FORMAL VERIFICATION METHODOLOGY FOR ASYNCHRONOUS SLEEP CONVENTION LOGIC CIRCUITS BASED ON EQUIVALENCE VERIFICATION

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

By

Mousam Hossain

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Electrical and Computer Engineering

July 2019

Fargo, North Dakota

North Dakota State University
Graduate School

Title

FORMAL VERIFICATION METHODOLOGY FOR ASYNCHRONOUS SLEEP CONVENTION LOGIC CIRCUITS BASED ON EQUIVALENCE VERIFICATION

By

Mousam Hossain

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

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Sudarshan K. Srinivasan
Chair

Dr. Scott C. Smith
Co-Chair

Dr. Ying Huang

Approved:

07/22/2019
Date

Dr. Benjamin Braaten
Department Chair

ABSTRACT

Sleep Convention Logic (SCL) is an emerging ultra-low power Quasi-Delay Insensitive (QDI) asynchronous design paradigm with enormous potential for industrial applications. Design validation is a critical concern before commercialization. Unlike other QDI paradigms, such as NULL Convention Logic (NCL) and Pre-Charge Half Buffers (PCHB), there exists no formal verification methods for SCL. In this thesis, a unified formal verification scheme for combinational as well as sequential SCL circuits is proposed based on equivalence checking, which verifies both safety and liveness. The method is demonstrated using several multipliers, MACs, and ISCAS benchmarks.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sudarshan Srinivasan, and my co-advisor, Dr. Scott Smith, for giving me this opportunity to pursue Master of Science degree at NDSU under their supervision. I am very grateful for their excellent academic guidance, support, motivations and advising throughout my degree, which undoubtedly helped me find the true joy in research. I would like to thank my committee member, Dr. Ying Huang, for her valuable suggestions and feedback. I am also thankful to the department of Electrical and Computer Engineering at NDSU for providing the financial assistance to support my education. Last but not the least, I would like thank my parents and sister, for their relentless efforts and support, without which none of this would be possible, and my dear husband for his emotional support during my difficult times.

DEDICATION

To my father Dr. Mossaraf Hossain, mother Dr. Basera Khatun, sister Dr. Shabnam Banu and my loving husband Ashiq Sakib. You are my rock.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

GHz	Giga-Hertz.
PVT	Process, Voltage, and Temperature.
ITRS	International Technology Roadmap for Semiconductors.
NCL	NULL Convention Logic.
DI	Delay Insensitive.
EMI	Electro Magnetic Interference.
MTNCL	Multi-Threshold NULL Convention Logic.
SCL	Sleep Convention Logic.
MTCMOS	Multi-Threshold Complementary Metal Oxide Semiconductor.
QDI	Quasi-Delay Insensitive.
PCHB	Pre-Charge Half Buffers.
DIMS	Delay Insensitive Circuits using Multi-Ring Structures.
Rfn	Request for Null.
Rfd	Request for Data.
PUN	Pull Up Network.
PDN	Pull Down Network.
DFT	Design for Testability.
WEB	Well-founded Equivalence Bisimulation.
HA	Half Adder.
FA	Full Adder.
SMT-Lib	Satisfiability Modulo Theory Library.
MAC	Multiply and Accumulate.

LIST OF SYMBOLS

$\forall$.....For All.

$\wedge$ .....Logical AND Operation.

$\neg$.....Logical NOT Operation.

$\Rightarrow$.....Implies.

$\in$ .....Belongs to.

1. INTRODUCTION

Over the past many decades, the semiconductor industry has been predominantly ruled by synchronous design paradigm. Majority of the electronic devices that we use today are synchronous in nature. Utilization of a global clock for synchronization and the assumption that all signals are binary have contributed significantly to the simplification of logic design. Clocked designs are deterministic in that all signals are sampled at a definite time interval, which makes testing and validation easier. However, today's semiconductor industry is facing a huge challenge in synchronous domain coping up with the ever increasing consumer expectations of faster, robust, and more compact devices. There are three major limiting factors: clock management, increasing power consumption, and susceptibility to process variations. As the operating frequency hits Giga-Hertz (GHz) level for faster designs, clock management becomes considerably challenging, resulting in multidimensional clock related issues, such as, clock-skew, clock jitter, clock distribution etc. Interconnect delay or wire delay, which was often ignored in previous low frequency designs, becomes a major contributor to the design complexity and timing analysis. Additional clock distribution networks are required to manage the clock to attain acceptable skew, which adds to the area overhead. Moreover, similar to the clock frequency, Moore's Law that has been the foundation of IC chip design for the past so many decades, is finally reaching a saturation. Further integration and miniaturization results in excessive power consumption in devices, reducing the overall efficiency. In nanoscale designs, leakage power consumption during idle mode turns out to be an important factor. The clock hinders the power performance as well. The clock driver unit is a large component with numerous gates that undergoes continuous switching to manage the clock, even in idle mode, when the circuit is not performing useful tasks. Moreover, Process, Voltage, and Temperature

(PVT) variations become very critical in deep submicron level designs. Because of the limitations new avenues of research are being conducted to tackle these issues.

Asynchronous design is a clockless approach that alleviates the major issues faced in the present synchronous domain. Like synchronous, asynchronous logic is based on the assumption that all signals are binary. However, it removes the synchronous assumption that time is discrete. Asynchronous designs do not use clock and eliminates all clock related issues. Synchronization between components are attained by a handshaking scheme. Transmission and reception of data is based on request and acknowledgement, which yields average-case performance, while synchronous design yields worse-case performance. In asynchronous design, transition or switching occurs in the components involved in computation only. This significantly improves power performance during active mode. Also, unlike synchronous design, there is no clock driver network that is always on, which further reduces power consumption. QDI designs are inherently robust against PVT variation that makes them an excellent design choice for extreme environmental applications.

These advantages over the synchronous domain have made the asynchronous paradigm quite popular in industry, as well as academic research as evidenced by the International Technology Roadmap for Semiconductors (ITRS). ITRS predicts that asynchronous logic will account for over 50% of the of the billion dollar semiconductor industry by 2027 [1].

1.1. NCL and SCL: Benefits

NULL Convention Logic [2] is the most widely used and popular Delay-Insensitive (DI) asynchronous paradigm. They have found numerous industrial applications in companies such as Theseus Logic, Ozark Integrated Circuits, Eta Compute etc. NCL circuits have much lower power consumption, glitch power, and Electro Magnetic Interference (EMI) as compared to

corresponding synchronous designs, by possessing an inherently idle behavior, which considerably reduces switching power consumption. They also require very less timing analysis; hence, design process is much faster as compared to synchronous system design. NCL designs are also correct by construction, and the framework is very similar to synchronous systems, due to which automation of these designs are similar to that of synchronous systems and presents a much flatter learning curve for designers.

Multi-Threshold NCL (MTNCL) or most commonly known as Sleep Convention Logic (SCL), is a modification over NCL architecture. With reduced supply voltages and sub-micron size miniaturizations, sub-threshold leakage power constitutes a considerable portion of the total power dissipation in the circuit. MTNCL or SCL framework was developed to address this issue, which combined NCL with Multi-Threshold CMOS (MTCMOS) techniques to further reduce leakage power consumption during idle mode, resulting in an ultra-low power design paradigm. Various SCL frameworks are illustrated in [3], which demonstrates that the best SCL framework outperforms corresponding NCL designs in all aspects, and were significantly more power and area efficient as compared to synchronous MTCMOS designs.

1.2. Motivation

The key to widespread utilization and acceptance of any new design or framework in the semiconductor design industry lies in the availability of standard tools for synthesis and validation. Though several design methodologies exist for constructing asynchronous circuits from their corresponding synchronous designs [4, 5, 6, 7, 8], the area of validation and formal verification is still being ventured into only recently. Formal verification is of utmost importance in current ASIC design flow. As most modern ASIC designs become increasingly vast and complex, ensuring functional correctness becomes a critical task, which cannot be done through

simulation alone. Simulation based functional testing can be only as effective as the test cases designed, which often leads to rare case bugs going undetected during the design validation phase. Also due to the increasing vastness of designs, 100% functional coverage is becoming quite impossible to achieve, as it takes a huge amount of time and money, which in turn would lead to delays in time-to-market schedule. Formal verification techniques are used in conjunction with testing in the semiconductor industry to detect the rare case bugs, and ensure complete functional correctness. In order to increase the acceptance and incorporation of asynchronous designs in the existing synchronous dominated semi-conductor industry, developing formal verification schemes for the various QDI asynchronous paradigms have recently become a rapidly growing research field. Few formal verification schemes have been developed for different QDI paradigms, but these techniques are not directly applicable to SCL circuits. Design for Testability technique [9] with very high-test coverage exist for SCL circuits, but these are not formal verification techniques. Therefore, this thesis work illustrates a formal verification technique based on equivalence checking for SCL circuits that ensures both safety and liveness of the circuits. This is the first known formal verification work applicable to SCL circuits.

1.3. Thesis Overview

The thesis is organized as follows: Chapter 2 provides an overview of the NCL and SCL frameworks, background, and illustrates the differences between the two architectures. The proposed formal verification method is presented in Chapter 3. The method to verify the safety, liveness, and handshaking checks for combinational as well as sequential SCL circuits are illustrated, followed by demonstration of the method verifying numerous benchmark circuits. Finally, conclusions and future work are discussed in Chapter 4.

2. ASYNCHRONOUS BACKGROUND

Asynchronous paradigm has a vast number of design methodologies that have been developed over the decades. Each of these design methodologies have their own advantages and disadvantages, which help to decide their utilization based on specific application requirements. All these models fall into two major categories based on delay assumptions – Bounded-Delay and Delay-Insensitive (DI).

In bounded-delay model of asynchronous designs, it is assumed that the delay of the various circuit elements and the connecting paths is known, or bounded within certain limits. Bounded delay models require extensive timing analysis to determine the delay in the datapath, so that it can be matched to the control path delay, to achieve synchronization between datapath and control path in the absence of clock. There are various asynchronous circuits based on bounded-delay model, such as, Huffman circuits [10], burst-mode circuits [11], and micropipelines [12]. However, the bounded-delay model has some limitations, such as worst case performance.

The Delay-Insensitive (DI) model is based on the primary assumption that the delays in both logic elements and wires are unbounded, which means that data at the inputs can arrive at any point in time. Hence, there is no bound on the delay of its arrival as well as on the delay of obtaining the correct output. Well-defined handshaking schemes are utilized along with specific completion detection mechanisms so that the receiver can notify its sender on the proper reception and computation of the received signals. The sender waits on an acknowledge signal from the receiver before sending the next set of data inputs. This allows a datapath element to start working on a new set of inputs early after finishing computation on the previous input data set, or stall the previous stage when more time is required to finish computation. This yields

average case performance instead of worst case performance as compared to the bounded-delay models. It also avoids hazards. While such a signaling protocol may somewhat complicate the circuit layout and implementations, it provides the advantage of separating circuit correctness from specific delay assumptions. However, despite the advantages in terms of not requiring complex timing analysis, practical circuits cannot be designed in DI paradigm due to lack of expressible conditionals [13]. Instead, Quasi-Delay Insensitive (QDI) methods are utilized for practical implementation, which allows a small relaxation on the unbounded delay assumption. The QDI model assumes that the component delays are much larger as compared to the interconnect delays within a component; i.e. wire forks within a component are isochronic in nature. In practice, circuits that are most commonly referred as Delay-Insensitive (DI) are actually QDI.

2.1. Delay Insensitive Methods: Related Works

Delay-Insensitive methods can be sub-divided into two main categories based on their synthesis levels: transistor-level delay-insensitive methods and gate-level delay insensitive methods. Martin's method [14] provides an approach to transistor level DI synthesis from high-level program description based on formal derivation using certain codes and theorems. But it does not target a previously pre-defined set of logic gates, hence not applicable directly to existing synchronous systems. Pre-Charge Half Buffer (PCHB) circuits are based on dynamic logic, and are synthesized at transistor level. PCHB utilizes a fine-grained pipelined architecture and provides design flexibility, which made this paradigm commercially successful.

Some popular gate-level delay-insensitive methods are developed by Seitz [15], Singh [16], Anantharaman [17], David [18] and the Delay Insensitive Circuits using Multi-Ring Structures (DIMS) approach by Sparso [19]. All of these methods incorporate completion

detection units in order to ensure correct circuit operation. Muller C-elements are used [20] as the only state-holding element in these circuits. C-elements operate such that the output changes to the input value only when all the inputs assume the same value, either Boolean '0' or '1'; otherwise the C-element output holds its previous value. This property of C-elements helps in achieving delay-insensitivity. All the above mentioned methods yield average-case performance compared to worst-case performance of bounded-delay models and synchronous circuits. Seitz's, Anatharaman's, and DIMS approach require the generation of full min-term expressions for all the output signals, which nullifies the scope of any optimizations. In comparison, David's and Singh's methods do not need full min-term generation. David designs DI circuits using four kind of subnets, namely n-input C-Element (CEN), n 2-input OR gates (ORN), DRN, which is extraction of individual rails of dual-rail outputs, and the 2-m dual input C-elements network (OUTN) producing the circuit output; n and m being the number of inputs and outputs of the DI circuit, respectively. Singh devices twelve modules, the various combinations of which can be used to design circuits in DI paradigm. The modules are of two main types: one 'user' module and the rest 'control' modules. User modules are designed by the designer and control modules are fixed functionality units (for routing data from input to output ports) with user defined data widths of inputs and outputs.

