STARS

University of Central Florida
STARS

Retrospective Theses and Dissertations

2001

Gate and Throughput Optimizations for NULL Convention Self-timed Digital Circuits

Scott Christopher Smith
University of Central Florida, smithsco@uark.edu

Find similar works at: https://stars.library.ucf.edu/rtd
University of Central Florida Libraries http://library.ucf.edu

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation
Smith, Scott Christopher, "Gate and Throughput Optimizations for NULL Convention Self-timed Digital Circuits" (2001). *Retrospective Theses and Dissertations*. 1359.
https://stars.library.ucf.edu/rtd/1359

University of Central Florida
STARS
Showcase of Text, Archives, Research & Scholarship

GATE AND THROUGHPUT OPTIMIZATIONS FOR NULL CONVENTION SELF-TIMED DIGITAL CIRCUITS

By

Scott Christopher Smith

2001

UCF

GATE AND THROUGHPUT OPTIMIZATIONS FOR NULL CONVENTION SELF-TIMED DIGITAL CIRCUITS

by

Scott Christopher Smith

MSEE, University of Missouri-Columbia, 1998
BSEE, University of Missouri-Columbia, 1996
BSCompE, University of Missouri-Columbia, 1996

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Engineering
in the field of Computer Architecture and Digital Systems
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2001

Major Professor: Dr. Ronald DeMara

ABSTRACT

NULL Convention Logic (NCL) provides an asynchronous design methodology employing dual-rail signals, quad-rail signals, or other Mutually Exclusive Assertion Groups (MEAGs) to incorporate data and control information into one mixed path. In NCL, the control is inherently present with each datum, so there is no need for worse-case delay analysis and control path delay matching. This dissertation focuses on optimization methods for NCL circuits, specifically addressing three related architectural areas of NCL design.

First, a design method for optimizing NCL circuits is developed. The method utilizes conventional Boolean minimization followed by table-driven gate substitutions. It is applied to design time and space optimal fundamental logic functions, a time and space optimal full adder, and time, transistor count, and power optimal up-counter circuits. The method is applicable when composing logic functions where each gate is a state-holding element; and can produce delay-insensitive circuits requiring less area and fewer gate delays than alternative gate-level approaches requiring full minterm generation.

Second, a pipelining method for producing throughput optimal NCL systems is developed. A relationship between the number of gate delays per stage and the worse-case throughput for a pipeline as a whole is derived. The method then uses this relationship to minimize a pipeline's worse-case throughput by partitioning the NCL

combinational circuitry through the addition of asynchronous registers. The method is applied to design a maximum throughput unsigned multiplier, which yields a speedup of 2.25 over the non-pipelined version, while maintaining delay-insensitivity.

Third, a technique to mitigate the impact of the NULL cycle is developed. The technique further increases the maximum attainable throughput of a NCL system by reducing inherent overheads associated with an integrated data and control path. This technique is applied to a non-pipelined 4-bit by 4-bit unsigned multiplier to yield a speedup of 1.61 over the standalone version.

Finally, these techniques are applied to design a 72+32×32 multiply and accumulate (MAC) unit, which outperforms other delay-insensitive/self-timed MACs in the literature. It also performs conditional rounding, scaling, and saturation of the output, whereas the others do not; thus further distinguishing it from the previous work. The methods developed facilitate speed, transistor count, and power tradeoffs using approaches that are readily automatable.

ACKNOWLEDGEMENTS

I would like to thank Theseus Logic, Inc. for their financial support and the opportunity to work with such novel and exciting technology. I would like to thank Dr. DeMara for his technical and editorial advice that has helped shape this work. I would like to thank the committee members who have taken the time to review and comment on this dissertation. I would also like to thank the state of Florida for the fellowships provided that have allowed me the opportunity to pursue this degree. But most of all I would like to thank my loving wife Tamara, for her patience and support in my continued education.

TABLE OF CONTENTS

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER 1. INTRODUCTION 1

1.1. Objective 1

1.2. History and Benefits of NCL 1

1.3. Research Challenges 5

1.4. Dissertation Overview 6

CHAPTER 2. PREVIOUS WORK 8

2.1. Overview of Asynchronous Methods 9

2.1.1. Gate-Level Delay-Insensitive Methods 11

2.1.2. Transistor-Level Delay-Insensitive Methods 12

2.2. Overview of NCL 12

2.2.1. Delay-Insensitivity 13

2.2.2. Logic Gates and Functional Blocks 15

2.2.3. Completeness of Input 20

2.2.4. Observability 22

2.2.5. NCL Registration 23

2.2.6. NCL Completion 29

CHAPTER 3. THRESHOLD COMBINATIONAL REDUCTION METHOD............31

3.1. Chapter Outline............32

3.2. TCR Method Definition............32

3.2.1. Method 1: Incomplete Functions............34

3.2.2. Method 2: Dual-Rail Optimizations............34

3.2.3. Method 3: Quad-Rail Optimizations............36

3.2.4. Performance Assessment............37

3.3. Application to Input-Complete Fundamental Logic Functions............38

3.4. Application to Full Adder............40

3.5. Application to Up-Counter............47

3.5.1. Method 1: Incomplete Functions............51

3.5.2. Method 2: Dual-Rail Encoding Optimizations............53

3.5.3. Method 3: Quad-Rail Encoding Optimizations............57

3.5.4. Other MEAG Optimizations............61

3.5.5. Up-Counter Performance Summary............64

CHAPTER 4. GATE-LEVEL PIPELINING OPTIMIZATIONS............66

4.1. Chapter Outline............67

4.2. Previous Work............67

4.2.1. Relation of NCL to Previous Work............70

4.3. Method Definition............72

4.3.1. Throughput Derivation............75

4.3.1.1. Idealized Completion Circuitry............77

4.3.1.2. Non-Zero Delay Completion Circuitry 79
4.3.2. Bit-Wise Completion 82
4.4. Application to Unsigned Multiplier 85
4.4.1. Pipelined Multipliers with Full-Word Completion 87
4.4.2. Summary of Multiplier Designs using Full-Word Completion 93
4.4.3. Applying Bit-Wise Completion 93
4.5. Conclusion 94
CHAPTER 5. NULL CYCLE REDUCTION TECHNIQUE 96
5.1. Introduction 96
5.2. NULL Cycle Reduction 97
5.2.1. Demultiplexer 99
5.2.2. Completion Detection Circuitry 100
5.2.3. Sequencer #1 100
5.2.4. Multiplexer 101
5.2.5. Sequencer #2 102
5.3. Simulation Results 103
CHAPTER 6. NCL MULTIPLY AND ACCUMULATE UNIT 106
6.1. Introduction 107
6.2. Previous Work 108
6.3. Self-Timed MAC Design Methods 109
6.3.1. Non-Pipelined Modified Baugh-Wooley MAC 111
6.3.1.1. Operation 111

6.3.1.2. Design Optimizations....115

6.3.1.3. Average Cycle Time Determination....117

6.3.2. Non-Pipelined Modified Booth2 MAC....118

6.3.2.1. Operation....118

6.3.2.2. Design Optimizations....120

6.3.2.3. Average Cycle Time Determination....120

6.3.3. Pipelined Modified Baugh-Wooley MAC....120

6.3.3.1. Operation....120

6.3.3.2. Throughput Maximization....122

6.3.4. Pipelined Modified Booth2 MAC....126

6.3.4.1. Operation....126

6.3.4.2. Throughput Maximization....128

6.3.5. Simulation Results....128

6.4. Carry-Propagate Adder Comparison....129

6.5. Gate Requirements for Proposed Designs....132

6.5.1. Modified Baugh-Wooley MAC....133

6.5.2. Modified Booth2 MAC....133

6.5.3. Array MAC....134

6.5.4. Modified Booth3 MAC....135

6.5.5. Modified Booth4 MAC....136

6.5.6. Combinational 2-Bit × 2-Bit MAC....137

6.5.7. Combinational 2-Bit × 3-Bit MAC....138

6.5.8. Combinational 2-Bit × 4-Bit MAC 139

6.5.9. Combinational 3-Bit × 3-Bit MAC 140

6.5.10. Quad-Rail MACs 142

6.6. Conclusion 142

CHAPTER 7. Conclusion 145

7.1. Summary 145

7.2. Future Work 147

LIST OF REFERENCES 150

LIST OF TABLES

I. Attributes of clocked Boolean and asynchronous methods 3

II. Attributes of self-timed methods 10

III. 27 NCL macros 36

IV. Performance characteristics of input-complete NCL logic functions....................40

V. Full adder using various delay-insensitive methods 43

VI. Delay-insensitive methods for *f(a, b, c, d) = a • b' • c • d'*..................................47

VII. Alternate designs for NCL up-counter increment circuit 65

VIII. Discrete timing chart for the idealized NCL cycle 77

IX. Discrete timing chart for the general NCL cycle...80

X. Stage delay and throughput for various multiplier designs 93

XI. Sequencer output...101

XII. NCR vs. pipelining for multiplier application 104

XIII. Saturation table 115

XIV. Propagation delay and gate count for 4-bit adders...131

XV. Algorithm, technology, and cycle time for various self-timed MACs 144

LIST OF FIGURES

1. Symbolic incompleteness of a Boolean AND gate....................................13

2. NCL AND function: Z = X • Y and associated waveforms 15

3. THmn threshold gate...16

4. Static CMOS implementation of a TH23 gate 16

5. NULL flowing through combinational circuitry...18

6. Completion detection of NULL output..19

7. DATA flowing through combinational circuitry 19

8. Completion detection of DATA output 19

9. DATA-to-DATA cycle time (T_{DD}) 20

10. Incomplete AND function: Z = X • Y 21

11. Conventional input-complete AND function: Z = X • Y 21

12. Incorrect XOR function: Z = X ⊕ Y (orphans may propagate through a gate)...........22

13. Correct XOR function: Z = X ⊕ Y (orphans may not propagate through any gate)...23

14. n-bit dual-rail registration 24

15. Initial register state..26

16. Register state after traversing combinational circuitry 27

17. Register state after NULL wavefront passes through downstream register 27

18. Register state after DATA wavefront passes through current register 27

19. Register state after NULL wavefront passes through upstream register 28

20. Static register state 28

21. Single-bit dual-rail register 29

22. Single-signal quad-rail register 30

23. N-bit completion component 30

24. TCR design flow 33

25. Conventional input-complete OR function: Z = X + Y 39

26. Conventional input-complete XOR function: Z = X ⊕ Y 39

27. Truth table for full adder 41

28. K-map for C_o output of full adder 41

29. K-map for S output of full adder 41

30. Optimized NCL full adder 41

31. Full adder using Anantharaman's approach or DIMS 43

32. Full adder using Seitz's approach 44

33. Full adder using David's approach 45

34. Full adder using Singh's approach 46

35. 4-bit up-counter block diagram 48

36. Up-counter with three-register feedback 48

37. Dual-rail 4-bit counter waveforms 49

38. 16-rail MEAG 4-bit counter waveforms 50

39. Quad-rail 4-bit counter waveforms 51

40. Boolean increment circuit 52

41. Increment circuit using incomplete AND functions 52

42. Increment circuit using dual-rail reduced minterm expressions 54

43. Increment circuit using dual-rail factored minterm expressions 55

44. Dual-rail increment circuit using complex gates 56

45. Karnaugh maps for quad-rail counter 57

46. Increment circuit using quad-rail reduced minterm expressions 58

47. Increment circuit using quad-rail factored minterm expressions 60

48. Quad-rail increment circuit using complex gates 61

49. 16-rail MEAG increment circuit 62

50. 16-rail MEAG register 63

51. Two-phase handshaking protocol 69

52. Four-phase handshaking protocol 69

53. GLP design flow 73

54. Sub-cycles of the NCL cycle 76

55. Pipeline showing NCL sub-cycle times 76

56. Full-word completion 84

57. Bit-wise completion 84

58. 4×4 multiplier block diagram 85

59. Non-pipelined, 1-stage 4×4 multiplier using full-word completion 86

60. Half adder 87

61. GEN_S7 component 87

62. 2-stage 4×4 multiplier using full-word completion 89

63. 3-stage 4×4 multiplier using full-word completion 90

64. 4-stage 4×4 multiplier using full-word completion 91

65. 7-stage 4×4 multiplier using full-word completion 92

66. 7-stage 4×4 multiplier using bit-wise completion 95

67. NCR architecture 98

68. 1-bit Demultiplexer 99

69. Sequence generator 101

70. 1-bit Multiplexer 102

71. NCL pipeline with one slow stage 105

72. MAC block diagram 111

73. Taxonomy of 72+32×32 MAC 111

74. Non-pipelined Modified Baugh-Wooley MAC 113

75. Output divisions for up-scaling, no scaling, and down-scaling 114

76. Non-pipelined Modified Booth2 MAC 119

77. Pipelined Modified Baugh-Wooley MAC 121

78. Pipelined Modified Booth2 MAC 127

79. 4-bit carry-lookahead adder 130

1.0 INTRODUCTION

1.1 Objective

This Ph.D. dissertation is intended to familiarize the reader with the syntax and semantics of NULL Convention Logic (NCL), to develop NCL design methods and optimization techniques, and to discuss analytical and experimental results. The main focus will be on architectural aspects of NCL as discussed at the gate level.

1.2 History and Benefits of NCL

Various design aspects of NCL were patented by Karl Fant and Scott Brandt in April of 1994 [1]. Acknowledging that clocked circuits unnecessarily restricted execution flow, consumed power proportional to the operating frequency, occupied significant device area for the clock tree, and greatly complicated the design process, they sought a clockless design approach. But eliminating clocks as in traditional asynchronous design presented race conditions and made timing optimizations like pipelining difficult. By eliminating clocks but retaining control information in the datapath, NCL aims at designing VLSI devices with greater ease, with a reduced power budget, lower electromagnetic interface effects, and reduced noise margins.

Karl Fant founded Theseus Logic, Inc., which began operations in Minnesota in January of 1996, to develop NCL-based Application Specific Integrated Circuits (ASICs)

and "soft cores" for electronics manufacturers. The company has demonstrated the viability of NCL technology through government programs with Honeywell, Lockheed Martin, the Defense Advanced Research Projects Agency (DARPA), the Ballistic Missile Defense Organization (BMDO), the US ARMY Communication Electronics Command (CECOM), and the National Security Agency (NSA). A privately held company, Theseus is now headquartered in Orlando, Florida and also has a research and development office in Sunnyvale, California.

In August 1999, Theseus and the University of Central Florida were awarded a state–funded grant for a joint research project involving NCL ASIC design and development of formal design methods for NCL. In October 1999 Theseus formed a strategic technology alliance with Motorola's Semiconductor Products Sector to jointly implement NCL versions of various Motorola microcontrollers. And in September 2000 Theseus formed a strategic technology alliance with Synopsys for development of NCL-based design tools. Many potential applications from mobile, handheld low-power DSP devices to general purpose CPUs lie ahead.

Table I lists the advantages of asynchronous design, both bounded-delay and delay-insensitive models, over clocked Boolean design. It shows that clocked Boolean design necessitates a global clock, where asynchronous design does not; and that only delay-insensitive methods have no glitch power, deliver average-case verses worse-case performance, and provide for ease of design reuse. Table I also lists that power, noise, and EMI are disadvantages for clocked Boolean circuits, but are advantages for their asynchronous counterparts, as detailed below.

Table I. Attributes of clocked Boolean and asynchronous methods.

Design Paradigm	Global Clock	Glitch Power	Performance Average-Case	Performance Worse-Case	Reuse Ease	Power	Noise	EMI
Clocked Boolean	Y	Y	N	Y	N	D	D	D
Bounded-Delay	N	Y	N	Y	N	A	A	A
Delay-Insensitive	N	N	Y	N	Y	A	A	A

Traditional clocked Boolean circuits suffer from the layout nightmare of clock distribution and require high power surges at the clock edge, when switching is most prevalent. Synchronous circuits also cannot operate at their maximum potential due to clock skew. These trends have led to a large revival of interest in the asynchronous approach.

In asynchronous design approaches each component in the system is not controlled by a clock signal. Thus, timing design margins are not required to compensate for clock skew. An asynchronous design theoretically should allow data to flow through a circuit at the maximum rate of the underlying switching technology being used. As the required inputs arrive, a function should be executed and its results sent to the required destination(s).

Nonetheless, traditional asynchronous design techniques have drawbacks of their own. An asynchronous circuit is traditionally designed as having a datapath and a control path. Since there is no clock to synchronize these two paths, there must be extensive timing analysis performed in order to determine the worse-case delay in the datapath. This delay must then be matched in the control path in order to synchronize the two paths without the use of a clock. This method of asynchronous circuit design is classified as

bounded-delay. Both clocked Boolean and bounded-delay designs suffer from the problem of limiting the maximum operating frequency based on the worse-case delay in the datapath. Bounded-delay design also alleviates the complex task of clock distribution, but it introduces another complex task of determining the worse-case datapath delay and matching this delay in the control path. An important benefit of NCL is asynchronous execution that is completely delay-insensitive, assuming that wire forks are isochronic [2, 3]. When designing in NCL there is no need for worse-case delay analysis and delay matching, which makes the NCL design process significantly less complex than traditional asynchronous design.

NCL on the other hand, allows a system to run at its maximum frequency regardless of the input. For inputs which traverse a path with minimal delay, the output will arrive much faster than for inputs which traverse a longer delay path. This property allows a NCL circuit to potentially operate faster than a traditional Boolean asynchronous design. NCL circuits are also much more adaptive, and facilitate easier reuse than Boolean asynchronous circuits, since timing analysis is unnecessary due to NCL's delay-insensitivity.

As the trend towards higher clock frequency continues, power consumption, noise, and electromagnetic interference (EMI) of synchronous designs increase significantly. PCs are becoming more widespread and consume an increasingly substantial percentage of the world's electrical power. With the absence of a clock, NCL systems promise to reduce power consumption, noise, and EMI. NCL circuits, designed using CMOS, also exhibit an inherent idle behavior since they only switch when useful

work is being performed, unlike clocked Boolean circuits that switch every clock pulse. NCL circuits adhere to monotonic transitions between DATA and NULL, so there is no glitching, unlike clocked Boolean circuits that produce substantial glitch power. NCL systems also distribute the demand for power over time and area, reducing the occurrence of hot spots, system noise, and peak power demand, unlike clocked Boolean circuits where all circuitry switches simultaneously at the clock edge. Furthermore, NCL systems are very tolerant of power supply variations such that cheaper power supplies can be used and voltage can be reduced dramatically to meet performance criterion while reducing power consumption. Therefore, a very fast NCL circuit can be run at a lower voltage to reduce power consumption when high performance is not required.

The initial version of Motorola STAR08 processor using NCL technology shows a 40% reduction in power and a 10 dB reduction in noise over its clocked Boolean counterpart, while operating at a comparable frequency. Since NCL circuits have been demonstrated to consume significantly less power than clocked Boolean designs, NCL has a promising future in the field of mobile electronics, where power consumption is a major design consideration.

1.3 Research Challenges

This dissertation focuses on three architectural areas of NCL, all related to circuit design and optimization. Since NCL is still conceptually young, there is no current formal method for designing optimal NCL circuits. NCL differs significantly from Boolean logic; so traditional Boolean techniques for circuit simplification cannot be

applied to NCL circuits without major modifications. Thus, the first goal is to devise a new formal method for NCL circuit simplification, such that optimal designs are readily obtained.

The unique structure of NCL lends itself to pipelining, even though a clock is not present. Since there is no clock in NCL to synchronize pipeline stages, the design of a NCL pipeline will be significantly different than a Boolean pipeline design. A related need is to develop a means for determining the maximum number of gate delays per stage to yield the maximum attainable throughput when pipelining a given design. Thus, the second goal is to develop a formal method for designing throughput optimal NCL systems.

The NULL cycle accounts for approximately half of the cycle time of a NCL circuit, therefore reducing the system's maximum attainable throughput by a factor of two. Thus, the third goal is to devise a technique to reduce the NULL cycle, further increasing system performance. This further speedup may be essential for especially time critical circuits.

1.4 Dissertation Overview

This dissertation is organized into seven chapters. Chapter 2 presents previous work and contains an in-depth discussion of fundamental NCL terminology, concepts, and components, which will provide the notation and basis for the rest of the dissertation. In Chapter 3, a formal method for designing different types of optimal combinational, simplified NCL circuits is developed. This method is then tested on the design of

fundamental logic functions, a full adder, and a 4-bit counter, with simulation times, gate counts, and transistor counts included. In Chapter 4, a formal method for producing pipelined designs, which yield the maximum attainable throughput, is devised. This method is tested on the design of a 4-bit by 4-bit multiplier, and includes comprehensive simulation times and pipeline stage information. Chapter 5 develops a technique for reduction of the NULL cycle, and applies it to a non-pipelined 4-bit by 4-bit multiplier. Chapter 6 details the design of a throughput and area optimal 72+32×32 MAC. Chapter 7 highlights the contributions of this dissertation and provides direction for future research.

2.0 PREVIOUS WORK

For the last two decades the focus of digital design has been primarily on synchronous, clocked architectures. However, as clock rates have significantly increased while feature size has decreased, clock skew has become a major problem. High performance chips must dedicate increasingly larger portions of their area for clock drivers to achieve acceptable skew, causing these chips to dissipate increasingly higher power. As these trends continue, the clock is becoming more and more difficult to manage. This has caused renewed interest in asynchronous digital design.

NULL Convention Logic (NCL) offers a delay-insensitive logic paradigm where control is inherent with each datum. NCL follows the so-called "weak conditions" of Seitz's delay-insensitive signaling scheme [4]. As with other delay-insensitive logic methods discussed herein, the NCL paradigm assumes that forks in wires are isochronic [2, 3]. The origins of various aspects of the paradigm, including the NULL (or spacer) logic state from which NCL derives its name, can be traced back to Muller's work on speed-independent circuits in the 1950s and 1960s [5].

Earlier work by Seitz presents an extensive discussion of delay-insensitive logic, illustrating its advantages over traditional clocked logic, and includes one approach to designing such circuits [2]. Some other methods of designing delay-insensitive circuits are detailed in [6, 7, 8, 9]. These techniques concentrate on developing circuits from a

standardized set of gates, while other techniques [10, 11] emphasize formal logic methods that directly yield designs at the transistor-level. In the application of CMOS technology, processors implemented with this type of signaling scheme include the MIPS R3000 [12] and another at Caltech [13], the FLYSIG processor at the University of Paderborn [14], the MSL16A at the Chinese University of Hong Kong [15], and the TITAC processor at the Toyko Institute of Technology [16]. NCL differs from the above mentioned methods in that they only utilize one type of state-holding gate, the *C-element* [5]. On the other hand, all NCL gates are state-holding. Thus, NCL optimization methods can be considered as a subclass of the techniques for developing delay-insensitive circuits using a pre-defined set of more complex components with built-in hysteresis behavior. In functions that do not require full minterm generation, such attributes may allow optimizations that produce smaller, faster delay-insensitive combinational circuits.

2.1 Overview of Asynchronous Methods

Asynchronous circuits fall into two main categories: delay-insensitive and bounded-delay models. Paradigms, like NCL, assume delays in both logic elements and interconnects to be unbounded, although they assume that wire forks are isochronic. This implies the ability to operate in the presence of indefinite arrival times for the reception of inputs. Completion detection of the output signals allows for handshaking to control input wavefronts. On the other hand, bounded-delay models such as *Huffman circuits* [17], *burst-mode circuits* [18], and *micropipelines* [19] assume that delays in both gates and wires are bounded. Delays are added based on worse-case scenarios to avoid hazard

conditions. This leads to extensive timing analysis of worse-case behavior to ensure correct circuit operation. Since NCL exhibits neither of these characteristics, bounded-delay models are not addressed further.

Table II summarizes the attributes of various self-timed methods. It lists that only micropipelines add explicit delays, while the other methods rely on completion detection; and that only micropipelines exhibit worse-case performance, verses the average-case performance of the other methods. Table II also shows that only Seitz's, Anantharaman's, and DIMS approaches require full minterm generation, while all approaches use C-elements exclusively for their state-holding gates, except for micropipelines that do not require any state-holding elements, NCL that utilizes numerous state-holding gates, and Martin's method that does not use a standardized set of gates but instead develops each element at the transistor level, as detailed below.

Table II. Attributes of self-timed methods.

Self-Timed Method	Explicit Delays Inserted	Completion Detection	Full Minterm Generation Required	State-Holding Gates	Performance	
					Average-Case	Worse-Case
Micropipelines	Y	N	N	None	N	Y
Seitz	N	Y	Y	C-elements	Y	N
DIMS	N	Y	Y	C-elements	Y	N
Anantharaman	N	Y	Y	C-elements	Y	N
Singh	N	Y	N	C-elements	Y	N
David	N	Y	N	C-elements	Y	N
NCL	N	Y	N	Numerous	Y	N
Martin	N	Y	N	N/A	Y	N

2.1.1 Gate-Level Delay-Insensitive Methods

Most gate-level delay-insensitive methods combine *C-elements* [5] with Boolean gates for circuit construction. A C-element behaves as follows: when all inputs assume the same value then the output assumes this value, otherwise the output does not change. Seitz's method [2] employs a sum of products network using AND and OR gates, combined with a network to OR both rails of all inputs together. The output of the OR network is then combined with the sum of products outputs, using C-elements, to produce the circuit outputs. DIMS [9] and Anantharaman's approach [7] are similar to each other in that each produces a sum of products circuit using OR gates and C-elements, instead of AND gates. Singh's method [8] combines small self-timed logic functions to produce the desired functionality, while David's method [6] produces self-timed circuits with *n* inputs and *m* outputs, composed of four subnets, *ORN*, *CEN*, *DRN*, and *OUTN*. ORN consists of *n* 2-input OR gates, which OR together both rails of each dual-rail input. CEN is an n-input C-structure, which is equivalent to an n-input C-element, whose inputs are the *n* outputs from ORN. DRN is a monotonic implementation of each rail of the dual-rail output(s). OUTN produces the circuit output and consists of 2m 2-input C-elements, each with the output of CEN as one input, and an output from DRN as the other input. Seitz's method, Anantharaman's approach, and DIMS require the generation of all minterms to implement a function, where a minterm is defined as the logical AND, or product, of input signals. While Singh's and David's methods do not require full minterm generation, they rely solely on C-elements for delay-insensitivity.

Since Seitz's and Anantharaman's approaches, along with DIMS, require the generation of all minterms, no optimization is possible. However, Singh's and David's approaches allow for some Boolean optimization to be performed, but they may not facilitate the same potential for optimization provided by NCL's many state-holding gates, as will be shown in Chapter 3.

2.1.2 Transistor-Level Delay-Insensitive Methods

Other delay-insensitive methods such as Martin's [30] consist of constructing transistor-optimized circuits from their Boolean equations through formal logic transformations. Most of the resulting transistor level circuits are state-holding. However, since these methods do not target a specific set of gates, they are not directly comparable to gate-level delay-insensitive methods, including NCL.

2.2 Overview of NCL

NCL gates are a special case of the logical operators or gates available in digital VLSI circuit design [20]. Such an operator consists of a set condition and a reset condition that the environment must ensure are not both satisfied at the same time. If neither condition is satisfied then the operator maintains its current state. A number of NCL-based designs have been commercially developed by Theseus Logic, Inc., which has formed strategic alliances with Motorola for microcontroller design and Synopsys for NCL-based design tool development.

2.2.1 Delay-Insensitivity

NCL uses symbolic completeness of expression [21] to achieve self-timed behavior. A symbolically complete expression is defined as an expression that only depends on the relationships of the symbols present in the expression without a reference to the time of evaluation. Traditional Boolean logic is not symbolically complete; the output of a Boolean gate is only valid when referenced with time. For example, assume it takes 1 ns for output Z of an AND gate to become valid once its inputs X and Y have arrived. As shown in Figure 1, suppose $X = 1$, $Y = 0$, and $Z = 0$, initially. If Y changes to 1, Z will change to 1 after 1 ns; so Z is not valid from the time Y changes until 1 ns later. Therefore output Z not only depends on the inputs X and Y, but time must also be referenced in order to determine the validity of Z. This can be critical when Z is used as an input to another circuit.

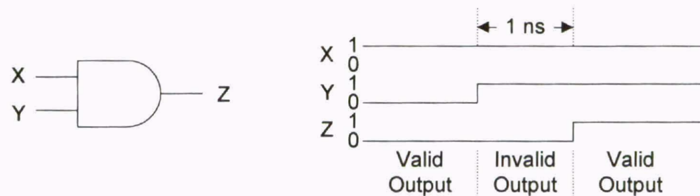

Figure 1. Symbolic incompleteness of a Boolean AND gate.

In particular, dual-rail signals, quad-rail signals, or other *Mutually Exclusive Assertion Groups* (MEAGs) can be used to incorporate data and control information into one mixed signal path to eliminate time reference [22]. A dual-rail signal, D, consists of two wires, D^0 and D^1, which may assume any value from the set {DATA0, DATA1, NULL}. The DATA0 state ($D^0 = 1$, $D^1 = 0$) corresponds to a Boolean logic 0, the

DATA1 state ($D^0 = 0, D^1 = 1$) corresponds to a Boolean logic 1, and the NULL state ($D^0 = 0, D^1 = 0$) corresponds to the empty set meaning that the value of *D* is not yet available. The two rails are mutually exclusive, so that both rails can never be asserted simultaneously; this state is defined as an illegal state. A quad-rail signal, *Q*, consists of four wires, Q^0, Q^1, Q^2, and Q^3, which may assume any value from the set {DATA0, DATA1, DATA2, DATA3, NULL}. The DATA0 state ($Q^0 = 1, Q^1 = 0, Q^2 = 0, Q^3 = 0$) corresponds to two Boolean logic signals, *X* and *Y*, where X = 0 and Y = 0. The DATA1 state ($Q^0 = 0, Q^1 = 1, Q^2 = 0, Q^3 = 0$) corresponds to X = 0 and Y = 1. The DATA2 state ($Q^0 = 0, Q^1 = 0, Q^2 = 1, Q^3 = 0$) corresponds to X = 1 and Y = 0. The DATA3 state ($Q^0 = 0, Q^1 = 0, Q^2 = 0, Q^3 = 1$) corresponds to X = 1 and Y = 1, and the NULL state ($Q^0 = 0, Q^1 = 0, Q^2 = 0, Q^3 = 0$) corresponds to the empty set meaning that the result is not yet available. The four rails of a quad-rail NCL signal are mutually exclusive, so no two rails can ever be asserted simultaneously; these states are defined as illegal states. Both dual-rail and quad-rail signals are space optimal delay-insensitive codes, requiring two wires per bit. Other higher order MEAGs are not typically wire count optimal, however they can be more power efficient due to the decreased number of transitions per cycle.

Consider the behavior of a symbolically complete AND function using NCL as shown in Figure 2. Assume it takes 1 ns for output *Z* of a NCL AND function to become valid once its inputs *X* and *Y* have arrived. Also, initially suppose *X* is DATA1, *Y* is DATA0, and *Z* is DATA0. Before the next set of inputs can be applied, all inputs must first transition to NULL, which causes the output to transition to NULL, 1 ns later. Once

the output has transitioned to NULL, the next input set can be applied. If the next input set consists of *X* = DATA1 and *Y* = DATA1, *Z* will become DATA1 after 1 ns, signaled by *Z* transitioning from NULL to DATA. Output *Z* will remain DATA1 until both inputs, *X* and *Y*, transition to NULL, due to the hysteresis behavior inherent in each threshold gate. Time is never referenced to determine the validity of *Z*. The 1 ns delay is an arbitrary gate transition delay and does not affect the validity of *Z*.

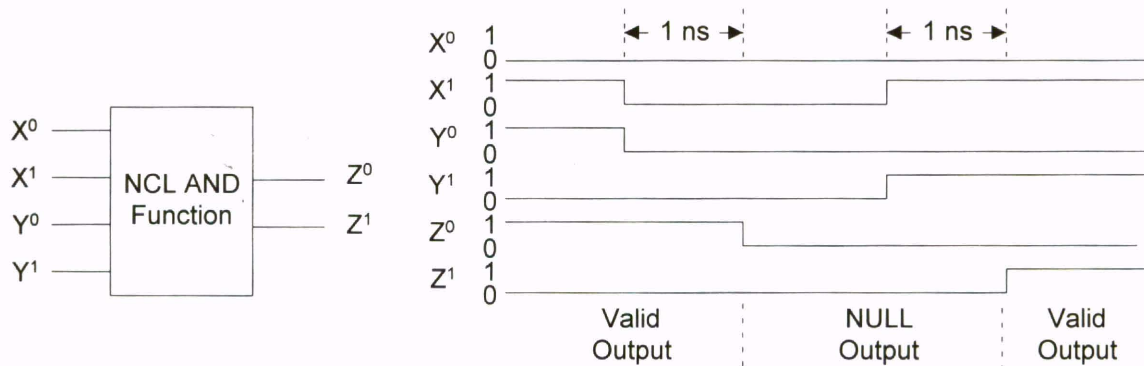

Figure 2. NCL AND function: Z = X • Y and associated waveforms.

2.2.2 Logic Gates and Functional Blocks

NCL uses *threshold gates* with *hysteresis* [23] for its composable logic elements. One type of threshold gate is the *THmn gate*, where $1 \leq m \leq n$, as depicted in Figure 3. A THmn gate corresponds to an operator with at least m signals asserted as its set condition and all signals de-asserted as its reset condition. THmn gates have *n* inputs. At least *m* of the *n* inputs must be asserted before the output will become asserted. Because threshold gates are designed with hysteresis, all asserted inputs must be de-asserted before the output will be de-asserted. Hysteresis is used to provide a means for monotonic

transitions and a complete transition of multi-rail inputs back to a NULL state before asserting the output associated with the next wavefront of input data. In a THmn gate, each of the n inputs is connected to the rounded portion of the gate. The output emanates from the pointed end of the gate. The gate's threshold value, m, is written inside of the gate. Figure 4 shows a static CMOS implementation of a TH23 gate, with inputs A, B, and C, and output Z. [23] details various design implementations (static, semi-static, and dynamic) of THmn gates.

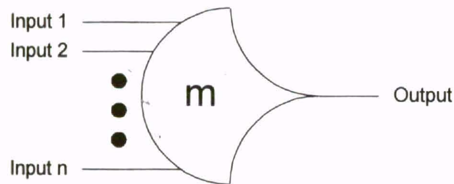

Figure 3. THmn threshold gate [21].

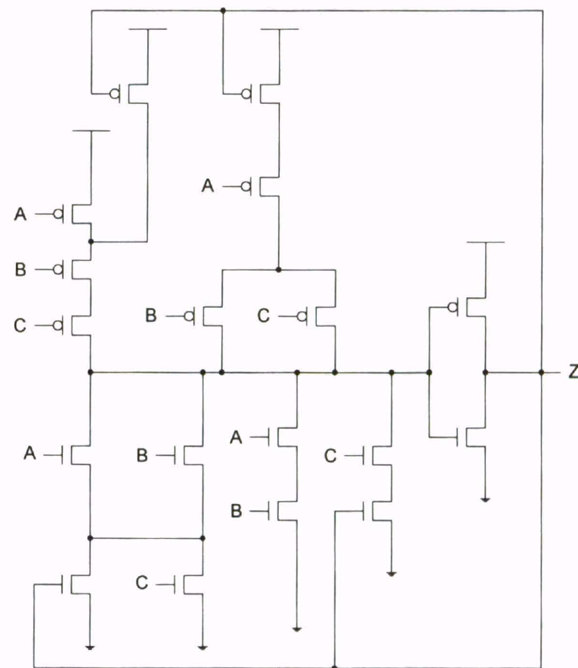

Figure 4. Static CMOS implementation of a TH23 gate.

Another type of threshold gate is referred to as a weighted threshold gate, denoted as THmnWw$_1$w$_2$...w$_R$. Weighted threshold gates have an integer value, $m \geq w_R > 1$, applied to *inputR*. Here $1 \leq R < n$; where n is the number of inputs; m is the gate's threshold; and w_1, w_2, ...w_R, are the integer weights of *input1, input2, ... inputR*,

respectively. For example, consider a TH34W2 gate, whose $n = 4$ inputs are labeled A, B, C, and D. The weight of input A, $W(A)$, is therefore 2. Since the gate's threshold, m, is 3, this implies that in order for the output to be asserted, either inputs B, C, and D must all be asserted, or input A must be asserted and any other input, B, C, or D must also be asserted. NCL threshold gates may also include a reset input to initialize the gate's output. Resetable gates are denoted by either a D or an N appearing inside the gate, along with the gate's threshold, referring to the gate being reset to logic 1 or logic 0, respectively.

By employing threshold gates for each logic rail, NCL is able to determine the output status without referencing time. Inputs are partitioned into two separate wavefronts, the NULL wavefront and the DATA wavefront. The NULL wavefront consists of all inputs to a circuit being NULL, while the DATA wavefront refers to all inputs being DATA, some combination of DATA0 and DATA1. Initially all circuit elements are reset to the NULL state. First, a DATA wavefront is presented to the circuit. Once all of the outputs of the circuit transition to DATA, the NULL wavefront is presented to the circuit. Once all of the outputs of the circuit transition to NULL, the next DATA wavefront is presented to the circuit. This DATA/NULL cycle continues repeatedly. As soon as all outputs of the circuit are DATA, the circuit's result is valid. The NULL wavefront then transitions all of these DATA outputs back to NULL. When they transition back to DATA again, the next output is available.

Figure 5 shows the primary functional blocks of a NCL circuit. The *NCL registration* stages act to control the DATA/NULL wavefronts, through their request input lines, K_i, and their request output lines, K_o. The *NCL completion* detects complete

DATA and NULL sets, where all outputs are DATA or all outputs or NULL, respectively, at the output of NCL registration. *NCL combinational circuits* provide the fundamental functionality of a NCL system. Since every NCL circuit continually cycles through NULL followed by DATA, one complete cycle will consist of NULL flowing through the combinational circuitry as shown in Figure 5, followed by NULL flowing through the completion circuitry as shown in Figure 6, followed by DATA flowing through the combinational circuitry as shown in Figure 7, and finally followed by DATA flowing through the completion circuitry, back to the input as shown in Figure 8. *rfn* refers to *request for NULL* and *rfd* refers to *request for DATA*. Each phase of this cycle, depicted in the Gantt chart of Figure 9, will be referred to here on out as the DATA-to-DATA cycle; and the period of this cycle will be called the DATA-to-DATA cycle time (T_{DD}). T_{DD} has an analogous role to the clock period in a synchronous system.

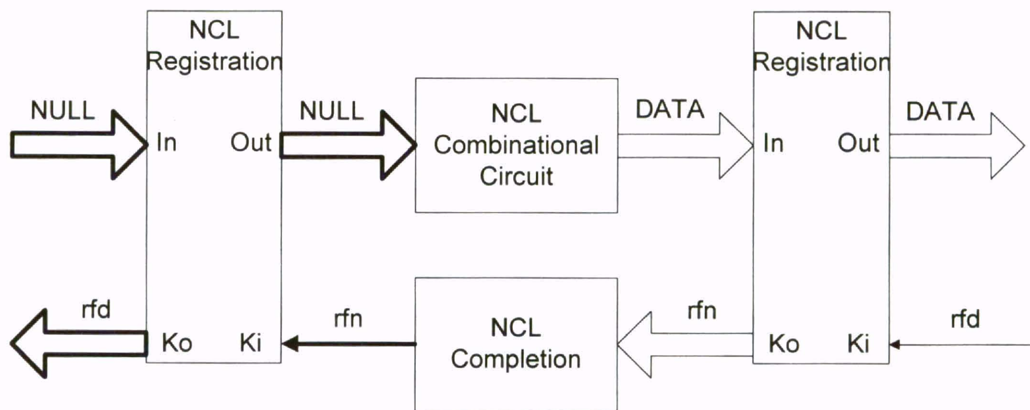

Figure 5. NULL flowing through combinational circuitry.

