


12-2014

# Technology Mapping, Design for Testability, and Circuit Optimizations for NULL Convention Logic Based Architectures

Farhad Alibeygi Parsan  
*University of Arkansas, Fayetteville*

Follow this and additional works at: <http://scholarworks.uark.edu/etd>

 Part of the [Digital Circuits Commons](#), and the [VLSI and Circuits, Embedded and Hardware Systems Commons](#)

---

## Recommended Citation

Alibeygi Parsan, Farhad, "Technology Mapping, Design for Testability, and Circuit Optimizations for NULL Convention Logic Based Architectures" (2014). *Theses and Dissertations*. 2119.  
<http://scholarworks.uark.edu/etd/2119>

This Dissertation is brought to you for free and open access by ScholarWorks@UARK. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of ScholarWorks@UARK. For more information, please contact [scholar@uark.edu](mailto:scholar@uark.edu), [ccmiddle@uark.edu](mailto:ccmiddle@uark.edu).

Technology Mapping, Design for Testability, and Circuit Optimizations for  
NULL Convention Logic Based Architectures

Technology Mapping, Design for Testability, and Circuit Optimizations for  
NULL Convention Logic Based Architectures

A dissertation submitted in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy in Electrical Engineering

by

Farhad Alibeygi Parsan  
Iran University of Science and Technology  
Bachelor of Science in Electrical Engineering, 2005  
Iran University of Science and Technology  
Master of Science in Electrical Engineering, 2008

December 2014  
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

---

Dr. Scott C. Smith  
Dissertation Director

---

Dr. Waleed K. Al-Assadi  
Committee Member

---

Dr. Jia Di  
Committee Member

---

Dr. Alan Mantooth  
Committee Member

---

Dr. Jingxian Wu  
Committee Member

## **ABSTRACT**

Delay-insensitive asynchronous circuits have been the target of a renewed research effort because of the advantages they offer over traditional synchronous circuits. Minimal timing analysis, inherent robustness against power-supply, temperature, and process variations, reduced energy consumption, less noise and EMI emission, and easy design reuse are some of the benefits of these circuits. NULL Convention Logic (NCL) is one of the mainstream asynchronous logic design paradigms that has been shown to be a promising method for designing delay-insensitive asynchronous circuits.

This dissertation investigates new areas in NCL design and test and is made of three sections. The first section discusses different CMOS implementations of NCL gates and proposes new circuit techniques to enhance their operation. The second section focuses on mapping multi-rail logic expressions to a standard NCL gate library, which is a form of technology mapping for a category of NCL design automation flows. Finally, the last section proposes design for testability techniques for a recently developed low-power variant of NCL called Sleep Convention Logic (SCL).

## **ACKNOWLEDGMENTS**

I am deeply grateful to my advisor Dr. Scott C. Smith, who introduced me to the world of asynchronous digital design and has provided me with guidance and financial support throughout the research and preparation of this dissertation. I would also like to thank my other advisory committee members, Dr. Al-Assadi, Dr. Di, Dr. Mantooth, and Dr. Wu. Also, a special thanks goes out to the faculty and staff at the University of Arkansas for their commitment to the students and to the University.

## **DEDICATION**

I dedicate my dissertation to my loving parents, Alidad Alibeygi Parsan and Mahnesa Mohammadi Tahroudi, whose affection, love, encouragement, and unconditional support made this work possible. I also dedicate this dissertation to my love, Jingyi Zhao, for her unwavering support and encouragement throughout my doctorate program.

## TABLE OF CONTENTS

<b>1</b>	<b>NCL OVERVIEW</b>	<b>1</b>
1.1	INTRODUCTION . . . . .	1
1.2	NCL DESIGN METHODOLOGY . . . . .	1
<b>2</b>	<b>NCL GATE DESIGN OPTIMIZATION</b>	<b>10</b>
2.1	INTRODUCTION . . . . .	10
2.2	DYNAMIC GATES . . . . .	11
2.3	SEMI-STATIC GATES . . . . .	13
2.4	DIFFERENTIAL GATES . . . . .	17
2.5	STATIC GATES . . . . .	18
2.6	COMPARISON OF DIFFERENT GATE STYLES . . . . .	22
2.6.1	MINIMUM POWER-SUPPLY VOLTAGE . . . . .	22
2.6.2	AVERAGE ENERGY PER OPERATION . . . . .	24
2.6.3	AVERAGE DELAY PER OPERATION . . . . .	25
2.6.4	AVERAGE POWER . . . . .	25
2.6.5	AREA . . . . .	25
2.6.6	NOISE SUSCEPTIBILITY . . . . .	26
2.6.7	DISCUSSION . . . . .	28
2.7	NEW STATIC GATES . . . . .	29
2.7.1	DESIGNING NEW STATIC GATES . . . . .	29
2.7.2	SIZING NEW STATIC GATES . . . . .	35
2.7.3	EXPERIMENTAL RESULTS . . . . .	36
2.8	CONCLUSION . . . . .	37
<b>3</b>	<b>NCL TECHNOLOGY MAPPING</b>	<b>38</b>
3.1	INTRODUCTION . . . . .	38
3.2	EXISTING NCL DESIGN AUTOMATION FLOWS . . . . .	40
3.3	CONVENTIONS . . . . .	47
3.4	ORIGINAL MAPPING ALGORITHM . . . . .	51
3.5	ORIGINAL MAPPING ALGORITHM CONSTRAINTS . . . . .	56
3.6	PROPOSED MAPPING ALGORITHM . . . . .	58
3.7	IMPROVEMENTS OVER THE ORIGINAL MAPPING ALGORITHM . . . . .	66
3.8	EXPERIMENTAL RESULTS . . . . .	69
3.9	CONCLUSION . . . . .	73
<b>4</b>	<b>SCL DESIGN FOR TESTABILITY</b>	<b>75</b>
4.1	INTRODUCTION . . . . .	75
4.2	PREVIOUS WORKS . . . . .	80
4.3	PROPOSED DESIGN FOR TESTABILITY METHODOLOGY . . . . .	82
4.3.1	MOTIVATION . . . . .	82
4.3.2	LOGIC FAULT ANALYSIS . . . . .	83
4.3.3	SLEEP SIGNAL FAULT ANALYSIS . . . . .	87

4.3.4	TEST PROCEDURE . . . . .	91
4.4	EXPERIMENTAL RESULTS . . . . .	96
4.5	CONCLUSION . . . . .	98
<b>5</b>	<b>REFERENCES</b>	<b>99</b>



## LIST OF FIGURES

1	Symbolically complete logic concept . . . . .	1
2	(a) Boolean AND gate versus (b) NCL AND circuit . . . . .	3
3	(a) NCL threshold gate symbol (b) a weighted NCL threshold gate . . . . .	4
4	NCL AND circuit . . . . .	6
5	NCL design framework . . . . .	7
6	DATA/NULL cycle . . . . .	7
7	A single-bit dual-rail NCL register . . . . .	8
8	NCL completion detection block . . . . .	8
9	(a) Structure of NCL dynamic gates (b) TH23 dynamic gate . . . . .	12
10	(a) Structure of NCL semi-static gates (b) TH23 semi-static gate . . . . .	14
11	Different feedback inverter weakening methods (a) standard method (b) resistive method (c) proposed diode-connected method . . . . .	15
12	(a) Structure of NCL differential gates (b) TH23 differential gate . . . . .	18
13	(a) Structure of NCL static gates (b) TH23 static gate . . . . .	19
14	Minimum power-supply voltage . . . . .	23
15	Average energy per operation . . . . .	23
16	Average delay per operation . . . . .	24
17	Average power . . . . .	26
18	Area . . . . .	27
19	Noise susceptibility . . . . .	27
20	(a) Original TH23 static gate (b) Proposed TH23 static gate . . . . .	29
21	A sizing example for (a) original (b) new static TH23 gates . . . . .	36
22	Automated NCL design flows . . . . .	43
23	Original grouping algorithm . . . . .	50
24	NCL implementation . . . . .	55
25	New grouping algorithm . . . . .	62
26	NCL implementation . . . . .	65
27	Dependency trees . . . . .	67
28	Single-variable term relocation . . . . .	70
29	SCL framework . . . . .	75
30	SCL gate structure . . . . .	76
31	SCL register implementation . . . . .	77
32	SCL completion detector . . . . .	78
33	SCL scan cell . . . . .	92
34	SCL scan chain design . . . . .	93
35	SCL scan cell design . . . . .	94
36	Modified version of the original SCL register for a single rail . . . . .	95

## LIST OF TABLES

1	Dual-rail encoding . . . . .	2
2	Standard NCL gate library . . . . .	5
3	Power, delay, and energy at 1.2v . . . . .	22
4	Original complex static gates versus the new versions . . . . .	32
5	Original C-elements versus the new versions . . . . .	33
6	Comparison between original and new static gate styles . . . . .	34
7	Comparison of NCL multipliers realized with different static gate styles . . . . .	36
8	Comparison of the two NCL design flow categories . . . . .	46
9	Library of the 27 standard NCL gates . . . . .	49
10	K-feasibility table . . . . .	52
11	K-feasible groups for each parent term . . . . .	52
12	Final k-feasible groups after first iteration . . . . .	53
13	Final groups . . . . .	55
14	A typical priority table . . . . .	60
15	K-feasible groups for each parent term . . . . .	61
16	K-feasible groups pushed into their corresponding priority levels . . . . .	64
17	Updated priority levels . . . . .	64
18	Final groups . . . . .	65
19	Comparison of the original and proposed grouping methods . . . . .	71
20	Mapping circuit components using the original and proposed grouping methods . . . . .	72
21	Mapping circuit components to a restricted subset of NCL gates . . . . .	72
22	Comparison of the two NCL design flow categories . . . . .	74
23	Experimental results . . . . .	97

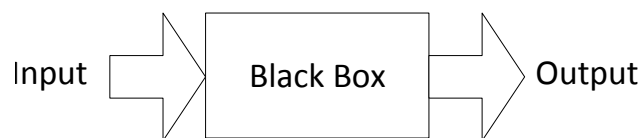
# 1 NCL OVERVIEW

## 1.1 INTRODUCTION

Delay-insensitive asynchronous circuits have been the target of a renewed research effort because of the advantages they offer over traditional synchronous circuits. Minimal timing analysis, inherent robustness against power-supply, temperature, and process variations, reduced energy consumption, less noise and EMI emission, and easy design reuse are some of the benefits of these circuits [1]. NULL Convention Logic (NCL) is one of the mainstream asynchronous logic design paradigms that has been shown to be a promising method for designing delay-insensitive asynchronous circuits [2–5]. NCL circuits are correct-by-construction [3], requiring very little timing analysis, if any. In today’s nanometer processes where meeting timing closure is becoming increasingly more difficult due to increasing clock rates and process variation, this quality is very attractive. NCL has been used for a number of industrial designs [4, 6], and is becoming more popular as design automation tools and techniques are being developed to automate the design process [4, 7–10].

## 1.2 NCL DESIGN METHODOLOGY

NCL is a delay-insensitive asynchronous logic design paradigm in which control is inherent within each datum. It follows the so-called “weak conditions” of Seitz’s delay-insensitive signal-



**Figure 1: Symbolically complete logic concept**

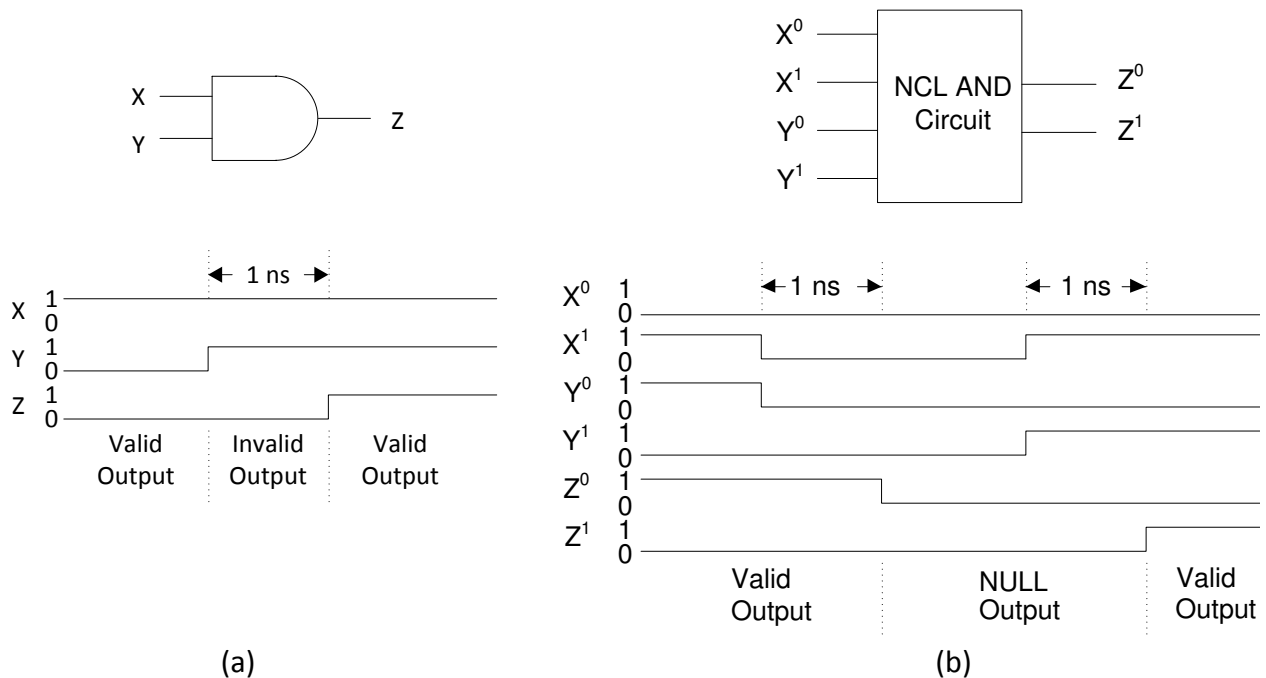
**Table 1: Dual-rail encoding**

	<b>DATA0</b>	<b>DATA1</b>	<b>NULL</b>	<b>Illegal</b>
<b>D<sup>0</sup></b>	1	0	0	1
<b>D<sup>1</sup></b>	0	1	0	1

ing scheme [11]. Similar to other delay-insensitive logic design paradigms, NCL assumes that wire forks are isochronic [12]. NCL is a “symbolically complete” logic meaning that the output validity is unambiguously determined regardless of time reference [3]. Figure 1 shows an unknown circuit inside a black box. Assuming that the unknown circuit is a traditional Boolean combinational circuit, once the inputs are asserted, it is impossible to determine when the outputs become valid unless the circuit’s delay is known. However, if the unknown circuit is using a symbolically complete logic, such as NCL, one can determine the output validity without needing to know the circuit’s delay. This is because NCL uses delay-insensitive codes for data communication, alternating between set and reset phases. In the set phase, data changes from spacer (called NULL) to a proper codeword (called DATA); and in the reset phase it changes back to NULL. NCL combines DATA and NULL into a single path presented by dual-rail, quad-rail, or in general, any Mutually Exclusive Assertion Group (MEAG) signals [13].

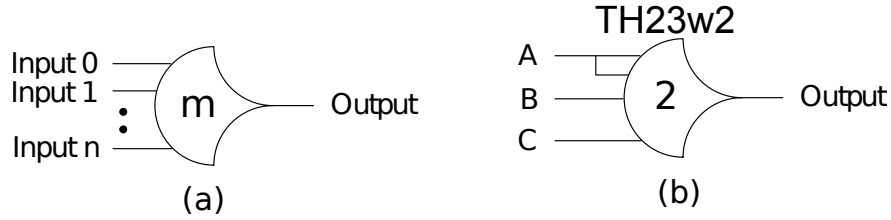
In practice, dual-rail signal encoding is more popular, since it is most similar to traditional Boolean logic. Table 1 shows the dual-rail signal encoding. A dual-rail signal,  $D$ , consists of two wires,  $D^0$  and  $D^1$ .  $D$  is logic 0 (DATA0) when  $D^0 = 1$  and  $D^1 = 0$ ; it is logic 1 (DATA1) when  $D^0 = 0$  and  $D^1 = 1$ ; and it is NULL when  $D^0 = 0$  and  $D^1 = 0$ .  $D^0$  and  $D^1$  are mutually exclusive, such that they are never asserted at the same time; doing so would produce an illegal codeword.

Figure 2 shows a simple Boolean AND gate versus a dual-rail NCL circuit that performs the same AND operation. For the Boolean AND gate, inputs  $X$ , and  $Y$ , and output  $Z$  use only one



**Figure 2: (a) Boolean AND gate versus (b) NCL AND circuit**

wire, but the dual-rail NCL AND circuit uses two wires for each input and output. For the Boolean AND gate, initially  $X = 1$  and  $Y = 0$ , so output  $Z$  is 0. For the NCL AND circuit, initially  $X$  is DATA1 ( $X^1 = 1, X^0 = 0$ ) and  $Y$  is DATA0 ( $Y^0 = 1, Y^1 = 0$ ); therefore, output  $Z$  is DATA0 ( $Z^0 = 1, Z^1 = 0$ ). For the Boolean AND gate, once input  $Y$  is asserted, output  $Z$  becomes invalid until the signal propagates through the AND gate and asserts the output (in this example after 1 ns). For the NCL AND circuit, however, before input  $Y$  changes to its next DATA value, all inputs must first transition to the NULL state (i.e., all input rails must go to 0) and we must wait until the output then transitions to NULL. At this point, the circuit is ready to accept a new DATA set, so  $X$  and  $Y$  can both change from NULL to DATA1. Consequently, output  $Z$  then changes from NULL to DATA1 after some time (1 ns in this example). An NCL circuit always cycles through NULL and DATA phases so the validity of the output can always be unambiguously determined by merely looking at the output. A NULL at the output means that the output is not valid and a DATA at the



**Figure 3: (a) NCL threshold gate symbol (b) a weighted NCL threshold gate**

output means that the output is valid. For a Boolean circuit, on the other hand, the output validity can only be determined if we know when the inputs change and the worst-case propagation delay of the circuit.

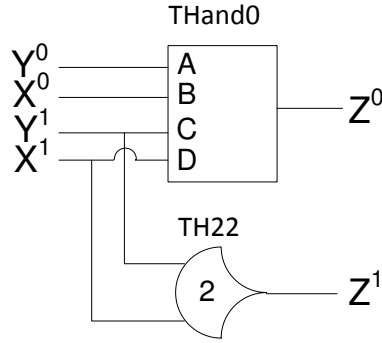
NCL circuits are comprised of 27 threshold gates with hysteresis [2]. Each gate is denoted as  $THmnWw_1w_2\dots w_r$  in which  $m$  is the threshold of the gate,  $n$  is the number of inputs, and  $w_r$  is the weight of input  $r$  if its weight is greater than 1. Figure 3(a) shows the symbol of an NCL gate. For an NCL gate with no weighted inputs, the output is asserted when at least  $m$  out of  $n$  inputs are asserted. As an example, the TH23 gate asserts its output when at least two out of three inputs are asserted; therefore, assuming the inputs are  $A$ ,  $B$ , and  $C$ , the set function of a TH23 gate can be expressed as  $F = AB + AC + BC$ . Figure 3(b) shows a TH23w2 gate, where input  $A$  has a weight of two. Therefore, asserting  $A$  alone asserts the gate output. The set function of the TH23w2 gate can then be expressed as  $F = A + BC$ .

The standard NCL gate library is shown in Table 2. Since NCL gates have hysteresis, once the output is asserted, it remains asserted until all the inputs are deasserted. Hysteresis behavior is required to ensure the delay-insensitivity of NCL circuits [2]. A non-weighted NCL gate with  $m = n$  (i.e., THnn) is a special case of NCL gates that is equivalent to an  $n$ -input C-element [14]. C-elements are well-known gates used in many other asynchronous logic design styles. A non-

**Table 2: Standard NCL gate library**

<b>NCL Gate</b>	<b>Set Function</b>
TH12	$A + B$
TH22	$AB$
TH13	$A + B + C$
TH23	$AB + AC + BC$
TH33	$ABC$
TH23w2	$A + BC$
TH33w2	$AB + AC$
TH14	$A + B + C + D$
TH24	$AB + AC + AD + BC + BD + CD$
TH34	$ABC + ABD + ACD + BCD$
TH44	$ABCD$
TH24w2	$A + BC + BD + CD$
TH34w2	$AB + AC + AD + BCD$
TH44w2	$ABC + ABD + ACD$
TH34w3	$A + BCD$
TH44w3	$AB + AC + AD$
TH24w22	$A + B + CD$
TH34w22	$AB + AC + AD + BC + BD$
TH44w22	$AB + ACD + BCD$
TH54w22	$ABC + ABD$
TH34w32	$A + BC + BD$
TH54w32	$AB + ACD$
TH44w322	$AB + AC + AD + BC$
TH54w322	$AB + AC + BCD$
THxor0	$AB + CD$
THand0	$AB + BC + AD$
TH24comp	$AC + BC + AD + BD$

weighted NCL gate with  $m = 1$  (i.e., TH1n) is another special case of NCL gates that is equivalent to an n-input Boolean OR gate. Among the 27 NCL gates, there are 3 gates (TH24comp, Thand0, THxor0) that are not actually threshold gates, but can be made by combining other threshold gates. These gates are included in the standard NCL gate library so that any function of 4 or fewer variables directly maps to one of these 27 NCL gates. Due to hysteresis, NCL gates act as memory elements; therefore, like any other memory element they have to be initialized. Initialization can be performed implicitly by asserting/deasserting all the gate inputs, or it can be done explicitly by



**Figure 4: NCL AND circuit**

adding a reset input to the gate. Depending on whether the reset signal asserts or deasserts the gate output, resettable gates are denoted with an ‘n’ (output deasserted) or a ‘d’ (output asserted) at the end of their name. Additionally, the output of an NCL gate can be provided in its inverted form; this is denoted by a small circle at the output of the gate symbol and a ‘b’ at the end of the gate name. Figure 4 shows how the NCL AND circuit in Figure 2 can be built using two NCL gates, based on the canonical SOP equations for both the rail1 and rail0 outputs, shown in equations 1 and 2, respectively, and mapping these to the set function of the gates shown in Table 2.

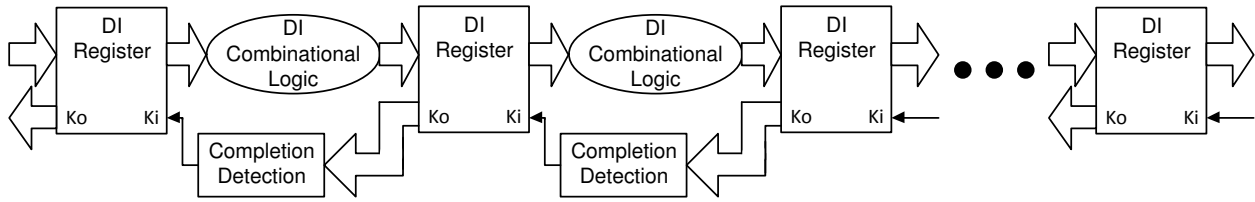
$$Z^1 = X^1Y^1 \quad (1)$$

$$Z^0 = X^0Y^0 + X^0Y^1 + X^1Y^0 \quad (2)$$

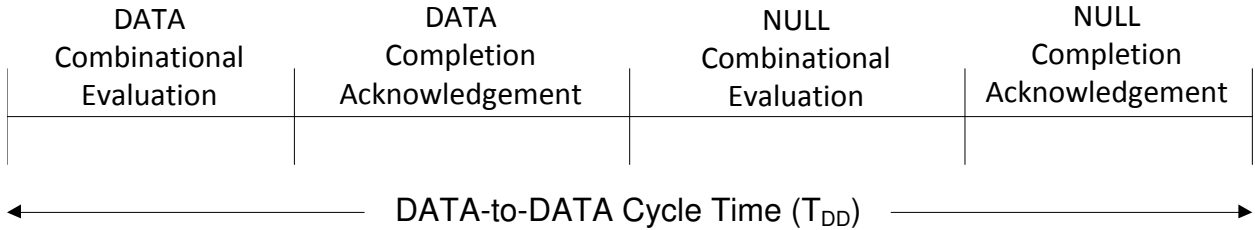
Therefore, output  $Z$  becomes DATA1 when both  $X$  and  $Y$  are DATA1 and it becomes DATA0 when either input is DATA0 and the other input is DATA (i.e., DATA0 or DATA1). Reference [2] elaborates on how to design more complex combinational logic circuits using NCL.

The NCL design framework consists of delay-insensitive (DI) Combinational Logic blocks



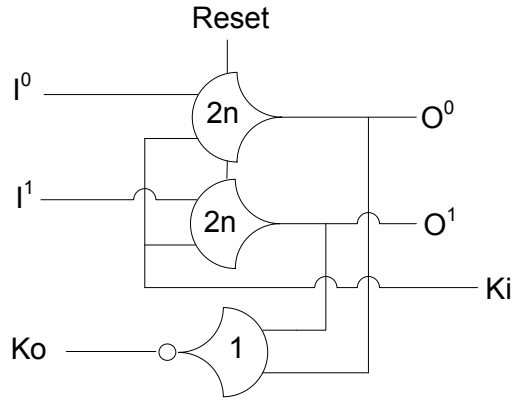


**Figure 5: NCL design framework**

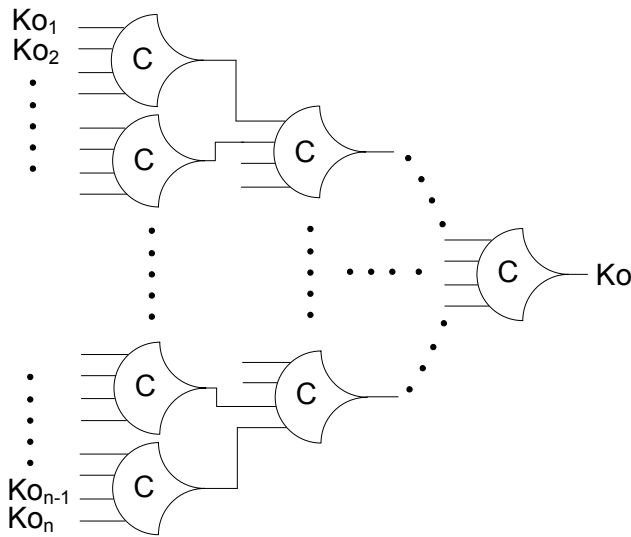


**Figure 6: DATA/NULL cycle**

sandwiched between DI Registers. This design framework, shown in Figure 5, is very similar to the traditional synchronous design framework, except that Completion Detection blocks are used to synchronize data communication instead of a global clock. Completion Detection block checks the output of a register to see if the previous DATA (NULL) has successfully propagated through the Combinational Logic; if so, it then allows the next NULL (DATA) to start propagating through the Combinational Logic.  $K_i$  and  $K_o$  are the handshaking signals used for requesting and acknowledging DATA and NULL wavefronts. A typical DATA/NULL cycle is shown in Figure 6. It starts with DATA propagating through a combinational block; once DATA passes the following register, the completion detection block acknowledges that DATA evaluation is finished and that NULL can now propagate. Then NULL propagates through the combinational block and clears the previous DATA; once NULL passes the register, the completion detection block acknowledges that NULL propagation is complete and allows the next DATA to start propagating through the combination block. The time period between two consequent DATA phases is called the DATA-to-DATA Cycle Time ( $T_{DD}$ ), and is a measure of an NCL pipeline's throughput.



