NOVEL RTD-BASED THRESHOLD LOGIC DESIGN AND VERIFICATION

Yexin Zheng

Thesis submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Dr. Chao Huang, Chair
Dr. Michael S. Hsiao
Dr. Patrick R. Schaumont

April 28th 2008

BLACKSBURG, VIRGINIA

KEYWORDS: resonant tunneling diode, threshold logic, reconfigurable structure, equivalence checking, SAT

Novel RTD-Based Threshold Logic Design and Verification

Yexin Zheng

Abstract

Innovative nano-scale devices have been developed to enhance future circuit design to overcome physical barriers hindering complementary metal-oxide semiconductor (CMOS) technology. Among the emerging nanodevices, resonant tunneling diodes (RTDs) have demonstrated promising electronic features due to their high speed switching capability and functional versatility. Great circuit functionality can be achieved through integrating heterostructure field-effect transistors (HFETs) in conjunction with RTDs to modulate effective negative differential resistance (NDR). However, RTDs are intrinsically suitable for implementing threshold logic rather than Boolean logic which has dominated CMOS technology in the past. To fully take advantage of such emerging nanotechnology, efficient design methodologies and design automation tools for threshold logic therefore become essential.

In this thesis, we first propose novel programmable logic elements (PLEs) implemented in threshold gates (TGs) and multi-threshold threshold gates (MTTGs) by exploring RTD/HFET monostable-bistable transition logic element (MOBILE) principles. Our three-input PLE can be configured through five control bits to realize all the three-variable logic functions, which is, to the best of our knowledge, the first single RTD-based structure that provides complete logic implementation. It is also a more efficient reconfigurable circuit element than a general look-up table which requires eight configuration bits for three-variable functions. We further extend the design concept to construct a more versatile four-input PLE. A comprehensive comparison of three- and four-input PLEs provides an insightful view of design tradeoffs between performance and area. We present the mathematical proof of PLE's logic completeness based on Shannon Expansion, as well as the

HSPICE simulation results of the programmable and primitive RTD/HFET gates that we have designed. An efficient control bit generating algorithm is developed by using a special encoding scheme to implement any given logic function.

In addition, we propose novel techniques of formulating a given threshold logic in conjunctive normal form (CNF) that facilitates efficient SAT-based equivalence checking for threshold logic networks. Three different strategies of CNF generation from threshold logic representations are implemented. Experimental results based on MCNC benchmarks are presented as a complete comparison. Our hybrid algorithm, which takes into account input symmetry as well as input weight order of threshold gates, can efficiently generate CNF formulas in terms of both SAT solving time and CNF generating time.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Dr Chao Huang, who has supported me thoughout my thesis with his patience and knowledge. I attribute the level of my Masters degree to his encouragement and effort and without him this thesis, would not have been completed or written.

I also wish to offer my appreciation to Dr. Michael S. Hsiao and Dr. Patrick R. Schaumont, both at Virginia Tech. Their suggestions, comments and additional guidance were invaluable to the completion of this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Although complementary metal-oxide semiconductor (CMOS) will continue dominating digital electronic circuits for the next 10-15 years [1], research advances of innovative nano-scale devices have visualized great opportunities to surpass physical barriers faced by the current semiconductor technology [2–4], which include field effect transistors (*e.g.*, SOI MOSFETs [5]), single electron devices (transistors [6], traps [7], and memories [8]), and quantum interference devices such as resonant tunneling diodes (RTDs) [9], *etc*.

Among these nanodevices, RTDs have shown promising circuit characteristics in improving both analog and digital circuits, due to their high speed switching capability and versatile functionality. For example, several RTD-based circuits have been reported working at clock frequencies of GHz, including the basic logic gates [10], flip-flops [11], analog-to-digital convertor [12], *etc*. In [13, 14], tunneling-based static random access memory (SRAM) cells have demonstrated a higher performance, smaller area, and lower standby power consumption compared to traditional SRAMs. The prototype and integration process of RTD-CMOS hybrid circuits were developed to achieve higher speed and lower power fabrication over pure CMOS circuits [15, 16]. To better understand and support design of

high performance RTD-based circuits, various RTD models have been proposed in [17–19], which help bridge the gap between quantum mechanical models, circuit simulation models, and measured results of fabricated RTDs. The authors in [20] improved transient convergence performance analysis by incorporating three modified algorithms into the RTD SPICE models.

Recently, research work of augmenting RTDs in conjunction with heterostructure field effect transistors (HFETs) have been proposed to modulate negative differential resistance (NDR) to achieve desirable circuit characteristics. In [21,22], a RTD/HFET threshold logic design called MOnostable-BIstable transition Logic Element (MOBILE) realized complex functionality with lower area and power consumption. The MOBILE threshold gate (TG) principle was then extended in [23] to implement multi-threshold TGs (MTTGs) by connecting three or more RTDs in series. Compared to Boolean logic, TG and MTTG designs can increase circuit functionality while reducing circuit levels and gate numbers. Although threshold logic was first proposed and investigated in the 1960s [24], motivated by the compelling potential of nanodevice-based threshold logic circuits, threshold logic design automation have gained renewed interests and become an active field. For example, both synthesis and automatic test pattern generation techniques have been lately developed in [25, 26]. Moreover, the intrinsic self-latching property of MOBILE devices enables the implementation of nanopipeline architectures without incurring latency overheads [27, 28]. By contrast, conventional pipelined structures require latches between pipeline stages to keep data valid, therefore lengthen delays.

However, since nanotechnologies are still at their early life stage, the development of efficient and effective design and verification methodologies is crucial and remain an open topic. In this thesis, we first propose the design of a novel programmable logic element

(PLE) structures implemented with programmable MOBILE TGs and MTTGs. The proposed PLEs are proved of complete logic functionality and an efficient configuration bits generation algorithm for PLE structures is also constructed. By adapting a nanopipeling scheme, PLE structures can support dynamic reconfigurability without incurring delay overheads. The novalties of this threshold logic element design are highlighted as follows:

- The simple and novel three- and four-input PLE structures are developed, which consist of three novel programmable gates and two primitive functional gates based on MOBILE TGs and MTTGs. The design simulation testifies functional correctness.

- Compared with [29], our circuit configuration is proved to be able to realize all the logic functions through properly setting the control bits which can be obtained by an effective encoding scheme. Furthermore, only five control bits need to be configured to realize a three-input logic function. This is more compact and efficient than a general look-up table (LUT) solution which requires eight configuration bits.

- By adapting a nanopipelining scheme, the PLE structure is designed to support dynamical reconfiguration without delay overheads. Comparisons between three- and four-input PLE implementations provide an insightful view of design tradeoff.

In addition to the threshold logic design, we also develop equivalence checking methodologies to help verify threshold logic designs. Novel techniques of formulating a given threshold logic in conjunctive normal form (CNF) that facilitate efficient SAT-based equivalence checking is proposed. Our goal is to generate CNF formulas efficiently in terms of both SAT solving and CNF generating time. Three different strategies of CNF generation from a given threshold logic representation of weight-threshold vectors are implemented. Experimental results based on MCNC benchmarks are provided as a comparison. Our hybrid algorithm outperforms previous methods by taking into account input symmetry as

well as input weight order of threshold gates. It can achieve an average SAT solving time reduction of 81.5% for equivalence checking between a Boolean and threshold representation.

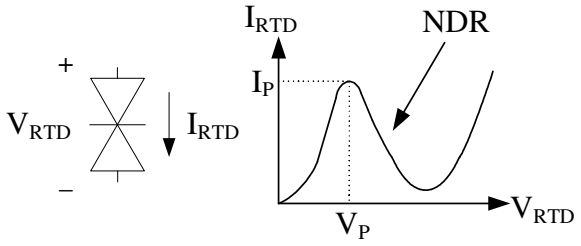
The rest of this thesis is organized as follows. Chapter 2 introduces the preliminary concepts and background materials. Chapter 3 presents the novel PLE topologies, its logic completeness and the algorithm to generate configuration bits. Chapter 4 introduces the equivalence checking techniques for threshold logic. Finally Chapter 5 concludes.

Chapter 2

Background

In this section, we introduce some preliminary concepts of RTD-based threshold logic design and verification, specifically, the MOBILE circuit, threshold function, clocking scheme for nanopipelining, unateness and cofactors, and SAT-based equivalence checking.

2.1 MOBILE



(a) (b)

Figure 2.1: RTD: (a) schematic symbol and (b) I-V characteristics

The RTD schematic symbol and its I-V curve are shown in Figure 2.1(a) and (b), respectively. RTD devices feature a nonlinear I-V characteristic called NDR. When current

I_{RTD} is smaller than its peak value I_P, it increases as V_{RTD} increases – the RTD is at a low resistance state. Once I_{RTD} reaches I_P, the RTD enters the NDR region and I_{RTD} decreases as V_{RTD} increases – the RTD switches to a high resistance state.

Great circuit functionalities can be enabled by exploiting this special NDR characteristic. A basic MOBILE circuit, proposed in [21,22], connects a load and driver RTD in series as shown in Figure 2.2(a). MOBILEs are bias rising-edge triggered and current controlled. Driven by a bias voltage V_{CLK} which oscillates between $0V$ and V_{DD}, the MOBILE circuit operates at a monostable or bistable state.

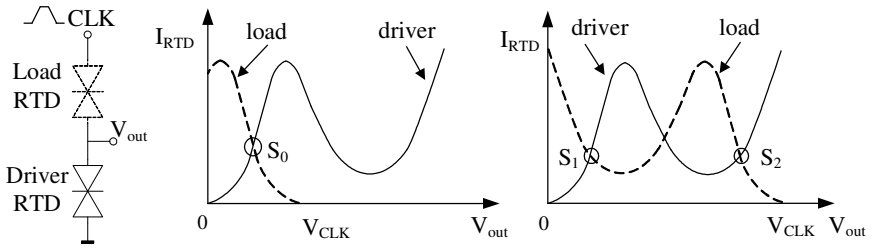


Figure 2.2: MOBILE: (a) basic circuit, (b) operating principle in monostable, and (c) operating principle in bistable

When bias voltage V_{CLK} is low, both RTDs are at their low resistance state and the MOBILE has only one stable state S_0 shown in the I-V chart of Figure 2.2(b). This is referred to *monostable*. When the bias voltage increases, the RTD with a smaller peak current I_P switches first from the low resistance state to a high resistance. The MOBILE can reach two possible states S_1 or S_2, which is referred to *bistable*, as shown in Figure 2.2(c). The resulting state of the MOBILE depends on which RTD (load or driver) switches first to its high resistance state, in other words, which RTD has a smaller peak current I_P. If the load RTD has a smaller I_p, the MOBILE becomes stable at S_1 and generates a low output (logic 0). Otherwise, the MOBILE switches to S_2 with a high output (logic 1). Assuming

that the current density is identical for both load and driver RTDs, the RTD's peak current is proportional to its area. Therefore, the functionality of a MOBILE circuit can be simply determined by the RTD sizes. Combined with RTD/HFET branches which are used to modulate the equivalent peak current, the MOBILE circuit can implement TGs and more complex MTTGs.

