FORMAL MODELING AND VERIFICATION METHODOLOGIES FOR QUASI-DELAY INSENSITIVE ASYNCHRONOUS CIRCUITS

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Ashiq Adnan Sakib

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Electrical and Computer Engineering

June 2019

Fargo, North Dakota

North Dakota State University
Graduate School

Title

FORMAL MODELING AND VERIFICATION METHODOLOGIES FOR QUASI-DELAY INSENSITIVE ASYNCHRONOUS CIRCUITS

By

Ashiq Adnan Sakib

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Scott C. Smith
Chair

Dr. Sudarshan K. Srinivasan
Co-Chair

Dr. Jacob Glower

Dr, Rhonda Magel

Approved:

06/19/2019	Dr. Benjamin Braaten
Date	Department Chair

ABSTRACT

Pre-Charge Half Buffers (PCHB) and NULL convention Logic (NCL) are two major commercially successful Quasi-Delay Insensitive (QDI) asynchronous paradigms, which are known for their low-power performance and inherent robustness. In industry, QDI circuits are synthesized from their synchronous counterparts using custom synthesis tools. Validation of the synthesized QDI implementation is a critical design prerequisite before fabrication. At present, validation schemes are mostly extensive simulation based that are good enough to detect shallow bugs, but may fail to detect corner-case bugs. Hence, development of formal verification methods for QDI circuits have been long desired. The very few formal verification methods that exist in the related field have major limiting factors. This dissertation presents different formal verification methodologies applicable to PCHB and NCL circuits, and aims at addressing the limitations of previous verification approaches. The developed methodologies can guarantee both safety (full functional correctness) and liveness (absence of deadlock), and are demonstrated using several increasingly larger sequential and combinational PCHB and NCL circuits, along with various ISCAS benchmarks.

ACKNOWLEDGEMENTS

I would like to thank first and foremost my advisor, Dr. Scott C. Smith, for providing me with an opportunity to pursue my Ph.D. at NDSU under his supervision. I would like to thank my co-advisor, Dr. Sudarshan K. Srinivasan, for sharing his knowledge on formal verification and making me competent in an area of studies in which I had no prior experience. I am very thankful for Drs. Smith's and Srinivasan's technical guidance, support, advice, motivations, and recommendations over the last five years, which made my research experience enjoyable. I would like to thank my committee members, Dr. Jacob Glower and Dr. Rhonda Magel, for their reviews and valuable inputs on the dissertation. I would like to thank the National Science Foundation (NSF) (Grant No. CCF-1717420) and the department of Electrical and Computer Engineering at NDSU for funding my doctoral research. I would also like to thank ECE department's administrative and technical staffs: Laura Dallmann, Jeffrey Erickson, and Eric Christianson, for their continued support and assists. Last but not the least I would like to thank my parents for all the sacrifices they made to get me where I am today, and my wife, Mousam, for being the emotional support during all my difficulties.

DEDICATION

To my parents, Mr. Md. Safiqul Islam Bhuiyan and Ms. Tazgira Zaman, my brother Abir Akib, and my loving wife Mousam Hossain.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

DEDICATION v

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ABBREVIATIONS xii

LIST OF SYMBOLS xiv

1. INTRODUCTION 1

  1.1. Background 1

  1.2. Motivation 3

  1.3. Research Challenges 4

  1.4. Dissertation Overview 5

2. OVERVIEW OF ASYNCHRONOUS DESIGN METHODS 6

  2.1. QDI Paradigms: An Overview 6

  2.2. QDI Background 8

  2.3. QDI Paradigms 10

    2.3.1. Pre-Charge Half Buffers (PCHB) 11

      2.3.1.1. Semi-static representation of PCHB circuits 11

      2.3.1.2. Static representation of PCHB circuits 15

    2.3.2. NULL Convention Logic (NCL) 18

      2.3.2.1. NCL combinational unit 21

      2.3.2.2. NCL registration unit 22

      2.3.2.3. NCL completion unit 24

  2.4. Related Verification Works 24

3. FORMAL MODELING AND VERIFICATION METHODS FOR QDI PCHB ASYNCHRONOUS CIRCUITS 29

3.1. Model Checking Based Verification Method 29

3.1.1. PCHB Formal Gate Model 29

3.1.2. Proposed Methodology for Verification 31

3.1.2.1. Input properties 32

3.1.2.2. Rack and Lack properties 33

3.1.2.3. Output properties 33

3.1.2.4. N-stage pipeline architecture and the global property 34

3.1.3. Results 36

3.1.4. Discussions 36

3.2. An Equivalence Verification Methodology for Combinational and Sequential QDI PCHB Asynchronous Circuits 37

3.2.1. Enumeration of All Possible PCHB Faults 37

3.2.2. Equivalence Verification of Combinational PCHB Circuits 47

3.2.2.1. Safety check of combinational PCHB circuits 47

3.2.2.2. Liveness and handshaking correctness check 51

3.2.2.3. Results 54

3.2.3. Equivalence Verification of Sequential PCHB Circuits 56

3.2.3.1. Safety check for sequential PCHB circuits 57

3.2.3.2. Liveness and handshaking correctness check 66

3.2.4. Detection of All Possible Faults 67

3.2.5. Results 68

3.3. Conclusions 70

4. FORMAL MODELING AND VERIFICATION METHODS FOR QDI NCL ASYNCHRONOUS CIRCUITS 72

4.1. Equivalence Verification for Combinational NCL Circuits 73

4.1.1. Functional Equivalence Check 75

4.1.2. Invariant Check 80

4.1.3. Handshaking Check 82

4.1.4. Combinational NCL Circuits Results 85

4.2. Equivalence Verification for Sequential NCL Circuits 86

4.2.1. Safety Check 86

4.2.2. Liveness Check 93

4.2.3. Sequential NCL Circuit Results 95

4.3. Conclusions 96

5. CONCLUSIONS 97

5.1. Summary 97

5.2. Future Works 99

REFERENCES 101

APPENDIX. LIST OF PUBLICATIONS 107

LIST OF TABLES

Table Page

1. Dual-Rail Signal Representation. 9
2. Quad-Rail Signal Representation. 9
3. Test Circuit Verification Times for Model Checking Approach. 36
4. Verification Results for Various Combinational PCHB Circuits Based on Equivalence Checking. 55
5. Verification Results for Various Sequential PCHB Circuits Based on Equivalence Checking. 70
6. Verification Results for Various C/L NCL Circuits. 86
7. Verification Results for Sequential NCL Circuits. 96

LIST OF FIGURES

Figure Page

1. C-element Functionality.......... 7

2. Four-phase Handshaking Protocol.......... 10

3. Semi-static PCHB Function Template [21].......... 12

4. Semi-static Implementation of PCHB NAND2 Circuit [23].......... 13

5. A Reset-to-NULL PCHB NAND2 Circuit.......... 14

6. Static Implementation of PCHB Circuits [23].......... 16

7. Set Functions for Each Rail of *Sum* and *Carry* of a PCHB Full Adder.......... 16

8. Static Implementation of PCHB Full Adder *Sum* Circuit [23].......... 17

9. Static Implementation of PCHB Full Adder *Carry* Circuit [23].......... 17

10. Generation of *Sleep* Signal for the PCHB Full Adder.......... 18

11. NCL Architecture Framework.......... 19

12. DATA Flow through Combinational Logic.......... 20

13. *rfn* Flow through Completion Logic.......... 20

14. NULL Flow through Combinational Logic.......... 20

15. *rfd* Flow through Completion Logic.......... 21

16. TH23 Gate as an Example of Non-weighted Threshold Gate.......... 21

17. TH24W22 Gate as an Example of Weighted Threshold Gate.......... 22

18. NCL Single Bit Reset-to-NULL Register.......... 23

19. (a) Reset-to-DATA0 NCL Register. (b) Reset-to-DATA1 NCL Register.......... 23

20. 16-bit NCL Completion Unit.......... 24

21. State Diagram of a PCHB *n*-input NAND Function.......... 31

22. PCHB 2x2 Multiplier Circuit.......... 48

23. Netlist Structure of PCHB 2x2 Multiplier Circuit.......... 49

24. Converted Boolean Netlist Structure. 50

25. *Fan_out* and *Comp_fanin* Structure. 53

26. Synchronous MAC Structure. 57

27. PCHB MAC Structure. 57

28. PCHB 4 + 2x2 MAC Datapath Connection. 58

29. PCHB 4 + 2x2 MAC Handshaking Connection. 59

30. PCHB 4 + 2x2 MAC Unit Netlist. 60

31. Equivalent Boolean Converted Netlist. 60

32. Equivalent Synchronous Circuit after Conversion. 63

33. Alternate Arrangement in FL3 and FL4 to Demonstrate Deadlock. 64

34. Formulation of Proof Obligation to Check Equivalence of PCHB_SEQ and SPEC_SEQ. 66

35. 3×3 NCL Multiplier Circuit. 75

36. (a) Input Incomplete NCL AND. (b) Input Complete NCL AND. 76

37. NCL Half Adder Circuit. 76

38. NCL Full Adder Circuit. 76

39. (a) 3x3 NCL Multiplier Netlist. (b) Converted Boolean Netlist. 78

40. Handshaking Connection for 3x3 NCL Multiplier. 83

41. *Reg_fanin* and *Ko_fanout* Lists for the 3x3 NCL Multiplier. 85

42. NCL MAC Unit. 88

43. 4 + 2x2 NCL MAC Datapath. 89

44. (a) 4 + 2x2 NCL MAC Netlist. (b) Converted Synchronous Equivalent Netlist. 90

45. Proof Obligation to Check Equivalence of *NCL_SYNC* and *SPEC_SYNC* Netlists. 92

46. Handshaking Connections for the 4 + 2x2 NCL MAC. 94

47. *Reg_fanin* and *Ko_fanout* Lists for the 4 + 2x2 NCL MAC. 95

LIST OF ABBREVIATIONS

EMI	Electro-magnetic Interference.
PVT	Process, Voltage, Temperature.
ITRS	International Technology Roadmap for Semiconductors.
ASIC	Application Specific Integrated Circuits.
DI	Delay Insensitive.
QDI	Quasi Delay Insensitive.
CAD	Computer Automated Design.
PCHB	Pre-Charge Half Buffer.
NCL	NULL Convention Logic.
DIMS	Delay Insensitive Min-Term Synthesis.
SOP	Sum of Products.
FPGAs	Field Programmable Gate Arrays.
SOC	System on Chip.
WSNs	Wireless Sensor Networks.
SCL	Sleep Convention Logic.
MTNCL	Multi-Threshold NULL Convention Logic.
CMOS	Complementary Metal Oxide Semiconductor.
LCD	Left Completion Detection.
RCD	Right Completion Detection.
Rfd	Request for Data.
Rfn	Request for NULL.
STG	Signal Transition Graph.
WEB	Well-Founded Equivalence Bisimulation.

TS	Transition System.
TSs	Transition Systems.
CTL	Computational Tree Logic.
AG	Always Global.
MAC	Multiply and Accumulate.
SMT	Satisfiability Modulo Theory.
SMT-Lib	Satisfiability Modulo Theory Library.
MSB	Most Significant Bit.
LSB	Least Significant Bit.
HA	Half Adder.
FA	Full Adder.

LIST OF SYMBOLS

$\forall$	For All.
$\wedge$	Logical AND Operation.
$\vee$	Logical OR Operation.
$\neg$	Logical NOT Operation.
$\oplus$	Logical XOR Operation.
$\rightarrow$	Implies.
$\in$	Belongs to.
$\notin$	Does Not Belong to.
$\neg$	Does Not Belong to.
U	Until.
$\subset$	Proper Subset.

1. INTRODUCTION

1.1. Background

Over the last few decades, the synchronous domain has evolved enormously with the industry demands in terms of technology scaling, high frequency operations, and attaining multi-functionality. Extensive research works over a period of time have resulted in the development of advanced support tools and automated design processes to significantly improve the productivity of designers in the related field. As a result, today's digital design industry is dominated by the clocked approach. Although the synchronous design has led to dramatic progress, it has hit major limitations. As the operating frequency gets into Giga-Hertz (GHz) region, clock management becomes a fundamental design challenge, and surfaces several clock related issues, such as, clock skew, clock jitter, timing closure, clock distribution, etc. The foundation of synchronous design is based on the clock as reference, where all subcomponents observe the clock propagation almost at the same time with certain reasonable approximations. Previously, concepts, like wire delays, were not taken into design consideration for mid-frequency operations, as wire delays were significantly smaller as compared to clock period. However, at GHz level the wire delays add to the clock period significantly to an extent that certain component(s) may get out of 'sync' because of reiteration, resulting in a malfunction [1]. Therefore, in recent synchronous design trends, addressing the timing violations have turned out to be designers' nightmare, even for a single chip design. An additional circuitry, called the clock driver, is used to manage the clock skew in high performance digital designs, which is responsible for attaining acceptable skew during circuit operation. The clock driver is a complex design and results in significant area overhead. However, there is a limit on how much area overhead is acceptable, as a larger chip area further increases the interconnect delays, and with faster clocks the skew may even get worse if not designed with precision.

Power dissipation is another paramount design challenge in the synchronous domain. High speed clocks result in increased switching activities in gates, deteriorating the power performance. For example, the largest arrangement of gates is the clock driver unit, which is always on and constantly switching even if no other parts have any task to perform. Moreover, higher level of integration in nanoscale devices adds significantly to the power dissipation. The leakage power consumption that was previously ignored for being too negligible, turns out to be a huge source of power loss in nanoscale designs. Process variability becomes another contributor to design challenges for designs with small feature size, resulting in design compromises, such as, stretching of timing margins in static timing analysis [2].

Asynchronous design, on the other hand, is a clockless approach. The absence of clock results in significantly less power consumption, noise, and Electro-magnetic Interference (EMI). Additionally, it also eradicates the clock skew, clock jitter, glitches, and wire delay issues. The synchronization and communication between the components are established using a handshaking mechanism. Unlike synchronous design, the system only switches when some tasks are to be performed, providing significantly improved power performance. Asynchronous structures are robust against process, voltage and temperature (PVT) variations, which enables desired tradeoff between power performance, voltage supply, and speed of operation depending on the design requirement. Furthermore, the tolerance against PVT variations make this domain an excellent choice for operations in extreme environmental conditions.

Because of the inherent advantages over synchronous domain, asynchronous design has gained popularity in the industry over the last two decades as evidenced by the International

Technology Roadmap for Semiconductors (ITRS). According to the ITRS, asynchronous circuits currently comprise around 32% of Integrated Circuit (IC) logic, compared to 20% in 2013, and estimates that they will account for over 50% of the multi-billion dollar semiconductor industry by 2027 [3].

Asynchronous circuits are very complex designs. Dual rail encoding, hysteresis, registration, and handshaking circuits add to the design complexity and increase the area overhead. However, the area overhead may be overlooked considering the increased robustness and ultra-low power applications. In addition, asynchronous paradigm is relatively new to the mainstream semiconductor industry, and therefore lacks advanced support and verification tools. Although there have been several tools that can automate the synthesis of asynchronous circuits form synchronous specification [4, 5, 6, 7, 8], developing verification methodology for different asynchronous models still remains an open challenge.

1.2. Motivation

Formal modeling and verification for design validation is a critical component of any commercial ASIC design flow. In formal methods, the correctness of operation is established using mathematical proofs. This allows formal methods to detect and flag corner-case bugs, as proof can correspond to a large set of possible test cases. Extensive simulation based testing schemes have been predominantly what has been used in the semiconductor industry before the infamous Intel FDIV bug incident. In 1994, Intel had to incur a loss of about $500 million, when a bug in the floating point unit of their premium processor went unnoticed through extensive testing, and only got detected after deployment. Since then, the semiconductor industry has aggressively incorporated formal verification into its design cycle for validation. Presently, testing and formal verification are implemented independently and complement each other to ensure complete functional correctness.

Although asynchronous design has managed to establish a growing interest in the industry and found numerous applications, the domain is still relatively new. The scopes of formal verification in asynchronous designs are not widely explored. Design validation is mostly simulation-based, which cannot guarantee full functional correctness. Therefore, formal verification methods applicable to different asynchronous models have been long desired in the industry. The work presented in this dissertation is an effort to address this issue to aid the widespread implementation of the clockless approach.

The verification methods developed and illustrated in this dissertation are applicable to Quasi-Delay Insensitive (QDI) asynchronous circuits, which is one of the two commercially successful asynchronous design paradigms. There exists very few verification methods for QDI circuits; and existing methods have several drawbacks. The goal of the work is to develop unified, fast, and scalable verification methods for QDI circuits by addressing the issues and overcoming the limitations of already existing schemes.

1.3. Research Challenges

In industry, QDI circuits are often synthesized from their synchronous counterparts utilizing Computer Automated Design (CAD) tools that cause the specification to undergo numerous transformations and optimizations, resulting in an asynchronous implementation. Any error in the synthesis tools will eventually result in an implementation error and is required to be detected. However, this task is not easy because of the huge structural dissimilarities between synchronous specification and asynchronous implementation. Traditional synchronous designs are deterministic in nature. Because of a reference clock, all the states in a design change concurrently. Hence, the verification turns out to be a relatively straightforward process of inspecting the delays of combinational units in between registers. Whereas, QDI asynchronous paradigms are more non-deterministic (e.g., inputs can start propagating through the circuit at any time and in any order, unlike synchronous circuits where all inputs start propagating through the circuit at the same time at a predetermined clock edge), which makes them very difficult to verify. Multi rail logic, arbitrary availability of data, hysteresis in combinational units, and intricate handshaking control mechanism add further to the verification complexity.

1.4. Dissertation Overview

This dissertation is divided into five main chapters. A detailed overview of QDI background, major QDI paradigms: Pre-Charge Half Buffers (PCHB) and NULL Convention Logic (NCL), and review of related verification works are presented in Chapter 2. Developed formal modeling and verification methodologies based on model checking and equivalence checking for QDI PCHB circuits are discussed in Chapter 3. Chapter 4 presents the developed verification methodology for QDI NCL circuits, followed by conclusions and directions for future work in Chapter 5.

2. OVERVIEW OF ASYNCHRONOUS DESIGN METHODS

Asynchronous circuits can be grouped into two major categories: *bounded-delay* model and *delay-insensitive* model. The key difference between the two categories are on their assumption on delays in gates and wires. Bounded-delay models are based on the assumption that the delay in gates and wires are bounded. Therefore, delays are added to avoid hazard conditions based on the worst-case scenarios, which results in an extensive timing analysis to guarantee correct functional operation of the circuit. *Micropipelines* [9], *Huffman circuits* [10], and *burst-mode circuits* [11] are some of the popular circuits based on bounded-delay model of asynchronous design. On the other hand, Delay-Insensitive (DI) circuits do not assume the delays on interconnects and gates. However, no practical circuits are possible due to lack of expressible conditionals in DI circuits [12]. Hence, Quasi-Delay Insensitive (QDI) circuits are used for practical use, which is based on an assumption that the wire forks within basic components, such as full adder and half adder, are isochronic; i.e., the wire delays within components are less than the logic element delays within the components. Interconnects between components are not required to follow the isochronic fork assumption. All the practical Delay-Insensitive circuits are essentially Quasi-Delay Insensitive. QDI model has two major advantages over bounded-delay model: 1) it requires very little timing analysis to ensure correct functionality, and 2) yields average case performance; whereas, bounded-delay model as well as synchronous paradigms yield worse case performance. As this research focuses on QDI asynchronous model, bounded-delay models are not addressed further.

2.1. QDI Paradigms: An Overview

Seitz's [13], Anantharaman's [14], Singh's [15], Delay Insensitive Min-term Synthesis (DIMS) [16], and David's [17] methods are popular designs based on gate-level delay insensitivity. All these methods utilize *C-elements* [18] along with Boolean gates to attain delay insensitivity. For an *n*-input C element, when all of the *n*-inputs assume a particular value then the output assumes the same value, otherwise the output remains unchanged. Fig. 1 shows the structure and behavior of a 2-input *C-element*. The output, C, becomes '1'/ '0' when A=B= '1'/ A= B= '0', otherwise it holds the previous value.

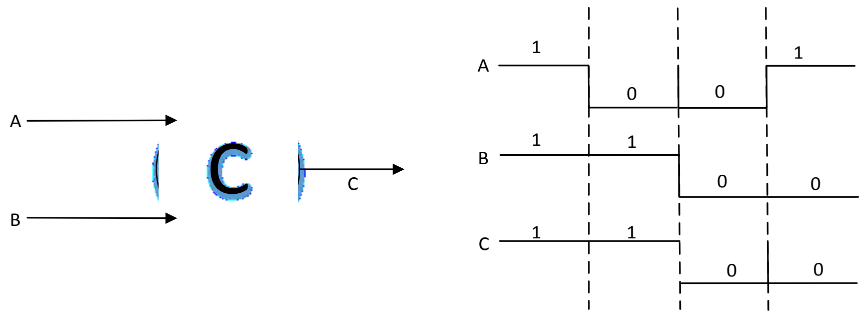


Figure 1. C-element Functionality.

In Seitz's method, both rails of individual inputs go through an OR gate. The function is implemented by generating all the minterms in sum-of-product (SOP) form; which is further combined with the OR network using a C-element, to generate the final outputs. Anantharaman's and DIMS methods also require the generation of all minterms; whereas, Singh's and David's methods do not require the generation of all minterms. Delay insensitive functionality is attained by combining smaller self-timed logic components in Singh's method. In David's method, four subnets, *ORN*, *CEN*, *DRN*, and *OUTN*, are utilized to construct self-timed logics. *ORN* is an OR network, outputs of which are inputs to n-input C element, *CEN*. *OUTN* is the network that produces the circuit outputs. *OUTN* consists of 2m 2-input C-elements, where 'm' is the number of outputs. One of the 2 inputs of the C-elements is the output of *CEN*, while the other input is the output of *DRN*, which is an implementation of individual rail of dual-rail outputs.

NULL Convention Logic (NCL) [19] circuits do not rely on C-elements only, instead this paradigm has a library of 27 threshold gates with state-holding capacity, which can implement any function with maximum four variables. This results in more design flexibility and scopes of optimization in NCL design flow. Phased Logic [20] is another delay-insensitive method that does not require the generation of full minterms set. The method transforms synchronous circuits to delay-insensitive versions, and was developed to ease the timing constraints. Pre-Charge Half Buffers (PCHB) [21] circuits are based on dynamic logic, and are synthesized at transistor level, unlike the above mentioned methods that are synthesized at the gate level. Common PCHB gates include typical Boolean functions and components (e.g., AND, NAND, OR, NOR, XOR, FA, HA, etc.) that are utilized in combination with C-elements to attain circuit functionality. Like NCL, PCHB also provides design flexibility to certain extent.

2.2. QDI Background

QDI circuits have three distinguishing features that sets them apart from their synchronous counterparts, which are discussed below:

1. Utilization of Multi-rail Logic: In synchronous circuits, data is encoded using binary with each bit represented by a single wire, while QDI uses multi-rail logic to represent a bit of data. This is because QDI uses symbolic completeness of expression [19] to achieve self-timed behavior, unlike traditional Boolean logic that are symbolic incomplete. A symbolic complete expression does not take the time of evaluation into consideration, rather depends on the relationships of the symbols present in the expression. In order to attain the symbolic completeness, QDI uses dual-rail or quad-rail logic. In case of dual-rail logic, a dual-rail signal D, is represented by two wires, D^0 and D^1. D can be either *DATA0, DATA1* or *NULL*, as can be seen from Table 1. *DATA0* and *DATA1* corresponds to Boolean logic '0' and '1', respectively. Both rails being de-asserted corresponds to a *NULL* state, which represents the unavailability of DATA. D^0 and D^1 are mutually exclusive, i.e., D^0 and D^1 cannot be asserted simultaneously.

Table 1. Dual-Rail Signal Representation.

Value	***D^1***	***D^0***	**Boolean Equivalent**
NULL	0	0	—
DATA0	0	1	Logic 0
DATA1	1	0	Logic 1
Illegal	1	1	XX

In case of quad-rail representation, a quad rail signal, *Q,* is represented using four wires, Q^0, Q^1, Q^2, and Q^3. *Q* can be either *DATA0*, *DATA1*, *DATA2*, *DATA 3*, or *NULL*, as can be seen from Table 2. Each DATA signal corresponds to two Boolean logic signals, *X* and *Y*. *DATA0*, *DATA1*, *DATA2*, and *DATA3* state correspond to ($X = 0$ and $Y = 0$), ($X = 0$ and $Y = 1$), ($X = 1$ and $Y = 0$), and ($X = 1$ and $Y = 1$), respectively. All four rails being de-asserted corresponds to a *NULL* state, which represents the unavailability of DATA. Q^0, Q^1, Q^2, and Q^3 are mutually exclusive, i.e., any 2 wires cannot be asserted simultaneously.

Table 2. Quad-Rail Signal Representation.

Value	***Q^3***	***Q^2***	***Q^1***	***Q^0***	**Boolean Equivalent**
NULL	0	0	0	0	____
DATA0	0	0	0	1	Logic 00
DATA1	0	0	1	0	Logic 01
DATA2	0	1	0	0	Logic 10
DATA3	1	0	0	0	Logic 11

2. Incorporates registration: QDI circuits have functional units/ combinational units as well as registration units. The registration and computation units are either separated (in case of NCL) or integrated together (in case of PCHB). The registration units for both NCL and PCHB are discussed in details in the later subsections.

3. Handshaking Protocol: QDI circuits do not have any clock as a reference. Hence, to synchronize and communicate between components, QDI circuits incorporate a very well-defined four-phased handshaking protocol utilizing the previously mentioned multi-rail data along with a *handshaking* signal. Completion detection units are used to detect whether the multi-rail signals are DATA or NULL. Fig. 2 shows the high level representation of the four-phase handshaking protocol. In Phase 1, the data channel is in the NULL state, and the receiver requests data by asserting the handshaking signal. In Phase 2, data is sent by the sender after receiving the handshaking request by setting the data channel to DATA. In Phase 3, the receiver gets the data and acknowledges this by deasserting the handshaking signal. In Phase 4, the sender resets the data channel back to NULL after receiving the handshaking acknowledgement. After the receiver sees NULL on the data channel, it can request the next data by asserting the handshaking signal, which is Phase 1 again. The implementation of the handshaking mechanism for different QDI paradigms is discussed in details afterwards.

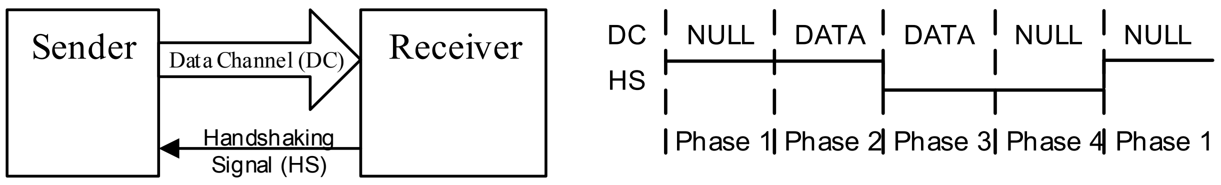


Figure 2. Four-phase Handshaking Protocol.

2.3. QDI Paradigms

NCL and PCHB are two major QDI design paradigms. Both paradigms are commercially successful and have been utilized in a number of applications by several semiconductor industries, including Intel, Achronix, Phillips, Eta-compute, Camgian/Theseus Logic, etc. Intel uses PCHB technology to produce fast Ethernet switch chips, whereas Achronix develops high-speed-low-power asynchronous Field Programmable Gate Arrays (FPGAs) based on synchronous interface with PCHB asynchronous core [22]. Camgian/Theseus Logic develops highly integrated mixed signal systems-on-chip (SOC) for wireless sensor nodes (WSNs), based on NCL [22].

2.3.1. Pre-Charge Half Buffers (PCHB)

As a QDI paradigm, PCHB circuits utilize multi-rail logic (most commonly dual-rail logic), incorporates registration units that are integrated with the functional unit, and includes a very well-defined handshaking scheme based on the four-phased mechanism, as discussed in section 2.2. Because of the integrated registration units PCHB circuits have state-holding functionality, known as *hysteresis*. *Dynamic*, *semi-static*, and *static* representation of PCHB circuits can be used to attain hysteresis. The *dynamic* approach is fastest, but not delay-insensitive, because this approach relies on output capacitance to hold the previous state. It does not utilize a feedback, which results in loss of charge in the capacitor (i.e., the previous state) after a period of time. Therefore, the dynamic representation is not popular. *Semi-static* approach utilizes a weak-feedback inverter arrangement to hold previous state. Although the transistors need to be carefully sized to balance current, *semi-static* approach is the most popular PCHB design method. *Static* approach incorporates additional pull-up and pull-down network at the output for state-holding, making it more robust and impervious to process variation and supply voltage, but with an area overhead. The semi-static and static representation of PCHB circuits are discussed in this chapter. Dynamic approach is not discussed further because of its limited applications.

2.3.1.1. Semi-static representation of PCHB circuits

PCHB circuits are synthesized at the transistor level utilizing pre-charge domino style CMOS logic. Fig. 3 shows a semi-static template for a two-input PCHB gate. Inputs *(X^0, X^1)*, *(Y^0,*

Y^1), and (Z^0, Z^1) are the two rails of dual-rail inputs, *X* and *Y*, and dual-rail output, *Z*, respectively. Every PCHB gates incorporate registration, i.e., the gate in addition to performing a logic function such as NAND, also stores the dual-rail output value, and therefore acts like a latch. The *F* block implements the specific logic function of the gate and the weak feedback inverters provide for latching. Control is accomplished using a handshaking protocol using Request (*Rack*) and Acknowledge (*Lack*) signals. *LCD* and *RCD* are the left and right completion detection units used to detect when the inputs and outputs respectively are DATA or NULL. The outputs of *LCD* and *RCD* go through a C-element to produce the *Lack* signal. When both inputs are NULL and the output is also NULL, then *Lack* will be asserted, requesting-for-data (*rfd*). Similarly, when both inputs are DATA and the output is also DATA, *Lack* will be deasserted, requesting-for-NULL (*rfn*). The function evaluates and the output becomes DATA whenever both *Lack* and *Rack* are *rfd* and the *X* and *Y* inputs are DATA. Note that the output can become DATA before both inputs are DATA, depending on *F*. An example is the NAND gate function, where the output is logic 1 when either input is logic 0.

