoration can be tackled from four levels: device level, circuit level, gate level and architecture level. A realization of this idea at the device level is the *Multi-Step Function* (MSF) MOSFET [11] with a stair-like $I$-$V$ curve. This device can be used as a current mirror or as a single transistor quantizer to construct robust multiple-valued CMCL circuits. However, the MSF MOSFET prototype is quite large ($500\mu\text{m}\times500\mu\text{m}$) and fabrication of this device requires process modification. Also, lack of a SPICE model for the MSF MOSFET means that it cannot be simulated using SPICE without using an equivalent circuit.

A study of this problem at the circuit level is disclosed in [12]. It can be seen that the different elements cause different levels of the adverse effect. For example, threshold elements cause little problem with logic levels. A constant element also functions as a 'constant' current source in most cases. The most problematic element is the current mirror. Although other types of current mirrors, such as cascode current mirrors and active current mirrors provide better signal integrity than a simple current mirror does, the signal still deteriorates stage by stage. The active current mirror also increases circuit size significantly. The MVL operators constructed by using the CMCL elements also suffers different degrees of signal deterioration. For example, output signals from *min*, *max*, and *cycle* operators are the worst among the MVL operators. *tsum* and *modsum* operators are worse than the *literal* operators. The problem caused by current mirrors must be compensated for.

The problem of self-restoration appears to be best solved at the architecture level. In 1994, Chang, et al. [13] proposed a hybrid-mode CMOS MVL architecture to attack this problem. The average cost of circuits using Chang's scheme seems lower

than other synthesis schemes as mentioned in the paper. As MVL circuits will not survive unless they possess an overwhelming advantage over binary logic circuits or at least have no fatal drawbacks, it is important to evaluate whether there is such an advantage or drawback of a MVL design architecture. Chang's scheme is not comparable to binary implementation in cost. Output signals are vulnerable because they are the difference of a P-type current mirror output and a N-type current mirror. Speed is slow as P-type thresholds are used. Also, there is static power consumption when inputs are logic 0.

The lack of appropriate EDA tools also retards the progress of MVL. The methodology of high-level design and synthesis is widely used in the binary design world, but the EDA tools for MVL designs are still in their infancy. MVL researchers have developed a number of algorithms and programs for minimization of MVL functions. In general, they are following in the same footsteps as the binary logic to develop synthesizers for MVL designs. A variety of minimization techniques have been proposed in the past. For circuit implementation of MVL designs these techniques are all based upon implementation-oriented multiple-valued algebra. The two well-known implementation-oriented multiple-valued algebras were proposed by C. M. Allen, et al. [5, 14] and Z. G. Vranesic, et al. [15] separately. Allen, et al. also proposed a minimization technique in the same paper based on *min*, *max*, and *literal* operators. In the Vranesic algebra, MVL functions are expressed and minimized by using *min*, *max*, and *cycle* operators. The definitions of these operators are given in Section 2.3.2. One of the most notable CAD tools is HAMLET (Heuristic Analyzer for Multiple-valued Logic Expression Translation) developed by Yurchak and Butler [16]. Some recent developments are proposed in [17, 18, 19, 20, 21, 22, 23].

Nowadays, most MVL researchers [24, 25] are generally of the opinion that MVL cannot survive on its own and has to coexist with binary logic. It is quite normal to use binary gates inside a MVL circuit to generate control signals or outside the circuits to interface with binary logic circuits. This implies that mixed-radix system architecture

will be a future trend of MVL designs. For a mixed-radix design, it is essential to allow communications between binary and MVL subsystems. Two bus structures to handle the interface between MVL and binary circuits were proposed by Z. G. Vranesic [4, 26]: a binary bus structure and a MVL bus structure. Figure 1.2 is a schematic diagram of the binary bus structure. Figure 1.3 is a schematic diagram of the MVL bus structure. Both bus structures can have the MVL circuits implementable by the same fabrication process used for binary logic circuits, thus minimizing process-related overhead. Therefore the use of MVL circuits in VLSI design must consider the ease of interfacing MVL sub-systems with binary sub-systems. To this end, EDA tools for MVL designs must incorporate capabilities to handle both the MVL and binary logic.

Considering the fact that a variety of binary logic synthesizers, such as Synopsys Designer Analyzer and Cadence Ambit, are commercially available and well accepted in the industry, it is interesting to see if those binary logic synthesizers can be extended to MVL synthesis. For example, the use of the Synopsys Design Analyzer for structural synthesis of VHDL description of MVL designs was studied in [12]. The use of binary logic synthesizers for MVL synthesis also allows us to explore the use of a single tool for a mixed-radix logic design.

Since MVL designs involve multiple logic levels, it was inevitable to verify them using analog simulators. Circuit simulators such as SPICE are widely available today. A major inconvenience of using a circuit simulator is the time required to perform the simulations. A more convenient method would be to use a high-level hardware description language (HDL) for circuit designs so that functional simulation of the circuit description can be performed. Nevertheless, both VHDL and Verilog were not designed for analog simulation as a primary requirement. The Accellera Verilog-AMS Technical Subcommittee and the IEEE 1076.1 Working Group have been working on analog and mixed-signal extensions to both languages. By taking advantages of VHDL's flexible value system and customizable bus resolution

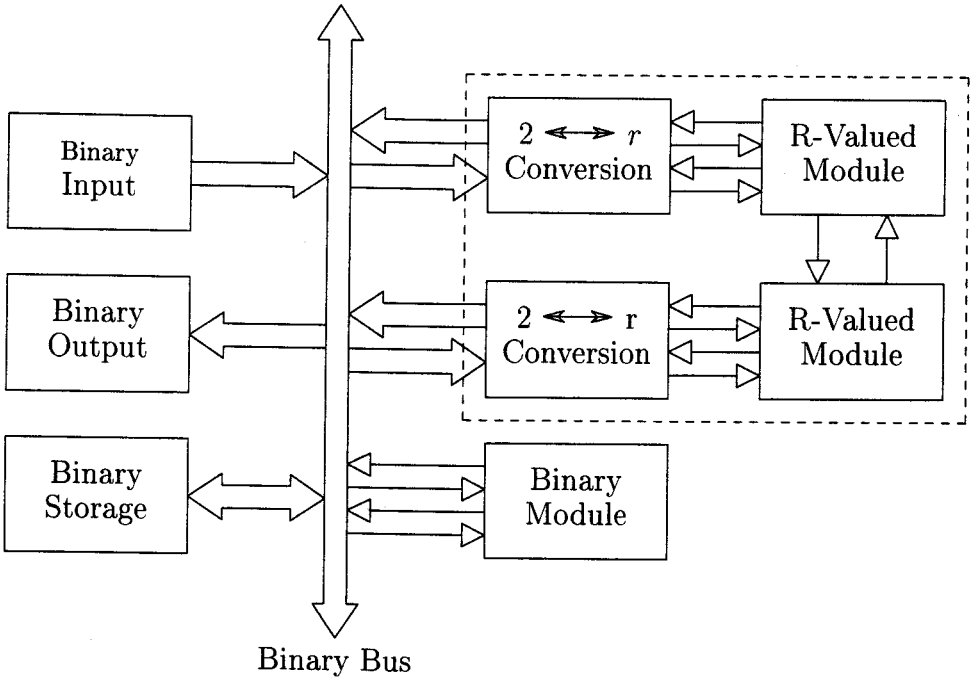



**Figure 1.2** Binary bus system architecture.

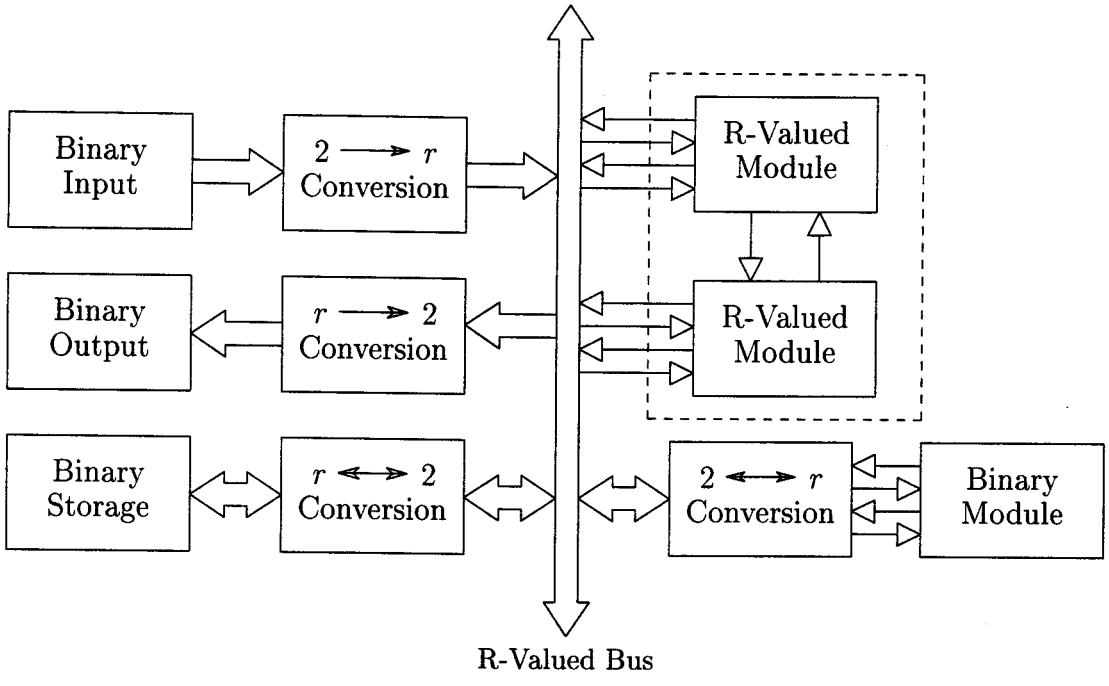



**Figure 1.3** Multiple-valued bus system architecture.

function [27, 28, 29, 30], a library which allows VHDL simulation of mixed CMCL and binary designs was proposed in the author's Masters thesis [12]. The library in [12] suffers from two drawbacks. First of all, the resolution function is so complex that maintenance and improvement are extremely difficult. A complex resolution function also slows down simulation speed. Secondly, the switch models are not robust enough to handle designs with more than two series-connected switches. It is therefore desirable to redesign the library to further increase the simulation speed while adding the capability to deal with various circuit topologies.

## 1.2 Research Objectives

The objectives of this thesis are to address the aforementioned problems and to propose solutions to the problems.

- To reduce the circuit size of MVL designs.
- To propose a self-restored MVL design architecture.
- To use a binary logic synthesizer for synthesis of the self-restored MVL design architecture.
- To design a new VHDL library for high-level simulation of mixed CMCL and binary logic designs.

## 1.3 Outline of Thesis

Chapter 2 gives the background material of MVL including multiple-valued algebra, notation, definitions, and circuit design.

Chapter 3 describes the self-restored design architecture and associated theorems. Design examples are also provided in this chapter to demonstrate the minimization scheme with the self-restored design architecture.

Chapter 4 describes the approaches of using the arithmetic *sum* and *diff* operators to reduce circuit size of the self-restored current-mode CMOS MVL design.

Chapter 5 presents two computer programs, WMS and M2B. The WMS program, working together with a binary logic synthesizer, is used to synthesize a MVL function based upon the self-restored design architecture. The M2B program, which is modified from the WMS program, is used for generating an equivalent binary logic circuit for a given MVL function.

Chapter 6 describes a VHDL library by which synthesized results can be verified using a VHDL simulator. It starts with the library structure, followed by the detailed design of VHDL packages, bus resolution function, circuit functions, operator functions, and binary function. Most importantly, interaction between resolution function and circuit functions to determine currents in interconnections will be described.

Chapter 7 compares the self-restored MVL design with other MVL design schemes and binary logic in terms of area, speed and power dissipation.

Thesis conclusions and potential future research work are given in Chapter 8.

# 2. Multiple-Valued Logic

MVL originated from doubt about the so-called *Law of the Excluded Middle* by ancient Greek philosophers. However, this topic had been only within the scope of philosophy [31, 32] until in the early twentieth century when mathematicians began devising multiple-valued algebras [33, 34, 35]. A variety of circuit realizations of MVL design have been widely discussed over the past three decades. The application of CMCL to MVL design has received more attention than VMCL in recent years. CMCL design allows higher radix and better noise margin than VMCL implementation of MVL functions.

In this chapter, fundamental knowledge of MVL is introduced with emphasis on the implementation-oriented multiple-valued algebra and notation, as well as CMCL circuit elements and design.

## 2.1 Definitions

In order to explain the multiple-valued algebra and MVL design, some definitions are given in this section. It should be noticed that there are no standard notation and definitions of MVL operations. In the following definitions, it is assumed that $a, b, c, k$ are elements of a finite set, defined as $\mathbf{R} = \{0, 1, \cdots, r-1\}$ where $r > 1$, and $a \leq b$ when they appear in a same definition. Also, it should be noted that Definitions 2.4 to 2.11 are unary operators while Definitions 2.12 and 2.13 are binary operators.

**Definition 2.1** A $r$-valued variable $X$ can take on values from $\mathbf{R} = \{0, 1, \cdots, r-1\}$ where $r > 1$.

**Definition 2.2** A $r$-valued function $F$ of $n$ variables is a mapping

$$F(X_1, X_2, \cdots, X_n) : \mathbf{R}^n \longrightarrow \mathbf{R} \tag{2.1}$$

where $X_i$ is a $r$-valued variable and $n \geq 1$.

**Definition 2.3** A $r$-valued-input binary-output function $F$ of $n$ variables is a mapping [36]

$$F(X_1, X_2, \cdots, X_n) : \mathbf{R}^n \longrightarrow \mathbf{B} \tag{2.2}$$

where $X_i$ is a $r$-valued variable, $\mathbf{B} = \{0, l\}$, $l \geq 1$, and $n \geq 1$.

**Definition 2.4** A *clockwise cycle* operator, denoted as $X^{c\rightarrow}$, is defined as [15, 33]:

$$X^{c\rightarrow} = (X + c) \bmod r \tag{2.3}$$

**Definition 2.5** A *counter-clockwise cycle* operator, denoted as $X^{c\leftarrow}$, is defined as [15]:

$$X^{c\leftarrow} = (X - c) \bmod r \tag{2.4}$$

Comparing Definition 2.4 and Definition 2.5, it can be found that $X^{c\leftarrow} = X^{r-c\rightarrow}$ because $(X - c) \bmod r = (X - c + r) \bmod r$.

**Definition 2.6** A *literal* of $X$, denoted as ${}^aX^b$, is defined as [14]:

$${}^aX^b = \begin{cases} r - 1 & \text{when } a \leq X \leq b \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

**Definition 2.7** A *complement of literal* of $X$, denoted as $\overline{{}^aX^b}$, is defined as [37]:

$$\overline{{}^aX^b} = \begin{cases} r - 1 & \text{when } X < a \text{ or } X > b \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

It should be noted that ${}^{0}X^{r-1} = r-1$ and $\overline{{}^{0}X^{r-1}} = 0$. That is, ${}^{a}X^{b}$ and $\overline{{}^{a}X^{b}}$ are constants when $a = 0$ and $b = r-1$.

**Definition 2.8** An *up literal* of $X$, denoted as ${}^{a}X$, is defined as [38]:

$$ {}^{a}X = \begin{cases} r-1 & \text{when } X \geq a \\ 0 & \text{otherwise.} \end{cases} \tag{2.7} $$

**Definition 2.9** A *complement of up literal* of $X$, denoted as $\overline{{}^{a}X}$, is defined as [39]:

$$ \overline{{}^{a}X} = \begin{cases} r-1 & \text{when } X < a \\ 0 & \text{otherwise.} \end{cases} \tag{2.8} $$

**Definition 2.10** A *down literal* of $X$, denoted as $X^{b}$, is defined as [39]:

$$ X^{b} = \begin{cases} r-1 & \text{when } X \leq b \\ 0 & \text{otherwise.} \end{cases} \tag{2.9} $$

**Definition 2.11** A *complement of down literal* of $X$, denoted as $\overline{X^{b}}$, is defined as [39]:

$$ \overline{X^{b}} = \begin{cases} r-1 & \text{when } X > b \\ 0 & \text{otherwise.} \end{cases} \tag{2.10} $$

It can be seem from the definitions that all of the *literal* operators are actually $r$-valued input binary output mapping functions. The *up literal* and *down literal* are both *threshold literals* [38].

**Definition 2.12** A *min* operation, denoted as $\bullet$, is defined as [5]:

$$ X \bullet Y = \begin{cases} X & \text{when } X \leq Y \\ Y & \text{otherwise,} \end{cases} \tag{2.11} $$

where $X$ and $Y$ are two $r$-valued variables.

**Definition 2.13** A *max* operation, denoted as $\dotplus$, is defined as [5]:

$$X \dotplus Y = \begin{cases} X & \text{when } X \geq y \\ Y & \text{otherwise,} \end{cases} \tag{2.12}$$

where $X$ and $Y$ are two $r$-valued variables.

**Definition 2.14** A *product term* (PT) is *min* operations of a non-zero constant in $\mathbf{R}$ and a set of unary operations on $X_1, X_2, \cdots, X_n$ where an unary operations is any one of Definitions 2.4 to 2.11.

**Definition 2.15** A *literal product term* (*literal* PT), denoted as

$$k \bullet {}^{a_1}\hat{X}_1^{b_1} \bullet {}^{a_2}\hat{X}_2^{b_2} \bullet \cdots \bullet {}^{a_n}\hat{X}_n^{b_n}, \tag{2.13}$$

is defined as a *min* operations of a set of *literals* and *complement of literals*, where $k$ is a non-zero constant in $\mathbf{R}$, ${}^{a_i}\hat{X}_i^{b_i}$ is either ${}^{a_i}X_i^{b_i}$ or $\overline{{}^{a_i}X_i^{b_i}}$, and $\bullet$ represents the *min* operation.

**Definition 2.16** A *literal minterm* is a product term of the form

$$k \bullet {}^{a_1}X_1{}^{a_1} \bullet {}^{a_2}X_2{}^{a_2} \bullet \cdots \bullet {}^{a_n}X_n{}^{a_n}, \tag{2.14}$$

which contains each variable of a function $F$.

**Definition 2.17** Two literal minterms, $k_a \bullet {}^{a_1}X_1{}^{a_1} \bullet {}^{a_2}X_2{}^{a_2} \bullet \cdots \bullet {}^{a_n}X_n{}^{a_n}$ and $k_b \bullet {}^{b_1}X_1{}^{b_1} \bullet {}^{b_2}X_2{}^{b_2} \bullet \cdots \bullet {}^{b_n}X_n{}^{b_n}$ where $k_a$ and $k_b$ are non-zero constants in $\mathbf{R}$, are said to be *distinct* if at least one ${}^{a_i}X_i{}^{a_i} \neq {}^{b_i}X_i{}^{b_i}$ regardless of $k_a$ and $k_b$. There can be a maximum of $r^n$ number of distinct literal minterms for a $n$-variable $r$-valued logic function [40].

For 2-variable 4-valued logic functions, the maximum number of distinct literal minterms is $4^2 = 16$.

## 2.2 Multiple-Valued Algebra

Multiple-valued algebras were developed independently by J. Łukasiewicz [35] and E. Post [33] in the 1920's. According to L. Bolc [35],

> "Łukasiewicz made an attempt to modify the propositional calculus so as to achieve a kind of conceptual synthesis of determinism and indeterminism. This gave rise to three-valued logic.".

Post developed the first functionally complete many-valued logic of order $n$ [33]. Unlike Łukasiewicz, however, Post was not led by any philosophical motivations [35].

The usage of multiple-valued algebra in circuit design emerged just three decades ago. The reason is that neither the original definition of Post algebra nor the modified definitions [34, 41] are implementation-oriented. In order to put MVL to practical use, several implementation-oriented algebras have been formulated, such as the algebra of Vranesic et al. [15], and the algebra of Allen and Givone [5]. The implementation-oriented multiple-valued algebras have similar definitions as that of Boolean algebra. For example, the implementation-oriented multiple-valued algebra of Allen and Givone uses *min*, *max* and *literal* operators. It can be seen the axioms shown in Table 2.1 hold for any $r$-valued variables, $X$, $Y$ and $Z$.

From definition of the *max* operator it can be seen that

$$ {}^{a}X^{b} = {}^{a_1}X^{b_1} \dotplus {}^{a_2}X^{b_2} \tag{2.15} $$

if and only if $a = a_1 \bullet a_2$, $b = b_1 \dotplus b_2$, $a_2 - 1 \leq b_1$, and $a_1 - 1 \leq b_2$. Similarly, from the definition of the *min* operator it can be seen that

$$ {}^{a}X^{b} = {}^{a_1}X^{b_1} \bullet {}^{a_2}X^{b_2} \tag{2.16} $$

if and only if $a = a_1 \dotplus a_2$, $b = b_1 \bullet b_2$, $a_1 \leq b_2$, and $a_2 \leq b_1$. Eqs. 2.15 and 2.16 are the major rules in the algebra of Allen and Givone for minimization. The multiple-valued

**Table 2.1** Axioms for $r$-valued variables $X$, $Y$, $Z$ of a multiple-valued algebra.

| | | |
|---|---|---|
| *closure* | $X + Y \in \mathbf{R}$ | $X \bullet Y \in \mathbf{R}$ |
| *idempotent* | $X + X = X$ | $X \bullet X = X$ |
| *commutative* | $X + Y = Y + X$ | $X \bullet Y = Y \bullet X$ |
| *associative* | $(X + Y) + Z = X + (Y + Z)$ | $(X \bullet Y) \bullet Z = X \bullet (Y \bullet Z)$ |
| *absorption* | $X + (X \bullet Y) = X$ | $X \bullet (X + Y) = X$ |
| *distributive* | $X + Y \bullet Z = (X + Y) \bullet (X + Z)$ | $X \bullet (Y + Z) = X \bullet Y + X \bullet Z$ |
| *null element* | $X + 0 = X$ | $X \bullet 0 = 0$ |
| *universal element* | $X + (r-1) = r-1$ | $X \bullet (r-1) = X$ |

algebra of Allen and Givone is equivalent to Boolean algebra when $r$ is 2. However, the only meaningful unary operator is the *complement* in Boolean algebra. Multiple unary operations can be used in a multiple-valued algebra. There are $r^r$ possible definitions for unary operations in theory. Some of the unary operators were given in Definitions 2.6 to 2.11. In order to simplify the design task and to take advantage of the IC fabrication process, other operators like *nand*, *nor*, *xor* and *xnor* are also used in binary logic designs even though *not*, *and* and *or* operators are functionally complete. MVL design also uses additional operators for the same reason. Two additional operators, *tsum* and *modsum*, are defined in 2.18, and 2.19.

**Definition 2.18** A *tsum* (truncated sum) operator, denoted as $\boxplus$, is defined as [42]:

$$\begin{aligned} X_1 \boxplus X_2 \boxplus \cdots \boxplus X_n &= tsum(X_1, X_2, \cdots, X_n) \\ &= min(X_1 + X_2 + \cdots + X_n, r-1) \end{aligned} \tag{2.17}$$

It should be noted that the *tsum* operation and the *max* operation give the same result if and only if only one $X_i \in \mathbf{R} = \{0, 1, \cdots, r-1\}$ is not zero at a time.

**Definition 2.19** A *modsum* (modulus sum) operator, denoted as $\oplus$, is defined as [43]:

$$\begin{aligned} X_1 \oplus X_2 \oplus \cdots \oplus X_n &= modsum(X_1, X_2, \cdots, X_n) \\ &= (X_1 + X_2 + \cdots + X_n) \bmod r \end{aligned} \tag{2.18}$$

It can be seen from Definition 2.19 that the *modsum* operator corresponds to the *xor* operator in binary logic when the logic radix $r = 2$.

There are $r^{r^2}$ possible definitions of binary MVL operators. Other binary MVL operators have been also proposed in the past (Appendix B), but only the above operators will be used in this thesis. An important MVL theory is also given in Theorem 2.1.

**Theorem 2.1** Any multiple-valued logic function $F$ of $n$ variables can be expressed in a max-of-products (MOPs) form [5]:

$$F = \sum_k \!\!+ \; c_k \bullet {a_{k_1}}X_1^{b_{k_1}} \bullet {a_{k_2}}X_2^{b_{k_2}} \bullet \cdots \bullet {a_{k_n}}X_n^{b_{k_n}} \tag{2.19}$$

where $c_k, a_{k_i}, b_{k_i} \in \mathbf{R}$.

The expression stated in Theorem 2.1 is not the only form to represent $n$-variable $r$-valued logic functions. For example, a more general form using both *literal* and *complement of literal* was proposed in [37]. The *max* operations can be also replaced with *tsum* [44, 45] or *modsum* [43] operations. Another form that uses arithmetic sum in place of the *max* operator will be described below.

**Lemma 2.1** Any $n$-variable $r$-valued logic function, $F$, can be expressed with a maximum of $r^n$ number of distinct literal minterms in a MOP form.

**Proof:** Let $(a_1, a_2, \cdots, a_n)$ be an element in $\mathbf{R}^n$ such that $F(a_1, a_2, \cdots, a_n) = k$ where $k \in \mathbf{R}$. Then $F$ can be represented by

$$F = \sum \!\!+ \; k \bullet {a_1}X_1^{a_1} \bullet {a_2}X_2^{a_2} \bullet \cdots \bullet {a_n}X_n^{a_n} \tag{2.20}$$

This expression is a MOP form. The maximum number of distinct literal minterms is $r^n$ by Definition 2.17.

**Definition 2.20** Two $n$-variable $r$-valued functions $F$ and $G$ are disjoint if and only if $F \bullet G = 0$. In other words, there is no $(a_1, a_2, \cdots, a_n) \in \mathbf{R}^n$ such that

$$F(a_1, a_2, \cdots, a_n) \neq 0 \quad \text{and} \quad G(a_1, a_2, \cdots, a_n) \neq 0. \tag{2.21}$$

**Lemma 2.2** For disjoint MVL functions $F_1, F_2 \cdots F_n$

$$F_1 \dotplus F_2 \dotplus \cdots \dotplus F_n = F_1 + F_2 + \cdots + F_n, \tag{2.22}$$

where $+$ is an arithmetic sum.

**Proof:** By Definition 2.20 there does not exist a $(a_1, a_2, \cdots, a_n) \in \mathbf{R}^n$ such that more than one $F_i \neq 0$ at a time where $i \in \{0, 1, \cdots, n\}$ since $F_1, F_2, \cdots, F_n$ are disjoint. Therefore, $F_1 \dotplus F_2 \dotplus \cdots \dotplus F_n = F_1 + F_2 + \cdots + F_n = F_i$ or $F_1 \dotplus F_2 \dotplus \cdots \dotplus F_n = F_1 + F_2 + \cdots + F_n = 0$ for any $(a_1, a_2, \cdots, a_n) \in \mathbf{R}^n$. As a result, the *max* operations on $F_i$ functions give the same result as the arithmetic sum operations do.

**Lemma 2.3** Distinct literal minterms are disjoint.

**Proof:** Assume $m_a = k_a \bullet {}^{a_1}X_1{}^{a_1} \bullet {}^{a_2}X_2{}^{a_2} \bullet \cdots \bullet {}^{a_n}X_n{}^{a_n}$ and $m_b = k_b{}^{b_1}X_1{}^{b_1} \bullet {}^{b_2}X_2{}^{b_2} \bullet \cdots \bullet {}^{b_n}X_n{}^{b_n}$ are two distinct literal minterms where $k_a \neq 0$ and $k_b \neq 0$. Then there is at least one $a_i \neq b_i$ by Definition 2.17; i.e., there does not exist a $(a_1, a_2, \cdots, a_n) \in \mathbf{R}^n$ such that $m_a \neq 0$ and $m_b \neq 0$. Therefore, $m_a$ and $m_b$ are disjoint.

**Theorem 2.2** Any multiple-valued logic function $F$ of $n$ variables can be expressed in an arithmetic-sum-of-products (ASOPs) form:

$$F = \sum_k{}^{+} c_k \bullet {}^{a_{k_1}}X_1{}^{b_{k_1}} \bullet {}^{a_{k_2}}X_2{}^{b_{k_2}} \bullet \cdots \bullet {}^{a_{k_n}}X_n{}^{b_{k_n}} \tag{2.23}$$

where $c_k, a_{k_i}, b_{k_i} \in \mathbf{R}$.

**Proof:** A $n$-variable $r$-valued logic function can be represented with $r^n$ distinct literal minterms by Lemma 2.1. Since distinct literal minterms are disjoint by Lemma 2.3, the *max* operations on the literal minterms can be replaced with arithmetic sum operations by Lemma 2.2.

**Lemma 2.4** ${}^aX^b = {}^aX \bullet X^b$ and $\overline{{}^aX^b} = \overline{{}^aX} + \overline{X^b}$.

**Proof:** The proof is self-evident by Definitions 2.6 to 2.11. ${}^aX^b = {}^aX^{r-1} \bullet {}^0X^b = {}^aX \bullet X^b$, and $\overline{{}^aX^b} = \overline{{}^aX^{r-1}} + \overline{{}^0X^b} = \overline{{}^aX} + \overline{X^b}$.

**Lemma 2.5** ${}^kX = \overline{X^{k-1}}$ and $X^{k-1} = \overline{{}^kX}$ for $k \geq 1$.

**Proof:** By Definitions 2.8 and 2.11 the *up literal* is equal to its corresponding *complement of down literal.* By Definitions 2.10 and 2.9 the *down literal* is equal to its corresponding to *complement of up literal.*

**Theorem 2.3** Any many-valued logic function $F$ of $n$ variables can be represented with *up literal*, *min* and *arithmetic sum* operators.

**Proof:** By Lemma 2.4, every literal in Eq. 2.23 can be represented with *min* operation of a *up literal* and a *down literal.* By Lemma 2.5, every *down literal* can be replaced with a *complement of up literal.* Therefore, Eq. 2.23 can be expressed in terms of *up literal*, *min* and *arithmetic sum.*

As will be described in Chapters 3 and 4, Theorems 2.2 and 2.3 are important to current-mode MVL design as the *arithmetic sum* operation can be realized by simply wiring signals together according Kirchhoff's Current Law (KCL). In other words, the *arithmetic sum* operation does not require additional circuit elements as opposed to *max* operators.

## 2.3 Circuit Realizations of Multiple-Valued Logic

Based upon the multiple-valued algebra defined in the previous section, different circuit implementations have been proposed in the past. The first successful approach to realization of MVL operators was using integrated injection logic ($I^2L$) circuits [46]. This technology led to a commercial multiple-valued IC. Unfortunately, the lack of acceptance of binary $I^2L$ directly caused the failure of multiple-valued $I^2L$ design [47]. Nowadays, multiple-valued circuits designers generally agree that current-mode logic, in particular that which uses CMOS technologies, is the most appropriate one for multiple-valued circuits [1].

Voltage-mode logic is not chosen as the potential candidate because it limits the available radix. Voltage-mode logic will be increasingly difficult to fabricate due to device tolerances and noise margins at low voltages. Despite this, an interesting possible exception is the applications of the neuron MOS ($\nu$MOS) transistor [48, 49]. With the exception of $\nu$MOS, current-mode CMOS logic will be the mainstream of MVL design since its range is less limited by the combination of device and supply properties [1].

In either CMCL or current-mode BiCMOS logic, two adjacent logic levels are normally different by $10\mu$A or $20\mu$A, defined as the *base current*, $I_0$. For example, if the base current is $20\mu$A, then logic 1 is represented by $20\mu\text{A}(I_0)$, logic 2 is represented by $40\mu\text{A}(2I_0)$, and so on. For higher radix ($r \geq 4$) MVL circuits, $10\mu$A is usually used as base current [50], but considering noise immunity and process variations, it is preferable to use $20\mu$A as the base current for ternary or quaternary logic. This thesis focuses on quaternary logic with the base current $I_0$ equal to $20\mu$A.

### 2.3.1 MVL Circuit Elements

The basic current-mode building elements for analog VLSI, including *constants*, *current mirrors*, *thresholds*, *switches*, and *sum* [51, 52, 53], have been successfully

applied to current-mode MVL operators in the past using 3$\mu$m or 5$\mu$m technologies [42, 54]. They are listed in Figure 2.1. The detailed descriptions of the circuit elements are covered in Appendix A.

It should be noted that an N-type threshold function as the *up literal*, while a P-type threshold function as the *down literal*. A current source is a *constant*. *Sum* is not shown in the figure as it is realized by simply wiring signals together.

### 2.3.2 Circuit Realization of MVL Operators

A variety of circuit realizations of MVL operators using the basic CMCL circuit elements have been proposed, fabricated, and tested in the past [17, 55, 56, 57, 58]. Two examples are shown in Figure 2.2. Figure 2.2($a$) is a multiple-input *min* operator. Figure 2.2($b$) is a multiple-input *tsum* operator. Circuit implementation of other operators and their detailed description are given in Appendix B. It should be noted there are no standard symbols for the MVL operators.

## 2.4 Multiple-Valued Logic Design

MVL design varies dramatically with choice of logic operators. Depending upon a set of logic operators, the resulting circuit and the minimization procedure are very different. C. M. Allen et al. [5] used a set of operators consisting of *min*, *max* and *literal*. Z. G. Vranesic et al. [15] used a set operators consisting of *min*, *max* and *cycle*. Researchers have compared several sets of MVL operators in order to obtain the most efficient expression for a given MVL function. In [17], comparison among the following three sets of operators has been made for 2-variable 3-valued logic functions.

**Set A:** *literal*, *min*, and *tsum*.

**Set B:** *literal*, *complement of literal*, *min*, and *tsum*.

**Set C:** *literal*, *cycle*, *complement of literal*, *complement of cycle*, *min*, and *tsum*.

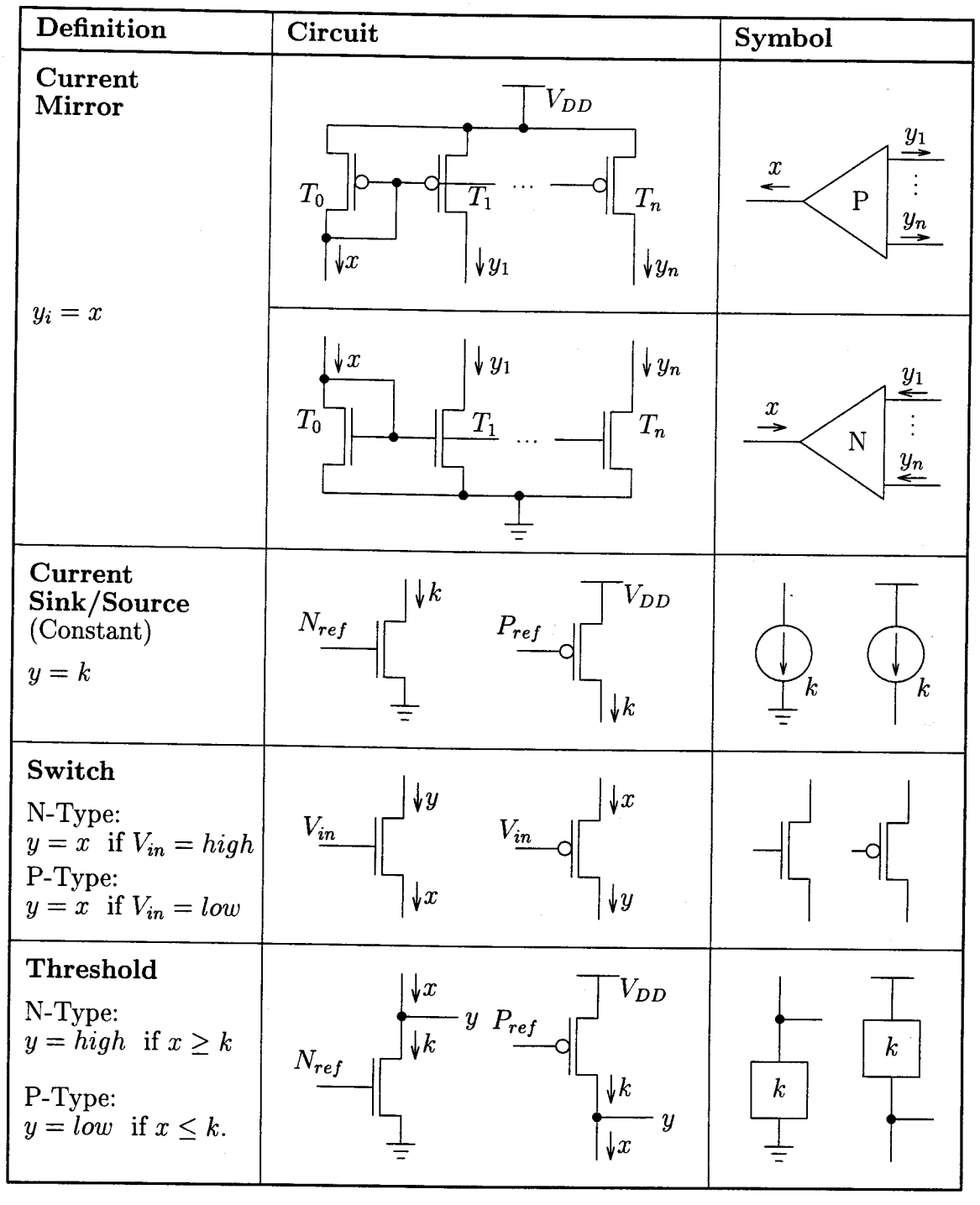
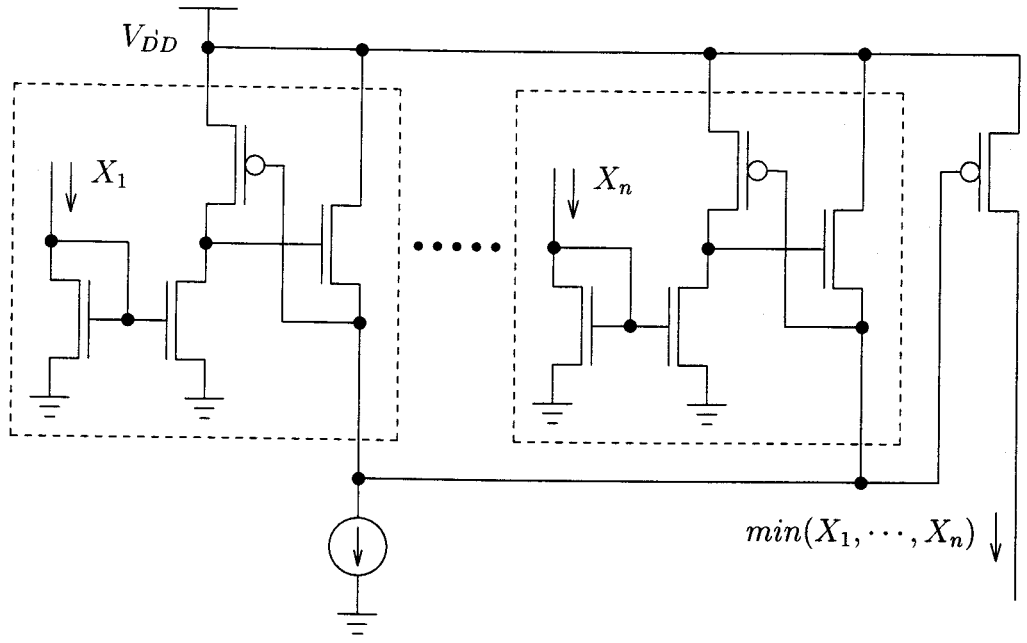
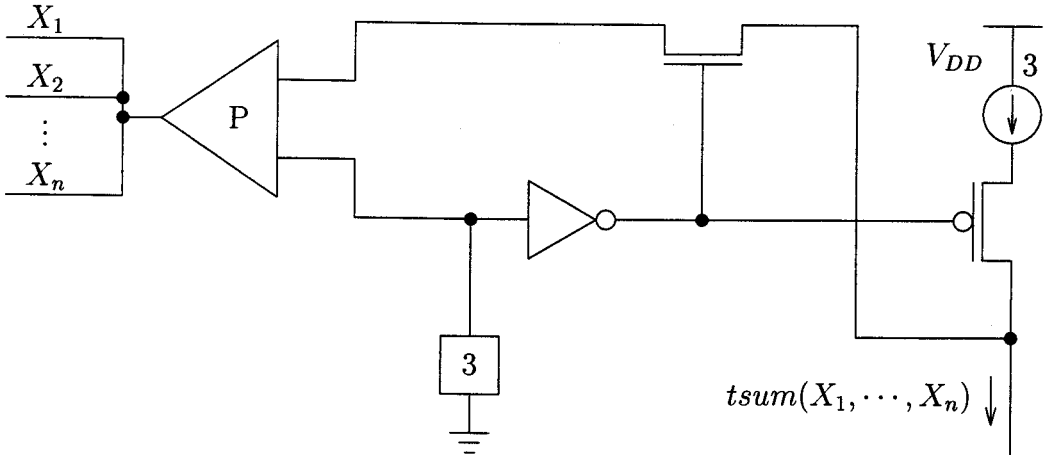



**Figure 2.1** CMCL circuit elements




(*a*)




(*b*)

**Figure 2.2** CMCL circuit realization of MVL operators. (*a*) Multiple-input *min* operator. (*b*) Multiple-input *tsum* operator

MVL design using different sets of operators can be more easily understood through examples. Figure 2.3($a$) shows an 2-variable 4-valued logic function minimized using Set A. 6 PTs are required for realizing this functions as shown below:

$$\begin{aligned} F \;=\; & (2 \bullet {}^{0}X_1{}^{0}) \bullet (2 \bullet {}^{0}X_0{}^{0}) \boxplus (2 \bullet {}^{2}X_1{}^{3}) \bullet (2 \bullet {}^{0}X_0{}^{0}) \boxplus \\ & (1 \bullet {}^{0}X_1{}^{0}) \bullet (1 \bullet {}^{2}X_0{}^{2}) \boxplus (1 \bullet {}^{2}X_1{}^{3}) \bullet (1 \bullet {}^{2}X_0{}^{2}) \boxplus \\ & (2 \bullet {}^{0}X_1{}^{0}) \bullet (2 \bullet {}^{3}X_0{}^{3}) \boxplus (2 \bullet {}^{2}X_1{}^{3}) \bullet (2 \bullet {}^{3}X_0{}^{3}) \end{aligned} \tag{2.24}$$

If Set B is used the same function can be expressed with 2 PTs as follows:

$$F = (2 \bullet \overline{{}^{1}X_1{}^{1}}) \bullet (2 \bullet \overline{{}^{1}X_0{}^{2}}) \boxplus (1 \bullet \overline{{}^{1}X_1{}^{1}}) \bullet (1 \bullet {}^{2}X_0{}^{2}) \tag{2.25}$$

As shown in Figure 2.3($b$), The function can be further minimized to a single PT using Set C. The truth table on the left is $2 \bullet \overline{{}^{1}X_1{}^{1}}$, the truth table on the right is $X_0{}^{\overset{3}{\rightarrow}}$, and the $\bullet$ between the two truth tables is the *min* operator; i.e., the single PT can be expressed as follows:

$$F = 2 \bullet \overline{{}^{1}X_1{}^{1}} \bullet X_0{}^{\overset{3}{\rightarrow}} \tag{2.26}$$

By comparing the PTs required for implementation of all $3^{3^2}$ functions, the results

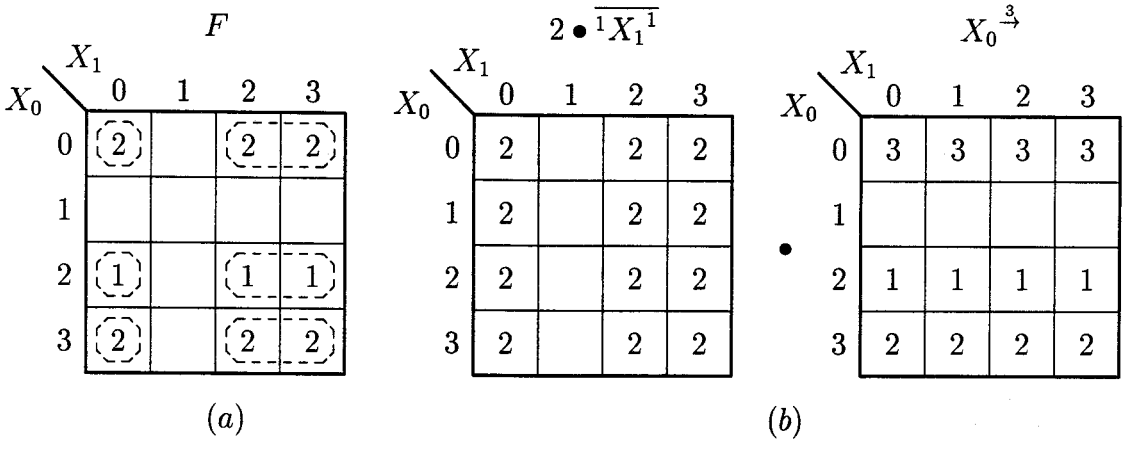



**Figure 2.3** MVL design using different sets of operators. ($a$) Set A. ($b$) Set C.

show that Set C provides a lower average and maximum number of PTs. However, the minimization is not obvious using Set C. As a result, heuristic techniques have been used to realize efficient circuits.

In [59] MVL functions are first expressed in a *sum-of-terms* (SOTs) form as:

$$F(X_1, X_2, \cdots, X_n) = T_1 \boxplus T_2 \boxplus \cdots \boxplus T_m \tag{2.27}$$

where $T_i$ is a *logic term.* For example, the 2-variable 4-valued functions shown in Figure 2.4 can be expressed as follows:

$$f(X_0, X_1) = 2({}^1X_1{}^1 \cup {}^1X_0{}^1) \boxplus 2({}^1X_1{}^2 \cup {}^1X_0{}^2) \tag{2.28}$$

The circuit realization of this equation is shown in Figure 2.5. This circuit includes both MVL circuit elements and MVL operators (i.e., *tsum*). However, a systematic approach to minimization was not mentioned in [59].

MVL functions can also be realized at the circuit level. In [13], Chang et al. proposed a hybrid mode CMOS MVL architecture. According to Chang's architecture, a given MVL function is expressed as an arithmetic sum of *min-of-down-literal* terms. For example, a 3-variable 4-valued logic function, $F = 2 \bullet {}^1X_1{}^2 \bullet {}^3X_1 \bullet X_2{}^1$, can be expressed as

$$F = 2 \bullet \overline{X_0{}^0} \bullet X_0{}^2 \bullet \overline{X_1{}^2} \bullet X_2{}^1. \tag{2.29}$$

Basically, Chang et al. use three subcircuits to implement a MVL function in this form. As shown in Figure 2.6, subcircuit 1 is a simple current-mirror; subcircuit 2 is a threshold comprised of a current source and a current comparator; and subcircuit 3 consists of a current mirror, two current sources and parallel-connected switches, where one current source is connected to the input of the current mirror through the switches and the other current source is directly connected to the output of the current mirror.

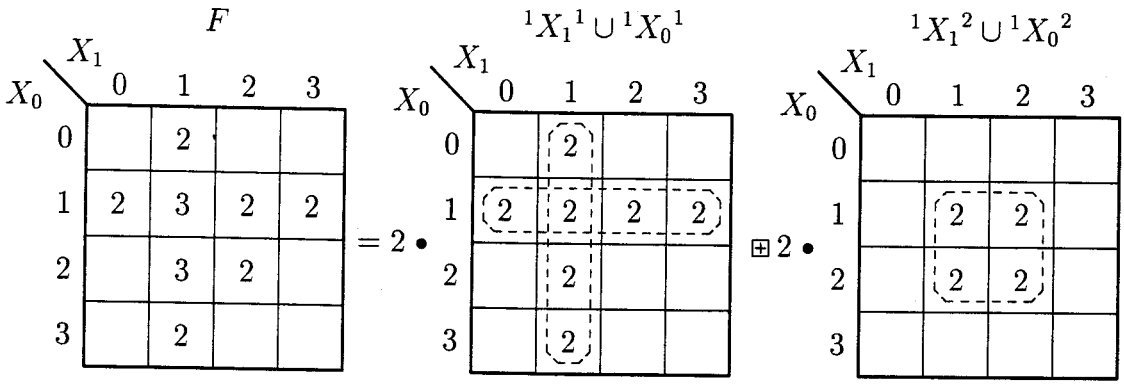



**Figure 2.4** A MVL function represented with two logic terms

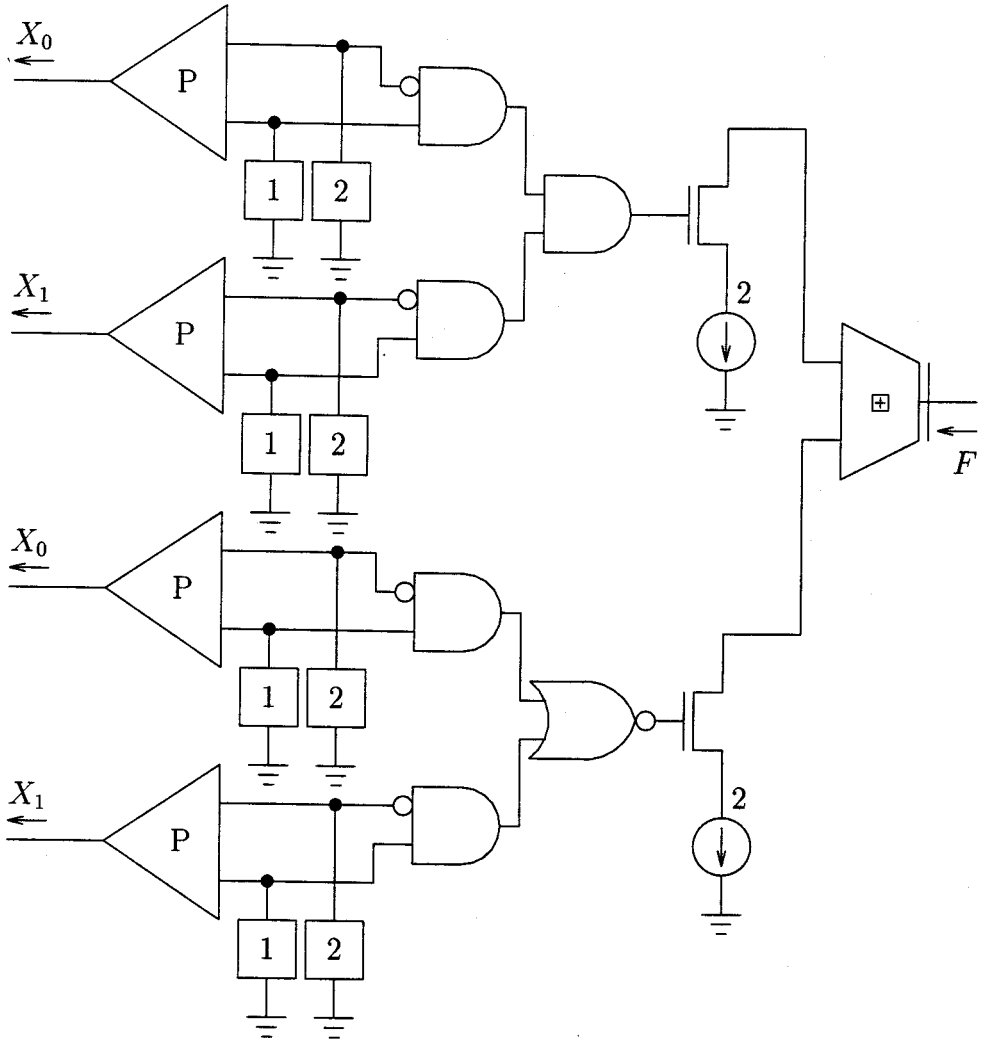



**Figure 2.5** Circuit realization of Eq. 2.28.

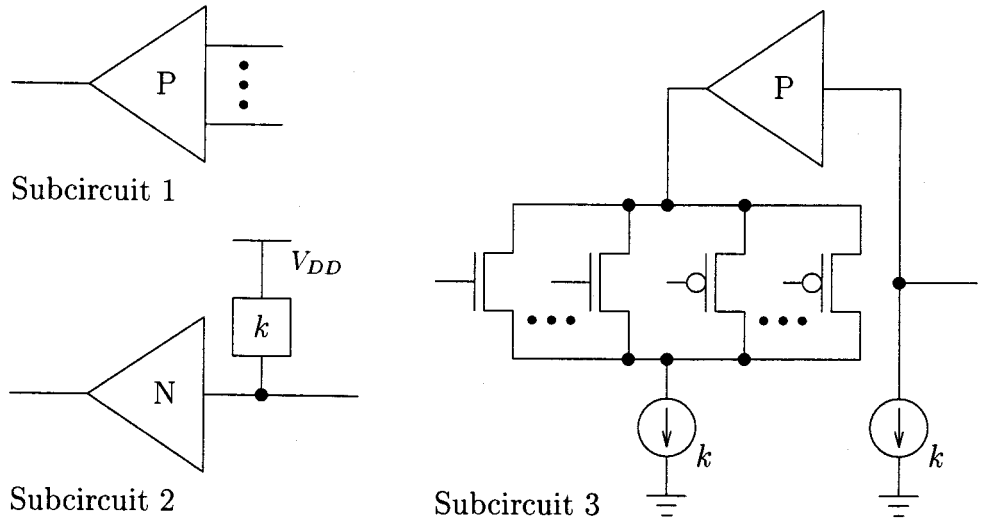



**Figure 2.6** Subcircuits for Chang's design scheme.

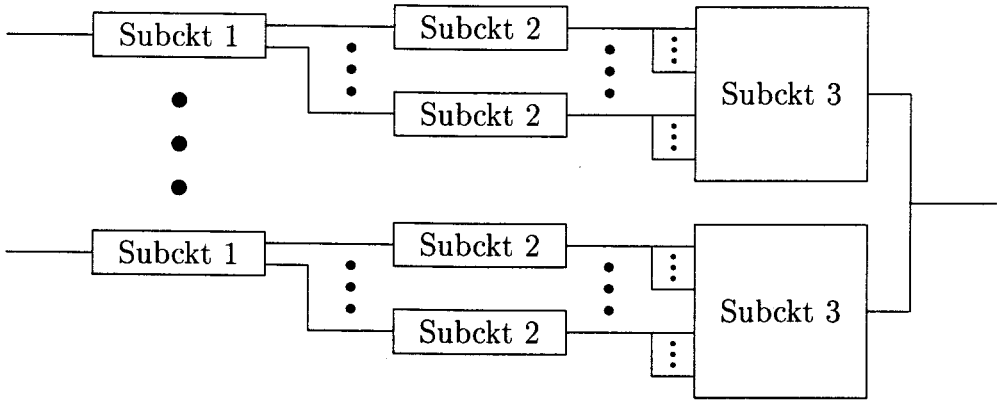



**Figure 2.7** General architecture for Chang's design scheme.

The general architecture of Chang's design scheme is shown in Figure 2.7.[1] To implement a MVL function, a current-mode MVL variable is duplicated by a subcircuit 1. The number of subcircuit 1 elements corresponds to the number of variables in a MVL function. The number of outputs of a subcircuit 1 is determined by how many different *literal* terms are related to that MVL variable. For example, there are two *literal* terms related to the variable $X_0$ in Eq. 2.29. The input $X_0$ should be connected to a subcircuit 1 with two outputs. It should be noted that *complement*

[1]It should be noted that Figure 2.7 is modified from the one in Chang's paper so as to more precisely match Chang's synthesis scheme.

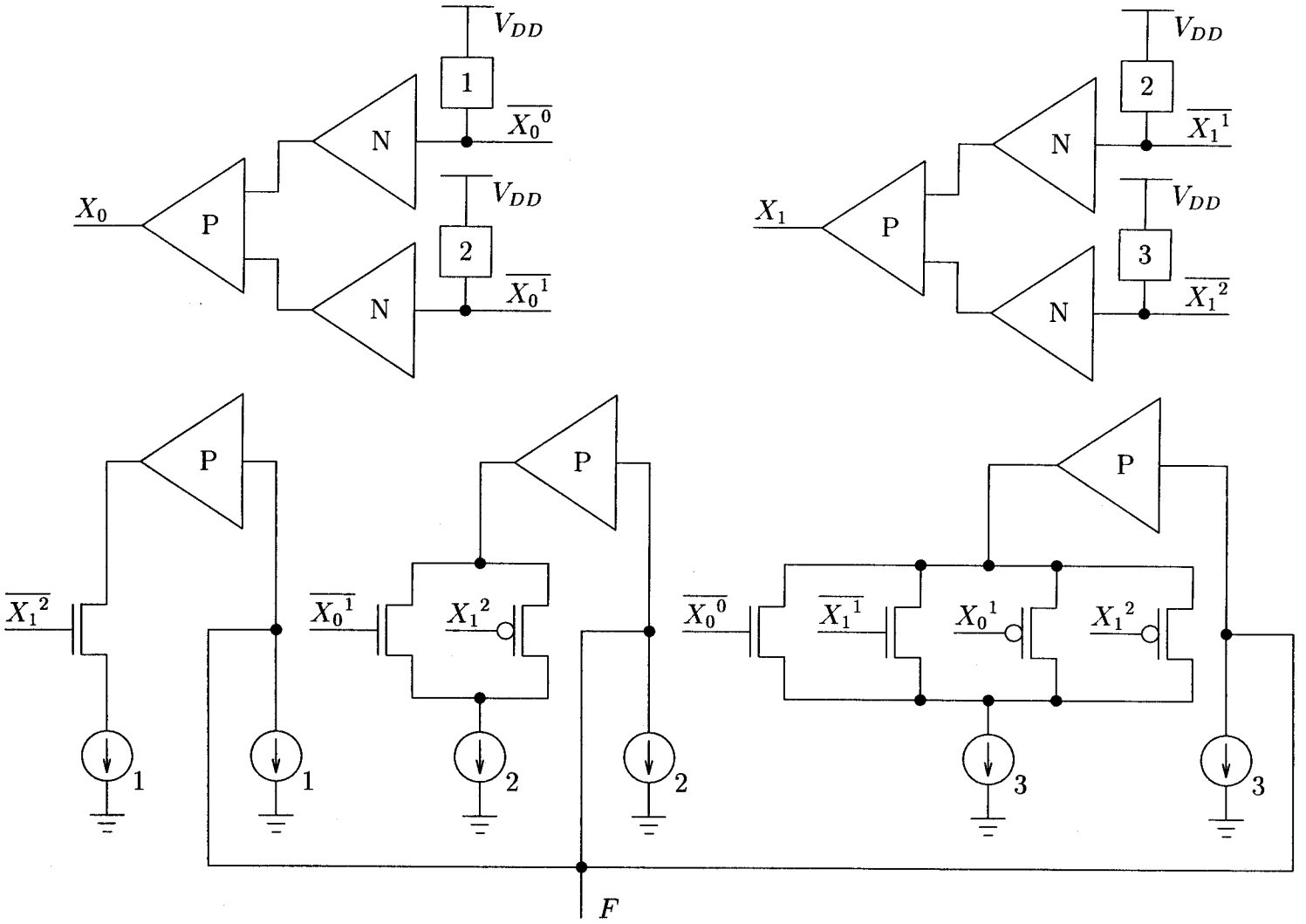



**Figure 2.8** Circuit realization of Eq. 2.31 according to Chang's synthesis scheme.

*of down literal* and *down literal* are considered as the same *literal* term here. Each output of a subcircuit 1 is connected with a subcircuit 2 input and converted into a voltage binary signal. That is,

$$V_i(X) = \begin{cases} \textit{binary high} & \text{if } X \geq i \\ \textit{binary low} & \text{otherwise.} \end{cases} \tag{2.30}$$

It should be noted that an output of a subcircuit 2 is actually a *complement of down literal.* Each voltage binary signal from a subcircuit 2 controls a switch in a subcircuit 3. Subcircuit 3 performs *min* operation on *literal* outputs from several subcircuit 2. Therefore, an output of a subcircuit 3 is a *min-of-down-literal.* The number of switches in a subcircuit 3 corresponds to the number of *down literal* terms in a *min-of-down-literal* term. For example, a subcircuit 3 for Eq. 2.29 has four switches. A N-type switch is used for a normal *literal* term, while a P-type switch is used for an inverted *literal* term. Outputs from subcircuit 3 are then wired together to perform arithmetic *sum* operation. The following equation is another example taken from Chang's paper [13].

$$F = 1 \bullet \overline{{X_1}^2} + 2 \bullet {X_1}^2 \bullet \overline{{X_0}^1} + 3 \bullet \overline{{X_1}^1} \bullet {X_1}^2 \bullet \overline{{X_0}^0} \bullet {X_0}^1 \tag{2.31}$$

The circuit realization of this function is shown in Figure 2.8.