**Figure 7: A single-bit dual-rail NCL register**



**Figure 8: NCL completion detection block**

A single-bit dual-rail NCL register is shown in Figure 7, where  $I^0$  and  $I^1$  are the input rails and  $O^0$  and  $O^1$  are the output rails. A single-bit NCL register is comprised of two TH2n gates and one inverting TH12 gate. When a combinational block is ready for DATA,  $Ki$  is asserted, allowing DATA to pass through the register; and once DATA is evaluated by the combinational block,  $Ki$  is deasserted, allowing NULL to pass through. The  $Ki$  signals of a multi-bit register are all connected together and connected to the output of the completion detection block of the next register.

The completion detection block detects whether there is a complete DATA/NULL set at

the output of a register. When a register's output is NULL (i.e., both output rails in Figure 7 are deasserted), the inverting TH12 gate is asserted to request the next DATA (rfd). When a register's output is DATA (i.e., either of the output rails in Figure 7 is asserted), the inverting TH12 gate is deasserted to request NULL (rfn). All  $Ko$  outputs of a multi-bit register are input to a completion detection block that asserts its output when all  $Ko$  signals are rfd, and deasserts its output when all  $Ko$  signals are rfn. An  $n$ -bit completion detection block, shown in Figure 8, is equivalent to an  $n$ -input C-element, comprised of THnn gates. The minimum number of levels required for a completion detection block is  $\lceil \log_4 n \rceil$ , where  $n$  is the number of  $Ko$  signals [2].

## **2 NCL GATE DESIGN OPTIMIZATION**

©2012 IEEE. Reprinted, with permission, from F. A. Parsan and S. C. Smith, “CMOS implementation of static threshold gates with hysteresis: A new approach,” in VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on, pp. 41–45, October 2012.

©2012 IEEE. Reprinted, with permission, from F. A. Parsan and S. C. Smith, “CMOS implementation comparison of NCL gates,” in Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on, pp. 394–397, August 2012.

In reference to IEEE copyrighted material which is used with permission in this dissertation, the IEEE does not endorse any of University of Arkansas’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to [http://www.ieee.org/publications\\_standards/publications/rights/rights\\_link.html](http://www.ieee.org/publications_standards/publications/rights/rights_link.html) to learn how to obtain a License from RightsLink.

### **2.1 INTRODUCTION**

NCL circuits utilize threshold gates with hysteresis to maintain delay insensitivity. The general form of an NCL gate is very similar to a C-element [15]. Several CMOS implementation schemes have been introduced for NCL gates, including: dynamic, static, semi-static, and differential [16–20]. Each implementation offers some advantages and has some drawbacks in terms of delay, area, and power consumption. It is important for an NCL circuit designer to choose the CMOS implementation that best fits an application. In this chapter, different CMOS implementa-

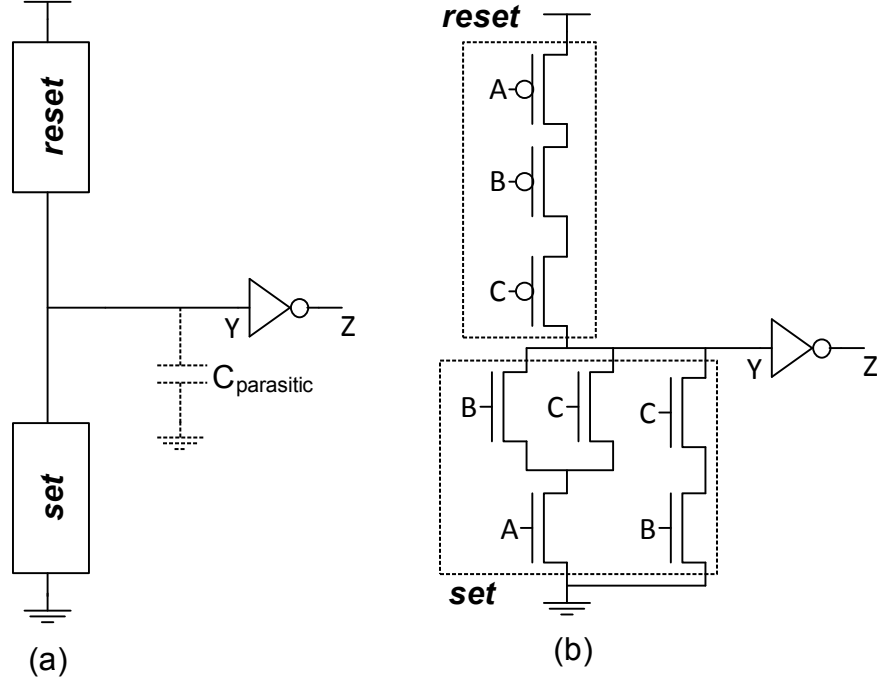
tions of NCL gates are introduced and their tradeoffs are discussed. In addition, a new approach to design static and semi-static NCL gates is proposed and compared with the traditional approaches in terms of delay, area, and energy consumption. It will be shown that the new gate designs offer faster operation, with a small increase in area, and consume almost the same (static) or less (semi-static) amount of energy.

This chapter is organized as follows: The first few sections discuss different CMOS implementations of NCL gates: Dynamic implementation in Section 2.2, Semi-static implementation in Section 2.3, Differential implementation in Section 2.4, and finally Static implementation in Section 2.5. Various implementations are then compared against each other in terms of area, delay, and power/energy consumption in Section 2.6. Finally, Section 2.7 details the proposed new static gate design and compares it to the traditional design in different ways.

## 2.2 DYNAMIC GATES

The dynamic implementation of NCL gates can be used in real-time computing applications where a minimum data rate is guaranteed so that the state information can be maintained on an isolated node. The structure of an NCL dynamic gate is shown in Figure 9(a).

The set block realizes the set function of an NCL gate, such that when the set function becomes true, the set block becomes active and discharges the internal node  $Y$ , causing output  $Z$  to be asserted. Similarly, when all inputs are deasserted, the reset block becomes active and charges the internal node  $Y$  to  $V_{DD}$ , causing output  $Z$  to be deasserted. In a CMOS implementation of NCL dynamic gates, the set block is a pull-down network of NMOS transistors, derived from the equations in Table 2 for each of the 27 NCL gates. On the other hand, the reset block is always a



**Figure 9: (a) Structure of NCL dynamic gates (b) TH23 dynamic gate**

series chain of PMOS transistors consisting of one transistor per input; therefore, NCL gates that have the same number of inputs have the same reset block. The reset function of an NCL gate with  $n$  inputs can be expressed as:

$$reset = I'_1 \bullet I'_2 \bullet \dots \bullet I'_n \quad (3)$$

where  $I_n$  represents input  $n$ . For most NCL gates, the set and reset functions are not complements of each other, so there are times when neither the set nor reset block is active. In a dynamic implementation, when neither is active, the internal node Y will be floating, so its value will be preserved on its parasitic capacitance,  $C_{\text{parasitic}}$ , for a few milliseconds before its charge leaks away, enabling the NCL gate to maintain its state, but only for a finite amount of time. Therefore, once the set function becomes true and the output is asserted, it remains asserted until the reset func-

tion becomes true and deasserts the output (hysteresis behavior). Figure 9(b) shows the dynamic implementation of a TH23 gate, whose set function is:

$$F = AB + AC + BC \quad (4)$$

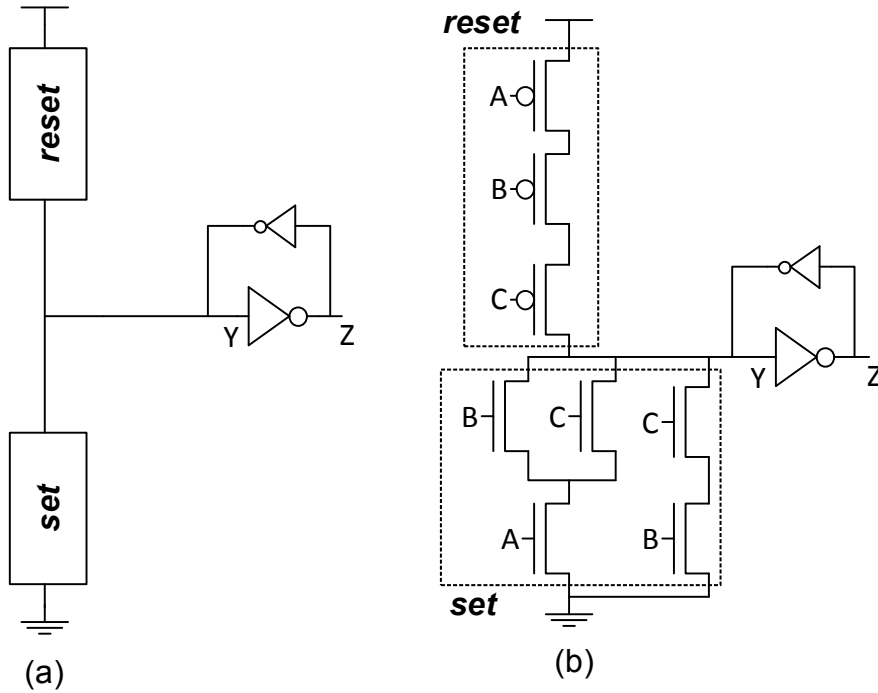
and the set function can then be factored to reduce the number of transistors:

$$F = A(B + C) + BC \quad (5)$$

The NCL dynamic implementation is the smallest and fastest NCL gate style, and consumes the least amount of energy; however, since its output cannot be held indefinitely when neither set nor reset is active, it is not considered a delay-insensitive solution. Moreover, since the state information is stored on a small parasitic capacitance, it is very vulnerable to noise and charge sharing effects, although the latter can be alleviated by transistor reordering in the pull-down network [16], careful transistor sizing, and post-layout simulations. For these reasons, dynamic NCL gates are rarely used in real applications.

### 2.3 SEMI-STATIC GATES

The semi-static (or pseudo-static) implementation of NCL gates utilizes feedback to maintain state information, and therefore, does not require a minimum input data rate, since it can hold the output state indefinitely. The structure of an NCL semi-static gate is shown in Figure 10(a). In a semi-static implementation, the state information is maintained via a staticizer, in the form of a weak feedback inverter. The weak feedback inverter compensates for the leakage current that

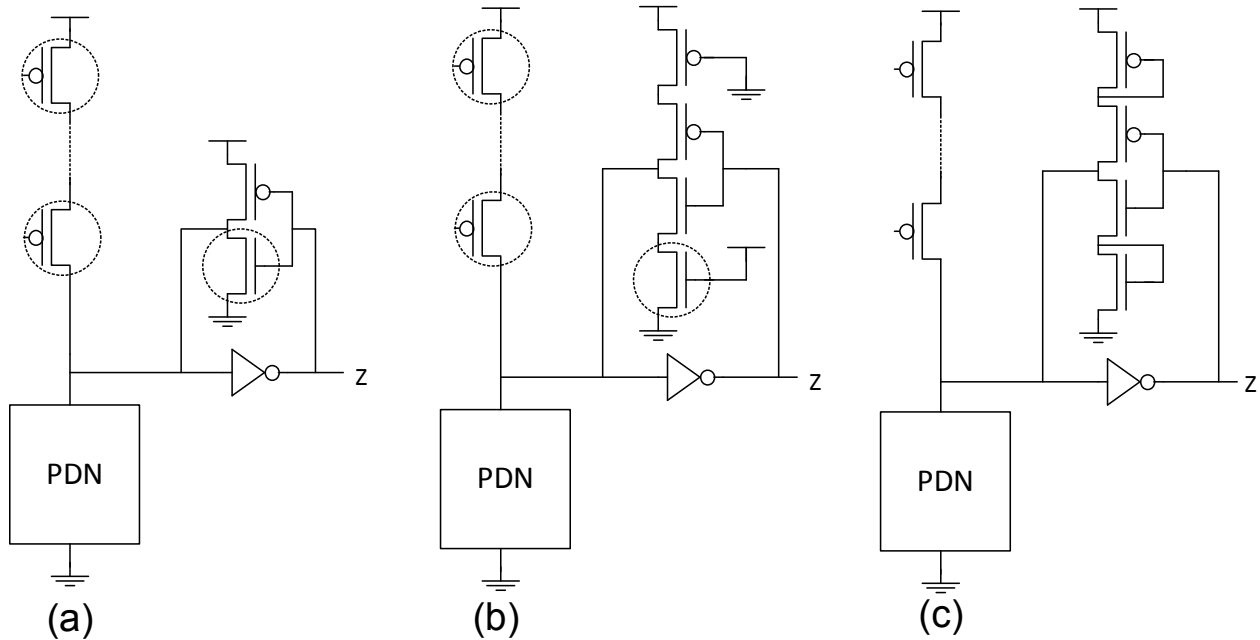


**Figure 10: (a) Structure of NCL semi-static gates (b) TH23 semi-static gate**

discharges the internal node  $Y$  when both set and reset blocks are inactive. This implementation is also more robust to noise and charge sharing effects because the weak feedback inverter, if carefully sized, can restore the value on the internal node  $Y$  in a reasonably short time. The semi-static implementation of a TH23 gate is shown in Figure 10(b).

Appropriate weak feedback inverter sizing is essential for correct operation of a semi-static gate. If a feedback inverter is made very weak, it will not be able to compensate for the leakage current on the internal node, and consequently, the charge on internal node  $Y$  will leak away and the gate output  $Z$  may become invalid or switch value altogether. On the other hand, a feedback inverter that is not weak enough will require a large contention current from the pull-down network (set block) or pull-up network (reset block) to switch the output value, in which case the gate's output may get stuck at a high or low value. The appropriate feedback inverter sizing also determines





**Figure 11: Different feedback inverter weakening methods (a) standard method (b) resistive method (c) proposed diode-connected method**

the performance of the gate. The weaker the feedback inverter, the more similar the semi-static implementation is to the dynamic implementation; therefore, it would be faster and would consume less energy. But, similar to the dynamic implementation, making the feedback inverter very weak makes the gate more vulnerable to noise and charge sharing effects. A more analytical discussion of semi-static C-elements, which are a special case of semi-static NCL gates, can be found in [15].

There are different ways of weakening the feedback inverter. In the standard way, shown in Figure 11(a), usually the length of the NMOS transistor in the feedback inverter is increased. This makes the feedback inverter weak enough to be overpowered by the reset block PMOS transistor chain. The length of the PMOS transistor in the feedback inverter can also be increased or left minimum-sized since the set block pull-down network (PDN) is made of NMOS transistors and, due to the higher mobility of NMOS transistors compared to PMOS transistors, the PDN is usually able to overpower the weak inverter's PMOS transistor. Besides increasing the length of the

feedback inverter's NMOS transistor, sometimes it is better to increase the width of the reset block PMOS transistor chain. The minimum set of transistors that usually need to be sized in a standard semi-static gate is shown with dashed circles in Figure 11(a). In order to save area, sometimes it is better to add series transistors with the feedback inverter [21]. This weakening method, called resistive method hereafter, is shown in Figure 11(b). Here, the added series transistors limit the current available to the feedback inverter, making it weaker. The minimum set of transistors that usually need to be sized is shown with dashed circles. Finally, saving even more area is possible by using diode-connected transistors in series with the feedback inverter, as proposed in Figure 11(c). Using this method, the feedback inverter becomes weak enough even with minimum-sized transistors; therefore, no sizing is usually required. This method also improves the gates in terms of delay and energy consumption significantly as will be shown later in Section 2.6. Again, weakening the feedback inverter makes the gate faster and less energy hungry, but the gate becomes more vulnerable to noise and charge sharing effects, so a trade-off is involved. In practice, optimal sizing of the feedback inverter is not trivial; a more analytic sizing approach is described in [22].

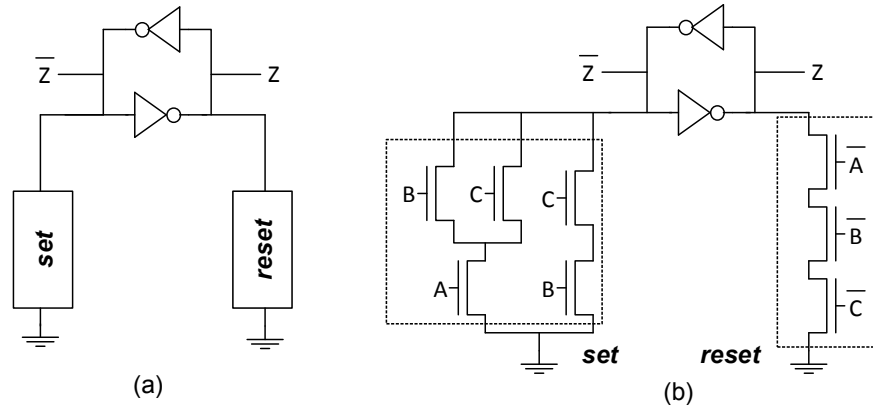
Among the other implementations of the NCL gates (except dynamic implementation), semi-static gates are usually considered to be small (i.e., having minimal number of transistors) and low-energy; however, this image of semi-static NCL gates significantly depends on the weak feedback inverter sizing. The relative sizing requirements for semi-static gates makes this implementation less robust to PVT variations. Also, due to the inherent contention between the set/reset blocks and the weak feedback inverter for switching the output, this implementation is usually slower than the other implementations. This contention can be minimized by appropriate weak

feedback inverter sizing, but it can never be removed. A comparison of various semi-static implementations with the other implementations can be found in [19].

## 2.4 DIFFERENTIAL GATES

The differential implementation of NCL gates [17,21] is most similar to a Differential Cascode Voltage-Switch Logic (DCVSL) implementation of Boolean gates [23], with the exception of using cross-coupled inverters instead of cross-coupled PMOS transistors. A differential NCL gate is shown in Figure 12(a). The major difference between the semi-static implementation of NCL gates and the differential implementation is that the reset block is now connecting output  $Z$  to ground through a pull-down network. Due to this change in the circuit structure, the reset block should use NMOS transistors instead of PMOS transistors, and therefore requires the input complements instead. Since each differential NCL gate provides both output  $Z$  and its complement,  $\bar{Z}$ , no extra logic is necessary to invert inputs. Figure 12(b) shows the differential implementation of a TH23 gate. In a differential NCL gate, asserting an output requires pulling the other output low through a pull-down network (either set or reset block); therefore, before outputs switch value, there is always a short time when both outputs become low. Since in a circuit realized with differential NCL gates, the inputs of each differential gate come from the outputs of other differential gates, this ensures that before a pull-down block becomes active, the other pull-down block becomes inactive first, therefore, no contention between pull-down blocks will ever happen.

Enabling the reset block to use higher-mobility NMOS transistors instead of PMOS transistors improves the differential implementation in several ways. These improvements are mainly because of the reset block being stronger than before so it can switch the state of the cross-coupled

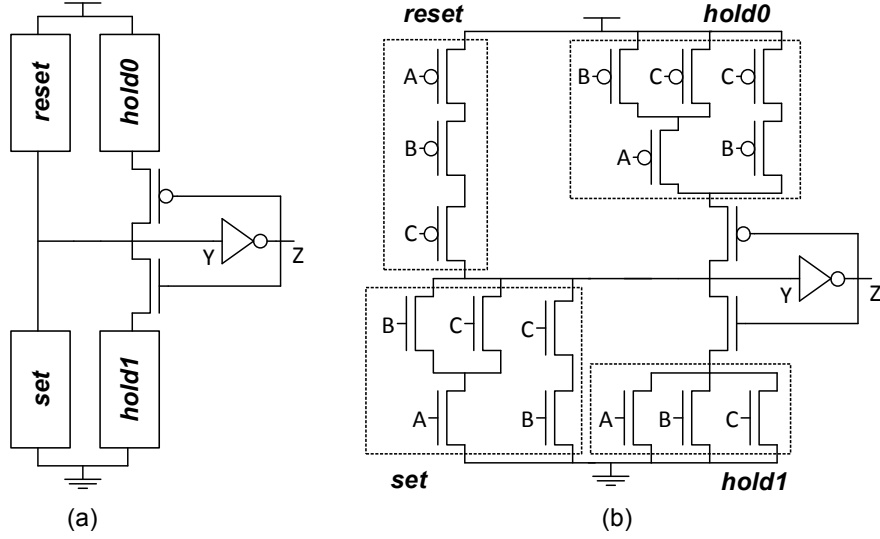


**Figure 12: (a) Structure of NCL differential gates (b) TH23 differential gate**

inverters with less effort. The immediate result being that the differential implementation is usually faster than the semi-static implementation. Also, less aggressive sizing is now required, so the differential implementation is usually smaller than the semi-static implementation. In fact, a differential NCL gate can usually use all minimum-sized transistors and still function correctly. In addition, due to the symmetry of the differential implementation, the cross-coupled inverters are usually sized equally and the whole structure is therefore less sensitive to sizing, and consequently, more robust to PVT variations.

## 2.5 STATIC GATES

All the CMOS NCL gate implementations discussed so far rely on either a parasitic capacitance to maintain state information, such as in the dynamic implementation, or rely on a simple feedback mechanism via an inverter, such as in the semi-static and differential implementations. As discussed, relying on the parasitic capacitance makes NCL gates vulnerable to leakage, noise, and charge sharing problems, and eliminates their delay-insensitivity, while a feedback inverter slows down the gates due to the intrinsic switching contention involved. A static NCL gate imple-



**Figure 13: (a) Structure of NCL static gates (b) TH23 static gate**

mentation removes all these drawbacks, offering faster and more reliable operation.

As depicted in Figure 13(a), static NCL gates are comprised of 4 transistor networks: set, reset, hold1, and hold0. Similar to other implementations, the set block determines the gate's functionality as one of the 27 NCL gates. Once the set function becomes true, the output is asserted. The output then remains asserted through the hold1 block until all inputs are deasserted. The hold1 block is simply made by ORing all inputs together; therefore, it is the same for gates having the same number of inputs. The hold1 function of a static NCL gate with  $n$  inputs can be expressed as:

$$hold1 = I_1 + I_2 + \dots + I_n \quad (6)$$

where  $I_n$  represents input  $n$ . Since both set and hold1 blocks contribute to asserting  $Z$  and maintaining its assertion, the set equation of a static NCL gate can be described as:

$$Z = set + (Z^- \bullet hold1) \quad (7)$$

where  $Z^-$  is the previous output value of the gate and  $Z$  is the new output value. As an example, as depicted in Figure 13(b), the TH23 gate has the following set and hold1 functions:

$$set = A(B + C) + BC$$

$$hold1 = A + B + C$$

In order to implement a static NCL gate in CMOS technology, the complement of  $Z$  is also required. The complement of  $Z$ , denoted as  $Z'$ , is realized with reset and hold0 blocks. The reset block, similar to the previous implementations, consists of all complemented inputs ANDed together. Once all inputs are deasserted, the reset block becomes active and deasserts the output. The output then stays deasserted through the hold0 block until new input values activate the set block to assert the output again. The reset equation of a static NCL gate can therefore be described as:

$$Z' = reset + (Z^- \bullet hold0) \quad (8)$$

It can be proven that the following relations exist between set, reset, hold1, and hold0 functions:

$$set = hold0' \quad (9)$$

$$reset = hold1' \quad (10)$$

Equation (10) can be directly inferred from the definition of reset and hold1 functions and DeMorgan's law; and equation (9) is the logical consequence of the fact that in a static implementation, the pull-up and pull-down networks must be complements of each other to avoid a short-circuit path or a floating node. According to the above equations, the hold0 and reset functions for a static TH23 gate are:

$$hold0 = A' (B' + C') + B'C'$$

$$reset = A'B'C'$$

as shown in Figure 13(b).

In contrast to the semi-static implementation, the static implementation of NCL gates is faster since output switching does not involve contention. It is also very robust to leakage, noise, and charge sharing since for any input combination the internal node  $Y$  is connected to either  $V_{DD}$  through the pull-up network, or GND through the pull-down network. Moreover, the switching threshold of static gates being typically around  $V_{DD}/2$  adds to their noise immunity. Additionally, transistor sizing in a static implementation only impacts its performance, not its functionality; therefore, the static implementation is very robust to PVT variations. Its main drawback is the area overhead from adding hold0 and hold1 blocks. For example, in the case of the TH23 gate, the static implementation shown in Figure 13(b) requires 20 transistors (18 transistors for optimized version to be discussed in section 2.7), while the semi-static and differential implementations only require 12 transistors. A more analytical discussion of static C-elements, that are a special case of static NCL gates, can be found in [15].

## 2.6 COMPARISON OF DIFFERENT GATE STYLES

In order to make a comparison between different implementations of NCL gates, a delay-insensitive  $4 \times 4$  NCL multiplier was designed and exhaustively simulated at the transistor-level using each type of gate. All simulations are performed using the 1.2V IBM CMRF8SF 130nm CMOS process. To make a fair comparison, all the gates utilize minimum-sized transistors except for the first two types of semi-static gates (standard, and resistive) that require sizing for correct operation. These semi-static gates are only sized to the point where they are functional under the supply voltage of 1.2V. Several design parameters including minimum supply voltage, average energy per operation, average delay per operation, average power, area, and noise susceptibility are being compared.

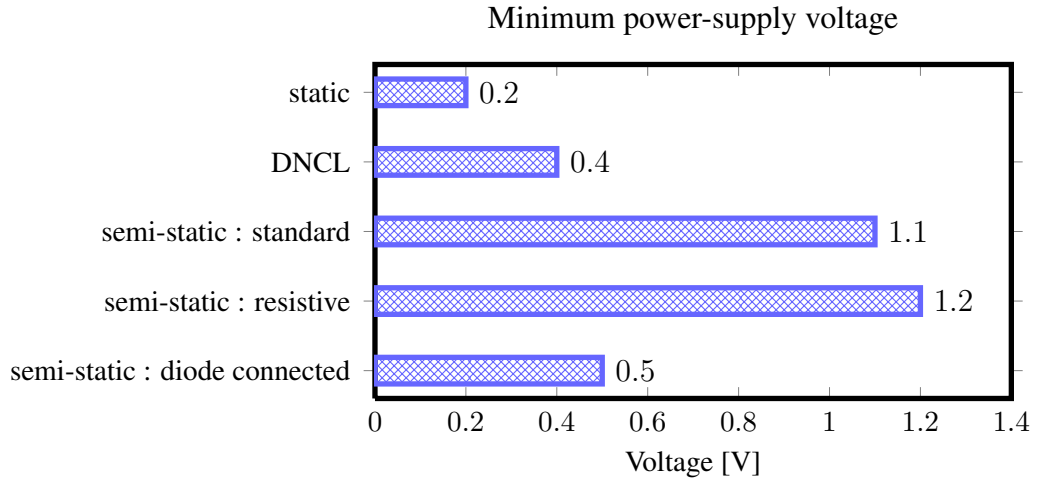
### 2.6.1 MINIMUM POWER-SUPPLY VOLTAGE

The minimum supply voltage under which the NCL multiplier can function correctly is shown in Figure 14 for each gate type. The static multiplier can operate with a supply voltage as low as 0.2V. The multipliers utilizing the first two types of semi-static gates have the highest minimum supply voltage due to the relative transistor sizing requirement for the PUN and the feedback inverter.