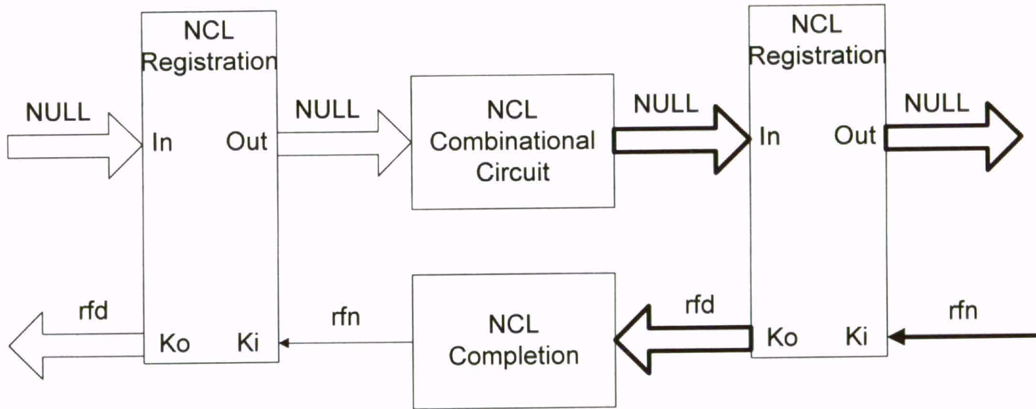

Figure 6. Completion detection of NULL output.

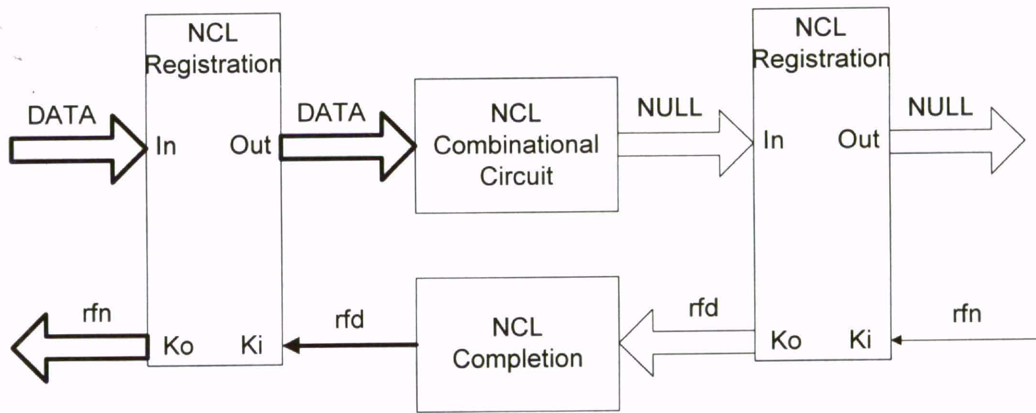

Figure 7. DATA flowing through combinational circuitry.

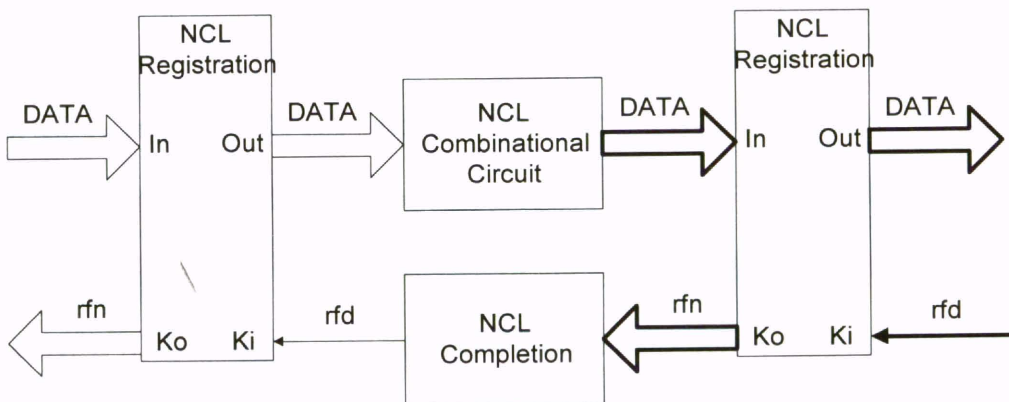

Figure 8. Completion detection of DATA output.

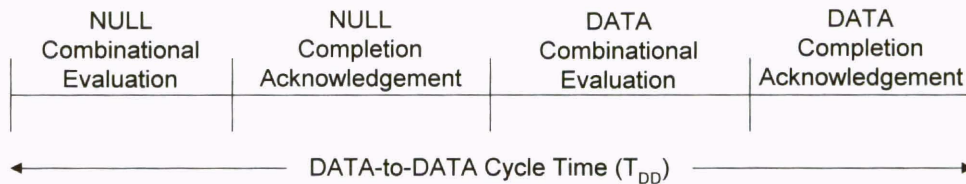

Figure 9. DATA-to-DATA cycle time (T_{DD}).

2.2.3 Completeness of Input

The *input-completeness* criterion [21], which NCL circuits must maintain in order to be delay-insensitive, requires that:

1. the outputs of a circuit may not transition from NULL to DATA until all inputs have transitioned from NULL to DATA, and
2. the outputs of a circuit may not transition from DATA to NULL until all inputs have transitioned from DATA to NULL.

In circuits with multiple outputs, it is acceptable for some of the outputs to transition without having a complete input set present, as long as all outputs cannot transition before all inputs arrive. This signaling scheme is equivalent to the "weak conditions" of delay-insensitive signaling defined by Seitz [4]. Consider the incomplete NCL AND function shown in Figure 10. The output can change from NULL to DATA0 without both inputs first transitioning to DATA. For instance, if A = DATA0 and B = NULL then C = DATA0, which breaks the completeness of input criterion. Figure 11 shows a complete NCL AND function since the output cannot transition until both inputs have transitioned.

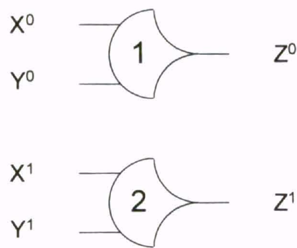

Figure 10. Incomplete AND function: Z = X • Y.

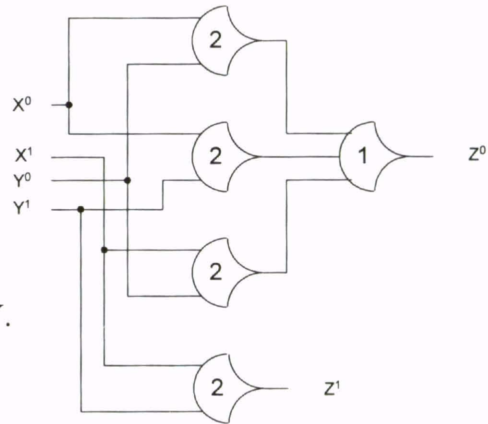

Figure 11. Conventional input-complete AND function: Z = X • Y.

Completeness of DATA can be ensured for an *N* input function as shown in Algorithm 2.1. If a function is complete with respect to DATA, it is also complete with respect to NULL due to the hysteresis functionality of every NCL gate. This completeness check takes $O(N \bullet 2^{N-1})$; however, this is unnecessary for many functions due to their inherent completeness. For example, the XOR function, the full adder, and the increment circuitry, all are inherently complete such that it is impossible to know the output without all of the inputs being known.

```
for (i = 1 to N) loop
    INPUT_i = NULL
    group INPUTS_j (1 ≤ j ≤ N, j ≠ i)
                        such that they form an N-1 bit word called REMAINDER
    for (k = 0 to 2^(N-1)-1) loop
        REMAINDER = k
        if (all output bits are DATA) then
            return (INCOMPLETE)
    end loop
end loop
return (COMPLETE)
```

Algorithm 2.1. Input-completeness pseudocode.

2.2.4 Observability

There is one more condition that must be met in order for NCL to retain delay-insensitivity. No *orphans* may propagate through a gate. An orphan is defined as a wire that transitions during the current DATA wavefront, but is not used in the determination of the output. Orphans are caused by wire forks and can be neglected through the isochronic fork assumption, as long as they are not allowed to cross a gate boundary. This *observability* condition ensures that every gate transition is observable at the output. Consider an incorrect version of an XOR function shown in Figure 12, where an orphan is allowed to pass through the TH12 gate. For instance, when X = DATA0 and Y = DATA0, the TH12 gate is asserted, but does not take part in the determination of the output, Z = DATA0. This orphan path is shown in boldface in Figure 12. A correct, fully observable version of the XOR function is given in Figure 13, where no orphans propagate through any gate. An orphan checker tool, as a Synopsys shell, is run on each design to ensure observability.

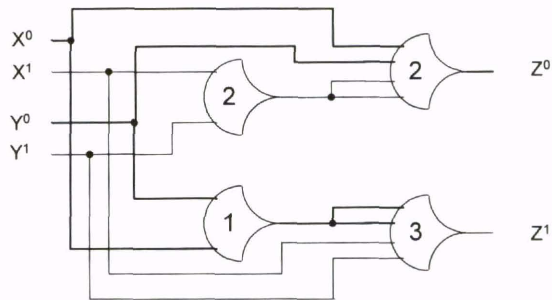

Figure 12. Incorrect XOR function: $Z = X \oplus Y$ (orphans may propagate through a gate).

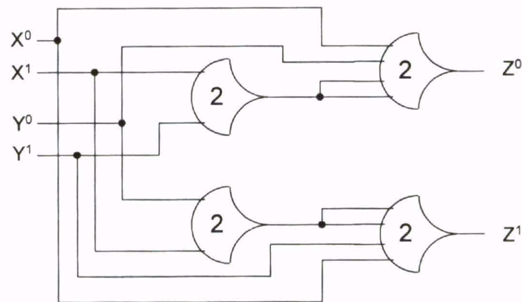

Figure 13. Correct XOR function: Z = X ⊕ Y
(orphans may not propagate through any gate).

2.2.5 NCL Registration

With the input-completeness and observability criteria met, a NCL circuit is therefore delay-insensitive, because the output will not transition until all of its inputs transition and two consecutive DATA wavefronts will always remain separated despite arbitrarily large gate delays. Henceforth, the circuit will wait indefinitely until it receives all of its inputs and the inputs traverse the logic, before requesting the next either NULL or DATA wavefront.

With this in mind, there must be a device that monitors the outputs of NCL circuits in order to detect when there is a complete DATA set or a complete NULL set, and upon detection of a complete output set, request the next wavefront. The NCL register, shown in Figure 14, does just that. When the request input line, K_i, is *rfd*, any of the register inputs, *I*, that are asserted are allowed to pass through their respective TH22 gate, to the output of the register. Likewise, when the request input line, K_i, is *rfn*, any of the register inputs, *I*, that are de-asserted are allowed to pass through their respective TH22 gate, to the output of the register. Only after all *n* inputs to the register have

transitioned to DATA, causing their respective outputs to transition to DATA as well, will the register's request output line, K_o, transition to *rfn*, meaning that the register has received the DATA wavefront and is requesting the NULL wavefront. And, only after all *n* inputs to the register have transitioned to NULL, causing their respective outputs to transition to NULL as well, will the register's request output line, K_o, transition to *rfd*, meaning that the register has received the NULL wavefront and is requesting the DATA wavefront.

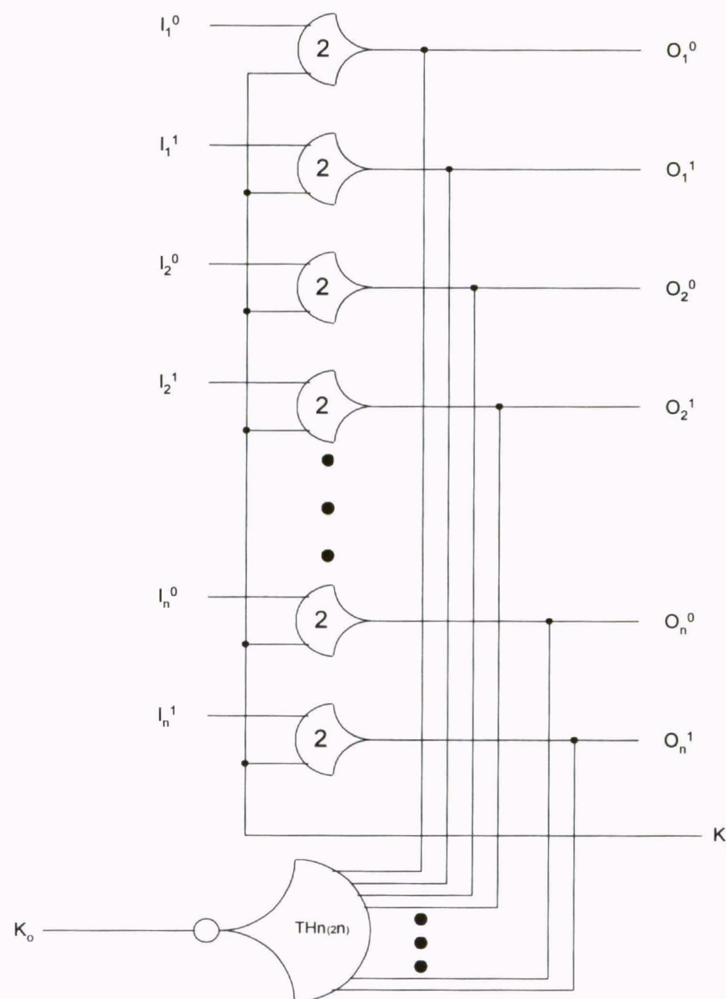

Figure 14: n-bit dual-rail registration.

The NCL register does not assure completeness of input, it only assures completeness of output. The NCL register will not request the NULL wavefront until the current DATA wavefront has been received; and likewise the next DATA wavefront will not be requested until the current NULL wavefront has been received. This would not prevent the NULL wavefront from being requested before all of the inputs become DATA, if the output was all DATA, caused by some inputs being DATA and combinational logic which is not complete with respect to its inputs.

Assume that the registers shown in Figure 15 have the following values: the output of the *upstream* register is DATA, so it is requesting NULL; the output of the *current* register is NULL, so it is requesting DATA; and the output of the *downstream* register is DATA, so it is requesting NULL. Also assume that the input to the upstream register is NULL and that the request input, K_i, to the downstream register is *rfn*. The NULL input to the upstream register will be blocked because the upstream register's request input line, K_i, is set to *rfd*. The DATA output from the upstream register will flow through the first set of combinational logic, to the input of the current register, while the NULL output of the current register flows through the second set of combinational logic to the input of the downstream register, as depicted in Figure 16. Once the DATA wavefront reaches the input of the current register, it is blocked, because the current register's request input line, K_i, is *rfn*. But when the NULL wavefront reaches the input of the downstream register, it is allowed to pass through to the output because the downstream register's request input line, K_i, is *rfn*. When every output of the downstream register transitions to NULL, the downstream register's request output line, K_o, will

transition to *rfd*, shown in Figure 17, which will allow the DATA wavefront at the input of the current register to pass through to the output of the current register and start flowing through the second set of combinational logic. When all outputs of the current register have transitioned to DATA, the request output line, K_o, of the current register will transition to *rfn*, as shown in Figure 18, which will allow the NULL wavefront at the input of the upstream register to pass through to the output of the upstream register and start flowing through the first set of combinational logic, as depicted in Figure 19. As shown in Figure 20, once the NULL wavefront has passed through the first set of combinational logic, the circuit will be in a static state; and no more transitions can occur until the request input line, K_i, of the downstream register transitions to *rfd*, signifying that the NULL wavefront at the output of the downstream register has been received by the next register after the downstream register. The registers will continue to control the NULL/DATA cycles in this fashion, insuring that the next wavefront is sent only after the current wavefront has produced all of its outputs.

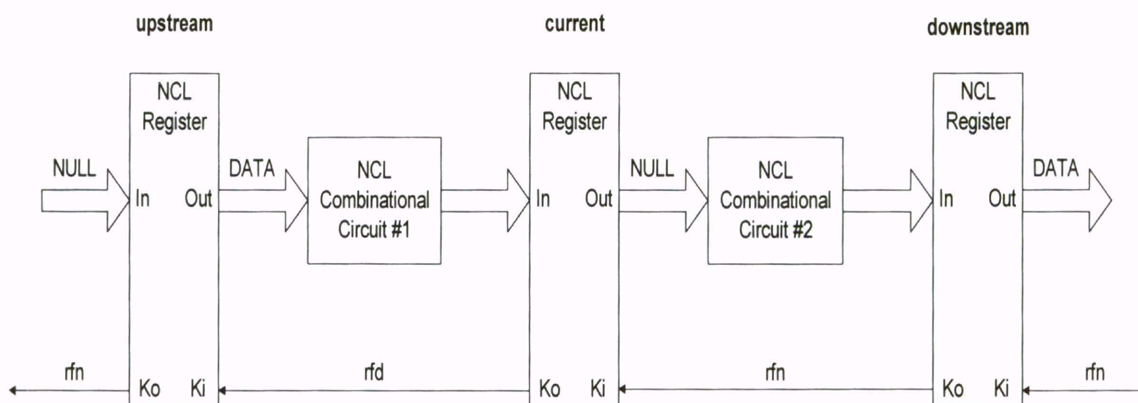

Figure 15. Initial register state.

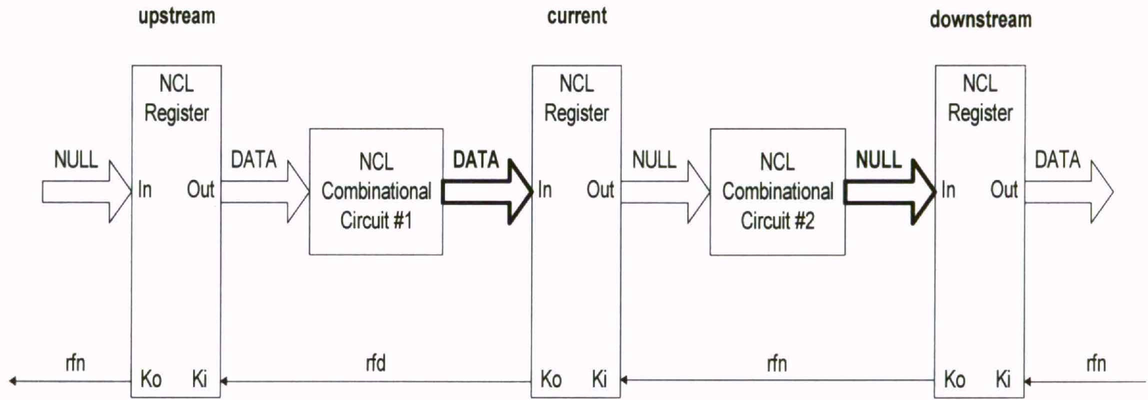

Figure 16. Register state after traversing combinational circuitry.

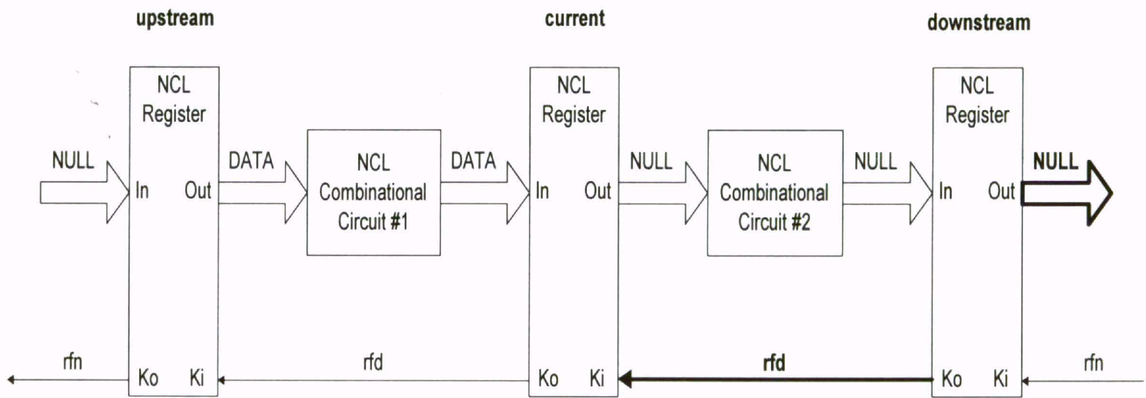

Figure 17. Register state after NULL wavefront passes through downstream register.

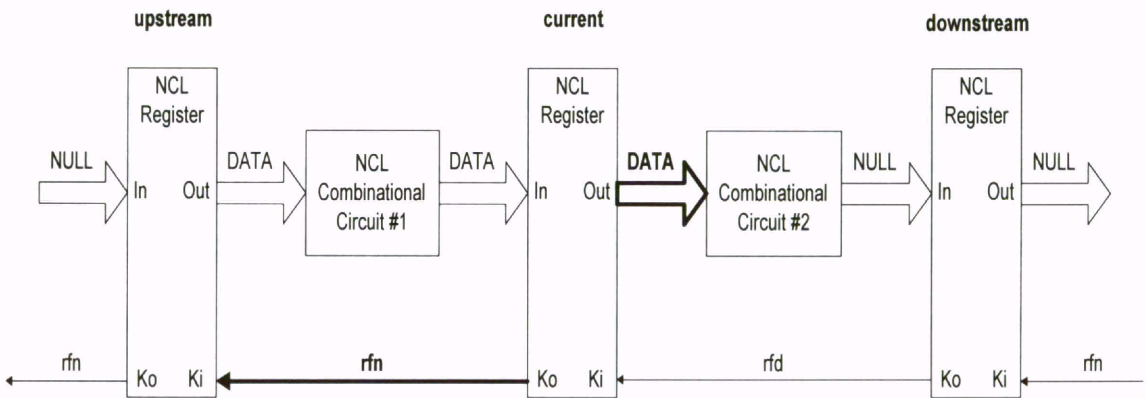

Figure 18. Register state after DATA wavefront passes through current register.

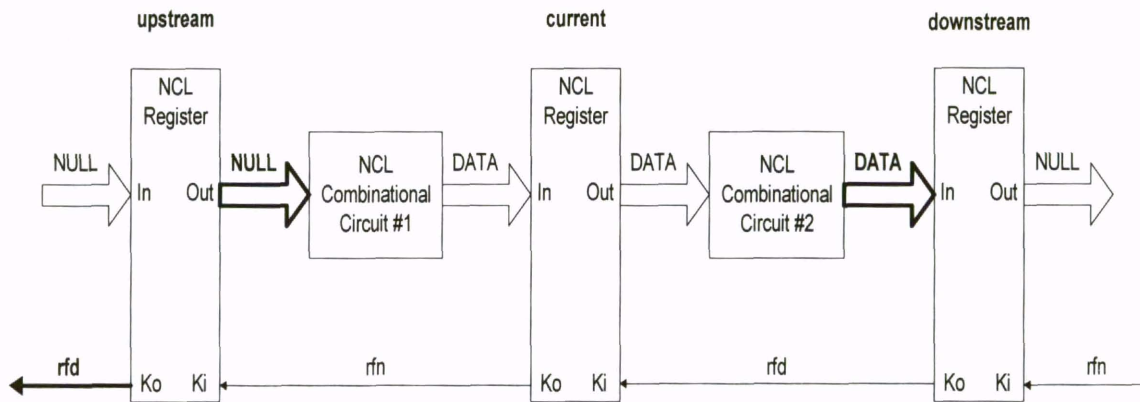

Figure 19. Register state after NULL wavefront passes through upstream register.

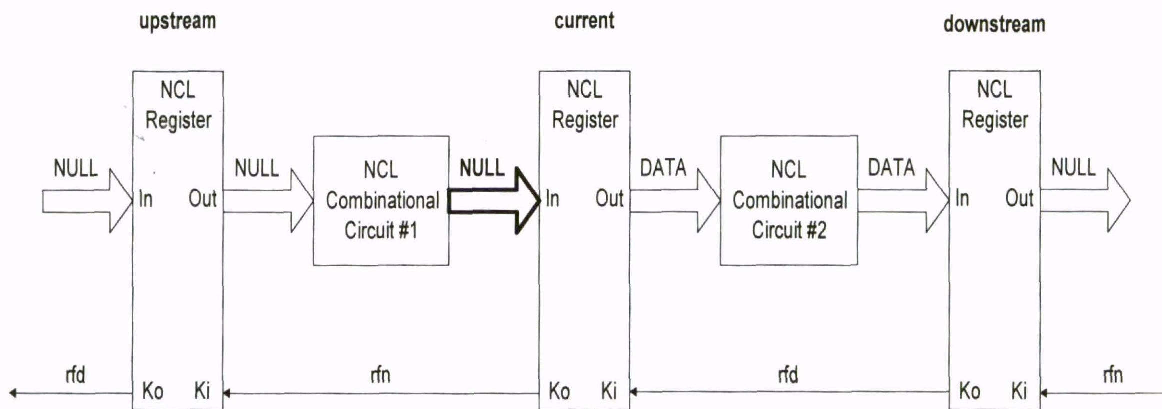

Figure 20. Static register state.

All NCL systems have at least two register stages, one at both the input and output; and all NCL systems with feedback have at least three register stages in the feedback loop [21]. This technique of organizing registers into a ring is fully discussed in [24, 9]. These register stages interact through handshaking to prevent DATA set_i from overwriting DATA set_{i-1} by ensuring that the two consecutive DATA sets are always separated by a NULL set.

2.2.6 NCL Completion

Actual NCL registration is realized through cascaded arrangements of single-bit dual-rail registers or single-signal quad-rail registers, depicted in Figure 21 and 22, respectively. Therefore, an N-bit register stage, comprised of N single-bit dual-rail NCL registers, requires N completion signals, one for each bit. The NCL *Completion* component, shown in Figure 23, uses these N K_o lines to detect complete DATA and NULL sets at the output of every register stage and request the next NULL and DATA set, respectively. The single-bit output of the completion component is connected to all K_i lines of the previous register stage. Since the maximum input threshold gate currently supported is the TH44 gate, the number of logic levels in the completion component for an N-bit register is given by $\lceil \log_4 N \rceil$. Likewise, the completion component for an N-bit quad-rail registration stage requires $\frac{N}{2}$ inputs, and can be realized in a similar fashion using TH44 gates. The registers shown in Figures 21 and 22 are reset to NULL. Either register could be instead reset to a DATA value by replacing exactly one of the TH22n gates with a TH22d gate.

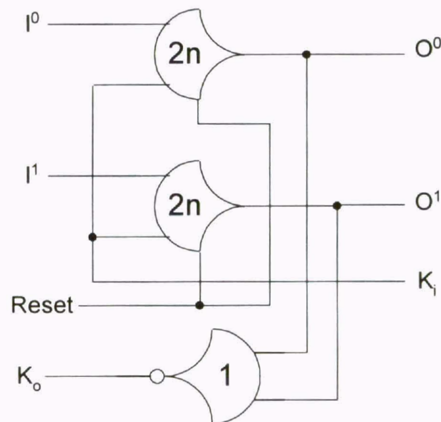

Figure 21. Single-bit dual-rail register.

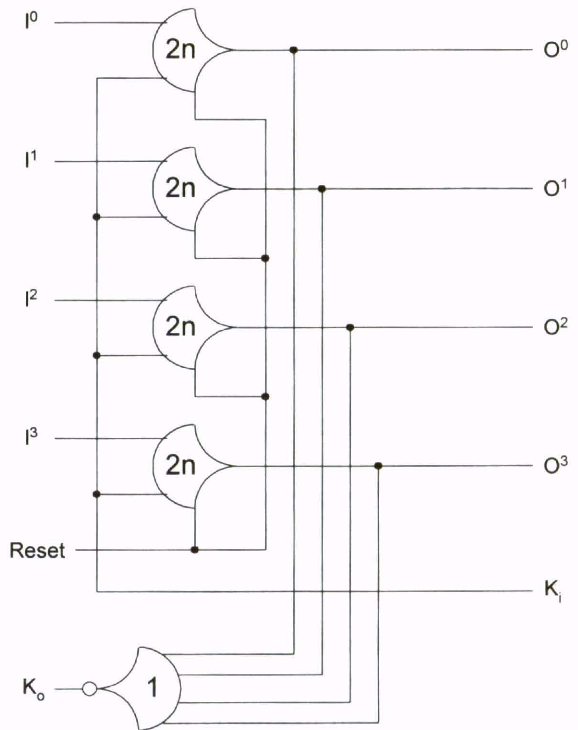

Figure 22. Single-signal quad-rail register.

Figure 23. N-bit completion component.

3.0 THRESHOLD COMBINATIONAL REDUCTION METHOD

Delay-insensitive logic design methods are developed using *Threshold Combinational Reduction (TCR)* within the NULL Convention Logic (NCL) paradigm. NCL logic elements are realized using 27 distinct transistor networks implementing the set of all functions of four or fewer variables, thus facilitating a variety of gate-level optimizations. TCR optimizations are formalized for NCL and then assessed by comparing levels of gate delays, gate counts, and transistor counts of the resulting designs. The methods are illustrated to produce fundamental logic functions, and a full adder with reduced critical path delay and transistor count over various alternative gate-level synthesis approaches. As an example of circuits employing feedback, TCR is applied to derive time and space optimized increment circuits for a 4-bit up-counter. Results demonstrate support for a variety of optimizations utilizing conventional Boolean minimization followed by table-driven gate substitutions. Whereas previous work on optimization of circuits constructed from logical operators has concentrated on transistor-sizing [25] and decomposition of high fan-in operators [26], this chapter will emphasize composable circuit construction utilizing a set of complex state-holding gates, and will illustrate circuit minimization techniques, their application, and associated tradeoffs.

3.1 Chapter Outline

This chapter is organized into five sections. In Section 3.2, the TCR method for optimizing combinational NCL circuits is developed. The method is demonstrated in Sections 3.3, 3.4, and 3.5. Section 3.3 presents optimal input-complete AND/NAND, OR/NOR, and XOR/NXOR logic functions, designed using TCR. Section 3.4 applies TCR to produce a delay-insensitive Full Adder that significantly reduces critical path delay and transistor count over previous gate-level delay-insensitive approaches. Section 3.5 illustrates the use of TCR to derive a variety of time and space optimized NCL increment circuitries for an up-counter with a feedback circuit.

3.2 TCR Method Definition

As depicted in Figure 24, the design process begins with a specification of the circuit functional behavior and desired optimization criteria. Circuit behavior is specified as Boolean logic expressions, truth tables, and/or narrative descriptions. The optimization criteria include parameters such as critical path delay, gate count, transistor count, or power consumption, that are to be minimized in the target design. Several alternate designs are generated, which are then assessed against the optimization criteria, allowing the preferred design to be selected for implementation.

First, a logic encoding scheme is selected such as dual-rail, quad-rail, or other MEAG representations, as depicted in Figure 24. Typically either dual-rail or quad-rail is chosen since these encodings yield the minimum of two wires per bit. If a dual-rail encoding is used, the next step is to select the optimization space in which minimization

will be performed. The proposed TCR design methods have been numbered “1”, “2”, and “3”, each with design steps labeled “A”, “B”, or “C”, appropriately.

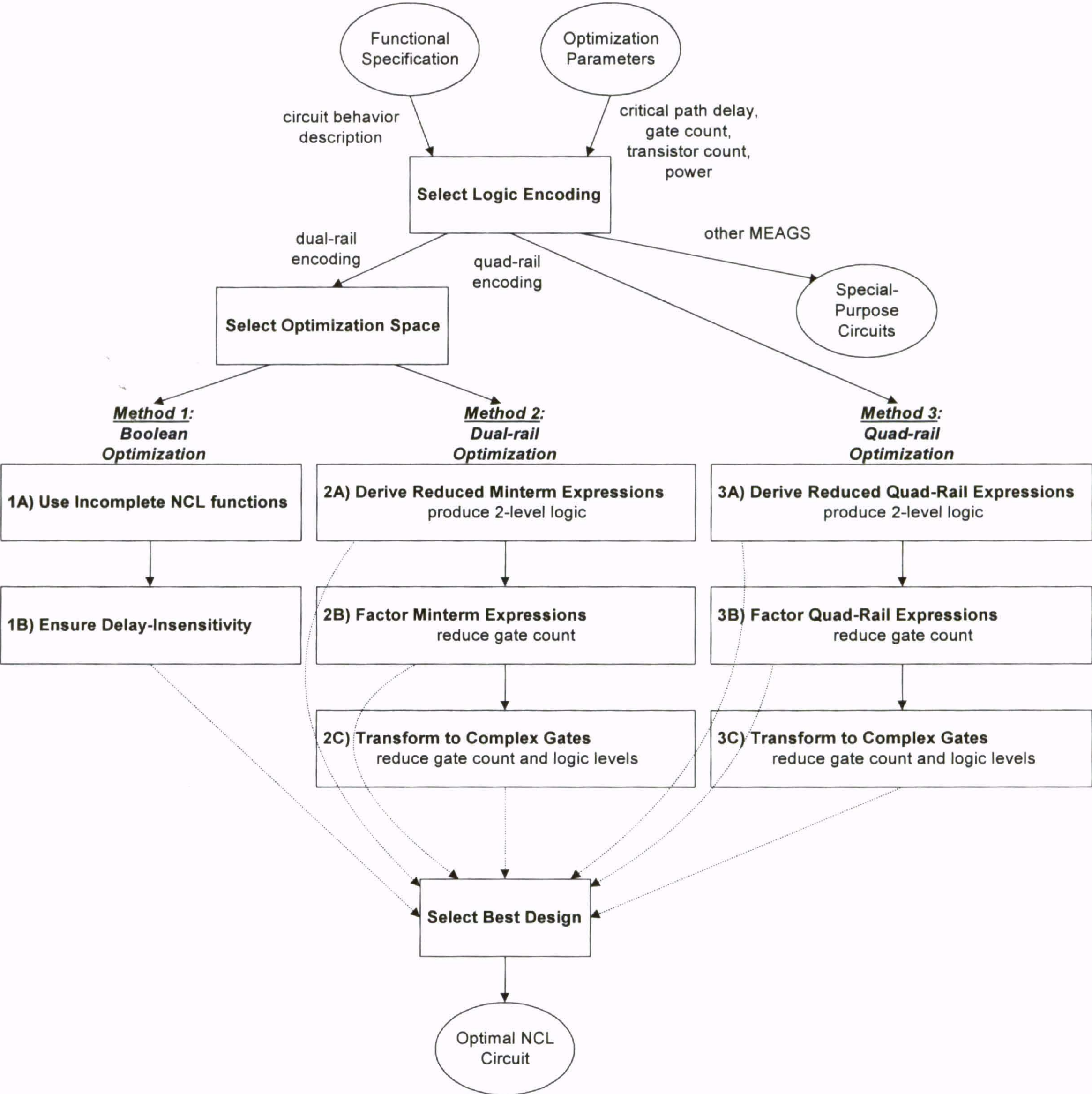

Figure 24. TCR design flow.

3.2.1 Method 1: Incomplete Functions

As depicted in Figure 24, Method 1 corresponds to Boolean optimization. Maximal use of incomplete NCL logic functions, such as the incomplete AND function shown in Figure 10, generates the individual outputs, while maintaining the completeness of input criterion for the circuit as a whole. For example, gates in Boolean designs that target the basic logical operators (AND, OR, XOR, NAND, NOR, NXOR, NOT) are directly mapped to a NCL design by using as many incomplete NCL functions as possible. As described in Step 1A of Figure 24, each Boolean gate is replaced with its NCL equivalent function, using incomplete versions whenever possible. Step 1B ensures input-completeness for the circuit as a whole by employing complete functions only for selected gates in the data path, so that the computation of an entire output set implies that the complete input set has arrived. The observability criterion must also be ensured.

3.2.2 Method 2: Dual-Rail Optimizations

Method 2 is based on dual-rail optimization. In Step 2A, the NCL circuit is optimized by using reduced minterm expressions for both rails of the output. These expressions are then mapped to TH1n and THnn gates. As in Boolean circuits, a Karnaugh map can be constructed for each output. The *0s* in the Karnaugh map refer to a signal's $rail^0$ line and the *1s* refer to a signal's $rail^1$ line. Reduced minterm expressions for both the *1s* and *0s* in the Karnaugh map are derived. After these expressions for the outputs have been obtained, an assessment must be made to ensure that the complete output set cannot be generated without all of the inputs being present. If under any timing

scenario, a complete output set can be generated without all of the inputs being present, the missing logic terms must be added to the reduced expressions to guarantee that the completeness of input criterion holds. This method will always generate two-level logic, given that threshold gates with a sufficiently large number of inputs are available. The first level will consist of THnn gates, to produce the required minterms; and the second level will consist of TH1n gates, which act to OR the minterms together to produce the desired outputs. Step 2A is similar to Anantharaman's approach [7] and DIMS [9]. In Step 2B, the common sub-expressions are factored and consolidated to reduce the gate count. Finally, the factored expressions for each rail are manipulated in Step 2C to obtain equations of the forms contained in Table III. The observability criterion must be ensured for every circuit output from Steps 2A, 2B, and 2C.

Table III lists the 27 transistor networks, along with their corresponding Boolean equations, used to construct NCL circuits. These 27 transistor networks, implemented as macros, constitute the set of all functions consisting of four or fewer variables. Since each rail of a NCL signal is considered a separate variable, a four variable function is not the same as a function of four literals, which would normally consist of eight variables. Twenty four of these macros can be realized using complex threshold gates, identical to the standard threshold gate forms for functions of four or fewer variables [27, 28, 29]. The other three macros could be constructed from threshold gate networks, but have been implemented as transistor networks to provide completeness. Table III also contains the transistor count for these 27 macros.

Table III. 27 NCL macros.

NCL Macro	Boolean Function	Transistor Count
TH12	A + B	6
TH22	AB	12
TH13	A + B + C	8
TH23	AB + AC + BC	18
TH33	ABC	16
TH23w2	A + BC	15
TH33w2	AB + AC	14
TH14	A + B + C + D	10
TH24	AB + AC + AD + BC + BD + CD	27
TH34	ABC + ABD + ACD + BCD	26
TH44	ABCD	20
TH24w2	A + BC + BD + CD	23
TH34w2	AB + AC + AD + BCD	22
TH44w2	ABC + ABD + ACD	23
TH34w3	A + BCD	19
TH44w3	AB + AC + AD	16
TH24w22	A + B + CD	18
TH34w22	AB + AC + AD + BC + BD	22
TH44w22	AB + ACD + BCD	24
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	20
TH24comp	AC + BC + AD + BD	18

3.2.3 Method 3: Quad-Rail Optimizations

For some circuits, it may be advantageous to use quad-rail optimization, referred to as Method 3 in Figure 24. Two dual-rail signals yield the same five logic states as one quad-rail signal, however using quad-rail logic signals may lead to a more efficient design. Quad-rail optimization follows the same steps as does dual-rail optimization. In Step 3A, the NCL circuit is optimized by using reduced minterm expressions for all four rails of the output. These expressions are then mapped to TH1n and THnn gates. As in

dual-rail optimization, a Karnaugh map can be constructed for each output, but instead of only *0s* and *1s*, corresponding to a signal's $rail^0$ and $rail^1$, respectively, the K-map also contains *2s* and *3s*, which correspond to a signal's $rail^2$ and $rail^3$, respectively. Reduced minterm expressions for the *0s*, *1s*, *2s*, and *3s* in the Karnaugh map are derived. After these expressions for the outputs have been obtained, an assessment must be made to ensure that the complete output set cannot be generated without all of the inputs being present. If under any timing scenario, a complete output set can be generated without all of the inputs being present, the missing logic terms must be added to the reduced expressions to guarantee that the completeness of input criterion holds. This method will always generate two-level logic, given that threshold gates with a sufficiently large number of inputs are available. The first level will consist of THnn gates, to produce the required minterms; and the second level will consist of TH1n gates, which act to OR the minterms together to produce the desired outputs. In Step 3B, the common sub-expressions are factored and consolidated to reduce the gate count. Finally, the factored expressions for each rail are manipulated in Step 3C to obtain equations of the forms contained in Table III. The observability criterion must be ensured for every circuit output from Steps 3A, 3B, and 3C.

3.2.4 Performance Assessment

To assess the performance of alternate designs, Synopsys, a commercial design tool, was used to simulate the circuits to generate their timing characteristics. All NCL circuits presented herein have been exhaustively tested and their average cycle time, T_{DD},

has been reported. The Synopsys technology library for the 27 macros is based on Spice simulations of static 0.25 μm CMOS gates, operating at 3.3V. Along with the average cycle time, the number of gates and transistors has also been tabulated for comparison. The design that best meets the desired criteria can then be selected for implementation.