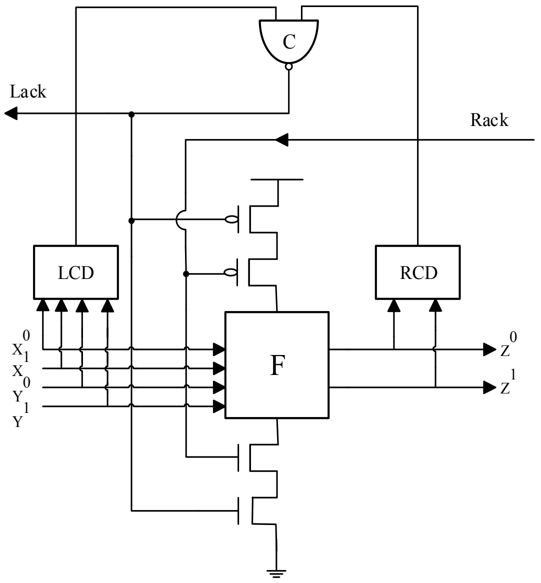


Figure 3. Semi-static PCHB Function Template [21].

Based on the template in Fig. 3, Fig. 4 shows a 2 input PCHB NAND gate with dual-rail inputs and outputs, *X* and *Y*, and *F*, respectively. The *Set* functions are implemented to achieve NAND2 functionality (*Set* $F^0 = X^1$ *AND* Y^1; *Set* $F^1 = X^0$ *OR* Y^0). The two 2-input NORs connected to the rails of dual rail inputs correspond to the *LCD*, while the 2-input NOR connected to the output signal rails correspond to the *RCD* in Fig. 3. The TH33 threshold gate structure used as a C-element will be discussed in Section 2.3.2.1. The weak inverter arrangement is used to hold the output DATA until pre-charged back to NULL to attain delay-insensitivity.

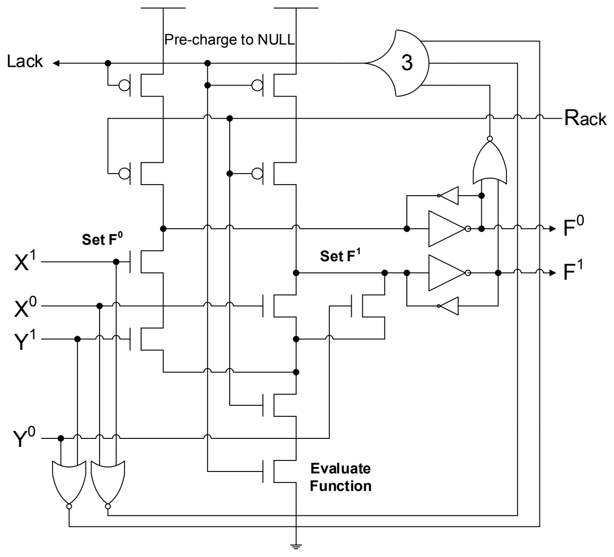


Figure 4. Semi-static Implementation of PCHB NAND2 Circuit [23].

The mechanism of DATA and NULL transitions for the NAND2 gate with the help of *Lack* and *Rack* signals is discussed below:

When *Lack* is logic 1 (*rfd*), the inputs will eventually become DATA; and when *Lack* is logic 0 (*rfn*), the inputs will eventually become NULL. The NAND function evaluates, and the output becomes DATA whenever both *Lack* and *Rack* are *rfd* and the *X* and *Y* inputs are DATA. If *Rack* is *rfd* and *Lack* is *rfn*, or vice versa, the state is held by the weak inverters. When *Lack* and *Rack* are both *rfn*, the output is pre-charged back to NULL. Whenever the inputs and outputs are all DATA, *Lack* changes to *rfn*; and when the inputs and output are all NULL, *Lack* changes to *rfd*.

PCHB gates can also include a *reset* input to initialize the gate's data output to NULL, DATA0, or DATA1, and the gate's *Lack* output to the appropriate value based on the data output's reset value (i.e., logic 1 if reset to NULL, or logic 0 if reset to DATA). A reset-to-NULL version of the NAND 2 gate (Fig. 4) is shown in Fig. 5.

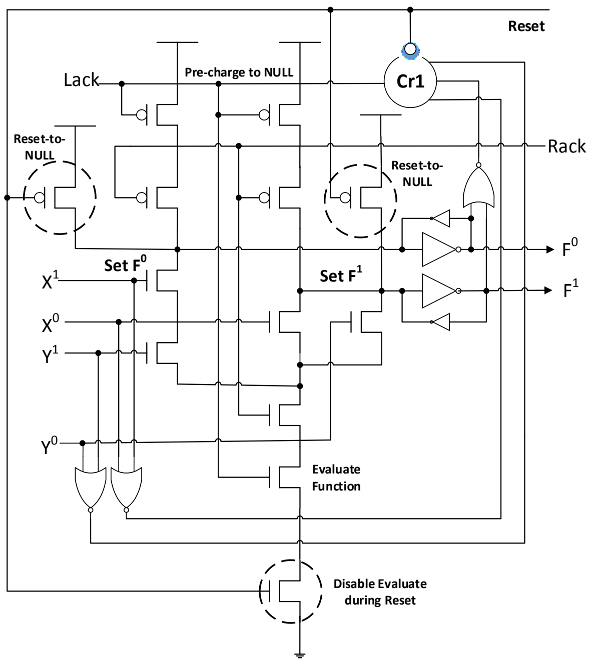


Figure 5. A Reset-to-NULL PCHB NAND2 Circuit.

Resettable version of PCHB gates can be used to utilize the gate as register/latch as well, resulting in design optimization. For example, sequential PCHB circuits require at least *2N+1* registers/latches in any feedback loop with *N* DATA tokens to avoid deadlock [24]. As PCHB gates themselves behave as latches, additional registers are not necessary if a feedback loop already contains enough PCHB gates. For example, if there are 3 or more PCHB gates in a feedback loop with 1 DATA token, then no additional registers are required. Additionally, a feedback loop can never be all NULL (N) or all DATA (D). For example, in a feedback loop with 3 registers and 1 DATA token, (NNN) and (DDD) are illegal states, whereas any other of the 6 combinations are valid. PCHB registers operate the same as regular PCHB gates, described above, where the function is output = input.

Handshaking logic between PCHB gates can be implemented using either full-word or bit-wise completion [25], or some combination of the two. Full-word completion requires that the *Lack* signal of each PCHB gate in level$_i$ be conjoined by one or more C-elements to produce a single *Lack* signal, whose output is connected to the *Rack* signal of each PCHB gate in level$_{i-1}$, where a gate's level is the longest path (in terms of number of PCHB gates) from the circuit's primary inputs to that gate's output. On the other hand, bit-wise completion only sends the completion signal from PCHB gate *b* back to each PCHB gate whose output is an input to gate *b*.

2.3.1.2. Static representation of PCHB circuits

The static representation template of any PCHB component is shown in Fig. 6. The circuit is comprised of *set* blocks and *hold0* blocks for each rail of dual-rail outputs, and a *sleep* signal. *Set* block computes the function of each output rail, while the *hold0* block is the complement of the *set* block. *Sleep* is generated by combining *Rack* and *Lack* through a C-element and an inverter. *Sleep* gets asserted when *Lack* and *Rack* are both *rfn*, and the outputs become NULL. *Sleep* is deasserted when Lack and *Rack* are both *rfd.* Under this scenario, the output evaluates whenever the inputs become available. The output will again go back to NULL only when *sleep* is re-asserted. The output should hold its DATA value if *sleep* remains deasserted.

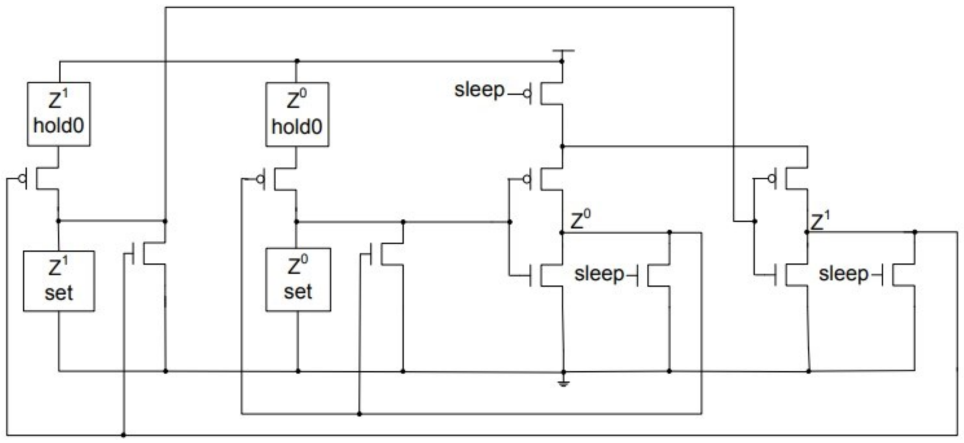


Figure 6. Static Implementation of PCHB Circuits [23].

To construct a static implementation of any PCHB component, e.g. a PCHB full adder, the *set* and *hold0* blocks must be determined. From Fig. 7, we can observe that the set block functions for *sum*, S, are: $S^1 = X^1Y^1Cin^1 + X^1Y^0Cin^0 + X^0Y^1Cin^0 + X^0Y^0Cin^1$, and $S^0 = X^0Y^0Cin^0 + X^0Y^1Cin^1 + X^1Y^0Cin^1 + X^1Y^1Cin^0$; set block function for *carry*, Co, are: $Co^1 = X^1Y^1 + X^1Cin^1 + Y^1Cin^1$, and $Co^0 = X^0Y^0 + X^0Cin^0 + Y^0Cin^0$. The *hold0* blocks for *sum* and *carry* are: $S^{1}{}' = X^{1}{}'Y^{1}{}'Cin^{1}{}' + X^{1}{}'Y^{0}{}'Cin^{0}{}' + X^{0}{}'Y^{1}{}'Cin^{0}{}' + X^{0}{}'Y^{0}{}'Cin^{1}{}'$; $S^{0}{}' = X^{0}{}'Y^{0}{}'Cin^{0}{}' + X^{0}{}'Y^{1}{}'Cin^{1}{}' + X^{1}{}'Y^{0}{}'Cin^{1}{}' + X^{1}{}'Y^{1}{}'Cin^{0}{}'$; $Co^{1}{}' = X^{1}{}'Y^{1}{}' + X^{1}{}'Cin^{1}{}' + Y^{1}{}'Cin^{1}{}'$; $Co^{0}{}' = X^{0}{}'Y^{0}{}' + X^{0}{}'Cin^{0}{}' + Y^{0}{}'Cin^{0}{}'$.

X \ YCin	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$S^1 = X^1Y^1Cin^1 + X^1Y^0Cin^0 + X^0Y^1Cin^0 + X^0Y^0Cin^1$

$S^0 = X^0Y^0Cin^0 + X^0Y^1Cin^1 + X^1Y^0Cin^1 + X^1Y^1Cin^0$

X \ YCin	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$Co^1 = X^1Y^1 + X^1Cin^1 + Y^1Cin^1$

$Co^0 = X^0Y^0 + X^0Cin^0 + Y^0Cin^0$

Figure 7. Set Functions for Each Rail of *Sum* and *Carry* of a PCHB Full Adder.

Fig. 8 and Fig. 9 shows the static PCHB full adder *sum* and *carry* implementations, respectively. Some transistors are shared for area optimization. Generation of the *sleep* signal is shown in Fig. 10, which has a *reset* input that initializes the *sleep* signal to logic 0.

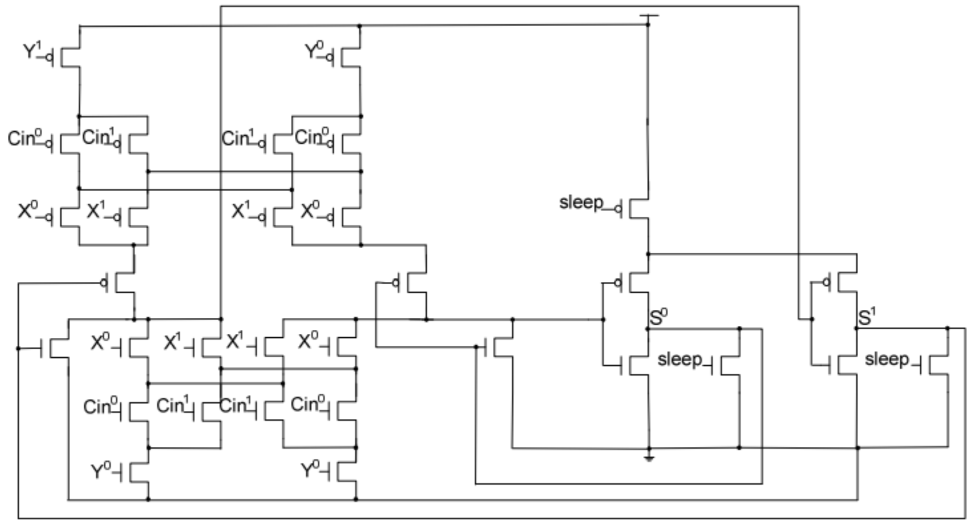


Figure 8. Static Implementation of PCHB Full Adder *Sum* Circuit [23].

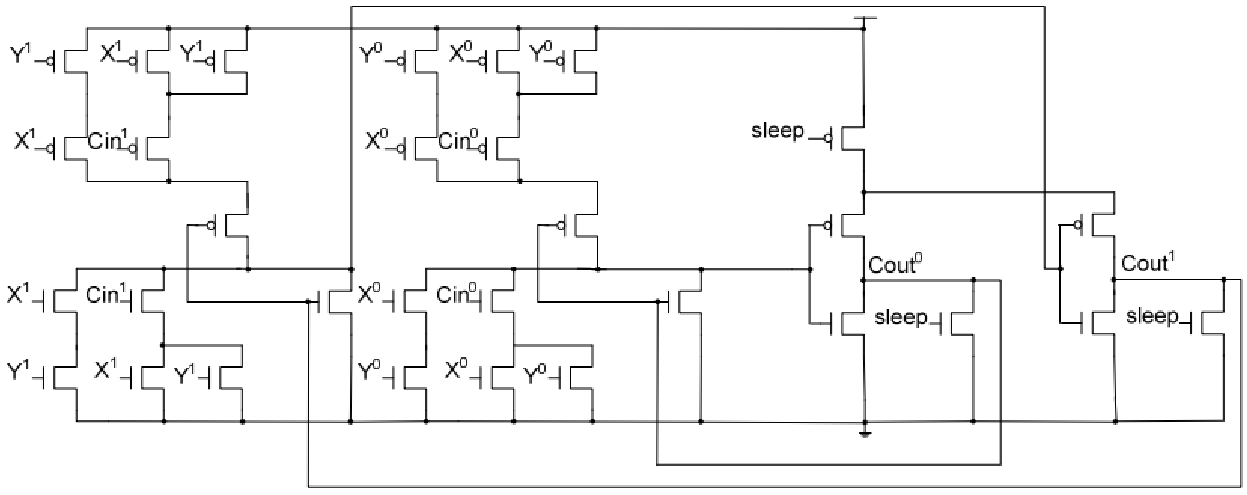


Figure 9. Static Implementation of PCHB Full Adder *Carry* Circuit [23].

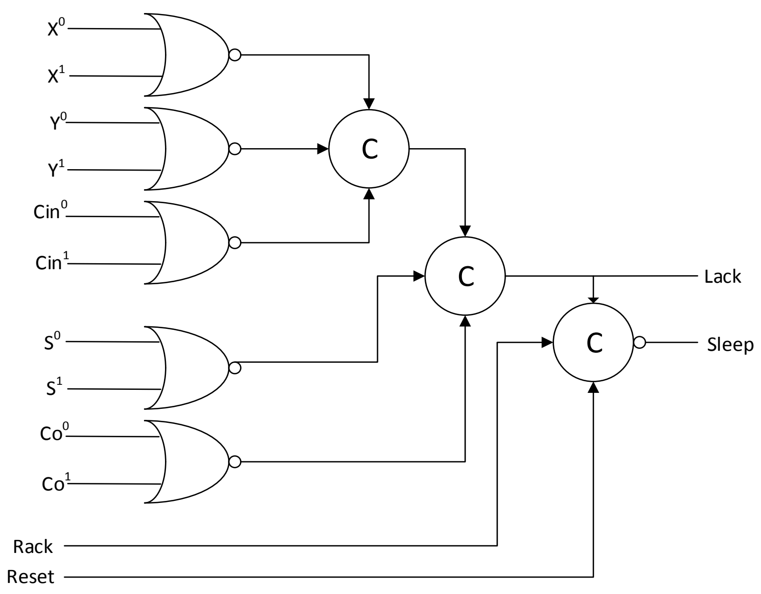


Figure 10. Generation of *Sleep* Signal for the PCHB Full Adder.

2.3.2. NULL Convention Logic (NCL)

Although NCL and PCHB are both QDI paradigms, both are structurally quite different from each other. The major distinguishing features are discussed below:

1. In PCHB, registration and functional units are integrated together, as discussed in previous subsection. However, in NCL, the registration and functional units are separated and independent.
2. NCL circuits are synthesized at the gate level; whereas, PCHB circuits are synthesized at transistor level.
3. NCL circuits have a library of 27 threshold gates (discussed later in this section) to attain delay-insensitivity, while PCHB gates do not utilize threshold gates. Instead, there are PCHB versions of individual Boolean gates, which are used to construct the circuit functionality.

4. NCL only utilizes registration outputs to generate the *acknowledge* signal. PCHB, on the other hand, uses both the inputs and outputs of registration stage to generate the *acknowledge* signal.

Fig. 11 shows the NCL architecture framework, which is quite similar to the synchronous systems. Each NCL combinational unit exists between two NCL registers. The NCL registers control the flow of DATA/NULL with the help of *request* signal (Ki), *acknowledge* signal (Ko), and completion detection unit.

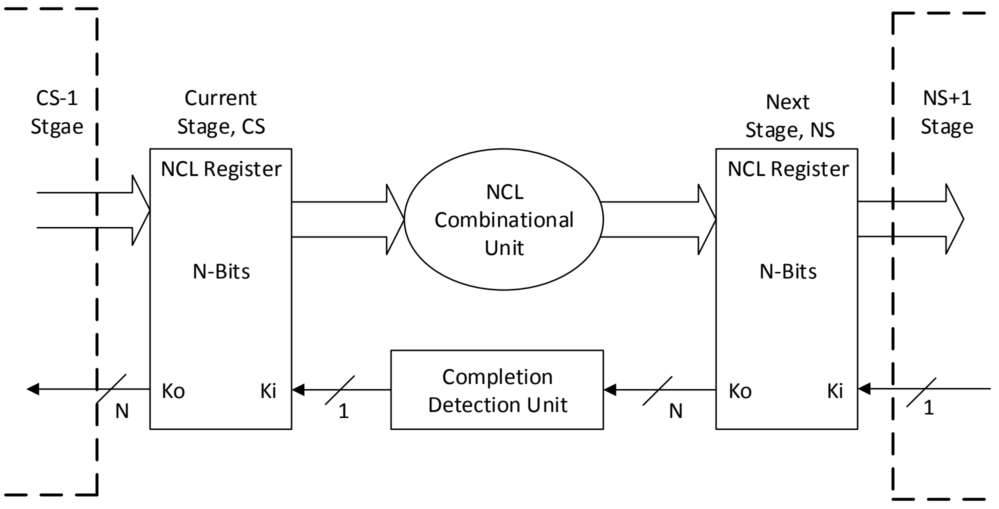


Figure 11. NCL Architecture Framework.
Adapted from [24].

The registration unit ensures that two different sets of DATA are always separated by a NULL wavefront through the *Ki* and *Ko* signals, establishing an alternating DATA, NULL, DATA, NULL… pattern. NCL combinational unit is comprised of threshold gates that construct a specific function. The completion detection unit detects whenever the NCL registration output is all DATA or all NULL. The complete flow of NULL and DATA in an NCL system is illustrated with the help of Fig. 12, 13, 14, and 15. One complete cycle corresponds to DATA flowing through the combinational logic (Fig. 12), followed by *rfn* flowing through the completion logic (Fig. 13), followed by NULL flowing through the combinational logic (Fig.

14), and then *rfd* flowing through the completion logic (Fig. 15). The flow of DATA/NULL and changes in *Ki* and *Ko* signals in each stage are highlighted in respective figures. The time required for a complete DATA/NULL propagation is similar to one synchronous clock cycle.

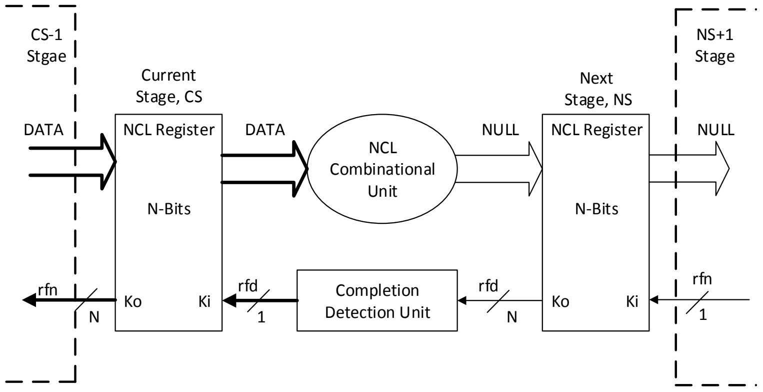


Figure 12. DATA Flow through Combinational Logic. Adapted from [24].

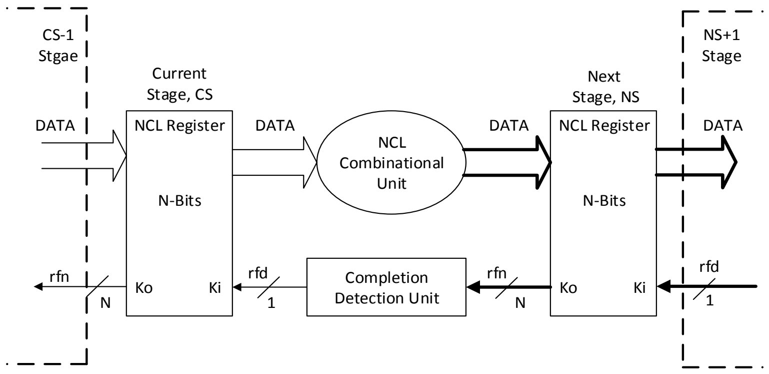


Figure 13. *rfn* Flow through Completion Logic. Adapted from [24].

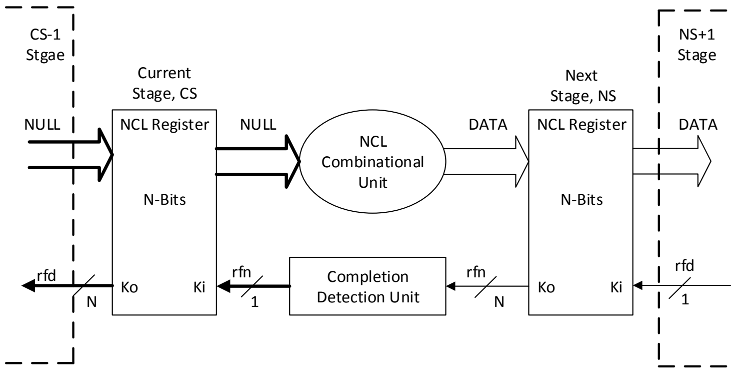


Figure 14. NULL Flow through Combinational Logic. Adapted from [24].

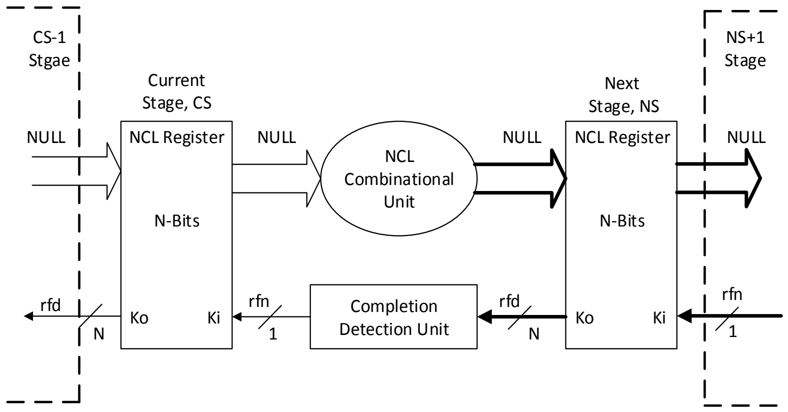


Figure 15. *rfd* Flow through Completion Logic.
Adapted from [24].

2.3.2.1. NCL combinational unit

The combinational unit is comprised of a combination of threshold gates from a library of 27 fundamental gates with state-holding capacity [26]. There are two types of threshold gates: non-weighted and weighted. The non-weighted threshold gates are represented as *THmn*, where *n* is the number of inputs and *m* ([1, n]) is the threshold; i.e., at least *m* inputs are required to be asserted to assert the output of the gate. Fig. 16 shows a TH23 gate as an example with 3 inputs and a threshold value of 2. The threshold value is written inside the gate structure. As *m=2*, at least two inputs from *A, B,* and *C* need to be asserted to generate the output. Hence, the equation of a TH23 gate is *AB+AC+BC*.

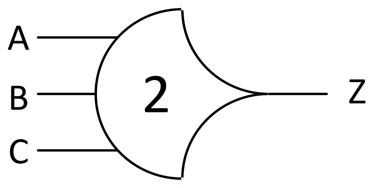


Figure 16. TH23 Gate as an Example of Non-weighted Threshold Gate.

The weighted threshold gates are denoted as *THmnWw*$_1$*, w*$_2$*... w*$_R$, where, w_R ($m \geq w_R > 1$) is the weight corresponding to input *R*. *m* and *n* are the threshold value and number of inputs respectively. Fig. 17 shows the TH24w22 gate as an example. As the threshold value, m=2; therefore, at least 2 inputs need to be asserted to assert the output. However, input 1 and 2 (*A* and *B*) are weighted with $(w_1, w_2) = 2$; hence, even if only *A* (or *B*) is asserted then the output Z will be asserted. Similarly, *C* and *D* both need to be asserted to meet the threshold of 2, as *C* and *D* have weights of 1. Hence, the equation of a TH24W22 gate is $A+B+CD$.

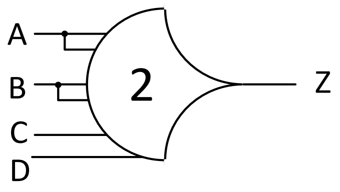


Figure 17. TH24W22 Gate as an Example of Weighted Threshold Gate.

All the 27 fundamental threshold gates ensure that once the output is asserted, all the inputs must be deasserted to de-assert the output, which is imperative to attain delay-insensitivity. Based on this behavior, a *THnn* gate is equivalent to an n-input C element, as output becomes logic 1/0 when all inputs are logic 1/0, otherwise, the output remains unchanged. In Fig. 4, the TH33 gate is used as a 3-input C element. NCL gates can also be designed with an additional *reset* input to initialize the output to either logic 0 or logic 1. A reset-to-1/reset-to-0 threshold gate is denoted with a *d*/*n* following the gate threshold in the gate symbol.

2.3.2.2. NCL registration unit

A single bit dual-rail NCL register structure is depicted in Fig. 18. A single bit NCL register uses two TH22 gates. A register can be resettable to either NULL, DATA0, or DATA1 by using resettable versions of the TH22 gates. For example, the register in Fig. 18 is reset-to-NULL register as both the TH22 gates are reset-to-0 type (TH22n). Similarly, the register can be designed as reset-to-DATA0/ reset-to-DATA1 by replacing one of the TH22n gates with a TH22d gate, which produces output rail^0/rail^1 output of the register, as shown in Fig. 19.

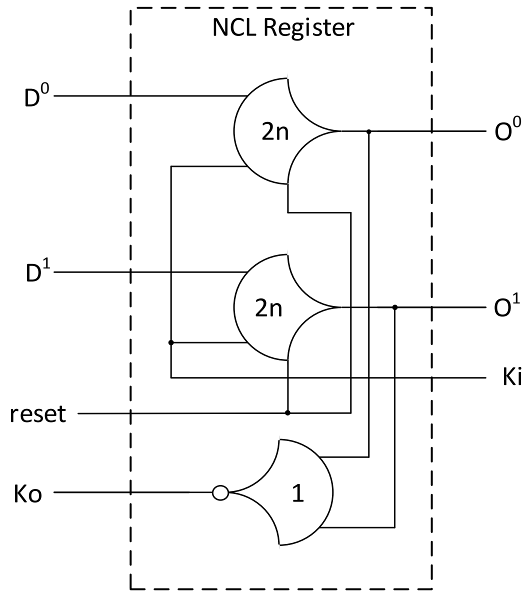


Figure 18. NCL Single Bit Reset-to-NULL Register.
Adapted from [24].

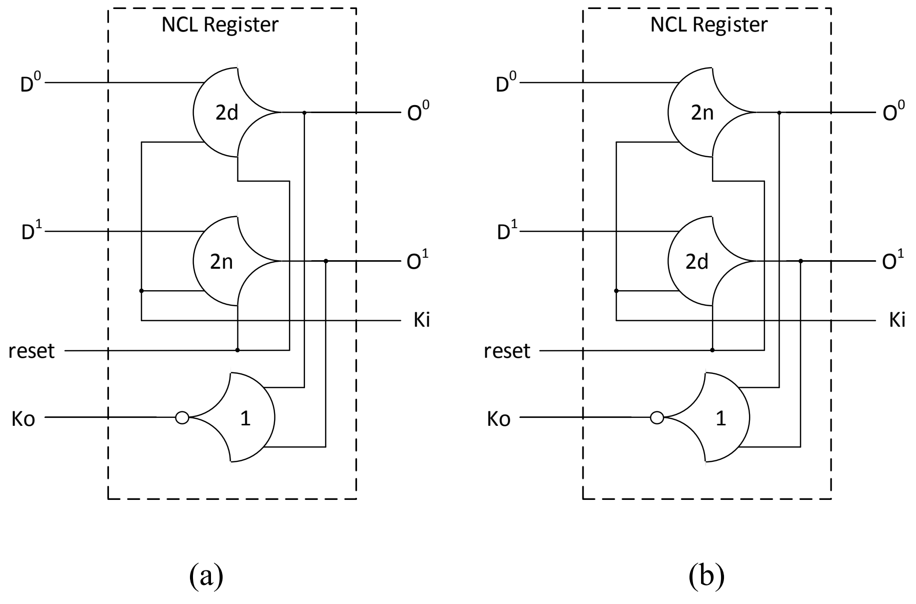


Figure 19. (a) Reset-to-DATA0 NCL Register. (b) Reset-to-DATA1 NCL Register.

