University of Arkansas, Fayetteville
ScholarWorks@UARK

Theses and Dissertations

5-2013

CAD Tools for Synthesis of Sleep Convention Logic

Parviz Palangpour
University of Arkansas, Fayetteville

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

Part of the Digital Circuits Commons, and the VLSI and Circuits, Embedded and Hardware Systems Commons

Recommended Citation
Palangpour, Parviz, "CAD Tools for Synthesis of Sleep Convention Logic" (2013). *Theses and Dissertations*. 755.
http://scholarworks.uark.edu/etd/755

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

CAD TOOLS FOR SYNTHESIS OF SLEEP CONVENTION LOGIC

CAD TOOLS FOR SYNTHESIS OF SLEEP CONVENTION LOGIC

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

By

Parviz M Palangpour
Missouri University of Science and Technology
Bachelor of Science in Computer Engineering, 2007
Missouri University of Science and Technology
Master of Science in Computer Engineering, 2010

May 2013
University of Arkansas

ABSTRACT

This dissertation proposes an automated flow for the Sleep Convention Logic (SCL) asynchronous design style. The proposed flow synthesizes synchronous RTL into an SCL netlist. The flow utilizes commercial design tools, while supplementing missing functionality using custom tools. A method for determining the performance bottleneck in an SCL design is proposed. A constraint-driven method to increase the performance of linear SCL pipelines is proposed. Several enhancements to SCL are proposed, including techniques to reduce the number of registers and total sleep capacitance in an SCL design.

This dissertation is approved for recommendation
to the Graduate Council.

Dissertation Director:

____

Dr. Scott C. Smith

Dissertation Committee:

____

Dr. Jia Di

____

Dr. Alan Mantooth

____

Dr. Jingxian Wu

DISSERTATION DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this dissertation when needed for research and/or scholarship.

Agreed ____

Parviz M Palangpour

Refused ____

Parviz M Palangpour

ACKNOWLEDGMENTS

I am deeply grateful to my advisor Dr. Scott C. Smith, who introduced me to the world of digital asynchronous design and has provided me with guidance, knowledge and financial support throughout the research and preparation of this dissertation. I would also like to thank my defense committee members, Dr. Di, Dr. Mantooth and Dr. Wu. Most importantly, I would like to thank my wife, Winnie, and my parents for their unconditional support and encouragement towards reaching my goals.

TABLE OF CONTENTS

1 INTRODUCTION **1**
1.1 Objectives 1
1.2 Design Challenges 1

2 BACKGROUND **4**
2.1 Introduction 4
2.2 Synchronous Clocking Schemes 4
2.3 Asynchronous Handshaking 5
2.4 Asynchronous Data Encoding 7
2.4.1 Bundled-Data Channels 8
2.4.2 One-Hot Encoded Channels 8
2.5 Slack Elasticity 10
2.6 Timing Models 10
2.7 Petri Networks 11
2.8 Asynchronous Design Styles 13
2.8.1 NULL Convention Logic 13
2.9 Asynchronous Synthesis Tools 16

3 MTCMOS POWER-GATING **20**

4 SLEEP CONVENTION LOGIC **22**
4.1 Introduction to SCL 22
4.2 SCL Function Block 22
4.3 SCL Register 23
4.4 SCL Completion Detector 26
4.5 SCL Final Completion Gate 26
4.6 SCL Pipeline Initialization 26
4.7 SCL Performance and Timing Assumptions 27

5 SYNCHRONOUS TO SCL CONVERSION **35**
5.1 Synchronous and SCL Equivalence 35
5.2 Extracting Connectivity Information from Netlists 37
5.3 Determining Acknowledge and Sleep Networks 42
5.4 Determining Pipeline Stages 44
5.5 Combining Pipeline Stages 45

6 SCL Performance Analysis **52**

7 SCL OPTIMIZATION TECHNIQUES **55**
7.1 SCL Embedded Registration 55
7.2 SCL Partially Slept Function Blocks 55
7.3 SCL Pipeline Standby Detection 57
7.4 SCL Pipelining 59

8 AUTOMATED SCL CONVERSION FLOW **63**
8.1 Generating the Single-Rail Netlist 63
8.2 Generating the Dual-Rail Netlist 63
8.3 Optimizing the Dual-Rail Netlist 65
8.4 Completing the SCL Netlist 66
8.5 Validating the SCL Netlist Equivalance 66
8.6 Experimental Results 66

9 CONCLUSION **68**

10 REFERENCES **69**

LIST OF FIGURES

1 Timing waveform for flip-flop. 5
2 Timing waveform for latch. 6
3 The 4-phase handshaking protocol. 7
4 Two asynchronous blocks communicating via a channel. 7
5 The 4-phase handshaking protocol using one-hot encoding. 9
6 Two asynchronous blocks communicating via a channel. 10
7 NCL linear pipeline of three registers. 15
8 A marked graph model of the NCL pipeline with critical paths highlighted. 15
9 NCL EC linear pipeline of three registers. 16
10 A marked graph model of the NCL EC pipeline with first race condition highlighted. 16
11 A marked graph model of the NCL EC pipeline with second race condition highlighted. 17
12 MTCMOS power-gating architecture [32]. 21
13 Basic architecture of SCL linear pipelines. 23
14 Transistor-level diagram of SCL threshold gate [37]. 24
15 Transistor-level diagram of SCL register [37]. 25
16 SCL linear pipeline with no combinational logic. 28
17 SCL pipeline with sleep buffers. 30
18 Marked graph model of SCL pipeline. 33
19 Marked graph model of SCL pipeline with race paths indicated by thick lines. 33
20 Signals for SCL pipeline with critical cycle indicated by dotted path. 34
21 Signals for SCL pipeline with race indicated by dashed and dotted paths. 34
22 Synchronous design with flip-flops. 38
23 Output trace of synchronous pipeline. 38
24 SCL pipeline translated from synchronous pipeline. 39
25 Output trace of 2-stage SCL pipeline. 39
26 SCL pipeline translated from synchronous pipeline. 39
27 Output trace of 3-stage SCL pipeline. 40
28 Synchronous pipeline with direct feedback on register. 40
29 SCL pipeline stage. 43
30 Abstract SCL datapath. 46
31 Abstract data path with each register partitioned into unique pipeline stages. 46
32 Acknowledgement network for partitioning in Figure 31 47
33 Sleep networks for partitioning in Figure 31 48
34 Acknowledgement network for merged partitioning. 49
35 Sleep networks for merged partitioning. 50
36 Abstract SCL pipeline with datapath loop. 50
37 Sleep networks for abstract data path in Figure 36. 51
38 A three stage SCL pipeline represented as a MG. 53
39 A three stage MG SCL with the second stage initialized to DATA. 54
40 Delay-dependent algorithm for partitioning slept and non-slept gates in F_i. 57
41 Greedy vertex coloring for partitioning slept and non-slept gates in F_i. 58
42 A three-stage SCL pipeline with standby-detection logic. 59

43 Pipeline configurations for 4x4 unsigned multiplier. 62
44 Efficient pipelining algorithm for linear SCL pipeline. 62
45 The flow for automated synchronous to SCL conversion. 64

LIST OF TABLES

1 Pipelining partitions for 4x4 unsigned multiplier. 61
2 Area of ISCAS'89 Designs 67

1 INTRODUCTION

1.1 Objectives

The objective of this Ph.D. dissertation is to develop tools that support an automated flow from a synchronous Register-Transfer Level (RTL) description to a gate-level netlist for Sleep Convention Logic (SCL). The tools developed in this dissertation leverage commercial software for logic synthesis while providing custom tools for implementing SCL handshaking and performance analysis. Experimental results are presented to validate the proposed design tools.

1.2 Design Challenges

Synchronous design methods have dominated the digital VLSI industry for the last several decades. However, as the industry moves towards smaller process geometries, achieving timing closure in synchronous designs has become increasingly challenging. To reach timing closure, the design must be verified to operate reliably across all expected operating conditions at the desired clock frequency. Specifically, as wire delays and process variation become more significant, distributing the global clock signal on complex ICs (integrated circuits) while meeting the clock-related timing constraints is becoming an increasingly difficult task. In order to account for varying delays, designers typically increase the timing margins in the clock period, which results in reduced performance.

Another rising issue in IC design is the growing dynamic and static power consumption. The switching of large clock distribution networks is responsible for a significant amount of dynamic power consumption in modern digital ICs. This has resulted in the industry adopting com-

plicated clocking schemes to reduce the power wasted by the clock distribution network. More recently, with semiconductor devices scaling deep into the submicron region, static power has now become a primary concern as well [12][6]. Several circuit-level techniques have been adopted to lower static power consumption; however, these techniques reduce static power at the expense of design complexity, area, and/or performance. While there are several techniques to reduce the dynamic and static power dissipation in synchronous circuits, as the design complexity increases timing closure can become even more difficult. Asynchronous circuits, specifically Sleep Convention Logic (SCL)[32], can address many of these synchronous design issues. Asynchronous circuits eliminate the clock signal and hence eliminate the large effort required to distribute the clock signal and verify the complicated clock-related timing constraints. In addition, the power wasted in the clock distribution network is eliminated and SCL has a built-in sleep mechanism that drastically reduces static power consumption. Many asynchronous styles, including SCL, utilize completion detection in order to adapt to varying delays. This means designers don't have to add any explicit timing margins to allow for variation in delay; the circuit simply adapts to the current operating conditions and operates at the fastest performance possible.

While asynchronous circuits offer many advantages, they have not seen widespread use in the VLSI industry. Synchronous design flows based on commercial automated design tools have been heavily used and improved over the course of the last twenty years. However, due to the lack of automated asynchronous design tools, asynchronous design flows have mainly been restricted to custom designs, which require substantially more design effort [9][21]. Without the support of automated design flows the time and costs associated with custom asynchronous designs are too

high for broad adoption by VLSI companies. The motivation behind this work is to develop an automated flow for the SCL asynchronous design style. Using the automated tools developed in this work, the advantages of asynchronous designs can be achieved at a much lower design effort than by a custom asynchronous design flow. In an effort to reduce the barrier to adoption, the flow leverages proven commercial synchronous design tools that are widely used in industry.

2 BACKGROUND

2.1 Introduction

Digital systems are typically composed of combinational logic blocks, which are separated by sequential elements. The sequential elements are used to safely synchronize the transfer of data between one combinational stage and the next. Synchronous designs utilize one or more periodic clock signals to control when the sequential elements pass the input data to the next stage. While the synchronization between sequential elements in a synchronous circuit is achieved using periodic clock signals, asynchronous circuits achieve synchronization through local handshaking signals between stages.

2.2 Synchronous Clocking Schemes

The large majority of the digital systems designed today are synchronous and utilize the edge-triggered flip-flop as the sequential element. Timing for a typical positive-edge-triggered flip-flop pipeline is shown in Figure 1. The clock signal is used to indicate when the sequential elements can safely sample their inputs. The clock period (t_c) indicates how often the flip-flops will sample their inputs and propagate the values to the next stage. Due to an inherent race condition in flip-flops, two timing constraints known as setup (t_{su}) and hold (t_{hold}) times must be satisfied. The setup time constraint requires that the input signal to a flip-flop does not change less than t_{su} before the active edge of the clock. The hold time constraint requires that the input signal to the flip-flop does not change less than t_{hold} after the active edge of the clock. Failure to satisfy these constraints can result in a disastrous condition known as metastability [10].

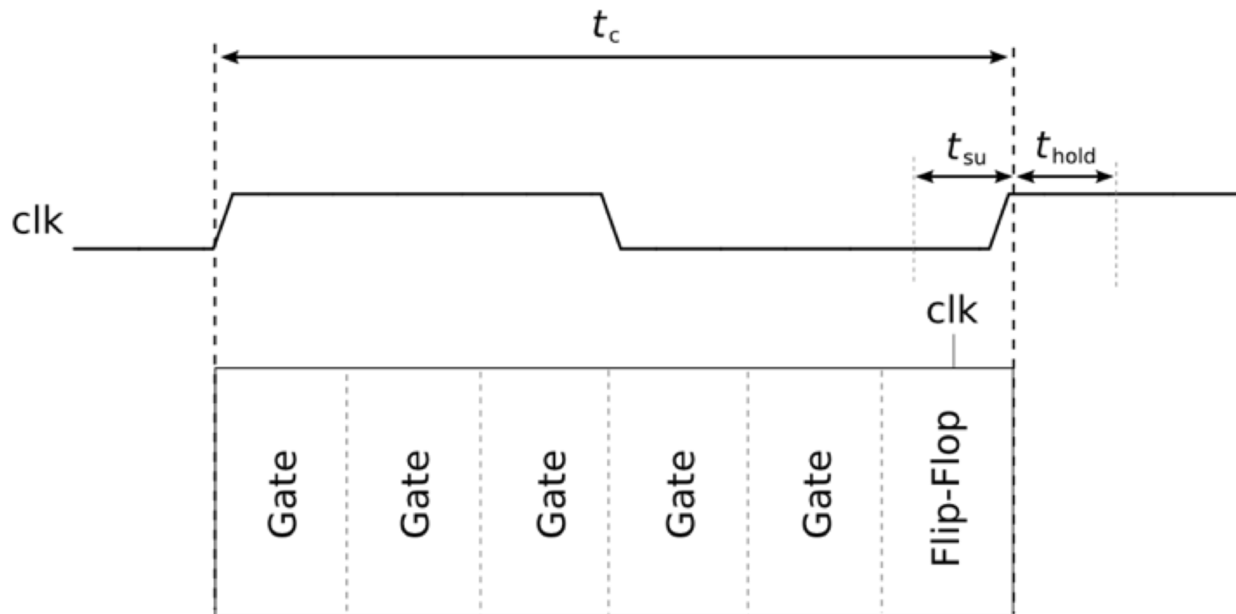

Figure 1: Timing waveform for flip-flop.

Flip-flops are often constructed from a pair of sequential elements known as latches. Latches have similar setup and hold time constraints to those discussed for flip-flops. However, latches are level-sensitive, which means the output of a latch follows the input as long as the clock input is high. Each of the latches inside the flip-flop is transparent for a different phase of the clock signal. The flip-flops can essentially be split into two separate latches, which are each controlled by a separate clock signal. The two clocks, ϕ_1 and ϕ_2 are inverted with respect to each other. A single clock cycle t_c now consists of two adjacent stages as opposed to a single stage in a flip-flop-based system, as illustrated in Figure 2.