3.3 Application to Input-Complete Fundamental Logic Functions

Several optimizations can be used to generate designs that are very competitive in terms of speed and area as compared to other self-timed approaches. For example, Figures 11, 25, and 26 show the conventional implementations of the logic functions: AND, OR, and XOR, respectively. Each of these may be obtained directly from their minterm form. Method 2 is readily applicable. Dual-Rail Encoding Optimization achieves significant reduction in both area and speed. TCR Step 2C can be applied directly from the minterm form to reduce the circuit complexity and improve performance. Specifically, consider the objective of realizing an optimized input-complete 2-input OR function: $Z = X + Y$. The minterm expression for Z^0 is given by: $Z^0 = X^0Y^0$, which directly maps to a TH22 gate in Table III. The minterm expression for Z^1 is given by: $Z^1 = X^1Y^1 + X^0Y^1 + X^1Y^0$, which directly maps to a THand0 gate. Similarly, an optimized input-complete 2-input AND function: $Z = X \bullet Y$ can be realized. The minterm expression for Z^0 is given by: $Z^0 = X^0Y^0 + X^0Y^1 + X^1Y^0$, which directly maps to a THand0 gate. The minterm expression for Z^1 is given by: $Z^1 = X^1 Y^1$, which directly maps to a TH22 gate. The derivation of an optimized 2-input XOR function: $Z = X \oplus Y$ is a bit more complex. The minterm expression for Z^0 is given by: $Z^0 = X^0Y^0 + X^1Y^1$,

which directly maps to a THxor0 gate. The minterm expression for Z^1 is given by: $Z^1 = X^1Y^0 + X^0Y^1$, which also directly maps to a THxor0 gate. However, two transistors can be eliminated for each rail of *Z* by adding the two *don't care* terms, representing the cases when both rails of either *X* or *Y* are simultaneously asserted. The new equations for Z^0 and Z^1 are as follows: $Z^0 = X^0Y^0 + X^1Y^1 + X^0X^1 + Y^0Y^1$ and $Z^1 = X^1Y^0 + X^0Y^1 + X^0X^1 + Y^0Y^1$, both of which now map to TH24comp gates.

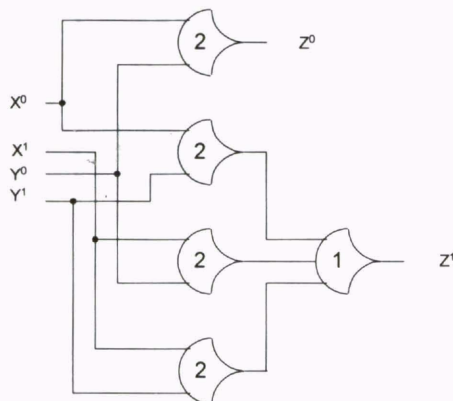

Figure 25. Conventional input-complete OR function: Z = X + Y.

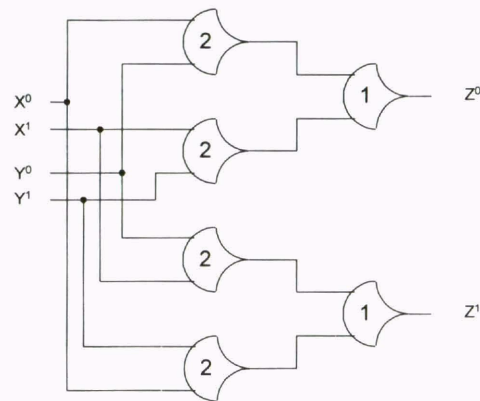

Figure 26. Conventional input-complete XOR function: $Z = X \oplus Y$.

As shown in Table IV, the AND, OR, and XOR functions produced using TCR outperform the conventional minterm designs in terms of both area and throughput. In particular, the TCR optimized AND and OR functions are 2.2-fold faster and require 43% fewer transistors than the conventional minterm designs. Furthermore, the optimized XOR function is 2.3-fold faster and requires 40% fewer transistors than the conventional minterm design. The inverse logic functions, NAND, NOR, and NXOR, can easily be

attained by exchanging the output rails of the AND, OR, and XOR functions, respectively.

Table IV. Performance characteristics of input-complete NCL logic functions.

Complete AND	Component List	Gate Delays	Gate Count	Transistor Count	T_{DD}
Conventional	4xTH22, 1xTH13	2	5	56	1.58 ns
TCR Method 2	1xTHand0, 1xTH22	1	2	32	0.71 ns

Complete OR	Component List	Gate Delays	Gate Count	Transistor Count	T_{DD}
Conventional	4xTH22, 1xTH13	2	5	56	1.58 ns
TCR Method 2	1xTHand0, 1xTH22	1	2	32	0.71 ns

XOR	Component List	Gate Delays	Gate Count	Transistor Count	T_{DD}
Conventional	4xTH22, 2xTH12	2	6	60	1.70 ns
TCR Method 2	2xTH24comp	1	2	36	0.75 ns

3.4 Application to Full Adder

The truth table for a full adder circuit is shown in Figure 27, where X and Y denote the input addends and C_i denotes the carry input. S and C_o denote the sum and carry output, respectively. This circuit can be extensively optimized using TCR Method 2. Applying TCR Step 2A, the K-map for the C_o output is obtained as shown in Figure 28, yielding: $C_o^0 = X^0Y^0 + C_i^0X^0 + C_i^0Y^0$ and $C_o^1 = X^1Y^1 + C_i^1X^1 + C_i^1Y^1$. Both functions directly map to a TH23 gate, so factoring in Step 2B is not necessary. Since a TH23 gate does not produce an output which is complete with respect to any of its inputs, there must be another output or set of outputs that enforce the completeness of input criterion. As explained below, the sum output, S, will enforce the completeness of input criterion for the circuit as a whole, thus allowing the carry output to be incomplete.

X	Y	Ci	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 27. Truth table for full adder.

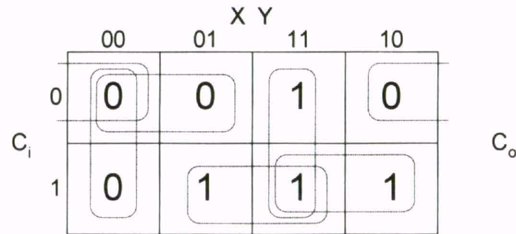

Figure 28. K-map for C_o output of full adder.

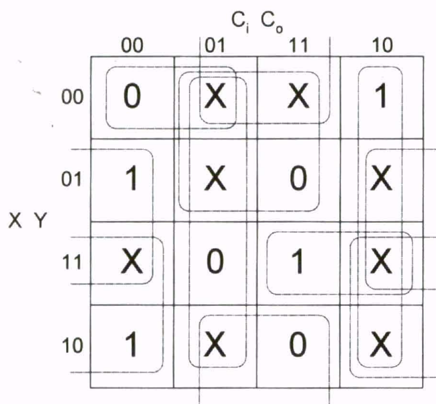

Figure 29. K-map for *S* output of full adder.

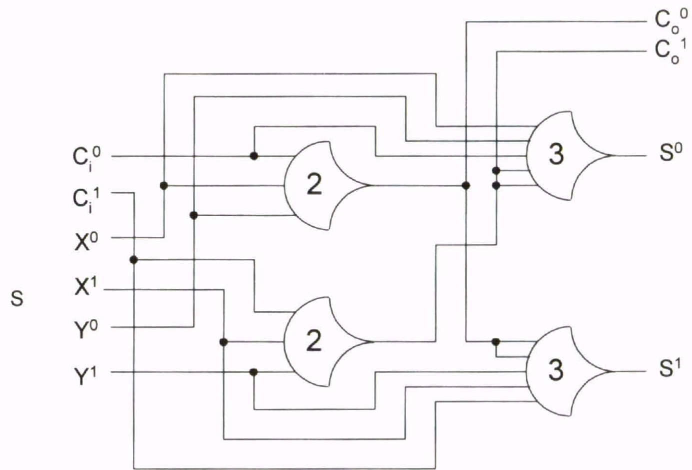

Figure 30. Optimized NCL full adder [21].

The K-map for *S*, based on *X*, *Y*, C_i, and the intermediate output C_o, is shown in Figure 29, with essential prime implicants covered. This cover yields: $S^0 = C_o^1X^0 + C_o^1Y^0 + C_o^1C_i^0 + X^0Y^0C_i^0$ and $S^1 = C_o^0X^1 + C_o^0Y^1 + C_o^0C_i^1 + X^1Y^1C_i^1$, both of which directly map to TH34W2 gates, so factoring in Step 2B is not necessary. C_o is taken as the *A* input such that $W(C_o) = 2$, as shown in Figure 30. Checking input-completeness, the carry output requires at least two inputs to be generated and the sum output requires either the carry output and one more input, or all three inputs to be

generated; so all three inputs are needed to generate the sum output. Therefore, the completeness of input criterion holds for the circuit as a whole.

As shown in Table V, the NCL design of the full adder produced using TCR optimizations can outperform those of other delay-insensitive methods, such as Anantharaman's and DIMS, Seitz's, David's, and Singh's approaches, shown in Figures 31, 32, 33, and 34, respectively. Here n-input C-elements are drawn as THnn gates since their functionality is identical. The NCL design has far fewer gates and transistors, while requiring fewer logic levels to produce the *carry* output, C_o. NCL also requires fewer logic levels to produce the *sum* output, *S*, than three of the five other methods, and has the same number of logic levels for *S* as the other two. Notice that the NCL full adder uses the carry output as an input to compute the sum output, whereas the other methods compute the sum and carry outputs independently. This is because for the other methods it is not practical to use the carry output to help generate the sum output. For the other methods the carry output is generated in the same number of logic levels, or more, as the sum output. Therefore, to use the carry output as an input for calculating the sum output would require more logic levels, as well as more gates. Besides NCL, only Seitz's full adder can be designed such that C_o can be computed before the C_i input is known for the cases A = DATA0, B = DATA0 and A = DATA1, B = DATA1. This optimization is important if the full adder component is to be used in an N-bit ripple-carry addition; since it allows the addition to be performed in $O(\log_2 N)$ on average instead of O(N). This optimization could be applied to DIMS, Anantharaman's approach, and David's method,

if their signaling scheme was slightly changed such that it coincided with the "weak conditions" of delay-insensitive signaling defined by Seitz [4] and used by NCL.

Table V. Full adder using various delay-insensitive methods.

Method	Design Level	Gate Delays for C_o	Gate Delays for S	Gate Count	Transistor Count
Seitz [4]	gate	2	3	18	154
Anantharaman [7]	gate	2	2	12	168
DIMS [9]	gate	2	2	12	168
David [6]	gate	3	3	20	186
Singh [8]	gate	6	4	19	192
TCR (Method 2)	gate	1	2	4	80
Martin [30]	transistor	1	1	3	42 or 34

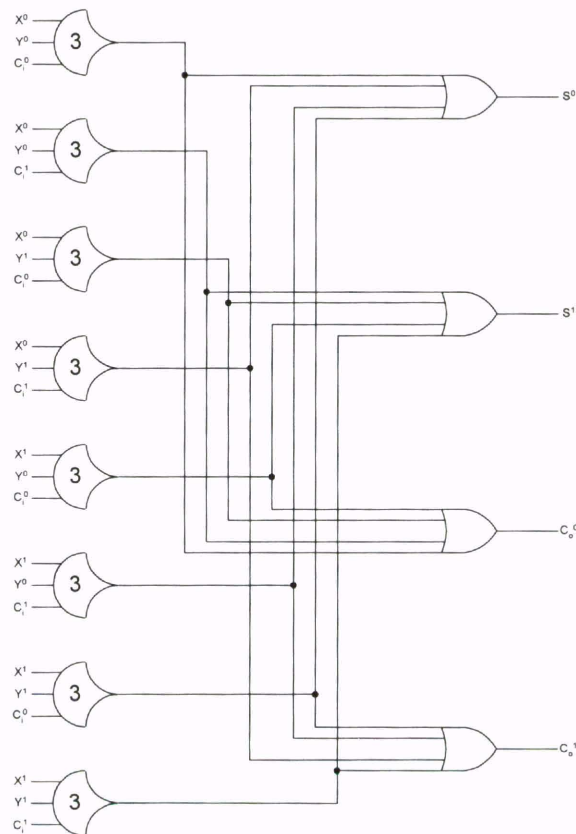

Figure 31. Full adder using Anantharaman's approach or DIMS [9].

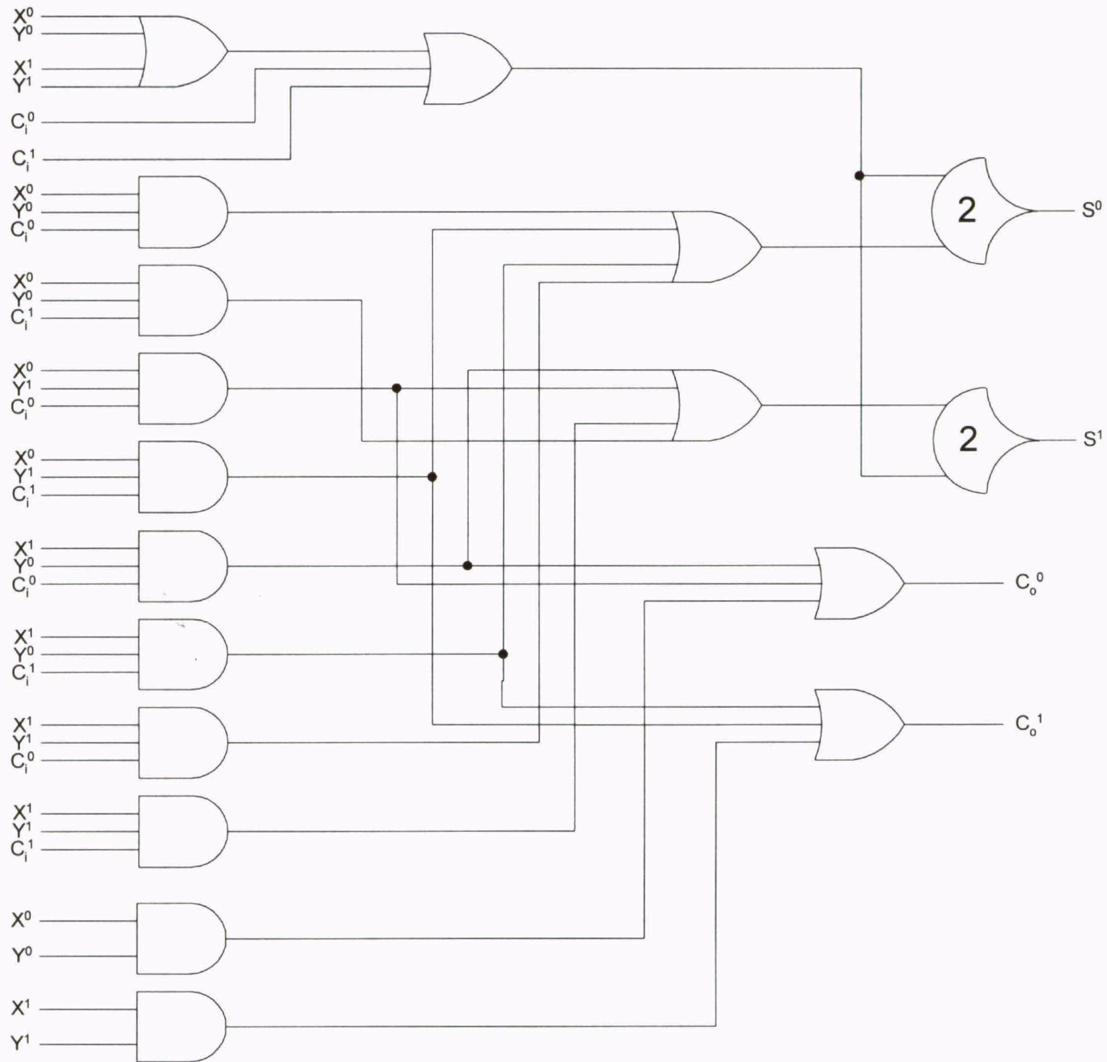

Figure 32. Full adder using Seitz's approach [4].

NCL circuits are often able to outperform other self-timed methods since NCL targets a wider range of logical operators whereas other methods target a more standard, restricted set. For example, the full adder can be further optimized by design methods at the transistor level as demonstrated by Martin [30]. His full adder requires three complex transistor networks: the first computes both rails of the sum output, while the second and third each compute one rail of the carry output. The resulting design consists of only 42

transistors, when the input and output inverters are included, or 34 transistors otherwise. However, this method is not directly comparable to the other above mentioned methods since it optimizes designs at the transistor level instead of targeting use of a predefined set of gates.

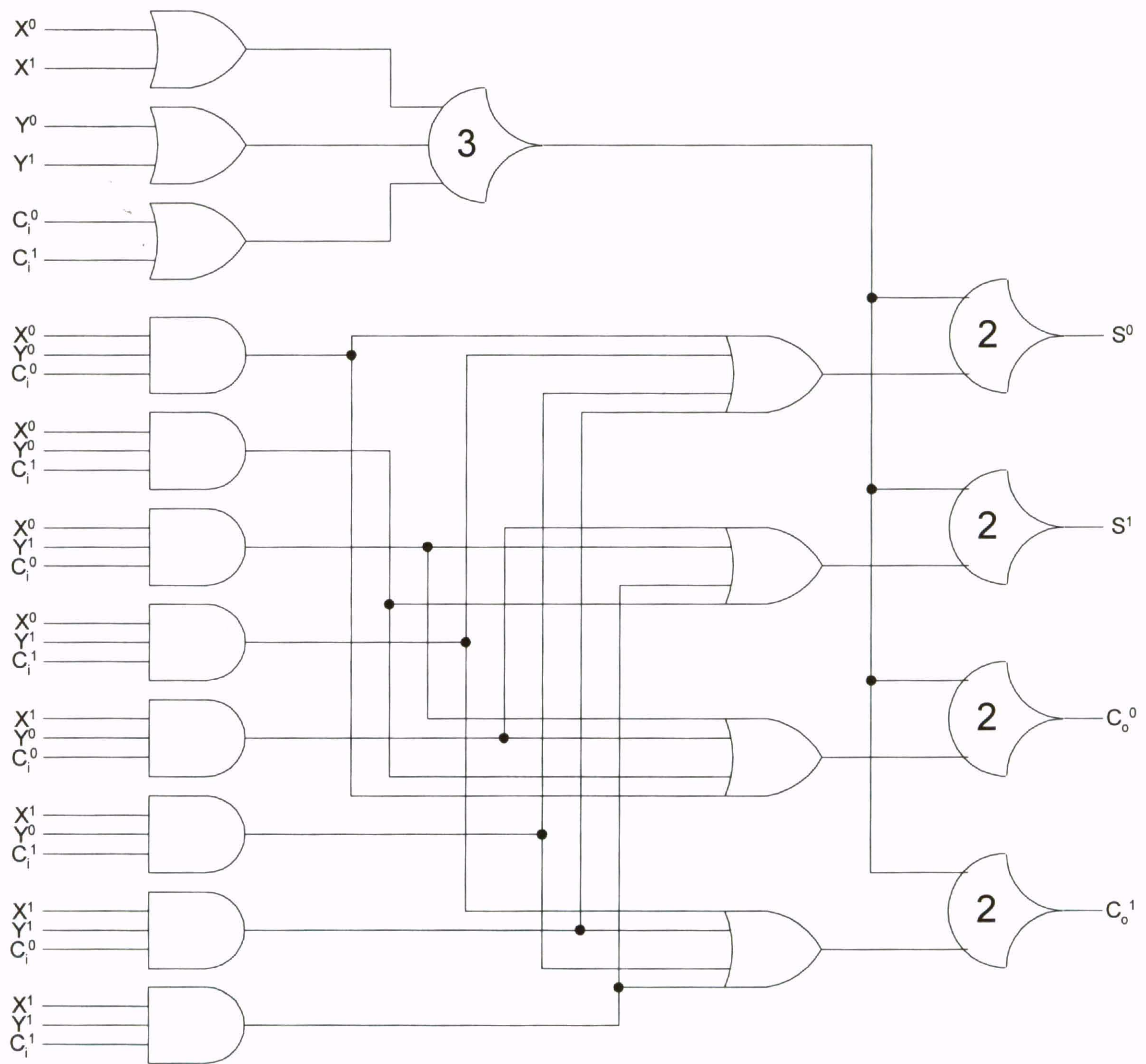

Figure 33. Full adder using David's approach.

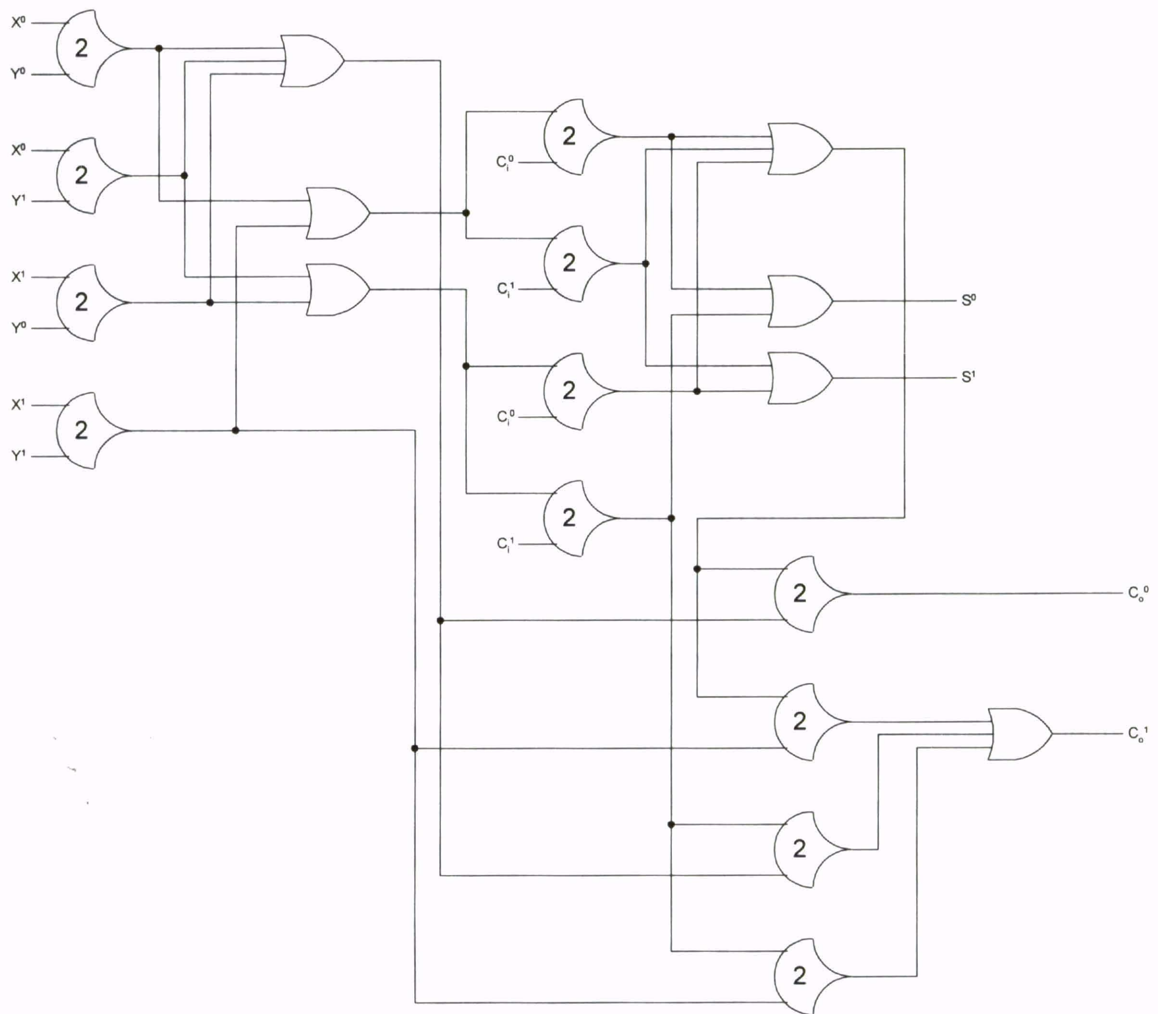

Figure 34. Full adder using Singh's approach.

As for general-purpose methods, DIMS, Seitz's method, and Anantharaman's approach require full minterm generation, so that no simplification is possible. DIMS and Anantharaman's approach cannot outperform NCL, and at best will be identical only if the NCL design requires full minterm generation. Seitz's approach can outperform NCL in terms of area, but not speed, for a limited class of circuits. These include functions

with 4 or more inputs, with one or few outputs, that contain almost all *1s* or *0s* in their truth table. These are the types of circuits that will receive little benefit from TCR optimizations. David's and Singh's approaches also favor these same classes of functions, and typically produce more efficient circuits than those obtainable by Seitz's approach. Singh's approach will require less area, but more delay than TCR for these types of functions, whereas David's approach will provide the same speed with significantly less area. For example, consider the function: $f(a, b, c, d) = a \bullet b' \bullet c \bullet d'$ [6]. Table VI shows that Seitz's, David's, and Singh's circuits are all better than those obtainable by TCR, in terms of area for this function and that Anantharaman's approach is the same. However, only David's approach outperforms TCR in both area and speed. David's approach is better because this function, and others like it, require full minterm generation in NCL to ensure input-completeness, so no simplification is possible by TCR methods.

Table VI. Delay-insensitive methods for $f(a, b, c, d) = a \bullet b' \bullet c \bullet d'$.

Method	Gate Delays	Gate Count	Transistor Count
Seitz [4]	4	25	250
Anantharaman [7]	3	21	368
DIMS [9]	3	21	368
David [6]	3	9	88
Singh [8]	4	15	168
NCL	3	21	368

3.5 Application to Up-Counter

A number of experiments based on the 4-bit counter shown in Figure 35 have been conducted. The specifications for this counter include a full NCL interface with

request and acknowledge signals labeled K_i and K_o, respectively. Functionality was specified to reset *count* to 0000b when the *reset* signal is applied, to increment *count* by 1 when *inc* = 1, and to keep *count* the same when *inc* = 0. The counter will rollover to 0000b when *count* = 1111b and *inc* = 1.

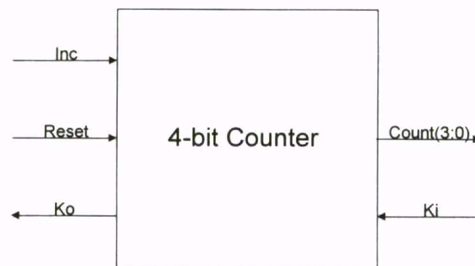

Figure 35. 4-bit up-counter block diagram.

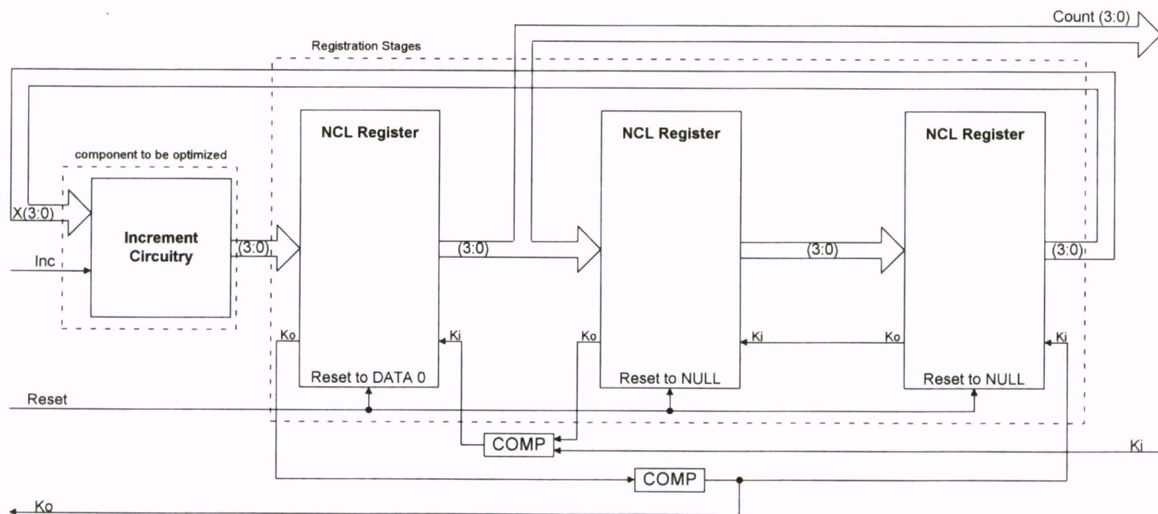

Figure 36. Up-counter with three-register feedback.

The functional design of the 4-bit counter, shown in Figure 36, will be the same for all counter models considered here. However, the *Increment Circuitry* will differ based on the particular TCR optimization method that is used. Figure 36 shows that there

are three NCL registers to feedback the current count to the increment circuitry. These *Registration Stages* act to control the DATA/NULL wavefronts, through their request in lines, K_i, and their request out lines, K_o. The completion logic (*COMP*) detects complete DATA and NULL sets, where all outputs are DATA or all outputs are NULL, respectively, at the output of NCL registration. The waveforms for the dual-rail, 16-rail, and quad-rail counters are shown in Figures 37, 38, and 39, respectively, with timing information depicted in nanoseconds. From these simulations the average DATA-to-DATA cycle time can be computed as follows: $T_{DD} = \frac{T_T}{32}$; where T_T is the total time for all input combinations and 32 is the number of combinations of the 5 circuit inputs (i.e. $2^5 = 32$). The timing information shown for the dual-rail and quad-rail waveforms is for their respective complex gate model.

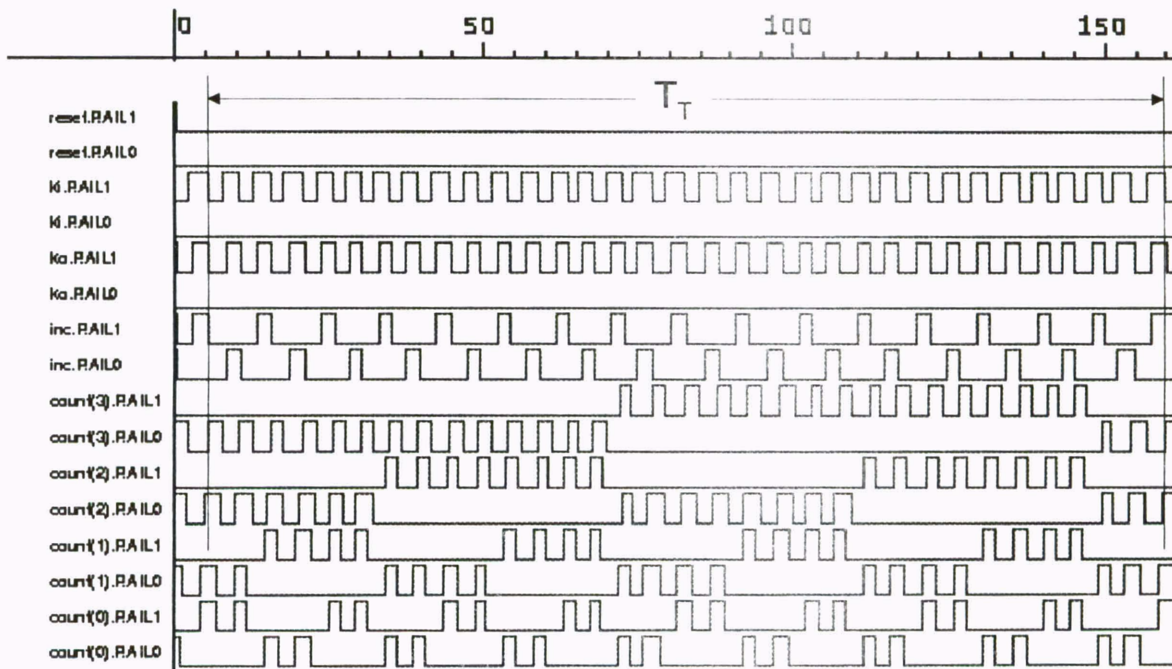

Figure 37. Dual-rail 4-bit counter waveforms.

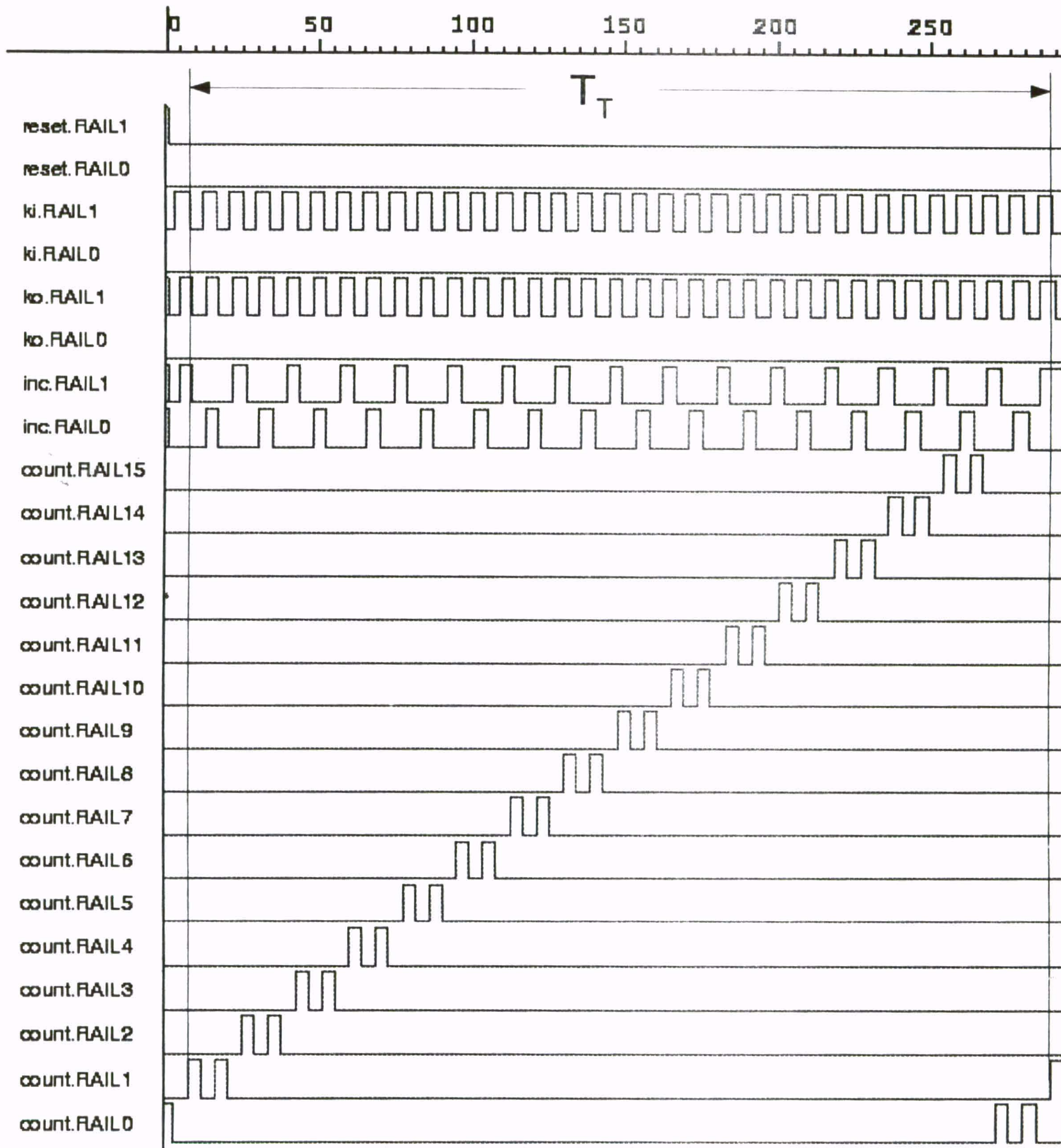

Figure 38. 16-rail MEAG 4-bit counter waveforms.

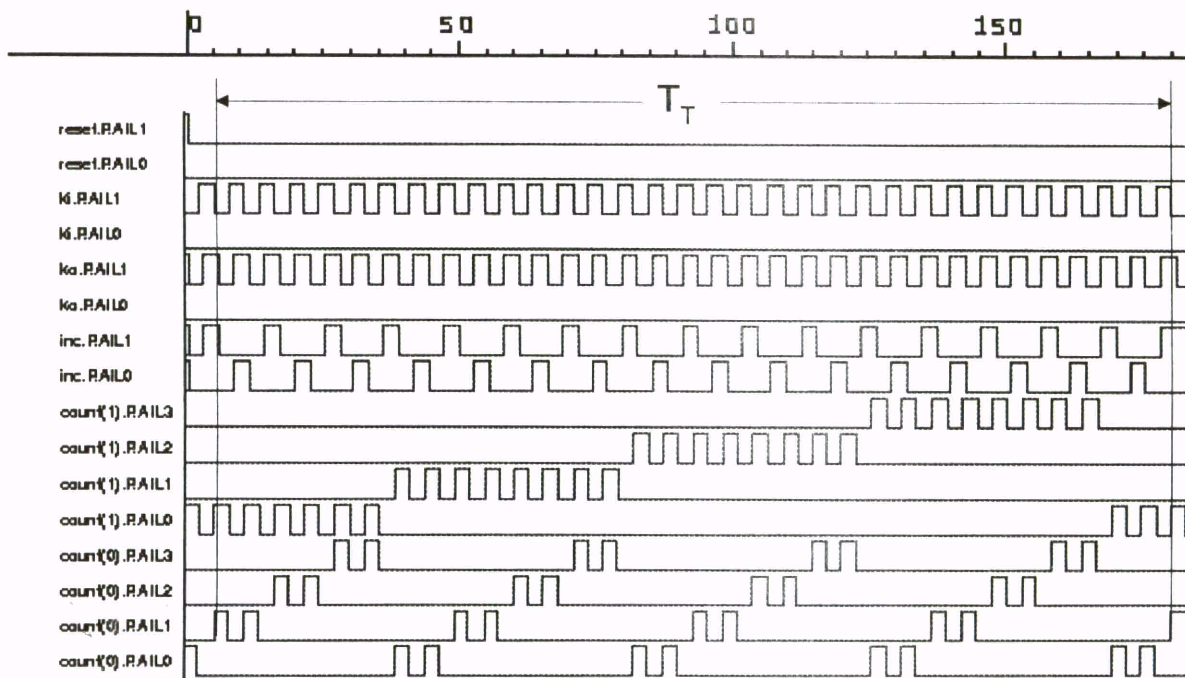

Figure 39. Quad-rail 4-bit counter waveforms.

3.5.1 Method 1: Incomplete Functions

This technique was applied to the optimized Boolean increment circuitry of the 4-bit counter shown in Figure 40, which is based on a carry look-ahead adder. The Boolean XOR gates were replaced with the XOR function described in Section 3.3, and the Boolean AND gates were replaced with incomplete versions of the AND function shown in Figure 10. The resulting logic diagram is depicted in Figure 41. The completeness of input criterion for the circuit as a whole is satisfied since all of the inputs are needed to produce a complete output set, due to the inherent completeness of input of an XOR function. This model has a worse-case path delay of two NCL gates in the increment circuitry. It consists of 14 NCL gates and T_{DD} was determined to be 4.81 ns using Synopsys.

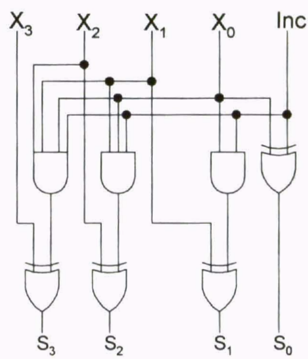

Figure 40. Boolean increment circuit.