The most popular gate-level QDI models are the NULL Convention Logic (NCL) [2] and Multi Threshold NULL Convention Logic (MTNCL), also known as Sleep Convention Logic (SCL) [3]. NCL has been widely accepted in today's semiconductor industry due to the various advantages it offers over the previous mentioned gate-level and transistor level DI models. NCL does not require full minterm generation which leads to better optimization scopes. NCL circuits have automated synthesis tools [8], and much work has been carried out in developing

optimization techniques for such circuits, based on Threshold Combinational Reduction (TCR) [21], NULL Cycle Reduction [22], glitch power reduction [23], throughput optimization using gate-level Pipelining [24], and optimization of NCL self-timed rings [25]. NCL has numerous threshold gates for realizing delay-insensitivity, which are also capable of executing Boolean functions of maximum four variables. NCL is also easily incorporable into current semiconductor industry due to similarity in framework with synchronous designs. This thesis deals with verification of SCL modules, which is a modification over NCL module, achieving low power and transistor count optimizations. Hence, in order to better understand SCL framework an overview of NCL framework is first provided in Section 2.2, followed by a detailed description of SCL in the Section 2.3.

2.2. NCL Overview

NCL circuits are different from their synchronous counterparts in that they use multi-rail logic, such as dual-rail logic and quad-rail logic, compared to the single rail encoding in synchronous. Dual-rail encoding is the most popular design choice. In dual-rail logic, two rails are used to encode each signal or bit of data. If D^0 and D^1 are the two rails of a dual rail signal, D, then *DATA0* is represented as D^1 = '0', D^0 = '1', which is equivalent to Boolean logic '0', *DATA1* is represented as D^1 = '1', D^0 = '0', which is equivalent to Boolean logic '1', and $D^1 = D^0$ = '0' corresponds to a *NULL* state representing absence of DATA. Similarly, in quad-rail logic, four wires are used to encode two bits of Boolean variables. Let Q^0, Q^1, Q^2, and Q^3 represent the four rails of quad-rail signal, Q, encoding two Boolean variables, A and B. The encoding scheme is shown in Table 1. Both dual-rail and quad-rail logic are referred to as one hot encoding schemes, which means that only one wire can be asserted at a time, whereas all other wires will remain de-asserted. More than one wire asserted simultaneously is an illegal state. This sort of

data encoding approach leads to monotonic transition from DATA to NULL for inputs and outputs. Thus, helps to eliminate timing reference from the circuits and achieve delay-insensitivity.

Table 1. Quad-Rail Encoding Scheme.

	Q^3	Q^2	Q^1	Q^0	**Boolean Equivalent**
NULL	0	0	0	0	Data absent
DATA0	0	0	0	1	A = 0, B = 0
DATA1	0	0	1	0	A = 0, B = 1
DATA2	0	1	0	0	A = 1, B = 0
DATA3	1	0	0	0	A = 1, B = 1

The operation of NCL model is discussed next. The main advantage of NCL architecture is its similarity with synchronous pipeline architecture, where each combinational unit is sandwiched between two registers. The basic M-stage NCL framework is shown in Fig. 1. For feed-forward NCL circuits, at least one set of input registers and output registers are necessary for correct flow of DATA/NULL. Multiple intermediate register stages can be added to improve throughput [26], as depicted in Fig. 1, where $R_2 – R_{M-1}$ are the intermediate registers. However, for sequential NCL circuits every feedback path requires at least *2N+1* registers in the feedback loop for *N* data tokens to avoid deadlock [40]. The NCL combinational unit, registration unit and the completion unit are described in the next sub-sections.

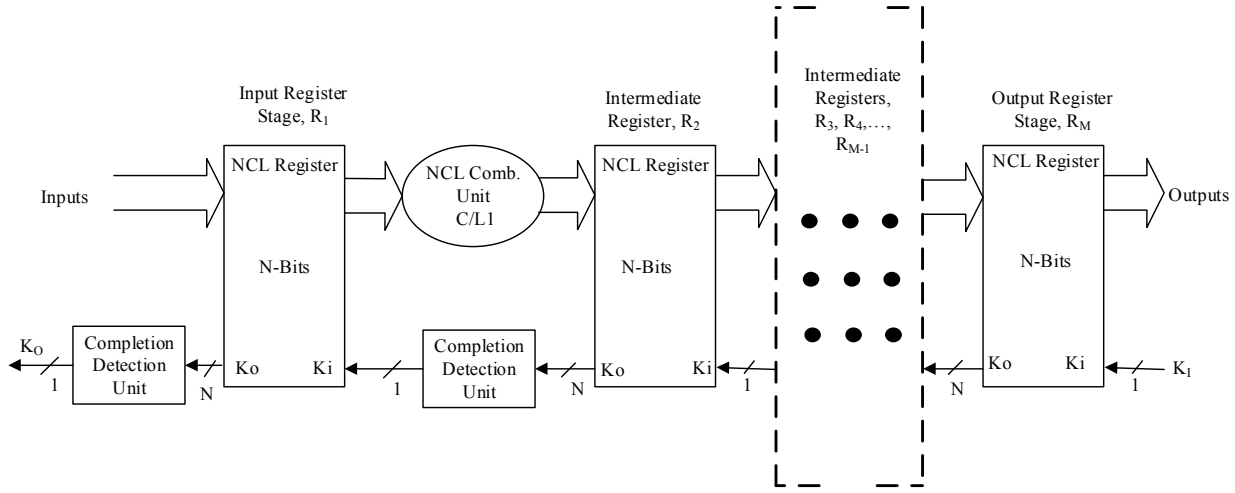
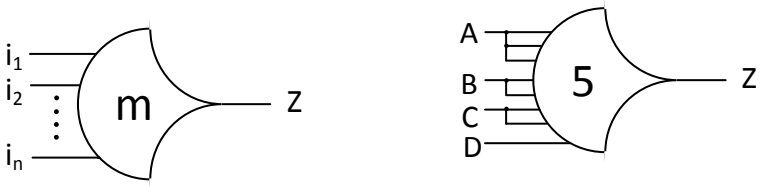


Figure 1. NCL Framework.
Adapted from [2].

2.2.1. NCL Combinational Unit

NCL combinational unit consists of 27 threshold logic gates [2] with hysteresis state holding capability. Individual threshold gates can be used to implement functions of four or fewer variables. Here, individual variable corresponds to each separate rail of a dual-rail signal. A simple NCL threshold gate is written as *THmn* gate is shown in Fig. 2(a), where *n* is the number of inputs (i_1, i_2... i_n) and the gate produces a single output (Z). *m* is the gate *threshold*, which is the minimum number of inputs that are required to be asserted to assert the output, *Z*. Similarly, the output will be de-asserted when all of the *n* inputs to the gate are de-asserted. Otherwise, the gate will hold its previous state. Threshold gates can also be of *weighted* type, where individual input can be assigned certain weights. Such a gate is represented symbolically as $THmnWw_1w_2w_3...w_k$, where *'W'* stands for weighted gate and w_1, w_2 ... w_k are the weights associated with $input_1$, $input_2$ ... $input_k$, respectively. A TH54W322 threshold gate is shown in Fig. 2(b). TH54W322 gate has 4 inputs with gate threshold value of 5. The weight of input *A* is 3, *B* and *C* are 2, and *D* has a weight of 1. As the gate threshold is 5, any combination of inputs that sums up to a weight of 5 can assert the gate. For example, both *A* and *B* getting asserted can

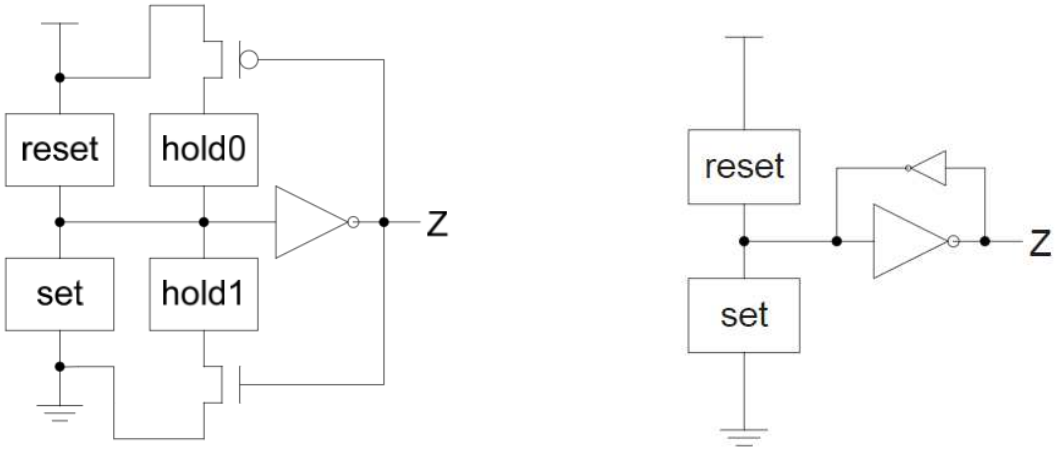
assert the gate output, *Z*, as they have a combined weight of 5. Therefore, the gate function can be written as Z = AB + AC + BCD.



(a) (b)

Figure 2. (a) THmn Gate (b) TH54W322 Gate.

There are two widely utilized transistor level implementations of these threshold gates – static and semi-static [27]. All these implementations have *set* and *reset, hold0* and *hold1* functions, arranged as shown in Fig. 3. *set* functions determine when the gate output will become asserted depending on when the threshold number of inputs become asserted. The *reset* function determines when the gate output will be de-asserted, which is when all gate inputs are de-asserted. In static implementation, as shown in Fig. 3(a), *hold0* and *hold1* functions are utilized to attain hysteresis state-holding functionality, whereas in case of semi-static implementation, the hysteresis is achieved by adding a weak inverter in feedback loop to the output as depicted in Fig. 3(b). *hold0* and *hold1* functions enable the gate to remain asserted or de-asserted until either the *reset* or *set* function is met, respectively. *reset* and *hold1* are generic in structure for an *n*-input threshold gate, whereas *set* and *hold0* functions vary depending on the NCL gate functionality. *reset* is derived by connecting all the complemented gate inputs in series. *hold1* function is complement of *reset* function, i.e. all the gate inputs connected in parallel. *set* functional block is simply the NMOS implementation of the Boolean function of the threshold gate obtained after further simplification. *hold0* is obtained by complementing the set function and applying further simplifications [2].



(a) (b)

Figure 3. (a) Static NCL Implementation (b) Semi-Static NCL Implementation [2].

For the TH54W322 gate the *set* function is AB + AC + BCD, as discussed earlier. The *reset* function is *A'B'C'D'*, the *hold1* function is *A+B+C+D*, and the *hold0* function is the complement of the *set* function. Based on the threshold gate implementation template on Fig. 3, the static and semi-static representation of the TH54W322 gate is shown in Fig. 4 and 5, respectively.

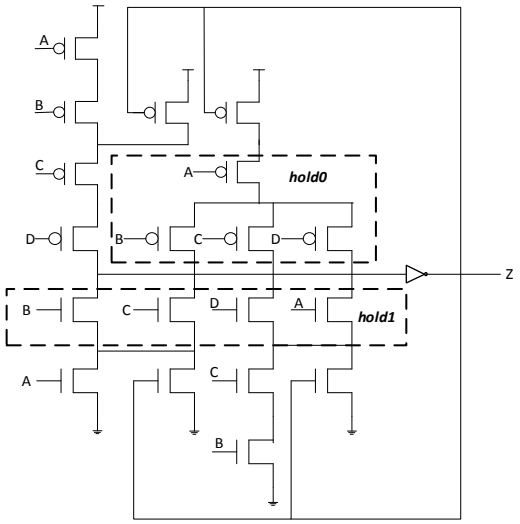


Figure 4. Static Implementation of TH54W322 Gate.

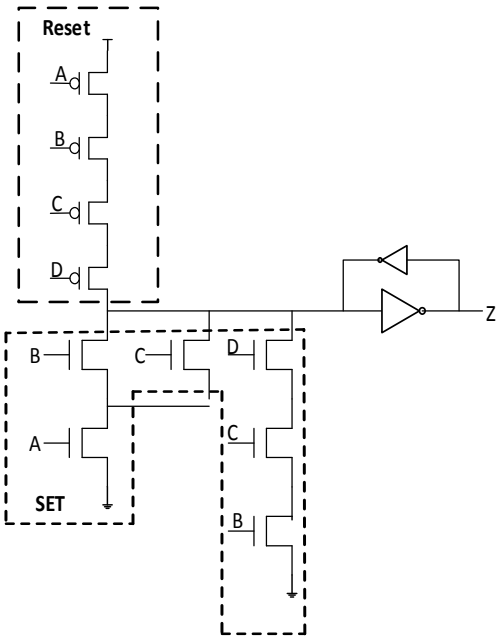


Figure 5. Semi-static Implementation of TH54W322 Gate.