2.3 Asynchronous Handshaking

The most commonly used handshaking protocol in asynchronous circuits is the 4-phase protocol, illustrated in Figure 3. When the sender has generated stable data it asserts the request

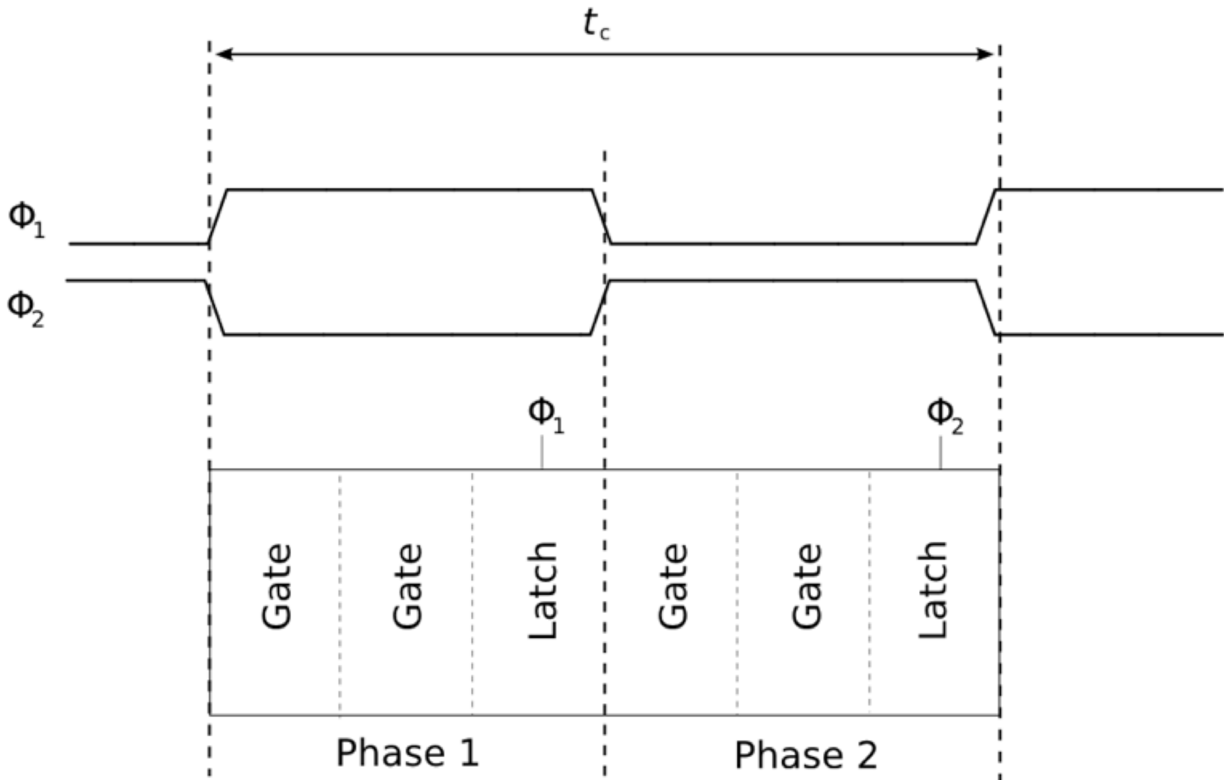

Figure 2: Timing waveform for latch.

(REQ) signal. The receiver can now sample the data and assert the acknowledge (ACK) signal. The sender can now reset the request signal, which is followed by the receiver resetting the acknowledge signal. The 4-phase handshaking protocol has now reset and is ready to transfer the next data token. The 4-phase protocol requires four transitions per data transfer, thus the name. One arrangement of two communicating blocks, F_1 and F_2 is shown in Figure 4. Note that the blocks F_1 and F_2 could be as low-level as two adjacent pipeline stages or as high-level as two communicating processors. As long as the two blocks communicate with a standard asynchronous handshaking scheme, no additional effort is required to match their communication rates. If one block attempts to communicate at a faster rate than the receiving block can tolerate, the handshaking protocol will ensure no data is lost. This is in contrast to synchronous design where additional

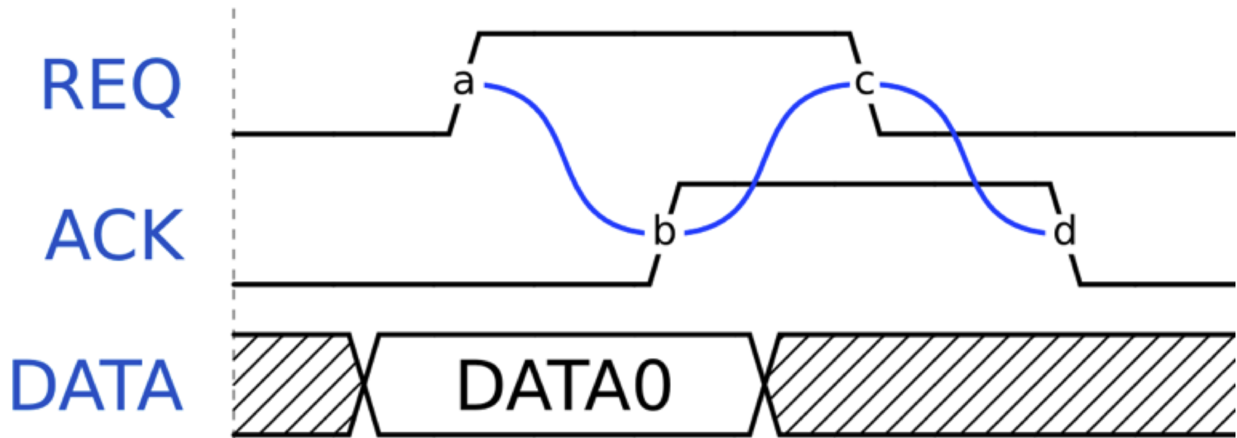

Figure 3: The 4-phase handshaking protocol.

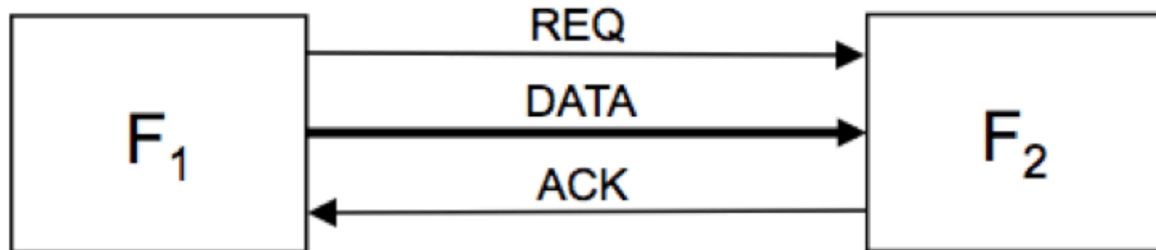

Figure 4: Two asynchronous blocks communicating via a channel.

design and verification effort is required to interface blocks that operate at different clock speeds. A data bus and its associated handshaking signals grouped together are referred to as a channel. The channel-based interfaces that form the input and outputs of asynchronous blocks make them modular, allowing simpler integration of blocks to form a complete system.

2.4 Asynchronous Data Encoding

A variety of different data encodings can be used in asynchronous circuits; a single asynchronous circuit may utilize multiple encodings. The most commonly used encoding in synchronous and asynchronous designs is single-rail. Here, single-rail encoding refers to binary en-

coded data where 2^n distinct symbols can be represented by the Boolean symbols '0' and '1' using n wires. In single-rail encoding, all possible combinations of '0' and '1' can represent valid data.

2.4.1 Bundled-Data Channels

Bundled-data is the most similar to synchronous data transfer and is the most popular data encoding used in asynchronous circuit design [35][8]. Bundled-data channels are simply a single-rail encoded data bus bundled with two additional signals representing the request and acknowledge handshaking signals. Hence, a channel that can transmit n-bits per transfer requires (n+2) physical wires. This is in contrast to a synchronous design, which requires only a single additional signal, the clock. Bundled-data asynchronous designs utilize the same flip-flop and latches used in synchronous designs. As a result, use of bundled-data dictates strict timing constraints similar to those found in synchronous design. The timing of the request signal in relation to the data becoming valid must be verified in the physical design; this results in complicated delay-matching that must be performed on each individual bundled-data channel.

2.4.2 One-Hot Encoded Channels

Instead of using separate signals for data and request, an alternative data encoding is to encode the validity of the data into the data signal. Consider a one-hot encoded signal of n-wires, which can represent n distinct symbols. A single wire can be asserted at one time while the others must remain low. The state where all n wires are low can be used to represent the absence of a symbol, referred to as the NULL state. This allows the validity of the data to be physically combined with the data itself. The assertion of a single wire indicates both the transmitted symbol

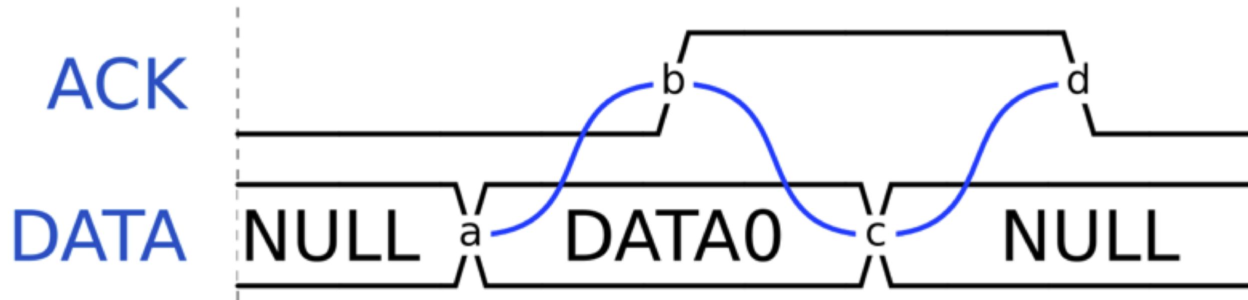

Figure 5: The 4-phase handshaking protocol using one-hot encoding.

as well as the fact that the symbol is valid and ready to be sampled. An OR gate can be used to detect the validity of data on a one-hot encoded channel. The component that accomplishes the detection of data is referred to as a completion detector. Since the validity of the data is encoded in the data, the request signal can be eliminated. The 4-phase handshaking protocol using a one-hot encoded channel is shown in Figure 5. Two asynchronous blocks communicating using a one-hot encoded channel is shown in Figure 6. This data-encoding scheme is the basis for delay-insensitive asynchronous communication. The most commonly used one-hot codes are 1-of-2 and 1-of-4, referred to as dual-rail and quad-rail, respectively. Dual-rail encoding is more widely used than quad-rail due to simplicity. However, quad-rail encoding has a power advantage due to the fact that it requires half the number of signal transitions compared to dual-rail. Transmitting a pair of Boolean values using dual-rail will require two dual-rail channels that each must switch a signal high to become valid data, while a single quad-rail channel only needs to switch a single wire to transmit the two Boolean values.

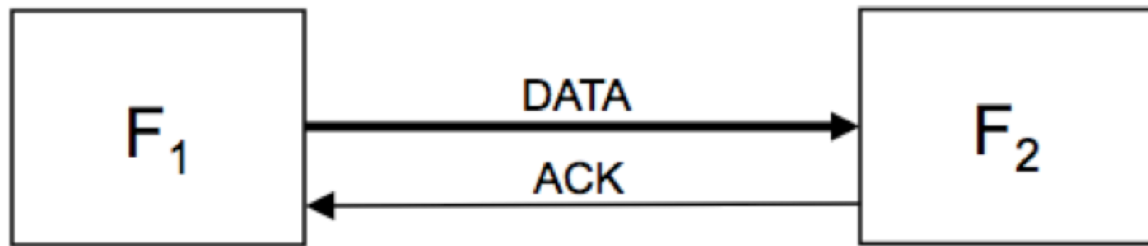

Figure 6: Two asynchronous blocks communicating via a channel.

2.5 Slack Elasticity

Synchronous designs lacking handshaking must anticipate the arrival of data after a fixed number of clock cycles. For instance, a path with three registers will result in a latency of three clock cycles. Increasing the number of registers on a path in a synchronous pipeline changes the behavior of the pipeline. However, due to the inherent handshaking in asynchronous circuits, additional registers can be inserted on a path and still maintain observational equivalence with the original pipeline. This property is referred to as slack elasticity [4].

2.6 Timing Models

The most distinctive attribute of any design style is the assumptions made with respect to the timing characteristics of signals. In synchronous and bundled-data asynchronous design, timing assumptions are made on the arrival of the clock or control signal relative to the arrival of data at each sequential element. These assumptions make the logical design straightforward while making the physical design more difficult. Ensuring the timing relationships between all related sequential elements and the respective combinational delays and wires in older CMOS processes was far less challenging. However, as device geometries shrink, the manufacturing variation increases.

The increasing delay uncertainty of wires and transistors poses a critical problem to the design of synchronous circuits due to the inherent assumption on delays. It's important to note that the timing failure of a single flip-flop in a fabricated multi-million-gate design can cause the entire design to be non-functional. As a result, the clock rates are reduced to increase the timing window in which the data or clock signals may arrive.

In contrast to the delay-dependent synchronous and bundled-data schemes, the most robust circuits are those that adhere to the Delay-Insensitive (DI) timing model. The devices and wires in DI circuits can take on any value and the circuit will still function correctly. However, it has been shown that the DI timing model is too restrictive to design practical circuits [17]. A slightly more relaxed delay model, referred to as Quasi Delay Insensitive (QDI), is similar to DI except it requires that all wire forks be isochronic, which means that wire delays within basic components, such as a full adder, are much less than the delay through a logic gate. Designs that adhere to the QDI timing model utilize one-hot encoded channels.

2.7 Petri Networks

Petri networks are a mathematical modeling language for distributed systems. A Petri net is a directed bipartite graph, in which vertexes can be either a transition or a place. A transition, often symbolized by a vertical bar or square, represents events that occur. Places, often symbolized by circles, represent conditions. Each place can contain zero or more tokens, represented by black dots inside the place, at any given moment. A place is said to be marked if it contains a token. Directed edges connect places to transitions and transitions to places. In this dissertation, a compressed format for illustrating Petri nets is used. The Petri net in Figure 8 uses text labels for transitions.

In addition, places are not explicitly shown unless initialized with a token, which is illustrated by a filled black circle; an edge between two transitions is assumed to represent two edges, separated by a place.

For the application of asynchronous performance modeling, a specific type of Petri net known as a Marked Graph (MG) is used. In a MG, every place can only have a single incoming edge from a transition and a single outgoing edge to a transition. Each transition can have multiple incoming and outgoing edges from and to places. For each transition the set of incoming places is called the preset while the set of outgoing places is called the postset. When all of the places in a transition's preset contain at least one token, the transition is said to fire, and one token will be removed from each place in the transition's preset and one token will be added to each place in the transition's postset. For the modeling of asynchronous circuits, a fixed time delay is assigned to each transition. The transitions only fire after their preset is satisfied and their fixed delay has elapsed. While MGs are a restricted form of Petri Nets, using MGs to model the performance of asynchronous circuits is appealing because the cycle-time of a MG is known to be:

$$\max_{c_i \in C_{MG}} \left(\frac{\sum_{v_i \in c_i} d(v_i)}{\sum_{v_i \in c_i} m(v_i)} \right) \tag{1}$$

where a cycle, c_i, is a sequence of nodes that start and end at the same node; C_{MG} is the set of all simple cycles in MG; $d(v_i)$ is the delay of node v_i; and $m(v_i)$ is the number of tokens initialized in node v_i [24]. In other words, the cycle time for a cycle is the sum of the delays of the transitions along the cycle, divided by the number of tokens in the cycle. The cycle time for the entire MG is equal to the largest cycle time of any cycle in the MG. This cycle time is the performance metric

for asynchronous circuits. However, enumerating over all the cycles in a MG is computationally expensive. The cycle time can be found using efficient algorithms for the maximum cycle mean problem; Karps algorithm has $O(|V| * |E|)$ time complexity, where V is the set of nodes and E is the set of edges in the MG. This provides a means for determining both the worst-case throughput and the critical cycle for an asynchronous design. The critical cycle is analogous to the critical path in a synchronous design.

2.8 Asynchronous Design Styles

There are several different asynchronous design styles, utilizing different data-encodings and timing assumptions, which make each design style advantageous for different applications. The most popular asynchronous design styles are the Pre-Charge Half Buffer (PCHB) [26], which is used in high-performance applications, and NULL Convention Logic (NCL) [7], which is used in lower performance applications.

2.8.1 NULL Convention Logic

NCL is a QDI (Quasi-Delay-Insensitive) asynchronous design style [7]. Each pair of adjacent registers communicates using the common 4-phase handshaking protocol. All combinational logic and registers in NCL are built using special threshold gates with hysteresis. An NCL THmn gate refers to a threshold gate that is asserted when at least m of the n inputs are asserted. NCL gates have hysteresis, such that once the gate is asserted it will remain asserted until all the inputs are de-asserted. The first condition required for an NCL circuit to be QDI is that the combinational logic between registers must be input-complete. Input-complete logic will only allow all outputs to

transition to DATA (NULL) after all inputs have transitioned to DATA (NULL). This often results in an area, performance, and/or power overhead but is crucial to achieve a QDI implementation in NCL. NCL gates must have hysteresis to enforce input-completeness with respect to NULL, such that a circuit's outputs cannot transition back to NULL until all inputs have become NULL. The second condition for an NCL circuit to be QDI is that the signal transitions in the combinational logic are observable, such that each gate that transitions during a DATA/NULL wavefront must contribute to a transition on an output of the combinational logic. This ensures that every gate output is returned to logical 0 before the circuit output is NULL, such that the circuit is ready to receive the next DATA wavefront.