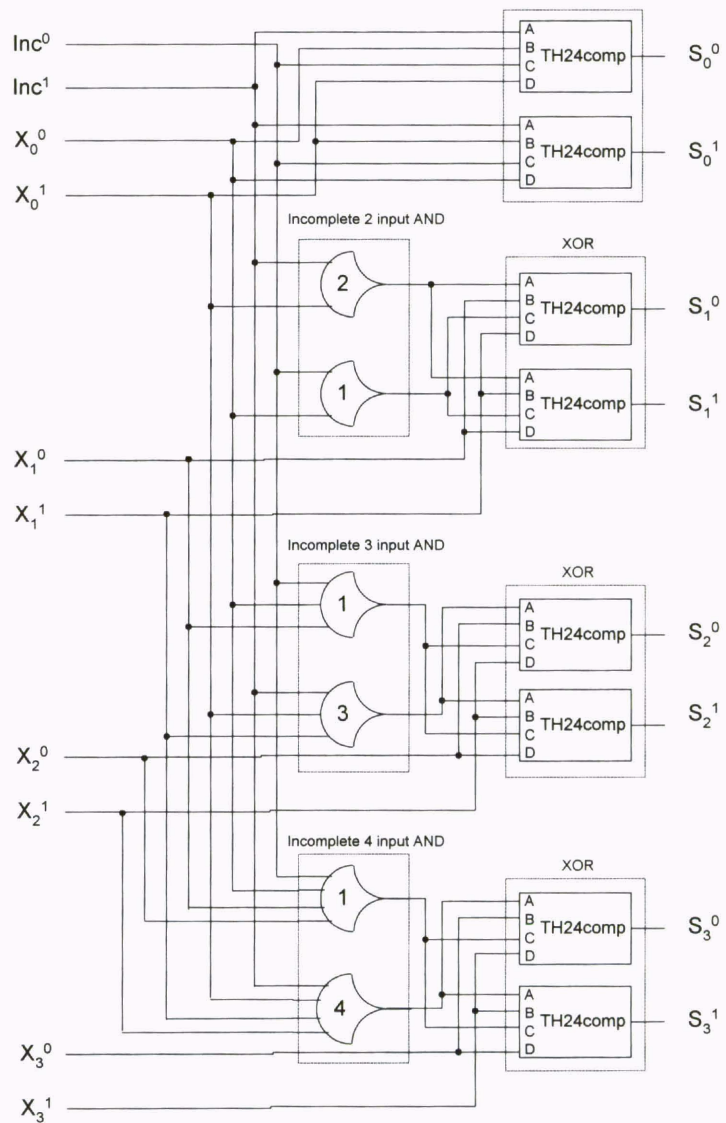

Figure 41. Increment circuit using incomplete AND functions.

3.5.2 Method 2: Dual-Rail Encoding Optimizations

The resulting logic diagram after deriving reduced minterm expressions from Step 2A is shown in Figure 42. This model has a theoretical worse-case path delay of 2 threshold gates in the increment circuitry. However, TH15 and TH55 gates are not supported in the 27 NCL macros, since they require 5 inputs. Therefore, the TH15 gate was realized by connecting a TH14 gate in series with a TH12 gate. However, this technique could not be applied to the TH55 gate, since this decomposition would violate the observablity criterion. Instead the two TH55 gates were decomposed into one TH44 gate and two TH22 gates, in order to maintain observability of every gate transition at the output. This decomposition is valid since every transition of the TH44 gate will result in exactly one of the two TH22 gates also transitioning. The decompositions caused the worse-case path delay to be three NCL gates, instead of two. The reduced minterm model consists of 39 gates, but only 36 gates are necessary if TH55 and TH15 gates are used. From Synopsys simulation, T_{DD} was determined to be 5.34 ns.

To further reduce the gate count, the expressions for S_1, S_2, and S_3 can be factored using Step 2B. This factoring increases the worse-case path delay from two NCL gates to three NCL gates. Since constructing TH55 and TH15 gates for the reduced minterm model from smaller gates caused a worse-case path delay of 3 threshold gates, factoring did not increase the depth of the critical path. The logic diagram for the increment circuitry factored form is shown in Figure 43. The factored minterm model consists of 28 gates, but only 27 gates are necessary if TH55 gates are used. From Synopsys simulation, T_{DD} was determined to be 5.28 ns.

UCF LIBRARY, ORLANDO, FL 32816

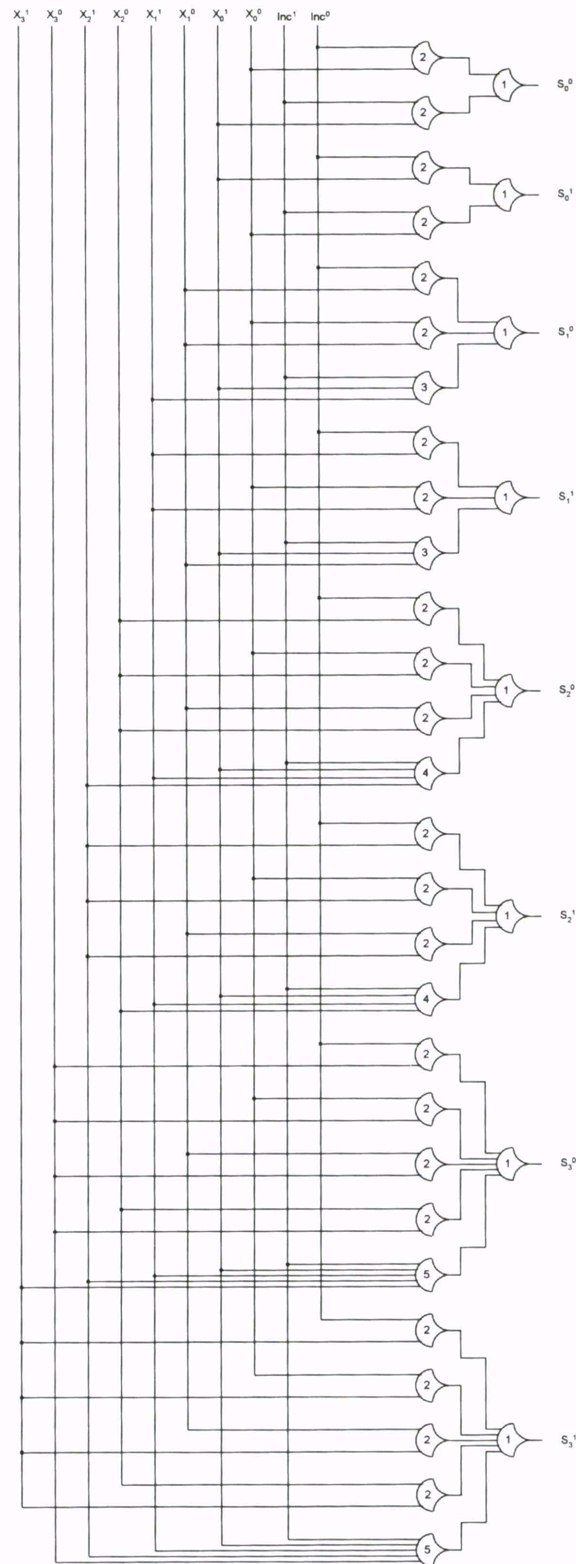

Figure 42. Increment circuit using dual-rail reduced minterm expressions.

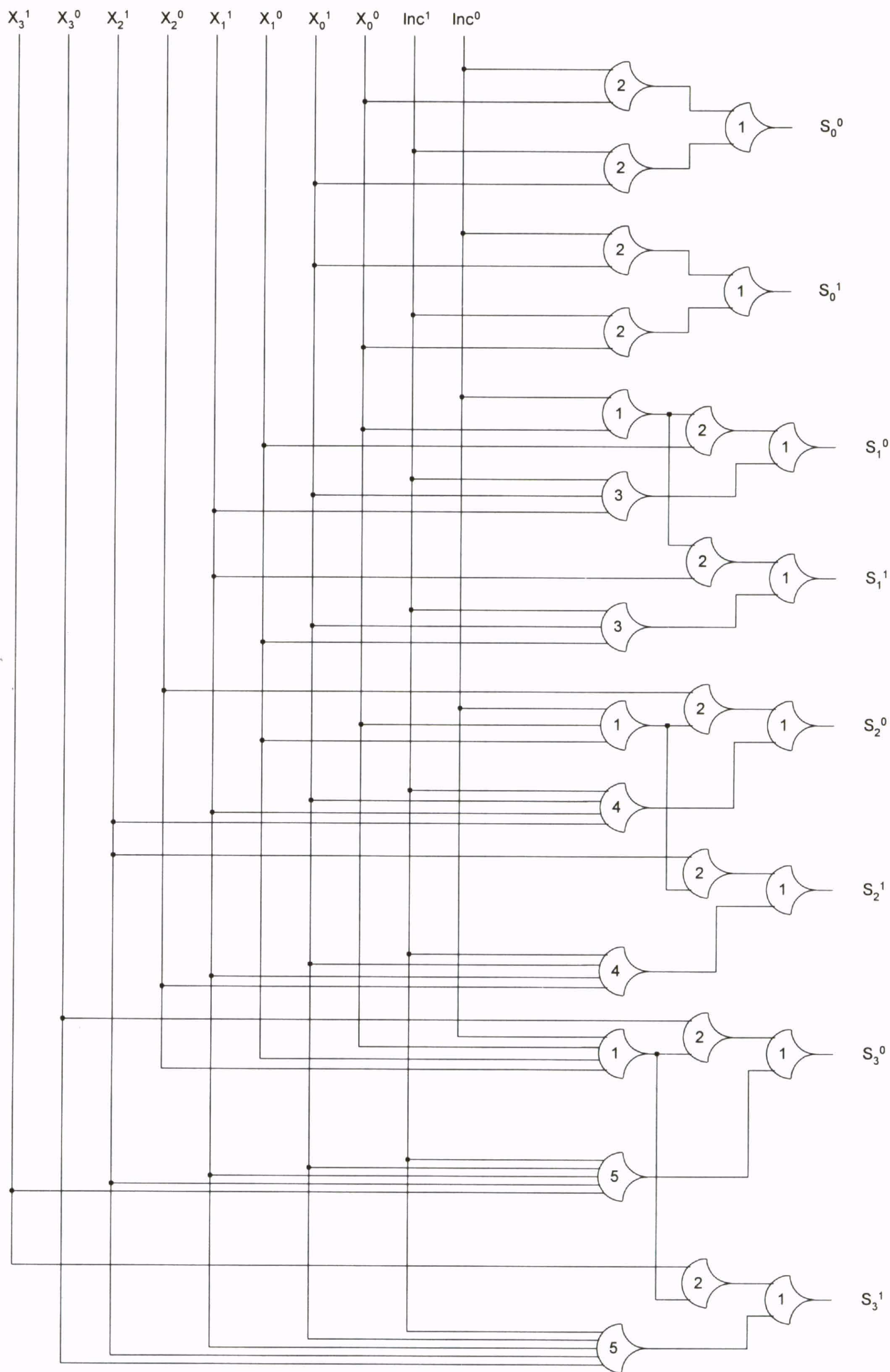

Figure 43. Increment circuit using dual-rail factored minterm expressions.

Step 2C maps the factored expressions to the full 27 macros in Table III, reducing both the number of gates and the number of logic levels. Note that the expressions for S_0, S_2, and S_3 can be mapped to TH24comp gates by adding two *don't care* terms as for the XOR function explained in Section 3.3. The logic diagram for the increment circuitry using complex gates is shown in Figure 44. It has a worse-case path delay of two NCL gates in the increment circuitry. The complex dual-rail model consists of 13 gates, and from Synopsys simulation T_{DD} was determined to be 4.81 ns.

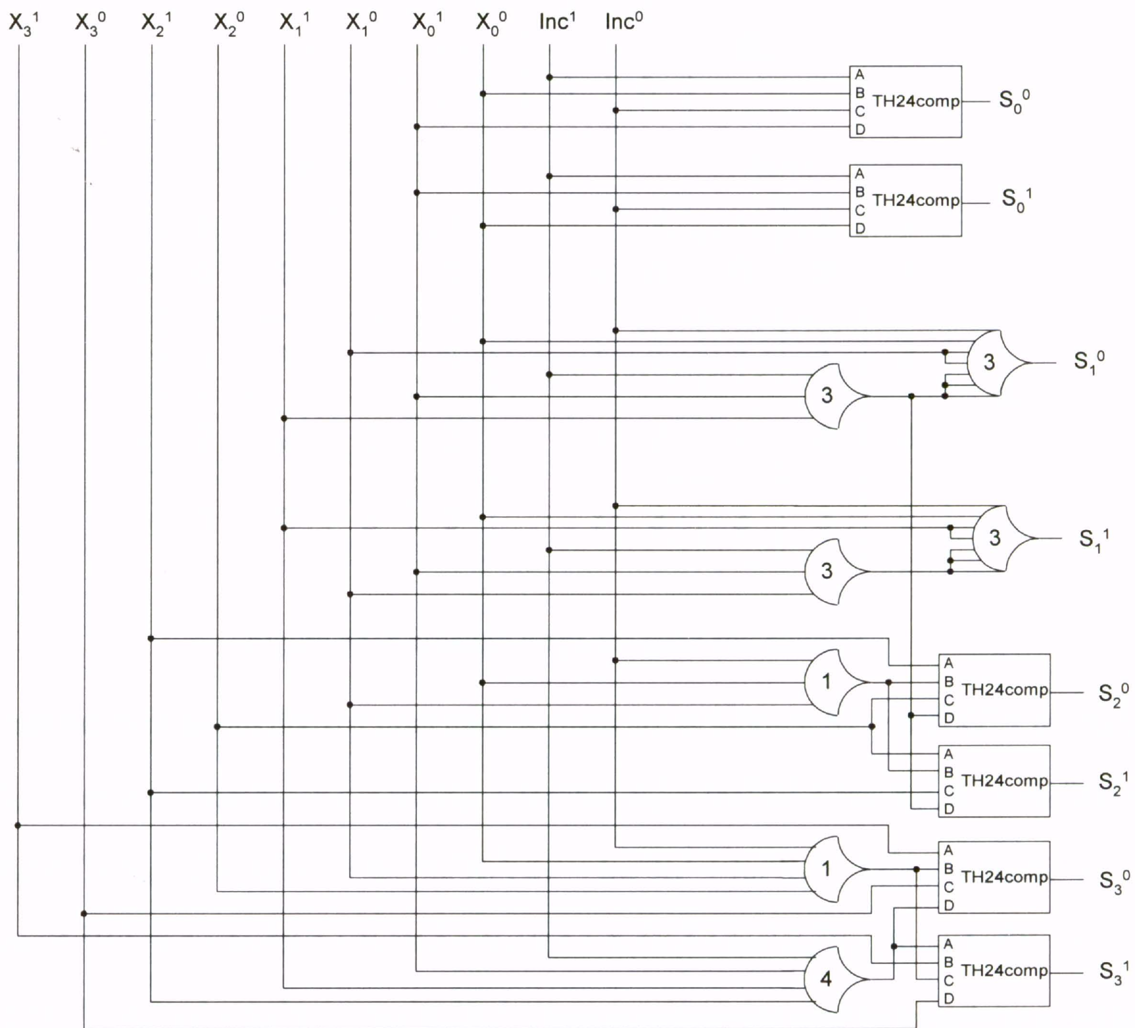

Figure 44. Dual-rail increment circuit using complex gates.

3.5.3 Method 3: Quad-Rail Encoding Optimizations

Quad-rail optimizations proceed in a similar fashion to dual-rail optimizations. In Step 3A, the NCL circuit is optimized by using reduced minterm expressions for all four rails of both outputs, S_0 and S_1, the low order two bits and the high order two bits, respectively, derived from the Karnaugh maps shown in Figure 45. Note that not all of the coverings that eliminate *Inc* are shown, so as not to clutter the drawing. The reduced

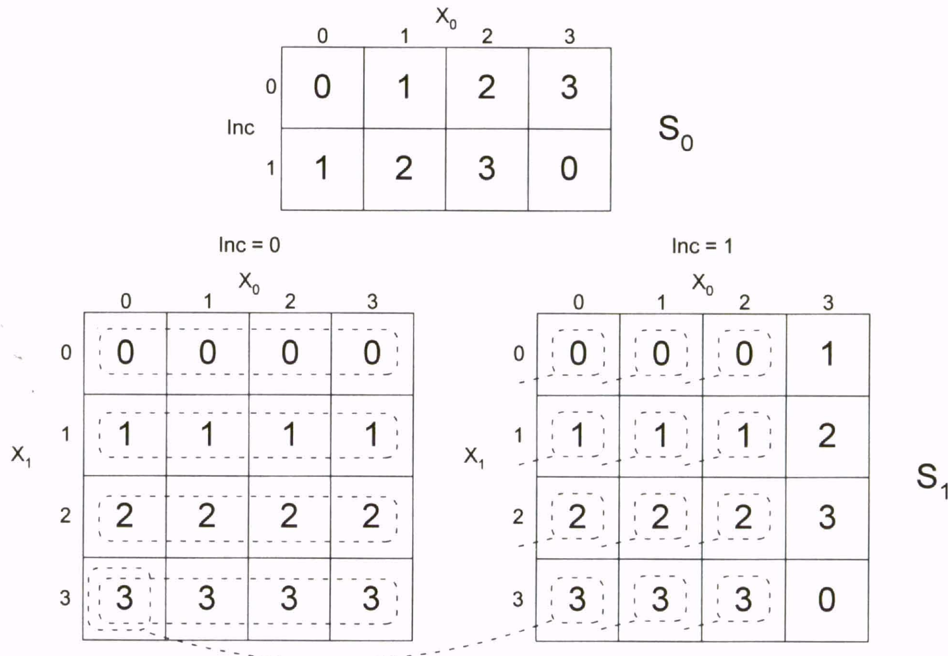

Figure 45. Karnaugh maps for quad-rail counter.

minterm expressions derived from these K-maps are as follows: $S_0^0 = Inc^0X_0^0 + Inc^1X_0^3$, $S_0^1 = Inc^0X_0^1 + Inc^1X_0^0$, $S_0^2 = Inc^0X_0^2 + Inc^1X_0^1$, $S_0^3 = Inc^0X_0^3 + Inc^1X_0^2$, $S_1^0 = Inc^0X_1^0 + X_0^0X_1^0 + X_0^1X_1^0 + X_0^2X_1^0 + Inc^1X_0^3X_1^3$, $S_1^1 = Inc^0X_1^1 + X_0^0X_1^1 + X_0^1X_1^1 + X_0^2X_1^1 + Inc^1X_0^3X_1^0$, $S_1^2 = Inc^0X_1^2 + X_0^0X_1^2 + X_0^1X_1^2 + X_0^2X_1^2 + Inc^1X_0^3X_1^1$, $S_1^3 = Inc^0X_1^3 + X_0^0X_1^3 + X_0^1X_1^3 + X_0^2X_1^3 + Inc^1X_0^3X_1^2$. These equations can now be directly mapped to TH1n and THnn gates to produce the reduced minterm model, shown in Figure 46. This

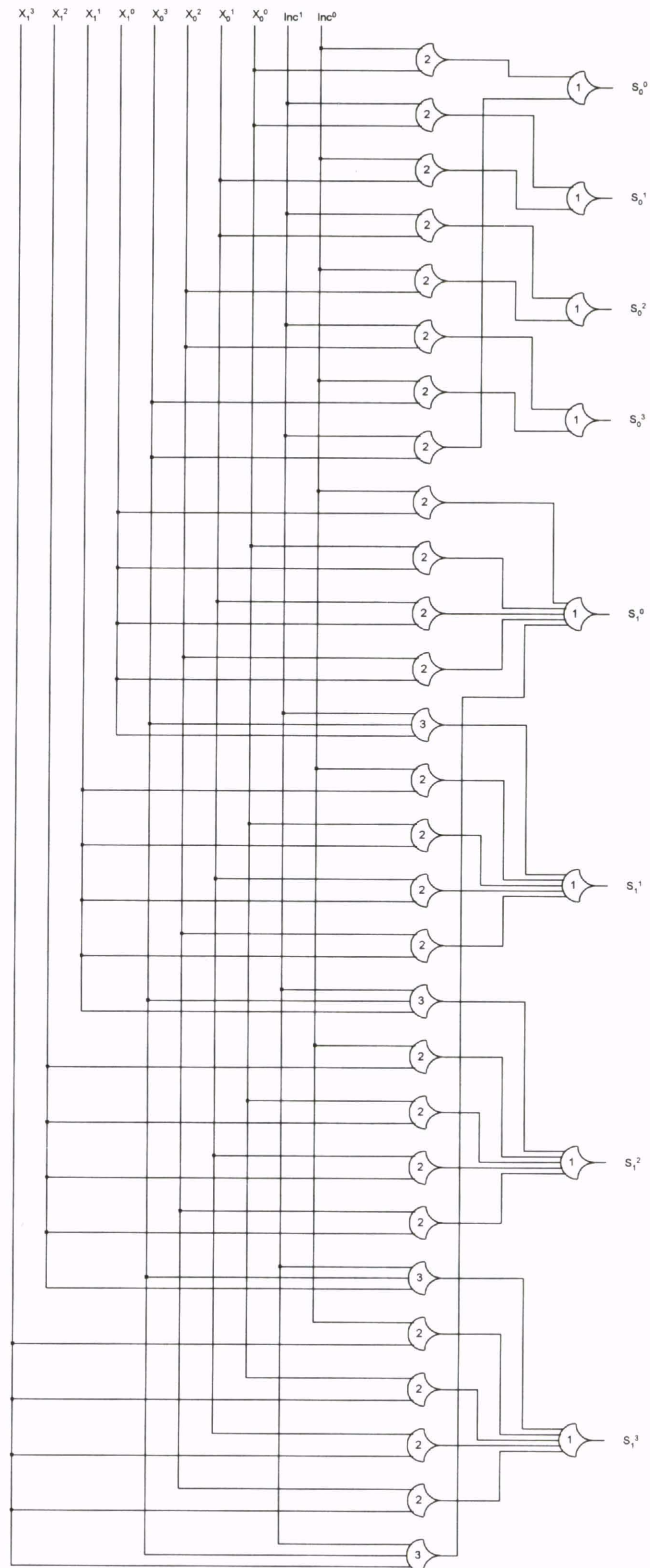

Figure 46. Increment circuit using quad-rail reduced minterm expressions.

model has a theoretical worse-case path delay of two NCL gates in the increment circuitry. However, TH15 gates are not supported in the 27 NCL macros, since they require 5 inputs. Therefore, the actual worse-case path delay is three NCL gates. The reduced minterm model consists of 40 gates, but only 36 gates are necessary if TH15 gates are used. From Synopsys simulation, T_{DD} was determined to be 5.59 ns.

To further reduce the gate count, the expressions for S_1 can be factored using Step 3B. This factoring increases the worse-case path delay from two NCL gates to three NCL gates. Since constructing TH15 gates for the reduced minterm model from smaller gates caused a worse-case path delay of 3 gates, factoring did not increase the depth of the critical path. The factored minterm model, shown in Figure 47, reduced the gate count to only 25 gates, and from Synopsys simulation, T_{DD} was determined to be 5.57 ns.

Step 2C maps the factored expressions to the full 27 macros in Table III, reducing both the number of gates and the number of logic levels. Note that the expressions for S_0 and S_1 can be mapped to TH24comp gates by adding two *don't care* terms as for the XOR function explained in Section 3.3. The logic diagram for the increment circuitry using complex gates is shown in Figure 48. It has a worse case path delay of two NCL gates in the increment circuitry. The complex quad-rail model consists of 10 gates and from Synopsys simulation T_{DD} was determined to be 5.47 ns.

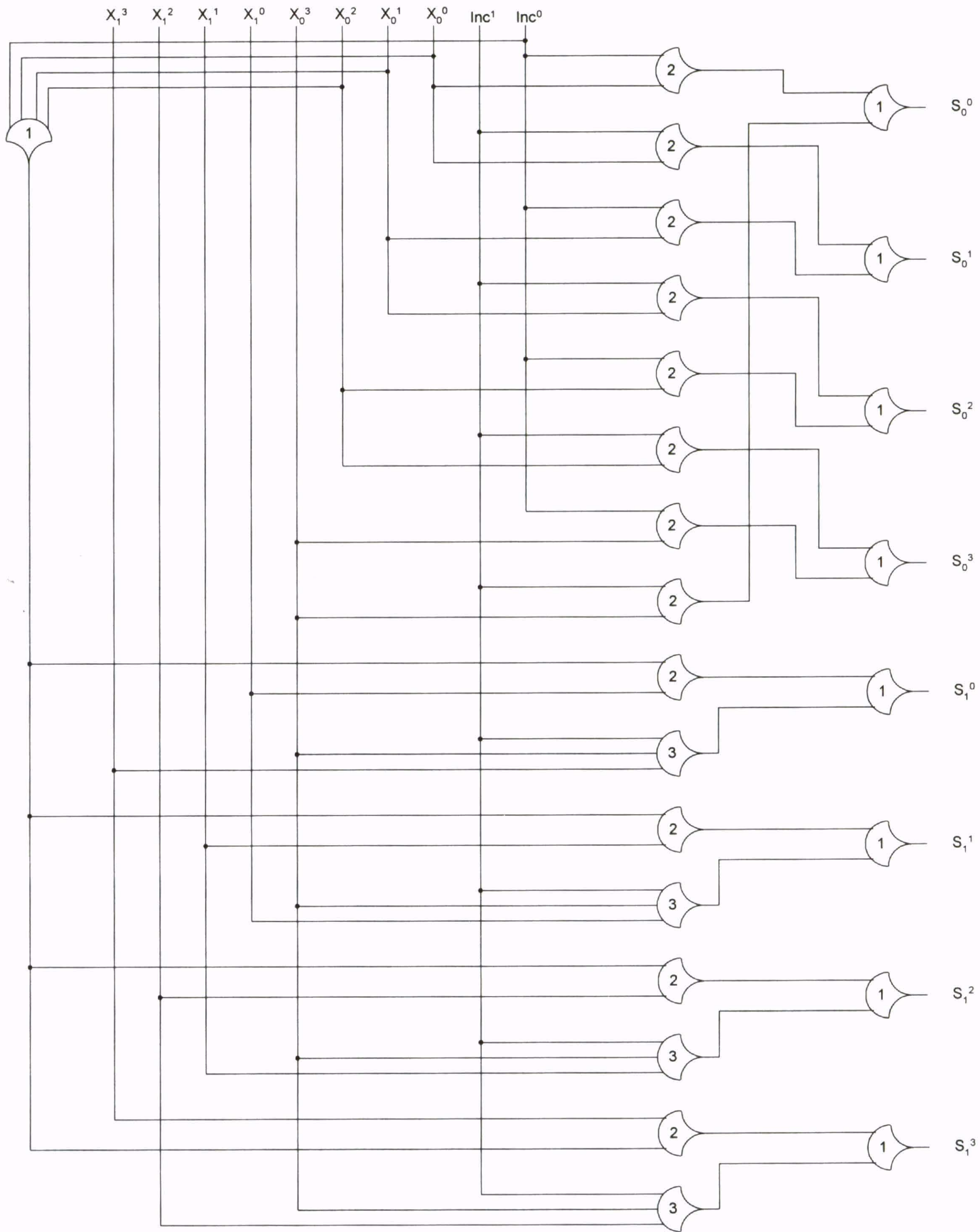

Figure 47. Increment circuit using quad-rail factored minterm expressions.

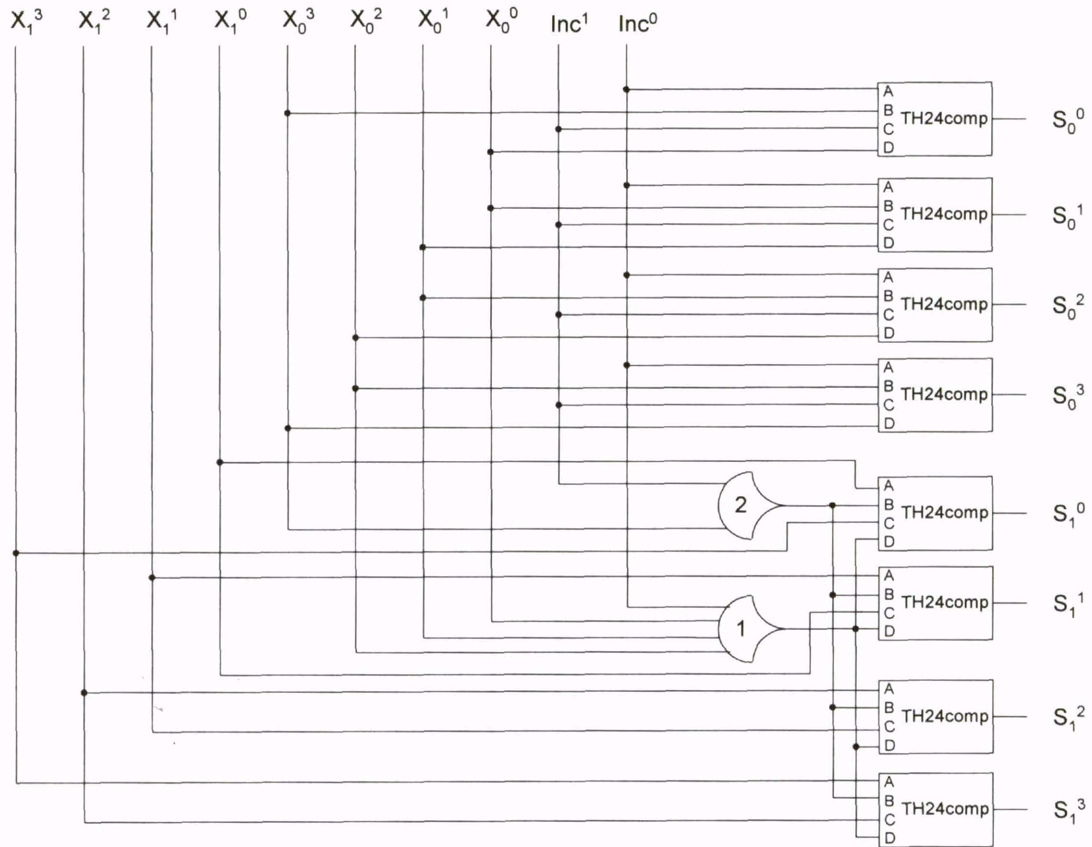

Figure 48. Quad-rail increment circuit using complex gates.

3.5.4 Other MEAG Optimizations

To reduce power, this technique was applied to design a 16-rail MEAG counter. The resulting increment circuitry is shown in Figure 49. Note that TH24comp gates can be used by adding two *don't care* terms as for the XOR function explained in Section 3.3. This model has a worse-case path delay of one NCL gate in the increment circuitry and consists of 16 NCL gates. However, a special 16-rail register, shown in Figure 50, was required to implement the feedback circuitry. The register is depicted as reset to NULL, however it could be instead reset to a DATA value by replacing exactly one of the TH22n

gates with a TH22d gate. This register requires two levels of logic to generate the K_o signal, instead of only one level required by both the dual-rail and quad-rail registers, causing the 16-rail MEAG counter to have a longer feedback path and therefore operate slower. Furthermore this 16-rail representation is exponential in the number of bits, reducing its applicability for general purpose designs. T_{DD} was determined to be 8.77 ns using Synopsys and the average power per cycle, denoted P_{DD}, was determined to be 5.37 μW using Cadence.

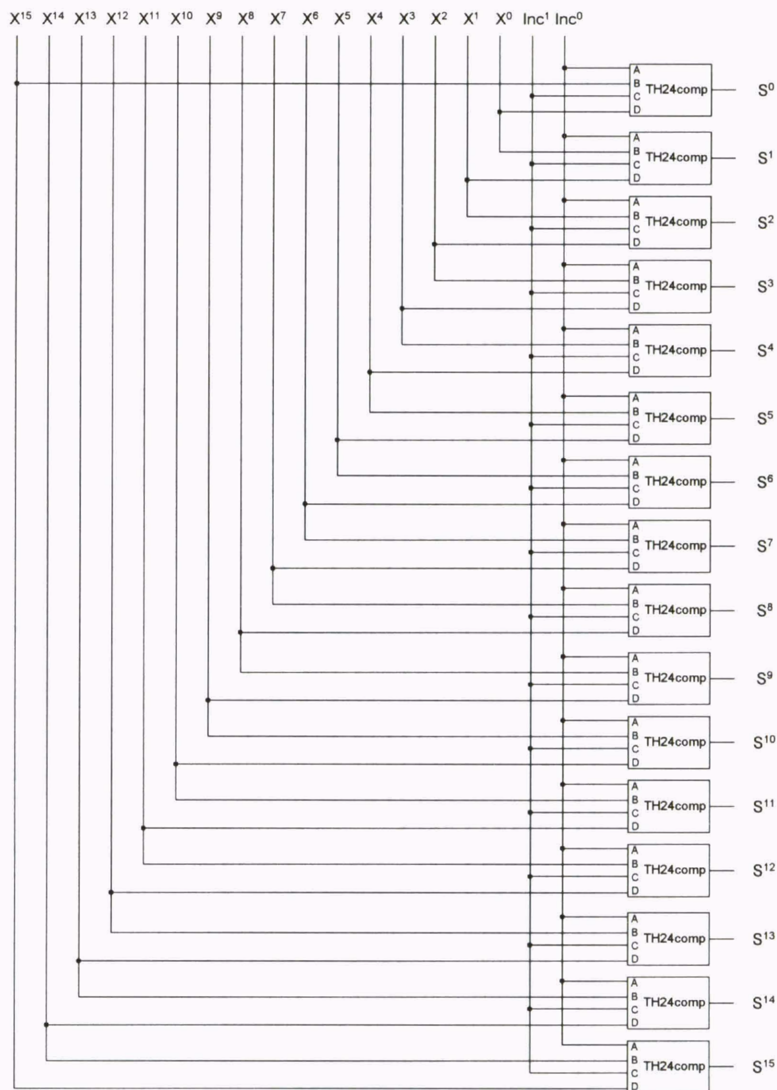

Figure 49. 16-rail MEAG increment circuit.

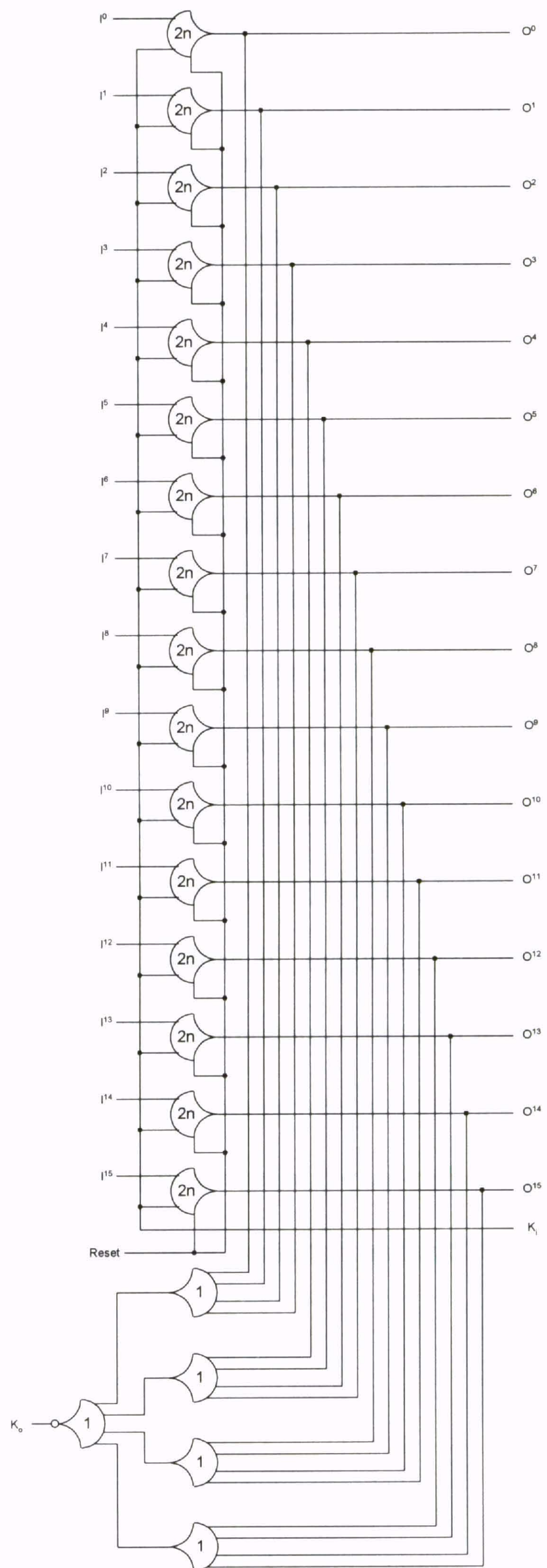

Figure 50. 16-rail MEAG register.

3.5.5 Up-Counter Performance Summary

Table VII lists the timing, gate counts, and transistor count for each of the eight counter models. It also lists the average power per operation for both the optimal dual-rail and quad-rail counters, as well as for the 16-rail MEAG counter. The *theoretical gate count* is the number of gates that would be required if TH55 and/or TH15 gates were used. Since these gates are not part of the 27 NCL macros, they have been constructed from existing gates, as discussed in Section 3.5.2, to yield the *actual gate count*. Table VII indicates that the factored forms of both the dual-rail and quad-rail circuits yield fewer gates and transistors, as well as smaller cycle times, compared to their original reduced forms. However, the complex gate models yield the best time and space performance for Method 2 and Method 3, as expected. The optimal design in terms of speed is generated from both Method 1 and Method 2C, although the design from Method 2C is preferred since it contains fewer gates and transistors. The most area efficient design is generated from Method 3C, requiring 22% fewer transistors than the speed optimal design of Method 2C. Furthermore, the most power efficient design is the 16-rail MEAG counter, requiring 63% less power than the optimal dual-rail design from Method 2C and 42% less power than the optimal quad-rail design from Method 3C, although it requires 36% and 73% more transistors and is 82% and 60% slower than the two, respectively.

Table VII. Alternate designs for NCL up-counter increment circuit.

Model Type	Theoretical Gate Count	Actual Gate Count	Transistor Count	T_{DD}	P_{DD}
1) Incomplete AND	14	14	216	4.81 ns	
2a) Reduced Dual-Rail	36	39	460	5.34 ns	
2b) Factored Dual-Rail	27	28	308	5.28 ns	
2c) Complex Dual-Rail	13	13	212	4.81 ns	14.44 μW
3a) Reduced Quad-Rail	36	40	440	5.59 ns	
3b) Factored Quad-Rail	25	25	266	5.57 ns	
3c) Complex Quad-Rail	10	10	166	5.47 ns	9.30 μW
16-rail MEAG	16	16	288	8.77 ns	5.37 μW

4.0 GATE-LEVEL PIPELINING OPTIMIZATIONS

Gate-Level Pipelining (GLP) techniques are developed to design throughput-optimal delay-insensitive NCL systems. Pipelined NCL systems consist of *Combinational, Registration*, and *Completion* circuits implemented using threshold gates equipped with hysteresis behavior. NCL *Combinational* circuits provide the desired processing behavior between *Asynchronous Registers* that regulate wavefront propagation. NCL *Completion* logic detects completed DATA or NULL output sets from each register stage. GLP techniques cascade registration and completion elements to systematically partition a combinational circuit and allow controlled overlapping of input wavefronts. Both full-word and bit-wise completion strategies are applied progressively to select the optimal size grouping of operand and output data bits. To illustrate the method, GLP is applied to a case study of a 4-bit by 4-bit unsigned multiplier, yielding a speedup of 2.25 over the non-pipelined version, while maintaining delay-insensitivity. Even though delay-insensitive design methods do not utilize clocked control signals, they are still amenable to significant throughput increases by pipelining of wavefronts. The objective of this chapter is to develop and illustrate a pipelining method for maximizing throughput of delay-insensitive systems at the gate level.

4.1 Chapter Outline

This chapter is organized into five sections. An overview of previous work is given in Section 4.2. In Section 4.3, the GLP method is developed. This method is then demonstrated in Section 4.4 by applying GLP to design an optimal 4-bit by 4-bit unsigned multiplier whose throughput is increased by 125% over the non-pipelined version. Section 4.5 concludes the 4×4 multiplier case study.

4.2 Previous Work

Pipelining facilitates temporal parallelism by partitioning a process into stages such that each stage operates simultaneously on different wavefronts of input operands. If a process that requires N time units can be partitioned into S identical stages then a steady-state throughput not to exceed S/N results per time unit may be realized. In practice numerous constraints, such as registration overhead between computational stages, limit the actual speedup achievable by pipelining. For instance, throughput limitations may be encountered as clocked Boolean circuits are partitioned to increasingly finer granularities. In particular, the clock period used to advance data between stages becomes increasingly dominated by the required design margins, including accommodations for clock skew. Clearly, asynchronous design methods need not provide design margins to accommodate clock skew. Nonetheless, they do possess their own constraints governing speedup by pipelining and can benefit substantially from optimized pipeline design strategies.

