FORMAL VERIFICATION METHODOLOGIES FOR NULL CONVENTION LOGIC CIRCUITS

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

By

Son Ngoc Le

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

Major Department:
Electrical and Computer Engineering

June 2020

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Formal Verification Methodologies for NULL Convention Logic Circuits

By

Son Ngoc Le

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:

Sudarshan K. Srinivasan
Chair

Scott C. Smith
Co-Chair

Dharmakeerthi Nawarathna

Kenneth Magel

Approved:

07/01/2020
Date

Benjamin Braaten
Department Chair

ABSTRACT

NULL Convention Logic (NCL) is a Quasi-Delay Insensitive (QDI) asynchronous design paradigm that aims to tackle some of the major problems synchronous designs are facing as the industry trend of increased clock rates and decreased feature size continues. The clock in synchronous designs is becoming increasingly difficult to manage and causing more power consumption than ever before. NCL circuits address some of these issues by requiring less power, producing less noise and electro-magnetic interference, and being more robust to Process, Voltage, and Temperature (PVT) variations. With the increase in popularity of asynchronous designs, a formal verification methodology is crucial for ensuring these circuits operate correctly. Four automated formal verification methodologies have been developed, three to ensure delay-insensitivity of an NCL circuit (i.e., prove Input-Completeness, Observability, and Completion-Completeness properties), and one to aid in proving functional equivalence between an NCL circuit and its synchronous counterpart. Note that an NCL circuit can be functionally correct and still not be input-complete, observable, or completion-complete, which could cause the circuit to operate correctly under normal conditions, but malfunction when circuit timing drastically changes (e.g., significantly reduced supply voltage, extreme temperatures). Since NCL circuits are implemented using dual-rail logic (i.e., 2 wires, $rail^0$ and $rail^1$, represent one bit of data), part of the functional equivalence verification involves ensuring that the NCL $rail^0$ logic is the inverse of its $rail^1$ logic. Equivalence verification optimizations and alternative invariant checking methods were investigated and proved to decrease verification times of identical circuits substantially. This work will be a major step toward NCL circuits being utilized more frequently in industry, since it provides an automated verification method to prove correctness of an NCL implementation and equivalence to its synchronous specification, which is the industry standard.

ACKNOWLEDGEMENTS

This dissertation is based upon work supported by the National Science Foundation under Grant No. CCF-1717420.

DEDICATION

To my parents, Ngoc and Tammy Le.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

DEDICATION v

LIST OF TABLES viii

LIST OF FIGURES ix

LIST OF ABBREVIATIONS x

1. INTRODUCTION 1

1.1. Formal Verification Overview 1

1.2. NULL Convention Logic (NCL) Overview 2

1.3. Dissertation Work 10

1.4. Related Work 11

2. NCL EQUIVALENCE VERIFICATION 13

2.1. Equivalence Verification for Combinational NCL Circuits 14

2.1.1. Functional Equivalence Check 15

2.1.2. Invariant Check 20

2.1.3. Combinational NCL Circuits Results 23

2.2. Equivalence Verification for Sequential NCL Circuits 23

2.2.1. NCL to Synchronous Reduction 23

2.2.2. Exploiting Dual-Rail Invariants for Equivalence Verification 34

2.3. Equivalence Verification Conclusion 39

3. NCL INPUT-COMPLETENESS VERIFICATION 42

3.1. Input-Completeness Verification 43

3.1.1. Input-Completeness Proof Obligation: NULL to DATA 43

3.1.2. Input-Completeness Proof Obligation: DATA to NULL 44

3.1.3. Input-Completeness Results 46

3.1.4. Input-Completeness Conclusion 48

4. NCL OBSERVABILITY VERIFICATION 50

4.1. Observability Verification 50

4.1.1. Observability Proof Obligation: NULL to DATA 50

4.1.2. Observability Proof Obligation: DATA to NULL 51

4.1.3. Observability Results 53

5. NCL COMPLETION-COMPLETENESS VERIFICATION 56

5.1. Completion-Completeness Previous Work 56

5.2. Completion-Completeness Verification 58

5.2.1. Completion-Completeness Proof Obligation 59

5.2.2. Completion-Completeness Results 61

5.2.3. Completion-Completeness Conclusion 62

6. CONCLUSION 64

6.1. Summary 64

6.2. Future Work 65

REFERENCES 67

APPENDIX. LIST OF PUBLICATIONS 70

LIST OF TABLES

Table Page

1. Dual-rail signal representation [5] 3
2. Fundamental NCL gates [5] 4
3. Predicates for invariant check 21
4. Predicates for revised invariant check 22
5. Verification results of various C/L NCL circuits (in sec.) 24
6. Verification results for sequential NCL circuits (in sec.) 34
7. Predicates for equivalence check 37
8. Dual-rail refinement results 38
9. Predicates for input-completeness check 45
10. Verification results of input-completeness (in sec.) 48
11. Predicates for observability check 52
12. Verification results of observability (in sec.) 54
13. Predicates for completion-completeness check 60
14. Verification results of completion-completeness (in sec.) 63

LIST OF FIGURES

Figure Page

1. NCL system framework [5] 3
2. Single-bit dual-rail register [5] 5
3. N-bit completion detection [5] 6
4. Full-word completion [8] 7
5. Bit-wise completion [8] 8
6. Input-incomplete dual-rail two-input AND circuit [5] 9
7. Input-complete dual-rail two-input AND circuit [8] 9
8. Unobservable dual-rail two-input XOR circuit [5] 10
9. NCL half-adder 16
10. NCL full-adder [5] 16
11. 3×3 NCL multiplier 17
12. (a) 3×3 NCL multiplier netlist, (b) Converted Boolean netlist 18
13. Multiply and accumulate (MAC) circuit: (a) Synchronous (b) NCL 25
14. 4 + 2 × 2 NCL MAC datapath 26
15. (a) 4 + 2 × 2 NCL MAC netlist, (b) Converted synchronous equivalent netlist 28
16. Depiction of proof obligation to check equivalence of NCL_SYNC and SPEC_SYNC netlists 30
17. Handshaking connections for the 4+2×2 NCL MAC 32
18. reg_fanin and ko_fanout lists for the 4+2×2 NCL MAC 33
19. (a) 3×3 NCL multiplier netlist (b) Converted netlist using method in Section 2 (c) Converted netlist using proposed Register Invariance Refinement 40
20. ISCAS-85 C432 M1 module nine-input NCL NAND that generates PA 55
21. Completion-incomplete NCL circuit 58

LIST OF ABBREVIATIONS

DI ....Delay-Insensitivity

EMI ....Electro-magnetic Interference

FA(s) ....Full-Adder(s)

HA(s)..Half-Adder(s)

IoT..Internet of Things

MAC ....Multiply and Accumulate

MTNCL ....Multi-Threshold NULL Convention Logic

NCL..NULL Convention Logic

PVT ....Process, Voltage, Temperature

QDI ....Quasi-Delay Insensitive

rfn...Request for NULL

rfd...Request for DATA

RTL ....Register-Transfer Level

SCL ....Sleep Convention Logic

SMT ....Satisfiability Modulo Theory

SMT-LIB..Satisfiability Modulo Theory Library

SOP ....Sum of Products

TO ....Timeout

TS(s)...Transition System(s)

WEB...Well-Founded Equivalence Bisimulation

WSN ....Wireless Sensor Networks

1. INTRODUCTION

1.1. Formal Verification Overview

Formal verification is an alternative approach to validation of a circuit design. The correctness of formal methodologies is based on mathematical proofs instead of individual test cases, which are used in the traditional test-based approach to validation. The benefit of using a formal method over the traditional test-based approach is that a single proof can be used to cover many test cases. As circuit size increases, the number of test cases for a system increases exponentially making it increasingly difficult to achieve an acceptable amount of test coverage using traditional methods. Due to this fact, formal verification has been shown to be crucial to ensuring design correctness and finding corner-case bugs that can be easily missed using the traditional test methodology. The semiconductor industry has started incorporating formal methods into their design cycle for validation after the floating-point bug was found on the Intel Pentium processor in 1994, which cost Intel $500 million to fix.

One of the more popular formal verification approaches that has been found to be extremely scalable and useful in semiconductor design is equivalence checking. Typically, a lot of time, money, and effort is invested into ensuring the correctness of a design. However, the design itself is never static, as it is continuously tinkered with and optimized. Equivalence checking technology can, with a high degree of automation and efficiency, check that the golden model (i.e., the design that has been extensively validated) and its derivative are functionally equivalent. Scalability is harnessed by exploiting the structural similarity of the golden model and its derivative. Examples of commercial equivalence checkers include IBM Sixth Sense, Jasper Gold Sequential Equivalence Checker, Calypto SLEC, Mishchenko EBCCS13, and Cadence Encounter Conformal Equivalence Checker.

1.2. NULL Convention Logic (NCL) Overview

Traditional digital design is reliant on operating clocks that dictate when data needs to be available in a circuit. To increase a circuit's speed the operating clock speed is increased. This along with the desire to have less area have caused many clock-related issues. These issues require the circuit designer to have more and more area dedicated to fix these clock related issues causing high power consumption and high complexity when designing. Asynchronous designs, like NULL Convention Logic (NCL) [1], are clockless designs and have benefits over their synchronous counterparts due to the lack of clocking within a circuit. Asynchronous circuits require less power, generate less noise, and produce less electro-magnetic interference (EMI) than the traditional clocked circuits.

NCL is a Quasi-Delay Insensitive (QDI) asynchronous design style that has been demonstrated to function in environments characterized by high radiation and extreme temperatures, both high and low, and is also very robust to process and voltage variations, all of which can cause traditional synchronous circuits to fail due to circuit timing [2]. Being able to function correctly in these extreme environments makes NCL designs very suitable for space exploration, the power industry, the automobile industry (internal combustion engines), oil/gas exploration, medical imaging instrumentation, the laser industry, superconducting computing and energy storage systems, and low voltage or low power applications such as wireless sensor networks (WSN) or Internet of Things (IoT).

The NCL system framework, shown in Fig. 1, depicts how an NCL circuit operates. In short, the delay-insensitive (DI) combinational logic is placed between DI registers. These registers use local handshaking and completion components to request and acknowledge alternating DATA and NULL wavefronts, as further explained below. To achieve delay-

insensitivity, meaning the circuit will operate regardless of when circuit inputs become available, NCL is typically constructed using dual-rail signals. A dual-rail signal consists of two wires, called rails, with $rail^0$ and $rail^1$ representing the two wires. With these two wires, four states are possible: 0b00 is known as the NULL state or absence of data; 0b01 and 0b10 are the DATA0 and DATA1 state corresponding to a Boolean 0 and 1, respectively; and 0b11 is an ILLEGAL state that will never occur in a properly operating circuit. The states of a dual-rail signal are shown in Table 1.

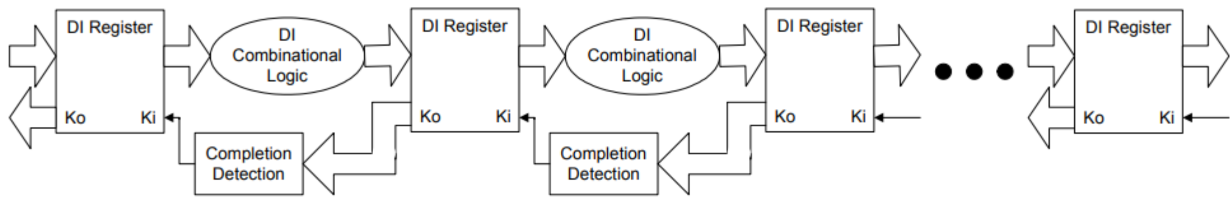


Figure 1. NCL system framework [5]

Table 1. Dual-rail signal representation [5]

	DATA0	DATA1	NULL	ILLEGAL
$rail^0$	1	0	0	1
$rail^1$	0	1	0	1

NCL consists of 27 fundamental gates, with each of these gates having a state-holding capability called hysteresis, meaning that once the gate becomes asserted, it stays asserted until all the inputs are de-asserted. These gates, called threshold gates, are described as $THmn$, where $1 \leq m \leq n$. n represents the number of inputs to the gate; and m represents the threshold value (i.e. how many of the inputs that must be asserted for the output to be asserted). There is also a weight mechanism which gives certain inputs more influence or weight in calculating whether the threshold has been reached. These gates are depicted as $THmnWw_1, \ldots, w_R,\ R < N$, where $w_1, \ldots, w_R$ are the integer weights of inputs that are more than weight 1. The function to assert each of these 27 gates is shown in Table 2.

Table 2. Fundamental NCL gates [5]

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

NCL circuits function under the propagation of NULL and DATA waves through the circuit. These waves contain handshaking protocols that are used to replace the clock in traditional designs. To ensure that DATA from one wavefront does not overwrite the DATA of another wavefront, a minimum of two register stages are required at the input and output of the circuit. Each single-bit dual-rail register component, as seen in Fig. 2, has seven signals. I^0 and I^1 represent the dual-rail input; O^0 and O^1 represent the dual-rail output; Reset is used to set the register output into a known state (i.e., registers can be reset to three states, NULL, DATA0, or

DATA1; and these are referred to as Reset-to-NULL, Reset-to-DATA0, and Reset-to-DATA1 registers, respectively); and K_i and K_o are handshaking signals used for communication between register stages. When the register output is NULL and needing DATA, the K_o value is 1, known as request for DATA (*rfd*). When in a DATA state and needing NULL, the K_o value is 0, known as request for NULL (*rfn*). The K_o values from each register are fed through Completion Detection circuitry, as shown in Fig. 3, to determine whether the current register stage needs a DATA wavefront or a NULL wavefront. The output generated from the Completion Detection circuitry is fed to the K_i inputs of the previous register stage.