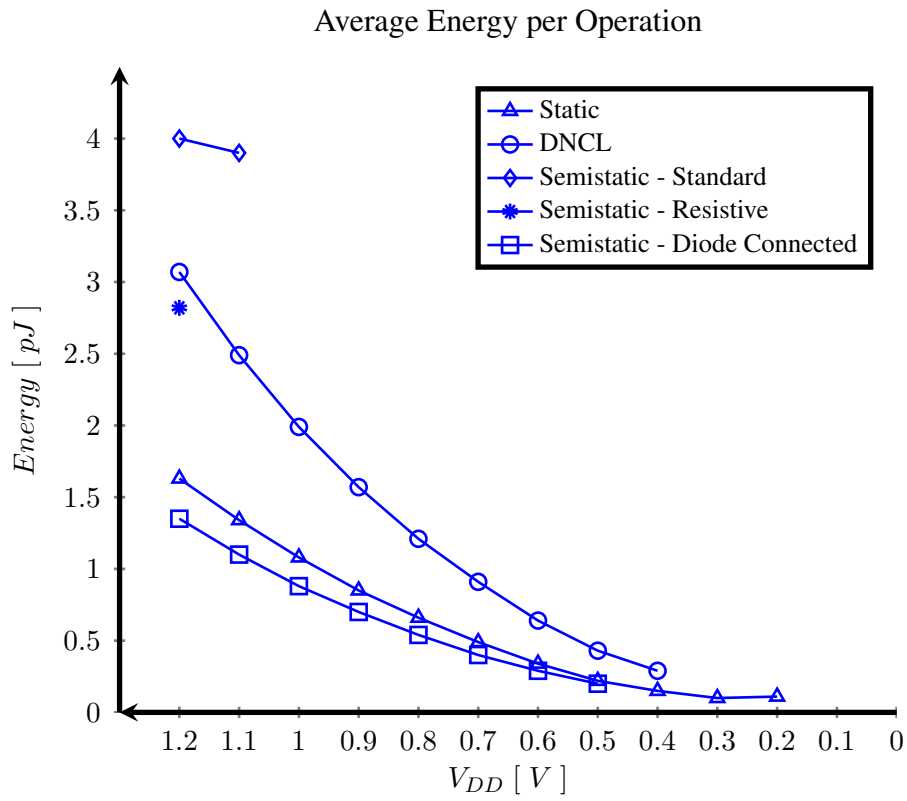
**Table 3: Power, delay, and energy at 1.2v**

Gate Type	Power ( $\mu$ w)	Delay per Operation (ns)	Energy per Operation (pJ)
static	431	3.8	1.63
DNCL	675	4.5	3.07
semi-static ( standard )	655	6	3.95
semi-static ( resistive )	538	5.2	2.82
semi-static ( diode connected )	459	2.9	1.35





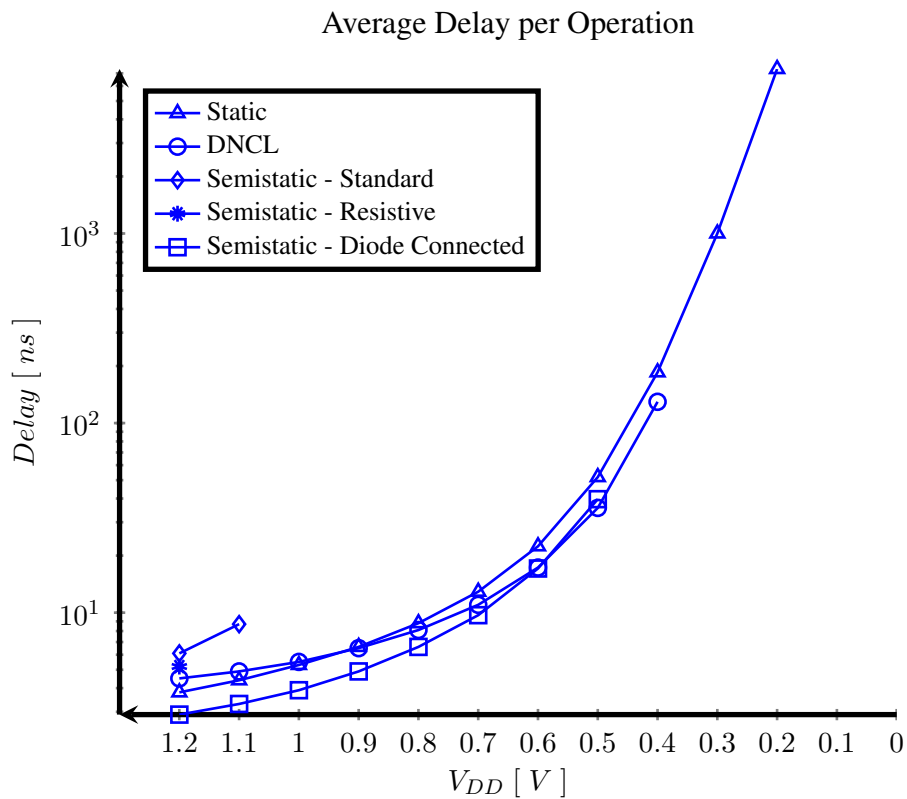
**Figure 14: Minimum power-supply voltage**



**Figure 15: Average energy per operation**

## 2.6.2 AVERAGE ENERGY PER OPERATION

Figure 15 shows the average energy per operation for different realizations versus different supply voltages. The multiplier realized with the diode-connected semi-static gates consumes the least energy per operation. On the other hand, the standard semi-static realization has the worst energy per operation. The energy per operation for each realization decreases as the supply voltage decreases. Under the supply voltage of 1.2V, the average energy per operation for different realizations is listed in Table 3. Based on this table, the multipliers that utilize the static or diode-connected semi-static gates consume less energy than the other realizations.



**Figure 16: Average delay per operation**

### **2.6.3 AVERAGE DELAY PER OPERATION**

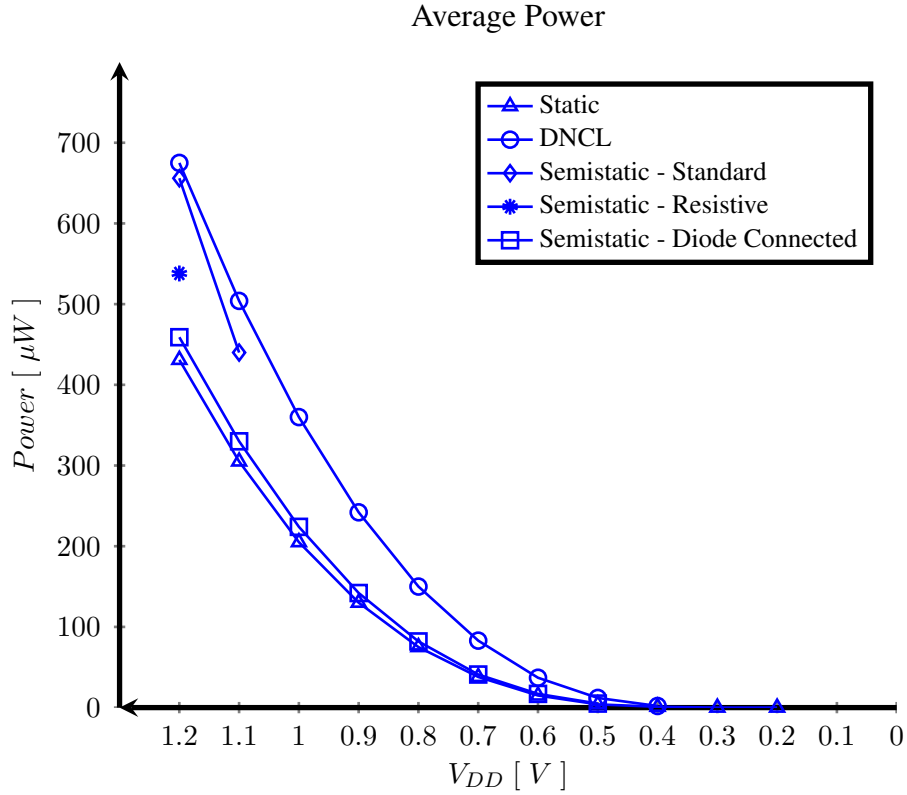
The average delay per operation for different realizations under the supply voltage of 1.2V is listed in Table 3. Based on this table, the diode-connected semi-static and the standard semi-static are the fastest and the slowest realizations, respectively. The average delay per operation versus different supply voltages is shown in Figure 16. This shows an exponential increase in the delay as the supply voltage decreases. Also, the DNCL realization is the fastest realization for supply voltages lower than 0.6v.

### **2.6.4 AVERAGE POWER**

Average power consumption for different realizations versus supply voltage is shown in Figure 17, where power consumption decreases as supply voltage decreases. Based on simulation results, the static and DNCL realizations are the least and most power consuming, respectively. Under the supply voltage of 1.2V, the average power consumption for different realizations is listed in Table 3.

### **2.6.5 AREA**

The total area for different realizations is shown in Figure 18. Total area is calculated by adding up the area of transistors used in each realization. As it is shown, DNCL and the standard semi-static realizations have the smallest and largest area, respectively. This could be predicted by considering the fact that DNCL gates use the least number of transistors and they are all minimum-sized. The diode-connected semi-static gates also use all minimum-sized transistors but they re-

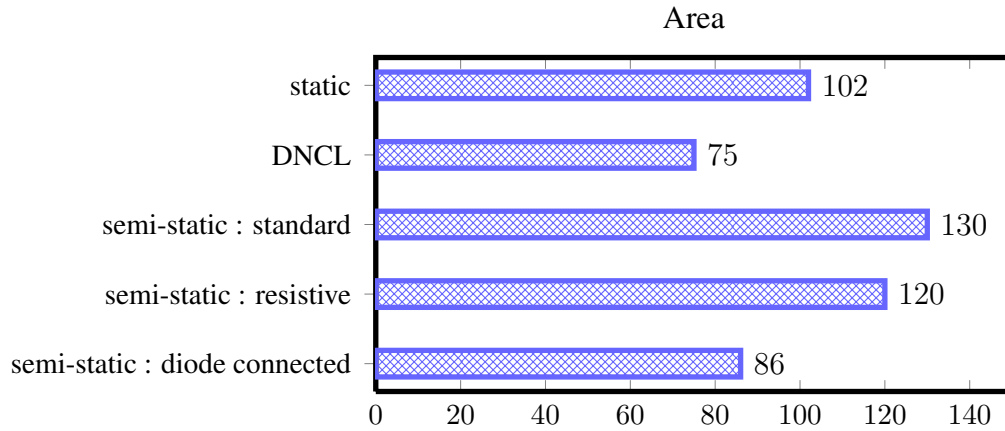


**Figure 17: Average power**

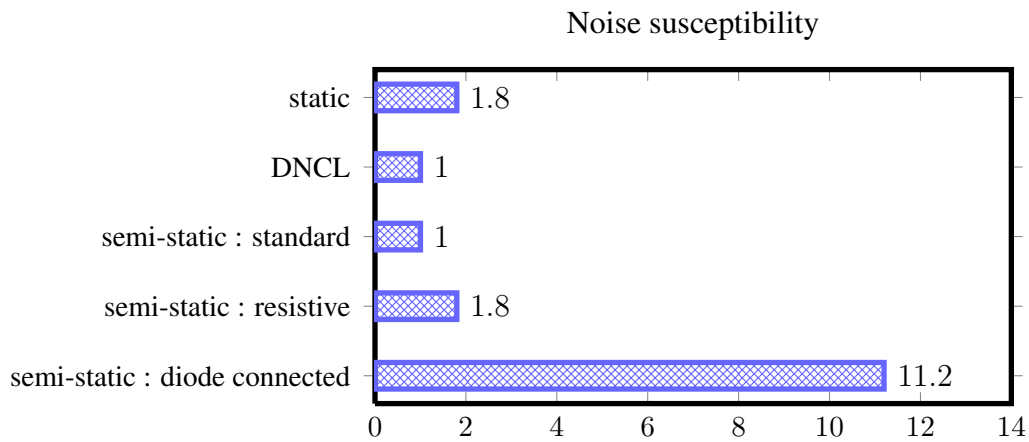
quire two additional transistors compared to DNCL gates to weaken the feedback inverter, thus they consume slightly more area (15%). The static realization was expected to utilize more area since the static gates have extra pull-up and pull-down networks to maintain hysteresis. The other semi-static realization were also expected to require more area due to transistor sizing requirement.

### 2.6.6 NOISE SUSCEPTIBILITY

The last parameter measured is noise susceptibility, motivated by the fact that in semi-static gates the feedback inverter should not be too weak such that it can resist the noise currents on the internal node. To test robustness to noise a typical gate (TH22) was selected and simulated for noise susceptibility. The testbench is comprised of a TH22 gate and a noise current source connected



**Figure 18: Area**



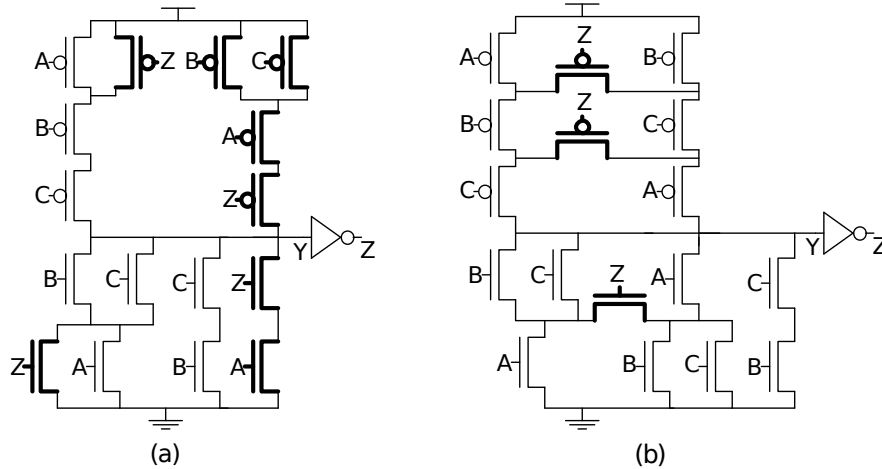
**Figure 19: Noise susceptibility**

to its internal node. The magnitude of the noise current is then swept to the point where the gate output switches. Simulation results show that the minimum current value for which the output switches for different gate types are as follows: static  $10.8\mu\text{A}$ , standard semi-static  $19.3\mu\text{A}$ , diode-connected semi-static  $1.7\mu\text{A}$ , DNCL  $19.3\mu\text{A}$ , resistive semi-static  $10.7\mu\text{A}$ , and supply feedback semi-static  $6.2\mu\text{A}$ . Based on simulation results the standard semi-static and the DNCL are the most noise robust implementations. Assuming the noise susceptibility of these two implementations is 1x, the noise susceptibility for the rest of the implementations is shown in Figure 19.

## 2.6.7 DISCUSSION

According to the simulation results, NCL circuits that utilize different implementations of NCL gates can excel in different design parameters. The static realization of NCL circuits can utilize very low supply voltages. Under these supply voltages, NCL circuits can be very low-energy/power but at the same time be much slower. Therefore, static realizations could be used in applications where low voltage/energy/power are desired but speed and area are not very critical. The DNCL realization utilizes less area and can operate under low-voltage conditions with notably high speed but consumes more energy and power.

The only advantage of the standard semi-static realization is its noise susceptibility, which is the same as DNCL; hence even when noise is an issue, DNCL would be preferred. The resistive semi-static realization has very close features to the standard semi-static except for lower power/energy consumption. The performance of the first two types of semi-static realizations is highly dependent to the sizing of the gates. Here, for the sake of a fair comparison, these semi-static gates were only sized to the point where they function correctly, but, one can achieve better performance by resizing them. Finally, the proposed diode-connected semi-static realization, although relatively weak in terms of noise susceptibility, seems to offer an excellent trade-off between the other design parameters; it requires the second least area, over DNCL, utilizes only slightly more power than the least power consuming static design, can operate under moderately low supply voltage conditions, and is the fastest realization under nominal supply voltage, while consuming the least amount of energy.



**Figure 20: (a) Original TH23 static gate (b) Proposed TH23 static gate**

## 2.7 NEW STATIC GATES

### 2.7.1 DESIGNING NEW STATIC GATES

In the previous section, area overhead was mentioned to be the main drawback of static NCL gates; however, sometimes it is possible to share transistors between each pair of pull-up (reset and hold0) or pull-down (set and hold1) networks to reduce area. For example, the direct static implementation of the TH23 gate, shown in Figure 13(b), consists of 20 transistors; but after sharing transistors, the optimized implementation only requires 18 transistors, as shown in Figure 20(a). There are two types of transistors in a static NCL gate: switchers, which contribute to switching the gate's output, and keepers, which only contribute to retaining the gate's state when neither set nor reset blocks are active. In Figure 20(a) the keepers are shown in boldface.

The development of the new static NCL gates is inspired by the observation that in a traditional static NCL gate, the hold0 and hold1 transistor networks are only used for retaining the gate's state when neither set nor reset functions are true. In other words, the hold0 and hold1 tran-

sistor networks only contribute to holding the output state but not switching it. The idea behind the new static NCL gates is to integrate the set and hold1 transistor networks as well as the reset and hold0 transistor networks into a single composite transistor network such that it involves more transistors in output switching. Figure 20(b) shows the application of this idea to the TH23 gate. The new gate structure differs from the original one in two ways. First, the reset network has been duplicated and rearranged, and then some extra PMOS transistors are added to realize the hold0 function by connecting appropriate nodes of the two PMOS transistor chains. Second, the hold1 function is realized by duplicating and flipping a portion of the set network and then connecting the middle nodes with an NMOS transistor. The new gate consists of 19 transistors, which is one transistor more than the original one; however, compared to the original gate, the number of keepers has been reduced from 8 to 3 (shown in boldface), while the number of switchers has increased from 8 to 14, resulting in faster switching compared to the original gate.

The correctness of the new gate structure can be easily proved using Boolean algebra. For the pull-up network, when  $Z = 1$  both PMOS keepers are off so the function of the pull-up network can be expressed as:

$$A'B'C' + B'C'A' = A'B'C' \quad (11)$$

which is the same as the function of the reset block, and when  $Z = 0$  both PMOS keepers turn on so the function of the pull-up network can be expressed as:

$$(A' + B')(B' + C')(C' + A') = A'(B' + C') + B'C' \quad (12)$$



which is the same as the function of the hold0 block. Similarly, for the pull-down network, when  $Z = 0$  the NMOS keeper is off so the function of the pull-down network can be expressed as:

$$(B + C)A + A(B + C) + BC = A(B + C) + BC \quad (13)$$

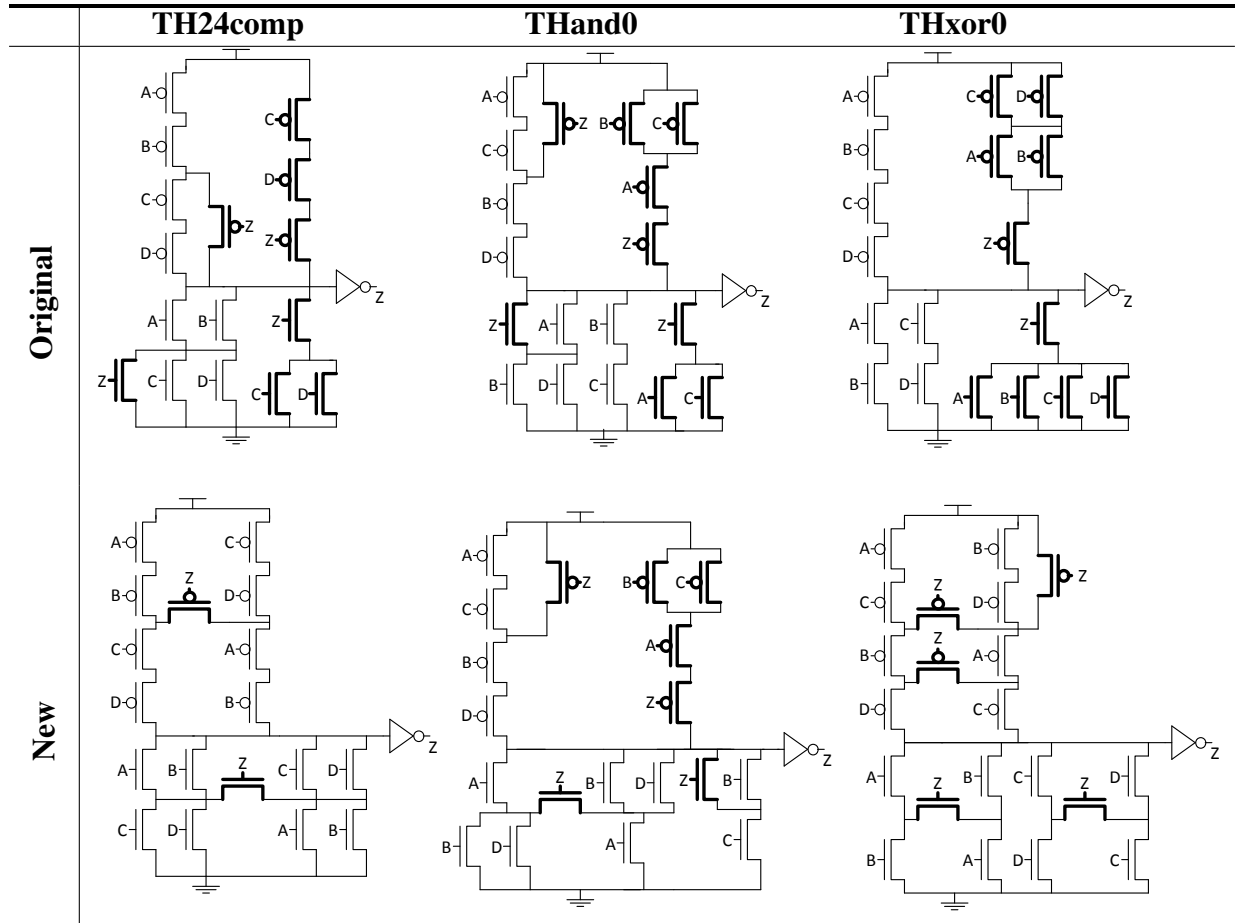
which is the same as the function of the set block, and when  $Z = 1$  the NMOS keeper turns on so the function of the pull-down network can be expressed as:

$$(B + C + A)(A + B + C) + BC = A + B + C \quad (14)$$

which is the same as the function of the hold1 block.

The new gate structure also speeds-up output switching in one additional way. Careful investigation shows that the number of transistors in a series chain for holding the gate's state when neither set nor reset functions are true has increased. For example, the hold0 path that was originally going through  $Z \rightarrow B \rightarrow C$  is now going through  $B \rightarrow Z \rightarrow B \rightarrow C$ , which is one transistor longer than the original path. Similarly, the hold1 path that was originally going through  $B \rightarrow Z$  is now going through  $B \rightarrow Z \rightarrow B$ . Hence, the new gate structure's transistor chain length for hold0 and hold1 paths has increased by one transistor. This is equivalent to weaker hold0 and hold1 networks (i.e., the paths have higher resistance); therefore, the set and reset networks can switch the gate's output faster. This might look confusing since, as mentioned before, the set and hold0 (and similarly reset and hold1) networks are complements of each other such that they are never asserted simultaneously; therefore, the set network never needs to overpower the hold0 network (or reset network overpower hold1). However, since at the time of switching there is a short moment

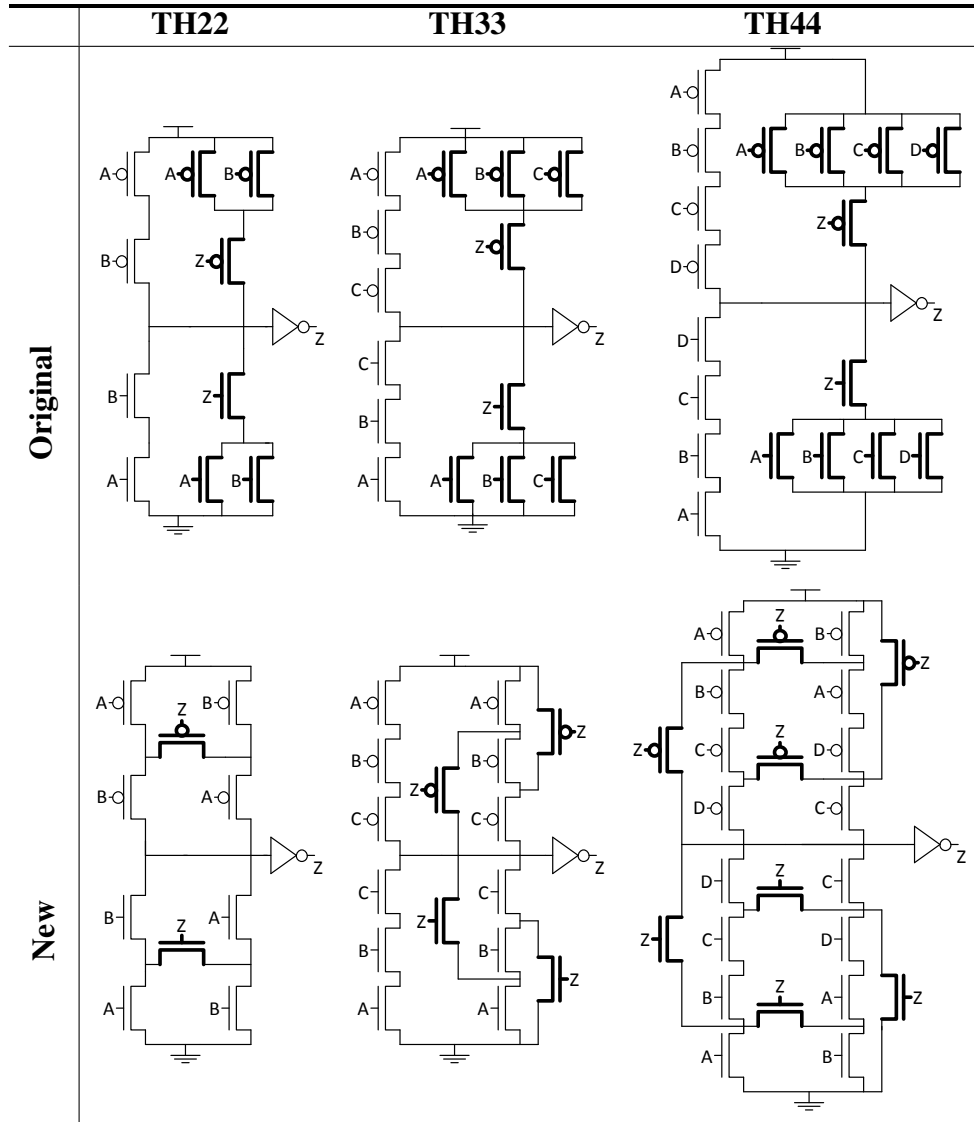
**Table 4: Original complex static gates versus the new versions**



when both pull-up and pull-down networks turn on and create a short-circuit path from  $V_{DD}$  to GND (similar to static Boolean gates), a pull-up (pull-down) network with higher resistance, and consequently less current flow, helps the pull-down (pull-up) network pull the internal node to GND ( $V_{DD}$ ) with less effort, resulting in faster switching. The last interesting feature of the new static gate structure is that it is more symmetric than the original structure, resulting in closer output rise/fall times.