## 2.5 Summary

A $r$-valued logic system allows $r^r$ unary operators and $r^{r^2}$ binary operators. Unlike Boolean algebra, MVL allows using multiple unary operators to implement a function effectively. Some of the definitions and their circuit realization are given in this Chapter. Other MVL operators and their current-mode CMOS circuits are described in Appendix B.

There are various approaches to implement MVL functions. MVL researchers have compared several different sets of operators for minimal number of PTs for a given set of MVL functions. Depending on the chosen set of operators, a MVL function is expressed with different number of PTs and the minimization algorithms are also different from each other.

# 3. Self-Restored Design Architecture - I

CMOS technologies have evolved dramatically over the past decades from $5\mu$m in the early 90s to $0.18\mu$m in 2000. According to the 2000 edition of International Technology Roadmap for Semiconductors (ITRS), $30n$m technology will be the mainstream process in 2014 [60]. The success of CMOS technology with binary logic design is underlied by an important fact that for all input assignments there is always a path from a voltage supply ($V_{DD}$ or $V_{SS}$) to the output and that the full supply voltages appear at the output. This feature leads to a *fully restored* or *self-restored* logic family. By taking advantage of the CMOS technologies, total gate account for binary logic design chips is breaking through 100 million.

However, it has been questionable that current-mode MVL design can be self-restored in a cost-effective way. This chapter addresses the self-restored techniques for current-mode CMOS MVL design. The theoretical basis will be described first, followed by explanation of the self-restored design architecture. The self-restored design architecture and one of its variants are described with examples. Other variants using arithmetic *sum* and *diff* operators are discussed in the next chapter.

## 3.1 Theoretical Analysis

A conventional CMOS binary logic design is self-restored as outputs are either connected to a voltage source through PMOS transistors or grounded through NMOS transistors as shown in Figure 3.1($a$). For example, Figure 3.1($b$) is a typical 2-input NAND gate. The output is connected to $V_{DD}$ through a PMOS transistor or grounded through two NMOS transistors. In view of this fact the block diagram of a

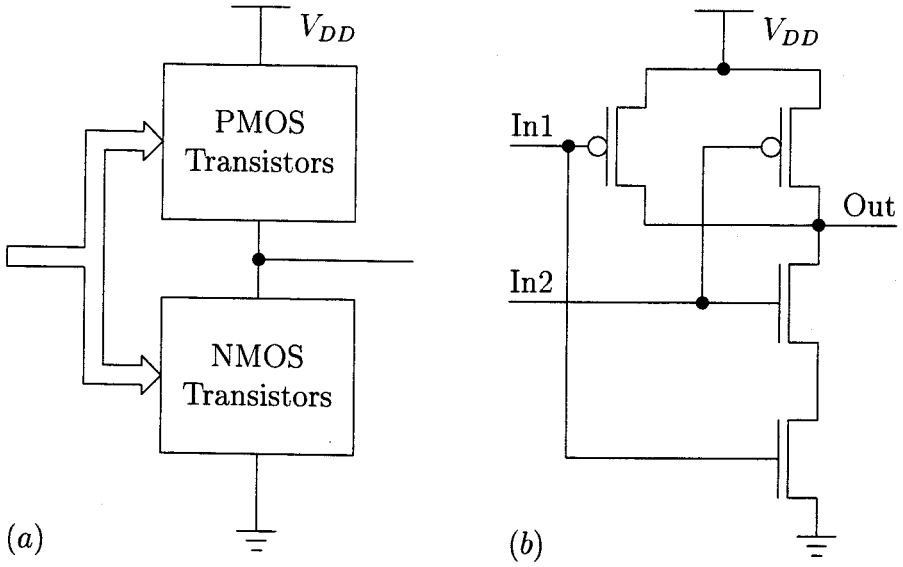



**Figure 3.1** (*a*) The conventional CMOS architecture for binary logic designs. (*b*) 2-input *nand* gate.

self-restored current-mode MVL design architecture should be similar to Figure 3.2 where output signals come directly from separate current sources through switches. Since current sources can be turned on/off through series-connected switches, the problem associated with a self-restored CMCL MVL design architecture becomes how to design a control circuit and how to arrange switches. It is well known that switches can be controlled using binary gates. Binary gates are preferable, as they

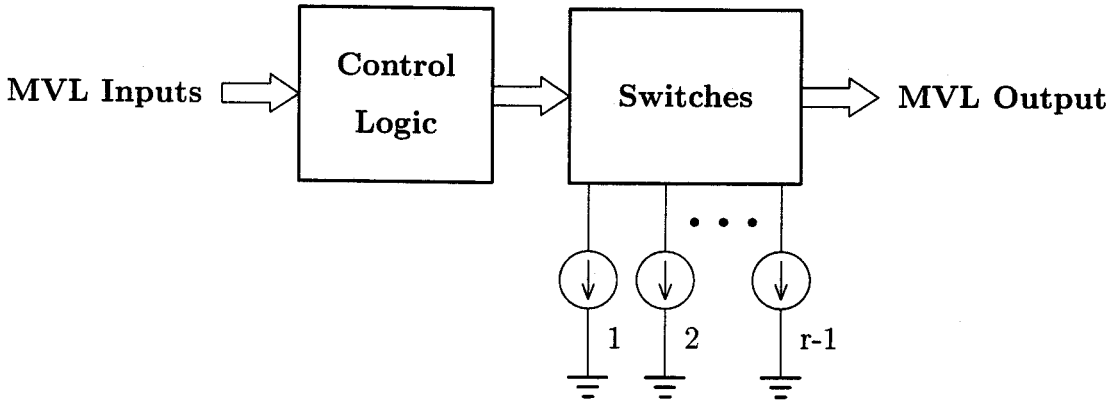



**Figure 3.2** A block diagram that shows the concept of a self-restored MVL architecture.

are self-restored and do not have adverse effects on output signal integrity. As well, binary design can be synthesized by using existing binary logic synthesizers such as Synopsys Design Compiler™. This further implies that less effort will be involved with the MVL synthesis. In order to make use of binary gates in the self-restored design architecture, current-mode MVL inputs signals must be converted to voltage-mode binary signals. It can be seen from the above preliminary analysis that a self-restored current-mode MVL design architecture should consist of three blocks: an input block for converting MVL signals to binary signals, a control block consisting of binary gates for controlling switches, and an output block consisting of switches and current sources where each current source connected with a switch. The fan-out problem can be solved by duplicating current sources in the output block. Figure 3.3 shows a general schematic. The operators such as *min*, *max* and *tsum* must not be included in the architecture because they adversely affect the output signals and increase circuit size drastically.

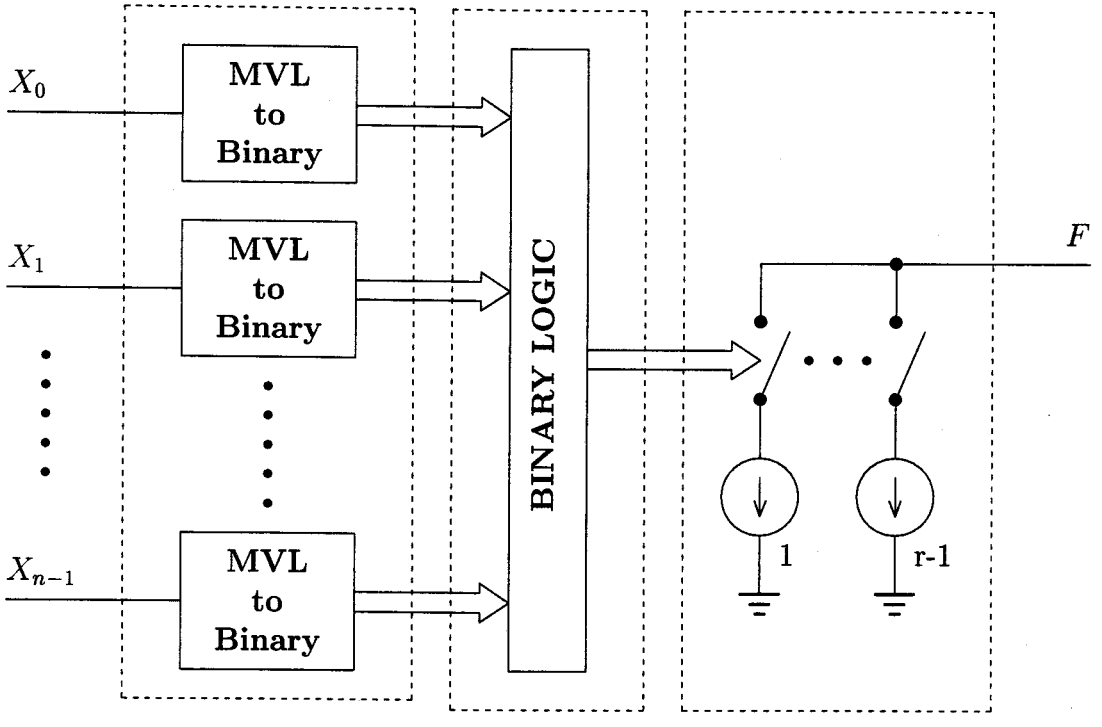



**Figure 3.3** The schematic diagram of a self-restored MVL architecture.

On the other hand, as stated by Lemma 2.1, any $n$-variable $r$-valued function can be expressed with the *max* operations on a maximum of $r^n$ number of distinct literal minterms. If the distinct literal minterms of a function, $F$, that result in the same logic value are grouped together, there are a maximum of $r-1$ groups. The minterms that result in logic 0 are not included since they are not to be implemented with circuits.

**Definition 3.1** For a $r$-valued function $F$ of $n$-variables, a subfunction, $F_j$, denoted as

$$F_j = \sum_k {}^+ \; j \bullet {}^{a_{jk_1}}X_1{}^{b_{jk_1}} \bullet {}^{a_{jk_2}}X_2{}^{b_{jk_2}} \bullet \cdots \bullet {}^{a_{jn_2}}X_n{}^{b_{jk_2}}, \tag{3.1}$$

is the MOPs of the distinct literal minterms of $F$ that result in only one logic value, $j$, of $F$, where $j, a_{jk_i}, b_{jk_i} \in \mathbf{R}$, $k \leq r^n$.

For example, referring to Figure 3.4, the 2-variable 4-valued function with its truth table shown in Figure 3.4($a$) can be expressed in terms of minterms in a MOP form as:

$$\begin{aligned} F \;=\; & 1 \bullet {}^0X_0{}^0 \bullet {}^1X_1{}^1 + 1 \bullet {}^0X_0{}^0 \bullet {}^2X_1{}^2 + 1 \bullet {}^1X_0{}^1 \bullet {}^1X_1{}^1 + 1 \bullet {}^1X_0{}^1 \bullet {}^2X_1{}^2 + \\ & 2 \bullet {}^2X_0{}^2 \bullet {}^2X_1{}^2 + 2 \bullet {}^2X_0{}^2 \bullet {}^3X_1{}^3 + 2 \bullet {}^3X_0{}^3 \bullet {}^2X_1{}^2 + 2 \bullet {}^3X_0{}^3 \bullet {}^3X_1{}^3 \end{aligned}$$

Therefore the subfunctions of $F$ are

$$\begin{aligned} F_1 \;&=\; 1 \bullet {}^0X_0{}^0 \bullet {}^1X_1{}^1 + 1 \bullet {}^0X_0{}^0 \bullet {}^2X_1{}^2 + 1 \bullet {}^1X_0{}^1 \bullet {}^1X_1{}^1 + 1 \bullet {}^1X_0{}^1 \bullet {}^2X_1{}^2 \\ F_2 \;&=\; 2 \bullet {}^2X_0{}^2 \bullet {}^2X_1{}^2 + 2 \bullet {}^2X_0{}^2 \bullet {}^3X_1{}^3 + 2 \bullet {}^3X_0{}^3 \bullet {}^2X_1{}^2 + 2 \bullet {}^3X_0{}^3 \bullet {}^3X_1{}^3 \\ F_3 \;&=\; 0 \end{aligned}$$

By Eqs. 2.15 and 2.16 as well as the axioms of of Chapter 2, subfunctions $F_1$ and $F_2$ can be minimized as:

$$F_1 \;=\; 1 \bullet {}^0X_0{}^0 \bullet {}^1X_1{}^1 + 1 \bullet {}^0X_0{}^0 \bullet {}^2X_1{}^2 + 1 \bullet {}^1X_0{}^1 \bullet {}^1X_1{}^1 + 1 \bullet {}^1X_0{}^1 \bullet {}^2X_1{}^2 +$$

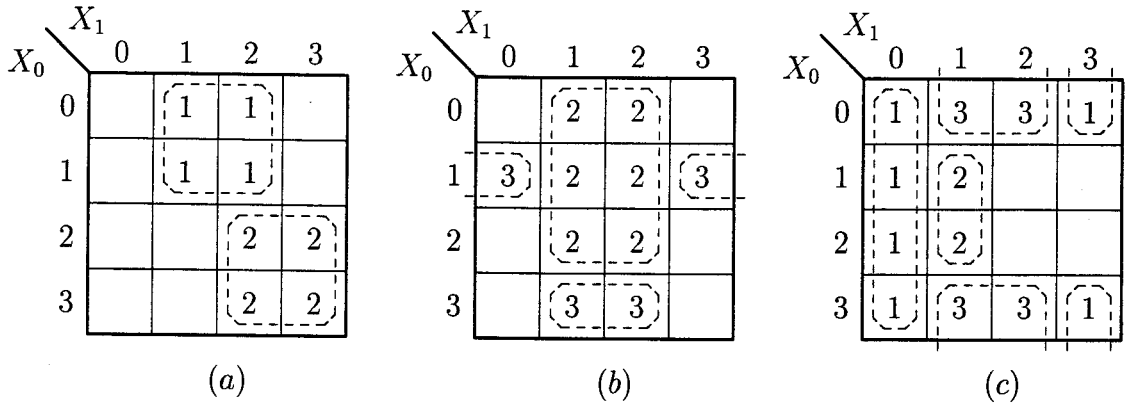



**Figure 3.4** Three examples of 2-variable 4-valued logic functions expressed in a MOP form.

$$
\begin{aligned}
&= 1 \bullet ({}^0X_0{}^0 \bullet ({}^1X_1{}^1 + {}^2X_1{}^2) + {}^1X_0{}^1 \bullet ({}^1X_1{}^1 + {}^2X_1{}^2)) \\
&= 1 \bullet ({}^0X_0{}^0 + {}^1X_0{}^1) \bullet ({}^1X_0{}^1 + {}^2X_0{}^2) \\
&= 1 \bullet {}^0X_0{}^1 \bullet {}^1X_1{}^2 \\
F_2 &= 2 \bullet {}^2X_0{}^2 \bullet {}^2X_1{}^2 + 2 \bullet {}^2X_0{}^2 \bullet {}^3X_1{}^3 + 2 \bullet {}^3X_0{}^3 \bullet {}^2X_1{}^2 + 2 \bullet {}^3X_0{}^3 \bullet {}^3X_1{}^3 \\
&= 2 \bullet ({}^2X_0{}^2 \bullet ({}^2X_1{}^2 + {}^3X_1{}^3) + {}^3X_0{}^3 \bullet ({}^2X_1{}^2 + {}^3X_1{}^3)) \\
&= 2 \bullet ({}^2X_0{}^2 + {}^3X_0{}^3) \bullet ({}^2X_1{}^2 + {}^3X_1{}^3) \\
&= 2 \bullet {}^2X_0{}^3 \bullet {}^2X_1{}^3
\end{aligned}
$$

Similarly, the subfunctions of the MVL truth table shown in Figure 3.4($b$) can be expressed as:

$$
\begin{aligned}
F_1 &= 0 \\
F_2 &= 2 \bullet {}^0X_0{}^2 \bullet {}^1X_1{}^2 \\
F_3 &= 3 \bullet {}^1X_0{}^1 \bullet \overline{{}^1X_1{}^2} + 3 \bullet {}^3X_0{}^3 \bullet {}^1X_1{}^2 ,
\end{aligned}
$$

and the subfunctions of the MVL truth table shown in Figure 3.4($c$) can be expressed as:

$$
F_1 = 1 \bullet {}^0X_1{}^0 + 1 \bullet \overline{{}^1X_0{}^2} \bullet {}^3X_1{}^3
$$

$$F_2 = 2 \bullet {}^1X_0{}^2 \bullet {}^1X_1{}^1$$
$$F_3 = 3 \bullet \overline{{}^1X_0{}^2} \bullet {}^1X_1{}^2 .$$

As can be seen from the examples, a benefit of using *literal* operations to represent a MVL function is that the MVL expression can be obtained from a truth table in a similar way to the minimization of Karnaugh maps. The basic concept is to combine the minterms according to Eqs. 2.15 and 2.16 for each subfunction.

Let $F_j = j \bullet V_j$; i.e.,

$$V_j = \sum_k \text{+} \; {}^{a_{jk_1}}X_1{}^{b_{jk_1}} \bullet {}^{a_{jk_2}}X_2{}^{b_{jk_2}} \bullet \cdots \bullet {}^{a_{jn_2}}X_n{}^{b_{jk_2}}, \tag{3.2}$$

By the definition of $F_j$ (*Definition 3.1*), a $r$-valued function $F$ can be decomposed into $r-1$ subfunctions as described below by Lemma 3.1.

**Lemma 3.1** Any $r$-valued functions can be represented with a maximum of $r-1$ subfunctions in the form:

$$\begin{aligned} F &= \sum_{j=1}^{r-1} \text{+} F_j = F_1 \text{+} F_2 \text{+} \cdots \text{+} F_{r-1} \\ &= \sum_{j=1}^{r-1} \text{+} j \bullet V_j = 1 \bullet V_1 \text{+} 2 \bullet V_2 \text{+} \cdots \text{+} (r-1) \bullet V_{r-1} \end{aligned} \tag{3.3}$$

As defined in Eq. 3.2, $V_j$ is a *max* of *literal* PTs. Every *literal* operation, ${}^{a_{jk_i}}X_i{}^{b_{jk_i}}$, is actually a multiple-valued input binary output mapping function (*Definition 2.3*), which is either 0 or $r-1$. Therefore, each literal PT is 0 or $r-1$ because the result of *min* operations on a series of literal PTs remains 0 or $r-1$ (*Definition 2.12*). $V_j$ also has only two output values, 0 or $r-1$ (*Definition 2.13*). In other words, $V_j$ can be considered as a binary function with $r-1$ as logic high and 0 as logic low. The *min* operation on $j$ and $V_j$ doesn't affect the binary feature of $V_j$. A subfunction $F_j = j \bullet V_j$ is still a binary function except that the value of binary high changes from

$r-1$ to $j$. Therefore, Eq. 3.3 indicates that a $r$-valued logic function is actually a *max* operation on $r-1$ binary logic subfunctions each having an unique logic high value. In other words, any $r$-valued logic function, $F$, of $n$ variables can be decomposed into a *max* of $r-1$ binary subfunctions. Referring to Fig. 3.4($a$), a 2-variable 4-valued logic function $F$ is decomposed into $1 \bullet V_1 + 2 \bullet V_2$ as shown in Figure 3.5, where

$$V_1 = {}^0X_0{}^1 \bullet {}^1X_1{}^2 \quad \text{and} \quad V_2 = {}^2X_0{}^3 \bullet {}^2X_1{}^3$$

$V_3$ does not appear in the figure because it is always 0.

When hardware implementation is considered, Eqs. 3.2 and 3.3 are not the best representations of a MVL function because they don't result in the most economical circuit design. For example, the CMOS implementation of a single *literal* needs on average 13 transistors [12]. Multi-input *min* and *max* circuits are not self-restored - nor are they small. Depending on the circuit implementation, the transistor counts of *min* and *max* operators are 9 to 13 [12]. Eq. 3.3 must be rewritten into other forms for smaller circuit realization.

**Lemma 3.2** The $V_j$ subfunctions of a MVL function are disjoint.

**Proof:** Assume $F_a = a \bullet V_a$ and $F_b = b \bullet V_b$ are two subfunctions of a function $F$. Since $F_a$ and $F_b$ consists of the distinct literal minterms of a function $F$, there are no

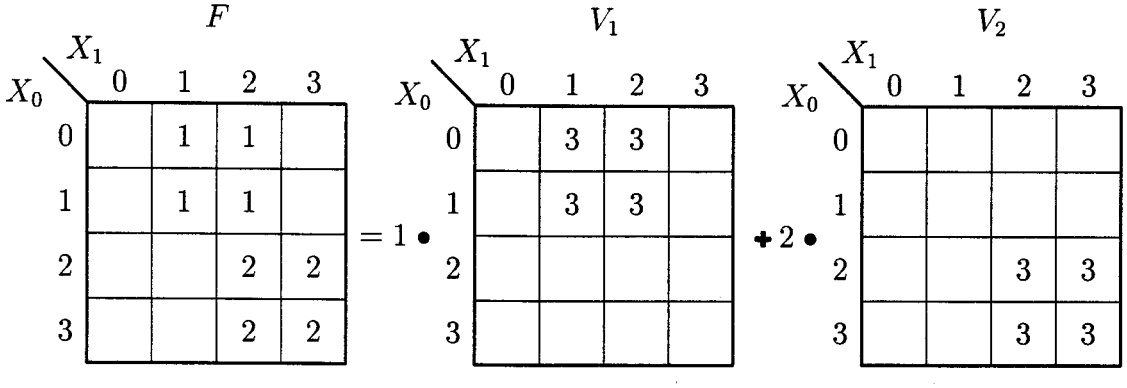



**Figure 3.5** Decomposition of a 2-variable 4-valued logic function into two binary logic functions.

equivalent minterms in $F_a$ and $F_b$. Accordingly, subfunctions $F_a$ and $F_b$ are disjoint and therefore $V_a$ and $V_b$ are disjoint as well.

In order words, a $r$-valued function $F$ can be decomposed to $r-1$ disjoint binary subfunctions. This leads to the Theorem 3.1 by Lemma 2.2.

**Theorem 3.1** Any MVL functions can be represented with an arithmetic sum of binary subfunctions, $V_j$, in the form:

$$F = 1 \bullet V_1 + 2 \bullet V_2 + \cdots + (r-1) \bullet V_{r-1} \tag{3.4}$$

That is, the *max* operations are replaced with algebraic sum operations. As described in Chapter 2, algebraic sums are preferred in current-mode design because it can be realized by simply wiring signals together (Kirchhoff's Current Law). Referring to Figure 3.3, the $V_j$ subfunctions can be implemented in the binary logic block since they are binary functions, and their outputs can then be used for controlling the current sources at the output end. The *literals* correspond to the MVL-to-binary blocks. The binary logic block consists of VMCL binary circuit. This implies that a binary logic synthesizer can be applied to MVL synthesis. The question now is how to realize a current-input voltage-output *literals* circuit and how to represent $V_j$ subfunctions in binary logic expressions.

## 3.2 Design Architecture

From Lemma 2.4 we know that ${}^aX^b = {}^aX \bullet X^b$. Therefore, Eq. 3.2 becomes

$$V_j = \sum_k \blacklozenge \; {}^{a_{jk_1}}X_1 \bullet {X_1}^{b_{jk_1}} \bullet {}^{a_{jk_2}}X_2 \bullet {X_2}^{b_{jk_2}} \bullet \cdots \bullet {}^{a_{jk_n}}X_n \bullet {X_n}^{b_{jk_n}} \tag{3.5}$$

In Eq. 3.5, ${}^{a_{jk_i}}X_i$ is a *(up literal)* ${X_i}^{b_{jk_i}}$ is a *(down literal)*. As mentioned in Chapter 2, the *up literal* and *down literal* operators are *threshold literals*. From a circuit design

point of view, they are threshold circuits. The CMOS *threshold literal* can be realized with only one transistor. The *up literal* is an N-type threshold, while the *down literal* is a P-type threshold.

The voltage output of a threshold is a binary signal. If threshold output signals are used directly as inputs for implementing a $V_j$ subfunction, the $V_j$ subfunction can be realized by a conventional voltage-mode binary circuit. By doing so, all of the *min* and *max* operations within a $V_j$ subfunction can be replaced by binary *and* and *or* operations. This means reduction of circuit size. The implementation of $j \bullet V_j$ does not need a *min* circuit either. The *min* operation on $j$ and $V_j$ can be realized by connecting a switch to a logic $j$ current source, where the switch is controlled by the binary circuit of $V_j$.

Since $X^{k-1} = \overline{{}^{k}X}$ by Lemma 2.5, Eq. 3.5 can be expressed in terms of *up literals* and *complement of up literals* only.

$$V_j = \sum_k {}^{+} \; {}^{a_{jk_1}}X_1 \bullet \overline{{}^{q_{jk_1}}X_1} \bullet {}^{a_{jk_2}}X_2 \bullet \overline{{}^{q_{jk_2}}X_2} \bullet \cdots \bullet {}^{a_{jk_n}}X_n \bullet \overline{{}^{q_{jk_n}}X_n} \qquad (3.6)$$

where $q_{jk_i} = b_{jk_i} + 1$ if $b_{jk_1} < r-1$ or $q_{jk_i} = r-1$ if $b_{jk_1} = r-1$ and can be omitted from a PT. The *up literal* should be used when applicable because an N-type threshold is faster than a P-type threshold [12]. The *complement of up literal* can be implemented by an N-type threshold with an inverter connected to its output. Two examples are given below. The 4-valued logic function shown in Figure 3.6($a$) can be expressed by

$$F = 1 \bullet 0 + 2 \bullet 0 + 3 \bullet V_3 = 3 \bullet {}^{1}X,$$

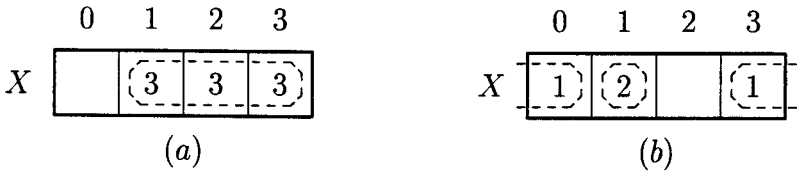



**Figure 3.6** Derivation of $V_j$ subfunctions from truth tables using the *up literal* operator.

and the 4-valued logic function shown in Figure 3.6($b$) can be expressed by

$$F = 1 \bullet V_1 + 2 \bullet V_2 + 3 \bullet 0 = 1 \bullet (\overline{{}^1X} + {}^3X) + 2 \bullet {}^1X \bullet \overline{{}^2X}.$$

Based on the above analysis, a general architecture of self-restored current-mode MVL designs is obtained as shown in Fig. 3.7. The design architecture consists of three blocks: an input block, a control block and an output block. The input block implements *up literal* operations, ${}^{a_{jk_i}}X_i$ and $\overline{{}^{q_{jk_i}}X_i}$, in each $V_j$ subfunctions by using current mirrors and thresholds. Since current-mode MVL signals are automatically converted to voltage-mode binary signals when ${}^{a_{jk_i}}X_i$ and $\overline{{}^{q_{jk_i}}X_i}$ are implemented by thresholds, additional MVL-binary conversion circuits are not required to interface with binary logic circuits. The control block is a voltage-mode binary design for realization of the binary counterpart of $V_j$, denoted as $v_j$. The output block is comprised of switches and current sources. Each switch is connected with a current

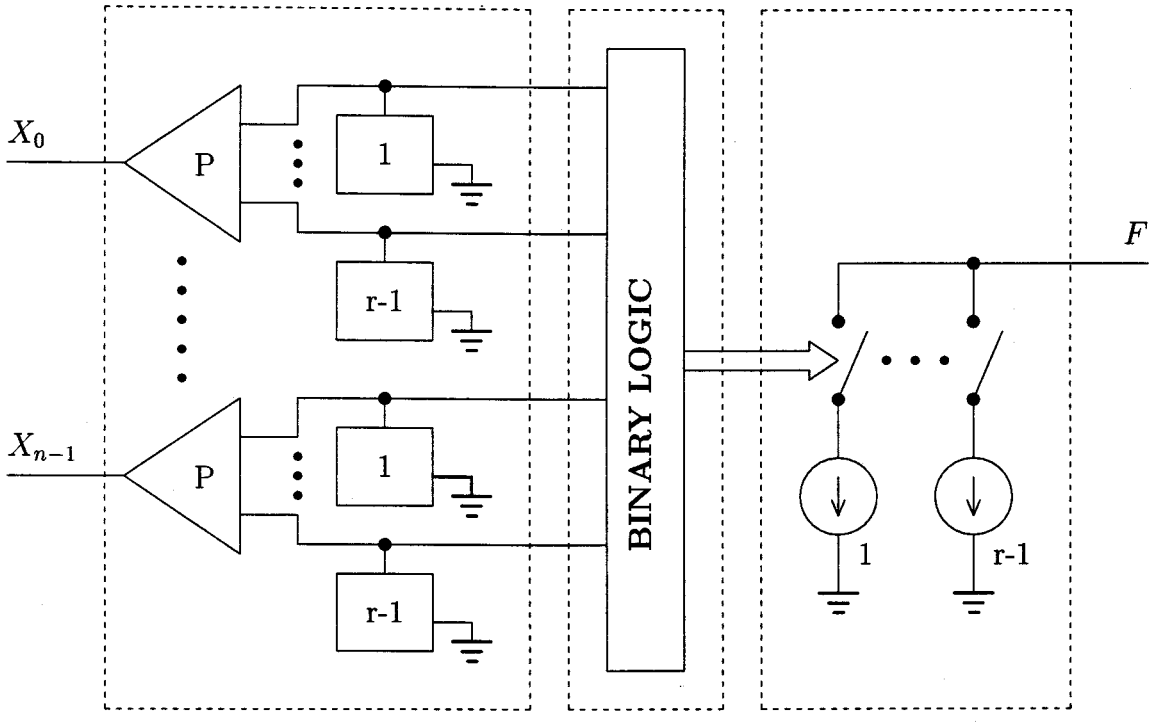



**Figure 3.7** A general self-restored MVL design architecture.

source. The binary output signals from the control block control switches in the output block. Therefore, each output of a current source corresponds to a $F_j = j \bullet V_j$. The outputs of the current sources are wired together to obtain an algebraic sum. Since $F_j$ subfunctions are disjoint, only one switch is on at a time. It should be noted that not all types of thresholds, switches and current sources are required for a MVL function. In other words, the total number of the threshold is always less than or equal to $(n-1) \times (r-1)$ while the number of switches and current sources should always be equal to or less than $(r-1)$.

As described in the previous sections, a MVL function expressed in terms of *literal* operators can be obtained from its truth table in a very similar way to the minimization of Karnaugh maps. The procedure is easy to those skilled in binary logic design. Let $x_{i,j}$ represent a binary signal from a logic $j$ threshold of variable $X_i$ and $x_{i,j}$ is written as $x_j$ for a single-variable MVL function; that is,

$$x_{i,j} = \begin{cases} \textit{binary high} & \text{when } X_i \geq j \\ \textit{binary low} & \text{otherwise.} \end{cases} \tag{3.7}$$

Then the binary function to be implemented by the control block for the MVL function in Figure 3.6($a$) is expressed as

$$v_3 = x_{0,1} = x_1 \ ,$$

and the binary functions to be implemented by the control block for the MVL function in Figure 3.6($b$) are expressed as

$$v_1 = \bar{x}_1 + x_3 \quad \text{and} \quad v_2 = x_1 \cdot \bar{x}_2,$$

where '+' represents the binary logic *or* operator and '$\cdot$' represents the binary logic *and* operator. Fig. 3.8 is the circuit realization of Fig. 3.6($b$). The input block includes a 3-output current mirror and three thresholds to detect logic 1, 2, and

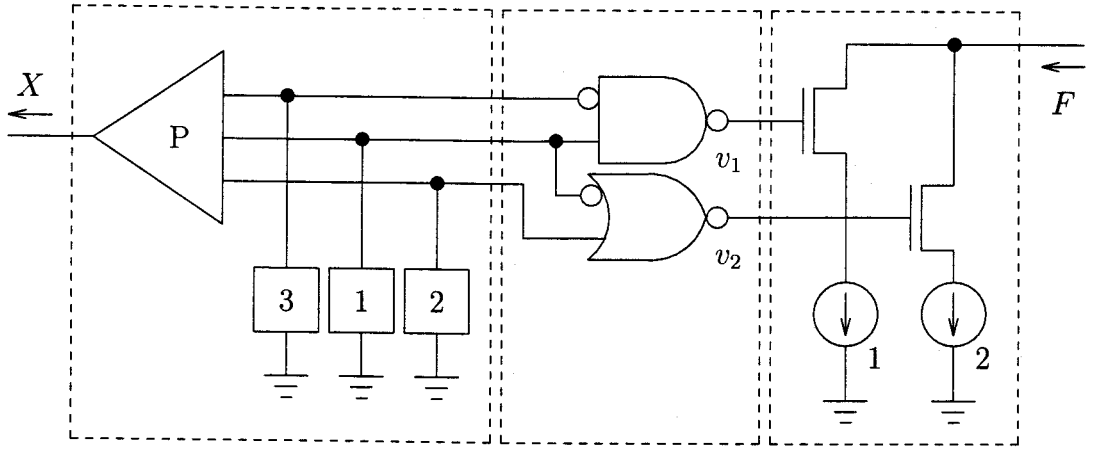



**Figure 3.8** MVL circuit realization of Figure 3.6($b$) according to the self-restored design architecture.

3, respectively. The output block includes two switches. The switch controlled by signals $v_1$ is connected with a logic 1 current source, while the switch controlled by $v_2$ is connected with a logic 2 current source. If the base current (i.e., difference between two adjacent logic levels) is $20\mu$A, then the logic 1 current source generates $20\mu$A and the logic 2 current source generates $2 \times 20\mu\text{A} = 40\mu\text{A}$. More examples are given in Section 3.4.

## 3.3 Variants of the Architecture

The self-restored architecture can be modified in various ways. For example, the control block can be implemented with CMOS structure as shown in Figure 3.1 instead of using binary gates. This section presents a simple variant. More variants will be presented in Chapter 4 where an approach of applying arithmetic operators to the design architecture is discussed.

The self-restored architecture can be modified to merge the control block into the output block by constructing an NMOS switch network connected to current sources. This design architecture then consists of two blocks: an input block for converting

MVL signals to binary signals and a control/output block of current sources and switches where each current source is connected with an NMOS switch network. In other words, all binary gates in Figure 3.7 are merged into the control/output block. The design of an NMOS switch network is very similar to the design of a CMOS binary gate at the transistor level. The difference is that only the NMOS configuration is of interest. The general design architecture based on this idea is shown in Figure 3.9. It should be noted that some inverters and current sources might not be used when implementing a given MVL function.

Circuit implementation of MVL functions according to this architecture follows the same idea described in Sections 3.1 and 3.2. For example, for the MVL function shown in Figure 3.6($b$), the same $v_j$ subfunctions, i.e., $v_1 = \bar{x}_1 + x_3$ and $v_2 = x_1 \cdot \bar{x}_2$, are needed to be realized with the NMOS transistors. Figure 3.10 is a circuit realization of Figure 3.6($b$) based upon this variant of the self-restored architecture. Assume a simple current mirror is used. The circuit shown in Figure 3.8 consists of 23 transistors and the circuit shown in Figure 3.10 consists of 17 transistors. The circuit implementation of MVL functions based upon the variant is smaller. However, it should be noted that the variant is not good for complex MVL functions such as multiple-input functions because series-connected NMOS transistors could deteriorate output signals from current sources. Also, a new set of library cells for the output block must be developed. The new library cells should include various combinations of NMOS switch network and current sources.

## 3.4 Examples

One of the advantages of the proposed architecture is that the minimization procedure is very easy to understand for those who are skilled in binary logic design. The minimization procedure will be more clear with reference to the following examples. Three examples will be presented in this section. Example 3.1 implements the truth

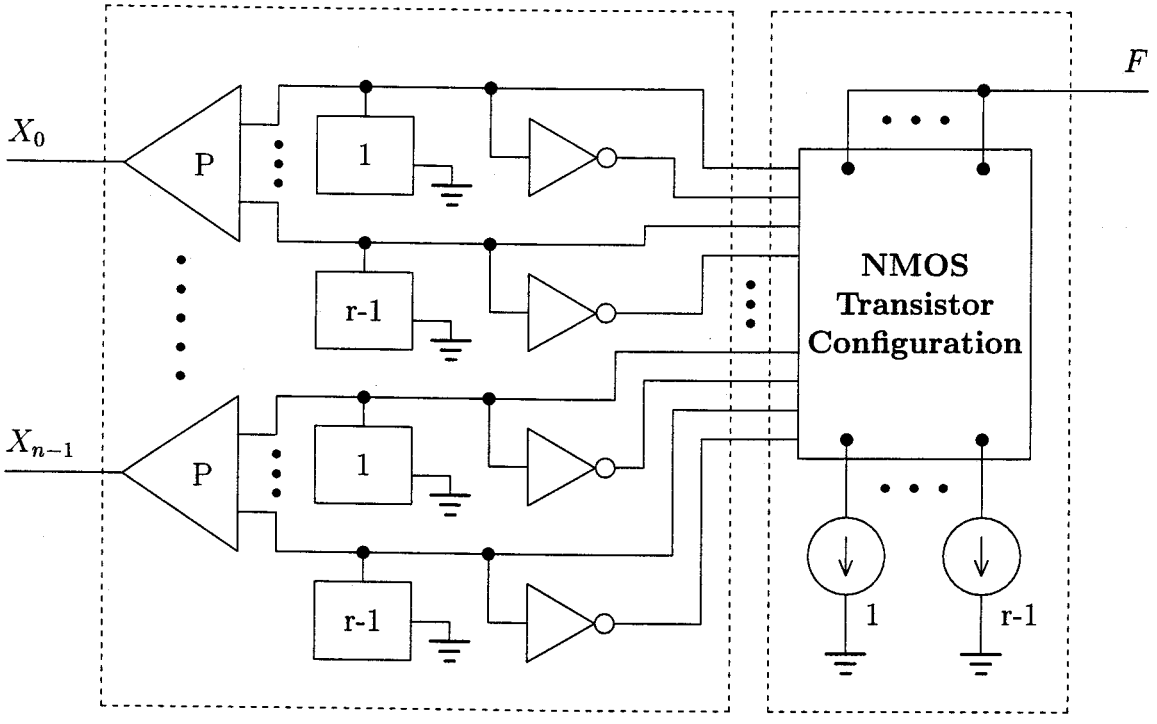



**Figure 3.9** An variant of the self-restored MVL design architecture.

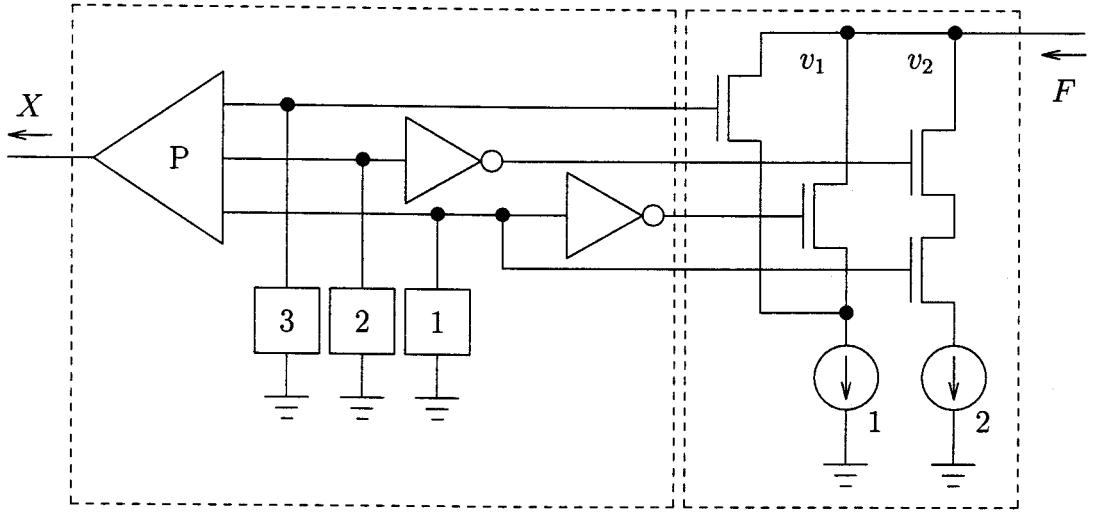



**Figure 3.10** MVL circuit realization of Figure 3.6($b$) according to the variant of the design architecture.

table shown in Figure 3.11($a$) which is taken from Chang's paper [13]. Example 3.2 implements the truth table shown in Figure 3.11($b$). Example 3.3 implements the truth table shown in Figure 3.11($c$).

**Example 3.1** First of all, consider the implementation of the truth table shown in Figure 3.11($a$). $V_j$ subfunctions can be obtained from the three implicants in the figure:

$$V_1 = {}^3X_1, \quad V_2 = {}^2X_0 \bullet \overline{{}^3X_1} \quad \text{and} \quad V_3 = {}^1X_0 \bullet \overline{{}^2X_0} \bullet {}^2X_1 \bullet \overline{{}^3X_1}. \tag{3.8}$$

Binary $v_j$ subfunctions for the control block can be obtained from Eq. 3.8:

$$v_1 = x_3, \quad v_2 = x_{0,2}\bar{x}_{1,3} \quad \text{and} \quad v_3 = x_{0,1}\bar{x}_{0,2}x_{1,2}\bar{x}_{1,3}. \tag{3.9}$$

The circuit implementation of these three expressions are shown in Figure 3.12. The total number of transistors is 34. This number is lower than Chang's implementation (37) as shown in Figure 2.8 on page 28.

**Example 3.2** In this example, the truth table shown in Figure 3.11($b$) is implemented using the NMOS structure according to the self-restored architecture of Fig-

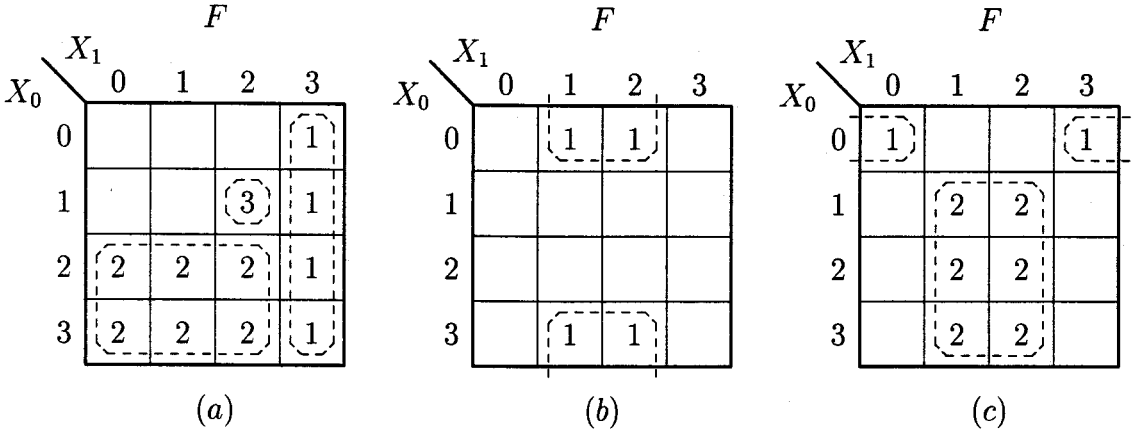



**Figure 3.11** Truth table of three examples.

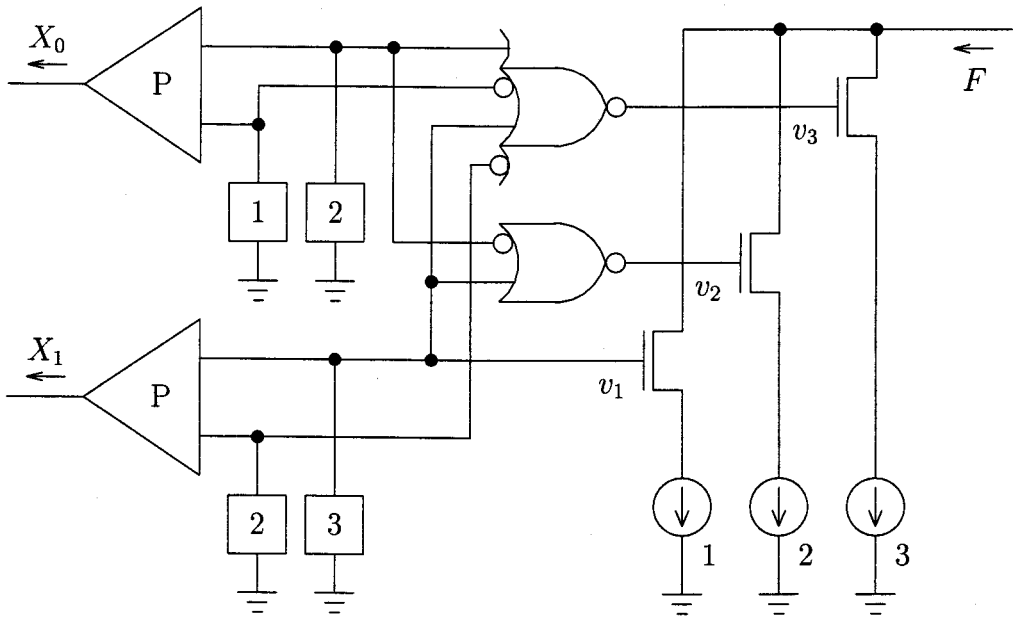



**Figure 3.12** Circuit realization of Example 3.1.

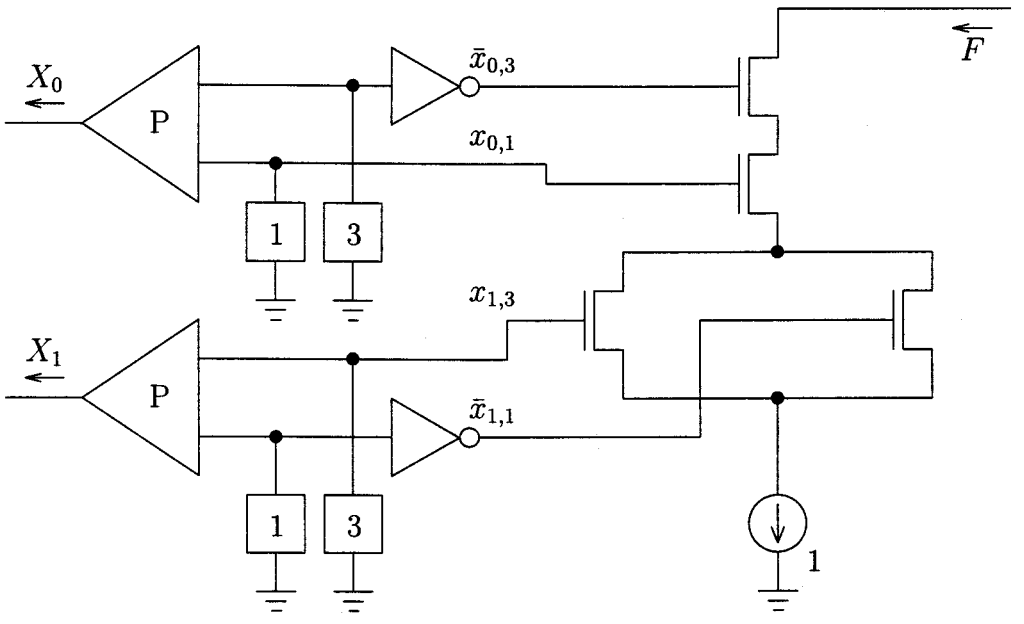



**Figure 3.13** Circuit realization of Example 3.2.

ure 3.9. First of all, the corresponding $v_j$ subfunction of the control block can be expressed as

$$v_1 = x_{1,1}\bar{x}_{1,3}(\bar{x}_{0,1} + x_{0,3}) = \overline{\bar{x}_{1,1} + x_{1,3} + x_{0,1}\bar{x}_{0,3}}. \tag{3.10}$$

The circuit realization of this expression is shown in Figure 3.13. The total transistor count is 19.

**Example 3.3** Another example is shown in Figure 3.11($c$). Again, the control block can be expressed with two binary functions, $v_1$ and $v_2$, where

$$v_1 = \bar{x}_{0,1}(\bar{x}_{1,1} + x_{1,3}) = \overline{x_{0,1} + \overline{(\bar{x}_{1,1} + x_{1,3})}}, \text{ and} \tag{3.11}$$

$$v_2 = x_{0,1}x_{1,1}\bar{x}_{1,3} = x_{0,1}\overline{(\bar{x}_{1,1} + x_{1,3})}. \tag{3.12}$$

The circuit implementation of these two expressions are shown in Figure 3.14. The total transistor count is 28.

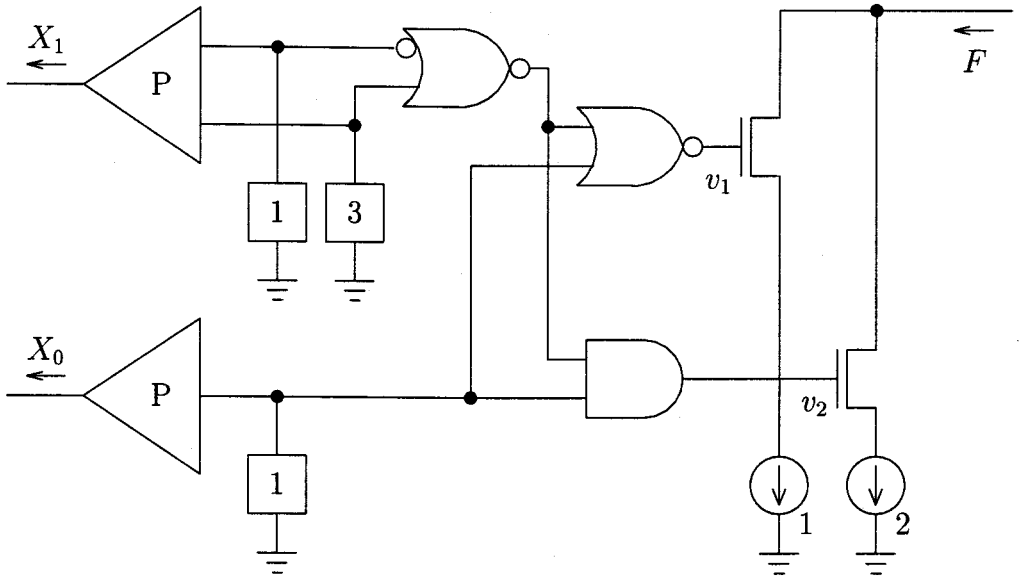



**Figure 3.14** Circuit realization of Example 3.3.

## 3.5 Summary

Through the above theoretical analysis and examples, it can be seen any $r$-valued logic function can be represented with $r-1$ binary $V_j$ subfunctions. Once a MVL function is decomposed into this form, it can be realized according to the proposed self-restored design architecture. The design architecture consists of three blocks, an input block, a control block and an output block. The most important part is the control block as it realizes the binary functions and the binary functions contain all the information of a MVL function. Since the control block is a binary circuit, it can be minimized using a binary logic synthesizer as will be described in Chapter 5.

The self-restored architecture is characterized by implementing MVL functions using those logic operators that correspond to current-mode circuit elements together with voltage-mode binary gates. Using voltage-mode binary gates to control output switches, output signals are restored simultaneously. N-type thresholds and binary gates are used within the design architecture so that extra MVL-binary or binary-MVL conversion circuits are not required to interface with binary circuits. Three examples are presented to illustrate MVL design using this architecture. The minimization procedure is easy to understand to those who are already skilled in binary logic design.

# 4. Self-Restored Design Architecture - II

A self-restored current-mode CMOS Multiple-Valued Logic (MVL) design architecture is proposed in Chapter 3, which consists of three blocks: an input block, a control block and an output block. The theoretical analysis of Section 3.1 also indicates that any $r$-valued logic functions can be realized with this design architecture. The self-restored architecture is advantageous over conventional operator-based MVL design schemes in many aspects in addition to self-restoration. With the self-restored architecture, logic minimization of MVL functions can take advantage of existing binary logic synthesizers. Also, *up literals* and binary logic gates that serve as part of the function implementation allow a design to interface with binary logic design without using extra MVL-binary encoders.

Similar to a binary logic design, a MVL function can be implemented in more than one way. Circuit size varies with different implementations. It is well known that the circuit size of a current-mode design can be reduced by using the arithmetic *sum* and *diff* operators because they are realized through simply wiring signal lines together according to Kirchhoff's Current Law. In other words, implementation of the *sum* and *diff* operations virtually does not need any transistors. Therefore, by using the *sum* and *diff* operators in self-restored MVL designs, a number of transistors should be saved. This chapter describes how these two arithmetic operators can be used in a self-restored MVL design and examines the size of resulting circuits. Section 4.1 focuses on the use of the *sum* operator in an output block along with a minimization approach using different types of output blocks. With the *sum* operator in an output block, minimization of a MVL function becomes more complicated since selection of the output block configuration will affect the control block. Proper minimization in

conjunction with appropriate choice of an output block results in a smaller circuit. Section 4.2 discusses the uses of the *sum* and the *diff* in an input block. Combinational use of the two operators in an input block as well as in both an input block and an output block at the same time is also described in this section. A summary is given in Section 4.3.

## 4.1 Use of the Sum Operator in an Output Block

Using the *sum* operator in an output block not only reduces the circuit size but also allows the use of other types of output blocks. According to the minimization approach for the self-restored MVL architecture proposed in Chapter 3, both the input block and the output block are fixed for a given function. For a 4-valued logic function, an output block must be either one or a combination of the subcircuits shown in Figure 4.1. Each subcircuit consists of a current source and a switch. For example, subcircuit 1 has a logic 1 current source. The current source is controlled by a binary signal $v_1$ through a switch $SW_1$. If the base current (the difference between two adjacent logic levels) is set to $20\mu$A, logic 1 current source generates $20\mu$A, logic 2 current source generates $40\mu$A, and so forth. If a MVL function has logic 1 output

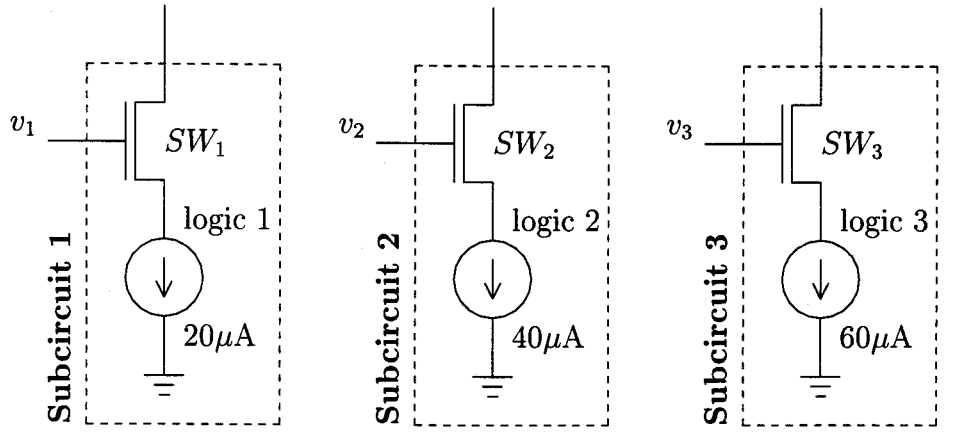



**Figure 4.1** Three output subcircuits of the self-restored 4-valued architecture.

only, the corresponding output block consists of a subcircuit 1. If a MVL function has both logic 2 and logic 3 outputs, then the corresponding output block consists of subcircuits 2 and 3. For simplification a single subcircuit output block, such as subcircuit 2, is denoted as {2}. An output block with two subcircuits, for example subcircuit 1 and subcircuit 3, is denoted as {1,3}. Two rules are followed by the self-restored architecture and its variants in Chapter 3; first it uses one subcircuit $i$ to implement all logic $i$ entries in a truth table, and the other is that only one subcircuit is activated at a time. As a consequence, identical output subcircuits are never used in an output block at the same time. Except for the single subcircuit output blocks, {1}, {2} and {3}, the possible combinations of subcircuits are {1,2}, {1,3}, {2,3}, and {1,2,3}. Other combinations of output blocks including {1,1}, {2,2}, {3,3} {1,1,1}, {1,1,2}, {1,1,3}, {1,2,3}, {1,2,2}, {2,2,2}, {2,2,3}, {2,3,3}, or {3,3,3}, are never used. This approach greatly simplifies MVL minimization because the only block that changes in a circuit is the control block and the control block can be realized by using a binary logic synthesizer. However, implementation of a MVL function with a limited selection of output blocks does not always end up with the smallest circuit. It is desirable to take the unused combinations into consideration during minimization so as to choose a better output block for a given MVL function.