The TH22 gates allows the input DATA to pass to the output only when the *request* signal, *Ki*, is *rfd* (logic 1). When the output is DATA, the *acknowledge* signal, *Ko*, becomes *rfn* (logic 0), requesting NULL at the inputs. The output returns to NULL only when both the register input is NULL and *Ki* is *rfn*. A N-bit NCL register is a combination of *N* single rail NCL registers with *N* dual-rail inputs, one *reset*, *N Ki* inputs, *N* dual-rail outputs, and *N Ko* outputs.

2.3.2.3. NCL completion unit

N Ko lines of *N*-bit NCL registers in a particular stage go through a completion unit, which is a tree structure comprising of C-elements (THnn gates). *N Ko* bits are converted to one bit with the help of the completion unit. The maximum number of inputs for a THnn gate is 4; hence, the number of logic levels in NCL completion unit for *N*-bit register is $\log_4 N$. For example, for 16-bit registers in a stage, the completion unit will have 2 levels of THnn gate arrangement, as shown in Fig. 20. For full-word completion, the single bit output from the completion component is fed into the *Ki* inputs of all the previous stage registers.

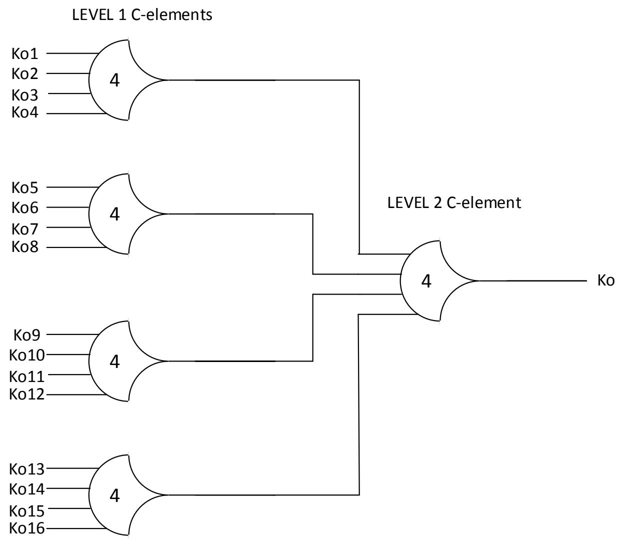


Figure 20. 16-bit NCL Completion Unit.

2.4. Related Verification Works

Several formal verification techniques have been implemented to verify the two major asynchronous design paradigms: bounded-delay and QDI. As discussed earlier in this section, the bounded-delay model is based on the assumption that the delay in all circuit components and wires is bounded, i.e., worse case delay can be calculated. Because of these timing constraints, most of the verification schemes for timed asynchronous models involve trace theory, Signal Transition Graph (STG) [27], and timed petri-nets. A Trace Theory based method [28] was proposed to verify various asynchronous circuits at the gate level, such as Huffman circuits and Muller circuits, where the circuit behavior is represented as sets of traces, and the correctness properties are modeled as petri-nets. An approach based on time-driven-unfolding of petri-nets is used to verify freedom from hazards in asynchronous circuits consisting of logic gates and Micropipelines [29]. However, timed-model based verification methods are not applicable for QDI circuits, which are based on exactly the opposite assumption, that circuit delays are unbounded and therefore indeterminate.

There exist several verification schemes specific to QDI circuits as well. Verbeek and Schmaltz [30] illustrate a deadlock-verification scheme for QDI circuits based on the Click Library [31]. Circuits based on this primitive library are structurally different from other QDI paradigms, such as NCL and PCHB. Moreover, this method does not verify the functional correctness (safety) of the circuit. Refinement based formal methods have been successful in verifying both bounded-delay and QDI asynchronous models. Desynchronized circuits, which are based on a bounded-delay structure, can be verified by a refinement based approach, as discussed in [32]. [33] presents a method to check the functional equivalence of NCL circuits against their synchronous counterparts using WEB refinement [34]. However, this technique suffers from state space explosion, since they model the QDI circuits as TSs, which become very complex for large circuits due to the non-deterministic behavior of QDI paradigms.

Input-completeness and observability are two critical properties of NCL circuits, which must be verified in order to ensure delay-insensitivity, since a circuit may function correctly under normal operating conditions while not being input-complete or observable, but may then malfunction under extreme timing scenarios, such as those caused by process, voltage, or temperature (PVT) variations. *Input-completeness* is a condition that mandates the outputs to transition from NULL to DATA only after all the inputs have transitioned from NULL to DATA. Similarly, the outputs may transition from DATA to NULL only after all the inputs have transitioned from DATA to NULL. *Observability* is a condition ensures that every gate that transitions is necessary to transition at least one of the outputs. A manual approach to checking input-completeness is outlined in [24], which requires an analysis of each output term. For example, in order for output Z to be input-complete with respect to input A, every product term in all rails of Z (in SOP format) must contain any rail of A. This ensures that Z cannot be DATA until A is DATA, and if Z is constructed solely out of NCL gates with hysteresis, the gate hysteresis ensures that Z cannot transition from DATA to NULL until A transitions from DATA to NULL. Hence, Z is input-complete with respect to A. However, this method cannot ensure input-completeness of relaxed NCL circuits [35], where not all gates contain hysteresis. Also, scalability is a problem with this approach, as the number of product terms that need to be verified grows exponentially as the number of inputs increase. Kondratyev et. al. [36] provide a formal verification approach for observability verification, which entails determining all input combinations that assert $gate_i$, then forcing $gate_i$ to remain de-asserted while checking that none of those input combinations result in all circuit outputs becoming DATA. This check is performed for all gates to ensure circuit observability; and if also applied to each circuit input (i.e., replace $gate_i$ with $input_i$ in the observability check explanation), will guarantee input-completeness. [37] illustrates an input-completeness verification method, which is not manual and can ensure the input-completeness of relaxed circuits; also, proposes an observability checking approach that is very similar to [36].

There also exist some methods directly applicable to QDI PCHB circuits, as described below, but these also have major limiting factors. In [38], a reverse synthesis-based approach that creates a high-level specification from a PCHB circuit is presented; however, in case of a bug, the methodology does not address the issue of finding the error. For example, if the QDI circuit is buggy (e.g., a completion signal is missing in a completion network, such that under some extreme timing scenarios the QDI circuit will malfunction), it is not clear if/how this will be preserved in the reverse synthesized output. Also, [38] is only applicable to control circuits, not datapath circuits. In [39], Shih et. al. developed a deadlock verification scheme for sequential PCHB circuits that detects deadlocks by transforming the asynchronous pipeline into a Time Marked Graph, removing all edges containing initial tokens, and then detecting any remaining cycles (i.e., a deadlock free circuit should be acyclic after removal of all initial token edges). The method effectively identifies deadlocks in any sequential PCHB circuit; however, it does not address verification of the combinational logic (C/L), neither functionality nor handshaking connections. [39] assumes that their optimized synthesis method for generating a combinational PCHB circuit from its Boolean specification, presented in [40], is correct. For example, inversion in a handshaking signal within the C/L would cause deadlock, and swapped rails of a dual-rail signal would produce incorrect results, but not deadlock the system, neither of which would be detected by [39]. In [41], a testing method is proposed for template based asynchronous circuits. Testing can detect shallow bugs, whereas formal verification effectively finds corner case bugs, which is why testing and formal verification are implemented independently in industry.

Therefore, the goal of the research work presented in this dissertation is to develop formal methods and verification methods for QDI PCHB and QDI NCL circuits, which guarantee *safety* (functional correctness) as well as *liveness* (absence of deadlock) of the circuits. The developed verification methods are discussed in details in the next sections.

3. FORMAL MODELING AND VERIFICATION METHODS FOR QDI PCHB ASYNCHRONOUS CIRCUITS

This chapter outlines the formal modeling and verification of QDI PCHB gates and circuits. The chapter provides a detailed description of the developed verification methodologies based on model checking and equivalence verification.

3.1. Model Checking Based Verification Method

The methodology is implemented in two folds: 1) modeling the gates as Transition Systems (TSs), unlike Boolean gates that are modeled as Boolean functions, while circuit models are obtained by composing the gate TS; 2) developing formal property templates that can be used to capture the correctness of any PCHB circuit that corresponds to a combinational Boolean circuit. The property templates account for both *safety* (circuit outputs are functionally correct) and *liveness* (circuit is never deadlocked).

3.1.1. PCHB Formal Gate Model

Since PCHB gates have memory and implement a handshaking protocol, we have developed a TS to model the behavior of a gate. This TS is shown in Fig. 21 and is one of the key contributions. The state variables of the model are *Lack*, *Rack*, *Din* (represents all dual-rail data inputs), and *Do* (represents the dual-rail data output). *Lack* and *Rack* are Boolean variables. *Din* can have three values: D (valid DATA on all inputs); N (NULL value on all inputs); and PD (Partial DATA, meaning at least one of the inputs is not available, i.e., has a NULL value). Below are the rules to compute the initial state and state transitions. *Do* can be either D (DATA) or N (NULL).

Generic Rules to Compute PCHB Gate Transition Systems.

```
1: Initialize: Lack =1, Rack=0, Din= N, Do =N
2: (Rack=0 ∧ Do= N) → Rack= 1
3: (Rack=1 ∧ Do = D) → Rack= 0
4: (Lack=1 ∧ Din = N) → Din = PD
5: (Lack=1 ∧ Din = PD) → Din = D
6: (Lack=1 ∧ Rack=1 ∧ Din =D ∧ Do =N) → (Do =D)
7: (Din = D ∧ Do = D) → Lack=0
8: (Lack=0 ∧ Rack=0) → (Do =N)
9: (Lack=0 ∧ Din = D) → Din = PD
10: (Lack=0 ∧ Din = PD) → Din = N
11: (Din = N ∧ Do = N) → Lack=1
12: (Lack=1 ∧ Rack =1 ∧ Din = PD ∧ Do =N) → (Do =D) **
```

The data value attained by *Do* depends on the gate function. Otherwise, the TS shown in Fig. 21 can be used to model any *n*-input PCHB gate. Rule 12 corresponds to the case when Partial DATA is sufficient to compute the output of the gate function, (e.g., if one of the inputs has a value of 0 for a NAND gate, then the output will be 1 regardless of the other inputs). In such cases, the PCHB gate will compute the output even before the other input data values are available. '*' represents intermediate states and the numbers in the TS correspond to the rules applied. The TSs of different PCHB gates were verified by performing Cadence transistor level simulation, changing one input in each step.

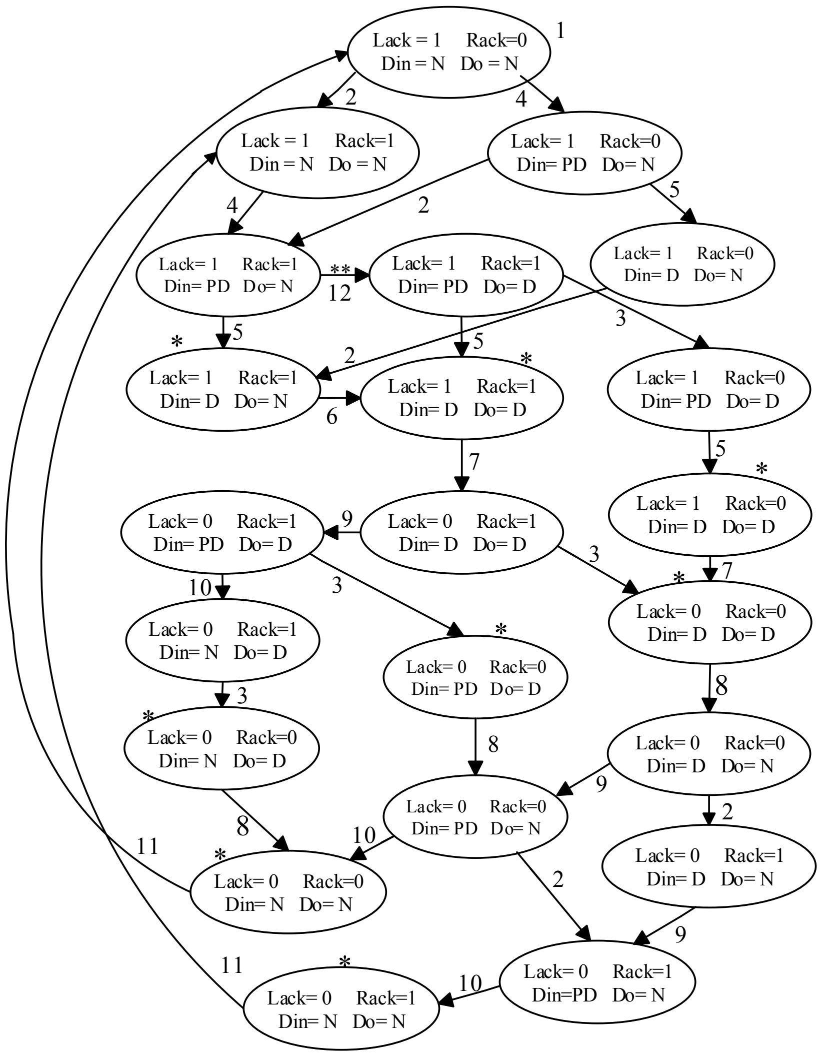

Lack = 1 Rack=0 Din = N Do = N
1
Lack = 1 Rack=1 Din = N Do = N
Lack= 1 Rack=0 Din= PD Do= N
Lack= 1 Rack=0 Din= D Do= N
Lack= 1 Rack=1 Din= PD Do= N
Lack= 1 Rack=1 Din= PD Do= D
Lack= 1 Rack=1 Din= D Do= N
Lack= 1 Rack=1 Din= D Do= D
Lack= 1 Rack=0 Din= PD Do= D
Lack= 1 Rack=0 Din= D Do= D
Lack= 0 Rack=1 Din= PD Do= D
Lack= 0 Rack=1 Din= D Do= D
Lack= 0 Rack=0 Din= D Do= D
Lack= 0 Rack=1 Din= N Do= D
Lack= 0 Rack=0 Din= PD Do= D
Lack= 0 Rack=0 Din= N Do= D
Lack= 0 Rack=0 Din= D Do= N
Lack= 0 Rack=0 Din= PD Do= N
Lack= 0 Rack=1 Din= D Do= N
Lack= 0 Rack=0 Din= N Do= N
Lack= 0 Rack=1 Din=PD Do= N
Lack= 0 Rack=1 Din= N Do= N
**
12
*

Figure 21. State Diagram of a PCHB *n*-input NAND Function.

3.1.2. Proposed Methodology for Verification

In model checking, the circuit to be verified is modeled as a TS and correctness properties are specified using a temporal logic. In the proposed approach, Computational Tree Logic (CTL) [42] is used to specify the correctness properties. A set of property templates is developed for PCHB circuits, which is the second contribution. The property templates can be used to verify any PCHB circuit that corresponds to a combinational Boolean circuit. Note that PCHB circuits themselves are not combinational as each gate incorporates registration and control in addition to logic function. The templates can be classified as a set of *local* templates and one *global* template. The *local* templates are applicable locally to each PCHB gate, and check for liveness of the circuit, which is the absence of deadlock. The *global* template checks for safety, i.e., that under all circumstances, the circuit output is always correct. NuSMV model checker [43] is utilized to check the CTL properties of PCHB circuits, modelled as TSs, which does not require any modifications to NuSMV. As used in the properties below, *AG* stands for Always Global, which means that it is always true; *A f1 U f2* means that *f1* is always true until *f2* is true.

3.1.2.1. Input properties

P1: AG ((Din = D) ^ Lack = 1 →A [(Din = D) U Lack = 0]).

P2: AG ((Din = N) ^ Lack = 0 →A [(Din = N) U Lack = 1]).

P1 states that if all inputs are DATA and the acknowledge signal (*Lack*) is *rfd*, then the inputs will not change until *Lack* becomes *rfn*. Similarly, *P2* states that if all inputs are NULL and *Lack* is *rfn*, the inputs will remain *NULL* until *Lack* becomes *rfd*.

P3: AG ((Din = D) ^ Lack = 0 →A [(Din = D) U ((Din = PD) ∨ Din = N))]).

P4: AG ((Din = N) ^ Lack = 1 →A [(Din= N) U ((Din = PD) ∨ Din = D))]).

P3 states that if all inputs are DATA, they will eventually change to Partial DATA or NULL if *Lack* is *rfn*. Similarly, *P4* states that if all inputs are NULL and *Lack* is *rfd*, they will eventually change to Partial DATA or DATA.

P5: AG ((Din = PD) ^ Lack = 1 → A [(Din = PD ^ Lack = 1) U (Din = D)]).

P6: AG ((Din = PD) ^ Lack = 0 →A [(Din = PD ^ Lack = 0) U (Din = N)]).

P5 states that if the inputs are Partial DATA and *Lack* is *rfd*, the inputs will eventually become DATA; and *P6* states that if the inputs are Partial DATA and *Lack* is *rfn*, the inputs will eventually become NULL.

3.1.2.2. Rack and Lack properties

P7: $AG ((Rack = 1) \wedge (Do = D) \rightarrow A [(Rack = 1) \wedge (Do = D)) U Rack = 0])$.

P8: $AG ((Rack = 0) \wedge (Do = N) \rightarrow A [(Rack = 0) \wedge (Do = N)) U Rack = 1])$.

P7 states that whenever the output is DATA and the Request signal (*Rack*) is *rfd* (logic 1), then *Rack* will eventually change to *rfn* (logic 0) to request NULL at the output. *P8* states that if *Rack* is *rfn* and there's a NULL at the output, *Rack* will eventually become *rfd* to request DATA at the output.

P9: $AG (Lack = 0 \rightarrow A [(Lack = 0) U ((Din = N) \wedge (Do = N))])$.

P10: $AG (Lack = 1 \rightarrow A [(Lack = 1) U ((Din = D) \wedge (Do = D))])$.

P9 states that if *Lack* is *rfn*, then it will remain *rfn* until all inputs and outputs become NULL. Similarly, *P10* states that if *Lack* is *rfd*, it will only change to *rfn* when all inputs and outputs become DATA.

3.1.2.3. Output properties

P11: $AG (Rack = 1 \wedge Lack =1 \wedge (Din = PD) \wedge (Do = N) \rightarrow A [(Rack = 1 \wedge Lack = 1 \wedge (Din = PD) \wedge (Do = N)) U (Rack = 1 \wedge Lack = 1 \wedge (Din = D) \wedge (Do = N)) \vee (Rack = 1 \wedge Lack =1 \wedge (Din = PD) \wedge (Do = D))])$.

P11 states that if *Rack* and *Lack* are both *rfd*, the inputs are Partial DATA, and the output is NULL, then in the next state either the inputs will become DATA, or the output may evaluate with the Partial DATA input if the logic allows. (e.g., for a NAND function, where at least one input is DATA0).

P12: $AG (Rack = 1 \wedge Lack = 1 \wedge (Din = D) \wedge (Do = N) \rightarrow A [(Rack = 1 \wedge Lack = 1 \wedge (Din = D) \wedge (Do = N)) U (Rack = 1 \wedge Lack = 1 \wedge (Din = D) \wedge (Do = D))])$.

P12 states that the output will evaluate when *Lack* = *Rack* = *rfd* and the inputs are DATA, as mentioned in Section 2.3.1.

P13: AG (Rack = 0 ∧ Lack = 0 ∧ (Din = D) ∧ (Do = D) →A [(Rack = 0 ∧ Lack = 0 ∧ (Do = D)) U (Rack = 0 ∧ Lack = 0 ∧ (Do = N))]).

P14: AG (Rack = 0 ∧ Lack = 0 ∧ (Din = N) ∧ (Do = D) → A [(Rack = 0 ∧ Lack = 0 ∧ (Din = N) ∧ (Do = D)) U (Rack = 0 ∧ Lack =0 ∧ (Din=N) ∧ (Do= N))]).

P15: AG (Rack = 0 ∧ Lack = 0 ∧ (Din = PD) ∧ (Do = D) →A [(Rack = 0 ∧ Lack = 0 ∧ (Do = D)) U (Rack = 0 ∧ Lack = 0 ∧ Do = N))]).

Properties *P13*-*P15* indicate that whenever *Lack* and *Rack* both become *rfn*, they cause the output to become NULL.

P16: AG ¬ (Do [0] =1 ∧ Do [1] =1).

P16 verifies that the output rails of the gate are never asserted simultaneously.

3.1.2.4. N-stage pipeline architecture and the global property

One of the major contributions of this work is modelling a *global property template* for *N*-stage PCHB circuits that checks the functional correctness of the circuit. The key problem here is that the input values corresponding to a certain output become available at different times. Therefore, we require a mechanism that keeps track of all input values corresponding to a particular output value. A pipeline structure is utilized to do this, which is included as part of the circuit model. The length of the pipeline is the number of stages in the circuit. Each stage in the pipeline includes variables that correspond to each of the inputs. Each output also has a variable in every pipeline stage. The pipeline keeps track of the DATA values for each input and preserves their order. When a complete set of inputs reach the end of the pipeline and all output values are DATA, the global property checks that for all such states, the output value is equal to the evaluation of the Boolean function *fc* when applied on the corresponding set of inputs. *fc* corresponds to the Boolean function implemented by the PCHB circuit (e.g., $X \oplus Y \oplus Ci$ for the sum output of a full adder). The logic for constructing the pipeline is given next followed by the global property.

Generic Model for N-stage Pipeline Structure Used for Global Property.

1: $(X_{m1}, X_{m2}, X_{m3}\ldots\ldots X_{mN}) \leftarrow \text{NULL}$; for each value of *m*.

2: $(X_{IN} \neq X_1) \wedge (X_1{=}X_2{=}X_3{=}\ldots\ldots = X_N = \text{NULL}) \rightarrow (X_1 = X_2 = X_3 = \ldots\ldots = X_N = X_{IN})$.

3: $(X_{IN} \neq X_1) \wedge (X_1 = X_2) \rightarrow (X_i = X_{IN})$ until $(X_i \neq X_{i+1})$; '*i*' ranges from 1 to *(N-1).*

4: $(X_{1N} = X_{2N} = X_{3N} = \ldots\ldots = X_{mN} = \text{DATA}) \mid (X_{1N} = X_{2N} = X_{3N} = \ldots\ldots = X_{mN} = \text{NULL}) \rightarrow (X_{mj} = X_{m(j-1)})$; for each value of *m*, where '*j*' ranges from *N* downto 2.

The variables of the pipeline are history variables in that they do not impact the functionality of the circuit but are used only for verification purposes. In the model above, *X*s are the variables in the pipeline stages and there are '*m*' variables in each stage. Step 1 initializes all pipeline stages to NULL. Step 2 flows the first DATA value through the entire pipeline when the pipeline is all NULL, such that all stages become DATA. Step 3 flows either a DATA or NULL through all initial stages of the pipeline until it is blocked by a NULL or DATA, respectively, such that there is always a NULL between every two DATA values. Step 4 shifts the entire pipeline for all inputs and outputs one stage to the right when the last stage of the pipeline for all inputs and outputs are either all DATA or all NULL. In this case when all DATA, we are required to check if the model satisfies the global property below, which ensures that the combinational PCHB function is equivalent to the corresponding Boolean function.

GP: AG ($X_{1N}= X_{2N} =X_{3N} =…..= X_{mN}$ = DATA →A [(X_{k+1}, X_{k+2}… X_m) = *fc* (X_1, X_2… X_k)]), where there are *k* inputs and *m-k* outputs.

3.1.3. Results

From basic PCHB functions to more complex multifunction circuits, the presented formal verification method was used to verify 11 PCHB circuits. NuSMV model checker v2.6 was used. The verified circuits are listed below in Table 3, along with their respective verification time. Note that in addition to connecting different PCHB gates together to form these circuits, completion logic, composed of C-elements, was also added, which were also modelled as TSs for NuSMV.

Several bugs were injected to test the methodology, including interchanging the rails of dual-rail signals, incorrect connections between *Lack* and *Rack*, malfunctioning completion detection circuitry, and incorrect logical function. Our proposed methodology flagged all bugs and also produced counter-examples to trace their paths. Verification was performed on an Intel® Core™ i7-4790 CPU @ 3.6 GHz with 32 GB of RAM and 64-bit operating system.

Table 3. Test Circuit Verification Times for Model Checking Approach.

Circuit	**Time (sec.)**	**Circuit**	**Time (sec.)**
Full Adder	21.54	16- bit MUX	204.4
ISCAS'85 C-17 [45]	138.62	32- bit MUX	413.23
1- bit MUX	11.33	2- to- 4 decoder	44.37
2- bit MUX	21.49	3- to- 8 decoder	163.37
4- bit MUX	50.65	4- to- 16 decoder	222.6
8- bit MUX	99.78	---	---

3.1.4. Discussions

PCHB circuits are very complex because each gate incorporates registration and control. Thus, a sequence of gates operates similar to a synchronous pipeline. Our model checking approach guarantees that under all circumstances, the PCHB circuit functions correctly. This is the first approach that we are aware of that verifies both safety and liveness for PCHB circuits. The verification engineer only has to fill-out the property templates to use this approach. However, scalability is the major limiting factor of this verification approach. Since PCHB gates each incorporate hysteresis state holding capability with a complex handshaking scheme, the corresponding transition system for a PCHB circuit is very complex, even for relatively simple circuits. For example, a simple PCHB n-input NAND gate contains 22 states in its TS, as shown in Fig. 21. When multiple gates are connected together to form a circuit, the state space increases exponentially. This causes state space explosion, which in turn results in an infeasible verification time.

Therefore, an alternate verification methodology is presented to circumvent having to deal with the complex transition system. This alternate method is a unified verification approach based on equivalence verification that also guarantees safety as well as liveness for any PCHB circuit; it is fast and highly scalable, and discussed in details in the next section.

3.2. An Equivalence Verification Methodology for Combinational and Sequential QDI PCHB Asynchronous Circuits

This section provides an enumeration of all possible faults that can occur during PCHB synthesis, illustrates the proposed verification approach for combination as well as sequential PCHB circuits, followed by a demonstration on how the method detects every possible faults.

3.2.1. Enumeration of All Possible PCHB Faults

The development of our verification method for PCHB circuits is based on the assumption that individual PCHB gates are fault-free, which is consistent with standard gate-level verification methodologies. The developed method ensures that no interconnections between gates are erroneous, and that the implemented PCHB function is equivalent to its Boolean/synchronous specification. Below is an enumeration of all possible faults that could occur in a PCHB circuit.

Case 1. *Faulty data connection*: Each PCHB gate receives its data inputs from the circuit's primary data inputs and/or other PCHB gate data outputs. A PCHB gate's data input or circuit output could be the wrong dual-rail signal. For example, the F output of PCHB $gate_i$ should be connected to the X input of PCHB $gate_j$; however, X is instead connected to the output of PCHB $gate_k$, which would result in a logical error, such that the specification and implementation circuits would not be functionally equivalent.

Case 2. *Swapped dual-rail connection*: In PCHB circuits, all data signals are dual-rail logic, where two wires together represent one bit of data, as detailed in Chapter 2. The rails of a dual-rail data input or circuit output could be unintentionally swapped. For example, if PCHB $gate_i$ output F is supposed to connect to PCHB $gate_j$ input X, this implies that F_i^0 and F_i^1 should connect to X_j^0 and X_j^1, respectively. However, swapping the dual-rail connections would result in F_i^0 and F_i^1 connected to X_j^1 and X_j^0, respectively, which would correspond to inversion of that signal, resulting in a logical error, such that the specification and implementation circuits would not be functionally equivalent.

Case 3. *Rails from different signals*: A PCHB gate's data input or circuit output could be incorrectly comprised of two different dual-rail signals' rails. For example, the X^1 input of PCHB gatei is connected to the F^1 output of PCHB $gate_j$ and the X^0 input of PCHB $gate_i$ is connected to the G^0 output of PCHB $gate_k$. This will result in the circuit deadlocking when F = DATA0 and G = DATA1, since X will never transition to DATA, and will result in X being an illegal value (i.e., $X^0 = 1$ and $X^1 = 1$) when F = DATA1 and G = DATA0.

Case 4. Rail Duplication: A PCHB gate's data input or circuit output could be incorrectly comprised of the same rail of a dual-rail signal. For example, both rails of the *X* input of PCHB gatei are connected to the F^1 output of PCHB $gate_j$. This will result in the circuit deadlocking when *F* = DATA0, since *X* will never transition to DATA, and will result in *X* being an illegal value when *F* = DATA1.

Case 5. Handshaking signal connected to data signal: A PCHB gate's data input or circuit output could be either partially or fully comprised of one or two handshaking network signals (i.e., PCHB gate *Rack*/*Lack* signal or C-element output), which would result in the affected dual-rail signal being stuck at NULL for some cases, causing circuit deadlock, and being an illegal value for other cases.

Case 6. Incorrect logic implementation: The functionality of the PCHB circuit is not equivalent to its specification. For example, the specification $F = AB+C$ is implemented as a PCHB circuit utilizing a 2-input AND gate, followed by a 2-input XOR gate, instead of a correct implementation that utilizes a 2-input AND gate followed by a 2-input OR gate.

Case 7. Non-PCHB gate in datapath: The datapath of PCHB circuits consists entirely of PCHB gates, which all have 1 or more data inputs, 1 or more data outputs, a *Rack* input and a *Lack* output. Any type of gate other than a PCHB gate in the datapath is an error, which may cause the circuit to deadlock or result in a logical error, such that the specification and implementation circuits would not be functionally equivalent.