A simple linear pipeline of NCL registers is shown in Figure 7. The registers consist of a pair of TH22 gates, also known as C-elements [23]. The NOR2 gates function as completion detectors and acknowledge the previous stage. The cycle time of the NCL pipeline can be derived from the marked graph model in Figure 8. As can be seen from the marked graph model, there are two critical cycles of events: $Q_1^D, F_1^D, Q_2^D, F_2^D, Q_3^D, Ko_3^{RFN}, Q_2^N, Ko_2^{RFD}$ and $Q_1^D, F_1^D, Q_2^D, Ko_2^{RFN}, Q_1^N, F_1^N, Q_2^N, Ko_2^{RFD}$. The cycle with the largest total delay determines the throughput for the NCL pipeline.

Early Completion is an enhancement that can increase throughput of a conventional NCL pipeline [31]. Early completion increases throughput by moving the completion detectors to in front of the registers and adds control logic which anticipates the latching of DATA or NULL. The speculation control logic is implemented by a final c-element. However, two race conditions are introduced by early completion [31]. The two race conditions are illustrated by the petri net models

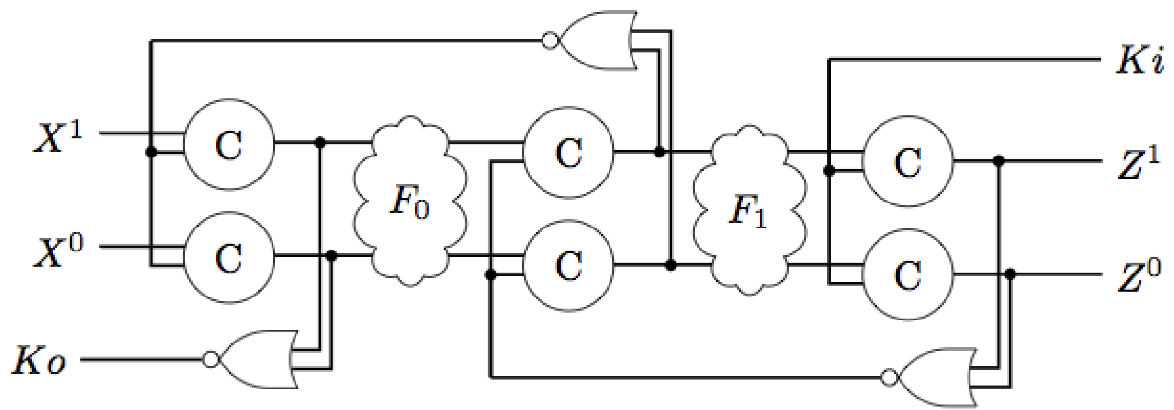

Figure 7: NCL linear pipeline of three registers.

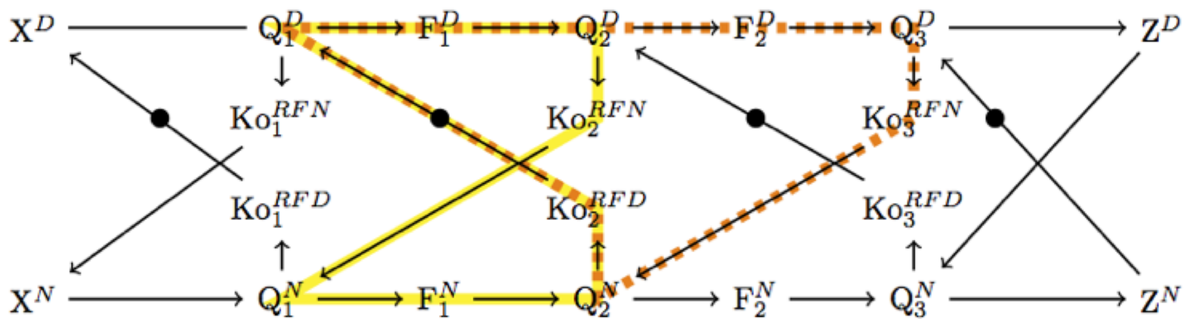

Figure 8: A marked graph model of the NCL pipeline with critical paths highlighted.

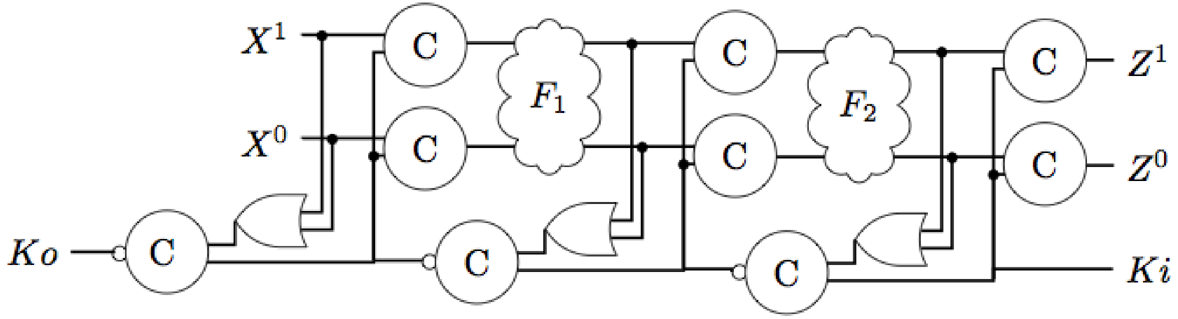

Figure 9: NCL EC linear pipeline of three registers.

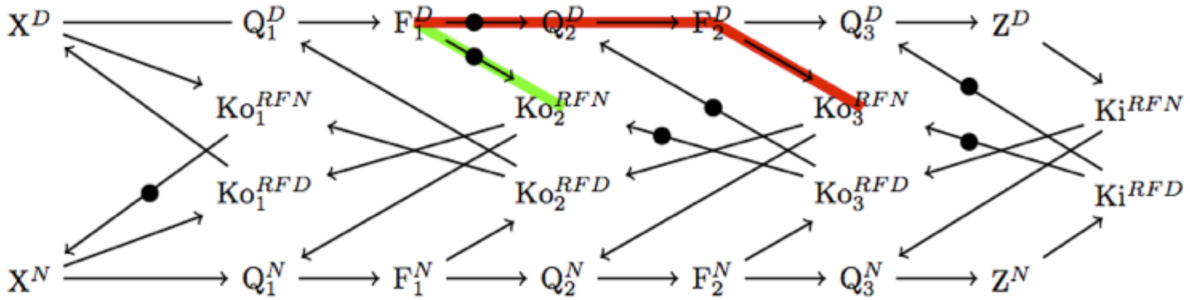

Figure 10: A marked graph model of the NCL EC pipeline with first race condition highlighted.

in Figures 10 and 11. The first condition is violated if a stage can transition its final completion gate before the preceding stage's final completion gate; in other words, the events Q_i^D, F_i^D, and Ko_{i+1}^{RFN} can occur before the final completion gate transitions, Ko_i^{RFN}. The second condition is violated if a stage's data output can transition before the following stage can register it; in other words, the events Ko_i^{RFN}, Q_{i-1}^N, and F_{i-1}^N can occur before the register latches the DATA, Q_i^D. Each condition is symmetric with respect to DATA/NULL.

2.9 Asynchronous Synthesis Tools

Several different asynchronous synthesis systems have been developed so far; some of the more complete efforts include the Cal-Tech Asynchronous Synthesis Tool (CAST)[18][19][20],

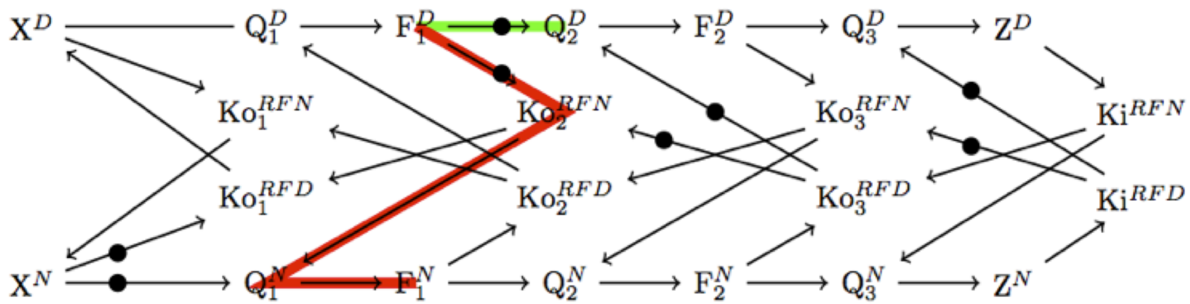

Figure 11: A marked graph model of the NCL EC pipeline with second race condition highlighted.

Balsa [1][2], NCL-X [14], Phased-Logic [16], De-synchronization [5], Weaver [29], Proteus [3], and the Unified NCL Environment (UNCLE)[27]. Each of these tools is designed to generate asynchronous circuits; however the approaches have some significant differences.

One of the fundamental differences in the tools is the choice of language for the design specification. Both CAST and Balsa utilize custom languages based on the CSP (Communicating Sequential Processes) language. The use of CSP-based design specifications has some advantages and disadvantages; while CSP-based languages allow for very elegant and concise descriptions of asynchronous channel-based systems, they require designers to use an entirely different language than used for synchronous design. This presents a serious barrier to adoption by synchronous design companies. Experienced synchronous designers who have been using VHDL and Verilog for decades must now become proficient in a new language. In addition, legacy designs written in VHDL or Verilog will need to be re-written in the appropriate language before they can be synthesized by CHP or Balsa. Although academic simulation tools have been developed for CHP and Balsa, the tools are fairly primitive compared to the commercial simulation tools that are available for VHDL and Verilog.

Commercial synchronous design tools have been developed and improved by companies for over twenty years. Developing competitive asynchronous design tools from scratch would require a very large effort. The more practical approach is to utilize as many commercial synchronous designs tools as possible. While the CAST and Balsa flows utilize entirely custom tools, NCL-X, Phased-Logic, Weaver, Proteus, and UNCLE use synchronous design tools for RTL synthesis and technology mapping, while using custom tools to supplement the missing procedures for asynchronous design. The end result is an asynchronous design that has been translated from a synchronous design.

Theseus Logic was the first to develop a partially automated flow from synchronous RTL to their NULL Convention Logic asynchronous design style. While the synchronous datapath logic was automatically translated to NCL, the designers had to manually instantiate NCL registers and connect their handshaking signals [30][15][22]. The NCL-X flow can be viewed as a the fully automated successor to Theseus Logic's initial flow. UNCLE is a more powerful set of tools, allowing designers to develop NCL/Balsa hybrid designs using synchronous RTL and providing automated acknowledgment network merging. However, the use of Balsa-like primitives must be manually instantiated in the RTL by a designer familiar with asynchronous design.

While the other QDI flows discussed here utilize dual-rail encoding, Phased Logic utilizes a unique data encoding such that each code-word corresponds to a data and a phase. Each encoded value alternates phase, making successive DATA encodings distinguishable without the need for a NULL spacer. The principle idea is that by removing the NULL spacer, unnecessary switching can be removed, resulting in reduced dynamic power. However, the Phased Logic flow requires

the use of complicated custom gates and a complicated conversion procedure.

The De-synchronization approach uses conventional synchronous design tools as well as conventional synchronous standard cell libraries. The approach is based on first translating a flip-flop based synchronous design to a latch-based synchronous design as discussed in Section 2.2. Each flip-flop is then split into a pair of latches and control logic is added to implement asynchronous channels between adjacent latches. Unlike the QDI flows which utilize multi-rail encodings, De-synchronization uses the bundled-data channels which are more area efficient. However, the synthesis procedure requires carful implementation of a matched delay line which may require a significant amount of analysis.

Both Weaver and Proteus translate a synchronous design into a high-performance PCHB asynchronous design. While the previously discussed conversion flows retain the same pipeline granularity of the original synchronous design, Weaver and Proteus translate the synchronous design into fine-grained pipelines. While the resulting PCHB designs are often significantly faster than the original synchronous design, the area of the PCHB design could be over ten times higher than the original synchronous design.

3 MTCMOS POWER-GATING

MTCMOS processes provide multiple transistors with different threshold voltages (V_{th}). Transistors with higher V_{th} are slower and have lower leakage current, while lower V_{th} transistors are faster but suffer from higher leakage current. MTCMOS can be utilized to reduce leakage power by disconnecting the power supply from portions of the circuit that are idle [25]. This power-gating is implemented using low-leakage high-V_{th} transistors, while the switching logic is implemented using faster low-V_{th} transistors. A high-V_{th} PFET transistor used to disconnect the circuit from the supply is referred to as a 'header', while a high-V_{th} NFET transistor used to disconnect the circuit from ground is referred to as a footer. The signal that is used to power-up or power-down a block is referred to as the sleep signal. A block-level diagram of power-gating using both a header and footer is illustrated in Figure 12. The control logic that generates the sleep signal is generally application-dependent.

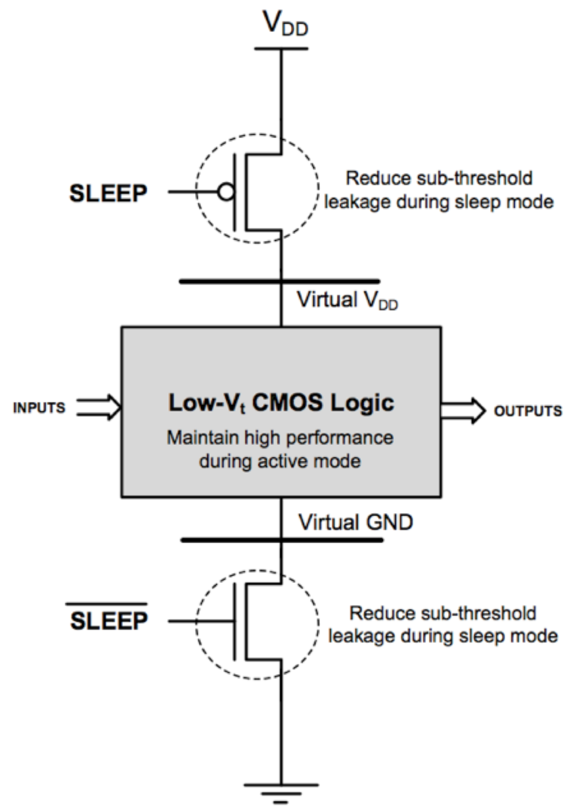

Figure 12: MTCMOS power-gating architecture [32].

4 SLEEP CONVENTION LOGIC

Sleep Convention Logic (SCL) was originally developed in [32], as summarized below, with the addition of analysis of the performance and timing assumptions in Section 4.7.

4.1 Introduction to SCL

SCL is a self-timed asynchronous pipeline style that offers inherent power-gating, resulting in ultra-low power consumption while idle. SCL combines the ideas of NCL with early completion and MTCMOS power-gating. The basic architecture of an SCL pipeline is shown in Figure 13. A single stage i of an SCL pipeline is composed of a register (R_i), a function block (F_i), a completion detector (CD_i) and a final completion gate (C_i). The MTCMOS power-gating sleep input of a block is denoted by s. Each stage communicates with the adjacent stages using the 4-phase handshaking protocol discussed in Section 2.4.2. Much like NCL, each pipeline stage in SCL undergoes alternating cycles of DATA evaluation and reset to NULL.