As well the output block for a 4-valued logic design can be one of various combinations of the output subcircuits shown in Figure 4.1. For example, instead of using all three subcircuits (i.e, {1,2,3}), subcircuit 1 and subcircuit 2 (i.e., {1,2}) can work cooperatively to generates 20$\mu$A, 40$\mu$A and 60$\mu$A. In this case, 20$\mu$A is generated by a 20$\mu$ current source, 40$\mu$A is generated by a 40$\mu$A current source, and 60$\mu$A is generated by turning on both switches connected to a 20$\mu$ current source and a 40$\mu$A current source, respectively. Alternatively, three subcircuit 1s (i.e., {1,1,1}) can be used to generates 20$\mu$A, 40$\mu$A and 60$\mu$A. In this case, 20$\mu$A is still generated by a 20$\mu$ current source, 40$\mu$A is generated by two 20$\mu$ current sources, and 60$\mu$A is generated by three 20$\mu$ current sources. It should be noted that any one of the combinations is

not absolutely better than another in terms of circuit size.

In the above two examples, both {1,2} and {1,1,1} can be used for generating 20$\mu$A, 40$\mu$A and 60$\mu$A because the *sum* operation is involved. However, a problem associated with the *sum* operator is that the output block is no longer fixed. It is now possible for the MVL function to be implemented with different output blocks. For example, a {2} output block can now be replaced with an {1,1} output block. A {1,2,3} output block can be replaced with other output blocks such as {1,1,2}, {1,1,3}, or {1,2,2} in addition to {1,2}, {1,1,1}, In other words, the flexibility offered by the *sum* operator complicates the MVL minimization process. As a result, the output block must be used together with proper minimization so that the circuit size of MVL designs can be reduced.

Further study has shown that selection of an output block is not obvious even for a single-variable MVL function. Figure 4.2 shows two different approaches to minimize a single-variable 4-valued logic function. It is very straightforward to use an {1,2} output block to implement this function as shown in Figure 4.2($b$) where the $v_j$ subfunctions are

$$v_1 = x_2 \cdot \bar{x}_3 \quad \text{and} \quad v_2 = x_3$$

The circuit realized according to this minimization is shown in Figure 4.3($a$). The circuit consists of 12 transistors. Another way to minimize the function is shown in Figure 4.2($c$) where the function is realized by using a {1,1} output block is used, i.e., two identical subcircuits are used in the output block. The two subcircuits are controlled by two $v_j$ subfunctions, $v_1'$ and $v_1''$, respectively, where

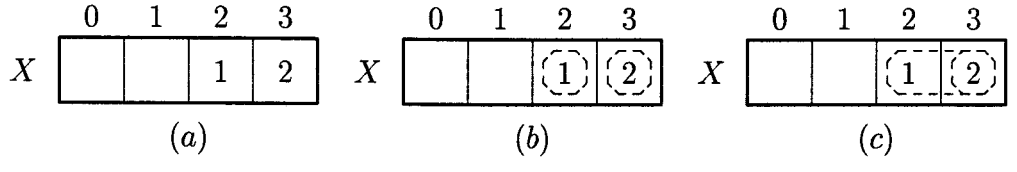
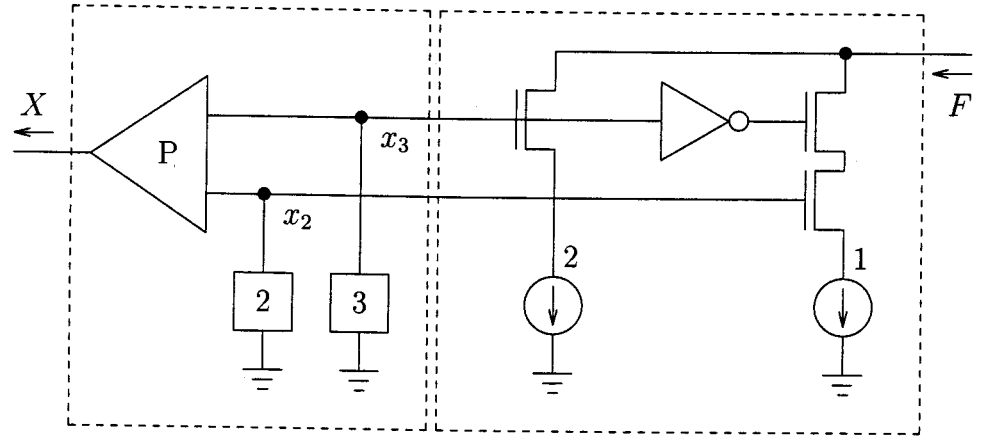
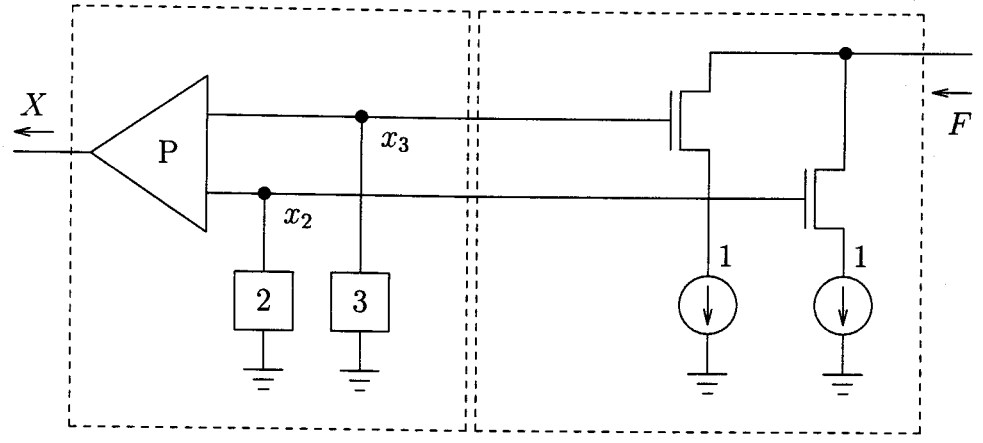



**Figure 4.2** ($a$) Truth table of a single-variable 4-valued function. ($b$) minimization by using {1,2} output block. ($c$) minimization by using {1,1} output block.




(*a*)




(*b*)

**Figure 4.3** Circuit realizations of a single-variable 4-valued function by using two different types of output blocks. (*a*) {1,2} output block. (*b*) {1,1} output block.

$$v_1' = x_2 \text{ and } v_1'' = x_3$$

Figure 4.3($b$) is the circuit realization according to these $v_j$ subfunctions. The resulting circuit consists of only 9 transistors. It can be seen from this example that a smaller circuit can be obtained by replacing a {2} output block with a {1,1} output block.

The selection of an output block is more complicated for a MVL function with multiple input variables. Figure 4.4($a$) is a 2-variable 4-valued function. If a {1,2,3} output block is chosen, the MVL function can be minimized into five implicants as shown in Figure 4.4($b$) and the three corresponding binary $v_j$ subfunctions are:

$$\begin{aligned} v_1 &= x_{0,1} \cdot \bar{x}_{0,2} \cdot x_{1,2} + x_{0,1} \cdot \bar{x}_{0,3} \cdot x_{1,3} \\ v_2 &= x_{0,2} \cdot x_{1,1} \cdot \bar{x}_{1,2} + x_{0,3} \cdot x_{1,1} \cdot \bar{x}_{1,3} \\ v_3 &= x_{0,2} \cdot \bar{x}_{0,3} \cdot x_{1,2} \cdot \bar{x}_{1,3} \end{aligned}$$

The total transistor count of this circuit implementation is in the range of 60 to 70 with different implementations. If a {1,2} output block is chosen, the MVL function can be minimized into two implicants as shown in Figure 4.4($c$) when the *sum* operator

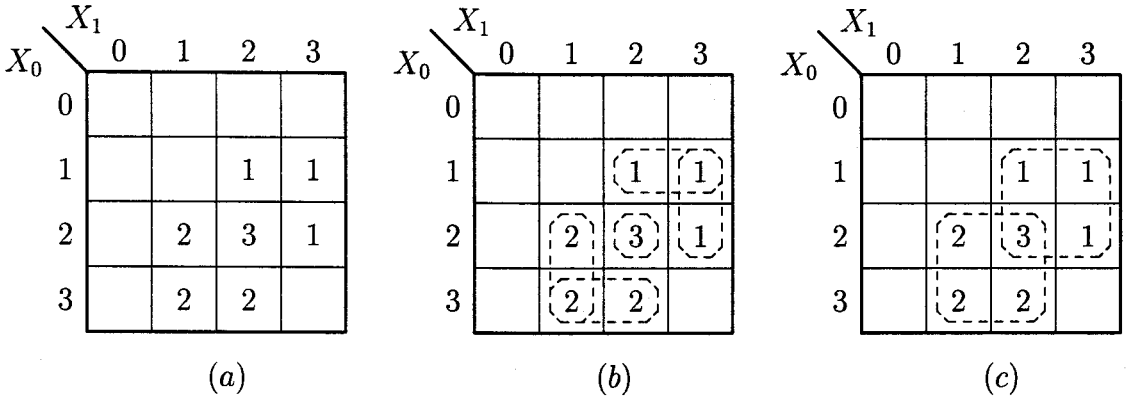



**Figure 4.4** ($a$) Truth table of a 2-variable 4-valued function. ($b$) minimization by using an {1,2,3} output block. ($c$) minimization by using an {1,2} output block.

is used and the two corresponding binary $v_j$ subfunctions are:

$$v_1 = x_{0,1} \cdot \bar{x}_{0,3} \cdot x_{1,2}$$

$$v_2 = x_{0,2} \cdot x_{1,1} \cdot \bar{x}_{1,3}$$

The circuit realization of these $v_j$ subfunctions is shown in Figure 4.5. The total transistor count of this circuit is 38 which is almost 40% smaller than the previous implementation. The {1,2} output block is better than the {1,2,3} output block in this case. It should be noted that the {1,2} output block instead of the {1,2,3} output block does not always end up with a smaller circuit.

Three more examples are shown in Figure 4.6 to demonstrate the concept of minimization using different types of output blocks. More complex functions can be minimized based upon the same concept.

The function shown in Figure 4.6($a$) can be minimized to two implicants by using

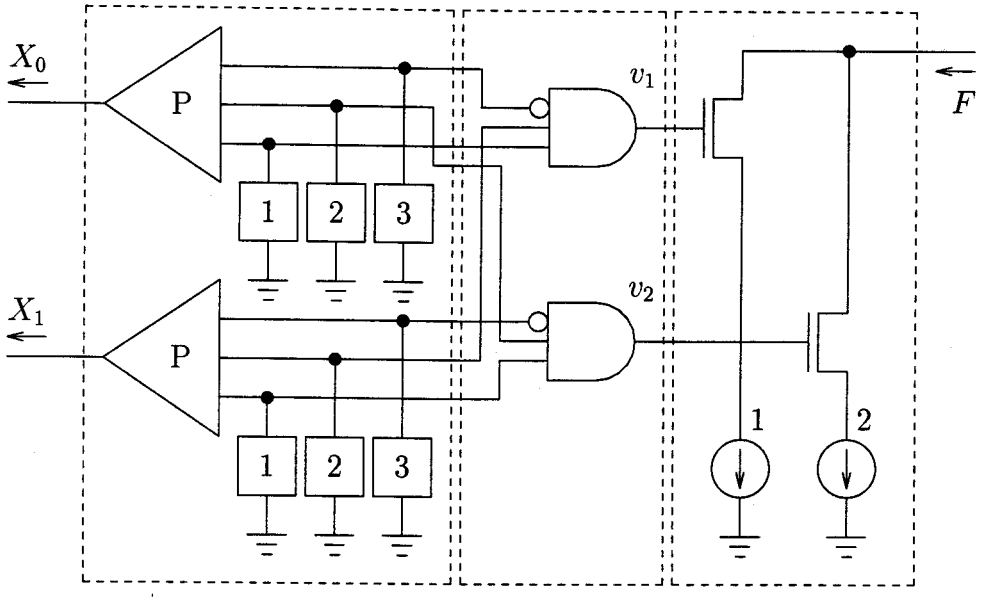



**Figure 4.5** Circuit realization of a 2-variable 4-valued logic function according to the minimization of Figure 4.4($c$).

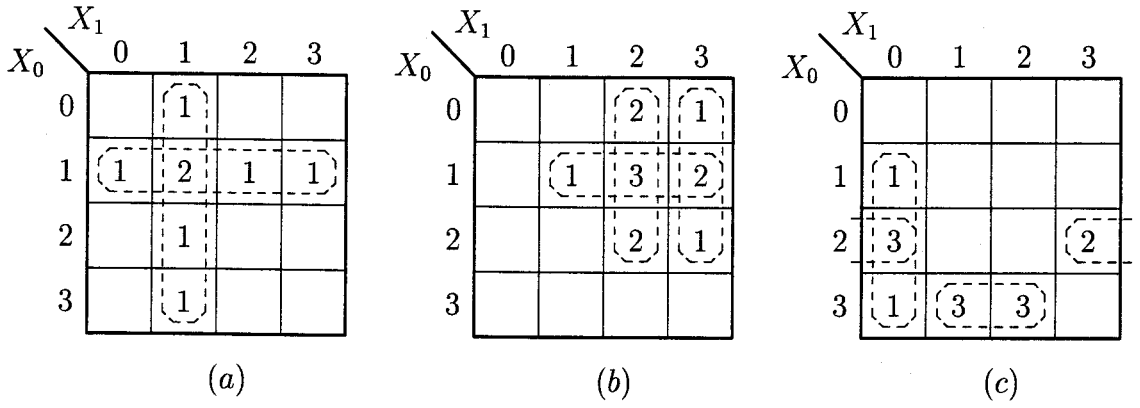



**Figure 4.6** Minimization of 2-variable 4-valued logic functions using the *sum* operation in an output block. (*a*) {1,1} output block. (*b*) {1,1,2} output block. (*c*) {1,2,3} output block.

a {1,1} output block, where

$$v_1' = x_{0,1} \cdot \bar{x}_{0,2} \quad \text{and} \quad v_1'' = x_{1,1} \cdot \bar{x}_{1,2}$$

and the resulting circuit consists of 24 transistors. Without using the *sum* operator in the output block, the same function is minimized to three implicants by using a {1,2} output block, the resulting circuit consists of 34 transistors.

The function shown in Figure 4.6(*b*) can be minimized to three implicants by using a {1,1,2} output block, where

$$\begin{aligned} v_1' &= x_{0,1} \cdot \bar{x}_{0,2} \cdot x_{1,1} \\ v_1'' &= \bar{x}_{0,3} \cdot x_{1,3} \\ v_2 &= \bar{x}_{0,3} \cdot x_{1,2} \cdot \bar{x}_{1,3} \end{aligned}$$

and the resulting circuit consists of 39 transistors. Alternatively, this function can be minimized to another set of implicants by using a {1,1,1} output block where

$$v_1' = x_{0,1} \cdot \bar{x}_{0,2} \cdot x_{1,1}$$

$$v_1'' = \bar{x}_{0,3} \cdot x_{1,2}$$
$$v_1''' = \bar{x}_{0,3} \cdot x_{1,2} \cdot \bar{x}_{1,3}$$

and the resulting circuit still consists of 39 transistors. Without using the *sum* operator in the output block, the same function is minimized to seven implicants by using a {1,2,3} output block; the resulting circuit consists of 68 transistors.

Figure 4.6($c$) can be minimized to three implicants by using {1,2,3} output block where

$$v_1 = x_{0,1} \cdot \bar{x}_{1,1}$$
$$v_2 = x_{0,2} \cdot \bar{x}_{0,3} \cdot (\bar{x}_{1,1} + x_{1,3})$$
$$v_3 = x_{0,3} \cdot x_{1,1} \cdot \bar{x}_{1,3}$$

and the resulting circuit consists of 44 transistors. Again, without using the *sum* operator in the output block, the function is minimized to five implicants by using the same {1,2,3} output block; the resulting circuit consists of 53 transistors.

The three examples show that circuit size can be reduced significantly through choice of a proper output block resulting in better minimization. The tradeoff is that the minimization becomes more complicated because the output block for a given function is not fixed and neither are the $v_j$ subfunctions. For a self-restored MVL design realized without using the *sum* operator most of the minimization work is done by a binary logic synthesizer. If the *sum* operator is used, a computer program that is able to take over part of minimization work from the binary logic synthesizer is needed. The program must work interactively with a binary logic synthesizer in order to choose an output block from a number of candidates and obtain a proper set of $v_j$ subfunctions.

## 4.2 Use of the Sum and Diff Operators in an Input Block

The *sum* operator can also be used in the input block of a self-restored design, but minimization becomes quite different from that previously discussed. For example, for a 2-variable 4-valued logic truth table, entries can now be merged diagonally or triangularly. The *diff* operator can be used in an input block in a similar manner with the *sum* operator except that implicants are flipped horizontally or vertically as will be described below. The *sum* and *diff* operators can also be used in an input block or in both an input block and an output block at the same time. Examples are given for each case to demonstrate how to use these two operators in an input block.

Figure 4.7 shows three examples of minimization of 2-variable 4-valued logic functions using the *sum* operator in an input block. The function shown in Figure 4.7($a$) can be minimized to a single implicant and the corresponding $v_j$ subfunction is expressed in mathematical form as

$$v_3 = high \text{ if } X_0 + X_1 > 3.$$

Circuit realization of this function is shown in Figure 4.8($a$). The total transistor

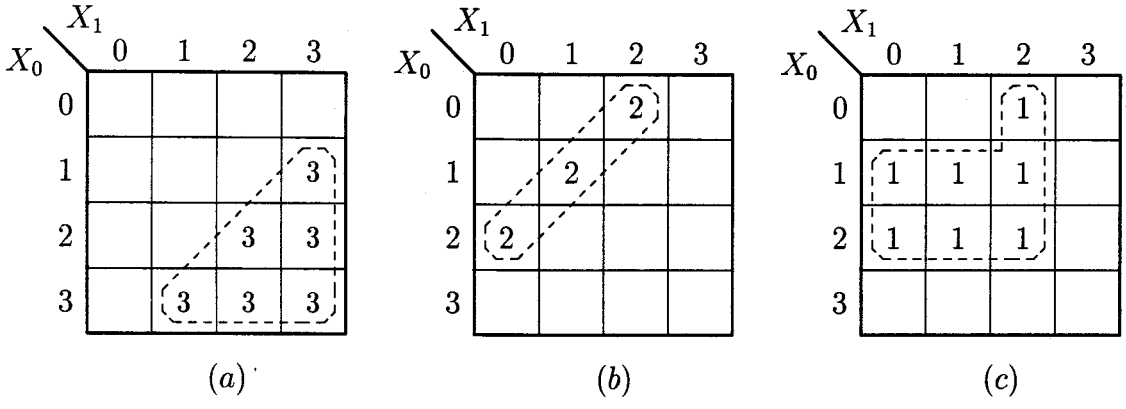



**Figure 4.7** Minimization of 2-variable 4-valued logic functions using the *sum* operation in an input block.

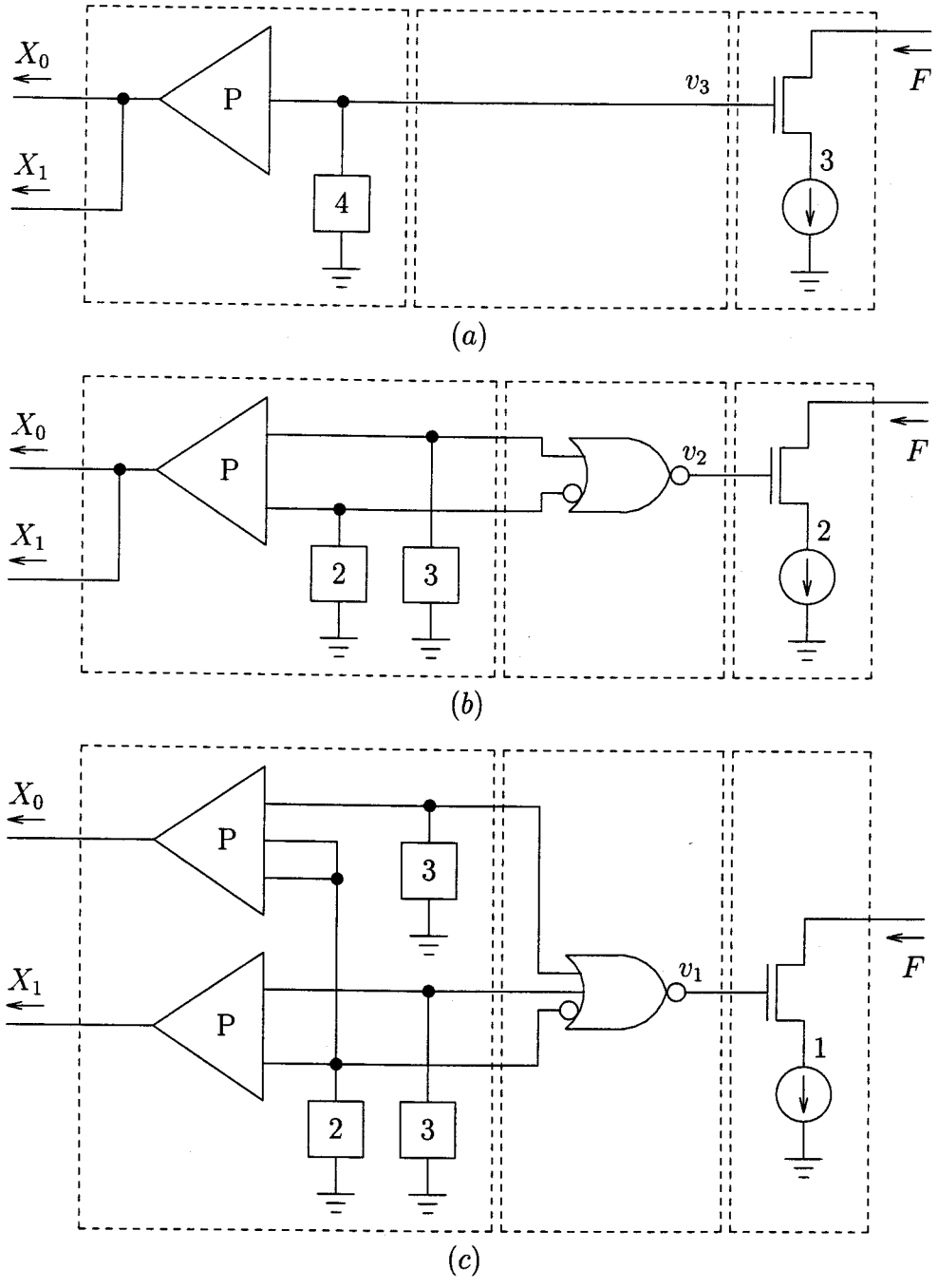



**Figure 4.8** Circuit realization of the 2-variable 4-valued logic functions in Figure 4.7.

count of this circuit is 5. The equivalent MVL circuit without using the *sum* operator consists of 31 transistors.

The function shown in Figure 4.7($b$) can be minimized to a single implicant and the correspond $v_j$ subfunction is expressed in mathematical form as

$$v_2 = high \text{ if } 1 < X_0 + X_1 < 3.$$

Circuit realization of this function is shown in Figure 4.8($b$). The total transistor count of this circuit is 13. The equivalent MVL circuit without using the *sum* operator consists of 41 transistors.

The function shown in Figure 4.7($c$) can also be minimized to a single implicant and the corresponding $v_j$ subfunction is expressed in mathematical form as

$$v_1 = high \text{ if } 2X_0 + X_1 > 1,\ X_0 < 3 \text{ and } X_1 < 3.$$

Circuit realization of this function is shown in Figure 4.8($c$). The total transistor count of this circuit is 20. The equivalent MVL circuit without using the *sum* operator consists of 21 transistors.

As shown from these examples, using the *sum* operator in an input block can reduce circuit size in certain cases. As mentioned previously, minimization of a MVL function using the *diff* operator in an input block is similar to that of using the *sum* operator except implicants are flipped horizontally or vertically. However, it should be noted logic minimization using *diff* operation instead of *sum* operation contains N-type current mirrors in resulting circuits. Figure 4.9 shows three examples of minimization of 4-valued logic functions using the *diff* operator in an input block.

The function shown in Figure 4.9($a$) can be minimized to a single implicant and the corresponding $v_j$ subfunction is expressed in mathematical form as

$$v_2 = high \text{ if } X_1 - X_0 > 1.$$

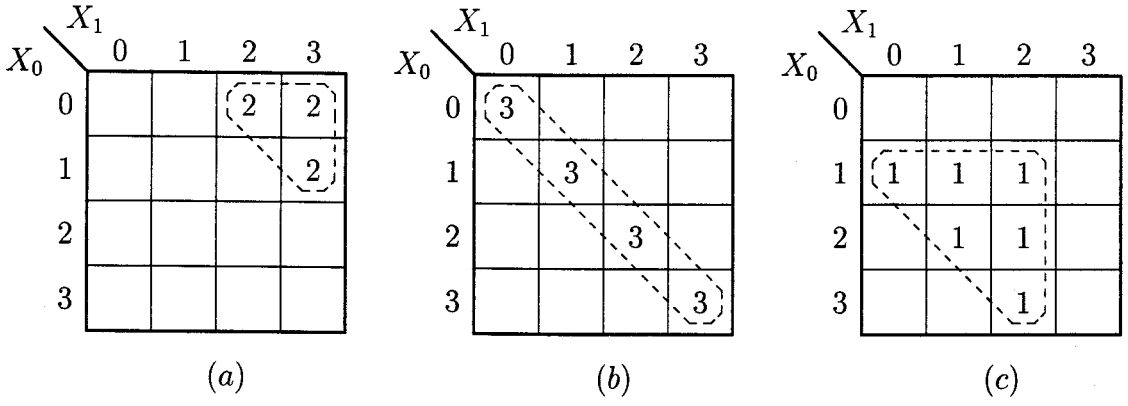



**Figure 4.9** Minimization of 2-variable 4-valued logic functions using the *diff* operation in an input block.

Circuit realization of this function is shown in Figure 4.10($a$). The total transistor count of this circuit is 9. The equivalent MVL circuit without using the *diff* operator consists of 26 transistors.

The function shown in Figure 4.9($b$) can be minimized to a single implicant and the corresponding $v_j$ subfunction is expressed in mathematical form as

$$v_3 = high \text{ if } X_0 - X_1 = 0.$$

Circuit realization of this function is shown in Figure 4.10($b$). The total transistor count of this circuit is 18. The equivalent MVL circuit without using the *diff* operator consists of 50 transistors.

The function shown in Figure 4.9($c$) can be minimized to a single implicant and the corresponding $v_j$ subfunction is expressed in mathematical form as

$$v_1 = high \text{ if } X_0 - X_1 < 2,\ X_0 < 3, \text{ and } X_1 < 3.$$

Circuit realization of this function is shown in Figure 4.10($c$). The total transistor count of this circuit is 21. The equivalent MVL circuit without using the *diff* operator consists of 34 transistors.

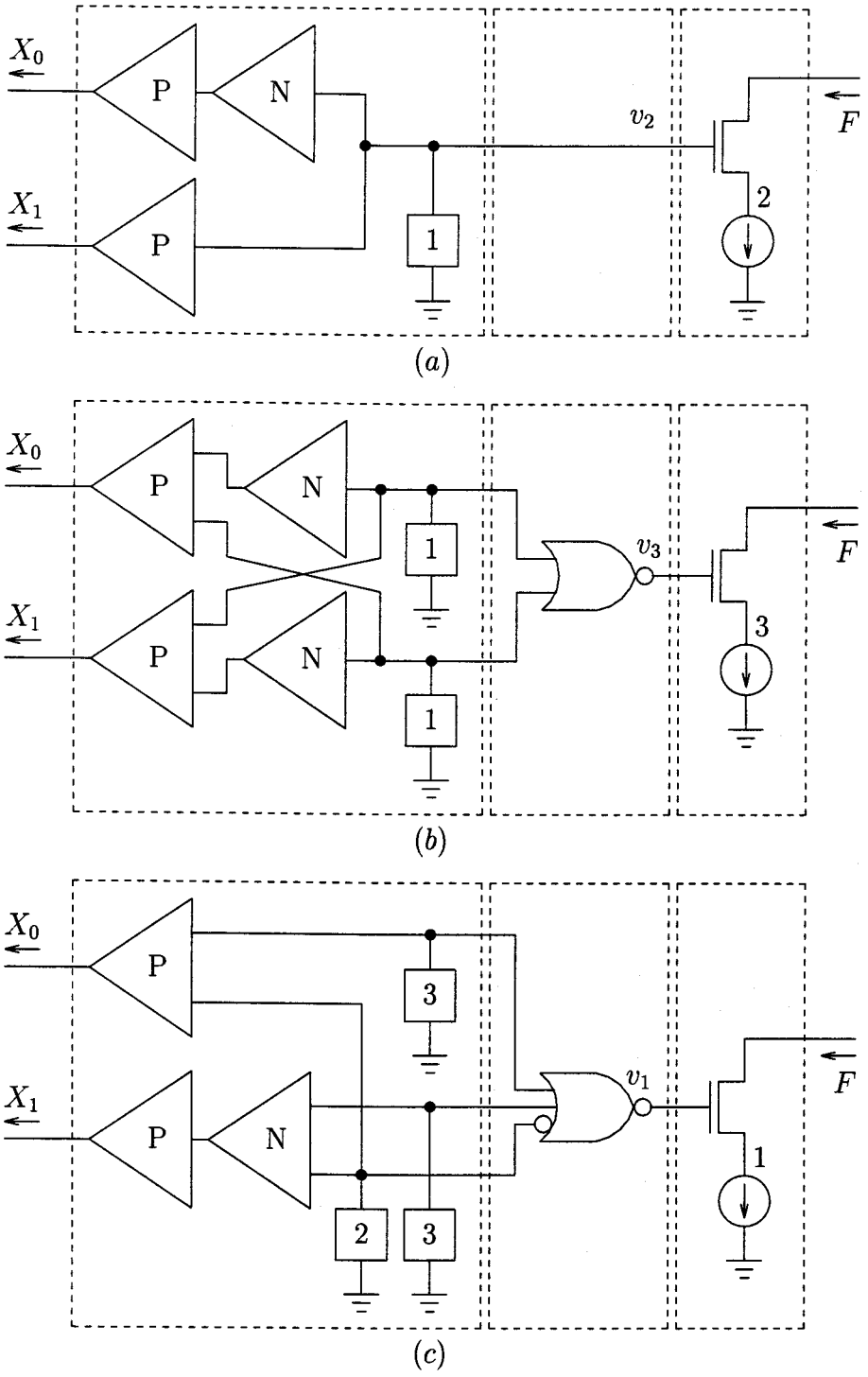



**Figure 4.10** Circuit realization of the 2-variable 4-valued logic functions in Figure 4.9.

Minimization using both the *sum* and the *diff* operators in an input block is suitable for some special MVL functions and results in smaller circuits. Figure 4.11 shows three examples of minimization of 2-variable 4-valued logic functions using both the *sum* and the *diff* operators in an input block.

The function shown in Figure 4.11($a$) can be minimized to a single implicant and the corresponding $v_j$ subfunction is expressed in mathematical form as

$$v_1 = high \text{ if } 2X_0 - X_1 > 2.$$

Circuit realization of this function is shown in Figure 4.12($a$). The total transistor count of this circuit is 10. The equivalent MVL circuit without using both the *sum* and the *diff* operators consists of 17 transistors.

The function shown in Figure 4.11($b$) can be minimized to a single implicant and the corresponding $v_j$ subfunction is expressed in mathematical form as

$$v_2 = high \text{ if } X_0 - 2X_1 < 0, \text{ and } X_0 < 3.$$

Circuit realization of this function is shown in Figure 4.12($b$). The total transistor count of this circuit is 18. The equivalent MVL circuit without using both the *sum* and the *diff* operators consists of 26 transistors.

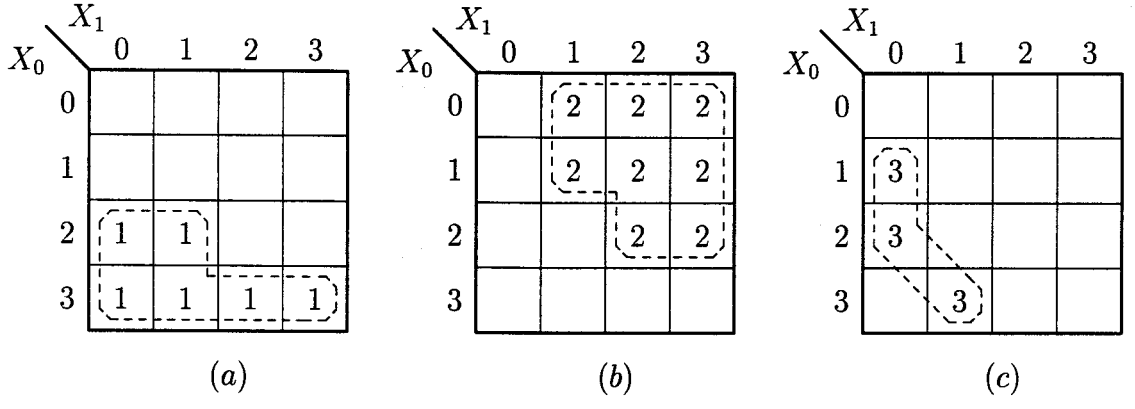



**Figure 4.11** Three examples of minimization of 4-valued logic functions using the *sum* and the *diff* operators in an input block.

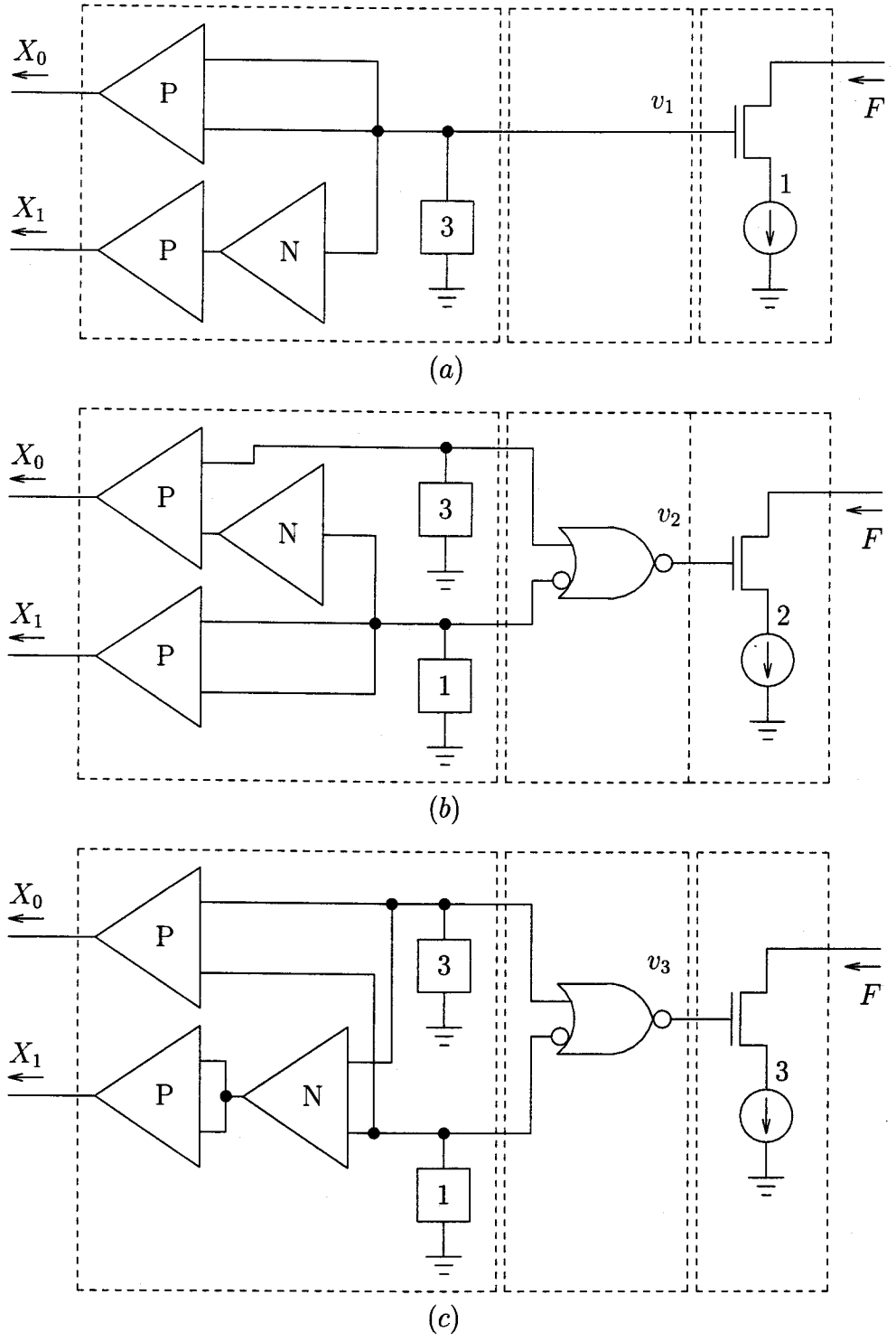



**Figure 4.12** Circuit realization of the 2-variable 4-valued logic functions in Figure 4.11.

The function shown in Figure 4.11($c$) can be minimized to a single implicant and the corresponding $v_j$ subfunction is expressed in mathematical form as

$$v_3 = high \text{ if } 0 < X_0 - 2X_1 < 3.$$

Circuit realization of this function is shown in Figure 4.12($c$). The total transistor count of this circuit is 19. The equivalent MVL circuit without using both the *sum* and the *diff* operator consists of 28 transistors.

To conclude this section, combined use of the *sum* and *diff* operators in both the input block and the output block is described through three examples. Circuit realization of these three example are not shown as they can be easily built based on the previous examples. Only the transistor counts are provided for comparison with equivalent MVL circuits. Needless to say, the tradeoff between the complexity of minimization and the resulting circuit size also applies here.

The function shown in Figure 4.13($a$) can be minimized to two implicants by using a {1,2} output block where the $v_j$ subfunctions are expressed in mathematical form as

$$\begin{aligned} v_1 &= x_{0,1} \cdot \bar{x}_{0,3} \cdot x_{1,1} \cdot \bar{x}_{1,3} \\ v_2 &= high \text{ if } X_0 + X_1 < 3 \end{aligned}$$

Circuit realization of this function needs 31 transistors. The equivalent MVL circuit without using the *sum* and the *diff* operator either in the input block or the output block consists of 53 transistors.

The function shown in Figure 4.13($b$) can be minimized to two implicants by using a {1,1} output block where the $v_j$ subfunctions are expressed in mathematical form as

$$v_1' = high \text{ if } 2X_0 - X_1 \leq 0 \text{ and } X_1 < 3$$

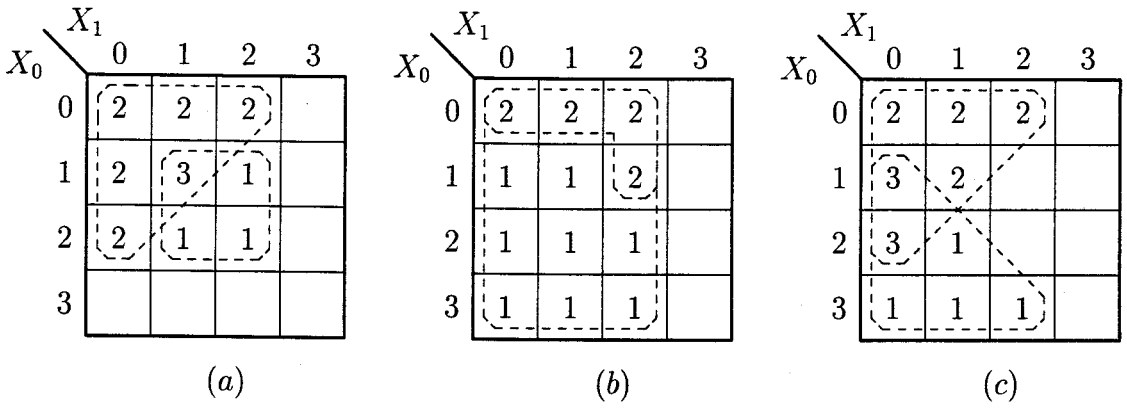



**Figure 4.13** Three examples of minimization of 4-valued logic functions using the *sum* and *diff* operations in both the input block and the output blocks.

$$v_1'' = \bar{x}_{1,3}$$

Circuit realization of this function needs 26 transistors. The equivalent MVL circuit without using the *sum* and the *diff* operator either in the input block or the output block consists of 37 transistors.

The function shown in Figure 4.13($c$) can be minimized to two implicants by using a {1,2} output block where the $v_j$ subfunctions are expressed in a mathematical form as

$$v_1 = high \text{ if } X_0 - X_1 > 0$$
$$v_2 = high \text{ if } X_0 + X_1 < 3$$

Circuit realization of this function needs 20 transistors. The equivalent MVL circuit without using the *sum* and the *diff* operator either in the input block or the output block consists of 63 transistors.

The transistor counts of the twelve examples in this section are summarized in the table shown below.

| Examples | Figure 4.7 | | | Figure 4.9 | | | Figure 4.11 | | | Figure 4.13 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | (a) | (b) | (c) | (a) | (b) | (c) | (a) | (b) | (c) | (a) | (b) | (c) |
| with *sum/diff* | 5 | 13 | 20 | 9 | 18 | 21 | 10 | 18 | 19 | 31 | 26 | 20 |
| without *sum/diff* | 31 | 41 | 21 | 26 | 50 | 34 | 17 | 26 | 28 | 53 | 37 | 63 |

It can be seen from the table that the circuit size of some self-restored MVL designs can be significantly reduced by using the *sum* and *diff* operators in an input block. The average transistor count of these twelve examples is 17.5, for MVL designs using the arithmetic operators, 33.5 for MVL designs without using the arithmetic operators, and 22.5 for the equivalent binary logic designs. In the best (worst) case, the circuit can even be reduced to about 16% (95%) of the original size. Even so, using the *sum* and *diff* operators does not always result in a smaller circuit. For example, it is obvious that using the *sum* and *diff* operators does not produce a smaller circuit for the MVL function shown in Figure 3.4(*a*). Figure 4.14(*a*) shows that a function using the *sum* and *diff* operators does not obviously result in a smaller circuit. When the arithmetic operators are used, the function can be minimized to three implicants as shown in Figure 4.14(*c*) and the corresponding $v_j$ subfunctions are

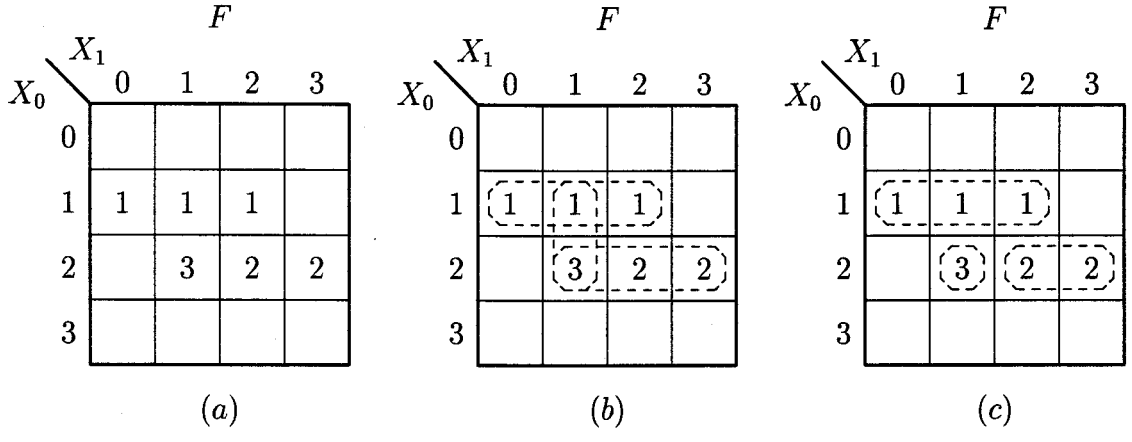



**Figure 4.14** An example showing that using the arithmetic operators might not result in a smaller circuit.

$$v_1 = \bar{x}_{13} \cdot x_{01} \cdot \bar{x}_{02} + x_{11} \cdot \bar{x}_{12} \cdot x_{01} \cdot \bar{x}_{03} \text{ and } v_2 = x_{12} \cdot \bar{x}_{02} \cdot \bar{x}_{03}$$

Without using the arithmetic operators, the function can still be minimized to three implicants as shown in Figure 4.14($b$) and the corresponding $v_j$ subfunctions are

$$v_1 = \bar{x}_{13} \cdot x_{01} \cdot \bar{x}_{02}, \quad v_2 = x_{12} \cdot x_{02} \cdot \bar{x}_{03} \text{ and } v_3 = x_{11} \cdot \bar{x}_{12} \cdot x_{02} \cdot \bar{x}_{02}$$

The former representation results in a slightly larger circuit. For non-obvious cases, a synthesis program must be able to work interactively with a binary logic synthesizer to determine $v_j$ subfunctions for this kind of MVL functions.

## 4.3 Summary

This chapter discusses the approaches of using the arithmetic *sum* and *diff* operators to reduce circuit size of a self-restored current-mode CMOS Multiple-Valued Logic (MVL) design. The *sum* and *diff* operators can be applied to an input block and/or an output block. The approach of using these arithmetic operators in an input block is quite different depending on how they are used in an output block. With proper minimization, circuit size of many self-restored MVL designs can be reduced significantly in term of transistor count. This chapter begins with the discussion of an approach to use the *sum* operator in an output block along with selection of an proper output block for a given MVL function, followed by an approach of using the *sum* and *diff* operators in an input block. Combined use of the two approaches is also discussed.

# 5. MVL Synthesis

A self-restored CMOS Multiple-Valued Logic (MVL) design architecture was proposed in Chapter 3, which consists of three blocks: an input block, a control block and an output block. The minimization procedure associated with the design architecture is also explained by examples. Theorem 3.1 stated that any $r$-valued multiple-valued function can be represented with $r-1$ disjoint binary $V_j$ subfunctions and implemented with this design architecture. Each $V_j$ subfunction consists of *up literal* PTs in a MOP form. Since the output of a *up literal* PTs is either 0 or $r-1$ because of the nature of the *up literal*, $V_j$ subfunctions can be converted into corresponding binary subfunctions, denoted as $v_j$, by replacing *min* with binary *and* and *max* with binary *or*. The binary $v_j$ subfunctions can then be realized by a control block using VMCL binary circuits. Replacing *min* and *max* operators with *and* and *or* operators implies possible reduction of resulting circuit size. This also implies that a binary logic synthesizer can be used for synthesis of a control block.

This chapter focuses on an approach to use a binary logic synthesizer with the self-restored MVL design architecture. It should be noticed that the approach to be described in this chapter works only for the self-restored architecture in Chapter 3. It does not extend to the variants of the self-restored architecture using the arithmetic *sum* and *diff* operators. Minimization of a MVL function using the two arithmetic operators, particularly in the input block, results in different solutions for the prime implicants as discussed in Chapter 4. This requires interative program executions and significantly increases the complexity and difficulties of finding a set of prime implicants that yield a smaller circuit. As stated in Chapter 8, the minimization algorithm of using the two arithmetic operators is one of the future research topics.

A wrapper program for MVL synthesis, WMS, will be discussed with examples to show how to apply the approach for the self-restored architecture. The WMS wrapper program works together with Synopsys Design Compiler to generate a self-restored MVL circuit in VHDL format from a MVL function in a truth table format. The current version can synthesize any-radix MVL functions with a reasonable number of input variables. Other features of the program include estimation of circuit size of the synthesized MVL circuits and automatic generation of Spectre netlists. The program is also able to generate a Verilog netlist for the control block. The Verilog netlist can be imported to the Cadence environment for Spectre simulation of the whole circuit or for other purposes.

Another wrapper program, M2B, is also designed based on the WMS program. The M2B program basically has the same features of the WMS wrapper program except that it generates the equivalent binary circuit from a given MVL truth table instead of a MVL circuit. The generated binary logic VHDL and SPICE netlists are used for comparison of the MVL design and binary design as will be discussed in Chapter 7.

## 5.1 WMS Wrapper Program

The binary $v_j$ subfunctions of a MVL function can be obtained from its truth table in a very similar way to the minimization of a Karnaugh map. As Karnaugh maps are based on Venn diagrams, only even number of minterms can be merged into an implicant. A MVL truth table, on the other hand, is not restricted to this rule. Several examples of 2-variable 4-valued logic functions are given in Chapters 3. The minimization method also applies to higher radix logic functions. Figure 5.1 shows truth tables of two 2-variable 5-valued logic functions. The binary $v_j$ subfunctions corresponding to the function of Figure 5.1($a$) are

$$v_1 = \bar{x}_{0,2} \cdot x_{1,1} \cdot \bar{x}_{1,2},$$

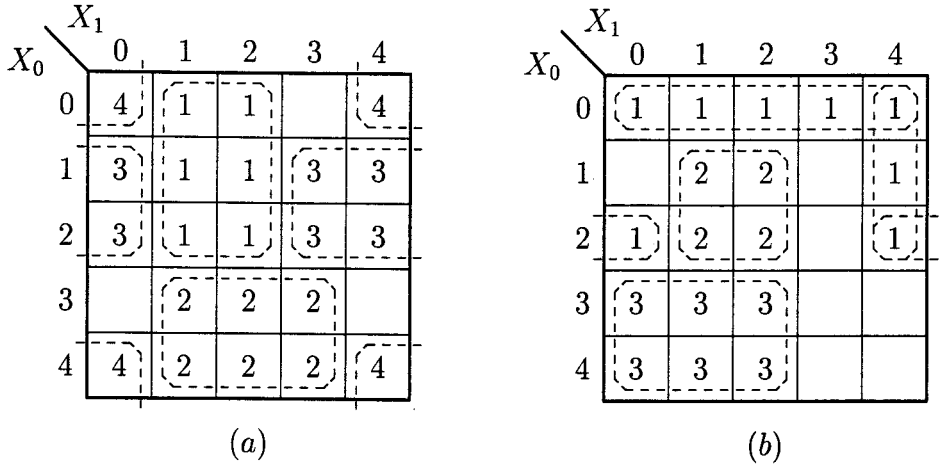



**Figure 5.1** Truth tables of two 5-valued logic functions.

$$\begin{aligned} v_2 &= x_{0,3} \cdot x_{1,1} \cdot \bar{x}_{1,4}, \\ v_3 &= x_{0,1} \cdot \bar{x}_{0,3} \cdot (\bar{x}_{1,1} + x_{1,3}) \text{ and} \\ v_4 &= (\bar{x}_{0,0} + x_{0,4}) \cdot (\bar{x}_{1,0} + x_{1,4}). \end{aligned} \tag{5.1}$$

Similarly, the corresponding binary $v_j$ subfunctions for the 2-variable 5-valued logic function of Figure 5.1($b$) can be expressed as

$$\begin{aligned} v_1 &= \bar{x}_{0,1} + \bar{x}_{0,3} \cdot x_{1,4} + x_{0,2} \cdot \bar{x}_{0,3} \cdot (\bar{x}_{1,1} + x_{1,4}), \\ v_2 &= x_{0,1} \cdot \bar{x}_{0,3} \cdot x_{1,1}\bar{x}_{1,3} \text{ and} \\ v_3 &= x_{0,3} \cdot \bar{x}_{1,3}. \end{aligned} \tag{5.2}$$

For a given MVL function, the control block implements the binary $v_j$ subfunctions. The input block and the output block for a given MVL function can be determined from the control block if the *sum* and *diff* operators are not considered in the binary $v_j$ subfunctions. Once a set of binary $v_j$ subfunctions is obtained, the MVL circuit is determined. That is, binary $v_j$ subfunctions contain all the information required for realizing a MVL function. For example, consider the 2-variable 5-valued

function of Figure 5.1($b$). It can be determined from the binary $v_j$ subfunctions of Eq. 5.2 that the control block must have three outputs, $v_1$, $v_2$ and $v_3$, where $v_1$ controls a switch connected with a logic 1 current source to generate $1 \times 20 = 20\mu$A, $v_2$ controls a switch connected with a logic 2 current source to generate $2 \times 20 = 40\mu$A. and $v_3$ controls a switch connected with a logic 3 current source to generate $3 \times 20 = 60\mu$A, It can also be determined from Eq. 5.2 that a 3-output current mirror is required for both inputs, $X_0$ and $X_1$. The 3-output current mirror for input $X_0$ has its outputs connecting to a logic 1 threshold, a logic 2 threshold and a logic 3 threshold, respectively in oder to generate binary signals, $x_{0,1}$, $x_{1,2}$ and $x_{0,3}$. The 3-output current mirror for input $X_1$ has its outputs connecting to a logic 1 threshold, a logic 2 threshold and a logic 3 threshold, respectively in oder to generate binary signals, $x_{1,1}$, $x_{1,3}$, and $x_{1,4}$. These binary signals including $x_{0,1}$, $x_{0,2}$, $x_{0,3}$, $x_{1,1}$, $x_{1,3}$, and $x_{1,4}$, are the inputs to the control block. Therefore, the control block is a 6-input 3-output binary design.

The binary $v_j$ subfunctions often have common input variables, which indicates they should be checked for the possibility of minimization before being realized with binary gates. For example, in Eq. 5.2, $v_1$, $v_2$ and $v_3$ are all functions of $x_{0,3}$ and $x_{1,3}$, and $v_1$ and $v_2$ are also functions of $x_{0,1}$. Since the binary $v_j$ subfunctions are binary functions, minimization of this multi-input multi-output binary control block can be done with a binary logic synthesizer. However, it should be noted that the $x_{i,j}$ variables are considered as independent variables by a binary logic synthesizer. While a binary logic synthesizer recognizes the fact that $\bar{x}_{u,v} \cdot x_{u,v} = 0$ and $\bar{x}_{u,v} + x_{u,v} = 1$, it does not recognize rules such as $x_{u,v}$ covers $x_{u,v+1}$ or $\bar{x}_{u,v+1}$ covers $\bar{x}_{u,v}$, Therefore, the binary $v_j$ subfunctions passed to a binary logic synthesizer are better expressed in terms of prime implicants instead of minterms. Passing prime implicants to a binary logic synthesizer also guarantees that input variables will not disappear during minimization by the binary logic synthesizer. Referring to Figure 5.1($b$) and Eq. 5.2, none of the input variables, $x_{0,1}$, $x_{0,2}$, $x_{0,3}$, $x_{1,1}$, $x_{1,3}$, and $x_{1,4}$, will be cancelled by

minimization. In brief, the task of MVL synthesis with respect to the self-restored architecture is reduced to finding prime implicants and passing them to a binary logic synthesizer.

The truth table method described in Chapter 3 is an effective way to simplify MVL functions which have a smaller number of input variables. When the number of input variables is large or if several functions must be simplified, use of a computer program is desirable. A wrapper program for MVL synthesis, WMS, was therefore developed for replacing manual deduction of binary $v_j$ subfunctions as well as many other tasks. The WMS wrapper program generates a structural model of the top level circuit in VHDL format of a given MVL function according to the self-restored architecture and also generates a separate VHDL file for binary $v_j$ subfunctions in terms of prime implicants. As part of the program, the WMS wrapper program then automatically invokes a binary logic synthesizer to synthesize the VHDL file and generate an actual circuit of the control block. The current version of the WMS wrapper program was designed to work with Synopsys Design Compiler, but any other binary logic synthesizer can also be used provided they accept VHDL. A number of other aspects were also considered during development for easy and fast comparison of a self-restored MVL design with an equivalent operator-based MVL design as well as with an equivalent binary design. As will be described in Chapter 7, the comparison involves many tasks. For example, in order to compare time delay and average power dissipation, it is necessary to run circuit simulations. In other words, SPICE netlists are required. A SPICE netlist can be generated from a given MVL truth table by schematic entry or manual input. Both approaches are time consuming and error prone. Automatic generation of SPICE netlists is preferred. The program should also be able to calculate the size of a synthesized MVL circuit in terms of transistor count so as to compare circuit size.

The algorithm for obtaining prime implicants of binary $v_j$ subfunctions is similar to the Quine-McClusky method [61, 62]. The major difference is the rules for

combining minterms. The Quine-McClusky method combines minterms by systematically applying the theorem $xy + x\bar{y} = x$. The WMS program combines minterms by applying the following rules.

**Rule 5.1** $^{a}X^{u} \blacklozenge {}^{v}X^{b} = {}^{a}X^{b}$ if $a \leq v \leq u \leq b$

**Rule 5.2** $^{a}X^{u} \blacklozenge {}^{v}X^{b} = {}^{v}X^{b}$ if $v \leq a \leq u \leq b$

**Rule 5.3** $^{0}X^{u} \blacklozenge {}^{v}X^{r-1} = \overline{{}^{u+1}X^{v-1}}$ if $u+1 \leq v-1$

**Rule 5.4** $\overline{{}^{a}X^{b}} = {}^{0}X^{a-1} \blacklozenge {}^{b+1}X^{r-1}$

where $^{0}X^{a-1} = 0$ if $a = 0$, and $^{b+1}X^{r-1} = 0$ if $b = r-1$.

**Rule 5.5** $^{0}X^{r-1} = r-1$

It can be seen from the rules that, unlike the Quine-McCluskey method, two minterms differing in only one literal do not necessarily cancel that literal or might not be even mergeable. Allen and Givone [14] and W. R. Smith [63] proposed algorithms similar to Quine-McCluskey method in the past, but their algorithms are based on different set of rules. More importantly, the algorithms are not suitable for implementation of disjoint binary $v_j$ subfunctions since a prime implicant in one scheme may not be considered as one in another scheme.