2.2 TG and MTTG

A TG [24] is defined as a logic gate with n binary input variables $\{x_i\}$ (i=1,2,. . .,n), a set of n positive or negative weights $\{w_i\}$ (i=1,2,. . .,n), and a numerical threshold T such that the binary output

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq T \\ 0 & \text{otherwise} \end{cases}$$

A TG can also be denoted by using a weight-threshold vector $[w_1, w_2, \ldots, w_n; T]$.

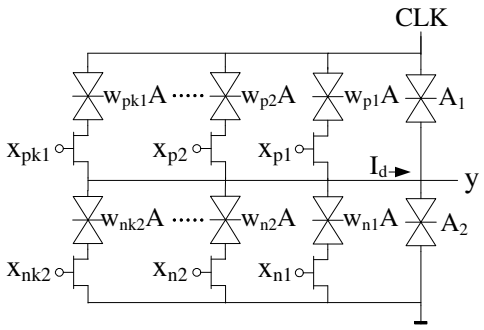


Figure 2.3: A generic RTD/HFET MOBILE TG

Figure 2.3 illustrates a generic TG topology based on MOBILE implementation [30], in which current controlling branches are connected in parallel with the MOBILE RTDs (with a load area A_1 and driver area A_2). The current controlling branches consist of a series connection of an RTD and a heterostructure field-effect transistor (HFET), where A

is the unit RTD area and weights w_{pi} (i=1,2,. . .,$k1$) and w_{nj} (j=1,2,. . .,$k2$) are determined by the corresponding RTD areas. The HFETs, which demonstrate high-frequency performance, behave like switches with x_{pi} (i=1,2,. . .,$k1$) and x_{nj} (j=1,2,. . .,$k2$) as the positive and negative binary inputs, respectively. We assume the unit area peak current density I_{pd} of each RTD is identical. The modulation current can be obtained by Kirchoff's current law as follows.

$$I_d = \sum_{i=1}^{k1} x_{pi} w_{pi} A I_{pd} - \sum_{j=1}^{k2} x_{nj} w_{nj} A I_{pd}, \qquad x_{pi}, x_{nj} \in \{0, 1\}$$

Given a numerical threshold T, the current threshold is calculated as I_t=$TA\times I_{pd}$. The relationship between the modulation current I_d and current threshold I_t determines the circuit output: $y = 1$ if $I_d \geq I_t$; otherwise $y = 0$. Therefore, this generic MOBILE TG implements a threshold function $[w_{p1}, w_{p2}, \ldots, w_{pk1}, -w_{n1}, -w_{n2}, \ldots, -w_{nk2}; T]$.

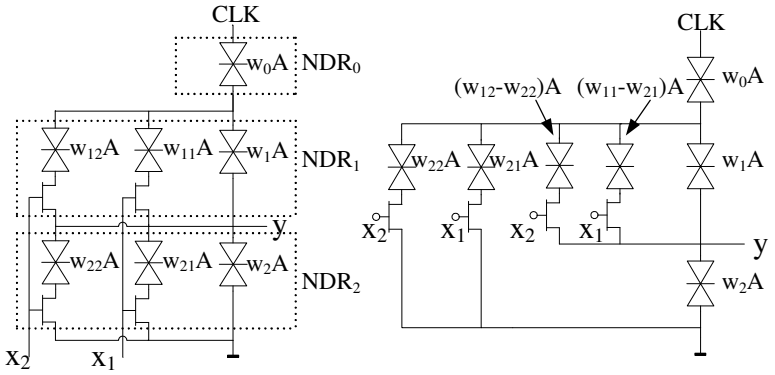


Figure 2.4: MOBILE MTTG: (a) basic topology and (b) improved topology

The concept of RTD/HFET MOBILE TG design can be further extended to implement MTTGs by connecting three or more RTDs in series [23]. Because of the same circuit operating principles as TGs, different MTTG functions can be realized by adjusting the RTD areas to obtain the required current relationship among different NDRs. Hence, the

determination of MTTG function can be simplified to calculate the effective areas [29]. For the example circuit shown in Figure 2.4(a), the effective NDR areas are w_0A, $(x_2w_{12} + x_1w_{11} + w_1)A$, and $(x_2w_{22} + x_1w_{21} + w_2)A$, respectively. For any given input combination, the NDR with the smallest effective area switches first to the high resistance state and the circuit outputs correspondingly. A programmable MTTG gate, therefore, can be achieved by using some of the inputs as control bits to configure the effective areas to implement different logic functions [29]. Figure 2.4(b) demonstrates an improved circuit topology of the same MOBILE MTTG as the circuit in Figure 2.4(a). This alternative implementation can achieve a smaller circuit area and consume less power [31]. The RTD-based circuits proposed in this paper are based on this improved topology.

2.3 MOBILE clocking scheme

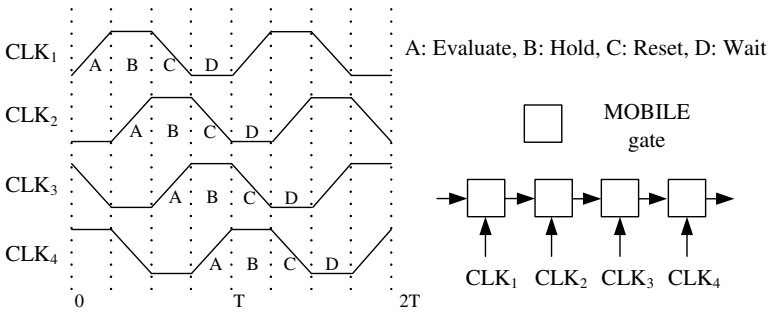


Figure 2.5: Cascaded MOBILE circuits and four-phase clocking scheme

MOBILE circuits are inherently self-latching as they can preserve the output values when bias voltage V_{CLK} is set to high. Therefore, this property together with a proper clocking scheme can enable nanopipelining operations [28]. A four-phase clocking scheme was introduced in [30] to operate cascaded MOBILE circuit stages (see Figure 2.5). In this scheme, each clock period T is divided to four phases with an equal time interval of $T/4$. Phase A is the evaluation phase during which the gate switches from monostable to bistable

and evaluates the output. In phase B, the gate holds the result. In the reset phase C, the load capacitor is discharged and the gate returns to its initial monostable state. The gate is inactive in the wait phase D. So in the clocking scheme illustrated in Figure 2.5, each clock is delayed by $T/4$ from the previous one to safeguard that the evaluation of a gate only starts after the output of its previous gate becomes valid.

2.4 Shannon expansion and threshold cofactors

Given an n-variable Boolean function $f(x_1, x_2, \ldots, x_n)$, its Shannon expansion with respect to variable x_i is

$$f(x_1, x_2, \ldots, x_n) = \bar{x}_i \cdot f_{\bar{x}_i} + x_i \cdot f_{x_i}$$

$$f_{\bar{x}_i} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

$$f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$

$$\forall x_i, \qquad i = 1, 2, \ldots, n$$

where $f_{\bar{x}_i}$ and f_{x_i} are referred to as the negative and positive cofactors, respectively.

The weight-threshold vectors of Shannon cofactors for threshold function can be computed as [24]:

$$f_{x_i} = [w_1, w_2, \ldots, w_{i-1}, w_{i+1}, \ldots, w_n; T - w_i]$$

$$f_{\bar{x}_i} = [w_1, w_2, \ldots, w_{i-1}, w_{i+1}, \ldots, w_n; T]$$

2.5 Unateness of threshold functions

A function $f(x_1, x_2, \ldots, x_n)$ is said to be positive (negative) unate in variable x_i, if and only if $f_{x_i} \supseteq f_{\bar{x}_i}$ ($f_{x_i} \subseteq f_{\bar{x}_i}$). A unique property of threshold function is that any threshold

function is a unate function, which means the function is either positive or negative unate in all its support variables $\{x_i\}$ (i=1,2,. . .,n) [24].

The main reason behind this property is that: if we assume $w_i > 0$, the threshold value $T - w_i$ of the positive cofactor f_{x_i} is smaller than the threshold value T for the negative cofactor $f_{\bar{x}_i}$, which implies $f_{x_i} \supseteq f_{\bar{x}_i}$ and f is positive unate on variable x_i. Similarly, $w_i < 0$ implies a negative unateness on variable x_i.

Theorem 1. *If a threshold function $f(x_1, x_2, \ldots, x_n)$ has a realization f =$[w_1, w_2, \ldots, w_n; T]$, the function $g(x'_1, x'_2, \ldots, x'_n) = f(x_1, x_2, \ldots, x_{i-1}, \bar{x}_i, x_{i+1}, \ldots, x_n)$ is also a threshold function and has a realization g=$[w'_1, w'_2, \ldots, w'_n; T']$ such that $w'_k = w_k$ for $k \neq i$, $w'_k = -w_k$ for $k = i$, and $T' = T - w_i$.*

This theorem [24] provides a method to transform a threshold function between positive and negative unateness in its variables. As an implication of Theorem 1, a threshold function with negative unate variables can be converted to an equivalent threshold function with all positive unate variables. In other words, all the input weights are positive.

Example 1. *Consider function $f(x_1, x_2, x_3) = [-1, 2, 1; 2]$ ($f = \bar{x}_1 x_2 + x_2 x_3$) as an example. Function f is a unate function positive in variable x_2 and x_3 and negative in variable x_1. By using theorem 1, f can be transformed to threshold function $g(x'_1, x'_2, x'_3) = [1, 2, 1; 3]$ which is equivalent to $f(\bar{x}_1, x_2, x_3)$ ($g = (\bar{x}_1)x_2 + x_2 x_3$). Now all the weights of the resulting function g are positive. Positive unateness in all the variables facilitates the analysis and computation in the equivalence checking algorithms discussed in Chapter 4.*

2.6 SAT-based equivalence checking

Combinational equivalence checking is one of the most widely used formal techniques in the verification of digital circuits. The equivalence checking problem discussed in this

work is to determine the equivalence of two circuit designs, of which at least one design is realized by threshold logic.

There are two main approaches used alternatively for equivalence checking problems. The first method is to convert the problem into a functionally canonical form such as Binary Decision Diagram (BDD) and the solution can be obtained from the resulting diagram. The second method is to model the problem as a satisfiability problem, which is the focus of this work.

SAT problems are usually formulated in CNF, which consists of the conjunction (logical AND) of several clauses and each clause is a disjunction (logical OR) of one or more literals. Figure 2.6 shows a single output miter structure [32] for SAT-based equivalence checking of two circuits, namely, A and B. In this structure, both circuits A and B have the same primary inputs, and each pair of the corresponding outputs are fed to an XOR gate. All the outputs of these XOR gates are then OR-ed together to generate the miter output. This miter structure is finally transformed into CNF to prove a constant 0 output (equivalence) through SAT solver.