One approach to pipelining asynchronous circuits was described in Ivan Sutherland's work on *micropipelines* [19]. This method employs *two-phase handshaking* supporting transmission of *bundled data*. Figure 51 shows a two-phase handshaking protocol. Two control wires, labeled *request* and *acknowledge*, are used to support an arbitrary number of data wires. In two-phase handshaking, both the rising and falling edges of the request and acknowledge signals are indicative of circuit behavior. A cycle begins with the sender setting the data lines and generating a request event by toggling the request line. When the request is received, the data is latched and the receiver generates an acknowledge event by toggling the acknowledge line. The cycle terminates when the sender receives the acknowledge signal, at which time the data lines may be set for the next cycle. The use of bundled data refers to the fact that the data lines and request signal are treated as a bundle. Data bundling implies that the data transmission delay cannot exceed the delay to transmit the request. Otherwise, the request event might reach the receiver prior to valid data, causing invalid data to be latched. Subsequent work on micropipelines [31, 32, 33] suggest that performance may be increased by using four-phase handshaking protocols. Four-phase handshaking also requires two control wires, *request* and *acknowledge*, along with an arbitrary number of data wires. But, in four-phase handshaking only one edge, either the rising or falling edge of the request and acknowledge signals, is active. The four-phase handshaking protocol is shown in Figure 52, using the rising edge as active. A cycle begins with the sender placing data on the bus and generating a request event by asserting the request line. When the request is received, the data is latched and the receiver generates an acknowledge event by asserting

UCF LIBRARY ORLANDO FL 32816

the acknowledge line. When the sender receives the acknowledge signal, the request signal is de-asserted and the data lines may be set for the next cycle. The cycle concludes with the acknowledge line being de-asserted, as precipitated by the de-assertion of the request line. Micropipelining techniques such as these are evident in several processors that have been designed and implemented using bundled data methods [34, 35].

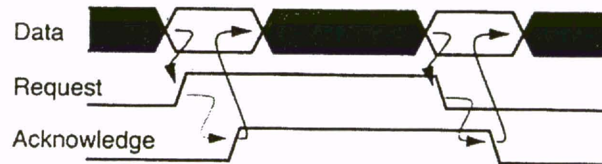

Figure 51. Two-phase handshaking protocol [19].

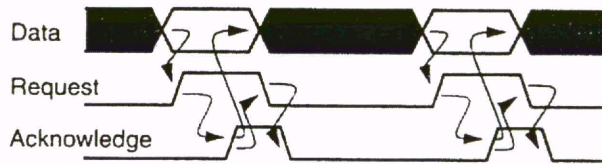

Figure 52. Four-phase handshaking protocol [33].

Another approach to pipelining asynchronous circuits is through the use of *wave pipelining*. Hauck and Huss [36] describe a technique that allows multiple data wavefronts to simultaneously propagate between two asynchronous registers by partitioning each combinational logic block with dynamic latches, controlled only by the request line. Synchronous wave pipelining and asynchronous micropipelining methods can be combined using these techniques. However, a potential limitation of eliminating the acknowledge signal is that delay-insensitive behavior may be compromised, thus making the protocol inelastic. Further work by Park and Chung [37] presents a modification to this approach in which both the number of latches and the number of delay elements can be reduced, resulting in higher throughput.

A third asynchronous pipelining approach uses delay-insensitive *multi-ring structures* [38]. This method employs a four-phase handshaking protocol using dual-rail signals for data representation and Delay-Insensitive Minterm Synthesis (DIMS) [9] for each functional block. It also presents a formal method for analyzing the performance of these multi-ring structures, based on signal transition graphs. Nonetheless, formal methods to design throughput-optimal multi-ring structures are not directly feasible due to underlying difficulties in partitioning of DIMS expressions.

In [39] Kim and Beerel present an optimal branch and bound algorithm to partition asynchronous circuits composed of precharge-logic blocks [12, 24] designed at the transistor level. The algorithm uses a labeled directed graph to represent the model being pipelined. However, this method is not directly amenable to pipelining NCL circuits due to the differences in the fundamental components.

4.2.1 Relation of NCL to Previous Work

For Sutherland's *micropipelines* using either two-phase or four-phase handshaking, the determination of the maximum throughput design for a given combinational circuit is straightforward. Since micropipelines assume bundled data and therefore employ single-rail signals, there is no completeness of input criterion that must be met when partitioning a circuit, therefore further partitioning cannot invalidate a design. Furthermore, delay is added in the control path such that completion detection is unnecessary, therefore further partitioning cannot decrease throughput. Thus, the design that will yield the maximum throughput is the one containing only one gate delay per

stage. Since micropipelines necessitate the addition of delay in the control path, they exhibit worse-case performance verses the average-case performance of NCL systems and are layout and process dependent unlike NCL systems. Micropipelines also assume bundled data such that synchronicity is required, while NCL systems require no synchronization so that inputs may arrive at any time and in any order. Therefore, NCL systems are potentially more independent than micropipelines.

Since the maximum throughput rate for *asynchronous wave pipelines* is determined by the difference between the longest and shortest path through the combinational logic, there is even more timing analysis required than for micropipelines. In asynchronous wave pipelines throughput will be maximized by designing the shortest and longest path to be nearly equal, therefore extensive timing analysis is required. Asynchronous wave pipelines are therefore very susceptible to process dependencies and environmental variations, unlike NCL. These fundamental differences between NCL and both micropipelines and asynchronous wave pipelines place NCL in a different class than either and would make direct comparisons difficult.

NCL circuits are in the same class as other delay-insensitive approaches [4, 6, 7, 8, 9], that were compared to NCL in Chapter 3. The functionality of NCL circuits is the same as those designed using the approaches presented in [4, 6, 7, 8, 9]. Thus, the NCL combinational circuit, as part of the NCL gate-level pipelining framework, could be replaced with an equivalent circuit designed using [4, 6, 7, 8, 9], and the resulting single-stage system would function correctly. This is exactly what delay-insensitive multi-ring structures are. Their framework is equivalent to that of NCL, except for the

combinational circuits, which use the approach described in [9]. But, since all of the basic gates used in the other delay-insensitive approaches, including delay-insensitive multi-ring structures, do not include hysteresis, their combinational designs cannot be partitioned, as can NCL combinational circuits. Thus, a given combinational circuit designed using [4, 6, 7, 8, 9] can either be used as a non-pipelined design, or if increased throughput is desired, each stage of the pipeline must be separately redesigned. Therefore a method which iteratively divides a combinational circuit of a delay-insensitive multi-ring structure to increase throughput cannot do so with little effort, as does the method presented herein for NCL; since after each iteration all combinational blocks which were divided would have to be redesigned to include input-completeness necessary for delay-insensitivity.

4.3 Method Definition

In Chapter 3 it was shown how to design an optimal NCL combinational circuit. So, starting with an N-level NCL combinational logic circuit, the design process for optimizing throughput begins, as depicted in Figure 53. Other criteria such as maximum latency and maximum area may also be considered during throughput optimization. Several alternate designs are generated which are then assessed against the optimization criteria, allowing the preferred design to be selected for implementation.

It is assumed that if a maximum latency bound is specified then it is at least one stage, and that if a maximum area bound is specified then it is at least as large as the non-pipelined design, otherwise the non-pipelined design will be output. If no maximum

latency or maximum area requirements are specified, then both are assumed to be infinity such that they are not considered in determining the optimal design. If more than one design has the same throughput, the one with the least latency will be chosen. If multiple designs have the same throughput and latency, the one with the least area will be chosen.

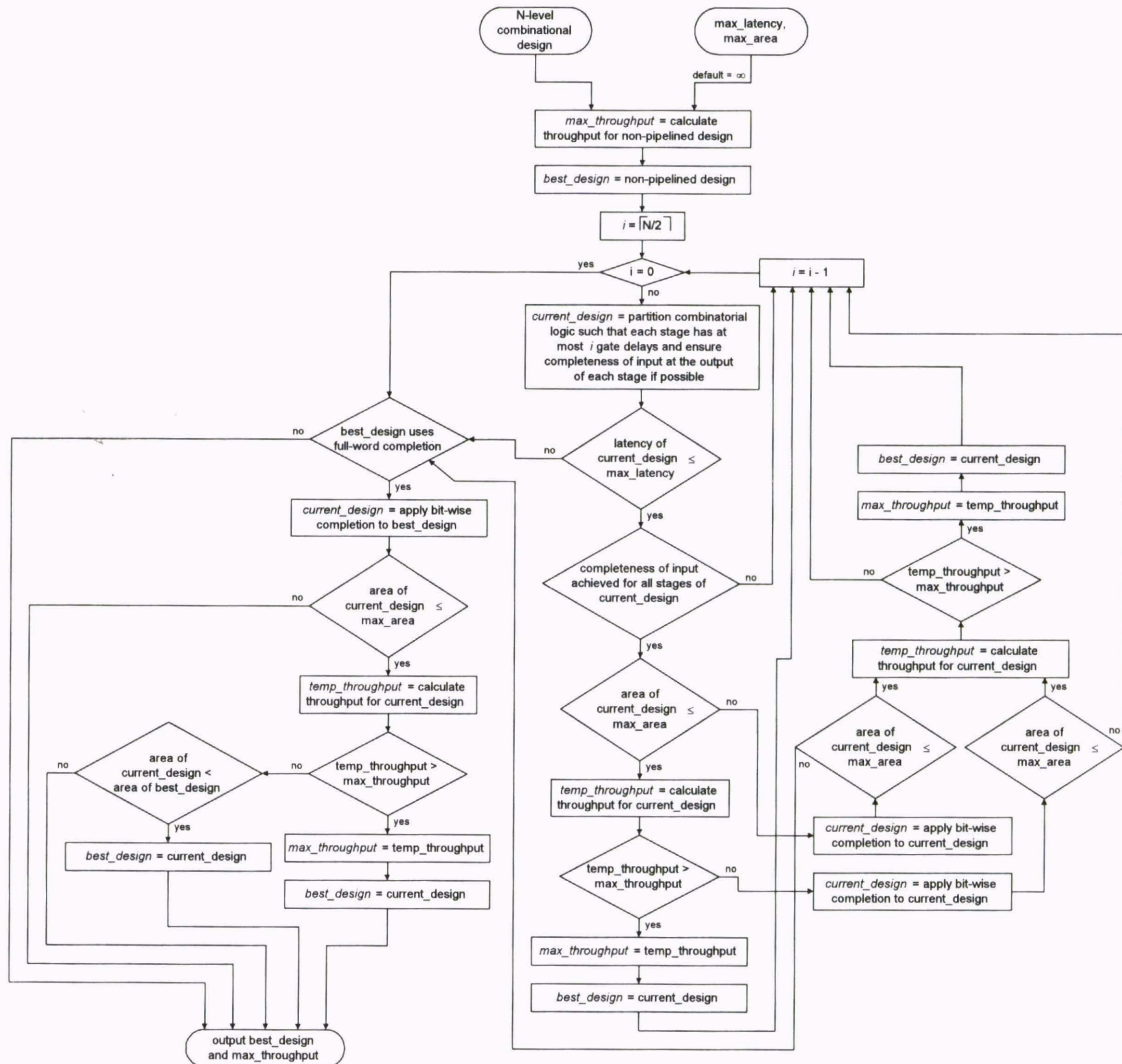

Figure 53. GLP design flow.

The original combinational circuit with no pipelining will always be input-complete since TCR only yields input-complete designs. Thus, starting with the combinational logic design and adding registration along with corresponding completion logic at the input and output will yield an initial 1-stage design. Partitioning this initial design, first into 2 stages, then further into as many as *N* stages may or may not produce better designs. First, completeness of input must be ensured at the output of each stage, as discussed in Chapter 2, otherwise the design will not be delay-insensitive and is therefore invalid. After input-completeness is ensured, the throughput for the current design must be calculated and compared to the throughput of the best design. If the current design's throughput is greater than that of the best design, it is designated as the best design, otherwise bit-wise completion is applied to the current design and the throughput is reevaluated. If the throughput of the current design using bit-wise completion is still not greater than that of the best design, the best design does not change since the current design doesn't increase throughput and has longer latency, otherwise the current design using bit-wise completion becomes the best design. As mentioned in Chapter 2 the completion delay is proportional to $\lceil \log_4 N \rceil$. Thus, if partitioning causes registers of significantly larger width to be required then the decrease in the combinational delay per stage will be offset by the increase in the completion delay such that the throughput of the system may not necessarily increase, as discussed in Section 4.3.1. If after traversing the loop of Figure 53 (`i=0`), which generates each subsequent pipelined design, or if the maximum latency or area requirements have been exceeded, then if the best design utilizes full-word completion, bit-wise completion is applied to this design to possibly

further increase throughput. If throughput is not increased the design with the least area is chosen since both designs will have the same throughput and latency. This is because application of bit-wise completion won't decrease throughput, as explained in Section 4.3.2, and doesn't impact the number of stages. The output of this flowchart will be the optimal design (`best_design`) that produces the maximum throughput (`max_throughput`), and does not exceed the maximum latency or maximum area requirements, if any were given.

4.3.1 Throughput Derivation

Quarter-cycle timing is used to determine the worst-case achievable throughput of a pipelined NCL system. The name is derived from the fact that the analysis requires each NCL cycle to be broken into its four sub-cycles. The NCL cycle is comprised of the DATA and NULL propagation through the combinational circuitry, as well as the generation of the request for DATA and request for NULL from the completion circuitry. The four sub-cycles that are contained in the NCL cycle are shown in Figure 54. *D* denotes the interval when any DATA bits are propagating through the combinational circuit, *N* denotes the interval when any NULL bits are propagating through the combinational circuit, *RFD* is the request for DATA generation, and *RFN* is the request for NULL generation. Assuming $K_o = rfd$, the cycle starts with DATA propagation and the sequence of the four sub-cycles is as follows: D, RFN, N, and RFD. The propagation delays associated with this sequence are labeled as follows: TD, TRFN, TN, and TRFD, respectively. TD and TN are defined to be the delay experienced by the slowest bit

through their respective sub-cycles. In this chapter TD, TRFN, TN, and TRFD are calculated in terms of gate delays, making the predicted throughput an estimate since different gates do have slightly different delays. If this method were to be automated, the actual delay of each gate would be used to calculate the predicted throughput.

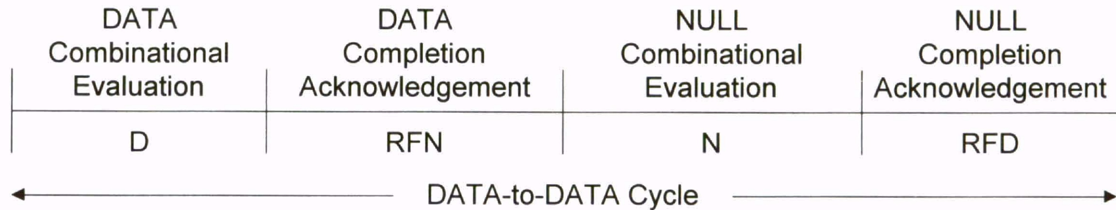

Figure 54. Sub-cycles of the NCL cycle.

The NCL cycle is bounded by the current registration stage, denoted as *i*, and the previous registration stage, denoted by *i-1*, as depicted in Figure 55. The calculation resulting in the maximum cycle time forms a lower bound on the throughput of the ith and $i\text{-}1^{th}$ registration pair. This process of bounding the throughput for registration pairs is repeated for all adjacent registration pairs in a pipelined configuration. The maximum value calculated over all adjacent registration pairs determines a lower bound on steady-state throughput for the entire design.

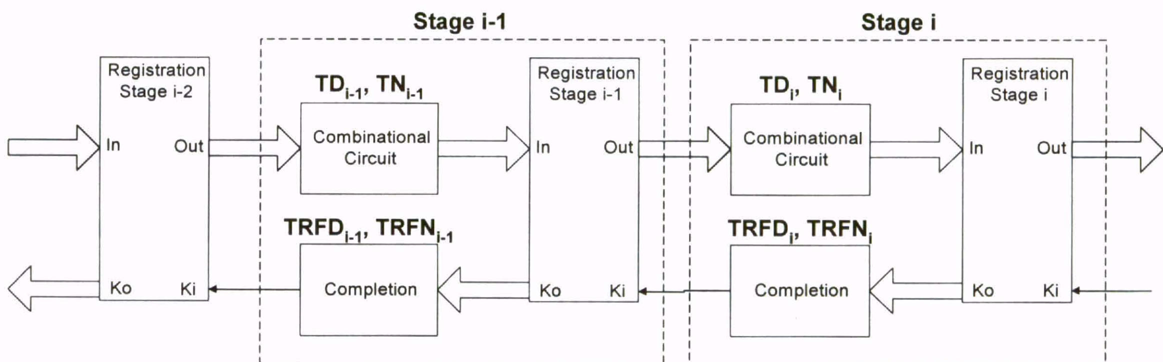

Figure 55. Pipeline showing NCL sub-cycle times.

4.3.1.1 Idealized Completion Circuitry

Consider the idealized case where TRFN and TRFD are assumed to be zero. The discrete timing chart in Table VIII identifies the interaction of $stage_i$ and $stage_{i-1}$ under these idealized conditions. For the initial state, the analysis begins with $stage_i$ and $stage_{i-1}$ both reset to NULL. At wavefront #1, DATA propagates through the combinational circuitry of $stage_{i-1}$, while $stage_i$ remains idle. At wavefront #2, NULL propagates through the combinational circuitry of $stage_{i-1}$, while DATA propagates through the combinational circuitry of $stage_i$. At wavefront #3, DATA propagates through the combinational circuitry of $stage_{i-1}$, while NULL propagates through the combinational circuitry of $stage_i$. This pattern of NULL propagating through $stage_{i-1}$, while DATA propagates through $stage_i$, followed by DATA propagating through $stage_{i-1}$, while NULL propagates through $stage_i$, repeats continuously and forms the simplified NCL cycle, shown in boldface in Table VIII.

Table VIII. Discrete timing chart for the idealized NCL cycle.

Stage	Sub-cycle	Initial State	Wavefronts				
			1	**2**	**3**	4	5
i-1	D_{i-1}		X		**X**		X
	N_{i-1}	X		**X**		X	
i	D_i			**X**		X	
	N_i	X			**X**		X

Using the above terminology, the worst-case DATA-to-DATA cycle time for $stage_i$ assuming idealized completion is:

$$T_{DD_i}^{idealized} = MAX\ (TN_{i-1}, TD_i) + MAX\ (TD_{i-1}, TN_i) \qquad \text{(eq. 4.1).}$$

Interpreting Equation 4.1 as a set of exclusive events implies exactly one of the following relationships:

either $T_{DD_i}{}^{idealized} = TN_{i-1} + TD_{i-1}$ (eq. 4.2), or

$T_{DD_i}{}^{idealized} = TN_{i-1} + TN_i$ (eq. 4.3), or

$T_{DD_i}{}^{idealized} = TD_i + TD_{i-1}$ (eq. 4.4), or

$T_{DD_i}{}^{idealized} = TD_i + TN_i$ (eq. 4.5).

Notice that Equations 4.2 and 4.5 are equivalent except for their stage index. Under the proposed method of evaluating each stage pair in increasing order to determine the global maximum value, Equation 4.2 would therefore have been evaluated in the previous registration pair calculations, so it does not need to be reevaluated in the current registration pair calculations. This is true for every registration pair except the first pair, stage 1 and stage 2. For the first registration pair, Equation 4.2 does need to be considered since there is no previous registration pair that incorporates this calculation.

Equation 4.3 considers the case of adjacent NULL propagation delays. Equation 4.4 considers the case of adjacent DATA propagation delays. Equation 4.5 considers the case of NULL and DATA propagation delays for a single registration stage. The pseudocode listed in Algorithm 4.1 calculates the worst-case throughput for an idealized N-stage NCL pipeline.

```
max_cycle_time = TD1 + TN1
for (i = 2 to N) loop
    temp_cycle_time = MAX(TNi-1 + TNi, TDi-1 + TDi, TDi + TNi)
    if (temp_cycle_time > max_cycle_time) then
        max_cycle_time = temp_cycle_time
    end if
end loop
worst_case_throughput = 1 / max_cycle_time
```

Algorithm 4.1. Calculation of worst-case throughput for an idealized N-stage pipeline.

Evaluation of the above loop is followed by taking the reciprocal of the maximum adjacent stage pair delay to obtain a lower bound on the pipeline's throughput.

4.3.1.2 Non-Zero Delay Completion Circuitry

Now the general case will be examined, where TRFN and TRFD are not zero. The discrete timing chart in Table IX shows the interaction of $stage_i$ and $stage_{i-1}$. For the initial state, assume $stage_i$ and $stage_{i-1}$ are both reset to NULL, so both stages will initially be requesting DATA. At wavefront #1, DATA propagates through the combinational circuitry of $stage_{i-1}$, while $stage_i$ remains idle. At wavefront #2, DATA propagates through the combinational circuitry of $stage_i$, while $stage_{i-1}$ requests NULL. At wavefront #3, NULL propagates through the combinational circuitry of $stage_{i-1}$, while $stage_i$ requests NULL. At wavefront #4, NULL propagates through the combinational circuitry of $stage_i$, while $stage_{i-1}$ requests DATA. At wavefront #5, DATA propagates through the combinational circuitry of $stage_{i-1}$, while $stage_i$ requests DATA. This pattern, from wavefront #2 to wavefront #5, repeats continuously and forms the generalized NCL cycle, shown in boldface in Table IX.

Table IX. Discrete timing chart for the general NCL cycle.

Stage	Sub-cycle	Initial State	Wavefronts								
			1	2	3	4	5	6	7	8	9
i-1	D_{i-1}		X				X				X
	N_{i-1}	X			X				X		
	RFD_{i-1}	X				X				X	
	RFN_{i-1}			X				X			
i	D_i			X				X			
	N_i	X				X				X	
	RFD_i	X					X				X
	RFN_i				X				X		

The worst-case cycle time for the generalized case of $stage_i$ is then given by:

$$T_{DD_i} = MAX(TD_i, TRFN_{i-1}) + MAX(TN_{i-1}, TRFN_i) + MAX(TN_i, TRFD_{i-1}) + MAX(TD_{i-1}, TRFD_i) \quad \text{(eq. 4.6).}$$

Interpreting Equation 4.6 as a set of exclusive events implies exactly one of the following relationships:

either $$T_{DD_i} = TD_i + TN_{i-1} + TN_i + TD_{i-1} \quad \text{(eq. 4.7), or}$$

$$T_{DD_i} = TD_i + TN_{i-1} + TN_i + TRFD_i \quad \text{(eq. 4.8), or}$$

$$T_{DD_i} = TD_i + TN_{i-1} + TRFD_{i-1} + TD_{i-1} \quad \text{(eq. 4.9), or}$$

$$T_{DD_i} = TD_i + TN_{i-1} + TRFD_{i-1} + TRFD_i \quad \text{(eq. 4.10), or}$$

$$T_{DD_i} = TD_i + TRFN_i + TN_i + TD_{i-1} \quad \text{(eq. 4.11), or}$$

$$T_{DD_i} = TD_i + TRFN_i + TN_i + TRFD_i \quad \text{(eq. 4.12), or}$$

$$T_{DD_i} = TD_i + TRFN_i + TRFD_{i-1} + TD_{i-1} \quad \text{(eq. 4.13), or}$$

$$T_{DD_i} = TD_i + TRFN_i + TRFD_{i-1} + TRFD_i \quad \text{(eq. 4.14), or}$$

$$T_{DD_i} = TRFN_{i-1} + TN_{i-1} + TN_i + TD_{i-1} \quad \text{(eq. 4.15), or}$$

$$T_{DD_i} = TRFN_{i-1} + TN_{i-1} + TN_i + TRFD_i \quad \text{(eq. 4.16), or}$$

$$T_{DD_i} = TRFN_{i-1} + TN_{i-1} + TRFD_{i-1} + TD_{i-1} \qquad \text{(eq. 4.17), or}$$

$$T_{DD_i} = TRFN_{i-1} + TN_{i-1} + TRFD_{i-1} + TRFD_i \qquad \text{(eq. 4.18), or}$$

$$T_{DD_i} = TRFN_{i-1} + TRFN_i + TN_i + TD_{i-1} \qquad \text{(eq. 4.19), or}$$

$$T_{DD_i} = TRFN_{i-1} + TRFN_i + TN_i + TRFD_i \qquad \text{(eq. 4.20), or}$$

$$T_{DD_i} = TRFN_{i-1} + TRFN_i + TRFD_{i-1} + TD_{i-1} \qquad \text{(eq. 4.21), or}$$

$$T_{DD_i} = TRFN_{i-1} + TRFN_i + TRFD_{i-1} + TRFD_i \qquad \text{(eq. 4.22).}$$

Observe that Equations 4.17 and 4.12 are equivalent except for their stage index, as in the simplified case. Thus, Equation 4.17 would have been evaluated in the previous registration pair calculations, so it does not need to be reevaluated in the current registration pair calculations, except for the first pair, stage 1 and stage 2. Equations 4.7 through 4.11, 4.14, 4.15, and 4.18 through 4.22, inclusive, can also be omitted based on the fact that they contain terms with overlapping time intervals. For example, consider Equation 4.11 containing TN_i, then from Equation 4.6, $TN_i > TRFD_{i-1}$, which means that RFD_{i-1} completes before N_i. Since D_{i-1} can begin as soon as RFD_{i-1} completes and RFD_{i-1} completes before N_i, then the intervals labeled D_{i-1} and N_i must at least partially overlap. Thus, Equation 4.11 can be disregarded since it does not take into account this overlap. To remove the overlap, TN_i could be replaced with $TRFD_{i-1}$, which would yield the existing equation, 4.13. Through a similar analysis, three other overlapping terms can be found. Therefore, any equation containing one or more of these overlapping pairs: TN_i and TD_{i-1}, TD_i and TN_{i-1}, $TRFN_i$ and $TRFN_{i-1}$, or $TRFD_i$ and $TRFD_{i-1}$ must also be invalid, leaving only three valid equations, 4.12, 4.13, and 4.16.

In particular, Equation 4.16 considers the case of adjacent NULL propagation delays, including the request times. Equation 4.13 considers the case of adjacent DATA propagation delays, including the request times. Equation 4.12 considers the case of NULL and DATA propagation delays for a single registration stage, including the request times. Based on this analysis, the pseudocode listed in Algorithm 4.2 can be used to calculate the worst-case throughput for a generalized N-stage NCL pipeline.

```
max_cycle_time = TRFD1 + TD1 + TRFN1 + TN1
for (i = 2 to N) loop
        temp_cycle_time =  MAX(TRFDi + TDi + TRFNi + TNi,
                                 TRFDi-1 + TDi-1 + TDi + TRFNi,
                                 TRFNi-1 + TNi-1 + TNi + TRFDi)
        if (temp_cycle_time > max_cycle_time) then
                max_cycle_time = temp_cycle_time
        end if
end loop
worst_case_throughput = 1 / max_cycle_time
```

Algorithm 4.2. Calculation of worst-case throughput for a generalized N-stage pipeline.

Evaluation of the above loop is followed by taking the reciprocal of the maximum adjacent stage pair delay to obtain a lower bound on the pipeline's throughput.

4.3.2 Bit-Wise Completion

In addition to minimizing stage delay, throughput may be further increased using *bit-wise completion*, briefly mentioned in [40]. Until now only full-word completion has been utilized, where the completion signal for each bit in register_i is conjoined by the completion component, whose single-bit output is connected to all K_i lines of register_{i-1}. On the other hand, bit-wise completion only sends the completion signal from bit b in

$register_i$ back to the bits in $register_{i-1}$ that took part in the calculation of bit *b*. This method may therefore require fewer logic levels than that of full-word completion, thus increasing throughput. Bit-wise completion will never reduce throughput, since in the worse case all bits of $register_{i-1}$ are used to calculate each bit of $register_i$, such that the completion logic and therefore throughput does not change by selecting bit-wise completion rather than full-word completion. Bit-wise completion may or may not require more logic gates and therefore transistors than full-word completion, thus bit-wise completion will be used if it increases throughput, or if throughput is the same as for full-word completion but area is reduced.

Figure 56 shows full-word completion for a combinational stage of six 2-input AND functions, generating all combinations of the 4-bit input *X*. Figure 57 shows bit-wise completion for the same six AND functions. There is only one level of logic in the completion components for the bit-wise completion approach verses two levels of logic in the completion component for the full-word completion approach. Also notice that four gates are required for bit-wise completion verses three gates for full-word completion, a difference of 8 additional transistors. To maximize throughput in this case, bit-wise completion would be selected in spite of its larger size since it reduces the completion logic path from two gate delays down to only one gate delay, which translates to an increase in throughput by Algorithm 4.2.

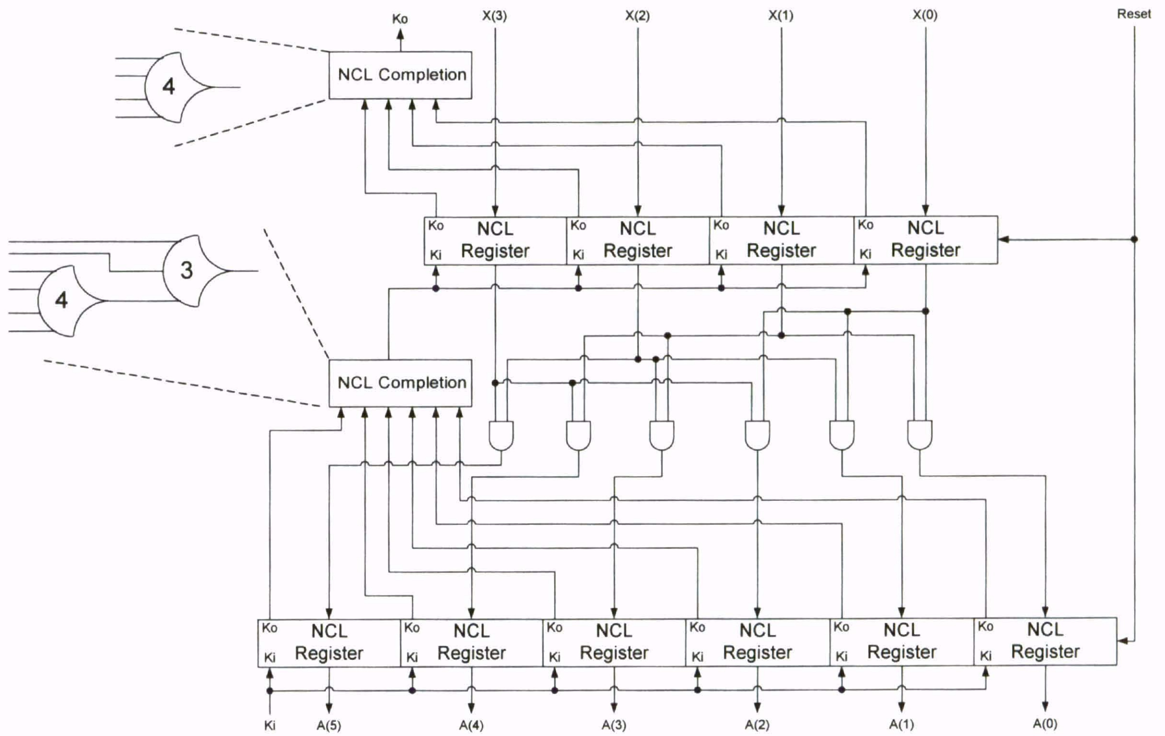

Figure 56. Full-word completion.

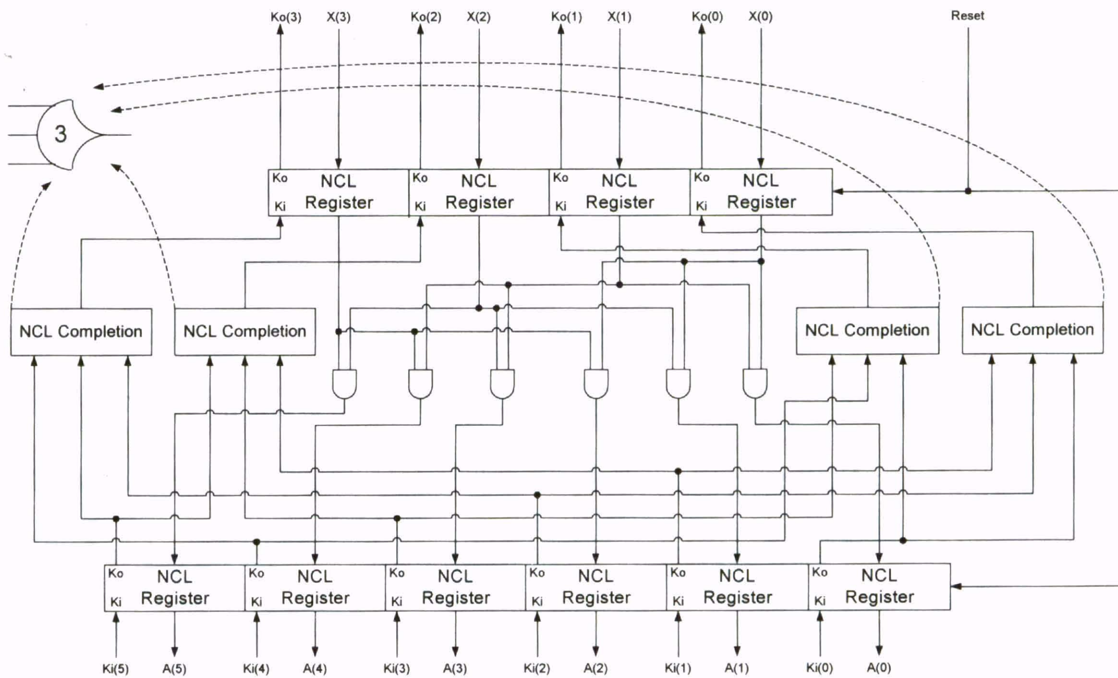

Figure 57. Bit-wise completion.

4.4 Application to Unsigned Multiplier

A number of designs based on the 4-bit by 4-bit multiplier shown in Figure 58 have been evaluated as a case study to assess the impact of GLP methods on throughput. The specifications for this multiplier were simply to perform an unsigned multiply of the two 4-bit input vectors, *X* and *Y*, and then output their 8-bit product, *S*. As with all NCL systems, a full NCL interface with request and acknowledge signals labeled K_i and K_o, respectively, is included for requesting and acknowledging complete DATA and NULL wavefronts.

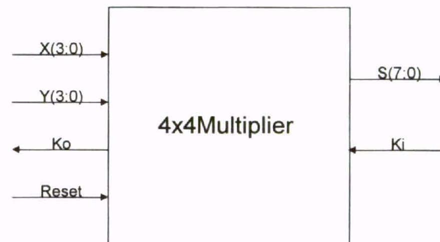

Figure 58. 4×4 multiplier block diagram.

The non-pipelined version of the 4×4 multiplier is shown in Figure 59. It consists of incomplete AND functions, denoted as *I* and depicted in Figure 10, as well as complete AND functions, denoted as *C* and developed in Chapter 3. The multiplier also utilizes half adders, as shown in Figure 60 and denoted *HA*, as well as full adders, as shown in Figure 30 and denoted *FA*. The last components of the multiplier include *GEN_S7*, as shown in Figure 61, and the completion components, denoted as *COMP*. Remember that the number of gate delays in the completion logic for an N-bit register is $\lceil \log_4 N \rceil$, as discussed in Chapter 2.

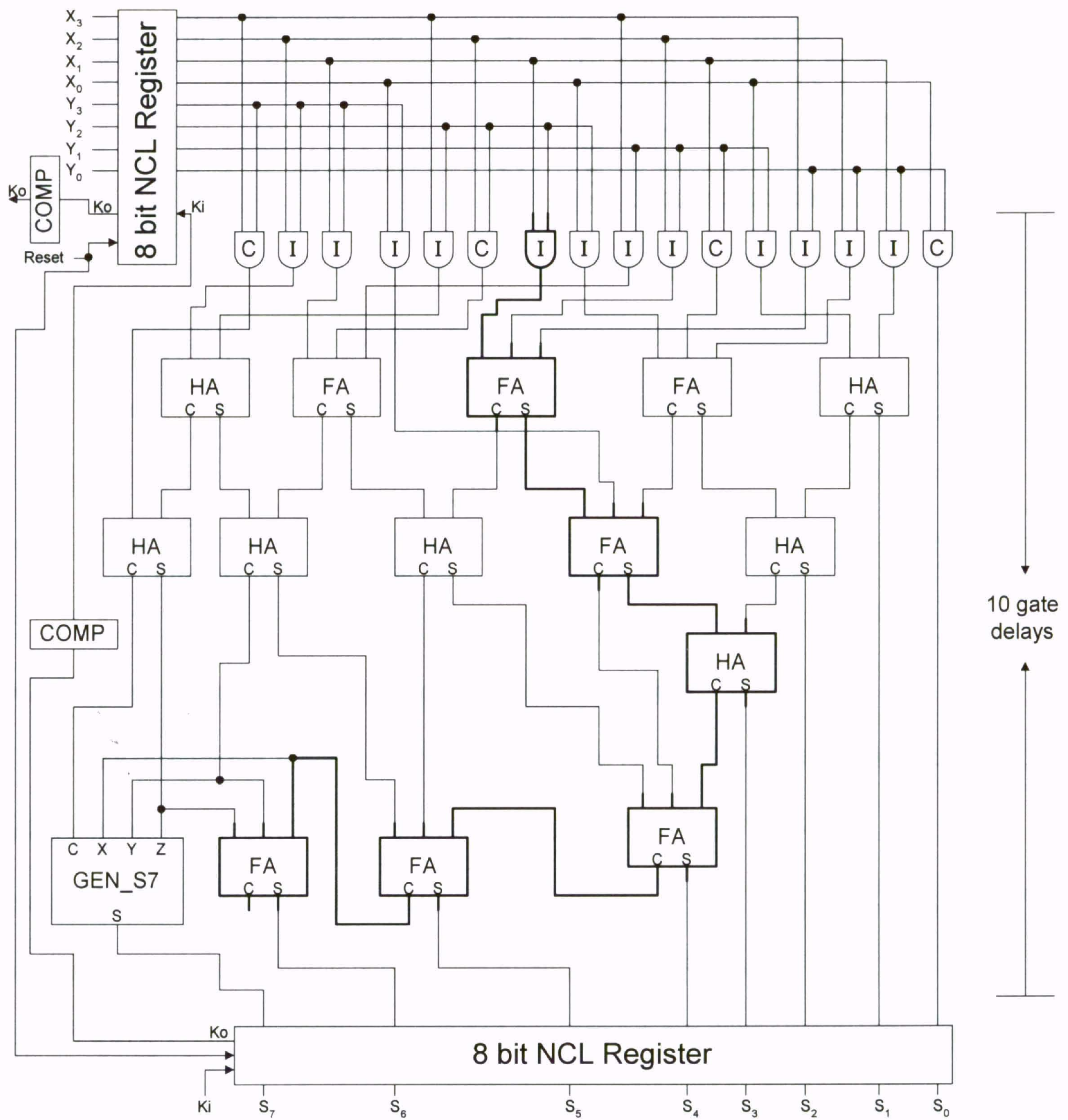

Figure 59. Non-pipelined, 1-stage 4×4 multiplier using full-word completion.

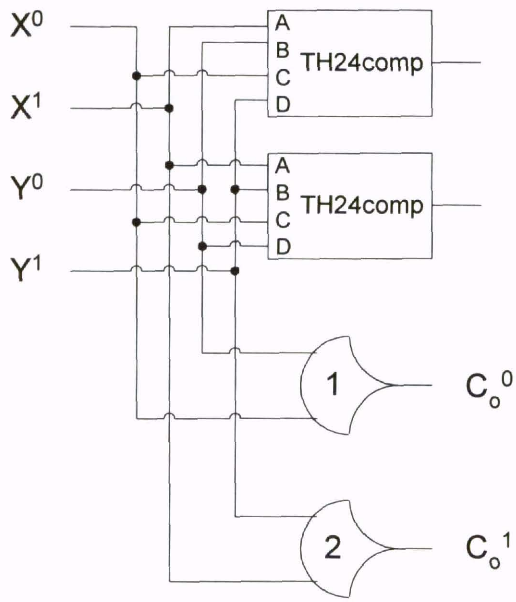

Figure 60. Half adder

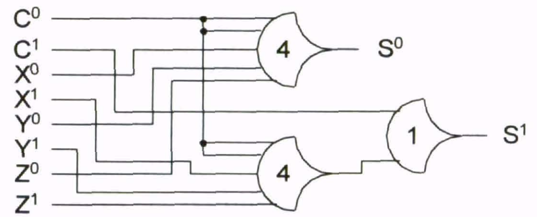

Figure 61. GEN_S7 component.