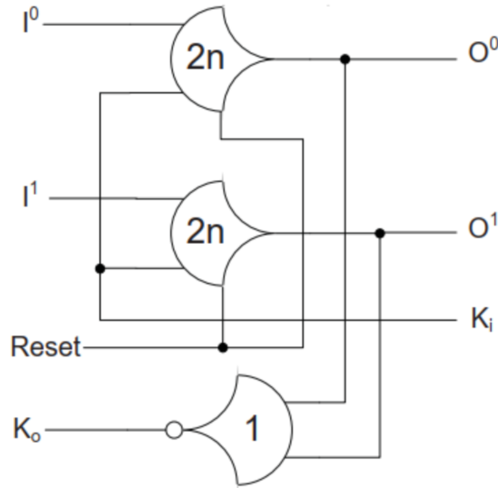


Figure 2. Single-bit dual-rail register [5]

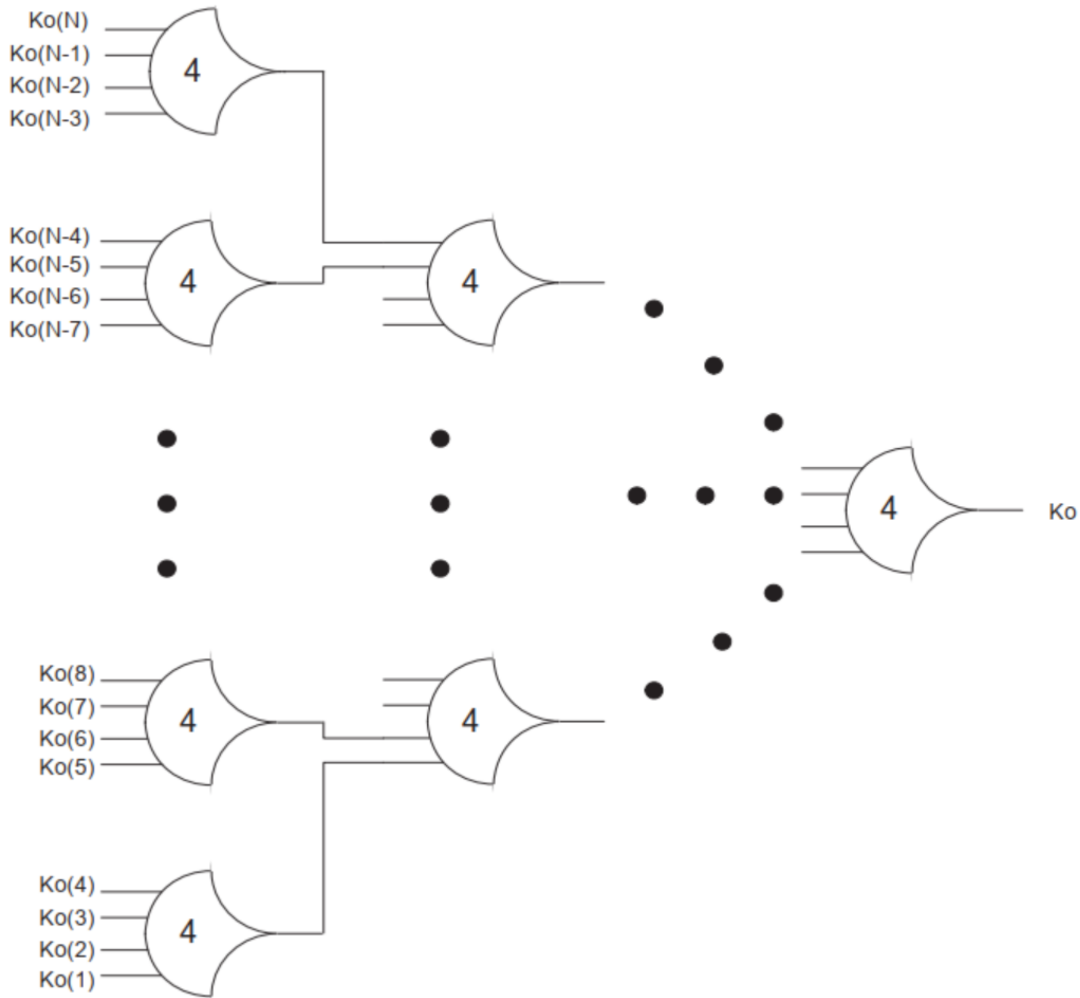


Figure 3. N-bit completion detection [5]

When pipelining two completion strategies can be used: full-word and bitwise-completion. The decision on which completion type to use is circuit dependent, with bit-wise completion having the possibility of reducing completion logic delay to increase throughput of the circuit, or possibly decreasing area. When implementing full-word completion, all the acknowledge bits from $register_i$ are fed into the same Completion Detection circuit to generate a single output that is then fed to all the request signals of $register_{i-1}$. When using bit-wise completion, a separate Completion Detection circuit is used to generate each request signal of $register_{i-1}$, whose inputs are only the acknowledge signals of the $register_i$ outputs calculated using

that particular $register_{i-1}$ output. Full-word and bit-wise completion strategies are demonstrated using Fig. 3 and 4, respectively. In Fig. 3, only one Completion Detection unit is required, which combines all the $register_i$'s Ko signals. Its output is then fed to every Ki signal in $register_{i-1}$. In contrast, there are four Completion Detection units in Fig. 4, one for each bit in $register_{i-1}$.

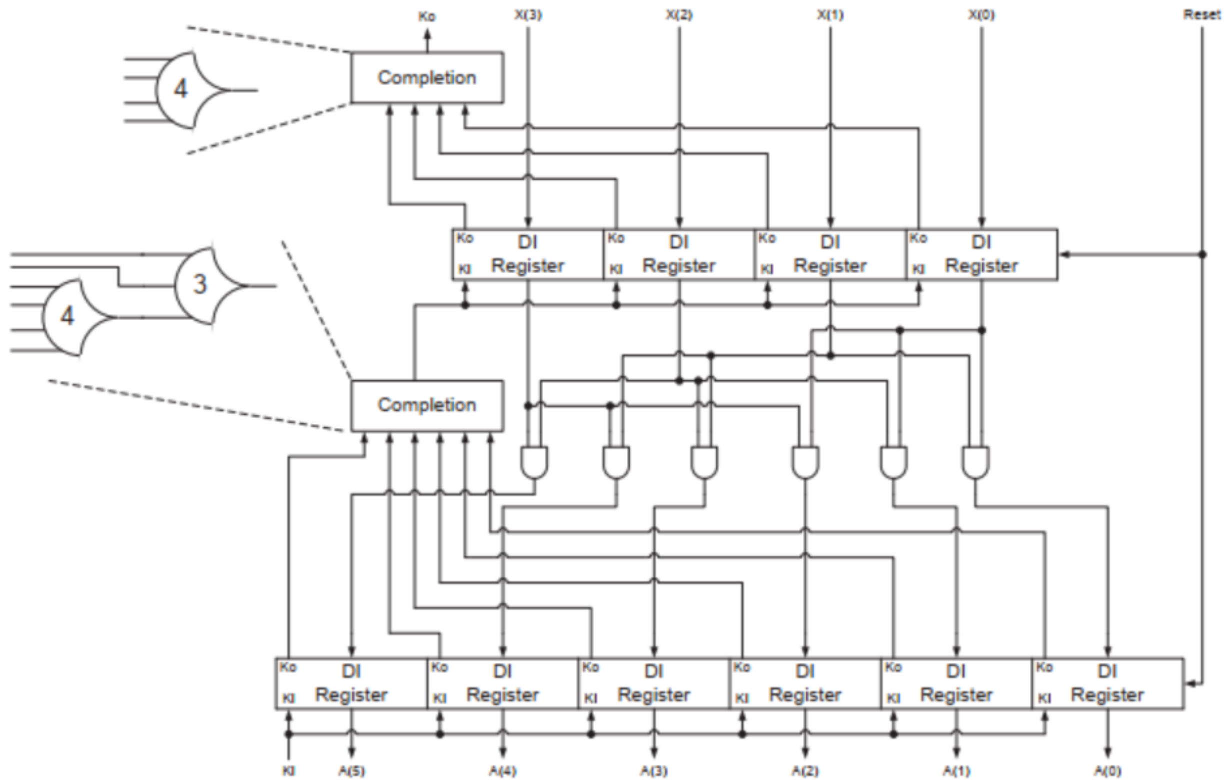


Figure 4. Full-word completion [8]

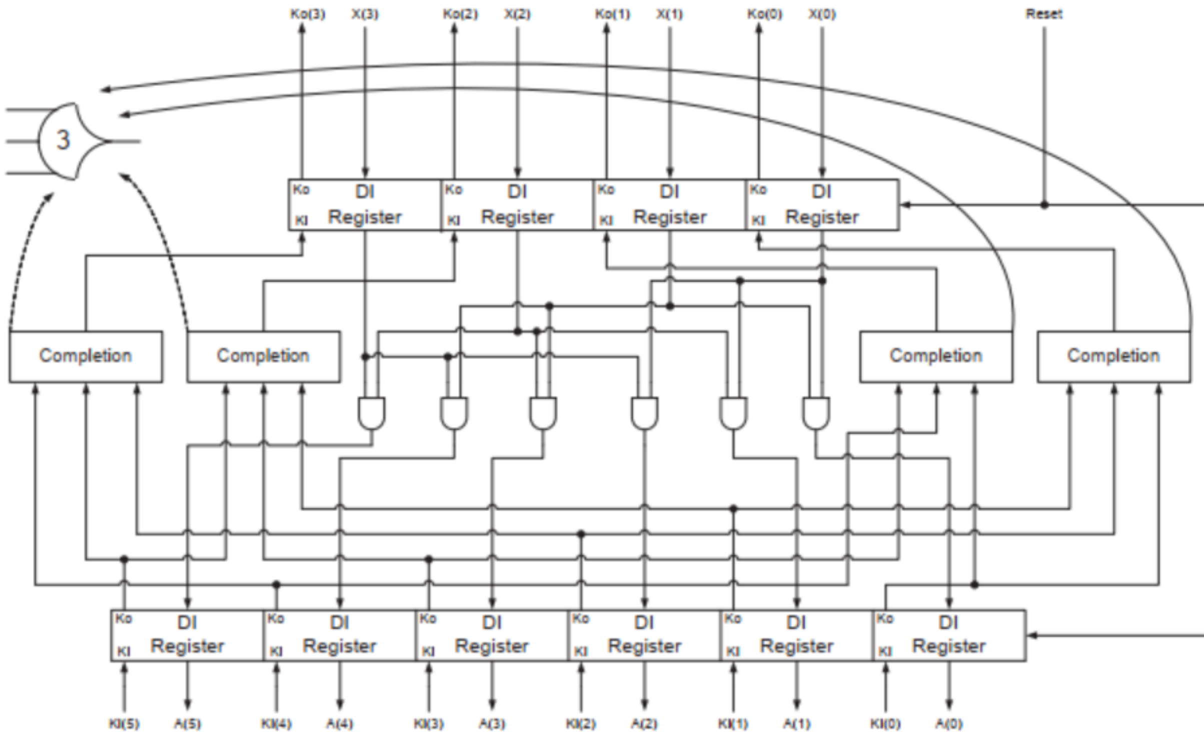


Figure 5. Bit-wise completion [8]

Aside from the logic being correct, NCL circuits must satisfy the following properties to ensure delay-insensitivity: input-completeness, observability, and completion-completeness. Input-completeness is a property that states that an NCL circuit's output may only transition from NULL to DATA after all of its inputs have transition from NULL to DATA, and conversely, that an NCL circuit's output may only transition from DATA to NULL after all of its inputs have transition from DATA to NULL. According to Seitz's "weak conditions", some of the outputs can transition in a circuit as long as at least one remains untransitioned until all inputs arrive. Violation of this property can be shown with the input-incomplete NCL AND function shown in Fig. 6. Assume the circuit is in a NULL state, and inputs X and Y are DATA0 and NULL, respectively; the TH12 gate would assert, therefore asserting Z^0 and making Z become DATA0, which violates the input-completeness property because the output Z has transitioned

from NULL to DATA before Y has transitioned from NULL to DATA. Contrarily, the output Z of the input-complete AND2 shown in Fig. 7 cannot transition until both inputs have become DATA, therefore making it input-complete even though the two implementations are functionally equivalent.

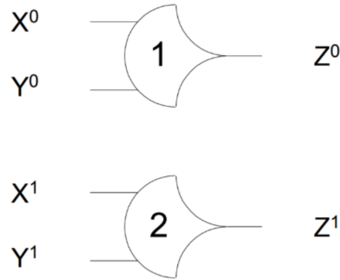


Figure 6. Input-incomplete dual-rail two-input AND circuit [5]

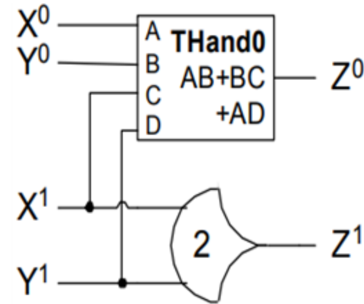


Figure 7. Input-complete dual-rail two-input AND circuit [8]

Observability is another property that must be satisfied for NCL circuits to be delay-insensitive. The observability property states that no orphans may propagate through a gate. An orphan is defined as a wire that transitions during the DATA wavefront but is not used to determine the output. This means that if a gate transitions from NULL to DATA or from DATA to NULL, that transition must be necessary to transition one of the outputs. Violation of this property can be shown in Fig. 8. Assume both X and Y are DATA0, which asserts the TH12 gate; however, the output of this gate is only connected to the TH33w2 gate that determines Z^1, which will not be asserted for this scenario, as the TH23w2 gate that determines Z^0 will instead be asserted. Hence, the TH12 gate is not observable, which can lead to an incorrect output when timing changes (i.e., the circuit is not delay-insensitive).

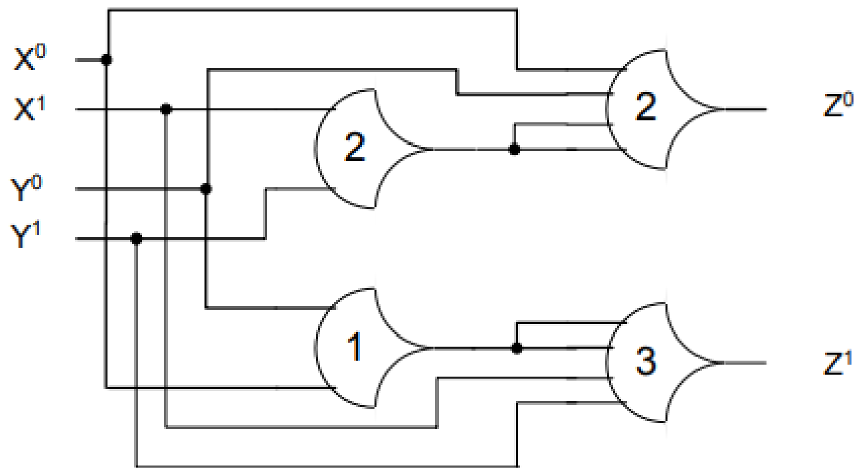


Figure 8. Unobservable dual-rail two-input XOR circuit [5]

Completion-Completeness is a property that must be checked for NCL circuits that utilize bit-wise completion along with some input-incomplete logic functions. It states that completion signals must be generated such that two adjacent DATA wavefronts cannot interact within a combinational logic component. This property is automatically satisfied when using full-word completion, since there is only a single handshaking signal for each stage that allows a DATA or NULL wavefront to enter the stage. Hence, as long as the circuit is input-complete, it is also completion-complete when using full-word completion. When using bit-wise completion, the input-completeness criterion does not fully ensure delay-insensitivity, because there are multiple request signals for each stage, which change at different times as the stage outputs change.