2.2.2. NCL Registration Unit

In NCL framework, the registration units serve to maintain the proper flow of DATA and NULL in the system. NCL combinational circuit consists of at least two register stages, one at the input and the other at the output, with the combinational logic placed in-between these two stages. Each register stage comprises a set of cascading dual-rail or quad-rail registers. Each register is internally made of TH22 gates. The structure of a single bit reset-to-NULL dual-rail register is shown in Fig. 6. One input of each TH22n (reset-to-zero TH22) gate comes from the $rail^1$ and $rail^0$ of the input, whereas the other input is the *request* signal, K_i. The two rails of the output are input to an inverted TH12 gate, which is a NOR gate. The output of the inverted TH12 gate produces the *acknowledge* signal, K_o. The register stages communicate between each other through a four-phase handshaking mechanism, using K_i and K_o. When K_i is request for data (*rfd*) i.e. logic '1' and the dual-rail input is DATA, then the dual-rail output of the register becomes DATA. Similarly, the output will be NULL only when the input is NULL and K_i is requesting

for NULL (*rfn*) i.e. logic '0'. When the output is DATA/NULL, K_o becomes 0/1, requesting for NULL/DATA at the input of the register. The register can be reset to either DATA0 or DATA1 by replacing one of the two TH22n gates with a TH22d gate, which is reset to logic 1.

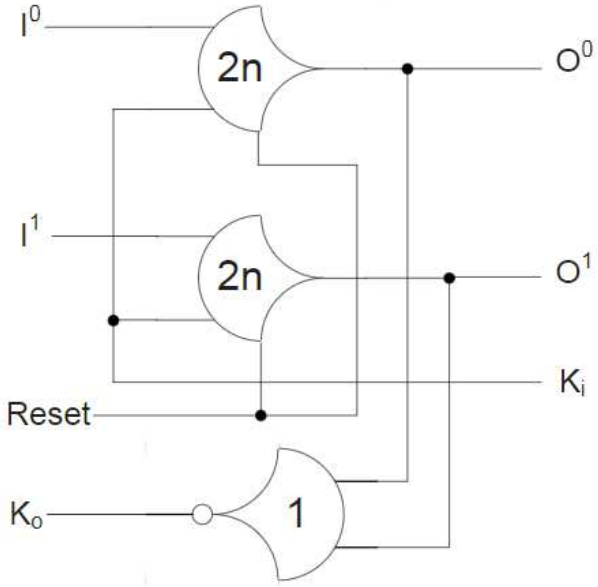


Figure 6. Single-bit Dual-rail Register Reset to NULL [2].

2.2.3. NCL Completion Unit

NCL completion unit is a network of THnn gates arranged in a tree structure. THnn gates are an N-input C-element. The completion unit takes the N-bit K_o outputs from the next stage registers and combines them to form a single-bit K_o output, which is fed back as the K_i input of the registers in the previous stage, as shown in Fig. 1. As the maximum number of inputs to a threshold gate is four, the number of levels of THnn gates for N-bit inputs is given by $\lceil log_4 N \rceil$. Fig. 7 shows an N-bit completion unit structure.

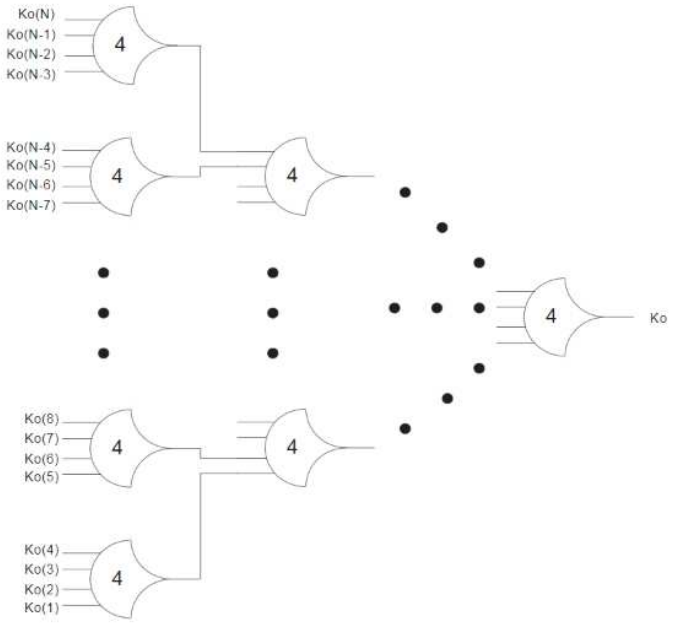


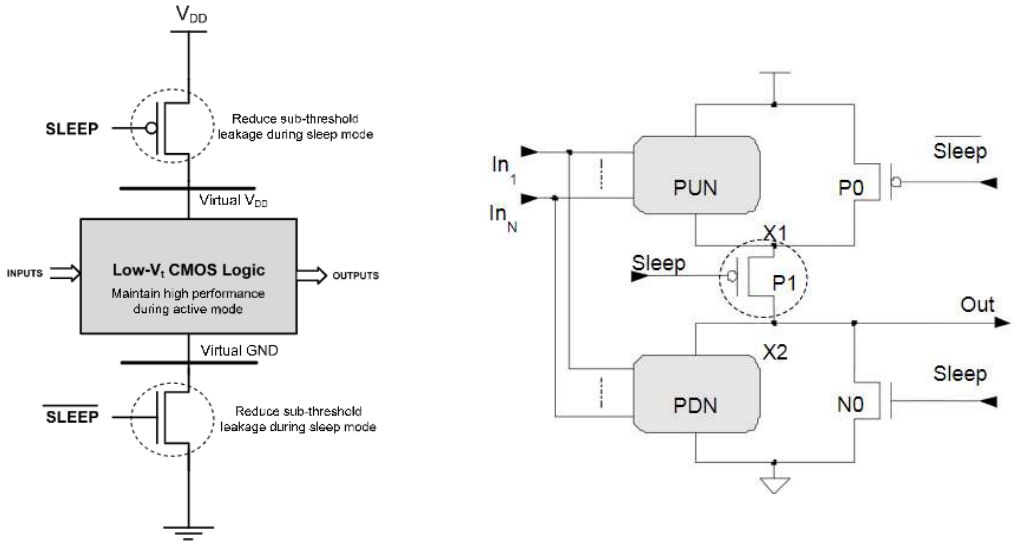
Figure 7. N-bit Completion Tree Structure [2].

2.3. Sleep Convention Logic Overview

Sleep Convention Logic (SCL) is an ultra-low power, high speed QDI asynchronous paradigm, which is a modified version of the popular NCL architecture [28]. SCL integrates the Multi-Threshold CMOS (MTCMOS) technique for leakage power reduction with NCL; hence, are often termed as Multi-Threshold NULL Convention Logic (MTNCL) [3].

MTCMOS technique is typically implemented using two or more threshold voltages in the circuit. In one kind of MTCMOS application, low threshold voltage transistors (low-V_t) are used to achieve faster switching speeds during active-mode, and high threshold voltage transistors (high-V_t), controlled by *Sleep* signal, are used to gate the power supply from the circuit during idle-mode, thus reducing sub-threshold leakage current, as shown in Fig. 8(a). But sizing of the *Sleep* controlled high-V_t transistor poses a serious design challenge for larger circuits. An alternate method is developed to tackle this issue, where MTCMOS technique is applied to each gate, as shown in Fig. 8(b). In this case, the gate logic is implemented in CMOS

using Pull-up (PUN) and Pull-down (PDN) networks. The PUN and PDN are separated by a high-V_t *Sleep* transistor, marked by a dotted circle in Fig. 8. In idle mode, *sleep* is asserted (Sleep = '1') and the *Sleep* transistor is turned OFF, which disconnects the PUN from the PDN. This arrangement thus reduces leakage power in the circuit during idle mode. The *P0* PMOS and *N0* NMOS transistors remain ON when *Sleep* is asserted, thus the output node is pulled down to '0'. Although MTCMOS technique reduces power consumption, they still have serious limitations in synchronous domain; such as, area overhead, possibility of losing data during cut-off mode, and logic partitioning. However, by incorporating MTCMOS with NCL, these drawback are eliminated in SCL architecture.



(a) (b)

Figure 8. (a) General MTCMOS Architecture (b) Boolean Gate Implementation [3].

A typical multi-stage SCL framework is shown in Fig. 9. Every pipeline stage in SCL framework comprises of a DI register block (R_i), a combinational block (C/L) and a completion detection unit (C_i). The self-timed phase alterations between DATA and NULL wave fronts and synchronization is achieved through the mutual handshaking between these three basic blocks of different pipeline stages, which will be explained in detail in the following sub-sections. There

are several architectures available for implementing handshaking in SCL circuits. The SECRII w/o *nsleep* architecture [3], as shown in Fig. 9, being the fastest one has been chosen for the circuit implementations in this thesis.

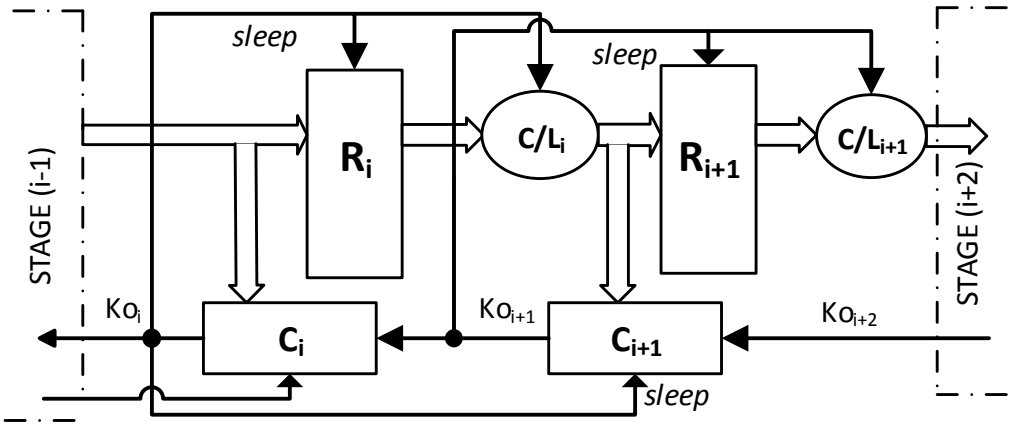


Figure 9. SCL SECRII Without *nsleep* Architecture Framework.
Adapted from [3].

2.3.1. SCL Combinational Unit

SCL circuits comprise of SCL threshold gates which are a variant of the 27 fundamental NCL gates, where *hold1* block, *reset* block, and the corresponding NMOS and PMOS bypass transistors of static implementation of NCL gates (Fig. 3(a)) are removed, and an additional *Sleep* signal is incorporated in each gate [3]. A modified version of this implementation is called the Static MTNCL or SMTNCL implementation, shown in Fig. 10, where the high-V_t transistor, separating the pull-up and pull-down networks (which was shown in Fig. 8(b)), is moved to the pull down network. All of the PMOS transistors are turned ON only when the inputs are logic '0' and *Sleep* = '1', i.e. in idle mode. They stay in this state until *Sleep* = '0' and the gate's *set* function evaluates to true. All of the PMOS transistors, except the one in the output inverter, are high-V_t in nature. In active mode (*Sleep* = '0'), the circuit performs the logic function implemented by the threshold gate. During idle mode, *Sleep* = '1' and *nSleep* (complement of

Sleep signal) = '0'. Hence, the low-V_t transistor at the output pulls the output node to ground during idle mode. Whereas, the high-V_t transistor in the Pull-down network cuts off the logic circuit from ground resulting in reduced leakage power.

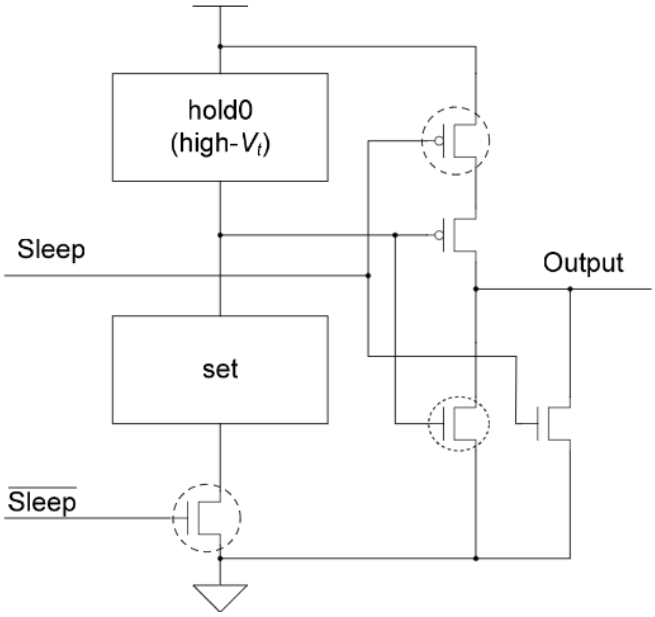


Figure 10. SMTNCL Gate Level Implementation [3].

As the above STMNCL implementation required *Sleep* signal and generation of its complement signal i.e. *nSleep*, further improvements were suggested to this architecture, which did not require the *nSleep* generation, thus reducing area and energy. SMTNCL w/o *nsleep* implementation is shown in Fig. 11. Here the *set* function is implemented in Bit-Wise MTNCL fashion (BWMTNCL) [29], such that, it is gated from the ground by having at least one high-V_t NMOS transistor in each path through *set* function to the ground, in a way that there are a minimum number of such high-V_t transistors. A SMTNCL TH54W22 gate w/o *nSleep* is shown in Fig. 12.