4.4.1 <u>Pipelined Multipliers with Full-Word Completion</u>

The throughput for the non-pipelined design is calculated using Algorithm 4.2, and is determined to be (24 gate delays)$^{-1}$. Here, $TRFD_1 = TRFN_1 = \lceil \log_4 8 \rceil = 2$ gate delays and $TN_1 = TD_1 = 10$ gate delays as given by the I, FA, FA, HA, FA, FA, and FA components along the critical path shown in bold face in Figure 59. Thus, $T_{DD} = TRFD_1 + TD_1 + TRFN_1 + TN_1 = 2 + 10 + 2 + 10 = 24$. Since the 4×4 multiplier has a longest path delay of 10 threshold gates, then from the flowchart in Figure 53, the 4×4 multiplier can be pipelined with either 5, 4, 3, 2, or 1 gate delays per stage, if completeness of input can be achieved for each such partition.

For a partition of 5 gate delays per stage, 2 stages are required, as shown in Figure 62. The throughput of this 2-stage design is determined to be (14 gate delays)$^{-1}$, as all equations from Algorithm 4.2 yield this same maximum cycle delay. For a partition of

4 gate delays per stage, 3 stages are required, as shown in Figure 63. The first and second stages only have 3 gate delays, while stage 3 has 4 gate delays. The throughput of this 3-stage design is determined to be $(12 \text{ gate delays})^{-1}$, as calculated from Algorithm 4.2 for stage 3. For a partition of 3 gate delays per stage, 4 stages are required, as shown in Figure 64. The first stage has 3 gate delays, stage 2 only has 2 gate delays, and stage 3 and stage 4 both have 3 gate delays. The throughput of this 4-stage design is determined to be $(10 \text{ gate delays})^{-1}$. The equations from Algorithm 4.2 for stage 1, stage 3, stage 4, and stages 3 and 4 combined all yield this result. For a partition of 2 gate delays per stage, 7 stages are required, as shown in Figure 65. The first stage and the fourth stage only have 1 gate delay, while the other stages all have 2 gate delays. The throughput of this 7-stage design is determined to be $(8 \text{ gate delays})^{-1}$. The equations from Algorithm 4.2 for stages 2, 3, 5, 6, and 7, as well as those for stages 2 and 3 combined, stages 5 and 6 combined, and stages 6 and 7 combined yield this result.

A partition into a single gate delay per stage cannot be achieved since the completeness of input criterion is unattainable using only one level of logic with a maximum gate fan-in of 4 inputs. This would require inserting a register between the two levels of logic within the full adder, which would violate the completeness of input criterion upon which it was designed.

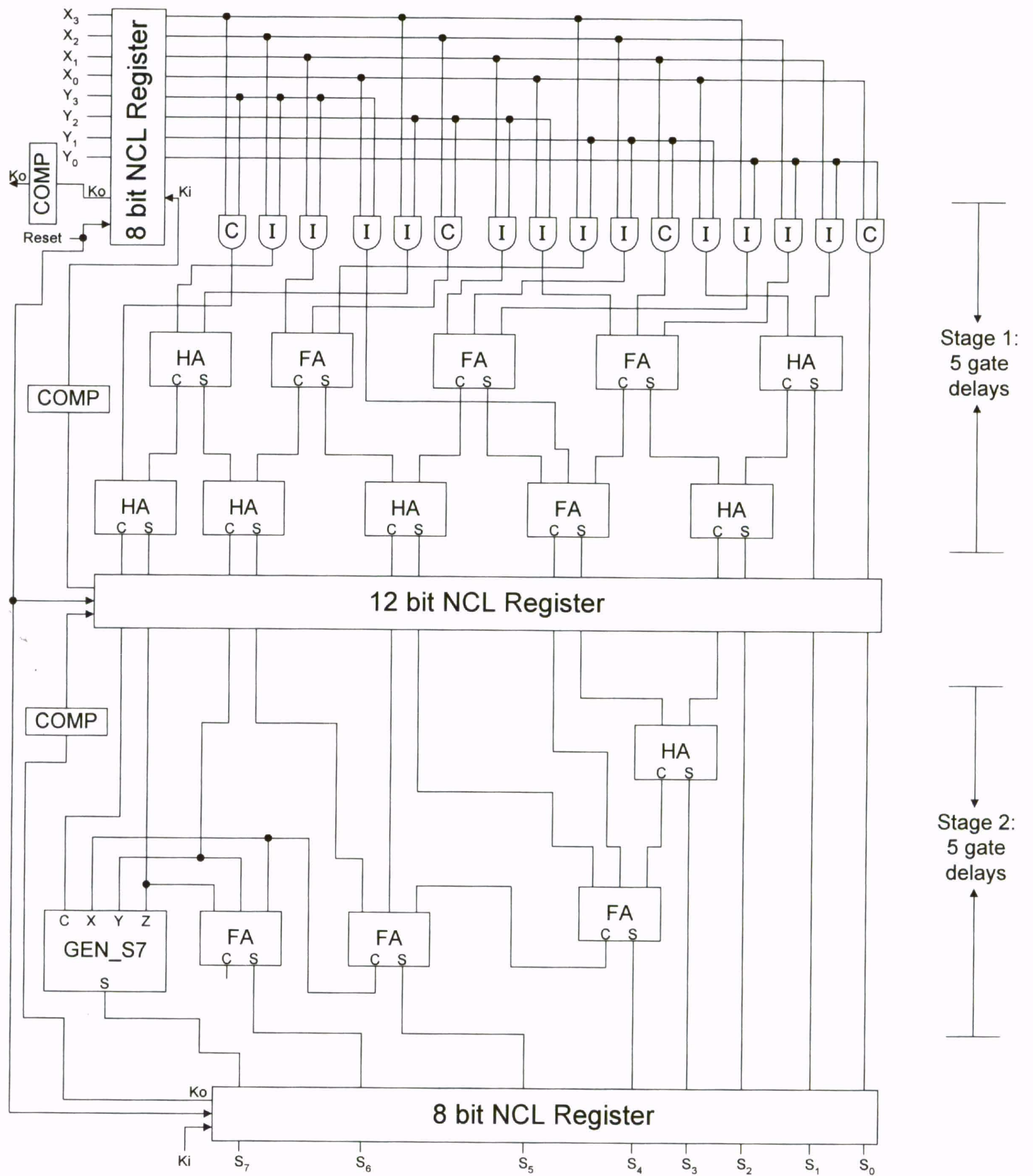

Figure 62. 2-stage 4×4 multiplier using full-word completion.

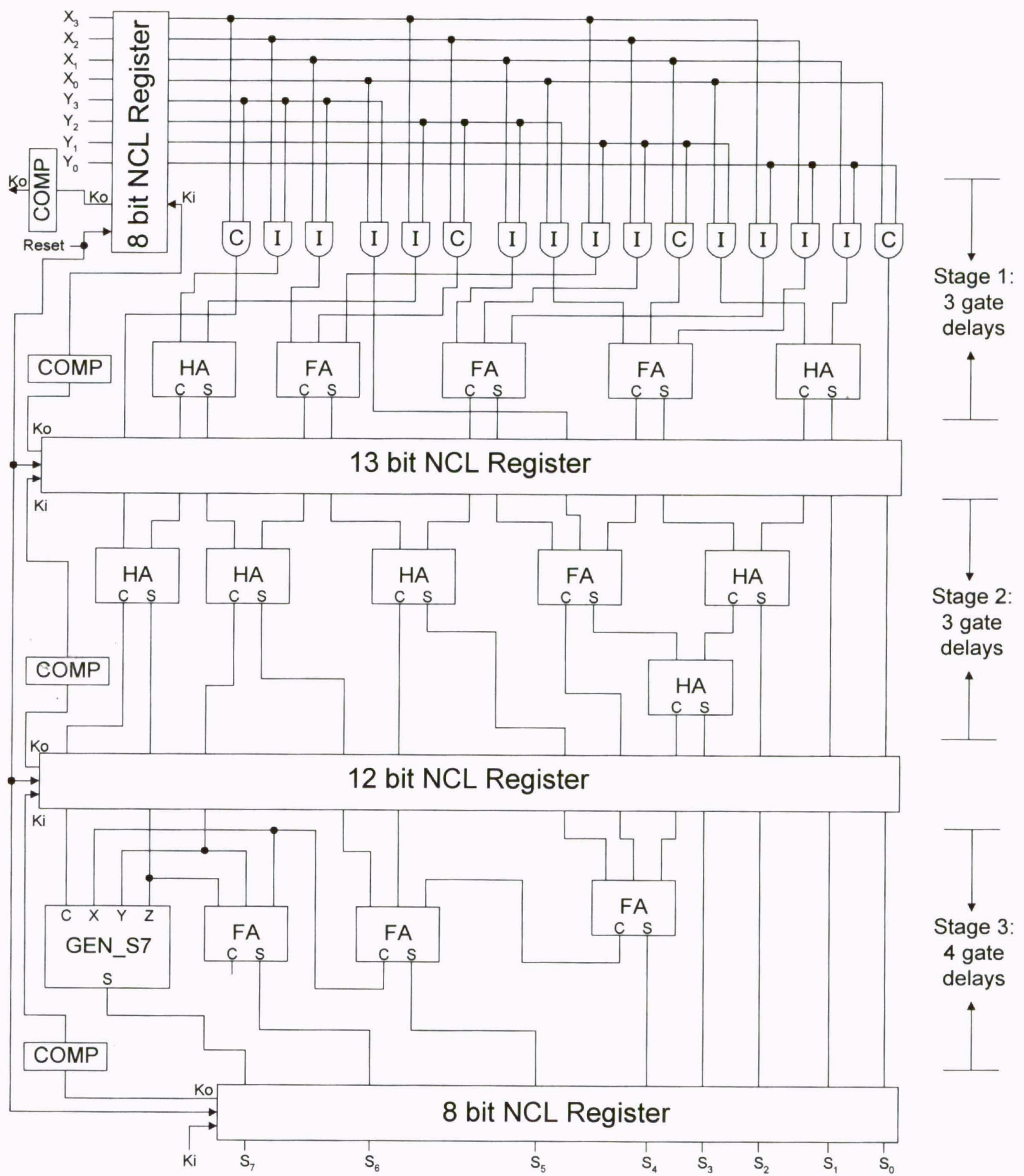

Figure 63. 3-stage 4×4 multiplier using full-word completion.

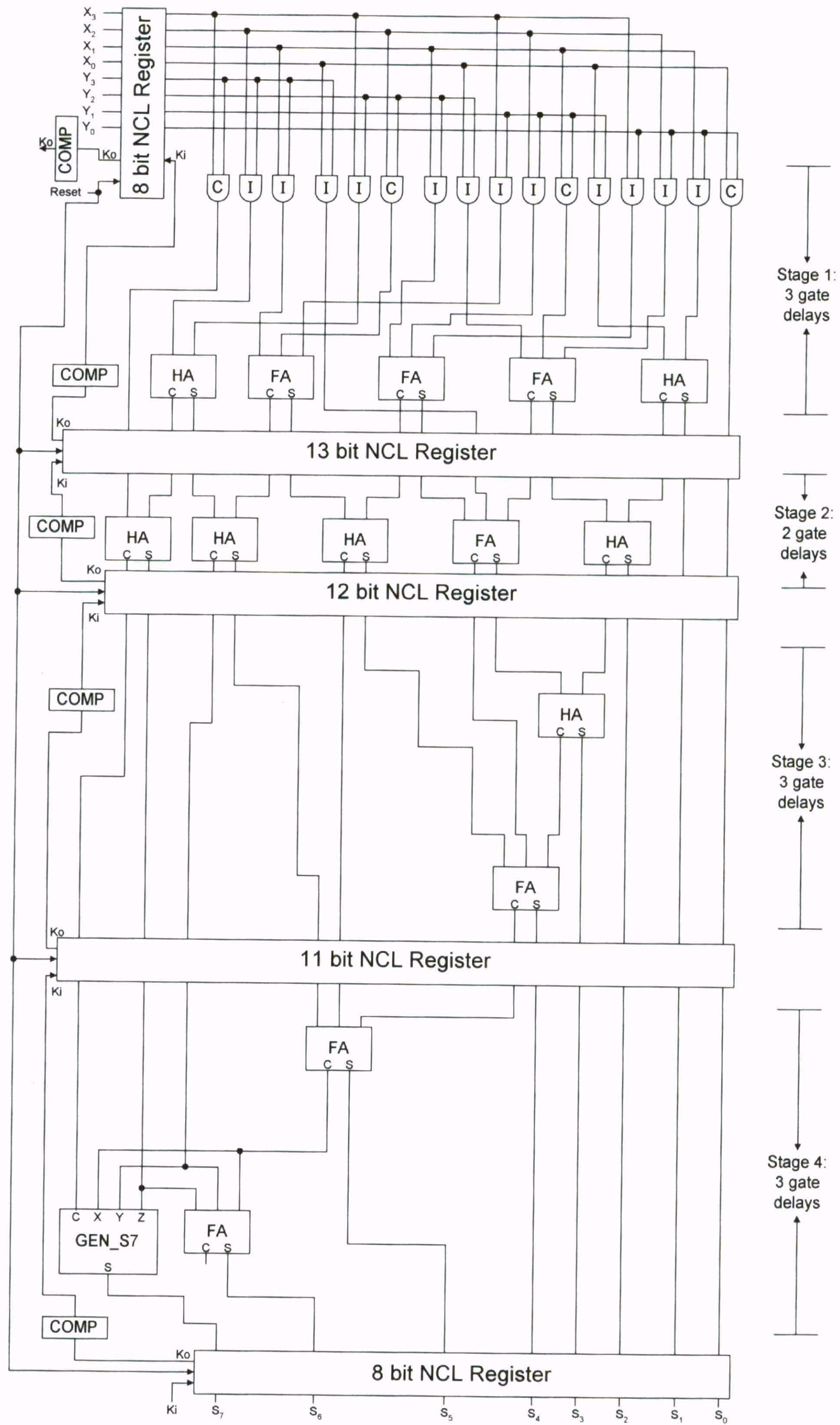

Figure 64. 4-stage 4×4 multiplier using full-word completion.

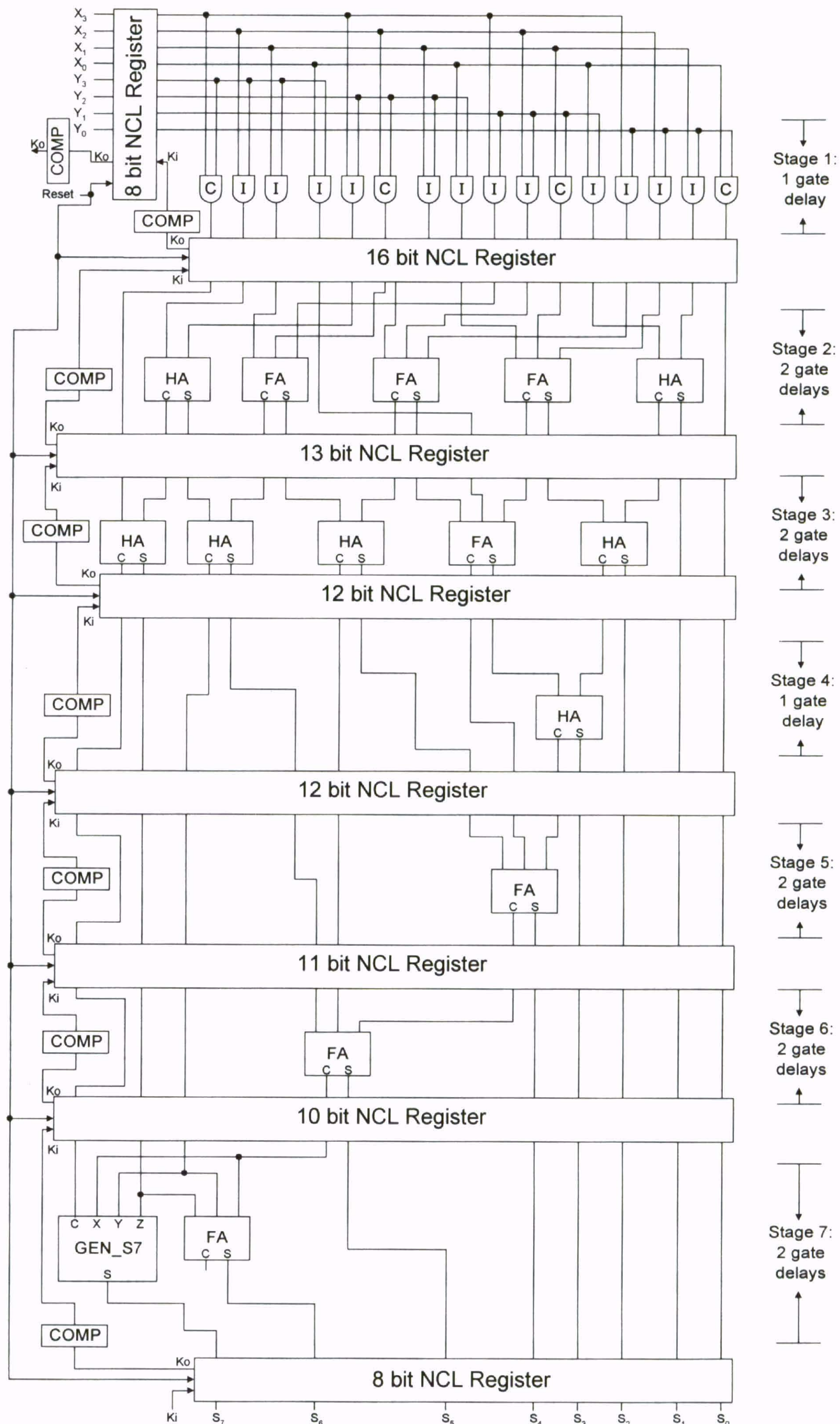

Figure 65. 7-stage 4×4 multiplier using full-word completion.

4.4.2 Summary of Multiplier Designs using Full-Word Completion

The maximum throughput when pipelining the 4×4 multiplier using full-word completion was (8 gate delays)$^{-1}$ as attained by the 7-stage design. Table X compares the throughputs attained from Synopsys simulation and shows that the 7-stage design indeed outperforms all other configurations, as expected by comparing the analytically predicted throughputs. This design has a 19% increase in throughput over the next highest throughput from the 4-stage multiplier, and an 83% increase in throughput over the original non-pipelined design. This increase in throughput was achieved at the expense of inserting 6 asynchronous registers along with corresponding completion logic, as dictated by the flowchart of Figure 53. The simulated throughput was obtained by averaging the throughputs resulting from all 256 possible combinations of input pairs.

Table X. Stage delay and throughput for various multiplier designs.

Multiplier Design	Maximum Combinational Delay per Stage (gate delays)	Maximum Completion Delay per Stage (gate delays)	Predicted Throughput (gate delays)$^{-1}$	Simulated Throughput (ns)$^{-1}$
1-stage	10	2	1/24 = 0.042	0.114
2-stage	5	2	1/14 = 0.071	0.150
3-stage	4	2	1/12 = 0.083	0.172
4-stage	3	2	1/10 = 0.100	0.176
7-stage	2	2	1/8 = 0.125	0.209

4.4.3 Applying Bit-Wise Completion

After traversing the loop of Figure 53 such that *i=0*, the highest throughput design utilized full-word completion. Bit-wise completion was applied to this design as specified by the flowchart. When switching from full-word completion to bit-wise completion the

incomplete AND functions had to be replaced with complete AND functions to satisfy the completeness of input criterion over the new completion sets. The resulting design, shown in Figure 66, reduced the completion logic from 2 gate delays to only 1 gate delay for all registers, thus increasing the throughput from (8 gate delays)$^{-1}$ to (6 gate delays)$^{-1}$. From Synopsys simulation throughput was determined to be 0.257 ns^{-1}, an increase of 21% over the design with an identical number of stages using full-word completion. Thus, the 7-stage 4×4 multiplier utilizing bit-wise completion optimizes throughput.

4.5 Conclusion

Since increasingly finer pipelining of the multiplier did not increase the completion delay, the most finely grained pipelined design was optimal. The non-pipelined design (Figure 59) required a maximum register width of 8 bits while the 7-stage pipelined design (Figure 65) required a maximum register width of 16 bits, and $\lceil \log_4 8 \rceil = \lceil \log_4 16 \rceil = 2$. However, if the 7-stage design required a maximum register width of 17 bits instead of 16 bits, the throughput for the 7-stage design using full-word completion would have been the same as for the 4-stage design using full-word completion. Thus, the 4-stage design using full-word completion would have been preferable over its 7-stage counterpart, since it would have had less latency. Bit-wise completion would still have had to be performed on the 7-stage design and possibly the 4-stage design to determine the overall optimal throughput design.

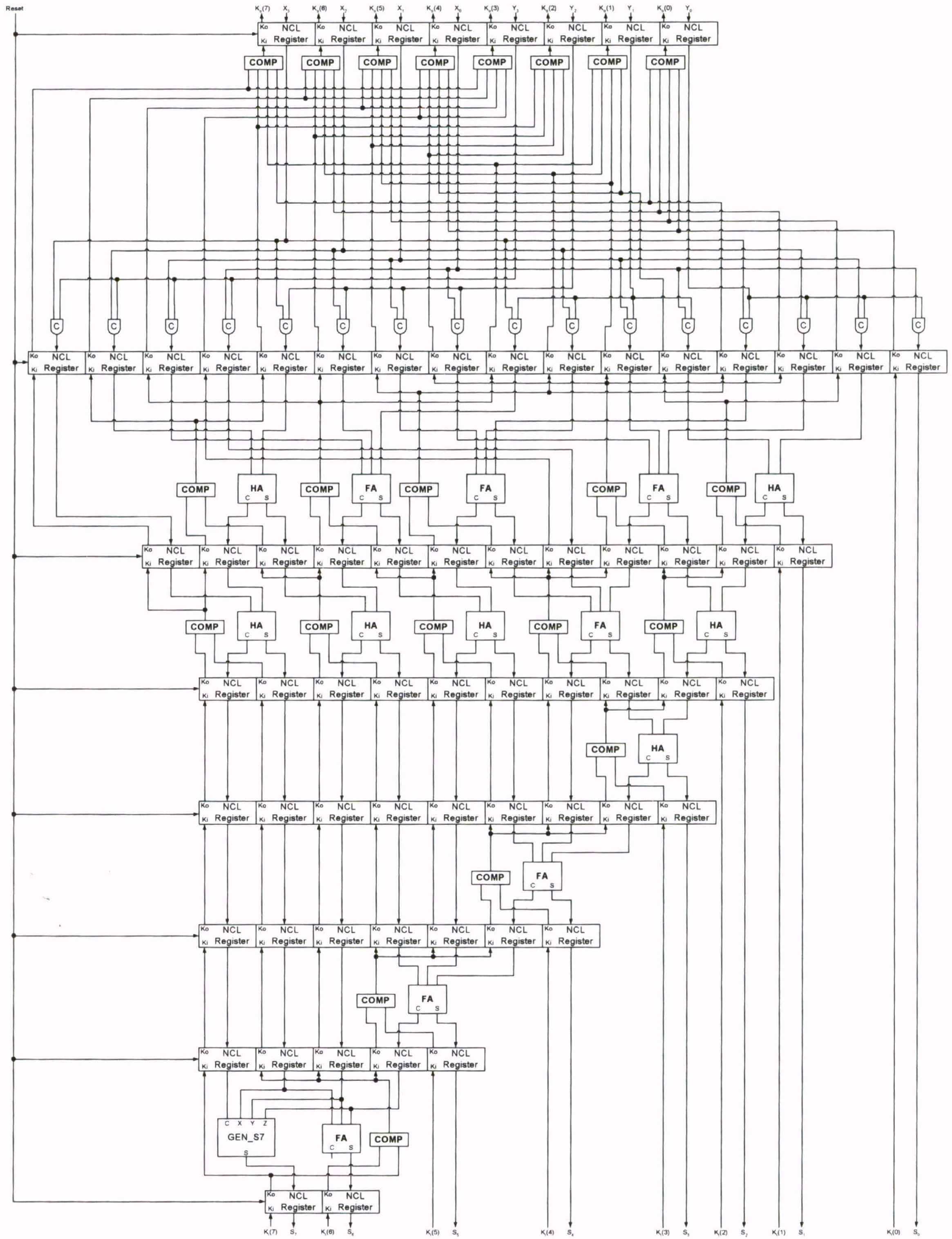

Figure 66. 7-stage 4×4 multiplier using bit-wise completion.

5.0 NULL CYCLE REDUCTION TECHNIQUE

A *NULL Cycle Reduction (NCR)* technique is developed to increase the throughput of delay-insensitive digital systems. NCR reduces the time required to flush complete DATA wavefronts, commonly referred to as the *NULL* or *Empty* cycle. The NCR technique exploits parallelism by partitioning input wavefronts such that one circuit processes a DATA wavefront, while its duplicate processes a NULL wavefront. To illustrate the technique, NCR is applied to a case study of a dual-rail non-pipelined 4-bit by 4-bit unsigned multiplier, yielding a speedup of 1.61 over the standalone version, while maintaining delay-insensitivity.

5.1 Introduction

Most multi-rail delay-insensitive logic paradigms employ both a DATA wavefront and a NULL wavefront in order to maintain delay-insensitivity [4, 6, 7, 8, 9, 21]. The DATA wavefront realizes circuit functionality, while the NULL wavefront flushes the previous DATA wavefront. The NULL cycle accounts for approximately half of the total cycle time, thus decreasing attainable throughput by a factor of two. The objective of this chapter is to develop and illustrate a technique for reducing the NULL cycle time such that throughput does not depend as heavily on the DATA flush time, yet still maintains delay-insensitivity.

Many architectures and algorithms employ the well-known divide and conquer strategy. The divide and conquer technique partitions a problem into smaller sub-problems that can be solved simultaneously, then merges their outputs to construct the solution to the original problem, thus reducing computation time. The NCR technique described herein also employs this divide and conquer strategy to increase the throughput of NCL systems. Successive input wavefronts are partitioned such that one circuit processes a DATA wavefront, while its duplicate processes a NULL wavefront. The first DATA/NULL cycle flows through the original circuit, while the next DATA/NULL cycle flows through the other circuit. The outputs of the two circuits are then multiplexed to form a single output stream.

5.2 NULL Cycle Reduction

The technique for reducing the NULL cycle, thus increasing throughput for any NCL system is shown in Figure 67. *NCL Circuit #1* and *NCL Circuit #2* have identical functionality and are both initialized to output NULL and request DATA upon reset. Both have an asynchronous NCL register at the input and output, while the combinational functionality can be designed using TCR described in Chapter 3. These circuits may also be pipelined as described in Chapter 4, to further increase throughput. The *Demultiplexer* partitions the input, *D*, into two outputs, *A* and *B*, such that *A* receives the first DATA/NULL cycle and *B* receives the second DATA/NULL cycle. The input continuously alternates between *A* and *B*. The *Completion Detection* circuitry detects when either a complete DATA or NULL wavefront has propagated through the

Demultiplexer, and requests the next NULL or DATA wavefront, respectively. *Sequencer #1* is controlled by the output of the Completion Detection circuitry and is used to select either output *A* or *B* of the Demultiplexer. Output *A* of the Demultiplexer is input to NCL Circuit #1 when requested by *Ki1*; and output *B* of the Demultiplexer is input to NCL Circuit #2 when requested by *Ki2*. The outputs of NCL Circuit #1 and NCL Circuit #2 are allowed to pass through their respective output registers, as determined by *Sequencer #2*, which is controlled by the external request, *Ki*. The Multiplexer rejoins the partitioned datapath by passing a DATA input on either *A* or *B* to the output, or asserting NULL on the output when both *A* and *B* are NULL. Figure 67 shows the state of the system when a DATA wavefront is being input, before its acknowledge flows through the Completion Detection circuitry, and when a DATA wavefront is being output, before it is acknowledged by the receiver.

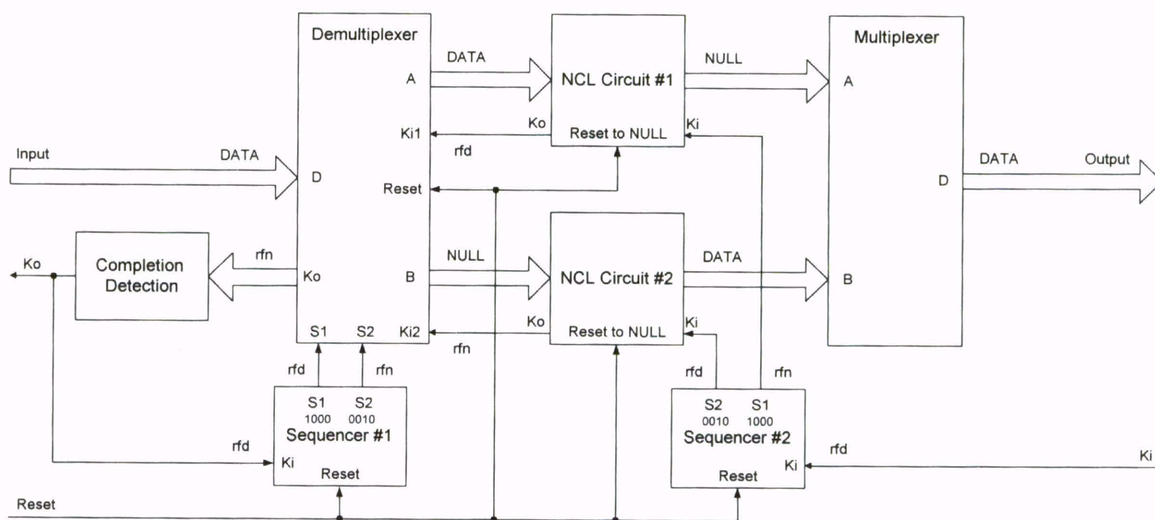

Figure 67. NCR architecture.

5.2.1 Demultiplexer

A logic diagram for one bit of the *Demultiplexer* is shown in Figure 68. Upon reset both *A* and *B* are initialized to NULL. When *S1* is asserted and *Ki1* is *rfd*, a DATA input on *D* will be passed to output *A*. Likewise, when *S2* is asserted and *Ki2* is *rfd*, a DATA input on *D* will be passed to output *B*. *Ko* becomes *rfd* when both *A* and *B* are NULL, and becomes *rfn* when either *A* or *B* is DATA. When *A* becomes DATA, it will return to NULL only after *S1* is de-asserted, *Ki1* becomes *rfn*, and the input, *D*, becomes NULL. Likewise, when *B* becomes DATA, it will return to NULL only after *S2* is de-asserted, *Ki2* becomes *rfn*, and the input, *D*, becomes NULL. Therefore, *A* and *B* can never both be DATA since *S1* and *S2* can never be simultaneously asserted and both *A* and *B* must be NULL before the next DATA wavefront is requested. Each bit of the Demultiplexer is the same, and the number of bits is determined by the width of the input datapath.

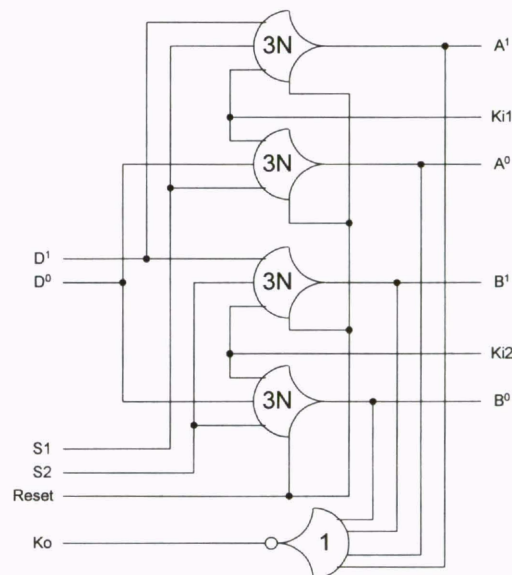

Figure 68. 1-bit Demultiplexer.

5.2.2 Completion Detection Circuitry

The *Completion Detection* circuitry is the same as that explained in Chapter 2 and shown in Figure 23. The number of *Ko* lines from the Demultiplexer is also determined by the width of the input datapath.

5.2.3 Sequencer #1

Sequencer #1 is controlled by the output of the Completion Detection circuitry and is used to select either output *A* or *B* of the Demultiplexer. Upon reset it selects output *A* to receive the first DATA/NULL cycle, after *Ki* becomes *rfd*. It then selects output *B* to receive the second DATA/NULL cycle. Sequencer #1 continuously alternates the DATA/NULL cycles between outputs *A* and *B*. A logic diagram of Sequencer #1 is shown in Figure 69. This is a 4-stage single-rail ring structure with one token, where a token is defined as a DATA wavefront with corresponding NULL wavefront, and two bubbles, where a bubble is defined as either a DATA or NULL wavefront occupying more than one neighboring stage [38]. When *Ki* becomes *rfd*, the DATA wavefront moves through the two NULL bubbles ahead of it, creating two DATA bubbles in its wake. Likewise, when *Ki* becomes *rfn*, the NULL wavefront moves through the two DATA bubbles ahead of it, creating two NULL bubbles in its wake. The DATA/NULL wavefront restricts the forward propagation of the NULL/DATA wavefront, respectively, for each change of *Ki*, limiting the forward propagation to only the two bubbles. A complete cycle of the Sequencer is shown in boldface and italics in Table XI. The cycle for *S1* is 1000, while the cycle for *S2* is 0010.

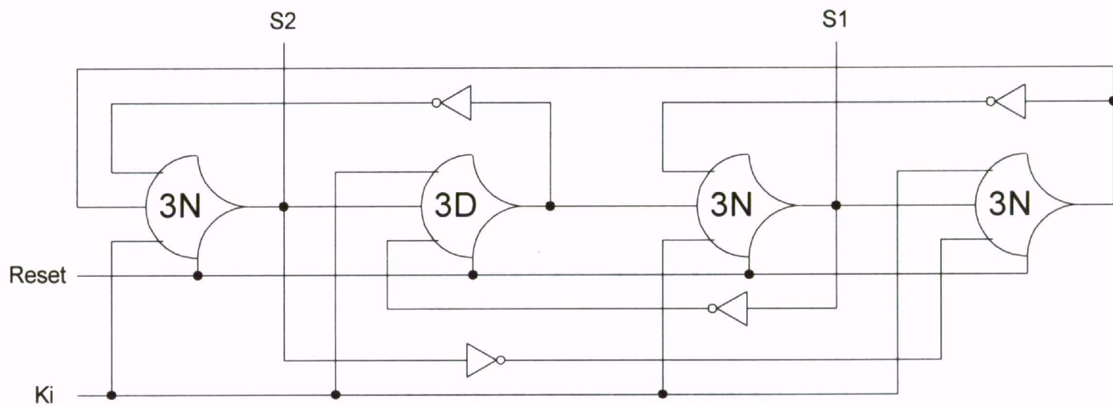

Figure 69. Sequence generator.

Table XI. Sequencer output.

Cycle #	Initial State	1	2	3	4	5	6	7	8
Reset	1	0	0	0	0	0	0	0	0
Ki	X	1	0	1	0	***1***	***0***	***1***	***0***
S1	0	1	0	0	0	***1***	***0***	***0***	***0***
S2	0	0	0	1	0	***0***	***0***	***1***	***0***

5.2.4 Multiplexer

A logic diagram for one bit of the *Multiplexer* is shown in Figure 70. It simply consists of two OR gates that pass a DATA input on either *A* or *B* to the output, *D*, or assert NULL on the output when both *A* and *B* are NULL. The Multiplexer does not require any select signals, since *A* and *B* can never simultaneously be DATA. This mutual exclusion is ensured by Sequencer #2, which controls the outputs of NCL Circuit #1 and NCL Circuit #2. Each bit of the Multiplexer is the same, and the number of bits is determined by the width of the output datapath.

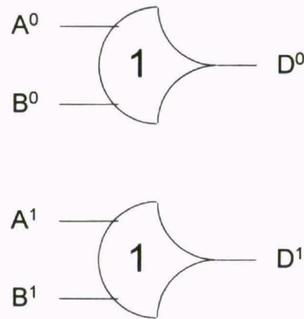

Figure 70. 1-bit Multiplexer.

5.2.5 Sequencer #2

Sequencer #2 is controlled by the external request, *Ki*, and is used to allow DATA and NULL wavefronts to flow through the output register of NCL Circuit #1 and NCL Circuit #2. Upon reset it selects NCL Circuit #1 to output the first DATA/NULL cycle, after *Ki* becomes *rfd*. It then selects NCL Circuit #2 to receive the second DATA/NULL cycle. Sequencer #2 continuously alternates the DATA/NULL cycles between NCL Circuit #1 and NCL Circuit #2. When *S1* is asserted, DATA will be output from NCL Circuit #1. Likewise, when *S2* is asserted, DATA will be output from NCL Circuit #2. When the output of NCL Circuit #1 becomes DATA, it will return to NULL only after *S1* is de-asserted. Likewise, when the output of NCL Circuit #2 becomes DATA, it will return to NULL only after *S2* is de-asserted. Therefore, NCL Circuit #1 and NCL Circuit #2 can never both output DATA since *S1* and *S2* can never be simultaneously asserted and the outputs of both circuits must be NULL before the next DATA wavefront is requested by asserting either *S1* or *S2*. The structure of Sequencer #2 is the same as that of Sequencer #1 shown in Figure 69.

5.3 Simulation Results

A case study of a dual-rail non-pipelined 4-bit by 4-bit multiplier, shown in Figure 59, has been evaluated to assess the impact of the NCR technique on throughput. The specifications for this multiplier were simply to perform an unsigned multiply of the two 4-bit input vectors, X and Y, and then output their 8-bit product, S. A full NCL interface with request and acknowledge signals labeled K_i and K_o, respectively, is provided for requesting and acknowledging complete DATA and NULL wavefronts. From Synopsys simulation it was determined that the standalone version of the dual-rail non-pipelined 4-bit by 4-bit multiplier had an average DATA-to-DATA cycle time of 8.75 ns with approximately equal DATA and NULL cycles. When the NCR technique was applied to this design, the NULL cycle was reduced to approximately ¼ of the DATA cycle. This resulted in an overall average DATA-to-DATA cycle time of only 5.43 ns, which corresponds to a 61% increase in throughput. Values for average throughput were obtained from the arithmetic mean of throughputs corresponding to all 256 possible pairs of input operands.

Table XII compares the throughput of the multiplier using NCR with the throughputs achieved by pipelining the multiplier as explained in Chapter 4. Table XII shows that the NCR technique is roughly comparable to pipelining for some applications, since it falls in between the 4-stage and 7-stage pipelined designs in terms of both throughput and gate count. Furthermore, it is not necessary to duplicate the entire circuit when applying the NCR technique. Rather, its benefits can be obtained without doubling area and power requirements by applying it to selective portions of a circuit, which

cannot be pipelined more finely due to the completeness of input criterion. However, if NCR was applied to $stage_i$ to boost throughput, both $stage_{i-1}$ and $stage_{i+1}$ may have to be non-functional stages to realize the full increase due to the adjacent DATA propagation delays of Equation 4.13 for determining throughput, as explained in Chapter 4. A non-functional stage can be easily added by inserting an additional asynchronous register. Thus, throughput of a pipelined design with a small number of slow stages can be readily boosted with relatively little cost by using NCR.

Table XII. NCR vs. pipelining for multiplier application.

Multiplier Design	**Maximum Combinational Delay per Stage** (gate delays)	**Maximum Completion Delay per Stage** (gate delays)	**Simulated Throughput** $(ns)^{-1}$	**Gate Count**
4-stage	3	2	0.176	264
NCR (1-stage)	10	2	0.184	365
7-stage	2	2	0.209	390

To illustrate this point, NCR was applied to only a single stage of the pipeline shown in Figure 71. Multiplier #1 and Multiplier #3 are both 2-stage unsigned multipliers with a worse-case stage delay of 5 gate delays, as depicted in Figure 62. Multiplier #2 is a non-pipelined unsigned multiplier consisting of 10 gate delays, as depicted in Figure 59. Therefore, the 10 gate delays of Multiplier #2 is much longer than the 5 gate delays per stage of the other multipliers, making Multiplier #2 a good candidate for NULL Cycle Reduction. Without NCR, the pipeline of Figure 71 operates with $T_{DD} = 8.42$ ns; however, with NCR only applied to Multiplier #2, T_{DD} is decreased to 6.96 ns, a speedup of 1.21. Henceforth, applying NCR to only slow stages in a pipeline can boost throughput

for the pipeline as a whole. Note that additional registration was not needed to form non-functional stages around the NCR stage, since these non-functional stages already existed when the multipliers were connected to form the pipeline of Figure 71, since each multiplier contains both an input and output register.