Case 8. Incorrect reset: Every PCHB gate includes a *reset* input to initialize the gate's data output to either NULL, DATA0, or DATA1, all of which must be connected to the circuit's external reset input, which itself must not be connected to any other gate input. A PCHB gate with incorrect *reset* value will either result in the circuit deadlocking or not being functionally equivalent to its specification. Take for example a Multiply and Accumulate unit (MAC), where all outputs should be reset to DATA0, according to its specification. If instead one or more of the PCHB implementation's outputs are reset to DATA1, the results of the first MAC operation will differ from its specification (i.e., $A_1 = A_0 + X_1 \times Y_1$, where $A_0 \neq 0$, vs. the correct implementation: $A_1 = 0 + X_1 \times Y_1$). If instead an output is reset to NULL, and all other PCHB gates in its respective feedback loop are also reset to NULL (this would be the typical reset state of these other gates), this would result in a feedback path with no DATA tokens, which would cause the circuit to deadlock.

Case 9. <u>Insufficient registers in a feedback loop</u>: The 4-phase QDI handshaking protocol utilized for PCHB circuits requires at least *2N+1* PCHB registers/latches in a feedback loop that contains *N* DATA tokens, in order to avoid deadlock [24]. For example, a feedback loop with a single DATA token, such as a MAC (i.e., $A_i = A_{i-1} + X_i \times Y_i$), requires at least 3 PCHB registers in every feedback path, otherwise the circuit will deadlock. Note that every PCHB gate includes an internal latch; so, for the MAC example, a feedback path that includes at least 3 PCHB gates is sufficient.

Case 10. <u>Missing handshaking signal</u>: Each PCHB gatej ($j \in [1, N]$) whose data input is a data output of PCHB *gate*$_i$, must acknowledge PCHB *gate*$_i$, resulting in the *Lack* output of each PCHB *gate*$_j$ being conjoined via an *N*-input C-element structure, whose output is the *Rack* input of PCHB *gate*$_i$. For example, if a data output of PCHB *gate*$_x$ is a data input of PCHB *gate*$_y$ and the *Lack* output of PCHB *gate*$_y$ is not an input to the C-element structure that generates the *Rack* input to PCHB *gate*$_x$, the circuit will deadlock under some timing scenarios.

Case 11. <u>Additional handshaking signal</u>: If the C-element structure that generates the *Rack* input for PCHB *gate*$_i$ contains a *Lack* input from PCHB *gate*$_j$, and a data output of PCHB *gate$_i$* is not a data input of PCHB *gate$_j$*, the circuit may deadlock or slowdown, but could also operate correctly. This isn't necessarily an error; however, the additional handshaking signal requires further inspection.

Case 12. External Lack error: The external *Lack* output synchronizes all circuit primary data inputs. Hence, the *Lack* outputs of all PCHB gates that have a circuit primary data input as a data input must be combined through a C-element structure to produce the external *Lack* output. Like Case 10, any missing *Lack* input to this C-element structure will cause the circuit to deadlock under some timing scenarios. Similar, but different to Case 11, any additional *Lack* input to this C-element structure is an error, which may cause the circuit to slowdown or deadlock.

Case 13. External Rack error: The external *Rack* input synchronizes all circuit primary data outputs. Hence, for each PCHB *gate$_i$* whose data output is a circuit primary data output, the external *Rack* input must either be the *Rack* input to *gate$_i$* or an input to the C-element structure that generates the *Rack* input for *gate$_i$* (as would be the case when an external data output is fed back to another PCHB gate). Like Case 10, if the external *Rack* input is missing from the C-element structure that generates the *Rack* input for a PCHB gate whose data output is a circuit primary data output, the circuit will deadlock under some timing scenarios.

Case 14. Non-C-element in handshaking circuitry: PCHB handshaking circuitry is composed entirely of C-element structures, which consist of 0 or more C-elements that combine *N* Lack signals into a single *Rack* signal or the external *Lack*. Hence, any gate other than a C-element in the PCHB handshaking circuitry is an error, which will cause the circuit to deadlock under some timing scenarios.

Case 15. *Data signal input to C-element*: As mentioned in Case 14, C-elements only occur in the handshaking circuitry to combine *Lack* signals; they are not utilized in the datapath. Hence, either rail of a data signal being an input to a C-element is an error, which will cause the circuit to deadlock under some timing scenarios.

Case 16. *Data signal input to PCHB gate Rack input*: As mentioned in Cases 10 and 13, a PCHB gate's *Rack* input may only be the output of a C-element, another PCHB gate's *Lack* output, or the external *Rack* input. Hence, either rail of a data signal being a PCHB gate's *Rack* input is an error, which will cause the circuit to deadlock.

Case 17. *C-element structure feedback*: As mentioned in Cases 10 through 13, a C-element structure combines multiple PCHB gate *Lack* outputs, and possibly the external *Rack* input, to generate PCHB gate *Rack* inputs or the external *Lack* output; hence, C-element structures are feedforward only, such that any feedback loop within a C-element structure is an error, which will cause the circuit to deadlock.

Case 18. *Shorted output*: An output of a C-element or any output of a PCHB gate cannot be directly connected to any other PCHB gate or C-element output, or any external input. This would result in a wire short, causing the affected signal to be undefined when the logical values of the shorted wires differed.

These 18 cases comprise all possible faults that could occur in a PCHB circuit synthesized from a Boolean/synchronous specification, comprised solely of PCHB gates and C-elements (i.e., no special asynchronous control elements, such as F-element, D-element, Tangram S-element, etc.). PCHB gates have *n* dual-rail inputs, *1* Rack input, *m* dual-rail outputs, and *1* Lack output. C-elements have *k* Boolean inputs and *1* Boolean output. In order to establish our claim that the abovementioned 18 cases comprise all possible faults, we analyze an exhaustive conjunction of PCHB gates, C-elements, and PCHB circuit inputs and outputs, from which only a small set of connections are legal, and a large set of connections are illegal/faulty. We then illustrate how every faulty connection can be categorized by at least one of the above 18 fault case scenarios.

Let us consider a PCHB circuit with *N* PCHB gates, *M* C-elements, *X* external data inputs, *Y* external data outputs, a single *Rack* input, a single *Lack* output, and a single *reset* input. Considering all possibilities, the dual-rail output of PCHB $gate_i$ has the following twelve interconnection scenarios: it could be connected to: i) dual-rail input(s) of other PCHB $gates_j$, where $j \neq i$, ii) external data output, iii) dual-rail input(s) of other PCHB gates, including $gate_i$, iv) *Rack* input of a PCHB gate, v) C-element input, vi) another PCHB gate data output, vii) a PCHB gate *Lack* output, viii) a C-element output, ix) external data input, x) external *Rack* input, xi) external *reset* input, or xii) external *Lack* output. Of these, only i) and ii) are possibly correct, and will be expanded upon later; all other interconnection scenarios, iii) through xii) are faulty. iii) corresponds to Case 9; iv) to Case 16, v) to Case 7 or 15, vi) through xi) to Case 18, and xii) to Case 12 or 18. For i), the dual-rail output of PCHB $gate_i$ could be correctly connected to the data inputs of PCHB $gates_j$, or could be incorrectly connected via a swapped rail connection (Case 2), being an input to a wrong PCHB gate (Case 1 or 6), or only being a partial input to a PCHB gate (Case 3 or 4). For ii), the dual-rail output of PCHB $gate_i$ could be correctly connected to an external data output, or could be incorrectly connected via a swapped rail connection (Case 2), being connected to the wrong external data output (Case 1), or only being partially connected to the external data output (Case 3 or 4).

Considering all possibilities, the *Lack* output of PCHB $gate_i$ has the following twelve interconnection scenarios: it could be connected to: i) *Rack* input of other PCHB $gates_j$, where *j*

$\neq i$, ii) input of one or more C-elements, iii) external *Lack* output, iv) *Rack* input of other PCHB gates, including $gate_i$, v) dual-rail input of a PCHB gate, vi) external data output, vii) PCHB gate data output, viii) another PCHB gate *Lack* output, ix) a C-element output, x) external data input, xi) external *Rack* input, or xii) external *reset* input. Of these, only i), ii), and iii) are possibly correct, and will be expanded upon later; all other interconnection scenarios, iv) through xii) are faulty. iv) corresponds to Case 9; v) and vi) to Case 5, and vii) through xii) to Case 18. For i), the *Lack* output of PCHB $gate_i$ could be correctly connected to the *Rack* input of other PCHB gates, or could be incorrectly connected by being the *Rack* input to a PCHB gate whose data output was not an input to $gate_i$ (Case 10). For ii), the *Lack* output of PCHB $gate_i$ could be correctly connected to C-element input(s), or could be incorrectly connected by being an input to a C-element structure that outputs the *Rack* input for a PCHB gate whose data output was not an input to $gate_i$ (Case 11). For iii), the *Lack* output of PCHB $gate_i$ could be correctly connected to the external *Lack* output, or could be incorrectly connected if the external data inputs are connected to PCHB gates other than $gate_i$ (Case 12).

Considering all possibilities, the output of C-element$_i$ has the following twelve interconnection scenarios: it could be connected to: i) input(s) of other C-elements$_j$, where $j \neq i$, ii) external *Lack* output, iii) *Rack* input of a PCHB gate, iv) input(s) of C-elements, including C-element$_i$, v) dual-rail input of a PCHB gate, vi) external data output, vii) PCHB gate data output, viii) PCHB gate *Lack* output, ix) another C-element output, x) external data input, xi) external *Rack* input, or xii) external *reset* input. Of these, only i), ii), and iii) are possibly correct, and will be expanded upon later; all other interconnection scenarios, iv) through xii) are faulty. iv) corresponds to Case 17; v) to Case 5 or 14, vi) to Case 5, and vii) through xii) to Case 18. For i), the output of C-elementi could be correctly connected to other C-element inputs, or could be incorrectly connected by being part of the C-element structure that produces the *Rack* input for PCHB *gate$_k$*, where PCHB *gate$_k$'s* data output was not an input to all PCHB gates whose *Lack* outputs are inputs to the C-element structure containing C-element$_i$ (Case 11), or by being connected to a C-element input within the same C-element structure, forming a feedback loop within the C-element structure (Case 17). For ii), the output of C-element$_i$ could be correctly connected to the external *Lack* output, or could be incorrectly connected if the external data inputs are connected to PCHB gates other than those whose *Lack* outputs are the inputs to the C-element structure containing C-element$_i$ (Case 12). For iii), the output of C-element$_i$ could be correctly connected to the *Rack* input of a PCHB gate, or could be incorrectly connected by being the *Rack* input to a PCHB gate whose data output was not an input to all PCHB gates whose *Lack* outputs are inputs to the C-element structure containing C-element$_i$ (Case 11).

Each external data input is treated similarly to a PCHB gate data output. Considering all possibilities, an external data input has the following eleven interconnection scenarios: it could be connected to: i) the dual-rail input of one or more PCHB gates, ii) external data output, iii) *Rack* input of a PCHB gate, iv) C-element input, v) another external data input, vi) PCHB gate data output, vii) a PCHB gate *Lack* output, viii) a C-element output, ix) external *Rack* input, x) external *reset* input, or xi) external *Lack* output. Of these, only i) and ii) are possibly correct, and will be expanded upon later; all other interconnection scenarios, iii) through xi) are faulty. iii) corresponds to Case 16, iv) to Case 7 or 15, v) through x) to Case 18, and xi) to Case 12 or 18. For i), the external data input could be correctly connected to the data inputs of PCHB gates, or could be incorrectly connected via a swapped rail connection (Case 2), being an input to a wrong PCHB gate (Case 1 or 6), or only being a partial input to a PCHB gate (Case 3 or 4). For ii), the external data input could be correctly connected to an external data output, or could be incorrectly connected via a swapped rail connection (Case 2), being connected to the wrong external data output (Case 1), or only being partially connected to the external data output (Case 3 or 4).

The external *Rack* input is treated similarly to a PCHB gate *Lack* output. Considering all possibilities, the external *Rack* input has the following ten interconnection scenarios: it could be connected to: i) *Rack* input of one or more PCHB gates, ii) input of one or more C-elements, iii) external *Lack* output, iv) dual-rail input of a PCHB gate, v) external data output, vi) PCHB gate data output, vii) PCHB gate *Lack* output, viii) a C-element output, ix) external data input, or x) external *reset* input. Of these, only i) and ii) are possibly correct, and will be expanded upon later; all other interconnection scenarios, iii) through x) are faulty. iii) corresponds to Case 12 or 18; iv) and v) to Case 5, and vi) through x) to Case 18. For i), the external *Rack* input could be correctly connected to the *Rack* input of PCHB gates, or could be incorrectly connected by being the *Rack* input to a PCHB gate whose data output was not an external data output (Case 11), or by not being connected to the *Rack* input of a PCHB gate whose output is an external data output (Case 13). For ii), the external *Rack* input could be correctly connected to one or more C-element inputs, or could be incorrectly connected by being an input to a C-element structure that outputs the *Rack* input for a PCHB gate whose data output is not an external output (Case 11), or by not being connected to a C-element input that is part of a C-element structure that generates the *Rack* input of a PCHB gate whose output is an external data output (Case 13).

Considering all possibilities, the external *reset* input has the following eleven interconnection scenarios: it could be connected to: i) the *reset* input of one or more PCHB gates, ii) the data input of a PCHB gate, iii) external data output, iv) *Rack* input of a PCHB gate, v) C-element input, vi) external data input, vii) PCHB gate data output, viii) PCHB gate *Lack* output,

ix) C-element output, x) external *Rack* input, or xi) external *Lack* output. Of these, only i) is possibly correct, and will be expanded upon later; all other interconnection scenarios, ii) through xi) are faulty. ii) through v) correspond to Case 8, and vi) through xi) to Case 18. For i), the external *reset* input is correctly connected only if connected to every PCHB gate's *reset* input. Additionally, the external *reset* input is used to initialize the PCHB circuit, which must be reset to a live state and match the reset state of its corresponding Boolean/synchronous specification circuit, both covered by Case 8.

In summary, the above exhaustive intersection of all possible interconnection combinations of PCHB gates, C-elements, and circuit inputs and outputs, proves that every possible faulty connection maps to at least one of the 18 cases presented in this section, thereby proving that these 18 cases do indeed comprise all possible faults that could occur in a PCHB circuit.

3.2.2. Equivalence Verification of Combinational PCHB Circuits

The developed methodology includes an equivalence verification scheme that verifies the functionality of a combinational PCHB circuit against its respective Boolean specification to ensure safety, and a graph-based approach to ensure liveness and handshaking correctness, both described below. The ability of the proposed methodology to detect all possible faults is addressed in Section 3.2.4.

3.2.2.1. Safety check of combinational PCHB circuits

The safety check requires two steps. First, a conversion algorithm takes the netlist of a combinational PCHB circuit as input and transforms that into a corresponding Boolean netlist. The generated Boolean circuit is then checked against the Boolean specification using an equivalence checker. To describe the methodology, the 2×2 PCHB multiplier, shown in Fig. 22, is used as an example. Note that although the PCHB multiplier is similar to a Boolean multiplier at the gate level, PCHB gate structures are far more complex. For example, a 2-input Boolean NAND gate only requires 4 transistors; whereas the 2-input reset-to-NULL PCHB NAND gate, shown in Fig. 5, requires 51 transistors to account for dual-rail signaling, registration, and handshaking control. In general, PCHB circuits require approximately 6-14 times more transistors than corresponding Boolean circuits due to their complex features, as can be seen in Table 4.

Fig. 23 shows the netlist format of the 2×2 PCHB multiplier. The first two lines correspond to all primary data inputs and outputs of the circuit, respectively. A dual-rail signal, *a0* is represented as *"a0_1a0_0",* where *a0_1* and *a0_0* are *rail*^{1} and *rail*^{0} of *a0*, respectively.

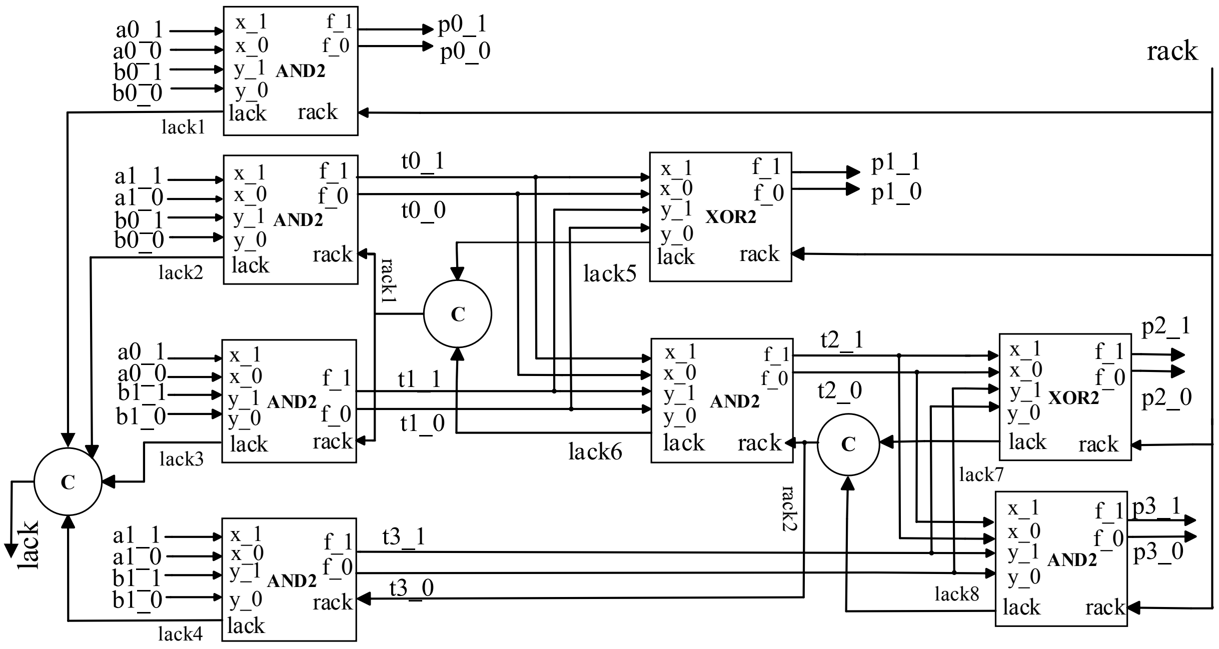


Figure 22. PCHB 2x2 Multiplier Circuit.

```
1. a0_1a0_0, a1_1a1_0, b0_1b0_0, b1_1b1_0
2. p0_1p0_0, p1_1p1_0, p2_1p2_0, p3_1p3_0
3. and2-N  1  a0_1a0_0,b0_1b0_0  rst  rack  lack1  p0_1p0_0
4. and2-N  1  a1_1a1_0,b0_1b0_0  rst  rack1  lack2  t0_1t0_0
5. and2-N  1  a0_1a0_0,b1_1b1_0  rst  rack1  lack3  t1_1t1_0
6. and2-N  1  a1_1a1_0,b1_1b1_0  rst  rack2  lack4  t3_1t3_0
7. xor2-N  2  t0_1t0_0,t1_1t1_0  rst  rack  lack5  p1_1p1_0
8. and2-N  2  t0_1t0_0,t1_1t1_0  rst  rack2  lack6  t2_1t2_0
9. xor2-N  3  t2_1t2_0,t3_1t3_0  rst  rack  lack7  p2_1p2_0
10. and2-N 3  t2_1t2_0,t3_1t3_0  rst  rack  lack8  p3_1p3_0
11. C2    lack5, lack6    rack1
12. C2    lack7, lack8    rack2
13. C4    lack1 ,lack2, lack3, lack4   lack
```

Figure 23. Netlist Structure of PCHB 2x2 Multiplier Circuit.

Lines 3 to 10 represent the individual PCHB gates used in the circuit. The first column of each of these lines represent the type of the gate in *"gate-reset_type"* format. As discussed previously, PCHB gates have state holding capability, and therefore must be initialized upon *reset*; *reset_type* denotes the *reset* value of the gate: NULL (N), DATA0 (D0), or DATA1 (D1). The number associated with the gate implies the number of gate inputs; e.g., *and2-N* represents a 2-input AND gate reset to NULL. The second column indicates the *level* of the gate, which is the longest path (in terms of number of PCHB gates) from the circuit's primary inputs to that gate's output. The remaining columns list the gate's data input(s), *reset* input, *Rack* input, *Lack* output, and data output(s), respectively. Type *Cn* in lines 11-13 represent an *n*-input C-element used to connect the PCHB handshake signals. Following *Cn* are its *n* inputs, and then its output.

The PCHB netlist is automatically converted into its corresponding Boolean netlist, shown in Fig. 24, using a developed algorithm. Each dual-rail signal, including the primary inputs/outputs, are replaced with a corresponding Boolean signal. A PCHB gate structure containing all information related to individual gates is created by traversing the netlist. A

Boolean gate structure is created by replacing each PCHB gate with its corresponding Boolean gate. Swapped rails of a dual-rail signal result in the introduction of an inverter. For example, if line 3 of Fig. 23 was instead *"and2-N 1 a0_0a0_1 ..."* this would result in the following additional line in Fig. 24: *"not 1 a0 a0_bar",* and line 3 of Fig. 24 to be changed to *"and2 2 a0_bar,b0 p0"*. Therefore, any bug causing unintended rail swap in the implementation will be detected, as the added inverter will result in functional inequivalence between the specification and PCHB implementation. If a PCHB gate input's $rail^1$ and $rail^0$ are not part of the same dual-rail signal, an error message is generated noting the misconnection between rails and where this occurs. Similarly, an error message is generated if any gate's data rails contain any handshaking signal(s) from other gates. A further check flags any bug that causes only one rail of a dual-rail signal to be connected to both rails of another dual-rail signal. And another check ensures that the circuit's external *reset* input is connected to all PCHB gates' *reset* input, and not connected to anything else, flagging any gate or connection that violates this.

```
1.  a0, a1, b0, b1
2.  p0, p1, p2, p3
3.  and2   1   a0,b0    p0
4.  and2   1   a1,b0    t0
5.  and2   1   a0,b1    t1
6.  and2   1   a1,b1    t3
7.  xor2   2   t0, t1   p1
8.  and2   2   t0, t1   t2
9.  xor2   3   t2, t3   p2
10. and2   3   t2, t3   p3
```

Figure 24. Converted Boolean Netlist Structure.

The converted Boolean netlist is then automatically encoded in the Satisfiability Modulo Theory Library (SMT-LIB) language, using Python, and is then input to an SMT solver to check for functional equivalence between the transformed Boolean version of the original PCHB circuit and its corresponding Boolean specification. For the 2×2 multiplier example, the SMT solver checks for the following safety property: $F_{PCHB\_Bool\_Equivalent}$ *(a0, a1, b0, b1)* = MUL *(a, b)*, where *(a1, a0)* and *(b1, b0)* are the (Most Significant Bit, Least Significant Bit) of *a* and *b*, respectively. We use the Z3 SMT solver [44] to check for equivalence verification, but any combinational equivalence checker could be used.

3.2.2.2. Liveness and handshaking correctness check

Liveness means absence of deadlock in a circuit. For combinational PCHB circuits, proper connections between handshaking signals ensures *liveness* and proper synchronization. The same PCHB netlist shown in Fig. 23, used as input for the safety check method, is also utilized as input for the liveness check to trace back the handshaking paths and C-element connections to verify proper handshaking, ensuring that every output generated by a particular input, acknowledges that input. Procedure 1 illustrates the algorithm that checks the handshaking connections.

A C-element structure and PCHB gate structure containing all information related to C-elements (i.e., C-element type, inputs, and outputs) and individual gates (i.e., gate type, level, data inputs, rack, lack, and outputs), respectively, is created by traversing the netlist (lines 1 and 2 in Procedure 1). For each PCHB gate, *i*, its output is compared with every other PCHB gate *j*'s inputs, $i \neq j$, to generate a fanout list, *fanout(i),* for PCHB gate *i* (line 3 in Procedure 1). For example, referring to Fig. 23, fanout for the *and2* gate on line 4 would contain the *xor2* gate on line 7 and the *and2* gate on line 8. For each PCHB gate, *i*, its *Rack* input is compared with every other PCHB gate *j*'s *Lack* output, $i \neq j$, and every C-element's output, to generate a completion fanin list, *comp_fanin(i),* for PCHB gate *i* (line 4 in Procedure 1). For example, referring to Fig. 23, *comp_fanin* for the *and2* gate on line 4 would contain the *xor2* gate on line 7 and the *and2* gate on line 8, since both of their *Lack* outputs are inputs to the C-element on line 11, whose output is the *Rack* input of the *and2* gate on line 4. Similarly, a *fanout* and *comp_fanin* list is generated for each external data input.

Procedure 1. Procedure to check handshaking connections

```
1: Create C_element_structure (PCHB_Netlist)
2: Create PCHB_gate_structure (PCHB_Netlist)
3: Create fanout (PCHB_gate_structure)
4: Create Comp_fanin (PCHB_gate_structure, C_element_structure)
5: for $i \leftarrow 1$ to num_pchb_gates do
6:     if fanout(i) = = Comp_fanin(i) then
7:         // no error
8:     else if $fanout(i) \subset Comp\_fanin(i)$ then
9:         for each variable v: $v \in Comp\_fanin(i)$ and $v \not\in fanout(i)$ do
10:            if $v.level \leq gate(i).level$ then
11:                report level error message
12:            else
13:                report warning message
14:            end if
15:        end for
16:    else
17:        report error message
18:    end if
19: end for
```

After *fanout* and *comp_fanin* for each PCHB gate and external data input is calculated, as shown in Fig. 25 for the 2×2 multiplier example, *fanout(k)* is checked to ensure that it is a subset of *comp_fanin(k),* for all PCHB gates and external data inputs (lines 5-19 in Procedure 1). Bit-wise completion results in *fanout(k)* being equal to *comp_fanin(k)*, while full-word completion results in *fanout(k)* being a proper subset of *comp_fanin(k)*, with the restriction that each gate that is in *comp_fanin(k)* and not in *fanout(k)* must be from the immediate subsequent level of gate/input *k*. *fanout(k)* not being a subset of *comp_fanin(k)* could result in deadlock, while *fanout(k)* being a proper subset of *comp_fanin(k)* but violating the level restriction described above, could either result in deadlock or may just decrease circuit performance. Hence, if *fanout(k)* is a proper subset of *comp_fanin(k)*, then each gate that is in *comp_fanin(k)* and not in *fanout(k)* is automatically inspected to ensure that it meets this level restriction. Even if the level restriction is met, a warning message is still generated to note the extra gate in the particular PCHB gate's *comp_fanin* list, to allow for easy manual inspection.

a0:	fanout: [1 3]	comp_fanin: [1 2 3 4]
a1:	fanout: [2 4]	comp_fanin: [1 2 3 4]
b0:	fanout: [1 2]	comp_fanin: [1 2 3 4]
b1:	fanout: [3 4]	comp_fanin: [1 2 3 4]
1:	fanout: 0	comp_fanin: 0
2:	fanout: [5 6]	comp_fanin: [5 6]
3:	fanout: [5 6]	comp_fanin: [5 6]
4:	fanout: [7 8]	comp_fanin: [7 8]
5:	fanout: 0	comp_fanin: 0
6:	fanout: [7 8]	comp_fanin: [7 8]
7:	fanout: 0	comp_fanin:0
8:	fanout: 0	comp_fanin:0

Figure 25. *Fan_out* and *Comp_fanin* Structure.

Additional checks ensure correct connection of the external *Rack* input, and proper generation of the external *Lack* output. The external *Rack* signal should be connected to the *rack* inputs of all gates that produce the circuit's external data outputs. Similarly, the *lack* outputs of all gates that take primary data inputs as their inputs should be conjoined via a C-element structure to generate the external *Lack* output. The developed algorithm generates an appropriate descriptive error message in case the PCHB circuit fails to satisfy any of these checks. Furthermore, it checks to ensure that no data signal is part of the handshaking connections, and that no handshaking signal is part of a data signal. All these checks are performed during the process of creating the PCHB gate and C-element structures (lines 1 and 2 in Procedure 1). Note that *fanout* 0 indicates an external output, while *comp_fanin* 0 denotes an external *Rack* input. The running time for this liveness check algorithm is $O\ (I+P)*(P+C)$, where I, P, and C are the number of external inputs, PCHB gates, and C-elements in the circuit, respectively.

3.2.2.3. Results

This equivalence verification methodology for combinational PCHB circuits has been demonstrated on several multipliers and ISCAS-85 [45] combinational circuit benchmarks; and the verification times are compared with the previous model checking based PCHB formal verification methodology, as discussed in Section 3.1. As shown in Table 4, the equivalence verification methodology presented herein is significantly faster than the model checking based approach for every circuit. Furthermore, this methodology was able to verify complex circuits with hundreds of gates, such as a 12×12 multiplier; whereas the model checking approach Timed Out (TO) for much smaller circuits, demonstrating the scalability of this approach. Note that DNS in Table 4 stands for "Did Not Simulate"; since the model checking approach Timed Out for a 4×4 multiplier, we can safely assume that it would time out for more complex circuits, such as ISCAS c432 and other higher order multipliers. 10x10Mul-B1, 10x10Mul-B2, 10x10Mul-B3, and 10x10Mul-B4 are some of the buggy circuits tested. In 10x10Mul-B1, a bug was introduced in the data signals by incorrectly connecting one gate's dual-rail input to another dual-rail signal. 10x10Mul-B2 represents a logic element bug, where a PCHB AND gate was replaced with a PCHB NAND gate. 10x10Mul-B3 represents a handshaking connection bug, where the *Lack* output from PCHB gate *i* was not included in the C-element structure that generated the *Rack* input to PCHB gate *j*, even though PCHB gate *j*'s data output was an input to PCHB gate *i*. 10x10Mul-B4 swaps the rails of one PCHB gate's input. In these, and every other buggy case tested, the proposed methodology detected and identified each bug very fast. Verification was performed using Z3 SMT solver [44] on an Intel® Core™ i7-4790 CPU with 32GB of RAM, running at 3.60 GHz. The verification times in Table 4 only include the Z3 runtime, as the netlist conversion times and time required to verify the handshaking signals were negligible in comparison.