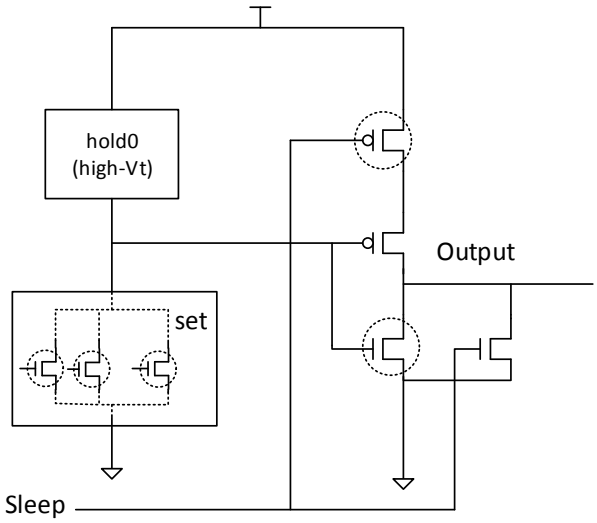


Figure 11. SMTNCL w/o *nsleep* Gate Level Implementation.
Adapted from [3].

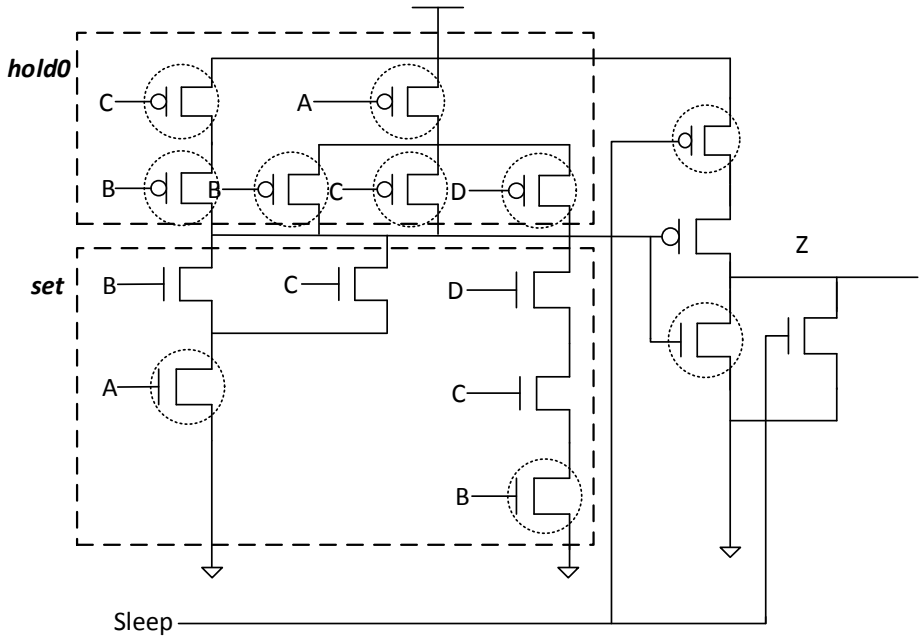


Figure 12. SMTNCL w/o *nSleep* Implementation of TH54W322 Gate.

2.3.2. SCL Registration Unit

SCL registration unit is an arrangement of multiple N-bit dual-rail Delay-Insensitive *sleep* registers in different stages. The transistor level diagram of a single bit dual-rail SCL register w/o *nSleep* is shown in Fig. 13. It consists of two TH22 gates in SMTNCL

implementation [3], with the *nSleep* signal removed and their sleep transistors combined. The two rails of the input I, I^0 and I^1, are fed into each TH22 gate and the register unit produces a dual-rail output O. An N-bit dual-rail SCL register stage is a combination of N single bit dual-rail SCL registers, each having the internal structure as shown in Fig. 13. In active mode, i.e. *Sleep* = '0', either I^0/ I^1 being asserted results in corresponding output rail, O^0/ O^1 to be asserted, respectively. The output remains latched at this value, irrespective of the input being asserted or de-asserted, until the *Sleep* mode is activated. When *Sleep* is asserted, the NMOS transistors at the output nodes are turned ON, resulting in output O^0 and O^1 being pulled down to ground. This condition is equivalent to the register unit passing a NULL input through to the output. Therefore, instead of waiting for the NULL wavefront to propagate through the stages, the register stage is slept to NULL by the *Sleep* signal, resulting in power saving during idle mode. Note that, unlike NCL, SCL registers do not possess *request* input (K_i) and *acknowledge* (K_o) output.

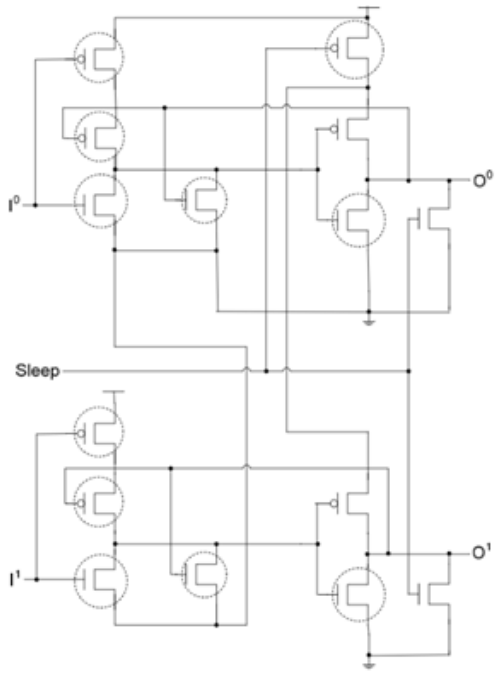


Figure 13. Slept DI Register w/o *nSleep* [3].

2.3.3. SCL Completion Unit and Handshaking Scheme

SCL completion component is based on NCL completion unit, in combination with Early Completion [2] and an additional *Sleep* signal input. Each completion unit comprises of *TH12/TH24comp* and *THnn* SCL gates arranged in a tree structure, followed by a final inverted NCL TH22 gate (without sleep). This tree structure is similar to the NCL completion unit structure, but different in that 1) all the gates (except the final inverting TH22 gate) can be driven to NULL by the pervious stage completion unit output, which is used as the *Sleep* signal for these gates, 2) The inputs to the tree structure are the register unit inputs in a stage, along with the output from subsequent stage completion component; whereas in NCL, inputs to the tree structure were only the K_o outputs from the registers in subsequent stage, as shown in Fig. 1. For example, as shown in Fig. 9, the dual-rail inputs to the stage i^{th} registers, R_i, are also input to the completion unit in stage *(i)*, C_i. These dual-rail signals along with the output from the subsequent stage completion unit, Ko_{i+1}, produces the output, Ko_i, of the completion component C_i. Ko_i is used as the *Sleep* input of the stage (*i)* Registration unit (R_i), Combinational Logic (C/L_i), and the stage (*i+1)* completion unit, C_{i+1}.

When the *Sleep* signal is asserted in sleep/idle mode, the registers and C/L block are all forced to a NULL value, which is equivalent to the NULL wavefront propagating through the circuits. The *Sleep* signal gets de-asserted when a new DATA value appears at the inputs. A 16 bit SCL completion unit structure is shown in Fig. 14. X_0 –X_{15} are the sixteen dual-rail inputs to a particular i^{th} stage register. Ko_{i+1} and Ko_{i-1} are the next and previous stage completion units' outputs, respectively. Ko_i is the output of the i^{th} stage completion unit.

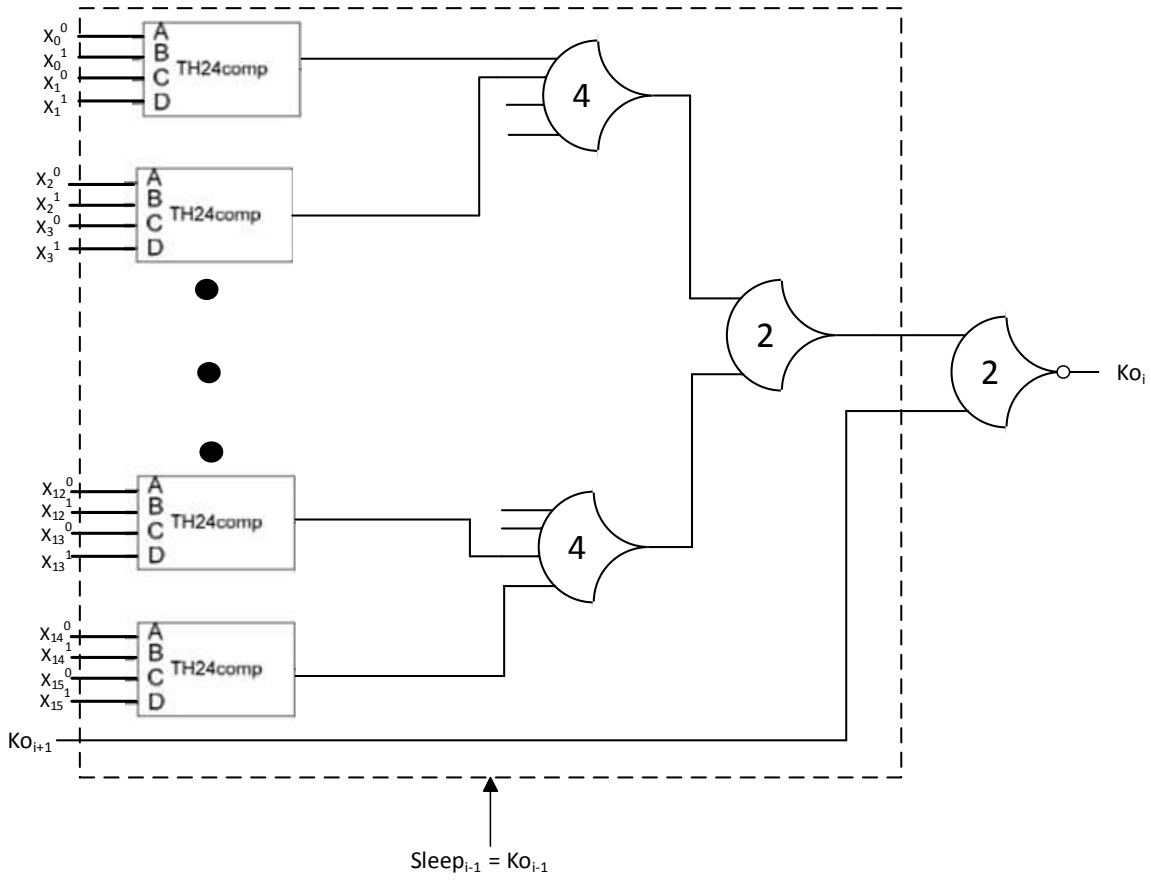


Figure 14. 16 bit SCL Early Completion Component.

3. FORMAL VERIFICATION METHOD FOR SCL CIRCUITS

Formal techniques are widely used in the industry to validate commercial designs before fabrication. Formal methods are based on proofs that can cover a large number of test cases that adds to its capability to detect corner case bugs. In industry, formal methods complement the traditional testing methods to guarantee complete functional correctness. At present, there are no formal verification schemes for SCL circuits. There exists few formal verification schemes for different QDI circuits, like, NCL and Pre-Charge Half Buffers (PCHB). However, those methods are not directly applicable to SCL circuits because of its unique structure. This chapter discusses some of the existing formal verification methods in asynchronous paradigms, their drawbacks, and the reasons behind not being directly applicable to QDI SCL circuits; followed by a detail illustration of the developed unified verification method for combinational and sequential SCL circuits.

3.1. Related Verification Work in QDI Paradigm

There have been some verification methodologies developed for asynchronous bounded delay model circuits based on trace theory, Signal Transition Graphs [30] etc. C. J. Meyers also developed a gate-level verification method for bounded delay models based on timed petri-nets [31]. However, bounded delay models consider the delays in both datapath and control path to ensure correctness of operation, hence are structurally very different from QDI paradigms. So these verification methodologies are not directly applicable to QDI circuits. There have been some formal verification schemes for QDI paradigms like NCL and PCHB paradigms, but none for QDI SCL circuits which can guarantee both safety and liveness of the circuits. [32] proposes an idea to verify NCL circuits based on the theory of WEB-Refinement [33]; where the specification and implementation are modeled as Transition Systems (TSs). However, due to the

extremely non-deterministic behavior of NCL circuits, the TSs become very complex with huge state-space. This results in a state space explosion, and infeasible verification time. [34] illustrates a model checking based approach for QDI PCHB circuits that also models the PCHB circuit as TSs; but this also suffers from state space explosion. Scalability is the major limiting factor for both the aforementioned methods. QDI SCL circuits are also non-deterministic, like NCL and PCHB; hence, we circumvent the idea of modelling the actual SCL circuit as TSs. A deadlock verification scheme for DI circuits, based on Click Library [35], is proposed in [36]. However, this method is not directly applicable, as SCL circuits are structurally very different from the circuits based on those primitive libraries. Also, it verifies only the liveness of the circuits and not the safety of the circuits. There exists several Design-For- Testability (DFT) based verification techniques for NCL [37] [38] and SCL circuits [9]. However, as discussed earlier, only testing is not sufficient to ensure complete functional correctness. Therefore, an alternate approach has been developed in this thesis that is scalable and tackles most of the limiting factors encountered in other methods for other QDI paradigms. It is the first ever formal verification method that is applicable to QDI SCL circuits.

3.2. Equivalence Verification Methodology for Sleep Convention Logic Circuits

In industry, QDI circuits are synthesized from their corresponding synchronous specifications. The specification goes through a series of transformation, which results in the synthesized circuit being structurally very different from the specification. For such scenario, equivalence checking is a widely used formal technique that checks for functional and logical equivalence between two structurally different systems.