The notation, $(a, b)$, represents a *literal* $^{a}X^{b-1}$ when $a < b$ or a *complement of literal* $\overline{{}^{b}X^{a-1}}$ when $a > b$. The only exceptions are $(r-1, 0)$ and $(0, 0)$. $(r-1, 0)$ represents $^{r-1}X^{r-1}$ and $(0, 0)$ represents a canceled variable. It should be noted that $a$ is never equal to $b$ unless both are zero. For example, for a 4-valued logic system, $(0, 1)$ means $^{0}X^{0}$, $(2, 3)$ means $^{2}X^{2}$, $(3, 2)$ means $\overline{{}^{2}X^{2}}$, and $(3, 0)$ means $^{3}X^{3}$. In addition, $(a_i, b_i)(a_j, b_j)$ represents a *min* operation on $(a_i, b_i)$ and $(a_j, b_j)$. Since *up literal* operators are to be used in the self-restored architecture, $(a, b)$ actually represents $^{a}X \bullet \overline{{}^{b}X}$ when $a < b$ or $^{a}X + \overline{{}^{b}X}$ when $a > b$. If $a = 0$ or $b = 0$, the corresponding variable associated with that *up literal* term is cancelled. Therefore,

$(0,1)$ means $\overline{{}^{1}X}$ and the corresponding binary logic term is $\bar{x}_1$. $(2,3)$ means ${}^{2}X \bullet \overline{{}^{3}X}$ and the corresponding binary logic term is $x_2 \cdot \bar{x}_3$. $(3,0)$ means ${}^{3}X$ and $(3,2)$ means ${}^{3}X + \overline{{}^{2}X}$ and the corresponding binary logic term is $x_3 + \bar{x}_2$. $(3,0)$ means ${}^{3}X$ and the corresponding binary logic term is $x_3$.

Table 5.1 shows the results of a step-by-step procedure (discussed below) for a 4-valued logic function given in Figure 5.2. In Figure 5.2, each minterm is labeled by a letter. For example, the five minterms for logic 1 are labeled with A, B, C, D and E respectively. These minterms are listed in Table 5.1 according to their logic values. For logic 3, A and B can be merged into one term by Rule 5.1. The new term is then appended at the end of the list and labeled as E. Since A and B are now covered by E, both are marked with $\surd$. Similarly, A and C are merged into F, and B and F are merged into G by Rule 5.3. D can not be merged with any other terms. E is covered by G by Rule 5.2 and is therefore marked with $\surd$. F is covered by G by Rule 5.2 and is therefore marked with $\surd$. The process stops when no further mergers can be made. The entries without $\surd$ means they are not covered by others and those are the prime implicants. Hence, the prime implicants for logic 3 is D:$(2,3)(3,0)$ and G:$(3,2)(2,3)$, where G covers B,F and in turn B,A,C. The lists of prime implicants for logic 2 and logic 3 can be obtained by using the same process. The prime implicants for logic 2 are I:$(3,1)(3,1)$ which covers E,H and in turn A,B,C,D. The prime implicants for logic 1 are K:$(2,3)(0,3)$ and L:$(1,3)(0,2)$, where K covers E,H and in turn E,B,D, and L covers F,I and in turn A,B,C,D. The corresponding binary $v_j$ subfunctions can be express as:

$$\begin{aligned} v_1 &= x_{0,1} \cdot \bar{x}_{0,3} \cdot \bar{x}_{1,2} + x_{0,2} \cdot \bar{x}_{0,3} \cdot \bar{x}_{1,3}, \\ v_2 &= (\bar{x}_{0,1} + x_{0,3}) \cdot (\bar{x}_{1,1} + x_{1,3}) \text{ and} \\ v_3 &= x_{0,2} \cdot \bar{x}_{0,3} \cdot x_{1,3} + x_{0,2} \cdot \bar{x}_{0,3} \cdot (\bar{x}_{1,2} + x_{1,3}). \end{aligned} \tag{5.3}$$

The pseudocode of a subprogram that implements this algorithm is shown in

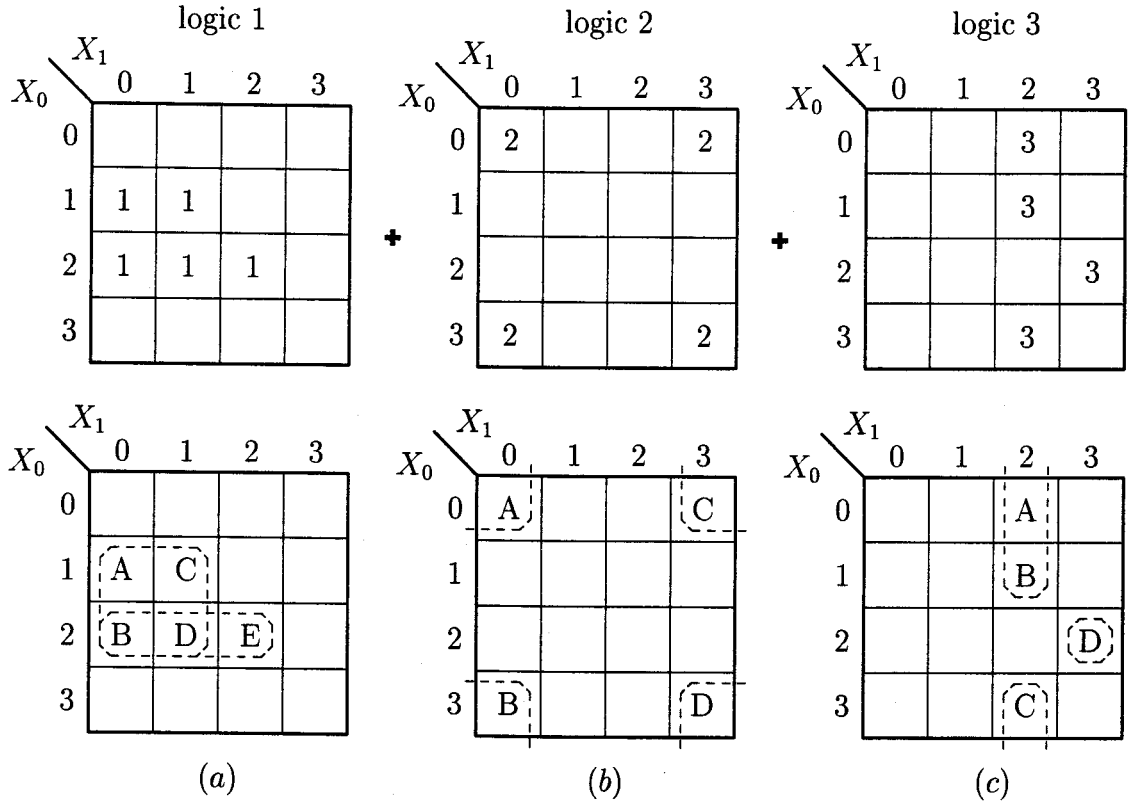



**Figure 5.2** A 4-valued logic function and its minterms.

**Table 5.1** The result of merging process yielding the prime implicants of the 2-variable 4-valued logic function of Figure 5.2.

| No. | logic 1 | | | logic 2 | | | logic 3 | | |
|---|---|---|---|---|---|---|---|---|---|
| A | A | (1,2)(0,1) | √ | A | (0,1)(0,1) | √ | A | (0,1)(2,3) | √ |
| B | B | (2,3)(0,1) | √ | B | (3,0)(0,1) | √ | B | (1,2)(2,3) | √ |
| C | C | (1,2)(1,2) | √ | C | (0,1)(3,0) | √ | C | (3,0)(2,3) | √ |
| D | D | (2,3)(1,2) | √ | D | (3,0)(3,0) | √ | D | (2,3)(3,0) | |
| E | E | (2,3)(2,3) | √ | A,B | (3,1)(0,1) | √ | A,B | (0,2)(2,3) | √ |
| F | A,B | (1,3)(0,1) | √ | A,C | (0,1)(3,1) | √ | A,C | (3,1)(2,3) | √ |
| G | A,C | (1,2)(0,2) | √ | B,D | (3,0)(3,1) | √ | B,F | (3,2)(2,3) | |
| H | B,D | (2,3)(0,2) | √ | C,D | (3,1)(3,0) | √ | | | |
| I | C,D | (1,3)(1,2) | √ | E,H | (3,1)(3,1) | | | | |
| J | D,E | (2,3)(1,3) | √ | | | | | | |
| K | E,H | (2,3)(0,3) | | | | | | | |
| L | F,I | (1,3)(0,2) | | | | | | | |

Figure 5.3. The subprogram starts with logic 1 and ends with logic $r-1$. It follows the algorithm to compare a pair of implicants at a time to see whether Rules 5.1 to 5.5 are applicable. If a start implicant covers a current implicant, then mark the current implicant. If the start implicant is covered by the current implicant, then mark the start implicant. If two implicants are mergeable, call the *merge_implicants()* function and mark both the start and current implicants. The *merge_implicants()* function merge the two implicants into a new implicant and check if there is an identical implicant on the list. If not, append the new implicant at the end of list.

Figure 5.4 is the pseudocode of the WMS core subprogram. After initialization and command option handling, the *get_minterm_list()* subprogram reads in a MVL function and generates a minterm list for each logic value. There are up to $r-1$ minterm lists. The core subprogram then calls the *get_prime_implicant_list()* subprogram to get prime implicants for each logic value based on the minterm lists as explained above through an example and the pseudocode shown in Figure 5.3. The resulting prime implicant lists correspond to binary $v_j$ subfunctions. Based on the prime implicant list, the *get_circuit_elements()* subprogram generates circuit topology which is used by the *generate_control_vhdl()*, *generate_top_vhdl()* and *generate_top_spice()* subprograms to generate VHDL and SPICE files.

The WMS wrapper program then invokes Synopsys Design Compiler to synthesize and optimize the behavioral VHDL description of the control block and to generate a structural VHDL description and a structural Verilog description of the control block. The structural Verilog code will be used later for generating a Spectre netlist of the control block. The WMS program also reports the total transistor count as well as circuit elements used in the input and output blocks.

As shown in Figure 5.5, the WMS program generates three VHDL files for the MVL function. One is the top level circuit description of the MVL functions which contains the structural description of the flattened input and output block circuit and an instantiated control block. The other two files are behavioral description of

```
procedure get_prime_implicant_list()
begin
        logic_value := 1
        while logic_value < radix
        begin
                if logic_value implicant list is not empty
                  start implicant := first implicant
                  while start implicant is not the last implicant
                  begin
                          current implicant := start implicant
                          while current implicant is not the last node
                          begin
                                  current implicant := next node
                                  compare start implicant and current implicant
                                  case compare result
                                          identical:
                                              remove current implicant
                                          start is covered by current :
                                              mark start implicant
                                          current is covered by start :
                                              mark current implicant
                                          mergeable :
                                              merge_implicants (start, current)
                                              mark both start and current implicants
                                          default:
                                              do nothing {can't merge}
                                  end case
                          end while
                          if start implicant is marked
                            delete start implicant
                          end if
                          start implicant := next implicant
                  end while
                end if
                increment logic_value
        end while
end get_prime_implicant_list
```

**Figure 5.3** Pseudocode of the subprogram for finding prime implicants.

```
procedure wms_core()
begin
        initialization()
        get_command_options()
        {generate minterm lists for each logic value of a MVL function}
        get_minterm_list()
        {get the prime implicant list for each logic value}
        get_prime_implicant_list()
        {derive binary v_j subfunctions and the corresponding circuits}
        get_circuit_elements()
        {generate behavioral VHDL code for control block}
        generate_control_vhdl()
        {generate top level structural VHDL code}
        generate_top_vhdl()
        {generate top level SPICE netlist}
        generate_top_spice()
end wms_core
```

**Figure 5.4** Pseudocode of the WMS core subprogram.

the control block: one for simulation and the other synthesis. The behavioral VHDL code for simulation is used to verify the synthesis result, which will be discussed in Chapter 6. The WMS program then invokes a binary logic synthesizer to optimize the synthesizable VHDL code of the control block in terms of area and to generate a structural VHDL description and a structural Verilog description of the control block. In the current version, the WMS program is designed to launch the Synopsys Design Compiler. The structural VHDL description is later automatically modified to the format that can be simulated by a VHDL simulator with the CMCL library. The structural Verilog code will be used later to generate a SPICE netlist of the control block. The binary logic synthesizer also saves the total transistor count of the control block in a report file. This information is sent back to the WMS program to estimate the total circuit size as well as the circuit elements used in the input and output blocks.

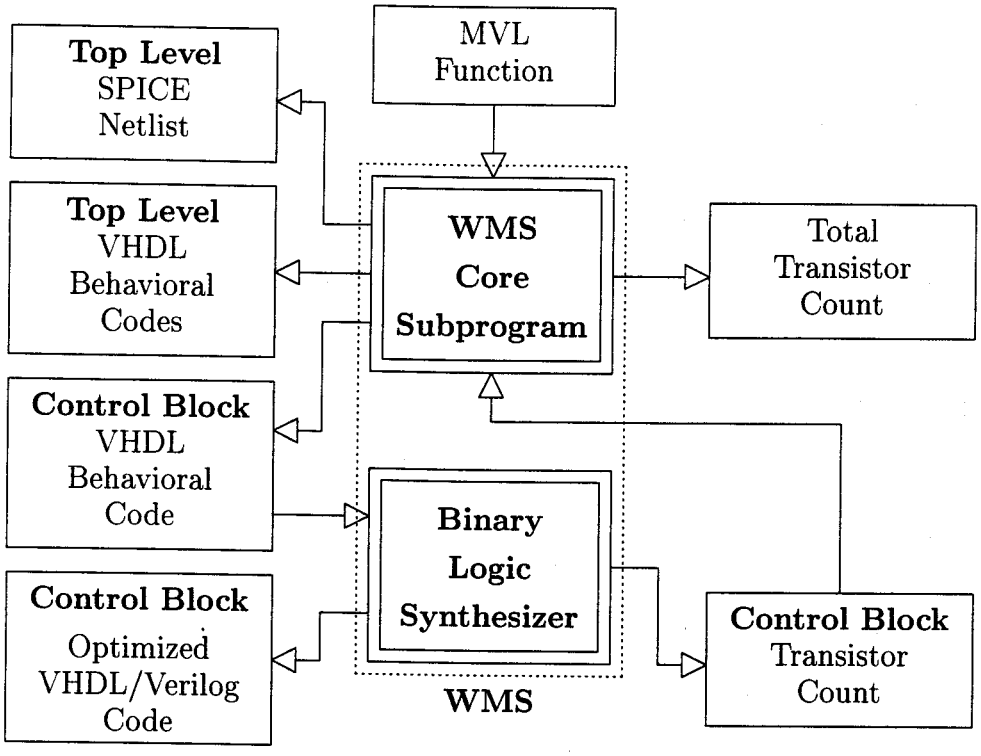



**Figure 5.5** Output files generated by the WMS wrapper program from a MVL function.

As the Cadence Spectre circuit simulator is used for power estimation, the SPICE netlist generated by the WMS program is in Spectre format. The Spectre file describes the top level circuit as well as *option* cards, *analysis* cards, *include* cards, local declarations, input sources, and output load. There are five *include* cards in a Spectre file:

- Global declarations such as $V_{DD}$, $V_{SS}$, $P_{ref}$, $N_{ref}$, etc.
- MVL library including MVL circuits such as current mirrors, current sources, thresholds, etc.
- Model file, i.e., SPICE parameters of PMOS and NMOS transistor models.
- Input file in which a variety of ideal and non-ideal input sources are defined.
- Control block. At this time, the Spectre file for the control block is still empty.

### 5.1.1 Generation of Spectre Netlist

The Spectre netlist of a control block is obtained according to the procedures shown in Figure 5.6. First of all, the Verilog code of the control block is automatically converted into a schematic by using the VerilogIn utility provided by Cadence. Cadence Analog Artist is then invoked on the schematic to generate a Spectre netlist from the schematic. Analog Artist provides an integrated environment for circuit simulation. A variety of application programs can be accessed through Analog Artist. The application programs include a waveform displayer, a calculator, a result browser, and various circuit simulators such as Spectre and HSPICE. Since a number of Spectre simulations are to be carried out to obtain an average power dissipation, batch simulation is preferred. As well, a control block has to be simulated with a corresponding top level circuit. Therefore, Analog Artist is only used for generating Spectre netlists for all control blocks. All MVL circuits were then simulated in batch mode afterward.

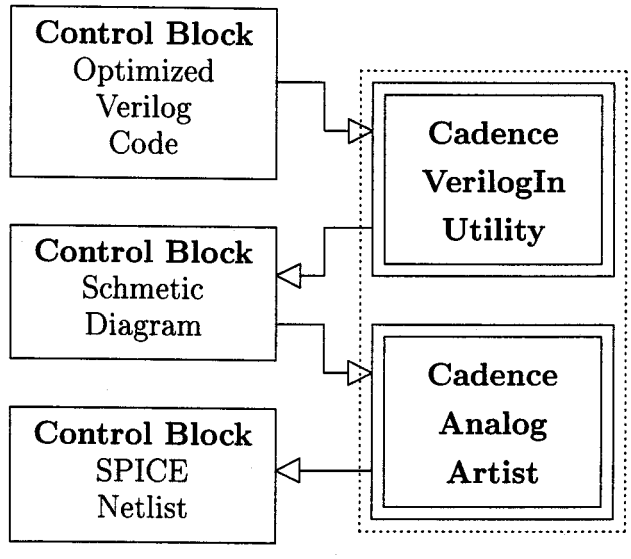
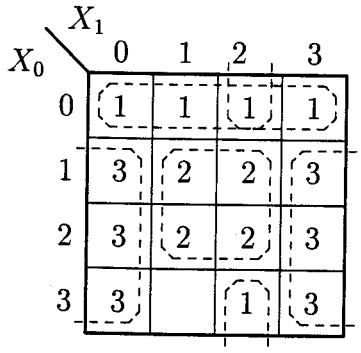



**Figure 5.6** Procedures to obtain Spectre netlist of a control block based on Verilog codes generated by the WMS program from a MVL truth table.

### 5.1.2 An Example

The truth table below represents a 2-variable 4-valued function.




To obtain the circuit for this function, it is required to obtain three $V_j$ subfunctions corresponding to logic values 1, 2 and 3. This truth table can be expressed with three binary subfunctions, $V_1$, $V_2$ and $V_3$, where

$$\begin{aligned} V_1 &= \overline{{}^1X_0} + {}^2X_1 \bullet \overline{{}^3X_1} \bullet (\overline{{}^1X_0} + {}^3X_0) \\ V_2 &= {}^1X_0 \bullet \overline{{}^3X_0} \bullet {}^1X_1 \bullet \overline{{}^3X_1} \text{ and} \\ V_3 &= {}^1X_0 \bullet (\overline{{}^1X_1} + {}^3X_1). \end{aligned} \tag{5.4}$$

The actual binary subfunctions implemented by the control block are

$$\begin{aligned} v_1 &= \bar{x}_{0,1} + x_{1,2} \cdot \bar{x}_{1,3} \cdot (\bar{x}_{0,1} + x_{0,3}) \\ v_2 &= x_{0,1} \cdot \bar{x}_{0,3} \cdot x_{1,1}\bar{x}_{1,3} \text{ and} \\ v_3 &= x_{0,1} \cdot (\bar{x}_{1,1} + x_{1,3}). \end{aligned} \tag{5.5}$$

It can be seen from these equations that the input block consists of a logic 1 threshold, a logic 3 threshold, as well as a 2-output current mirror for variable $X_0$, and a logic 1 threshold, a logic 2 threshold, a logic 3 threshold, and a 3-output current mirrors for variable $X_1$. The output block has three switches. The switch controlled by $v_i$ is connected with a $i \times 20\mu$A current source. For example, the switch controlled by $v_2$ is connected with a $40\mu$A current source.

The result of the procedure of getting prime implicants carried out by the WMS program is shown in Table 5.2. It can be seen from the table that the the prime implicants for logic 1 consists of $(3,1)(2,3)$ and $(0,1)(0,0)$, the prime implicants for logic 2 $(1,3)(1,3)$, and the prime implicants for logic 3 $(1,0)(3,1)$. As mentioned previously, $(3,1)(2,3)$ corresponds to $x_{1,2} \cdot \bar{x}_{1,3} \cdot (\bar{x}_{0,1} + x_{0,3})$ and $(0,1)(0,0)$ means $X_2$ is canceled by Rule 5.5 and corresponds to $\bar{x}_{0,1}$. The result conforms with $v_1$ in Eq. 5.5. Similarly, $(1,3)(1,3)$ for logic 2 corresponds to $x_{0,1} \cdot \bar{x}_{0,3} \cdot x_{1,1} \cdot \bar{x}_{1,3}$ and the result conforms with $v_2$ in Eq. 5.5, and $(1,0)(3,1)$ corresponds to $x_{0,1} \cdot (\bar{x}_{1,1} + x_{1,3})$ and the result conforms with $v_3$ in Eq. 5.5.

The output files generated by the WMS wrapper program are shown in Figures 5.7 to 5.10. Figure 5.7 is the structural VHDL code of the top level circuit of the given MVL function. Figure 5.8 is the VHDL code of the control block, i.e., behavioral

**Table 5.2** The result of merging process yielding the prime implicants of a 2-variable 4-valued logic function.

| No. | logic 1 | | | logic 2 | | | logic 3 | | |
|---|---|---|---|---|---|---|---|---|---|
| A | A | (0,1)(0,1) | √ | A | (1,2)(1,2) | √ | A | (1,2)(0,1) | √ |
| B | B | (0,1)(1,2) | √ | B | (1,2)(2,3) | √ | B | (2,3)(0,1) | √ |
| C | C | (0,1)(2,3) | √ | C | (2,3)(1,2) | √ | C | (3,0)(0,1) | √ |
| D | D | (0,1)(3,0) | √ | D | (2,3)(2,3) | √ | D | (1,2)(3,0) | √ |
| E | E | (3,0)(2,3) | √ | A,B | (1,2)(1,3) | √ | E | (2,3)(3,0) | √ |
| F | A,B | (0,1)(0,2) | √ | A,C | (1,3)(1,2) | √ | F | (3,0)(3,0) | √ |
| G | A,D | (0,1)(3,1) | √ | B,D | (1,3)(2,3) | √ | A,B | (1,3)(0,1) | √ |
| H | B,C | (0,1)(1,3) | √ | C,D | (2,3)(1,3) | √ | A,D | (1,2)(3,1) | √ |
| I | C,D | (0,1)(2,0) | √ | E,H | (1,3)(1,3) | | B,C | (2,0)(0,1) | √ |
| J | C,E | (3,1)(2,3) | | | | | B,E | (2,3)(3,1) | √ |
| K | C,F | (0,1)(0,3) | √ | | | | C,F | (3,0)(3,1) | √ |
| L | C,G | (0,1)(2,1) | √ | | | | C,G | (1,0)(0,1) | √ |
| M | D,F | (0,1)(3,2) | √ | | | | D,E | (1,3)(3,0) | √ |
| N | D,H | (0,1)(1,0) | √ | | | | E,F | (2,0)(3,0) | √ |
| O | D,K | (0,1)(0,0) | | | | | F,M | (1,0)(3,0) | √ |
| P | | | | | | | H,J | (1,3)(3,1) | √ |
| Q | | | | | | | I,N | (2,0)(3,1) | √ |
| R | | | | | | | K,P | (1,0)(3,1) | |

```
LIBRARY CCMVL, IEEE;
USE CCMVL.cmcl.ALL, IEEE.std_logic_1164.ALL;

ENTITY top IS
        PORT ( in0, in1 : INOUT node; fxout : INOUT node );
END top;

ARCHITECTURE structural OF top IS
        SIGNAL x0t1, x0t3, x1t1, x1t2, x1t3 : node := initNODE;
        SIGNAL y1, i1 : node := initNODE;
        SIGNAL y2, i2 : node := initNODE;
        SIGNAL y3, i3 : node := initNODE;

        COMPONENT nth
            GENERIC ( wgt : REAL );
            PORT ( inoutA : INOUT node );
        END COMPONENT;
        COMPONENT nsw
                PORT ( ctrl : IN node; inoutA, inoutB : INOUT node );
        END COMPONENT;
        COMPONENT control
            PORT ( x0t1, x0t3, x1t1, x1t2, x1t3 : INOUT node; y1, y2, y3 : INOUT node );
        END COMPONENT;
        COMPONENT pcm2
            -- GENERIC ( wgt1, wgt2 : REAL );
            PORT ( inA : INOUT node; outA, outB : INOUT node );
        END COMPONENT;
        COMPONENT pcm3
            -- GENERIC ( wgt1, wgt2 : REAL );
            PORT ( inA : INOUT node; outA, outB, outC : INOUT node );
        END COMPONENT;

        FOR ALL:nth     USE ENTITY CCMVL.nth(behavioral);
        FOR ALL:nsw     USE ENTITY CCMVL.nsw(behavioral);
        FOR ALL:control USE ENTITY CCMVL.control(behavioral);
        FOR ALL:pcm2    USE ENTITY CCMVL.pcm2(behavioral);
        FOR ALL:pcm3    USE ENTITY CCMVL.pcm3(behavioral);
BEGIN
        XCM0 : pcm2     PORT MAP (in0, x0t1, x0t3);
        XCM1 : pcm3     PORT MAP (in1, x1t1, x1t2, x1t3);
        TH01 : nth      GENERIC MAP (wgt => 0.5)
                        PORT MAP (x0t1);
        TH03 : nth      GENERIC MAP (wgt => 2.5)
                        PORT MAP (x0t3);
        TH11 : nth      GENERIC MAP (wgt => 0.5)
                        PORT MAP (x1t1);
        TH12 : nth      GENERIC MAP (wgt => 1.5)
                        PORT MAP (x1t2);
        TH13 : nth      GENERIC MAP (wgt => 2.5)
                        PORT MAP (x1t3);
        XCTR : control  PORT MAP (x0t1, x0t3, x1t1, x1t2, x1t3, y1, y2, y3);
        NSW1 : nsw      PORT MAP (y1, i1, fxout);
        SRC1 : nth      GENERIC MAP (wgt => 1.0)
                        PORT MAP (i1);
        NSW2 : nsw      PORT MAP (y2, i2, fxout);
        SRC2 : nth      GENERIC MAP (wgt => 2.0)
                        PORT MAP (i2);
        NSW3 : nsw      PORT MAP (y3, i3, fxout);
        SRC3 : nth      GENERIC MAP (wgt => 3.0)
                        PORT MAP (i3);
END structural;
```

**Figure 5.7** Top level VHDL code generated by the WMS wrapper program.

```
LIBRARY CCMVL, IEEE;
USE CCMVL.cmcl.ALL, IEEE.std_logic_1164.ALL;

ENTITY control IS
        PORT (  x0t1, x0t3, x1t1, x1t2, x1t3 : INOUT node;
                y1, y2, y3 : INOUT node );
END control;

ARCHITECTURE behavioral OF control IS
BEGIN
        y1 <= force(
        -- For (3,1) (2,3) :
                ((x0t3.B OR (NOT x0t1.B)) AND (x1t2.B AND (NOT x1t3.B))) OR
        -- For (0,1) (0,0) :
                ((NOT x0t1.B)));
        y2 <= force(
        -- For (1,3) (1,3) :
                ((x0t1.B AND (NOT x0t3.B)) AND (x1t1.B AND (NOT x1t3.B))));
        y3 <= force(
        -- For (1,0) (3,1) :
                ((x0t1.B) AND (x1t3.B OR (NOT x1t1.B))));

END behavioral;
```

**Figure 5.8** VHDL code of control block for simulation generated by the WMS wrapper program.

```
LIBRARY  IEEE;
USE  IEEE.std_logic_1164.ALL;

ENTITY control IS
        PORT (  x0t1, x0t3, x1t1, x1t2, x1t3 : IN std_logic;
                y1, y2, y3 : OUT std_logic );
END control;

ARCHITECTURE behavioral OF control IS
BEGIN
        y1 <=
        -- For (3,1) (2,3) :
                ((x0t3 OR (NOT x0t1)) AND (x1t2 AND (NOT x1t3))) OR
        -- For (0,1) (0,0) :
                ((NOT x0t1));
        y2 <=
        -- For (1,3) (1,3) :
                ((x0t1 AND (NOT x0t3)) AND (x1t1 AND (NOT x1t3)));
        y3 <=
        -- For (1,0) (3,1) :
                ((x0t1) AND (x1t3 OR (NOT x1t1)));

END behavioral;
```

**Figure 5.9** VHDL code of control block for synthesis generated by the WMS wrapper program.

```
library CCMVL, IEEE;

use CCMVL.cmcl.ALL, IEEE.std_logic_1164.all;

package CONV_PACK_control is

-- define attributes
attribute ENUM_ENCODING : STRING;

end CONV_PACK_control;

library CCMVL, IEEE;

use CCMVL.cmcl.ALL, IEEE.std_logic_1164.all;

use work.CONV_PACK_control.all;

entity control is

   port( x0t1, x0t3, x1t1, x1t2, x1t3 : IN node;  y1, y2, y3 : out
         std_logic);

end control;

architecture SYN_behavioral of control is

   component NOR4D1
      port( Z : OUT node;  A1, A2, A3, A4 : IN node);
   end component;

   component AO31D1
      port( Z : OUT node;  A1, A2, A3, B : IN node);
   end component;

   component AOI21D1
      port( Z : OUT node;  A1, A2, B : IN node);
   end component;

   component INVD0
      port( Z : OUT node;  A : IN node);
   end component;

   signal n4, n5, n6 : std_logic;

begin

   U7 : NOR4D1 port map( Z => y2, A1 => x0t3, A2 => x1t3, A3 => n4, A4 => n5);
   U8 : AO31D1 port map( Z => y1, A1 => x1t2, A2 => n6, A3 => x0t3, B => n5);
   U9 : AOI21D1 port map( Z => y3, A1 => x1t1, A2 => n6, B => n5);
   U10 : INVD0 port map( Z => n4, A => x1t1);
   U11 : INVD0 port map( Z => n5, A => x0t1);
   U12 : INVD0 port map( Z => n6, A => x1t3);

end SYN_behavioral;
```

**Figure 5.10** Optimized VHDL code of control block generated by the Synopsys Design Compiler.

description of the binary $v_j$ subfunctions. As will be described in Chapter 6 the VHDL codes are in the format for MVL simulation with the CMCL library. Figure 5.9 is the VHDL codes of the control block for synthesis; i.e., the code is in the format that can be read into Synopsys Design Compiler for minimization. All three files are automatically generated by the WMS program from the truth table. Comparing the two behavioral VHDL codes for the control block, it can be found that both signal types and formats of output equations are different. The signals in the VHDL file for simulation are declared as 'node', while the the signals in the synthesizable VHDL file are declared as 'std_logic'. The 'std_logic' type is the IEEE standard binary logic signal declaration for VHDL, while the 'node' type is defined in the CMCL package for VHDL simulation of mixed CMCL MVL and VMCL binary logic designs. The complete package is attached as Appendix C.1 and the detailed description is given in Chapter 6. Also, every output equation in the simulation file is assigned to an Boolean expression by using the 'force' function, which converts 'std_logic' type to 'node' type. Figure 5.10 is the structural VHDL code of the control block after being minimized by the Synopsys Design Compiler. It should be noted that `xitj` in the VHDL codes corresponds to $x_{i,j}$.

## 5.2 M2B Program

In order to compare a number of self-restored MVL designs and equivalent binary logic designs on speed, area and power dissipation, it is desirable to have a computer program similar to the synthesis program which is able to generate a Spectre netlist of an equivalent binary logic circuit form a MVL truth table. Such a program, M2B, was therefore developed. A large portion of the M2B program is the same as the WMS wrapper program. Basically, the M2B program works in a similar fashion as the WMS program does. First of all, the M2B program converts a MVL function into equivalent binary functions, and then generates a file containing Boolean equations.

Figure 5.11 shows two Karnaugh maps to explain how a MVL function is converted to its equivalent binary logic functions. Figure 5.11($a$) is a binary Karnaugh map equivalent to the truth table shown in Figure 3.11($b$) on page 45. Figure 5.11($b$) is a binary Karnaugh map equivalent to Figure 3.11($c$) also on page 45. An example is given in Figure 5.12 to show a MVL function (column MVL) is converted to Boolean equations (column Boolean Equations in Synopsys format) by the M2B program according to its equivalent binary function (column Binary). It should be noted that the format of Boolean equations is acceptable by the Synopsys Design Compiler. Unlike the WMS program that passes prime implicants to Synopsys Design Compiler, the M2B program passes minterms instead. As mentioned in Section 5.1, the input variables of binary $v_j$ subfunctions come from outputs of *up literals* and their complements. A binary logic synthesizer does not trace back their original relationships during minimization and therefore is unable to recognize, for example, $x_1 \cdot \bar{x}_3$ covers $x_1 \cdot \bar{x}_2$ for a single-variable MVL function. Those variables, $x_1$, $x_2$ and $x_3$, are deemed as independent variables by a binary logic synthesizer. Equivalent binary expressions however do not have such a problem and are not necessary to be in the form of prime implicants before being passed to a binary logic synthesizer.

The complete procedures for obtaining the Spectre netlists are shown in Figure 5.13. As shown in Figure 5.13, the M2B program, working together with a binary logic synthesis, to optimize the Boolean equations in terms of area and to generate VHDL or Verilog codes as well as transistor count. The remaining procedures for generating Spectre netlists are omitted because they are same as Figure 5.6, are already covered in Section 5.1.1.

## 5.3 Summary

An approach to using a binary logic synthesizer for MVL synthesis of the self-restored CMOS MVL design architecture is discussed in this chapter. A computer

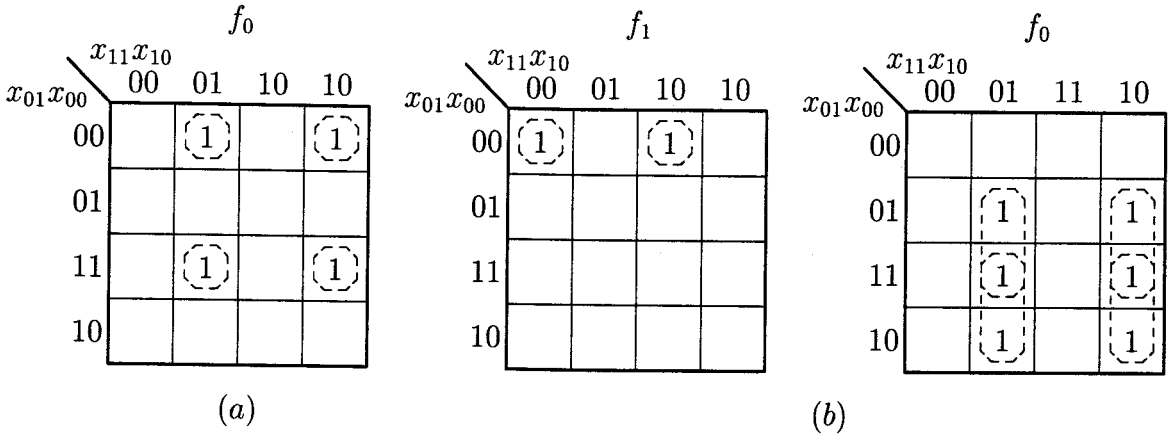



**Figure 5.11** (*a*) Corresponding binary Karnaugh map of Figure 3.11(*b*). (*b*) Corresponding binary Karnaugh map of Figure 3.11(*c*).

| MVL | | | Binary | | | | | | Boolean Equations in Synopsys Format |
|---|---|---|---|---|---|---|---|---|---|
| $X_1$ | $X_0$ | $F$ | $x_{11}$ | $x_{10}$ | $x_{01}$ | $x_{00}$ | $f_1$ | $f_0$ | |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | `.design_name control` |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | `.inputnames x00 x01 x10 x11` |
| 0 | 2 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | `.outputnames f0 f1` |
| 0 | 3 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | |
| 1 | 0 | 2 | 0 | 1 | 0 | 0 | 1 | 0 | `f0=((x0'*x1'*x2'*x3')+` |
| 1 | 1 | 3 | 0 | 1 | 0 | 1 | 1 | 1 | `(x0'*x1'*x2 *x3')+` |
| 1 | 2 | 2 | 0 | 1 | 1 | 0 | 1 | 0 | `(x0'*x1'*x2'*x3 )+` |
| 1 | 3 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | `(x0 *x1'*x2 *x3 )+` |
| 2 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | `(x0'*x1 *x2'*x3')+` |
| 2 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | `(x0 *x1 *x2 *x3')+` |
| 2 | 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | `(x0 *x1 *x2'*x3 )+` |
| 2 | 3 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | `(x0 *x1'*x2 *x3'))` |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | |
| 3 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | `f1=((x0 *x1'*x2'*x3')+` |
| 3 | 2 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | `(x0 *x1'*x2 *x3 )+` |
| 3 | 3 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | `(x0 *x1'*x2 *x3'))` |

**Figure 5.12** The conversion of a MVL function into Boolean equations by the M2B program.

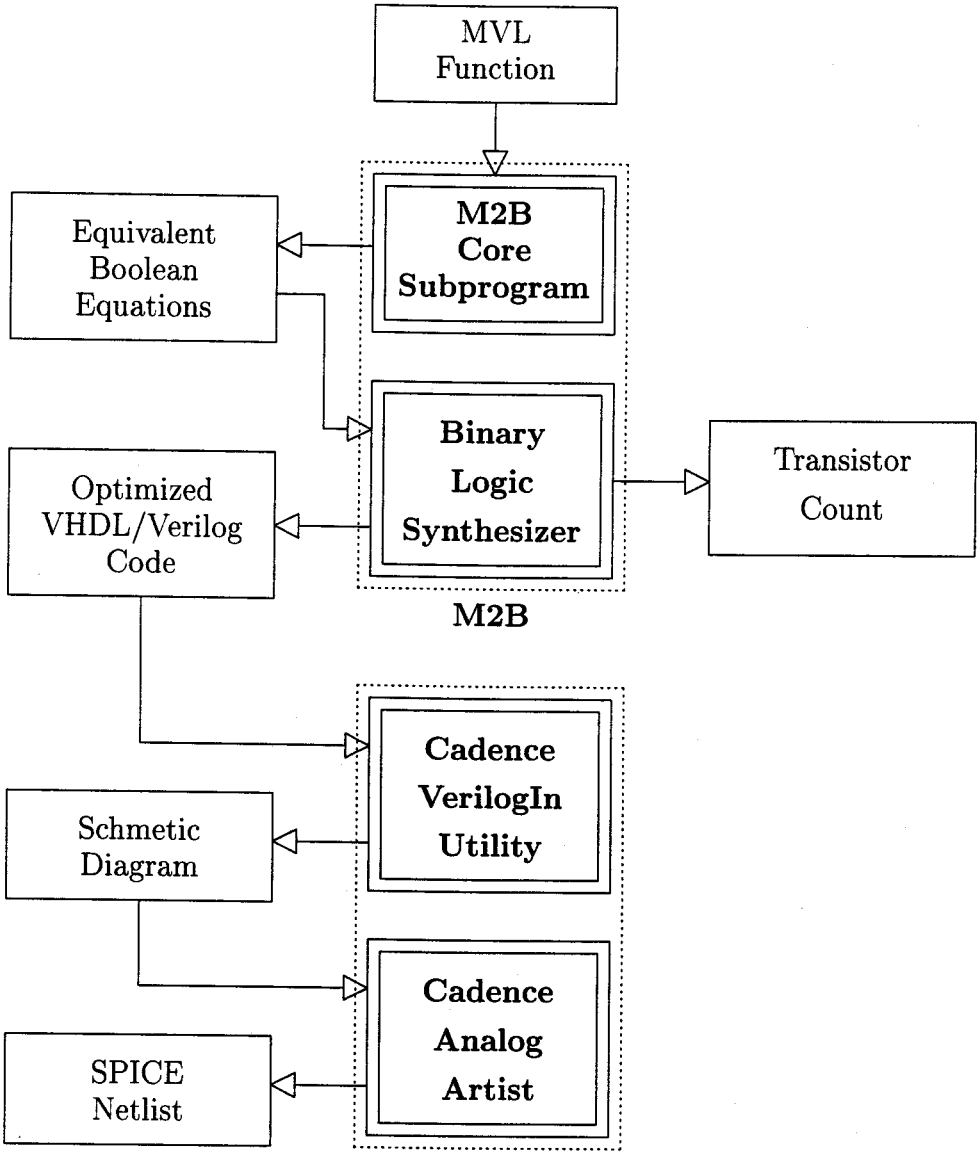



**Figure 5.13** Procedures to obtain Spectre netlist of an equivalent binary logic circuit from a MVL truth table by using the M2B program.

program for this purpose, WMS, was also developed to work with Synopsys Design Compiler to synthesize MVL functions according to the self-restored architecture without considering the arithmetic *sum* and *diff* operators. For a given MVL function, the WMS program obtains a list of minterms for each logic values from the functional description in a truth table format. Based on the minterm lists, the program derives prime implicants and corresponding disjoint binary $v_j$ subfunctions. The algorithm used to to derive prime implicants is similar to Quine-McCluskey method, but the rules of combining two terms are completely different. The binary $v_j$ subfunctions contains all the information required to construct a self-restored circuit for its corresponding MVL function. The WMS program generates several VHDL files for both simulation and synthesis from binary $v_j$ subfunctions. The VHDL simulation files are ready for simulation by using a VHDL simulator with the CMCL library to verify synthesized results. The details of the VHDL simulation will be covered in the next chapter. The VHDL synthesis file is passed to Synopsys Design Compiler for synthesis. The program also generates Spectre netlist for the top level circuit as well as a Verilog code of the control block. The Verilog code can be imported to Cadence Analog Artist for generating Spectre netlist of the control block. By combining the top level Spectre netlist and the control block Spectre netlist, circuit level simulations can be conducted to measure time delay and power consumption.

Based on the WMS wrapper program, a M2B program was also developed. For a given MVL function in a truth table format, the M2B program, working cooperatively with Synopsys Design Compiler and Cadence Analog Artist, generates equivalent binary logic circuit in three formats: VHDL code, Verilog code and Spectre netlist.

# 6. VHDL Simulation

In order to verify the self-restored architecture and test the WMS program, a tremendous amount of simulation must be carried out on the resulting circuits. As the self-restored MVL circuits consists of CMCL circuit elements such as current source and current mirror, it was inevitable to verify designs using analog simulators. Circuit simulators such as SPICE are widely available today. A major inconvenience of using a circuit simulator is the time required to perform the simulations. A more convenient method would be to use a high-level hardware description language (HDL) for circuit designs so that functional simulation of the circuit description can be performed. Nevertheless, both VHDL and Verilog were not designed for analog simulation as a primary requirement. The Accellera Verilog-AMS Technical Subcommittee and the IEEE 1076.1 Working Group have been working on analog and mixed-signal extensions to both languages.

Despite this, by taking advantages of VHDL's flexible value system and customizable bus resolution function [27, 28, 29, 30], a library (referred to as the "CMCL library" hereafter) which allows VHDL simulation of CMCL designs was proposed in the author's Masters thesis [12]. The CMCL library has undergone a number of refinements and is being improved on a continuing basis. In particular, the old CMCL library in [12] suffers from two drawbacks. First of all, the bus resolution function is so complex that maintenance and improvement are extremely difficult. Secondly, the switch model is not robust enough to handle various circuit topologies.

This chapter discusses the design and use of a new CMCL library for use with VHDL simulators. Starting with the library specification in Section 6.1, this chapter will provide the background knowledge of the CMCL library specification, followed by

the implementation details of new VHDL packages and library with emphasis on the bus resolution function, circuit functions, operator functions, and binary functions in Section 6.2. The interaction between the bus resolution function and circuit functions to determine currents in interconnections will also be explained. Section 6.3 presents two examples to show the usage of the CMCL library. Both VHDL simulation and SPICE transient analysis results are presented for comparison.

## 6.1 Library Specification

During the development of the CMCL library, a number of decisions had to be considered in its implementation. The first task in the development of the CMCL library was to specify what the library was expected to accomplish. Where possible, the CMCL library was written in a general way. This section describes the specification and structure of the VHDL library.

### 6.1.1 Current Distribution

The most complicated feature in the CMCL library is the current distribution. An example will explain the effect it had on the implementation of the library. A simple MVL circuit is shown in Figure 6.1($a$). It consists of an input MVL signal, an N-type current mirror, a P-type current mirror, and an N-type threshold. The threshold has been set to a value of $2.5I_0$, where $I_0$ is the base current (i.e., the difference between two adjacent logic levels). As long as the input current, $I_{in}$, is below $2.5I_0$, the circuit operates normally and $I_2 = I_1 = I_{in}$. When the input current exceeds $2.5I_0$, a resolution problem exists. Since the threshold element cannot conduct more than $2.5I_0$, a method of setting the P-type current mirror's output to $2.5I_0$ must be provided. Thus the threshold controls the value of the current $I_2$.

A variation of the simple MVL circuit is shown in Figure 6.1($b$). An N-type switch has been added between the P-type current mirror and the N-type current threshold.

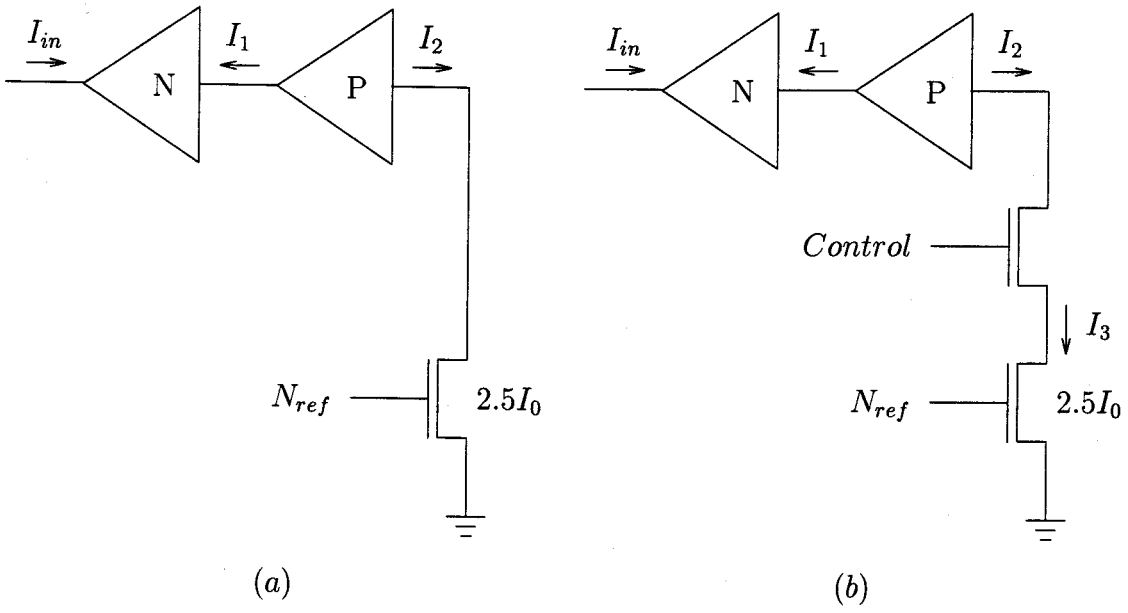



**Figure 6.1** CMCL bus resolution example circuits.

First consider the case where the control signal to the switch causes it to be OFF. In this case no current must flow from the P-type current mirror output or through the N-type threshold. The switch controls the value of the current $I_2$ and $I_3$ (i.e., both are $0.0\mu$A). Next consider the case where the control signal to the switch causes it to be ON. In this case the operation of the circuit is as above. The switch passes current without modification (note that $I_2$ must be equal $I_3$).

This implies that all MVL inputs are, in fact, of mode INOUT since they must also be capable of driving the output connection when necessary. This also implies that all MVL outputs are of mode INOUT since they must also be capable of being driven when necessary. This further implies that the various components (or entities) must have different driving types to resolve a node. In order to implement these refinements in the CMCL library, the mode of the PORTs on the MVL entities was set to INOUT and the basic MVL entities were assigned driving types. The latter further forced the use of aggregate type for MVL signals.

### 6.1.2 Terminal Types

Before designing a bus resolution function it is necessary to classify the terminals of elements which affect resolved logic value of a node. Basically, CMCL elements can be classified into the following three major categories in terms of terminal types.

**Category 1:** Single-terminal elements including N-type current source, P-type current source, N-type threshold, and P-type threshold.

**Category 2:** Multi-terminal elements including N-type current mirror and P-type current mirror.

**Category 3:** Binary gates.

If a current sink is deemed to be the same as a N-type threshold and a P-type current source is deemed as the same as a P-type threshold, only the following terminals will affect the resolved logic value of a node.

- Output of N-type current source (same as N-type threshold).
- Output of P-type current source (same as P-type threshold).
- Input of N-type current mirror.
- Output of N-type current mirror.
- Input of P-type current mirror.
- Output of P-type current mirror.
- Output of binary gates.

It should be noted that input terminals of a CMOS binary gate do not affect the logic value of a node in the ideal case because their input terminals are normally gate terminals of either PMOS or NMOS transistors. In other words, the gate capacitors of MOS transistors are simply charged or discharged by their driving current or voltage

signals. From the digital simulation point of view, charging and discharging is related to timing processing. On the other hand, the output terminals of CMOS binary gates cannot be connected to a pure MVL node because output terminals of CMOS binary gates would source or sink an error output current ($\sim$100$\mu$A) during switching and in turn, cause errors at this node. Excluding binary gates, only six different types of ports are left; output of current sink, output of current source, input of NCM, output of NCM, input of PCM, and output of PCM.

### 6.1.3 Binary Logic Packages

A VHDL package is a means of sharing portions of code that are used frequently for VHDL modeling, simulation, and synthesis. Based on packages, models as small as a cell or as large as a whole system can then be built. A VHDL package consists of two sections, *package declarations* and *package body*, which can include component declarations, signal declarations, disconnection specification, attribute declarations, type declarations, and subprograms. The package declarations define the interface for the package. The package body specifies the actual behavior of the package [64]. All the items specified in the package are accessible to other design units either by selection or directly [65].

The CMCL library must be able to handle both current-mode MVL and voltage-mode binary logic. As mentioned in Chapter 2 to 4, binary logic circuits are used in a multiple-valued CMCL system to generate control signals for switches or for other purposes through a threshold. As these binary logic signals are directly connected MVL nodes, the aggregate type of MVL signal must also include binary logic state of a node. Instead of defining all the necessary types and functions for the binary logic, existing binary packages should be used. There are many choices for the VHDL packages for binary logic. In order to make the CMCL library as portable as possible, The CMCL library should use the industrial standard VHDL packages. STD and IEEE libraries and packages were selected for this purpose. Extensive use has been made

of these two packages. The libraries *STD.standard*, *STD.texio*, *IEEE.std_login_1164*, and *IEEE.std_logic_texio* are all used. The details of these package can be found in most popular VHDL books [29, 64] and will not be described.

### 6.1.4 CMCL Packages

The CMCL library should contain several VHDL packages in order for easy maintenance and later enhancement. Where applicable, the packages should be designed in a way to avoid compilation of one package causing recompilation of other packages.

i. **CMCL Package:** The basic concept of the CMCL package is similar to that of binary logic packages. A variety of types and constants should be declared first. A bus resolution function should then be defined based on the types. However, multiple-valued CMCL circuits differ from binary logic circuits in that the output signal of a multiple-valued CMCL element is much more sensitive to $W$:$L$ ratios of PMOS or NMOS transistors and loadings. This difference must be considered carefully in the CMCL package. Debug capability should also be included that supports debugging output in ASCII format. Additions should be made to the CMCL library to use this feature. The use of ASSERTions to monitor the user's circuit specification, i.e., improperly connected entities, should also be added. An example of this is checking that current sources and current mirrors are not connected to an output from a binary gate. Finally, basic element and operator functions should be designed based on the CMCL package.

ii. **CIRCUIT Package:** This package implements functions of the CMCL circuit elements. The functions should include every basic CMCL circuit elements: *switches*, *thresholds*, *current sources*, and *current mirrors*. Both N-type and P-type versions should be available. The sign of current should be defined in terms of driving signals. All N-type driving signals have negative value, while

all P-type driving signals have positive value. For instance current flowing into both input and output of an N-type current mirror or an N-type threshold is negative, current flowing out from both input and output of a P-type current mirror or a P-type threshold is positive.

iii. **OPERATOR Package:** This package implements functions of the CMCL MVL operators and binary operators. Operator overload functions should be supported, including vector forms. MVL functions should include *min*, *max*, *tsum*, *ltrl* (*literal* operation), *ltrlb* (*complement of literal* operation), *cyc* (*clockwise cycle* operation), and *cycb* (*counter-clockwise cycle* operation). Binary logic functions should include *and*, *nand*, *or*, *nor*, *xor*, *xnor* and *not*.

### 6.1.5 Library Structure

Considering parameter transfer, logic simulation, and logic synthesis, VHDL simulation models in the CMCL library are divided into three types. Type 1 are simple entities including the basic CMCL building elements. Type 2 are binary entities including basic binary gates. Type 3 are complex entities including MVL operators and macrocells made of simple entities. A diagram showing the organization of the CMCL library is shown in Figure 6.2.

The simple behavioral entities should be developed by using the circuit functions defined in the CIRCUIT package. Re-defining each circuit element as a separate entity is an inevitable step for not only structural modeling of complex entities but also parameter transfer through GENERIC mapping. The parameters can be timing information, transistor size, fan-in, fan-out, and so on. It is possible to create a behavioral model for each element without using the CIRCUIT package. Functions must then be redundantly defined in models.

The complex entities should have two units: structural description and behavioral description. The structural description should be created by using the simple entities

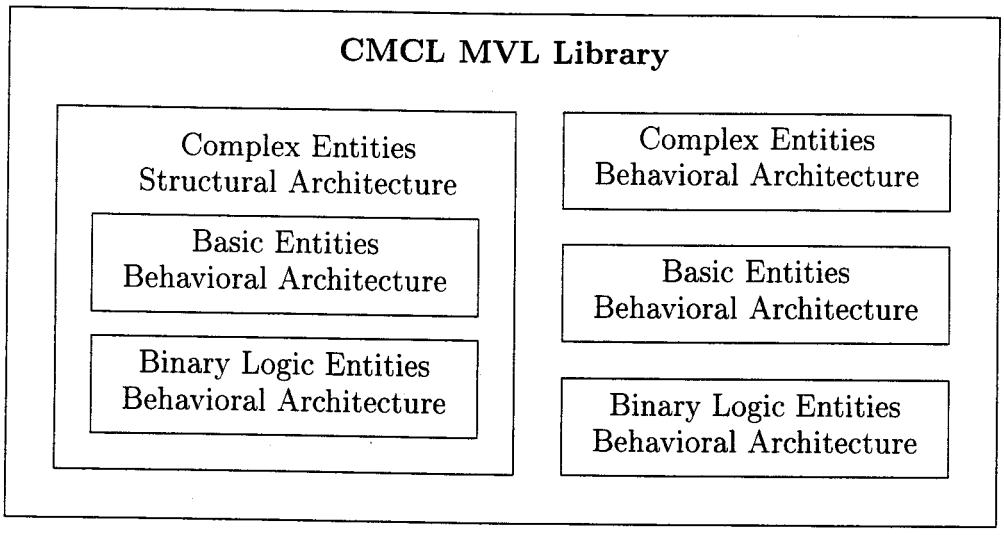



**Figure 6.2** Block diagram of the CMCL library.

and binary gates. The behavioral description should be created by using the functions defined in the OPERATOR package.

### 6.1.6 Library Cells

The entities contained in the CMCL library should be based upon the necessary basic MVL circuits [25] and standard binary logic gates. As well, a number of more complex MVL functions were implemented [42, 17]. A list of cells implemented in the CMCL library is given in Table 6.1.

For the basic CMCL behavioral entities such as N-type threshold and P-type threshold, a generic port should be used for setting the logic level they can detect. Similarly, a generic port should also be used in the current sink and current source entities to set the logic level that they can generate. Although only three types of NCM and PCM entities are shown in the table, current mirror entities with more than 3 inputs can be easily created based on the these models. A list of standard binary entities is also given in Table 6.1. All the entities have two inputs except the inverter. A general purpose test-bench was developed to allow verification of models. As shown in Table 6.1, a set of complex MVL structural and behavioral

**Table 6.1** VHDL models of the CMCL cells.

| Circuit Elements | | Logic Operators | |
|---|---|---|---|
| Cell | Description | Cell | Description |
| ncm1 | 1-Output NCM | and2 | 2-Input AND gate |
| ncm2 | 2-Output NCM | nand2 | 2-Input NAND gate |
| ncm3 | 3-Output NCM | or2 | 2-Input OR gate |
| pcm1 | 1-Output PCM | nor2 | 2-Input NOR gate |
| pcm2 | 2-Output PCM | xor2 | 2-Input EXOR gate |
| pcm3 | 3-Output PCM | xnor2 | 2-Input EXNOR gate |
| nth | N-type Threshold | inv | Inverter |
| pth | P-type Threshold | min2 | 2-Input Min |
| snk | Current Sink | max2 | 2-Input Max |
| src | Current Source | tsum | Truncated Sum |
| nsw | N-type Switch | cycle | Clockwise Cycle |
| psw | P-type Switch | ltrl | Literal |

entities are also included. Most of the complex MVL circuits are implementations of desirable operators. Where appropriate, N-type and P-type versions are available by using different architecture names. An N-type version should use 'n_type' for its architecture name, while a P-type version should use 'p_type' for its architecture name. Other models, for example, multiple-input *min* and *max*, *counter-clockwise cycle*, *complement of literal*, *window literal*, and *complement* can be created easily based on the available models.

## 6.2 Library Design

The second task in the development of the CMCL library was to implement the library specification. This section describes the design of the bus resolution function and the modeling of circuit elements, emphasizing how they work out branch current values of a node through seamless interactions. Type declarations, which are closely related to the bus resolution function, will be explained first.