Converting traditional static gates to the new ones is not always easy and straightforward, especially for more complex gates. Additionally, although in the case of the TH23 gate there was only one transistor overhead for the new gate style, sometimes area overhead is more than a few

**Table 5: Original C-elements versus the new versions**



transistors, resulting in an area versus delay tradeoff. Based on how complex the gate is, sometimes it is possible to partially apply this technique (e.g., to only the pull-up or pull-down network, or even just a portion of them). Table 4 shows the design of a few complex NCL gates using both the original and the new method, with keeper transistors shown in boldface. The first row of gates pertains to the original design, while the second row shows the new designs. Comparing the new versions with the original ones shows that the number of keepers has been reduced in all the new

**Table 6: Comparison between original and new static gate styles**

Gate	TPLH [ps]			TPHL [ps]			Energy [fJ]			Transistors		
	New	Original	Improve	New	Original	Improve	New	Original	Improve	New	Original	Overhead
TH22	155	168	7.70%	83	123	32.20%	18.4	18.5	0.60%	12	12	0
TH33	174	197	11.40%	128	193	33.90%	20.2	19.4	-4.50%	18	16	2
TH44	198	226	12.00%	183	262	30.20%	23.1	20.2	-14.30%	26	20	6
TH44w2	200	214	6.40%	179	198	9.70%	22.3	20.6	-7.80%	25	22	3
TH23	172	180	4.50%	115	207	44.30%	20	20.3	1.40%	19	18	1
TH34w2	191	194	1.70%	150	222	32.10%	21.6	20.3	-6.30%	27	22	5
TH24comp	160	188	14.80%	134	217	38.20%	19.8	20.4	3.00%	20	18	2
THxor0	167	189	12.00%	142	255	44.20%	20.4	20.7	1.80%	23	20	3
TH22n	167	189	11.50%	86	138	37.80%	18.7	18.9	1.30%	16	16	0
THand0	180	195	7.50%	195	252	22.80%	20.6	21.2	2.80%	21	20	1
Average	177	194	9.00%	139	207	32.50%	20.5	20.1	-2.20%	20.7	18.4	2.3

versions. For the THand0 gate, the pull-up network could not be converted to utilize fewer keepers, so it is not changed. Table 5 compares the original and the new static C-elements. For the TH22 gate, the new version is equivalent to the symmetric C-element design in [12]. As mentioned before, converting the original static design to the new one is not always easy and does not follow strict rules. However, the following guidelines are helpful:

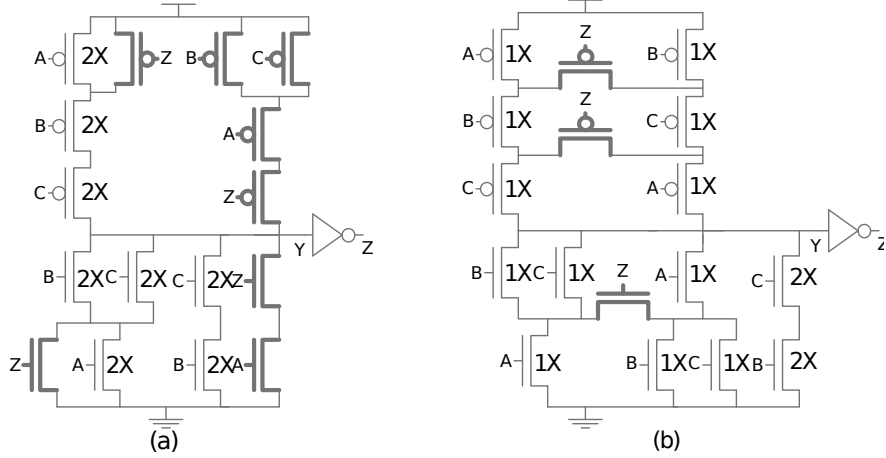
- Remove the hold1/hold0 networks from the original design
- Duplicate the set/reset networks
- Rearrange/flip the duplicated networks and connect their internal nodes to the original network by adding keepers such that the hold1/hold0 functionality is ensured
- If the new structure requires more keepers in the pull-up or pull-down networks then try to apply this technique partially or just use the original design

Table 6 shows a comparison between the new gates with the original ones in terms of delay, area, and energy consumption. The gates are implemented and simulated using the IBM CMOS9SF 90nm CMOS process. All simulations are performed under the following conditions: typical process corner, nominal power supply voltage of 1.2 V, temperature of 27 °C, and capacitive

load of 10 fF. Both high-to-low (TPHL) and low-to-high (TPLH) propagation delays are included in this table. The simulation results show that on average the new gates offer 9% improvement in TPLH and 32.5% improvement in TPHL, with a 2.2% increase in energy consumption and an average of 2.3 additional transistors per gate. For the results in Table 6, all transistors are minimum-sized and the results are averaged over all possible input combinations.

## 2.7.2 SIZING NEW STATIC GATES

The new static gate design speeds up switching with a reasonable area overhead (2.3 transistors per gate). However, the new static gates have the potential to be smaller than the original static gates when both gate styles are properly sized for faster switching. For example, assume that the TH23 gates in Figure 20 need to be sized. Since only the switchers are responsible for output switching, one can double their width while allowing the keepers to stay minimum-sized. This is shown in Figure 21. The keepers are all minimum-sized (1X) in this figure, so their size is not shown. The size of the switchers in the original static gate, however, has doubled, even for the parallel switchers, in order to account for when only one of them contributes to output switching. The switchers in the new static gate are then sized such that they provide the same pull-up/pull-down resistance as the original static gate on the switching paths. Finally, the output inverter for each gate can be sized such that it offers a balanced output rise/fall time targeting a certain output load. Assuming that the output inverters would have almost similar (or comparable) sizes, the new static gate would be smaller than the original one, shown by adding up the size of transistors for each gate. In the case of the TH23 gate, the original gate size is 24X while the new gate size is 19X.



**Figure 21: A sizing example for (a) original (b) new static TH23 gates**

**Table 7: Comparison of NCL multipliers realized with different static gate styles**

Gate Style	Original Minimum	New Minimum	Original Sized	New Sized
Delay per operation [ns]	1.45	1.05	1.29	0.98
Energy per operation [pJ]	1.29	1.28	2.62	2.34
Area [ $\mu\text{m}^2$ ]	59.4	62.6	122.2	111.3
Minimum VDD [V]	0.25	0.26	0.22	0.27

### 2.7.3 EXPERIMENTAL RESULTS

In order to measure the performance of the new static gate style at the circuit level and compare it to the original static gate style, a delay-insensitive NCL  $4 \times 4$  pipelined multiplier [24] was simulated at the transistor level using each gate style. The results, averaged over all 256 input combinations, are shown in Table 7. All simulations are performed under the following conditions: typical process corner, nominal power supply voltage of 1.2 V, and temperature of 27 °C. In order to measure the minimum power supply voltage for each variation of multiplier,  $V_{DD}$  is dropped to the point where the NCL multiplier outputs wrong data or completely stalls due to deadlock [2].

For the minimum-sized gates, the multiplier using the new gate style is 27% faster and requires 5% more area, with approximately the same energy per operation and the same low-

voltage operation capability. Table 7 also shows the comparison between the multipliers utilizing sized static gates. The multiplier realized with the new sized gates is now both faster (24%) and smaller (8%) than the multiplier using the original sized gates. In addition, the energy per operation is now 10% lower, but the minimum power supply voltage has increased by 22%.

## **2.8 CONCLUSION**

In this chapter, different CMOS implementations of NCL gates were introduced and their trade-offs were discussed in detail. It was shown that each implementation has its own advantages and drawbacks for designing NCL circuits. Additionally, new semi-static and static gate designs were proposed. The new semi-static gate style was shown to significantly improve the NCL circuit performance in many ways. Also the new static gate style was compared to the original static style in terms of delay, energy, and area consumption, showing that the new gate style is significantly faster, while requiring slightly more area and energy for minimum sized gates. After sizing the gates, it was shown that the new static gate style is faster, and requires less area and energy. These conclusions are all supported by transistor-level simulation of a delay-insensitive NCL pipelined multiplier implemented with different gate styles.

### **3 NCL TECHNOLOGY MAPPING**

©2014 IEEE. Reprinted, with permission, from F. A. Parsan, W. K. Al-Assadi, and S. C. Smith, “Gate mapping automation for asynchronous NULL convention logic circuits,” *Very Large Scale Integration (VLSI) Systems*, IEEE Transactions on, vol. 22, no. 1, pp. 99–112, 2014.

In reference to IEEE copyrighted material which is used with permission in this dissertation, the IEEE does not endorse any of University of Arkansas’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to [http://www.ieee.org/publications\\_standards/publications/rights/rights\\_link.html](http://www.ieee.org/publications_standards/publications/rights/rights_link.html) to learn how to obtain a License from RightsLink.

#### **3.1 INTRODUCTION**

Asynchronous logic design paradigms such as NULL Convention Logic (NCL) [2–5] offer many advantages over the conventional synchronous designs. These advantages include inherent robustness against power-supply, temperature, and process variations, less noise/EMI, and easy design reuse. NCL circuits are correct-by-construction [3] requiring very little timing analysis, if any. In today’s nanometer processes where meeting timing closure is becoming more and more difficult due to increasing clock rates and decreasing feature sizes this quality can be very attractive. NCL is also one of the few asynchronous design paradigms that has been used for a number of industrial designs [4, 6].

The main obstacle preventing the widespread application of asynchronous design paradigms,



such as NCL, in industry is the lack of standard CAD tools that support automating the design process. Several NCL design automation flows have been proposed in the literature so far [4, 7–9]. Technology mapping, in particular, is the heart of such design flows since the performance of the mapped circuits is directly determined by the efficiency of technology mapping. In this chapter, the problem of mapping multi-rail logic expressions to an NCL gate library, which is a form of technology mapping for a certain class of NCL design flows is being addressed. Mapping multi-rail logic expressions to NCL gates is also an essential part of a custom NCL design flow, and in contrast to mapping a Boolean logic expression to a limited set of basic Boolean gates, an efficient manual mapping of multi-rail logic expressions to 27 NCL gates is not trivial, especially when the logic expressions contain many product terms (explained later). In this chapter, a multi-rail logic expression mapping algorithm is proposed and compared to the only other existing multi-rail logic expression mapping algorithm in the literature; and it is shown that the proposed algorithm can outperform the existing one in several ways.

The rest of the chapter is organized as follows: Existing NCL design automation flows are first discussed in Section 3.2 and then some conventions are established for the rest of the chapter in Section 3.3. In Section 3.4, the original mapping algorithm is explained in detail; and its constraints are then discussed in Section 3.5. Section 3.6 presents our proposed mapping algorithm; and its improvements over the original algorithm are discussed in Section 3.7. In Section 3.8, the experimental results for each mapping algorithm are presented by running the developed Perl scripts on some typical multi-rail logic expressions and circuit components. Finally, the conclusion is presented in Section 3.9.

### 3.2 EXISTING NCL DESIGN AUTOMATION FLOWS

Several NCL design automation flows have been proposed in the literature [4, 7–9]. Any NCL design flow must address two main issues: synthesizing a synchronous register-transfer level (RTL) design into an NCL netlist without violating input-completeness and observability, and optimizing the synthesized circuit under a predefined cost function. All existing NCL design automation flows can convert a synchronous RTL design into an input-complete and observable NCL netlist; however, they differ in their technology mapping approach and optimization level.

The design flow in [4] starts with a synchronous RTL design written in VHDL. The RTL design must explicitly specify the location of registers, either through inference or instantiation. These registers are later replaced with NCL registers and the required handshaking signals and completion detection components are added. The rest of the RTL design that describes the combinational blocks can be a synthesizable behavioral description. The RTL design can be functionally verified by simulating it in any VHDL simulator using a single-rail multi-valued signal type (NCL\_LOGIC) that includes NULL ('N' value). After verification, the combinational logic is synthesized into a so-called “3NCL” gate-level netlist comprised of only 2-input Boolean gates using Synopsys Design Compiler. At the time of synthesis, Design Compiler treats NULL ('N' value) as a *don't-care* so it does not affect the synthesized circuit. Next, the single-rail multi-valued signals in the 3NCL netlist are converted to dual-rail signals, and each 2-input Boolean gate is macro-expanded to its DIMS-style [25] equivalent. The resultant circuit is now called a “2NCL” circuit comprised of only 2-input C-elements (i.e. TH22) [15] and OR gates. A 2NCL circuit built this way is proved to be input-complete and observable by design [26, 27], but it is not optimized. The

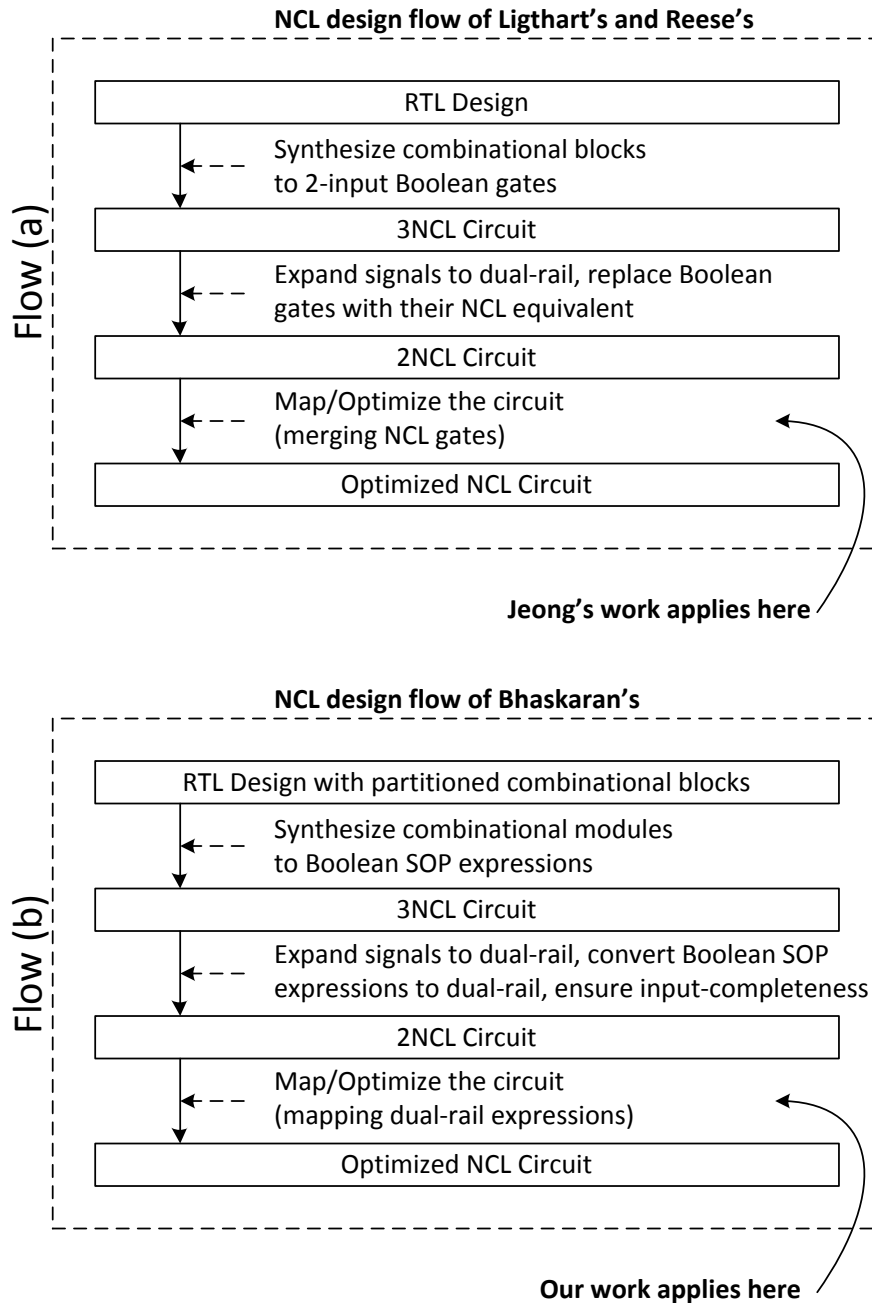
2NCL circuit is then optimized by replacing each 2-input C-element with its Boolean set phase equivalent (i.e., a 2-input AND gate); and a multi-level logic minimization is performed by using Design Compiler again. The resulting so-called smoothed and optimized netlist is finally mapped to NCL complex gates by running Design Compiler for the last time. This NCL design flow heavily depends on the sophisticated synchronous synthesis tools (Design Compiler, here) and only targets area reduction for optimization.

Reference [7] introduces systematic and general technology mapping and cell merger methods that can be used for designing all types of timing-robust asynchronous threshold networks, including NCL circuits. These methods are able to optimize both the datapath and control portions of NCL circuits, and they can target either area or delay or a combination of the two as the cost function. However, these methods are not a comprehensive NCL design flow (i.e., starting from an RTL description to an NCL netlist), but post-processing optimization steps that can add to other NCL design flows, such as the one in [4], to further optimize area or delay of the final circuit. In other words, the input to these methods is an already synthesized and working NCL circuit that has not been optimized yet.

The NCL design automation flow in [8] is very different than other existing NCL design flows in that it relies much less on the synchronous synthesis tools. This design flow is loosely based on the TCR method explained in [28]. Similar to the design flow in [4], it starts with a synchronous RTL design in which registers are explicitly defined; but in contrast, the designer also has to partition the combinational blocks into combinational modules with appropriate sizes. These combinational modules can be described behaviorally but must be connected in a structural manner

to form the entire combinational block. The combinational modules are initially synthesized into Boolean sum-of-products (SOP) expressions, such that each output of a combinational module is described in terms of its inputs. This step can be performed with just about any synthesis tool. The resultant Boolean SOP expressions are then converted to dual-rail SOP expressions and minimized using custom scripts. Finally, a minimum number of the minimized dual-rail SOP expressions are selected to become input-complete (according to the weak conditions of Seitz [11]) and the final minimized and input-complete SOP expressions are mapped to NCL gates by using a custom mapping algorithm, to be discussed in detail later.

Finally, Unified NCL Environment (UNCLE) [9] is a very recent interesting work on automating the NCL design flow. Conceptually, this flow is similar to the one in [4] but with more flexibility. The design flow starts with Verilog RTL code, which is initially synthesized to a library of Boolean gates, such as AND2, OR2, XOR2, inverter, D-flip-flop, D-latch, and other gates that are either black boxes for special use, or are complex gates such as Full Adder and MUX2, which have a direct, optimized NCL implementation. Synopsys Design Compiler or Cadence Encounter RTL Compiler are used in this step. In contrast to [4], here, the designer does not have to specify explicitly the location of registers in the RTL design. Next, the signals are converted to dual-rail, and all Boolean gates are replaced with their NCL implementation. The handshaking signals for the dual-rail registers are then generated and connected to make a functional NCL circuit. The rest of the UNCLE flow is related to optimizing the circuit by performing some optional optimization steps such as latch balancing, relaxation [29, 30], cell merging, and net buffering. An extension of the UNCLE design flow allows for a control-driven design style (Balsa style [31, 32]) versus the



**Figure 22: Automated NCL design flows**

data-driven style, which can potentially result in a smaller and more energy efficient NCL circuit.

The current NCL design flows can be roughly divided into two main categories. These categories are shown in Figure 22. The steps shown in both categories are simplified to show the essence of each work. Both design flows start with an RTL design, however, the RTL design in

flow (b) requires the combinational blocks to be partitioned into smaller modules (usually having at most 4 inputs and any number of outputs). This extra constraint is equivalent to the constraint of synthesizing combinational blocks to only 2-input Boolean gates in flow (a). In fact, both of these constraints are imposed by the fan-in restrictions of the standard NCL gate library. If these constraints are not set, the synthesized circuit may contain gates (flow (a)) or product terms (flow (b)) that cannot be mapped to NCL gates without breaking them into smaller gates or smaller product terms. In contrast to synchronous circuits, breaking gates or product terms to smaller pieces is not trivial in NCL circuits, since this can potentially introduce gate orphans [7]. Moreover, breaking the large gates or product terms to smaller pieces is usually more expensive in terms of area and delay compared to avoiding them in the first place. For these reasons, all the current NCL design flows (including the algorithm proposed here) assume that the synthesized circuits do not contain large gates or large product terms by setting appropriate constraints on the circuit before or during synthesis.

As depicted in Figure 22, both categories require the initial RTL design to be synthesized. However, in flow (a) the combinational blocks are synthesized into 2-input Boolean gates, while in flow (b) each combinational module is synthesized into Boolean SOP expressions describing its outputs in terms of its inputs. In other words, the 3NCL circuit result from each flow is different since one contains Boolean gates (flow (a)) and the other contains Boolean SOP expressions (flow (b)). As explained previously, although both flows use synthesis tools, flow (a) requires more sophisticated synthesis tools for synthesis and later for mapping and optimization. The next step in both flows is to expand the single-rail signals to dual-rail signals and replace the Boolean gates

with their NCL equivalent (flow (a)) or convert the Boolean SOP expressions to dual-rail SOP expressions (flow (b)). The resultant 2NCL circuits are again different because one contains NCL gates while the other contains dual-rail SOP expressions.

The 2NCL circuits then need to be optimized and mapped before a robust optimized NCL circuit is achieved. Each one of the previous works has developed its own optimization and mapping techniques. Reference [4] initially uses Design Compiler for a constrained minimization of the 2NCL circuit, and then uses a template-based cell merger method to further optimize the circuit. Reference [9] also allows merging adjacent gates with no fan-out to more complex gates to save area. Reference [7], however, offers more systematic and general cell merging and technology mapping techniques, which can be added to flow (a) to further optimize the final circuit. Similarly, reference [8] has its own grouping (to be explained later) and mapping algorithm, but as mentioned before, an alternative algorithm that can result in a more optimized circuit is proposed in this chapter. Figure 22 also shows where reference [7] and our proposed algorithm fit in the existing NCL design flows. Although reference [7] and our proposed algorithm are only used in the last step of the NCL design flow, but it is the most important step, since it determines the efficiency of the final NCL circuit.

Each one of the two NCL design flow categories has its own merits and drawbacks. The first advantage of flow (b) is its minimum reliance on the expensive sophisticated synthesis CAD tools, while flow (a) heavily uses them. Although reusing the existing rich set of synchronous CAD tools for designing NCL circuits is beneficial, enabling the NCL design flow to use simpler and cheaper CAD tools makes it more affordable.

**Table 8: Comparison of the two NCL design flow categories**

<i>Flow (a)</i>	<i>Flow (b)</i>
Requires more sophisticated synthesis tools	Can work with cheaper and simpler synthesis tools
Provides less control over the final circuit structure	Provides more control over the final circuit structure
Debugging the final circuit is more difficult	Debugging the final circuit is easier
Has area overhead due to not considering the weak conditions of Seitz	Takes advantage of the weak conditions of Seitz to reduce area
Suitable for less custom designs	Suitable for more custom designs

Another advantage of flow (b) is that it allows the designer to have more control over the structure of the final circuit. This control comes by explicitly defining the combinational modules through partitioning the combinational blocks. On the other hand, in flow (a) the whole combinational block is synthesized into 2-input Boolean gates; moreover, the subsequent optimization and mapping steps can potentially remove some of the circuit nodes. Consequently, the resultant circuit will have a less informative structure, which makes the debugging process harder if any problem is encountered later in the design phase. Although partitioning the design to smaller modules can be considered an advantage, it puts more burden on the designer, making flow (b) more appropriate for custom designs.

Finally, in contrast to flow (a), flow (b) takes advantage of the weak conditions of Seitz by only making a minimum number of outputs of the combinational modules input-complete, which leads to a smaller (and potentially faster) circuit. However, in flow (a), all the 2-input Boolean gates are expanded to their input-complete NCL equivalents without exception, which significantly increases area. The final area and speed of the synthesized circuits is design-dependent and can be better or worse for each flow. A summary of the advantages and disadvantages of each flow category is shown in Table 8. Additionally, several circuits were designed using both flows, and



the results are compared in Section 3.8.

This chapter addresses the problem of mapping multi-rail logic expressions to an NCL gate library. It particularly focuses on the only other existing multi-rail logic expression mapping algorithm in the literature, which is introduced in [8] and will be called the original algorithm hereafter. The original algorithm will be discussed in detail, and a new mapping algorithm will be proposed that can further reduce the area and delay of the mapped NCL circuits. In contrast to the original algorithm, the new mapping algorithm can target different cost functions or use a subset of NCL gates for mapping. The proposed mapping algorithm is not only helpful for being used as a part of the NCL design automation flow, but also can be used as a standalone tool in assisting the custom NCL design when the derived multi-rail logic expressions become so large that finding the optimal mapping for them becomes difficult. In summary, this chapter makes the following contributions:

- A new algorithm for mapping multi-rail logic expressions to an NCL gate library is proposed that improves area and delay of the mapped circuits.
- The new mapping algorithm can target different cost functions or use only a desired subset of NCL gates for mapping.
- The new mapping algorithm can be incorporated as part of the NCL design automation flow or be used as a standalone tool in a custom NCL design flow to assist mapping large multi-rail logic expressions.