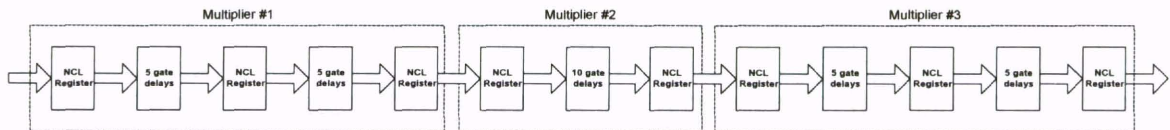

Figure 71. NCL pipeline with one slow stage.

6.0 NCL MULTIPLY AND ACCUMULATE UNIT

The TCR and GLP techniques developed in earlier chapters are illustrated in the context of a sophisticated arithmetic application. Approaches for maximizing throughput of self-timed multiply and accumulate units (MACs) are developed and assessed using NCL. It is shown that the self-timed MAC throughput optimization problem can be transformed into the selection of the multiplication algorithm requiring the fewest number of gates. A number of alternative MAC algorithms are compared and contrasted in terms of throughput and area to determine which design will yield the maximum throughput with the least area. It was determined that two algorithms that meet these criteria well are *Modified Baugh-Wooley* and *Modified Booth2*. Dual-rail non-pipelined versions of these algorithms were first designed using the *Threshold Combinational Reduction (TCR)* method described in Chapter 3. The non-pipelined designs were then optimized for throughput using the *Gate-Level Pipelining (GLP)* method described in Chapter 4. Finally, each design was simulated using Synopsys to quantify the advantage of the dual-rail pipelined *Modified Baugh-Wooley* MAC, which yielded a speedup of 2.5 over its initial non-pipelined version. This design also required 20% fewer gates than the dual-rail pipelined *Modified Booth2* MAC that operated at the same throughput. The resulting design employs a three-stage feed-forward multiply pipeline connected to a four-stage feedback multifunctional loop to perform a $72+32\times32$ MAC in 12.7 ns on

average using a 0.25 μm CMOS process at 3.3V, thus outperforming other delay-insensitive/self-timed MACs in the literature.

6.1 Introduction

This chapter evaluates a number of both bitwise and digitwise multiplication algorithms suitable for self-timed MAC design. The bitwise algorithms include *Array Structured* multiplication and multiplication using the *Modified Baugh-Wooley* algorithm. Digitwise algorithms include *Modified Booth* multiplication as well as combinational *N-Bit* × *M-Bit* multiplication. These algorithms are compared in terms of throughput and area to first maximize steady-state throughput and then minimize total gate count within the NCL multi-rail paradigm. This chapter considers 2^s-complement operands with rounding, scaling, and saturation of the output.

The chapter is organized into six sections. An overview of previous work is given in Section 6.2. In Section 6.3, the non-pipelined and pipelined versions of both the Modified Baugh-Wooley and Modified Booth2 MACs are designed; and their throughputs are estimated analytically and also simulated. Section 6.4 details the rationale for selecting a ripple-carry adder over a carry-lookahead adder for carry-propagation. In Section 6.5 the above designs, along with a variety of others, are compared in terms of gate count. Section 6.6 provides conclusions and compares the NCL MAC developed herein to other delay-insensitive/self-timed MACs.

6.2 Previous Work

Approaches to self-timed MAC design are an area of recent interest [41, 42, 43]. Self-timed MAC design itself presents some interesting design considerations such as feedback loop throughput maximization, carry-propagate adder selection, and multiplication algorithm selection. As detailed in Section 6.3.3.2, throughput is maximized for a self-timed feedback loop by inserting enough, but not too many, asynchronous registers. In Section 6.4 it is shown that for NCL, a ripple-carry adder is better than a carry-lookahead adder since timing is based on average-case scenarios. And as explained in Section 6.3.5, the throughput of a pipelined self-timed MAC is independent of the selected multiplication algorithm, making the best choice the algorithm requiring the least area.

The *Modified Baugh-Wooley* algorithm, the *Array* algorithm, and the *Modified Booth* algorithm for multiplication are all described in [44]. The *Modified Baugh-Wooley* algorithm removes the need for negatively weighted bits present in the traditional 2^s-complement multiplication algorithm by modifying the most significant bit of each partial product and the last row of partial products, and by adding two extra bits to the partial product matrix. This allows for summation of the partial products without using special adders equipped to handle negative inputs and without increasing the height of a tree of 3-input, 2-output carry-save adders.

Array multiplication of 2^s-complement numbers also begins with each partial product bit generated according to the Modified Baugh-Wooley algorithm. Its distinguishing characteristic is the technique for partial product summation. In the

Modified Baugh-Wooley algorithm the partial products are summed using a Wallace tree [44], which reduces the number of partial products by a factor of $\frac{2}{3}$ after each level of the tree and requires $O(\log_2 N)$ time and O(N) space, where *N* denotes the number of partial products [45]. On the other hand, Array multiplication reduces the number of partial products by one at each level, therefore this method requires both O(N) time and space [45].

The *Modified Booth* algorithms reduce the number of partial products to be summed by partitioning the multiplier into groups of overlapping bits, which are then used to select multiples of the multiplicand for each partial product. Consider, for example an N-bit by N-bit 2^s-complement multiply. Using the Modified Booth2 algorithm the multiplier is partitioned into overlapping groups of three bits, each of which selects a partial product from the following list: +0, +M, +2M, -2M, -M, and –0, where *M* represents the multiplicand. This recoding reduces the number of partial products from *N* to $\lfloor \frac{N+2}{2} \rfloor$. The tradeoff is more logic in the recoding portion of the multiplier in exchange for fewer partial products to sum.

6.3 Self-Timed MAC Design Methods

A block diagram for the MACs developed in this chapter is shown in Figure 72. Each MAC unit performs a 32-bit by 32-bit fixed-point fractional multiply, accepting (signed × signed), (signed × unsigned), and (unsigned × unsigned) 2^s-complement operands. The product may be added to or subtracted from the 72-bit accumulator. The MAC also supports 2^s-complement and convergent rounding, up-scaling and down-

scaling, output saturation, and it includes a multiply only option. The output is the 72-bit 2^s-complement result along with a bit to detect overflow.

The taxonomy in Figure 73 is useful to illustrate relationships between some possible multiplication algorithms that could be used in a self-timed MAC design. These include bitwise algorithms such as *Array* multiplication and the *Modified Baugh-Wooley* algorithm; and digitwise algorithms like *Modified Booth* as well as combinational *N-Bit* × *M-Bit* multiplication. The Modified Booth algorithms [44] considered were Booth2, Booth3, and Booth4, as higher radix Booth recodings incur an excessive number of gates, as discussed in Section 6.4.5. The N-Bit × M-Bit algorithms considered were 2-Bit × 2-Bit, 2-Bit × 3-Bit, 2-Bit × 4-Bit, and 3-Bit × 3-Bit combinational multiplication, since larger operand implementations are not competitive in terms of gate count, as discussed in Section 6.4.9. For all of these algorithms both dual-rail and quad-rail encodings were assessed and compared in terms of throughput and area to determine that the dual-rail pipelined Modified Baugh-Wooley MAC achieves highest throughput with the fewest number of gates. The next best performing approach is dual-rail Modified Booth2, which was also implemented as both a pipelined and non-pipelined design for comparison. For each design in Section 6.3, the circuit operation, optimization, and performance are discussed in that order. Unless otherwise stated, designs are implemented in dual-rail logic.

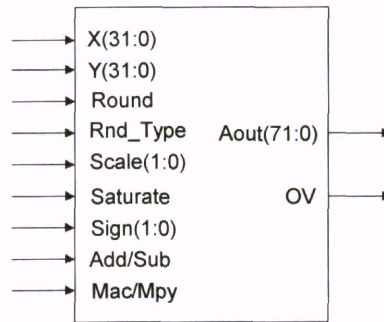

Figure 72. MAC block diagram.

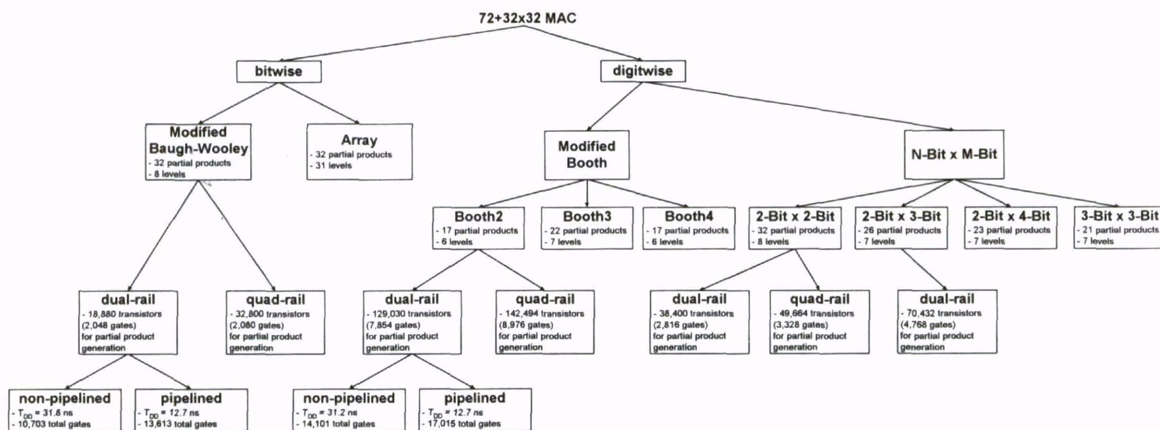

Figure 73. Taxonomy of 72+32×32 MAC.

6.3.1 Non-Pipelined Modified Baugh-Wooley MAC

6.3.1.1 Operation

The structure of the non-pipelined Modified Baugh-Wooley MAC is shown in Figure 74. NCL enables several optimizations as discussed in Section 6.3.1.2. In Phase 1, the multiplication begins by generating all of the partial products that can be generated in one gate delay. Next, these partial products are used in the first level of the Wallace tree, while the last row of partial products and most significant bit of each partial product,

requiring two gate delays, are generated. Concurrently, the previous value in the accumulator is shifted, if necessary, to account for the type of multiplication being performed. It is complemented if the result is to be subtracted from the accumulator, or is zeroed if multiply only is specified. Next, the modified accumulator and the uncombined partial products are used, along with the output from the first level of the Wallace tree, as the input to the second level of the Wallace tree. After this, there are six more Wallace tree levels before the partial products are reduced to two 65-bit words, where a ripple-carry addition is performed. The rationale for selecting a ripple-carry adder is detailed in Section 6.4.

During the summation of the partial products in Phase 1, Phase 2 begins with the multiply sign and the accumulate sign being generated as inputs to overflow detection. Also, the control signals are ensured for input-completeness in order for the MAC to remain delay-insensitive, as described in Chapter 2. After the ripple-carry addition, the result is again shifted if necessary to account for the type of multiplication being performed and is complemented if the result is to be subtracted from the accumulator.

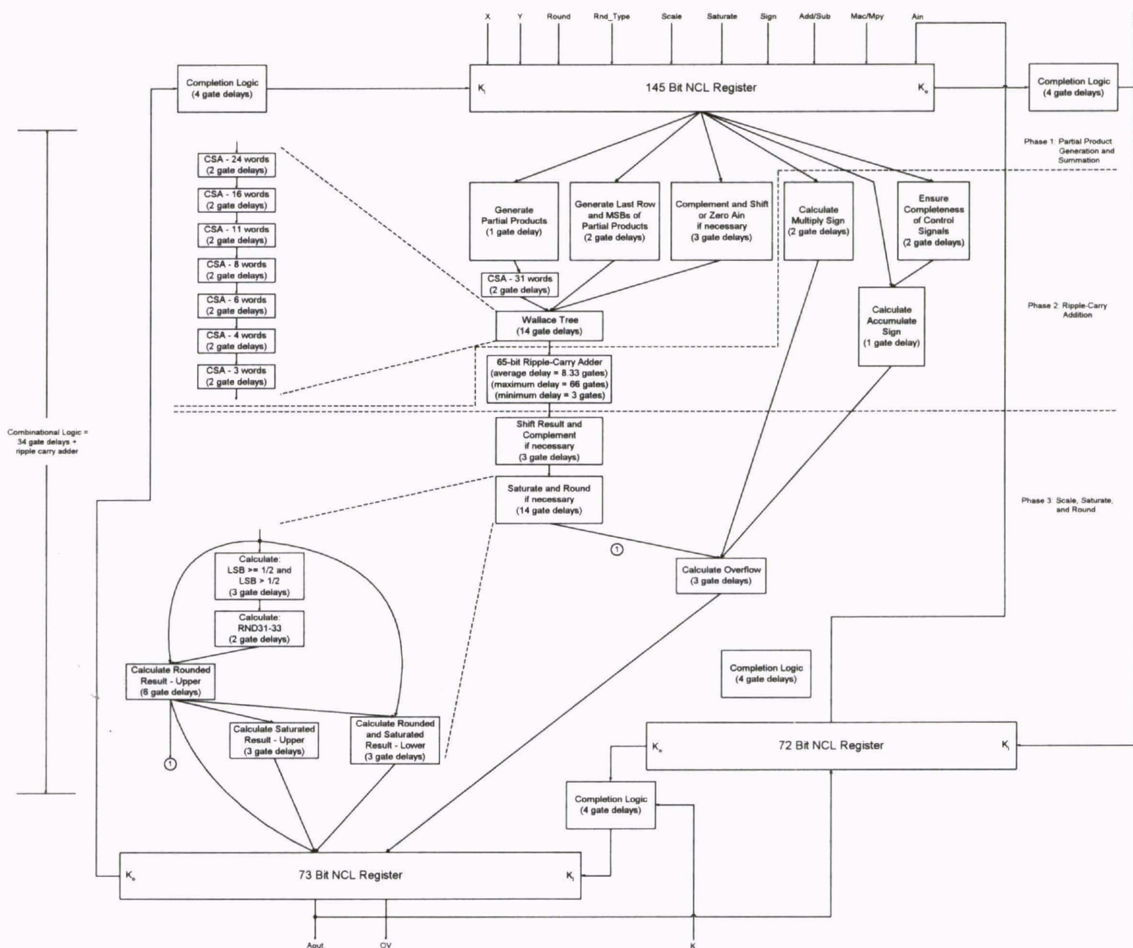

Figure 74. Non-pipelined Modified Baugh-Wooley MAC.

In Phase 3, the result can then be rounded and saturated if required. To round the result it is determined if the lower portion (LSB) is greater than or equal to 0.5, greater than 0.5, or less than 0.5. The LSB is contained in either the lower 31, 32, or 33 bits, depending on whether up-scaling, no scaling, or down-scaling is selected, respectively, as shown in Figure 75. After this is determined, a rounding bit is generated to be added to the upper portion of the result (MSB), based on the LSB and the selected rounding algorithm, either 2^s-complement or convergent rounding, described in Algorithm 6.1 and

Algorithm 6.2, respectively. Next, this bit, either RND31, RND32, or RND33, is added to the MSB of the result using a carry-lookahead adder. After the carry-lookahead addition, the result can then be saturated as shown in Table XIII, by checking bits 71, 64, and 63. While the result is processed by the saturation logic, the overflow bit is generated from bit 71 and the multiply and accumulate signs calculated earlier. The result is then output and fed back to the input register through an additional asynchronous register such that there are three registers in the feedback loop to prevent a lockup scenario as explained in Chapter 2.

a)

71 64	63 31	30 0
Extension	MSB	LSB

b)

71 64	63 32	31 0
Extension	MSB	LSB

c)

71 64	63 33	32 0
Extension	MSB	LSB

Figure 75. Output divisions for a) up-scaling, b) no scaling, and c) down-scaling.

```
if (LSB >= 0.5) then
        MSB = MSB + 1
else if (LSB < 0.5) then
        MSB = MSB
end if
LSB = 0
```

Algorithm 6.1. 2^s-complement rounding.

```
if (LSB > 0.5) then
        MSB = MSB + 1
else if (LSB < 0.5) then
        MSB = MSB
else if (LSB = 0.5) and (the least significant bit of MSB = 0) then
        MSB = MSB
else if (LSB = 0.5) and (the least significant bit of MSB = 1) then
        MSB = MSB + 1
end if
LSB = 0
```

Algorithm 6.2. Convergent rounding.

Table XIII. Saturation table.

B_{71}	B_{64}	B_{63}	Saturated Result	Saturated and Rounded Result
0	0	0	No Change	Result of Rounding Algorithm
0	0	1	00 7FFF FFFF	00 7FFF 0000
0	1	0	00 7FFF FFFF	00 7FFF 0000
0	1	1	00 7FFF FFFF	00 7FFF 0000
1	0	0	FF 8000 0000	FF 8000 0000
1	0	1	FF 8000 0000	FF 8000 0000
1	1	0	FF 8000 0000	FF 8000 0000
1	1	1	No Change	Result of Rounding Algorithm

6.3.1.2 Design Optimizations

There are two optimizations considered: the first is architectural and the second is NCL-specific. The first optimization deals with accumulation. The accumulator is shifted and complemented at the beginning and added to the second level of the Wallace tree, and the result is then shifted and complemented again following the ripple-carry addition to reduce the circuit delay. The shifting accounts for the various multiply types: (signed × signed), (signed × unsigned), and (unsigned × unsigned), while the complementing is used for subtraction from the accumulator. The alternative is to shift

and 2^s-complement the two outputs of the Wallace tree and then accumulate. This approach results in four words to be summed before the ripple-carry addition: the accumulator, the two shifted and complemented partial products, and the extra bit to be added to the least significant bit of each partial product due to their required 2^s-complementing. In the second approach, the four extra words that need to be summed before the ripple-carry addition can begin require two carry-save adders. This optimization will always reduce the critical path by twice the worst-case propagation delay of a full adder. In this design four gate delays were eliminated from the critical path.

Other optimizations include partial product generation facilitated through completeness optimizations in NCL. All partial products except for the most significant bits and the last partial product are directly generated by AND functions. To ensure completeness of the X and Y inputs only the X_iY_j partial products, where $i = j$ and $30 \geq i, j \geq 0$, require the use of *complete AND functions*, developed in Chapter 3. The rest of the partial products, X_iY_j, where $i \neq j$, can be generated using *incomplete AND functions*, depicted in Figure 10. Since the incomplete AND functions require 14 fewer transistors than the complete AND functions, and can be used for 930 of the 961 AND functions required for partial product generation, a net total of 13,020 transistors were saved in this design.

6.3.1.3 Average Cycle Time Determination

To determine the average cycle time for the MAC, the average cycle time for a ripple-carry adder was required. A C-language program was written that calculates the number of occurrences of each possible number of gate delays for an N-bit ripple-carry adder, from the minimum number of three gate delays for no carries, to the maximum number of N+1 gate delays for a carry occurring at each adder. The program then calculates the weighted average of the number of occurrences of each scenario to determine the expected average number of gate delays for the N-bit ripple-carry adder, assuming that all inputs are equiprobable. With N = 65, as in this design, the program calculates T_{DD} = 8.33 gate delays. With the average number of gate delays for the ripple-carry adder known, the calculation of T_{DD} follows Algorithm 4.2 in Chapter 4, as the average number of gate delays through the combinational logic for both DATA and NULL plus the number of gate delays through the completion circuitry for both DATA and NULL. Since the delay in the completion logic is 4 gates and the number of gate delays through the combinational circuitry is 34 plus the average delay of the ripple-carry adder, determined to be 8.33 from the program, $T_{DD} = (2 \times 4) + (2 \times (34 + 8.33)) = 92.66$ gate delays, accounting for both the DATA and NULL cycle. Simulation results are presented in Section 6.3.5. Experience with the program for a range of values of parameter *N* indicates logarithmic behavior for the ripple-carry addition as corroborated by [45].

6.3.2 Non-Pipelined Modified Booth2 MAC

6.3.2.1 Operation

The structure of the non-pipelined Modified Booth2 MAC is shown in Figure 76. In Phase 1, the multiplication begins by generating all of the partial products and the shifted and complemented, or zeroed, accumulator value, since both of these operations require three gate delays. Next, the partial products and the modified accumulator are combined through the first of six levels of the Wallace tree. The two partial products output from the Wallace tree are used in a 67-bit ripple-carry addition. The Modified Booth2 MAC requires a 67-bit ripple-carry addition, verses the 65-bit ripple-carry addition required in the Modified Baugh-Wooley MAC, since the Modified Booth2 MAC has two less Wallace tree levels, each of which reduces the length of the ripple-carry addition by one.

During the summation of the partial products in Phase 1, Phase 2 begins with the multiply sign and the accumulate sign being generated as inputs to overflow detection. Also, the control signals and the multiplier and multiplicand, *X* and *Y*, respectively, are ensured for completeness in order to maintain delay-insensitivity. Both *X* and *Y* must be ensured here because they are not implicitly complete in the partial product generation circuitry, as they are in the Modified Baugh-Wooley design, ensured by selectively complete AND functions. After the ripple-carry addition, the result is again shifted, if necessary, to account for the type of multiplication being performed and is complemented if the result is to be subtracted from the accumulator.

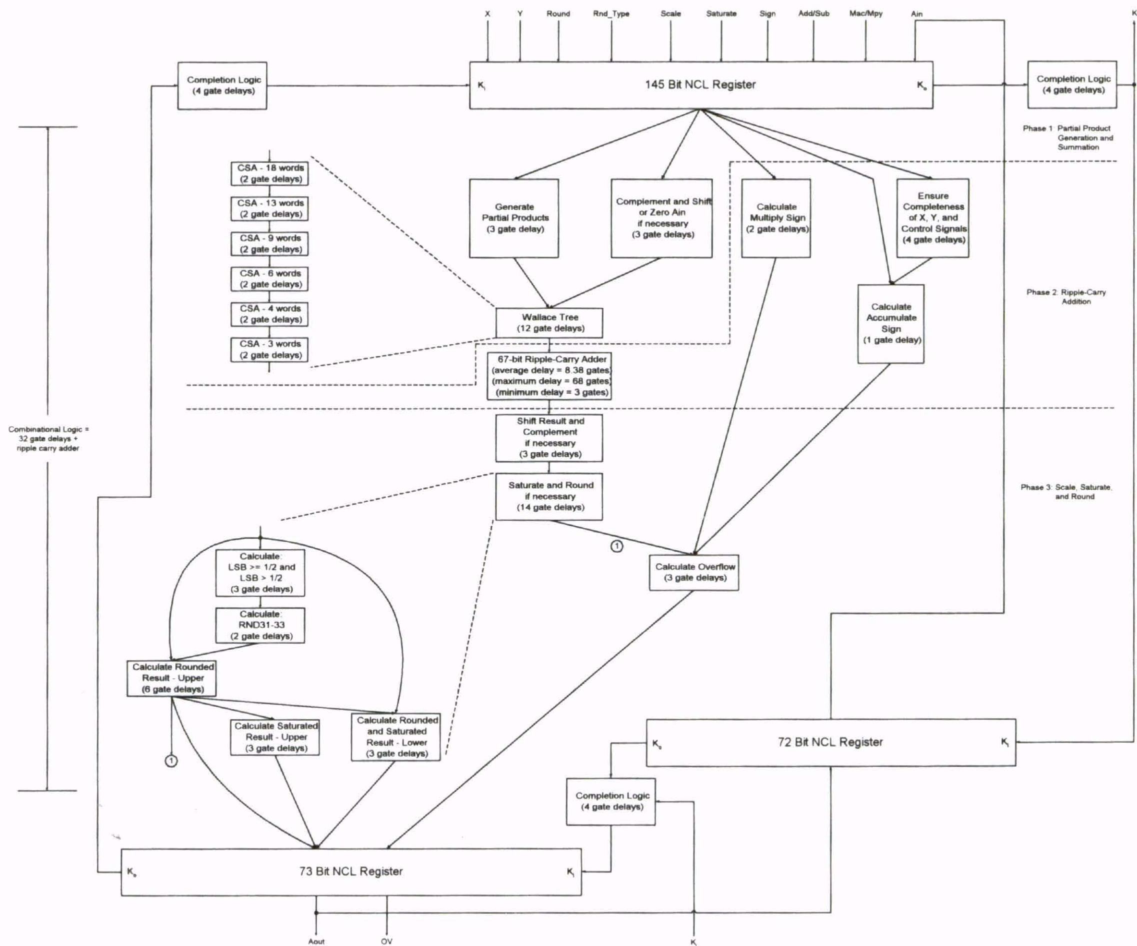

Figure 76. Non-pipelined Modified Booth2 MAC.

In Phase 3, the result can then be rounded and saturated if required and the overflow bit generated in exactly the same manner as for the Modified Baugh-Wooley MAC. The result is then output and fed back to the input register through an additional asynchronous register such that there are the required three registers in the feedback loop.

6.3.2.2 Design Optimizations

The same optimizations for selecting multiplication type and adding/subtracting the partial products to/from the accumulator used in the Modified Baugh-Wooley design, explained in Section 6.3.1.2, were implemented in the Modified Booth2 design.

6.3.2.3 Average Cycle Time Determination

T_{DD} can be calculated from Algorithm 4.2 in Chapter 2, as described in Section 6.3.1.3. Since the delay in the completion logic is 4 gates and the number of gate delays through the combinational circuitry is 32 plus the average of the ripple-carry adder determined to be 8.38 from the C-program, $T_{DD} = (2 \times 4) + (2 \times (32 + 8.38)) = 88.76$ gate delays, accounting for both the DATA and NULL cycle. Therefore, the Modified Booth2 algorithm should be faster than the Modified Baugh-Wooley algorithm for the non-pipelined MAC designs.

6.3.3 Pipelined Modified Baugh-Wooley MAC

6.3.3.1 Operation

The structure of the pipelined Modified Baugh-Wooley MAC is shown in Figure 77. The first stage begins by generating all of the partial products that can be generated in one gate delay. Next, these partial products are used in the first level of the Wallace tree, while the remaining partial products that require two gate delays are generated. The remaining partial products, along with the output from the first level of the Wallace tree, are then used as the input to the second level of the Wallace tree.

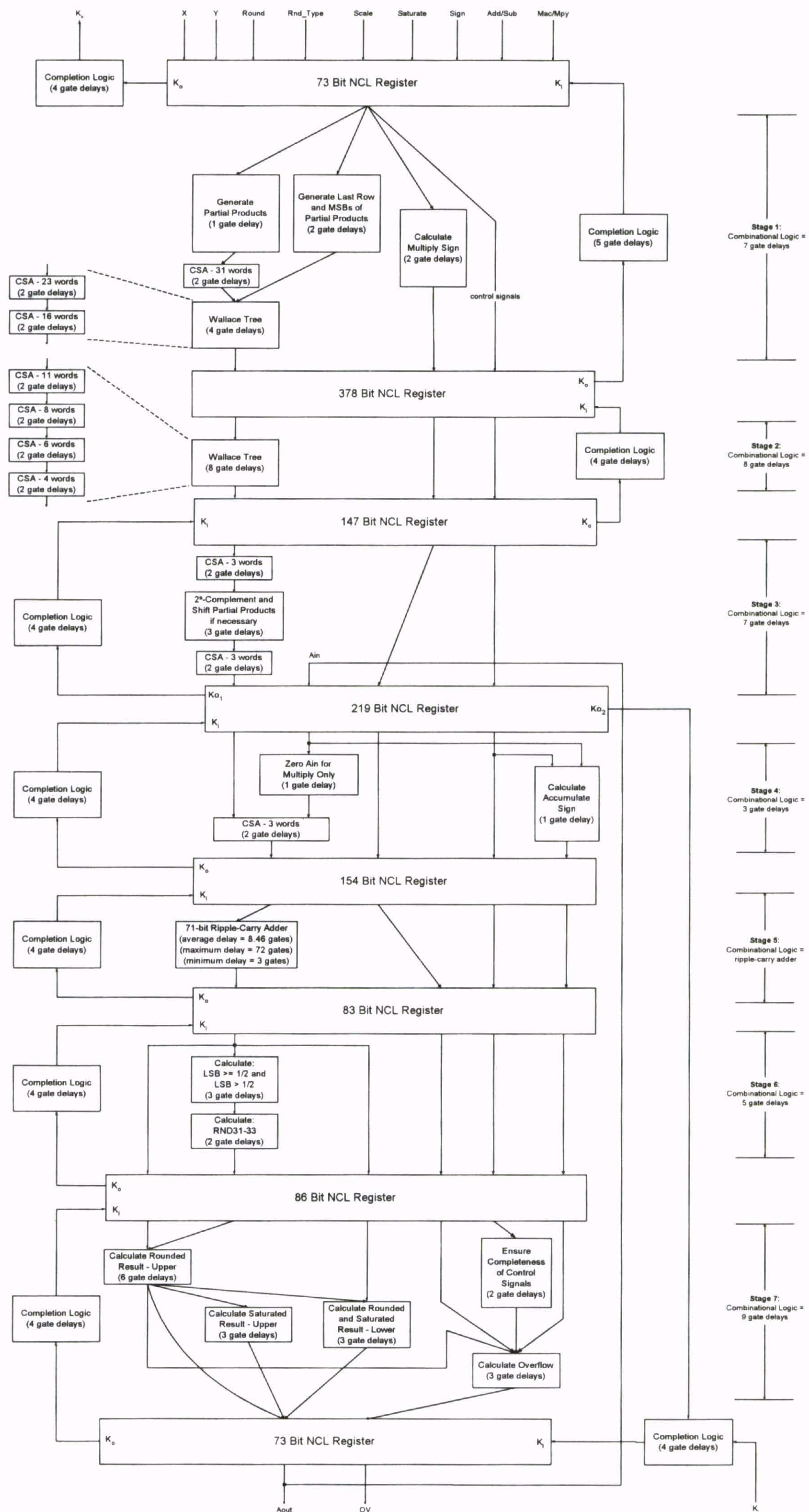

Figure 77. Pipelined Modified Baugh-Wooley MAC.

Stage 1 also contains the third level of the Wallace tree along with the multiply sign generation. The second stage consists of four more levels of the Wallace tree. Stage 3 begins with the final level of the Wallace tree, followed by the shifting and 2^s-complementing of the Wallace tree output, if necessary, to account for the type of multiplication being performed and for subtraction from the accumulator. The third stage also contains another carry-save adder, required because of the 2^s-complement operation. Stage 4 begins the feedback loop and contains the circuitry to zero *Ain* for the multiply only function and the final carry-save adder to add *Ain* to the Wallace tree output. The fourth stage also generates the accumulate sign. The fifth stage consists solely of a 71-bit ripple-carry adder. Stage 6 contains the first part of the rounding logic, while Stage 7 contains the remaining rounding logic along with the saturation circuitry, control signal completeness logic, and overflow detection circuitry, as explained in Section 6.3.1.1.

6.3.3.2 Throughput Maximization

An effective approach for pipelining a self-timed MAC begins with minimization of the feedback loop. This is in part because the feed-forward portion of the MAC can be pipelined to a fine granularity as long as completeness is ensured at each stage boundary. This enables the throughput of the feed-forward path to be at least as great as that of the feedback loop. To do this, it is preferable to postpone the addition of *Ain* with the partial products until absolutely necessary. Moreover, the subtraction and multiply mode selection method can be revised such that it reduces the number of operations required in the feedback loop. To increase throughput in the non-pipelined design, *Ain* was

complemented and shifted, or zeroed, and the result from the ripple-carry adder was complemented and shifted. However, for the pipelined design, the two outputs of the Wallace tree can be 2^s-complemented and shifted, allowing the shifting and complementing of *Ain* followed by the shifting and complementing of the result to be removed from the feedback loop. This is replaced instead by the 2^s-complementing and shifting of the final two partial products, followed by an extra carry-save adder in the feed-forward portion of the design. The zeroing of *Ain* for the multiply only function is still required to be performed within the feedback loop. In the pipelined implementation, this change eliminates five gate delays from the feedback path with no additional latency in the pipeline. The corresponding logic is relocated to the feed-forward portion of the design. Partitioning the feed-forward portion into three stages with a maximum of 8 gate delays per stage allows the inclusion of the additional logic without decreasing overall throughput.

After the feedback logic of the MAC is minimized, it can be pipelined by inserting asynchronous registers as described in Chapter 4. It was shown in [38] that a feedback loop containing *N* tokens, where a token is defined as a DATA wavefront with corresponding NULL wavefront, requires 2N bubbles for maximum throughput, where a bubble is defined as either a DATA or NULL wavefront occupying more than one neighboring stage. This allows for each DATA and NULL wavefront to move through the feedback loop independently. Since the feedback loop in the MAC design only contains one token, two bubbles are necessary to maximize throughput. A token requires two stages, one stage for the DATA portion and one stage for the NULL portion, while

each bubble requires one stage. Therefore, the feedback loop was partitioned into four stages for maximum throughput.

The front end of the feedback loop was partitioned as shown in Figure 77. Partitioning of the ripple-carry adder is not advisable since this would incur extra gate delays on the critical path. Inserting a register in the middle of the ripple-carry addition would tend to lessen the benefits of its asynchronous behavior by increasing the $O(\log_2 N)$ average time for an N-bit ripple-carry addition, since $\log_2 N1 + \log_2 N2 > \log_2 N$; where $N = N1 + N2$, $N \geq 6$, and $N1, N2 \geq 3$. The last two stages were divided to minimize the worst-case delay of each stage. The Upper Rounding logic for the most significant 41 bits of the result can be partitioned into a 5 gate delay circuit followed by a 1 gate delay circuit, without violating the input-completeness criteria. Alternately, inserting a register between this partition would result in Stage 6 having 10 gate delays and Stage 7 having 4 gate delays. The 10 gate delays of Stage 6 in this alternate design would exceed the 9 gate delays of Stage 7 in the current design. Furthermore, simulation shows both finer and coarser partitionings decrease throughput.

Throughput can be further increased using partial bitwise completion, described in Chapter 4, where the feed-forward output joins the feedback input. Two separate completion logic blocks are appropriate. The first, whose input is Ko_1, only acknowledges the inputs from the feed-forward circuit; the second, whose input is Ko_2, only acknowledges the feedback inputs. This optimization can decrease the inter-dependencies between the feedback loop and the feed-forward path to boost throughput an additional 2%.

Finally, the feed-forward portion is pipelined such that its throughput is at least as great as that of the feedback loop. In other words, the output from the feed-forward portion of the design must always be available when the feedback input is ready. Therefore, the minimum forward path through the feedback loop must be determined. Since the minimum delay through a ripple-carry adder is 3 gates and the delay for each register is 1 gate, the minimum forward path through the feedback loop is $3 + 3 + 5 + 9 + (5 \times 1) = 25$ gate delays, as indicated on the right side of Figure 77. In order to ensure that the feedback loop will never wait on input from the feed-forward portion, the maximum cycle time of the feed-forward pipeline must not exceed 25 gate delays. Decreasing the cycle time of the feed-forward portion to less than 25 gate delays will not increase the throughput as a whole. Therefore, this MAC optimization problem is transformed to ensuring a maximum cycle time of 25 gate delays for the feed-forward portion of the design, while adding as few asynchronous registers as possible. Following the method described in Chapter 4 for pipelining NCL circuits, it was determined that the addition of two asynchronous registers, as shown in Figure 77, would result in a maximum cycle time of 24 gate delays for the feed-forward circuitry. Furthermore, simulation shows that finer partitioning does not increase throughput, while coarser partitioning decreases throughput.

6.3.4 Pipelined Modified Booth2 MAC

6.3.4.1 Operation

The structure of the pipelined Modified Booth2 MAC is shown in Figure 78. The first stage begins by generating all of the partial products, which are then input to the first of two levels of the Wallace tree. Stage 1 also contains the multiply sign generation and the completeness generation for the multiplier and multiplicand, *X* and *Y*, respectively, since they are not implicitly complete in the partial product generation circuitry. The second stage consists of three more levels of the Wallace tree. Stage 3 begins with the final level of the Wallace tree, followed by the shifting and 2^s-complementing of the Wallace tree output, if necessary, to account for the type of multiplication being performed and for subtraction from the accumulator. The third stage also contains another carry-save adder, required because of the 2^s-complement operation. Stage 4 begins the feedback loop and contains the circuitry to zero *Ain* for the multiply only function and the final carry-save adder to add *Ain* to the Wallace tree output. The fourth stage also generates the accumulate sign. The fifth stage consists solely of a 71-bit ripple-carry adder. Stage 6 contains the first part of the rounding logic, while Stage 7 contains the remaining rounding logic along with the saturation circuitry, control signal completeness logic, and overflow detection circuitry, as detailed in Section 6.3.1.1.

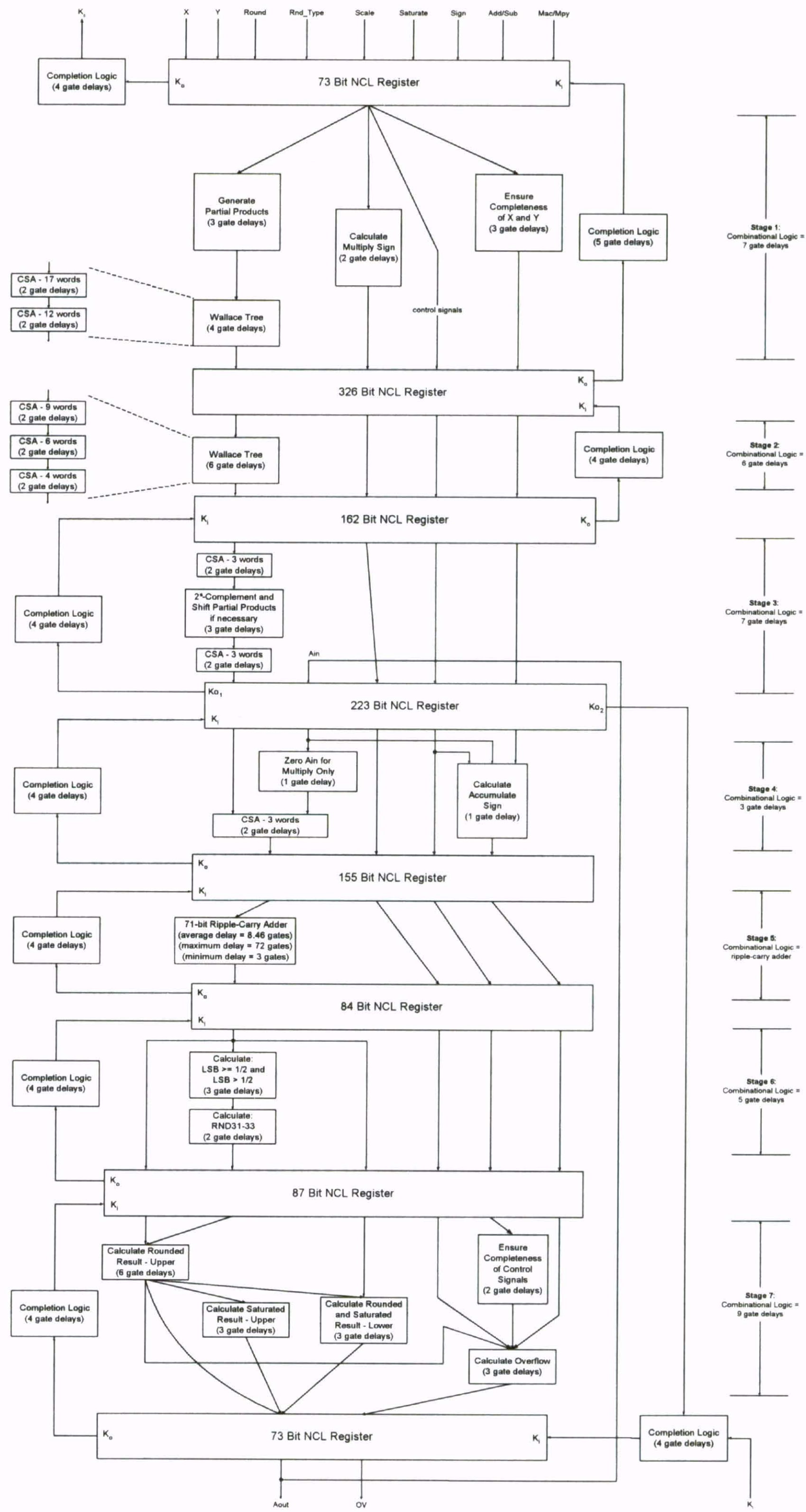

Figure 78. Pipelined Modified Booth2 MAC.