### 6.2.1 Type Declarations

The most important declarations in the new VHDL package are the new logic value system and the resolved signal. This section therefore focuses on declarations related to these two items. Other declarations are explained in Appendix C.1. Since logic levels are represented in terms of current in the CMCL design, the logic values can simply be defined as REAL type. According to the specification, the aggregate type for the MVL signals must consist of at least current and binary logic. In this implementation, a resolved MVL signal, *node*, is defined in a CMCL package as follows:

```
TYPE unode IS RECORD
    I   : current_vec;
    B   : std_logic;
    W   : weight_vec;
    PDR : real;
    NDR : real;
    EQ  : integer;
END RECORD;
TYPE     unode_vec IS ARRAY (NATURAL RANGE <>) OF unode;
FUNCTION mresolved (s: unode_vec) RETURN unode;
SUBTYPE  node IS mresolved unode;
```

The major subelements of the aggregate type are current vector (`I`), binary logic (`B`) and weight vector (`W`). The size of the current vector and weight vector is equal to the number of signal driving types defined in `node_type`. The `node_type` is defined as:

```
TYPE node_type IS (
    unknown,        -- 0
    ncmin,          -- 1
    pcmin,          -- 2
    ncmout,         -- 3
    nthio,          -- 4
    pcmout,         -- 5
    pthio,          -- 6
    bout;           -- 7
```

```
    swio);          -- 8
TYPE current_vec    IS ARRAY (0 TO typeno-1) OF real;
TYPE weight_vec     IS ARRAY (0 TO typeno-1) OF real;
```

where ncmin represents the input of N-type current mirrors; ncmout is the output of N-type current mirrors, nthio is the input/output of N-type thresholds and current sinks, pcmin is the input of P-type current mirrors, pcmout is the output of P-type current mirrors, pthio is the input/output of P-type thresholds and current sources, swio is the input/output of bidirectional switches, bout is the output of binary gates. The first entry in the current vector corresponds to the unknown driving current, the second entry in the current vector corresponds to the driving current from the input of N-type current mirror, etc. This way a signal can be assigned a value more easily. For example, partial code of an overloaded force function in the CMCL package is shown below:

```
FUNCTION force ( s : REAL; t : node_type ) RETURN node IS
    VARIABLE result : node := initNODE;
BEGIN
    result.I(node_type'POS(t)) := s;
    .....
    RETURN result;
END force;
```

where the value s is assigned to the current_vector of the variable, result, at a position where the t located in the node_type. For example, if t is pthio and s is 20, then I(6) is 20 because node_type'POS(pthio) is 6. It should be noted that a node must be initialized by initNODE, which is defined in UTILITY package as:

```
CONSTANT initCUR  : current_vec := ( 0 TO typeno-1 => 0.0);
CONSTANT initMVL  : mvl_vec     := ( 0 TO typeno-1 => 0.0);
CONSTANT initWGT  : weight_vec  := ( 0 TO typeno-1 => 0.0);
CONSTANT initNODE : unode := (initCUR, initMVL, 'Z', initWGT,
                              0.0, 0.0, 0.0, 0, unknown);
```

Initialization is very important. Otherwise, the default value of a type will be used. For example, without initialization, the initial value of `result.EQ` in the above example will be -2147483648.

The weight vector contains information of the transistor W/L ratio for each driving type. `B` is the binary logic state of a node. `PDR` and `NDR` are total positive and negative driving currents, respectively. As will be described later in this section, `EQ` is used by circuit functions and the bus resolution function to determine if a node reaches equilibrium. The current is declared as a subtype of REAL so as to allow more flexibility in signal definitions.

A global constant, *base_current*, is set to the MVL logic step size in $\mu$A. For example, a *base_current* setting of 20.0 corresponds to an $I_0$ of 20$\mu$A. If *base_current* is set to 1.0, display output indicating logic levels is obtained.

### 6.2.2 Bus Resolution

In addition to the declarations and functions another important issue in a VHDL package is how to handle multiple-driven nodes, namely, the *bus resolution* problem. The resolution function is usually the first function implemented in a package body. The purpose of the resolution function is to resolve the state of a node driven by multiple drivers. In a binary logic package, the bus resolution problem is usually handled by a resolution function in cooperation with a resolution table constant [29].

The bus resolution function must be kept as simple as possible in order to attain fast simulation and easy implementation. Unlike old versions of the bus resolution functions in [66] which try to resolve current distribution of each interconnection, the resolution function used here is designed to execute three tasks: (1) combine all driving signals together, (2) check if a node reaches equilibrium, and (3) determine the binary logic state of a node. The *mresolved* resolution function is only about 150 lines of code, which is about half the size of the older version. Accompanied with collecting the information regarding the network topology of node, the resolved signal

also generates a new event to invoke the processes of MVL entities connected to the node as will be described shortly. The entities then recalculate their input and/or output currents based on the resolved signal. For example, referring to Fig. 6.1($a$), assume $I_{in}$ is logic 0. If the base current, $I_0$, is set to 20$\mu$A, the initial value of $I_2$ is

```
I2.I   = (0.0, 0.0, 0.0, 0.0,-50.0, 0.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 1.0, 0.0, 0.0, 0.0)
I2.B   = 0
I2.PDR = 0.0
I2.NDR = -50.0
I2.EQ  = 1
```

I2.B is 0 because the sum of I2.PDR and I2.NDR is negative. Since the sum of the elements of I2.I is not zero, i.e., this node has not reached equilibrium, the resolution function increments I2.EQ, which invokes the P-type current mirror and the N-type current source. In response to the new event, both entities recalculate their output currents. The result is that the output of the P-type current mirror remains 0$\mu$A and the output of the N-type current source is reduced to 0$\mu$A. Therefore, $I_2$ becomes

```
I2.I   = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0)
I2.B   = 0
I2.PDR = 0.0
I2.NDR = -50.0
I2.EQ  = 0
```

and I2.EQ is reset to 0 by the resolution function because the sum of the elements of I2.I is zero. As will be seen below, a simple resolution function also greatly simplifies the bi-directional switch models. Although the resolution function does not provide accurate analog results as circuit simulators do, it can quickly and reliably verify MVL circuits based on the self-restored architecture.

### 6.2.3 Circuit Functions

The circuit functions are implemented in a CIRCUIT package, including *nth_proc*, *pth_proc*, *ncmin_proc*, *ncmout_proc*, *pcmin_proc*, *pcmout_proc*, *nsw_proc*, and *psw_proc*. As can be seen from the list, each circuit function corresponds to a terminal type mentioned in 6.1.2. MVL entities are then implemented based on these circuit functions. The circuit functions work tightly with the resolution function to determine currents in interconnections. Again, referring to Figure 6.1($a$), assume $I_{in}$ is logic 1. If the base current, $I_0$, is set to 20$\mu$A, the initial value of $I_2$ is

```
I2.I   = (0.0, 0.0, 0.0, 0.0,-50.0,20.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 1.0, 0.0, 0.0, 0.0)
I2.B   = 0
I2.PDR =  20.0
I2.NDR = -50.0
I2.EQ  = 1
```

`I2.B` is still 0 because the sum of `I2.PDR` and `I2.NDR` is negative. Since the sum of the elements of `I2.I` is not zero, i.e., this node has not reached equilibrium, the resolution function increments `I2.EQ`, which invokes the *pcmout_proc* in the P-type current mirror and the *nth_proc* in the N-type current source. In response to the event, both entities recalculate their output currents. The result is that the output of the P-type current mirror remains 20$\mu$A and the output of the N-type current source is reduced from 50$\mu$A to 20$\mu$A. The resolution function resets `I2.EQ` to 0 because the sum of the elements of `I2.I` is zero. Therefore, $I_2$ becomes

```
I2.I   = (0.0, 0.0, 0.0, 0.0,-20.0,20.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 1.0, 0.0, 0.0, 0.0)
I2.B   = 0
I2.PDR =  20.0
I2.NDR = -50.0
I2.EQ  = 0
```

Now assume $I_{in}$ is 60$\mu$A. The initial value of $I_2$ is

```
I2.I   = (0.0, 0.0, 0.0, 0.0,-50.0,60.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 1.0, 0.0, 0.0, 0.0)
I2.B   = 1
I2.PDR =  60.0
I2.NDR = -50.0
I2.EQ  = 1
```

I2.B is changed to 1 by the resolution function because the sum of I2.PDR and I2.NDR is positive. Similarly, since the sum of the elements of I2.I is not zero, the resolution function increments I2.EQ, which invokes the *pcmout_proc* in the P-type current mirror and the *nth_proc* in the N-type current source to recalculate their output currents. The result is that the output of the P-type current mirror is reduced from 60$\mu$A to 50$\mu$A and the output of the N-type current source remains 50$\mu$A. Therefore, $I_2$ becomes

```
I2.I   = (0.0, 0.0, 0.0, 0.0,-50.0,50.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 1.0, 0.0, 0.0, 0.0)
I2.B   = 1
I2.PDR =  60.0
I2.NDR = -50.0
I2.EQ  = 0
```

I2.EQ is reset to 0 by the resolution function because the sum of the elements of I2.I is zero again.

Bi-directional switch models are different from other circuit models. The simulation algorithms for circuits involving bi-directional switches are known as *relaxation* algorithms. An algorithm which models a bi-directional switch that can affect only the circuits directly connected to it is known as a *local relaxation* algorithm. A local relaxation algorithm implemented in VHDL was proposed by Coelho et al. and successfully used for VHDL switch-level simulation [29]. Unfortunately this algorithm

does not apply to CMCL design. As mentioned in Section 6.1, bi-directional switch ports affect the logic value of a node in a very particular way. When a switch is turned on each switch INOUT port must transfer the the circuit topology from one side to the other side so that a new network is formed. When a switch is turned off the circuit connected with either side should be evaluated individually. This switch model, which can be modeled for VMCL binary gates very easily, is quite complex as a multiple-valued CMCL element. In fact, this was the most complex model in the older version of the CMCL library. However, thanks to the simplified node type and bus resolution function, transferring circuit information from one side to the other side becomes much easier.

Referring to Figure 6.1($b$), assume the switch is initially OFF and and $I_{in}$ is logic one, i.e., 20$\mu$A. Since the switch is OFF, the *nsw_proc* in the switch model will simply assign `initNODE` to its two INOUT ports. Therefore, $I_2$ and $I_3$ eventually become

```
I2.I   = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0)
I2.B   = 1
I2.PDR = 20.0
I2.NDR =  0.0
I2.EQ  = 0

I3.I   = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
I3.W   = (0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0)
I3.B   = 0
I3.PDR =   0.0
I3.NDR = -50.0
I3.EQ  = 0
```

When the switch is turned on, the switch model generates a new event by setting `W(node_type)'POS(swio)` to 1 to force recalculation of current distribution. Therefore, the resolution function changes $I_2$ and $I_3$ to

```
I2.I   = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

```
I2.W   = (0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0)
I2.B   = 1
I2.PDR = 20.0
I2.NDR =  0.0
I2.EQ  = 0

I3.I   = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
I3.W   = (0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0)
I3.B   = 0
I3.PDR =   0.0
I3.NDR = -50.0
I3.EQ  = 0
```

In response to the new event, the *pcmout_proc* in the P-type current mirror and the *nth_proc* in the N-type current source also initialize their output currents. As a result, new resolved results of $I_2$ and $I_3$ are

```
I2.I   = (0.0, 0.0, 0.0, 0.0, 0.0,20.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0)
I2.B   = 1
I2.PDR = 20.0
I2.NDR =  0.0
I2.EQ  = 0

I3.I   = (0.0, 0.0, 0.0, 0.0,-50.0, 0.0, 0.0, 0.0, 0.0)
I3.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 0.0, 0.0, 0.0, 1.0)
I3.B   = 0
I3.PDR =   0.0
I3.NDR = -50.0
I3.EQ  = 0
```

In response to the new event, the switch model transfer the resolved value from one end to the other to force a new event on both $I_2$ and $I_3$. The resolution function generate a new results as

```
I2.I   = (0.0, 0.0, 0.0, 0.0,-50.0,20.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 1.0, 0.0, 0.0, 1.0)
```

```
I2.B   = 0
I2.PDR =  20.0
I2.NDR = -50.0
I2.EQ  = 0

I3.I   = (0.0, 0.0, 0.0, 0.0,-50.0,20.0, 0.0, 0.0, 0.0)
I3.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 1.0, 0.0, 0.0, 1.0)
I3.B   = 0
I3.PDR =  20.0
I3.NDR = -50.0
I3.EQ  = 0
```

I2.B is changed to 0 by the resolution function and $I_2$ and $I_3$ are the same. Similar to the situation described before with reference to to Figure 6.1($b$), the resolution function increments I2.EQ and I3.EQ since $I_2$ and $I_3$ have not reached equilibrium, which invokes the *pcmout_proc* in the P-type current mirror and the *nth_proc* in the N-type current source to recalculate their output currents. The result is that the output of the N-type current source changes its output from $-50\mu$A to $20\mu$A and the output of the P-type current mirror remains $20\mu$A. Therefore, $I_2$ remain unchanged and $I_3$ becomes

```
I3.I   = (0.0, 0.0, 0.0, 0.0,-20.0,20.0, 0.0, 0.0, 0.0)
I3.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 1.0, 0.0, 0.0, 1.0)
I3.B   = 0
I3.PDR =  20.0
I3.NDR = -50.0
I3.EQ  = 0
```

The switch then transfers the new resolved $I_3$ to $I_2$ so that $I_2$ also becomes

```
I2.I   = (0.0, 0.0, 0.0, 0.0,-20.0,20.0, 0.0, 0.0, 0.0)
I2.W   = (0.0, 0.0, 0.0, 0.0,  1.0, 1.0, 0.0, 0.0, 1.0)
I2.B   = 0
I2.PDR =  20.0
I2.NDR = -50.0
I2.EQ  = 0
```

The new resolved $I_2$ is transfered to $I_3$ by the switch. Since $I_2$ and $I_3$ are the same, simulation stops until the next new event.

### 6.2.4 Library Cell Modeling

All of the basic MVL elements discussed in Chapter 2 are modeled as library cells by defining their functions in the CIRCUIT package and then creating behavioral descriptions by using the functions. Although only single-output, two-output and three-output current mirror models were created, these models can easily be expanded to handle any number of outputs. A cell model using GENERIC for parameter transfer is shown in Figure 6.3. It should be noted each model must start with a LIBRARY clause and a USE clause so that the CMCL package and other packages are visible to the models.

As mentioned previously one of the reasons to re-define each circuit element as a cell is for parameter transfer through GENERIC mapping. In this model `wgt` in the GENERIC port is used to specify the width ratio $W_{out}$:$W_{in}$ of the output transistor and the input transistor of the N-type current mirror, `ncm1`. For example, `wgt<=2` means the width of the output transistor is two times larger than that of the input transistor of the same current mirror. Every port is a bidirectional INOUT port since all input ports must be capable of driving the output connection when necessary and all output ports must also be capable of being driven when necessary. In the above example, all of the ports are defined as a bidirectional port with an initial value of `initNODE`. As mentioned in Section 6.2.1 initialization is very important. Otherwise, the default value of REAL number is assigned to each branch current.

Based on the basic VHDL element models, the structural models for self-restored CMCL MVL circuits as well as operators such as *min*, *tsum*, *literal*, and *cycle* can be easily created. For the behavioral operator models the modeling approach is the same as that of CMCL elements. That is, the function is declared in the OPERATOR package and then the behavioral description was created by using the OPERATOR

```
LIBRARY CCMVL; USE CCMVL.cmcl.ALL, CCMVL.circuit.ALL;
LIBRARY IEEE; USE IEEE.std_logic_1164.ALL;

ENTITY ncm IS
  GENERIC ( win, wout : REAL := 1.0 );
  -- it's very important to set the iniNODE to all ports.
  PORT ( inA : INOUT node := initNODE; outA : INOUT node := initNODE);
END ncm;

ARCHITECTURE behavioral OF ncm IS
  SIGNAL x, y : REAL := 0.0;
BEGIN
  PROCESS(inA, outA)
    VARIABLE inA_delay, outA_delay : node := initNODE;
    VARIABLE last_inA_change, last_outA_change : TIME := 0 NS;
    VARIABLE x_int, x_ext, y_int, y_ext : REAL := 0.0;
    VARIABLE total_in, total_pos, total_neg : REAL := 0.0;
    VARIABLE equ : INTEGER := 0;
    VARIABLE bin_state : std_logic := 'Z';
    VARIABLE state_not_determined : boolean := false;
  BEGIN
    -- input event : recalculate output no matter of output event.
    IF inA'EVENT THEN
      ncmin_proc ( inA, outA, win, wout, x, y, inA_delay, outA_delay,
                   last_inA_change, last_outA_change,
                   x_int, x_ext, y_int, y_ext,
    total_in, total_pos, total_neg,
                   equ, bin_state, state_not_determined );
    -- output event : check if total current is greater than 0.
    -- if greater than 0 and no pcmin, then reduce current output.
    ELSE
      ncmout_proc ( inA, outA, wout, x, y, inA_delay, outA_delay,
                    last_inA_change, last_outA_change,
                    x_int, x_ext, y_int, y_ext,
     total_in, total_pos, total_neg,
                   equ, bin_state, state_not_determined );
    END IF;
  END PROCESS;
END behavioral;
```

**Figure 6.3** VHDL cell model of the single-output NCM.

```
LIBRARY CCMVL, IEEE;
USE CCMVL.cmcl.ALL,  CCMVL.operator.ALL, IEEE.std_logic_1164.ALL;

ENTITY and2 IS
  port ( inA, inB : INOUT node; outA : INOUT node := initNODE);
END and2;

ARCHITECTURE behavioral OF and2 IS
BEGIN
  outA <= CCMVL.operator."and"(inA, inB);
END behavioral;
```

**Figure 6.4** VHDL model of the binary 2-input *and* gate.

package. Structural models for the CMCL MVL operators are shown in Appendix D.2. To bind the components and models, a configuration file must be created for each architecture.

Cell models can be separated to a plurality of files or merged into a package. In either case each file must begin with the LIBRARY and the USE clauses to access the CMCL package and other packages. In this implementation all the simple MVL entities are declared and implemented in the CKT_ENT and CKT_ARCH files, respectively (refer to Appendix D.1); all the complex entities are declared and implemented in the MVL_ENT and MVL_ARCH files, respectively (refer to Appendix D.2); and all the binary entities are declared in the BIN_ENT and BIN_ARCH files, respectively (refer to Appendix D.3).

Behavioral models of binary gates were created by using the *std_logic_1164* package. An example of these behavioral models for binary gates is shown in Figure 6.4. It can be seen from the example that the ports for binary gates are also of the type of node. This avoids type conversion when a binary input or output is connected to a MVL element. Also the overloaded and function is double-quoted because 'and' is a reserved word in VHDL.

### 6.2.5 Simulation Models

To run VHDL simulation the first thing to do is to create simulation models. A simulation model does two things. First, it instantiates the model to be verified. Secondly, it contains the input stimulus. VHDL can be used effectively as a stimulus language. There are two approaches to create a VHDL simulation model; the *test bench approach* and the *simulation model approach* [29].

The test bench approach has a separate input stimulus entity and model instantiation entity. This stimulus entity has one OUT port and in its architecture body the test vectors to be applied to the OUT port are declared. The force values and time delay can be passed to the test vectors through the GENERIC ports. An instantiation entity is then created to connect the stimulus entity to the circuit under test. When this test bench is simulated, the stimulus entity provides stimulus to the circuit, which in response to these stimuli, sends results to the output until exhaustion of all stimuli. It is also able to include assertions in the vectors to provide automatic test for proper circuit operation.

The simulation model approach, on the other hand, has the circuit under test and the test vectors combined into one VHDL entity. When the simulation model is simulated stimulus processes defined in the model provide stimuli to the component, which sends results back to output ports in responding to this stimulus.

Comparing the above two approaches, the test bench approach is better than the simulation model approach for verifying a number of MVL circuits generated by the WMS program because the input stimulus is used repeatedly. However, during verification of the CMCL library it is frequently necessary to change the input stimulus and observe simulation results. Therefore, all the VHDL simulations were carried out using the simulation model approach. Yet another way to give input stimulus is to use a VHDL simulator vendors' proprietary stimulus commands. However, significant differences exist between VHDL simulators from various vendors. Considering the portability of simulation models, the proprietary stimulus commands were not used.

## 6.3 Examples

While being developed, the CMCL library was tested on a number of circuits to verify correct operation of the models. This section describes two of the test circuits. Although only Leapfrog and NC-Sim[1] simulation results will be discussed, the CMCL library was also verified by importing it into Synopsys VSS[2] and performing the simulation. All of the VHDL simulation results were confirmed.

### 6.3.1 Quaternary Full Adder

The first example is to demonstrate how a circuit can be described in VHDL by using the CMCL library. A circuit realization implementing a quaternary full adder (QFA) function [50] is shown in Fig. 6.5. The circuit uses a number of entities in the design, namely, N-type and P-type current mirrors, N-type threshold, N-type switch, and current sink. $A_{in}$ and $B_{in}$ are the two radix 4 (values <0123>) MVL signals to be added while $C_{in}$ is the carry-in signal from a previous stage (value <01>). $I_{net}$ is the sum of the inputs $A_{in}$, $B_{in}$ and $C_{in}$. $I_1$ and $I_2$ are duplicated current of $I_{net}$. The actual current of $I_2$ is subject to the threshold, Xnth . If $I_{net}$ is greater than $3.5I_0$ (i.e., 3.5 times the base current), the threshold generates a binary high signal and limits $I_1$ to $3.5I_0$; otherwise the threshold generates a binary low signal. The binary logic signal controls switches Xnsw1 and Xnsw2 through Xinv1 and Xinv2. When $I_{net}$ is more than $3.5I_0$, Xnsw1 and Xnsw2 are ON which causes $S_{out} = I_1 - 4I_5 = I_{net} - 4I_0$ and $C_{out} = I_0$. When $I_{net}$ is less than $3.5I_0$, $S_1$ and $S_2$ are OFF which causes $S_{out} = I_1 = I_{net}$ and $C_{out} = 0$.

The VHDL description for the QFA is shown in Fig. 6.6. It should be noted that multiple driving signals (Ain, Bin, and Cin) are to be delivered to the *modsum* through a single input (Inet). The VHDL code was compiled with the CMCL library.

[1]Leapfrog and NC-Sim are trademarks of Cadence Design Systems, Inc.
[2]VSS is a trademark of Synopsys, Inc.

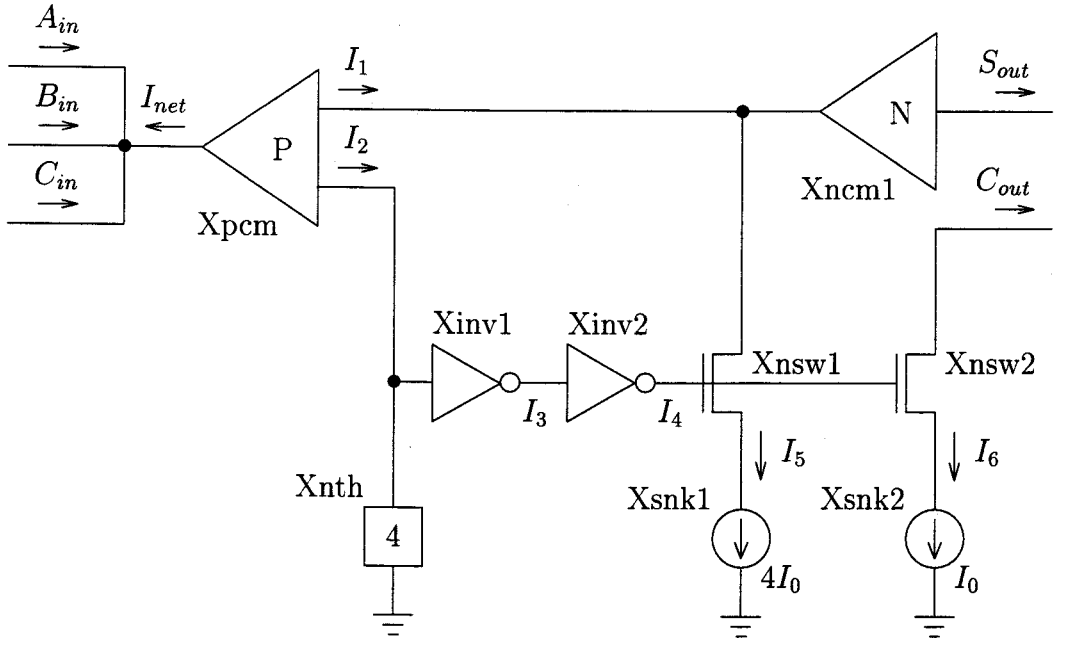



**Figure 6.5** Circuit realization of a QFA function.

NC-Sim was then used to simulate the circuit. A part of the SignalScan[3] output is shown in Figure 6.7. As can be seen from the simulation results, the QFA functions as intended. Internal current are as expected. HSPICE transient analysis circuit simulation was also carried out to verify the function of the implemented QFA as shown in Figure 6.8. Ideal current sources were used to generate the input currents. The outputs of the QFA are connected to a load so that the transient response of the QFA could be measured. The HSPICE simulation output agrees with the NC-Sim simulation.

### 6.3.2 A Matching and Squaring Circuit

Fig. 6.9 is one segment of a matched filter [66]. This example is chosen since functional verification at the circuit level is very time consuming. The designers required an alternative for shortening the design cycle. The circuit consists of 32 bit-slice blocks and an analog processing block. As shown in Figure 6.10, each bit-

[3]SignalScan is a trademark of Cadence Design Systems, Inc.