### 3.3 CONVENTIONS

Assume there is a dual-rail function  $F$  expressed as a sum-of-products (SOP) of certain dual-rail signals. For example,  $F^1 = A^1B^1$ , and  $F^0 = A^0B^0 + A^1B^0 + A^0B^1$ . This is an input-complete dual-rail AND function.  $F^1$  corresponds to rail-1 of function  $F$  and comprises only one product

term, while  $F^0$  corresponds to rail-0 of function  $F$  and comprises three product terms.  $F^0$  consists of 4 distinct variables but includes 6 variables ( $A^0$  and  $B^0$  are repeated).  $F^1$  consists of 2 distinct variables and includes 2 variables. For the sake of simplicity, for the rest of the chapter, the multi-rail SOP expression will be referred to as the SOP expression and the product term will be referred to as the term. Assuming that each rail of a multi-rail signal is an independent variable, the SOP expression for  $F^1$  and  $F^0$  can be presented in a simpler form. For example, assuming  $A^0=A$ ,  $B^0=B$ ,  $A^1=C$ , and  $B^1=D$ , the SOP expression for  $F^0$  reduces to  $AB+BC+AD$ , which maps to an NCL THand0 gate, and  $F^1$  reduces to  $CD$ , which maps to an NCL TH22 gate. The standard NCL gate library is comprised of 27 gates, as shown in Table 9 (transistor numbers are for static CMOS implementation [2]). In practice, any SOP expression of 4 or fewer distinct variables can be directly mapped to one of these 27 NCL gates [2]. This property will be called  $k$ -feasibility, where  $k$  can be 2, 3, or 4 due to the fan-in limitation of the NCL gate library. In the previous example,  $F^0$  is 4-feasible because it consists of 4 distinct variables, while  $F^1$  is 2-feasible.

If an SOP expression contains more than 4 distinct variables, then it must be partitioned into sufficiently small pieces of 4 or fewer distinct variables before being mapped to NCL gates. This partitioning process is equivalent to dividing the SOP expression into several groups of terms such that each group contains 4 or fewer distinct variables (i.e., is  $k$ -feasible) and therefore can be directly mapped to an NCL gate. This process is called grouping. As an example, assume  $F^1$  is  $A^0B^1C^1+A^0B^1D^1+A^1B^1+B^1C^0+B^1D^0+A^1C^1$ . We will see that  $F^1$  can be divided into several  $k$ -feasible groups:  $A^0B^1C^1+A^1B^1+A^1C^1$ ,  $A^0B^1D^1+B^1C^0$ , and  $B^1D^0$ , where each group is  $k$ -feasible; thus, it can be mapped to an NCL gate. However,  $F^1$  can be alternatively divided into several

**Table 9: Library of the 27 standard NCL gates**

<i>Gate</i>	<i>Set Function</i>	<i># Transistors</i>
TH12	$A + B$	6
TH22	$AB$	12
TH13	$A + B + C$	8
TH23	$AB + AC + BC$	18
TH33	$ABC$	16
TH23w2	$A + BC$	14
TH33w2	$AB + AC$	14
TH14	$A + B + C + D$	10
TH24	$AB + AC + AD + BC + BD + CD$	26
TH34	$ABC + ABD + ACD + BCD$	24
TH44	$ABCD$	20
TH24w2	$A + BC + BD + CD$	20
TH34w2	$AB + AC + AD + BCD$	22
TH44w2	$ABC + ABD + ACD$	23
TH34w3	$A + BCD$	18
TH44w3	$AB + AC + AD$	16
TH24w22	$A + B + CD$	16
TH34w22	$AB + AC + AD + BC + BD$	22
TH44w22	$AB + ACD + BCD$	22
TH54w22	$ABC + ABD$	18
TH34w32	$A + BC + BD$	17
TH54w32	$AB + ACD$	20
TH44w322	$AB + AC + AD + BC$	20
TH54w322	$AB + AC + BCD$	21
THxor0	$AB + CD$	20
THand0	$AB + BC + AD$	19
TH24comp	$AC + BC + AD + BD$	18

other k-feasible groups; in other words, grouping is not unique. The question is how to group the terms such that the mapped circuit is optimal in terms of a certain cost function. This chapter aims to answer this question through the proposed grouping algorithm. Each k-feasible group is finally mapped to a distinct NCL gate, so group and gate are equivalent terms in this chapter. Also, mapping and grouping refer to the same activity in this chapter and are interchangeable.

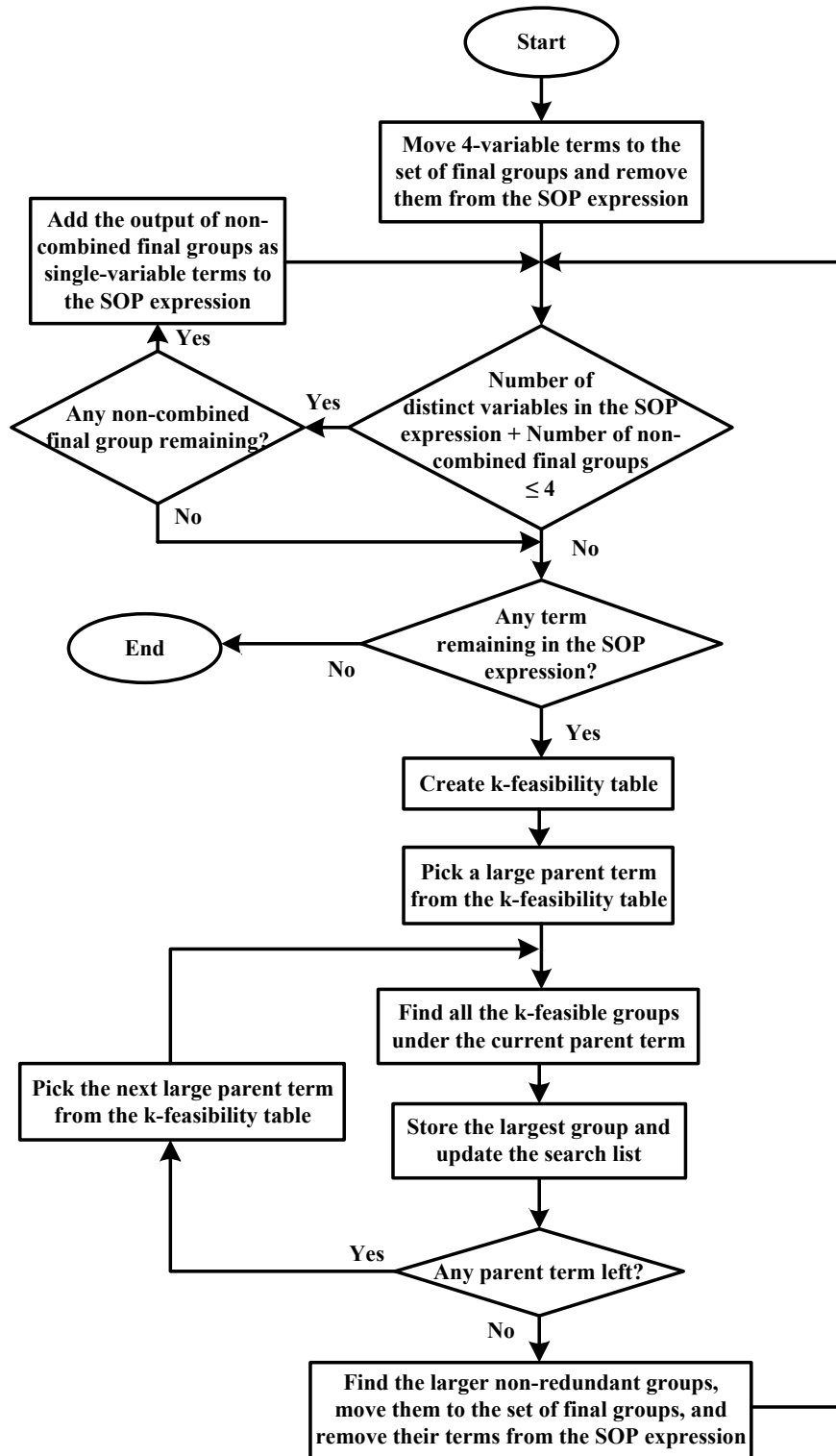


Figure 23: Original grouping algorithm

### 3.4 ORIGINAL MAPPING ALGORITHM

After several synthesis and optimization steps in [8], each combinational module in the initial synchronous RTL design is finally expressed as dual-rail SOP expressions describing its outputs in terms of its inputs. Then, each SOP expression is separately mapped to NCL gates using the grouping algorithm in [8], as shown in Figure 23. The algorithm starts with a multi-rail SOP expression as input and then groups the product terms of the SOP expression such that an area-efficient implementation is obtained. Taking  $F^1 = A^0B^1C^1 + A^0B^1D^1 + A^1B^1 + B^1C^0 + B^1D^0 + A^1C^1$  as an example, the first step in the original grouping algorithm is to remove all 4-variable terms from the initial SOP expression, because according to the NCL gate library, these terms cannot be grouped with any other terms to make larger groups and are always realized using TH44 gates. Thus, any 4-variable term in the initial SOP expression can be individually regarded as a group and moved to the set of final groups. Since  $F^1$  has no 4-variable term, no action is needed. Next, the distinct variables in the SOP expression are counted; if the number is less than 4, the previously grouped terms can be potentially combined with the SOP expression to make 4-feasible groups (this can potentially reduce area and will be discussed later).  $F^1$  contains 7 distinct variables, so the algorithm proceeds to the next decision box, where it is determined whether the SOP expression is empty or not. The initial SOP expression has not yet been modified, so the algorithm proceeds.

The next step in the original grouping algorithm is to create a k-feasibility table (Table 10). Each column in this table corresponds to one of the terms in the SOP expression. This term is called a “parent term,” and under each parent term in the k-feasibility table, the other terms (same size or smaller) that can be combined with the parent term such that their combination includes 4

**Table 10: K-feasibility table**

$A^0B^1C^1$	$A^0B^1D^1$	$A^1B^1$	$B^1C^0$	$B^1D^0$	$A^1C^1$
$A^0B^1D^1$	$A^0B^1C^1$	$B^1C^0$	$A^1B^1$	$A^1B^1$	$A^1B^1$
$A^1B^1$	$A^1B^1$	$B^1D^0$	$B^1D^0$	$B^1C^0$	$B^1C^0$
$B^1C^0$	$B^1C^0$	$A^1C^1$	$A^1C^1$	$A^1C^1$	$B^1D^0$
$B^1D^0$	$B^1D^0$				
$A^1C^1$					

**Table 11: K-feasible groups for each parent term**

$A^0B^1C^1$	$A^0B^1D^1$	$A^1B^1$
$A^0B^1C^1+A^0B^1D^1$	$A^0B^1D^1+A^1B^1$	$A^1B^1+B^1C^0+B^1D^0$
$A^0B^1C^1+A^1B^1+A^1C^1$	$A^0B^1D^1+B^1C^0$	$A^1B^1+B^1C^0+A^1C^1$
$A^0B^1C^1+B^1C^0$	$A^0B^1D^1+B^1D^0$	$A^1B^1+B^1D^0+A^1C^1$
$A^0B^1C^1+B^1D^0$		

or fewer distinct variables are listed. For example, the first column in Table 10 corresponds to the parent term  $A^0B^1C^1$ . Since the combination of  $A^0B^1D^1$  with the parent term includes 4 distinct variables, it is added under the parent term. The rest of the table is built the same way.

Now, starting from the larger parent terms and proceeding to the smaller ones (i.e., first 3-variable parent terms, then 2-variable parent terms, and finally single-variable parent terms), one parent term is picked at a time, and all the possible k-feasible groups are found by combining the parent term with the other terms listed under it. In Table 10, the parent term  $A^0B^1C^1$  can be combined with the terms listed under it to form the following k-feasible groups:  $\{A^0B^1C^1+A^0B^1D^1, A^0B^1C^1+A^1B^1+A^1C^1, A^0B^1C^1+B^1C^0, \text{ and } A^0B^1C^1+B^1D^0\}$ . Note that in the original grouping method, when the terms are combined the largest possible groups are formed, and the temporary small groups are discarded. For example, since  $A^0B^1C^1+A^1B^1$  can be combined with  $A^1C^1$  to make the larger group  $A^0B^1C^1+A^1B^1+A^1C^1$ , the original smaller group is discarded.

Using the same procedure, all the k-feasible groups for the other parent terms are derived.

**Table 12: Final k-feasible groups after first iteration**

$A^0B^1C^1$	$A^0B^1D^1$	$A^1B^1$
$A^0B^1C^1+A^1B^1+A^1C^1$	$A^0B^1D^1+A^1B^1$	$A^1B^1+B^1C^0+B^1D^0$

The k-feasible groups for  $F^1$  are shown in Table 11 (repeated groups under different parent terms are not shown). In the original grouping method, in order to reduce the runtime, after finding k-feasible groups for each parent term, the largest group is selected, and the rest of the groups are removed. Also, the selected group is added to a “search list” to avoid finding the same group for the other parent terms (this is possible because, for example, both parent terms  $A^0B^1C^1$  and  $A^0B^1D^1$  can form the same group  $A^0B^1C^1+A^0B^1D^1$ ). Table 12 shows the largest group under each parent term. A group is the largest of all groups under a parent term if it has more terms than the other groups. If all groups under a parent term have the same number of terms, the one that has more variables is the largest group. In the original grouping method, if groups of the exact same size fall under a parent term, the first group is always selected and the rest are discarded. For example, in Table 11, all the groups under parent  $A^0B^1D^1$  have the same size, so the first one is selected, and the rest are discarded. It will be shown that this approach of selecting the largest group can potentially result in non-optimal grouping; the new grouping algorithm addresses this limitation.

The original grouping algorithm then proceeds with choosing the largest non-redundant groups from the set of largest groups found under each parent term, and moving them to the set of final groups. A group is considered non-redundant if it does not share a common term with any other group. Shared terms are not allowed in the set of final groups since they introduce gate orphans and therefore impact the delay-insensitivity of NCL circuits. This issue arises because when several gates share a common term, there is a possibility that all their outputs transition during

the same DATA phase, but in that case only the fastest transition is acknowledged by the output and the other slower transitions are not acknowledged. In the original grouping method, non-redundant groups are found as follows: starting from the first group in the list, a group is selected and compared to the other groups; if it shares a common term with any other group, then the larger group is selected and the smaller group is discarded. This procedure continues until all the smaller redundant groups are removed. The remaining groups, which are large and non-redundant, are moved to the set of final groups. For example, in Table 12,  $A^0B^1C^1+A^1B^1+A^1C^1$  is first compared to  $A^0B^1D^1+A^1B^1$ , and because they both share  $A^1B^1$ , the smaller group (i.e.,  $A^0B^1D^1+A^1B^1$ ) is removed. Next,  $A^0B^1C^1+A^1B^1+A^1C^1$  is compared to  $A^1B^1+B^1C^0+B^1D^0$ , and for the same reason,  $A^1B^1+B^1C^0+B^1D^0$  is removed. At the end, only  $A^0B^1C^1+A^1B^1+A^1C^1$  remains; and it is moved to the set of final groups. This method of finding the largest non-redundant groups will be proven to be inefficient, and its limitations will be addressed in the new grouping method.

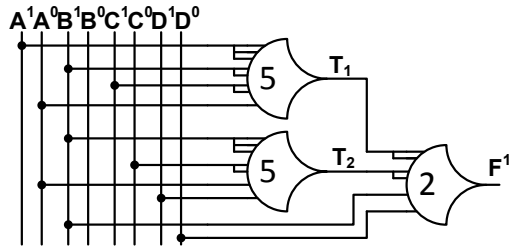
At the end of the first iteration in the original grouping algorithm, the selected groups are moved to the set of final groups, and their terms are removed from the initial SOP expression. In the previous example, the selected group (i.e.,  $A^0B^1C^1+A^1B^1+A^1C^1$ ) is removed from the initial SOP expression, reducing it to  $A^0B^1D^1+B^1C^0+B^1D^0$ .

The same procedure is then repeated for the new SOP expression until all the terms in the expression are grouped. For the previous example, at the end of the second iteration,  $A^0B^1D^1+B^1C^0$  is selected and moved to the set of final groups. After removing this group from the SOP expression, the initial SOP expression reduces to  $B^1D^0$ . At the beginning of the third iteration, the condition in the first decision box in the grouping algorithm is satisfied because only two variables



**Table 13: Final groups**

<i>Final Groups</i>
$T1 = A^0B^1C^1 + A^1B^1 + A^1C^1$
$T2 = A^0B^1D^1 + B^1C^0$
$F^1 = B^1D^0 + T1 + T2$



**Figure 24: NCL implementation**

remain in the SOP expression (i.e.,  $B^1$  and  $D^0$ ). This situation normally means that the last group will be smaller, but sometimes it is possible to add more variables to the remaining SOP expression to make the last group larger and potentially reduce the total number of final groups. The current set of final groups can be used for this purpose because the final groups eventually need to be ORed together in order to form the output. Therefore, the outputs of the current set of final groups are added as single-variable terms to the current SOP expression at this step. In the previous example, two groups already exist in the set of final groups. Let us call the output of the first group  $T1$  and the output of the second group  $T2$ . After adding  $T1$  and  $T2$  to the SOP expression, the new expression becomes  $B^1D^0 + T1 + T2$ . At the end of the third iteration,  $B^1D^0 + T1 + T2$  is selected as a final group and is removed from the SOP expression; therefore, the SOP expression becomes empty, and the grouping algorithm ends. Table 13 shows the final groups and Figure 24 shows its NCL implementation.

If  $T1$  and  $T2$  were not added to the SOP expression, then  $B^1D^0$  would be selected as the third final group. Assuming the output of this group was called  $T3$ , another group (and also another

iteration) would be required to OR the output of the final groups (i.e.,  $F^1 = T1+T2+T3$ ). Therefore, adding the output of the final groups to the SOP expression when it has less than 4 variables can potentially reduce the number of final groups and consequently produce a more area-efficient circuit. Although this approach appears helpful, in some cases it can increase both the area and the delay of a circuit. This problem can be resolved by adding a post-processing step as will be discussed in the new grouping algorithm.

### **3.5 ORIGINAL MAPPING ALGORITHM CONSTRAINTS**

Several potential constraints of the original grouping algorithm were introduced in the previous section. In this section, these constraints are discussed in more detail and some examples are provided. In the next section, our proposed grouping algorithm is introduced, and these constraints are addressed.

The original grouping method only targets area as the cost function and tries to minimize area by finding larger groups and decreasing the number of final groups. It also assumes that all 27 NCL gates are available for grouping. However, in practice, some NCL gates may not be available, or a designer could be interested in targeting delay as the cost function by using a certain set of low-latency NCL gates for mapping. The original grouping method cannot provide such a degree of flexibility. Moreover, the original grouping method usually requires several iterations to find the final groups (multi-pass grouping). Based on the original grouping algorithm shown in Figure 23, all the steps, from creating a k-feasibility table to combining terms with the parents, finding the largest group under each parent, and finally finding the largest non-redundant groups, must be performed every time the main loop is executed. This execution is computationally expensive and

could be alleviated by engaging in less computation in the main loop.

As explained previously, in the original grouping method, after finding all  $k$ -feasible groups under a parent term, the largest group is selected and the rest are discarded. However, if the  $k$ -feasible groups under a parent term have exactly the same size, then the first group is selected as the largest group. This approach of selecting the largest group can sometimes result in an inefficient grouping. For example, assume  $F^1 = A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0 + A^0D^0 + B^0C^1D^1$ . In this case, the 4-feasible groups under parent  $A^0B^1$  would be  $\{A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0$ , and  $A^0B^1 + A^0C^0 + B^1C^0 + A^0D^0\}$ . Now, if the first group is selected, the final grouping would be  $\{T1 = A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0$  (TH44w322),  $T2 = B^0C^1D^1$  (TH33), and  $F^1 = A^0D^0 + T1 + T2$  (TH24w22)}. However, if the second group under parent  $A^0B^1$  is selected as the largest group, the new grouping would be  $\{T1 = A^0B^1 + A^0C^0 + B^1C^0 + A^0D^0$  (TH44w322),  $T2 = B^0C^1D^1 + A^0D^1$  (TH54w32), and  $F^1 = T1 + T2$  (TH12)}. Based on Table 9, this grouping saves 6 transistors compared to the first grouping and is also faster (see Section 3.8).

The original grouping method is also not efficient in terms of finding the largest non-redundant groups. As discussed in the previous section, starting from the first group, each group is compared to the other groups, and if they share common terms, the largest one is selected, and the other one is discarded. For example, assume  $F^1 = B^1C^0D^0 + C^1D^0 + A^0B^1 + A^0C^1 + A^0D^1 + B^0C^1$ . The set of the largest groups after the first iteration will contain  $\{B^1C^0D^0 + C^1D^0$ ,  $C^1D^0 + A^0B^1 + A^0C^1$ ,  $A^0B^1 + A^0C^1 + A^0D^1$ ,  $A^0C^1 + A^0D^1 + B^0C^1\}$ . Using the original method to find the largest non-redundant groups results in only  $C^1D^0 + A^0B^1 + A^0C^1$  remaining at the end. However, a more careful examination reveals that selecting two other large non-redundant groups, which the original grouping

method would never select, is indeed a better choice (i.e.,  $A^0B^1+A^0C^1+A^0D^1$  and  $B^1C^0D^0+C^1D^0$ ). This alternative choice not only decreases the algorithm runtime but also can result in a smaller and faster circuit. In fact, if the first choice is used, the final groups would be  $\{T1 = A^0B^1+A^0C^1+C^1D^0$  (THand0),  $T2 = A^0D^1+B^0C^1$  (THxor0),  $T3 = B^1C^0D^0+T1$  (TH34w3),  $F^1 = T2+T3$  (TH12)} $\}$ , but if the second choice is used, the final groups would be  $\{T1 = A^0B^1+A^0C^1+A^0D^1$  (TH44w3),  $T2 = B^1C^0D^0+C^1D^0$  (TH54w32),  $F^1 = B^0C^1+T1+T2$  (TH24w22)} $\}$ . The second grouping saves 11 transistors and is faster (see Section 3.8).

Finally, the approach of adding the final groups as single-variable terms to the initial SOP expression to make larger groups is not always helpful. In fact, this approach can potentially result in a larger circuit or increase the logic levels unnecessarily and consequently make the final circuit slower. This technique is usually only beneficial if it can reduce the total number of final groups. For example, assume  $F^1 = A^0B^1C^0+A^0B^0C^1+A^1B^1C^0+A^1B^0C^1+B^1C^0D^1$ . Using the original grouping method, the final groups would be  $\{T1 = A^0B^1C^0+A^1B^1C^0$  (TH54w22),  $T2 = A^0B^0C^1+A^1B^0C^1$  (TH54w22),  $T3 = B^1C^0D^1+T1$  (TH34w3),  $F^1 = T2+T3$  (TH12)} $\}$ . In this example, combining  $T1$  with  $B^1C^0D^1$  cannot decrease the number of final groups, but it adds to the delay of the circuit. In fact, a grouping in which  $T3 = B^1C^0D^1$  (TH33) and  $F^1 = T1+T2+T3$  (TH13) maintains the same number of transistors but cuts the circuit delay almost in half (see Section 3.8).

### 3.6 PROPOSED MAPPING ALGORITHM

The standard NCL gate library is comprised of 27 gates, each of which has several properties. Some of these properties are technology-independent, such as the set function and the number of transistors, while some are technology-dependent, such as area and delay. The main idea of the

new grouping algorithm is to sort the list of NCL gates based on a cost function prior to grouping. At the time of grouping, the gates occupying a higher position in the sorted list are considered more important and will have a higher priority in grouping. Moreover, in contrast to the original grouping method, the list of NCL gates is no longer required to include all 27 gates as long as it contains enough gates to cover any given SOP expression. This usually means that the gate list must contain at least all of the NCL AND (THnn) and OR (TH1n) gates.

The first step in using the proposed grouping algorithm is to select a set of NCL gates to which the SOP expressions are to be mapped. The selected NCL gates are then sorted based on a cost function, and each gate is assigned a priority number accordingly. At the time of grouping, this priority number is used to determine which groups are most desirable. The cost function is usually area or delay. For example, if the cost function is area, the gates that cover more terms with less area will have a higher priority. In other words, let  $F$  be a multi-rail SOP expression. Assume that function  $F$  can be implemented (denoted by  $\langle \bullet \rangle$ ) by the set of NCL gates  $\{H_0, H_1, \dots, H_n\}$ :

$$\{H_0, H_1, \dots, H_n\} \langle \bullet \rangle F$$

If there exists a single NCL gate  $K$  that implements the same function:

$$K \langle \bullet \rangle F$$

Then  $K$  is said to have a higher priority than any NCL gate in the set of  $\{H_0, H_1, \dots, H_n\}$

if:

**Table 14: A typical priority table**

<i>Priority Level</i>	<i>NCL Gate</i>	<i>Set Function</i>	<i>3v &amp; 2v Terms Covered</i>	<i>Variables Covered</i>
1	TH24	AB+AC+AD+BC+BD+CD	6	12
2	TH34w22	AB+AC+AD+BC+BD	5	10
3	TH34	ABC+ABD+ACD+BCD	4	12
4	TH34w2	BCD+AB+AC+AD	4	9
5	TH44w322	AB+AC+AD+BC	4	8
	TH24comp	AC+BC+AD+BD		
6	TH44w2	ABC+ABD+ACD	3	9
7	TH44w22	ACD+BCD+AB	3	8
8	TH54w322	BCD+AB+AC	3	7
9	TH24w2	BC+BD+CD+A	3	7
10	TH23	AB+AC+BC	3	6
	TH44w3	AB+AC+AD		
	THand0	AB+BC+AD		
11	TH54w22	ABC+ABD	2	6
12	TH54w32	ACD+AB	2	5
13	TH34w32	BC+BD+A	2	5
14	TH33w2	AB+AC	2	4
	THxor0	AB+CD		
15	TH34w3	BCD+A	1	4
16	TH33	ABC	1	3
17	TH24w22	CD+A+B	1	4
18	TH23w2	BC+A	1	3
19	TH22	AB	1	2

$$Area(K) \leq \sum_{i=0}^n Area(H_i)$$

As an example, consider  $F(A,B,C,D) = AB+ACD+BCD$ .  $F$  can be implemented using a single gate, TH44w22, or it can be implemented using several other gates such as  $\{H_0 = AB$  (TH22),  $H_1 = ACD+BCD$  (TH54w22),  $H_2 = H_0+H_1$  (TH12)} $\}$ . Since TH44w22 has less area (#transistors) compared to the sum of the area for TH54w22, TH22, and TH12, it has a higher priority than all of them.