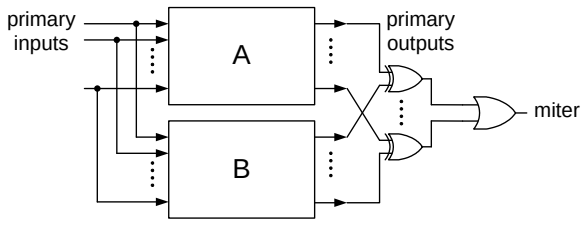


Figure 2.6: Miter circuit

Chapter 3

Programmable threshold logic element design

In this chapter, we propose novel three- and four-input programmable logic elements (PLEs), which are implemented with RTD/HFET MOBILE TGs and MTTGs. Our designs are mostly related to the work in [29]. However, for three input variables, their work can only implement 143 functions with three series-connected RTDs and 213 functions with four RTDs, out of a total of 256 logic functions (three-input functions have eight minterms, therefore altogether $2^8 = 256$ different functions). Contrastingly, our PLEs are mathematically proved to be able to realize all the logic functions, in other words, 256 functions for three inputs. To the best of our knowledge, this is the first RTD-based single structure design which can provide logic complete implementation. Based on the proposed PLE structures, an efficient control bits generation algorithm is also built to take full advantage of PLE structures.

3.1 Programmable logic element

3.1.1 Three-input PLE

The three-input PLE implementation is shown in Figure 3.1, which can realize all 256 logic functions by setting the control bits $\{c_1, c_2, \ldots, c_5\}$. The logic completeness and configuration details will be addressed in Sections 3.2 and 3.3, respectively.

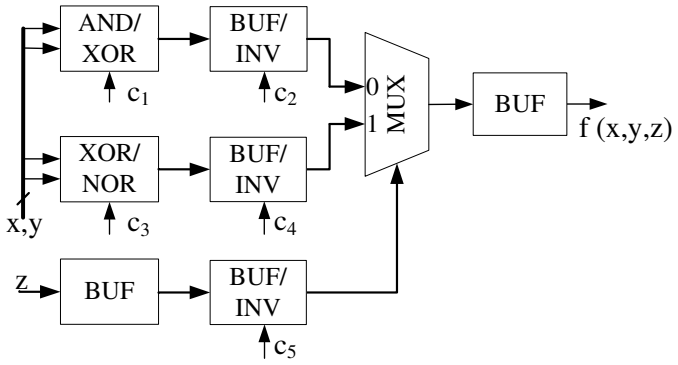


Figure 3.1: Three-input PLE

The three-input PLE is composed of three programmable gates, AND/XOR, XOR/NOR, and BUF/INV, and two primitive functional gates, MUX (multiplexer) and BUF (buffer). The programmable gates and MUX that we have designed use the improved MTTG topology introduced in Section 2.2, and BUF is implemented by a basic RTD/HFET TG [31]. The gate designs are illustrated in Figure 3.2(a)-(e). The area of each RTD is given in the figures and A denotes the unit RTD area. Each of these three programmable gates can realize two different boolean functions depending on the value of the control bit. For example, gate AND/XOR shown in Figure 3.2(a) has x_1 and x_2 as inputs, c as control bit, and y as output. When $c = 0$, gate AND/XOR acts as a logic *AND*, otherwise an *XOR*. The selection of the logic functions of the programmable gates by the control bit is presented in Table 3.1.

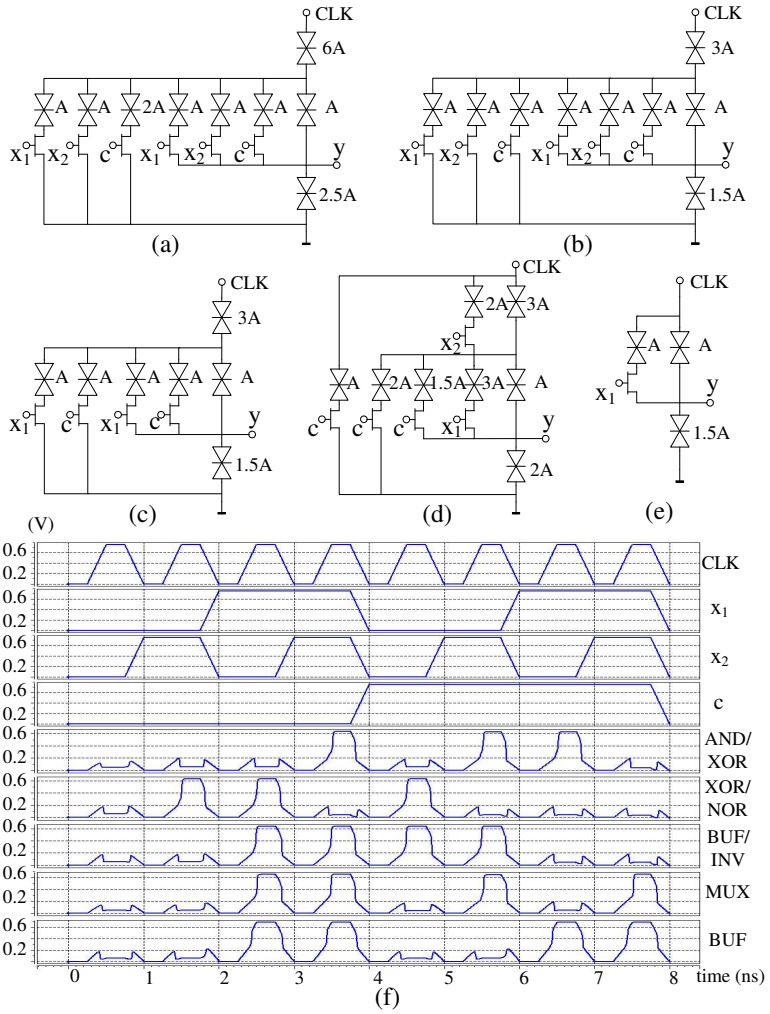


Figure 3.2: PLE MOBILE gate designs and HSPICE simulation: (a) AND/XOR, (b) XOR/NOR, (c) BUF/INV, (d) MUX, (e) BUF, and (f) HSPICE simulation

In the PLE shown in Figure 3.1, variable z selects an x-y function branch based on its positive or negative phase through a MUX. A BUF is used for z to patch up the signal path to two stages to synchronize data arrivals at the evaluation phase of the MUX. Another buffer is inserted at the output of the MUX in order to complete a whole four-phase clock cycle.

The functions of all the programmable and primitive RTD/HFET MOBILE gates that

Table 3.1: Logic function selection of primitive programmable gates

Control bit	Programmable gates			
	AND/XOR	XOR/NOR	BUF/INV	MUX
$c = 0$	$y = x_1x_2$	$y = x_1 \oplus x_2$	$y = x_1$	$y = x_1$
$c = 1$	$y = x_1 \oplus x_2$	$y = \overline{x_1 + x_2}$	$y = \overline{x_1}$	$y = x_2$

we have designed are verified through HSPICE simulation by using circuit models described in [20]. The RTD HSPICE subcircuit model consists of a voltage controlled current source I_{RTD} connected in parallel with a capacitor C, the combination of which is connected in series with a resistor R. The nonlinear current source represents the RTD $I - V$ characteristics. The RTD circuit parameters are set according to the measured results in [30] as follows: peak voltage $V_p = 0.27V$, peak current density $9KA/cm^2$, and capacitance $4fF/\mu m^2$. The HFET is a depletion-type transistor with a threshold voltage $V_T = -0.1V$. The unit RTD area A is set to $2\mu m^2$. The circuits are driven by clock V_{CLK} with an amplitude of $0.75V$. Fig. 3.2(f) presents the HSPICE simulation results at a clock frequency of $1GHz$.

3.1.2 Four-input PLE

In our three-input PLE design, a BUF is added at the MUX output to serve as the fourth nanopipelining stage to adapt the four-phase clocking scheme. We also designed a four-input PLE as an alternative approach to fit in the four clock phases, as shown in Figure 3.3, by duplicating two structures of the first three stages of the three-input PLE and connecting them to a MUX which serves as a fourth stage. In this manner, a fourth input w is added (as the final MUX selection) to implement all four-input logic functions. The comparisons of three- and four-input PLE implementations are discussed in Section 3.5.

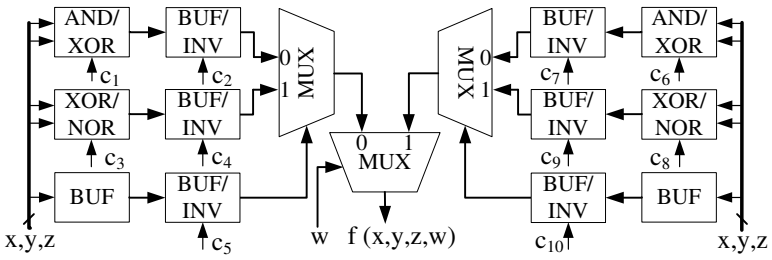


Figure 3.3: Four-input PLE

3.2 Logic completeness

The ability of realizing all 256 three-variable logic functions $f(x, y, z)$ by using our three-input PLE is not as obvious as a three-input LUT. In order to prove the logic completeness, we define the literal set **L** as

$$\mathbf{L} = \{l_i\} = \{\bar{x}_1, x_1, \bar{x}_2, x_2, \ldots, \bar{x}_n, x_n, \}, \quad i = 1, 2, \ldots, 2n$$

Accordingly, the cofactor set **F** is defined as

$$\mathbf{F} = \{f_i\} = \{f_{\bar{x}_1}, f_{x_1}, f_{\bar{x}_2}, f_{x_2}, \ldots, f_{\bar{x}_n}, f_{x_n}, \}, \quad i = 1, 2, \ldots, 2n$$

Shannon Expansion introduced in Section 2.4 can be implemented by simply using a MUX with x_i as the select bit and the positive and negative cofactors connected to the positive and negative MUX inputs, respectively. Therefore, the three-input PLE shown in Figure 3.1 is a three-variable function by Shannon Expansion on variable z. The positive or negative cofactor is one of the total 16 x-y functions. Unfortunately, the combination of an AND/XOR and BUF/INV gate (the negative branch) as well as the combination of an XOR/NOR and BUF/INV gate (the positive branch) cannot implement all the 16 x-y functions. We denote the function set of $\{x\bar{y}, \bar{x}y, x + \bar{y}, \bar{x} + y\}$ that cannot be implemented as unavailable set S_u and the set of the rest 12 functions as available set S_a.