1.3. Dissertation Work

The contents of this dissertation can be split into two main formal verification categories, equivalence checking, and checking of NCL properties, which are divided into their respective chapters. Chapter 2 presents two methods used to decrease equivalence verification time of NCL

circuits. Chapter 3, 4, and 5 all revolve around defining formal properties to check for input-completeness, observability, and completion-completeness of a circuit, respectively.

1.4. Related Work

Vidura et al. [3] 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 [4] 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. 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 signal transitioning order of QDI paradigms.

A manual approach to checking input-completeness is outlined in [5]. To check a circuit for input-completeness, an analysis has to be done on 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 [6], 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. [7] 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. Our approach for observability checking, detailed in Section 4, is very similar to [7], while our approach checks input-completeness for all inputs simultaneously using only two proof obligations, as detailed in Section 3. The completion-completeness property was demonstrated to be required for NCL circuits utilizing bit-wise completion in [8], and methods were presented to ensure that circuits were completion-complete; however, an algorithm to determine whether a circuit was completion-complete was not included.

2. NCL EQUIVALENCE VERIFICATION[1]

In working with the approach described in [3], we 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, which makes verification intractable.

We have therefore developed an alternate approach to circumvent having to deal with the NCL TS. 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 [9]. 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.

---

[1] The functional equivalence check and invariant checks documented in this chapter 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. The dual-rail invariant work was done independently by Son.

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.

2.1. 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 3 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.

2.1.1. Functional Equivalence Check

A 3×3 NCL multiplier, shown in Fig. 11, 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. 6, 7, 9, and 10, respectively. All signals in Fig. 11 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. 11 includes one intermediate register stage to increase throughput.

The netlist of the NCL 3×3 multiplier is shown in Fig. 12(a). 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.

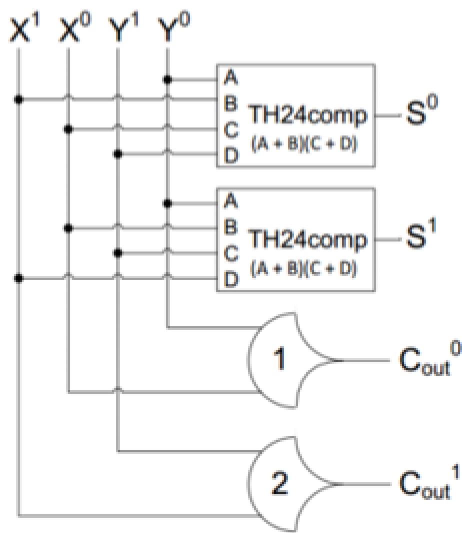


Figure 9. NCL half-adder

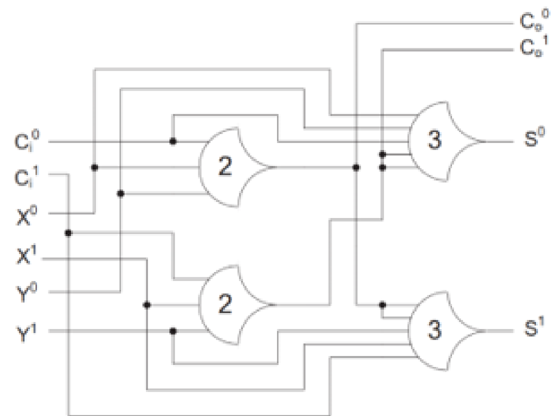


Figure 10. NCL full-adder [5]

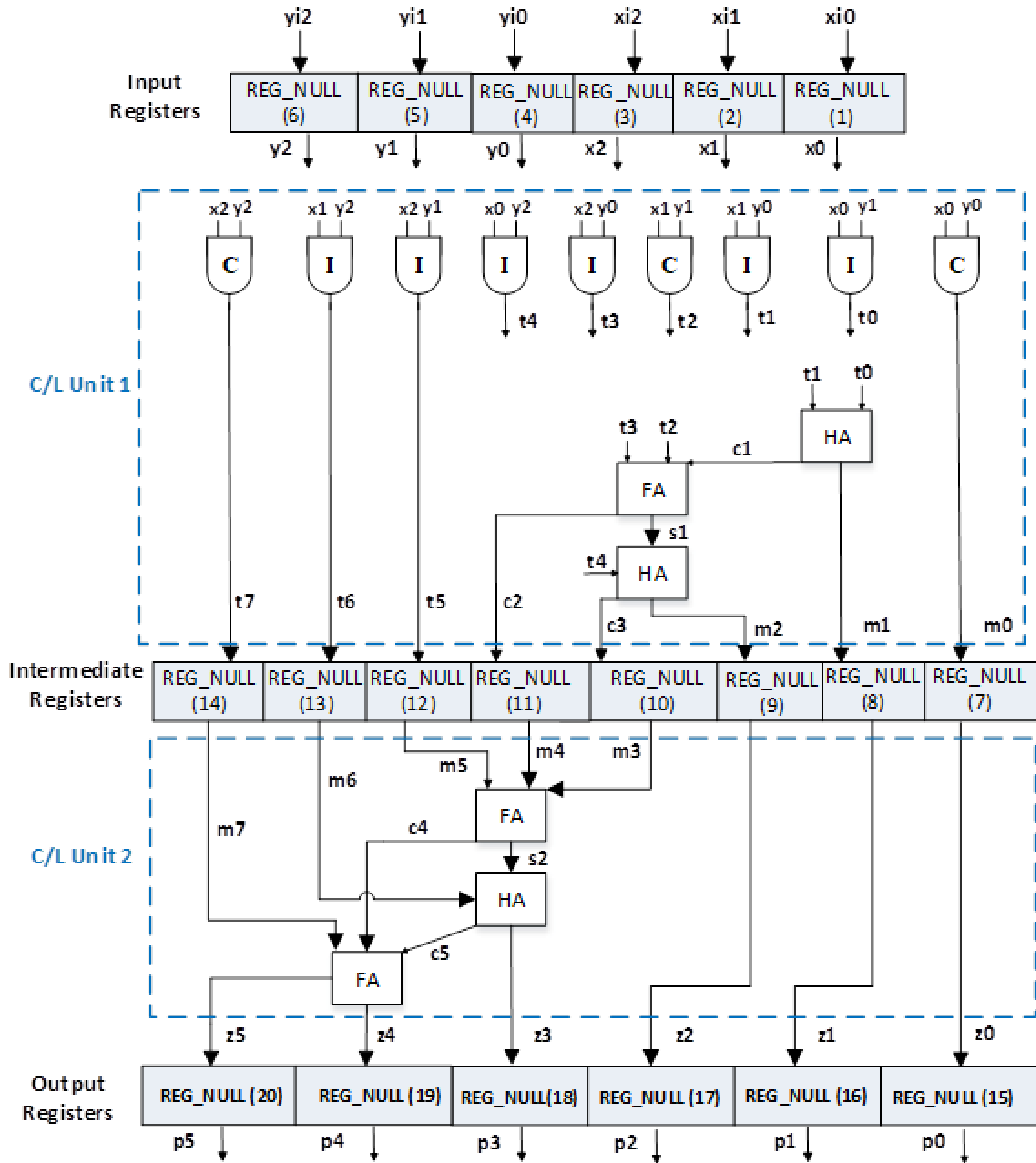


Figure 11. 3×3 NCL multiplier

```
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
```

(a)

```
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
```

(b)

Figure 12. (a) 3×3 NCL multiplier netlist, (b) Converted Boolean netlist

The NCL netlist is input to a conversion algorithm that converts it into an equivalent Boolean netlist, as shown in Fig. 12(b) for the Fig. 11 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 two 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. 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 2: Equivalence Verification for Sequential NCL Circuits.

The converted Boolean netlist is automatically encoded in the Satisfiability Modulo Theory Library (SMT_LIB) language [9], 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 3×3 multiplier example, the SMT solver checks for the following safety property: $F_{NCL_Bool_Equiv.}$ (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 [10] 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.

2.1.2. 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 $r^1, \dots, r^j$, k C/L threshold gates $g^1, \dots, g^k$, q dual-rail inputs $i^1, \dots, i^q$, and l dual-rail outputs $o^1 < R^0, R^1 >, \dots, o^l < R^0, R^1 >$, where R^0 and R^1 are the output's rail^0 and rail^1, respectively. The predicates for this invariant check are shown in Table 3. p_0 indicates that all registers in are reset-to-NULL. p_1 and p_2 state that all threshold gates and *Ki* register inputs are initialized to logic 0 and 1, respectively. p_3 indicates that all inputs are DATA. p_4 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 $({g_B}^1, \dots, {g_B}^k)$. p_5 states that the rails of each

dual-rail output are complements of each other. The proof obligation, PO0, 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.

Table 3. Predicates for invariant check

$\boldsymbol{p_n}$	**Predicate**
p_0	$\bigwedge_{n=1}^{n=j} ({r_A}^n = 0b00)$
p_1	$\bigwedge_{n=1}^{n=k} ({g_A}^n = 0)$
p_2	$\bigwedge_{n=1}^{n=j} ({Ki_A}^n = 1)$
p_3	$\bigwedge_{n=1}^{n=q} (({i_A}^n = 0b01) \vee ({i_A}^n = 0b10))$
p_4	$({g_B}^1, \dots, {g_B}^k) = NCLStep({i_A}^1, \dots, {i_A}^q)$
p_5	$\bigwedge_{n=1}^{n=l} {o_B}^n < R^0 > \neq {o_B}^n < R^1 >$

$$\textbf{PO0}: \{\, p_0 \wedge\ p_1 \wedge\ p_2 \wedge\ p_3 \wedge p_4\} \rightarrow\ p_5$$

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 PO1 and its predicates are listed in Table 4, where the extracted stage has j dual-rail inputs $i^1, \dots, i^j$, 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 rail⁰ and rail¹, respectively. Predicate p_0 indicates that all stage inputs are valid DATA; p_1 indicates that all NCL threshold gates in the stage are initialized to 0; p_3 corresponds to a NULL to DATA transition of the stage; and p_3 states that the rails of each dual-rail output are complements of each other. The predicates for PO1, states that after a NULL to DATA transition of the stage with all possible valid DATA inputs, that each output's rail⁰, R^0, is always the inverse of its respective rail¹ output, R^1.

Table 4. Predicates for revised invariant check

p_n	Predicate
p_0	$\bigwedge_{n=1}^{n=j}((i_A{}^n = 0b01) \vee (i_A{}^n = 0b10))$
p_1	$\bigwedge_{n=1}^{n=m}(g_A{}^n = 0)$
p_2	$(g_B{}^1, \ldots, g_B{}^m) = NCLStep(i_A{}^1, \ldots, i_A{}^j)$
p_3	$\bigwedge_{n=1}^{n=k} o_B{}^n < R^0 > \neq o_B{}^n < R^1 >$

$$\textbf{PO1}: \{\, p_0 \wedge \, p_1 \wedge \, p_2\} \rightarrow \, p_3$$

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, which only utilizes each gate's set function, that 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).

2.1.3. Combinational NCL Circuits Results

The methodology has been demonstrated on several multipliers and ISCAS-85 [11] combinational circuit benchmarks, as shown in Table 5. umultN represents a non-pipelined N-bit×N-bit unsigned multiplier. The NCL-to-Boolean netlist conversion time was negligible compared to the safety and invariant checks performed by the Z3 SMT solver [10] 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 umult10-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 results 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 is the same wire. Z3 reported all functional and invariant bugs along with a counter example; and our handshaking check tool identified and reported the location of all inserted completion logic bugs.

2.2. Equivalence Verification for Sequential NCL Circuits

2.2.1. NCL to Synchronous Reduction

As described in Section 2.1, our equivalence verification methodology proved to be a fast and scalable approach for C/L NCL circuits. Hence, in this section we extend that approach to

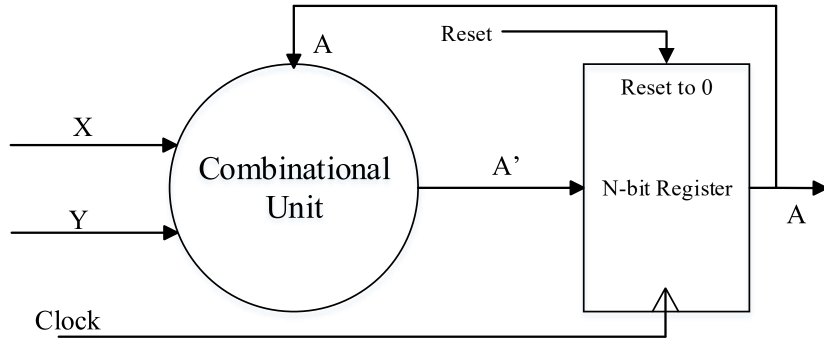
verify both safety and liveness of sequential NCL circuits, which is more complex due to datapath feedback.

Table 5. Verification results of various C/L NCL circuits (in sec.)