For the rest of the chapter, for the sake of comparison with the original grouping method,

**Table 15: K-feasible groups for each parent term**

$A^0B^1C^1$	$A^0B^1D^1$	$A^1B^1$	$B^1C^0$	$B^1D^0$	$A^1C^1$
$A^0B^1C^1+A^0B^1D^1$	$A^0B^1D^1+A^1B^1$	$A^1B^1+B^1C^0+B^1D^0$	$B^1C^0+B^1D^0$	$B^1D^0$	$A^1C^1$
$A^0B^1C^1+A^1B^1+A^1C^1$	$A^0B^1D^1+B^1C^0$	$A^1B^1+B^1C^0+A^1C^1$	$B^1C^0$		
$A^0B^1C^1+B^1C^0$	$A^0B^1D^1+B^1D^0$	$A^1B^1+B^1D^0+A^1C^1$			
$A^0B^1C^1+B^1D^0$	$A^0B^1D^1$	$A^1B^1+B^1C^0$			
$A^0B^1C^1$		$A^1B^1+B^1D^0$			
		$A^1B^1+A^1C^1$			
		$A^1B^1$			

let us assume that the cost function is area and that all of the NCL gates are available for grouping. A typical priority table is shown in Table 14, in which the gates are sorted based on how many 3-variable and 2-variable terms they cover. If some gates cover the same number of terms, the one that covers more variables is assigned a higher priority; if they also cover the same number of variables, then they are assigned the same priority number. Note that the NCL OR gates (TH1n) are missing from this table because they are used in the last step of the proposed grouping algorithm when the final groups are ORed. This step will be discussed later in more detail.

The proposed grouping algorithm is shown in Figure 25. In summary, the algorithm begins with moving all the 4-variable terms to the set of final groups, just as in the original grouping method. Next, all k-feasible groups are found by using a k-feasibility table and combining all parent terms with the other terms, as discussed in the original grouping method. The k-feasible groups are then arranged into priority levels according to Table 14. Next, starting from the highest non-empty priority level, the non-redundant groups in each priority level are found, their terms are removed from all the other groups, and the priority levels are updated accordingly. This procedure continues until all priority levels become empty.

Let us take the same SOP expression that was used to explain the original grouping algo-

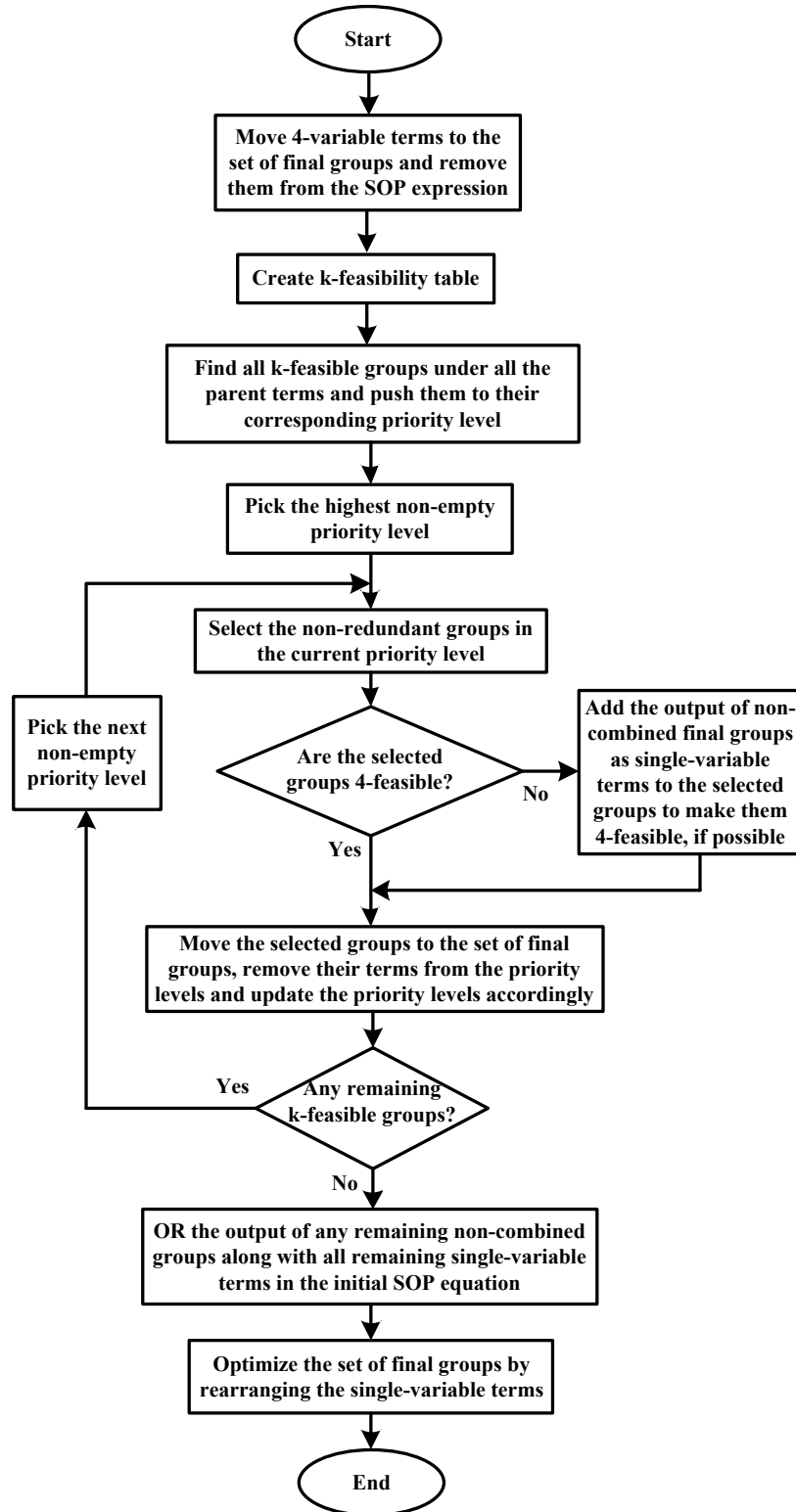


Figure 25: New grouping algorithm



rithm as an example. Assume  $F^1 = A^0B^1C^1 + A^0B^1D^1 + A^1B^1 + B^1C^0 + B^1D^0 + A^1C^1$ . Based on the algorithm shown in Figure 25, the first step is to remove all 4-variable terms from the SOP expression and move them to the set of final groups for the same reason that they were removed in the original grouping method.  $F^1$  contains no 4-variable terms, so this step is skipped. All k-feasible groups are then found by using the k-feasibility table, as discussed in the original grouping method. Table 15 shows the result. A major difference exists between the k-feasible groups found with the original grouping method and the ones found with the proposed grouping method. As mentioned previously, with the original grouping method, only larger k-feasible groups are stored, and the smaller temporary k-feasible groups are discarded. In the proposed grouping algorithm, however, all possible combinations of terms are stored. For example, for the parent  $A^1B^1$  in Table 15,  $A^1B^1 + B^1C^0 + B^1D^0$ ,  $A^1B^1 + B^1C^0$ , and  $A^1B^1$  are all stored. This enables the proposed grouping algorithm to find all the final groups in one iteration (one-pass grouping) as opposed to the original grouping algorithm that requires several iterations (see Section 3.4). Finding all k-feasible groups does not require more processing compared to the original grouping method since the process of finding larger k-feasible groups inherently includes finding smaller ones.

The next step is to push the groups into their corresponding priority levels, as demonstrated in Table 16. In this table, under each priority level, one or more groups from Table 15 are listed. The empty priority levels are not shown. The algorithm then picks one priority level at a time, starting from the higher, non-empty priority levels, and then selects the non-redundant groups in each priority level. In Table 16, the highest priority level is 8, and this level includes only one group (i.e.,  $A^0B^1C^1 + A^1B^1 + A^1C^1$ ), which is selected as a non-redundant group. The selected groups are

**Table 16: K-feasible groups pushed into their corresponding priority levels**

<i>8</i>	<i>10</i>	<i>11</i>	
$A^0B^1C^1+A^1B^1+A^1C^1$	$A^1B^1+B^1C^0+B^1D^0$ $A^1B^1+B^1C^0+A^1C^1$ $A^1B^1+B^1D^0+A^1C^1$	$A^0B^1C^1+A^0B^1D^1$	
<i>12</i>	<i>14</i>	<i>16</i>	<i>19</i>
$A^0B^1C^1+B^1C^0$	$A^1B^1+B^1C^0$	$A^0B^1C^1$	$A^1B^1$
$A^0B^1C^1+B^1D^0$	$A^1B^1+B^1D^0$	$A^0B^1D^1$	$B^1C^0$
$A^0B^1D^1+A^1B^1$	$A^1B^1+A^1C^1$		$B^1D^0$
$A^0B^1D^1+B^1C^0$	$B^1C^0+B^1D^0$		$A^1C^1$
$A^0B^1D^1+B^1D^0$			

**Table 17: Updated priority levels**

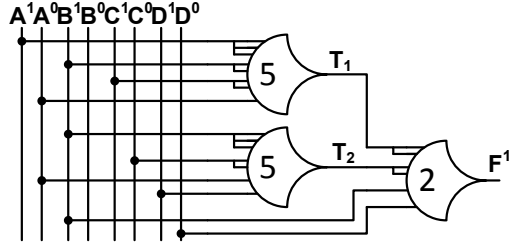
<i>12</i>	<i>14</i>	<i>16</i>	<i>19</i>
$A^0B^1D^1+B^1C^0$	$B^1C^0+B^1D^0$	$A^0B^1D^1$	$B^1C^0$
$A^0B^1D^1+B^1D^0$			$B^1D^0$

then examined to determine if they are 4-feasible (i.e., comprised of 4 distinct variables); if they are not, the output of the previously found final groups are added to them as single-variable terms to make them 4-feasible, if possible (provided that the new group is still implementable in the chosen NCL gate library). This approach is almost similar to the one used in the original grouping method. This step may not be required if the cost function is delay and can be skipped without causing problems for the whole grouping algorithm (in contrast to the original grouping method which will fail). Since the selected group in the priority level 8 is already 4-feasible, no action is required.

The selected groups are finally moved to the set of final groups, their terms are removed from the priority levels, and the priority levels are updated accordingly. For example, after removing the terms of  $A^0B^1C^1+A^1B^1+A^1C^1$  from the priority levels,  $A^1B^1+B^1C^0+B^1D^0$ , which already belongs to the priority level 10, reduces to  $B^1C^0+B^1D^0$ , which now belongs to the priority level

**Table 18: Final groups**

<i>Final Groups</i>
$T1 = A^0B^1C^1 + A^1B^1 + A^1C^1$
$T2 = A^0B^1D^1 + B^1C^0$
$F^1 = B^1D^0 + T1 + T2$



**Figure 26: NCL implementation**

14 according to Table 14. The updated priority levels are shown in Table 17. This procedure is repeated for the other priority levels until all the product terms are grouped and the priority levels become empty. In Table 17, the next highest priority level is 12, which contains two groups. Because these two groups share a common term, they are dependent, and one of them must be removed. In this example, it does not matter which one is removed (as will be explained later), so the first group is selected and the other is removed. The selected group is then moved to the set of final groups, and its terms are removed from the priority levels. This leaves only  $B^1D^0$ , which can be combined with the other final groups as shown in Table 18. The corresponding NCL implementation is shown in Figure 26. For this example, the original grouping method and the proposed one produce the same grouping result, but this is not always the case as explained in the next section.

The proposed grouping method always preserves delay-insensitivity of the initial dual-rail function. In order to prove it, one must prove that the proposed grouping method preserves input-completeness and observability.

*Theorem 1:* Grouping preserves input-completeness.

*Proof:* Let dual-rail function  $F$  be input-complete with regard to a certain variable  $X$ . Hence, all product terms of  $F$  (both  $F^1$  and  $F^0$ ) include variable  $X$  (either  $X^1$  or  $X^0$ ) [2]. Grouping is the process of combining (Boolean OR) the product terms into  $k$ -feasible groups and then ORing these groups to produce the output. Since grouping does not modify the product terms, each term will still contain variable  $X$ ; therefore,  $F$  will still be input-complete with respect to  $X$ . ■

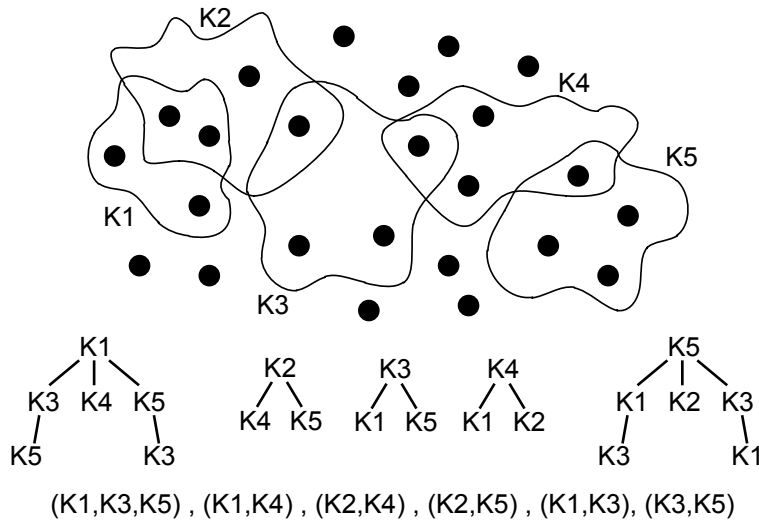
*Theorem 2:* Grouping preserves observability.

*Proof:* A circuit is observable when it does not include any gate orphans. In other words, any transition on the output of a gate must be followed by a transition on a primary output. Since grouping only combines the product terms into  $k$ -feasible groups and then ORs them to produce the final output, an assertion on the output of a gate will pass through the network of OR gates and will assert a primary output. Therefore, the output of all the gates is observable. ■

### 3.7 IMPROVEMENTS OVER THE ORIGINAL MAPPING ALGORITHM

Several advantages of the proposed grouping algorithm, such as the ability to use different cost functions, the ability to map to a subset of NCL gates, and one-pass grouping, were pointed out in the previous section. In this section, the details of some of the previously mentioned steps are explained, and it is shown how these steps improve the overall mapping process.

With the proposed grouping method, the non-redundant groups under a priority level are selected efficiently. Finding the maximum number of non-redundant groups is a special case of the *maximal independent set* problem [33]. This problem is NP-complete, so it is unlikely that it can be solved in polynomial time; however, since the number of groups in each priority level is always small (due to the small size of each combinational module, see Section 3.2), the runtime



**Figure 27: Dependency trees**

is always small. Many algorithms have been proposed for this problem in the literature [34, 35], but for our special case a simple approach as explained in Figure 27 was used. In contrast to the approach that is used in the original grouping method, this approach always finds the maximum number of non-redundant groups in a given set of groups. In Figure 27, each circle represents a number of non-redundant groups in a given set of groups. In Figure 27, each circle represents a term, and each enclosed area represents a group in the current priority level. These groups are named  $K1$  through  $K5$ . In this example, each group covers 4 terms, and some of the groups have common terms, and therefore are redundant. The terms that are not covered by any of the groups do not belong to the current priority level. In order to find the maximum number of non-redundant groups, a dependency tree must be made for each group. The following steps explain how to make a dependency tree for a group: First, a group is selected as the root (e.g.,  $K1$ ); then, all the groups that share a common term with the root are removed ( $K2$ ), and the rest of the groups are added as branches to the root ( $K3, K4, K5$ ). Then, one of the branches is selected as the new root (e.g.,  $K3$ ), and the other branches are added as branches to the new root if they share no common term with the new root (only  $K5$ ); otherwise, they are removed ( $K4$ ). This procedure, which is the same for

all other branches, continues until all the branches are removed. The same procedure is followed for the other groups until a dependency tree is created for each group. The dependency trees for the previous example are shown in Figure 27. Each path (starting from the root to the bottommost branch) in a dependency tree represents a set of non-redundant groups. For example, in the first tree of Figure 27, there are two sets of non-redundant groups, i.e., (K1, K3, K5) and (K1, K4). The order of groups is not important, so (K1, K5, K3) is considered the same as (K1, K3, K5) and is ignored. The other sets of non-redundant groups are shown in Figure 27. A set of non-redundant groups that includes more groups is clearly more desirable. In this example, (K1, K3, K5) offers the maximum number of non-redundant groups.

The dependency tree method does not always return a unique solution. Assume a scenario in which the dependency tree method returns multiple sets of non-redundant groups, each having the same number of groups. In this case, the proposed grouping method will try all the possible scenarios recursively in order to find the most optimal grouping. For example, in Table 17 under priority level 12, there are two dependant groups of the same size. One scenario could be to select  $A^0B^1D^1+B^1C^0$  as the final group and proceed with the grouping, and the other scenario could be to select  $A^0B^1D^1+B^1D^0$ . In practice, the new grouping method would consider both scenarios before selecting the optimal grouping. Technically, considering all the scenarios will result in a tree that must be traversed in order to find the most optimal grouping. A *depth-first search (DFS)* approach is used to traverse the resultant tree, which takes  $O(|V|+ |E|)$  time, where  $V$  and  $E$  are the number of vertices and edges of the tree, respectively [36].

Finally, the process of using the output of the current final groups as single-variable terms to

make larger groups is corrected in the proposed grouping algorithm, by adding an extra optimization step. As mentioned previously, this approach is usually helpful only when it can reduce the number of final groups. Moreover, if the cost function is delay, combining the final groups can be undesirable even if it reduces the number of final groups. The added optimization step in Figure 25 will explore relocating the single-variable terms in the set of final groups (without increasing the logic levels) to find the optimal grouping. This optimization step is shown in Figure 28, where  $M$  is the mapping before optimization. In this step, starting from the first logic level, each logic level is searched to find the groups that contain a single-variable term. The standard NCL gate library contains 8 gates with this property:  $A+BCD$  (TH34w3),  $A+BC+BD$  (TH34w32),  $A+BC$  (Th23w2),  $A+BC+BD+CD$  (TH24w2),  $A+B+CD$  (TH24w22),  $A+B$  (TH12),  $A+B+C$  (TH13), and  $A+B+C+D$  (TH14). If any of the above groups exists in the current logic level, the single-variable terms are moved to the next logic level, and the area/delay of the new mapping is calculated and compared to the previous one in order to find the best solution. The running time is a linear function of the number of gates in each mapping.

### 3.8 EXPERIMENTAL RESULTS

In order to compare the original and the proposed grouping methods, both methods were implemented using the Perl programming language. The developed Perl scripts were then tested on several SOP expressions and circuit components. Table 19 shows the results of running the developed scripts on some of the examples used in this chapter. For each example, the groups resulting from both grouping methods are shown. Moreover, the corresponding number of transistors (#T), area, delay, and runtime are also compared. Both grouping scripts were run on an Intel Core 2 Duo

```

1: function OPTIMIZE_MAPPING( $M$ )
2:    $S = \{\text{gates with single-variable terms in their set function}\}$ 
3:   for level  $L = 1$  to last level in  $M$  do
4:     for each gate  $g_i$  in level  $L$  do
5:       if  $g_i \in S$  then
6:         move single-variable terms of  $g_i$  to level  $L + 1$ 
7:         if  $\text{cost}(\text{new } M) < \text{cost}(\text{old } M)$  then
8:           save the new  $M$ 
9:         else
10:          revert to the old  $M$ 
11:        end if
12:      end if
13:    end for
14:  end for
15: end function

```

**Figure 28: Single-variable term relocation**

2.4 GHz machine with 4GB RAM, and the area and delay information came from an NCL physical cell library realized in a 1.8V, 0.18 $\mu$ m TSMC CMOS process. Table 19 shows that the new grouping method decreases area and delay of the mapped SOP expressions, while the runtime may increase based on the SOP expression. Although the new grouping method enables one-pass grouping and decreases the processing time in the main loop, as opposed to the original grouping method, the extra optimization steps that are missing in the original grouping method (see Section 3.7) can sometime increase the total runtime.

In order to observe the benefits of the proposed grouping method at the circuit level, this method was used to design several circuit components as listed in Table 20. In this table, the first two components are used in a quad-rail NCL multiplier [37], and the last three are used in an NCL 8051 ALU [38]. The experimental results show up to a 10% area reduction in the LOGIC component and up to a 39% delay improvement in the Q33MUL component.

Table 21 shows the result of mapping the circuit components in Table 20 using the new



**Table 19: Comparison of the original and proposed grouping methods**

	Resulted Groups	# T	Area [ $\mu\text{m}^2$ ]	Delay [ns]	Runtime [ms]
<b>1</b>	$F = A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0 + A^0D^0 + B^0C^1D^1$				
<b>Original</b>	$X = A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0$ (TH44w322) $Y = B^0C^1D^1$ (TH33) $F = A^0D^0 + X + Y$ (TH24w22)	52	271	660	13.5
<b>Proposed</b>	$X = A^0B^1 + A^0C^0 + A^0D^0 + B^1C^0$ (TH44w322) $Y = A^0D^1 + B^0C^1D^1$ (TH54w32) $F = X + Y$ (TH12)	46	254	434	13.8
<b>2</b>	$F = B^1C^0D^0 + C^1D^0 + A^0B^1 + A^0C^1 + A^0D^1 + B^0C^1$				
<b>Original</b>	$X = A^0B^1 + A^0C^1 + C^1D^0$ (THand0) $Y = A^0D^1 + B^0C^1$ (THxor0) $Z = B^1C^0D^0 + X$ (TH34w3) $F = Y + Z$ (TH12)	63	350	734	13.9
<b>Proposed</b>	$X = A^0B^1 + A^0C^1 + A^0D^1$ (TH44w3) $Y = B^1C^0D^0 + C^1D^0$ (TH54w32) $F = B^0C^1 + X + Y$ (TH24w22)	52	293	704	29.7
<b>3</b>	$F = A^0B^1C^0 + A^0B^0C^1 + A^1B^1C^0 + A^1B^0C^1 + B^1C^0D^1$				
<b>Original</b>	$X = A^0B^1C^0 + A^1B^1C^0$ (TH54w22) $Y = A^0B^0C^1 + A^1B^0C^1$ (TH54w22) $Z = B^1C^0D^1 + X$ (TH34w3) $F = Y + Z$ (TH12)	60	368	739	5.6
<b>Proposed</b>	$X = A^0B^1C^0 + A^1B^1C^0$ (TH54w22) $Y = A^0B^0C^1 + A^1B^0C^1$ (TH54w22) $Z = B^1C^0D^1$ (TH33) $F = X + Y + Z$ (TH13)	60	352	447	11.4

grouping method when only a restricted subset of NCL gates is available. As explained before, this is not possible in the original grouping method. In this experiment, half of the gates in the standard NCL gate library were removed and grouping was performed with the other half. The results show that the mapped circuits have more area and delay compared to the case with all the NCL gates available, since less optimization is possible with a restricted subset of gates.

Finally, although the proposed grouping method cannot be directly compared to the other previous works (see Section 3.2), an indirect comparison is possible through incorporating the new

**Table 20: Mapping circuit components using the original and proposed grouping methods**

Component	Grouping Method	Area [ $\mu\text{m}^2$ ]	Delay [ns]	Runtime [ms]
Q33MUL	Original	1021	597	79
	Proposed	977	363	65
Q322ADD	Original	1665	822	94
	Proposed	1626	534	80
FLAGS	Original	1581	781	26
	Proposed	1543	493	87
BCDAD	Original	1498	712	21
	Proposed	1498	712	38
LOGIC	Original	5899	415	296
	Proposed	5313	415	486

**Table 21: Mapping circuit components to a restricted subset of NCL gates**

Component	Grouping Method	Area [ $\mu\text{m}^2$ ]	Delay [ns]	Runtime [ms]
Q33MUL	All gates	977	363	65
	Limited gates	1061	415	55
Q322ADD	All gates	1626	534	80
	Limited gates	1979	1338	69
FLAGS	All gates	1543	493	87
	Limited gates	1906	1882	96
BCDAD	All gates	1498	712	38
	Limited gates	1680	841	35
LOGIC	All gates	5313	415	486
	Limited gates	5899	415	401

grouping algorithm in a design flow such as flow (b). For this purpose, the proposed grouping algorithm was added to the NCL design flow (b) and compared against the NCL design flow (a) by synthesizing several circuits listed in Table 22. In flow (a), the UNCLE toolset [9] is used to convert synchronous circuits to NCL and then the resultant circuits are further optimized using the ATN\_OPT toolset [7]. Both of these tools (i.e., UNCLE and ATN\_OPT) and the detailed instructions on how to use them are available online for public access at the time of writing this chapter.

In flow (b), the methods described in [8] were used to partition a synchronous circuit to smaller modules and derive the optimal input-complete dual-rail output SOP expressions, which were then mapped to NCL gates using the proposed grouping method to build the final circuit. In Table 22, several circuits are listed. These circuits include decoders (2-to-4 line and 3-to-8 line), logic blocks (4-bit and 8-bit), parity checkers (4-bit and 8-bit), comparators (4-bit and 8-bit), and multipliers (4×4 and 8×8). For each circuit the corresponding number of inputs (#i), outputs (#o), and NCL gates (#g) is also listed. Based on the results, the area and delay advantage for each flow is design-dependent and can be better or worse. Flow (b) particularly works better for the circuits that have more outputs than inputs, so more area reduction becomes possible through making a restricted set of outputs input-complete (i.e., using weak conditions of Seitz). On the other hand, flow (a) works better when more high-level components are inferred at synthesis time. For example, in the case of the multipliers, since they are mainly synthesized to full-adders, the UNCLE tool replaces these components with the manually optimized NCL full-adders (when converting the 3NCL circuit to 2NCL circuit), which results in much better area as compared to flow (b). At this time flow (b) does not take advantage of hand-optimized NCL components; however, this feature can be incorporated into the flow in future versions.

### **3.9 CONCLUSION**

A new mapping algorithm was proposed and compared to the original one for mapping multi-rail logic expressions to the NCL gate library. Both mapping algorithms were implemented in the Perl programming language and tested on some multi-rail logic expressions and circuit components. The proposed mapping algorithm was shown to outperform the original one in terms

**Table 22: Comparison of the two NCL design flow categories**

Circuit	Flow (a)			Flow (b)		
	#i/#o/#g	Area [ $\mu\text{m}^2$ ]	Delay [ns]	#i/#o/#g	Area [ $\mu\text{m}^2$ ]	Delay [ns]
<b>dec2x4</b>	2/4/14	1294	632	2/4/8	529	316
<b>dec3x8</b>	3/8/28	2588	632	3/8/20	915	493
<b>logic4</b>	8/12/24	2426	316	8/12/24	1739	236
<b>logic8</b>	16/24/48	4853	316	16/24/48	3478	236
<b>par4</b>	4/1/6	711	710	4/1/6	711	473
<b>par8</b>	8/1/14	1659	1656	8/1/14	1659	710
<b>comp4</b>	8/3/48	4593	2765	8/3/66	5824	2200
<b>comp8</b>	16/3/104	9978	5294	16/3/146	12968	4818
<b>mul4x4</b>	8/8/100	9933	3338	8/8/128	11476	3342
<b>mul8x8</b>	16/16/418	41641	8279	16/16/640	57310	8122

of area and delay reduction. It was shown that the new mapping algorithm can result in up to 10% area reduction and 39% delay reduction. Although the new mapping algorithm performs mapping in one pass requiring less computation in the main loop, its average runtime is higher compared to the original method because the new mapping algorithm incorporates several extra optimization steps. Also, in contrast to the original mapping algorithm, the new mapping algorithm can map a multi-rail expression to a restricted subset of NCL gates and can target any cost function.