Theorem 2. *Given a three-variable logic function $f(x, y, z)$, there exists at least one variable, whose Shannon expansion cofactors belong to the unavailable set S_u.*

In order to prove it, let us begin with some preliminary concepts. A Boolean function can be canonically expressed as the sum of minterms. For a three-variable function $f(x, y, z)$, it has eight possible minterms: $\{m_1, m_2, \ldots, m_8\}$ representing $\{\bar{x}\bar{y}\bar{z}, \bar{x}\bar{y}z, \ldots, xyz\}$, respectively. Hence, we use a 8×1 matrix $\mathbf{X}$ to canonically represent f.

$$\mathbf{X} = \begin{pmatrix} x_1 & x_2 & \ldots & x_8 \end{pmatrix}^T$$

$$x_i = \begin{cases} 1 & \text{if } m_i \in f \\ 0 & \text{if } m_i \notin f \end{cases}$$

The expansion cofactor matrix $\mathbf{W}$ is constructed to represent the cofactors of Shannon Expansions with respect to all the variables, which can guide us to quickly determine whether an expansion yields S_u cofactors or not. The general $\mathbf{W}$ matrix for n variables is defined as

$$w_{ij} = \begin{cases} 0 & \text{if minterm } m_j\text{'s cofactor } f_i \text{ is 0} \\ 1 & \text{if minterm } m_j\text{'s cofactor } f_i \in S_a \\ 3 & \text{if minterm } m_j\text{'s cofactor } f_i \in S_u \end{cases}$$
$$\forall i = 1, 2, \ldots, 2n \text{ and } j = 1, 2, \ldots, 2^n$$

Here the weights $\{1, 3\}$ are chosen to distinguish the S_u functions from S_a. As for three

input variables, $\mathbf{W}$ is an 6×8 matrix given as follows.

$$\mathbf{W}_{6\times 8} = \begin{pmatrix} 1 & 3 & 3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 3 & 3 & 1 \\ 1 & 3 & 0 & 0 & 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 & 3 & 1 \\ 1 & 0 & 3 & 0 & 3 & 0 & 1 & 0 \\ 0 & 1 & 0 & 3 & 0 & 3 & 0 & 1 \end{pmatrix} \begin{matrix} f_{\bar{x}} \\ f_x \\ f_{\bar{y}} \\ f_y \\ f_{\bar{z}} \\ f_z \end{matrix}$$
$$m_1\ m_2\ m_3\ m_4\ m_5\ m_6\ m_7\ m_8$$

where row i represents cofactor f_i in the cofactor set $\mathbf{F} = \{f_i\} = \{f_{\bar{x}}, f_x, f_{\bar{y}}, f_y, f_{\bar{z}}, f_z\}$, and column j represents minterm m_j of $\{m_1, m_2, \ldots, m_8\}$.

We then define our cofactor encoding matrix $\mathbf{F}^*$ as

$$\mathbf{F}^* = \begin{pmatrix} f_x^* & f_{\bar{x}}^* & f_y^* & f_{\bar{y}}^* & f_z^* & f_{\bar{z}}^* \end{pmatrix}^T = \mathbf{W} \cdot \mathbf{X}$$

so that the encoded value of f_i^* is calculated to be 3 or 5 for those cofactors f_i belong to S_u. In other words, if function $f(x, y, z)$ cannot be implemented on the three-input PLE by using input z as the MUX select bit as shown in Figure 3.1, at least one of the two encoded cofactors $f_{\bar{z}}^*$ and f_z^* is equal to either 3 or 5.

We now prove the theorem by contradiction.

Proof. If the theorem is not true, for all the three encoded cofactor pairs (negative and positive), $\{f_{\bar{x}}^*, f_x^*\}$, $\{f_{\bar{y}}^*, f_y^*\}$, and $\{f_{\bar{z}}^*, f_z^*\}$, at least one encoded cofactor of each pair is equal to 3 or 5. In other words, the theorem is true if we can prove that no matrix $\mathbf{X}$ can yield a 3, 5, or both for all the three encoded cofactor pairs at the same time.

- *Case 1:* Every encoded cofactor pair has a 3. An encoded cofactor of value 3 relates to a violating Shannon cofactor that is a $(x\bar{y})$-style function. Consider matrix $\mathbf{W}_{6\times 8}$:

each column (one minterm) only covers two 3s (two corresponding cofactors). Function $f(x, y, z)$ should contain at least two minterms to result in three 3s. Without losing generality, suppose f has minterm m_2 which results in $f^*_{\bar{x}} = 3$ ($f_{\bar{x}} \in S_u$) and $f^*_{\bar{y}} = 3$ ($f_{\bar{y}} \in S_u$). To satisfy the case that every encoded cofactor pair has a 3, at least one of the encoded cofactors of $f^*_{\bar{z}}$ and f^*_z equals 3. Therefore, at least one of the four minterms $\{m_3, m_4, m_5, m_6\}$ is contained in function f. However, no matter which minterm belongs to f, the value of $f^*_{\bar{x}}$ or $f^*_{\bar{y}}$ no longer stays equal to 3, which contradicts the assumption we have. Hence, case 1 is impossible.

- *Case 2:* At least one of the three encoded cofactor pairs has a 5, and the other two pairs have either a 3 or 5. An encoded cofactor of value 5 relates to a violating Shannon cofactor that is a $(x + \bar{y})$-style function. Also without lost generality, suppose minterms m_1, m_2, and $m_4 \in f$ that makes $f_{\bar{x}} \in S_u$ and the cofactor encoding matrix $\mathbf{F}^* = (5\ 0\ 4\ 3\ 1\ 4)$. Since $f^*_y = 3$, only the expansion on variable z is now feasible. To meet the assumption that all the three pairs have either a 3 or 5, f must contain other minterms. In other words, either $f^*_{\bar{z}}$ or f^*_z should equal 5. If ${f_{\bar{z}}}^* = 5$, m_3 and m_7 are contained in f, which will result in $f^*_{\bar{y}} = 4$ and $f^*_y = 7$. Or m_5 and m_7 are contained in f, which will result in $f^*_{\bar{y}} = 7$ and $f^*_y = 6$. Under these two scenarios, Shannon Expansion on y flips from infeasible to feasible. If $f^*_z = 5$, m_8 must belong to f, which results in $f^*_{\bar{y}} = 4$ and $f^*_y = 4$. Now expansion on y becomes feasible. If either m_6 or m_3 is in f, correspondingly, $f^*_{\bar{y}}$ or f^*_y will be 5. However, this will change the value of $f^*_{\bar{x}}$ or f^*_z and contradict with the assumption $f_{\bar{x}} = 5$. Hence, case 2 is impossible.

Combining case 1 and case 2, we conclude that it is impossible that all the three encoded cofactor pairs have a 3, 5, or both at the same time. In other words, no such a function

$f(x, y, z)$ whose expansion on every variable can yield a S_u cofactor.

□

Next, we will use an example to demonstrate how to use encoded cofactor matrix $\mathbf{F}^*$ to choose an expansion variable.

Example 2. *Consider a logic function $f(x, y, z) = xy + y\bar{z} + x\bar{z}$. It can be expressed as the sum of minterms:*

$$f(x, y, z) = \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z} + xyz$$

The corresponding minterm representation for f is

$$\mathbf{X} = (0\,0\,1\,0\,1\,0\,1\,1)^T$$

Therefore,

$$\mathbf{F}^* = \begin{pmatrix} 1 & 3 & 3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 3 & 3 & 1 \\ 1 & 3 & 0 & 0 & 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 & 3 & 1 \\ 1 & 0 & 3 & 0 & 3 & 0 & 1 & 0 \\ 0 & 1 & 0 & 3 & 0 & 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 3 \\ 5 \\ 7 \\ 1 \end{pmatrix}$$

Since $f^*_{\bar{x}} = f^*_{\bar{y}} = 3$ and $f^*_x = f^*_y = 5$, Shannon expansions on variable x and y cannot be implemented on our three-input PLE. Actually Shannon Expansion on variable x generates

$$f = \bar{x}(y\bar{z}) + x(y + \bar{z})$$

The cofactors $f_{\bar{x}} = y\bar{z}$ and $f_x = y + \bar{z}$ both belong to S_u. A similar result can be derived by expansion on variable y. Since neither $f^*_{\bar{z}}$ nor f^*_z equals 3 or 5, we choose to expand on

variable z, which yields

$$f = \bar{z}(x + y) + z(xy)$$

Figure 3.4 shows the PLE implementation for this example. The control bits generating algorithm is presented in Section 3.3.

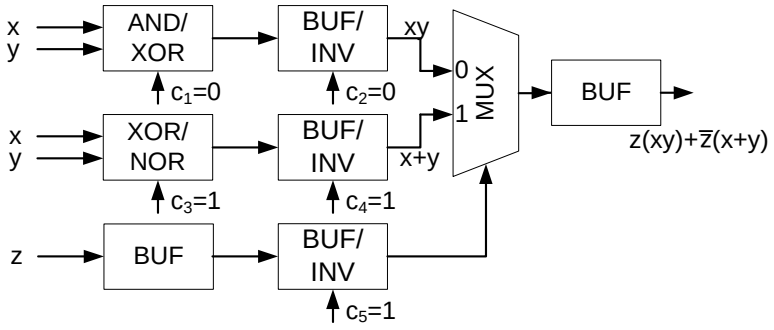


Figure 3.4: A configuration example

After picking up a feasible expansion variable, both the negative and positive cofactors of function f belong to the available set S_a and can be respectively mapped to a pair of available boolean functions {AND, NAND, OR, NOR, XOR, XNOR}. As both of the expansion variable and its complement can be fed as the select bit of the MUX, the order of the boolean function pairs can be exchanged. However, since each input branch of the MUX can only implement four of the six boolean functions ({AND, NAND, XOR, XNOR} for negative branch and {OR, NOR, XOR, XNOR} for positive branch), there are still six unordered function pairs that cannot be mapped onto a PLE. They are (AND, AND), (NAND, NAND), (OR, OR), (NOR, NOR), (AND, NAND), and (OR, NOR). Fortunately, the mapping between a cofactor and its boolean function implementation is a many-to-many mapping. Alternative function pairs always exist to result in a feasible mapping.

Based on the previous discussion, we see that the PLE structure is a simple yet powerful logic element. It can realize all 256 three-input functions with a proper configuration of

the inputs and control bits. Compared with current FPGA SRAM-based LUTs, our PLE requires only five bits to configure any three-input function rather than eight bits for a 3-LUT.

3.3 Control bits generation

In Section 3.2, an expansion cofactor matrix $\mathbf{W}$ is introduced to quickly determine the feasibility of Shannon Expansions. We similarly construct a cofactor mapping matrix $\mathbf{W}'$ to derive the control bits for three-input PLE implementations. We use a weighted binary encoding scheme. For simplicity, consider the case of expansion on variable z. Since the resulting cofactors have four possible terms $\{\bar{x}\bar{y}$,$\bar{x}y$,$x\bar{y}$ and $xy\}$, we assign four different binary weights $\{1, 2, 4, 8\} = \{2^0, 2^1, 2^2, 2^3\}$ to these four possible cofactors. Therefore, the cofactor mapping matrix can be expressed as:

$$\mathbf{W}' = \begin{pmatrix} 1 & 2 & 4 & 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 4 & 8 \\ 1 & 2 & 0 & 0 & 4 & 8 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 & 4 & 8 \\ 1 & 0 & 2 & 0 & 4 & 0 & 8 & 0 \\ 0 & 1 & 0 & 2 & 0 & 4 & 0 & 8 \end{pmatrix}$$

Since the mapping cofactors derived through $\mathbf{F}' = \mathbf{W}' \cdot \mathbf{X}$ are integers $\in [0, 15]$, a one-to-one mapping to the 16 two-variable functions, we can easily determine the boolean functions by checking their binary encoded values.