The proposed method requires two steps. The first step ensures the safety, i.e. the functional correctness of the circuit. The high level idea behind the *safety* check is to convert the

SCL combinational/sequential circuit into an equivalent Boolean/synchronous circuit with the help of a conversion algorithm. The reduced circuit is then checked for equivalence with the actual Boolean/synchronous specification.

The second step checks for the handshaking connections between components that ensures the *liveness* (absence of deadlock) of the circuit. The *safety* and *liveness* check for combinational and sequential SCL circuits is described in detail in subsections 3.2.1 and 3.2.2, respectively. Section 3.3 tabulates our results in terms of verification times and capability to detect faults when our method was applied to various increasing order combinational and sequential SCL circuits.

3.2.1. Equivalence Verification Method for Combinational SCL Circuits

Fig. 15 shows an SCL 2x2 multiplier unit, implemented using SECRII w/o *nsleep* architecture explained in chapter 2. The circuit performs multiplication of two 2 bit dual-rail numbers, *xi (1:0)* and *yi (1:0)*, where (*xi1*, *xi0*) and (*yi1*, *yi0*) are the two bits of *xi* and *yi*, respectively; and outputs a 4 bit dual-rail output, *p (3:0),* where *p3*, *p2*, *p1*, and *p0* are the four bits of *p*. The combinational unit comprises of SCL AND gates and SCL Half Adder (HA) components. Internally these components are implemented using SCL threshold gates. The threshold gate level structure of SCL AND gate and SCL HA unit is shown in Figs. 16 and 17, respectively. Registers in STAGE 1 and 3 are the input and output registers, respectively; and the STAGE 2 register is an intermediate pipelining register used to increase throughput. All the registers in a combinational SCL circuit are initialized to NULL (i.e. reset-to-NULL) at the beginning of operation. *Comp1*, *Comp2*, and *Comp3* are the completion units that generate the *sleep* signals for their respective stages' registers, combinational logic, and next stage completion unit, in compliance with SECRII SCL framework. K_i is the external *request* input, and $Ko = k_{o1}$ is

the external *acknowledge* output. *SLP* is an external *sleep* input that sleeps the STAGE 1 completion unit.

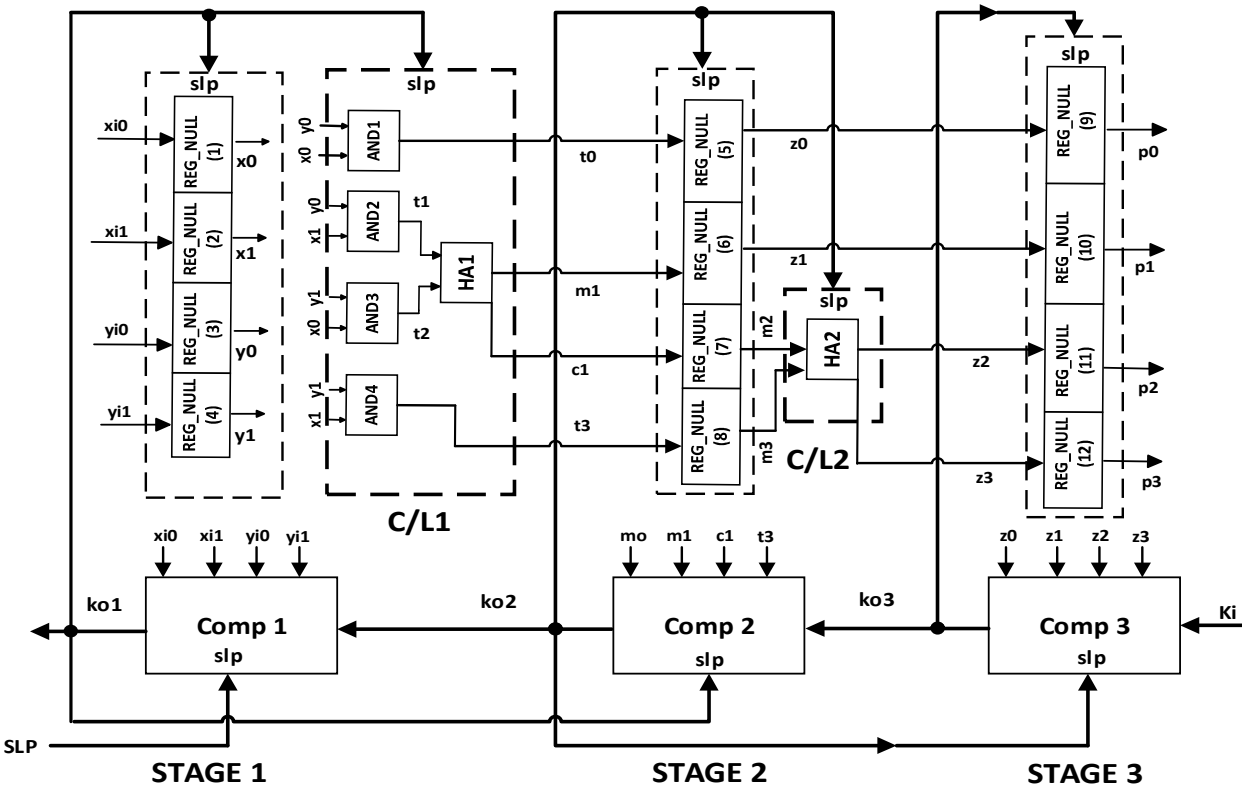


Figure 15. SCL 2x2 Multiplier.

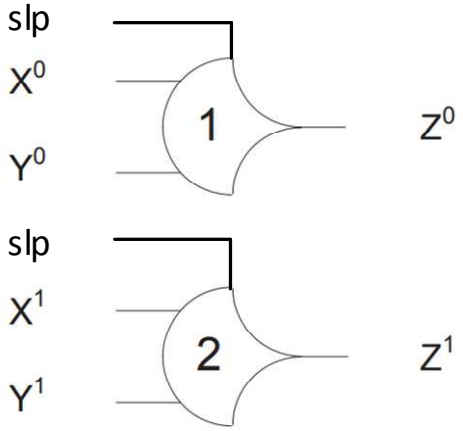


Figure 16. SCL AND2 Structure.

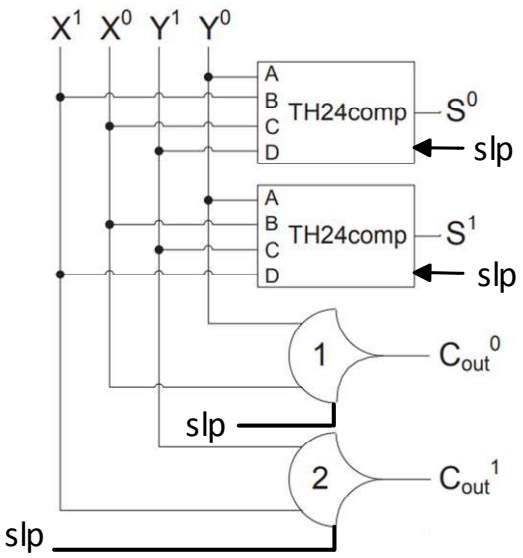


Figure 17. SCL Half-Adder (HA) Structure.
Adapted from [2].

3.2.1.1. Safety check for combinational SCL circuits

The netlist of the SCL 2×2 multiplier is shown in Fig. 18. The line numbers are used for ease of reference only and do not appear in the actual netlist. The first two lines denote the set of primary inputs and primary outputs, respectively. For any dual-rail signal, *a*, *a_0* and *a_1* denote a^0 and a^1, respectively. Lines 3-18 denote the gate structure of the C/L, where the first column indicates the *type* of SCL gate, the second column refers to the *level* of the gate (i.e., the longest path to the gate from any primary input, not counting registers), and the third, fourth, and fifth columns correspond to the gate's *data inputs*, *sleep input*, and *output*, respectively. Lines 19-30 are the 1-bit dual-rail SCL registers, where the first column corresponds to the reset type i.e. Reg_NULL for reset-to-NULL registers, columns 2 and 3 are the data $input^0$ and $input^1$ rails, respectively, column 4 is the *sleep* input, and columns 5 and 6 are the data $output^0$ and $output^1$ rails, respectively. Lines 31-33 are the three completion units, where *Comp* in the first column denotes an early completion component, the second column indicates the K_i input that comes

from the following stage completion, the third column includes all of the completion unit's *data inputs*, the fourth column is the sleep input, and the last column is the *output*.

```
1.  xi0_0,xi1_0,yi0_0,yi1_0,xi0_1,xi1_1,yi0_1,yi1_1
2.  p0_0,p1_0,p2_0,p3_0,p0_1,p1_1,p2_1,p3_1
3.  th22  1   x0_1,y0_1   ko1  t0_1
4.  th12  1   x0_0,y0_0   ko1  t0_0
5.  th22  1   x1_1,y0_1   ko1  t1_1
6.  th12  1   x1_0,y0_0   ko1  t1_0
7.  th22  1   x0_1,y1_1   ko1  t2_1
8.  th12  1   x0_0,y1_0   ko1  t2_0
9.  th22  1   x1_1,y1_1   ko1  t3_1
10. th12 1   x1_0,y1_0   ko1  t3_0
11. th24comp  2   t1_0,t2_0,t1_1,t2_1   ko1   m1_1
12. th24comp  2   t1_0,t2_1,t2_0,t1_1   ko1   m1_0
13. th22  2    t1_1,t2_1   ko1  c1_1
14. th12  2    t1_0,t2_0   ko1  c1_0
15. th24comp  3   m2_0,m3_0,m2_1,m3_1   ko2   z2_1
16. th24comp  3   m2_0,m3_1,m3_0,m2_1   ko2   z2_0
17. th22  3   m2_1,m3_1   ko2    z3_1
18. th12  3   m2_0,m3_0   ko2    z3_0
19.  Reg_NULL  xi0_0 xi0_1   ko1   x0_0 x0_1
20.  Reg_NULL  xi1_0 xi1_1   ko1   x1_0 x1_1
21.  Reg_NULL  yi0_0 yi0_1   ko1   y0_0 y0_1
22.  Reg_NULL  yi1_0 yi1_1   ko1   y1_0 y1_1
23.  Reg_NULL  t0_0 t0_1   ko2   z0_0 z0_1
24.  Reg_NULL  m1_0 m1_1   ko2   z1_0 z1_1
25.  Reg_NULL  c1_0 c1_1   ko2   m2_0 m2_1
26.  Reg_NULL  t3_0 t3_1    ko2   m3_0 m3_1
27.  Reg_NULL  z0_0 z0_1   ko3   p0_0 p0_1
28.  Reg_NULL  z1_0 z1_1   ko3   p1_0 p1_1
29.  Reg_NULL  z2_0 z2_1   ko3   p2_0 p2_1
30.  Reg_NULL  z3_0  z3_1   ko3    p3_0  p3_1
31.  Comp   ko2   xi0_0,xi0_1,xi1_0,xi1_1,yi0_0,yi0_1,yi1_0,yi1_1   SLP   ko1
32.  Comp   ko3   t0_0,t0_1,m1_0,m1_1,c1_0,c1_1,t3_0,t3_1   ko1   ko2
33.  Comp   Ki    z0_0,z0_1,z1_0,z1_1,z2_0,z2_1,z3_0,z3_1    ko2   ko3
```

Figure 18. 2x2 SCL Multiplier Netlist.

Note that this SCL netlist is generated by processing the original gate level SCL netlist to order the components and to combine all of the completion unit gates into the single completion component as shown in the netlist (Fig. 18). Each completion component, such as *Comp2* in Fig. 15, is comprised of *TH12/TH24comp* gates and *THnn SCL* gates arranged in a tree structure, followed by a final inverted NCL *TH22* gate (without sleep), as shown in Fig. 14. The *Comp* units in Fig. 18 are abstracted from the gate level completion unit structures as shown in Fig. 19.

```
th12   1   xi0_0,xi0_1  SLP   r0_1
th12   1  xi1_0,xi1_1   SLP   r1_1
th12   1  yi0_0,yi0_1   SLP   r2_1
th12   1  yi1_0,yi1_1   SLP   r3_1
th44   2  r0_1,r1_1,r2_1,r3_1   SLP   r4_1
nclth22  3   ko2,r4_1  ko1
th12  2   t0_0,t0_1  ko1  r5_1
th12  3   m1_0,m1_1   ko1   r6_1
th12  6  c1_0,c1_1   ko1   r7_1
th12  9  t3_0,t3_1   ko1   r8_1
th44  10  r5_1,r6_1,r7_1,r8_1  ko1   r9_1
nclth22  11  ko3,r9_1   ko2
th12  2  z0_0,z0_1   ko2    r10_1
th12  3  z1_0,z1_1   ko2    r11_1
th12  6  z2_0,z2_1   ko2    r12_1
th12  9  z3_0,z3_1   ko2   r13_1
th44 10 r10_1,r11_1,r12_1,r13_1 ko2  r14_ 1
nclth22  11 Ki,r14_1  ko3
```

Figure 19. Netlist of Gates Comprising *Comp* Units before Abstraction.

nclTH22 gate in Fig. 19 corresponds to the last inverting NCL threshold gate in each completion unit. Its representation in the netlist is similar to that of SCL gates. The first column corresponds to the *type* of gate i.e. always *nclTH22* (for completion components), second and third columns represent the *level* and *inputs* of the gate, respectively; followed by the *output* in

the last column. Unlike SCL gate structure in the netlist, the *nclTH22* gate has no *sleep* column since it is an NCL gate. When processing the original SCL netlist to obtain the abstracted completion component shown in the netlist Fig. 18, our developed tool ensures that the completion structures are internally correct i.e., all data inputs to a completion unit must go to *TH12/TH24comp* gates, and their outputs form a tree of SCL *THnn* type gates, whose output, along with the K_i input, is input to an inverted *nclTH22* gate that produces the K_o output, and that all SCL gates have the same *sleep* input. The safety check method first reduces the SCL netlist, as shown in Fig. 18, into an equivalent Boolean netlist, as shown in Fig. 20, which correlates to the equivalent Boolean 2x2 multiplier block diagram shown in Fig. 21. Each SCL C/L gate is replaced with its corresponding Boolean function, omitting the *sleep* input. Each rail of a dual-rail signal is treated as a distinct Boolean signal, which requires the addition of an inverter for each primary circuit input, to generate its complement to replace each input's $rail^0$, used in the C/L, as shown in lines 3-6.