## 4 SCL DESIGN FOR TESTABILITY

### 4.1 INTRODUCTION

Sleep Convention Logic (SCL), also known as Multi-Threshold NULL Convention Logic (MTNCL) [39], is a variant of NCL that takes advantage of MTCMOS power-gating technique to further reduce the power consumption. The application of MTCMOS to NCL circuits comes with interesting architectural changes that ultimately results in area and performance advantage as well. SCL was originally developed in [2]. SCL combines the idea of NCL with early completion [40] and MTCMOS power-gating. SCL framework is shown in Figure 29. The SCL framework is very similar to NCL. Similar to NCL, each pipeline stage contains a combinational logic function block ( $F_i$ ), a register block ( $R_i$ ), and a completion detector block ( $CD_i$ ). SCL requires an extra gate to synchronize between DATA and NULL phases. This extra gate is a simple resettable C-element with inverted output which will be called the completion C-element ( $C_i$ ) here. Similar to NCL, SCL uses dual-rail encoding for signals. Combinational logic blocks in SCL are made of threshold gates and implement unate functions where no logic inversions are allowed. SCL utilizes fine-grained power gating by incorporating sleep signal in every single gate. The sleep signal, denoted as  $s$ , once asserted disconnects the output of each gate from  $V_{DD}$  and pulls it to GND resulting in

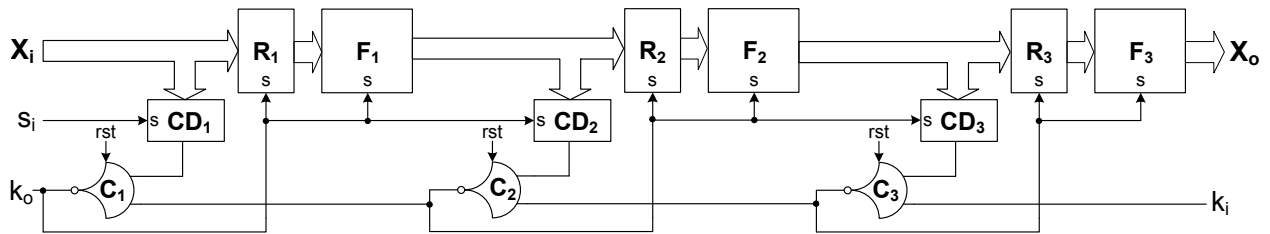
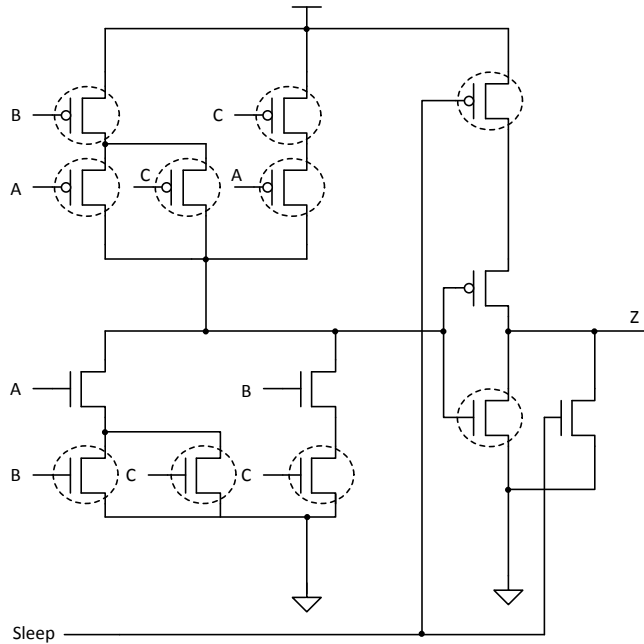


Figure 29: SCL framework



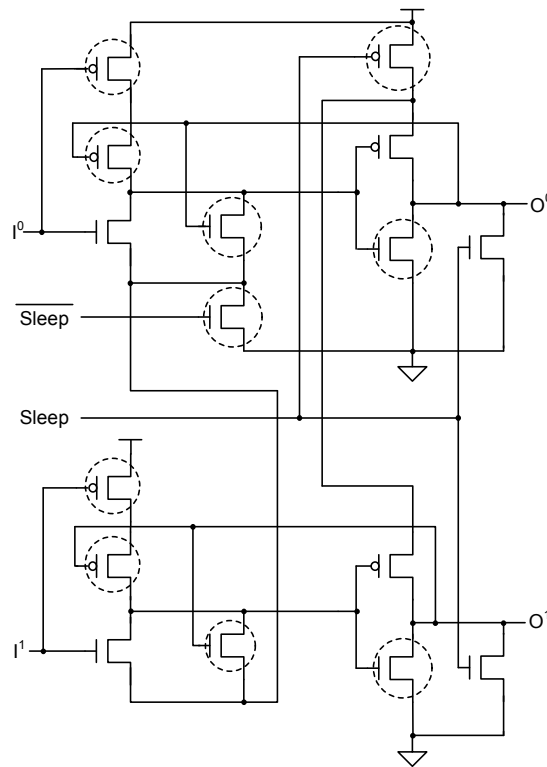
**Figure 30: SCL gate structure**

a NULL state at the output of the corresponding block. The output of an SCL gate then remains low until sleep signal is deasserted and the input values satisfy its set function. Consequently, SCL circuits also alternate between DATA and NULL phases although in SCL the NULL state is forced via sleep signal rather than by propagation of a NULL wavefront through the circuit.

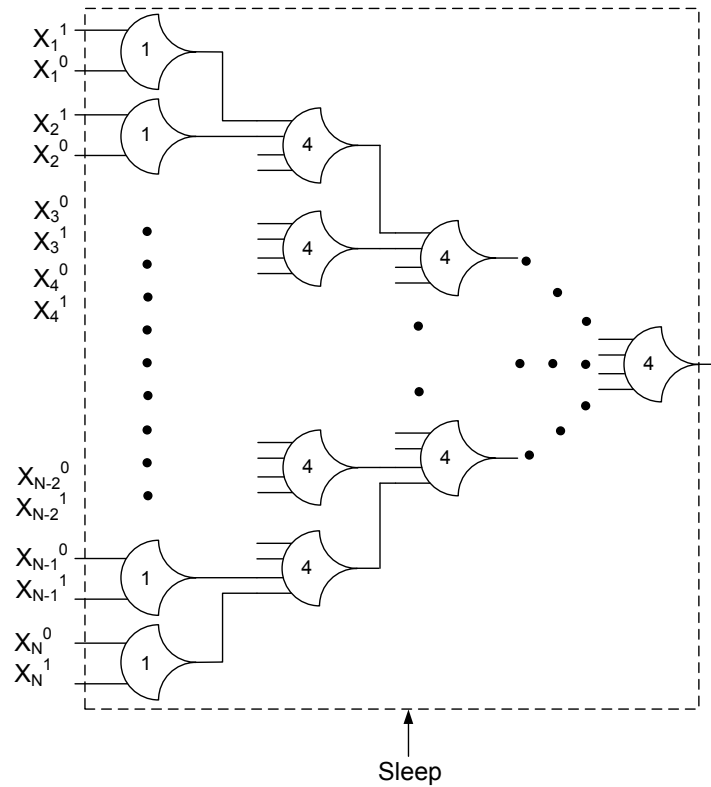
Figure 30 shows an SCL gate (TH23). Similar to NCL gates each SCL gate is made of a set block and a hold0 block, however, in contrast to NCL gates, the reset and hold1 blocks are removed. The reset block is removed since the reset functionality is now performed through the sleep mechanism. Asserting the sleep signal disconnects the output inverter from  $V_{DD}$  and pulls the gate's output ( $z$ ) to GND which will effectively reset the gate. Since the inputs of each SCL gate come from the outputs of preceding SCL gates, the internal node would then precharge to  $V_{DD}$  completing the reset phase. The hold1 block is removed in an SCL gate since the hold1

block's job was to add hysteresis to NCL gates in order to ensure input-completeness with regard to NULL. In SCL circuits, however, since all the gates within the combinational blocks are forced to reset by asserting the sleep signal, input-completeness to NULL is inherently ensured and no NULL wavefront propagation is needed anymore. In Figure 30 the circled transistors are high- $V_{th}$  in order to reduce leakage current when the gate is in sleep mode and consequently reduce the static power consumption. Other variants of SCL gates also exist. A comparison of some of these variants can be found in [41] and [42].

An SCL single-bit dual-rail register, shown in Figure 31, is essentially made of two latches, one for each signal rail. Once the sleep signal is asserted, both outputs of the register go to low signaling a NULL value. The outputs remain low until a new DATA value arrives and the sleep signal is deasserted. The register outputs will then get asserted according to the DATA value and



**Figure 31: SCL register implementation**



**Figure 32: SCL completion detector**

remain asserted even if the inputs of the register are all deasserted. This latching behavior is special because once an output is asserted by an input it cannot be deasserted by the same input; it only gets deasserted by the sleep signal. In other words, the latches can be considered one way.

An SCL completion detector is similar to the one used in NCL early completion except for using SCL gates instead of normal NCL gates so that the completion detector could also be put into sleep mode in NULL phase. The implementation of an SCL completion detector is shown in Figure 32. Finally, each stage of an SCL pipeline requires a completion C-element. This resettable C-element with inverted output is required to synchronize sleep signals and is critical for a safe DATA/NULL phase alternation; therefore, it must always be active during pipeline operation and it cannot be put to sleep. In fact this is the only gate in an SCL pipeline that never sleeps.



SCL circuits have several advantages over the traditional NCL circuits. Most of these advantages are the direct result of applying sleep mechanism to the circuit. The first obvious advantage is reducing the static power consumption due to power-gating through sleep transistors. Since NULL phase is now forced through sleep signal rather than waiting for a NULL wavefront to propagate through the circuit, the gates do not need to have hysteresis because input-completeness with regard to NULL is inherently ensured by sleeping all the gates. Removing hysteresis from NCL gates results in a significant amount of area saving. Also, since the combinational blocks in an SCL pipeline cannot wake up until a complete DATA set is available at the input of their preceding register, input-completeness to DATA is explicitly ensured. As a result, no extra logic is required to be added to a combinational block to make it input-complete with regard to DATA. Finally, observability in SCL circuits is also ensured via the sleep mechanism since any potential orphan is explicitly cleared between each two adjacent DATA phases once sleep signal is asserted. Relaxing the NCL circuits from input-completeness and observability requirements results in a significant area reduction and boosts the performance of SCL circuits. A case study comparing different NCL and SCL variants is demonstrated in [41,43]. The main disadvantage of SCL circuits is that it is no longer strictly QDI since it uses early completion. As described in [40] early completion technique involves some timing assumptions. In addition, sleep signal must be appropriately buffered in order to ensure that all gates reset in a timely manner. Since these timing assumptions are usually trivial SCL can still be considered almost QDI.

Design for testability (DFT), is a major concern in today's semiconductor industry in order to reduce test time and consequently decrease time-to-market. In contrast to NCL circuits, no

DFT methodology has been developed for SCL circuits so far. The current NCL specific DFT techniques cannot be directly used for SCL circuits due to the structural differences caused by introducing sleep mechanism. However, SCL circuits can partially use these techniques as discussed in section 4.3. The NCL specific DFT techniques in the literature are discussed in section 4.2.

## 4.2 PREVIOUS WORKS

Current ATPG tools do not support asynchronous circuit styles such as NCL due to asynchronous feedback paths and absence of clock signal. There are mainly two approaches to make NCL circuits testable: limited insertion of control/observation points to increase fault coverage [44], and synchronous modeling of NCL pipelines to make them compatible with synchronous ATPG tools and using scan chain technique [45].

The first approach to make NCL circuits testable focuses on finding nodes that are not easily controllable or observable and then inserts additional control signals or observation points to improve testability. For example, the output of each completion detector in an NCL pipeline is connected to the register of the previous stage. This creates an asynchronous feedback path that is not easily controllable by synchronous ATPG tools especially when the signal is buried in a deep pipeline. Breaking the feedback path and inserting an XOR gate controlled by a primary input provides a test point that improves controllability.

Moreover, in an NCL pipeline some nodes may not be easily observable at primary outputs. These nodes themselves can be made primary outputs to increase observability but this is not feasible if there are many unobservable nodes. Alternatively several unobservable nodes can be consolidated to a single primary output via an XOR tree. The XOR tree, however, takes up a

lot of space especially for more complex designs with several pipeline stages and also increases the number of primary outputs significantly so it is not very practical. Alternatively, scannable observation latches can be used where nodes are flagged as unobservable by the ATPG tools in order to increase observability without significantly increasing the primary outputs.

The previous methods as discussed in [44], although improve testability, take up significant space and cannot offer a high fault coverage. For example, the original fault coverage of a  $4 \times 4$  pipelined NCL multiplier was reported to be 16%. After adding XOR gates to the circuit to improve controllability and observability the fault coverage increased to only 21%. Finally the insertion of scannable observation latches increased the fault coverage to 45% which is still not acceptable. In order to cope with low fault coverage then reference [44] proposes to break the internal feedback path inside every NCL gate and insert a latch. This technique is then shown to improve testability significantly (almost 100% fault coverage) but it comes with substantial area increase which makes it impractical.

The second approach to make NCL circuits testable, as described in [45], starts with modeling the NCL pipeline and how stuck-at faults impact its behavior. It first proves that in an acyclic NCL pipeline with  $M$  stages, the faults in completion detectors can be checked by applying at most  $\frac{M}{2} + 1$  pairs of {DATA, NULL} to the pipeline. If there is a stuck-at fault at any node inside the completion detectors, the pipeline is guaranteed to stall. The value of DATA is not important and, in fact, the same DATA value can be repeated. It is further shown that faults in the registers can be eliminated by fault collapsing based on fault dominance. Consequently, both completion detectors and registers can be dropped from stuck-at fault checking.

The original NCL pipeline can then be considered as a purely NCL combinational logic after removing the register and completion detector circuitry. Then proving that irredundant NCL combinational circuits are fully testable for all stuck-at faults, a method is described to generate test vectors using conventional ATPG tools. In this method each NCL gate is replaced with its set phase equivalent boolean function. Then conventional ATPG tools are used to generate test patterns for all its stuck-at-0 faults since NCL circuits are unate and in set phase only rising transitions happen within the circuit. The stuck-at 1 faults are then checked by padding the generated test vectors with NULL values because if a DATA value asserts a node in DATA phase, the following NULL phase will deassert it according to the NCL two phase operation protocol. For cyclic pipelines where there are feedback loops in the data-path, the pipeline needs to be converted to an acyclic pipeline before using the previous method. Partial-scan technique can be used for this purpose.

### **4.3 PROPOSED DESIGN FOR TESTABILITY METHODOLOGY**

#### **4.3.1 MOTIVATION**

The NCL testing methods described in the previous section can only be partially used for SCL circuits. This is a result of adding sleep mechanism to the circuit. The sleep mechanism violates some of the assumptions that were made for the previous methods. For example, removal of completion detector and register blocks from fault checking is no longer possible because in SCL existence of stuck-at faults may not necessarily lead to a pipeline stall. The SCL registers also have a special behavior requiring a new scan cell design that is compatible with SCL logic. Due to differences such as the ones mentioned, the SCL design for testability methodology must

be investigated separately in its own context.

As discussed before, each stage of an SCL pipeline is made of four separate blocks: combinational logic function ( $F_i$ ), completion detector ( $CD_i$ ), register ( $R_i$ ), and completion C-element ( $C_i$ ). Since the stuck-at faults in each block can impact the SCL pipeline in different ways, each block is analyzed separately. Also in the following analysis it is assumed that the sleep signals are fault-free. The effect of stuck-at faults on sleep signals is analyzed in section 4.3.3.

### 4.3.2 LOGIC FAULT ANALYSIS

**Faults on Completion C-element Signals** As mentioned before, the completion C-element is needed to synchronize sleep signals and allow a safe propagation of DATA/NULL wavefronts. Also the completion C-element is the only gate in an SCL pipeline which does not have sleep capability, hence, it is implemented like a normal C-element as used in NCL logic. Since in each DATA/NULL phase all the inputs and consequently the output of a completion C-element must make a transition for the corresponding DATA/NULL wavefront to propagate through the pipeline the following theorem can be deduced:

*Theorem 1:* In an SCL pipeline, all stuck-at faults on the inputs and output of all the completion C-elements can be detected by allowing a single {DATA, NULL} pair to propagate through the pipeline.

*Proof:* Assume the SCL pipeline is reset to all-NULL state where all the stages are in sleep mode and the output of all the completion C-elements including the  $K_i$  and  $K_o$  signals are high. By providing a single DATA set at the pipeline input, the DATA should propagate all the way through the pipeline and produce a valid DATA set at the pipeline output only if the inputs and output of

completion C-elements make a single transition each. Once a valid DATA set is detected at the pipeline output,  $K_i$  is deasserted and a NULL set is provided at the pipeline input. The NULL set then causes each pipeline stage to consecutively enter the sleep state, which eventually produces a valid NULL set at the pipeline output only if the inputs and output of completion C-elements make another transition each. Consequently, a complete propagation of a {DATA, NULL} pair through the pipeline requires that the inputs and output of each completion C-element make exactly one low-to-high and one high-to-low transition which ensures that there is no stuck-at-0 or stuck-at-1 fault on the inputs and output of completion C-elements. ■

**Faults in Completion Detector** As mentioned before, the stuck-at faults in a completion detector may not necessarily result in a pipeline stall. For example, a stuck-at-1 fault on the output of any gate in a completion detector (except the final gate) is hidden by the sleep signal. In other words, in NULL phase, even if a gate's output is stuck-at-1, the completion detector will still produce a 0 at its output once the sleep signal is asserted, as long as the last gate in the completion detector has no stuck-at-1 fault. This is in fact a consequence of using reset signal to force the completion detector to get clear rather than allowing a NULL propagation to clear it.

Note that if the output of the last gate in the completion detector is stuck-at-1, the pipeline will stall after a while. This is easy to prove since the output of the last gate is in fact an input to a completion C-element and according to theorem 1 a stuck-at fault on the input of a completion C-element can be detected by propagating a single {DATA, NULL} pair through the pipeline. Also note that the fact that the sleep signal hides the stuck-at-1 faults in the completion detector, does not mean the circuit will work correctly. In fact, stuck-at-1 faults can produce premature RFN in the

DATA phase, running the risk of sleeping the combinational block and, therefore, clearing DATA wavefront before a complete DATA set is produced at the output of the combinational logic. Once the incomplete DATA set propagates to the next pipeline stage, it may or may not cause deadlock due to the fact that SCL combinational logic is not input-complete.

In contrast to detecting the stuck-at-1 faults in the completion detector that is not straightforward, stuck-at-0 faults always result in a deadlock so detecting them is easy. This is due to the fact that in DATA phase to generate a high at the output of the completion detector, all the gates within the completion detector must transition to high. Therefore, if even a single transition does not happen due to a stuck-at-0 fault, the output of the completion detector cannot go to high which eventually results in a deadlock since the output of the completion detector is an input of the completion C-element (see theorem 1).

**Faults in Combinational Logic** Combinational logic blocks in SCL are unate. In DATA phase the gates within a combinational block can only make low-to-high transitions and in NULL phase they can only make high-to-low transitions. This might imply that an approach similar to [45] can be used to detect stuck-at faults in the combinational logic but in fact it is not possible for two reasons. The first reason is that SCL combinational logic is not input-complete so, in contrast to NCL, a stuck-at-0 fault on a signal may not necessarily stop it from producing a valid output DATA set hence the pipeline may not stall. And the second reason is that because in SCL the NULL state is forced by the sleep signal rather than by the propagation of a NULL wavefront, a stuck-at-1 fault on the output of a gate may be hidden by the gates at its fanout if those gates can be properly put to sleep mode. In other words, a valid NULL set may be produced at the output of a combinational

logic block regardless of any internal stuck-at-1 faults as long as the last level of logic can be properly put to sleep mode. Again, although the internal stuck-at-1 faults are not troublesome in the NULL phase and cannot stall the pipeline, but in DATA phase they will produce wrong output so they must be detected.

In summary, in contrast to NCL, in SCL stuck-at faults in the combinational logic blocks may not necessarily lead to a pipeline stall, therefore, a different approach is needed to detect them as proposed in theorem 2:

*Theorem 2:* In an SCL pipeline, when the sleep signal is disabled, each combinational logic block behaves exactly like a traditional synchronous combinational logic block; therefore, the well-known synchronous combinational ATPG techniques can be used to detect its stuck-at faults.

*Proof:* When the sleep signal is disabled (i.e. tied to GND), the SCL gate in Figure 30 reduces to a normal Boolean threshold gate that is made of a set block that defines its functionality, a hold0 block that is the complement of its set function (as in normal static Boolean gates), and an output inverter. Since an SCL combinational block is made of SCL gates, disabling the sleep signal therefore makes the whole combinational block a normal synchronous combinational block made of normal Boolean threshold gates. And finally, it has been proved [46] that irredundant networks made of unate Boolean gates are fully testable; therefore, traditional synchronous combinational ATPG techniques can be used to detect stuck-at faults. ■

**Faults in Register** According to theorem 2, traditional combinational ATPG tools can be used to generate test patterns for detecting stuck-at faults in the SCL combinational blocks. These test



patterns are then needed to be applied to each combinational block through a scan chain design similar to a synchronous approach. This implies that the SCL registers must be augmented to have functionalities that are expected from a traditional scan cell. The SCL scan cell design and its corresponding testing method will be discussed in section 4.3.4.

### 4.3.3 SLEEP SIGNAL FAULT ANALYSIS

The analysis performed in the previous section was based on the assumption that the sleep signals are fault-free. But in reality the sleep signals can also be subject to stuck-at faults. In this section the effect of stuck-at faults on sleep signals is analyzed. In the SCL pipeline, as shown in Figure 29, each sleep signal generated by the output of a completion C-element is forked to a register block, a combinational logic block, and a completion detector block. The effect of a faulty sleep signal on each one of these blocks is analyzed separately.

**Sleep Signal Fork to Registers** A sleep signal that forks to a register block can be either stuck-at-0 or stuck-at-1. In the case of a stuck-at-1 fault, the register outputs get stuck to 0, causing the register to output NULL all the time. This can be easily detected since no DATA set can then propagate through the pipeline, causing the pipeline to stall. This conclusion is, however, based on the assumption that all the forks of the sleep signal are stuck-at-1 simultaneously. This is not a realistic assumption since stuck-at-1 faults could randomly happen on one or more forks rather than all of them. In this case, the register output will not be a complete NULL set and, therefore, it cannot be said that it stalls the pipeline. This is because in SCL combinational logic blocks are not input-complete and, therefore, an incomplete input DATA set may be still able to generate a

complete output DATA set. This consequently hides the stuck-at-1 faults making their detection difficult. A number of solutions can be thought of in this case. One solution could be to require that the combinational blocks must be input-complete for the purpose of testability. This is very expensive since extra logic needs to be added to the combinational logic blocks which increases area, delay, and power consumption of the design. Another solution would be to generate “smart” test patterns that make up for the lack of input-completeness and cause the pipeline to stall anyway. Since the current ATPG tools are all designed for synchronous circuits and, therefore, are oblivious to asynchronous logic styles and concepts such as input-completeness, this also does not seem a practical solution at this time. Fortunately, there is a cheap and practical solution based on the scan-chain design that will be discussed in section 4.3.4.

In the case of a stuck-at-0 fault on the sleep signal, the register outputs will never get back to NULL once they are set to DATA. As discussed before, this is due to the special design of SCL registers that makes them act like one-way latches that can be set by their input but cannot be reset without the intervention of sleep signal. When the output of registers do not get properly reset by sleep signal and the next DATA phase comes (note that this is possible in SCL since RFD is generated by the sleep signal rather than by the propagation of a NULL wavefront) it will cause registers to output illegal value (i.e. both rails of a dual-rail signal being high simultaneously) if the new DATA set is different than the previous DATA set. The propagation of illegal values through the pipeline can then be interpreted as a sign of stuck-at-0 on the sleep signal. Again, similar to the stuck-at-1 case, the stuck-at-0 fault may not affect all the forks of the sleep signal, rather a few of them, making the detection of illegal values more difficult. For example, if only a single fork of the

sleep signal is stuck-at-0 then the illegal value generated by that single register bit may resolve to a legal DATA value at the output of the combinational logic, therefore, hiding the stuck-at-0 fault. In section 4.3.4 it will be shown that the same scan-chain technique that is used to detect stuck-at-1 faults on the forks of a sleep signal can be used to also detect stuck-at-0 faults.

**Sleep Signal Fork to Completion Detectors** A sleep signal that forks to a completion detector block can be either stuck-at-0 or stuck-at-1. When it is stuck-at-1, the completion detector output is always 0, and therefore, the pipeline stalls according to theorem 1. Here the stuck-at-1 fault on any single fork of the sleep signal will have the same effect; therefore, finding stuck-at-1 faults on the sleep signal and its forks is easy and, in fact, it can be performed at the same time of testing the completion C-element signals.

When the sleep signal is stuck-at-0, the completion detector still functions correctly. This is because the role of the sleep signal to reset the gates is not really needed in a completion detector because in the NULL phase the propagation of NULL wavefront can effectively do the same job, albeit with a higher delay. In other words, disabling the sleep signal or removing it altogether from a completion detector only leads to performance and power consumption penalty and has no functionality impact whatsoever. As a result, the sleep signal is redundant in terms of testability, and therefore, untestable.

In summary, from above analysis we come to the conclusion that the sleep signal fork to completion detector is automatically tested for stuck-at-1 faults at the time of testing the completion C-elements and is untestable for stuck-at-0 faults since they do not impact the functionality of the completion detector logic. As a result, sleep signal forks to completion detectors need no further

consideration.

**Sleep Signal Fork to Combinational Logic Blocks** A sleep signal that forks to a combinational logic block can be either stuck-at-0 or stuck-at-1. When it is stuck-at-1, the combinational logic will be in sleep mode at all times and no DATA set can then propagate through it, resulting in a deadlock. This conclusion is again based on the assumption that all the forks of the sleep signal are stuck-at-1 simultaneously which is not very likely. In many cases it is only a few forks of the sleep signal that are stuck-at-1, in which case, the combinational block may still work correctly, or at least not cause a deadlock, depending on the DATA being processed at the time. Fortunately, through fault collapsing, the stuck-at-1 faults on the sleep signal forks can be detected during stuck-at-0 fault checking on the output of the gates within the combinational block.