4.2 SCL Function Block

The SCL function block is implemented using SCL threshold gates to perform the required logic function. An SCL function block has 1-of-M encoded data inputs and outputs; the logic implemented by the function block must be strictly unate and thus free of any logical inversions. The low static power consumption in SCL is achieved by utilizing MTCMOS power-gating. Each SCL threshold gate utilizes high-V_{th} sleep transistors to provide gate-level power-gating. When the sleep signal of an SCL gate is asserted, the power is disconnected through the sleep transistor

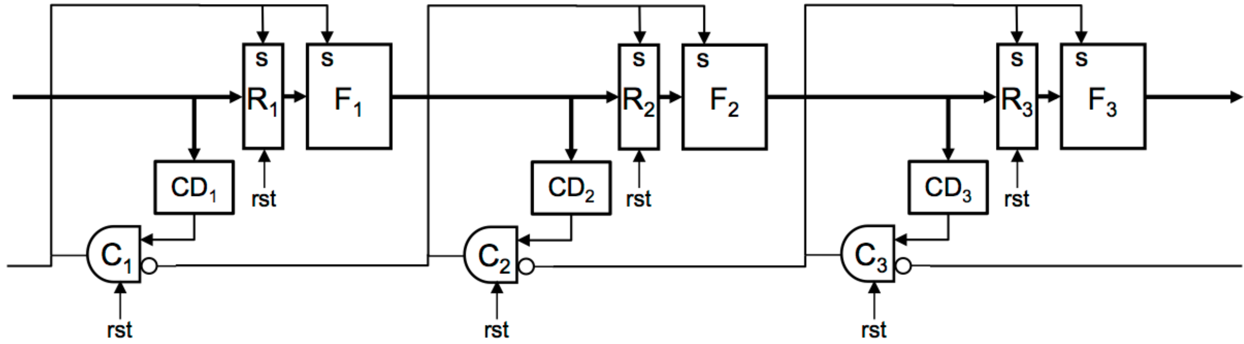

Figure 13: Basic architecture of SCL linear pipelines.

and the output of the gate is pulled to logical 0. Conversely, the gate cannot evaluate to a logical 1 until both sleep is de-asserted and the input values satisfy the threshold of the gate. An example of an SCL TH23 is shown in Figure 14, where the high-V_{th} transistors are circled.

4.3 SCL Register

The SCL register plays a similar role to a synchronous latch. Each M-rail SCL register has M input rails and M output rails. The transistor-level diagram of a dual-rail SCL register is shown in Figure 15. When the sleep input is asserted the register goes into a power-gated state and the outputs are pulled to logical 0. After the sleep signal is released the register comes out of the power-gated state and is ready for one of the input rails to be asserted. Once an input rail is asserted the corresponding output rail will be asserted and remain asserted until the sleep signal is asserted. Note the latching behavior that results in the output rails remaining asserted regardless of the input rails is distinguished from the strictly combinational SCL threshold gates.

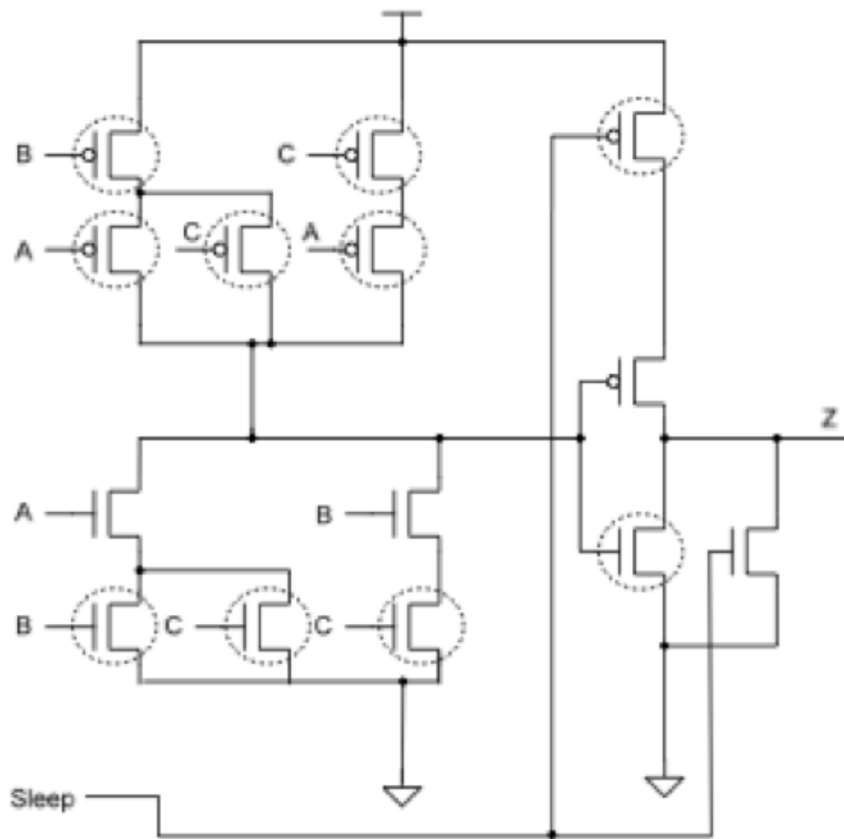

Figure 14: Transistor-level diagram of SCL threshold gate [37].

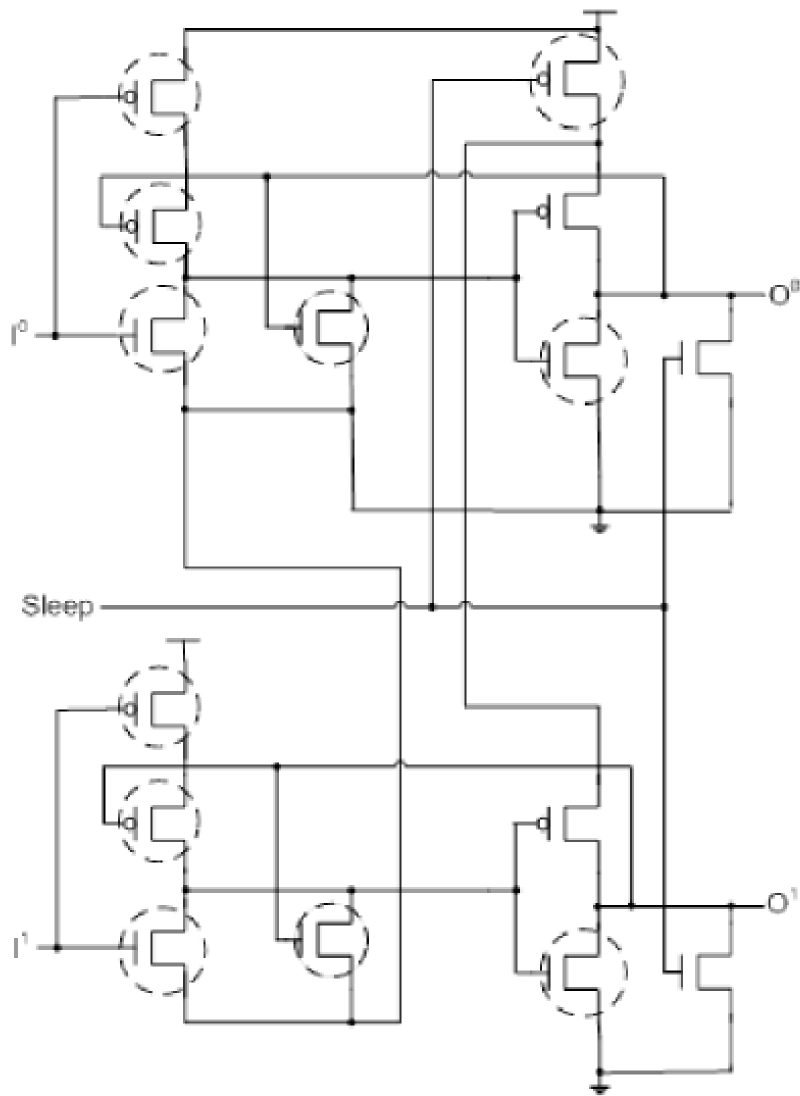

Figure 15: Transistor-level diagram of SCL register [37].

4.4 SCL Completion Detector

As SCL is derived from NCL with early completion, the completion detector CD_i checks for the presence or absence of DATA at the input to registers in stage i. The first level of logic in the completion detectors consists of NOR gates that generate logical 0 when the input has transitioned to DATA and logical 1 when the input has transitioned to NULL. A fan-in tree consisting of C-elements is used to combine the outputs of the NOR gates and generate a single acknowledge output.

4.5 SCL Final Completion Gate

A final completion gate, C_i, is needed, which is simply a C-element that implements the control logic that acknowledges stage $i-1$ and puts the pipeline stage i in the sleep state. Stage i will exit the sleep state as soon as CD_i has detected that the inputs are DATA and stage $i+1$ has acknowledged NULL. As stage i exits the sleep state, R_i will latch the DATA present at the input and F_i will generate DATA at the stage output. Stage i will then enter the sleep state as soon as CD_i has detected that the inputs are NULL and stage $i+1$ has acknowledged the generated DATA. Observe that the stages will only exit (enter) sleep after all the inputs have become DATA (NULL).

4.6 SCL Pipeline Initialization

Similar to NCL, each pipeline stage in an SCL system must be initialized to a specific state to function correctly. A global reset signal is used to force the components of each pipeline stage into the desired initial state. The registers in each SCL pipeline stage can be initialized to a NULL

or DATA state. The initialization overhead for the reset-to-NULL pipeline stages is low because only the completion final gates need to be initialized; by initializing the completion final gates to a logical 1, the registers and threshold gates in the stage will be forced to sleep upon reset, which cause the pipeline stage to generate a NULL. However, the reset-to-DATA pipeline stages require that each of the registers in the stage be initialized to DATA0 or DATA1. In order for the DATA to be able to propagate through the threshold gates of a reset-to-DATA pipeline stage, the completion final gate must be initialized to a logical 0. It is possible to initialize adjacent pipeline stages to NULL; however, it's not possible to initialize adjacent pipeline stages to DATA. If two adjacent pipeline stages are initialized to DATA, the first DATA will attempt to overwrite the second DATA. The two DATA wavefronts will become joined in a single DATA wavefront since there is not a NULL wavefront to separate them.

4.7 SCL Performance and Timing Assumptions

It is important to determine the analytical cycle time of an SCL pipeline as well as the relative timing assumptions (RTAs) needed to guarantee correct operation [34]. In order to analyze the performance and timing assumptions of SCL we have to consider how multiple pipeline stages interact. The interaction between pipeline stages can be analyzed by observing how a DATA propagates through an initially empty pipeline. Consider the generic three stage SCL pipeline illustrated in Figure 16. Assume all of the pipeline stages are initialized to NULL, which means each C_i is initialized to logical 1. The input to the first stage, X, is driven by an ideal source that can generate a DATA immediately after C_1 is asserted and generate a NULL immediately after C_1 is de-asserted. A marked graph model of the pipeline is given in Figure 18, and the timing

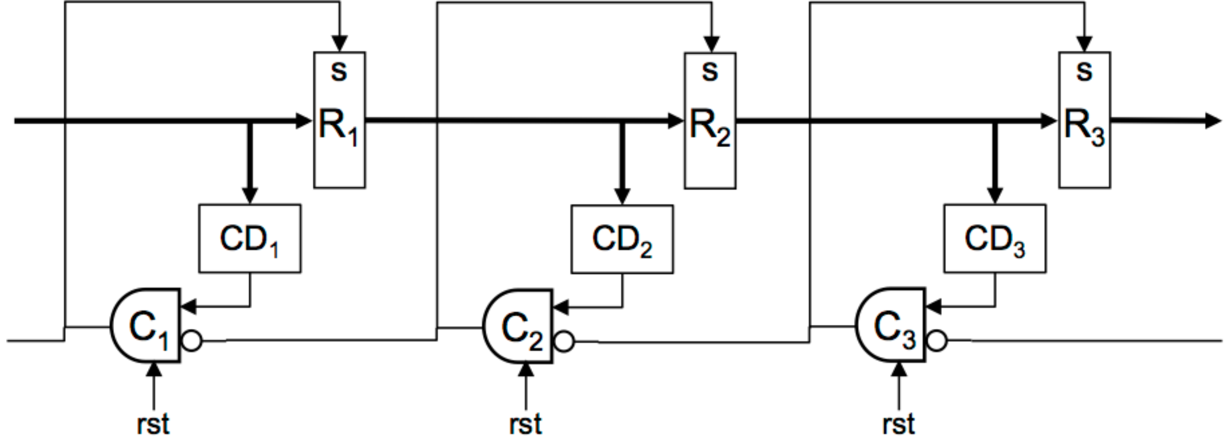

Figure 16: SCL linear pipeline with no combinational logic.

waveforms are given in Figure 20.

The operation for the i-th SCL pipeline stage can be summarized as follows. When a DATA reaches the input of the stage it causes the output of the completion detector to be de-asserted ($CD_i \downarrow$). Once stage $i+1$ has entered the sleep state, stage i can exit the sleep state, simultaneously acknowledging the DATA from its predecessor and starting the evaluation phase ($C_i \downarrow$). The evaluation phase begins with the register latching the DATA ($R_i \uparrow$). After stage $i-1$ has generated NULL, causing ($CD_i \uparrow$), and stage $i+1$ has acknowledged the DATA generated by stage i ($C_{i+1} \downarrow$), stage i can enter the sleep state, simultaneously acknowledging the NULL from its predecessor and starting the reset phase ($C_i \uparrow$). As stage i enters the reset phase, the register is reset to NULL ($R_i \downarrow$). Due to the acknowledgment of DATA ($C_i \downarrow$) before the DATA is actually latched ($R_i \uparrow$) there is a race condition, as illustrated in Figures 19 and 21. The proceeding stage must maintain the DATA long enough for the register to be able to latch it.

$$R_i \uparrow \prec R_{i-1} \downarrow \tag{2}$$

The relative timing assumption between two adjacent stages can be expressed as

$$T_{R_i,DATA} < T_{C_{i-1}} + T_{R_{i-1},NULL} \tag{3}$$

where $T_{R_i,DATA}$ ($T_{R_i,NULL}$) is the delay for register R_i to propagate DATA (NULL) upon de-assertion (assertion) of sleep. If RTA 2 is not satisfied for all pairs of adjacent pipeline stages, DATA can be lost. The forward latency of stage i ($T_{latency_i}$) is

$$T_{latency_i} = T_{CD_i} + T_{C_i} + T_{R_i} \tag{4}$$

The cycle time of the pipeline can be derived from either the marked graph model or the timing waveforms. The critical cycle of events for the pipeline is

$$C_1 \downarrow, R_1 \uparrow, CD_2 \downarrow, C_2 \downarrow, R_2 \uparrow, CD_3 \downarrow, C_3 \downarrow, C_2 \downarrow \tag{5}$$

It can be observed from the marked graph model that a symmetric critical loop exists that involves the registers transitioning to NULL. However, due to the design of the SCL registers and threshold gates, the delay for propagating DATA is significantly larger than the delay for transitioning to NULL. Therefore, the cycle time (T_{cycle}) for the pipeline is given by

$$T_{cycle} = 4 * T_{C-element} + 2 * T_{CD} + 2 * T_{R,DATA} \tag{6}$$

The discussion has focused on a simple SCL pipeline, however a more complete pipeline is