6.3.4.2 Throughput Maximization

The throughput maximization procedure for the feedback loop follows that of the pipelined Modified Baugh-Wooley design, explained in Section 6.3.3.2. The minimum forward path through the feedback loop is also 25 gate delays, and is independent of the selected multiplication algorithm. Addition of as few as two asynchronous registers, as shown in Figure 78, results in a maximum cycle time of 24 gate delays for the feed-forward portion. Since the feedback loop for the pipelined Modified Booth2 and Baugh-Wooley designs are the same, and the feedback loop is the limiting factor of throughput maximization for each, the two designs should have the same throughput.

6.3.5 Simulation Results

Before the average cycle time was determined for the designs, each was extensively tested with various data patterns and control inputs to verify correct operation. Once correct operation is established, representative MAC operations need to be selected to provide an adequate comparison of their throughputs. A candidate operation is Aout = $\sum_{i=0}^{N}(X_i \times Y_i)$; where $X_i = X_0 + (2^{-21} \times i)$ and $Y_i = Y_0 + (2^{-11} \times i)$ with N chosen to be 255. This allows a variety of computations to be performed such that any unusually short or long operations will not significantly skew the average cycle time. For instance, in my testbench X_0 and Y_0 were randomly selected such that X_0 = A61C039Dh = -0.702270077076 and Y_0 = F0046718h = -0.124865639955. Also, (signed × signed) multiplication was selected and rounding, scaling, and saturation were disabled. The same operation was also performed in a C-language program and the result

from this program agreed with the results from each of the simulated designs:

Aout = 05A0B13C0E04A37000h = 11.2554087704.

Both the non-pipelined and pipelined Modified Baugh-Wooley and Booth2 MAC designs were simulated using Synopsys in order to compare their throughputs to ensure that the relative values were consistent with the predicted results. The Synopsys technology library for the NCL gates is based on static 3.3V, 0.25 μm CMOS implementations. The average cycle time, T_{DD}, for the non-pipelined Modified Baugh-Wooley MAC was determined to be 31.8 ns; while T_{DD} for the non-pipelined Modified Booth2 MAC was determined to be 31.2 ns. Therefore, the non-pipelined Modified Booth2 MAC is faster than the non-pipelined Modified Baugh-Wooley MAC, as anticipated in Section 6.3.2.3. As for the pipelined designs, the Modified Baugh-Wooley and Booth2 MACs were anticipated to run at the same speed due to the fact that the feedback path was the same in both designs. The simulations of the two pipelined designs confirm this since they both have an average cycle time of 12.7 ns.

6.4 Carry-Propagate Adder Comparison

In [45] it was shown that the worse-case throughput for an N-bit ripple-carry adder was O(N), verses the $O(\log_2 N)$ worse-case throughput for an N-bit carry lookahead adder, when using 2-input gates. Since NCL uses gates with a maximum of 4 inputs, the worse case throughput for an NCL carry-lookahead adder is proportional to $\log_4 N$. Consider the 4-bit carry-lookahead adder depicted in Figure 79. Each of the AND and OR gates can be replaced with incomplete versions of the NCL AND and OR

functions, respectively, described in Chapter 2, while the XOR gates can be replaced with the NCL XOR function, developed in Chapter 3. The resulting design is complete with respect to all inputs. Likewise, a 4-bit ripple-carry adder can be constructed by connecting 4 full adders, shown in Figure 30, in series.

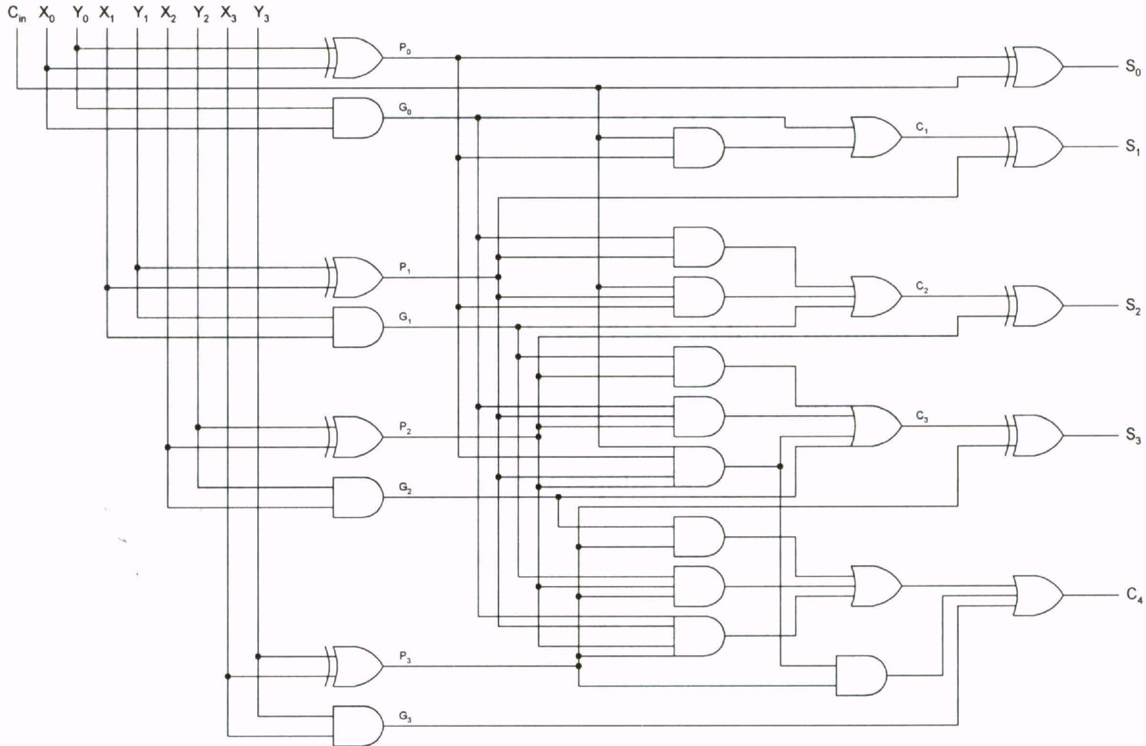

Figure 79. 4-bit carry-lookahead adder.

Table XIV compares the 4-bit versions of the carry-lookahead adder and the ripple-carry adder. It demonstrates that the two are comparable in terms of worse-case gate delays, but that the carry-lookahead adder requires more than three times as many gates. Comparing an N-bit addition using 4-bit carry-lookahead adders in series verses an N-bit ripple-carry adder, shows that the two approaches will require the same number of gate delays in the worst-case within a tolerance of ±1, depending on the size of N.

Furthermore, the 4-bit carry-lookahead adder described above is not fully observable due to redundancies in the carry calculations. To make it fully observable would require additional logic gates and logic levels, thus making it even less desirable.

Table XIV. Propagation delay and gate count for 4-bit adders.

	Gate Delays					Gate Count
	S_0	S_1	S_2	S_3	C_4	
Carry-Lookahead Adder	2	4	4	4	4	54
Ripple-Carry Adder	2	3	4	5	4	16

Another option is to construct an N-bit carry-lookahead adder, such that all carries are generated in parallel. Take for example the 71-bit addition required for the pipelined MACs designed in this chapter. To generate S_{70} requires a 71-bit AND function and a 71-bit OR function. Both of these functions require $O(\log_4 71)$, however portions can be performed in parallel, such that the two functions together only require 7 gate delays. Adding an additional gate delay for the *generate* and *propagate* calculation as well as for the final XOR function, causes the worse-case delay to be 9 gates. This is much smaller than the 72 gate worse-case delay of a 71-bit ripple-carry adder. However, since NCL is a delay-insensitive paradigm, its throughput is determined by the average-case delay and not the worse-case delay. Furthermore, the average-case delay for an N-bit ripple carry adder is only $O(\log_2 N)$ [45], which is 8.46 gate delays for a 71-bit ripple-carry addition, as determined by the C-language program described in Section 6.3.1.3. The average-case delay for the carry-lookahead adder would also be slightly reduced, but not by much since many of the path lengths are synonymous with the worse-case delay. Therefore, the

average-case delays for the 71-bit ripple-carry adder and carry-lookahead adder are comparable.

Above it was shown that the 4-bit carry-lookahead adder required more than three times the number of gates required by the 4-bit ripple-carry adder; therefore the 71-bit carry-lookahead adder will require at least three times the number of gates as the 71-bit ripple-carry adder. This indicates that the 71-bit ripple-carry adder would be preferred over the 71-bit carry-lookahead adder since they have comparable average-case delays and the ripple-carry adder is much smaller. Moreover, the 71-bit carry-lookahead adder described above is not fully observable. To make it fully observable would require additional logic gates and logic levels, thus making it even less desirable. Extending the above analysis to adders of arbitrary length, it can be stated that for any value of N, a NCL ripple-carry adder should outperform the identically sized NCL carry-lookahead adder.

6.5 Gate Requirements for Proposed Designs

In Section 6.3.3.2 and Section 6.3.4.2 it was shown that the throughput of a pipelined self-timed MAC design is limited by the feedback loop, independent of the feed-forward portion. This is due to the fact that the feed-forward portion can be readily pipelined to a fine granularity to match or exceed the throughput of the feedback loop. Since the feedback loop performs accumulation independent of the selected multiplication algorithm, the throughput of the MAC as a whole is independent of the

multiplication algorithm. This is demonstrated by the pipelined versions of the Modified Baugh-Wooley and Booth2 MACs operating with the same cycle time.

The design objective stated in the abstract is to obtain the highest throughput MAC using the fewest gates. Since the throughput of the pipelined MAC does not depend on the multiplication algorithm, the MAC throughput optimization problem can be transformed into the selection of the multiplication algorithm that requires the least amount of area to implement. The following sections will compare various algorithms to determine which requires the least gate count.

6.5.1 Modified Baugh-Wooley MAC

Since both the non-pipelined and pipelined designs were implemented in VHDL, the actual number of gates can be tabulated. The non-pipelined design requires 10,703 gates, while the pipelined design uses 13,613 gates, as shown in Figure 73. For both of these designs approximately 2,048 gates were from partial product generation with 32 complete AND functions and 992 incomplete AND functions.

6.5.2 Modified Booth2 MAC

Since both the non-pipelined and pipelined versions of this design were also implemented in VHDL, the actual number of gates can again be tabulated. The non-pipelined design used 14,101 gates, while the pipelined design used 17,015 gates, as shown in Figure 73. For both of these designs approximately 7,854 gates were from the partial product generation. Even though the Booth2 recoding eliminates two levels in the

Wallace tree, the additional gates required in the partial product generation outpace the savings. This causes the pipelined Modified Booth2 design to contain 3,402 more gates than the pipelined Modified Baugh-Wooley design. The Modified Booth2 MAC requires 405 fewer adders, which is 1,620 fewer gates, since each adder contains four gates. However, it requires approximately 5,806 additional gates for partial product generation. Since both designs operate with the same cycle time, the preferred design is the pipelined Modified Baugh-Wooley MAC, since it requires less area. This is even more evident when the number of transistors for partial product generation is compared. Since the number of transistors for the Modified Baugh-Wooley partial product generation can be greatly reduced as explained in Section 6.3.1.2, even though the number of gates remain the same, the transistor requirement for partial product generation of the two designs magnifies this differential, as shown in Figure 73. The partial product generation for the Modified Booth2 design requires 3.8-fold more gates than for the Modified Baugh-Wooley design, but 6.8-fold more transistors, due to the more sophisticated gates required in the recoding logic.

6.5.3 Array MAC

Both the Array MAC and the Modified Baugh-Wooley MAC use the same logic to generate the partial products and both require O(N) area for the partial product summation, as explained in Section 6.2. However, the Modified Baugh-Wooley MAC only requires $O(\log_2 N)$ gate delays for the partial product summation, while the Array MAC requires O(N) gate delays. Therefore, many more asynchronous registers would be

required to partition the feed-forward circuitry of the Array MAC than the two required for the Modified Baugh-Wooley MAC, in order to achieve the same throughput. Hence, the Array MAC would require approximately the same number of adders as the Modified Baugh-Wooley MAC, but would require many more asynchronous registers, causing it to contain many more gates than the Modified Baugh-Wooley MAC. However, the structure of the Array MAC is very regular compared to the irregular structure of the Modified Baugh-Wooley MAC, which could make it more desirable when layout is taken into consideration, despite its larger size.

6.5.4 Modified Booth3 MAC

The Modified Booth3 multiplication algorithm partitions the multiplier into overlapping groups of four bits, each of which selects a partial product from the following list: +0, +M, +2M, +3M, +4M, -4M, -3M, -2M, -M, and –0, where *M* represents the multiplicand. For the 32-bit × 32-bit multiplication, this decoding theoretically reduces the number of partial products from 17 for the Modified Booth2 algorithm to only 11. However, the +3M and –3M partial products cannot be obtained by simple shifting and/or complementing, like the others. These partial products are referred to as hard multiples. Therefore, two actual partial products must be used to represent each theoretical partial product to avoid the ripple-carry addition that would be required to compute both the +3M and –3M partial products. Any +3M partial product is represented by a +2M and a +M partial product, while any -3M partial product is represented by a -2M and a -M partial product. Since each theoretical partial product must be represented

by two partial products, the actual number of partial products for the Modified Booth3 MAC is 22, and the number of Wallace tree levels required to sum these partial products is 7. This is more than the 17 partial products required for the Modified Booth2 design, which can be summed using only 6 Wallace tree levels. Therefore, a Modified Booth3 MAC requires more adders to sum the partial products than would the Modified Booth2 MAC. Furthermore, the partial product generation requires scanning four multiplier bits at a time for the Modified Booth3 algorithm, verses only three bits which are simultaneously scanned in the Modified Booth2 algorithm. This requires more complex recoding logic for the Modified Booth3 algorithm. Since the Modified Booth3 algorithm requires more adders and more recoding logic than the Modified Booth2 algorithm, and increases the depth of the Wallace tree, it requires more gates than the Modified Booth2 design.

6.5.5 Modified Booth4 MAC

The Modified Booth4 multiplication algorithm also suffers from the problem of hard multiples. It partitions the multiplier into overlapping groups of five bits, each of which selects a partial product from the following list: +0, +M, +2M, +3M, +4M, +5M, +6M, +7M, +8M, -8M, -7M, -6M, -5M, -4M, -3M, -2M, -M, and –0, where *M* represents the multiplicand. The hard multiples are +3M, +5M, +6M, +7M, -7M, -6M, -5M, and -3M. However, if the hard multiples were to be generated through ripple-carry addition, the +6M and –6M multiples could be obtained simply by shifting the +3M and –3M multiples, respectively. For the 32-bit × 32-bit multiplication, this decoding theoretically

reduces the number of partial products from 17 for the Modified Booth2 algorithm to only 9. However, since the hard multiples require two partial products to represent each theoretical partial product, the actual number of partial products required is 17. The most significant partial product cannot be a hard multiple and therefore only requires one partial product for its representation. The actual number of partial products for the Modified Booth4 MAC is the same as for the Modified Booth2 MAC. The only difference is the partial product generation, which requires scanning five multiplier bits at a time for the Modified Booth4 algorithm, verses only three bits which are simultaneously scanned in the Modified Booth2 algorithm. This requires more complex recoding logic for the Modified Booth4 algorithm. Therefore, the Modified Booth4 MAC requires more gates than the Modified Booth2 MAC. Furthermore, higher radix Modified Booth algorithms can be expected to exhibit similar characteristics.

6.5.6 Combinational 2-Bit × 2-Bit MAC

The 2-Bit × 2-Bit partial product generation partitions both the multiplier and multiplicand into 16 groups of two bits that do not overlap. Each 2-bit multiplier, 2-bit multiplicand pair generates 4 bits of partial product. Every 2-bit multiplier group generates two rows of partial products since each 2-bit multiplier, 2-bit multiplicand pair generates 4 bits and each consecutive group of 4 bits is shifted two places due to the 2-bit partitioning of the multiplicand. This results in consecutive groups of 4 bits generated from one 2-bit multiplier group to be overlapped by two bits. Since there are sixteen 2-bit multiplier groups and each group generates two partial products, there are a total of 32

partial products. Since this number of partial products is the same as for the Modified Baugh-Wooley design, both designs will require the same number of gates to sum the partial products. Therefore, the only difference between the two designs is the partial product generation. The 2-Bit × 2-Bit partial product generation requires approximately 2,816 gates, while the Modified Baugh-Wooley partial product generation only requires approximately 2,048 gates, as shown in Figure 73. Hence, the 2-Bit × 2-Bit algorithm requires approximately 768 more gates than does the Modified Baugh-Wooley algorithm, making it less area efficient. This is even more evident when the transistor count for the partial product generation is compared. The Modified Baugh-Wooley partial product generation requires approximately 18,880 transistors, while the 2-Bit × 2-Bit partial product generation requires approximately 38,400 transistors, more than twice as many.

6.5.7 Combinational 2-Bit × 3-Bit MAC

The 2-Bit × 3-Bit partial product generation partitions the multiplier into 16 groups of two bits, and the multiplicand into 10 groups of three bits with 1 group of two bits, such that no groups overlap. Each 2-bit multiplier, 3-bit multiplicand pair generates 5 bits of partial product. Every 2-bit multiplier group generates two rows of partial products since each 2-bit multiplier, 3-bit multiplicand pair generates 5 bits and each consecutive group of 5 bits is shifted three places due to the 3-bit partitioning of the multiplicand. All two-row partial products generated from one 2-bit multiplier group contain an unused slot every third bit position, such that every third bit position in a two-row partial product only contains one bit rather than two bits, as in the other bit positions.

Since there are sixteen 2-bit multiplier groups and each group generates two partial products, 32 partial products are anticipated. However, because of the unused slots, there are actually only 26 rows of partial products, which can be summed in 7 Wallace tree levels. The multiplier could also be partitioned into 10 groups of three bits with 1 group of two bits, with the multiplicand partitioned into 16 groups of two bits, such that no groups overlap. This alternate partitioning also produces 26 rows of partial products. Recall that the Booth2 design, which has 17 rows of partial products that can be summed in 6 levels of Wallace tree, saved 405 adders or 1,620 gates in the partial product summation, as discussed in Section 6.5.2. Since the 2-Bit × 3-Bit algorithm requires 26 rows of partial products, which can be summed in 7 Wallace tree levels, this algorithm cannot utilize fewer adders than the Booth2 algorithm. Therefore, the number of gates saved by the reduced Wallace tree of the 2-Bit × 3-Bit algorithm is no more than 1,620. The number of gates required to generate the partial products for the 2-Bit × 3-Bit algorithm is approximately 4,768, a difference of approximately 2,720 additional gates than for the Modified Baugh-Wooley partial product generation. Therefore, the 2-Bit × 3-Bit algorithm would require at least 1,100 more gates than the Modified Baugh-Wooley design since it can save no more than 1,620 gates in the Wallace tree and requires an additional 2,720 gates for partial product generation.

6.5.8 Combinational 2-Bit × 4-Bit MAC

The 2-Bit × 4-Bit partial product generation partitions the multiplier into 16 groups of two bits, and the multiplicand into 8 groups of four bits, such that no groups

overlap. Each 2-bit multiplier, 4-bit multiplicand pair generates 6 bits of partial product. Every 2-bit multiplier group generates two rows of partial products since each 2-bit multiplier, 4-bit multiplicand pair generates 6 bits and each consecutive group of 6 bits is shifted four places due to the 4-bit partitioning of the multiplicand. All two-row partial products generated from one 2-bit multiplier group contain two unused slots every fourth bit position, such that for every four bit positions in a two-row partial product only two contain two bits while the other two contain only one bit. Since there are sixteen 2-bit multiplier groups and each group generates 2 partial products, 32 partial products are anticipated. However, because of the unused slots, there are actually only 23 rows of partial products, which can be summed in 7 Wallace tree levels. The multiplier and multiplicand could also be partitioned vise-versa, resulting in the same number of partial product rows. Since this design also requires 7 Wallace tree levels, as did the 2-Bit × 3-Bit design, it could not possibly save more than 1,620 gates in the Wallace tree, as explained in Section 6.5.7. The partial product generation is also more complicated than for the 2-Bit × 3-Bit partial product generation since more inputs are required. Therefore, partial product generation for this design requires at least as many gates as for the 2-Bit × 3-Bit design. Hence, this design must require more gates than the Modified Baugh-Wooley MAC, following the logic of Section 6.5.7.

6.5.9 Combinational 3-Bit × 3-Bit MAC

The 3-Bit × 3-Bit partial product generation partitions both the multiplier and multiplicand into 10 groups of three bits, with one group of two bits, such that no groups

overlap. Each 3-bit multiplier, 3-bit multiplicand pair generates 6 bits of partial product. Every 3-bit multiplier group generates two rows of partial products since each 3-bit multiplier, 3-bit multiplicand pair generates 6 bits and each consecutive group of 6 bits is shifted three places due to the 3-bit partitioning of the multiplicand, such that all consecutive groups of 6 bits generated from one 3-bit multiplier group overlap by three bits. The last multiplier group is only two bits, so for each 2-bit multiplier, 3-bit multiplicand pair, 5 bits of partial product are generated. This 2-bit multiplier group generates two rows of partial products since each 2-bit multiplier, 3-bit multiplicand pair generates 5 bits and each consecutive group of 5 bits is shifted three places due to the 3-bit partitioning of the multiplicand. These last two rows of partial products contain an unused slot every third bit position, such that every third bit position in the last two-row partial product only contains one bit rather than two bits, as in the other bit positions. Since there are ten 3-bit multiplier groups and one 2-bit multiplier group, each of which generates 2 partial products, 22 partial products are anticipated. However, because of the unused slots generated by the 2-bit multiplier group, there are actually only 21 rows of partial products, which can be summed in 7 Wallace tree levels. Since this design also requires 7 Wallace tree levels, as did the 2-Bit × 3-Bit design, it could not possibly save more than 1,620 gates in the Wallace tree, as explained in Section 6.5.7. The partial product generation is also more complicated than for the 2-Bit × 3-Bit partial product generation since more inputs are required. Therefore, partial product generation for this design requires at least as many gates as for the 2-Bit × 3-Bit design. Hence, this design must require more gates than the Modified Baugh-Wooley MAC, following the logic of

Section 6.5.7. Furthermore, any larger sized N-Bit × M-Bit algorithms would not be likely to reduce the number of gates due to their increasing complexity.

6.5.10 Quad-Rail MACs

To test the feasibility of quad-rail multiplication, a quad-rail 4-bit × 4-bit unsigned multiplier was designed, implemented, and tested. The resulting design operated with the same throughput as its dual-rail counterpart but required slightly more than twice as many gates, showing that a quad-rail encoding is not as efficient for realizing multiplication. Furthermore, quad-rail partial product generation circuitry was designed for each of the algorithm types shown in Figure 73; and the resulting quad-rail designs required at least 2% more gates and 10% more transistors than their dual-rail counterparts.

6.6 Conclusion

In Section 6.3 it was shown how to design and then pipeline both a self-timed Modified Baugh-Wooley MAC and Modified Booth2 MAC in order to achieve maximum throughput. Throughput maximization was accomplished by first minimizing the feedback loop and then partitioning the feed-forward path such that its throughput was at least as great as that of the feedback loop, since the feedback loop was determined to be the limiting factor to increasing throughput. Section 6.3 also showed that the feedback loop did not depend on the chosen multiplication algorithm, and therefore the throughput also did not depend on the multiplication algorithm, although a faster

multiplication algorithm would decrease latency of an isolated multiply. This was substantiated through simulations of both the pipelined Modified Baugh-Wooley MAC and the pipelined Modified Booth2 MAC, which both had the same throughput.

Since it was shown that the throughput of the MAC did not depend on the multiplication algorithm, the self-timed MAC throughput optimization problem was transformed into selecting the multiplication algorithm requiring the fewest gates. Section 6.5 compared the area of multiple MAC designs using various multiplication algorithms. The best design is therefore the one that requires the fewest number of gates to implement. It was also shown in Section 6.5 that the pipelined Modified Baugh-Wooley design required the least amount of area, and was therefore the best design based on the criteria of the highest throughput with the least area. The dual-rail pipelined Modified Baugh-Wooley MAC yielded a speedup of 2.5 over its initial non-pipelined version and required 20% fewer gates than the dual-rail pipelined Modified Booth2 MAC that operated with the same throughput.

Table XV compares this optimized NCL MAC to other delay-insensitive/self-timed MACs in the literature, showing that the 3.3V, 0.25 μm CMOS NCL MAC outperforms the other designs. [41] describes a serial-parallel MAC using the methods and tools developed at Caltech [46] for design of delay-insensitive circuits. In [41] an 8+4×4 MAC was fabricated using 5V, 2 μm CMOS technology that operated at 37 ns; and an extrapolation to larger word sizes was presented. Using this extrapolation it was determined that a 64+32×32 MAC would operate at 901 ns, much slower than the NCL MAC, as expected, since the implemented algorithm is not fully parallel. [42] describes a

self-timed 16+8×8 MAC designed using SCCVSL (single-rail CMOS cascode voltage switch logic) and fabricated in 0.6 μm technology. This MAC employs the parallel Booth2 algorithm, and has an average cycle time of about 90 ns. A third self-timed MAC described in [43] was designed in single-ended dynamic logic [47], utilizing conditional evaluation along with the traditional Array multiplication algorithm. Conditional evaluation allows for rows with a zero bit product to be multiplexed around, to reduce energy and delay. In [43] a 16+8×8 MAC was simulated using 3.3V, 0.35 μm CMOS technology, to determine the average cycle time of 7.8 ns. This delay information was then used in [43] to estimate the average cycle time for a 32+16×16 MAC as approximately 24 ns. These comparisons indicate that the NCL-based dual-rail pipelined Modified Baugh-Wooley MAC developed herein outperforms the three above mentioned methods, even after technology adjustments. Furthermore, the NCL MAC supports rounding, scaling, and saturation, whereas the other MACs discussed herein do not. Without the rounding, scaling, and saturation the NCL MAC performance could be more than doubled.

Table XV. Algorithm, technology, and cycle time for various self-timed MACs.

MAC Type	Algorithm	Technology	Avg. Cycle Time
72+32×32	Modified Baugh-Wooley	3.3V, 0.25 μm CMOS	12.7 ns
64+32×32 [41]	Serial-Parallel	5V, 2 μm CMOS	901 ns
16+8×8 [42]	Modified Booth2	0.6 μm CMOS	90 ns
16+8×8 [43]	Conditional Evaluation	3.3V, 0.35 μm CMOS	7.8 ns
32+16×16 [43]	Conditional Evaluation	3.3V, 0.35 μm CMOS	24 ns

7.0 CONCLUSION

While much remains to be learned in regard to the application of NCL, the techniques developed herein provide a basis for the design and optimization of NCL systems. A method for designing optimized NCL combinational circuits was developed, as well as a method for pipelining these combinational circuits such that optimum throughput is achieved. Furthermore, a technique to mitigate the impact of the NULL cycle on throughput was presented.

7.1 Summary

When full minterm generation is not required, TCR can produce delay-insensitive circuits that require less area and fewer logic levels than alternative gate-level approaches, as demonstrated in Chapter 3. TCR is applicable when composing logic functions where each gate is a state-holding element. The TCR method combines techniques such as incomplete functions, quad-rail encodings, reduced minterm expressions, and factored minterm expressions for reducing gate count. It then employs a mapping of the factored minterm equations to a set of 27 macros, which constitute the set of all functions consisting of four or fewer variables. A number of case studies validate the utility and potential for automation of the proposed method. Using TCR methods,

design parameters including critical path delay, gate count, transistor count, and power can be readily traded-off and optimized.

These results were further extended to a gate-level pipelining strategy for circuits composed of state-holding elements to maximize throughput of combinational circuits produced by TCR methods in Chapter 4. Since the GLP method successively partitions an N-level NCL combinational logic design first into 2 stages, then further into as many as *N* stages, it can produce an optimal pipelined NCL system with significantly increased throughput over its original non-pipelined design. The GLP process may also be partially applied to design maximum throughput systems under the constraints of latency and/or area bounds. The GLP method combines both full-word completion as well as bit-wise completion for designing the optimal system. A case study of a 4×4 multiplier substantiates the utility and potential for automation of the proposed method, as the throughput of the non-pipelined 4×4 multiplier was increased by 125%. GLP was applied to a dual-rail NCL design in Chapter 4; but it can also be applied to a quad-rail NCL design, by inserting quad-rail registers, rather than dual-rail registers.

Although NCL requires both a DATA wavefront and a NULL wavefront, which reduces the maximum attainable throughput by approximately half, a technique can be used to reduce this inherent throughput loss. In Chapter 5, the NCR method of partitioning delay-insensitive systems into two concurrent paths such that one circuit processes a DATA wavefront, while its duplicate processes a NULL wavefront, thus significantly increasing throughput, was developed. A 4-bit by 4-bit multiplier case study indicates a speedup of 1.61 over the standalone design. Furthermore, this technique could

also be applied to other delay-insensitive methods [4, 6, 7, 8, 9] as well. Moreover, it is not necessary to duplicate the entire circuit when applying the NCR technique. Rather, its benefits can be obtained without doubling area and power requirements by applying it to selective portions of a circuit, which cannot be pipelined more finely due to the completeness of input criterion. Thus, throughput of a pipelined design with a small number of slow stages can be readily boosted with relatively little cost by using NCR.

Finally, the methods presented herein were applied to design a 72+32×32 MAC that outperformed other delay-insensitive/self-timed MACs in the literature, including a 32+16×16 design using single-ended dynamic logic, utilizing conditional evaluation along with the traditional Array multiplication algorithm. This method of conditional evaluation was analyzed in the context of NCL showing that it would require additional gates, greater power dissipation, and a larger cycle time when compared to the normal Array multiplication algorithm, making it undesirable for NCL implementation. This is due to the proportionality differences between the NCL full adder and select logic verses the same two components implemented in single-ended dynamic logic. Furthermore, the NCL MAC supports rounding, scaling, and saturation, whereas the other MACs discussed herein do not. Without the rounding, scaling, and saturation the NCL MAC performance could be more than doubled.

7.2 Future Work

The utility of the TCR and GLP methods has been demonstrated in Chapter 3 and Chapter 4, respectively. The next step is to incorporate both of these methods into the

Synopsys design tools such that NCL circuits can be synthesized from high level, algorithmic descriptions and can then be automatically pipelined to optimize throughput.

Moreover, the throughput of NCL systems can be further increased by applying an *early completion* method described in [40] or by applying *2D-pipelining* described in [48]. Early completion performs the completion detection for registration $stage_i$ at the input of the register, instead of at the output of the register as previously described. This method requires that the single-rail completion signal from registration $stage_{i+1}$, ko_{i+1}, be used as an additional input to the completion detection circuitry for registration $stage_i$, to maintain delay-insensitivity. However, early completion necessitates an assumption of equipotential regions [4], making the design potentially more delay-sensitive. 2D-pipelining not only partitions a circuit between functional component boundaries, but also between bit slices, forming a complex 2-dimentional pipeline.

In Chapter 5, NCR was applied to a dual-rail NCL design utilizing full-word completion. However, it can also be applied to a quad-rail NCL design, by modifying the Demultiplexer and the Multiplexer to handle quad-rail signals, or to a design utilizing bit-wise completion by modifying the Demultiplexer only. Finally, the current MAC design utilizes combinational logic to determine if rounding, scaling, and saturation are required. However, the datapath could be steered through the rounding, scaling, and saturation logic, if required, through the use of a demultiplexer at the input and a multiplexer at the output, similar to the NCR technique. This alternate approach would reduce the cycle time for operations not requiring rounding, scaling, and saturation, at the expense of an

increase in the cycle time for operations where rounding, scaling, or saturation is required.

LIST OF REFERENCES

[1] Karl M. Fant and Scott A. Brandt, *NULL Convention Logic Systems*, US patent 5,305,463 April 19, 1994.

[2] A. J. Martin, "Programming in VLSI," in *Development in Concurrency and Communication*, Addison-Wesley, pp. 1 – 64, 1990.

[3] K. Van Berkel, "Beware the Isochronic Fork," *Integration, The VLSI Journal,* Vol. 13, No. 2, pp. 103-128, 1992.

[4] C. L. Seitz, "System Timing," in *Introduction to VLSI Systems*, Addison-Wesley, pp. 218-262, 1980.

[5] D. E. Muller, "Asynchronous Logics and Application to Information Processing," in *Switching Theory in Space Technology*, Stanford University Press, pp. 289-297, 1963.

[6] Ilana David, Ran Ginosar, and Michael Yoeli, "An Efficient Implementation of Boolean Functions as Self-Timed Circuits," *IEEE Transactions on Computers*, Vol. 41, No. 1, pp. 2-10,1992.

[7] T. S. Anantharaman, "A Delay Insensitive Regular Expression Recognizer," *IEEE VLSI Technology Bulletin*, Sept. 1986.

[8] N. P. Singh, *A Design Methodology for Self-Timed Systems*, Master's Thesis, MIT/LCS/TR-258, Laboratory for Computer Science, MIT, 1981.

[9] J. Sparso, J. Staunstrup, M. Dantzer-Sorensen, Design of Delay Insensitive Circuits using Multi-Ring Structures. *Proceedings of the European Design Automation Conference*, pp. 15-20, 1992.

[10] A. J. Martin, "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits," *Distributed Computing*, Vol. 1, No. 4, pp. 226-234, 1986.

[11] C. H. (Kees) van Berkel, *Handshake Ciruits: An Intermediary Between Communicating Processes and VLSI*, Ph.D. Thesis, Eindhoven University of Technology, 1992.

[12] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and Tak Kwan Lee, "The Design of an Asynchronous MIPS R3000 Microprocessor," *Proceedings of the 17th Conference on Advanced Research in VLSI*, pp. 164 – 181, 1997.

[13] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The Design of an Asynchronous Microprocessor," *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351-373, 1989.

[14] W. Hardt and B. Kleinjohann, "FLYSIG: Dataflow Oriented Delay-Insensitive Processor for Rapid Prototyping of Signal Processing," *Proceedings of the Ninth International Workshop on Rapid System Prototyping*, pp. 136-141, 1998.

[15] P. K. Tsang, C. C. Cheung, K. H. Leung, T. K. Lee, and P. H. W. Leong, "MSL16A: An Asynchronous Forth Microprocessor," *Proceedings of the IEEE Region 10 Conference*, Vol. 2, pp. 1079 –1082, 1999.

[16] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura, "TITAC: Design of a Quasi-Delay-Insensitive Microprocessor," *IEEE Design & Test of Computers*, Vol. 11, No. 2, pp. 50-63, 1994.

[17] S. H. Unger, *Asynchronous Sequential Switching Circuits*, Wiley, New York, 1969.

[18] S. M. Nowick and D. L. Dill, "Synthesis of Asynchronous State Machines Using a Local Clock," *Proceedings of ICCAD*, pp.192-197, 1991.

[19] Ivan E. Sutherland, "Micropipelines," *Communications of the ACM*, Vol. 32, No. 6, pp. 720-738, 1989.

[20] A. Martin, "The Limitations to Delay-Insensitivity in Asynchronous Circuits," *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*: pp. 263-278, 1990.

[21] Karl M. Fant and Scott A. Brandt, "NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis," *International Conference on Application Specific Systems, Architectures, and Processors*, pp. 261-273, 1996.

[22] T. Verhoff, "Delay-Insensitive Codes – An Overview," *Distributed Computing,* Vol. 3, pp. 1-8, 1988.

[23] Gerald E. Sobelman and Karl M. Fant, "CMOS Circuit Design of Threshold Gates with Hysteresis," *IEEE International Symposium on Circuits and Systems* (II), pp. 61-65, 1998.

[24] T. E. Williams, *Self-Timed Rings and Their Application to Division*, Ph.D. Thesis, CSL-TR-91-482, Department of Electrical Engineering and Computer Science, Stanford University, 1991.

[25] S. M. Burns, *Performance Analysis and Optimization of Asynchronous Circuits*, Ph.D. Thesis, CS-TR-91-1, Caltech, 1991.

[26] S. M. Burns, "General Conditions for the Decomposition of State Holding Elements," *Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 48-57, 1996.

[27] M. L. Dertouzos, *Threshold Logic: A Synthesis Approach*, Cambridge, M. I. T. Press, 1965.

[28] Lewis & Coates, *Threshold Logic*, New York: John Wiley & Sons, Inc., 1967.

[29] C. Sheng, *Threshold Logic*, New York: Ryerson Press, 1969.

[30] A. J. Martin, "Asynchronous Datapaths and the Design of an Asynchronous Adder," *Formal Methods in System Design*, Vol. 1, No. 1, pp. 117-137, 1992.

[31] Paul Day and J. Viv. Woods, "Investigation into Micropipeline Latch Design Styles," *IEEE Transactions on VLSI Systems*, Vol. 3, No. 2, pp. 264-272, 1995.

[32] K. Yun, P. Beerel, and J. Arceo, "High-Performance Asynchronous Pipeline Circuits," *Advanced Research in Asynchronous Circuits and Systems*, pp. 17-28, 1996.

[33] Stephen B. Furber and Paul Day, "Four-Phase Micropipeline Latch Control Circuits," *IEEE Transactions on VLSI Systems*, Vol. 4, No. 2, pp. 247-253, 1996.

[34] J. D. Garside, S. B. Furber, and S. H. Chung, "AMULET3 Revealed," *Proc. Async '99*, pp. 51 – 59, 1999.

[35] N.C. Paver, P. Day, C. Farnsworth, D.L. Jackson, W.A. Lien, J. Liu, "A Low-Power, Low Noise, Configurable Self-Timed DSP," *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 32-42, 1998.

[36] O. Hauck and S. A. Huss, "Asynchronous Wave Pipelines for High Throughput Datapaths," *IEEE International Conference on Electronics, Circuits, and Systems*, Vol. 1, pp. 283 –286, 1998.

[37] Chansub Park and Duckjin Chung, "Modified Asynchronous Wave-Pipelining," *Electronics Letters*, Vol. 36, No. 4, pp. 295 –297, 2000.

[38] Jens Sparso and Jorgen Stanstrup, "Design and Performance Analysis of Delay Insensitive Multi-Ring Structures," *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences*, Vol.1, pp. 349 –358, 1993.

[39] S. Kim and P. A. Beerel, "Pipeline Optimization for Asynchronous Circuits: Complexity Analysis and an Efficient Optimal Algorithm," *IEEE/ACM International Conference on Computer Aided Design*, pp. 296 –302, 2000.

[40] M. Singh and S. M. Nowick, "High-Throughput Asynchronous Pipelines for Fine-Grain Dynamic Datapaths," *Proceeding of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 198 –209, 2000.

[41] C. D. Nielsen and A.J. Martin, "Design of a Delay-Insensitive Multiply and Accumulate Unit," *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences*, Vol. 1, pp. 379 –388, 1993.

[42] T. Tang, C. Choy, P. Siu, and C. Chan, "Design of Self-Timed Asynchronous Booth's Multiplier," *Proceedings of the ASP-DAC Design Automation Conference*, pp. 15-16, 2000.

[43] V. A. Bartlett and E. Grass, "A Low-Power Concurrent Multiplier-Accumulator Using Conditional Evaluation," *The 6th IEEE International Conference on Proceedings of ICECS*, Vol. 2, pp. 629 - 633, 1999.

[44] Behrooz Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, Oxford University Press, New York, 2000.

[45] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, McGraw-Hill Book Company, New York, 1995.

[46] A. J. Martin, "Synthesis of Asynchronous VLSI Circuits," *Formal Methods for VLSI Design*, pp. 237-283, 1990.

[47] G. E. Sobelman and D. Raatz, "Low-power Multiplier Design using Delayed Evaluation," *Proceedings of the International Symposium on Circuits and Systems*, pp. 1564-1567, 1995.

[48] U. Cummings, A. Lines, and A. Martin, "An Asynchronous Pipelined Lattice Structure Filter," *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 126-133, 1994.

DATE DUE

MAR 26 2003

NOV 14 2005

Printed in USA