Similar to the SCL netlist structure, the first two lines in the converted netlist correspond to the set of primary inputs and primary outputs, respectively. Each subsequent line corresponds to a C/L gate, where the first column denotes the *type* of gate, the second column denotes the gate's *level*, the third column denotes the gate's *inputs*, and the fourth column denotes the gate's *output*. Note that the C/L *sleep* input, SCL registers, and completion units are removed, since these are not utilized in the Boolean circuit; and their connections will be verified as part of the liveness check, explained in the following sub-section.

```
1.  xi0_1,xi1_1,yi0_1,yi1_1
2.  p0_0,p0_1,p1_0,p1_1,p2_0,p2_1,p3_0,p3_1
3.  not   1   xi0_1   xi0_0
4.  not   1   yi0_1   yi0_0
5.  not   1   xi1_1   xi1_0
6.  not   1   yi1_1   yi1_0
7.  th12   2   xi0_0,yi0_0   p0_0
8.  th22   1   xi0_1,yi0_1   p0_1
9.  th12   2   xi1_0,yi0_0   t1_0
10. th22   1   xi1_1,yi0_1   t1_1
11. th12   2   xi0_0,yi1_0   t2_0
12. th22   1   xi0_1,yi1_1   t2_1
13. th12   2   xi1_0,yi1_0   t3_0
14 th22    1   xi1_1,yi1_1    t3_1
15. th24comp   3   t2_0,t1_1,t1_0,t2_1   p1_0
16. th24comp   3   t2_0,t1_0,t2_1,t1_1   p1_1
17. th12   3   t2_0,t1_0   c1_0
18. th22   2   t1_1,t2_1   c1_1
19. th24comp   4   c1_0,t3_1,t3_0,c1_1   p2_0
20. th24comp   4   c1_0,t3_0,c1_1,t3_1   p2_1
21. th12   4   c1_0,t3_0   p3_0
22. th22   3   t3_1,c1_1   p3_1
```

Figure 20. Converted Equivalent Boolean Netlist.

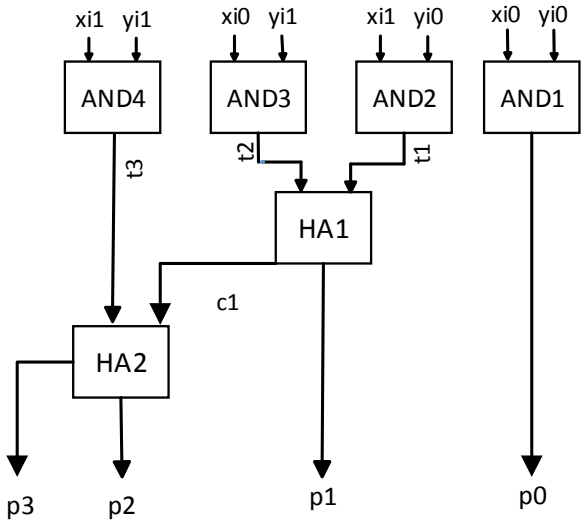


Figure 21. Equivalent Boolean 2x2 Multiplier Circuit.

The Boolean netlist obtained from the SCL netlist is then checked against the specification Boolean function. Z3 [39] SMT solver was used to perform this check. Our tool auto converts this Boolean equivalent netlist into SMT Lib language which is then fed into Z3. For the example 2×2 multiplier with two 2-bit dual-rail inputs, *x(1:0)* and *y(1:0)*, the SMT solver checks the following property: $F_{SCL\_Bool\_Equv}$ *(x0_1, x1_1, y0_1, y1_1)* → *MUL (x(1:0), y(1:0))*, where $F_{SCL\_Bool\_Equv}$ is the function corresponding to the Boolean circuit obtained from converting the SCL circuit to be verified and *MUL* corresponds to the Boolean specification of the multiplier circuit. *(x1, x0)* and *(y1, y0)* are the (MSB, LSB) of *x* and *y*, respectively. It also checks that the *rail0* and *rail1* outputs in the converted netlist are complements of each other: i.e., $Rail^1(P) \rightarrow \neg\, Rail^0(P)$.

3.2.1.2. Handshaking check for combinational SCL circuits

The 2×2 SCL multiplier in Fig. 15 implements the SECRII w/o *nsleep* architecture, where the registers, completion units, and C/L are all slept during the NULL cycle. In this architecture, the output of a completion unit in a particular stage is responsible to *sleep* the registers, combinational logic of that stage as well as the next-stage completion unit. The proper connection of handshaking signals between the various units in the framework ensures the correct functioning of the SCL circuit without any deadlock. In Fig. 15, the output of the STAGE 1 completion unit, *ko1*, is the *sleep* input of the STAGE 1 registers (1-4), the STAGE 1 C/L, *C/L1*, and the STAGE 2 completion unit, *Comp2*. An algorithm was developed that takes an SCL netlist, like the one shown in Fig. 18, and converts it into a graph structure, where each register, threshold gate, and completion unit are modeled as nodes. The directed edges going into and out from a node correspond to the inputs and outputs of that particular node, respectively. The algorithm traverses the graph to gather the needed information in order to verify that the

handshaking exactly follows the SCL protocol. For registers, completion units, and threshold gates, the following information is stored: data inputs, *sleep* input, and data output(s). After gathering this information, the following handshaking checks are performed:

- **Register *sleep* and Completion data inputs:**

Each stage register's data inputs must be exactly the same as for the stage's completion unit; and the completion unit output must be the register's *sleep* input. As an example, the inputs of registers (1-4) in STAGE1 are also the data inputs to completion unit, Comp1. Hence, the output of Comp1, *ko1*, is the *sleep* input for registers (1-4).

- **Sleep for C/L:**

Each completion unit sleeps its stage's register and C/L, such that every SCL C/L gate's *sleep* input should be the same as its preceding register's *sleep* input. Hence, for each gate, *i*, a *gate_fanin (i)* list is created, that traces back all inputs of $gate_i$ to their originating registers. For example, the TH12 gate on line 14 of Fig. 18, corresponds to the TH12 gate that generates the $c1^0$ carry output of the HA1 in C/L1 in Fig 15. Tracing this gate's inputs back to their generating registers yields x1_0, y0_0, x0_0, y1_0, resulting in a *gate_fanin* list of Reg1, Reg2, Reg3, Reg4, which all have the same *sleep* input as the TH12 gate. Once the *gate_fanin* list for all gates are computed, all registers in each *gate_fanin(i)* list are inspected to ensure that they all have the same *sleep* input, and that this sleep input is also the *sleep* input for $gate_i$. If the *gate_fanin* list contains registers from multiple stages (i.e., different *sleep* inputs), or if the gate's *sleep* input differs from its corresponding input register's *sleep* input, then an error message is generated.

- **Completion output, and *slp* and *Ki* inputs:**

$Comp_i$'s *Ki* input must be the output of $Comp_{i+1}$, and its *sleep* input must be the output of $Comp_{i-1}$. In Fig. 15, $Comp_2$'s *Ki* input is the output of $Comp_3$, and its *sleep* input is the output of $Comp_1$. The first and last stages are slightly different. $Comp_1$'s (i.e., the completion unit whose

data inputs are the circuit's external data inputs) *sleep* input must be the external *SLP* input, and its output must be the external *Ko* output; the last stage completion's (i.e., the completion unit associated with the register that produces the external data outputs) *Ki* input must be the external *Ki* input.

3.2.2. Equivalence Verification Method for Sequential SCL Circuits

Verification of sequential SCL circuits are much more complex because of datapath feedback, which requires at least *2N+1* SCL registers in a feedback loop with *N* DATA tokens in order to avoid deadlock [40]. Hence, common practice when synthesizing an SCL circuit from its synchronous specification is to replace every synchronous register with three SCL registers, reset to NULL, DATA, NULL. A simple 4+2x2 SCL MAC (i.e. a 2-bit wide multiplier with 4-bit wide accumulator) is taken as example to elaborate this further, as well as to explain our verification method for sequential circuits. For the 4+2×2 MAC, the output register is replaced with registers numbered 5-8 and 15-22, as shown in the Fig. 22 SCL implementation of this circuit. In addition to the fed back accumulator output (*acci(3:0)*), the STAGE 1 registers (registers 1-8) also include the external data inputs, *xi (1:0)* and *yi (1:0)*, while the STAGE 2 register is an additional reset-to-NULL register included to increase performance. Note that STAGE 1 and 2 could be combined into a single stage, or STAGE 3 could be removed without causing deadlock. The C/L is comprised of SCL 2-input AND functions (AND2), Half Adders (HAs) and Full Adders (FAs), similar to the ones shown in Figs. 17 and 18, in the previous sub-section. The FA SCL internal threshold gate structure is shown in Fig. 23, comprising of SCL *TH23* and *TH34w2* gates.

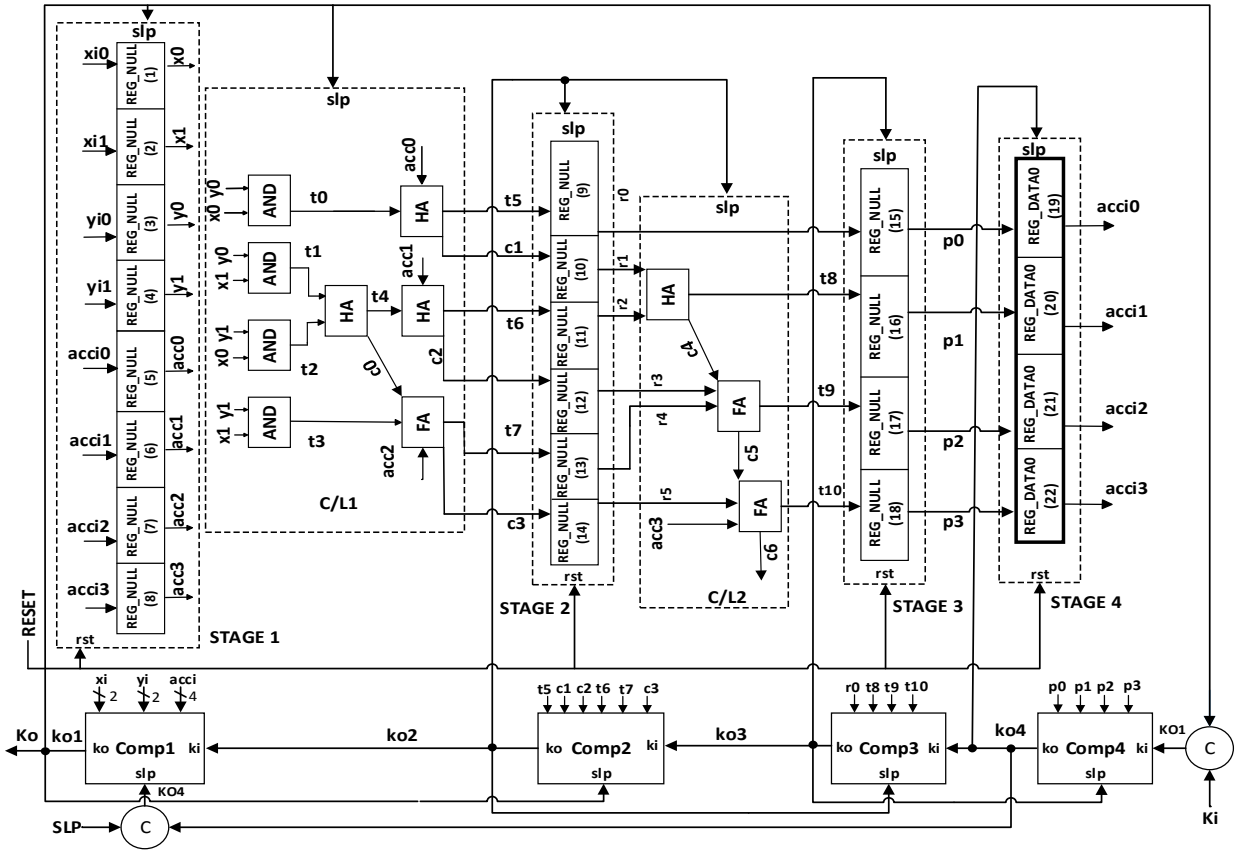


Figure 22. 4+2×2 SCL MAC.