Table 4. Verification Results for Various Combinational PCHB Circuits Based on Equivalence Checking.

PCHB Circuits	**# Transistors in Boolean Circuit**	**# Transistors in PCHB Implementation**	**Proposed Method Time (sec)**	**Model Checking Time (sec)**
Full Adder	36	266	<0.01	21.54
ISCAS C-17	24	320	0.01	138.62
4-to-16 decoder	136	1,284	0.01	222.6
32- bit MUX	386	4,836	0.23	413.20
4x4 Multiplier	456	3,324	0.06	**TO**
8x8 Multiplier	2,256	16,620	12.18	**DNS**
10x10 Multiplier	3,660	26,960	741.84	**DNS**
11x11 Multiplier	4,488	33,070	7323.07	**DNS**
12x12 Multiplier	5,400	39,812	62,823.67	**DNS**
10x10Mul-B1	3,660	26,960	1.01	**DNS**
10x10Mul-B2	3,660	26,960	0.06	**DNS**
10x10Mul-B3	3,660	26,960	0.84	**DNS**
10x10Mul-B4	3,660	26,960	0.79	**DNS**
ISCAS c432	816	7,362	1.31	**DNS**

3.2.3. Equivalence Verification of Sequential PCHB Circuits

As described in the section above, the equivalence verification methodology proved to be a much faster and more scalable approach for combinational PCHB circuits, compared to the previous model checking method. Hence, in this section the approach is extended to the verification of sequential PCHB circuits, which is far more complex due to datapath feedback.

The verification procedure requires three steps. In the first step, a sequential PCHB circuit is converted to an equivalent synchronous circuit. We utilize the theory of WEB-refinement [34] to compare the synchronous netlist generated from the PCHB circuit with the original synchronous specification, as the notion of correctness. The major advantage of applying WEB-refinement to the generated equivalent synchronous circuit instead of the actual PCHB circuit is that a synchronous circuit is much more deterministic compared to its PCHB equivalent, which makes the verification time much faster. The generated synchronous circuit, specification synchronous circuit, and the WEB-refinement property are automatically encoded in the SMT-LIB language. The resulting equivalence property is then checked using an SMT solver. In the second step, we check the handshaking connections between components, which is similar to the combinational PCHB handshaking check, discussed in Section 3.2.2.2. The third step consists of applying the method developed by Shih et. al. for deadlock verification of sequential PCHB circuits. Since this third step is fully detailed in [39], it is not discussed further herein, besides its brief overview in Section 2.4.

To describe the methodology, a Multiply and Accumulate (MAC) unit is used as an example circuit. Fig. 26 shows a synchronous MAC, where $A' = A + X \times Y$; and Fig. 27 shows the equivalent PCHB version. Two registers are shown in the PCHB version such that all feedback loops contain at least 3 registers to avoid deadlock, since PCHB gates themselves act as latches and the PCHB C/L contains at least one gate in every feedback path. However, some of the feedback paths only require one register, as shown in Fig. 28, to be discussed later. Although the synchronous and PCHB MACs seem similar, they are structurally very different. Synchronous registers are clocked, whereas alternating DATA/NULL transitions in PCHB are maintained via C-elements and a well-defined handshaking scheme.

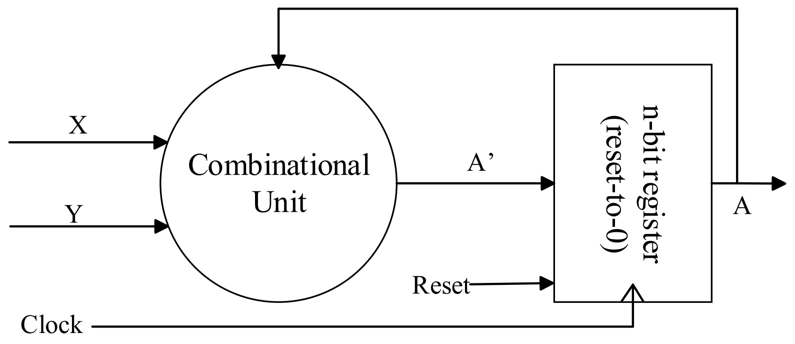


Figure 26. Synchronous MAC Structure.

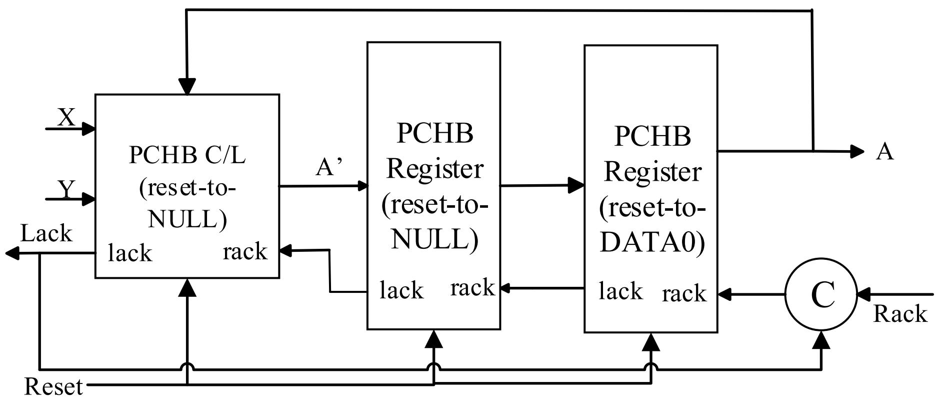


Figure 27. PCHB MAC Structure.

3.2.3.1. Safety check for sequential PCHB circuits

Fig. 28 shows the datapath connection (without handshaking signals) diagram of a 4 + 2×2 PCHB MAC. (X_1, X_0) and (Y_1, Y_0) are the two bits of inputs X and Y, respectively. The product of X and Y is added with the 4-bit accumulator output, A, where A_3 and A_0 are the MSB and LSB, respectively. Each input and output are dual-rail signals. *HA* and *FA* are the PCHB half-adder and full-adder components [46], respectively. The highlighted components in Fig. 28 are reset-to-NULL/DATA0 PCHB registers. Fig. 29 shows the handshaking connection between components for the *4 + 2x2* PCHB MAC, which will be discussed in the next section.

There are four feedback loops (*FL*) in Fig 28, utilizing different register arrangements to better illustrate the verification approach for different cases. The first loop, *FL1*, contains one reset-to-NULL HA component (*HA_N*) followed by two reset-to-DATA0 (*Reg_D0*) registers; a *Reg_D0* register preceded by 2 *HA_N* components and 2 reset-to-NULL PCHB FA (*FA_N*) components are utilized in *FL2* and *FL3*, respectively, to meet the three register minimum requirement for the one DATA token in each; and *FL4* contains a reset-to-DATA0 PCHB component (*FAs_D0)* preceded by *Reg_N* and *Reg_D0* components.

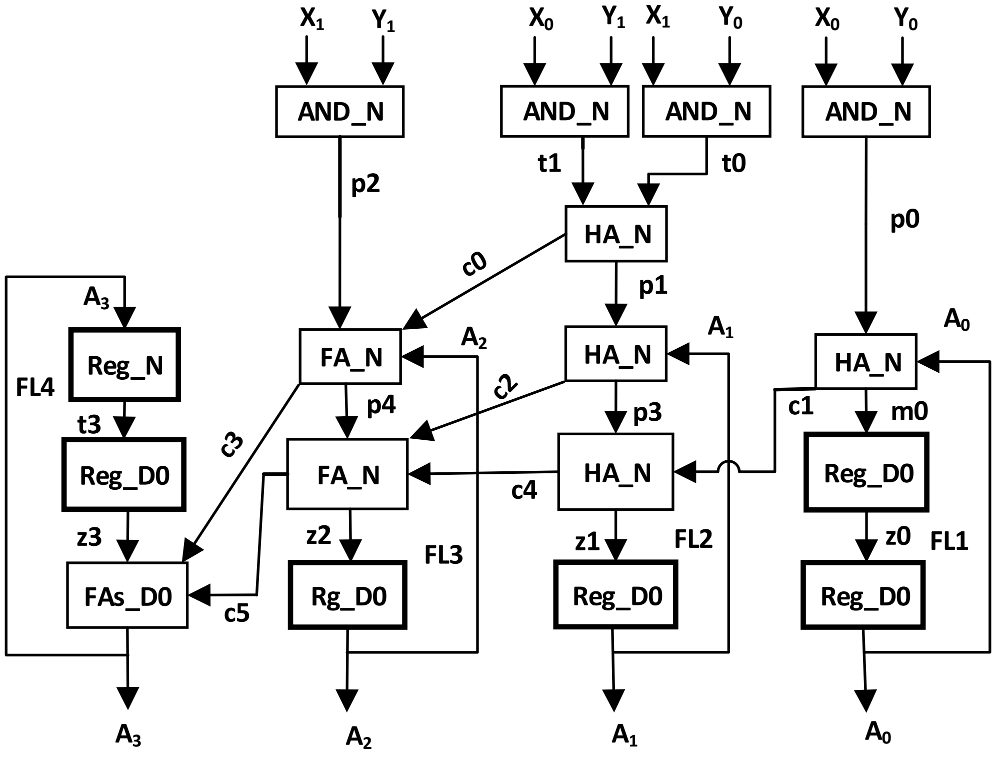


Figure 28. PCHB 4 + 2x2 MAC Datapath Connection.

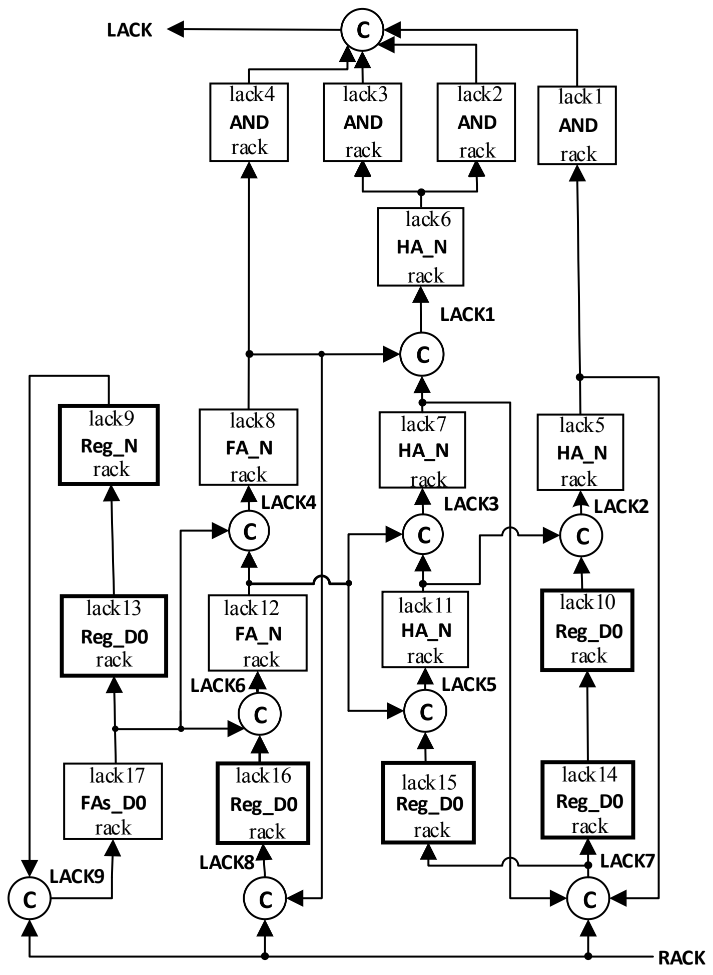


Figure 29. PCHB 4 + 2x2 MAC Handshaking Connection.

Fig. 30 shows the netlist of the PCHB 4+2×2 MAC, following the same structure as described in Section 3.2.2.1. The first 2 lines are the circuit inputs and outputs, respectively; lines 3-13 are the PCHB C/L gates; lines 14-19 are the PCHB registers; and lines 20-29 are C-elements used in the handshaking network. Note that the *HA* and *FA* components each contain two outputs, *sum* followed by *carry. FAs_D0* is a modified *FA* component with only *sum* output.

This sequential PCHB netlist is then automatically converted into its equivalent synchronous netlist, depicted in Fig. 31, similar to the conversion process described in Section 3.2.2.1 for combinational PCHB circuits. Each dual-rail signal is replaced with a corresponding Boolean signal; and all handshaking signals and C-elements are eliminated.

```
1. x0_1x0_0,x1_1x1_0,y0_1y0_0,y1_1y1_0
2. a0_1a0_0,a1_1a1_0,a2_1a2_0,a3_1a3_0
3. and2-N  1  x0_1x0_0,y0_1y0_0  reset  lack5  lack1  p0_1p0_0
4. and2-N  1  x1_1x1_0,y0_1y0_0  reset  lack6  lack2  t0_1t0_0
5. and2-N  1  x0_1x0_0,y1_1y1_0  reset  lack6  lack3  t1_1t1_0
6. and2-N  1  x1_1x1_0,y1_1y1_0  reset  lack8  lack4  p2_1p2_0
7. ha2-N  2  p0_1p0_0,a0_1a0_0   reset  LACK2  lack5  m0_1m0_0,c1_1c1_0
8. ha2-N  2  t0_1t0_0,t1_1t1_0    reset  LACK1  lack6  p1_1p1_0,c0_1c0_0
9. ha2-N  3  p1_1p1_0,a1_1a1_0   reset  LACK3  lack7  p3_1p3_0,c2_1c2_0
10. fa3-N  3  p2_1p2_0,a2_1a2_0,c0_1c0_0  reset  LACK4  lack8  p4_1p4_0,c3_1c3_0
11. ha2-N  4  p3_1p3_0,c1_1c1_0  reset  LACK5  lack11  z1_1z1_0,c4_1c4_0
12. fa3-N  5  p4_1p4_0,c2_1c2_0,c4_1c4_0  reset  LACK6  lack12  z2_1z2_0,c5_1c5_0
13. fas3-D0  6  z3_1z3_0,c3_1c3_0,c5_1c5_0  reset  LACK9  lack17  a3_1a3_0
14. Reg_D0  m0_1m0_0  reset   lack14    lack10    z0_1z0_0
15. Reg_D0  z0_1z0_0    reset    LACK7     lack14    a0_1a0_0
16. Reg_D0  z1_1z1_0    reset    LACK7     lack15    a1_1a1_0
17 .Reg_D0  z2_1z2_0    reset    LACK8     lack16    a2_1a2_0
18. Reg_N   a3_1a3_0    reset    lack13    lack9     t3_1t3_0
19. Reg_D0  t3_1t3_0    reset    lack17    lack13    z3_1z3_0
20. C2  lack7,lack8  LACK1
21. C2  lack10,lack11  LACK2
22. C2  lack11,lack12  LACK3
23. C2  lack12,lack17  LACK4
24. C2  lack15,lack12  LACK5
25. C2  lack16,lack17  LACK6
26. C3  RACK,lack5,lack7  LACK7
27. C2  RACK,lack8  LACK8
28. C2  RACK,lack9  LACK9
29. C4  lack1,lack2,lack3,lack4  LACK
```

Figure 30. PCHB 4 + 2x2 MAC Unit Netlist.

```
1.   x0,x1,y0,y1
2.   a0,a1,a2,a3
3.   and2  1   x0,y0   p0
4.   and2  1   x1,y0   t0
5.   and2  1   x0,y1   t1
6.   and2  1   x1,y1   p2
7.   ha2   2   p0,a0   m0,c1
8.   ha2   2   t0,t1    p1,c0
9.   ha2   3   p1,a1    p3,c2
10. fa3    3   p2,a2,c0  p4,c3
11. ha2    4   p3,c1   z1,c4
12. fa3    5   p4,c2,c4   z2,c5
13. fas3   6    a3,c3,c5    r1
14. Reg_0       m0    a0
15. Reg_0       z1     a1
16. Reg_0       z2     a2
17. Reg_0       r1     a3
```

Figure 31. Equivalent Boolean Converted Netlist.

Procedure 2 illustrates the conversion algorithm for PCHB components (i.e., gates and registers). A PCHB component structure, *PCHB_comp*, containing all information related to individual components (i.e., gate type, reset type, data inputs, reset input, rack, lack, and data outputs) is created for each component by traversing the netlist (line 1 in Procedure 2).

Procedure 2. Procedure to generate synchronous circuit from PCHB circuit

```
1: Create PCHB_comp (PCHB_Netlist)
2: for i←1 to num_pchb_components do
3:     if PCHB_comp(i).gate_type != Reg then
4:         if PCHB_comp(i).reset_type == D0 or D1 then
5:             for j←1 to num_outputs do
6:                 if PCHB_comp(i).output(j) connected to rst-to-DATA comp then
7:                     report warning message
8:                 else
9:                     Add PCHB_comp Register after PCHB_comp(i).output(j)
10:                end if
11:            end for
12:        end if
13:    else if PCHB_comp(i).reset_type == N then // PCHB_comp(i) is Reg_N
14:            merge PCHB gates separated by PCHB_comp(i)
15:            delete PCHB_comp(i)
16:    else if PCHB_comp(i).output(1) connected to rst-to-DATA comp then
17:            report warning message
18:            merge PCHB gates separated by PCHB_comp(i)
19:            delete PCHB_comp(i)
20:    end if
21: end for
22: for i←1 to num_pchb_components do
23:        convert_to_synchronous (PCHB_comp(i))
24: end for
```

Each PCHB gate/component (e.g., AND gate or FA component), excluding registers, is replaced with its corresponding Boolean gate/component, which does not include a *reset* input. Every reset-to-NULL register, *Reg_N*, is eliminated by setting its output equal to its input, as those are added in feedback loops to avoid deadlock or to increase throughput via slack matching [47], and have no corresponding functionality in the equivalent synchronous circuit. Each reset-to-DATA register is replaced with its corresponding resettable synchronous register, except for paths where its output is input to another reset-to-DATA register/component, in which case it's eliminated like a reset-to-NULL register described previously, since adjacent reset-to-DATA registers/components only represent a single DATA token, which corresponds to a single synchronous register. In addition to each reset-to-DATA PCHB gate/component being replaced with its corresponding Boolean gate/component, as previously mentioned, an additional resettable synchronous register is also added following its output(s), except again for paths where an output is input to another reset-to-DATA register/component, in which case no synchronous register is added. Referring to Procedure 2, lines 2-21 add and delete PCHB register components, according to the description above, such that after exiting that for loop, any remaining PCHB register component directly corresponds to a synchronous register. Lines 22-24 then replace each PCHB component with its corresponding synchronous/Boolean component. For example, referring to Fig. 28, the two adjacent Reg_D0 components in FL1 are replaced with a single synchronous Reg_0 component, shown on line 14 in Fig. 31; the Reg_D0 components in FL2 and FL3 are each replaced with a single synchronous Reg_0 component, shown on line 15 and 16 in Fig. 31, respectively; and in FL4, the FAs_D0 component is replaced by a Boolean full adder followed by a Reg_0 component, shown on lines 13 and 17 in Fig. 31, respectively, while the preceding *Reg_N* and *Reg_D* components are eliminated. The equivalent converted synchronous circuit is depicted in Fig. 32.

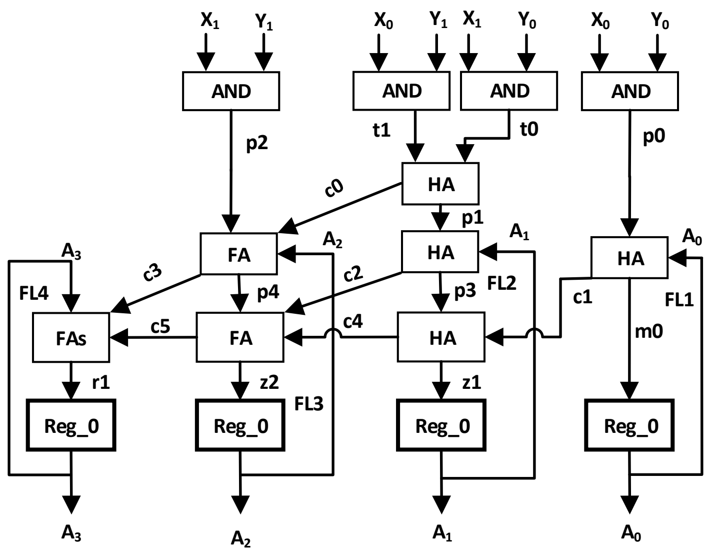


Figure 32. Equivalent Synchronous Circuit after Conversion.

While consecutive reset-to-DATA registers/components are not necessarily incorrect, as discussed in the example above, typically only a single reset-to-DATA register/component is utilized to represent a DATA token. Therefore, our conversion tool generates a warning message that flags all adjacent reset-to-DATA components for manual inspection (lines 7 and 17 in Procedure 2), since this could cause deadlock upon reset in some cases without violating our safety or handshaking checks. For example, replacing the middle FA_N with FA_D0 and swapping it with the last Reg_D0 in Fig. 28 FL3, will result in the circuit snippet shown in Fig. 33. Converting this circuit results in the same converted netlist as Fig. 32; however, this reset state will result in deadlock (i.e., *LACK8* will never become initialized, as can be seen from the data and handshaking signal transitions in Fig. 33). Even if the C-element that produces *LACK8* was replaced with a resettable version to initialize *LACK8*, the circuit will still deadlock upon reset. For example, if *LACK8* was reset to 0, *A2* would transition to NULL causing the sum output of FA_N in FL3, *p4*, to never become DATA (i.e., the initial *A2* data would be lost before being fed back). If instead *LACK8* was reset to 1, the *c5* input to FAs_D0 would never become NULL and therefore its *lack17* output would never become 1. If one transitioned the other would be able to transition; however, neither can transition without the other transitioning first. This type of reset state error is easily detected in simulation, since the circuit will deadlock after reset before completing its first DATA/NULL cycle. For our example in Fig. 28, the two Reg_D0 components in FL1 and the Reg_D0 and FAs_D0 components in FL4 would be flagged for manual inspection.

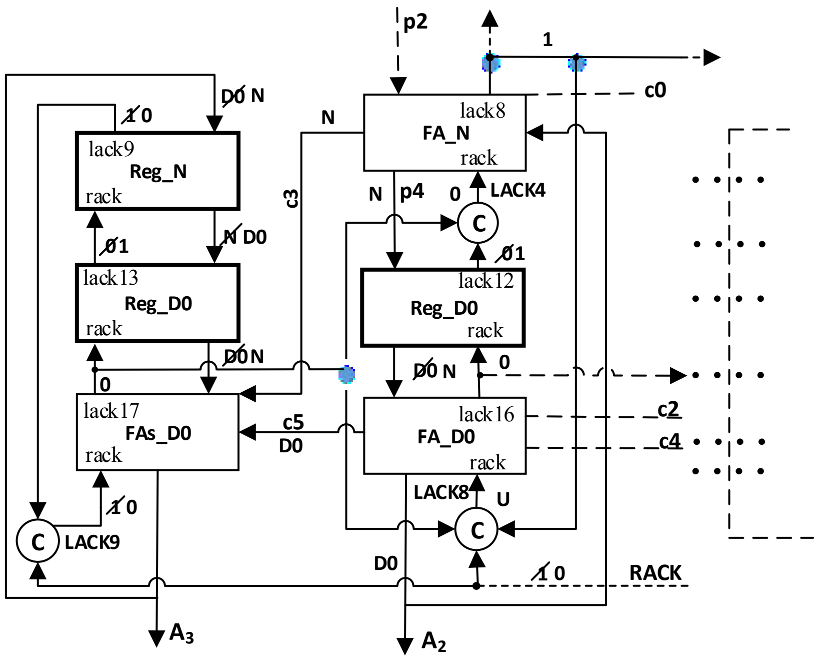


Figure 33. Alternate Arrangement in FL3 and FL4 to Demonstrate Deadlock.

Functional equivalence checking of sequential PCHB circuits is more complicated than for combinational PCHB circuits described in Section 3.2.2.1. Sequential circuits require states and transitions between the states, such that both specification and implementation can be modeled as a transition system (TS). These TSs are then checked for equivalence using the theory of WEB refinement [34], which utilizes two functions, *rank* and *refinement-map*, to distinguish between finite stuttering and infinite stuttering (deadlock) states, and to map implementation states to corresponding specification states, respectively. However, since the registers of the specification sequential circuit, *SPEC_SEQ*, and those of the equivalent synchronous circuit, *PCHB_SEQ*, automatically generated from the PCHB circuit as described above, have a one-to-one mapping, there is no stuttering. Moreover, the *refinement-map* function is just a projection of each implementation stage register to its corresponding specification stage register. Hence, the correctness proof obligation can be reduced to Proof Obligation 1, shown below.

Proof Obligation 1 *:* $\{\forall s :: s \in S_{PCHB\_SEQ} ::$

$[u = Reg\_Proj(s) \wedge w = Step_{PCHB\_SEQ}(s) \wedge \quad v = Step_{SPEC\_SEQ}(u)]$

$\Rightarrow w = v\}$

Fig. 34 is used to explain the proof obligation. *s* is a state of *PCHB_SEQ*, and *u* is a *SPEC_SEQ* state, which includes a projection of register values of state *S*. $Step_{PCHB\_SEQ}$ and *StepSPEC_SEQ* are two functions that step *PCHB_SEQ* and *SPEC_SEQ* once, respectively. *w* and *v* are the next states of *s* and *u* respectively. The proof obligation states that *PCHB_SEQ* and *SPEC_SEQ* are equivalent iff the projection values of all registers in state *w* equal the values of their corresponding registers in state *v*. The proof obligation is encoded in SMT-LIB, and checked using an SMT solver.

Additionally, since *PCHB_SEQ* and *SPEC_SEQ* have a one-to-one register mapping, the register reset values are compared to ensure that the two circuits have the same initial reset state. If a one-to-one register mapping does not exist, this is an error; for example, a PCHB register being reset to NULL instead of DATA, as it was supposed to be, could result in 1 fewer register in *PCHB_SEQ*.

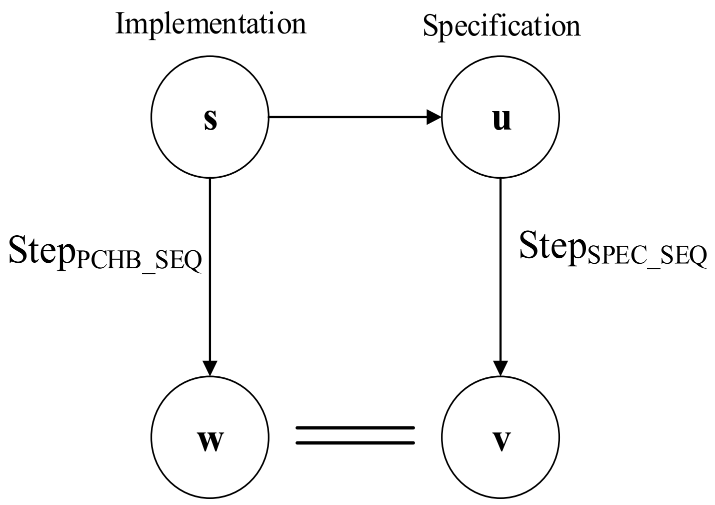


Figure 34. Formulation of Proof Obligation to Check Equivalence of PCHB_SEQ and SPEC_SEQ.

3.2.3.2. Liveness and handshaking correctness check

Fig. 29 shows the handshaking connections between components for the *4+2×2* PCHB MAC, where *RACK* and *LACK* are the external *request* and *acknowledge* signals, respectively. The same handshaking check algorithm detailed in Section 3.2.2.2 is utilized to check the handshaking connections, with the only difference being the level restriction check for full-word completion (i.e., each gate that is in *comp_fanin(k)* and not in *fanout(k)* must be from the immediate subsequent level of gate/input *k*), since this level restriction no longer holds due to datapath feedback. Therefore, we do not require *level* for PCHB register components; and *level* for other PCHB gates is ignored for sequential circuits. To demonstrate this, we have utilized partial full-word completion in Fig. 29, by combining the request generation of the two LSB registers. The *comp_fanin* list of the 2 LSB Reg_D0 components will contain one additional signal each, *lack7* for the LSB register in *FL1* and *lack5* for the next LSB register in *FL2*. This is not an error, but may slow down the circuit. The handshaking algorithm generates a warning under such scenario, highlighting the additional signal for further inspection.

Following the above handshaking check, the method developed by Shih et. al. for deadlock verification [39] is applied, as mentioned at the beginning of Section 3.2.3. This method, along with our handshaking correctness check, guarantees liveness for sequential PCHB circuits.

3.2.4. Detection of All Possible Faults

Section 3.2.1 enumerates the faults that could occur in a PCHB circuit comprised solely of PCHB gates and C-elements (i.e., no special asynchronous control elements, such as F-element, D-element, Tangram S-element, etc.), and proves that the 18 faults listed comprise all possible faults. Below, we show how the proposed methodology detects all 18 of these faults. *Cases 1-8* correspond to datapath faults, which are detected in our *safety* check; *Cases 9-17* correspond to handshaking faults, which are detected in our *liveness* check; and *Case 18* corresponds to electrical faults, which can occur in either the datapath or handshaking circuitry, and are detected in our *safety* or *liveness* check, respectively.

Case 1: Faulty data connection, *Case 2: Swapped dual-rail connection*, *and Case 6: Incorrect logic implementation*, would all result in functional inequivalence between implementation and specification, and would be detected by the SMT solver. *Case 3: Rails from different signals*, *Case 4: Rail Duplication*, *Case 5: Handshaking signal connected to data signal*, *and Case 7: Non-PCHB gate in datapath*, would all be detected in the PCHB-to-Boolean netlist conversion algorithm, described in Sections 3.2.2.1 and 3.2.3.1. For *Case 8: Incorrect reset*, the external *reset* input not being connected to all PCHB gates' *reset* input would be detected in the PCHB-to-Boolean netlist conversion algorithm, while an incorrect *reset* value would be detected in the register *reset* value comparison check described at the end of Section 3.2.3.1.