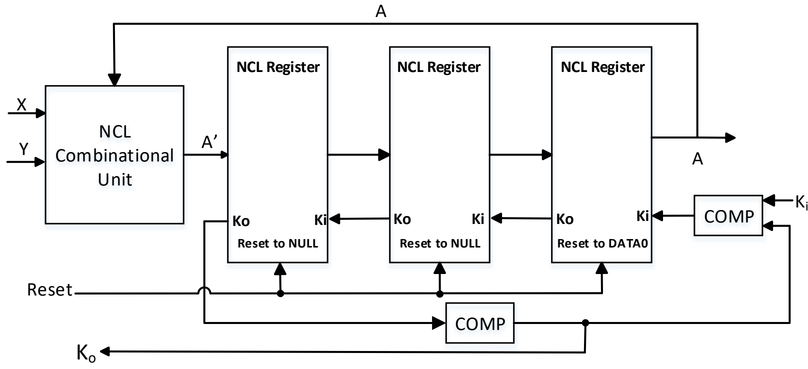
Circuit	Functional Check	Invariant Check	Total Time
ISCAS C17	0.01	0.01	0.02
umult2	0.02	0.01	0.03
umult3	0.04	0.02	0.06
umult6	0.32	0.33	0.65
umult8	10.62	6.79	17.41
umult10	683.49	192.39	875.88
ISCAS C432	1.03	1.06	2.09
umult10-B1	0.08 (B)	0.10 (B)	0.18
umult10-B2	0.06 (B)	192.39	192.45
umult10-B3	683.49	192.39	875.88
umult10-B4	683.49	0.08 (B)	683.57
umult10-B5	0.10 (B)	0.09 (B)	0.19

To describe our methodology, we will use an unsigned Multiply and Accumulate (MAC) unit as an example circuit. Fig. 13(a) shows a synchronous MAC, where A' = A+X×Y; and Fig. 13(b) 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 [5].

Hence, at least 3 NCL registers are needed in the MAC feedback loop to avoid deadlock, as shown in Fig. 13(b). 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.



(a)



(b)

Figure 13. Multiply and accumulate (MAC) circuit: (a) Synchronous (b) NCL

Fig. 14 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 [5]). (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.

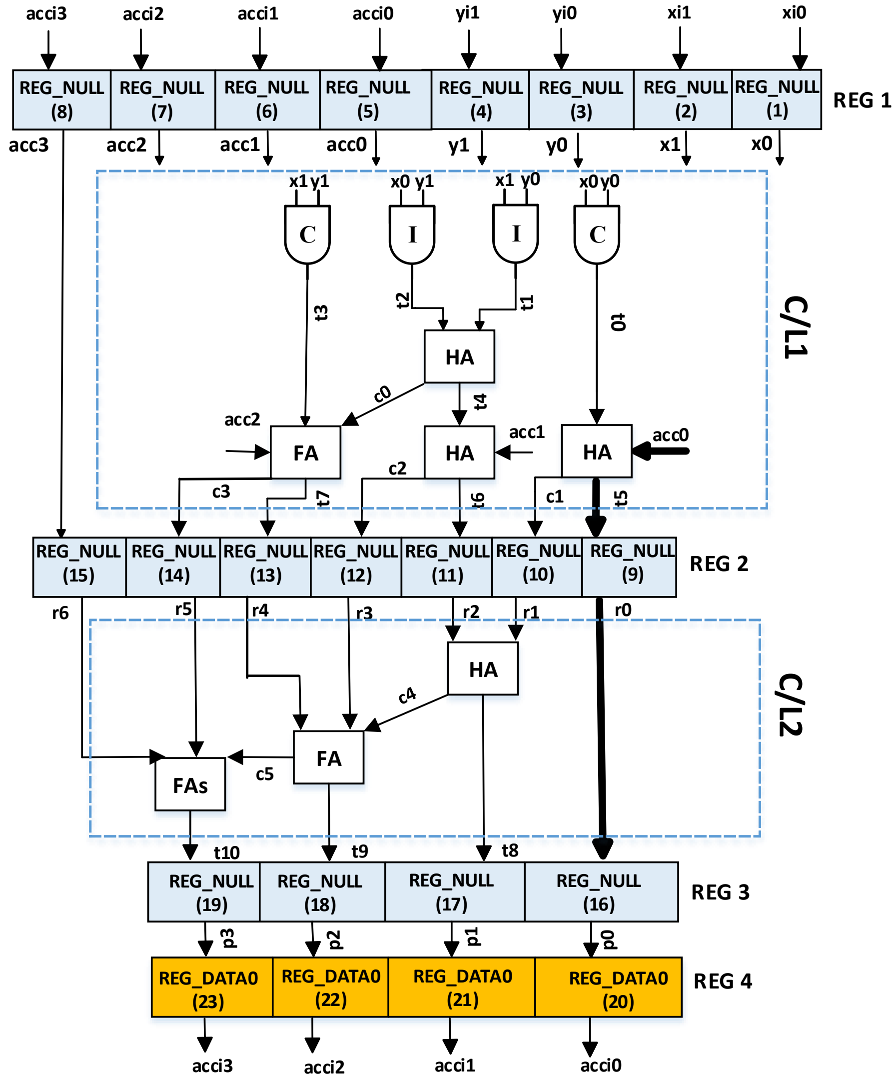


Figure 14. 4 + 2 × 2 NCL MAC datapath

All signals shown in Fig. 14 are dual-rail signals. *HA* and *FA* are the NCL half-adder and full-adder components, shown in Figs. 9 and 10, 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 (same as the HA *sum* output shown in Fig. 9, to compute its *sum* output. The highlighted components in Fig. 14 are the NCL registers.

Fig. 15(a) shows the netlist of the NCL 4+2×2 MAC, following the same structure as described in Section 2. 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.

2.2.1.1. Safety

Safety verification 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 [4] 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 invariant for each C/L stage, the same as previously discussed in Section 2.1.

The converted netlist (NCL-SYNC) is depicted in Fig. 15(b). The conversion algorithm for sequential NCL circuits is slightly different than for C/L NCL circuits, described in Section 2.1, 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.

```
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
```

(a) (b)

Figure 15. (a) 4 + 2 × 2 NCL MAC netlist, (b) Converted synchronous equivalent netlist

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, we use transition systems 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 [5] defines what it means for an implementation transition system to be functionally equivalent to a specification transition system. Therefore, we use the theory of WEB refinement for checking equivalence for sequential circuits.

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. Therefore, the correctness proof obligations required for verifying WEB refinement can be reduced to the proof obligation depicted in Fig. 16, where 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*; *StepSYNC_NCL* and *StepSYNC_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 below in PO2, and checked using an SMT solver.

$$\boldsymbol{PO2}: \{\forall s :: s \in S_{SyncNCL}$$
$$:: \left[u = RegProj(s) \wedge w = Step_{SyncNCL}(s) \wedge v\right.$$
$$\left.= Step_{SyncSPEC}(u)\right] \Rightarrow RegProj(w) = v\}$$

After verifying function equivalence, the rail^0 outputs of each C/L stage must also be checked to ensure safety, as detailed in Section 2.2.1.1. Note that for sequential circuits, which include datapath feedback, the first invariant check method that checks the entire circuit simultaneously will not work; hence, the second, much faster method that performs the invariant check independently for each stage is utilized.

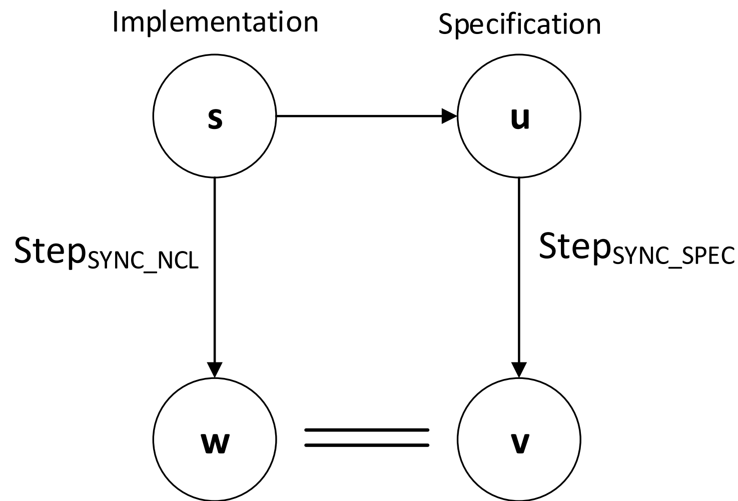


Figure 16. Depiction of proof obligation to check equivalence of NCL_SYNC and SPEC_SYNC netlists

2.2.1.2. Liveness

Fig. 17 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 2. 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. 17, 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, 9-14. Fig. 18 shows the reg_fanin and ko_fanout lists for each register in the 4+2×2 NCL MAC example.

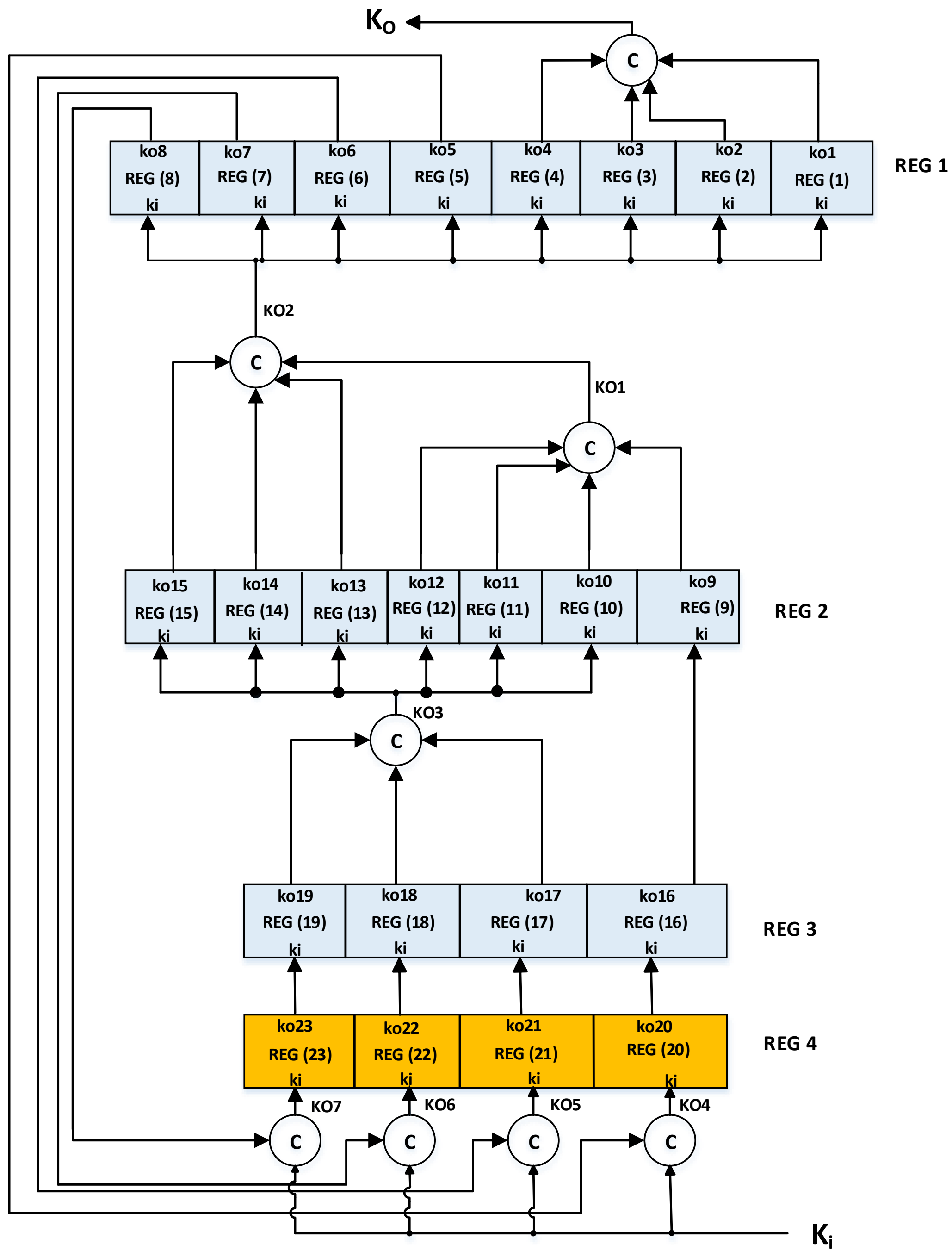


Figure 17. Handshaking connections for the 4+2×2 NCL MAC

After verifying handshaking correctness, each stage's C/L must also be checked for input-completeness and observability, utilizing the methods detailed in Sections 15.3.4 and 15.3.5, respectively, to guarantee liveness.

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, 12, 13, 14,15]
18: reg_fanin: [10, 11, 12, 13]	ko_fanout: [10,11, 12, 13, 14,15]
19: reg_fanin: [10,11, 12, 13,14,15]	ko_fanout: [10,11, 12, 13, 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 18. reg_fanin and ko_fanout lists for the 4+2×2 NCL MAC

2.2.1.3. Sequential NCL Circuit Results

The verification results for sequential NCL circuits, including functional equivalence and handshaking checks, are shown in Table 6. Since the invariant, input-completeness, and observability checks are exactly the same for combinational and sequential NCL circuits, these results are not included in Table 6. Test circuits include multiple MAC units and one ISCAS-89 benchmark, s27 [12]. The MAC units are represented as A+M×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 2.2. Z3 reported all functional bugs along with a counter example; and our handshaking check tool identified and reported the location of all inserted completion logic bugs.

Table 6. Verification results for sequential NCL circuits (in sec.)

Circuit	Functional Check	Handshaking Check	Total Time
ISCAS s27	0.01	0.0019	0.0119
4+2×2MAC	0.01	0.0045	0.0145
8+4×4MAC	0.05	0.79	0.84
12+6×6MAC	0.77	2.33	3.10
16+8×8MAC	47.55	21.74	69.18
20+10×10MAC	2643.99	163.65	2807.64
20+10×10MAC-B1	0.11 (B)	163.65	163.76
20+10×10MAC-B2	0.13 (B)	163.65	163.78
20+10×10MAC-B3	2643.99	169.84 (B)	2813.83
20+10×10MAC-B4	2643.99	159.33 (B)	2803.32
20+10×10MAC-B5	0.20 (B)	163.65	163.85

2.2.2. Exploiting Dual-Rail Invariants for Equivalence Verification

In the previous sections, NCL equivalence verification is broken down into two basic steps. In the first step, an NCL netlist is converted into its Boolean/synchronous equivalent netlist. The conversion from an NCL netlist to its Boolean/synchronous equivalent netlist is done by first converting all NCL combinational logic (C/L) gates with their relaxed counterparts, which have the same function but without hysteresis. Next, each primary circuit input is transformed from a dual-rail representation to a single bit that represents the $rail^1$ input. The $rail^0$ value is generated by negating the $rail^1$ input, which is then used internally. Furthermore, all Reset-to-NULL register components (denoted as Reg_NULL herein) were removed, and their inputs connected directly to their outputs. All Reset-to-DATA register components (denoted as Reg_DATA herein) were replaced with corresponding 2-bit resettable synchronous registers to capture both the $rail^1$ and $rail^0$ values. In the second step, the converted Boolean/synchronous

implementation netlist is then encoded in SMT-LIB along with its corresponding synchronous specification, and the two are verified to be equivalent using the theory of Well-Founded Equivalence Bisimulation (WEB) refinement [5]. An SMT solver was used to check the equivalence property. These steps proved very beneficial because the model of the converted Boolean/synchronous circuit is much more deterministic in regard to when signals transition, resulting in much faster verification times. This section proposes the Dual-Rail Register Invariant technique that modifies the conversion technique described above for the register components.