Figure 3.5 describes the pseudo code of the control bits generating algorithm. It derives the input connection and control bits configuration $C(f)$ for a given function f. First, the cofactor mapping matrix W' and minterm representation X of function f are multiplied to

Input: f
Output: $C(f)$
1: generate matrix X for function f
2: $\mathbf{F'} = \mathbf{W'} \cdot \mathbf{X}$
3: **for** every feasible expansion variable v
4: $B_{\bar{v}} \leftarrow f'_{\bar{v}}$
5: $B_v \leftarrow f'_v$
6: **if** $map(B_{\bar{v}}, B_v)$=positive **then**
7: $c_5 = 0$, get c_1, c_2, c_3 and c_4
8: **return** $C(f)$
9: **else if** $map(B_{\bar{v}}, B_v)$=negative **then**
10: $c_5 = 1$, get c_1, c_2, c_3 and c_4
11: **return** $C(f)$

Figure 3.5: Control bits generating algorithm

generate the mapping cofactors $\mathbf{F'}$ (Line **1-2**). As we discussed in Section 3.2 that not all the Shannon Expansions can be implemented on PLEs, we need to perform a feasibility check on the resulting mapping cofactors and choose a feasible one. Because the four cofactor functions in S_u are encoded as $\{2, 4, 11, 13\}$ under this binary encoding scheme, we just search in the variable order and pick up the first expansion whose both encoded positive and negative cofactors do not belong to $S'_u = \{2, 4, 11, 13\}$.

Then we map the encoded cofactors f'_v and $f'_{\bar{v}}$ to the available boolean set $\mathbf{B}$=$\{AND$, $NAND, OR, NOR, XOR, XNOR\}$. If boolean functions $B_{\bar{v}}$ and B_v that realize the negative and positive cofactors can be implemented as the negative and positive PLE branches respectively ($map(B_{\bar{v}}, B_v)$=positive), variable v is connected to the select bit of the MUX with $c_5 = 0$. The corresponding control bits that configure the programmable gates are generated (Line **6-8**). However, due to the asymmetry of the positive and negative PLE branches, it may happen that the positive and negative cofactors can only be mapped to the negative and positive branches, respectively ($map(B_{\bar{v}}, B_v)$=negative). Under such a circumstance, the control bit c_5 is set to 1 which feeds v's complement to the MUX's select

bit (Line **9-11**).

Example 3. *Consider function $f(x,y,z) = xy + y\bar{z} + x\bar{z}$ in Example 2 again. The minterm matrix representation of f is* $\mathbf{X}$=$(0\ 0\ 1\ 0\ 1\ 0\ 1\ 1)^T$. *Then we calculate the mapping cofactors* $\mathbf{F}'$ *using the cofactor mapping matrix:*

$$\mathbf{F}' = \begin{pmatrix} 1 & 2 & 4 & 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 4 & 8 \\ 1 & 2 & 0 & 0 & 4 & 8 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 & 4 & 8 \\ 1 & 0 & 2 & 0 & 4 & 0 & 8 & 0 \\ 0 & 1 & 0 & 2 & 0 & 4 & 0 & 8 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 13 \\ 4 \\ 13 \\ 14 \\ 8 \end{pmatrix}$$

Checking the encoded cofactors, we find out that $f'_{\bar{x}} = f'_{\bar{y}} = 4 \in S'_u$ and $f'_x = f'_y = 13 \in S'_u$. This implies that only z can be used as the expansion variable. The encoded cofactors with respect to variable z are mapped to boolean functions: $f'_{\bar{z}} = 14 = (1110)_2 \Rightarrow f_{\bar{z}} = xy + x\bar{y} + \bar{x}y = x + y$ (OR gate) and $f'_z = 8 = (1000)_2 \Rightarrow f_z = xy$ (AND gate). Because $B_{\bar{v}} = OR$ and $B_v = AND$ can only be implemented on the positive and negative PLE branches respectively ($map(B_{\bar{v}}, B_v)$=negative), the control bit c_5 is set to 1 to invert z. The control bits of the negative branch $c_1 = 0$ and $c_2 = 0$ are required to configure an AND gate for $f_z = xy$, while the control bits of the positive branch $c_3 = 1$ and $c_4 = 1$ are required to configure an OR gate for $f_{\bar{z}} = x + y$. The final configuration to realize function $f(x,y,z) = z \cdot (xy) + \bar{z} \cdot (x + y)$ is shown in Figure 3.4.

Because of the fact that the four-input PLE is composed of two three-input PLEs, the control bits of a four-input PLE can be derived by a proper modification of the aforementioned algorithm targeting three-input PLEs. For a four-input function $f(x,y,z,w)$, at first

an input variable is selected randomly, for example w. Thus the function f can be expressed as Shannon Expansion on variable w: $f = w \cdot f_w + \bar{w} \cdot f_{\bar{w}}$. Then f_w and $f_{\bar{w}}$ are two three-input functions that can be implemented by the two three-input PLE branches of the four-input PLE structure (see Figure 3.3). The control bits generating algorithm for three-input PLE is executed twice to obtain the control bits for both function f_w and $f_{\bar{w}}$. Altogether ten control bits are generated corresponding to $\{c_1, c_2, \ldots, c_{10}\}$ of the four-input PLE shown in Figure 3.3.

3.4 Dynamic reconfigurability

Generally speaking, one of the performance challenges of dynamic reconfiguration is the relatively long reconfiguration time caused by the requirement of loading a large amount of configuration data through limited internal bandwidth. Thanks for the inherent self-latching property of MOBILE devices, the PLE structure can easily relieve this design bottleneck without introducing any overhead.

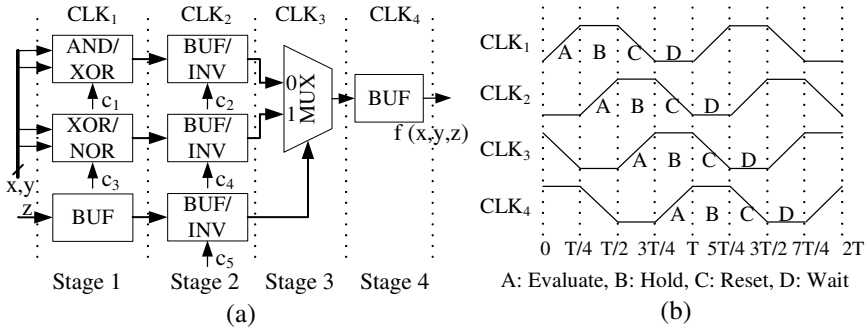


Figure 3.6: Nanopipelining: (a) pipeline stages and (b) clocking scheme

Figure 3.6(a) shows the separation of the PLE's four nanopipelining stages. As described in Section 2.3, four overlapping four-phase clocks (CLK_1, CLK_2, CLK_3, and CLK_4 illustrated in Figure 3.6(b)) are supplied to the corresponding stages to facilitate

nanopipelining operations. Under this clocking scheme, the MOBILE-based circuits of each stage require the output values of their previous stage to be valid only at the evaluation phase (phase A). Even the inputs change after the evaluation phase, the self-latching property of MOBILE circuits keeps the output values stable during the hold phase (phase B). Therefore, the hold, reset, and wait phases can be used to reconfigure the input connections and control bits.

Suppose the PLE functionality is dynamically reconfigured every clock cycle T (reconfiguration cycle). In the PLE pipeline, the inputs to the stage-1 gates are required to be valid only during the clock phase $[0, T/4]$, $[T, 5T/4]$, ..., $[nT, nT + T/4]$ (evaluation phase). Thus the time interval from the end of an evaluation phase to the beginning of next valid evaluation phase, can be used to reconfigure the PLE stage-1 gates including input connection and control bits c_1 and c_3. An example of such a clock phase for stage-1 is $[T/4, T]$ with a reconfiguration slack of $3T/4$. The stage-2 gates, similarly, can take advantage of the hold, reset, and wait phases of CLK_2 (*e.g.*, $[T/2, 5T/4]$) to reconfigure the control bits c_2, c_4, and c_5. The overlapping reconfiguration slacks of different configuration objects form the pipelining reconfiguration scheme. The advantage of this pipelining reconfiguration scheme is that the inactive clock phases of the MOBILE circuits are fully utilized to avoid performance degrading.

3.5 Experimental results

We evaluated our three- and four-input PLE structures in terms of area and performance. MCNC benchmarks were implemented on an array of PLEs. Berkeley's synthesis and verification software ABC [33] was used to extract three- and four-variable logic functions from the benchmark applications.

Table 3.2: Area and performance comparisons of three- and four-input PLE implementations

Circuit	three-input PLE				four-input PLE				Comparisons			
	Level	PLEs (w/o)	PLEs (1DR)	PLEs (Red.%)	Level	PLEs (w/o)	PLEs (1DR)	PLEs (Red.%)	Latency (w/o)	Area (w/o)	L×A (w/o)	L×A (1DR)
9symml	7	111	43	61	6	78	31	60	1.17	0.71	0.83	0.81
alu4	16	381	61	84	12	287	70	76	1.33	0.66	0.88	0.58
apex6	8	369	102	72	6	238	95	60	1.33	0.78	1.04	0.71
apex7	8	107	27	75	5	82	32	61	1.60	0.65	1.04	0.68
cc	3	39	18	54	2	31	21	32	1.50	0.63	0.95	0.64
count	10	56	19	66	6	37	7	81	1.67	0.76	1.27	2.27
dalu	16	585	114	81	11	398	79	80	1.45	0.73	1.06	1.04
des	9	1925	522	73	6	1534	556	64	1.50	0.63	0.95	0.70
rot	12	300	97	68	8	240	91	62	1.50	0.63	0.95	0.80
z4ml	3	16	8	50	3	11	6	45	1.00	0.73	0.73	0.67

The area and performance comparisons of the three- and four-input PLE implementations are summarized in Table 3.2. The level of the circuit, total number of PLEs without dynamic reconfiguration (*w/o*), total number of PLEs with dynamic reconfiguration cycle of $1T$(*1DR*), and reduction of PLE numbers by using dynamic reconfiguration (*Red.%*) are presented for both implementations in major columns *three-input PLE* and *four-input PLE*. The comparisons between two implementations are computed in ratios of three- to four-input on four metrics: *Latency*, *Area*, $L \times A$ *(w/o)*, and $L \times A$ *(1DR)*. Since the latency is proportional to the circuit level, the ratio of latency is the ratio of circuit level under the assumption that the implementations are working at the same clock frequency. The area is proportional to the total number of PLE employed and PLE area.

Since the four-input PLE structure is larger in terms of granularity, it requires less number of total PLEs to implement the function thus reducing the circuit level and overall latency. However this more powerful PLE structure takes approximately twice of area compared to a three-input PLE. Although the total number of the PLEs is reduced, the total area required is still larger than the implementation based on three-input PLEs. If we

consider the latency-area product without reconfiguration, the three-input PLE-based implementations are slightly better and the selection between these two structures is a tradeoff between performance and area cost. When reconfiguration is implemented, the three-input PLE solutions are more favorable especially from the area cost perspective.

The comparison of PLE numbers required for implementation with and without dynamic reconfiguration demonstrates an average total area reduction of 65% if reconfiguration is applied. Combined with the discussion in Section 3.4, the reconfiguration process enables area reduction without incurring performance overheads by utilizing the inactive clock phase of MOBILE circuit.