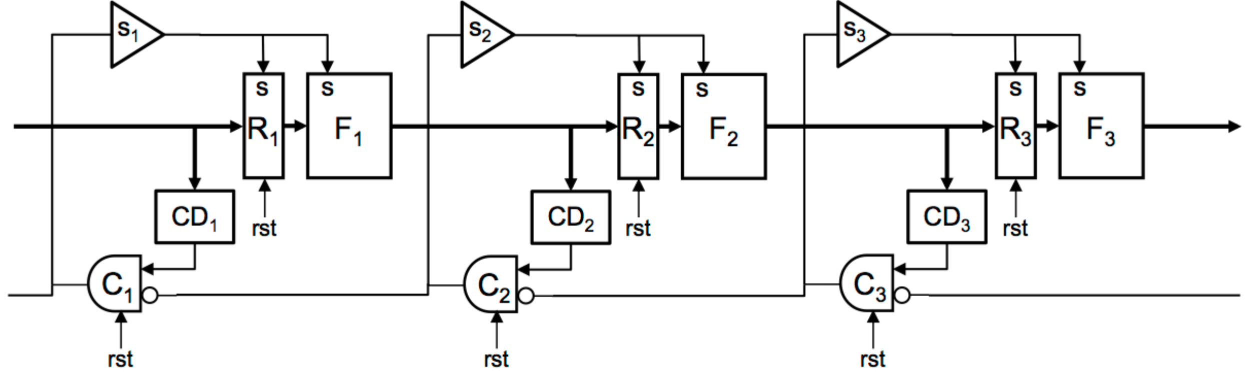

Figure 17: SCL pipeline with sleep buffers.

illustrated in Figure 17. This SCL pipeline has two additional components, functional blocks (F_i) and sleep buffers (s_i). Due to the combined capacitance presented by the sleep pins of registers and function blocks, sleep buffers are often needed for each pipeline stage. These sleep buffers introduce additional delay that must be considered. While RTA 2 is still valid, RTA 3 becomes

$$T_{s_i} + T_{R_i,DATA} < T_{C_{i-1}} + T_{s_{i-1}} + T_{R_{i-1},NULL} \tag{7}$$

In the previous pipeline, CD_{i+1} will directly monitor when R_i transitions to NULL. While NULL is strictly propagated through the threshold gates in NCL, NULL is generated by a sleep signal in SCL. Often, the function blocks contain multiple levels of threshold gates. The threshold gates in a multi-level function block, F_i, can be partitioned into the set of final gates that drive the next stage, F_{F_i}, and the set of remaining internal gates, F_{I_i}. In SCL pipelines, only the threshold gates in the final level of logic F_{F_i}, can be directly observed by CD_{i+1}. As a result, the set of internal gates F_{I_i} are unobservable. Consider the scenario where a single threshold gate suffers from a slower then expected transition to logical 0. Assume the slow threshold gate is in the first

level of a multi-level function block, F_i. If DATA(t) causes the slow gate to transition to logical 1 and the gate remains logical 1 during the subsequent evaluation phase of DATA(t+1), F_i can produce an incorrect result. This is possible because during the reset phase of stage i, CD_{i+1} is unable to determine that the slow gate has not yet transitioned back to logical 0. Thus, the addition of any level of logic beyond the registers results in a race condition

$$(g_k \downarrow) \in F_{I_i} \prec C_i \downarrow \tag{8}$$

which states that every internal threshold gate in function block F_i should transition back to logical 0 before the next evaluation phase of stage i begins. In order to place timing bounds on this RTA we need to determine how quickly stage i, once the reset phase has begun, can begin the next evaluation phase. The slower of two events will cause stage i to begin the next evaluation phase, the acknowledgement of NULL by $C_{i+1} \uparrow$ or the detection of DATA by $CD_i \downarrow$. The delay of the fastest path from $C_i \uparrow$, to $C_{i+1} \uparrow$ is defined as $\min(T_{C_i\uparrow, C_{i+1}\uparrow})$. The delay of the fastest path from $C_i \uparrow$, to $CD_i \downarrow$ is defined as $\min(T_{C_i\uparrow, CD_i\downarrow})$. The delay of the slowest path from $C_i \uparrow$, to $g_k \downarrow$ is defined as $\max(T_{C_i\uparrow, g_k\downarrow})$. Therefore, RTA 8 can be expressed as

$$\max(T_{C_i\uparrow, g_k\downarrow}) < \max(\min(T_{C_i\uparrow, C_{i+1}\uparrow}, \min(T_{C_i\uparrow, CD_i\downarrow})) \tag{9}$$

which must be satisfied for each gate, g_k, in F_{I_i}. Observe that $\min(T_{C_i\uparrow, CD_i\downarrow})$ can be increased by decreasing the rate at which DATA is inserted into the pipeline. By artificially slowing down the rate that DATA is inserted into an SCL pipeline, RTA 9 can be satisfied, just as the clock period can

be increased to satisfy setup constraints in a synchronous design. Now that a pipeline with function blocks is being analyzed, the forward latency and cycle time must be revisited. The forward latency of stage i becomes

$$T_{latency_i} = T_{CD_i} + T_{C_i} + T_{s_i} + T_{R_i} + T_{F_i} \quad (10)$$

The delay for the evaluation phase of pipeline stage i, which begins after $C_i \downarrow$, can be expressed as

$$T_{eval_i} = T_{s_i} + T_{R_i,DATA} + T_{F_i,DATA} \quad (11)$$

where T_{s_i} is the delay through the sleep buffer, s_i, and $T_{F_i,DATA}$ is the delay through the function block, F_i. Conversely, the delay for the reset phase of pipeline stage i, which begins after $C_i \uparrow$, can be expressed as

$$T_{reset_i} = T_{s_i} + T_{F_i,NULL} \quad (12)$$

where $T_{F_i,NULL}$ is the delay of the threshold gates in F_i, which directly drive CD_{i+1}. Note that while T_{eval_i} is a function of the delay through the whole function block, F_i, T_{reset_i} is only a function of the delay of the final threshold gates in F_i. The complete SCL cycle time can be written as

$$T_{cycle} = 4 * T_{C-element} + 2 * T_{eval} + 2 * T_{CD} \quad (13)$$

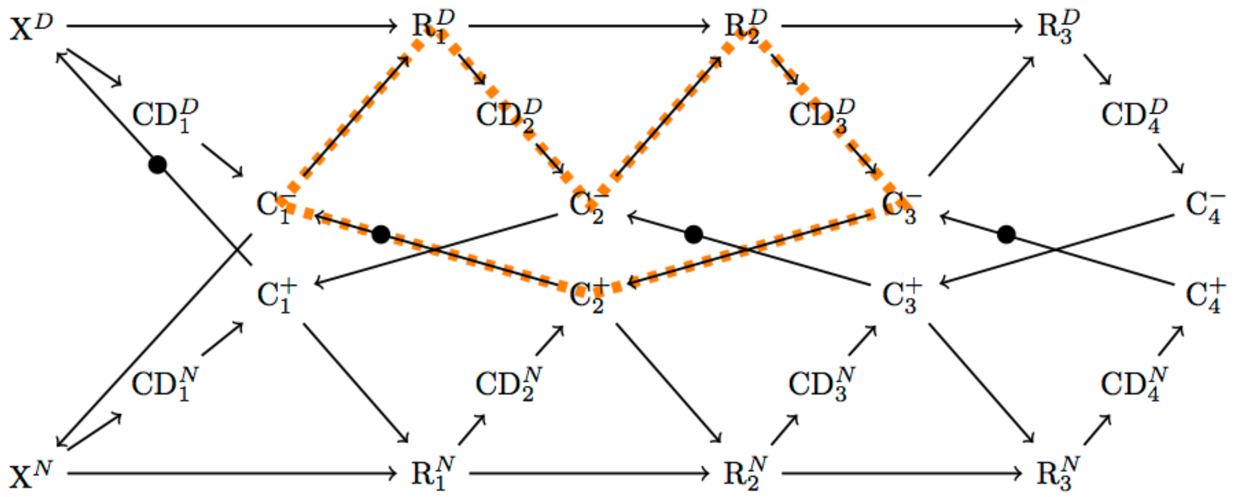

Figure 18: Marked graph model of SCL pipeline.

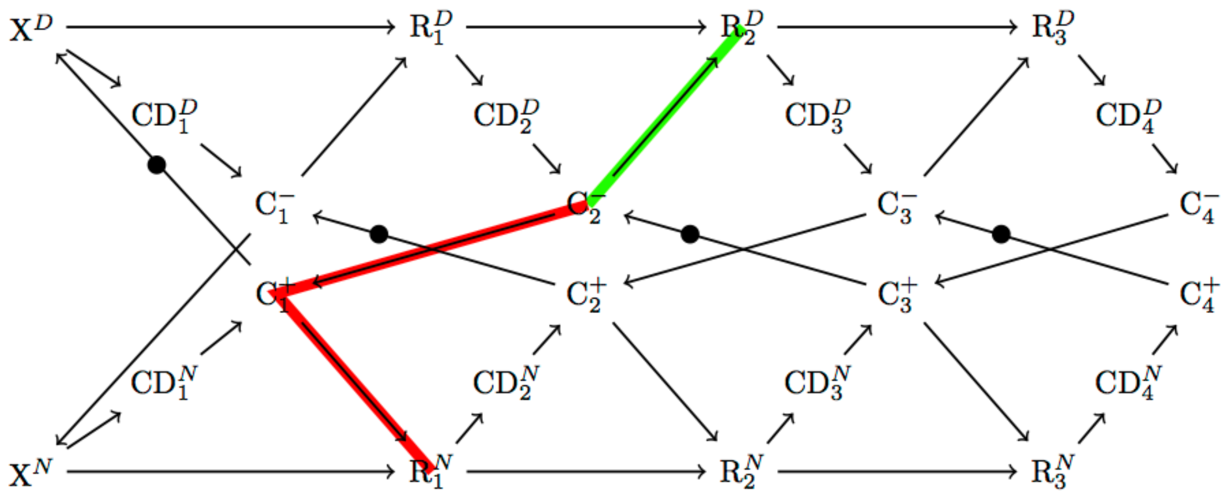

Figure 19: Marked graph model of SCL pipeline with race paths indicated by thick lines.

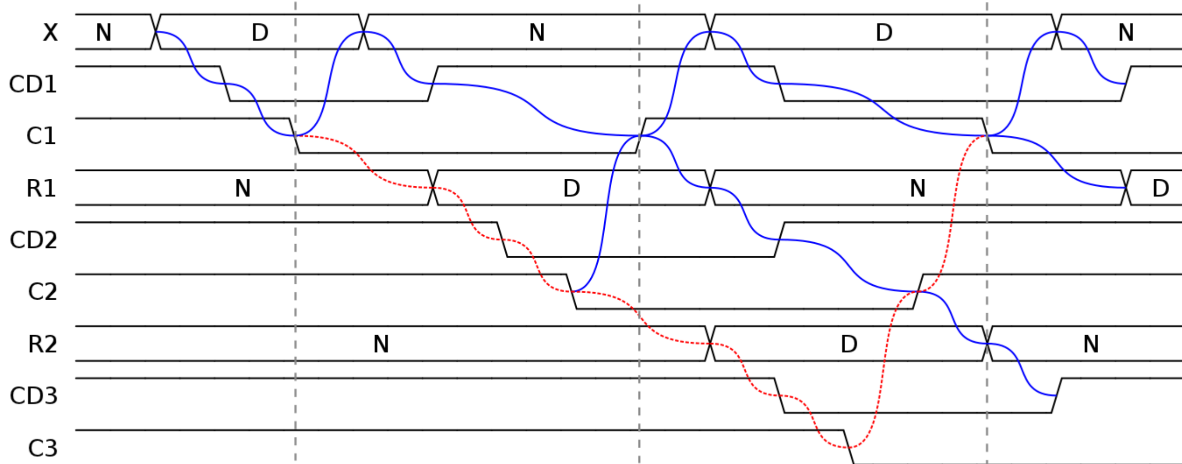

Figure 20: Signals for SCL pipeline with critical cycle indicated by dotted path.

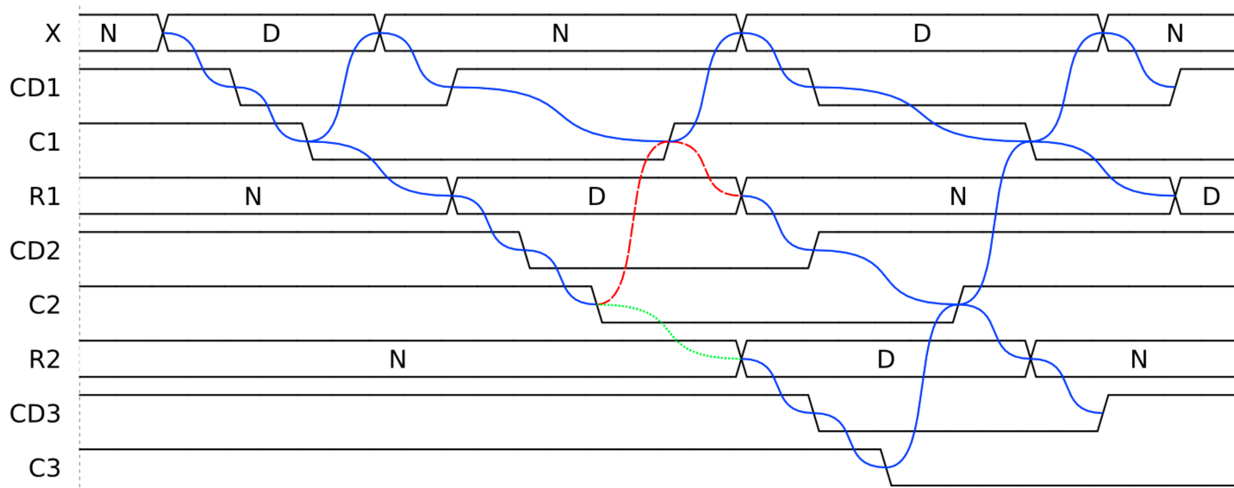

Figure 21: Signals for SCL pipeline with race indicated by dashed and dotted paths.

5 SYNCHRONOUS TO SCL CONVERSION

As most digital systems designed today utilize flip-flops this dissertation will focus on translating flip-flop based synchronous blocks to an equivalent SCL block. The synchronous block is assumed to utilize a single clock, and every flip-flop is assumed to operate on the same active edge. It is important to first discuss how equivalence between the synchronous and SCL block is defined.

5.1 Synchronous and SCL Equivalence

In this work a synchronous circuit and its translated SCL circuit are considered to be equivalent if the two circuits are observationally equivalent. Two systems are said to be observationally equivalent if an external agent cannot differentiate them by comparing their observable traces [4]. The synchronous and SCL circuits have a sequencing event that signals the validity of data between one pipeline stage and the next. The sequencing event for a synchronous circuit is defined as the active clock edge and the sequencing event for an SCL circuit is defined as the transition of a 1-of-M encoded signal from NULL to DATA. The observable trace for SCL circuits can be obtained by simply removing the NULL wavefronts generated at the outputs. In other words, given the same input vector, the values clocked out of the synchronous block must be identical to the DATA values generated by the SCL block. Consider the simple two-stage synchronous block illustrated in Figure 22. The timing behavior of the synchronous block is illustrated in Figure 23. Given an input vector of bits, **I**, one bit will be consumed at the input of the synchronous block and one bit will be generated at the output of the synchronous block at each sequencing event. As shown in

Figure 23, an input vector $\mathbf{I} = \{I_0, I_1\}$ results in an output vector $\mathbf{O_{sync}} = \{0, 0, I_0, I_1\}$. The first two elements in $\mathbf{O_{sync}}$ are the values initialized in the first and second flip-flop. If each flip-flip in the synchronous design were to be substituted for a reset-to-NULL SCL register, the resulting SCL pipeline would be that in Figure 24. The timing behavior of the SCL pipeline is show in Figure 25. The flow of DATA wavefronts through the SCL block is straightforward since the SCL pipeline acts as a FIFO: the i-th DATA inserted into the block is the i-th DATA generated at the output of the block. Therefore, the same input vector, $\mathbf{I}$, results in an output vector $\mathbf{O_{SCL}} = \{I_0, I_1\}$. As $\mathbf{O_{sync}}$ is not equal to $\mathbf{O_{SCL}}$ the proposed SCL block in Figure 24 is not equivalent to the synchronous block in Figure 22. In order to create an equivalent SCL block we must emulate the values that are initialized in the synchronous block's flip-flops. This can be accomplished by replacing the original flip-flops in the synchronous block with an equivalent reset-to-DATA register in the SCL block. Since the flip-flops in Figure 22 are initialized to a logic 0 we must replace them with SCL registers that are initialized to DATA0. As discussed in Section 4.6, pipeline stages that are initialized to DATA cannot be adjacent to other pipeline stages that are initialized to DATA. As a result, we must insert an additional reset-to-NULL register between the two reset-to-DATA registers. The resulting pipeline is shown in Figure 26. The SCL block in Figure 26 is now said to be equivalent, since $\mathbf{O_{sync}} = \mathbf{O_{SCL}}$. The resulting SCL block has three pipeline stages, which is the minimum number of pipeline stages required for the SCL block to be equivalent to the synchronous block in Figure 22.