2.2.2.1. Safety

The 3×3 unsigned NCL multiplier that implements the function p(5:0) = xi(2:0) × yi(2:0), as shown in Fig. 11 without its completion logic, will be used as the example circuit to show the circuit transformation done in the previous work and contrast that to the proposed Dual-Rail Register Invariant. It is comprised of several components including dual-rail inputs and outputs, input-complete NCL AND functions (represented with a C inside the AND symbol), input-incomplete NCL AND functions (represented with an I inside the AND symbol), NCL Half-Adders (HA) and Full-Adders (FA), and dual-rail Reset-to-NULL registers (REG_NULL).

To accompany the circuit in Fig. 11, its netlist, and the netlist of the previous work's circuit transformation, are shown in Fig. 19(a) and 19(b), respectively. In Fig. 19(a), 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, 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.

2.2.2.2. Previous Circuit Transformation

For Fig. 19(b), each NCL gate from Fig. 19(a) is replaced with its corresponding Boolean gate without hysteresis, and the dual-rail primary inputs are replaced by their respective $rail^1$ input, which are then complemented by inserting invertors (lines 3-8) to generate their corresponding $rail^0$ signals. The Reg_NULL components are removed by connecting their inputs to their outputs; and the handshaking C-elements are also removed.

2.2.2.3. Proposed Dual-Rail Register Invariant

Instead of removing the Reg_NULL components by connecting their inputs to their outputs, the proposed Dual-Rail Register Invariant removes the Reg_NULL components by connecting their $rail^1$ inputs to their corresponding $rail^1$ outputs, and then generates each $rail^0$ output by inverting its corresponding $rail^1$ input. This transformation is possible due to the inherent NCL property where the $rail^1$ and $rail^0$ values are inverses of each other in the DATA phase. Note that both this inverse signal property and correctness of the NULL phase are checked as part of the NCL formal verification method presented in [13]. The proposed Dual-Rail Register Invariant allows the SMT solver to trim the circuit by removing all logic solely

used to generate the Reg_NULL rail^0 inputs, replacing this instead with a single inverter, as shown in Fig. 19(c).

2.2.2.4. Proof Obligation

While the circuit transformation is different for the two above techniques, the proof obligation remains the same. To describe the circuits without loss of generality, assume an NCL circuit has p dual-rail inputs and q dual-rail outputs, while its Boolean/synchronous specification has p and q Boolean inputs and outputs, respectively. Let $i^1, \ldots, i^p$ represent the Boolean circuit inputs, $O_{NCL}{}^1, \ldots, O_{NCL}{}^q$ be the dual-rail output values after symbolically stepping the converted NCL circuit using inputs $i^1, \ldots, i^p$, and $O_{sync}{}^1, \ldots, O_{sync}{}^q$ be the Boolean output values after symbolically stepping the Boolean/synchronous specification using inputs $i^1, \ldots, i^p$. The predicates in Table 7 are used to construct the Equivalence Proof Obligation as follows:

Table 7. Predicates for equivalence check

p_n	**Predicate**
p_0	$(O_{NCL}{}^1, \ldots, O_{NCL}{}^q) = NCLStep(i^1, \ldots, i^p)$
p_1	$(O_{SYNC}{}^1, \ldots, O_{SYNC}{}^q) = SyncStep(i^1, \ldots, i^p)$
p_2	$\bigwedge_{n=1}^{n=q} \left(rail1(o_{NCL}{}^n) = o_{sync}{}^n\right)$

$$\boldsymbol{PO3}: \{\, p_0 \wedge p_1 \,\} \rightarrow p_2.$$

p_0 represents the symbolic step of the converted NCL circuit and p_1 the symbolic step of the Boolean/synchronous specification, both using inputs $i^1, \ldots, i^p$, with the values of the circuit outputs recorded in $O_{NCL}{}^1, \ldots, O_{NCL}{}^q$ and $O_{sync}{}^1, \ldots, O_{sync}{}^q$, respectively. p_2 indicates that the rail^1 symbolic outputs of the converted NCL circuit are equal to those of the Boolean/synchronous specification.

For comparison, the same unsigned NCL Multiply and Accumulate (MAC) circuits as in [13], with increasing operand sizes to show scalability, were used. These implement the function acci = acci + xi × yi, as shown in Fig. 14 for a 4+2×2 NCL MAC, without its completion logic. As shown in the Fig. 14 example, each MAC's C/L is partitioned into 2 stages by inserting a Reset-to-NULL register between the last carry-save adder and the final ripple-carry adder; and the feedback loop contains 4 registers for increased performance. Note that the proposed Dual-Rail Register Invariant can also be applied to Reset-to-DATA registers, resulting in their replacement with a single synchronous register, plus an inverter to generate the $rail^0$ output, instead of the previous conversion technique that required 2 synchronous registers, as described in Section 2.

Table 8. Dual-rail refinement results

Circuit	Speedup
8+4×4 MAC	1.14
12+6×6 MAC	1.17
16+8×8 MAC	2.75
20+10×10 MAC	1.31
22+11×11 MAC	3.63
24+12×12 MAC	TO/67,599 sec
16+8×8 MAC-5 Reg	3.44
20+10×10 MAC-5 Reg	1.46

The Z3 SMT solver [10] was used to check for equivalence, but any combinational equivalence checker could be used. Table 8 lists the verification results, where the first column indicates the MAC size, and the second column is speedup (i.e., equivalence verification time using the method described in [13] divided by equivalence verification time using the proposed Dual-Rail Register Invariant). Timeout (TO) denotes that the verification time exceeded one day. The last 2 rows in Table 8 are for MACs with an additional Reset-to-NULL register inserted between the partial product generation circuitry (i.e., AND functions) and the first carry-save adder.

The results show speedups ranging from 14% - 263% for the various 4-register MACS, and an additional speedup of 25% and 11% when adding an extra 5th register stage in the 16+8×8 and 20+10×10 MACs, respectively. Note that the 24+12×12 MAC timed out using the previous approach in [13] but was successfully verified in less than 1 day utilizing the proposed Dual-Rail Register Invariant.

2.3. Equivalence Verification Conclusion

Section 2 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. 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. Section proposes the Dual-Rail Register Invariant technique to speedup equivalence checking of NCL circuit implementations with respect to their Boolean/synchronous specifications. It has been shown to significantly reduce equivalence checking times, and to further speedup equivalence checking when additional pipeline stages are added to the NCL circuit, as this allows for more usage of the proposed technique.

The proposed Dual-Rail Register Invariant technique is also applicable to equivalence verification of Sleep Convention Logic (SCL) circuits [14] with respect to their boolean/synchronous specifications, as the circuit transformation and safety verification are the same for SCL and NCL, only liveness verification differs [15].

```
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
```

(a)

```
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
```

(b)

```
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. not   p0_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. not   t6_1   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. not   t5_1   t5_0
25. th22 xi2_1,yi2_1 t7_1
26. not   t7_1   t7_0
27. th24comp   t0_0,t1_0,t0_1,t1_1  p1_1
28. not   p1_1   p1_0
29. th22   t0_1, t1_1   c1_1
30. th12   t0_0,t1_0  c1_0
31. th23   t3_1,t2_1,c1_1  c2_1
32. not   c2_1   c2_0
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. not   p2_1   p2_0
37. th22   s1_1,t4_1  c3_1
38. not   c3_1   c3_0
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. not   p3_1   p3_0
45. th22   s2_1,t6_1  c5_1
46. th12   s2_0,t6_0  c5_0
47. th23   t7_1,c4_1,c5_1   p5_1
48. not   p5_1   p5_0
49. th34w2    p5_0,t7_1,c4_1,c5_1  p4_1
50. not    p4_1   p4_0
```

(c)

Figure 19. (a) 3×3 NCL multiplier netlist (b) Converted netlist using method in Section 2 (c) Converted netlist using proposed Register Invariance Refinement

Future work includes investigating additional refinements and invariants to potentially further reduce verification time, such as applying the technique described herein, to generate rail^0 as the inverse of rail^1, at the outputs of every NCL C/L function (e.g., HA, FA, AND), instead of only at the register boundaries. This would require NCL C/L functions to be reconstructed from a flattened NCL gate netlist, where the two wires of a dual-rail signal are disassociated. Additionally, the invariant check from [13] that ensures that the rail^1 and rail^0 outputs of each NCL register are always inverses of each other during the DATA phase would need to be checked at the output of each NCL C/L function, instead of only at the register boundaries.

The framework of this verification methodology can also be applied to other QDI paradigms, such as MTNCL and PCHB. For MTNCL, the functional checking and invariant checking methods are essentially the same as for NCL, but the handshaking check is slightly different [15].

3. NCL INPUT-COMPLETENESS VERIFICATION

An automated formal verification approach for ensuring input-completeness of NULL Convention Logic (NCL) circuits [1] is proposed. NCL circuits have the benefit that they can operate in extreme environments where traditional synchronous circuits fail due to significant fluctuations in circuit timing. Input-completeness is a critical property to ensure correct functioning of NCL circuits in extreme environments and therefore is required to be verified. Note that an NCL circuit can be functionally correct and still not be input-complete, which could cause the circuit to operate correctly under normal conditions, but malfunction only when the circuit timing is substantially changed (e.g. operating in a very hot or cold environment such as outer space).

NULL Convention Logic (NCL) circuits [1] are a type of Quasi-Delay Insensitive (QDI) asynchronous design style that has been demonstrated to function in environments characterized by high radiation exposure, and high or low temperatures or large temperature fluctuations, where synchronous circuit counterparts fail [2]. The ability of NCL circuits to function correctly in extreme environments makes them very suitable for space exploration, the power industry, the automobile industry (internal combustion engines), oil/gas exploration, medical imaging instrumentation, the laser industry, superconducting computing and energy storage systems, and low voltage or low power applications such as wireless sensor networks (WSN) or Internet of Things (IoT).

Synchronization of NCL circuits happens via the propagation of NULL and DATA waves through the circuit, utilizing handshaking instead of a traditional clock signal. Dual-rail signals are used for data representation. A NULL state (absence of data) is represented by 0b00 and a DATA state is represented as either 0b01 (0 in Boolean) or 0b10 (1 in Boolean). The state

0b11 is an ILLEGAL state. As mentioned in Section 2, to achieve delay-insensitivity, all NCL circuits must satisfy two properties, input-completeness and observability. In order for a combinational NCL circuit to be input-complete, its outputs may not all transition from NULL to DATA until all inputs have transitioned from NULL to DATA, and conversely, may not all transition from DATA to NULL until all inputs have transitioned from DATA to NULL [16]. Note that some outputs can transition to DATA (NULL) before all inputs are DATA (NULL) as long as all outputs cannot become DATA (NULL) until all inputs are DATA (NULL) [17]. Observability ensures that every gate asserted during a DATA wavefront propagates through the circuit to cause at least one circuit output to be asserted. In this paper, an automated formal verification approach to check input-completeness of NCL circuits is proposed. The efficiency of the proposed approach is demonstrated using 37 NCL circuit benchmarks.

3.1. Input-Completeness Verification

The proposed approach for input-completeness verification is as follows. Two proof obligations (POs) have been formulated, one for NULL to DATA transition and one for DATA to NULL transition. The POs are generic and can be applied to any NCL combinational circuit and can be automatically checked using a decision procedure such as a Satisfiability Modulo Theories (SMT) solver. The PO for the NULL to DATA transition is described next.

3.1.1. Input-Completeness Proof Obligation: NULL to DATA

Without loss of generality, assume an NCL circuit has m threshold gates, p dual-rail inputs, q dual-rail outputs. Let ${g_A}^1, \dots, {g_A}^m$ represent Boolean variables that correspond to the current state of the NCL threshold gates before $step\ A$ and ${g_B}^1, \dots, {g_B}^m$ represent the same threshold gates' state after $step\ A$. Let ${i_A}^1, \dots, {i_A}^p$ represent the circuit inputs for $step\ A$, and ${i_B}^1, \dots, {i_B}^p$ for $step\ B$. Let ${o_A}^1, \dots, {o_A}^q$ be the circuit output values after symbolically stepping

the circuit using inputs $i_A{}^1, \dots, i_A{}^p$ and threshold gates states $g_A{}^1, \dots, g_A{}^m$. Let $o_B{}^1, \dots, o_B{}^q$be the circuit output values after symbolically stepping the circuit using $i_B{}^1, \dots, i_B{}^p$ and threshold gate states $g_B{}^1, \dots, g_B{}^m$. The proof obligation predicates for input-completeness are given in Table 9.

p_0 indicates that no dual-rail inputs are in an illegal state. p_1 states that all the threshold gate's current output values are 0, which indicates that the circuit is in the NULL state before a DATA transition. p_2 indicates that at least one of the dual rail inputs is NULL, and p_3 indicates that at least one of the dual-rail outputs is NULL. Proof obligation $PO2$ below is used to check input-completeness of the NULL to DATA transition of the circuit. $PO2$ states that if none of the inputs are ILLEGAL, all current threshold gate outputs are 0, and at least one of the dual-rail inputs is NULL, then at least one of the dual-rail outputs must be NULL. Since the dual-rail inputs in the proof obligation are symbolic, the SMT solver checks this property for all possible input combinations.