Chapter 4

Equivalence checking for threshold logic designs

Motivated by the compelling potential of emerging nanotechnologies, most recently, threshold logic design automation has started to become an active research field. Several synthesis and optimization tools have been proposed to automatically yield multi-level threshold logic implementations. However, much of verifying synthesized threshold circuits remains an open topic. An important question in verifying equivalence of threshold networks is how to determine the logic function realized by a given TG of n inputs. A naive solution is to try all the 2^n input combinations and determine the ON-set of the function to generate a sum of products (SOP) representation. Clearly, this is an exponential time solution and therefore not practical. A previously developed equivalence checking method for threshold circuits [34] generates a maximally factored form to construct the corresponding Boolean expression diagram (BED) for each threshold gate in the circuit. It then performs equivalence checking by generating the BDD from BED representations. Considering that BDD-based methods may suffer from a high need of memory resources, many researchers have

looked into alternatives and found that SAT is more robust and flexible than BDDs [35–39]. Therefore, efficient methodologies to bridge Boolean SAT success and threshold logic network verification would greatly benefit nanotechnology design automation.

4.1 Related work

Previous researches on SAT have provided solutions to a general set of inequality problems, namely, pseudo-Boolean (PB) problems or 0-1 integer linear programming problems [40–42]. The work in [41] transforms the BDD representation of PB constraints directly to CNF clauses without introducing extra variables. In [40,42], methods are proposed to derive CNF formulas by building arithmetic circuits, such as adders, from inequalities, which are linear in size. However, these PB methods cannot be directly applied to equivalence checking of threshold logic. PB problems contain additional objective functions and the CNF for inequalities is converted in implication relation. But for equivalence checking of threshold logic, the SAT solver is not trying to find assignments to satisfy all the inequalities. Instead, the SAT solver checks a TG's functionality, in other words, to check the output is 1 or 0 if the threshold inequalities are satisfied or not. Therefore, for threshold equivalence checking, CNFs are generated to represent the equivalence relationship between inputs and outputs according to the inequalities. Another drawback of the linear method is that SAT solvers tend to have inferior performance on the CNF instances it generated. The main reason is that the linear method, because of using adders as arithmetic circuit elements, yields a lot of XOR relations. These XOR constraints generate few implications so that all the variables must be bound before propagation, and therefore hinder current conflict-clause-based SAT solving.

On the other hand, the threshold logic based on nano-scale devices, such as RTDs, has

some special features that a general PB to CNF conversion technique does not take advantage of. First, most of the practical weights and threshold to implement a complex functionality are small integers, *e.g.*, less than 5 according to the synthesis results in [25]. Besides, since threshold logic is a superset of Boolean logic, (any Boolean primitive gate can be implemented by a single TG but not vice verse), some symmetric properties of Boolean gates still remain in TGs. These characteristics of emerging nanotechnology implementations make equivalence checking become a unique problem where previously developed techniques generate sub-optimal results.

From the above discussion, we see that a good-quality threshold network equivalence checking methodology should address the issues of efficient CNF generating and SAT solving, and specifically should suit emerging nanotechnology design features. Therefore, in this thesis, we propose three novel techniques to formulate a given threshold logic network in CNF. Among them, the hybrid algorithm which combines decision tree traversal and efficient Boolean to CNF transformation, achieves better performance of equivalence checking for nano-threshold networks than previous methods.

4.2 SAT-based equivalence checking methodologies

We present our SAT-based equivalence checking methodologies for threshold networks in this section. The definitions used in our algorithms are given as follows.

f: a given threshold function of weight-threshold vector

x_i: the i^{th} fanin variable

f_{x_i}: the positive cofactor of f with respect to input x_i

$f_{\bar{x}_i}$: the negative cofactor of f with respect to input x_i

$cnf(f)$: the CNF representation of f

cls: current CNF clauses

$W(f)$: a queue of input variables in a descending order of weights

$Q(f)$: a queue of variables that hybrid algorithm searches

$P(f)$: a set of variables representing the products in the SOP form of f

Before formulating a weight-threshold vector to final CNF clauses, a preprocessing step transforms all the negative weights (negative unateness) to positive using Theorem 1. This reduces complexity of the subsequent analysis and cofactoring process, and can be easily recovered by negating corresponding CNF variables during output.

Next, we propose our three different algorithms to generate the CNF representation of a given threshold circuit.

4.2.1 Path search with weight ordering

The path search with weight ordering algorithm is shown in Figure 4.1. Given a threshold gate of n inputs with function $f = [w_1, w_2, \ldots, w_n; T]$, it keeps cofactoring the function into its positive and negative cofactors according to Theorem 2 with respect to the first input variable remaining in $W(f)$. The input variable assignments along the searching path form a CNF clause. This recursive expansion continues on the newly generated cofactor functions until it reaches a constant cofactor (cofactor function equals to either 0 or 1). Then the algorithm backtracks and picks another path until the entire search tree is explored and a complete search yields the final CNF representation of the given function.

Function $check\_constant$ returns 0 or 1 if the threshold function reaches constant 0 or 1, respectively. Since all the threshold functions are transformed to positive unate, the principle behind the $check\_constant$ function can be efficiently achieved as: the function returns 0 if $T \leq 0$ and 1 if $\sum_{i=1}^{n} w_i < T$.

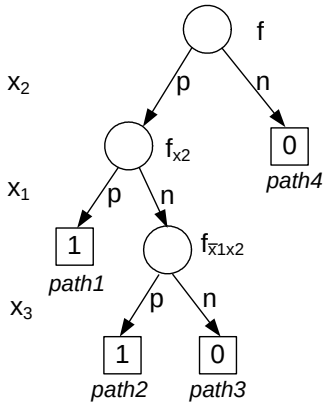
Example 4. *To demonstrate the idea of path search with weight ordering, let us consider*

path_weight()
Input: f, cls, $W(f)$
Output: $cnf(f)$
1: **if** check_constant (f)= 0 or 1 **then**
2: $cnf(f) \wedge cls$
3: **else**
4: x_i=dequeue($W(f)$)
5: $cnf(f) \wedge$ path_weight(f_{x_i}, $cls \vee \bar{x}_i$, $W(f)$)
6: $cnf(f) \wedge$ path_weight($f_{\bar{x}_i}$, $cls \vee x_i$, $W(f)$)
7: **return** $cnf(f)$

Figure 4.1: Path search with weight ordering algorithm

*a threshold function $f = [1, 2, 1; 3]$ ($f = x_1x_2 + x_2x_3$). The algorithm chooses to first cofactor on x_2 which is the input variable with the maximum weight (the first element in queue $W(f)$). Cofactoring on variable x_2 generates a positive cofactor $f_{x_2} = [1, 1; 1]$ ($f_{x_2} = x_1 + x_3$). We continue the process to obtain further positive cofactor $f_{x_1x_2} = [1; 0]$ ($f_{x_1x_2} = 1$), which becomes a constant. Then the variable assignments along the path (*path1 *in Figure 4.2(a)) form a CNF clause as illustrated in Figure 4.2(b). The algorithm backtracks and searches for the entire input combinations for all the possible paths and finally finishes with a complete CNF representation.*

The advantage of this algorithm is twofold. First, during the depth-first search, no extra variables are added, which is more likely to reduce the subsequent SAT solving workload [41]. Second, considering the relation between the input and threshold values, an input with a larger weight influences more than an input with a smaller weight. In other words, cofactoring on the input with a larger weight is more likely to lead toward a constant cofactor, thus terminates the recursive process faster. Therefore, in our algorithm, cofactoring with weight ordering results in an efficient conversion process and CNF representation.



(a)

$$\text{path1}: \quad f \vee \neg x_2 \vee \neg x_1$$
$$\text{path2}: \quad f \vee \neg x_2 \vee x_1 \vee \neg x_3$$
$$\text{path3}: \quad \neg f \vee \neg x_2 \vee x_1 \vee x_3$$
$$\text{path4}: \quad \neg f \vee x_2$$

(b)

Figure 4.2: Example: (a) search tree of path search with weight ordering, and (b) CNF of path search with weight ordering algorithm

However, this algorithm suffers a similar problem as the linear algorithm [40] that assignments may be difficult to propagate since the number of variables in each clause may be large.

4.2.2 Path search for SOP representation

Generally speaking, path search algorithms can capture both ON- and OFF-set terms (not necessarily minterms though) of a given threshold function. Since a Boolean function can be represented in SOP form, the CNF formulas can also be built in a similar manner. Figure 4.3 demonstrates our algorithm of path search for SOP representation. The difference between the path search with weight ordering and path search for SOP is that, the latter algorithm searches for constant 1 cofactors only, which correspond to the function's ON-set terms.

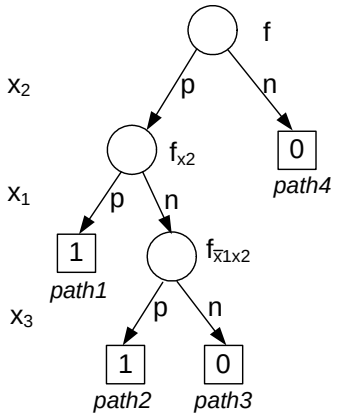
The path search for SOP algorithm searches for the ON-set terms and adds clauses in the same way as the path search with weight ordering algorithm (Lines **3-15**). After a complete search of the decision tree, all the products found are OR-ed together with the corresponding CNF clauses added (Lines **1-2**).

```
path_sop()
Input: f, cls, P(f), W(f)
Output: cnf(f)
1:  if all paths have been searched then
2:    cnf(f) ∧ cnf for sum of P(f)
3:  else if check_constant (f)= 1 then
4:    if cls has only 1 literal then
5:      P(f) ← the only literal
6:      cnf(f) ∧ cls
7:    else
8:      add a new variable v for the resulting product
9:      P(f) ← v
10:     cnf(f) ∧ (cls ∨ v)
11: else
12:   x_i=dequeue(W(f))
13:   cnf(f) ∧ path_sop(f_{x_i}, cls ∨ x̄_i, P(f), W(f))
14:   cnf(f) ∧ path_sop(f_{x̄_i}, cls ∨ x_i, P(f), W(f))
15: return cnf(f)
```

Figure 4.3: Path search for SOP representation

Example 5. *Consider the same threshold function discussed in Example 4. The search tree is shown in Figure 4.4(a). Two paths that lead to a constant 1 cofactor ($path1$ and $path2$) are picked up as the product terms and two new variables (n_1 and n_2) are introduced to denote them. Finally, the CNF clauses of an OR gate (sum) is added to complete the SOP representation (see Figure 4.4(b)).*

Although this algorithm requires a complete search of the decision tree as the path search with weight ordering algorithm does, it alleviates the assignment propagation problem as it reduces the number of those clauses that contain a large number of literals. On the other hand, this SOP form is intrinsically a way of Boolean representation, which increases the probability of equivalent internal nodes between two different circuit realizations, and thus improves SAT solving efficiency.



(a)

path1 : $n_1 \vee \neg x_2 \vee \neg x_1$

path2 : $n_2 \vee \neg x_2 \vee x_1 \vee \neg x_3$

SOP : $f \vee \neg n_1$

$f \vee \neg n_2$

$\neg f \vee n_1 \vee n_2$

(b)

Figure 4.4: Example: (a) search tree of path search for SOP, and (b) CNF of path search for SOP

4.2.3 Hybrid algorithm

The success of the path search for SOP algorithm lies in the recovery of Boolean information from the weight-threshold vector. Based on this observation, the hybrid algorithm is developed which can efficiently capture Boolean information and maintain the advantage of path search. For most of the Boolean primitive gates, two inputs are interchangeable without changing the output functionality. This symmetric property is also true in the weight-threshold vector of a threshold function, as two inputs sharing the same weight are interchangeable. Thus it is more likely that a threshold function with an identical input weight is functionally equivalent to a Boolean gate such as AND ($[1, 1, \ldots, 1; n]$), OR ($[1, 1, \ldots, 1; 1]$), *etc*.