In bundled-delay asynchronous circuits, flip-flop-based synchronous designs can be translated into latch-based asynchronous designs via the de-synchronization method. The de-synchronization

method proposed splitting each flip-flop in the synchronous design into a pair of latches, one of which is initialized to DATA [5]. In the case of the two-stage synchronous design presented earlier, and any linear pipeline, it would be sufficient to replace each flip-flop in the synchronous design with a reset-to-NULL followed by a reset-to-DATA register; however, this mapping is insufficient for synchronous circuits with feedback. Consider the simple finite-state machine (FSM) in Figure 28. If each flip-flop is replaced by a reset-to-NULL and reset-to-DATA SCL register, the resulting design will contain a data-path cycle consisting of only two pipeline stages. Any data-path loop in a SCL pipeline must contain at least three pipeline stages or the pipeline will dead-lock [32]. Therefore, an additional reset-to-NULL register must be inserted into the feedback path. One observation from this example is to simply replace each flip-flop in the synchronous design with a triplet of SCL registers with the middle register being reset-to-DATA. While this scheme is simple and results in an SCL design that is equivalent to the synchronous design, it may insert unnecessary registers. To reduce register overhead, the method proposed in this work inserts a third additional reset-to-NULL SCL register only on feedback paths.

5.2 Extracting Connectivity Information from Netlists

In this work, a netlist-graph is a directed graph $G_n = (V, E)$, where V is the set of nodes in the netlist and E is the set of directed edges connecting the cells. Here, $V = PI \cup PO \cup CC \cup SC$, where PI is the set of primary inputs, PO is the set of primary outputs, CC is the set of combinational cells and SC is the set of sequential cells. The four sets PI, PO, CC, and SC are mutually disjoint. A path $p_{i,j}$ is defined as a sequence of edges in E, starting from node v_i and ending at node v_j. The set of all paths that exist in G is defined as P_G. The combinational transitive

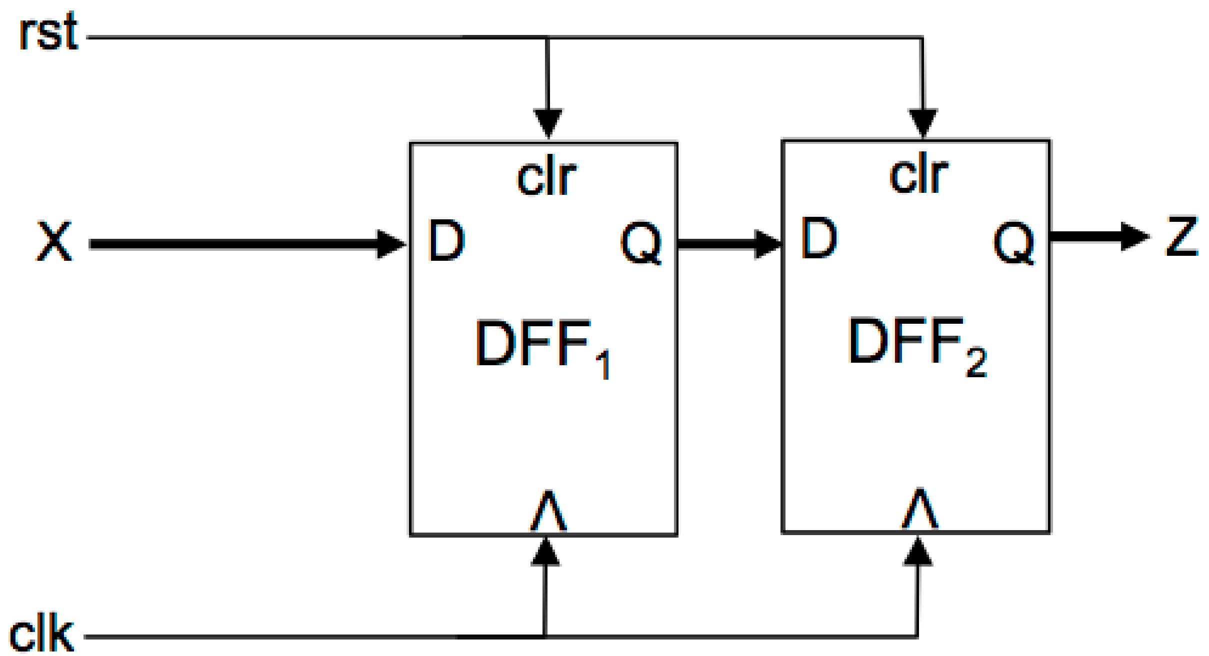

Figure 22: Synchronous design with flip-flops.

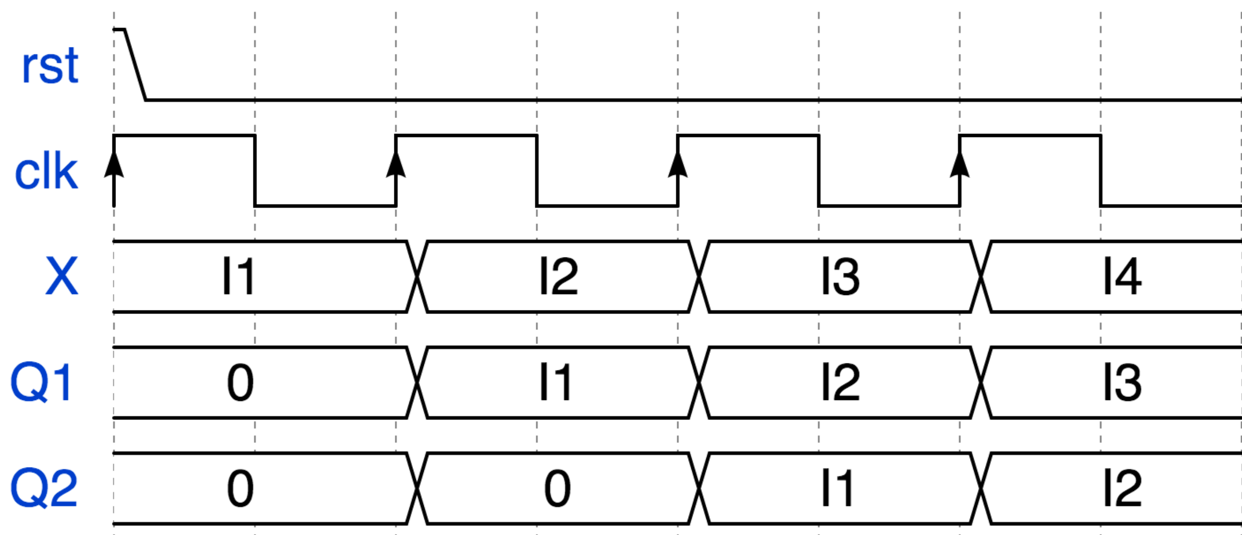

Figure 23: Output trace of synchronous pipeline.

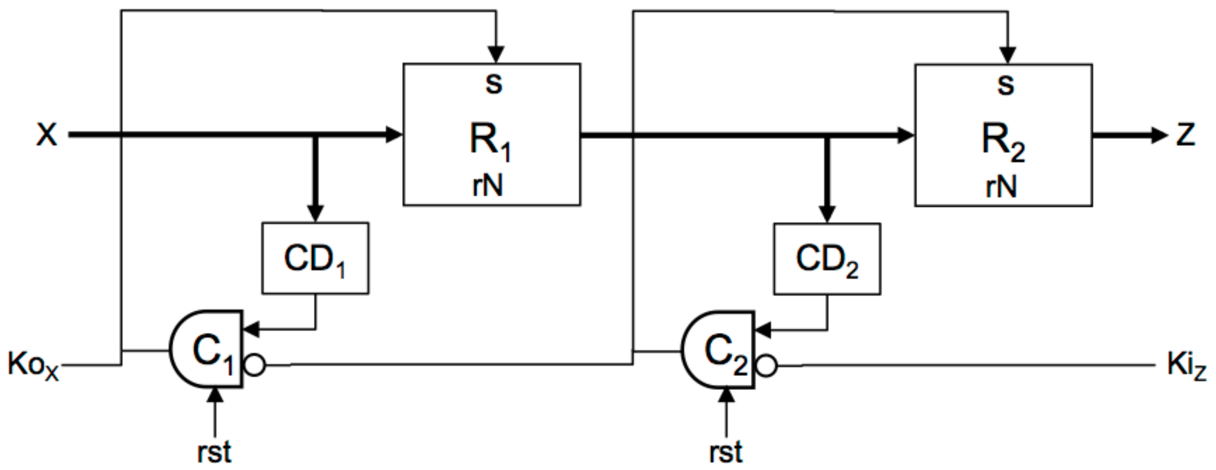

Figure 24: SCL pipeline translated from synchronous pipeline.

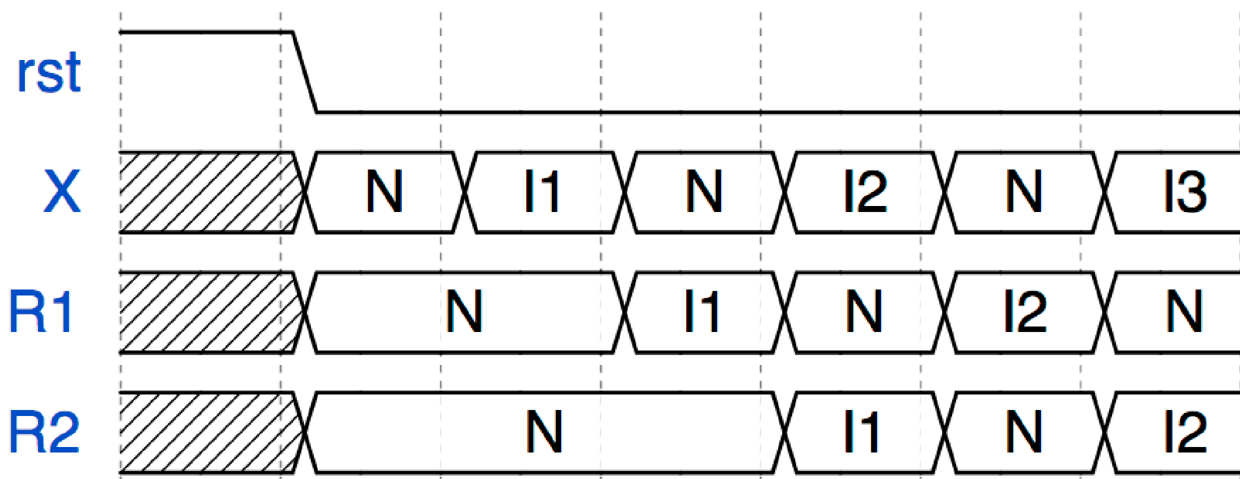

Figure 25: Output trace of 2-stage SCL pipeline.

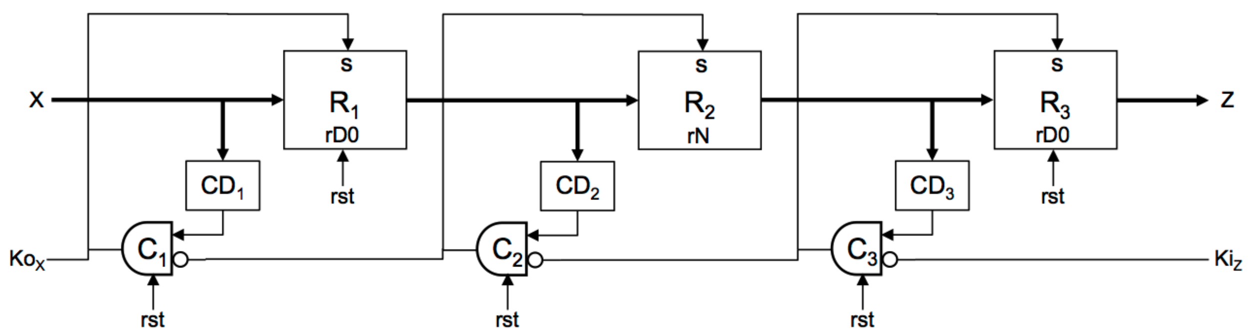

Figure 26: SCL pipeline translated from synchronous pipeline.

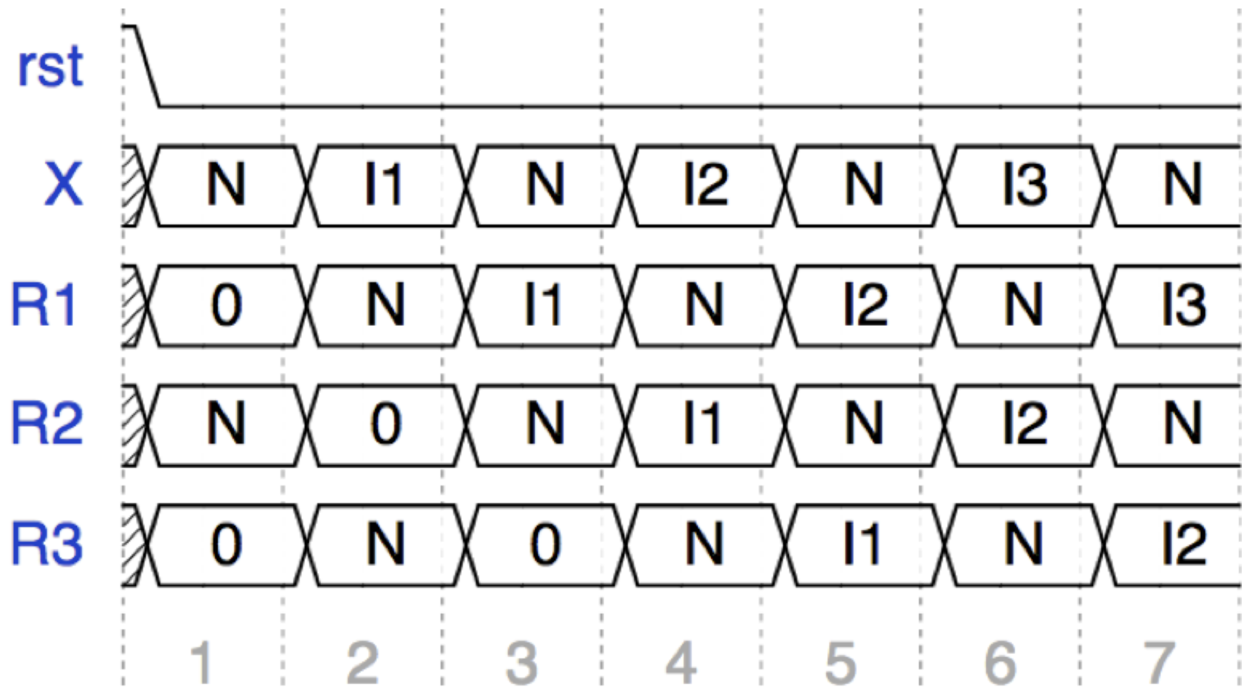

Figure 27: Output trace of 3-stage SCL pipeline.