Case 9: Insufficient registers in a feedback loop, would be detected by the verification procedure proposed by Shih et. al. [39], which follows our handshaking check, as described in Section 3.2.3.2. *Cases 10-13: Missing handshaking signal*, *Additional handshaking signal*, *External Lack error*, *and External Rack error*, would all be detected in the handshaking correctness check, described in Sections 3.2.2.2 and 3.2.3.2. *Cases 14-17: Non-C-element in handshaking circuitry*, *Data signal input to C-element*, *Data signal input to PCHB gate Rack input*, and *C-element structure feedback*, would all be detected by the algorithm that generates the *fanout* and *comp_fanin* for each PCHB gate and external data input, described in Section 3.2.2.2.

Case 18: Shorted output, would be detected by the PCHB-to-Boolean netlist conversion algorithm for shorted PCHB gate data outputs, and by the *fanout/comp_fanin* generation algorithm for shorted PCHB *Lack* or C-element outputs. Hence, our proposed methodology will detect all 18 fault cases, which in Section 3.2.1 were proved to comprise all possible PCHB circuit faults; therefore, our proposed methodology guarantees full functional correctness for any PCHB circuit composed solely of PCHB gates and C-elements (i.e., no special asynchronous control elements, such as F-element, D-element, Tangram S-element, etc.).

3.2.5. Results

The proposed methodology is demonstrated by verifying several different sized MACs and ISCAS sequential circuit benchmarks [48], as shown in Table 5, which lists the verification time for each circuit. MAC circuits could be considered a special case, since they only contain non-interacting feedback loops; however, the ISCAS benchmarks are more general and contain various interacting feedback paths. PCHB MAC circuits are complex sequential pipeline structures, with each PCHB component acting as a state holding element. For example, an optimized 20+10×10 PCHB MAC requires 18,612 transistors, whereas its synchronous counterpart only requires 4,710 transistors, demonstrating the substantially increased complexity of PCHB circuits vs. their synchronous equivalent. The PCHB-to-Boolean netlist conversion time and the time to generate *fanout* and *comp_fanin* for each PCHB gate was negligible compared to the time to perform the safety check by the Z3 SMT solver [44]; additionally, the handshaking check was also negligible, as seen in Table 5. Note that Table 5 does not include the deadlock verification times, as this algorithm was developed by Shih et. al. and fully detailed in [39].

To check our methodology, we injected bugs into the 20+10×10 MAC, corresponding to all 18 fault cases, except for *Case 9: Insufficient registers in a feedback loop*, as the method to detect this fault was already developed by Shih et. al. in [39]. The –Bn multipliers in Table 5 are the buggy circuits, where *n* corresponds to a *Case n* bug; and the (B) in either the Safety Check or Handshaking Check column denotes which check detected the bug. -B8i corresponds to a register *reset* to an incorrect value, while B8ii corresponds to a signal other than the external *reset* being connected to a PCHB gate's *reset* input.

In every case, the proposed methodology detected the bug and produced a counterexample and/or descriptive error message, in order to assist in identifying where the error occurred. Verification was performed using the same computer described in Section 3.2.2.3.

Table 5. Verification Results for Various Sequential PCHB Circuits Based on Equivalence Checking.

PCHB Circuits	**Verification Time (in sec.)**			**PCHB Circuits**	**Verification Time (in sec.)**		
	Safety Check	**Hand-shaking Check**	**Total Time**		**Safety Check**	**Hand-shaking Check**	**Total Time**
4+2x2 MAC	0.01	0.0053	0.0153	20+10x10 MAC- B6	0.23 (B)	1.7862	2.0162
8+4x4 MAC	0.07	0.0278	0.0978	20+10x10 MAC- B7	0.0109 (B)	1.7862	1.7971
16+8x8 MAC	12.06	0.7893	12.8493	20+10x10 MAC- B8i	0.17 (B)	1.7862	1.9562
20+10x10 MAC	813.03	1.7862	814.8162	20+10x10 MAC- B8ii	0.0206 (B)	1.7862	1.8068
ISCAS s27	0.01	0.0010	0.0110	20+10x10 MAC- B10	813.03	3.0563 (B)	816.0863
ISCAS s208	0.17	0.0378	0.2078	20+10x10 MAC- B11	813.03	2.5998 (B)	815.6298
ISCAS s298	0.23	0.0459	0.2759	20+10x10 MAC- B12	813.03	1.8368 (B)	814.8668
ISCAS s444	0.76	0.1008	0.8608	20+10x10 MAC- B13	813.03	1.8989 (B)	814.9289
20+10x10 MAC- B1	0.11 (B)	1.7862	1.8962	20+10x10 MAC- B14	813.03	0.9063 (B)	813.9363
20+10x10 MAC- B2	0.14(B)	1.7862	1.9262	20+10x10 MAC- B15	813.03	1.1856 (B)	814.2156
20+10x10 MAC- B3	0.0324 (B)	1.7862	1.8186	20+10x10 MAC- B16	813.03	1.1121 (B)	814.1421
20+10x10 MAC- B4	0.0175 (B)	1.7862	1.8037	20+10x10 MAC- B17	813.03	0.8555(B)	813.8855
20+10x10 MAC- B5	0.0154 (B)	1.7862	1.8016	20+10x10 MAC- B18	0.1333 (B)	1.7862	1.9195

3.3. Conclusions

Formal verification methodologies for QDI PCHB paradigms have long been desired in industry. In this chapter, two developed formal verification methods for QDI PCHB circuits have been illustrated in details. The first method based on model checking is the first known formal verification method for PCHB circuits that checks for both safety and liveness. However, scalability is an issue with the approach. The second method is based on equivalence checking, which is one of the popular formal verification approaches that is highly scalable and efficient. This chapter discusses the first ever equivalence checking based formal verification methodology for QDI PCHB circuits. This second approach is scalable, fast, and applicable to any combinational or sequential PCHB circuit comprised entirely of PCHB gates and C-elements. We have demonstrated that the proposed approach detects all possible faults, thereby guaranteeing correctness, and have proved that the 18 fault cases presented herein comprise all faults that could possibly occur in a PCHB circuit comprised entirely of PCHB gates and C-elements.

4. FORMAL MODELING AND VERIFICATION METHODS FOR QDI NCL ASYNCHRONOUS CIRCUITS[1]

Vidura et al. [33] have previously developed an approach for verifying the equivalence of an NCL circuit against a synchronous circuit. They use the theory of Well-Founded Equivalence Bisimulation (WEB) refinement [34] as the notion of equivalence. In WEB refinement, both the circuit to be verified (here the NCL circuit) and the specification circuit (here the synchronous circuit) are modeled as transition systems (TSs), which capture the behavior of the circuit as a set of states and transitions between the states. WEB refinement essentially defines what it means for two TSs to be functionally equivalent. Their approach performs symbolic simulation on both the NCL circuit and the synchronous circuit to generate the TSs corresponding to both circuits. A decision procedure is then used to verify that the two TSs satisfy the WEB refinement property.

In working with the above approach, it is found that because NCL circuits exhibit highly non-deterministic behaviors, the corresponding TSs are very complex, even for relatively simple circuits. This complexity leads to two issues. First is state space explosion. Second, it becomes very difficult to compute the reachable states of the resulting TS. Computing reachable states is important because unreachable states often flag numerous spurious counterexamples that makes verification intractable.

[1]The functional equivalence check and invariant checks documented in this dissertation were a collaborative work between Ashiq Sakib, Son Le, Scott Smith, and Sudarshan Srinivasan. The conversion of NCL combinational circuits to equivalent Boolean circuits and conversion of NCL sequential circuits to equivalent synchronous circuits were done by Ashiq. Equivalence checking for combinational circuits was done by Ashiq. An automated tool to generate the initial equivalence proof for the sequential logic and proofs of sequential circuits was done by Son. For the invariant check, all combinational circuits were done by Ashiq and the sequential circuits were done by modifying Son's equivalence models. The handshaking check algorithms for both combinational and sequential NCL circuits were developed and implemented by Ashiq.

An alternate approach to circumvent having to deal with the NCL TS is discussed in this chapter. The strategy behind the methodology is similar to the equivalence verification method for PCHB circuits, as discussed in Section 3.2. The high-level idea is to perform structural transformation on the NCL circuit netlist to convert the NCL circuit into an equivalent synchronous circuit. The converted synchronous circuit is then compared against the specification synchronous circuit, using WEB refinement as the notion of correctness. The converted synchronous circuit, specification synchronous circuit, and the WEB refinement property are then automatically encoded in the Satisfiability Modulo Theory Library (SMT-Lib) language [49]. The resulting equivalence property is then checked using an SMT solver. Additional checks need to be performed to ensure that the NCL circuit is live (i.e., deadlock free). Thus, the overall verification has three high-level steps: (1) Conversion from NCL to synchronous; (2) Verification of converted synchronous against specification synchronous; (3) Additional checks on original NCL circuit to ensure liveness. The methodology can also be used to check the equivalence of two NCL circuits by applying the conversion technique to both NCL circuits to obtain two corresponding synchronous circuits, verifying these two synchronous circuits against each other, and performing the additional liveness checks on both NCL circuits.

4.1. Equivalence Verification for Combinational NCL Circuits

In industry, asynchronous NCL circuits are typically synthesized from their synchronous counterparts. Throughout the synthesis and optimization process, the synchronous specification undergoes several transformations, resulting in major structural differences between the implemented NCL circuit and its synchronous specification. For this kind of scenario, equivalence checking is a widely used formal verification method that checks for logical and functional equivalence between two different circuits.

NCL verification based on equivalence checking has proved to be a unified, fast, and scalable approach that eliminates most of the limiting factors of previous verification works in the field. The NCL equivalence verification method requires 5 steps, as described below and detailed in the following sub sections:

1) The netlist of an NCL circuit to be verified is converted into a corresponding Boolean/synchronous netlist, which is modeled in the SMT-Lib language using an automated script that we developed. The converted netlist is then checked against its corresponding Boolean/synchronous specification using an SMT solver to test for functional equivalence.

2) Step 1 only checks the converted circuit's signals corresponding to the original NCL circuit's rail^1 signals with their equivalent Boolean/synchronous specification external outputs or register outputs; hence, the original NCL circuit's rail^0 signals must also be ensured to be inverses of their respective rail^1 signals, in order to guarantee safety after passing Step 1.

3) The NCL netlist is then automatically converted into a graph-structure, and information related to the handshaking control is gathered by traversing the graph. This information is utilized to analyze the handshaking correctness of the circuit in order to check for deadlock.

4 and 5) Once the NCL circuit passes Step 2, each combinational logic (C/L) block must be verified to be both input-complete (Step 4) and observable (Step 5) in order to guarantee liveness of the circuit under all timing scenarios. The input completeness and observability check is already established and illustrated in details in [37]. Hence, input-completeness and observability check are not discussed further in this dissertation.

4.1.1. Functional Equivalence Check

A 3×3 NCL multiplier, shown in Fig. 35, is used as an example to illustrate the equivalence verification procedure for combinational NCL circuits. NCL multipliers use input-incomplete NCL AND functions (denoted with an I inside the AND symbol), input-complete NCL AND functions (denoted with a C inside the AND symbol), NCL Half-Adders (HA), and NCL Full-Adders (FA), which all consist of a combination of NCL threshold gates, as shown in Figs. 36a, 36b, 37, and 38, respectively. All signals in Fig. 35 are dual-rail; and all registers are reset-to-NULL, denoted as *REG_NULL*. In addition to the I/O registers, the multiplier in Fig. 35 includes one intermediate register stage to increase throughput.

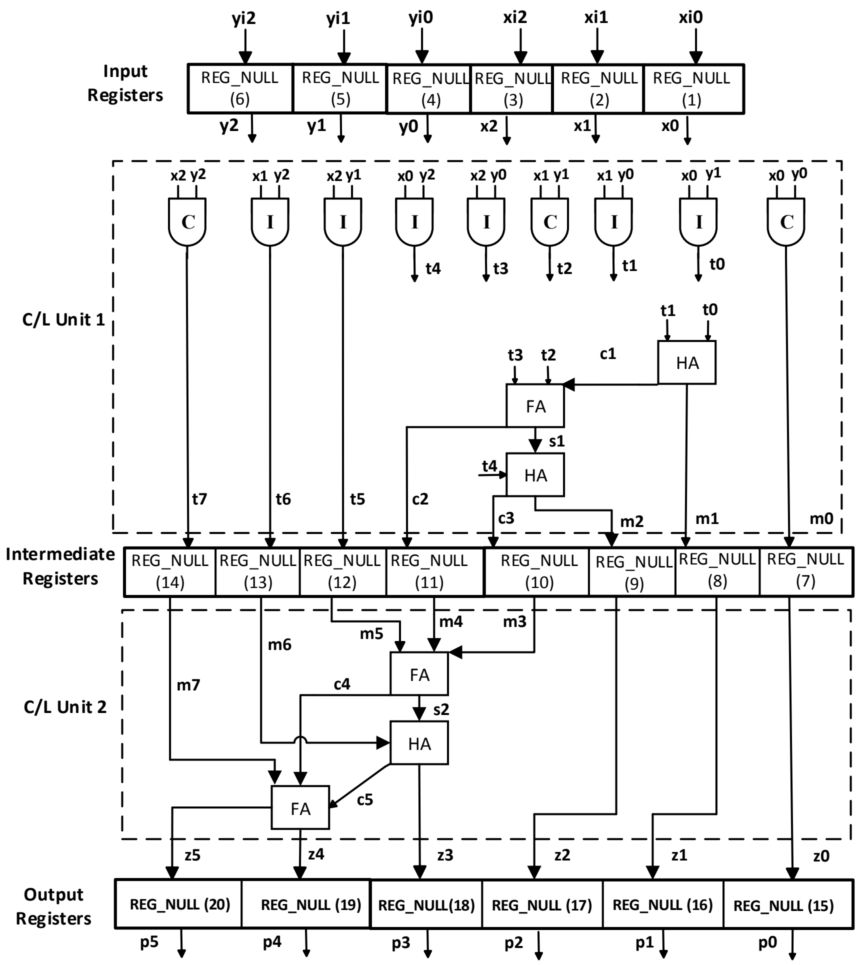
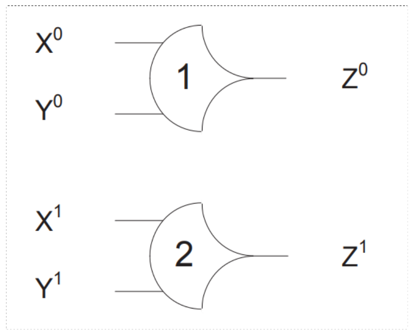
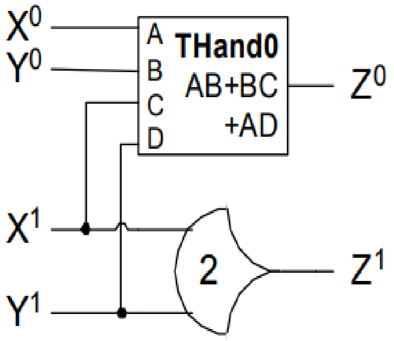


Figure 35. 3×3 NCL Multiplier Circuit.



(a)



(b)

Figure 36. (a) Input Incomplete NCL AND. (b) Input Complete NCL AND.

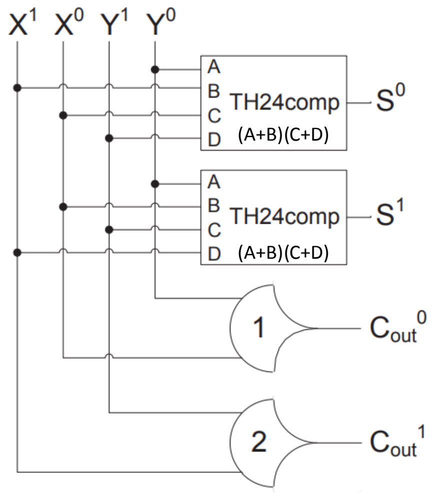


Figure 37. NCL Half Adder Circuit.

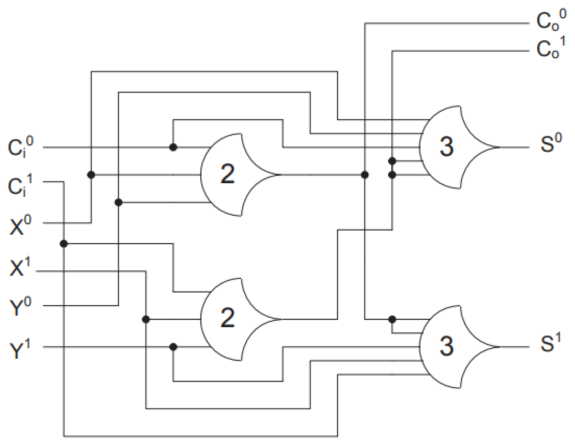


Figure 38. NCL Full Adder Circuit.

The netlist of the NCL 3×3 multiplier is shown in Fig. 39a. The first two lines indicate all primary inputs and primary outputs, respectively. Lines 3-44 correspond to the NCL C/L threshold gates, where the first column is the type of gate, the second column lists the gate's inputs, in comma separated format starting with input *A*, and the last column is the gate's output. Lines 45-64 correspond to 1-bit NCL registers, where the first column is the reset type of the register (i.e., _NULL, _DATA0, or _DATA1, for reset to NULL, DATA0, or DATA1, respectively), the second column denotes the register's level (i.e., the depth of the path through registers without considering the C/L in-between). For the 3×3 multiplier example, there are 3 stages of registers, with levels 1, 2, and 3, starting from the input registers), the third and fourth columns are the register's rail^0 and rail^1 data inputs, respectively, the fifth and sixth columns are the register's *Ki* input and *Ko* output, respectively, and the seventh and eighth columns are the register's rail^0 and rail^1 data outputs, respectively. Lines 65-72 correspond to the C-elements (i.e., THnn gates) used in the handshaking control circuitry, where the first column is *Cn*, with *n* indicating the number of inputs to the C-element, the second column lists the inputs in comma separated format, and the last column is the C-element's output. For example, C4 on line 65 is a 4-input C-element.

The NCL netlist is input to a conversion algorithm that converts it into an equivalent Boolean netlist, as shown in Fig. 39b for the Fig. 39a example. Each NCL C/L gate is replaced with its corresponding Boolean gate that has the same set function, but no hysteresis; each internal dual-rail signal is already represented as 2 Boolean signals, the first for rail^1 and the second for rail^0, so no changes are needed for these; and each primary dual-rail input is replaced with that signal's rail^1, as this corresponds to the equivalent Boolean signal.

```
1.   xi0_0, xi0_1, xi1_0, xi1_1, … , yi1_0, yi1_1,yi2_0, yi2_1
2.   p0_0,p0_1, p1_0, p1_1,…,p5_0,p5_1
3.   th22 x0_1,y0_1  m0_1
4.   thand0 y0_0,x0_0,y0_1,x0_1  m0_0
5.   th22   x0_1,y1_1  t0_1
6.   th12   x0_0,y1_0  t0_0
7.   th22   x0_1,y2_1  t4_1
8.   th12   x0_0,y2_0  t4_0
9.   th22   x1_1,y0_1  t1_1
10.  th12   x1_0,y0_0  t1_0
11.  th22   x1_1,y1_1  t2_1
12.  thand0   y1_0,x1_0,y1_1,x1_1  t2_0
13.  th22   x1_1,y2_1  t6_1
14.  th12   x1_0,y2_0  t6_0
15.  th22   x2_1,y0_1  t3_1
16.  th12   x2_0,y0_0  t3_0
17.  th22   x2_1,y1_1  t5_1
18.  th12   x2_0,y1_0  t5_0
19.  th22   x2_1,y2_1  t7_1
20.  thand0   y2_0,x2_0,y2_1,x2_1  t7_0
21.  th24comp   t0_0,t1_0,t0_1,t1_1  m1_1
22.  th24comp   t0_0,t1_1,t1_0,t0_1  m1_0
23.  th22   t0_1, t1_1  c1_1
24.  th12   t0_0,t1_0  c1_0
25.  th23   t3_0,t2_0,c1_0  c2_0
26.  th23   t3_1,t2_1,c1_1  c2_1
27.  th34w2    c2_0,t3_1,t2_1,c1_1   s1_1
28.  th34w2    c2_1,t3_0,t2_0,c1_0   s1_0
29.  th24comp    s1_0,t4_0,s1_1,t4_1   m2_1
30.  th24comp    s1_0,t4_1,t4_0,s1_1   m2_0
31.  th22    s1_1,t4_1   c3_1
32.  th12    s1_0,t4_0   c3_0
33.  th23    m5_0,m4_0,m3_0   c4_0
34.  th23    m5_1,m4_1,m3_1   c4_1
35.  th34w2    c4_0,m5_1,m4_1,m3_1   s2_1
36.  th34w2    c4_1,m5_0,m4_0,m3_0   s2_0
37.  th24comp    s2_0,m6_0,s2_1,m6_1   z3_1
38.  th24comp    s2_0,m6_1,m6_0,s2_1   z3_0
39.  th22    s2_1,m6_1   c5_1
40.  th12    s2_0,m6_0   c5_0
41.  th23    m7_0,c4_0,c5_0   z5_0
42.  th23    m7_1,c4_1,c5_1   z5_1
43.  th34w2    z5_0,m7_1,c4_1,c5_1  z4_1
44.  th34w2    z5_1,m7_0,c4_0,c5_0  z4_0
45.  Reg_NULL   1   xi0_0 xi0_1   KO3   ko1   x0_0 x0_1
46.  Reg_NULL   1   xi1_0 xi1_1   KO3   ko2   x1_0 x1_1
47.  Reg_NULL   1   xi2_0 xi2_1   KO3   ko3   x2_0 x2_1
48.  Reg_NULL   1   yi0_0 yi0_1   KO3   ko4   y0_0 y0_1
49.  Reg_NULL   1   yi1_0 yi1_1   KO3   ko5   y1_0 y1_1
50.  Reg_NULL   1   yi2_0 yi2_1   KO3   ko6   y2_0 y2_1
51.  Reg_NULL   2   m0_0 m0_1   ko15   ko7   z0_0 z0_1
52.  Reg_NULL   2   m1_0 m1_1   ko16   ko8   z1_0 z1_1
53.  Reg_NULL   2   m2_0 m2_1   ko17   ko9   z2_0 z2_1
54.  Reg_NULL   2   c3_0 c3_1   KO4   ko10   m3_0 m3_1
55.  Reg_NULL   2   c2_0 c2_1   KO4   ko11   m4_0 m4_1
56.  Reg_NULL   2   t5_0 t5_1   KO4   ko12   m5_0 m5_1
57.  Reg_NULL   2   t6_0 t6_1   KO4   ko13   m6_0 m6_1
58.  Reg_NULL   2   t7_0 t7_1   KO5   ko14   m7_0 m7_1
59.  Reg_NULL   3   z0_0 z0_1   Ki   ko15   p0_0 p0_1
60.  Reg_NULL   3   z1_0 z1_1   Ki   ko16   p1_0 p1_1
61.  Reg_NULL   3   z2_0 z2_1   Ki   ko17   p2_0 p2_1
62.  Reg_NULL   3   z3_0 z3_1   Ki   ko18   p3_0 p3_1
63.  Reg_NULL   3   z4_0 z4_1   Ki   ko19   p4_0 p4_1
64.  Reg_NULL   3   z5_0 z5_1   Ki   ko20   p5_0 p5_1
65.  C4   ko7,ko8,ko9,ko10   KO1
66.  C4   ko11,ko12,ko13,ko14   KO2
67.  C2   KO1,KO2   KO3
68.  C3   ko18,ko19,ko20   KO4
69.  C2   ko19,ko20      KO5
70.  C3   ko4,ko5,ko6   KO6
71.  C3   ko1,ko2,ko3   KO7
72.  C2   KO7,KO6   KO
```

```
1.   xi0_1, xi1_1, xi2_1, yi0_1, yi1_1, yi2_1
2.   p0_0,p0_1, p1_0, p1_1,…,p5_0,p5_1
3.   not    xi0_1   xi0_0
4.   not    xi1_1   xi1_0
5.   not    xi2_1   xi2_0
6.   not    yi0_1   yi0_0
7.   not    yi1_1   yi1_0
8.   not    yi2_1   yi2_0
9.   th22    xi0_1 ,yi0_1   p0_1
10.  thand0    yi0_0,xi0_0,yi0_1,xi0_1   p0_0
11.  th22    xi0_1,yi1_1   t0_1
12.  th12    xi0_0,yi1_0   t0_0
13.  th22    xi0_1,yi2_1   t4_1
14.  th12    xi0_0,yi2_0   t4_0
15.  th22  xi1_1,yi0_1 t1_1
16.  th12  xi1_0,yi0_0 t1_0
17.  th22  xi1_1,yi1_1 t2_1
18.  thand0  yi1_0,xi1_0,yi1_1,xi1_1 t2_0
19.  th22  xi1_1,yi2_1 t6_1
20.  th12  xi1_0,yi2_0 t6_0
21.  th22  xi2_1,yi0_1 t3_1
22.  th12  xi2_0,yi0_0 t3_0
23.  th22  xi2_1,yi1_1 t5_1
24.  th12  xi2_0,yi1_0 t5_0
25.  th22  xi2_1,yi2_1 t7_1
26.  thand0    yi2_0,xi2_0,yi2_1,xi2_1   t7_0
27.  th24comp    t0_0,t1_0,t0_1,t1_1   p1_1
28.  th24comp    t0_0,t1_1,t1_0,t0_1   p1_0
29.  th22    t0_1, t1_1   c1_1
30.  th12    t0_0,t1_0   c1_0
31.  th23    t3_0,t2_0,c1_0   c2_0
32.  th23    t3_1,t2_1,c1_1   c2_1
33.  th34w2    c2_0,t3_1,t2_1,c1_1   s1_1
34.  th34w2    c2_1,t3_0,t2_0,c1_0   s1_0
35.  th24comp    s1_0,t4_0,s1_1,t4_1   p2_1
36.  th24comp    s1_0,t4_1,t4_0,s1_1   p2_0
37.  th22    s1_1,t4_1   c3_1
38.  th12    s1_0,t4_0   c3_0
39.  th23    t5_0,c2_0,c3_0   c4_0
40.  th23    t5_1,c2_1,c3_1   c4_1
41.  th34w2    c4_0,t5_1,c2_1,c3_1   s2_1
42.  th34w2    c4_1,t5_0,c2_0,c3_0   s2_0
43.  th24comp    s2_0,t6_0,s2_1,t6_1   p3_1
44.  th24comp    s2_0,t6_1,t6_0,s2_1   p3_0
45.  th22    s2_1,t6_1   c5_1
46.  th12    s2_0,t6_0   c5_0
47.  th23    t7_0,c4_0,c5_0    p5_0
48.  th23     t7_1,c4_1,c5_1   p5_1
49.  th34w2     p5_0,t7_1,c4_1,c5_1   p4_1
50.  th34w2     p5_1,t7_0,c4_0,c5_0   p4_0
```

Figure 39. (a) 3x3 NCL Multiplier Netlist. (b) Converted Boolean Netlist.

The rail^1 primary inputs are then inverted to produce internal signals corresponding to what used to be the rail^0 primary inputs, as these are utilized in the internal logic. The first two lines in the converted netlist are the list of primary inputs and outputs, respectively, where the inputs correspond to the original NCL netlist's rail^1 inputs, and the outputs include both rail^0 and rail^1 outputs. Lines 3-8 in the converted netlist are the added inverters used to produce the equivalent signals to the original rail^0 inputs, as these were removed in the conversion. The format of each gate is the same as explained above for the NCL netlist. All *Reg_NULL* components are removed during conversion by setting their data outputs equal to their data inputs, since these have no corresponding functionality in the equivalent Boolean circuit. Purely C/L circuits will not include *Reg_DATA* components, as these correspond to synchronous registers; these will be discussed in Section 4.2: Equivalence Verification for Sequential NCL Circuits.

The converted Boolean netlist is automatically encoded in the Satisfiability Modulo Theory Library (SMT_LIB) language [49], using a conversion tool we developed, which is then input to an SMT solver to check for functional equivalence with the corresponding specification. For the 3x3 multiplier example, the SMT solver checks for the following safety property: $F_{\text{NCL\_Bool\_Equiv.}}$ (*x2_1, x1_1, x0_1, y2_1, y1_1, y0_1*) = MUL (*x*, *y*), where (*x2_1, x1_1, x0_1*) and (*y2_1, y1_1, y0_1*) are the *x* and *y rail^1* inputs, respectively, starting with the MSB. We use the Z3 SMT solver [44] to check for equivalence, but any combinational equivalence checker could be used. Note that only the rail^1 outputs need to be checked here, as these correspond to the Boolean specification circuit outputs. The rail^0 outputs will be utilized for the invariant check, described next.

4.1.2. Invariant Check

Since only the rail^1 outputs are utilized for the functional equivalence check, the rail^0 outputs must also be checked to ensure safety. To address correctness of the rail^0 outputs, an additional SMT invariant proof obligation is required for the original NCL circuit, which states that in any reachable NCL circuit state where the outputs are all DATA, every rail^0 output must be the inverse of its corresponding rail^1 output.

One way to achieve this is to initialize all registers to NULL, all C/L gate outputs to 0, and all register *Ki* inputs to *rfd* (i.e., logic 1), then make all the primary inputs DATA (i.e., represented in SMT as all combinations of valid DATA) and step the circuit. This will allow the input DATA to flow through all stages of the circuit, generating all possible combinations of valid DATA at the primary outputs. For each primary dual-rail output, the invariant is then checked to ensure that the rail^0 output is the inverse of its corresponding rail^1 output. For a C/L circuit with *j* registers $r^1, \ldots, r^j$, *k* C/L threshold gates $g^1, \ldots, g^k$, *q* dual-rail inputs $i^1, \ldots, i^q$, and *l* dual-rail outputs $o^1{<}R0, R1{>}, \ldots, o^l{<}R^0, R^1{>}$, where *R0* and *R1* are the output's rail^0 and rail^1, respectively, the proof obligation for this invariant check is shown below as Proof Obligation 1. Predicate *P1* indicates that all registers in step *A* are reset-to-NULL. *P2* and *P3* state that all threshold gates and *Ki* register inputs are initialized to logic 0 and 1, respectively. *P4* indicates that all step *A* inputs are DATA. *P5* represents the symbolic step of the circuit with all threshold gates set to 0 and all inputs set to DATA, with the new values of the threshold gates stored in ($g_B{}^1 \ldots g_B{}^k$). *P6* states that the rails of each dual-rail output are complements of each other. The proof obligation, *PO1*, indicates that if DATA is allowed to flow from the primary inputs to the primary outputs, then for all possible valid DATA inputs, each output's rail^0, R^0, is always the inverse of its respective rail^1 output, R^1.