Our hybrid algorithm, shown in Figure 4.5, can facilitate detecting a Boolean primitive gate in a threshold representation. It first analyzes the input weights of the TG, picking up the input variables (say m variables), whose weights are the most common in the threshold function. The rest of the inputs are ordered on input weight as a descending queue $Q(f)$. The cofactoring search process according to the variables in $Q(f)$ is recursively carried out

until a constant cofactor is reached or all the variables in $Q(f)$ have been assigned values (Lines **1-9**).

```
hybrid()
Input: f, cls, Q(f)
Output: cnf(f)
1: if check_constant (f)= 0 or 1 then
2:    cnf(f) ∧ cls
3: else if Q(f) is empty then
4:    if f is a new function then
5:       add new variable for f
6:    else
7:       use the existing variable for f
8:    cnf(f) ∧ cls
9:    cnf(f) ∧ gate_check (f)
10: else
11:   x_i=dequeue(Q(f))
12:   cnf(f) ∧ hybrid(f_{x_i}, cls ∨ x̄_i, Q(f))
13:   cnf(f) ∧ hybrid(f_{x̄_i}, cls ∨ x_i, Q(f))
14: return cnf(f)
```

Figure 4.5: Hybrid algorithm

After this process, since the variables with the same weight have not been cofactored yet (they are not in $Q(f)$), the remaining cofactor is an m-input threshold function with an identical weight w. Function $gate\_check$ is then called to map the cofactor to a Boolean gate if possible. Otherwise, cofactoring on the remaining m-input threshold function continues. Since negative weights have been converted to positive, all the possible Boolean primitive gates that a unate threshold function can be mapped to are AND and OR gates. Other Boolean gates, such as NAND and NOR, and a subset of non-Boolean functions, such as $f = x_1\bar{x}_2$, are all converted to AND or OR gates by the negative to positive unate preprocessing. Based on this observation, the function can be mapped to an AND gate if $mw \geq T > (m-1)w$ or an OR gate if $w \geq T > 0$.

Such a strategy can effectively enhance the conversion process. If a mapping exists, the algorithm can quickly generate the CNF formulas for the known Boolean gates rather than an exhaustive path search. It can also result in a simplified CNF. Another improvement in this algorithm is that, every time the path search process terminates when the remaining inputs have the same weight, the algorithm checks the threshold cofactor to determine if the same cofactor function has been explored before (Lines **4-7**). If yes, the same CNF variable can be used to avoid introducing extra variables and save SAT solving efforts.

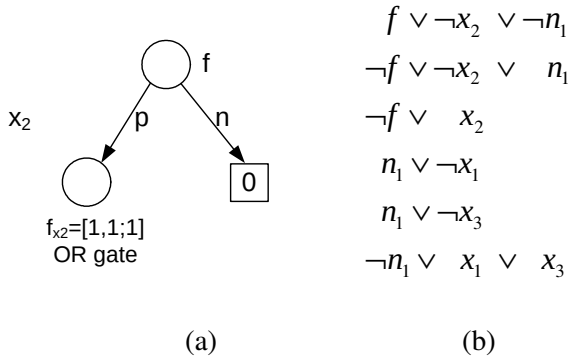


Figure 4.6: Example: (a) search tree, and (b) CNF of hybrid algorithm

Example 6. *Consider the same threshold function $f = [1, 2, 1; 3]$ again. Weight 1 is the most common weight value. Therefore, the second input x_2 is added into $Q(f)$. After the termination of path search process, we have a constant cofactor $f_{\bar{x}_2} = 0$ and $f_{x_2} = [1, 1; 1]$. The latter is thus mapped to a two-input OR gate (see Figure 4.6(a)). A new variable is added for f_{x_2} and the CNF clauses of the OR gate is constructed accordingly as shown in Figure 4.6(b).*

4.3 Experimental results

We evaluated our proposed algorithms based on MCNC benchmarks. The threshold logic synthesizer TELS [25] was used to generate the threshold logic implementations. The equivalence checking between a Boolean and threshold networks and between two threshold networks was conducted. In order to provide a comprehensive comparison of different equivalence checking techniques, the metrics including the numbers of CNF variables and clauses, CNF generating time, and SAT solving time are presented. The experiments were run on an Intel Xeon $3GHz$ workstation with $2GB$ memory running Linux operating system.

4.3.1 Equivalence checking between Boolean and threshold

The set-up of this experiment consists of three steps. In the first step, the MCNC benchmarks are synthesized to Boolean implementations through ABC synthesis tool [33]. Then we convert the Boolean synthesis results to CNF formulas [43] as the Boolean representations. During the second step, the weight-threshold vector representations of the threshold networks are obtained by TELS. Based on these synthesis results, we generate the CNF formulas of the threshold networks using our proposed techniques: path search with weight ordering ($Weight$), path search for SOP representation (SOP), and hybrid algorithm ($Hybrid$). In the last step, the miters are built and Berkmin561 [39] SAT solver is employed to solve the CNF instances.

We compare our work with the approaches for general PB problems [40,41] and specifically for threshold logic [34]. Under column $Linear$ are the CNF instances generated by the linear transformation method proposed in [40]. We also provide the results of modified equivalence checking process in [34] (column BED), by solving CNF formulas rather

than BDD from their BED representation in order to give a fair comparison. The results of CNF transformed via BDD with no extra variables using method in [41] are included under column BDD.

Table 4.1 summarizes the equivalence checking results between Boolean and threshold logic. Columns *Var* and *Cls* refer to the numbers of variables and clauses of the final miter CNF instances, respectively, while column *Time* refers to the SAT solving time in seconds. The best time of each benchmark is highlighted in bold. The last three circuits shown in Table 4.1 are large combinational circuits obtained by unrolling the sequential circuits for 15 timeframes. Figure 4.7 compares the CNF generating time of various algorithms.

Table 4.1: CNF instance and SAT solving time comparisons among different techniques (between Boolean and threshold)

	$Linear$			BED			BDD		
Circuit	Var	Cls	Time (s)	Var	Cls	Time (s)	Var	Cls	Time (s)
i8	7814	36269	6.18	6674	17997	3.84	2296	9884	1.03
t481	9866	52203	41.17	9150	24266	8.51	2270	10970	2.53
too_large	26897	137401	531.79	23445	61823	186.74	6623	28213	102.98
i10	8709	33572	9.79	6387	17652	5.31	3889	14046	3.89
des	16947	81078	39.87	15623	42628	20.68	6267	24012	5.05
s526_15	10237	38202	112.75	7395	21241	60.04	4937	17194	65.94
s1196_15	26462	96384	40.96	18594	52977	26.31	12624	45615	14.25
s1488_15	36548	139701	2069.24	26828	76521	1202.43	16598	64761	1698.40
	$Weight$			SOP			$Hybrid$		
Circuit	Var	Cls	Time (s)	Var	Cls	Time (s)	Var	Cls	Time (s)
i8	2296	9241	0.93	3960	14283	0.65	2417	9483	**0.47**
t481	2270	10506	2.28	5304	22765	2.88	2311	10588	**0.36**
too_large	6623	28179	100.99	13346	43986	4.84	6690	28313	**2.27**
i10	3889	12656	3.50	5697	17488	3.25	4077	13032	**2.10**
des	6267	23916	4.29	9434	34591	3.45	6269	23920	**2.47**
s526_15	4937	16325	59.86	7396	22545	66.33	5430	17311	**46.31**
s1196_15	12624	41037	10.73	19517	59855	12.78	14512	44813	**8.44**
s1488_15	16598	56061	1332.26	26393	84201	1362.12	19883	62631	**1028.90**

As demonstrated in Table 4.1, the CNF instances generated by $Linear$ takes the longest SAT solving time, although the transformation is the most efficient. The main reason is that the linear conversion introduces many XOR relations, which increase workloads to the SAT solver. The BDD method, on the other hand, achieves CNF instances with less number of variables and clauses, and thus a shorter SAT solving time. Although it does not add any additional variables into the formulas, this technique may suffer from a long CNF generating time as shown in Figure 4.7, because its search space is exponential to the number of inputs.

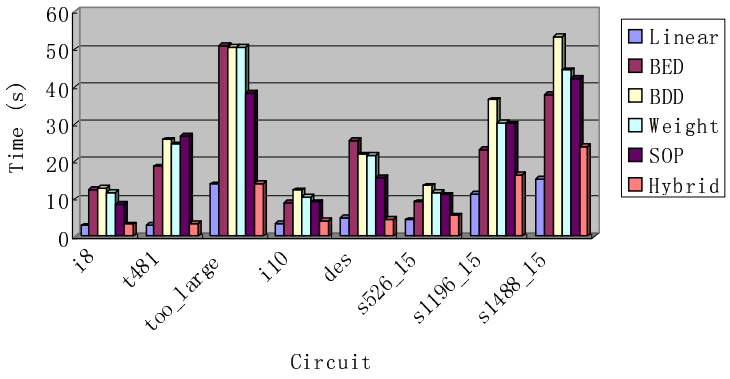


Figure 4.7: CNF formula generating time comparisons

Our $Weight$ method achieves some improvements over the basic algorithms. Cofactoring on the variable with a larger weight enables a quicker termination of solution space search and simplifies the final CNF representation. The SOP strategy can be interpreted as a way of converting threshold logic to Boolean first and then to CNF. The results show that using Boolean as an intermedia can provide better results.

As shown in Table 4.1, our $Hybrid$ technique yields superior performance in SAT solving time for all the benchmarks. Since input symmetry exists in most threshold gates, taking advantage of this property can help quickly identify the cofactor function and generate a neat CNF representation. Although in the worst case, the search space of $Hybrid$ is still

exponential, the CNF generating time on average is comparable to $Linear$ for practical applications. Compared to $Linear$, $Hybrid$ can achieve an average reduction of 81.5% on the critical SAT solving time with little conversion time penalty. In the best case of miter circuit *too_large*, it can achieve a $234X$ speedup for SAT solving.

4.3.2 Equivalence checking between two threshold networks

For the future threshold design, equivalence checking between two threshold networks will become an important aspect in verification. Therefore, we also evaluated the efficiency of proposed algorithms for equivalence checking between two threshold logic networks. The experimental set-up between two threshold networks are similar to that between Boolean and threshold. The same threshold to CNF conversion algorithms are used for two threshold networks, which are obtained by TELS's basic synthesis and synthesis with redundancy removal. The experimental results of SAT solving time are given in Table 4.2, which demonstrate a similar trend as the experiments between Boolean and threshold. The hybrid algorithm achieves an average of 96.4% time reduction and upto 99.8% in the best case.