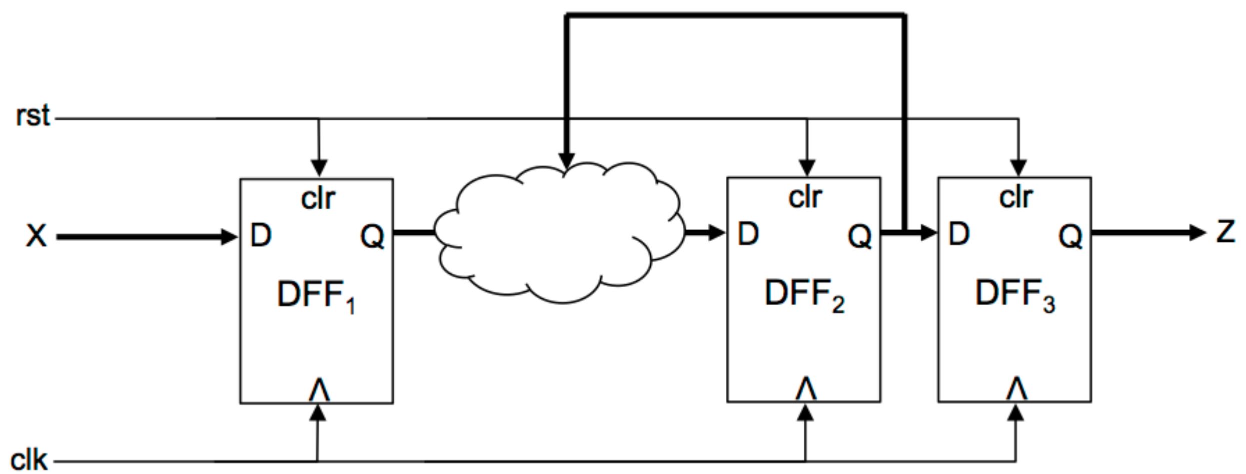

Figure 28: Synchronous pipeline with direct feedback on register.

fanout of a node, $CTFO(v_i)$, is defined as the set of nodes reachable from v_i through a path that does not go through a sequential cell. The combinational transitive fanin of a node, $CTFI(v_i)$, is defined as the set of nodes that can reach v_i through a path that does not go through a sequential cell. Note that a path that does not go through a sequential cell may begin and end with a sequential cell.

In this work a register-graph is a directed graph $G_r = (V, E)$. While the netlist-graph contains nodes that represent registers and gates, each node in the register-graph represents a single SCL pipeline stage. The function R is defined as $R : V \rightarrow \{v_i, v_j\}$, which maps a pipeline stage node to a set of registers. The function r is defined as $r : V \rightarrow \{0, 1\}$, which maps a pipeline stage node to 0 if the pipeline stage is reset-to-NULL and 1 if the pipeline stage is reset-to-DATA. An edge $e_{i,j} = (ps_i, ps_j)$ in G_r represents a combinational path from pipeline stage ps_i to pipeline stage ps_j. A register-graph is initially extracted from the synchronous netlist G_n.

The initial register-graph contains a vertex for every register in the synchronous netlist, which are all reset-to-DATA. Any datapath that forms a closed loop must contain at least three pipeline stages. In addition, two adjacent pipeline stages cannot be initialized to DATA. These two rules are satisfied by first inserting a reset-to-NULL register directly on the output of any reset-to-DATA register that has a combinational feedback path. The pipeline stage v_i can be easily determined to have a combinational feedback path from the register-graph by checking if the edge $e_{i,i}$ exists.

Since all datapath loops in synchronous designs must contain a flip-flop, this guarantees all loops consist of at least one reset-to-DATA stage and one reset-to-NULL stage. Now, a reset-

to-NULL register is inserted directly on the input of any reset-to-DATA register. Thus, a reset-to-NULL pipeline stage is guaranteed to be inserted between adjacent reset-to-DATA pipeline stages, and all loops now consist of at least one reset-to-DATA stage and two reset-to-NULL stages.

5.3 Determining Acknowledge and Sleep Networks

As discussed in Section 4.5, each SCL pipeline stage contains a completion final gate that receives an acknowledgment signal. A complete pipeline stage is shown in Figure 29. In this section the completion detectors and final C-elements will not be included in pipeline diagrams; the acknowledgement signal entering a pipeline stage, Ki, is assumed to be connected to the inverted input of the final C-element, and the acknowledge signal exiting the pipeline stage, Ko, is assumed to be the output of the final C-element, as shown in Figure 29.

The logic network that generates the acknowledgement signal that enters the pipeline stage is referred to as an *acknowledgement network*. Recall that each SCL register must belong to a single pipeline stage and the set of registers that belong to pipeline stage ps_i has been defined as $R(ps_i)$. Each SCL pipeline stage must receive a combined acknowledgment from all the pipeline stages it directly contributes to. The set of pipeline stages that are driven by stage ps_i is defined as $ACK(ps_i)$, which can be derived from the register-graph:

$$ACK(ps_i) = \{ps_j : \exists e_{i,j} \in E\} \tag{14}$$

Each SCL threshold gate must receive a sleep signal that indicates when the gate should enter and exit the sleep state. The logic network that generates this sleep signal is referred to as

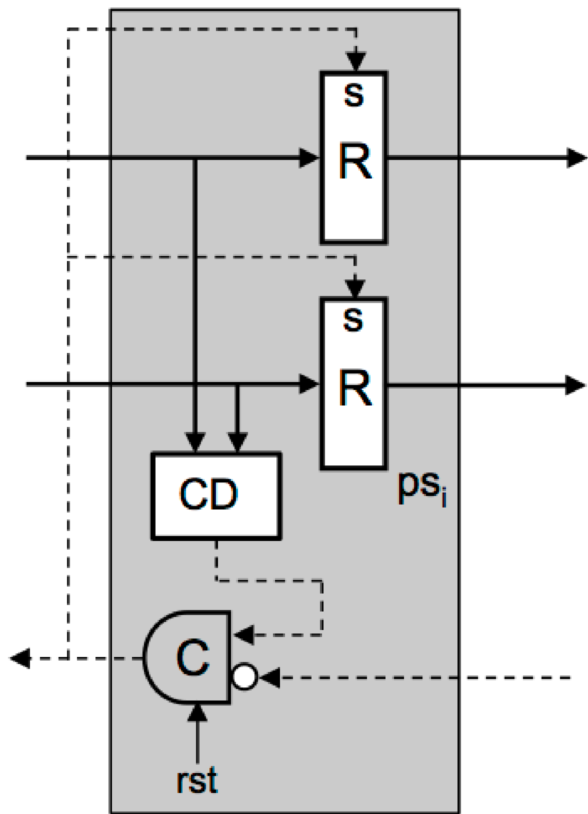

Figure 29: SCL pipeline stage.

a *sleep network*. A single SCL threshold gate may be driven by registers in one or more pipeline stages. The sleep network combines the acknowledge signals from all the pipeline stages that drive the gate. The set of pipeline stages that drive the threshold gate v_i is defined as $SLEEP(v_i)$:

$$SFI(v_i) = \{v_j : v_j \in CTFI(v_i) \land v_j \in SC\} \tag{15}$$

$$SLEEP(v_i) = \{R^{-1}(v_j) : v_j \in SFI(v_i)\} \tag{16}$$

where $R^{-1}(v_j)$ maps the register, v_i, to the pipeline stage, ps_i, it belongs to. If $SLEEP(v_i) = SLEEP(v_j)$, then SCL threshold gates v_i and v_j belong to the same sleep domain and share the same sleep network.

5.4 Determining Pipeline Stages

For a given design there are multiple ways to group registers into SCL pipeline stages. One approach is to group each register into a unique pipeline stage. This was the approach used in the Weaver project [29]; however, this results in a large area overhead for acknowledgement networks. Recall that each SCL pipeline stage must receive a combined acknowledgement signal, which satisfies expression 14, as well as generate a single acknowledgement output via the completion final gate. Consider the abstract datapath shown in Figure 30. Each register can be grouped into a separate pipeline stage, resulting in five pipeline stages, shown in Figure 31. This approach results in the following acknowledge and sleep networks, illustrated in Figures 32 and 33, respectively. Note that each of the gates, v_2, v_3 and v_4 belong to a different sleep domain.

$$ACK(ps_0) = \{ps_2, ps_3\}$$

$$ACK(ps_1) = \{ps_3, ps_4\}$$

$$SLEEP(v_2) = \{ps_0\}$$

$$SLEEP(v_3) = \{ps_0, ps_1\}$$

$$SLEEP(v_4) = \{ps_1\}$$

One alternative approach is to merge the first pair of registers into a single pipeline stage, such that $R(ps_0) = \{v_0, v_1\}$. The new acknowledge networks can be derived from the acknowledge networks in the first approach. This merged approach results in the following acknowledge and sleep networks, illustrated in Figures 34 and 35, respectively.

$$ACK_{merged}(ps_0) = ACK(ps_0) \cup ACK(ps_1) = \{ps_2, ps_3, ps_4\}$$

$$SLEEP_{merged}(v_2) = SLEEP_{merged}(v_3) = SLEEP_{merged}(v_4) = \{ps_0\}$$

The merged approach generally results in less area overhead because as the number of pipeline stages are reduced, the number of acknowledge and sleep networks are also reduced. In NCL, the first approach is typically preferred when performance is critical because while there are more acknowledge networks, each acknowledge network is smaller, which generally results in less delay. However, in SCL, increasing the number of pipeline stages that drive a single threshold gate results in a larger sleep network, which generally increases delay.

5.5 Combining Pipeline Stages

The SCL pipeline stages can be iteratively merged to reduce the number of pipeline stages, which reduces the overhead discussed in the previous section. Pairs of pipeline stages that share a common driven pipeline stage are considered for merging, similar to [27]. For initialization

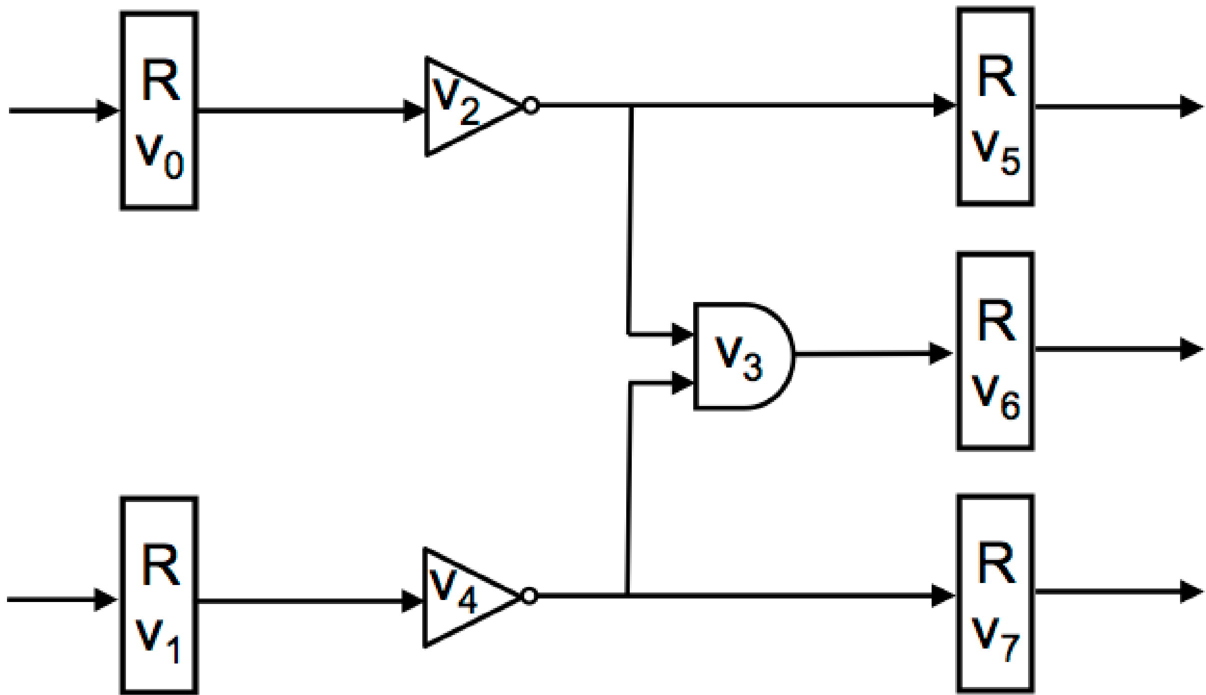

Figure 30: Abstract SCL datapath.

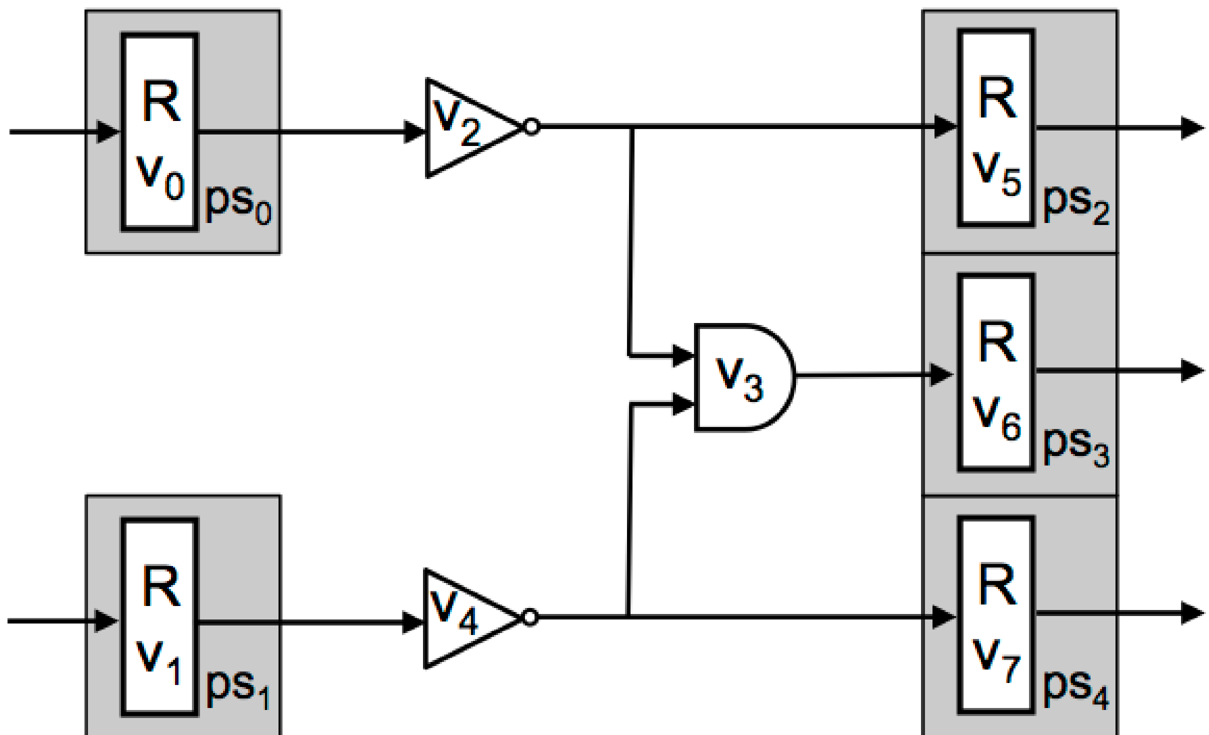

Figure 31: Abstract data path with each register partitioned into unique pipeline stages.