Finally, similar to the completion detectors, stuck-at-0 faults on the sleep signal forks have no functionality impact on the combinational logic blocks, and therefore, are not testable. This is because in NULL phase, as long as the register preceding each combinational block is properly put to sleep mode, the propagation of NULL wavefront will reset all the gates within the combinational logic regardless of any stuck-at-0 fault on the sleep signal forks. However, this may require some timing analysis to make sure that the combinational block gets completely cleared before the next DATA phase starts.

In summary, the stuck-at faults on the sleep signal forks are either untestable or can be ignored due to fault collapsing, therefore, these faults also do not need to be considered any further.

#### 4.3.4 TEST PROCEDURE

**Fault Analysis Summary** After analyzing different fault scenarios and how they impact the SCL pipeline we can now devise a methodology to perform testing. Before that let us summarize the various conclusions that were drawn in the previous section:

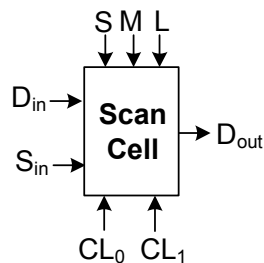
- Based on theorem 1, by allowing a single {DATA, NULL} pair to propagate through the SCL pipeline, all the stuck-at faults on the inputs and output of all the completion C-elements can be detected.
- Based on theorem 2, by disabling the sleep signal the combinational logic blocks will become normal synchronous circuit that can then be checked for stuck-at faults using traditional ATPG tools.
- The stuck-at faults on the sleep signal forks within a combinational logic block are either untestable (stuck-at-0 faults) or can be ignored through fault collapsing (stuck-at-1 faults).
- The stuck-at faults on the sleep signal forks within a completion detector block are either untestable (stuck-at-0 faults) or can be detected during the test of the completion C-elements (stuck-at-1 faults).
- The stuck-at faults on the sleep signal forks within a register block are best tested through a scan-chain design to be discussed. Scan-chain design is also needed to apply the ATPG generated test patterns to the combinational logic blocks.

**Replacing Registers with Scan Cells** Similar to a synchronous scan-based testing approach, the SCL registers need to be replaced with scan cells in order to shift in the test patterns generated by ATPG, capture the results, and shift the results out. Figure 33 shows the interface of such a scan cell.

Since SCL uses dual-rail encoding, each register bit is made of two scan cell, one for each rail.  $D_{in}$  is the main input which could be any rail of a dual-rail input signal.  $S_{in}$  is the scan input and  $D_{out}$  is the output of the scan cell. In a scan chain configuration, the  $D_{out}$  of each scan cell is connected to the  $S_{in}$  of the next scan cell.  $M$  is the test mode selection signal. When  $M = 0$  the

scan cell is in normal mode but when  $M = 1$  the scan cell enters the test mode. In normal mode the scan cell operates exactly like an SCL register but in test mode it behaves like a traditional LSSD-type scan cell [47] where data can be shifted from  $S_{in}$  to  $D_{out}$  through the non-overlapping clock signals  $CL_0$  and  $CL_1$ . In test mode, once the test patterns are applied to the combinational logic and a sufficient amount of time is passed, the outputs of the combinational logic can be loaded into scan cells using signal  $L$ . Finally,  $S$  is the sleep signal that puts the register in sleep mode when scan cell is in normal mode.

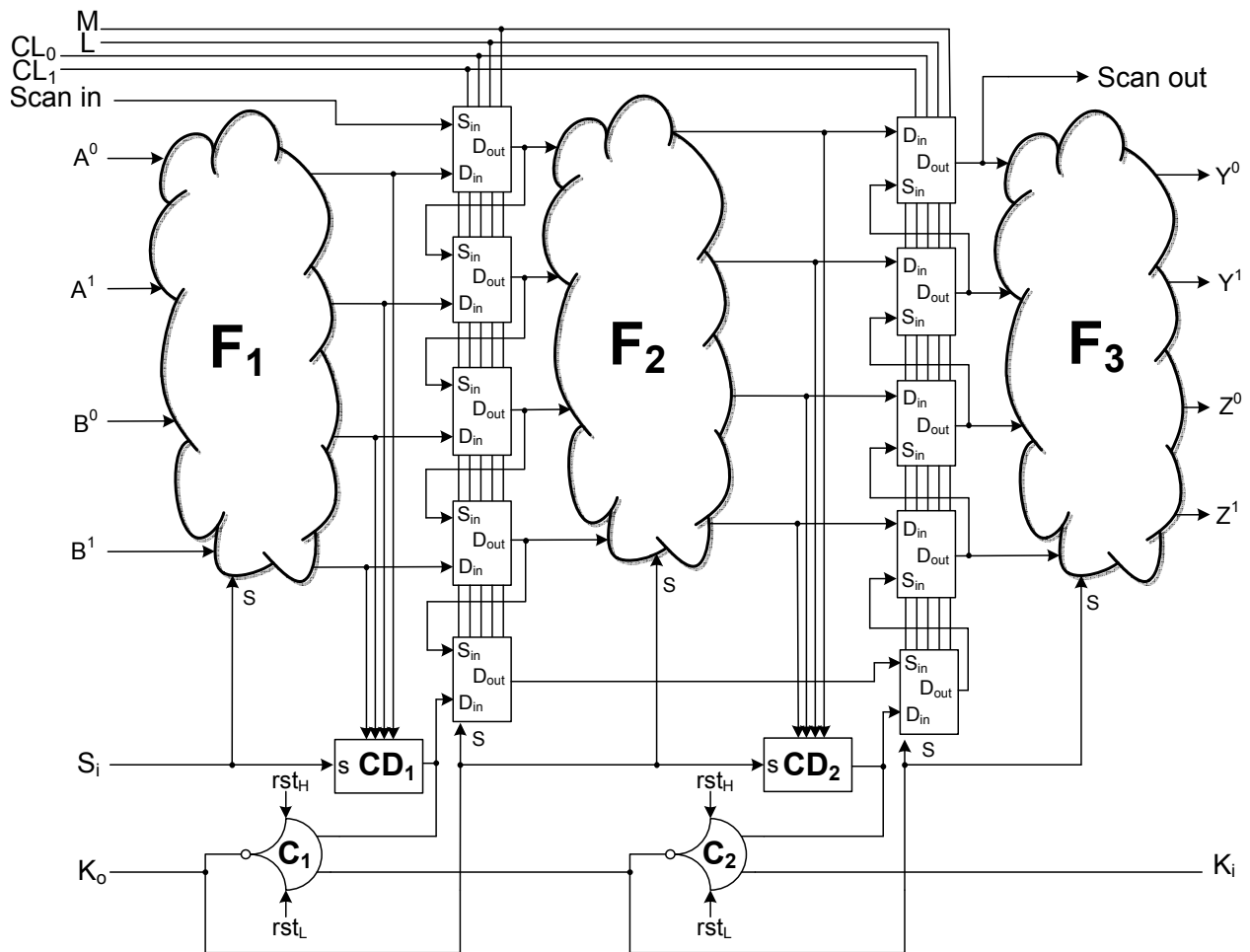
Figure 34 shows a typical SCL pipeline with two primary inputs ( $A$  and  $B$ ) and two primary outputs ( $Y$  and  $Z$ ) when registers are replaced with scan cells. Similar to a traditional scan chain design the scan cells form a long shift register in the test mode so that the test vectors can be shifted in and the results can be shifted out. There are, however, two major differences compared to the original SCL pipeline in Figure 29. First, the output of the completion detector is also fed to a scan cell. As discussed in section 4.3.2, stuck-at-0 faults on the inputs and output of gates within the completion detector block can be easily detected since they cause the pipeline to stall. However, detecting stuck-at-1 faults is not as easy since the sleep signal hides those faults. Therefore, in order to detect stuck-at-1 faults, a traditional ATPG method must be used similar to the case of combinational logic faults. Since the output of a completion detector is not readily available for



**Figure 33: SCL scan cell**

observation, adding an extra scan cell can solve the problem. The overhead associated with this extra scan cell is negligible considering that only a single additional scan cell is needed per pipeline stage. The second difference of the new SCL pipeline is adding an additional input signal,  $rst_L$ , to the completion C-element gates. This additional signal disables the sleep signal in test mode by forcing it to 0. Signal  $rst_H$ , however, does the same thing that  $rst$  does in the original pipeline, i.e., initializing the circuit to an all-NULL state by putting all the blocks into sleep mode.

Figure 35 shows the implementation of an SCL scan cell. The design is made of two D-latches, one of them being made out of the original SCL register by reconfiguring it using signals

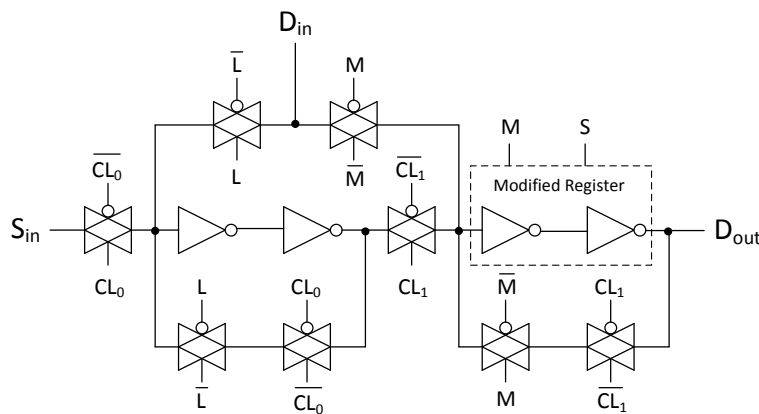


**Figure 34: SCL scan chain design**

$M$  and  $S$ . The modified version of the original SCL register for a single rail is shown in Figure 36. The modified version makes use of three additional transistors to cut the feedback path and make the first half of the register look like an inverter when  $M = 1$ . For the second half of the register to look like an inverter it is enough to just disable the sleep signal, i.e.  $S = 0$ . The whole implementation requires 32 transistors, however, in a dual-rail implementation each rail requires its own scan cell so a dual rail signal requires twice the number of transistors.

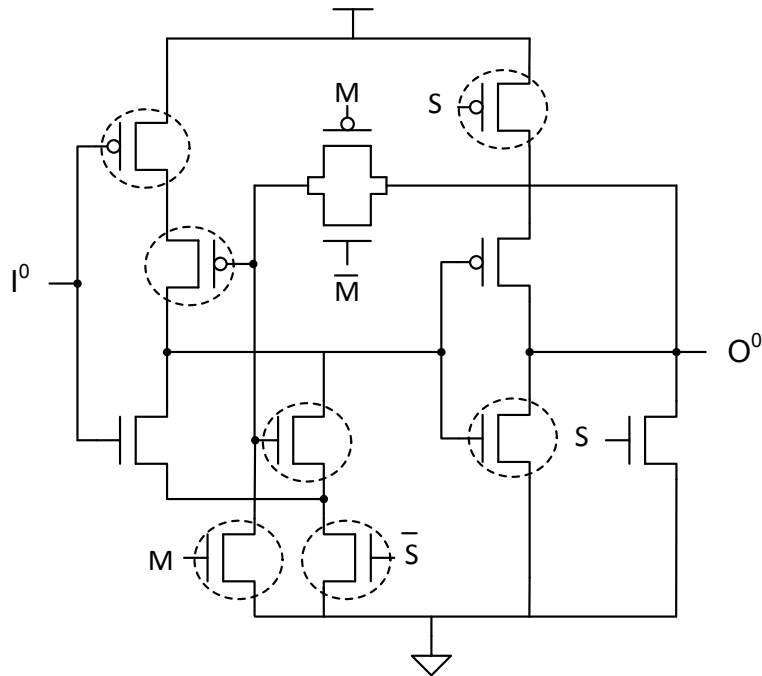
**Performing the Test** The testing procedure starts with testing the scan cells first. Similar to a synchronous scan test a *shift test* [48] can be initially used to detect most stuck-at faults associated with scan cells and ensure the correctness of shifting operation. The circuit is then placed in test mode by asserting  $M$  and  $rst_L$ . This will disable the sleep signals and set the scan cells in a shift register configuration. A *toggle sequence* [48], 00110011... , of length  $N + 4$  is then shifted in and out, where  $N$  is the number of scan cells. The toggle sequence generates various transitions on  $S_{in}$  and  $D_{out}$  signals to capture most of the stuck-at faults associated with the scan cells.

The shift test serves an additional purpose in the case of SCL pipeline test. The shift test



**Figure 35: SCL scan cell design**





**Figure 36: Modified version of the original SCL register for a single rail**

also detects stuck-at-1 faults on the sleep signal forks in the registers. Since every bit of the toggle sequence needs to pass all the scan cells, a stuck-at-1 fault on the sleep signal of even a single scan cell causes all the 1's in the toggle sequence to change to 0; therefore, the output sequence will be all 0's.

A similar approach can be used to detect all the stuck-at-0 faults on the sleep signal forks in the registers. This time an all-1 sequence, 1111..., is shifted into the registers. Then while in test mode,  $rst_H$  signal is asserted temporarily followed by asserting  $rst_L$  again. Asserting  $rst_H$  causes the sleep signal of all the scan cells to get asserted and asserting  $rst_L$  returns the circuit to test mode. If there are no stuck-at-0 faults on the sleep signal forks, all the registers must then get cleared, causing the output sequence to be all 0's. The presence of even a single 1 in the output sequence indicates the existence of a stuck-at-0 fault on a sleep signal fork. In fact, the number of 1's in the output sequence shows how many of the sleep signal forks are stuck-at-0.

At the end of the second shift test all the stuck-at faults on the sleep signal forks in the registers are tested. The second testing step is to apply a single {DATA, NULL} pair to the SCL pipeline in normal mode and let it propagate all the way from primary inputs to primary outputs. Based on theorem 1 this will detect all the stuck-at faults on the inputs and output of completion C-elements. Additionally, as discussed earlier this also detects all the stuck-at-0 faults on the gates within the completion detector blocks and all the stuck-at-1 faults on the sleep signal forks in the completion detector blocks.

At the end of the second testing step only stuck-at faults in the combinational logic blocks and stuck-at-1 faults on the gates in the completion detector blocks remain. According to theorem 2, by disabling the sleep signal the combinational logic blocks (as well as the completion detector blocks) become normal synchronous circuits. Therefore, traditional ATPG tools can be used to generate test patterns to detect the remaining faults. These test patterns can then be applied to the circuit according to the following protocol:

- put the circuit in test mode by asserting  $M$  and  $rst_L$  signals.
- shift in a test vector using the non-overlapping clock signals  $CL_0$  and  $CL_1$ .
- wait for sufficient amount of time for the test data to propagate through the circuit and then load the results by pulsing signal  $L$  followed by  $CL_1$  in a non-overlapping clock format.
- shift out the results using the non-overlapping clock signals  $CL_0$  and  $CL_1$  while shifting in the next test vector

#### 4.4 EXPERIMENTAL RESULTS

Table 23 shows the application of the testing methodology to some circuits. A 32-bit comparator, a 32-to-5 line priority encoder, and a  $32 \times 32$ -bit multiplier were used for case study. All the designs are made of two pipeline stages. The designs were initially synthesized by the UNCLE

**Table 23: Experimental results**

	<b>Comparator 32-bits</b>	<b>Encoder 32-to-5</b>	<b>Multiplier 32×32</b>
<b>Number of I/O pins</b>	138	80	260
<b>SCL gate count</b>	474	543	7630
<b>Transistor count w/o scan</b>	5212	5866	95728
<b>Transistor count with scan</b>	5698	6674	104632
<b>Length of scan chain</b>	20	34	382
<b>Area penalty due to scan</b>	9.3 %	13.7 %	9.3 %
<b>Total faults</b>	4030	4236	63108
<b>Detected Faults</b>	3948	3537	62075
<b>Redundant Faults</b>	78	669	473
<b>Undetected Faults</b>	4	30	560
<b>Fault Coverage</b>	98.0 %	83.5 %	98.4 %
<b>Test Coverage</b>	99.9 %	99.2 %	99.1 %
<b>Number of Test Vectors</b>	143	44	370
<b>CPU time</b>	0.8	1.4	54.1

tool [9] to SCL netlists and then the Synopsys Tetramax ATPG tool was used to generate test patterns for stuck-at faults according to the test protocol described in section 4.3.4. Table 23 is divided into two sections where in the first section information about the area of each design before and after the scan insertion is provided. This includes the original number of I/O pins, the original gate count, transistor count before and after the scan insertion, length of scan chain, and finally the area penalty due to scan insertion. Experimental results show that the area penalty due to scan insertion is in an acceptable range even in the case of the 32×32-bit multiplier with a long scan chain length. The area penalty is, however, a function of the number of registers and the complexity of combinational blocks and can be more or less for other circuits.

The second section of Table 23 provides information about the total number of stuck-at faults in each circuit, and the number of faults that are detected or not detected due to redundancy or other reasons. The total number of faults excludes the faults that are detected during shift test

or during the propagation of a {DATA, NULL} pair as described before. Table 23 also provides the fault coverage and test coverage for each case and shows the number of test vectors required to achieve that coverage as well as the CPU time taken to find those test vectors. The experiment results show that the methodology is very effective at achieving a high test coverage (more than 99%) in rather short time.

The relatively lower fault coverage especially in the case of the encoder is mainly due to the redundant faults that are caused by the redundant logic produced during synthesis. Ignoring these redundant faults increases the fault coverage to 99.9%, 99.2%, and 99.1% for the comparator, encoder, and multiplier, respectively. Table 23 also shows that the ratio of the undetected faults to the total number of faults is negligible (less than 1%). Additional experiments with Tetramax showed that most of these undetected faults can be detected by using a higher *abort limit* for the ATPG tool at the cost of higher CPU time.

## 4.5 CONCLUSION

In this chapter the problem of testing SCL circuits for stuck-at faults was investigated. The faults were initially divided into two separate categories: faults on logic gates, and faults on sleep signal forks. The faults within each category were then analyzed separately and a design for testability methodology was proposed based on the fault analysis. Finally, the proposed design for testability methodology was validated through experimental results and it was shown that the methodology provides a high test coverage (more than 99%) at the cost of an acceptable area overhead.

## 5 REFERENCES

- [1] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A designer's guide to asynchronous VLSI*. Cambridge University Press, 2010.
- [2] S. C. Smith and J. Di, "Designing asynchronous circuits using NULL convention logic (NCL)," *Synthesis Lectures on Digital Circuits and Systems*, vol. 4, no. 1, pp. 1–96, 2009.
- [3] K. M. Fant, *Logically determined design: clockless system design with NULL convention logic*. John Wiley & Sons, 2005.
- [4] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial HDL synthesis tools," in *Advanced Research in Asynchronous Circuits and Systems, 2000.(ASYNC 2000) Proceedings. Sixth International Symposium on*, pp. 114–125, IEEE, April 2000.
- [5] K. M. Fant and S. A. Brandt, "NULL convention logic: a complete and consistent logic for asynchronous digital circuit synthesis," in *Application Specific Systems, Architectures and Processors, 1996. ASAP 96. Proceedings of International Conference on*, pp. 261–273, IEEE, August 1996.
- [6] J. McCardle and D. Chester, "Measuring an asynchronous processor's power and noise," in *Synopsys Users Group (SNUG) Conference*, pp. 66–70, 2001.
- [7] C. Jeong and S. M. Nowick, "Technology mapping and cell merger for asynchronous threshold networks," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, pp. 659–672, April 2008.
- [8] B. Bhaskaran, *Automated Synthesis and Cycle Reduction Optimization for Asynchronous NULL Convention Circuits Using Industry-Standard CAD Tools*. PhD thesis, University of Missouri – Rolla, Rolla, MO, 2007.
- [9] R. B. Reese, S. C. Smith, and M. A. Thornton, "UNCLE—an RTL approach to asynchronous design," in *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, pp. 65–72, IEEE, May 2012.
- [10] F. Parsan, W. Al-Assadi, and S. Smith, "Gate mapping automation for asynchronous NULL convention logic circuits," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 99–112, 2014.
- [11] C. L. Seitz, "System timing," *Introduction to VLSI systems*, pp. 218–262, 1980.
- [12] K. van Berkel, "Beware the isochronic fork," *Integration, the VLSI journal*, vol. 13, pp. 103–128, June 1992.
- [13] T. Verhoeff, "Delay-insensitive codes—an overview," *Distributed Computing*, vol. 3, no. 1, pp. 1–8, 1988.
- [14] D. E. Muller, "Asynchronous logics and application to information processing," *Switching Theory in Space Technology*, pp. 289–297, 1963.

- [15] M. Shams, J. C. Ebergen, and M. I. Elmasry, "Modeling and comparing CMOS implementations of the c-element," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 6, pp. 563–567, December 1998.
- [16] G. E. Sobelman and K. Fant, "CMOS circuit design of threshold gates with hysteresis," in *Circuits and Systems, 1998. ISCAS'98. Proceedings of the 1998 IEEE International Symposium on*, vol. 2, pp. 61–64, IEEE, June 1998.
- [17] S. Yancey and S. C. Smith, "A differential design for c-elements and NCL gates," in *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pp. 632–635, IEEE, August 2010.
- [18] F. A. Parsan and S. C. Smith, "CMOS implementation of static threshold gates with hysteresis: A new approach," in *VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on*, pp. 41–45, IEEE, October 2012.
- [19] F. Parsan and S. Smith, "CMOS implementation comparison of NCL gates," in *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pp. 394–397, August 2012.
- [20] F. Parsan and S. Smith, "Cmos implementation of threshold gates with hysteresis," *VLSI-SoC: From Algorithms to Circuits and System-on-Chip Design*, vol. 418, pp. 196–216, 2013.
- [21] M. Shams, J. C. Ebergen, and M. I. Elmasry, "Optimizing CMOS implementations of the c-element," in *Computer Design: VLSI in Computers and Processors, 1997. ICCD'97. Proceedings., 1997 IEEE International Conference on*, pp. 700–705, IEEE, October 1997.
- [22] L. Ding and P. Mazumder, "On circuit techniques to improve noise immunity of CMOS dynamic logic," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 9, pp. 910–925, 2004.
- [23] L. Heller, W. Griffin, J. Davis, and N. Thoma, "Cascode voltage switch logic: A differential CMOS logic family," in *Solid-State Circuits Conference. Digest of Technical Papers. 1984 IEEE International*, vol. 27, pp. 16–17, IEEE, February 1984.
- [24] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguson, "Delay-insensitive gate-level pipelining," *Integration, the VLSI journal*, vol. 30, pp. 103–131, October 2001.
- [25] I. David, R. Ginosar, and M. Yoeli, "An efficient implementation of boolean functions as self-timed circuits," *Computers, IEEE Transactions on*, vol. 41, pp. 2–11, January 1992.
- [26] J. Sparsø and J. Staunstrup, "Delay-insensitive multi-ring structures," *Integration, the VLSI journal*, vol. 15, pp. 313–340, October 1993.
- [27] C. D. Nielsen, "Evaluation of function blocks for asynchronous design," in *Proceedings of the conference on European design automation*, pp. 454–459, IEEE Computer Society Press, September 1994.

- [28] S. C. Smith, R. F. DeMara, J. S. Yuan, D. Ferguson, and D. Lamb, "Optimization of NULL convention self-timed circuits," *INTEGRATION, the VLSI journal*, vol. 37, pp. 135–165, August 2004.
- [29] C. Jeong and S. M. Nowick, "Optimization of robust asynchronous circuits by local input completeness relaxation," in *Design Automation Conference, 2007. ASP-DAC'07. Asia and South Pacific*, pp. 622–627, IEEE, January 2007.
- [30] C. Jeong and S. M. Nowick, "Block-level relaxation for timing-robust asynchronous circuits based on eager evaluation," in *Asynchronous Circuits and Systems, 2008. ASYNC'08. 14th IEEE International Symposium on*, pp. 95–104, IEEE, April 2008.
- [31] A. Bardsley and D. Edwards, *Balsa: An asynchronous circuit synthesis system*. University of Manchester, 1998.
- [32] D. Edwards, A. Bardsley, L. Janin, L. Plana, and W. Toms, "Balsa: A tutorial guide.," *The University of Manchester*, 2006.
- [33] C. D. Godsil, G. Royle, and C. Godsil, *Algebraic graph theory*, vol. 8. Springer New York, 2001.
- [34] J. M. Robson, "Algorithms for maximum independent sets," *Journal of Algorithms*, vol. 7, no. 3, pp. 425–440, 1986.
- [35] F. V. Fomin, F. Grandoni, and D. Kratsch, "A measure & conquer approach for the analysis of exact algorithms," *Journal of the ACM (JACM)*, vol. 56, no. 5, p. 25, 2009.
- [36] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen, *Introduction to algorithms*. The MIT press, 2001.
- [37] S. K. Bandapati, S. C. Smith, and M. Choi, "Design and characterization of NULL convention self-timed multipliers," *Design & Test of Computers, IEEE*, vol. 20, pp. 26–36, December 2003.
- [38] B. Hollosi, M. Barlow, G. Fu, C. Lee, J. Di, S. C. Smith, H. A. Mantooth, and M. Schupbach, "Delay-insensitive asynchronous ALU for cryogenic temperature environments," in *Circuits and Systems, 2008. MWSCAS 2008. 51st Midwest Symposium on*, pp. 322–325, IEEE, August 2008.
- [39] J. Di and S. C. Smith, "Ultra-low power multi-threshold asynchronous circuit design," June 2012. US Patent 8,207,758.
- [40] S. C. Smith, "Speedup of self-timed digital systems using early completion," in *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pp. 98–104, IEEE, April 2002.
- [41] L. Zhou, *Ultra-low power and radiation hardened asynchronous circuit design*. PhD thesis, University of Arkansas, Fayetteville, AR, 2012.

- [42] F. Parsan, J. Zhao, and S. Smith, “Scl design of a pipelined 8051 alu,” in *Circuits and Systems (MWSCAS), 2014 IEEE 57th International Midwest Symposium on*, pp. 885–888, IEEE, Aug 2014.
- [43] L. Zhou, S. C. Smith, and J. Di, “Bit-wise MTNCL: An ultra-low power bit-wise pipelined asynchronous circuit design methodology,” in *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pp. 217–220, IEEE, August 2010.
- [44] V. Satagopan, B. Bhaskaran, W. K. Al-Assadi, S. C. Smith, and S. Kakarla, “DFT techniques and automation for asynchronous NULL conventional logic circuits,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 15, pp. 1155–1159, October 2007.
- [45] A. Kondratyev, L. Sorensen, and A. Streich, “Testing of asynchronous designs by ”inappropriate” means. synchronous approach,” in *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pp. 171–180, April 2002.
- [46] D. R. Schertz and G. Metze, “On the design of multiple fault diagnosable networks,” *Computers, IEEE Transactions on*, vol. C-20, no. 11, pp. 1361–1364, 1971.
- [47] E. B. Eichelberger and T. W. Williams, “A logic design structure for lsi testability,” in *Proceedings of the 14th ACM/IEEE design automation conference*, pp. 462–468, IEEE Press, 1977.
- [48] M. Bushnell and V. D. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*, vol. 17. Springer, 2000.