3.1.2. Input-Completeness Proof Obligation: DATA to NULL

When NCL circuits are constructed using only threshold gates with hysteresis, ensuring input-completeness of the NULL to DATA transition guarantees input-completeness of the DATA to NULL transition. Reason for this is because gate hysteresis ensures that a gate output cannot transition to 0 until all its inputs transition to 0. However, there are NCL based designs that are comprised of both threshold gates with hysteresis and Boolean gates. These types of circuits are called relaxed NCL circuits. Hence, for relaxed NCL circuits, input-completeness of the DATA to NULL transition must also be checked.

To formulate the DATA to NULL proof obligation, the circuit must first be symbolically initialized with all possible threshold gate outputs after a transition from NULL to DATA. This is done by first initializing the circuit to the NULL state (i.e., all threshold gates are set to 0) and

then stepping the circuit with valid symbolic DATA (i.e., not NULL and not illegal) inputs, identified as $step\ A$. The symbolic values of the threshold gates from $step\ A$ are retained, and the circuit is symbolically stepped again with new inputs, identified as $step\ B$, which represents the DATA to NULL transition.

Table 9. Predicates for input-completeness check

$\boldsymbol{p_n}$	***Predicate***
p_0	$\bigwedge_{n=1}^{n=p} \sim (i_A{}^n = 0b11)$
p_1	$\bigwedge_{n=1}^{n=m} (g_A{}^n = 0)$
p_2	$\bigvee_{n=1}^{n=p} (i_A{}^n = 0b00)$
p_3	$\bigvee_{n=1}^{n=q} (o_A{}^n = 0b00)$
p_4	$\bigwedge_{n=1}^{n=p} ((i_A{}^n = 0b01) \vee (i_A{}^n = 0b10))$
p_5	$(g_B{}^1, \ldots, g_B{}^m) = NCLStep(i_A{}^1, \ldots, i_A{}^p)$
p_6	$\bigwedge_{n=1}^{n=p} ((i_B{}^n = i_A{}^n) \vee (i_B{}^n = 0b00))$
p_7	$\bigvee_{n=1}^{n=p} (i_B{}^n = i_A{}^n)$
p_8	$\bigvee_{n=1}^{n=q} ((o_B{}^n = 0b01) \vee (o_B{}^n = 0b10))$

$$\boldsymbol{PO4}: \{\, p_0 \wedge p_1 \wedge p_2 \,\} \rightarrow p_3$$

p_1 initializes all threshold gate outputs to 0 before $step\ A$. p_4 indicates that all $step\ A$ inputs are DATA. p_5 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 $(g_B{}^1, \dots, g_B{}^m)$. p_6 indicates that each input for $step\ B$ is either the same DATA value it was for $step\ A$, or has transitioned to NULL. p_7 indicates that at least one of the inputs for $step\ B$ is still DATA; and p_8 indicates that at least one of the outputs of $step\ B$ remains DATA. The final proof obligation for input-completeness of the DATA to NULL transition is given below as $PO3$. It states that after initializing the circuit to the NULL state and symbolically stepping the circuit with all possible DATA inputs to generate all possible DATA states, that if at least one dual-rail input remains DATA while other inputs may transition to NULL, at least one of the outputs must remain DATA, meaning that the circuit has not fully transitioned to the NULL state, because all inputs have not yet transitioned to NULL. Like the NULL to DATA proof obligation, all inputs are symbolic, so the SMT solver checks all combinations.

$$\boldsymbol{PO5}: \{\, p_1 \wedge p_4 \wedge p_5 \ \wedge p_6 \wedge p_7 \,\} \rightarrow p_8$$

3.1.3. Input-Completeness Results

Verification of the two proof obligations can be performed using an SMT solver. The benchmarks used for verification were NxN unsigned dual-rail NCL multipliers ranging from 3 to 20 bits, as well as the ISCAS-85 C432 27-channel interrupt controller [18]. To perform verification, both the circuit and the POs needed to be encoded in the SMT-LIB [9] language. This was performed automatically using a developed tool that took as input the netlist of the circuit and generated both the circuit model and PO specifications in SMT-LIB format. PO checking was preformed using the z3 SMT solver [10]. Verification experiments were run on a Intel® Core™ i7-4790 CPU with 32GB of RAM, running at 3.60 GHz.

The results of all benchmarks are listed in Table 10, where the first column is the Circuit Name, the second column is the verification time for the NULL to DATA proof obligation of a

correct input-complete implementation, the third column is the verification time for the NULL to DATA proof obligation of an incorrect input-incomplete implementation, and columns four and five report the verification times for the DATA to NULL proof obligations for input-complete and input-incomplete implementations, respectively. $umultN$ represents an $N\text{-}\ bit\ x\ N\text{-}\ bit$ unsigned multiplier that was constructed using only NCL gates with hysteresis. $r - umultN$ represents a relaxed version of $umultN$ where NCL gates are replaced with their Boolean equivalent gates when hysteresis is not required for input-completeness. Any verifications that had a run time greater than one day is listed as Timeout (TO).

The benchmark multipliers were designed with input-complete AND functions to generate the X_iY_i partial products and input-incomplete AND functions for the X_iY_j partial products, where $i \neq j$, but without the intermediate NCL register (i.e., a single stage with only input and output registers [18]). To create the buggy non-relaxed versions, $1 \leq k \leq N$ was chosen at random and the input-complete AND function used to generate the X_kY_k partial product was replaced with an input-incomplete version. NCL half-adders (HAs) and full-adders (FAs) are inherently input-complete and therefore cannot be made input-incomplete when constructed only using NCL gates with hysteresis. The relaxed version of each multiplier was constructed by taking the non-relaxed version and replacing the TH22 gate within the input-incomplete AND and HAs components with a Boolean AND gate. Buggy relaxed circuits were constructed by relaxing one of the following: either the TH22 or THand0 gate in a X_iY_i partial product AND component, a TH24comp gate in a HA component, or either a TH34w2 or TH23 gate in a FA component. The ISCAS-85 C432 circuit was designed using input-incomplete functions when possible while maintaining input-completeness. The buggy version was obtained

by replacing one of the input-complete 3-input NAND functions that calculate RC, in Module M3 [19], with an input-incomplete version. Z3 reported all bugs along with a counter example.

Table 10. Verification results of input-completeness (in sec.)

Circuit	N to D	Buggy N to D	D to N	Buggy D to N
umult3	0.02	0.01	0.03	0.04
umult4	0.02	0.05	0.06	0.0.06
umult5	0.09	0.05	0.12	0.11
umult6	0.11	0.15	0.38	0.24
umult7	0.38	0.27	1.49	1.23
umult8	1.44	0.49	5.47	3.60
umult9	5.30	2.37	22.38	1.28
umult10	20.22	8.92	102.42	18.45
umult11	54.09	2.99	430.29	22.81
umult12	236.00	8.21	1909.44	23.17
umult13	885.30	3.85	7401.11	15.11
umult14	3424.89	114.41	34961.6	8.26
umult15	9663.01	19.41	**TO**	112.55
r-umult3	0.02	0.02	0.04	0.07
r-umult4	0.02	0.02	0.06	0.07
r-umult5	0.05	0.04	0.10	0.08
r-umult6	0.15	0.12	0.42	0.07
r-umult7	0.39	0.12	1.48	0.11
r-umult8	1.38	1.43	6.38	0.17
r-umult9	4.74	5.17	28.03	0.20
r-umult10	16.26	19.02	146.95	0.20
r-umult11	58.04	46.53	642.80	0.31
r-umult12	215.75	228.47	3635.01	0.35
r-umult13	729.11	34.97	15663.24	0.40
r-umult14	3045.99	4104.45	80213.90	0.68
r-umult15	10561.11	9974.39	**TO**	0.31
C432	0.062	0.068	0.074	0.94

3.1.4. Input-Completeness Conclusion

An approach to automated verification of input-completeness for NCL circuits is presented. It ensures input-completeness of combinational NCL circuits comprised solely of gates with hysteresis, as well as relaxed NCL circuits that contain some gates without hysteresis; whereas the previous manual approach for ensure input-completeness [5] is not applicable to

relaxed NCL circuits. It also ensures input-completeness for all inputs simultaneously, whereas [7] must check each input separately. The proposed approach is efficient; however, scalability can be improved and will be a topic for future work.

4. NCL OBSERVABILITY VERIFICATION

Observability requires every gate transition to be observable at the output, which means that every gate that transitions is necessary to transition at least one output. Observability of every gate in every C/L stage is required for NCL circuits to be QDI; an unobservable gate in any stage may cause the circuit to deadlock under some timing scenarios.

4.1. Observability Verification

Observability can be proven in a similar fashion to input-completeness. Two proof obligations are needed for each C/L gate, one for the NULL to DATA transition, and the other for the DATA to NULL transition. The proof obligations, like those for input-completeness, have been encoded in a decidable fragment of first order logic and are automatically checked using an SMT solver. The PO for the NULL to DATA transition is described next.

4.1.1. Observability Proof Obligation: NULL to DATA

To verify observability, a check must be performed on each C/L gate. For each gate $g^1, \dots, g^m$ assertion of that gate is first computed, denoted as $f_1, \dots, f_m = 1$, respectively. During the NULL to DATA observability verification of $f_1, \dots, f_m = 1$, where $1 \leq n \leq m$, the output of g^n is forced to 0. Simulation of a circuit with g^n forced to 0 is called a Gn0 simulation, and the resulting function is $nclcktGn0(i^1, \dots, i^p)$. To formulate the DATA to NULL observability proof obligation, the circuit must first be symbolically initialized with all possible threshold gate outputs that assert g^n after a transition from NULL to DATA. This is done by first initializing the circuit to the NULL state (i.e., all threshold gates are set to 0) and then stepping the circuit with valid symbolic DATA (i.e., not NULL and not illegal) inputs, identified as $step\ A$. The symbolic values of the threshold gates from $step\ A$ are retained as ${g_B}^1, \dots, {g_B}^m$, and the circuit is symbolically stepped again with new inputs, identified as $step\ B$, which represents the DATA to

NULL transition. During the verification of g^n, where $1 \leq n \leq m$, the output of g^n is forced to 1. Simulation of a circuit with g^n forced to 1 is called a Gn1 simulation, and the resulting function is $nclcktGn1(i^1, \ldots, i^p)$. The predicates required for observability are listed below in Table 11.

p_0 states that all the threshold gates' current output value is 0, which indicates that the circuit is in the NULL state before a DATA transition. p_1 indicates that every circuit input is valid DATA. p_3 assigns the outputs of the NCL circuit for a Gn0 simulation, where the output of g^n, the gate under test, is forced to 0. p_4 enables only valid input combinations that would assert g^n to be used to step the circuit in p_3. Finally, p_5 ensures that at least one of the outputs is NULL. The proof obligation to test observability of the NULL to DATA transition is given below as *PO4*, which tests observability of all gates, $g^1, \ldots, g^m$. If true for g^n, this ensures that there is at least one output that will not be asserted if g^n is not asserted, for all sets of inputs in which g^n should be asserted, therefore proving that g^n is observable for the NULL to DATA transition.

$$\boldsymbol{PO6}: \bigwedge_{n=1}^{n=m} (\{ p_0 \wedge p_1 \wedge p_3 \wedge p_4 \} \rightarrow p_5)$$

4.1.2. Observability Proof Obligation: DATA to NULL

Like input-completeness, NCL circuits consisting only of NCL gates with hysteresis are inherently observable for the DATA to NULL transition if observable for the NULL to DATA transition, since gate hysteresis ensures that a gate output cannot transition to 0 until all its preceding gates' outputs transition to 0. However, this is not the case for relaxed NCL circuits, which are comprised of both threshold gates with hysteresis and Boolean gates. Hence, for relaxed NCL circuits, observability of the DATA to NULL transition must also be checked.

Table 11. Predicates for observability check

p_n	$Predicate$
p_0	$\bigwedge_{n=1}^{n=m} (g_A{}^n = 0)$
p_1	$\bigwedge_{n=1}^{n=p} ((i_A{}^n = 0b01) \vee (i_A{}^n = 0b10))$
p_2	$(g_B{}^1, \ldots, g_B{}^m) = NCLStep(i_A{}^1, \ldots, i_A{}^p)$
p_3	$(o_A{}^1, \ldots, o_A{}^q) = nclcktGn0(i_A{}^1, \ldots, i_A{}^p)$
p_4	$f_n = 1$
p_5	$\bigvee_{n=1}^{n=q} (o_B{}^n = 0b00)$
p_6	$\bigwedge_{n=1}^{n=p} (i_B{}^n = 0b00)$
p_7	$(o_B{}^1, \ldots, o_B{}^q) = nclcktGn1(i_B{}^1, \ldots, i_B{}^p)$
p_8	$\sim \bigwedge_{n=1}^{n=q} (o_B{}^n = 0b00)$

p_0 initializes all threshold gate outputs to 0 before $step\ A$. p_1 indicates that all $step\ A$ inputs are DATA. p_2 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 $(g_B{}^1, \ldots, g_B{}^m)$. p_4 enables only valid input combinations that would assert g^n to be used to step the circuit in p_2. p_6 indicates that all inputs for $step\ B$ have transitioned to NULL. p_7 assigns the outputs of the NCL circuit for a Gn1 simulation, where the output of g^n, the gate under test, is forced to 1. Finally, p_8 ensures that all outputs are not NULL. The proof obligation to test observability of the DATA to NULL transition is given below as *PO5*, which tests observability of all gates, g^1, ..., g^m. If true for g^n, this ensures that following a NULL to DATA transition that asserts g^n, there is at least one output that will not become NULL during the subsequent DATA to NULL transition while g^n remains asserted, therefore proving that g^n is observable for the DATA to NULL transition.

$$\boldsymbol{PO7}: \bigwedge_{n=1}^{n=m} (\{ p_0 \wedge p_1 \wedge p_2 \wedge p_4 \wedge p_6 \wedge p_7 \} \rightarrow p_8)$$

4.1.3. Observability Results

Verification of the proof obligations for observability can be performed using any SMT solver. To perform observability verification, we developed a tool to automatically generate the circuit model and proof obligation specifications, encoded in SMT-LIB format, from the original circuit netlist, such as the one shown in Fig. 12(a) for the 3×3 multiplier. For the verification results presented here, N-bit×N-bit unsigned dual-rail NCL multipliers were used as benchmarks, where $3 \le N \le 13$. The ISCAS-85 C432 27-channel interrupt controller circuit was also used as a benchmark [19]. The verification proof obligations were checked using the Z3 SMT solver on an Intel® Core™ i7-4790 CPU with 32GB of RAM, running at 3.60 GHz.