Table 4.2: SAT solving time comparisons among proposed techniques (between two threshold networks)

Circuit	$Linear$	BED	BDD	$Weight$	SOP	$Hybrid$
i8	17.32	6.74	1.80	1.45	1.13	**0.77**
t481	256.50	30.93	6.90	5.94	7.35	**0.56**
too_large	2201.52	901.05	325.11	308.62	25.87	**8.24**
i10	21.52	9.03	4.92	4.15	4.16	**2.62**
des	89.46	50.06	5.39	4.94	4.36	**1.71**

Chapter 5

Conclusion

In this thesis, an emerging nanotechnology, namely RTD is introduced with its versatile threshold functionality enabled by MOBILE principle. Then novel three- and four-input circuit elements, based on MOBILE TG and MTTG implementations, are proposed. The functional correctness of the circuit is verified by HSPICE simulation. An efficient control bit generating algorithm is developed to configure the structures to realize all three- and four-variable logic functions. Due to the self-latching property of MOBILE circuits, the reconfigurability achieves an average 65% area reduction without delay overheads.

Then we proposed novel techniques of formulating a given threshold logic network in CNF to facilitate efficient SAT-based equivalence checking for emerging nanotechnology threshold designs. Three strategies of CNF generation, namely, path search with weight ordering, path search for SOP representation, and hybrid search, were developed. The hybrid algorithm, which takes into account the input symmetry property as well as input weight ordering of threshold gates, generates CNF instances more efficiently than other techniques.

It achieves an average SAT solving time reduction of 81.5% for equivalence checking between a Boolean and threshold representation. and 96.4% for equivalence checking between two threshold representations. The methodologies we proposed can be applied to general threshold logic to aid future design and verification of emerging nanotechnologies.

Because this is the first work for configurable structure design and SAT-based equivalence checking targeting threshold logic, there is much room for future improvement. For example, this work demonstrates that the PLE structure is suitable for dynamic reconfiguration. The design of such configurable architecture and reconfiguration scheme that facilitates an efficient reconfiguration process is desired. For the SAT-based equivalence checking problem, recent research shows that the identification of equivalence points or cutpoints, can help exploit structural similarities and decompose the problem into smaller pieces thus enables an incremental equivalence checking scheme. Therefore, a static or dynamic exploration of signal relationships greatly benefits the equivalence checking process. We hope the advancement in design methodology and design automation tools for threshold logic can help pave the road for future logic design using nanotechnologies.

Bibliography

[1] *International Technology Roadmap for Semiconductors (ITRS)*, http://www.itrs.net.

[2] W. Porod, C. S. Lent, G. H. Bernstein, A. O. Orlov, I. Amlani, G. L. Snider, and J. L. Merz, "Quantum-dot cellular automata: Computing with coupled quantum dots," *Int. J. Electronics*, vol. 86, no. 5, pp. 549–590, 1999.

[3] M. R. Stan, P. D. Franzon, S. C. Goldstein, J. C. Lach, and M. M. Ziegler, "Molecular electronics: From devices and interconnect to circuits and architecture," *Proc. IEEE*, vol. 91, no. 11, pp. 1940–1957, Nov. 2003.

[4] A. DeHon and K. K. Likharev, "Hybrid CMOS/nanoelectronic digital circuits: Devices, architectures, and design automation," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2005, pp. 375–382.

[5] V. A. Sverdlov, T. J. Walls, and K. K. Likharev, "Nanoscale silicon MOSFETs: A theoretical study," *IEEE Trans. Electron Devices*, vol. 50, no. 9, pp. 1926–1933, Sept. 2003.

[6] K. K. Likharev, "Single-electron transistors: Electrostatic analogs of the DC SQUIDS," *IEEE Trans. Magnetics*, vol. 23, no. 2, pp. 1124–1145, Mar. 1987.

[7] T. A. Fulton, P. L. Gammel, and L. N. Dunkleberger, "Determination of coulomb-blockade resistances and observation of the tunneling of single electrons in small-tunnel-junction circuits," *Physical Reveiw Letters*, vol. 67, no. 22, pp. 3148–3151, Nov. 1991.

[8] S. Tiwari, F. Rana, H. Hanafi, A. Hartstein, E. F. Crabbe, and K. Chan, "A Silicon nanocrystals based memory," *Appl. Phys. Lett.*, vol. 68, no. 10, pp. 1377–1379, Mar. 1996.

[9] L. L. Chang, L. Esaki, and R. Tsu, "Resonant tunneling in semiconductor double barriers," *Appl. Phys. Lett.*, vol. 24, no. 12, pp. 593–595, June 1974.

[10] W. Williamson, 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 J. Solid-State Circuits*, vol. 32, no. 2, pp. 222–231, Feb. 1997.

[11] H. Matsuzaki, T. Itoh, and M. Yamamoto, "A novel high-speed flip-flop circuit using RTDs and HEMTs," in *Proc. Great Lake Symp. VLSI*, Mar. 1999, pp. 154–157.

[12] T. P. E. Broekaert, B. Brar, J. P. A. van der Wagt, A. C. Seabaugh, F. J. Morris, T. S. Moise, E. A. Beam, and G. A. Frazier, "A monolithic 4-bit 2-Gsps resonant tunneling analog-to-digital converter," *IEEE J. Solid-State Circuits*, vol. 33, no. 9, pp. 1342–1349, Sept. 1998.

[13] J. P. A. van der Wagt, "Tunneling-based SRAM," *Proc. IEEE*, vol. 87, no. 4, pp. 571–595, Apr. 1999.

[14] J. P. Shen, G. Kramer, S. Tehrani, and H. Goronkin, "Static random access memories based on resonant interband tunneling diodes in the InAs/GaSb/AlSb material system," *IEEE Electron Device Letters*, vol. 16, no. 5, pp. 178–180, May 1995.

[15] M. Bhattacharya, S. Kulkarni, A. Gonzalez, and P. Mazumder, "A prototyping technique for large-scale RTD-CMOS circuits," in *Proc. Int. Symp. Circuits & Systems*, May 2000, pp. 635–638.

[16] J. I. Bergman, J. Chang, Y. Joo, B. Matinpour, J. Laskar, N. M. Jokerst, M. A. Brooke, B. Brar, and E. Beam, "RTD/CMOS nanoelectronic circuits: Thin-film InP-based resonant tunneling diodes integrated with CMOS circuits," *IEEE Electron Device Letters*, vol. 20, no. 3, pp. 119–122, Mar. 1999.

[17] J. N. Schulman, H. J. De Los Santos, and D. H. Chow, "Physics-based RTD current-voltage equation," *IEEE Electron Device Letters*, vol. 17, no. 5, pp. 220–222, May 1996.

[18] Z. Yan and M. J. Deen, "New RTD large-signal DC model suitable for PSPICE," *IEEE Trans. Computer-Aided Design of Integrated Circuits & Systems*, vol. 14, no. 2, pp. 167–172, Feb. 1995.

[19] J. Sun, G. I. Haddad, P. Mazumder, and J. N. Schulman, "Resonant tunneling diodes: Models and properties," *Proc. IEEE*, vol. 86, no. 4, pp. 641–660, Apr. 1998.

[20] M. Bhattacharya and P. Mazumder, "Augmentation of SPICE for simulation of circuits containing resonant," *IEEE Trans. Computer-Aided Design of Integrated Circuits & Systems*, vol. 20, no. 1, pp. 39–50, Jan. 2001.

[21] K. Maezawa, T. Akeyoshi, and T. Mizutani, "Functions and applications of monostable-bistable transition logic elements (MOBILEs) having multiple-input terminals," *IEEE Trans. Electron Devices*, vol. 41, no. 2, pp. 148–154, Feb. 1994.

[22] 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, no. 3, pp. 80–82, Mar. 1998.

[23] M. J. Avedillo, J. M. Quintana, H. Pettenghi, P. M. Kelly, and C. J. Thompson, "Multi-threshold threshold logic circuit design using resonant tunneling devices," *Electron Letters*, vol. 39, no. 21, pp. 1502–1504, Oct. 2003.

[24] C. Sheng, *Threshold Logic*, Academic Press, New York, NY, 1969.

[25] R. Zhang, P. Gupta, L. Zhong, and N. K. Jha, "Synthesis and optimization of threshold logic networks with application to nanotechnologies," in *Proc. Design Automation & Test Europe Conf.*, Feb. 2004, pp. 904–909.

[26] P. Gupta, R. Zhang, and N. K. Jha, "An automatic test pattern generation framework for combinational threshold logic networks," in *Proc. Int. Conf. Computer Design*, Oct. 2004, pp. 540–543.

[27] P. Mazumder, S. Kulkarni, M. Bhattacharya, J. P. Sun, and G. I. Haddad, "Digital circuit applications of resonant tunneling devices," *Proc. IEEE*, vol. 86, no. 4, pp. 664–686, Apr. 1998.

[28] P. Gupta and N. K. Jha, "An algorithm for nano-pipelining of circuits and architectures for a nanotechnology," in *Proc. Design Automation & Test Europe Conf.*, Feb. 2004, pp. 974–979.

[29] M. J. Avedillo, J. M. Quintana, and H. P. Roldan, "Increased logic functionality of clocked series-connected RTDs," *IEEE Trans. Nanotechnology*, vol. 5, no. 5, pp. 606–611, Sept. 2006.

[30] C. Pacha, U. Auer, C. Burwick, P. Glosekotter, A. Brennemann, W. Prost, F. J. Tegude, and K. F. Goser, "Threshold logic circuit design of parallel adders using resonant tunneling deviecs," *IEEE Trans. Very Large Scale Integration Systems*, vol. 8, no. 5, pp. 558–572, Oct. 2000.

[31] H. Pettenghi, M. J. Avedillo, and J. M. Quintana, "Using multi-threshold threshold gates in RTD-based logic design, a case study," in *Euro. Nano. System*, Dec. 2005, pp. 14–16.

[32] D. Brand, "Verification of large synthesized designs," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 534–537.

[33] Berkeley Logic Synthesis & Verification Group, *ABC: A system for sequential synthesis and verification*, http://www.eecs.berkeley.edu/∼alanmi/abc/.

[34] T. Gowda, S. Vrudhula, and G. Konjevod, "Combinational equivalence checking for threshold logic circuits," in *Proc. Great Lake Symp. VLSI*, Mar. 2007, pp. 102–107.

[35] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2001, pp. 114–121.

[36] H. Zhang, "SATO: An efficient propositional prover," in *Proc. Int. Conf. Automated Deduction*, July 1997, pp. 272–275.

[37] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 593–595, May 1999.

[38] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Efficient SAT solver," in *Proc. Design Automation Conf.*, June 2001, pp. 530–535.

[39] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust SAT solver," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2002, pp. 142–149.

[40] J. P. Warners, "A linear-time transformation of linear inequalities into conjuctive normal form," *Information Processing Letters*, vol. 68, no. 2, pp. 63–69, Oct. 1998.

[41] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Generic ILP versus specialized 0-1 ILP: An update," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2002, pp. 450–457.

[42] N. Een and N. Sorensson, "Translating pseudo-Boolean constraints in SAT," *J. Satisfiability, Boolean Modeling & Computation*, vol. 2, pp. 1–26, Feb. 2006.

[43] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. Computer-Aided Design of Integrated Circuits & Systems*, vol. 11, no. 1, pp. 4–15, Jan. 1992.