```
LIBRARY CCMVL; USE CCMVL.cmcl.ALL, CCMVL.operator.ALL;
LIBRARY IEEE; USE IEEE.std_logic_1164.ALL;
ENTITY qfa IS
  PORT ( Inet, Sout, Cout : INOUT node := initNODE );
END qfa;
ARCHITECTURE structural OF qfa IS
SIGNAL i1, i2, i3, i4, i5, i6 : node := initNODE;
COMPONENT nth
  GENERIC ( wgt : REAL );
  PORT ( inoutA : INOUT node);
END COMPONENT;
COMPONENT ncm
  PORT ( inA : INOUT node; outA : INOUT node);
END COMPONENT;
COMPONENT pcm
  PORT ( inA : INOUT node; outA : INOUT node);
END COMPONENT;
COMPONENT pcm2
  PORT ( inA : INOUT node; outA, outB : INOUT node);
END COMPONENT;
COMPONENT nsw
  PORT ( ctrl : IN node; inoutA, inoutB : INOUT node );
END COMPONENT;
COMPONENT inv
  PORT ( inA : IN node; outA : OUT node);
END COMPONENT;
BEGIN
  XPCM  : pcm2  PORT MAP (Inet, i1, i2);
  XNTH  : nth   GENERIC MAP (3.5) PORT MAP (i2);
  XINV1 : inv   PORT MAP (i2, i3);
  XINV2 : inv   PORT MAP (i3, i4);
  XNSW1 : nsw   PORT MAP (i4, i1, i5);
  XSNK1 : nth   GENERIC MAP (4.0) PORT MAP (i5);
  XNCM  : ncm   PORT MAP (i1, Sout);
  XNSW2 : nsw   PORT MAP (i4, i6, Cout);
  XSNK2 : nth   GENERIC MAP (1.0) PORT MAP (i6);
END structural;
```

**Figure 6.6** VHDL code of the QFA circuit shown in Fig. 6.5.

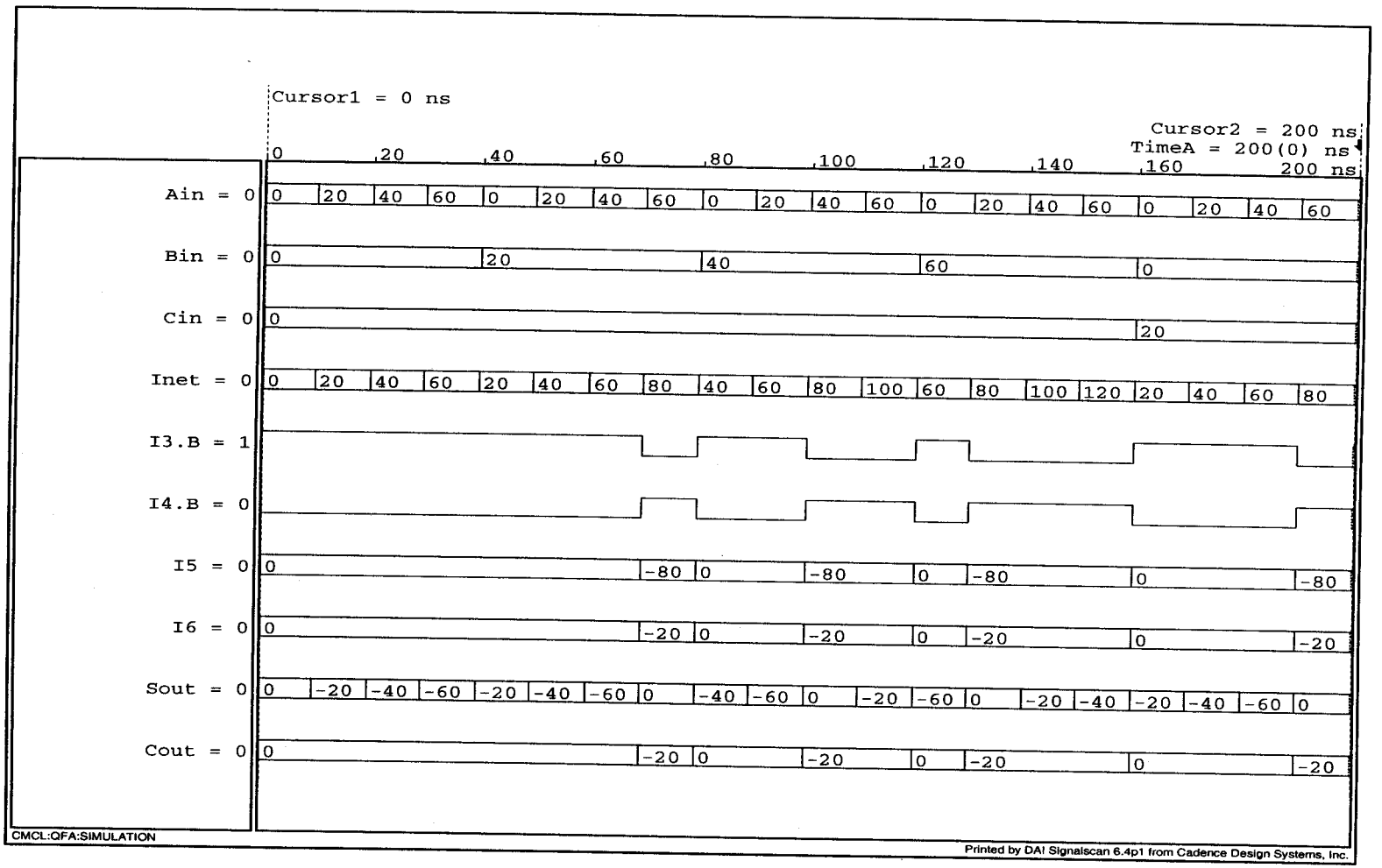



**Figure 6.7** VHDL simulation of the QFA circuit using Cadence NC-Sim.

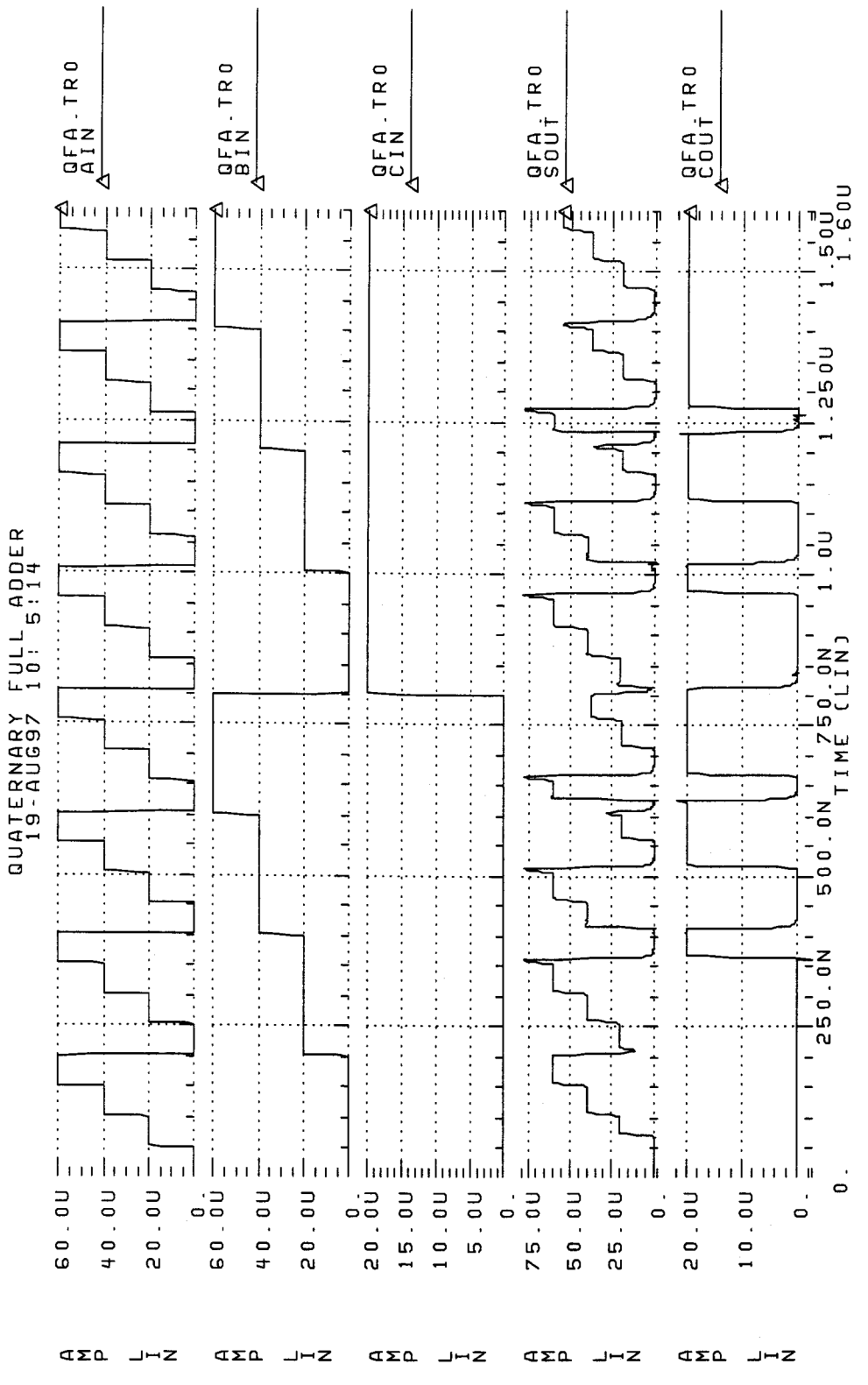



**Figure 6.8** HSPICE simulation of the QFA circuit.

slice block further consists of three D-type flip-flops (DFFs) for latching input signals and codes as well as two match processing blocks comparing input signals and codes. DFF0 and DFF1 are signal registers while DFF2 is a code register. The two matching blocks compare outputs from DFFs and generate current outputs when they match. A circuit implementation of the matching blocks is shown in Figure 6.11. The match processing block, MATCH_PROC0, compares Sig0_In and Code_In and sinks $5\mu$A when the states match, and the other match processing block, MATCH_PROC1, compares Sig1_In and Code_In and sinks $10\mu$A when the states match. The amount of current sunk by a bit-slice block is summarized in Figure 6.12. The current outputs from the 32 bit-slice blocks are connected together so that the total current input to Imatch from the 32 bit-slice blocks ranges from $0\mu$A to $32 \times (5\mu\text{A} + 10\mu\text{A}) = 480\mu$A.

The analog processing block contains an absolute differencing block (not shown) and a current squaring block (not shown) to perform an operation described in equation form by $|\text{Imatch} - 240\mu\text{A}|^2$. In order to provide the absolute value of Imatch $- 240\mu$A, two current differencing blocks are used. The circuit used to implement the current squaring block was taken from [67]. The output current is sourced out of the current squaring block and follows the relationship

$$I_{out} = \frac{I_{in1}^2}{8I_{in2}} \quad \text{iff} \quad |I_{in1}| < 4I_{in2} \quad \text{and} \quad I_{in2} > 0 \tag{6.1}$$

The input to $I_{in1}$ is provided by the output of the current differencing blocks, and will therefore have a maximum value of $480\mu\text{A} - 240\mu\text{A} = 240\mu$A. Thus $I_{in2}$ must be at least $240\mu\text{A}/4 = 60\mu$A.

The VHDL description of the matching and squaring circuit was compiled using Leapfrog with the CMCL library. Leapfrog was then used to simulate the circuit. Fig. 6.13 shows partial Leapfrog simulation output. Transient analysis was also carried out by using Spectre[4] to verify the function of the implemented circuit. Fig. 6.14

[4]Spectre is a trademark of Cadence Design Systems, Inc.

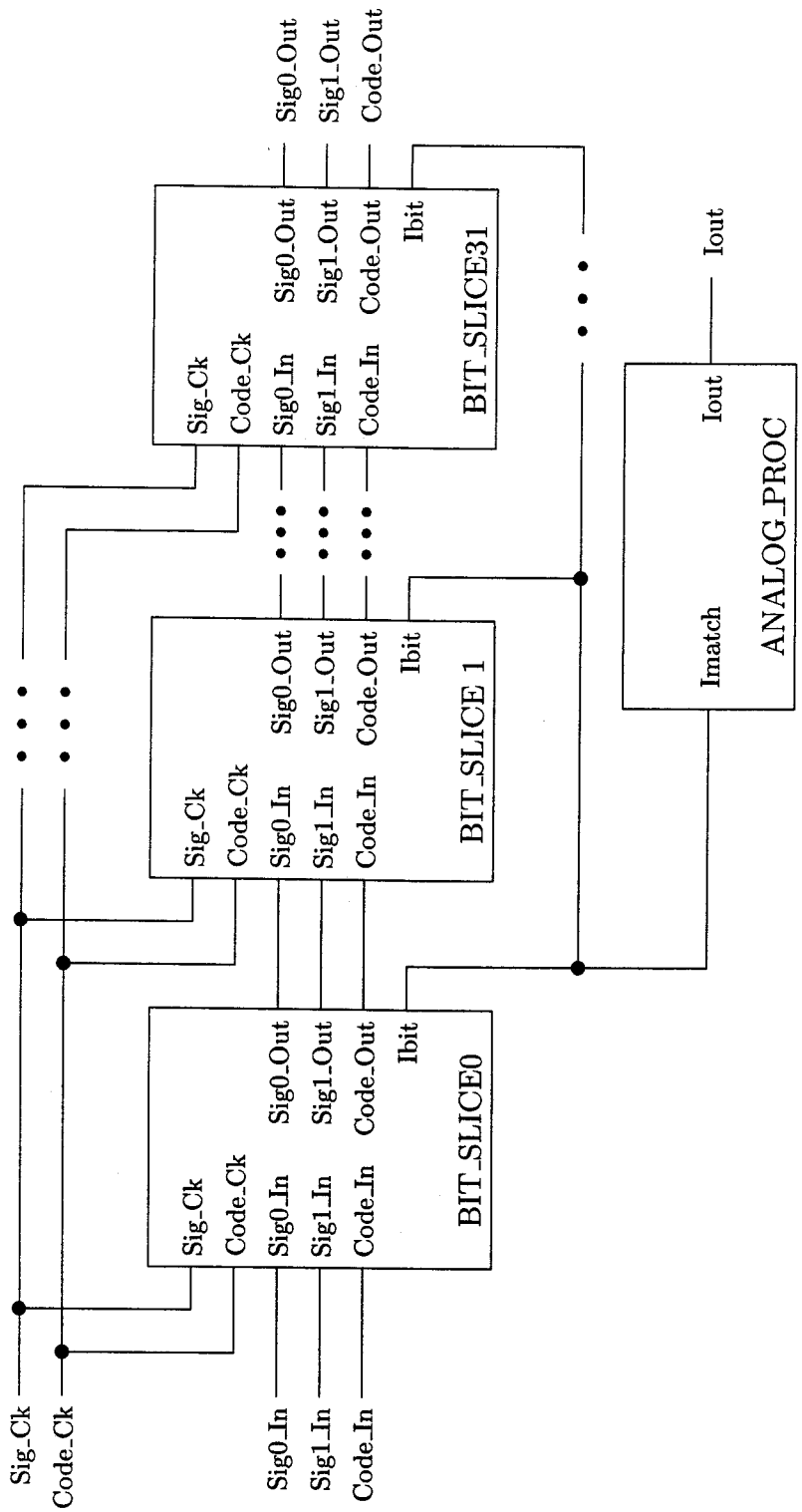



**Figure 6.9** Block diagram of one segment of a match filter.

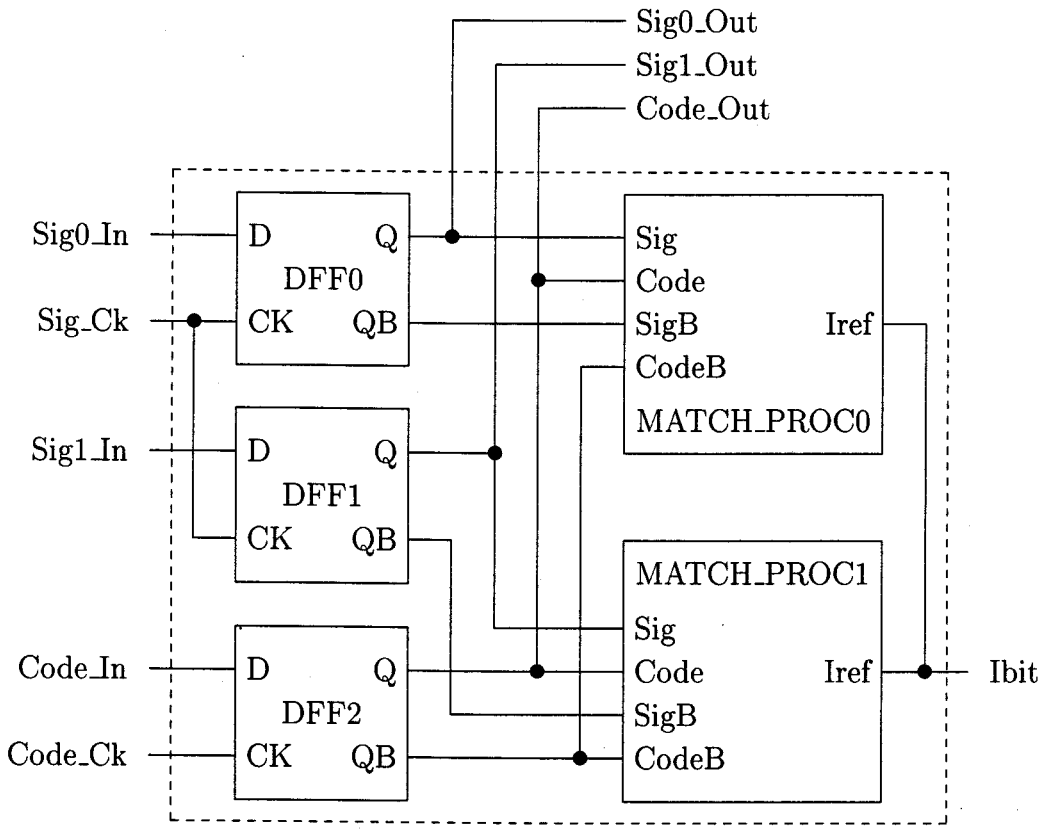



**Figure 6.10** Block diagram of the bit-slice block.

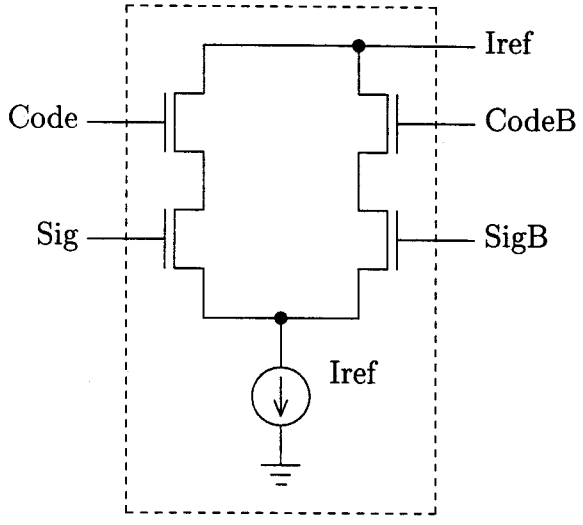



**Figure 6.11** Circuit diagram of the matching block.

| Code | Sig1 | Sig0 | Ibit |
|---|---|---|---|
| 0 | 0 | 0 | 15$\mu$A |
| 0 | 0 | 1 | 10$\mu$A |
| 0 | 1 | 0 | 5$\mu$A |
| 0 | 1 | 1 | 0$\mu$A |
| 1 | 0 | 0 | 0$\mu$A |
| 1 | 0 | 1 | 5$\mu$A |
| 1 | 1 | 0 | 10$\mu$A |
| 1 | 1 | 1 | 15$\mu$A |

**Figure 6.12** Current sunk by a bit-slice block.

shows the Spectre simulation result from 0.0$\mu$s to 100$\mu$s. The Code Register (DFF2) was loaded with all ones. The two Signal Register (DFF0 and DFF1) inputs were tied together and was initially loaded with zeros so that the number of matches starts at 0 which results in a maximum output of

$$I_{out} = \frac{|0\mu A - 240\mu A|^2}{8 \times 60\mu A} = 120\mu A. \tag{6.2}$$

The Signal register inputs were then supplied with alternating runs of 32 ones a nd 32 zeros. The number of matches increases to 16 which result in a minimum output of

$$I_{out} = \frac{|240\mu A - 240\mu A|^2}{8 \times 60\mu A} = 0\mu A \tag{6.3}$$

and then the number of matches increases to 32 resulting in the maximum output

$$I_{out} = \frac{|480\mu A - 240\mu A|^2}{8 \times 60\mu A} = 120\mu A. \tag{6.4}$$

The number of matches then ramps down to 16 and to 0 to finish the test.

In Fig. 6.13, Code_In is the voltage supplied to Code_In of BIT_SLICE0, Sig1_In and Sig0_In are the voltages supplied to Sig1_In and Sig0_In of BIT_SLICE0, Code_Out is Code_Out from BIT_SLICE31, Sig1_Out and Sig0_Out are Sig1_Out and Sig0_Out from BIT_SLICE31, and Imatch is the sum of total current outputs from the 32 bit-slice blocks, and Iout is the output current waveform from the analog processing block. Similarly, in Fig. 6.14, /Code_In is the voltage supplied to Code_In of BIT_SLICE0, /Sig1_In and /Sig0_In are the voltages supplied to Sig1_In and Sig0_In of BIT_SLICE0, /Code<31> is Code_Out from BIT_SLICE31, Sig1<31> is Sig1_Out from BIT_SLICE31, and /R3/PLUS is the output current waveform, Iout. By comparing Iout in Fig. 6.13 and /R3/PLUS in Fig. 6.14 around 65$\mu$s, it can be seen that the VHDL simulation functionally matches the Spectre simulation. The peak value of /R3/PLUS is approximately 120$\mu$A which is the same as the peak value of Iout in

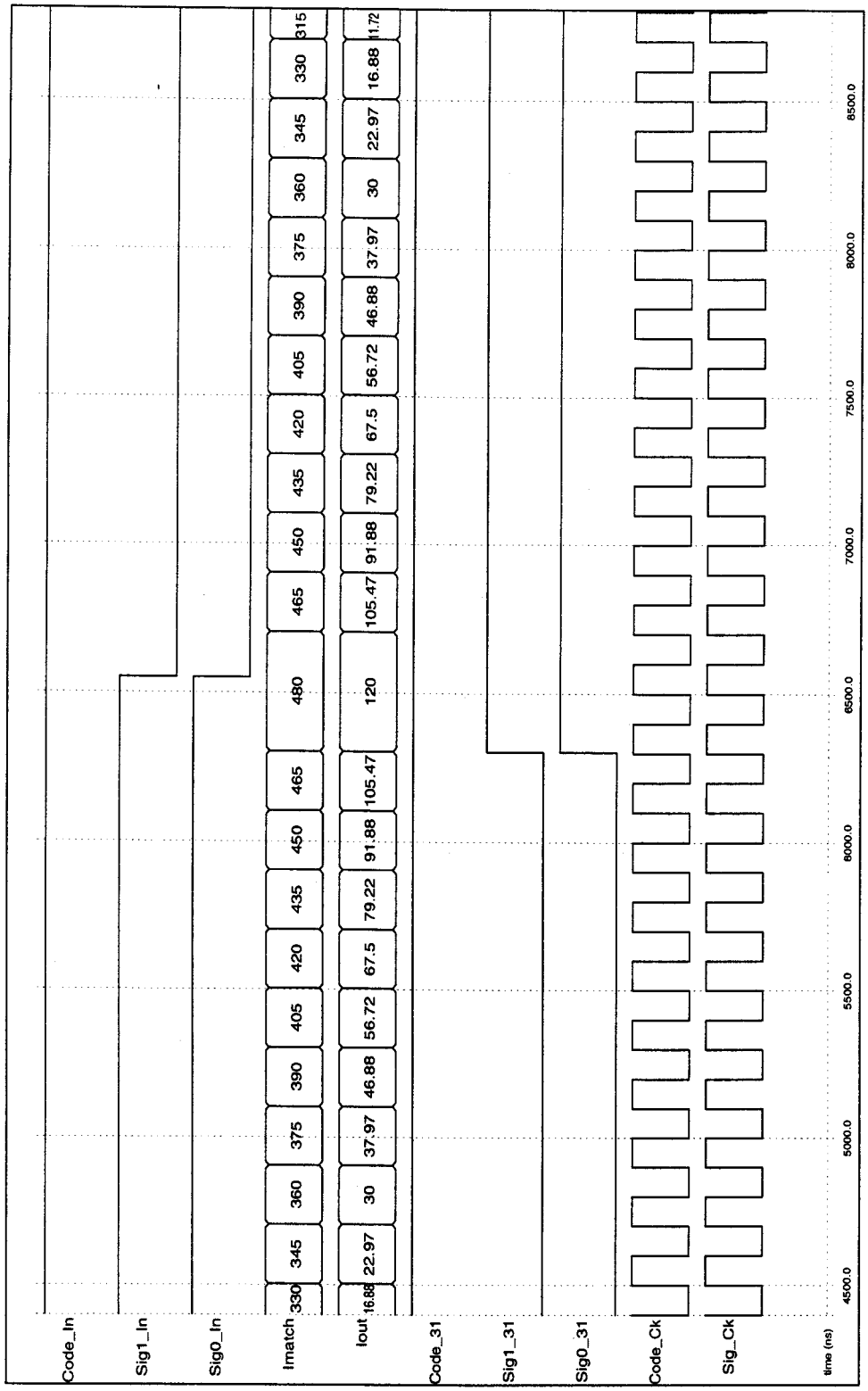



**Figure 6.13** VHDL simulation of the matching and squaring circuit.

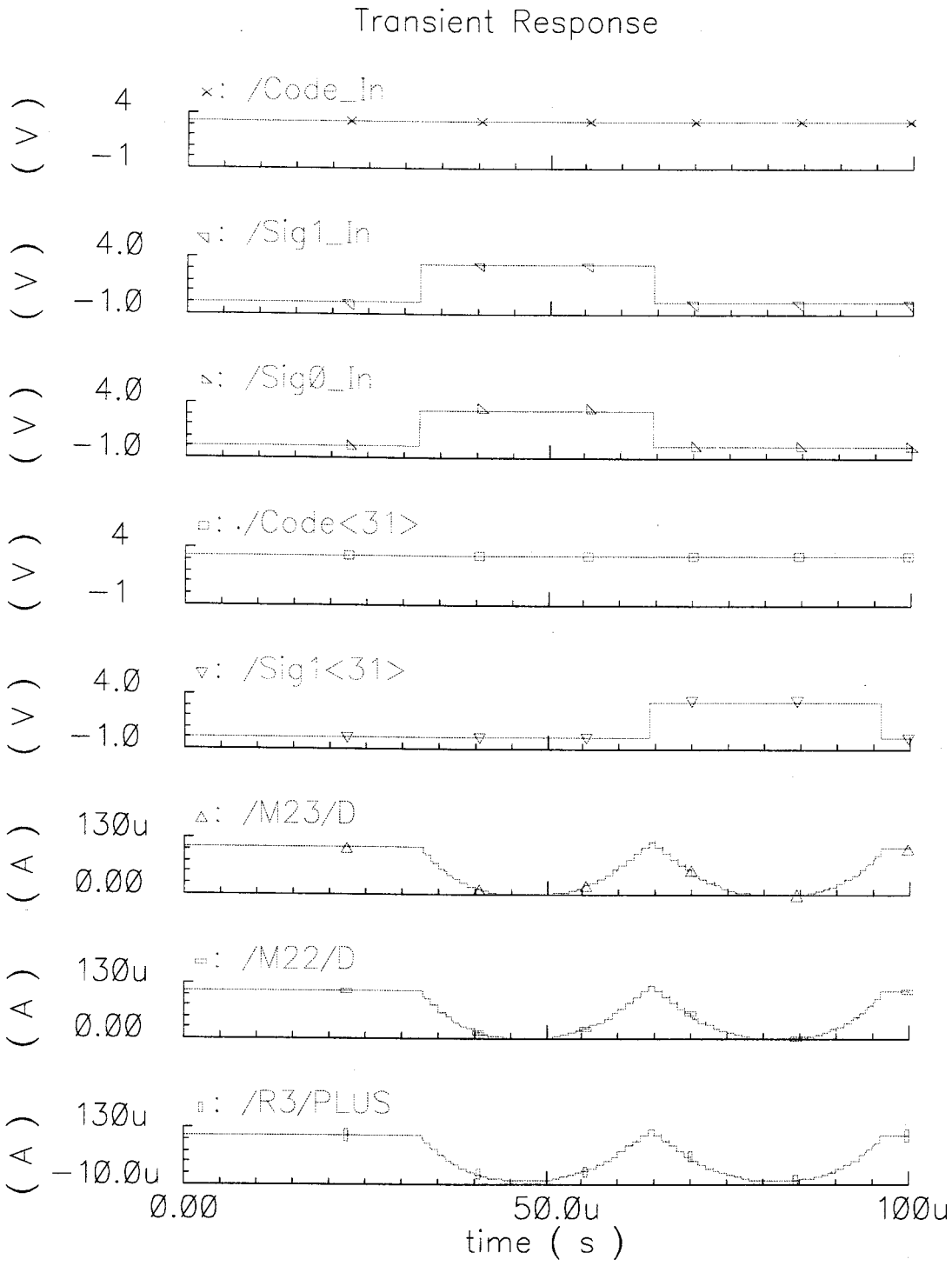



**Figure 6.14** Circuit simulation of the matching and squaring circuit.

the VHDL simulation. More importantly, the VHDL simulation is much faster than the Spectre simulation. The Spectre simulation takes about two and half hours on an UltraSparc machine with 128MB memory, while the VHDL simulation takes only 15 seconds on the same machine.

## 6.4 Summary

A new library for VHDL simulation of CMCL MVL design is discussed in this chapter. The new library allows faster simulation of a broader range of CMCL circuits as a result of the new bus resolution function and the new CMCL circuit element functions. The library has basic MVL entities (behavioral), complex MVL entities (behavioral and structural) as well as standard binary logic gates. A bus resolution function working cooperatively with the basic MVL entities allow MVL logic levels (currents) in individual connections to be output.

Design examples of a quaternary full adder and partial circuit of a matched filter are presented along with both VHDL and circuit simulation results. The design examples verify that the CMCL library allows the VHDL simulation of current-mode CMOS logic using Leapfrog. Spectre was used to confirm that the VHDL simulations were correct. The circuits were also verified by importing the CMCL library into VSS and performing the simulation. All of the VHDL simulation results were confirmed.

The library is being improved on a continuing basis. The next step is to define the CMCL library to incorporate better debugging and performance. As well, extensive cell characterization must take place to ensure the cell-delay figures for the models are correct.

# 7. Results

A self-restored CMOS MVL design architecture and its variants were proposed in Chapters 3 and 4. An approach of using a binary logic synthesizer for MVL synthesis was also discussed in Chapter 5. A computer program, WMS, was developed to implement the approach. The program works together with the Synopsys Design Compiler to generate self-restored MVL circuits in both VHDL and Verilog formats. The generated VHDL files can then be verified by using a VHDL simulator with the CMCL library. The Verilog files can be imported to the Cadence environment for Spectre simulation or other purposes. Based on the WMS program, another computer program, M2B, was also developed to derive an equivalent binary logic circuit for a given MVL function.

This chapter focuses on comparison of self-restored MVL design with conventional operator-based MVL design and with binary logic design. The results will be discussed from three aspects: circuit size, time delay, and power dissipation. Section 7.1 presents comparison between the self-restored MVL architecture and the conventional operator-based MVL design as well as the comparing procedures. Section 7.2 presents comparison of the self-restored MVL architecture with the binary logic design.

## 7.1 Comparison with Operator-Based MVL Designs

The self-restored architecture is advantageous over the conventional operator-based MVL design schemes in many aspects in addition to self-restoration. With

the self-restored architecture, a binary logic synthesizer can be used for synthesis of MVL functions. It is also believed that the self-restored architecture can result in smaller circuit size because of using *up literal* in place of *literal* and using binary *and/or* operators in place of *min/max* operators.

Many examples were manually designed and compared with operators-based design prior to the development of the WMS program. In most of the cases MVL circuits implemented with the self-restored architecture are smaller than the operator-based MVL design. Figures 7.1 to 7.3 are examples taken from [59]. Figure 7.1 shows three examples for comparing with operator-based design using the HAMLET program. Figure 7.2 shows three examples for comparing with operator-based design using the Set C operators as described in Chapter 2. Figure 7.3 shows three examples for comparing with operator-based design using other approaches.

When *max* is used, Figure 7.1($a$) can be expressed as:

$$\begin{aligned} F \;=\; & ((1 \bullet {}^{0}X_1{}^{2}) \bullet (1 \bullet {}^{0}X_0{}^{1})) + (1 \bullet {}^{1}X_0{}^{1}) + \\ & ((1 \bullet {}^{0}X_1{}^{0}) \bullet (1 \bullet {}^{0}X_0{}^{2})) + ((1 \bullet {}^{1}X_1{}^{2}) \bullet (1 \bullet {}^{3}X_0{}^{3})) + \\ & ((2 \bullet {}^{0}X_1{}^{2}) \bullet (2 \bullet {}^{1}X_0{}^{1})) + ((3 \bullet {}^{1}X_1{}^{1}) \bullet (3 \bullet {}^{1}X_0{}^{1})). \end{aligned} \tag{7.1}$$

When *tsum* is used, Figure 7.1($a$) can be expressed with fewer number of PTs:

$$\begin{aligned} F \;=\; & ((1 \bullet {}^{0}X_1{}^{2}) \bullet (1 \bullet {}^{0}X_0{}^{1})) \boxplus ((1 \bullet {}^{1}X_1{}^{3}) \bullet (1 \bullet {}^{1}X_0{}^{1})) \boxplus \\ & ((1 \bullet {}^{0}X_1{}^{0}) \bullet (1 \bullet {}^{1}X_0{}^{2})) \boxplus ((1 \bullet {}^{1}X_1{}^{2}) \bullet (1 \bullet {}^{3}X_0{}^{3})) \boxplus \\ & ((2 \bullet {}^{1}X_1{}^{1}) \bullet (2 \bullet {}^{1}X_0{}^{1})). \end{aligned} \tag{7.2}$$

Since a *literal* operator consists of 13 transistors, a *min* operator consists of 7 transistors, and a *tsum* operator consists of 9 transistors, circuit realization of Eq. 7.2 needs $10 \times 13 + 4 \times 7 + 9 = 167$ transistors. That actual circuit size is more than this number because current mirrors are frequently used for reversing current direction and level

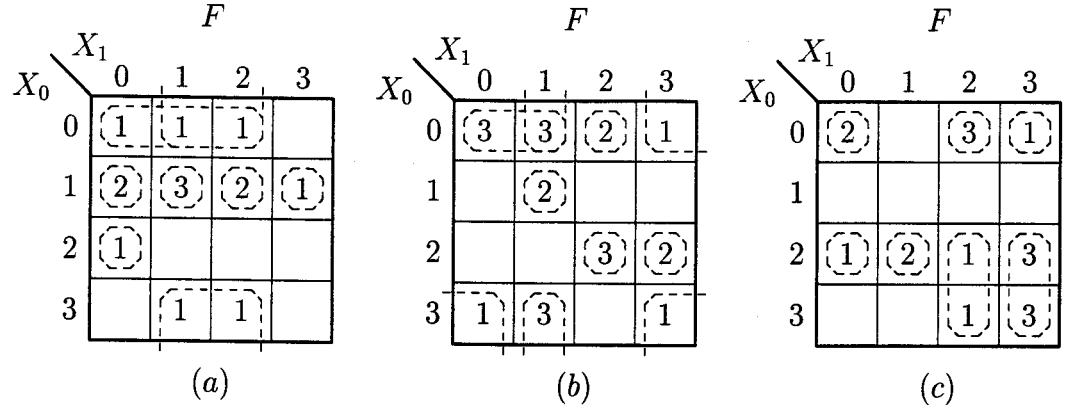



**Figure 7.1** Truth tables of three examples for comparing with operator-based MVL design using the HAMLET program.

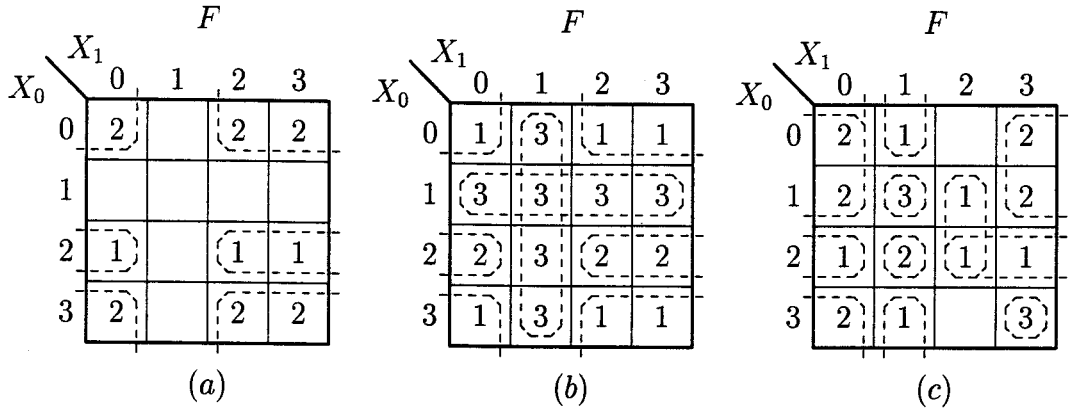



**Figure 7.2** Truth tables of three examples for comparing with operator-based MVL design using the Set C operators.

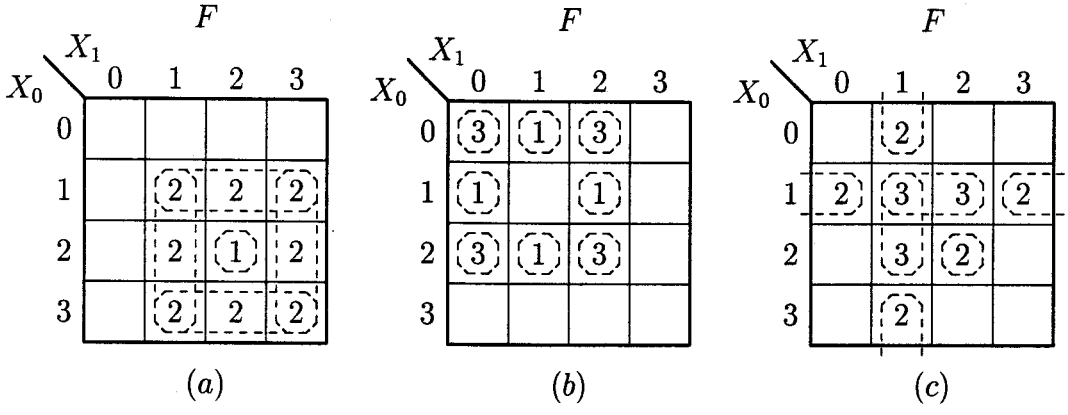



**Figure 7.3** Truth table of three examples for comparing with MVL design using other approaches.

restorer circuits are required for recovering signals. Additionally, more than one *tsum* operator is required due to the limitation of maximum input current. If interface with a binary logic module is needed, extra binary-MVL or MVL-binary converters make the number even greater. The same function can be implemented by the self-restored architecture with 44 transistors, which is about 25% of the operator-based design.

Figure 7.1($b$) can be expressed as:

$$\begin{aligned} F = {} & (1 \bullet {}^{0}X_0{}^{0}) \boxplus ((1 \bullet {}^{2}X_1{}^{2}) \bullet (1 \bullet {}^{0}X_0{}^{0})) \boxplus ((1 \bullet {}^{0}X_1{}^{0}) \bullet (1 \bullet {}^{3}X_0{}^{3})) \boxplus \\ & ((1 \bullet {}^{3}X_1{}^{3}) \bullet (1 \bullet {}^{3}X_0{}^{3})) \boxplus ((3 \bullet {}^{1}X_1{}^{1}) \bullet (3 \bullet {}^{3}X_0{}^{3})) \boxplus \\ & ((2 \bullet {}^{0}X_1{}^{1}) \bullet (2 \bullet {}^{0}X_0{}^{0})) \boxplus ((2 \bullet {}^{1}X_1{}^{1}) \bullet (2 \bullet {}^{1}X_0{}^{1})) \boxplus \\ & ((2 \bullet {}^{3}X_1{}^{3}) \bullet (2 \bullet {}^{2}X_0{}^{2})) \boxplus ((3 \bullet {}^{2}X_1{}^{2}) \bullet (3 \bullet {}^{2}X_0{}^{2})). \end{aligned} \tag{7.3}$$

Circuit realization of Eq. 7.3 needs more than $17 \times 13 + 8 \times 7 + 9 = 286$ transistors. The same function can be implemented by the self-restored architecture with 72 transistors.

Figure 7.1($c$) can be expressed as:

$$\begin{aligned} F = {} & ((1 \bullet {}^{3}X_1{}^{3}) \bullet (1 \bullet {}^{0}X_0{}^{0}) \boxplus ((1 \bullet {}^{0}X_1{}^{1}) \bullet (1 \bullet {}^{2}X_0{}^{2}) \boxplus \\ & ((1 \bullet {}^{1}X_1{}^{2}) \bullet (1 \bullet {}^{2}X_0{}^{3}) \boxplus ((2 \bullet {}^{0}X_1{}^{0}) \bullet (2 \bullet {}^{0}X_0{}^{0}) \boxplus \\ & ((3 \bullet {}^{2}X_1{}^{2}) \bullet (3 \bullet {}^{0}X_0{}^{0}) \boxplus ((3 \bullet {}^{3}X_1{}^{3}) \bullet (3 \bullet {}^{2}X_0{}^{3}). \end{aligned} \tag{7.4}$$

Circuit realization of Eq. 7.4 needs more than $12 \times 13 + 6 \times 7 + 9 = 207$ transistors. The same function can be implemented by the self-restored architecture with 66 transistors.

Figure 7.2($a$) can be expressed as:

$$F = (2 \bullet \overline{{}^{1}X_1{}^{1}}) \bullet X_0^{\overrightarrow{3}}. \tag{7.5}$$

Circuit realization of Eq. 7.5 needs more than $1 \times 13 + 1 \times 7 + 14 = 34$ transistors

as a *cycle* operator consists of 14 transistors. The same function can be implemented by the self-restored architecture with 38 transistors.

Figure 7.2($b$) can be expressed as:

$$F = X_0^{\overleftarrow{3}} \boxplus (1 \bullet {}^0X_0{}^0) \boxplus (2 \bullet {}^1X_1{}^1) \bullet X_0^{\overrightarrow{3}}. \tag{7.6}$$

Circuit realization of Eq. 7.6 needs more than $2 \times 13 + 2 \times 14 + 7 + 9 = 58$ transistors. The same function can be implemented by the self-restored architecture with 38 transistors.

Figure 7.2($c$) can be expressed as:

$$F = (1 \bullet {}^1X_0{}^2) \boxplus (X_1^{\overleftarrow{3}} \bullet X_0^{\overleftarrow{3}}) \boxplus ((1 \bullet {}^1X_1{}^1) \bullet (1 \bullet {}^1X_0{}^2)). \tag{7.7}$$

Circuit realization of Eq. 7.7 needs more than $3 \times 13 + 2 \times 14 + 2 \times 7 + 9 = 90$ transistors. The same function can be implemented by the self-restored architecture with 86 transistors.

Figure 7.3($a$) can be expressed as:

$$F = ((2 \bullet {}^1X_1{}^3) \bullet (2 \bullet {}^1X_0{}^3)) - ((1 \bullet {}^2X_1{}^2) \bullet (1 \bullet {}^2X_0{}^2)). \tag{7.8}$$

where '$-$' means arithmetic difference. That is, the output is $(2 \bullet {}^1X_1{}^3) \bullet (2 \bullet {}^1X_0{}^3)$ subtracted by $(1 \bullet {}^2X_1{}^2) \bullet (1 \bullet {}^2X_0{}^2)$. If the *min* operator on $(2 \bullet {}^1X_1{}^3)$ and $(2 \bullet {}^1X_0{}^3)$ is P-type circuit, the *min* operator on $(1 \bullet {}^2X_1{}^2)$ and $(1 \bullet {}^2X_0{}^2)$ must be N-type circuit, and vice versa. Alternatively, a current mirror can be used for reversing current direction when both *min* operators are implemented with the same type of circuit. Circuit realization of Eq. 7.8 needs more than $4 \times 13 + 2 \times 7 = 66$ transistors. The same function can be implemented by the self-restored architecture with 34 transistors.

Figure 7.3($b$) can be expressed as:

$$F = (((3 \bullet {}^{0}X_1{}^{2}) \bullet (3 \bullet {}^{0}X_0{}^{2})) + ((3 \bullet {}^{0}X_1{}^{2}) \bullet (1 \bullet {}^{1}X_0{}^{1}))) - (((2 \bullet {}^{1}X_1{}^{1}) \bullet (2 \bullet {}^{0}X_0{}^{2})) + ((2 \bullet {}^{0}X_1{}^{2}) \bullet (2 \bullet {}^{1}X_0{}^{1}))) \quad (7.9)$$

where '+' means arithmetic sum, which does not need extra transistors. Circuit realization of Eq. 7.9 needs more than $8 \times 13 + 4 \times 7 = 132$ transistors. The same function can be implemented by the self-restored architecture with 54 transistors.

Figure 7.3($c$) can be expressed as:

$$F = 2 \bullet ({}^{1}X_1{}^{1} \cup {}^{1}X_0{}^{1}) \boxplus 2 \bullet ({}^{1}X_1{}^{2} \cap {}^{1}X_0{}^{2}) \quad (7.10)$$

where '$\cup$' means the binary *or* operation and '$\cap$' means the binary *and* operation. Circuit realization of Eq. 7.10 needs more than 65 transistors as described in [59]. The same function can be implemented by the self-restored architecture with 66 transistors.

The transistor counts of these examples are summarized in Table 7.1. The table shows the average size of self-restored design is 55.33, while average size of operator-based design is 122.78 which is 2.22 times larger. However, more accurate comparison

**Table 7.1** Transistor count of MVL functions of Figures 7.1, 7.2 and 7.3.

| **Figure** | 7.1 | | | 7.2 | | | 7.3 | | |
|---|---|---|---|---|---|---|---|---|---|
| | ($a$) | ($b$) | ($c$) | ($a$) | ($b$) | ($c$) | ($a$) | ($b$) | ($c$) |
| **Equation** | 7.2 | 7.3 | 7.4 | 7.5 | 7.6 | 7.7 | 7.8 | 7.9 | 7.10 |
| **Self-Restored** | 44 | 72 | 66 | 38 | 38 | 86 | 34 | 54 | 66 |
| **Operator-Based**† | 167 | 286 | 207 | 34 | 58 | 90 | 66 | 132 | 65 |

† The actual transistor count is larger than the number shown in this table.

was needed to support this statement. To this end, an experiment was conducted to obtain the average number of transistors of 2-variable 4-valued functions. Considering the fact that there are $4^{4^2}$ possible combinations for 2-variable 4-valued functions, a statistical approach is preferable. Monte Carlo simulation (MCS) was adopted to estimate the number of functions required for obtaining the estimation of average number of transistors.[1] Through repeated simulations of different set of pseudo-randomly generated functions, it is found that MCS results begin converging to a value between 50 to 150 functions. That means a number larger than 150 is a good choice for the number of functions. Two MCS results of 200 functions and 500 functions are shown in Figure 7.4. Considering CPU power and the fact that a larger number of functions results in better confidence interval, 500 2-variable 4-valued logic functions were pseudo-randomly generated and then synthesized using the WMS program together with the Synopsys Design Compiler using the CMC BiCMOS synthesis library. To assure correct synthesized results, the synthesized circuits were simulated using a VHDL simulator with the CMCL library for quick verification. Circuit simulations were then carried out in order to obtain an average power dissipation and timing information at the same time.

Table 7.2 shows the transistor count of the 500 pseudo-randomly generated functions. It should be noted the transistor count of each function is a fractional number because the transistor sizes and interconnection area within library cells were also considered by the Synopsys Design Compiler. The average transistor count of the 500 MVL circuits is 79.62. The 95% confidence interval [68, 69] for this average value is $(78.66, 80.58)$. It should be noted that the numbers are higher than the actual transistor count since the transistor sizes and interconnection area within library cells is also taken into consideration when they are generated by the Synopsys Design Compiler. It should also be noted the average transistor count of the 500 self-restored MVL designs is based on the self-restored architecture shown in Figure 3.7.

[1]Based on the private communication with Dr. Jinting Wang of Mathematics Department.

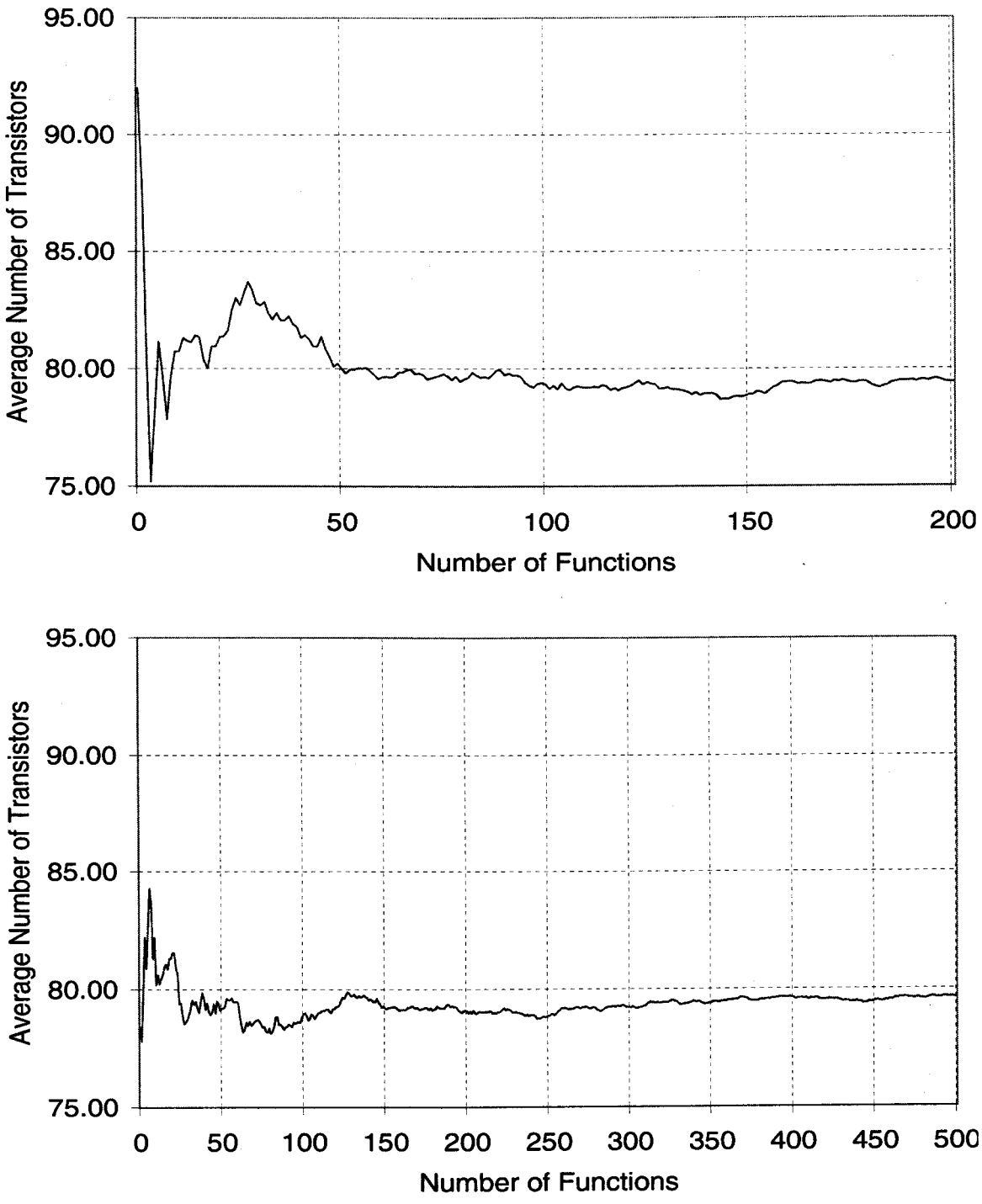



**Figure 7.4** Monte Carlo simulation results of 200 functions and 500 functions.

**Table 7.2** Transistor count of 500 2-variable 4-valued logic functions.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 92.0 | 74.0 | 79.6 | 77.6 | 78.4 | 68.4 | 92.8 | 79.6 | 91.6 | 84.4 |
| **2** | 83.2 | 59.6 | 77.6 | 61.6 | 77.2 | 70.4 | 78.8 | 103.2 | 80.8 | 78.8 |
| **3** | 66.4 | 92.0 | 88.8 | 67.2 | 84.0 | 88.8 | 96.8 | 73.2 | 92.0 | 85.2 |
| **4** | 59.2 | 64.0 | 69.2 | 94.0 | 89.2 | 79.6 | 74.8 | 80.4 | 62.8 | 95.6 |
| **5** | 90.0 | 105.6 | 68.4 | 80.0 | 75.6 | 83.6 | 67.2 | 76.8 | 73.6 | 76.4 |
| **6** | 96.0 | 57.2 | 82.4 | 86.0 | 92.4 | 78.4 | 102.0 | 78.4 | 79.2 | 80.0 |
| **7** | 70.8 | 71.2 | 86.0 | 66.8 | 93.2 | 82.4 | 99.6 | 88.0 | 89.2 | 84.0 |
| **8** | 65.2 | 92.4 | 82.0 | 82.0 | 78.8 | 71.2 | 88.0 | 81.2 | 96.0 | 70.4 |
| **9** | 94.8 | 83.6 | 85.2 | 89.6 | 62.8 | 70.8 | 102.4 | 86.8 | 68.0 | 78.4 |
| **10** | 89.6 | 73.6 | 90.8 | 59.6 | 90.4 | 69.2 | 82.8 | 74.8 | 99.6 | 80.4 |
| **11** | 80.8 | 79.6 | 85.2 | 77.6 | 60.0 | 84.4 | 81.6 | 94.4 | 78.8 | 76.4 |
| **12** | 87.6 | 78.0 | 106.8 | 97.2 | 85.2 | 80.8 | 77.2 | 83.2 | 76.4 | 42.8 |
| **13** | 79.6 | 84.4 | 85.2 | 74.8 | 75.6 | 77.6 | 65.6 | 76.0 | 100.0 | 58.4 |
| **14** | 80.4 | 75.2 | 96.8 | 78.4 | 83.6 | 82.4 | 83.6 | 86.0 | 94.8 | 57.2 |
| **15** | 85.6 | 90.4 | 66.4 | 91.6 | 82.8 | 91.2 | 76.8 | 76.4 | 71.6 | 84.8 |
| **16** | 80.0 | 74.0 | 82.0 | 79.2 | 86.0 | 79.6 | 79.2 | 92.8 | 91.2 | 99.2 |
| **17** | 65.2 | 60.0 | 69.2 | 58.4 | 82.8 | 86.8 | 92.0 | 71.2 | 91.2 | 70.0 |
| **18** | 73.6 | 88.0 | 91.6 | 72.0 | 77.2 | 81.2 | 85.6 | 96.0 | 71.6 | 89.6 |
| **19** | 98.0 | 68.4 | 69.2 | 76.0 | 89.2 | 67.2 | 80.8 | 99.2 | 87.6 | 67.2 |
| **20** | 80.8 | 90.8 | 81.2 | 83.6 | 82.0 | 81.6 | 72.4 | 74.4 | 81.2 | 82.8 |
| **21** | 89.2 | 83.6 | 81.6 | 77.6 | 86.4 | 75.2 | 72.4 | 78.0 | 74.4 | 86.0 |
| **22** | 82.0 | 86.4 | 88.0 | 79.6 | 81.2 | 66.4 | 96.0 | 72.4 | 66.8 | 82.0 |
| **23** | 87.2 | 93.6 | 79.6 | 90.4 | 66.4 | 82.8 | 72.8 | 75.6 | 81.2 | 81.6 |
| **24** | 102.8 | 92.8 | 74.4 | 76.0 | 75.2 | 83.6 | 90.8 | 57.2 | 76.0 | 74.0 |
| **25** | 95.2 | 59.2 | 83.2 | 77.6 | 48.8 | 83.6 | 69.2 | 68.8 | 74.4 | 69.2 |
| **26** | 74.4 | 91.6 | 90.0 | 73.6 | 80.4 | 85.6 | 71.2 | 71.2 | 76.8 | 68.0 |
| **27** | 97.6 | 70.4 | 97.2 | 88.0 | 64.4 | 71.6 | 76.4 | 84.0 | 96.0 | 80.8 |
| **28** | 96.0 | 76.0 | 74.4 | 78.8 | 69.6 | 68.4 | 88.4 | 82.0 | 80.4 | 62.0 |
| **29** | 73.6 | 56.0 | 63.6 | 68.4 | 80.8 | 91.2 | 82.8 | 73.6 | 62.8 | 75.6 |
| **30** | 66.4 | 78.8 | 78.8 | 70.0 | 82.0 | 62.0 | 72.4 | 81.6 | 76.8 | 90.8 |
| **31** | 79.6 | 86.0 | 53.6 | 61.2 | 85.2 | 87.6 | 57.2 | 98.0 | 76.8 | 61.6 |
| **32** | 87.6 | 69.2 | 64.0 | 75.2 | 88.0 | 84.8 | 68.4 | 84.4 | 77.2 | 85.6 |
| **33** | 66.0 | 78.8 | 81.2 | 89.6 | 90.0 | 98.0 | 68.4 | 66.4 | 84.8 | 95.2 |
| **34** | 74.4 | 74.8 | 82.0 | 80.4 | 73.2 | 70.4 | 91.6 | 94.8 | 98.4 | 112.0 |
| **35** | 91.6 | 73.6 | 85.2 | 78.4 | 83.6 | 70.4 | 88.0 | 84.0 | 85.6 | 79.6 |
| **36** | 71.2 | 70.4 | 84.8 | 64.0 | 70.0 | 82.8 | 100.4 | 82.0 | 92.8 | 49.6 |
| **37** | 81.6 | 66.4 | 84.0 | 75.6 | 72.0 | 75.6 | 84.8 | 77.6 | 78.4 | 80.4 |
| **38** | 89.2 | 92.0 | 82.0 | 86.8 | 96.0 | 93.2 | 87.6 | 88.4 | 72.8 | 70.4 |
| **39** | 69.2 | 61.2 | 91.6 | 73.6 | 94.8 | 96.0 | 78.8 | 84.8 | 66.4 | 66.8 |
| **40** | 76.8 | 90.4 | 95.2 | 99.2 | 70.0 | 78.8 | 80.4 | 76.4 | 87.2 | 87.2 |
| **41** | 62.8 | 78.4 | 70.4 | 56.4 | 60.8 | 61.6 | 84.8 | 82.4 | 70.4 | 82.4 |
| **42** | 86.4 | 80.4 | 71.6 | 79.6 | 90.4 | 87.2 | 68.4 | 90.8 | 84.8 | 85.6 |
| **43** | 73.2 | 68.8 | 88.0 | 80.8 | 63.2 | 70.4 | 90.0 | 83.6 | 97.2 | 74.4 |
| **44** | 68.4 | 51.6 | 76.4 | 48.8 | 73.6 | 81.6 | 84.4 | 84.4 | 82.8 | 71.2 |
| **45** | 80.4 | 82.8 | 64.0 | 90.4 | 84.4 | 71.6 | 70.4 | 77.2 | 76.4 | 98.0 |
| **46** | 100.4 | 76.0 | 69.6 | 73.6 | 94.8 | 56.8 | 90.4 | 75.6 | 66.8 | 73.6 |
| **47** | 56.8 | 91.6 | 65.6 | 104.0 | 60.4 | 67.2 | 85.2 | 77.6 | 84.4 | 82.8 |
| **48** | 65.6 | 84.8 | 88.8 | 86.0 | 104.0 | 71.2 | 68.4 | 86.0 | 81.6 | 82.8 |
| **49** | 60.0 | 74.4 | 78.0 | 68.8 | 72.8 | 93.2 | 67.6 | 84.0 | 86.4 | 73.6 |
| **50** | 86.0 | 81.6 | 90.8 | 94.0 | 66.0 | 84.8 | 74.8 | 72.4 | 61.6 | 101.6 |

In order to compare with conventional operator-based MVL designs it is necessary to know the average circuit size of 2-variable 4-valued functions in terms of transistor count using the conventional operator-based MVL design schemes. The information of such experimental data has not been found. However, according to [59], the average number of product terms (PTs) for 2-variable 4-valued logic functions expressed in a *sum-of-product* form is 6.62 based on Set A operators (*literal*, *min*, and *tsum*) and 6.02 for Set B operators (*literal*, *complement of literal*, *min*, and *tsum*) using HAMLET (Gold) [42, 17], and 5.53 for Set C operators (*literal*, *cycle*, *complement of literal*, *complement of cycle*, *min*, and *tsum*).

The circuit size of conventional operator-based MVL designs can be estimated based on the transistor counts of the MVL operators. Referring to Appendix B, a level restorer uses 19 transistors, a *tsum* operator uses 9 transistors, a *min* operator uses 7 transistors, *literal* and *complement of literal* operators use 13 transistors, a *cycle* uses 14 transistors. The following assumptions were also made for the estimation. First, Set C operators are used. Secondly, the average number of transistors of a unary operator is the average of a *literal* operator and a *cycle* operator; i.e., $(13 + 14)/2 = 13.5$. Thirdly, a multi-input *tsum* operator is used in place of all *max* operators to implement a MVL function. Fourthly, a level restorer is required for every four stages and a MVL function is considered as on stage; i.e, each MVL function shares $19 \times 0.25 = 4.75$ transistors of a level restorer. Based on those assumptions, an estimated transistor count of the conventional operator-based design can be calculated for different number of unary operators in a PT using the equation:

$$5.53 \times (13.5 \times x) + 7 \times (x - 1) + 9 + 4.75$$

where $x$ is the number of unary operators in a PT, 5.53 is the average number of PTs for 2-variable 4-valued logic functions, 13.5 is the average number of transistors of unary operators, 7 is the number of transistors of the *min* operator, 9 is the number of transistors of the *tsum* operator, and 4.75 is 1/4 of the number of transistors of the

level restorer. If each PT has only one unary operator, the total number of transistors is $5.53 \times 13.5 + 9 + 4.75 \approx 89$. If each PT has 1.5 unary operators on average, the total number of transistors is $5.53 \times (13.5 \times 1.5 + 7 \times 0.5) + 9 + 4.75 \approx 145$. If each PT has two unary operators, the total number of transistors is $5.53\ (13.5 \times 2 + 7) + 9 + 4.75 \approx 202$. These numbers are 1.1 to 2.5 times larger than that of the MVL design based on the self-restored architecture shown in Figure 3.7.

The time delays in a current-mode CMOS MVL design are dominated by threshold elements. Figures 7.5 and 7.6 are two typical simulation results of 2-variable 4-valued logic functions. The functions are realized with both the self-restored architecture and the operator-based design by using Nortel BiCMOS $0.8\mu$ technology. The outputs of both designs are shown in the figures. With a P-type current mirror as load, the propagation delay for both the self-restored architecture and the operator-based design ranges from $5ns$ to $15ns$ approximately, dependent on different state transitions. The average propagation delay is $10ns$ approximately. It can be seen from the figures that even though the self-restored design and the operator-based design have very close propagation delay, the self-restored design has much shorter rise and fall time delays ($1ns$ approximately) than the operator-base design ($5ns$ approximately).

It is well known that static power is the major dissipation source in an conventional operator-based CMCL MVL design. Since a small number of binary gates are also used in the MVL operators implemented with CMCL and also due to the switching activities of current sources, the total power dissipation slowly increases with frequency. In the self-restored MVL design of Figure 3.7, static power dissipation and dynamic power dissipation are equally important because static power dominates the power dissipation in the input and output blocks while dynamic power dominates the power dissipation in the control block. The total power dissipation increases with the circuit size ratio between the control block and the input/output blocks. Assume the maximum number of transistors in an input block and an output block for a 2-variable 4-valued function. That is, an input block consists of two 3-output P-type current

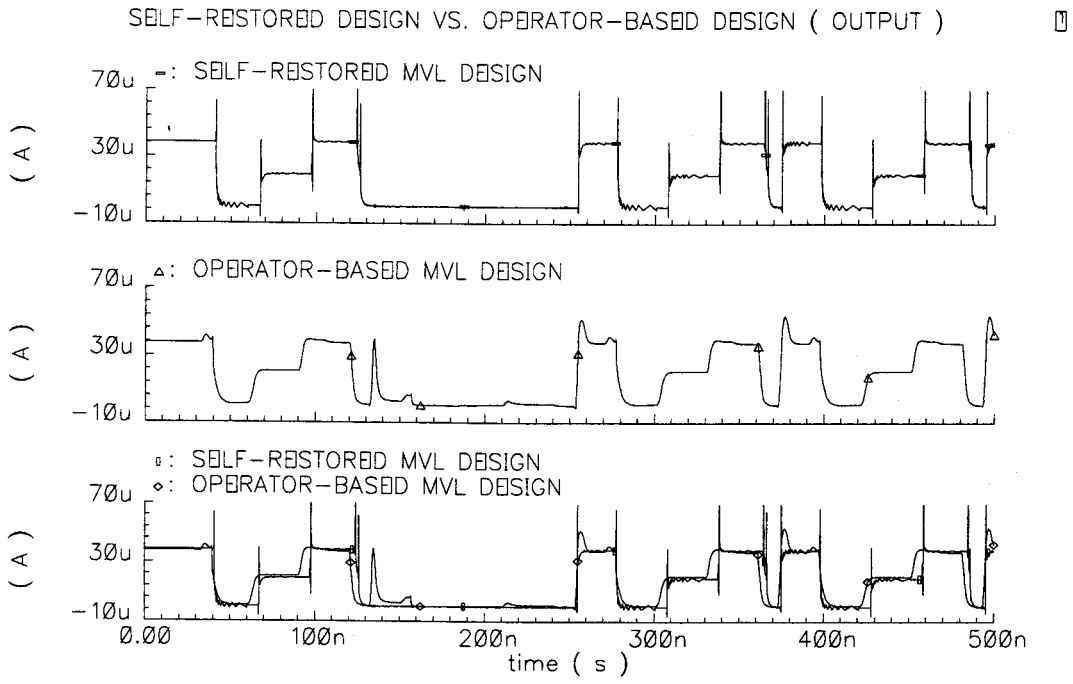



**Figure 7.5** Spectre simulation of transient analysis of a 2-variable 4-valued logic function for measuring delay times.

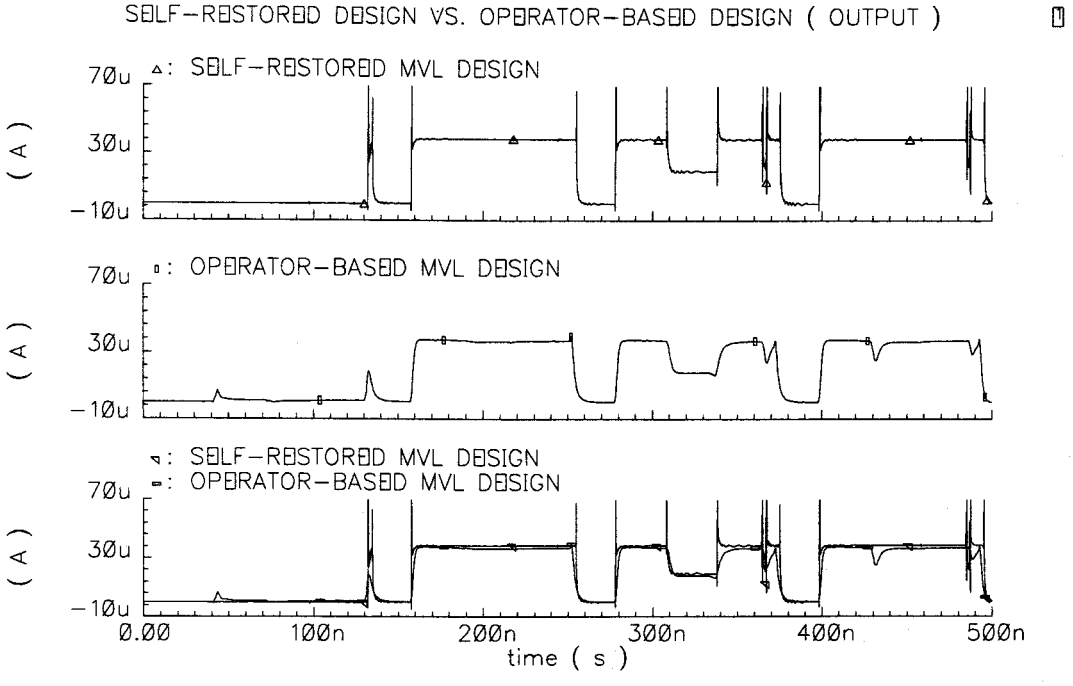



**Figure 7.6** Spectre simulation of transient analysis of a 2-variable 4-valued logic function for measuring delay times.

mirrors and a N-type threshold connected to each output of the current mirrors, while an output block consists of three current sources and three switches. The total transistor count of an input block and an output block is $2 \times (3 \times 2 + 1) + (3 \times 2) = 20$. Since the average transistor count of 2-variable 4-valued functions is 79.62, the estimated transistor count of a control block is $79.62 - 20 = 59.62$. Therefore, the ratio of circuit sizes between the control block and the input/output blocks for 2-variable 4-valued functions is around 3 : 1. Based on the above calculation, it can be expected that the power dissipation increases with frequency faster than the the conventional operator-based design.

Figures 7.7 and 7.8 are typical current waveforms flowing from $V_{DD}$ to $V_{SS}$ for self-restored designs and equivalent operator-based designs. For each analysis, an average power dissipation is obtained by multiplying the average current flowing from $V_{DD}$ to $V_{SS}$ with $V_{DD}$ which is 5 volts in this case. For example, the average currents of the two self-restored MVL designs in Figures 7.7 and 7.8 are 251.6$\mu$A and 306.7$\mu$A respectively, while the average currents of the equivalent operator-based designs are 271.7$\mu$A and 524.0$\mu$A respectively. The average power dissipations of the two self-restored designs are 1258.0$\mu$W and 1533.5$\mu$W, respectively. The average power dissipations of the two operator-based designs are 1358.5$\mu$W and 2620.0$\mu$W, respectively. In both figures, the top sub-figure is the current waveform of the self-restored MVL design, the middle sub-figure is the current waveform of a equivalent operator-based MVL design, and the bottom sub-figure is combined waveforms of the self-restored MVL design and the equivalent operator-based MVL design for comparison. Multiple transient analyses ranging from 10MHz to 50MHz for each of these two examples were also simulated using the Cadence Spectre simulator. The upper bound of the frequency is set to 50MHz because of the restriction on the propagation delay. As mentioned above, the propagation delay falls in the range of 5$ns$ to 15$ns$ approximately. It is safe to set the maximum propagation delay to 20$ns$. From figures 7.9 and 7.10 it can be seen that the power dissipation of a self-restored design

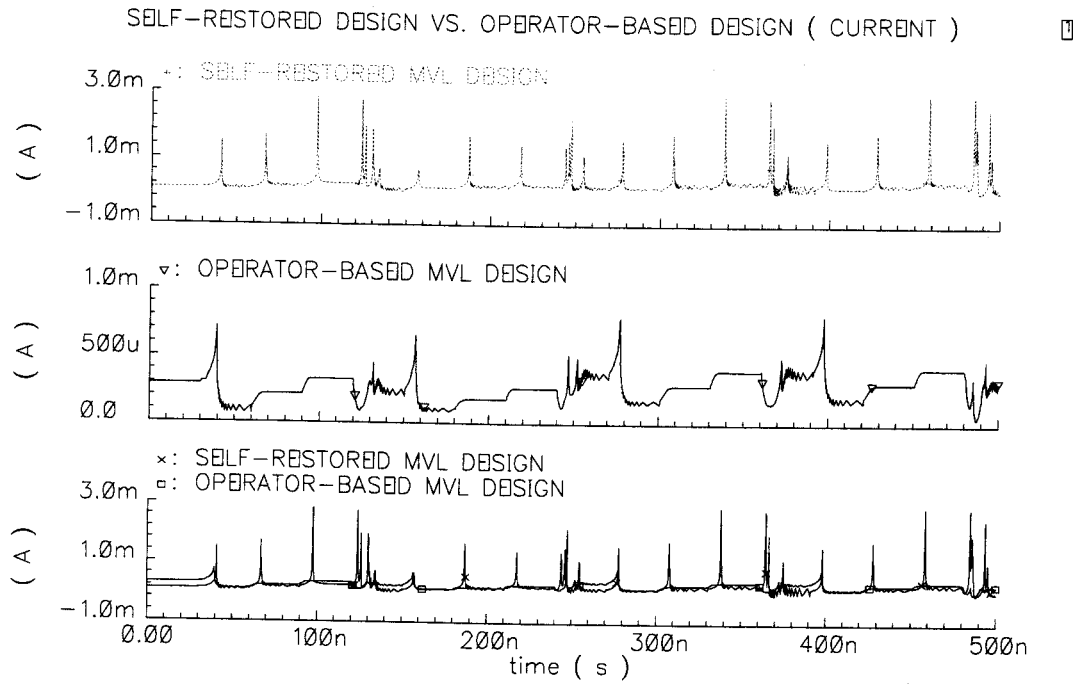



**Figure 7.7** Spectre simulation of transient analysis of the same 2-variable 4-valued logic function shown in Figure 7.5 for measuring total current from $V_{DD}$ to $V_{SS}$.

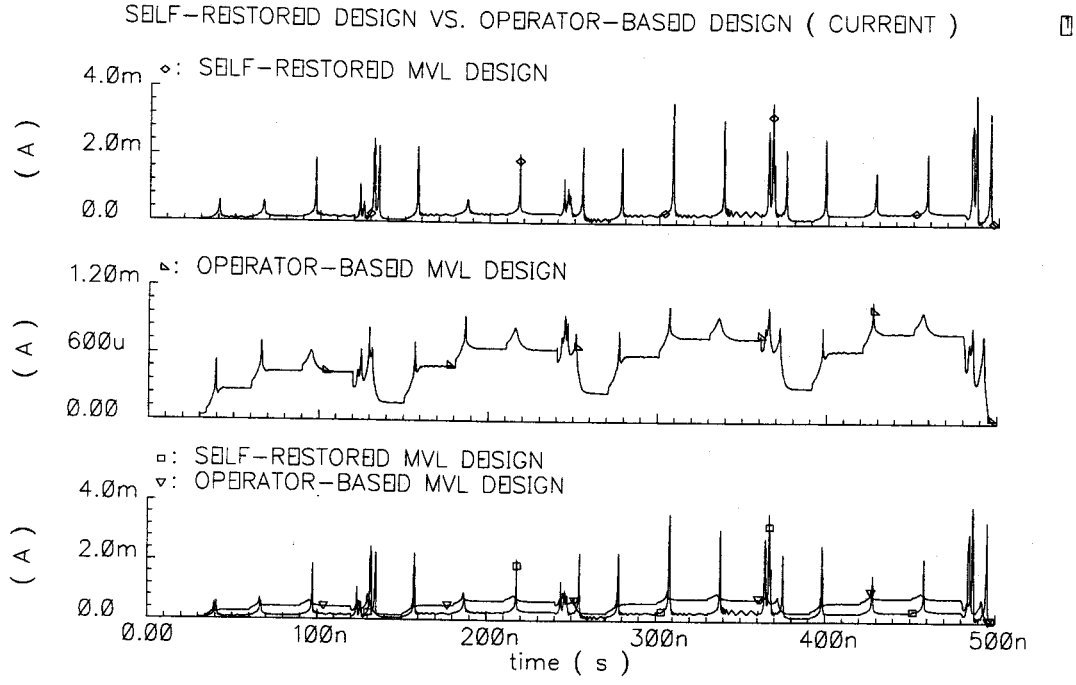



**Figure 7.8** Spectre simulation of transient analysis of the same 2-variable 4-valued logic function shown in Figure 7.6 for measuring total current from $V_{DD}$ to $V_{SS}$.

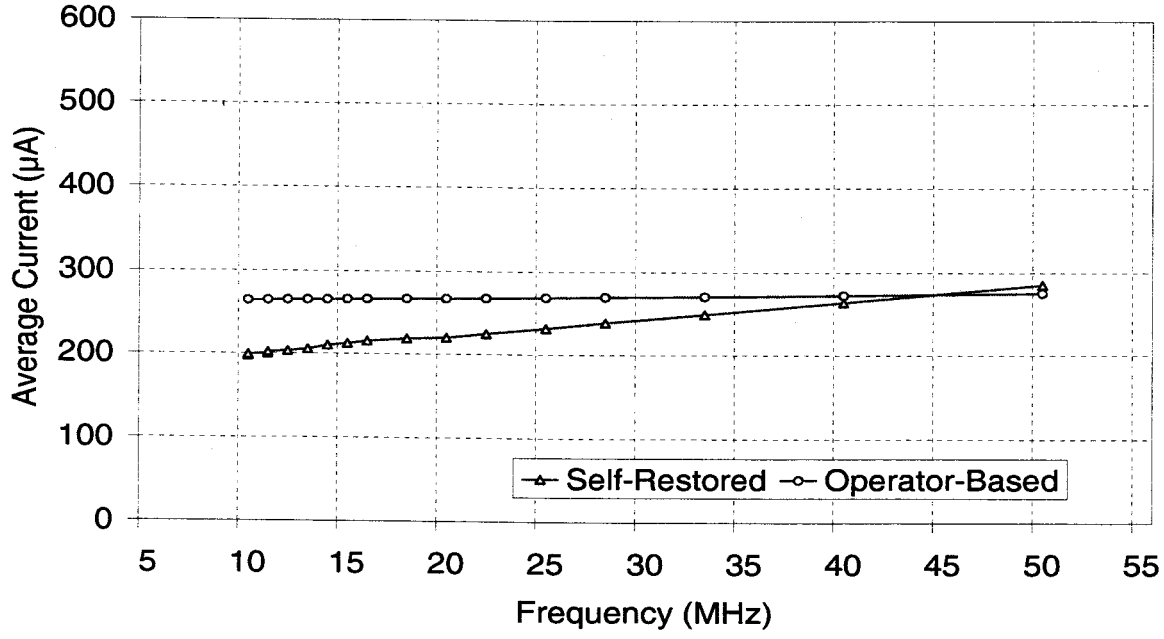



**Figure 7.9** Power dissipation of the same 2-variable 4-valued logic function shown Figure 7.5 from 10MHz to 50MHz.

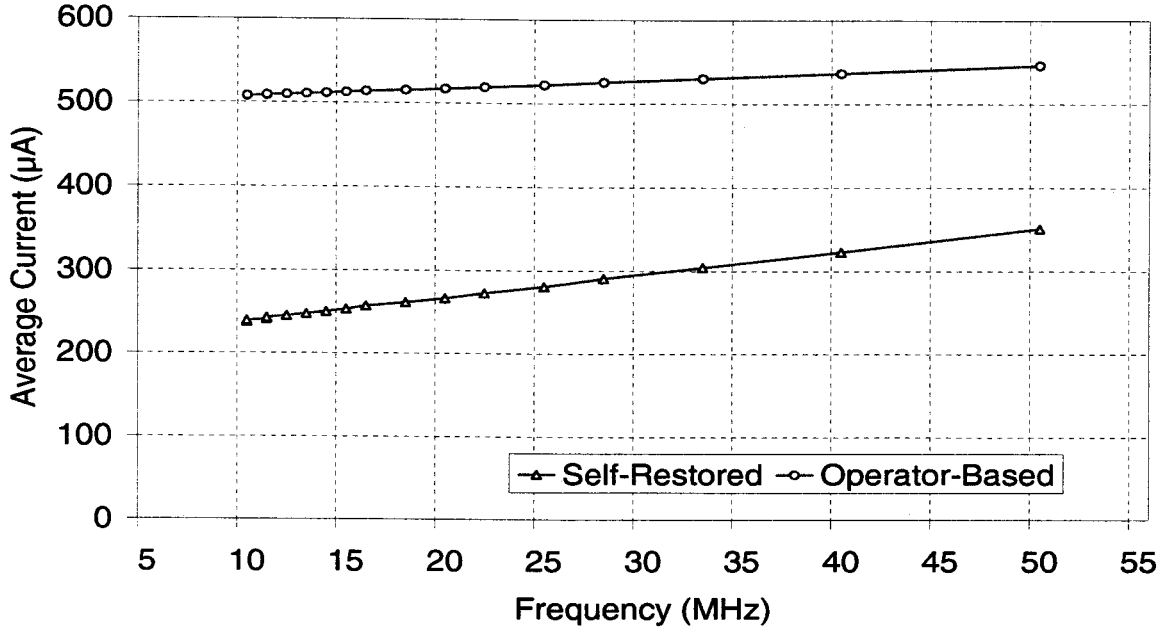



**Figure 7.10** Power dissipation of the same 2-variable 4-valued logic function shown Figure 7.6 from 10MHz to 50MHz.

increases with frequency faster than the equivalent operator-based design. Referring to Figure 7.9 the self-restored design consumes more power than the equivalent operator-based design when the frequency is higher than 45MHz. In most cases, a self-restored design consume less power than the equivalent operator-based design as shown in Figure 7.10. The differences are likely caused by the smaller circuit size of the self-restored designs.

During Spectre simulations, it was found that the threshold outputs in a self-restored design should be connected with a buffer to drive a control block which consists of binary gates. Otherwise, power dissipation would be higher due to slower rise/fall time delays. Therefore, the WMS program was modified so that it can generate circuits with or without buffers added to each threshold output. Table 7.3 summarizes the average current at 50MHz of thirty 2-variable 4-valued logic functions implemented by the self-restored architecture with and without a buffer connected to each threshold output. The average current in circuits with buffers is about half that of those without buffers.

To conclude this section, it is also interesting to see the comparison between the self-restored architecture and Chang's scheme. Three MVL functions are chosen to

**Table 7.3** Comparison of average current of 30 2-variable 4-valued logic function circuits implemented with and without buffers for threshold outputs.

| with buf | w/o buf | with buf | w/o buf | with buf | w/o buf |
|---|---|---|---|---|---|
| 412.6 | 1079.0 | 333.5 | 837.9 | 388.1 | 1171.0 |
| 388.8 | 1042.0 | 456.5 | 1317.0 | 385.2 | 1157.0 |
| 450.6 | 1150.0 | 423.2 | 1481.0 | 377.8 | 968.4 |
| 455.3 | 1347.0 | 368.9 | 1034.0 | 361.5 | 804.0 |
| 426.6 | 1213.0 | 364.9 | 992.9 | 488.3 | 1510.0 |
| 370.5 | 781.6 | 421.4 | 1172.7 | 410.5 | 1123.0 |
| 354.3 | 1006.0 | 397.3 | 992.8 | 387.4 | 1187.0 |
| 431.3 | 983.8 | 352.6 | 1029.0 | 460.2 | 1103.0 |
| 404.2 | 1097.0 | 437.5 | 1430.0 | 421.8 | 986.8 |
| 390.9 | 1166.0 | 386.7 | 976.4 | 418.2 | 1448.0 |

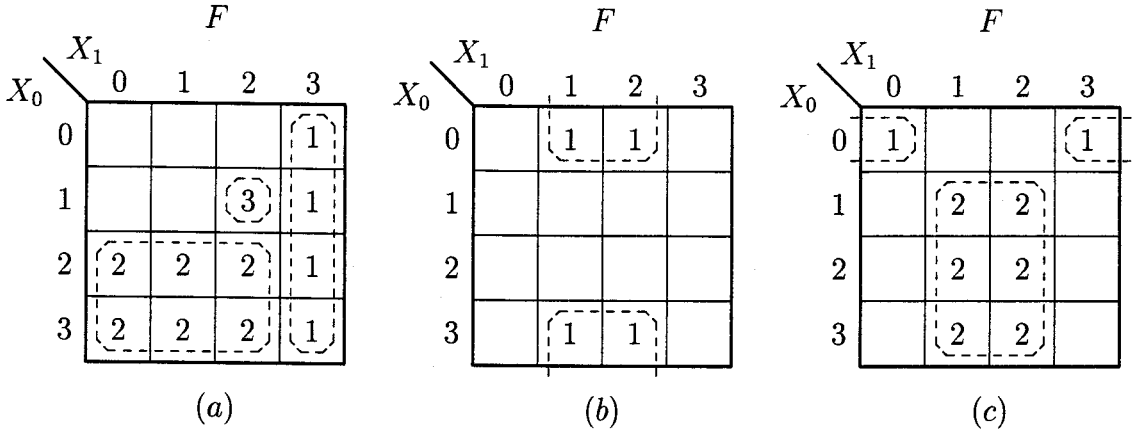



**Figure 7.11** Truth table of three examples from Chapter 3.

compare with Chang's scheme. The truth tables are shown in Figure 7.11, which is the same as Figure 3.11 on page 45. The circuit realizations of these functions according to the self-restored architecture are shown in Figure 3.12 on page 46, Figure 3.13 on page 46, and Figure 3.14 on page 47 in Chapter 3, respectively. The total number of transistors of the three functions are 34, 19, and 28, respectively.

Figure 7.11($a$) is an example taken from Chang's paper. As shown on page 28 of Chapter 2, the circuit realization needs 37 transistors, which uses 3 more transistors than the self-restored design as shown in Figure 3.12 on page 46. According to Chang's synthesis procedure, the MVL function of Figure 7.11($b$) can be expressed as:

$$\begin{aligned} F &= 1 \bullet ({}^{1}X^{2} \bullet {}^{0}Y^{0} + {}^{1}X^{2} \bullet {}^{3}Y^{3}) \\ &= 1 \bullet ({}^{1}X \bullet X^{2} \bullet {}^{0}Y \bullet Y^{0} + {}^{1}X \bullet X^{2} \bullet {}^{3}Y \bullet Y^{3}) \\ &= 1 \bullet (\overline{X^{0}} \bullet X^{2} \bullet \overline{Y^{3}} \bullet Y^{0} + \overline{X^{0}} \bullet X^{2} \bullet \overline{Y^{2}} \bullet Y^{3}). \end{aligned} \tag{7.11}$$

Referring to Figures 2.7 and 2.8 on page 28, implementation of this function needs one 2-output subcircuit 1 (3 transistors), one 3-output subcircuit 1 (4 transistors), five subcircuit 2 ($5 \times 3 = 15$ transistors), and two subcircuit 3 each with four switches ($2 \times (4+4) = 16$ transistors). The total transistors are 38, which is 2 times larger than the self-restored design (19 transistors) as shown in Figure 3.13 on page 46. Similarly,

according to Chang's method, the truth table of Figure 7.11($c$) can be expressed by a MVL function $F = 1 \bullet F_1 + 2 \bullet F_2$, where:

$$\begin{aligned} F_1 &= {}^0X^0 \bullet {}^0Y^0 + {}^3X^3 \bullet {}^0Y^0 \\ &= {}^0X \bullet X^0 \bullet {}^0Y \bullet Y^0 + {}^3X \bullet X^3 \bullet {}^0Y \bullet Y^0 \\ &= \overline{X^3} \bullet X^0 \bullet \overline{Y^3} \bullet Y^3 + \overline{X^2} \bullet X^3 \bullet \overline{Y^3} \bullet Y^3 \qquad (7.12) \\ F_2 &= {}^1X^2 \bullet {}^3Y^1 = {}^1X \bullet X^2 \bullet {}^3Y \bullet Y^1 = \overline{X^0} \bullet X^2 \bullet \overline{Y^2} \bullet Y^1 \qquad (7.13) \end{aligned}$$