The verification results are listed in Table 12, where the first column is the Circuit Name, the second column is the verification time for the NULL to DATA proof obligation and the third column is the verification time for the DATA to NULL proof obligation. $umultN$ represents an N-bit×N-bit unsigned multiplier constructed using only NCL gates with hysteresis, while $r-umultN$ represents a relaxed version of the N-bit×N-bit multiplier, where NCL gates are replaced with Boolean gates when hysteresis is not required. Timeout (TO) is listed in the verification results when the verification time exceeded one day.

The test multipliers were designed exactly the same as the ones used for testing input-completeness (i.e., input-complete AND functions generate the X_iY_i partial products, and input-incomplete AND functions generate the X_iY_j partial products, where $i \neq j$). To create buggy multipliers that were input-complete but not observable, a HA was chosen at random and the XOR function to generate its sum (i.e., the two TH24comp gates in Fig. 9 was replaced with the

unobservable XOR function, shown in Fig. 8. To check observability of relaxed circuits, the M1 module of the ISCAS-85 C432 benchmark [20] was used, where the 9-input NAND function that generates *PA* was composed of two relaxed input-incomplete 4-input AND functions, followed by an input-complete 2-input AND function, and then an input-complete 2-input NAND function, as shown in Fig. 20. To create a buggy version that was input-complete but not observable, any of the 4 gates comprising the 2-input AND function or the 2-input NAND function shown in Fig. 20 could be relaxed. The test times reported for the circuits are for testing every single gate for observability, even if a previous gate was found to be unobservable. Therefore, the time to detect a buggy circuit will be less than or equal to the reported times since the rest of the gates would no longer need to be tested once an unobservable gate was identified. Z3 reported all bugs along with a counter example.

Table 12. Verification results of observability (in sec.)

Circuit	N to D	D to N
umult4	0.001	0.001
umult5	8.203	8.944
umult6	13.7599	16.1921
umult7	27.8229	36.528
umult8	54.062	105.4979
umult9	138.3139	412.605
umult10	363.7079	1968.434
umult11	902.046	9657.475
umult12	2384.504	52093.64
umult13	5797.037	TO
C432 M1	1.53	3.882

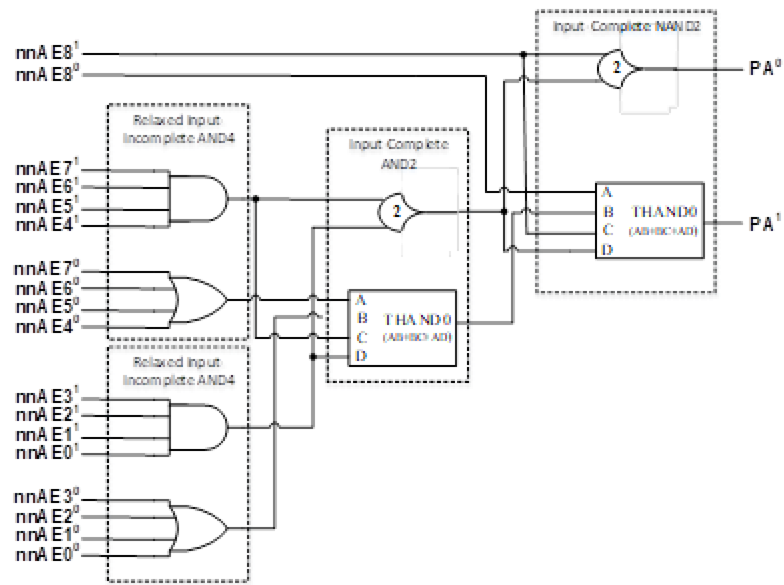


Figure 20. ISCAS-85 C432 M1 module nine-input NCL NAND that generates PA

5. NCL COMPLETION-COMPLETENESS VERIFICATION

In order to achieve delay-insensitivity, NCL circuits must be input-complete and observable [5]. Input-completeness requires that all outputs of a combinational circuit may not transition from NULL to DATA until all inputs have transitioned from NULL to DATA, and that all outputs of a combinational circuit may not transition from DATA to NULL until all inputs have transitioned from DATA to NULL. In circuits with multiple outputs, it is acceptable according to Seitz's "weak conditions" of delay-insensitive signaling [17], for some of the outputs to transition without having a complete input set present, as long as all outputs cannot transition before all inputs arrive. Observability requires that no orphans may propagate through a gate, where an orphan is defined as a wire that transitions during the current DATA wavefront but is not used in the determination of the output. NCL circuits that utilize the bit-wise completion strategy along with input-incomplete logic functions/components must also be completion-complete [8] to ensure delay-insensitivity. Completion-completeness requires that completion signals only be generated such that no two adjacent DATA wavefronts can interact within any combinational logic (C/L) component. Note that completion-completeness is inherent when using full-word completion. While [13] presents automated formal methods for ensuring that NCL circuits utilize correct handshaking, and are input-complete and observable, this paper describes an automated formal method to ensure that NCL circuits are also completion-complete.

5.1. Completion-Completeness Previous Work

The need for completion-completeness was demonstrated in [8] by showing a number of example NCL circuits that utilized proper handshaking connections and were input-complete and observable, but still were not delay-insensitive, since they allowed two adjacent DATA wavefronts to interact within a C/L component. Take for example the partial product

generation circuit for X(1:0) × Y(1:0) utilizing bit-wise completion, as shown in Fig. 21. AND functions Y(1) • X(1) and Y(0) • X(0) are input-complete, as shown in Fig. 7, while the other two AND functions are input-incomplete, as shown in Fig. 6, such that the entire circuit is input-complete (i.e., all outputs cannot become DATA until all inputs are DATA). To show that this circuit is not completion-complete, and therefore not delay-insensitive, let *Xi* and *Yi* be 00_2 and 11_2, respectively, which would result in *PPi* = 0000_2; and let *Xi+1(0)* = DATA1 and *Yi+1(1)* = DATA0, which would result in *PPi+1(1)* = DATA0, where the subscript, *i*, refers to the wavefront. Now, assume that the signals transition as follows, starting from the NULL state (i.e., all dual-rail signals are NULL and all *Ki* and *Ko* signals are rfd): *Xi* changes to DATA (i.e., 00_2), *Yi(0)* changes to DATA (i.e., DATA1), and *Yi(1)* remains NULL. This causes *PPi(2:0)* to become 000_2, as expected, while *PPi(3)* remains NULL, which in turn causes *Kic0* and *Kic1* to become rfn, allowing NULL to flow through these two input registers. This in turn causes *PPi(1:0)* to become NULL, assuming their respective *Ki* signals are rfn, which transitions *Kic0* to rfd, allowing *Xi+1(0)* = DATA1 to flow through its register. *Yi(1)* now finally transitions to DATA1, which causes two adjacent DATA wavefronts, *Xi+1(0)* and *Yj(1)*, to interact within the combinational logic, which violates the completion-completeness criterion; and this produces *PPi+1(1)* = DATA1, which is incorrect. In addition to showing how to manually determine if an NCL circuit is completion-complete, [8] also presented a variety of methods to make NCL circuits completion-complete, so that they would be delay-insensitive. For this example, either the two input-incomplete AND functions could be replaced with input-complete versions, or the completion logic sets would need to be modified. The work herein presents an automated method to formally verify that an NCL circuit is completion-complete.

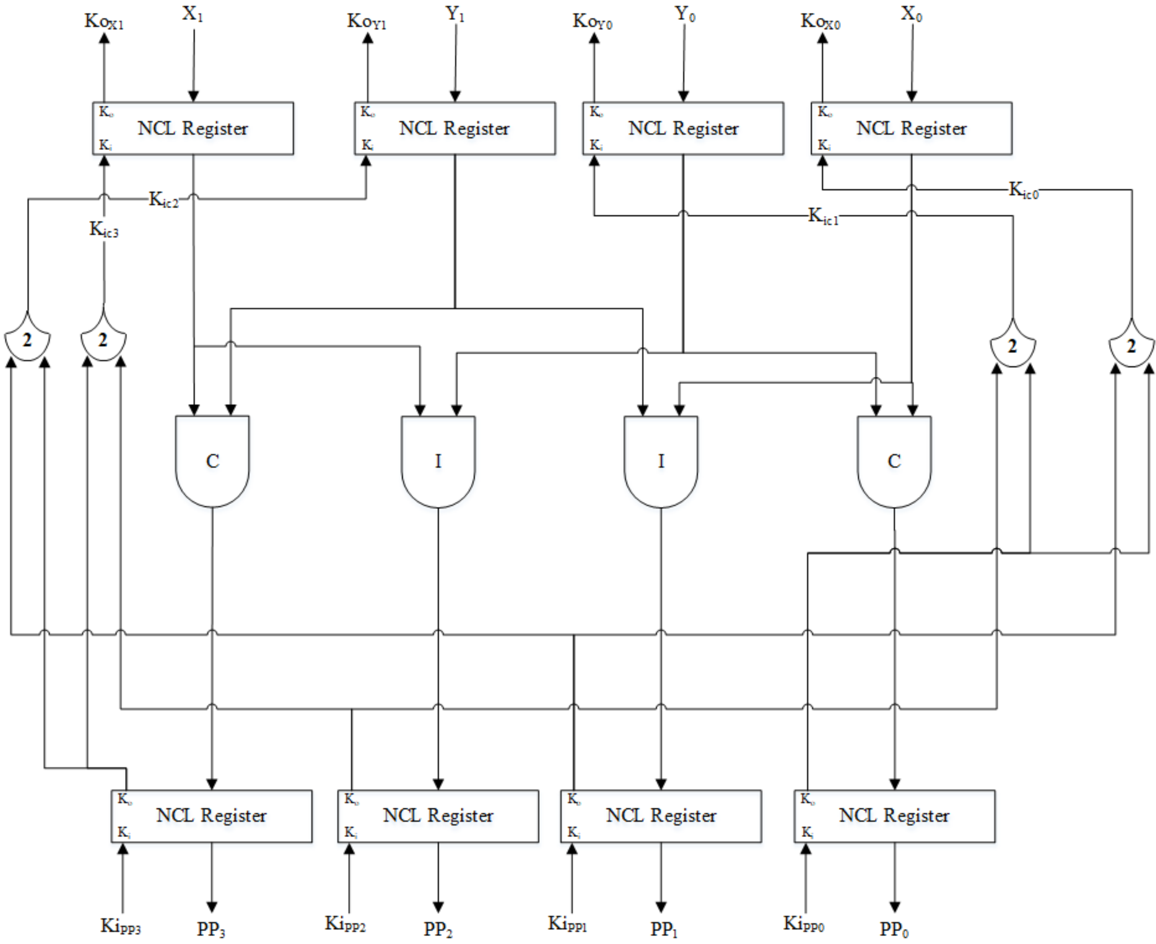


Figure 21. Completion-incomplete NCL circuit

5.2. Completion-Completeness Verification

The proposed completion-completeness verification is as follows. The NCL circuit is partitioned into stages, and each stage is handled independently. If a stage has p inputs, then p proof obligations are required for that stage. Below, we describe the generic proof obligation (PO) template that must be applied to each input of a circuit stage. The approach described using this template can be applied to any arbitrary NCL circuit. The POs are formulated such that they can be automatically checked using a Satisfiability Modulo Theories (SMT) solver [9].

5.2.1. Completion-Completeness Proof Obligation

Without loss of generality, an NCL circuit stage is assumed to have m threshold gates, p dual-rail inputs, and q dual-rail outputs, and include a p-bit input register and q-bit output register, as shown in Fig. 21 for p=q=4. To formulate the proof, three separate symbolic steps of the NCL circuit are required, denoted as Steps A, B, and C, respectively. ${g_{A/B/C/D}}^1, \dots, {g_{A/B/C/D}}^m$, are Boolean variables that represent the current state of the threshold gates for the corresponding symbolic step. ${i_{A/B/C}}^1, \dots, {i_{A/B/C}}^p$, are the symbolic values applied to the circuit inputs for the corresponding symbolic step; and ${i_{A/B/C}}^k$ represents the circuit input that is being verified. ${o_{A/B/C/D}}^1, \dots, {o_{A/B/C/D}}^q$, are the output values acquired during the corresponding step of the circuit with current state and input values mentioned above. ${Ko_{A/B/C/D}}^1, \dots, {Ko_{A/B/C/D}}^p$ *and* ${Kic_{A/B/C/D}}^1, \dots, {Kic_{A/B/C/D}}^p$ represent the K_o outputs and K_i inputs, respectively, of the NCL input register for the corresponding step, as labeled in Fig. 21. ${Ki_{A/B/C}}^1, \dots, {Ki_{A/B/C}}^q$ corresponds to the K_i inputs to the output register for the respective steps. Note that K_o, K_{ic}, and Ko are all threshold gate outputs, and are therefore accounted for in variable g.