Proof Obligation 1:

$P1: \bigwedge_{n=1}^{j} (r_A{}^n = 0b00)$
$P2: \bigwedge_{n=1}^{k} (g_A{}^n = 0)$
$P3: \bigwedge_{n=1}^{j} (Ki_A{}^n = 1)$
$P4: \bigwedge_{n=1}^{q} (i_A{}^n = 0b01) \vee (i_A{}^n = 0b10)$
$P5: (g_B{}^1, \ldots, g_B{}^k) = NCLStep(i_A{}^1, \ldots, i_A{}^q)$
$P6: \bigwedge_{n=1}^{l} o_B{}^n <R^0> = \neg\, o_B{}^n <R^1>$

$\textbf{\textit{PO1}}: \{P1 \wedge P2 \wedge P3 \wedge P4 \wedge P5 \Rightarrow P6\}$

An alternative, faster method to check invariants is to check each NCL circuit stage independently. To do this, we developed an algorithm that reads the original NCL circuit netlist and separately extracts each circuit stage. Then, for each extracted stage, we set all gate outputs to 0, all stage inputs to DATA, and step the circuit, such that the stage's outputs become all possible combinations of valid DATA. Finally, the invariant is checked for each of the stage's dual-rail outputs to ensure that its $rail^0$ is the inverse of its corresponding $rail^1$. The proof obligation for this second invariant check method is shown below as Proof Obligation 2, where the extracted stage has j dual-rail inputs $i^1, \ldots, i^j$, m threshold gates $g^1, \ldots, g^m$, and k dual-rail outputs $o^1<R^0, R^1>, \ldots, o^k<R^0, R^1>$, where R^0 and R^1 are the output's rail^0 and rail^1, respectively. Predicate *P1* indicates that all stage inputs are valid data; *P2* indicates that all NCL threshold gates in the stage are initialized to 0; *P3* corresponds to a NULL to DATA transition of the stage; and *P4* states that the rails of each dual-rail output are complements of each other. The Proof Obligation states that after a NULL to DATA transition of the stage with all possible valid DATA inputs, that each output's rail^0, R^0, is always the inverse of its respective rail^1 output, R^1.

Proof Obligation 2:

$P1: \bigwedge_{n=1}^{j} (i_A{}^n = 0b01) \vee (i_A{}^n = 0b10)$
$P2: \bigwedge_{n=1}^{m} (g_A{}^n = 0)$
$P3: (g_B{}^1, \ldots, g_B{}^m) = NCLStep(i_A{}^1, \ldots, i_A{}^j)$
$P4: \bigwedge_{n=1}^{k} o_B{}^n <R^0> = \neg\, o_B{}^n <R^1>$

$\textbf{\textit{PO2:}} \{P1 \wedge P2 \wedge P3 \Rightarrow P4\}$

This second invariant check method is much faster than the first, since it breaks the problem into a set of smaller invariant checks (i.e., one per stage), whereas the first method checks the invariant for the entire circuit all at once. For example, method 2 is 38% faster for a 2-stage 10×10 multiplier, and becomes even faster when the circuit includes additional stages. Note that for both invariant check methods, the NCL gates are modeled in SMT as Boolean functions (i.e., no hysteresis), since invariant checking only requires the NULL to DATA transition that only utilizes each gate's set function, which is the same for the Boolean and NCL state-holding gate implementations. This optimization reduces the invariant check time by approximately half (e.g., 377 sec. vs 192 sec. for a non-pipelined 10-bit × 10-bit unsigned multiplier).

4.1.3. Handshaking Check

Liveness means absence of deadlock in a circuit. For combinational NCL circuits, proper connections between handshaking signals, along with observable and input-complete C/L, ensures liveness. The same NCL netlist shown in Fig. 39a, used as input for the functional equivalence and invariant checks, is also utilized as input for the liveness checks. For the handshaking check, the NCL netlist is automatically converted into a graph structure, and the handshaking paths and C-element connections are traced back to verify proper handshaking, ensuring that every register input acknowledges all preceding stage register outputs that took part in its calculation. For each NCL register, *i*, its dual-rail input is traced back through its preceding C/L to identify every NCL register's dual-rail output that took part in its calculation, generating a fan-in list, *reg_fanin(i)*. For example, referring to Fig. 35, *reg_fanin(8)* would be *1, 2, 4, 5*, since *x0*, *x1*, *y0*, and *y1* are all used to generate *m1*. Also, for each NCL register, *i*, its *Ko* output is traced through the C-element handshaking circuitry to identify every NCL register's *Ki* input that register$_i$'s *Ko* output took part in calculating, generating a *Ko* fanout list, *ko_fanout(i)*. For example, referring to Fig. 40, which shows the handshaking circuitry for the 3×3 multiplier example, *ko_fanout(8)* would be *1, 2, 3, 4, 5, 6*, since *ko8* takes part in the generation of the *Ki* input for all of the preceding stage's registers (i.e., 1-6).

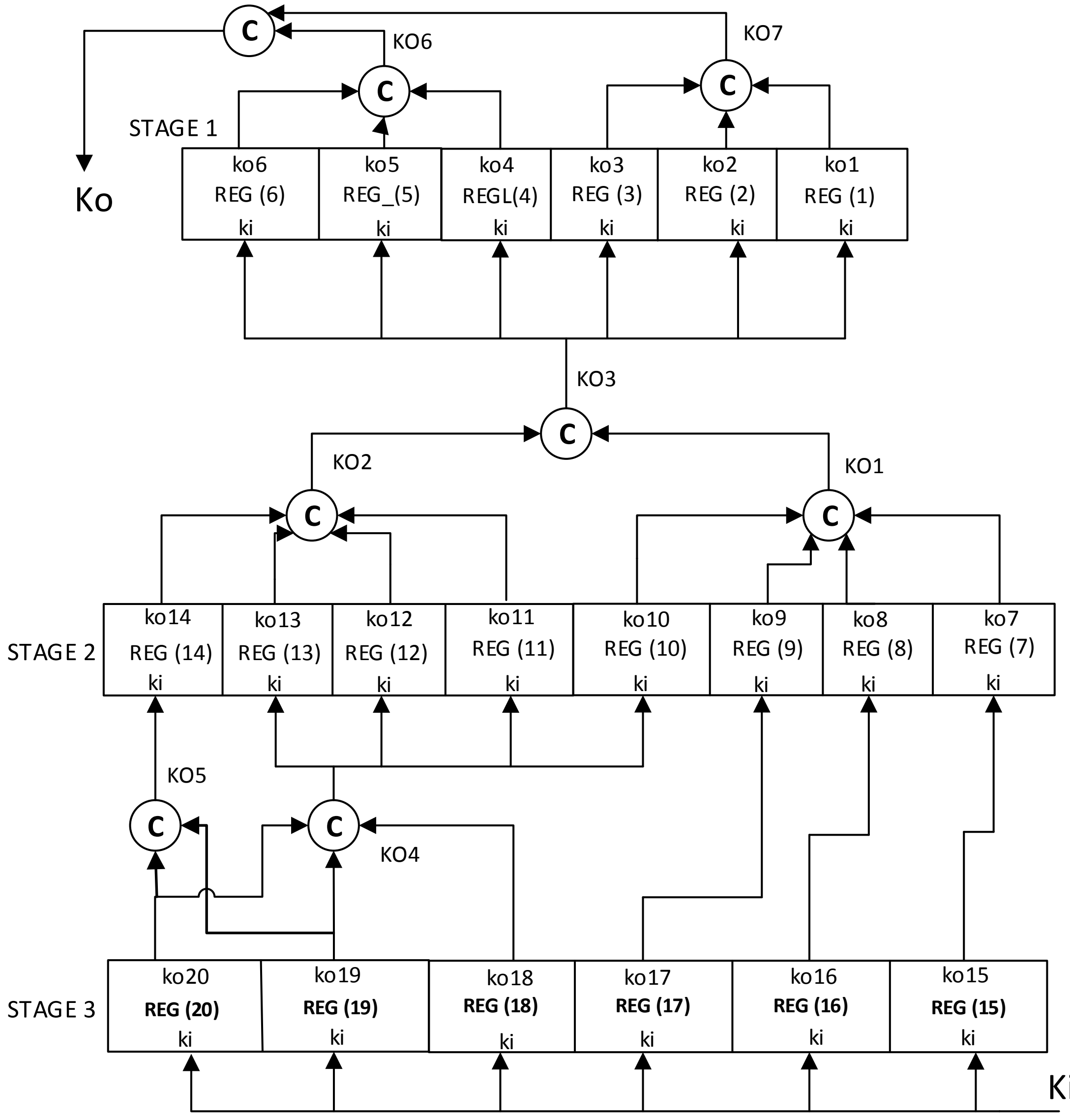


Figure 40. Handshaking Connection for 3x3 NCL Multiplier.

After *reg_fanin* and *ko_fanout* for each NCL register is calculated, as shown in Fig. 41 for the 3×3 multiplier example, *reg_fanin(k)* is checked to ensure that it is a subset of *ko_fanout(k),* for all NCL registers. Note that 0 in *reg_fanin* denotes a primary data input; and 0 in *ko_fanout* denotes the external *Ko* output. Bit-wise completion results in *reg_fanin(k)* being equal to *ko_fanout(k)*, while full-word completion results in *reg_fanin(k)* being a proper subset of *ko_fanout(k)*, with the restriction that each register that is in *ko_fanout(k)* and not in *reg_fanin(k)* must be from the immediate preceding register stage of register *k*. *reg_fanin(k)* not being a subset of *ko_fanout(k)* could result in deadlock, while *reg_fanin(k)* being a proper subset of *ko_fanout(k)* but violating the stage restriction described above, could either result in deadlock or may just decrease circuit performance. Hence, if *reg_fanin(k)* is a proper subset of *ko_fanout(k)*, then each register that is in *ko_fanout(k)* and not in *reg_fanin(k)* is automatically inspected to ensure that it meets this stage restriction. If not, a warning message is generated denoting the extra register in that particular register's *ko_fanout* list, to allow for easier manual inspection. For the Fig. 40 example, the first stage utilizes full-word completion, while the second stage uses bit-wise completion.

An additional check is needed to ensure correct connection of the external *Ki* input, namely that the external *Ki* should be the *Ki* input to every register that produces a primary data output. The developed algorithm generates an appropriate descriptive error message in case the NCL circuit fails to satisfy any of these handshaking checks. Furthermore, it checks to ensure that no data signal is part of the handshaking circuitry, and that no handshaking signal is part of a data signal.

```
1:  reg_fanin: 0                    ko_fanout: 0
2:  reg_fanin: 0                    ko_fanout: 0
3:  reg_fanin: 0                    ko_fanout: 0
4:  reg_fanin: 0                    ko_fanout: 0
5:  reg_fanin: 0                    ko_fanout: 0
6:  reg_fanin: 0                    ko_fanout: 0
7:  reg_fanin: [1, 4]               ko_fanout: [1, 2, 3, 4, 5, 6]
8:  reg_fanin: [1, 2, 4, 5]         ko_fanout: [1, 2, 3, 4, 5, 6]
9:  reg_fanin: [1, 2, 3, 4, 5, 6]   ko_fanout: [1, 2, 3, 4, 5, 6]
10: reg_fanin: [1, 2, 3, 4, 5, 6]   ko_fanout:[ 1, 2, 3, 4, 5, 6]
11: reg_fanin: [1, 2, 3, 4, 5]      ko_fanout: [1, 2, 3, 4, 5, 6]
12: reg_fanin: [3, 5]               ko_fanout:[ 1, 2, 3, 4, 5, 6]
13: reg_fanin: [2, 6]               ko_fanout: [1, 2, 3, 4, 5, 6]
14: reg_fanin: [3, 6]               ko_fanout: [1, 2, 3, 4, 5, 6]
15: reg_fanin: [7]                  ko_fanout: [7]
16: reg_fanin: [8]                  ko_fanout: [8]
17: reg_fanin: [9]                  ko_fanout: [9]
18: reg_fanin: [10, 11, 12, 13]     ko_fanout: [10, 11, 12, 13]
19: reg_fanin: [10,11,12,13,14]     ko_fanout: [10, 11, 12, 13, 14]
20: reg_fanin: [10,11,12,13,14]     ko_fanout: [10, 11, 12, 13, 14]
```

Figure 41. *Reg_fanin* and *Ko_fanout* Lists for the 3x3 NCL Multiplier.

4.1.4. Combinational NCL Circuits Results

The methodology has been demonstrated on several multipliers and ISCAS-85 [45] combinational circuit benchmarks, as shown in Table 6. *umult*N represents a non-pipelined N-bit × N-bit unsigned multiplier. The NCL-to-Boolean netlist conversion time was negligible compared to the safety check and invariant check performed by the Z3 SMT solver [44] on an Intel® Core™ i7-4790 CPU with 32GB of RAM, running at 3.60 GHz. To test the methodology, we injected several bugs. The *umult*10-Bn multipliers are circuits with *n* different kinds of bugs, and the (B) in either the Functional Check, Invariant check, or Handshaking Check column denotes which check detected the bug. The –B1 bug incorrectly swaps rails of a dual-rail signal. –B2 represents a faulty data connection. For example, the *F* output of NCL $gate_i$ should be connected to the *X* input of NCL $gate_j$; however, *X* is instead connected to the output of NCL $gate_k$, which would result in a logical error. –B3 corresponds to an incorrect handshaking connection; and external *Ki* and *Ko* bugs are represented by –B4. –B5 denotes a rail-duplication error, where $rail^0$ and $rail^1$ of a particular signal are the same wire.

Table 6. Verification Results for Various C/L NCL Circuits.

Circuits	Functional Check (sec.)	Invariant Check (sec.)	Handshaking Check (sec.)	Total Time (sec.)
ISCAS *c17*	0.01	0.01	0.0020	0.0220
umult2	0.02	0.01	0.0997	0.1297
umult3	0.04	0.02	0.1087	0.1687
umult6	0.32	0.33	0.8238	1.4738
umult8	10.62	6.79	9.3090	26.719
umult10	683.49	192.39	70.370	946.25
ISCAS *c432*	1.03	1.06	3.0111	5.1011
umult10*-B1*	0.08 (B)	0.10 (B)	70.370	70.550
umult10*-B2*	0.06 (B)	192.39	70.370	262.82
umult10*-B3*	683.49	192.39	69.1538 (B)	945.034
umult10*-B4*	683.49	0.08 (B)	72.0235 (B)	755.5935
umult10*-B5*	0.1 (B)	0.09 (B)	70.37	71.37

4.2. Equivalence Verification for Sequential NCL Circuits

4.2.1. Safety Check

As demonstrated in Section 4.1, proposed equivalence verification methodology proved to be a fast and scalable approach for C/L NCL circuits. Hence, in this section that approach is further extended to the verification of sequential NCL circuits, which is far more complex due to datapath feedback.

The verification procedure requires two steps. In the first step, we take a sequential NCL circuit and convert it to an equivalent synchronous circuit. We utilize the theory of WEB-refinement [34] to compare the synchronous netlist generated from the NCL circuit with the original synchronous specification, as the notion of correctness. The major advantage of applying

WEB-refinement to the generated equivalent synchronous circuit instead of the actual NCL circuit is that a synchronous circuit is much more deterministic compared to its NCL equivalent, which makes the verification time much faster. The generated synchronous circuit, specification synchronous circuit, and the WEB-refinement property are automatically encoded in the SMT-LIB language. The resulting equivalence property is then checked using an SMT solver. In the second step, we check the handshaking connections between components, which is similar to the C/L NCL handshaking check, discussed in Section 4.1.3.

To describe this methodology, an unsigned Multiply and Accumulate (MAC) unit is utilized as an example circuit. Fig. 26 shows a synchronous MAC, where $A' = A + X \times Y$; and Fig. 42 shows the equivalent NCL version. The 4-phase QDI handshaking protocol utilized for NCL circuits requires at least $2N+1$ NCL registers in a feedback loop that contains N DATA tokens, in order to avoid deadlock [24]. Hence, at least 3 NCL registers are needed in the MAC feedback loop to avoid deadlock, as shown in Fig. 42. Although the synchronous and NCL MACs seem similar, they are structurally very different. Synchronous registers are clocked, whereas alternating DATA/NULL transitions in NCL are maintained via C-elements and a well-defined handshaking scheme. K_i and K_o are the external *request* input and *acknowledge* output, respectively.

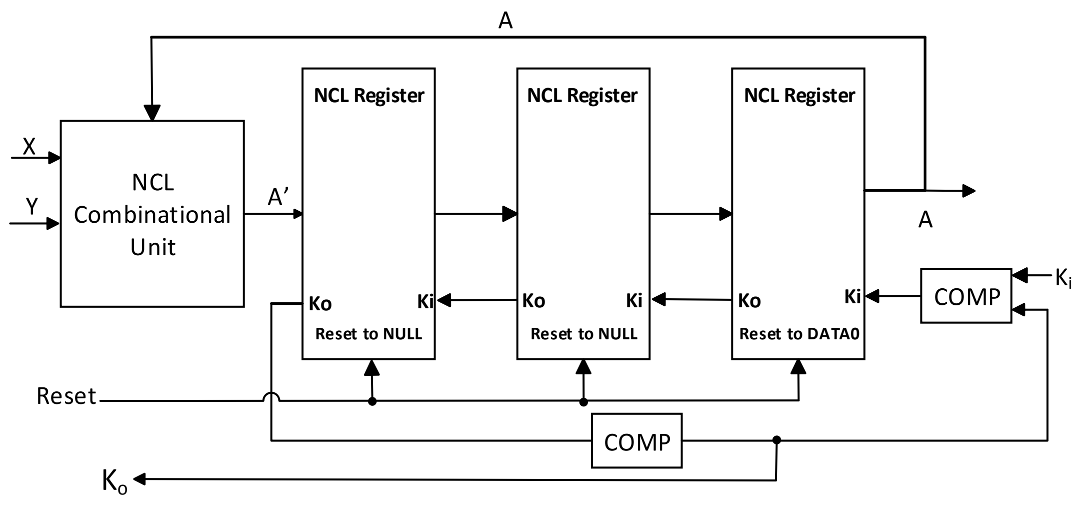


Figure 42. NCL MAC Unit.

Fig. 43 shows the datapath diagram for a *4 + 2×2* NCL MAC with 2 C/L stages and 4 registers in the feedback loop (note that including a 4th register in the feedback loop increases throughput compared to using the minimum required 3 registers, since this allows the DATA and NULL wavefronts to flow more independently [24]. (Xi_1, Xi_0) and (Yi_1, Yi_0) are the two bits of inputs *Xi* and *Yi*, respectively. The product of *Xi* and *Yi* is added with the 4-bit accumulator output, *Acci*, where $Acci_3$ and $Acci_0$ are the MSB and LSB, respectively. All signals shown in Fig. 43 are dual-rail signals. HA and FA are the NCL half-adder and full-adder components, shown in Figs. 37 and 38, respectively; and FAs is a full-adder component without a carry output; hence, it utilizes two 2-input XOR functions, each comprised of two TH24comp gates, to compute its *sum* output.

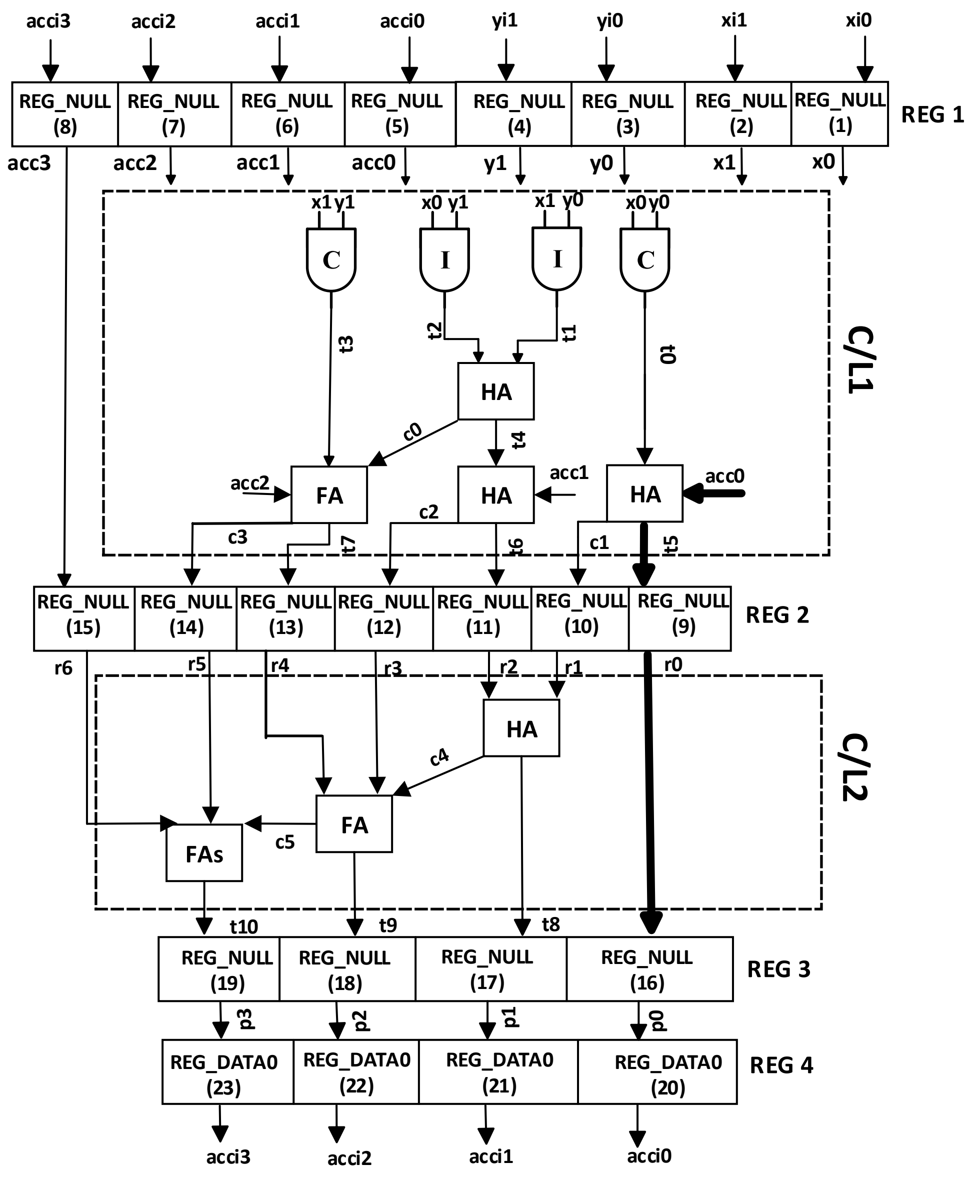


Figure 43. 4 + 2x2 NCL MAC Datapath.

Fig. 44a shows the netlist of the NCL 4+2×2 MAC, following the same structure as described in Section 4.1.1. The first 2 lines are the circuit inputs and outputs, respectively; lines 3-38 are the NCL threshold gates; lines 39-61 are the NCL registers; and lines 62-69 are C-elements used in the handshaking network.

```
1.  xi0_0, xi0_1, xi1_0, xi1_1, yi0_0, yi0_1, yi1_0, yi1_1
2.  acci0_0,acci0_1,acci1_0,acci1_1,…,acci3_0,acci3_1
3.  th22   x0_1,y0_1  t0_1
4.  thand0  y0_0,x0_0,y0_1,x0_1  t0_0
5.  th12   x1_0,y0_0  t1_0
6.  th22   x1_1,y0_1  t1_1
7.  th12   x0_0,y1_0  t2_0
8.  th22   x0_1,y1_1  t2_1
9.  th12   x1_0,y1_0  t3_0
10.  th22   x1_1,y1_1  t3_1
11.  th24comp   t2_0,t1_1,t1_0,t2_1  t4_0
12.  th24comp   t2_0,t1_0,t2_1,t1_1  t4_1
13.  th12   t2_0,t1_0  c0_0
14.  th22   t1_1,t2_1  c0_1
15.  th24comp   acc0_0,t0_1,t0_0,acc0_1  t5_0
16.  th24comp   acc0_0,t0_0,acc0_1,t0_1  t5_1
17.  th12   acc0_0,t0_0  c1_0
18.  th22   t0_1,acc0_1  c1_1
19.  th24comp   acc1_0,t4_1,t4_0,acc1_1  t6_0
20.  th24comp   acc1_0,t4_0,acc1_1,t4_1  t6_1
21.  th12   acc1_0,t4_0  c2_0
22.  th22   t4_1,acc1_1  c2_1
23.  th23   t3_0,acc2_0,c0_0  c3_0
24.  th23   t3_1,acc2_1,c0_1  c3_1
25.  th34w2   c3_1,t3_0,acc2_0,c0_0  t7_0
26.  th34w2   c3_0,t3_1,acc2_1,c0_1  t7_1
27.  th24comp   r1_0,r2_1,r2_0,r1_1  t8_0
28.  th24comp   r1_0,r2_0,r1_1,r2_1  t8_1
29.  th12   r1_0,r2_0  c4_0
30.  th22   r2_1,r1_1  c4_1
31.  th23   r4_0,r3_0,c4_0  c5_0
32.  th23   r4_1,r3_1,c4_1  c5_1
33.  th34w2   c5_1,r4_0,r3_0,c4_0  t9_0
34.  th34w2   c5_0,r4_1,r3_1,c4_1  t9_1
35.  th24comp   r5_0,r6_1,r6_0,r5_1  c6_0
36.  th24comp   r5_0,r6_0,r5_1,r6_1  c6_1
37.  th24comp   c5_0,c6_1,c6_0,c5_1  t10_0
38.  th24comp   c5_0,c6_0,c5_1,c6_1  t10_1
39.  Reg_NULL 1  xi0_0,xi0_1  KO2  ko1  x0_0,x0_1
40.  Reg_NULL 1  xi1_0,xi1_1  KO2  ko2  x1_0,x1_1
41.  Reg_NULL 1  yi0_0 yi0_1  KO2  ko3  y0_0 y0_1
42.  Reg_NULL 1  yi1_0 yi1_1  KO2  ko4  y1_0 y1_1
43.  Reg_NULL 1  acci0_0 acci0_1  KO2  ko5  acc0_0 acc0_1
44.  Reg_NULL 1  acci1_0 acci1_1  KO2  ko6  acc1_0 acc1_1
45.  Reg_NULL 1  acci2_0 acci2_1  KO2  ko7  acc2_0 acc2_1
46.  Reg_NULL 1  acci3_0 acci3_1  KO2  ko8  acc3_0 acc3_1
47.  Reg_NULL 2  t5_0  t5_1  ko16  ko9   r0_0  r0_1
48.  Reg_NULL 2  c1_0  c1_1  KO3  ko10  r1_0  r1_1
49.  Reg_NULL 2  t6_0 t6_1  KO3  ko11  r2_0 r2_1
50.  Reg_NULL 2  c2_0 c2_1  KO3  ko12  r3_0 r3_1
51.  Reg_NULL 2  t7_0 t7_1  KO3  ko13  r4_0 r4_1
52.  Reg_NULL 2  c3_0 c3_1  KO3  ko14  r5_0 r5_1
53.  Reg_NULL 2  acc3_0 acc3_1  KO3  ko15  r6_0 r6_1
54.  Reg_NULL 3  r0_0 r0_1  ko20  ko16  p0_0 p0_1
55.  Reg_NULL 3  t8_0 t8_1  ko21  ko17  p1_0 p1_1
56.  Reg_NULL 3  t9_0 t9_1  ko22  ko18  p2_0 p2_1
57.  Reg_NULL 3  t10_0 t10_1  ko23  ko19  p3_0 p3_1
58.  Reg_DATA0 4  p0_0  p0_1  KO4  ko20  acci0_0  acci0_1
59.  Reg_DATA0 4  p1_0  p1_1  KO5  ko21  acci1_0  acci1_1
60.  Reg_DATA0 4  p2_0  p2_1  KO6  ko22  acci2_0  acci2_1
61.  Reg_DATA0 4  p3_0  p3_1  KO7  ko23  acci3_0  acci3_1
62.  C4  ko9,ko10,ko11,ko12  KO1
63.  C4  ko13,ko14,ko15,KO1  KO2
64.  C3  ko17,ko18,ko19  KO3
65.  C2  Ki,ko5  KO4
66.  C2  Ki,ko6  KO5
67.  C2  Ki,ko7  KO6
68.  C2  Ki,ko8  KO7
69.  C4  ko1,ko2,ko3,ko4  KO
```

```
1. xi0_1, xi1_1,  yi0_1, yi1_1
2.acci0_0,acci0_1,acci1_0,acci1_1,…,acci3_0,acci3_1
3.  not  xi0_1  xi0_0
4.  not  yi0_1  yi0_0
5.  not  xi1_1  xi1_0
6.  not  yi1_1  yi1_0
7.  th12   xi0_0,yi0_0  t0_0
8.  th22   xi0_1,yi0_1  t0_1
9.  th12   xi1_0,yi0_0  t1_0
10.  th22   xi1_1,yi0_1  t1_1
11.  th12   xi0_0,yi1_0  t2_0
12.  th22   xi0_1,yi1_1  t2_1
13. th12   x1_0,y1_0  t3_0
14.  th22   x1_1,y1_1  t3_1
15.  th24comp   t2_0,t1_1,t1_0,t2_1  t4_0
16.  th24comp   t2_0,t1_0,t2_1,t1_1  t4_1
17.  th12   t2_0,t1_0  c0_0
18.  th22   t1_1,t2_1  c0_1
19.  th24comp   acci0_0,t0_1,t0_0,acci0_1  t5_0
20.  th24comp   acci0_0,t0_0,acci0_1,t0_1  t5_1
21.  th12   acci0_0,t0_0  c1_0
22.  th22   t0_1,acci0_1  c1_1
23.  th24comp   acci1_0,t4_1,t4_0,acci1_1  t6_0
24.  th24comp   acci1_0,t4_0,acci1_1,t4_1  t6_1
25.  th12   acci1_0,t4_0  c2_0
26.  th22   t4_1,acci1_1  c2_1
27.  th23   t3_0,acci2_0,c0_0  c3_0
28.  th23   t3_1,acci2_1,c0_1  c3_1
29.  th34w2   c3_1,t3_0,acci2_0,c0_0  t7_0
30.  th34w2   c3_0,t3_1,acci2_1,c0_1  t7_1
31.  th24comp   c1_0,t6_1,t6_0,c1_1  t8_0
32.  th24comp   c1_0,t6_0,c1_1,t6_1  t8_1
33.  th12   c1_0,t6_0  c4_0
34.  th22   t6_1,c1_1  c4_1
35.  th23   t7_0,c2_0,c4_0  c5_0
36.  th23   t7_1,c2_1,c4_1  c5_1
37.  th34w2   c5_1,t7_0,c2_0,c4_0  t9_0
38.  th34w2   c5_0,t7_1,c2_1,c4_1  t9_1
39.  th24comp   c3_0,acci3_1,acci3_0,c3_1  c6_0
40.  th24comp   c3_0,acci3_0,c3_1,acci3_1  c6_1
41.  th24comp   c5_0,c6_1,c6_0,c5_1  t10_0
42.  th24comp   c5_0,c6_0,c5_1,c6_1  t10_1
43.  Reg_0   t5_0  t5_1   acci0_0  acci0_1
44.  Reg_0   t8_0  t8_1   acci1_0  acci1_1
45.  Reg_0   t9_0  t9_1   acci2_0  acci2_1
46.  Reg_0   t10_0  t10_1   acci3_0  acci3_1
```

Figure 44. (a) 4 + 2x2 NCL MAC Netlist. (b) Converted Synchronous Equivalent Netlist.