To implement the function $F_1$ needs 25 transistors, including a 3-output subcircuit 1, three subcircuit 2, and eight switches in subcircuit 3. To implement the function $F_2$ needs another 25 transistors, including a 4-output subcircuit 1, four subcircuit 2, and four switches in subcircuit 3. Therefore, Chang's architecture needs 50 transistors in total. The circuit is almost twice as large as the the self-restored architecture (28 transistors) as shown in Figure 3.12 on page 46. In addition to lower implementation cost, the new architecture is also advantageous over Chang's scheme in other aspects such as virtually no static power consumption when inputs are logic 0 and synthesis of MVL functions using a binary logic synthesizer. The minimization of MVL functions according to the self-restored architecture is also easier to understand to those who are already skilled in binary logic design.

## 7.2 Comparison with the Binary Logic Design

The optimal logic radix in terms of implementation cost, as discussed in Chapter 1, is greater than 2. As mentioned in Section 7.1, a self-restored MVL design is generally smaller than an operator-based MVL design. It is interesting to compare the self-restored MVL design and binary logic design to see whether a MVL circuit is smaller than its equivalent binary logic circuit.

Like the MVL case, many examples are compared manually. In order to compare

with Chang's design at the same time, the three MVL function shown in Figure 7.11 are chosen. Figure 7.12($a$) is the same truth table of Figure 7.11($a$). The Karnaugh map of two equivalent binary functions are shown in Figure 7.12($b$). These two binary functions can be minimized for standard CMOS structure as follows:

$$\begin{aligned} f_1 &= \bar{x}_{10}x_{01} + x_{11}\bar{x}_{10}x_{00} + x_{11}\bar{x}_{10}x_{01} \\ &= \bar{x}_{10}x_{01} + x_{11}\bar{x}_{10}(x_{00} + x_{01}), \quad \text{and} \qquad (7.14) \\ f_0 &= x_{11}x_{10} + x_{11}\bar{x}_{01}x_{00} = x_{11}(x_{10} + \bar{x}_{01}x_{00}). \qquad (7.15) \end{aligned}$$

The number of transistors needed to implement these two functions is 28 ($10 \times 2 = 20$ plus 4 inverters) as shown in Figure 7.13. It should be noted that only the PMOS structure is shown in the figure (the NMOS structure is complementary). Comparing with the self-restored implementation which consists of 34 transistors as shown in Figure 3.12 on page 46 and Chang's scheme which consists of 37 transistors as discussed in Section 7.1, the binary counterpart is smaller.

Referring to the Karnaugh map of Figure 7.14($b$) the corresponding binary function is

$$\begin{aligned} f &= \bar{x}_{11}x_{10}\bar{x}_{01}\bar{x}_{00} + \bar{x}_{11}x_{10}x_{01}x_{00} + x_{11}\bar{x}_{10}\bar{x}_{01}\bar{x}_{00} + x_{11}\bar{x}_{10}x_{01}x_{00} \\ &= \bar{x}_{11}x_{10}(\bar{x}_{01}\bar{x}_{00} + x_{01}x_{00}) + x_{11}\bar{x}_{10}(\bar{x}_{01}\bar{x}_{00} + x_{01}x_{00}) \\ &= (\bar{x}_{11}x_{10} + x_{11}\bar{x}_{10})(\bar{x}_{01}\bar{x}_{00} + x_{01}x_{00}). \qquad (7.16) \end{aligned}$$

Therefore, as shown in Figure 7.15, the standard CMOS implementation of this function needs 24 transistors ($8 \times 2 = 16$ plus 4 inverters), which uses 5 more transistors than the self-restored MVL design as shown in Figure 3.13, but 14 less than Chang's scheme (38 transistors as discussed in Section 7.1).

Another example is shown in Figure 7.16($a$). The Karnaugh maps of the equivalent binary logic functions are given in Figure 7.16($b$). These two binary functions $f_1$ and

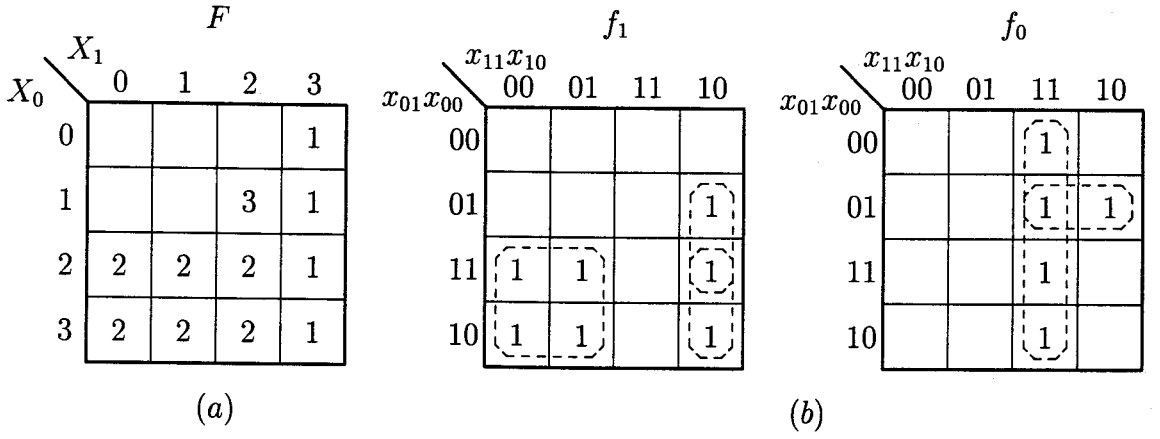



**Figure 7.12** The MVL truth table of Figure 7.11($a$) and its equivalent binary functions.

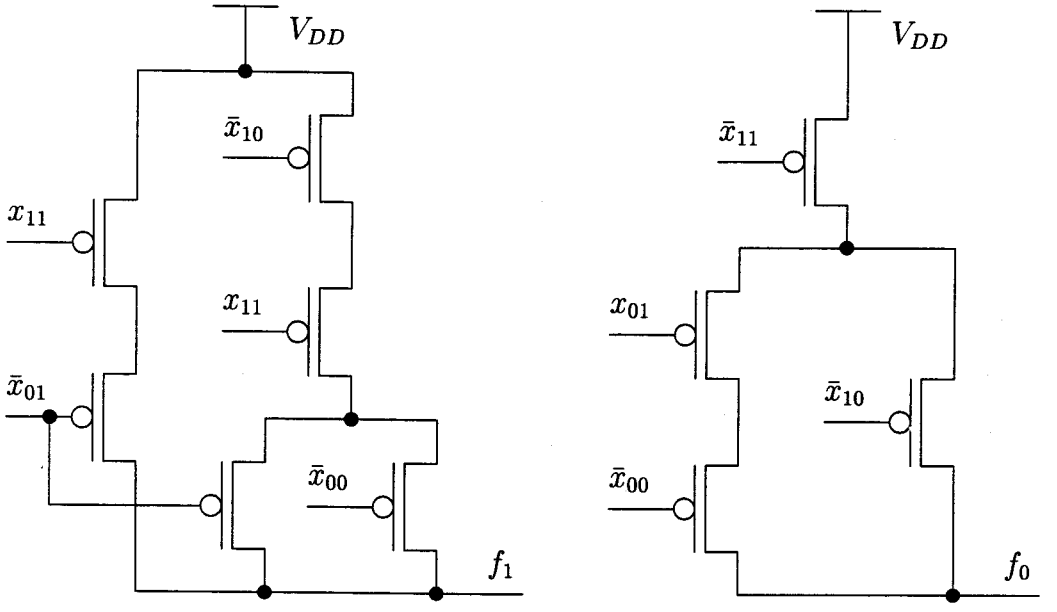



**Figure 7.13** The PMOS structure of circuit realization of Figure 7.12($b$).

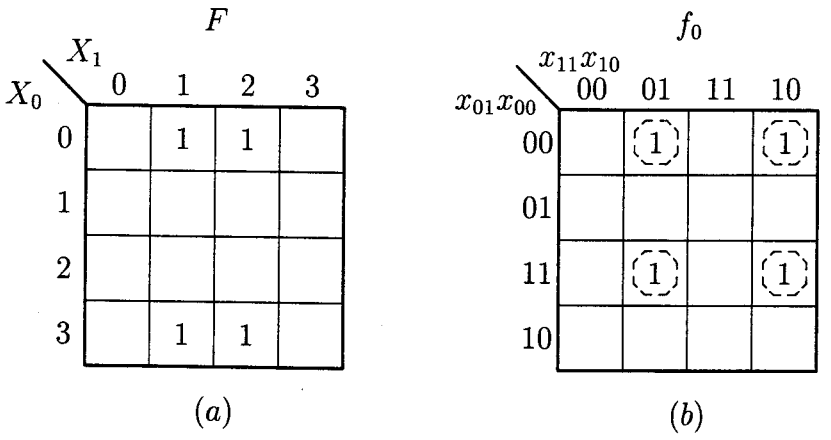



**Figure 7.14** The MVL truth table of Figure 7.11($b$) and the Karnaugh map of its equivalent binary functions.

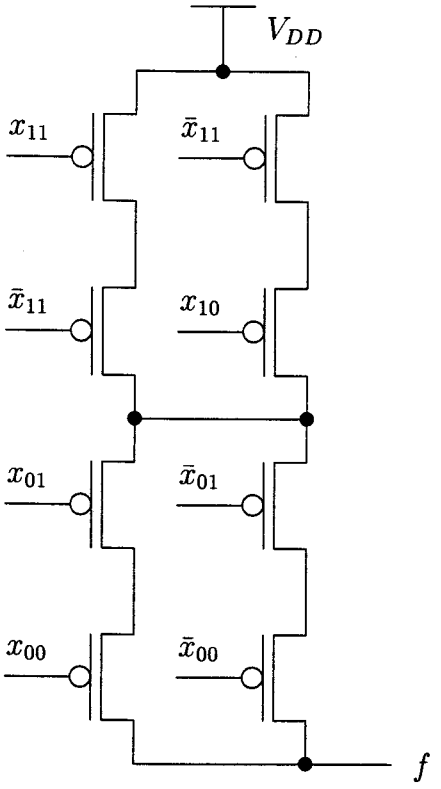



**Figure 7.15** The PMOS structure of circuit realization of Figure 7.14($b$).

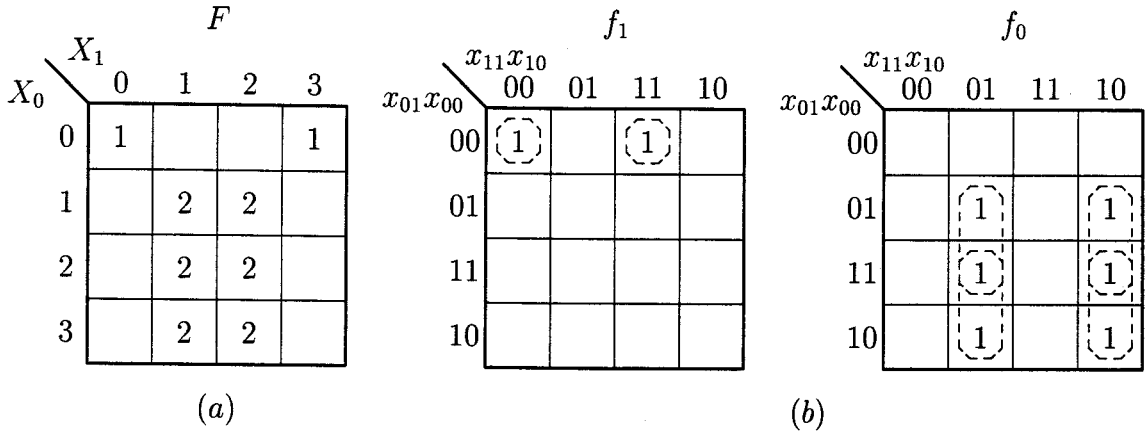



**Figure 7.16** The MVL truth table of Figure 7.11($c$) and its equivalent binary functions.

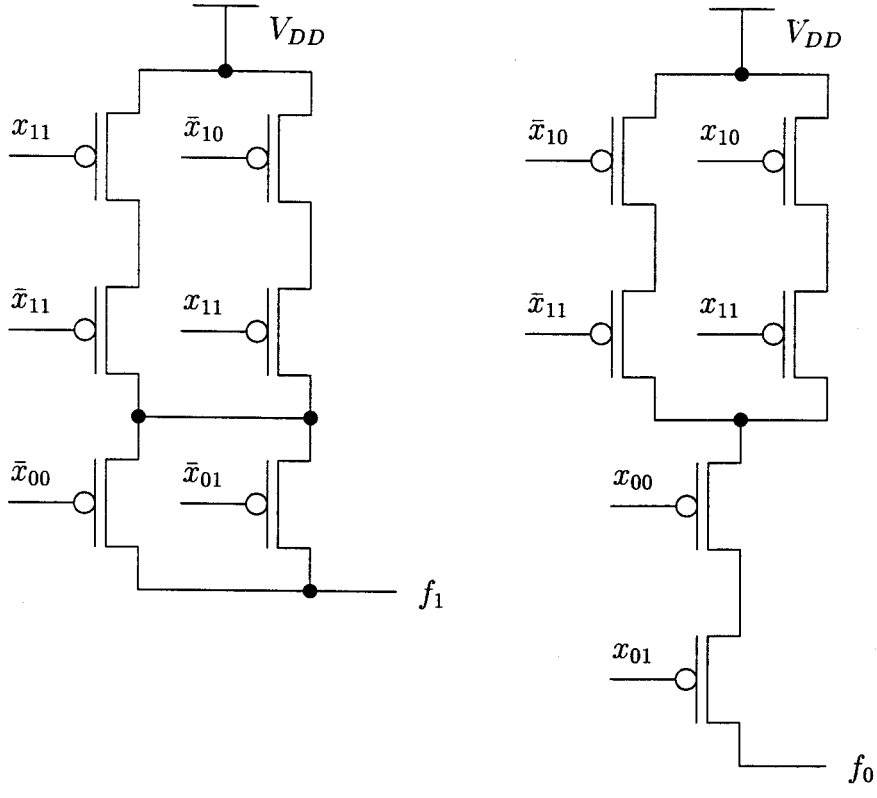



**Figure 7.17** The PMOS structure of circuit realization of Figure 7.16($b$).

$f_0$ can be expressed as:

$$f_1 = (\bar{x}_{11}x_{10} + x_{11}\bar{x}_{10})(x_{01} + x_{00}), \quad \text{and} \tag{7.17}$$

$$f_0 = (\bar{x}_{11}\bar{x}_{10} + x_{11}x_{10})\bar{x}_{01}\bar{x}_{00} \tag{7.18}$$

As shown in Figure 7.17, the circuit realization of these two binary functions needs 36 transistors ($12 \times 2$ plus 3 inverters). That is, the CMOS binary implementation of the same function uses 8 more transistors than the self-restored MVL design as shown in Figure 3.14, but is smaller than Chang's realization (50 transistors as discussed in Section 7.1).

Considering the fact that MVL design can reduce interconnection area by a factor of $\log_2 r$ for one-dimensional effect and $(\log_2 r)^2$ for two-dimensional effect, self-restored MVL design appears superior to binary logic design in terms of circuit size. The 500 random MVL functions, which were used for comparison of the self-restored MVL design and the operator-based MVL design, were used for the comparison with the binary logic design. The equivalent binary logic circuits as well as their transistor count and Spectre netlists were obtained following procedures in Figure 5.13.

Table 7.4 lists transistors counts of the 500 equivalent binary logic circuits. As mentioned previously in Section 7.1, the transistor count of each function is a fractional number because the transistor sizes and interconnection area within library cells were also considered by the Synopsys Design Compiler. The average transistor count of the 500 binary logic circuits is 46.67. This number is about 58.6% of the average transistor counts of the MVL circuits obtained in Section 7.1. This seems contradictory to the theories of optimal logic radix. However, it should be noted that interconnections are not taken into consideration in the above comparison. According to reports [70, 71], interconnections can occupy as much as 70% of a chip area designed with binary logic circuits. Therefore, the average total circuit size of an equivalent binary logic circuit is about $46.67/0.3 = 155.57$. The average total circuit size of a

**Table 7.4** Transistor count of equivalent binary logic circuits of the 500 MVL functions of Table 7.2.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 50.0 | 53.2 | 38.8 | 50.8 | 54.8 | 50.0 | 39.6 | 59.6 | 44.0 | 69.20 |
| **2** | 46.4 | 30.8 | 46.0 | 32.0 | 45.2 | 58.4 | 33.2 | 40.8 | 42.0 | 57.60 |
| **3** | 40.0 | 54.0 | 50.0 | 55.6 | 35.6 | 53.2 | 56.8 | 60.4 | 59.6 | 50.00 |
| **4** | 45.6 | 34.0 | 54.0 | 46.0 | 45.6 | 51.2 | 43.6 | 54.0 | 32.0 | 53.20 |
| **5** | 42.4 | 45.2 | 37.6 | 38.0 | 46.8 | 49.2 | 39.2 | 43.6 | 34.0 | 32.80 |
| **6** | 55.6 | 49.6 | 57.6 | 46.0 | 60.4 | 56.4 | 38.8 | 54.8 | 39.6 | 56.80 |
| **7** | 52.4 | 43.6 | 27.6 | 34.0 | 47.2 | 51.2 | 46.8 | 38.8 | 53.6 | 49.20 |
| **8** | 44.8 | 62.8 | 32.8 | 60.8 | 39.6 | 42.0 | 72.8 | 47.2 | 40.0 | 50.80 |
| **9** | 49.2 | 32.4 | 52.4 | 48.0 | 36.0 | 50.0 | 48.8 | 48.0 | 30.8 | 33.60 |
| **10** | 64.8 | 44.0 | 43.6 | 36.0 | 53.6 | 54.0 | 56.8 | 42.8 | 46.0 | 39.60 |
| **11** | 38.8 | 48.8 | 44.4 | 51.6 | 36.0 | 50.0 | 44.4 | 43.2 | 46.4 | 29.60 |
| **12** | 74.4 | 42.8 | 38.4 | 40.0 | 57.2 | 54.0 | 50.8 | 52.0 | 45.6 | 25.60 |
| **13** | 57.2 | 53.2 | 55.6 | 47.6 | 32.8 | 45.6 | 43.2 | 36.0 | 49.2 | 26.40 |
| **14** | 43.6 | 49.6 | 44.4 | 42.8 | 38.0 | 46.8 | 46.0 | 40.0 | 60.8 | 53.20 |
| **15** | 44.4 | 52.0 | 41.2 | 63.2 | 69.2 | 54.4 | 55.2 | 36.4 | 49.6 | 51.60 |
| **16** | 38.0 | 36.0 | 52.4 | 39.2 | 47.2 | 49.6 | 54.0 | 68.4 | 43.2 | 44.40 |
| **17** | 44.0 | 30.4 | 61.2 | 33.6 | 44.4 | 45.2 | 61.6 | 41.2 | 60.0 | 47.20 |
| **18** | 38.8 | 64.8 | 42.0 | 55.6 | 33.2 | 53.2 | 39.2 | 47.2 | 50.4 | 57.60 |
| **19** | 60.4 | 48.0 | 38.0 | 52.8 | 52.0 | 49.2 | 65.6 | 48.8 | 59.2 | 47.60 |
| **20** | 38.8 | 46.8 | 50.0 | 56.4 | 40.4 | 40.8 | 44.8 | 52.4 | 50.4 | 47.20 |
| **21** | 55.2 | 46.4 | 58.0 | 60.0 | 34.8 | 52.4 | 30.8 | 54.4 | 43.6 | 46.80 |
| **22** | 33.6 | 54.4 | 59.6 | 58.4 | 58.0 | 47.2 | 43.2 | 28.8 | 47.6 | 47.20 |
| **23** | 49.2 | 45.2 | 33.6 | 51.2 | 30.0 | 33.6 | 42.0 | 42.0 | 31.6 | 42.40 |
| **24** | 48.4 | 45.6 | 59.6 | 51.2 | 33.2 | 53.6 | 46.4 | 30.8 | 54.0 | 41.20 |
| **25** | 50.0 | 46.4 | 52.0 | 48.0 | 40.8 | 43.6 | 40.4 | 42.8 | 44.4 | 45.60 |
| **26** | 48.0 | 46.8 | 40.8 | 46.8 | 50.0 | 62.4 | 54.8 | 57.2 | 40.0 | 42.80 |
| **27** | 55.2 | 38.0 | 52.4 | 42.4 | 42.0 | 44.8 | 51.2 | 43.6 | 58.4 | 53.20 |
| **28** | 46.4 | 47.2 | 30.0 | 53.2 | 45.6 | 48.0 | 59.2 | 50.0 | 46.8 | 51.20 |
| **29** | 54.0 | 41.6 | 52.8 | 48.4 | 43.6 | 47.6 | 48.4 | 45.2 | 39.6 | 54.00 |
| **30** | 53.2 | 43.2 | 20.0 | 56.4 | 42.8 | 44.0 | 48.0 | 51.2 | 32.8 | 32.80 |
| **31** | 48.4 | 56.4 | 52.4 | 46.8 | 43.6 | 48.8 | 50.0 | 53.2 | 42.0 | 28.80 |
| **32** | 43.6 | 41.6 | 41.6 | 50.4 | 51.2 | 42.0 | 43.2 | 38.0 | 55.2 | 41.20 |
| **33** | 51.2 | 38.8 | 44.4 | 55.2 | 49.2 | 61.6 | 39.2 | 63.2 | 51.6 | 48.80 |
| **34** | 28.4 | 45.2 | 38.8 | 40.0 | 35.2 | 50.0 | 44.0 | 51.6 | 46.4 | 53.60 |
| **35** | 45.2 | 48.0 | 37.2 | 51.2 | 48.4 | 34.8 | 59.6 | 46.0 | 68.4 | 36.00 |
| **36** | 36.4 | 49.2 | 45.2 | 44.0 | 49.2 | 53.2 | 43.2 | 53.2 | 65.2 | 48.80 |
| **37** | 40.0 | 36.4 | 50.8 | 38.4 | 50.4 | 40.0 | 60.4 | 40.8 | 48.4 | 48.80 |
| **38** | 34.8 | 46.8 | 45.2 | 54.4 | 53.6 | 48.0 | 34.8 | 59.2 | 52.4 | 42.00 |
| **39** | 58.8 | 37.2 | 36.8 | 47.2 | 38.8 | 56.4 | 41.2 | 49.6 | 38.0 | 43.20 |
| **40** | 52.0 | 51.6 | 32.0 | 51.6 | 37.2 | 28.4 | 43.2 | 50.0 | 34.4 | 40.00 |
| **41** | 48.4 | 39.2 | 42.0 | 39.2 | 34.0 | 40.0 | 47.2 | 53.2 | 48.0 | 52.40 |
| **42** | 67.2 | 52.4 | 44.0 | 44.0 | 55.2 | 43.2 | 46.0 | 44.8 | 38.8 | 50.40 |
| **43** | 36.0 | 46.4 | 42.0 | 56.8 | 48.0 | 57.2 | 57.2 | 49.2 | 46.8 | 50.00 |
| **44** | 61.2 | 35.2 | 44.0 | 36.0 | 27.2 | 42.0 | 46.0 | 36.8 | 58.0 | 45.60 |
| **45** | 45.2 | 44.4 | 32.8 | 27.2 | 43.2 | 50.4 | 52.0 | 36.8 | 54.4 | 61.60 |
| **46** | 25.6 | 48.8 | 47.2 | 50.0 | 47.6 | 26.8 | 44.0 | 29.6 | 46.0 | 44.80 |
| **47** | 36.8 | 54.4 | 50.4 | 54.0 | 39.2 | 56.0 | 48.4 | 66.0 | 65.6 | 52.80 |
| **48** | 48.0 | 44.8 | 47.6 | 34.8 | 50.4 | 48.0 | 50.0 | 57.6 | 28.4 | 47.60 |
| **49** | 38.0 | 30.4 | 50.8 | 58.4 | 51.6 | 51.2 | 42.0 | 55.6 | 62.0 | 53.20 |
| **50** | 51.2 | 50.4 | 38.8 | 63.6 | 26.8 | 54.4 | 45.2 | 48.0 | 48.4 | 40.80 |

2-variable 4-valued logic function is about $79.62+(155.57\times 0.7)/(\log_2 4)^2 = 106.84$ to $79.62+(155.57\times 0.7)/\log_2 4 = 134.07$. If 50% of a chip area is occupied by interconnections, the average total circuit size of an equivalent binary logic circuit is about $46.67/0.5 = 93.34$. The average total circuit size of a 2-variable 4-valued logic function is about $79.62+(93.34\times 0.5)/(\log_2 4)^2 = 91.29$ to $79.62+(93.34\times 0.5)/\log_2 4 = 102.96$. If 30% of a chip area is occupied by interconnections, the average total circuit size of an equivalent binary logic circuit is about $46.67/0.7 = 66.67$. The average total circuit size of a 2-variable 4-valued logic function is about $79.62 + (66.67 \times 0.3)/(\log_2 4)^2 = 84.62$ to $79.62 + (66.67 \times 0.3)/\log_2 4 = 89.62$. Let $k$ be the ratio of interconnection area on a chip, $a$ be the one-dimensional ratio, and $b$ be the two-dimensional ratio. The following equation will be satisfied for 2-variable 4-valued logic functions:

$$\frac{46.67}{(1-k)} = 79.62 + \frac{46.67}{(1-k)\times(a\log_2 4 + b(\log_2 4)^2)} \times k. \tag{7.19}$$

That is,

$$k = \frac{32.95\times(2a+4b)}{79.62\times(2a+4b) - 46.67} \tag{7.20}$$

If $a = 1$ and $b = 0$, i.e., two-dimensional interconnection area is not considered, then $k \approx 58.54\%$. In other words, if the percentage of interconnection area is greater than 58.54, the average self-restored MVL circuit size is smaller than that of equivalent binary circuits. The value of $k$ decreases when $a$ decreases ($b$ increases) as shown in the following table.

| $a$ | 1.0 | 0.9 | 0.8 | 0.7 | 0.6 | 0.5 |
|---|---|---|---|---|---|---|
| $b$ | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 |
| $k$ | 58.54 | 56.42 | 54.76 | 53.43 | 52.34 | 51.43 |

As discussed in Section 7.1 the average transistor count (79.62) of the 500 self-restored MVL designs are obtained by using the WMS program according to the self-restored architecture of Figure 3.7. As shown in Figure 7.18, binary $v_j$ subfunc-

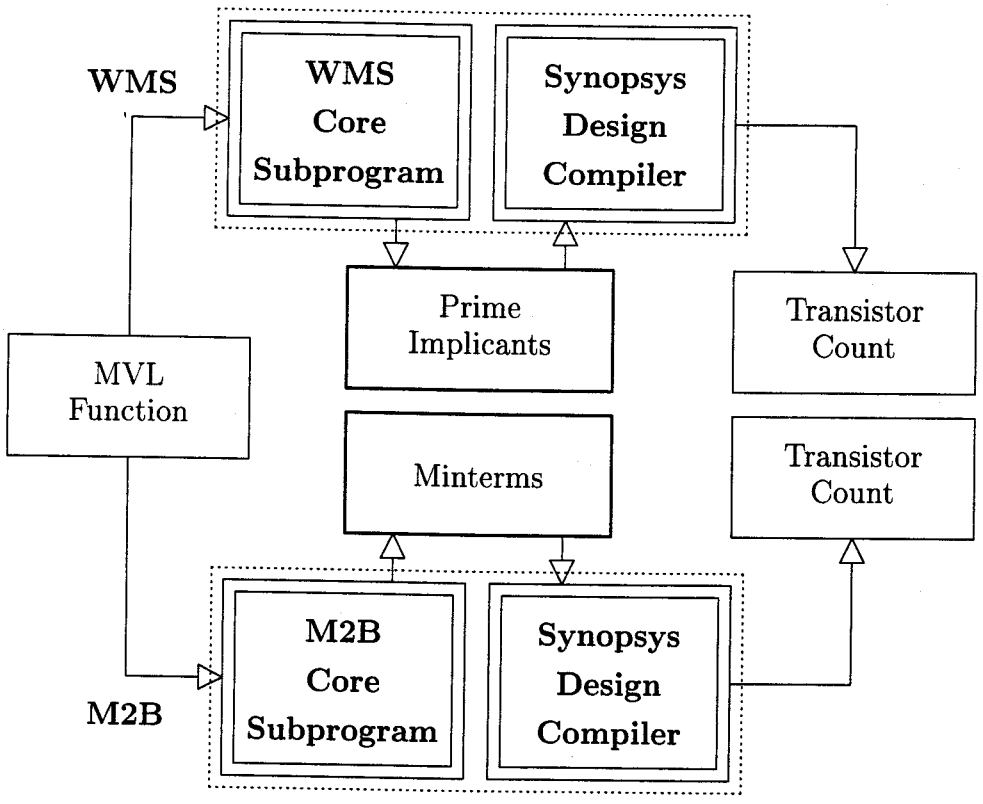



**Figure 7.18** Synthesis flow of a MVL function in terms of transistor count using the WMS program and the M2B program.

tions in terms of prime implicants are passed to the Synopsys Design Compiler for optimization in terms of area in the MVL case, while equivalent binary functions in terms of minterms are used in the binary logic case. As described in Chapter 5, a binary logic synthesizer may not remove redundant prime implicants as it does not recognize the relationships between the *up literal* outputs, $x_0, x_1, \cdots, x_{n-1}$. Therefore, the resulting circuits are slightly larger. Such a problem does not exist in the binary logic case because equivalent binary functions are to be synthesized by a binary logic synthesizer. The Synopsys Design Compiler is able to optimize equivalent binary functions in terms of area no matter whether the binary functions are expressed in terms of minterms or redundant prime implicants. As described previously in Chapter 4, the circuit size of self-restored MVL designs can be further reduced by choosing

an appropriate output block. For example, Figure 4.4($b$) shows proper use of output blocks can result in smaller circuit size. The tradeoff of removing redundant prime implicants or choosing a better output block is that multiple runs of the Synopsys Design Compiler are needed. More than one iteration between the WMS program and a binary logic synthesizer is needed to obtain a smaller circuit size for a given MVL function. Such a procedure is not implemented in the current version of the WMS program.

In addition, if the NMOS self-restored architecture of Figure 3.9 is used, the size of the control block will be reduced to half. An accurate number would better be obtained by using a dedicated synthesis library for the architecture, but the design of a synthesis library needs a substantial amount of work. Alternatively, an estimation of the average transistor count for 2-variable 4-valued functions implemented with the NMOS self-restored architecture shown in Figure 3.9 is carried out based on the result of 79.62 for the self-restored architecture of Figure 3.7. Assume the largest input/output blocks; that is, each input needs a 3-output current mirror and three thresholds and the output block has three switches and three current sources. The total number of transistors of the input and output blocks according to Figure 3.7 is $2(3+4)+3+3=20$. When the self-restored architecture of Figure 3.9 is used, the number of transistors in the NMOS configuration block is $(79.62-20)/2 = 29.81$. Therefore, the estimated average transistor count of 2-variable 4-valued functions implemented with the self-restored architecture of Figure 3.9 is $20-3+28.81 = 46.81$. Three transistors are deducted in the above calculation because the three switches in the output block are not required in the architecture of Figure 3.9. This number is similar to the average transistor count of equivalent binary logic circuits. The conventional operator-based design is approximately 1.9 to 4.3 times larger than that of the NMOS self-restored architecture shown in Figure 3.9.

For certain MVL functions the circuit size can be reduced significantly by using the *sum* and *diff* operations in the input block or the output block as discussed in

**Table 7.5** Comparison of the 12 examples in Chapter 4 and their equivalent binary circuits.

| Examples | Figure 4.7 | | | Figure 4.9 | | | Figure 4.11 | | | Figure 4.13 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | (*a*) | (*b*) | (*c*) | (*a*) | (*b*) | (*c*) | (*a*) | (*b*) | (*c*) | (*a*) | (*b*) | (*c*) |
| with *sum/diff* | 5 | 13 | 20 | 9 | 18 | 21 | 10 | 18 | 19 | 31 | 26 | 20 |
| without *sum/diff* | 31 | 41 | 21 | 26 | 50 | 34 | 17 | 26 | 28 | 53 | 37 | 63 |
| Binary | 22 | 24 | 18 | 12 | 18 | 28 | 10 | 14 | 24 | 30 | 30 | 40 |

Chapter 4. The transistor counts of the twelve MVL examples shown in the Chapter 4 and their equivalent binary circuits are summarized in the Table 7.5. The average transistor count of MVL design using the arithmetic operators is 17.50 while the the average transistor count of equivalent binary logic circuit is 22.50. The average transistor count of MVL design without using the arithmetic operators is 35.58.

Since power dissipation varies with frequency, multiple runs of transient analyses at different frequencies have to be conducted in order to get an average power dissipation. Considering simulation time and disk space available, 30 of the same 500 functions were simulated using the Cadence Spectre. This number is also recommended by statisticians such that the Central Limit Theorem can safely be applied [68, 69]. Multiple transient analyses ranging from 10MHz to 50MHz for each of the 30 MVL functions and equivalent binary logic functions were simulated by the Cadence Spectre using the Nortel 0.8$\mu$ BiCMOS technology. As mentioned in Section 7.1, the upper bound of the frequency is set to 50MHz because of the propagation delay. For each analysis, an average power dissipation is obtained by multiplying the average current flowing from $V_{DD}$ to $V_{SS}$ with $V_{DD}$ which is 5 volts in this case. Figures 7.19 and 7.20 show two example of total currents from $V_{DD}$ to $V_{SS}$ of two 2-variable 4-valued functions and equivalent binary logic functions. The MVL functions are the same as those shown in Figure 7.7 and 7.8. In both figures, the top sub-figure is

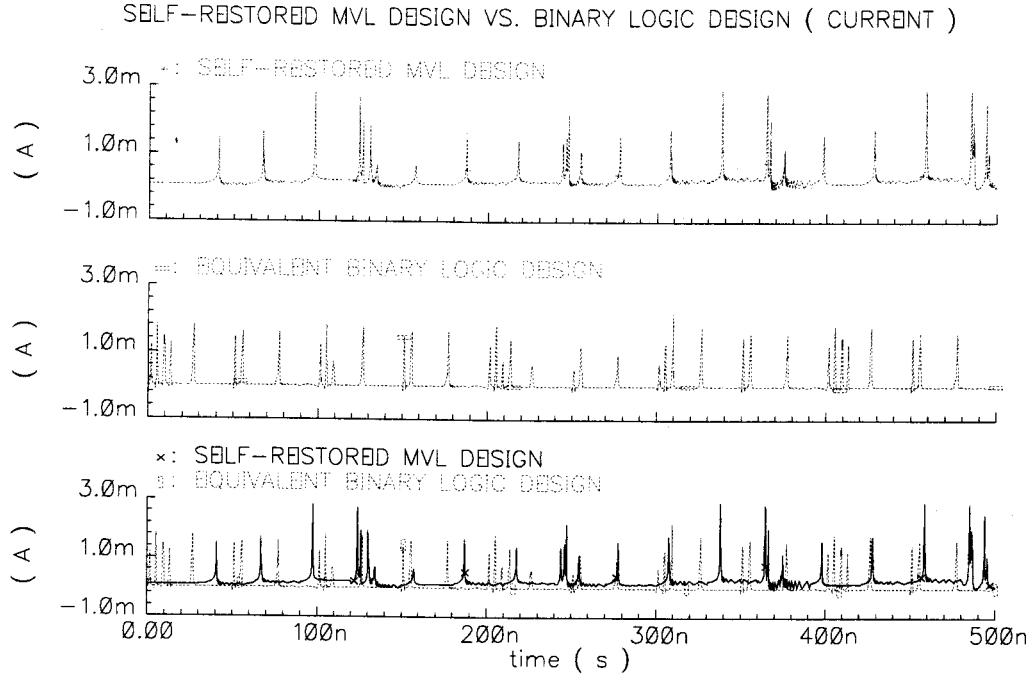



**Figure 7.19** Spectre simulation of transient analysis showing total current from $V_{DD}$ to $V_{SS}$ of a 2-variable 4-valued logic function and its equivalent binary logic function.

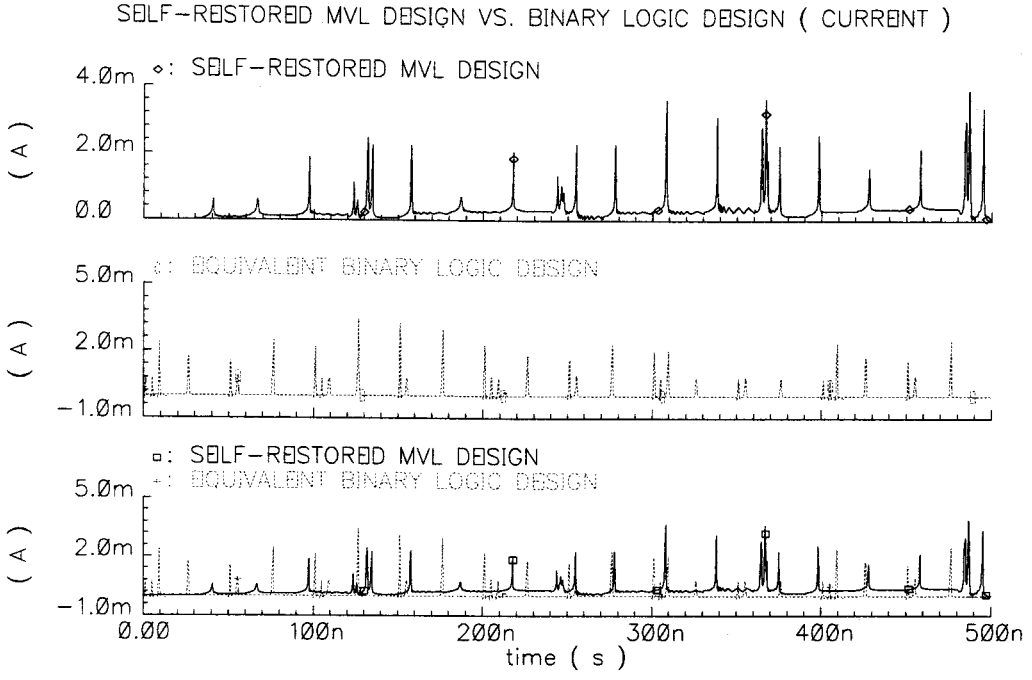



**Figure 7.20** Spectre simulation of transient analysis showing total current from $V_{DD}$ to $V_{SS}$ of a 2-variable 4-valued logic function and its equivalent binary logic function.

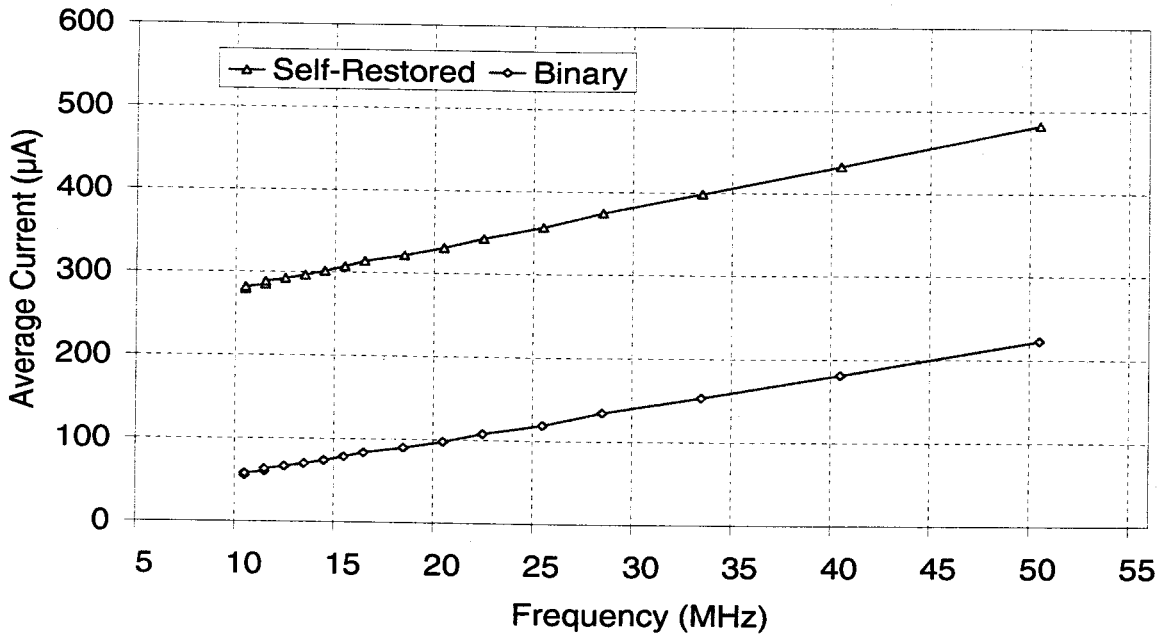



**Figure 7.21** Average current of 30 2-variable 4-valued logic functions and equivalent binary logic functions from 10MHz to 50MHz.

the current waveform of the self-restored MVL design, the middle sub-figure is the current waveform of a equivalent binary logic design, and the bottom sub-figure is combined waveforms of the self-restored MVL design and the equivalent binary logic design for comparison.

The average current of the 30 2-variable 4-valued logic circuits with buffers and equivalent binary logic circuits from 10MHz and 50MHz is shown in Figure 7.21 and the data is shown in Table 7.6. The average current of the MVL circuits is from 279$\mu$A at 10MHz to 482.81$\mu$A at 50MHz. The average current of the binary logic circuits is 55.73$\mu$A at 10MHz to 222.67$\mu$A at 50MHz. That is, the MVL circuits consume on average about 5.0 times more power than the binary circuits at 10MHz and 2.17 times more power at 50MHz.

A possible way to reduce the power dissipation of the self-restored architecture is to use 10$\mu$A as the base current. However, lower base current slows down threshold

**Table 7.6** Average current ($\mu$A) of 30 2-variable 4-valued logic functions and equivalent binary functions at frequencies from 10MHz to 50MHz.

| Frequency (MHz) | MVL ($\mu$A) | Binary ($\mu$A) |
|---|---|---|
| 10.0 | 279.12 | 55.73 |
| 10.5 | 281.95 | 57.94 |
| 11.1 | 284.94 | 60.50 |
| 11.8 | 288.49 | 63.36 |
| 12.5 | 292.26 | 66.49 |
| 13.3 | 296.34 | 70.06 |
| 14.3 | 301.56 | 73.79 |
| 15.4 | 307.02 | 78.51 |
| 16.7 | 313.78 | 83.74 |
| 18.2 | 321.26 | 89.90 |
| 20.0 | 330.77 | 97.61 |
| 22.2 | 342.32 | 107.69 |
| 25.0 | 356.62 | 118.77 |
| 28.6 | 374.76 | 134.41 |
| 33.3 | 399.04 | 153.21 |
| 40.0 | 432.75 | 181.22 |
| 50.0 | 482.81 | 222.67 |

circuit elements and is more susceptible to noise and process variations. Alternatively, by using the NMOS configuration for the control/output block, a considerable amount of power can be saved when circuits operate at higher frequencies due to reduced dynamic power dissipation. Figures 7.22 and 7.23 shows average currents of two 2-variable 4-valued functions realized by the self-restored architecture of Figure 3.7 (labeled as SR STD), the self-restored architecture using the NMOS configuration (labeled as SR NMOS), and equivalent binary logic circuit (labeled as Binary). The average current of the SR NMOS consumes less power than SR STD, but SR NMOS still consumes more power than equivalent logic circuits.

Spectre simulation results also show that MVL circuits are slower than binary circuits. The rise/fall time delays and propagation delays of self-restored MVL designs

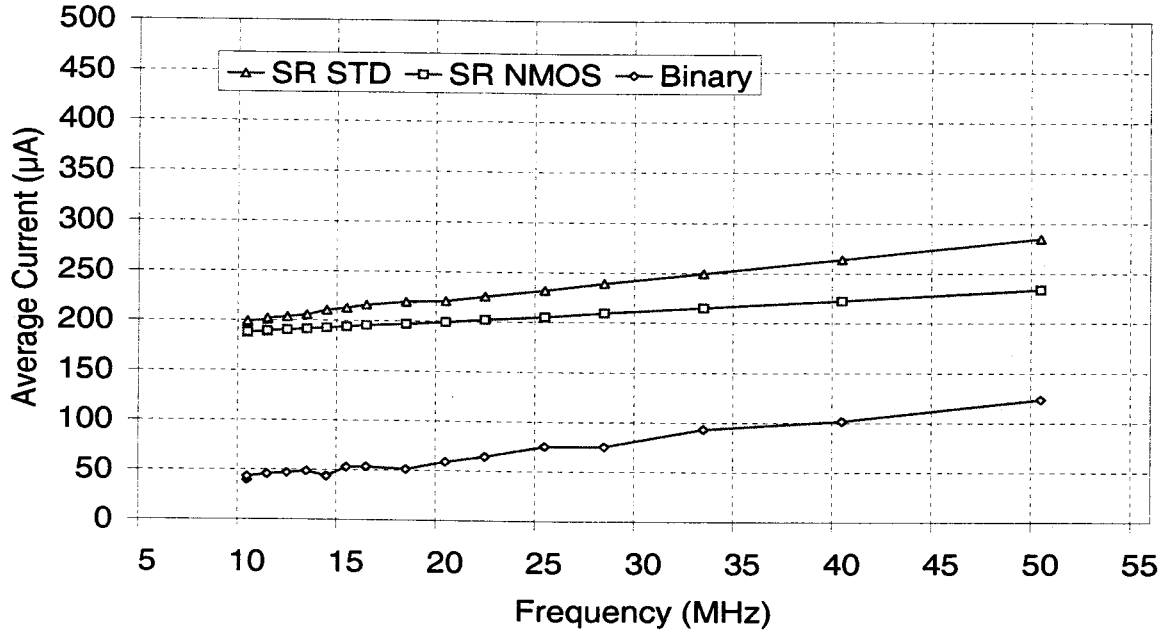



**Figure 7.22** Power dissipation of the same 2-variable 4-valued logic function shown Figure 7.5 from 10MHz to 50MHz.

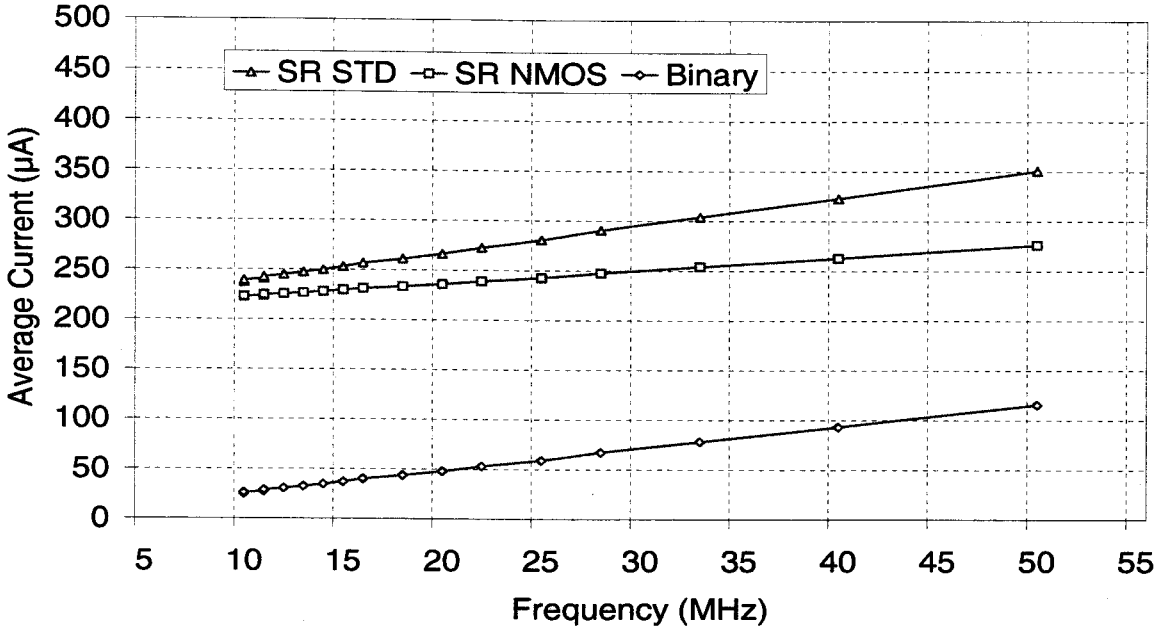



**Figure 7.23** Power dissipation of the same 2-variable 4-valued logic function shown Figure 7.6 from 10MHz to 50MHz.

are on the order of $10ns$, while the time delays of equivalent binary logic circuits are on the order of $1ns$. That is, the MVL circuits are about 10 times slower than binary circuits. Such a difference is mainly caused by the threshold circuit elements.

## 7.3 Circuit Examples

In Sections 7.1 and 7.2, comparison between the self-restored design and the conventional operator-based design as well as with equivalent binary logic design are discussed in terms of area, speed and power. The comparison is based on pseudo-randomly generated 2-variable 4-valued functions. The circuit examples in Chapters 3 and 4 are not particularly for certain practical functions either. As such, this section presents three practically used circuits to conclude the description of the self-restored design architecture.

### 7.3.1 Majority Circuit

The general equation $F$ for a majority circuit with $2n+1$ inputs can be expressed as:

$$F = \begin{cases} 1 & \text{when } X_1 + X_2 + \cdots + X_{2n+1} \geq n+1 \\ 0 & \text{otherwise} \end{cases} \tag{7.21}$$

where $n > 1$ and $X_0, X_1, \cdots, X_{2n+1} \in \{0, 1\}$. That is, when more than half of the inputs are 1, output is 1. Circuit realization of this equation according to the self-restored design architecture is shown in dashed lines in Figure 7.24. The input block circuit consists of a single-output P-type current mirror and an N-type threshold. The control/output block consists of a switch and a current source. The input and control/output blocks use 5 transistor. This number does not increase with the number of inputs as far as the total current to the P-type current and the N-type threshold is under the allowable maximum current of devices. As shown in Figure 7.24, the majority circuit can also interface with binary logic design by including $2n+1$ switches

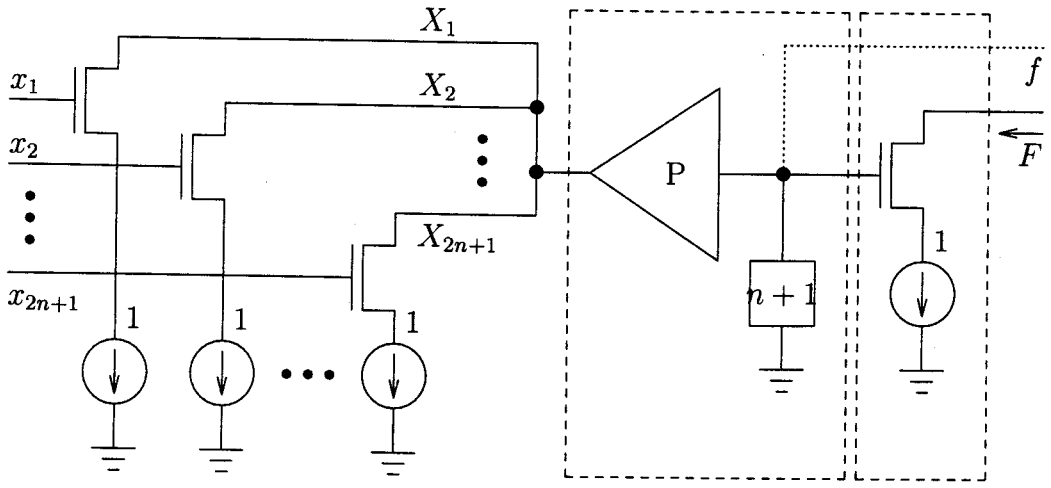



**Figure 7.24** CMCL realization of a majority circuit according to the self-restored architecture.

and current sources at the input end with each switch controlled by a binary input $x_i$. The binary output $f$, which is shown in a dotted line in the figure, is

$$f = \begin{cases} 1 & \text{when } x_1 + x_2 + \cdots + x_{2n+1} \geq n+1 \\ 0 & \text{otherwise} \end{cases} \tag{7.22}$$

where $+$ is arithmetic sum. The total number of transistors including the interface is $2(2n+1)+5 = 4n+7$. For example, a 3-input majority circuit ($n=1$) consists of 11 transistors and a 5-input majority circuit ($n=2$) consists of 15 transistors.

VMCL implementation of a 3-input binary majority circuit and a 5-input binary majority circuit is shown in Figure 7.25. As shown in the figure, the 3-input binary majority circuit uses 12 transistors and the 5-input majority circuit uses 38 transistors. Both circuits are larger than the corresponding CMCL majority circuits according to the self-restored architecture. The 3-input binary majority circuit uses one more transistor than the 3-input CMCL majority circuit, while the 5-input binary majority circuits is 2.53 times larger than the 5-input CMCL majority circuit. The difference in circuit size becomes larger with the increase of the number of inputs.

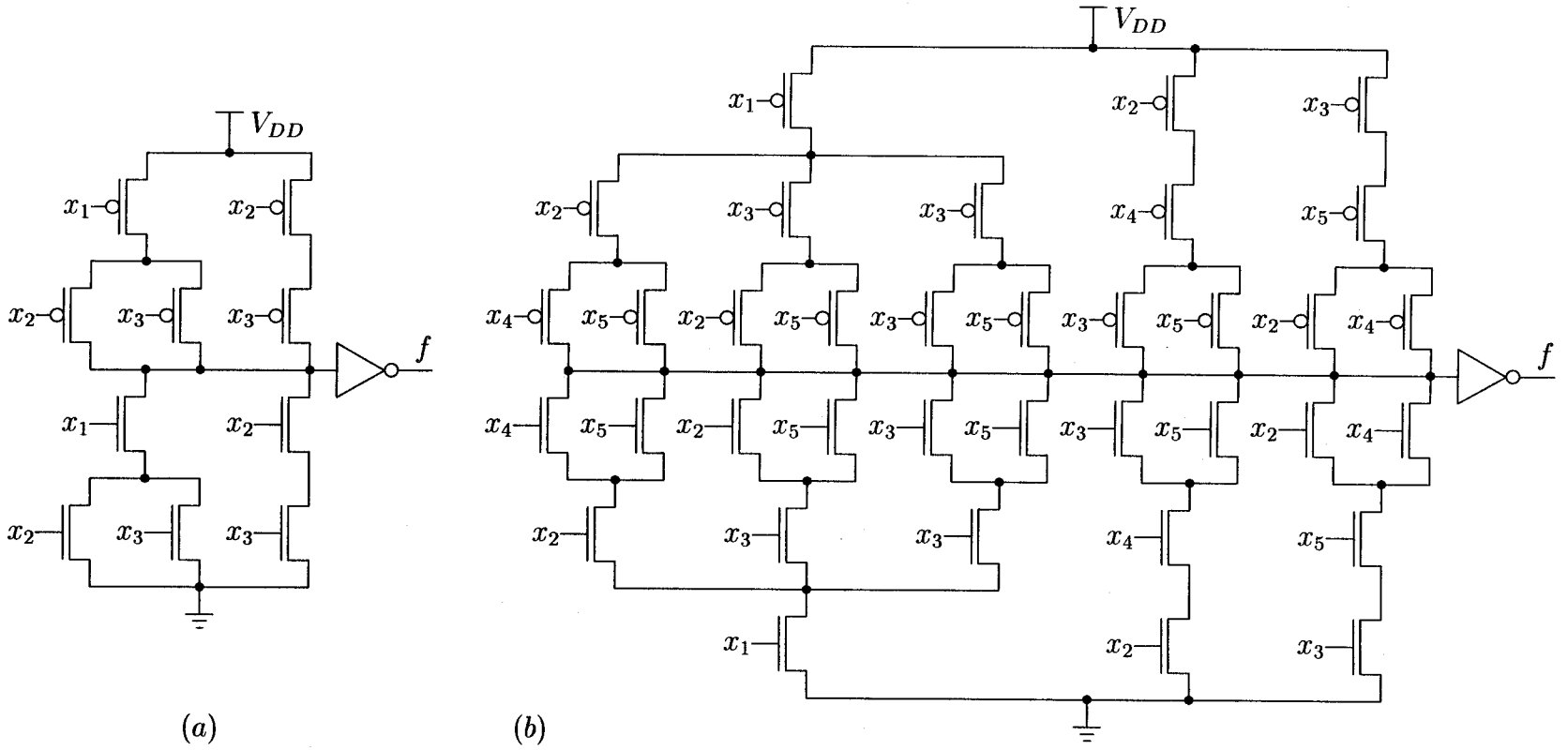



**Figure 7.25** VMCL realization of binary major circuits. ($a$) 3-input majority circuit. ($b$) 5-input majority circuit.

### 7.3.2 Ripple Carry Adder

Let $A$ and $B$ be addends, $C_i$ carry in, $S$ sum and $C_o$ carry out. Figure 7.26($a$) shows the truth table of a ternary half-adder (THA) and a ternary full-adder (TFA), and Figure 7.26($b$) shows the truth tables of a quaternary half-adder (QHA) and a quaternary full-adder (QFA), where $U = A + B + C_i$. $U$ is in the range of 0 to $2r - 1$ for a $r$-valued full-adder. For the THA, $U$ is in the range of 0 to 4 since $A$ and $B$ are in the range of 0 to 2 and $C_i$ is 0. For the TFA, $U$ is in the range of 0 to 5 since $A$ and $B$ are in the range of 0 to 2 and $C_i$ is either 0 or 1. Similarly, $U$ is in the range of 0 to 6 for the QHA since $A$ and $B$ are in the range of 0 to 3 and $C_i$ is 0, and $U$ is in the range of 0 to 7 for the QFA since $A$ and $B$ are in the range of 0 to 3 and $C_i$ is either 0 or 1.

A THA can be expressed as a single-input 2-output 5-valued functions and A TFA can be expressed as a single-input 2-output 6-valued functions. Similarly, a QHA

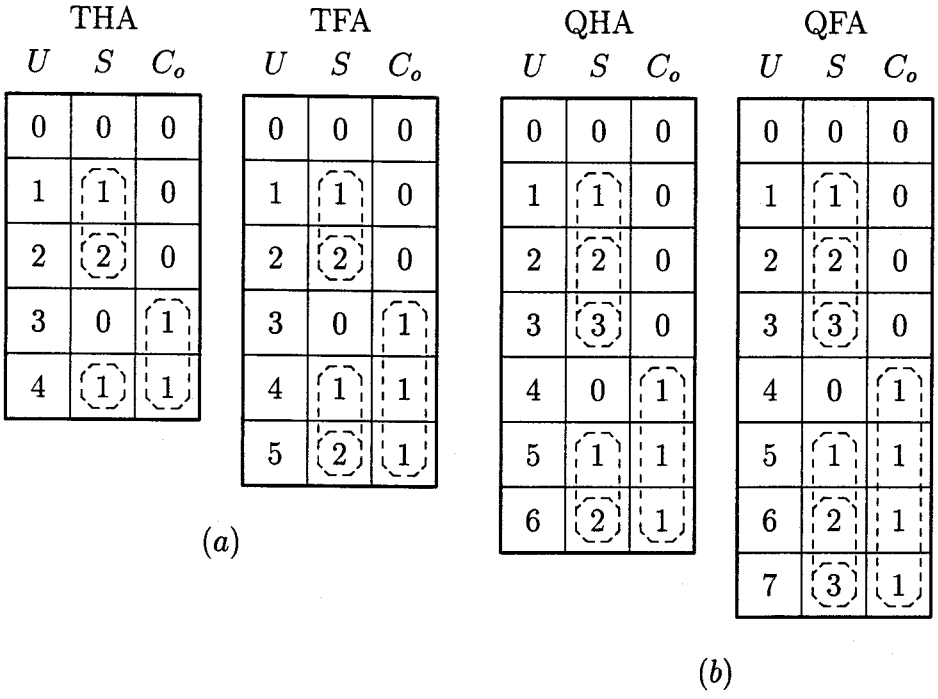



**Figure 7.26** Truth tables of 1-bit adders. ($a$) Ternary half-adder (THA) and ternary full-adder (TFA). ($b$) Quaternary half-adder (QFA) and quaternary full-adder (QFA).

becomes a single-input 2-output 7-valued function and a QFA becomes a single-input 2-output 8-valued function. Referring to the truth table of Figure 7.26($a$), the binary $v_j$ subfunctions for $S$ of the TFA can be expressed as

$$\begin{aligned} v_1' &= x_1\bar{x}_3 + x_4 \\ v_1'' &= x_2\bar{x}_3 + x_5 \end{aligned}$$

the binary $v_j$ subfunction for $C_o$ of the TFA can be expressed as

$$v_1 = x_3 \tag{7.23}$$

Circuit realization of the TFA is shown in Figure 7.27($a$). The wiring of $x_1, x_2, x_3, x_4, x_5$ to the switches in the control/output block is not shown in the figure. Only a corresponding signal name is shown at the control terminal of each switch. It should be also noted that each P-type switch could be replaced with an N-type switch and an inverter. The total number of transistors is 23 (21 plus an inverter). A THA can be obtained by removing the N-type switch controlled by $x_5$ in the output block, and the total number of transistors of the resulting circuit is 22.

Similarly, referring to the truth table of Figure 7.26($a$), the binary $v_j$ subfunctions for $S$ of the QFA can be expressed as

$$\begin{aligned} v_1' &= x_1\bar{x}_4 + x_5 \\ v_1'' &= x_2\bar{x}_4 + x_6 \\ v_1''' &= x_3\bar{x}_4 + x_7 \end{aligned}$$

the binary $v_j$ subfunction for $C_o$ of the QFA can be expressed as

$$v_1 = x_4 \tag{7.24}$$

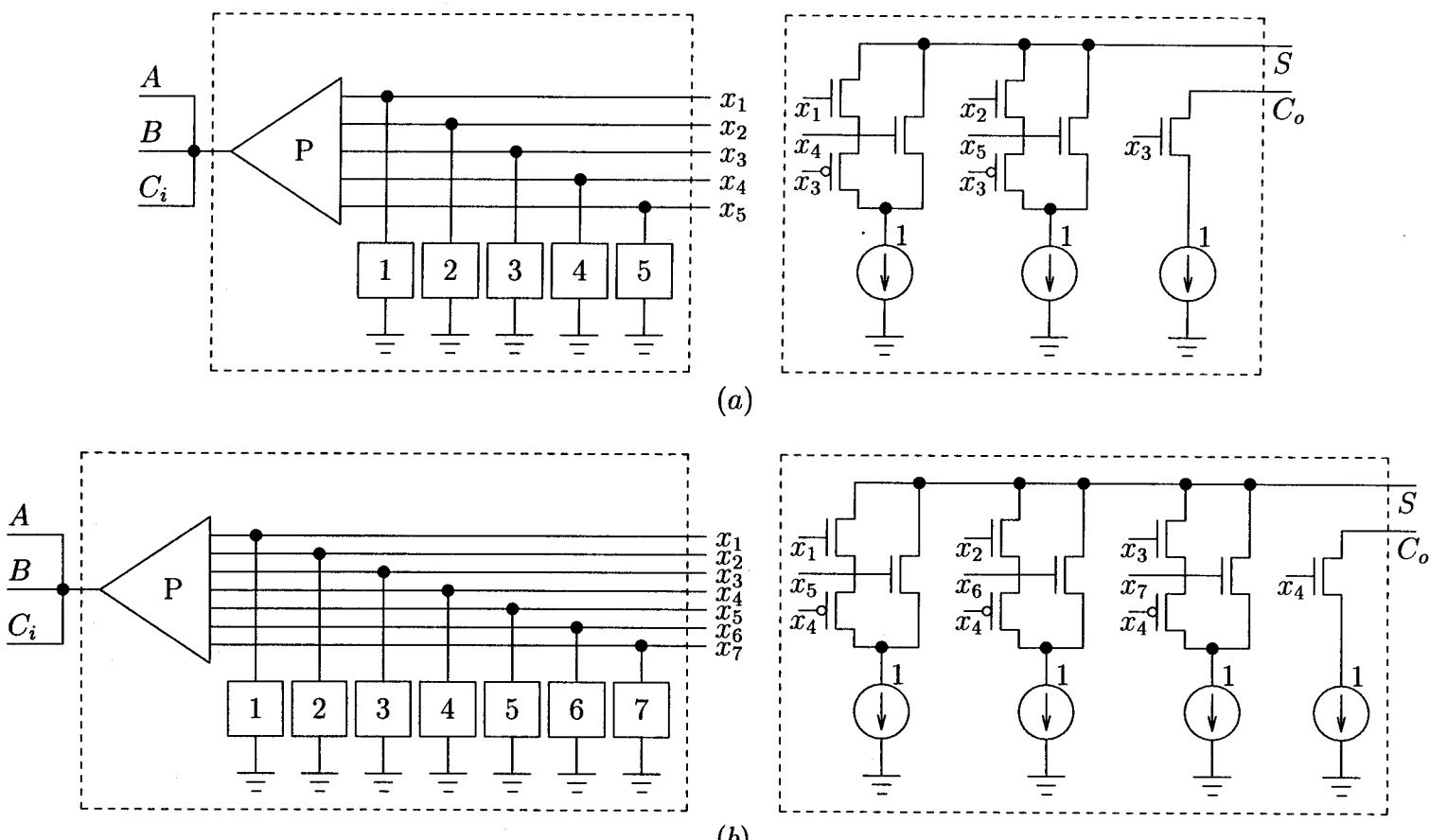



**Figure 7.27** CMCL realization of ripple carry adders according to the self-restored design architecture. (*a*) Ternary full-adder. (*b*) Quaternary full-adder.

Circuit realization of the QFA is shown in Figure 7.27($b$). The wiring of $x_1, x_2, \cdots, x_5$ to the switches in the control/output block is again omitted, and only a signal name is shown at the control terminal of each switch. The total number of transistors is 31 (29 plus an inverter). A QHA can be constructed by removing the N-type switch controlled by $x_7$ and the total number of transistor is 30. An $n$-bit ripple carry adder can be constructed by cascading $n$ 1-bit adders by connecting $C_o$ of bit $i$ to $C_i$ of the bit $i+1$. The total transistors of a $n$-bit quaternary ripple carry adders is therefore $31n - 1$. A transistor is deducted from the equation because the least significant bit (bit 0) uses a half-adder.

A 1-bit quaternary adder is equivalent to a 2-bit binary adder. The following comparison will show that the circuit size of a CMCL quaternary full-adder is smaller than that of two binary adders. Two different circuit realizations of the binary full-adder are shown in Figure 7.28 and 7.29 [72]. Figure 7.28 shows a typical VMCL circuit realization of the binary full-adder. Figure 7.29 shows a VMCL circuit realization of the binary full-adder using transmission gates. Both circuits use 28 transistors. The circuit size of two binary full-adders is about 1.8 times larger than a quaternary full-adder. As described above, an $n$-bit ripple carry adder can be constructed by cascading $n$ 1-bit adders. In general, a binary $n$-bit ripple carry adder will always be approximately 1.8 times larger than the size of an equivalent quaternary ripple carry adder ($n/2$ bits).

### 7.3.3 Tally Circuit

The function of a binary tally circuit is to count the numbers of bits that are set to logic high in $n$-bit inputs. The tally circuit is important to a digital implementation of the matched filter that is briefly discussed in Section 6.3.2. It is reported in [73] that a 32-input 33-output binary tally circuit constructed with steering logic [74] using transmission gates requires 2,304 transistors, where only one of the outputs is logic high at any time; i.e., if an output $y_i$ is logic high when there are $i$ bits of inputs are

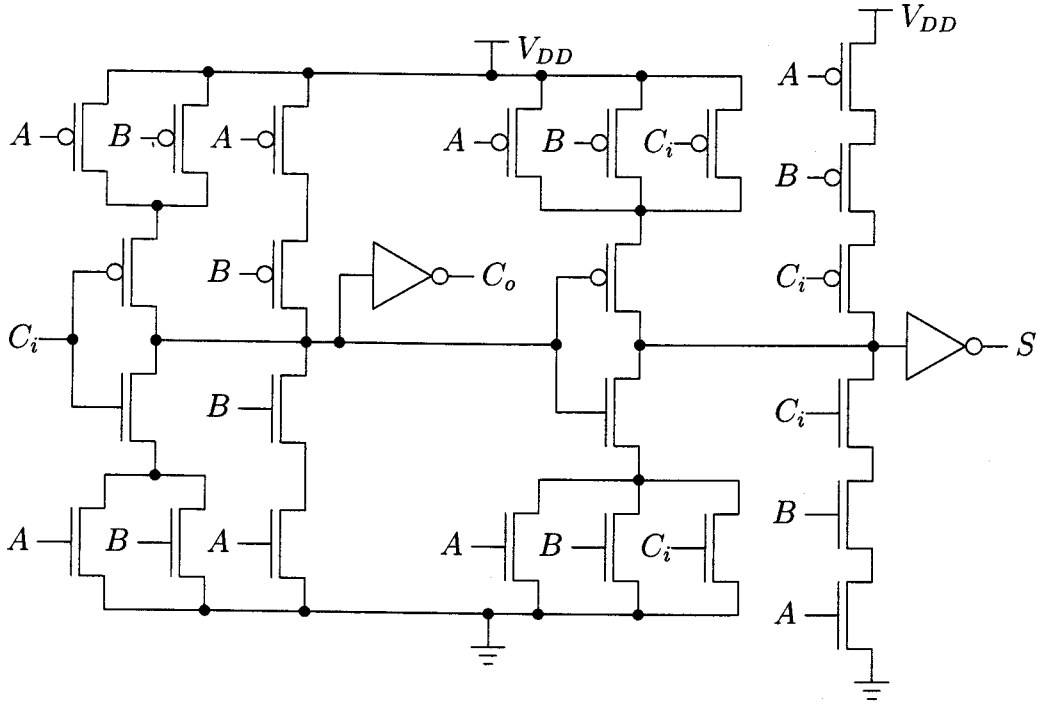



**Figure 7.28** A circuit realization of the binary full-adder.

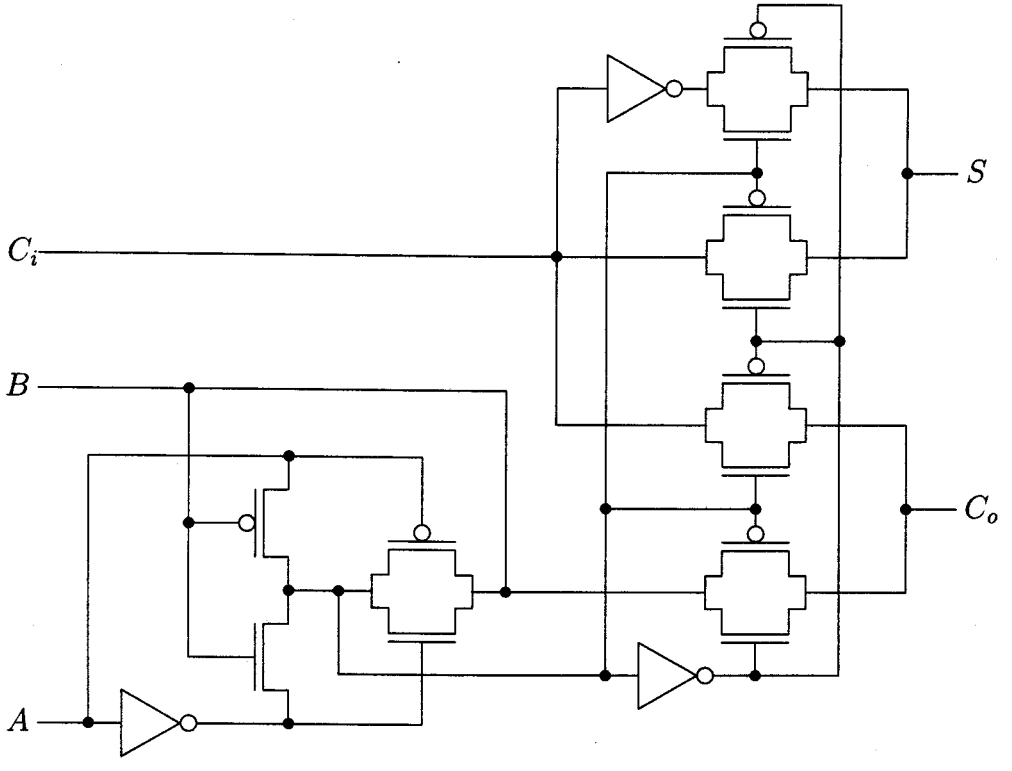



**Figure 7.29** A circuit realization of the binary full-adder using transmission gates.

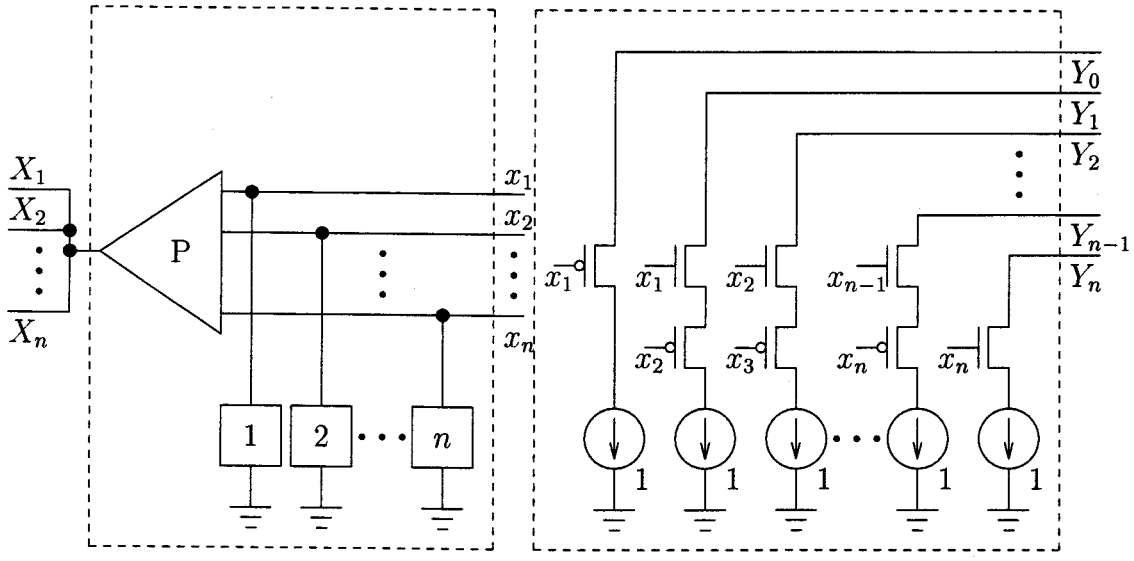



**Figure 7.30** Circuit realization of a $n$-input $n+1$ output tally circuit according to the self-restored design architecture.

logic high. If none of the inputs are logic high, then $y_0$ is logic high. A CMCL circuit realization of the $n$-input $n+1$ output tally circuit is shown in Figure 7.30, where the input block consists of a $n$-output current mirros and $n$ threshold elements while each output $Y_i$ is connected to a logic 1 current source through two series-connected switches. It should be noted that each P-type switch could be replaced with an N-type switch and an inverter. The sum of inputs, $X_1 + X_2 + \cdots + X_n$, tells how many input bits are at logic 1. When the sum of inputs, $X_1 + X_2 + \cdots + X_n$, is 0. $x_1, x_2, \cdots, x_n$ are logic low and therefore, only $Y_0$ is 1. When the sum of inputs is 1, only $x_1$ is logic high and therefore, $Y_1$ is 1. When the sum of inputs is $i$, $x_1, x_2, \cdots, x_i$ are logic high and $x_{i+1}, x_{i+2}, \cdots, x_n$ are logic low and therefore only $Y_i$ is 1. When all the inputs are 1, only $x_n$ is logic high and therefore only $Y_n$ is 1. It can be seen from Figure 7.30 that the input block uses $2n+1$ transistors and the output block uses $5n+1$ transistors including $2n$ switches, $2n$ inverters and $n+1$ current soruces. The total transistor count of the CMCL $n$-input $n+1$ output tally circuit is $7n+2$. A 32-input 33-output tally circuit requires 226 transistors, which is less than 10% of the equivalent binary tally circuit using transmission gates.

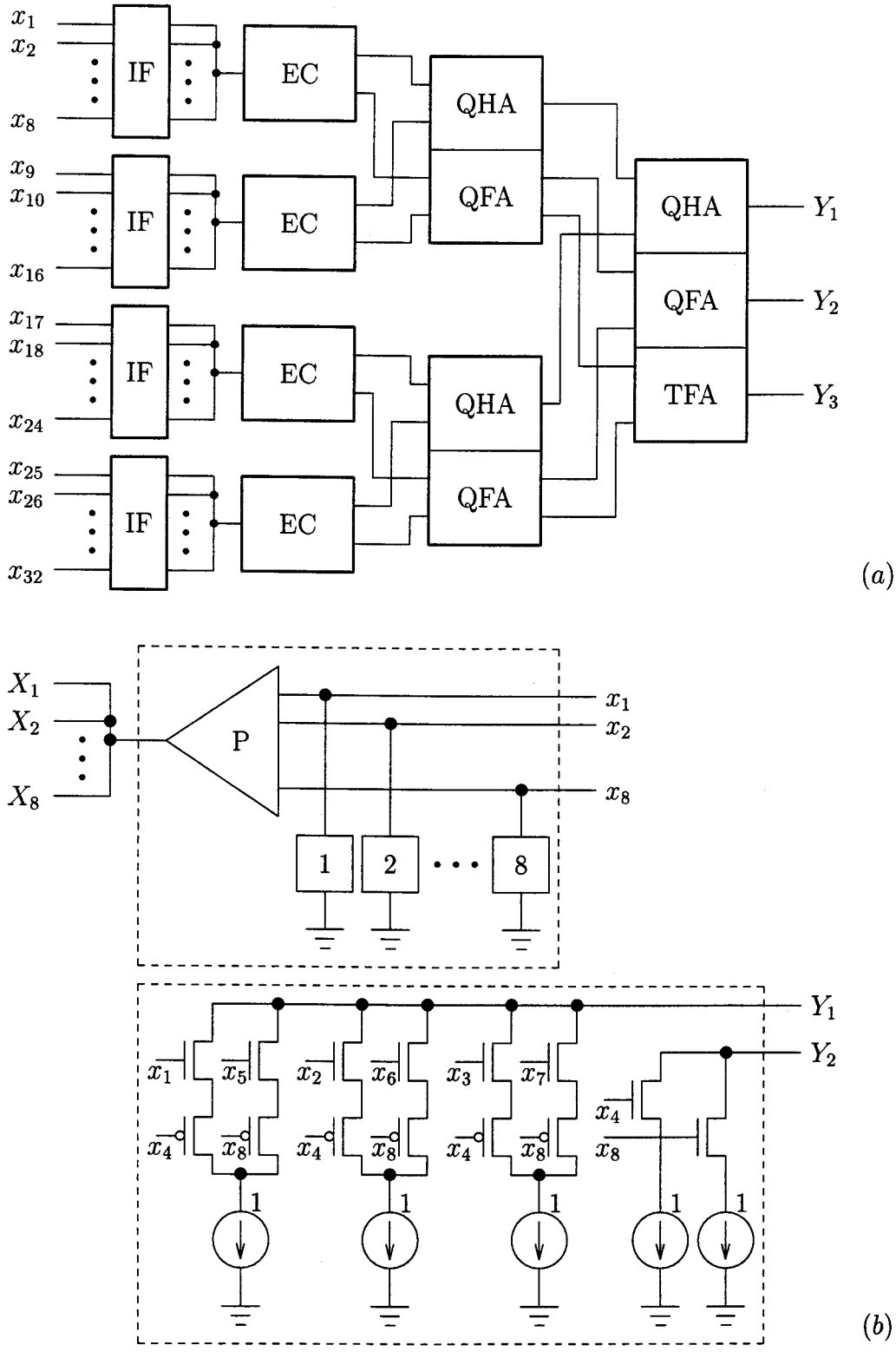



**Figure 7.31** CMCL realization of a 32-bit tally circuit using the adders according to the self-restored design architecture. (*a*) Block diagram. (*b*) Encoder circuit.

Figure 7.31($a$) shows the block diagram of a 32-bit quaternary tally circuit. It should be noted that this tally circuit has 3 quaternary outputs, $Y_1$, $Y_2$ and $Y_3$. For example, when 5 of inputs are at logic 1, $Y_3Y_2Y_1$ is quaternary $011_4$; when 16 of inputs are at logic 1, $Y_3Y_2Y_1$ is $100_4$. The largest output for $Y_3Y_2Y_1$ is $200_4$, that is equivalent to decimal 32. The 32-bit binary inputs are converted to current signals using four 8-bit interface (IF) circuit. The IF circuit is the same as the one used in Figure 7.24 where each input controls a switch that is connected a logic 1 current source. The output from IFs are fed to encoder (EC) circuits to obtain 2-bit quaternary outputs. Figure 7.31($b$) is the CMCL realization of the EC circuit according to the self-restored design architecture. It should be noted that the signals, $x_1, x_2, \cdots, x_8$ and $Y_1, Y_2$ in Figure 7.31($b$) and Figure 7.31($a$) are different signals. The encoded 2-bit outputs are summed up by a 2-bit quaternary adder that consists of a QHA and a QFA. The outputs from the two 2-bit adders are then summed up by a 3-bit quaternary adder that consists of a QHA, a QFA and a TFA. A TFA is used at the most significant bit because the largest number from a 2-bit quaternary adder at the previous stage is quaternary $100_4$ that is equivalent to decimal 16. The transistor count of the 32-bit CMCL tally circuit is calculated as follows. The four 8-bit IFs require $4 \times 8 \times 2 = 64$ transistors. The four ECs requires $4 \times 34 = 136$ transistors. Assume a QHA and a QFA are the same in size. The total number of transistors of six QFA and a TFA is $6 \times 31 + 23 = 209$. As a result, the 32-bit quaternary tally circuit use 409 transistors, which is about 1/5 the size of the 32-bit binary tally circuit using transmission gates.

## 7.4 Summary

This chapter presents comparison results of the self-restored MVL design with the operator-based MVL design and with the binary logic design. The comparison results are discussed from three aspects: area, timing, and power. 500 pseudo-randomly generated MVL functions were used to obtain an average circuit size in terms of

transistor count. A selection (30) of the 500 MVL functions were then simulated by using the Cadence Spectre circuit simulator to obtain average power dissipation and timing information. The same 500 MVL functions were also used to obtain an average transistor count of equivalent binary logic circuits. Equivalent binary logic circuits of the same 30 MVL functions were then simulated by using Spectre to obtain an average power dissipation and timing information.

The comparison results show that the self-restored MVL design architecture can implement MVL functions without sacrificing circuit size, power, and speed. The average transistor count of 500 2-variable 4-valued functions according to the self-restored architecture of Figure 3.7 is 79.62 with using the CMC BiCMOS Synopsys library. The average transistor count of equivalent operator-based circuits is at least 1.1 to 2.5 times larger dependent upon how many unary operators are in one product term. Compared with the estimated circuit size of the self-restored MVL design using NMOS configuration for the control/output blocks, the conventional operator-based MVL design is 1.9 to 4.3 times larger. The time delays are close to each other because threshold circuit elements dominate time delays in both cases. As for the power dissipation, self-restored MVL designs consume similar amount of power with operator-based MVL designs at 50MHz, while consumes less power at lower frequency.

The comparison results with binary logic design show that the self-restored MVL design architecture of Figure 3.7 is competitive to the binary logic design in terms of circuit size when both circuits and interconnections are considered. Dependent upon the ratio of the one-dimensional effect and two-dimensional effect, the average circuit size of 2-variable 4-valued logic functions is smaller than that of equivalent binary logic functions when the ratio of interconnection area to circuit area is greater than 58.54. The comparison results also show that the circuit size of equivalent binary logic circuits is similar to estimated circuit size of the self-restored MVL design using NMOS configuration for the control/output blocks without considering interconnection area. The circuit size can be further reduced with a better choice of an output block and

using *sum* and *diff* operations. Circuit simulations using the Cadence Spectre show the self-restored MVL architecture is about 10 times slower as a result of the threshold circuit elements. As for the power dissipation, the MVL circuits consume about 5.0 times more power than the binary circuits at 10MHz and 2.17 times more power at 50MHz.