The predicates used to construct the completion-completeness PO are shown in Table 13. p_0 indicates that all threshold gate output values are 0 for Step A, which indicates that the circuit is in the NULL state before a DATA transition. p_1 indicates that all circuit inputs during Step A are NULL. p_2 indicates that all circuit K_i inputs are 1, which indicates that the circuit is in a rfd state. p_3 symbolically steps the circuit stage under test using dual-rail inputs $({i_A}^1, \dots, {i_A}^p)$, K_i inputs $({Ki_A}^1, \dots, {Ki_A}^q)$ and threshold gate values $({g_A}^1, \dots, {g_A}^m)$, and stores the gate output values to $({g_B}^1, \dots, {g_B}^m)$. p_4 indicates that the circuit stage input being tested for completion-completeness, ${i_B}^k$, remains NULL, and all other inputs are DATA for Step B. p_5

Table 13. Predicates for completion-completeness check

p_n	Predicate
p_0	$\bigwedge_{n=1}^{n=m} (g_A{}^n = 0)$
p_1	$\bigwedge_{n=1}^{n=p} (i_A{}^n = 0b00)$
p_2	$\bigwedge_{n=1}^{n=q} (Ki_A{}^n = 1)$
p_3	$(g_B{}^1, \dots, g_B{}^m, Ko_B{}^1, \dots, Ko_B{}^p) = NCLStep(i_A{}^1, \dots, i_A{}^p, g_A{}^1, \dots, g_A{}^m, Ki_A{}^1, \dots, Ki_A{}^q)$
p_4	$\bigwedge_{n=1}^{n=p} (i_B{}^n = \begin{cases} 0b00, n = k \\ (0b01 \vee 0b10), n \neq k \end{cases})$
p_5	$(g_C{}^1, \dots, g_C{}^m, Ko_C{}^1, \dots, Ko_C{}^p) = NCLStep(i_B{}^1, \dots, i_B{}^p, g_B{}^1, \dots, g_B{}^m, Ki_B{}^1, \dots, Ki_B{}^q)$
p_6	$\bigwedge_{n=1}^{n=p} (i_C{}^n = 0b00)$
p_7	$\bigwedge_{n=1}^{n=q} (Ki_C{}^n = \begin{cases} 0, & o_B{}^n = (0b01 \vee 0b10) \\ 1, & o_B{}^n = 0b00 \end{cases})$
p_8	$(g_D{}^1, \dots, g_D{}^m, Ko_D{}^1, \dots, Ko_D{}^p) = NCLStep(i_C{}^1, \dots, i_C{}^p, g_C{}^1, \dots, g_C{}^m, Ki_C{}^1, \dots, Ki_C{}^q)$
p_9	$\bigwedge_{n=1}^{n=p} \begin{cases} (Ko_D{}^n = Kic_C = 1), n = k \\ \sim(Ko_D{}^n \barwedge Kic_C), n \neq k \end{cases}$

symbolically steps the circuit stage under test using the Step B inputs, and stores the gate output values to $(g_C{}^1, \dots, g_C{}^m)$. p_6 indicates that all circuit inputs during Step C are set to NULL. p_7 is used to constrain the K_i inputs for Step C, such that if a particular circuit output is DATA, then its corresponding K_i input is constrained to 0, indicating rfn; and if the circuit output is NULL, then its corresponding K_i input is constrained to 1, indicating rfd. p_8 symbolically steps the circuit stage under test using the Step C inputs and stores the gate output values to $(g_D{}^1, \dots, g_D{}^m)$. p_9 checks the K_{ic} and K_o values of the input register to ensure that they are

correct for input i^k being constrained to NULL (i.e., K_{ic} and K_o for input $register_k$ should both be 1, since i^k never transitioned from NULL; and the rest of the input register bits' K_{ic} and K_o should not both be 1, as this would allow the subsequent DATA wavefront to pass through into the C/L, thus violating the completion-completeness criteria). The completion-completeness proof obligation is constructed as follows:

$$\boldsymbol{PO8}: \{ p_0 \wedge p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7 \wedge p_8 \} \rightarrow p_9$$

At a high level, these predicates restrict the input under test so that it stays NULL and is therefore requesting DATA for all three symbolic steps (A, B, and C). The other inputs are not constrained and can transition from NULL to DATA and back to NULL, as allowed by their respective handshaking signals. If the circuit is completion-complete, then the unconstrained inputs can transition from DATA to NULL, but not back to DATA before the constrained input transitions to DATA and then to NULL. An unconstrained input that could transition back to DATA indicates that the circuit is not completion-complete. Essentially, p_9 indicates that none of the unconstrained inputs could transition back to DATA, which is what the SMT solver is checking. This can be observed when looking back to the Fig. 20 example in Section 5. The property is violated when the constrained input, *Y(1)* is tested, as both K_{ic0} and K_{oX0} are 1 after NCLStep C, such that $X_{i+1}(0)$ would be allowed to pass through into the C/L. If the solver can prove the PO, then this indicates that the circuit is completion-complete w.r.t. input *k*. If, however, there is a violation, then the solver will provide a counter example to the proof obligation, which can then be used to trace the source of the completion-completeness violation.

5.2.2. Completion-Completeness Results

For the verification results presented herein, partial-product generation of N-bit × N-bit unsigned dual-rail NCL multipliers were used as benchmarks, where $4 \leq N \leq 64$. The

verification proof obligations were checked using the Z3 SMT solver [10] running on an Intel® Core™ i5-6600k CPU with 16GB of RAM, operating at 3.50 GHz; however, any SMT solver could be used. The results are listed in Table 14, where the first column is *N*, corresponding to an N-bit × N-bit dual-rail NCL unsigned multiplier partial product generation circuit. The second column is the verification time in seconds of completion-complete multipliers that are constructed using only input-complete AND2 components, shown in Fig. 7. The third column is the verification time in seconds of completion-incomplete multipliers, where the input-complete AND2 components are replaced with their input-incomplete version, shown in Fig. 6, for partial products X_iY_j, where $i \neq j$. These are used to test the time to detect circuits that are input-complete but not completion-complete. The time reported for each completion-complete circuit is the total time to verify that all inputs are completion-complete; and the time reported for each completion-incomplete circuit is the total time until one input is found to be completion-incomplete. Z3 detected all completion-incomplete circuits and provided a counter example.

In addition to the multiplier partial product generation circuits, the other two circuits described in [8] where tested as well: a) the final stage of an unsigned multiplier with completion-complete and completion-incomplete GEN_S7 components, and b) the six 2-input AND function circuit. The developed automated completion-completeness verification method correctly verified the completion-complete versions and flagged the completion-incomplete circuits.

5.2.3. Completion-Completeness Conclusion

This chapter presents the first automated methodology for formal verification of completion-completeness of NCL circuits. The results are very promising, as even a 64 × 64 multiplier partial product generation circuit could be fully verified in 19.5 minutes. The

limitation to verification using the computer described above was not verification timeout of more than one day, but storage limitation as circuit size grew. Techniques to further improve efficiency and scalability could be explored as future work.

Table 14. Verification results of completion-completeness (in sec.)

N	Completion-Complete	Completion-Incomplete
4	2.785	.186
8	2.785	.186
12	7.131	.315
16	15.315	.525
20	30.135	.797
24	49.909	1.142
28	78.887	1.628
32	115.078	2.092
36	169.248	2.745
40	229.685	3.278
44	314.026	3.967
48	429.539	4.776
52	564.778	5.739
56	709.051	6.835
60	1042.379	8.469
64	1170.315	10.883

6. CONCLUSION

Proposed throughout this dissertation are new or improvements on existing formal verification methods for NULL Convention Logic (NCL) circuits. The goal of this dissertation work was to develop automated formal verification techniques that would check for equivalence when compared to a synchronous design and ensure delay-insensitivity through property checking of input-completeness, observability, and completion-completeness. Although optimizations and alternative methodologies to make formal techniques more scalable are always areas to investigate, this work serves as a baseline to help better integrate Quasi-Delay Insensitive digital design into the semiconductor industry.

6.1. Summary

In Chapter 1, an overview of formal verification and NULL Convention Logic (NCL) circuits are presented and serve as a background and motivation as to why this dissertation work was required. Chapter 2 dives into an existing equivalence checking technique and explores ways to improve the scalability of the pre-existing methodologies. The results were very promising, with the NCL to Synchronous Reduction reducing the state explosion significantly from previous works. The NCL to Synchronous Reduction was then further improved using an exploitation of the invariant property of NCL circuits at the register stage outputs. With this invariant, significant speedups were observed, allowing for verification of a 24+12×12 MAC, whereas the previous method would timeout. This is an exciting prospect because as the MAC size grows, the area and number of components grows along with it, but at an exponential pace. The rest of the dissertation focuses on circuit properties that ensure delay-insensitivity of NCL circuits. The contents of Chapters 3 and 4 revolve around property checking of input-completeness and observability for any arbitrary NCL circuit in an automated fashion. This was the first formal

method developed for input-completeness where all inputs were checked simultaneously, decreasing the amount of proofs required to ensure input-completeness. Previous input-completeness checks were done by checking observability of a circuit at the inputs. From the work in Chapter 4, it was concluded that this previous method was more complex and required more verification time than the proposed input-completeness check. Finally, in Chapter 5, a formal method was developed for checking the completion-completeness property of NCL circuits using bit-wise completion. As mentioned before, when developing NCL circuits, bit-wise completion is a useful tool to potentially increase throughput and decreases area. Unlike circuits that utilize full-word completion, ensuring input-completeness of circuits using bit-wise completion does not guarantee delay-insensitivity, which is the justification for the completion-completeness work. Those results are extremely promising, allowing verification of a 64×64 NCL multiplier partial-product generation circuit in less than 20 minutes.

6.2. Future Work

As with most formal verification techniques, additional refinements can be researched to improve scalability of previous techniques. For most of the methods presented herein, this could be a topic of future work, as verification times grow exponentially along with circuits size. For equivalence verification, the proposed invariant technique in Section 2.2.2 could be further applied at a more granular level (e.g. at the NCL component level) to potentially further reduce verification time, and could be a topic to investigate in the future. The property checking methods developed could be investigated for additional refinement techniques like the ones applied to equivalence checking. These refinements would change how the circuitry is modeled, but the proof obligation would stay the same. Another area that can be looked into is incorporating the work presented here into a commercial tool, such as JasperGOLD Sequential

Equivalence Checker, which is more prevalent in industry. The focus of most of this dissertation has been safety and liveness of NCL circuits, with problems arising from design flaws. A future area of research could be development of formal methods to detect circuitry inserted for malicious purposes, such as leaking private information.

REFERENCES

[1] K. M. Fant and S. A. Brandt, "NULL Convention Logic TM: a complete and consistent logic for asynchronous digital circuit synthesis," in *Application Specific Systems, Architectures and Processors, 1996. ASAP 96. Proceedings of International Conference on*, Aug 1996, pp. 261–273.

[2] **S. Le**, S. K. Srinivasan and S. C. Smith, "Automated verification of input completeness for NCL circuits," in *Electronics Letters*, vol. 54, no. 20, pp. 1158-1160, 4 10 2018. doi: 10.1049/el.2018.6068

[3] V. Wijayasekara, S. K. Srinivasan, and S. C. Smith, "Equivalence Verification for NULL Convention Logic (NCL) Circuits," *32nd IEEE International Conference on Computer Design* (ICCD), pp. 195-201, October 2014.

[4] 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.

[5] S. C. Smith and J. Di, "Designing Asynchronous Circuits using NULL Convention Logic (NCL)," *Synthesis Lectures on Digital Circuits and Systems*, Morgan & Claypool Publishers, Vol. 4/1, July 2009.

[6] C. Jeong and S. M. Nowick, "Optimization of robust asynchronous circuits by local input completeness relaxation," *Asia and South Pacific Design Automation Conference*, Jan. 2007, pp. 622–627.

[7] A. Kondratyev, L. Neukom, O. Roig, A. Taubin, and K. Fant, "Checking delay-insensitivity: 10^4 gates and beyond," *8th International Symposium on Asynchronous Circuits and Systems*, April 2002, pp. 149–157.

[8] S. C. Smith, “Completion-Completeness for NULL Convention Digital Circuits Utilizing the Bit-wise Completion Strategy,” International Conference on VLSI, pp. 143-149, June 2003.

[9] D. Monniaux, “A survey of Satisfiability Modulo Theory” [online]. Available: https://hal.archives-ouvertes.fr/hal-01332051/document [Accessed May 25, 2020].

[10] 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.

[11] D. Bryan, “The ISCAS '85 benchmark circuits and netlist format” [online]. Available: https://ddd.fit.cvut.cz/prj/Benchmarks/iscas85.pdf [Accessed: May 25, 2020].

[12] [Online] http://www.pld.ttu.ee/~maksim/benchmarks/iscas89/bench/, Accessed: May 25, 2019.

[13] A. A. Sakib, **S. Le**, S. C. Smith, and S. K. Srinivasan, “Formal Verification of NCL Circuits,” in Asynchronous Circuit Applications, pp. 309-338, IET, December 2019.

[14] L. Zhou, R. Parameswaran, F. Parsan, S. C. Smith, and J. Di, “Multi-Threshold NULL Convention Logic (MTNCL): An Ultra-Low Power Asynchronous Circuit Design Methodology,” Journal of Low Power Electronics and Applications, Vol. 5/2, pp. 81-100, May 2015.

[15] M. Hossain, A. A. Sakib, S. K. Srinivasan, and S. C. Smith, “An Equivalence Verification Methodology for Asynchronous Sleep Convention Logic Circuits,” IEEE International Symposium on Circuits and Systems, pp. 1-5, May 2019.

[16] S. C. Smith, R. F. DeMara, J. S. Yuan, D. Ferguson, and D. Lamb, “Optimization of null convention self-timed circuits,” *Integr. VLSI J.*, vol. 37, no. 3, pp. 135–165, Aug. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.vlsi.2003.12.004.

[17] C. L. Seitz, *Introduction to VLSI Systems*. Boston, MA, USA: Addison Wesley Longman Publishing Co., Inc., 1979, ch. System Timing, pp.218–262.

[18] “ISCAS-85 c432 27-channel interrupt controller” [online]. Available: http://web.eecs.umich.edu/~jhayes/iscas.restore/c432.html [Accessed: May 25, 2020].

[19] “ISCAS-85 c432 27-channel interrupt controller Module M3” [online]. Available: http://web.eecs.umich.edu/~jhayes/iscas.restore/c432m3.html [Accessed: May 25, 2020].

[20] “ISCAS-85 c432 27-channel interrupt controller Module M1” [online]. Available: http://web.eecs.umich.edu/~jhayes/iscas.restore/c432m1.html [Accessed: May 25, 2020].

APPENDIX. LIST OF PUBLICATIONS

- **S. N. Le**, S. K. Srinivasan and S. C. Smith, "Automated Verification of Input-Completeness for NCL Circuits," in *Electronics Letters*, vol. 54, no. 20, pp. 1158-1160, 2018.
- A. A. Sakib, **S. N. Le**, S. C. Smith, and S. K. Srinivasan, "Chapter 15: Formal Verification of NCL Circuits", in *Asynchronous Circuit Applications*, Institution of Engineering and Technology (IET), London, UK, pp. 309-338, 2019.
- **S. N. Le**, S. K. Srinivasan, S. C. Smith, "Exploiting Dual-Rail Register Invariants for Equivalence Verification of NCL Circuits", *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2020 (accepted).
- **S. N. Le**, S. K. Srinivasan, S. C. Smith, "Formal Verification of Completion-Completeness for NCL Circuits", *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2020 (accepted).