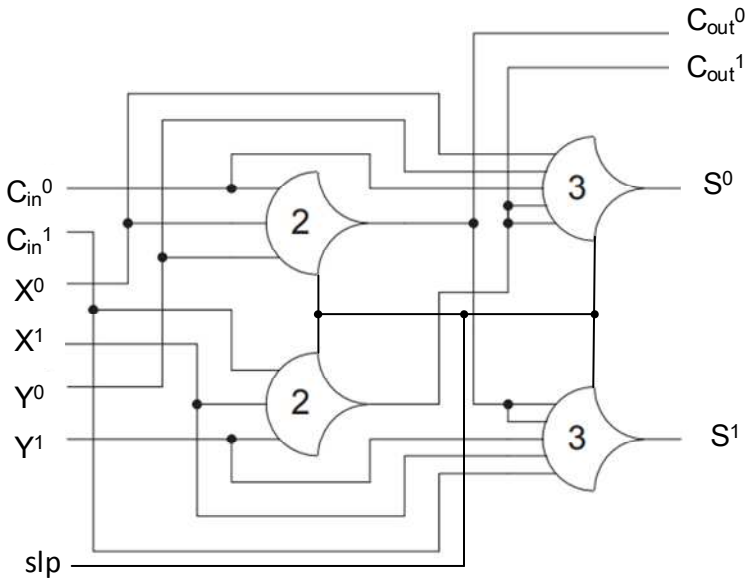


Figure 23. SCL Full Adder Block (FA).

3.2.2.1. Safety check for sequential SCL circuits

The netlist structure for a sequential SCL circuit is similar to that for a combinational circuit, shown in Fig. 24. Here also the *Comp* units are obtained through abstraction form an original netlist of SCL threshold gates and *nclTH22* gates. The only differences are the inclusion of C-elements in the feedback loop handshaking, and reset-to-DATA registers, REG_DATA0 in this case. The algorithm to convert the SCL netlist into its equivalent synchronous netlist is also similar to the combinational SCL conversion described in Section 3.2.1.1, with the following additions: each reset-to-DATA register is converted into an equivalent 2-bit Boolean register, one bit for the SCL register's $rail^1$ output and the other for its $rail^0$ output. Same as for the combinational circuit conversion, the reset-to-NULL registers and all *sleep* signals are omitted from the converted synchronous netlist, as these will be verified in the subsequent handshaking check. For sequential circuits, the additional C-elements are also omitted from the converted synchronous netlist and will also be verified in the handshaking check.

The converted Boolean netlist and the corresponding equivalent Boolean circuit for the 4+2x2 MAC are shown in Figs. 25 and 26 respectively. The equivalence check of sequential is not as straightforward as that of the combinational SCL circuits. Sequential circuits have states and transitions. The theory of WEB-Refinement [33] is utilized to check for equivalence between the converted synchronous netlist and the original synchronous specification.

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

Figure 24. 4+2x2 MAC SCL Netlist.

```
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, acci2_0, acci2_1, acci3_0, acci3_1
3. not   1   xi0_1   xi0_0
4. not   1   yi0_1   yi0_0
5. not   1   xi1_1   xi1_0
6. not   1   yi1_1   yi1_0
7. th12   2   xi0_0,yi0_0   t0_0
8. th22   1   xi0_1,yi0_1   t0_1
9. th12   2   xi1_0,yi0_0   t1_0
10. th22   1   xi1_1,yi0_1   t1_1
11. th12   2   xi0_0,yi1_0   t2_0
12. th22   1   xi0_1,yi1_1   t2_1
13. th12   2   xi1_0,yi1_0   t3_0
14. th22   1   xi1_1,yi1_1   t3_1
15. th24comp   3   t2_0,t1_1,t1_0,t2_1   t4_0
16. th24comp   3   t2_0,t1_0,t2_1,t1_1   t4_1
17. th12   3   t2_0,t1_0   c0_0
18. th22   2   t1_1,t2_1   c0_1
19. th24comp   3   acci0_0,t0_1,t0_0,acci0_1   p0_0
20. th24comp   3   acci0_0,t0_0,acci0_1,t0_1   p0_1
21. th12   3   acci0_0,t0_0   c1_0
22. th22   2   t0_1,acci0_1   c1_1
23. th24comp   4   acci1_0,t4_1,t4_0,acci1_1   t6_0
24. th24comp   4   acci1_0,t4_0,acci1_1,t4_1   t6_1
25. th12   4   acci1_0,t4_0   c2_0
26. th22   4   t4_1,acci1_1   c2_1
27. th23   4   t3_0,acci2_0,c0_0   c3_0
28. th23   3   t3_1,acci2_1,c0_1   c3_1
29. th34w2   4   c3_1,t3_0,acci2_0,c0_0   t7_0
30. th34w2   5   c3_0,t3_1,acci2_1,c0_1   t7_1
31. th24comp   5   c1_0,t6_1,t6_0,c1_1   p1_0
32. th24comp   5   c1_0,t6_0,c1_1,t6_1   p1_1
33. th12   5   c1_0,t6_0   c4_0
34. th22   5   t6_1,c1_1   c4_1
35. th23   6   t7_0,c2_0,c4_0   c5_0
36. th23   6   t7_1,c2_1,c4_1   c5_1
37. th34w2   7   c5_1,t7_0,c2_0,c4_0   p2_0
38. th34w2   7   c5_0,t7_1,c2_1,c4_1   p2_1
39. th23   7   acci3_0,c3_0,c5_0   c6_0
40. th23   7   acci3_1,c3_1,c5_1   c6_1
41. th34w2   8   c6_1,acci3_0,c3_0,c5_0   p3_0
42. th34w2   8   c6_0,acci3_1,c3_1,c5_1   p3_1
43. Reg_DATA   p0_0   p0_1   acci0_0   acci0_1
44. Reg_DATA   p1_0   p1_1   acci1_0   acci1_1
45. Reg_DATA   p2_0   p2_1   acci2_0   acci2_1
46. Reg_DATA   p3_0   p3_1   acci3_0   acci3_1
```

Figure 25. Converted Boolean 4+2x2 MAC SCL Netlist.

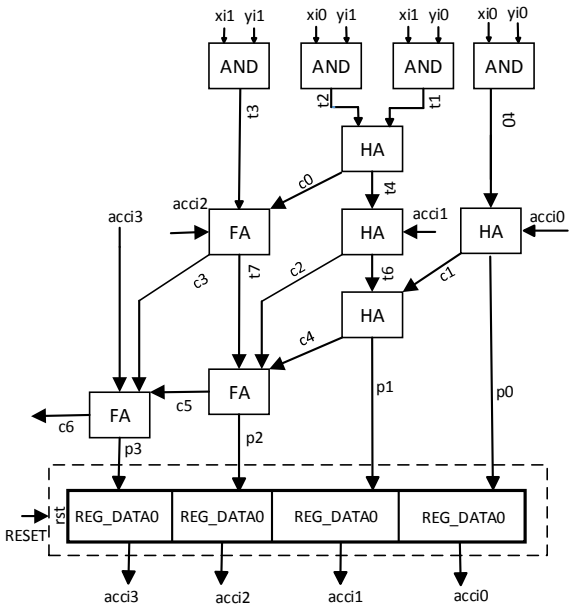


Figure 26. Converted Equivalent 4+2x2 MAC Boolean Circuit.

WEB-refinement uses two functions: *rank* and *refinement-map*. *Rank* functions differentiate between finite transitions and infinite stuttering (deadlock). The implementation Transition System (TS) may look very different from specification TS, which is tackled by the *refinement-map* functions. These functions map the specification states with implementation states. However, in our case, $rail^1$ registers have a one-to-one mapping with the synchronous register; hence, there is no stutter. It is assumed that the I\O mapping and the register mapping between the specification and implementation circuits are provided, resulting in a reduced proof obligation given below and demonstrated using Fig. 27.

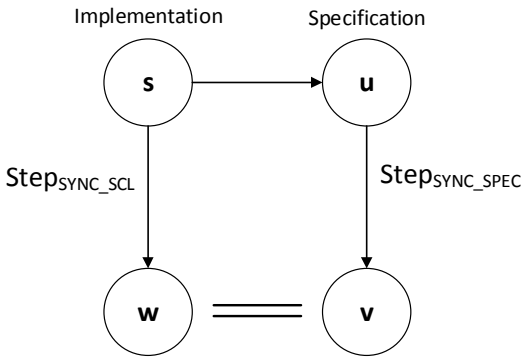


Figure 27. Proof Obligation to Check Equivalence of Sequential SCL Circuits.

In Fig. 27, *s* is an implementation state, i.e. a state in the reduced synchronous SCL circuit. *u* is a state in specification that is a projection of the values of the $rail^l$ registers from state *s*. $Step_{SYNC\_SCL}$ and $Step_{SYNC\_SPC}$ are the single step functions of the SCL implementation and specification, respectively. The proof obligation states that if *u* is obtained by projecting the $rail^l$ values of *s*, *w* is the next state of the implementation state *s*, and *v* is the next state of *u*, then the corresponding projection of values from the $rail^l$ registers of the *w* state must be equivalent to the values of the corresponding registers in the *v* state.

Proof Obligation:

$$PO :\{\forall s:: s \in S_{SYNC\_SCL}:: [u = Reg\_Proj\ (s) \wedge w = Step_{SYNC\_SCL}\ (s) \wedge v = Step_{SYNC\_SPEC}\ (u)]$$

$$\Rightarrow Reg\_Proj\ (w) = v\}.$$

The converted netlist, synchronous specification, and equivalence check proof obligation are modeled in SMT-LIB, and the properties checked using the Z3 SMT solver. For the 4+2×2 MAC, the following properties are checked: Considering symbolic state transitions for any current state values of 2-bit primary inputs, *x_cs(1:0)* and *y_cs(1:0)*, and 4-bit accumulator values *acc_cs(3:0)*, the next state of the converted synchronous netlist obtained from the SCL implementation should be equivalent to the next state of the synchronous specification; i.e.,

$$F_{\text{SCL\_Sync\_Eq}}\ (x0\_cs, x1\_cs, y0\_cs, y1\_cs, acc0\_cs, acc1\_cs, acc2\_cs, acc4\_cs) \rightarrow F_{Sync\_Spec}$$

(x_cs(1:0), y_cs(1:0), acc_cs(3:0)). For each register in the converted synchronous netlist, it is also checked that its output $rail^0$ and $rail^1$ are complements of each other.

3.2.2.2. Handshaking check for sequential SCL circuits

Handshaking verification is the same as for combinational SCL circuits, except that the first and last registers of a feedback loop include an extra C-element, as shown in Fig. 22, which requires the following two additional checks: The *Ki* input for a feedback loop's output register's completion (e.g., Comp4 in Fig. 22) must be the combination (via a C-element) of its downstream register's completion's *Ko* (external *Ki* input in Fig. 22) and its feedback loop input register's completion's *Ko* (*ko1* in Fig. 22), instead of only its downstream register's completion's *Ko*, as in combinational SCL circuits. The *sleep* input (*slp*) for a feedback loop's input register's completion (e.g., Comp1 in Fig. 22) must be the combination (via a C-element) of its upstream register's completion's *Ko* (external *SLP* input in Fig. 22) and its feedback loop output register's completion's *Ko* (*ko3* in Fig. 22), instead of only its upstream register's completion's *Ko*, as in combinational SCL circuits. The C-element are represented as shown in line 61 and 62 in Fig. 24, where the first column represents an n-input C element (*Cn*), followed by the inputs in ',' separated format and output of the C element in second and third columns, respectively.

3.2.3. Results

To demonstrate the verification of combinational SCL circuits, several multipliers and ISCAS benchmarks were verified; whereas, MACs and an ISCAS benchmark (s27) were verified to demonstrate the verification of sequential SCL circuits, as shown in Table 2. The algorithms described in Section 3.2.2 were implemented using Python. Z3 SMT solver [39] was used on an Intel® Core™ i7- 4790 CPU with 32GB of RAM running at 3.60 GHz.to check for functional

equivalence. *NxN MUL* stands for an *N*-bit combinational multiplier, whereas *2N+NxN MAC* stands for an *N*-bit multiplier unit with *2N*-bit accumulator. Additionally, a number of buggy circuits were tested, including circuits with erroneous handshaking signals, such as an incorrect sleep signal connection to a C/L gate (i.e., 20+10×10 MAC B1), a C/L gate with one data input being a signal's rail^0 instead of the correct rail^1 (i.e., 20+10×10 MAC B2) and the combinational logic having incorrect logic elements (20+10x10 MAC B3). For all buggy cases, the proposed approach was able to flag the errors, providing a descriptive message indicating the erroneous connection for handshaking errors, and producing counter examples to trace back the error path (via the SMT solver) for cases of functional in-equivalence. Since B2 was caught during the safety check, its verification time was much less than for B1, which was detected in the handshaking check. Time to convert an SCL netlist to its equivalent Boolean/synchronous netlist was negligible compared to safety and handshaking check times; therefore, was not included in Table 2. It is also to be observed from the table that while the safety check for higher order circuits took more time, this was only when the circuit was correct. Also, with increasing number of levels in a circuit the verification time increases. However, in case of any functional bugs in the circuit (like B2 and B3 in the table), the developed method was able to catch those bugs very fast (in less than a second for B2 and B3, as shown in the table).