The converted netlist (NCL-SYNC) is depicted in Fig. 44b. The conversion algorithm for sequential NCL circuits is slightly different than that for C/L NCL circuits, described in Section 4.1.1, since the sequential NCL circuit contains reset-to-DATA registers, which are replaced with a 2-bit resettable synchronous register, 1 bit for each rail of the corresponding NCL dual-rail register. Like for C/L NCL circuits, all reset-to-NULL registers, handshaking signals, and C-elements are eliminated; and all C/L NCL gates are replaced with their corresponding relaxed (i.e., Boolean) gate.

The NCL-SYNC netlist must next be checked against the synchronous specification (SPEC-SYNC) netlist for equivalence. When verifying C/L NCL circuits, the circuit functionality could be specified as a Boolean function. However, since sequential circuits involve states and transitions, transition systems are used as the formal model to capture the behaviors of both the NCL-SYNC netlist as well as the SPEC-SYNC netlist. The theory of WEB refinement [34] defines what it means for an implementation transition system to be functionally equivalent to a specification transition system, as discussed in Section 3.2.3.1. Therefore, the theory of WEB refinement is implemented for checking equivalence in the NCL sequential case as well.

The theory of WEB refinement allows for stutter between the implementation transition system and the specification transition system. What this means is that multiple but finite transitions of the implementation can match to a single specification transition. *Rank* functions (functions that map circuit states to natural numbers) are used to distinguish finite stutter from deadlock (infinite stutter). Another characteristic of WEB refinement is the use of *refinement maps*, which are functions that map implementation states to specification states. Refinement maps allow for the implementation and specification to be specified at significantly different abstraction levels. However, since the *rail*^{1} registers of NCL-SYNC and the registers of SPEC-SYNC have a one-to-one mapping, there is no stutter between these two transition systems, and the refinement is simply a projection of the *rail*^{1} registers of the implementation state to the registers of the specification state.

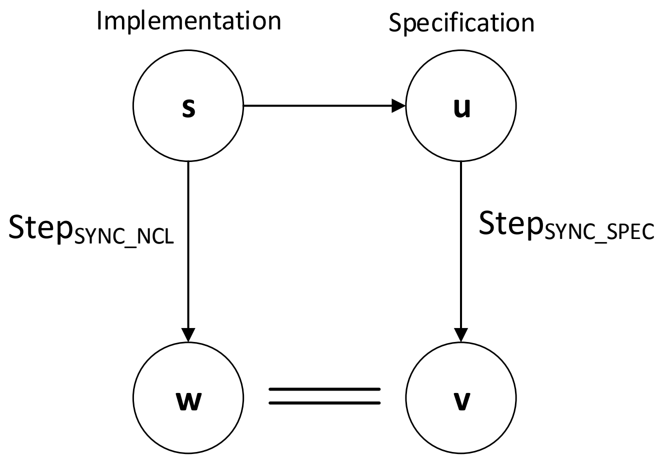


Figure 45. Proof Obligation to Check Equivalence of *NCL_SYNC* and *SPEC_SYNC* Netlists.

Therefore, the correctness proof obligations required for verifying WEB refinement can be reduced to the proof obligation given below and shown in Fig. 45. In the figure, *s* is a state of NCL-SYNC. *u* is a SPEC-SYNC state obtained by projecting the values of the rail^1 registers from state *s*. *Step$_{SYNC\_NCL}$* and *Step$_{SYNC\_SPEC}$* are the functions that correspond to a single step of the NCL-SYNC circuit and the SPEC-SYNC circuit, respectively. *w* is the state obtained by stepping NCL-SYNC from state *s*; and *v* is the state obtained by stepping SPEC-SYNC from state *u*. The proof obligation states that the two circuits are functionally equivalent if for every state *s* of NCL-SYNC, the corresponding projection of values from the *rail*^{1} registers of the *w* state are equivalent to the values of the corresponding registers in the *v* state. This proof obligation can be encoded in the SMT-LIB language, as shown in Proof Obligation 3, *PO3,* below, and checked using an SMT solver.

Proof Obligation 3:

$$PO3 : \{ \forall s :: s \in S_{SYNC\_NCL} :: [u = Reg\_Proj\,(s) \wedge w = Step_{SYNC\_NCL}\,(s) \wedge v = Step_{SYNC\_SPEC}\,(u)]$$
$$\Rightarrow Reg\_Proj\,(w) = v\}.$$

After verifying function equivalence, the rail^0 outputs of each C/L stage must also be checked to ensure safety, as detailed in Section 4.1.2. Note that for sequential circuits, which include datapath feedback, the first invariant check method that checks the entire circuit simultaneously won't work; hence, the second, much faster method that performs the invariant check independently for each stage is utilized.

4.2.2. Liveness Check

Fig. 46 shows the handshaking connections for the *4 + 2×2* NCL MAC. Full-word completion is used by the input register, Reg 1, to generate a single *Ko*. Full-word completion is also utilized between Reg 1 and Reg 2, since bit-wise completion would have the same delay and require more area. Partial bit-wise completion is utilized between Reg 2 and Reg 3, since full bit-wise completion would have the same delay and require more area. Bit-wise completion is utilized between Reg 3 and Reg 4, and for the output register, Reg 4. The handshaking check for sequential NCL circuits is essentially the same as that for C/L NCL circuits, described in Section 4.1.3. The only addition is calculating a feedback register's *level*, which should be assigned the same *level* as other registers that share its *Ki* input signal, or 1 level more than its previous register, if its *Ki* input signal is not shared with another register already assigned a level. For the MAC example in Fig. 46, feedback registers 5-8 would be assigned level 1, since they share their *Ki* input with the other level 1 registers, 1-4; and feedback register 15 would be assigned level 2, since it shares its *Ki* input with other level 2 registers, 10-14.

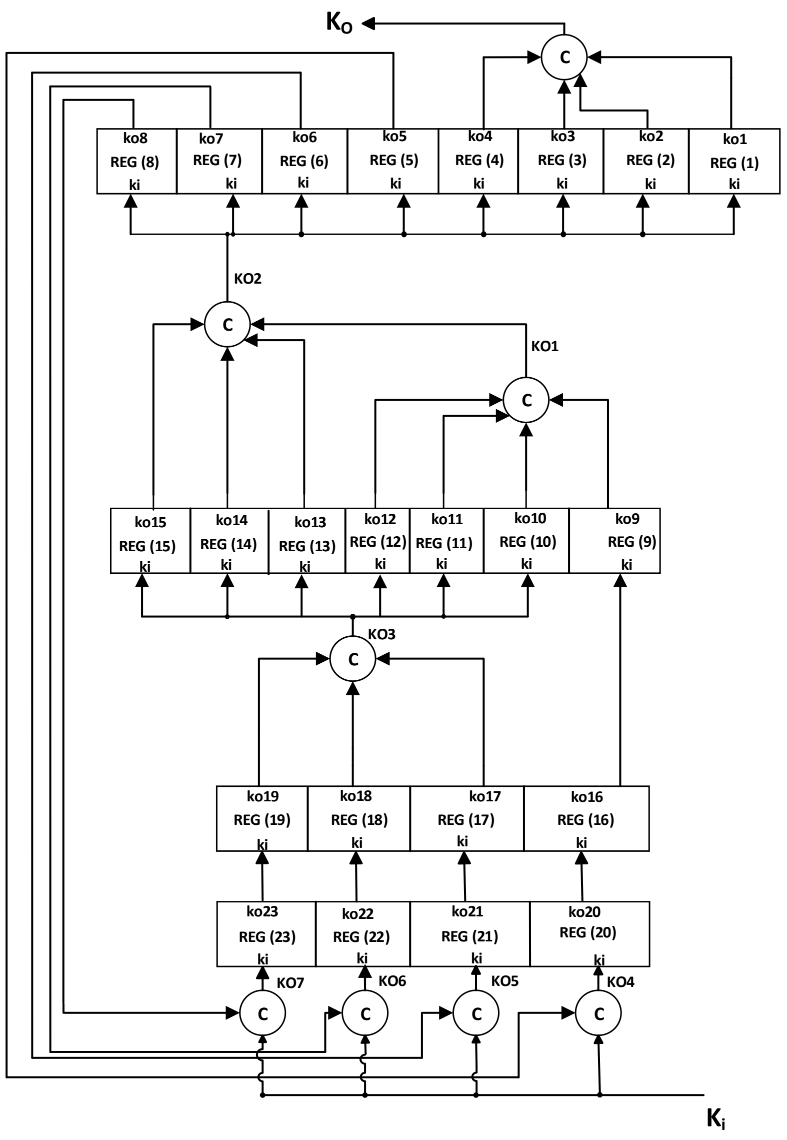


Figure 46. Handshaking Connections for the 4 + 2x2 NCL MAC.

Fig. 47 shows the *reg_fanin* and *ko_fanout* lists for each register in the *4+ 2×2* NCL MAC example. After verifying handshaking correctness, each stage's C/L must also be checked for input-completeness and observability, utilizing the methods detailed in [37].

1: reg_fanin: 0	ko_fanout: 0
2: reg_fanin: 0	ko_fanout: 0
3: reg_fanin: 0	ko_fanout: 0
4: reg_fanin: 0	ko_fanout: 0
5: reg_fanin: [20]	ko_fanout: [20]
6: reg_fanin: [21]	ko_fanout: [21]
7: reg_fanin: [22]	ko_fanout: [22]
8: reg_fanin: [23]	ko_fanout: [23]
9: reg_fanin: [1, 3, 5]	ko_fanout: [1,2,3,4,5,6,7,8]
10: reg_fanin: [1, 3, 5]	ko_fanout: [1,2,3,4,5,6,7,8]
11: reg_fanin: [1, 2, 3, 4, 6]	ko_fanout: [1,2,3,4,5,6,7,8]
12: reg_fanin: [1, 2, 3, 4, 6]	ko_fanout: [1,2,3,4,5,6,7,8]
13: reg_fanin: [1, 2, 3, 4, 7]	ko_fanout: [1,2,3,4,5,6,7,8]
14: reg_fanin: [1, 2, 3, 4, 7]	ko_fanout: [1,2,3,4,5,6,7,8]
15: reg_fanin: [8]	ko_fanout: [1,2,3,4,5,6,7,8]
16: reg_fanin: [9]	ko_fanout: [9]
17: reg_fanin: [10, 11]	ko_fanout: [10,11,…14,15]
18: reg_fanin: [10, 11, 12, 13]	ko_fanout: [10,11,…14,15]
19: reg_fanin: [10,11…,14,15]	ko_fanout: [10,11,…14,15]
20: reg_fanin: [16]	ko_fanout: [16]
21: reg_fanin: [17]	ko_fanout: [17]
22: reg_fanin: [18]	ko_fanout: [18]
23: reg_fanin: [19]	ko_fanout: [19]

Figure 47. *Reg_fanin* and *Ko_fanout* Lists for the 4 + 2x2 NCL MAC.

4.2.3. Sequential NCL Circuit Results

The verification results for sequential NCL circuits, including functional equivalence and handshaking checks, are shown in Table 7. Since the invariant checks are exactly the same for combinational and sequential NCL circuits, these results are not included in Table 7. Test circuits include multiple MAC units and one ISCAS-89 benchmark, s27 [48]. The MAC units are represented as $A + M \times N$, where A, M, and N represent the length of the accumulator, multiplicand, and multiplier, respectively. The same types of bugs were tested for the MACs as tested for the multipliers, and the same machine was used to perform the sequential circuit verification, both as described at the end of Section 4.1.4.

Table 7. Verification Results for Sequential NCL Circuits.

Circuits	**Safety Check (sec.)**	**Handshaking Check (sec.)**	**Total Time (sec.)**
ISCAS s27	0.01	0.0019	0.0119
4+2×2MAC	0.01	0.0045	0.0145
8+4×4MAC	0.05	0.7852	0.8352
12+6×6MAC	0.77	2.331	3.101
16+8×8MAC	47.55	21.7411	69.2911
20+10×10MAC	2643.99	163.6463	2807.6363
20+10×10MAC-B1	0.11 (B)	163.6463	163.7563
20+10×10MAC-B2	0.13 (B)	163.6463	163.7763
20+10×10MAC-B3	2643.99	169.8422 (B)	2813.8322
20+10×10MAC-B4	2643.99	159.3253 (B)	2803.3153
20+10×10MAC-B5	0.20 (B)	163.6463	163.8463

4.3. Conclusions

This chapter presents a novel methodology for formally verifying the correctness (both safety and liveness) of combinational and sequential NCL circuits. The approach includes methods for ensuring handshaking correctness, and functional correctness of both rail^1 and rail^0 outputs, which along with the existing input-completeness and observability check can guarantee correct operation under all timing scenarios. The presented methodology is applicable to both NCL circuits designed using only NCL gates with hysteresis, as well as relaxed NCL circuits, where NCL gates with hysteresis are replaced with their Boolean equivalent gate when hysteresis is not required for input-completeness and/or observability.

5. CONCLUSIONS

The major limiting factor that has hindered the widespread implementation and application of asynchronous paradigms is the lack of design and verification tools. In today's industry, formal verification is of utmost importance in ASIC design flow, and there exists very few formal verification methodologies for asynchronous circuits. The goal of the Ph.D. research work illustrated in the dissertation was to develop unified, scalable, and fast verification methodologies with the potential to meet commercial standards. While scopes and areas of optimization remain to be ventured, the methods developed herein have enormous prospects to be the support tool, which will facilitate the growing interest of asynchronous domain in industry as well as academic research.

5.1. Summary

The Quasi-Delay Insensitive (QDI) design is one of the most commercially successful asynchronous design models. QDI circuits have two major paradigms: NULL Convention Logic (NCL) and Pre-Charge Half Buffers (PCHB). Although both are QDI paradigms, they are structurally very different from each other, as demonstrated in Chapter 2. Different verification methodologies applicable to PCHB and NCL circuits have been discussed in the dissertation.

Formal modeling and verification methodologies for QDI PCHB circuits were demonstrated in Chapter 3. The first developed method was based on model checking, which is a widely utilized formal verification method in industry. To the best of my knowledge, this method was the first published work on verification of combinational PCHB circuits capable of verifying both *safety (*functional correctness) and *liveness* (absence of deadlock). This work had three major contributions: 1) formal modeling of PHCB gates and circuits by developing a set of generic rules to compute the Transition System (TS) of any *n*-input PCHB gate, 2) developing a set of *local* properties applicable locally to every PCHB gate in a circuit to verify the *liveness*, and 3) developing an *N-stage* pipeline architecture to track the DATA inputs and their corresponding outputs to check for the functional equivalence between the Boolean specification and PCHB implementation. The method was demonstrated on different combinational PCHB circuits along with buggy circuits. The approach successfully detected all the bugs and provided counter-examples to trace back the path of error. However, the circuit state space increases exponentially with every added gate, resulting in state-space explosion and an infeasible verification time for higher order circuits. Therefore, scalability remains an issue with the model checking based approach.

To overcome the scalability issue, an alternate verification methodology was proposed for combinational as well as sequential PCHB circuits based on equivalence checking. The method relies on a structural reduction of the PCHB circuit to convert it to an equivalent synchronous circuit. The reduced equivalent circuit is then checked against the synchronous specification utilizing WEB refinement as a notion of functional equivalence ensuring the safety of the circuits. A developed graph based approach checks for the handshaking correctness and liveness of the circuit. An enumeration of all possible structural faults that could occur in a PCHB circuit comprised of PCHB components and C-elements was discussed. The verification method was demonstrated on several different types of circuits, such as, higher order multipliers, multiply and accumulate units, ISCAS combinational benchmarks, and ISCAS sequential benchmarks. It was also proven by demonstration that all possible errors could be detected by the proposed methodology.

The equivalence verification methodology for PCHB circuits was further extended to be applicable to NCL combinational and sequential circuits. The methodology required considerable modifications because of the structural dissimilarities between NCL and PCHB paradigms. The fundamental idea of the safety check based on structural reduction and equivalence checking remains the same. However, the implementation steps were quite different. For NCL circuits, the separated registration units, threshold gates, and both rails' functionalities were added into consideration to develop the safety check model. Two different invariant check mechanisms were discussed to check the rails of each dual rail signals. The first method initializes all the registration and combinational stages as NULL, and flows a symbolic DATA set through all stages to check every stage together, while the second method checks each stage individually. It was shown that the second method was 1.4 times faster than the first approach for a 2-stage 10x10 multiplier. Furthermore, modeling the threshold gates as Boolean functions made the invariant check 2x faster than modeling the gates with hysteresis.

For NCL, the handshaking check remains a graph-based approach similar to PCHB, but requires certain modifications. Mostly because in case of NCL, only registration units along with completion components control handshaking, while in PCHB each component contributes in handshaking control along with C elements. Since PCHB gates consist of dual-rail inputs and output(s), invariant, input-completeness, and observability checking are not required, as these are ensured within the primitive PCHB gates themselves.

5.2. Future Works

As part of our future work, the intention is to tailor the method presented herein to be compatible with existing commercial equivalence checkers, such as Jasper Gold Sequential Equivalence Checker, Cadence Encounter Conformal Equivalence Checker, Calypto SLEC, etc., in order to further improve verification time. This work has the potential to be developed into the first ever commercial equivalence checker for QDI PCHB and NCL circuits.

Additionally, the NCL verification method can be extended to work with embedded registration [24], where some of the NCL registers are combined with the combinational logic to reduce area, latency, and power, and increase throughput.

REFERENCES

1. M. T. Bohr et al., “Interconnect scaling-the real limiter to high performance ulsi,” in *International Electron Devices Meeting*. Institute of Electrical & Electronic Engineers, Inc (IEEE), 1995, pp. 241–244.

2. A. Taubin, J. Cortadella, L. Lavagno, A. Kondratyev, and A. Peeters, “Design automation of real-life asynchronous devices and systems,” *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 1, pp. 1–133, 2007.

3. International Technology Roadmap for Semiconductors, 2013 Edition [Online], http://www.itrs2.net/2013-itrs.html, Accessed on: Feb. 2, 2018.

4. N. Andrikos, L. Lavagno, D. Pandini, and C. P. Sotiriou, “A fully-automated desynchronization flow for synchronous circuits,” in *44th ACM/IEEE Design Automation Conference*, June 2007, pp. 982–985.

5. E. Kilada and K. S. Stevens, “Control network generator for latency insensitive designs,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE ’10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 1773–1778. [Online]. Available: http://dl.acm.org/citation.cfm?id=1870926.1871354.

6. M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, “Asynchronous design using commercial hdl synthesis tools,” in *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, ser. ASYNC ’00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 114–. [Online]. Available: http://dl.acm.org/citation.cfm?id=785166.785308.

7. K. S. Stevens, Y. Xu, and V. Vij, “Characterization of asynchronous templates for integration into clocked cad flows,” in *15th IEEE Symposium on Asynchronous Circuits and Systems (ASYNC),* May 2009, pp. 151–161.

8. R. B. Reese, S. C. Smith, and M. A. Thornton, “Uncle - an rtl approach to asynchronous design,” in *ASYNC*, J. Sparsø, M. Singh, and P. Vivet, Eds. IEEE Computer Society, 2012, pp. 65–72.

9. I. E. Sutherland, “Micropipelines,” *Communications of the ACM*, Vol. 32/6, pp. 720–738, 1989. DOI: 10.1145/63526.63532.

10. S. H. Unger, Asynchronous Sequential Switching Circuits, Wiley, New York, 1969.

11. S. M. Nowick and D. L. Dill, “Synthesis of asynchronous state machines using a local clock,” Proceedings of ICCAD, pp.192-197, 1991.

12. Alain J. Martin, “The limitations to delay-insensitivity in asynchronous circuits,” in *Proceedings of the sixth MIT conference on Advanced research in VLSI (AUSCRYPT '90)*, William J. Dally (Ed.). MIT Press, Cambridge, MA, USA, 263-278.

13. C. L. Seitz, “System timing,” *in Introduction to VLSI Systems*, Addison-Wesley, pp. 218–262, 1980.

14. T. S. Anantharaman, “A delay insensitive regular expression recognizer,” in *IEEE VLSI Technical Bulletin*, Sept. 1986.

15. N. P. Singh, “A design methodology for self-timed systems,” *Master’s Thesis*, MIT/LCS/TR258, Laboratory for Computer Science, MIT, 1981.

16. J. Sparso, J. Staunstrup, M. Dantzer-Sorensen, “Design of delay insensitive circuits using multi-ring structures,” in *Proceedings of the European Design Automation Conference*, pp. 15–20, 1992. DOI: 10.1109/EURDAC.1992.246271.

17. I. David, R. Ginosar, and M. Yoeli, "An efficient implementation of Boolean functions as self-timed circuits," in *IEEE Transactions on Computers*, Vol. 41/1, pp. 2–10, 1992. DOI: 10.1109/12.123377.

18. D. E. Muller, "Asynchronous logics and application to information processing," in *Switching Theory in Space Technology*, Stanford University Press, pp. 289-297, 1963.

19. K. M. Fant and S. A. Brandt, "NULL Convention Logic: a complete and consistent logic for asynchronous digital circuit synthesis," in *International Conference on Application Specific Systems, Architectures, and Processors*, pp. 261–273, 1996. DOI: 10.1109/ASAP.1996.542821.

20. D. H. Linder and J. H. Harden, "Phased Logic: supporting the synchronous design paradigm with delay-insensitive circuitry," in *IEEE Transactions on Computers*, Vol. 45/9, pp. 1031–1044, 1996. DOI: 10.1109/12.537126.

21. A. J. Martin, M. Nystrom, "Asynchronous techniques for system-on chip design," *Proceedings of the IEEE*, vol. 94/6, pp. 1089-1120, 2006.

22. S. C. Smith, "The future of asynchronous logic", *Circuit Advances & Emerging Technologies Track 2015 CMOS Emerging Technologies Research Conference Presentation Slides*, vol. 2, pp. 46, May 2015.

23. L. Zhou and S. C. Smith, "Static implementation of Quasi-Delay-Insensitive Pre-Charge Half-Buffers", in *IEEE Midwest Symposium on Circuits and Systems*, 2010, pp. 636-639.

24. S. C. Smith and J. Di, *Designing Asynchronous Circuits using NULL Convention Logic (NCL),* ser. Synthesis Lectures on Digital Circuits and Systems. Morgan & Claypool Publishers, 2009.

25. Scott C. Smith, "Completion-Completeness for NULL convention digital circuits utilizing the bit-wise completion strategy," *in International Conference on VLSI*, 2003, pp. 143-149.

26. Gerald E. Sobelman and Karl M. Fant, “CMOS circuit design of threshold gates with hysteresis,” in *IEEE International Symposium on Circuits and Systems* (II), pp. 61-65, 1998.

27. Moon, C.W., Stephan, P.R. & Brayton, R.K, “Specification, synthesis, and verification of hazard-free asynchronous circuits,” in *Journal of VLSI Signal Processing (1994)* 7: 85. https://doi.org/10.1007/BF02108191.

28. C. J. Myers, *Asynchronous Circuit Design.* New York: Wiley, 2001.

29. A. Semenov and A. Yakovlev, "Verification of asynchronous circuits using time petri net unfolding," in *33rd Design Automation Conference Proceedings, 1996*, Las Vegas, NV, USA, 1996, pp. 59-62.

30. F. Verbeek and J. Schmaltz, “Verification of building blocks for asynchronous circuits,” *in ACL2*, ser. EPTCS, R. Gamboa and J. Davis, Eds., vol. 114, 2013, pp. 70–84.

31. A. Peeters, F. te Beest, M. de Wit & W. Mallon, “Click elements: An implementation style for data-driven compilation,” *in Proceedings of the IEEE Symposium on Asynchronous Circuits and Systems (ASYNC’10),* pp. 3-14.

32. S. K. Srinivasan and R. S. Katti, “Desynchronization: design for verification,” *in FMCAD*, P. Bjesse and A. Slobodova, Eds. FMCAD ´ Inc., 2011, pp. 215–222.

33. Vidura M. Wijayasekara, S. K. Srinivasan and S. C. Smith, "Equivalence verification for NULL Convention Logic (NCL) circuits," in *ICCD*, 2014, pp. 195-201.

34. P. Manolios, “Correctness of pipelined machines,” in *FMCAD* 2000, ser. LNCS, W. A. Hunt, Jr. and S. D. Johnson, Eds., vol. 1954. Springer-Verlag, 2000, pp. 161–178.

35. C. Jeong and S. M. Nowick, “Optimization of robust asynchronous circuits by local input completeness relaxation,” *in 2007 Asia and South Pacific Design Automation Conference*, Jan 2007, pp. 622–627.

36. A. Kondratyev, L. Neukom, O. Roig, A. Taubin, and K. Fant, “Checking delay-insensitivity: 104 gates and beyond,” *in Proceedings Eighth International Symposium on Asynchronous Circuits and Systems*, April 2002, pp. 149–157.

37. A. A. Sakib, S. Le, S. C. Smith, and S. K. Srinivasan, “Chapter 15: Asynchronous Circuit Verification,” Asynchronous Circuit Applications, Institution of Engineering and Technology (IET), London, UK (Accepted).

38. S.J. Longfield, R. Manohar, “Inverting Martin Synthesis for verification”, *in ASYNC*, 2013, pp. 150-157.

39. C. Shih, Y. Lai and J. R. Jiang, “SPOCK: Static performance analysis and deadlock verification for efficient asynchronous circuit synthesis”, *in IEEE/ACM ICCAD,* 2015, pp. 442-449.

40. C. Chuang, Y. Lai, J. R. Jiang, “Synthesis of PCHB-WCHB hybrid quasi-delay insensitive circuits”, *in Design Automation Conference (DAC)*, 2014, pp. 1-6.

41. B. Ghavami, H. Zarandi, A. Salrpour, H. Pedram, "Diagnosis of faults in template based asynchronous circuits", *International Symposium on System-on-Chip*, pp. 38-41, 2009.

42. Edmund M. Clarke, Orna Grumberg, Doron A. Peled, Model Checking, MIT Press, 1999.

43. "NuSMV: a new symbolic model checker", *NuSMV home page*, [online] Available: http://nusmv.fbk.eu.

44. L. M. de Moura and N. Bjørner, “Z3: An efficient smt solver*”, in TACAS, ser. Lecture Notes in Computer Science*, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

45. D. Bryan, “The ISCAS '85 benchmark circuits and netlist format,” [Online], https://ddd.fit.cvut.cz/prj/Benchmarks/iscas85.pdf, Accessed on: December 4, 2018.

46. A. J. Martin, A. Lines, R. Manohar, U. Cummings, and M. Nystroem, “Pipelined asynchronous processing,” U.S. Patent: 6,658,550, Dec. 2, 2003.

47. P. Prakash, A. J. Martin, “ Slack matching quasi-delay-insensitive circuits”, in *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC),* 2006, pp. 10-204.

48. [Online] http://www.pld.ttu.ee/~maksim/benchmarks/iscas89/bench/, Accessed on: Dec 4, 2018.

49. D. Monniaux, “ A survey of Satisfiability Modulo Theory,” [online]. Available: https://hal.archives-ouvertes.fr/hal-01332051/document [Accessed on 30 Apr 2019]

APPENDIX. LIST OF PUBLICATIONS

- Book Chapter: A. A. Sakib, S. Le, S. C. Smith, and S. K. Srinivasan, “Chapter 15: Formal Verification of NCL Circuits”, Asynchronous Circuit Applications, Institution of Engineering and Technology (IET), London, UK, 2019 (Accepted).
- Refereed Journal: A. A. Sakib, S. C. Smith, and S. K. Srinivasan, “Formal Modeling and Verification of PCHB Asynchronous Circuits”, in IEEE Transactions on VLSI (under review).
- Refereed Technical Conference: A. A. Sakib, S. C. Smith, and S. K. Srinivasan, "Formal Modeling and Verification for Pre-Charge Half Buffer Gates and Circuits," IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), 2017, pp. 519-522.
- Refereed Technical Conference: A. Sakib, S. C. Smith, and S. K. Srinivasan, "An Equivalence Verification Methodology for Combinational Pre-Charge Half Buffer Asynchronous Circuits.” IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), 2018, pp. 767-770.
- Refereed Technical Conference: M. Hossain, A. A. Sakib, S. C. Smith, and S. K. Srinivasan, "An Equivalence Verification Methodology for Asynchronous Sleep Convention Logic Circuits,” IEEE International Symposium on Circuits and Systems (ISCAS), 2019, pp. 1-5.