# 8. Conclusions and Future Work

This thesis proposed a new MVL design architecture along with a synthesis scheme. The MVL design architecture is characterized by realizing MVL functions with smaller average circuit size and self-restored output signals, which are the two primary objectives of the thesis. A theoretical analysis was carried out to determine the possibility of using a binary logic synthesizer for MVL synthesis according to the self-restored MVL design architecture, which is the third objective of the thesis. A computer program was developed for this design architecture based on the theoretical analysis. Working together with a binary logic synthesizer, the program generates an area-optimized circuit for a given MVL function according to the self-restored MVL design architecture. An additional computer program was also designed to automatically derive equivalent binary logic circuits for a given MVL function for comparison purposes. By using the computer programs, this thesis compared the self-restored MVL design architecture and its variants with other MVL design schemes and with binary logic design in three aspects: area, speed and power. In addition, in order to verify synthesized results, a new VHDL library was designed for fast functional verification of CMCL design with a VHDL simulator, which is the fourth and the final objective of the thesis.

## 8.1 Conclusions

The proposed MVL design architecture, consisting of a current-mode input block, a voltage-mode control block and a current-mode output block, provides self-restored signals because outputs always come from the current sources directly. Moreover,

MVL functions implemented with this design architecture result in smaller circuit size than the conventional operator-based MVL designs using *min*, *max*, *tsum*, *literal*, *cycle*, etc., yet without sacrificing speed and power. Using Monte Carlo simulation results with a 95% confidence interval, 500 2-variable 4-valued logic functions were pseudo-randomly generated for comparing the self-restored design architecture with other MVL design schemes. The average transistor count of 2-variable 4-valued functions implemented with the self-restored architecture is 79.62 using the CMC BiCMOS synthesis library. The average transistor count of equivalent operator-based circuits is 1.1 to 2.5 times larger than the self-restored architecture. The time delay and average power dissipation were similar. The average propagation delay for both the self-restored architecture and the the conventional operator-based designs is $10ns$ approximately, but the self-restored design has shorter rise and fall time delays ($1ns$ approximately) than the operator-base design ($5ns$ approximately). The power dissipation of the self-restored architecture increases with frequency, while the conventional operator-based MVL designs increase with frequency considerably slower because the main source of power dissipation is static power. The self-restored designs consume similar amount of power with the operator-based MVL designs at 50MHz ($2414\mu$W on average), but consumes less power at lower frequency ($1395\mu$W at 10MHz on average).

The self-restored architecture also offers other advantages. Unlike conventional operator-based MVL design schemes which usually needs new minimization algorithms for different sets of operators, a binary logic synthesizer can be used for MVL synthesis. In the self-restored architecture MVL functions are decomposed into $r-1$ disjoint binary subfunctions, and those binary subfunctions are realized in the voltage-mode binary control block. Therefore, a binary logic synthesizer can be used for synthesis of the control block. A computer program, WMS, was developed, which is capable of working with the Synopsys Design Compiler to create a self-restored MVL circuit from a MVL function. The program is able to calculate the size of the

resulting MVL circuit in terms of transistor count for comparing the circuit size of a self-restored MVL design with an equivalent operator-based MVL design or with an equivalent binary logic design. Other features include automatic generation of VHDL and Verilog format circuit files and SPICE netlists. SPICE netlists are necessary to compare time delay and average power dissipation. Since N-type threshold and binary logic gates are used to implement a MVL function, direct interface to binary modules is feasible. There is no need for extra binary-MVL or MVL-binary conversion circuits to interface with other binary circuits.

The self-restored MVL architecture was also compared with equivalent binary logic designs using the same 500 pseudo-randomly generated functions. The comparison results show that the self-restored MVL design architecture is comparable with the binary logic design in terms of circuit size when both circuits and interconnections are considered. For example, the average circuit size of 2-variable 4-valued logic functions is smaller than that of equivalent binary logic functions when the ratio of interconnection area to circuit area is greater than 51.43 (two-dimensional interconnection effect) to 58.54 (one-dimensional interconnection effect). For some of the example logic functions considered, the self-restored architecture results in a smaller transistor count than the binary logic design even without considering interconnection area. The average current (power) of the binary logic circuits is 55.73$\mu$A (278.65$\mu$W) at 10MHz to 222.67$\mu$A (1113.35$\mu$W) at 50MHz. The self-restored MVL circuits consume on average about 5.0 times more power than the binary circuits at 10MHz and 2.17 times more power at 50MHz. The self-restored MVL architecture is about 10 times slower because of the threshold elements.

Many variants to the design architecture were discussed in the thesis. Some of these allow for reduction in circuit size and power dissipation of the self-restored MVL architecture. In the NMOS variant, the control block and the output block can be merged into one block by replacing the binary gates in the control block with NMOS

transistors. The NMOS transistors are connected to current sources functioning as both a switch circuit and a control circuit. This way the size of the control block is cut in half. The estimated transistor count of 2-variable 4-valued function implemented with this architecture is 46.81 on average. This number is comparable to the average transistor count of equivalent binary logic circuits (46.67). The conventional operator-based design is approximately 1.9 to 4.3 times larger than that of the self-restored MVL designs using this NMOS variant architecture. The average power dissipation for the NMOS variant is also smaller because of smaller average circuit size.

The circuit size of the self-restored MVL designs can be further reduced by selecting an appropriate output block and using the *sum* and *diff* operations. The *sum* and *diff* operators can be applied to an input block and/or an output block. With iterative minimization, circuit size of many self-restored MVL designs can be reduced significantly in terms of transistor count because these two arithmetic operations can be realized by simply wiring signals together according to Kirchhoff's Current Law. The tradeoff is that the minimization becomes more complicated. For a self-restored MVL design realized without using the arithmetic operators most of the minimization work is done by a binary logic synthesizer. If the arithmetic operators are used, the WMS program must be able to work interactively with a binary logic synthesizer through multiple runs to determine which set of binary subfunctions results in the smaller circuit. When the *sum* operator is used in an output block along with the appropriate choice of an output block, the resulting circuit size can be reduced by as much as 40% for certain MVL functions. The arithmetic *sum* and *diff* operators can be used in an input block separately or in combination. Some of the example logic functions show that the average transistors count is reduced from 35.58 to 17.50 as compared to the MVL designs without using the arithmetic operators.

The self-restored MVL circuits generated by the WMS program were verified by VHDL simulations using a new VHDL library (CMCL library) for CMCL designs.

The new CMCL library allows faster simulation of a broader range of CMCL circuits as a result of the new node type, the new bus resolution function and the new CMCL circuit element functions. The bus resolution function and the circuit element functions are the most important part of the CMCL library. Library cells are created based on the circuit element functions, and they interact with each other according to the bus resolution function. The new bus resolution function is more concise and robust. Unlike the old versions, the new bus resolution function does not resolve each branch current of a node; it only combines all driving signals, checks current equilibrium and determines binary logic state. Branch currents are now determined by the circuit element functions. This approach is completely different from the conventional usage of a bus resolution function. The new bus resolution function has about 150 lines of code, which is about half the size of the older version. As a bus resolution function is one of the determining factors of the simulation speed, more concise coding means faster simulation. As a result of the new bus resolution function and new node type, a new switch circuit function using the local relaxation method is greatly simplified because transferring circuit information from one side of the switch to the other side becomes much easier.

The CMCL library has basic MVL cells (behavioral), complex MVL cells (behavioral and structural) as well as standard binary logic gates. For a given MVL function, the WMS program automatically generates VHDL description files for the resulting circuit using the CMCL library cells. Instead of using a time-consuming circuit simulator like other researchers do today, a quick functional verification of CMCL circuits can be conducted with the CMCL library. The matching and squaring circuit example shows the Spectre simulation takes about two and half hours on an UltraSparc 143MHz machine with 128MB memory, while the VHDL simulation takes only 15 seconds on the same machine. The VHDL simulation using the CMCL library provides a much faster functional verification than circuit simulation.

## 8.2 Future Work

Significant and successful designs of MVL ICs have been done in the last twenty years, including MVL $I^2L$ and ECL circuits, MVL ROMs and CMOS MVL circuits. However, as pointed out by D. Etiemble [47],

> "the tremendous progress of binary circuit performance is the major reason for the difficulties to establish a niche for MVL circuits in the binary world."

Implementation techniques of MVL circuits must be further improved. Some future work is proposed as follows:

**1. Deep-submicron implementation**

Current-mode CMOS logic (CMCL) has been believed to be a better candidate for circuit realization of MVL functions because CMCL allow higher logic radix and better signal integrity. Unfortunately, these advantages are disappearing as the feature size of CMOS technologies are becoming smaller. One of the reasons is that the maximum source-drain current ($I_{DS}$) is decreasing. For a 2-volt, 0.25$\mu$ CMOS technology, $I_{DS}$ is as low as 100$\mu$A to 150$\mu$A for a minimum sized transistor. This value is barely large enough for 4-valued design where the base current is 20$\mu$A. Reduction of the base current might not be a viable solution because of the variation of $I_{DS}$ caused by short-channel effects [12, 47, 75] and the nature of submicron technologies [76]. It has been reported that the variations in $I_{DS}$ for deep-submicron technologies could be as high as 60%. In view of this variation it cannot be guaranteed that the existing circuit designs would still function correctly when technologies are scaled down to submicron or deep-submicron, let alone satisfying timing constraints and power budget. The self-restored current-mode MVL design architecture might be the best chance for MVL circuits to find a niche in the binary world while CMOS still dominates the IC technologies. However, the self-restored MVL circuits must be verified with deep-submicron processes.

## 2. Circuit Realization with Nanoelectronic Devices

The advance of MVL technology depends much on the development of devices that are inherently suitable for MVL, just as CMOS for binary VLSI systems. The application of negative differential resistance (NDR) devices to MVL design have long been studied by MVL researchers [77]. Nanoelectronic devices are promising candidates due to their multiple stable states. Among nanoelectronic devices, resonant tunneling devices are attracting increasing attention due to their potential application to practical circuits [78, 79, 80, 81].

The output current of a resonant tunneling device can be made to peak at one or more values of the control voltage. Moreover, the widths of each peak and valley in the current-voltage ($I - V$) characteristic can be programmed by the appropriate choice of tunnel barrier/quantum well material, dimensions and doping densities. These properties make resonant tunneling devices attractive for implementing MVL systems. Resonant tunneling diodes (RTD) are a type of resonant tunneling device. RTD-based MVL circuits have been applied to memory [82], analog-to-digital converters [83], and multiplexers [84]. Among such RTD-based MVL circuits Waho et al. [85, 86, 87] propose a monostable-to-multistable transition logic (MML) quantizer, in which series-connected RTDs are used to produce multiple-valued signals. Because the MML quantizer has multiple thresholds, a possible use of the MML quantizer in the self-restored MVL design architecture is to replace thresholds in the input block and current sources in the output block resulting in smaller circuit size.

Another interesting nanoelectronic device is quantum dot (QD). QD devices are still under development. The $I$-$V$ curve of a QD shows that the current is in the range of $p$A [88]. One interesting application of QDs to MVL is the superpass-transistor (SPT) built on a conceptual lateral-resonant-tunneling QD transistor (LQT). The SPT is a voltage-controlled analog switch which has multiple ON-OFF switching states corresponding to different logic values of the control voltage. The switching state corresponding to a logic value of the control voltage is programmed in the device

structure. A possible realization of the SPT is proposed in [89, 90]. The SPT could be used in the self-restored MVL design architecture to replace multiple switches in the output block and to further reduce the circuit size.

### 3. Power Estimation of MVL designs

By power estimation we generally refer to the estimation of the average power dissipation of a circuit. Chip heating and temperature are directly related to the average power [91]. Power dissipation can be estimated at different levels, such as system-level, behavioral-level or register-transfer level (RTL), gate-level, switch-level, and circuit-level. System-level normally provides the fastest power estimation with least accuracy, while circuit-level normally provides the most accurate power estimation with the lowest speed. As pointed out in [92], level-by-level power analysis and estimation can provide faster and more accurate results.

Research on high-level (including system-level and RTL) power estimation techniques on binary logic design has just begun [93, 92, 94, 95], while gate-level estimation techniques on binary logic design are more mature for practical use. For example, an event-driven simulation algorithm for gate-level estimation on binary logic design was discussed in [96], which reports a speed increase of 2 to 3 orders of magnitude over SPICE. It is interesting to examine the possibility of applying the existing gate-level estimation techniques to the MVL designs so that a fast power comparison between different MVL design schemes or between a MVL design scheme and the binary logic design can be done without using a time-consuming circuit simulator such as SPICE.

### 4. MVL Synthesis Library

A synthesis library should be also developed based on the one that the author developed in [12] so that the VHDL simulator for MVL and the synthesizer can share one cell library. EDA tools cannot share libraries unless a standard database is available. A logic synthesizer defines library cells in a "technology file". The formats of technology files are different for logic synthesizers from different vendors.

For example, Synopsys Design Compiler does not understand the technology file for Cadence Ambit. VHDL simulators, on the other hand, can share *source* files provided proprietary PLI (Programming Language Interface) coding is not involved, because VHDL is a standard language. But the compiled designs or libraries are not compatible. The binary files are specific to simulators. For example, Cadence NC-VHDL does not understand a VHDL design compiled by Mentor ModelSim. That is, logic synthesizers use proprietary technology files, while VHDL simulators use the standard VHDL language but the compiled files are not standard. A shared library between a VHDL simulator and a logic synthesizer means unified cell names, port names, etc. If cell names, port names, etc. are the same, a VHDL design can be used for both simulation and synthesis.

The synthesis library should also include cells for the self-restored design architecture using the NMOS configuration for the control/output block as shown in Figure 3.9. The estimated result discussed in Chapter 7 revealed that the average transistor count of 2-variable 4-valued functions using the NMOS configuration is approximately the same as that of equivalent binary logic functions. A synthesis library is required for obtaining a more accurate number of transistors.

### 5. Improvement on the WMS Program

The WMS program can be improved on a variety of aspects. For example, the algorithm for searching prime implicants used by the WMS program needs to be improved because it slows down exponentially with the increase of input variables. Another aspect is the minimization using the *sum* and *diff* operators in the input/output blocks. The examples in Chapter 4 have shown that circuit size can be reduced using the arithmetic operators. However, it can be expected that the development of a minimization algorithm exploiting the two arithmetic operators is technically difficult due to an increased number of possible solutions. A good starting point might be the use of the *sum* operator only, such as restricting the use of the *sum* operator in the output block. Other features that can be implemented in the WMS

program include behavioral synthesis and the capability to handle the don't care condition. By incorporating these features, the WMS program allows MVL design at the behavioral level and in turn a faster design cycle.

# References


[1] K. C. Smith and P. G. Gulak, "Prospects for multiple-valued integrated circuits," *IEICE Trans. on Electronics*, vol. E76-C, pp. 372–382, Mar. 1993.

[2] M. Kameyama, S. Kawahito, and T. Higuchi, "A multiplier chip and multiple-valued bidirectional current-mode logic circuits," *IEEE Computer*, vol. 21, pp. 43–56, Apr. 1988.

[3] G. Agraham, "Variable radix multistable integrated circuits," *IEEE Computer*, vol. 7, pp. 42–59, Sept. 1974.

[4] S. L. Hurst, "Multiple-valued logic - Its status and its future," *IEEE Trans. on Computers*, vol. C-33, pp. 1160–1179, Dec. 1984.

[5] C. M. Allen and D. D. Givone, "The Allen-Givone implementation oriented algebra," in *Computer Science and Multiple-Valued Logic : Theory and Applications*, New York, U.S.A.: North-Holland, 1977.

[6] K. Shimabukuro and M. Kameyama, "Architecture of a parallel multiple-valued arithmetic VLSI processor using adder-based processing elements," *IEICE Trans. on Electronics*, vol. E76-C, pp. 463–471, Mar. 1993.

[7] Z. Zilic and Z. G. Vranesic, "Multiple-valued logic in FPGA," in *Proceedings of the 36th Midwest Symposium on Circuits and Systems*, (Detroit, U.S.A.), pp. 1553–1556, 1993.

[8] Z. Tang, O. Ishizuka, and H. Matsumoto, "Multiple-valued static random-access-memory design and application," *IEICE Trans. on Electronics*, vol. E76-C, pp. 403–411, Mar. 1993.

[9] G. Atwood, A. Fazio, D. Mills, and B. Reaves, "Intel StrataFlash memory technology overview," *Intel Technology Journal*, no. 4, 1997.

[10] A. Fazio and M. Bauer, "Intel StrataFlash memory development and implementation," *Intel Technology Journal*, no. 4, 1997.

[11] S. Karasawa and K. Yamahouchi, "Multi-step function MOS transistor circuits," *IEICE Trans. on Electronics*, vol. E76-C, pp. 357–363, Mar. 1993.

[12] H. Y. Teng, "VHDL simulation of multiple-valued logic," Master's thesis, University of Saskatchewan, Department of Electrical Engineering, Saskatoon, Canada, 1996.

[13] Y. J. Chang and C. L. Lee, "Synthesis of multi-variable mvl functions using hybrid mode CMOS logic," in *Proceeding of the 24th International Symposium on Multiple-Valued Logic (ISMVL)*, (Boston, MA, USA), pp. 35–41, 1994.

[14] C. M. Allen and D. D. Givone, "A minimization technique for multiple-valued logic systems," *IEEE Trans. on Computers*, vol. C17, pp. 964–971, Mar. 1968.

[15] Z. G. Vranesic, E. S. Lee, and K. C. Smith, "A many-valued algebra for switching systems," *IEEE Trans. on Computers*, vol. C19, pp. 964–971, Mar. 1970.

[16] J. M. Yurchak and J. T. Butler, "HAMLET – An expression compiler/optimizer for the implementation of heuristics to minimize multiple-valued programmable logic arrays," in *Proceedings of the 20th International Symposium on Multiple-Valued Logic (ISMVL)*, (Charlotte, U.S.A.), pp. 144–152, May 1990.

[17] A. K. Jain, R. J. Bolton, and M. H. Abd-El-Barr, "CMOS multiple-valued logic design - Part II : Function realization," *IEEE Trans. on Circuits and Systems-I*, vol. 40, pp. 515–522, Aug. 1993.

[18] A. M. Haidar and J. Urawa-Shi, "Optimization of multiple-valued logic functions based on petri nets," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E77-A, pp. 1067–1616, Oct. 1994.

[19] A. M. Haidar and M. Morisue, "Logic synthesis and optimization algorithm of multiple valued logic functions," *IEICE Transactions on Information and Systems*, vol. E77-D, pp. 1106–1114, Oct. 1994.

[20] Y. Hata, N. Kamiura, and K. Yamato, "On input permutation technique for multiple-valued logic synthesis," in *Proceeding of the 25th International Symposium on Multiple-Valued Logic (ISMVL)*, (Bloomington, IN, USA), pp. 170–175, 1995.

[21] Y. Hata, T. Jozumi, and K. Yamato, "Minimization of multiple-valued logic expressions with Kleenean coefficients," *IEICE Trans. on Information and Systems*, vol. E79-D, pp. 189–195, Mar. 1996.

[22] G. Caruso, "Local cover technique for the minimization of multiple-valued input binary-valued output functions," *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E79-A, pp. 110–117, Jan. 1996.

[23] G. W. Dueck and J. T. Butler, "Heat quench algorithm for the minimization of multiple-valued programmable logic arrays," *Computers and Electrical Engineering*, vol. 22, pp. 103–107, Mar. 1996.

[24] D. Etiemble and M. Israel, "Comparison of binary and multivalued ICs according to VLSI criteria," *IEEE Computer*, vol. 21, pp. 28–42, Apr. 1988.

[25] K. C. Smith, "Multiple-valued logic - A tutorial and appreciation," *IEEE Computer*, vol. 21, pp. 17–27, Apr. 1988.

[26] Z. G. Vranesic, "Multivalued signalling in daisy chain bus control," in *Proceeding of the 9th International Symposium on Multiple-Valued Logic (ISMVL)*, (San Francisco, CA, USA), pp. 14–18, 1979.

[27] A. Rushton, *VHDL for Logic Synthesis.* New York, U.S.A.: Wiley, 1998.

[28] A. M. Dewey, *Analysis and Design of Digital Systems with VHDL.* Boston, U.S.A.: PWS Publishing, 1997.

[29] D. R. Coelho, *The VHDL Handbook.* Boston, U.S.A.: Kluwer Academic Publishers, 1989.

[30] R. Cottrell, "High level simulation and hardware description languages," in *Analogue-Digital ASICs : circuit techniques, design tools and applications*, London, U.K.: Peregrinus on behalf of the Institution of Electrical Engineers, 1991.

[31] J. B. Rosser and A. R. Turquette, *Many-Valued Logics.* Amsterdam, Holland: North-Holland, 1952.

[32] A. A. Zinov'ev, *Philosophical Problems of Many-Valued Logic.* Dordrecht, Holland: D. Reidel, 1963.

[33] E. L. Post, "Introduction to a general theory of elementary proposition," *American Journal of Mathematics*, vol. 43, pp. 163–185, 1921.

[34] G. Epstein, *Multiple-Valued Logic Design : An Introduction.* Philadelphia, U.S.A.: Institute of Physics, 1993.

[35] L. Bolc and P. Borowik, *Many-Valued Logics 1 : Theoretical Foundations.* New York, U.S.A.: Springer-Verlag, 1992.

[36] T. Sasao, "EXMIN2: A simplification algorithm for exclusive-OR-sum-of products expressions for multiple-valued-input two-valued-output functions," *IEEE Trans. on Computer-Aided Design*, vol. 12, pp. 621–632, May 1993.

[37] S. Y. H. Su and A. A. Sarris, "The relationalship between multivalued switching algebra and boolean algebra under different definition of complement," *IEEE Trans. on Computers*, vol. C-21, pp. 479–485, May 1972.

[38] J. E. Birk and D. E. Farmer, "An algebraic method for designing multivalued logic circuits using principally binary components," *IEEE Trans. on Computers*, vol. C-24, pp. 1101–1104, Nov. 1975.

[39] K. C. Smith, "The prospects of multivalued logic: A technology and application view," *IEEE Trans. on Computers*, vol. C-30, pp. 619–634, Sept. 1981.

[40] G. Epstein, "The lattice theory of Post algebra," *Trans. of the American Mathematical Society*, vol. 95, pp. 300–317, May 1960.

[41] J. C. Muzio and T. C. Wesselkamper, *Multiple-Valued Switching Theory.* Boston, U.S.A.: A. Hilger, 1986.

[42] A. K. Jain, R. J. Bolton, and M. H. Abd-El-Barr, "CMOS multiple-valued logic design - Part I : Circuit implementation," *IEEE Trans. on Circuits and Systems-I*, vol. 40, pp. 503–514, Aug. 1993.

[43] G. W. Dueck and D. M. Miller, "A 4-valued PLA using the MODSUM," in *Proceedings of 16th International Symposium on Multiple-Valued Logic (ISMVL)*, (Blackburg, U.S.A.), pp. 232–240, May 1986.

[44] H. G. Kerkhoff, *Theory, Design, and Applications of Digital Charge-Coupled Devices.* PhD thesis, Twente University of Technology, Enschede, Netherlands, 1984.

[45] G. W. Dueck, *Algorithms for the Minimization of Binary and Multiple-Valued Logic Functions.* PhD thesis, University of Manitoba, Department of Computer Science, Winnipeg, Canada, 1988.

[46] T. T. Dao, "Threshold $I^2L$ and its application to binary symmetric functions and multivalued logic," *IEEE Journal of Solid-State Circuits*, vol. SC-12, pp. 463–472, Oct. 1977.

[47] D. Etiemble, "On the performance of multivalued integrated circuits : Past, present and future," *IEICE Trans. on Electronics*, vol. E76-C, pp. 364–371, Mar. 1993.

[48] T. Shibata and T. Ohmi, "Neuron MOS voltage-mode circuit technology for multiple-valued logic," *IEICE Trans. on Electronics*, vol. E76-C, pp. 347–356, Mar. 1993.

[49] T. Ohmi and T. Shibata, "The concept of four-terminal devices and its significance in the implementation of intelligent integrated circuits," *IEICE Trans. on Electronics*, vol. E77-C, pp. 1032–1041, July 1994.

[50] K. W. Current, "Current-mode CMOS multiple-valued logic circuits," *IEEE Journal of Solid-State Cricuits*, vol. 29, pp. 95–107, Feb. 1994.

[51] P. E. Allen and D. R. Holberg, *CMOS Analog Circuit Design.* New York, U.S.A.: Rinehart and Winston, Inc., 1987.

[52] R. L. Geiger, P. E. Allen, and N. R. Strader, *VLSI Design Techniques for Analog and Digital Circuits.* New York, U.S.A.: McGraw-Hill, 1990.

[53] K. R. Laker and W. M. C. Sansen, *Design of Analog Integrated Circuits and Systems.* New York, U.S.A.: McGraw-Hill, 1994.

[54] T. Yamakawa, "CMOS multivalued circuits in hybrid mode," in *Proceeding of 15th International Symposium on Multiple-Valued Logic (ISMVL)*, (Kingston, Canada), pp. 144–151, 1985.

[55] M. Sasaki, T. Inoue, Y. Shirai, and F. Ueno, "Fuzzy multiple-input maximum and minimum circuit in current mode and their analyses using bounded-difference equations," *IEEE Trans. on Computer*, vol. C-39, pp. 768–774, June 1990.

[56] T. Yamakawa and T. Mike, "The current-mode fuzzy logic integrated circuits fabricated by the standard CMOS process," *IEEE Trans. on Computer*, vol. C-35, pp. 161–167, Feb. 1986.

[57] M. Sasaki, K. Taniguchi, Y. Ogata, F. Ueno, and T. Inoue, "An implementation of multiple-valued logic and fuzzy logic circuits using 1.5V Bi-CMOS current-mode circuit," *IEICE Trans. on Information and Systems*, vol. E76-D, pp. 571–576, May 1993.

[58] C. Y. Huang, C. J. Wang, and B. D. Liu, "Modular current-mode multiple input minimum circuit for fuzzy logic controllers," *Electronics Letters*, vol. 32, pp. 1067–1069, June 1996.

[59] A. K. Jain, *Multiple-Valued Logic Design in Current-Mode CMOS.* PhD thesis, University of Saskatchewan, Department of Electrical Engineering, Saskatoon, Canada, 1994.

[60] "International technology roadmap for semiconductor: Overall roadmap technology characteristics," 2000.

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

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

[63] W. R. Smith, "Minimization of multivalued functions," in *Computer Science and Multiple-Valued Logic : Theory and Applications*, New York, U.S.A.: North-Holland, 1977.

[64] D. L. Perry, *VHDL.* New York, U.S.A.: McGraw-Hill, 1991.

[65] J. M. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard, *VHDL Designer's Reference.* Boston, U.S.A.: Kluwer Academic Publishers, 1992.

[66] D. Teng, R. J. Bolton, A. K. Jain, R. D. Schmitz, and D. E. Dodds, "A VHDL library for current-mode CMOS multiple-valued logic," in *Proceedings of the 7th IEEE Pacific Rim Conference on Communications, Computers and Signal Processing,* (Victoria, Canada), Aug. 1999.

[67] K. Bult and H. Wallinga, "A class of analog CMOS circuit based on the square-law characteristic of an MOS transistor in saturation," *IEEE Journal of Solid-State Circuits,* vol. SC-22, pp. 357–365, June 1987.

[68] R. H. Leaver and T. R. Thomas, *Analysis and Presentation of Experimental Results.* New York, U.S.A.: Halsted Press, 1974.

[69] J. Devore and R. Peck, *Statistics: The Exploration and Analysis Data.* Belmont, California, U.S.A.: Wadsworth, 1993.

[70] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays.* Boston, U.S.A.: Kluwer Academic Publisher, 1992.

[71] J. T. Butler, *Multiple-valued logic in VLSI design.* Los Alamitos, California, U.S.A.: IEEE Computer Society, 1991.

[72] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design.* Massachusetts, U.S.A.: Addison-Wesley, 1993.

[73] B. Persson, "A mixed-signal ASIC for CDMA code synchronization," Master's thesis, University of Saskatchewan, Department of Electrical Engineering, Winnipeg, Canada, 2001.

[74] R. H. Katz, *Contemporary Logic Design.* Redwood City, CA, U.S.A.: Cummings Publishing, 1994.

[75] Y. P. Tsividis, *Operation and Modelling of the MOS Transistor.* New York, U.S.A.: McGraw-Hill, 1987.

[76] I. Katz, "What system designers need to know about deep-submicron ASICs," *EDN,* vol. 17, pp. 69–73, Feb. 1995.

[77] G. Abraham, "Multiple-valued negative resistance integrated circuits," in *Computer Science and Multiple-Valued Logic : Theory and Applications,* New York, U.S.A.: North-Holland, 1977.

[78] W. W. III, S. B. Enquist, D. H. Chow, H. L. Dunlap, S. Subramaniam, P. Lei, G. H. Bernstein, and B. K. Gilbert, "12 GHz clocked operation of ultralow power interband resonant tunneling diode pipelined logic gates," *IEEE Journal of Solid-State Circuits*, vol. 32, pp. 222–231, Feb. 1997.

[79] J. P. A. van der Wagt, A. C. Seabaugh, and E. A. B. III, "RTD/HFET low standby power SRAM gain cell," *IEEE Electron Device Letters*, vol. 19, pp. 7–9, Jan. 1998.

[80] K. Maezawa, H. Matsuzaki, M. Yamamoto, and T. Otsuji, "High-speed and low-power operation of a resonant tunneling logic gate MOBILE," *IEEE Electron Device Letters*, vol. 19, pp. 80–82, Mar. 1998.

[81] T. P. E. Broekaert, B. Brar, J. P. A. van der Wagt, A. C. Seabaugh, F. J. Morris, E. A. B. III, and G. A. Frazier, "A monolithic 4-bit 2-Gsps resonant tunneling analog-to-digital converter," *IEEE Journal of Solid-State Circuits*, vol. 33, pp. 1342–1349, Sept. 1999.

[82] A. C. Seabaugh, Y. C. Kao, and H. C. Yuan, "Nini-state resonant tunneling diode memory," *IEEE Electron Device Letters*, vol. 13, pp. 479–481, June 1992.

[83] S. J. Wei, H. C. Lin, R. C. Potter, and D. Shupe, "A self-latching A/D converter using resonant tunneling diodes," *IEEE Journal of Solid-State Circuits*, vol. 28, pp. 697–700, June 1993.

[84] H. L. E. Chan, M. Bhattacharya, and P. Mazumder, "Compact multiple-valued multiplexers using negative differential resistance devices," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 1151–1155, Aug. 1996.

[85] T. Waho, "Resonant tunneling transistor and its application to multiple-valued logic circuits," in *Proceedings of the 1995 25th International Symposium on Multiple-Valued Logic*, (Monterey, CA, U.S.A.), pp. 130–138, 1995.

[86] T. Waho, K. J. Chen, and M. Yamamoto, "A novel multiple-valued logic gate using resonant-tunneling devices," *IEEE Electron Device Letters*, vol. 17, pp. 223–225, May 1996.

[87] T. Waho, K. J. Chen, and M. Yamamoto, "Resonant-tunneling diode and HEMT logic circuits with multiple thresholds and multilevel output," *IEEE Journal of Solid-State Circuits*, vol. 33, pp. 268–274, Feb. 1998.

[88] K. Yano, T. Ishii, T. Sano, T. Mine, F. Murai, T. Hashimoto, T. Kobayashi, T. Kure, and K. Seki, "Single-electron memory for giga-to-tera bit storage," *Proceedings of the IEEE*, vol. 87, pp. 633–651, Apr. 1999.

[89] X. Deng, T. Hanyu, and M. Kameyama, "Quentum-device-oriented multiple-valued logic system based on a superpass gate," *IEICE Trans. Information and Systems*, vol. E78-D, pp. 951–958, Aug. 1995.

[90] X. Deng, T. Hanyu, and M. Kameyama, "Multiple-valued logic network using quentum-device-oriented superpass gates and its minimization," *IEE Proc. Circuits, Devices and Systems*, vol. 142, pp. 163–185, Oct. 1995.

[91] F. N. Najm, "A survey of power estimation techniques in VLSI circuits," *IEEE Trans. on VLSI Systems*, vol. 2, pp. 446–455, Dec. 1994.

[92] E. Macii, M. Padram, and F. Somenzi, "High-level power modeling, estimation, and optimization," *IEEE Trans. on Computer-Aided Design*, vol. 17, pp. 1061–1079, Nov. 1999.

[93] L. Luca, R. Hodgson, and P. Siegel, "System-level power estimation and optimization," in *Proceedings of the 1998 International Symposium on Low-Power Design Electronics and Design*, (Monterey, CA, U.S.A.), pp. 173–178, 1998.

[94] M. Nemani and F. N. Najm, "Toward a high-level power estimation capability," *IEEE Trans. on Computer-Aided Design*, vol. 15, pp. 588–598, June 1996.

[95] M. Nemani and F. N. Najm, "High-level area and power estimation for VLSI circuits," in *Proceedings of the 1997 International Symposium on Computer-Aided Design*, (San Jose, CA, U.S.A.), pp. 114–119, 1997.

[96] S. Y. Yuan, K. H. Chen, J. Y. Jou, and S. Y. Kuo, "Static power analysis for power-driven synthesis," *IEE Proceedings: Computers and Digital Techques*, vol. 145, pp. 89–95, Mar. 1998.

[97] P. Antognetti and G. Massobrio, *Semiconductor Device Modeling with SPICE.* New York, U.S.A.: McGraw-Hill, 1988.

[98] U. Çilingiroğlu, *Systematic Analysis of Bipolar and MOS Transistor.* Morwood, MA, U.S.A.: Artech House, 1993.

[99] C. Andre, T. Salama, D. G. Nairn, and H. W. Singor, "Current mode A/D and D/A converter," in *Analog IC Design : The Current-Mode Approach*, London, U.K.: Peregrinus on behalf of the Institution of Electrical Engineers, 1990.

[100] T. Serrano and B. Linares-Barranco, "The active-input regulated-cascode current mirror," *IEEE Trans. on Circuits and Systems I : Fundamental Theory and Application*, vol. 41, pp. 464–467, June 1994.

[101] D. G.Nairn and C. A. T. Salama, "Current-mode algorithmic analog-to-digital converters," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 997–1004, Aug. 1990.

[102] Meta-Software, Inc., Campbell, U.S.A., *HSPICE User's Manual*, Dec. 1992.

[103] S. Bibyk and M. Ismail, "Neural network building blocks for analog MOS VLSI," in *Analogue IC Design : the current-mode approach*, London, U.K.: Peregrinus on behalf of the Institution of Electrical Engineers, 1990.

[104] M. Davio, "Synthesis of discrete functions using $I^2L$ technology," *IEEE Trans. on Computers*, vol. C-30, pp. 653–661, Sept. 1981.

[105] S. P. Onneweer and H. G. Kerkhoff, "High-radix current-mode CMOS circuits based on the truncated-difference operator," in *Multiple-valued logic in VLSI design*, Los Alamitos, California, U.S.A.: IEEE Computer Society, 1991.

# Appendices

Please note that hardcopy of the appendices are not included with the thesis due to space concerns. The appendices are available on the accompanying CD-ROM in PDF or text formats. Along with the appdendices, the CD-ROM also includes the complete source code and the executables of the WMS and M2B programs, as well as the LaTeX files and figures of the thesis.

**CD-ROM Contents**

| **Folders** | **Descriptions** |
|---|---|
| **1. appendices** | Appendices Folder |
| • appendix_a | Appendix A CMCL Elements |
| • appendix_b | Appendix B MVL Operators |
| • appendix_c | Appendix C CMCL Packages |
| • appendix_d | Appendix D CMCL Cell Models |
| **2. programs** | Programs Folder |
| • wms | WMS Program Folder |
| • m2b | M2B Program Folder |
| **3. thesis** | Thesis Folder |