Comparing our verification times with that of [34], we found that our method was able to verify up to 12x12 QDI SCL multiplier without timing out, whereas the model-checking based verification method for QDI PCHB circuits [34] timed out for 4x4 PCHB multiplier. Furthermore, comparing the verification times for 8x8 MAC NCL circuit using WEB Refinement [32] with that of 16 + 8x8 MAC SCL circuit using our verification methodology, we found that our method was ~1000 times faster. Even considering this difference between QDI

NCL, PCHB, and SCL circuits, it can be concluded that our equivalence verification method is a definite improvement in terms of scalability and speed.

Table 2. Verification Results for Various SCL Circuits.

SCL Circuit	**# of Gates**	**# of C/L levels**	**Verification Times (in sec.)**		
			Safety Check	***Handshaking Check***	***Total time***
6x6 MUL	260	22	0.32	0.0259	0.3459
8x8 MUL	440	30	10.62	0.055	10.675
10x10 MUL	670	38	683.49	0.1536	683.64
12x12 MUL	946	46	49,963.05	0.316	49,963
ISCAS c17 [41]	37	3	0.01	0.002	0.012
ISCAS c432 [41]	445	23	1.03	0.0468	1.0768
ISCAS s27 [42]	60	5	0.09	0.002	0.092
12+6x6 MAC	373	25	1.69	0.031	1.721
16+8x8 MAC	592	33	12.03	0.1007	12.131
20+10x10 MAC	858	41	1,581.72	0.213	1581.93
24+12x12 MAC	1173	50	1,40,780.13	0.4078	1,40,780.5
20+10x10 MAC B1	858	41	1483.22	0.3403	1483.5603
20+10x10 MAC B2	858	41	0.17	0.213	0.383
20+10x10 MAC B3	858	41	0.27	0.213	0.483

4. CONCLUSIONS

Sleep Convention Logic (SCL) is an emerging ultra-low power Quasi-Delay Insensitive (QDI) asynchronous design paradigm with enormous potential for industrial applications. Design validation is a critical concern before commercialization. Unlike other QDI paradigms, such as NULL Convention Logic (NCL) and Pre-Charge Half Buffers (PCHB), there exists no formal verification methods for SCL circuits. The goal of the research illustrated in this thesis was to develop a unified and scalable formal verification scheme for combinational as well as sequential SCL circuits, with the potential to meet commercial standards.

4.1. Summary

Power consumption and clock management are the two major design challenges faced by today's semiconductor industry in the synchronous domain. At nanoscale level, design factors that were previously less significant, such as wire-delays and leakage power, have become more crucial. Also, synchronous design gets more vulnerable to process variation (power, voltage, temperature) in deep submicron region. On the other hand, the Delay Insensitive (DI) paradigm of asynchronous domain is known for its robust architecture against process variation. They have low-power applications and requires no complex timing analysis, which resulted in an increasing popularity of this domain over the last few decades. NULL Convention Logic (NCL) is one such commercially successful DI paradigm. Multi-Threshold NULL Convention Logic (MTNCL), also known as Sleep Convention Logic (SCL), is a modification over the NCL architecture incorporating Multi-Threshold CMOS (MTCMOS) logic to further improve power performance. A detailed description of NCL and MTNCL architecture is provided in chapter 2.

Previous verification works related to different QDI paradigms and existing Design-for-Testability method for QDI SCL circuits were discussed in chapter 3; followed by an illustration

of the developed formal modeling and verification methodology SCL circuits. The fundamental idea behind the verification methodology was to perform structural reduction on the complex SCL implementation to convert it to an equivalent Boolean/synchronous circuit. The reduced circuit was checked for equivalence with the Boolean/synchronous specification. Procedures to verify the functionality, liveness, and handshaking connections based on the developed method were discussed in details. The method was demonstrated using several increasing order multipliers as well as Multiply and Accumulate (MAC) units, and ISCAS benchmarks.

4.2. Scopes for Future Work

The equivalence verification methodology demonstrated in this thesis is the first known formal verification work for SCL circuits. The method is applicable to both combinational and sequential SCL circuits. However, the method currently only works for sequential circuits without multiple interactive feedback loops, such as a MAC, since the handshaking is otherwise much more complicated, and requires the development of an algorithm to map each register in the converted synchronous circuit to its corresponding register in the original synchronous specification. These problems will be tackled in future work, such that the developed approach will be applicable to any arbitrary sequential circuit. Scalability of the approach can be improved further using abstraction techniques, and a commercial equivalence checker instead of an SMT solver.

REFERENCES

1. International Technology Roadmap for Semiconductors, 2013 Edition [Online], http://www.itrs2.net/2013-itrs.html, Accessed on: Feb. 2, 2018.
2. S. C. Smith and J. Di, *Designing Asynchronous Circuits using NULL Convention Logic (NCL)*, ser. Synthesis Lectures on Digital Circuits and Systems. Morgan & Claypool Publishers, 2009.
3. L. Zhou, R. Parameswaran, F. Parsan, S. 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, no. 2, May, pp. 81–100, 2015.
4. M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial hdl synthesis tools," in *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, ser. ASYNC '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 114 [Online]. Available: http://dl.acm.org/citation.cfm?id=785166.785308.
5. N. Andrikos, L. Lavagno, D. Pandini, and C. P. Sotiriou, "A fully-automated desynchronization flow for synchronous circuits," in *44th ACM/IEEE Design Automation Conference*, June 2007, pp. 982–985.
6. K. S. Stevens, Y. Xu, and V. Vij, "Characterization of asynchronous templates for integration into clocked cad flows," in *15th IEEE Symposium on Asynchronous Circuits and Systems (ASYNC),* May 2009, pp. 151–161.
7. E. Kilada and K. S. Stevens, "Control network generator for latency insensitive designs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10.

3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 1773–1778. [Online]. Available: http://dl.acm.org/citation.cfm?id=1870926.1871354.

8. R. B. Reese, S. C. Smith, and M. A. Thornton, "Uncle - an rtl approach to asynchronous design," in *ASYNC*, J. Sparsø, M. Singh, and P. Vivet, Eds. IEEE Computer Society, 2012, pp. 65–72.
9. F. Parsan, S. C. Smith, W. K. Al-Assadi, "Design for Testability of Sleep Convention Logic", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 24, pp. 743-753, 2015.
10. S. H. Unger, "Asynchronous Sequential Switching Circuits", Wiley, New York, 1969.
11. S. M. Nowick and D. L. Dill, "Synthesis of Asynchronous State Machines Using a Local Clock", Proceedings of ICCAD, pp.192-197, 1991.
12. Ivan E. Sutherland, "Micropipelines", Communications of the ACM, Vol. 32, No. 6, pp. 720-738, 1989.
13. A. Martin, "The Limitations to Delay-lnsensitivity in Asynchronous Circuits", Advanced Research in VLSI: Proceedings of the Sixth MIT Conference: pp. 263-278, 1990.
14. A. J. Martin, "Asynchronous Datapaths and the Design of an Asynchronous Adder", Formal Methods in System Design, Vol. 1, No. 1, pp. 117-137, 1992.
15. C. L. Seitz, "System Timing", in Introduction to VLSI Systems, Addison-Wesley, pp. 218-262, 1980.
16. N. P. Singh, A Design Methodology for Self-Timed Systems, Master's Thesis, MIT/LCS/TR-258, Laboratory for Computer Science, MIT, 1981.
17. T. S. Anantharaman, "A Delay Insensitive Regular Expression Recognizer", IEEE VLSI Technology Bulletin, Sept. 1986.

18. Ilana David, Ran Ginosar, and Michael Yoeli, “An Efficient Implementation of Boolean Functions as Self-Timed Circuits”, IEEE Transactions on Computers, Vol. 41, No. 1, pp. 2-10,1992.
19. J. Sparso, J. Staunstrup, M. Dantzer-Sorensen, Design of Delay Insensitive Circuits using Multi-Ring Structures. Proceedings of the European Design Automation Conference, pp. 15-20, 1992.
20. D. E. Muller, “Asynchronous Logics and Application to Information Processing”, in Switching Theory in Space Technology, Stanford University Press, pp. 289-297, 1963.
21. Scott C. Smith, Ronald F. DeMara, Jiann S. Yuan, D. Ferguson, and D. Lamb. "Optimization of NULL convention self-timed circuits." INTEGRATION, the VLSI journal 37, no. 3 (2004): 135-165.
22. S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguson, "Speedup of Delay-Insensitive Digital Systems Using NULL Cycle Reduction," in Proceedings of the 2001 International Workshop on Logic and Synthesis (IWLS-01), Granlibakken, California, U.S.A., pp. 185-189, June 12-15, 2001.
23. N. Weng, J. S. Yuan, R. F. DeMara, D. Ferguson, and M. Hagedorn, "Glitch Power Reduction for Low Power IC Design," in Proceedings of the Ninth Annual NASA Symposium on VLSI Design, pp. 7.5.1-7.5.7, Albuquerque, New Mexico, U.S.A., November 8-9, 2000.
24. Scott C. Smith, Ronald F. DeMara, Jiann S. Yuan, M. Hagedorn, and D. Ferguson. "Delay-insensitive gate-level pipelining." Integration, the VLSI journal 30, no. 2 (2001): 103-131.
25. W. Kuang, J. S. Yuan, R. F. DeMara, M. Hagedorn, and K. Fant, “Performance Analysis and Optimization of NCL Self-timed Rings,” IEE Proceedings on Circuits, Devices, and Systems,

Vol. 150, No. 3, June, 2003, pp. 167–172. ISSN:1350-2409 Inspec Accession Number: 7665699.

26. P. Prakash, A. J. Martin, Slack matching quasi-delay-insensitive circuits", in 12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 2006, pp. 10-204.
27. Gerald E. Sobelman and Karl M. Fant, "CMOS Circuit Design of Threshold Gates with Hysteresis", IEEE International Symposium on Circuits and Systems (II), pp. 61-65, 1998.
28. K. M. Fant and S. A. Brandt, "NULL convention logic: a complete and consistent logic for asynchronous digital circuit synthesis," In Proc. IEEE International Conference on Application Specific Systems, Architectures and Processors, August 1996, pp. 261–273.
29. Zhou, L.; Smith, S.C.; Di, J. Bit-Wise MTNCL: An Ultra-Low Power Bit-Wise Pipelined Asynchronous Circuit Design Methodology. In Proceedings of the IEEE Midwest Symposium on Circuits and Systems, Seattle, WA, USA, 1–4 August 2010; pp. 217–220.
30. Moon, C.W., Stephan, P.R. & Brayton, R.K. Journal of VLSI Signal Processing (1994) 7: 85.
31. C. J. Myers, *Asynchronous Circuit Design.* New York: Wiley, 2001.
32. Vidura M. Wijayasekara, S.K.Srinivasan and S. C. Smith, "Equivalence verification for NULL Convention Logic (NCL) circuits," In Proc. IEEE 32nd International Conference on Computer Design (ICCD), Oct 2014, pp. 195-201.
33. P. Manolios, "Correctness of pipelined machines," In Proc. Formal Methods in Computer-Aided Design–FMCAD 2000, ser. LNCS, Springer-Verlag, W. A. Hunt, Jr. and S. D. Johnson, Eds., vol. 1954, 2000, pp. 161–178.

34. A. A. Sakib, S. C. Smith, and S. K. Srinivasan, “Formal modeling and verification for pre-charge half buffer gates and circuits”, In Proc. IEEE International Midwest Symposium on Circuits and Systems, August 2017, pp. 519-522.
35. A. Peeters, F. te Beest, M. de Wit & W. Mallon, “Click elements: An implementation style for data-driven compilation,” In Proc. IEEE Symposium on Asynchronous Circuits and Systems (ASYNC’10), 2010, pp. 3-14.
36. F. Verbeek and J. Schmaltz, “Verification of building blocks for asynchronous circuits”, In Proc. ACL2, ser. EPTCS, R. Gamboa and J. Davis, Eds., vol. 114, 2013, pp. 70–84.
37. V. Satagopan, B. Bhaskaran, W. K. Al-Assadi, S. C. Smith, and S. Kakarla, “DFT techniques and automation for asynchronous NULLconventional logic circuits,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 15, no. 10, pp. 1155–1159, October 2007.
38. W. Al-Assadi and S. Kakarla, “Design for test of asynchronous NULL Convention logic (NCL) circuits”, Journal of Electronic Testing, vol. 25, no. 1, 2009, pp. 117–126.
39. 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.
40. T. E. Williams, “Self-Timed Rings and Their Application to Division”, Ph.D. Thesis, CSL-TR-91-482, Department of Electrical Engineering and Computer Science, Stanford University, 1991.
41. D. Bryan, The ISCAS ‘85 benchmark circuits and netlist format [online] Available: https://ddd.fit.cvut.cz/prj/Benchmarks/iscas85.pdf. [Accessed Jul. 10, 2019].
42. F. Brglez, D. Bryan, and K. Kozminski, “Combinational profiles of sequential benchmark circuits,” In Proc. Int’l. Symp. Circuits and Systems, 1989, pp. 1929-1934.

APPENDIX. PUBLICATION LIST

- Refereed Technical Conference: **M. Hossain**, A. A. Sakib, S. C. Smith, and S. K. Srinivasan, "An Equivalence Verification Methodology for Asynchronous Sleep Convention Logic Circuits," IEEE International Symposium on Circuits and Systems (ISCAS), 2019, pp. 1-5.