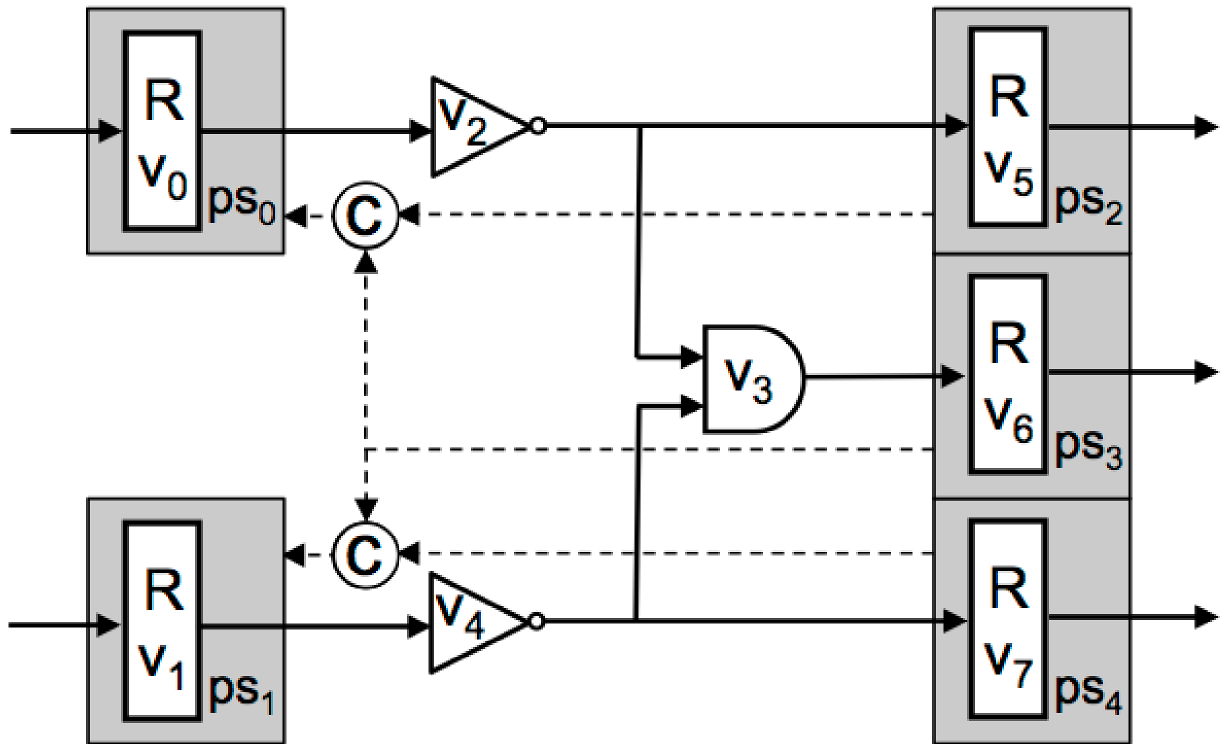

Figure 32: Acknowledgement network for partitioning in Figure 31

purposes, reset-to-DATA and reset-to-NULL pipeline stages are not merged together. Similarly, merges that would result in a pipeline stage driving a combination of both reset-to-DATA and reset-to-NULL stages are not allowed. In addition to these initialization-related merges, it is important to prevent other merges that can result in dead-lock. Consider the pipeline configuration in Figure 36. Register v_8 is reset-to-DATA while registers v_0 and v_1 are reset-to-NULL. The pipeline stages ps_0 and ps_1 both drive the pipeline stage ps_2, thus they are candidates for being merged. However, the merging of pipeline stage ps_0 and ps_1 results in the configuration shown in Figure 37, which results in dead-lock. There are two issues illustrated in this example. While the pre-merged configuration had a cycle of three pipeline stages (ps_0, ps_1, ps_3), the merged configuration has a cycle of only two pipeline stages (ps_0, ps_3). This merge can be prevented by not allowing merges that result in a stage driven by the merged stage to also drive the merged stage. The second issue is illustrated

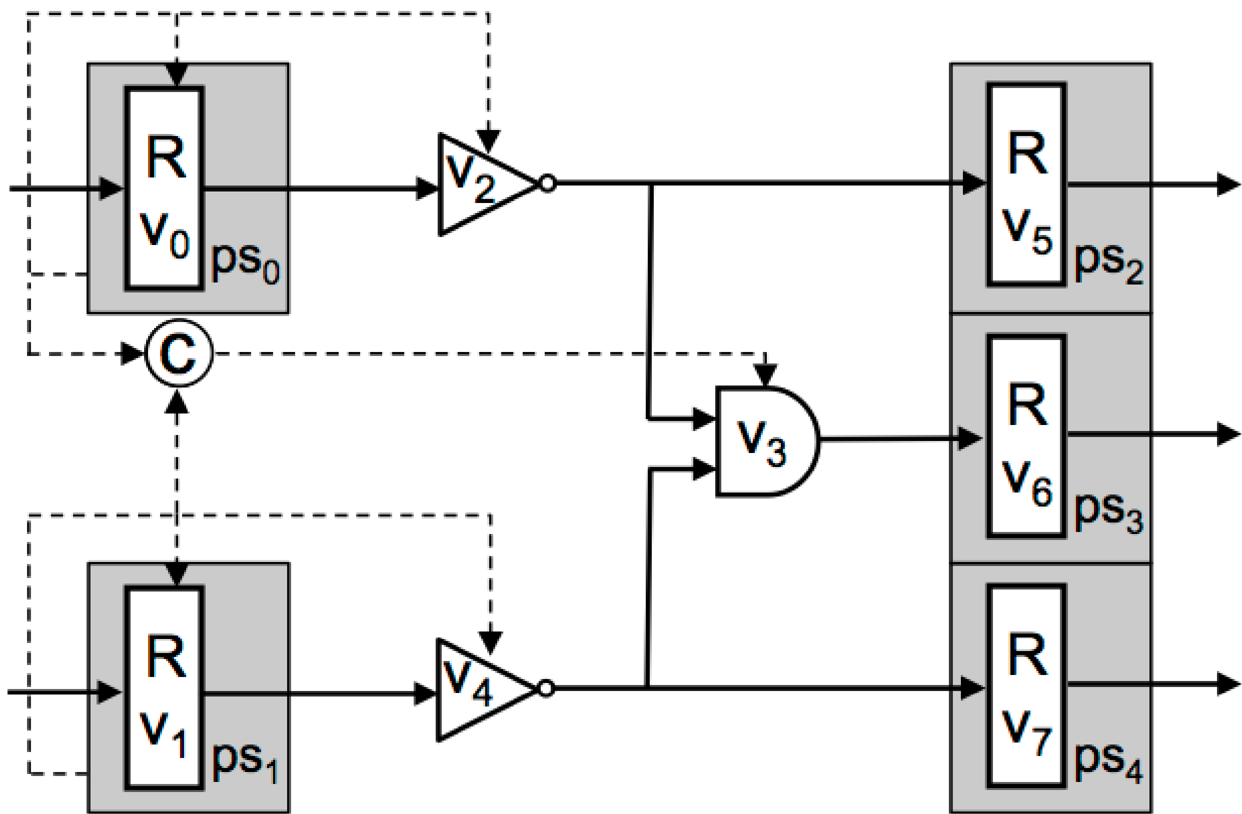

Figure 33: Sleep networks for partitioning in Figure 31

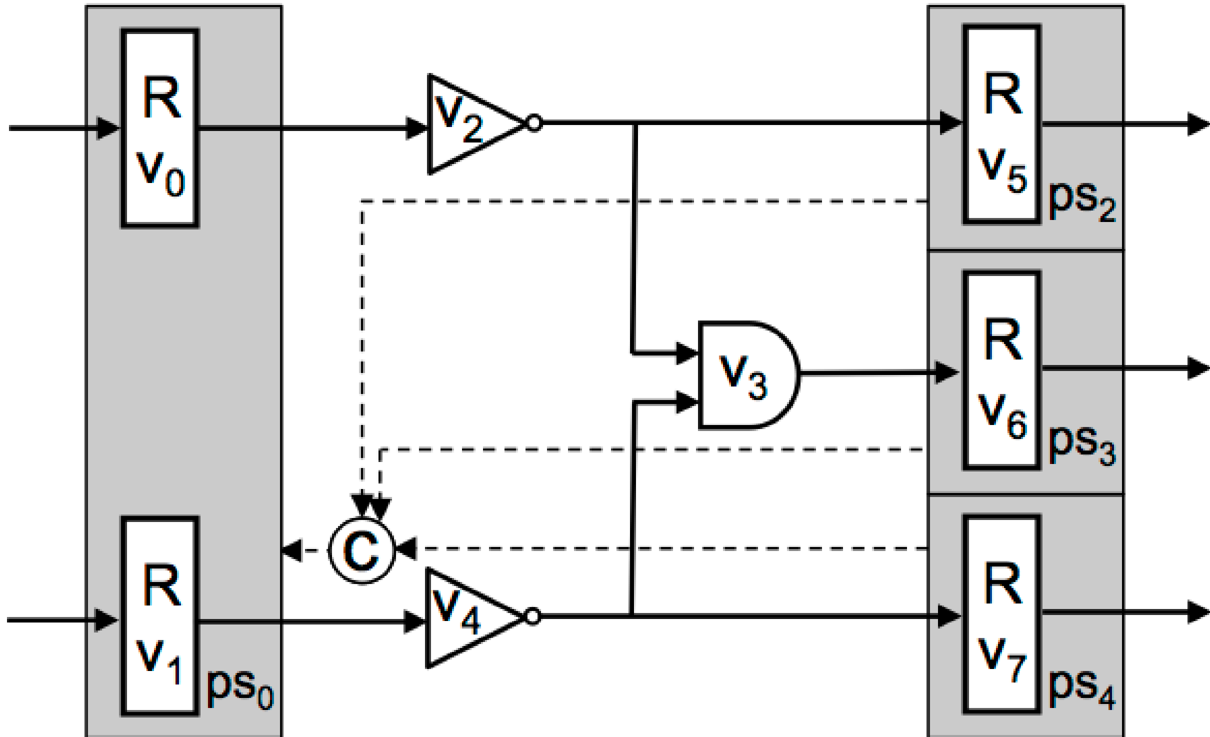

Figure 34: Acknowledgement network for merged partitioning.

in the sleep-network for the merged-configuration, shown in Figure 37. Observe that registers v_0 and v_1 will exit sleep when the stage ps_0 acknowledges DATA. However, both registers in the stage can only receive DATA after v_2 has exited sleep and propagated DATA, resulting in a cyclic dependency. This condition is avoided by not merging any two stages if a combinational path

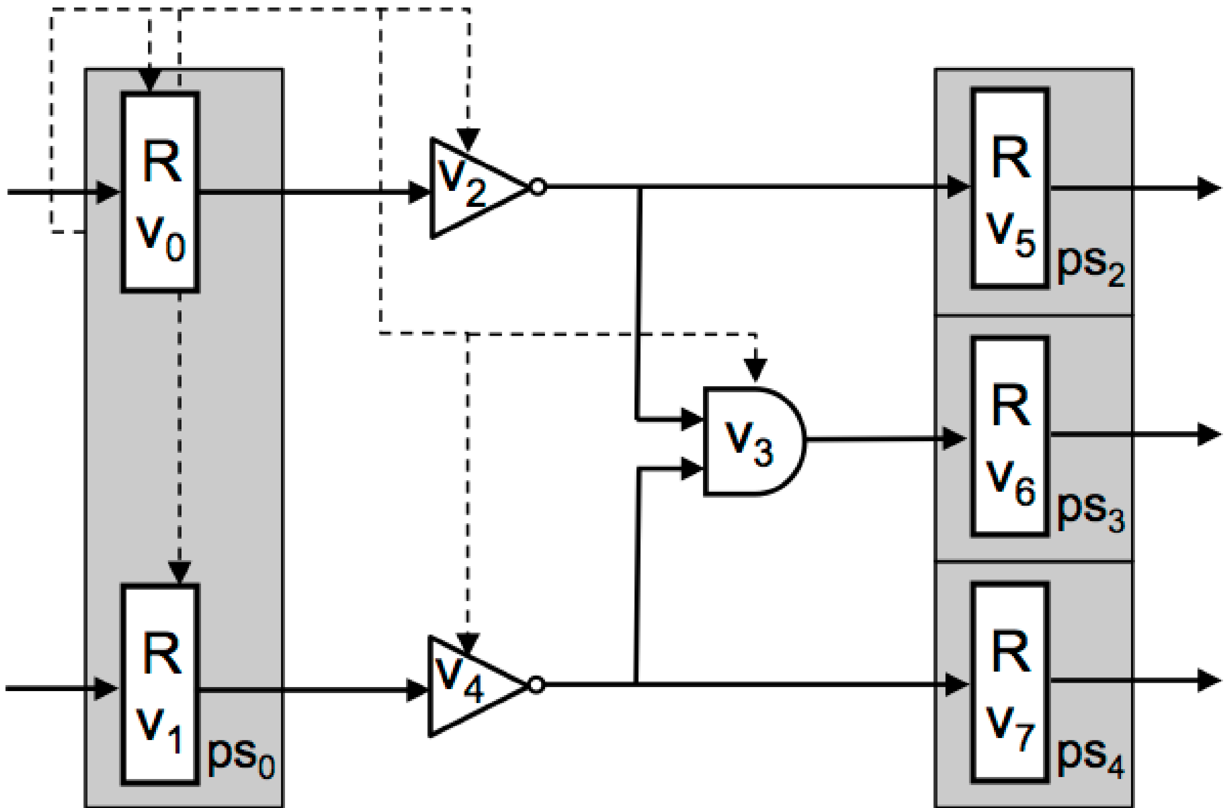

Figure 35: Sleep networks for merged partitioning.

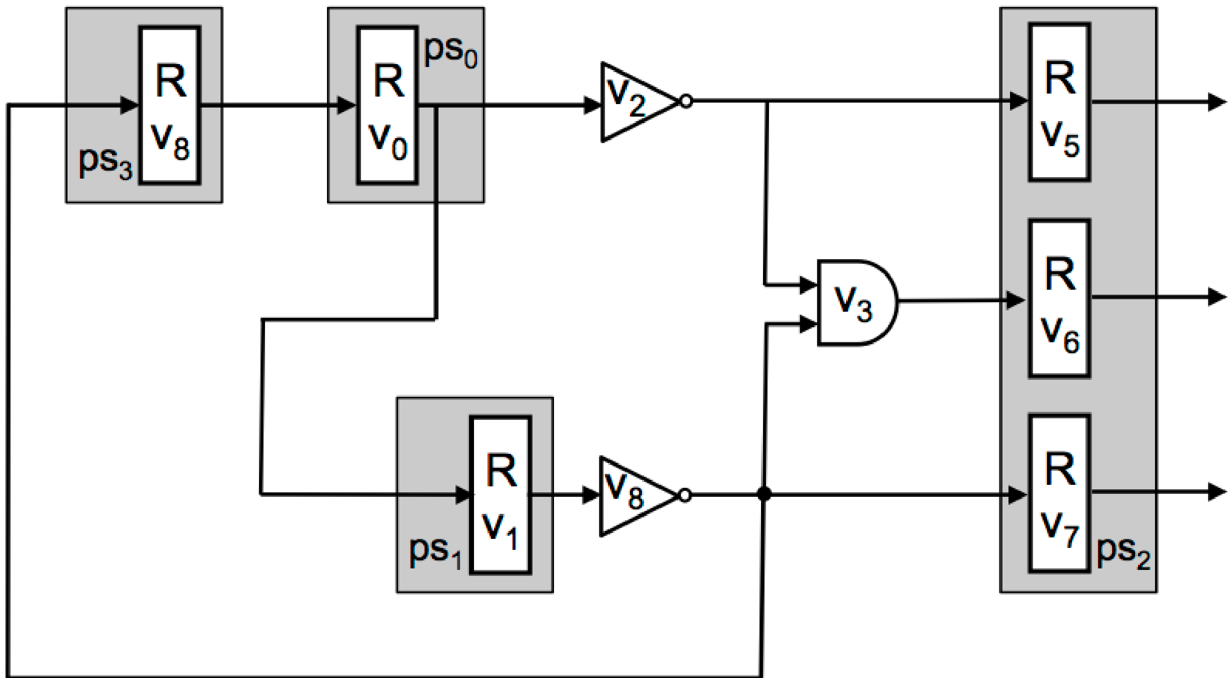

Figure 36: Abstract SCL pipeline with datapath loop.

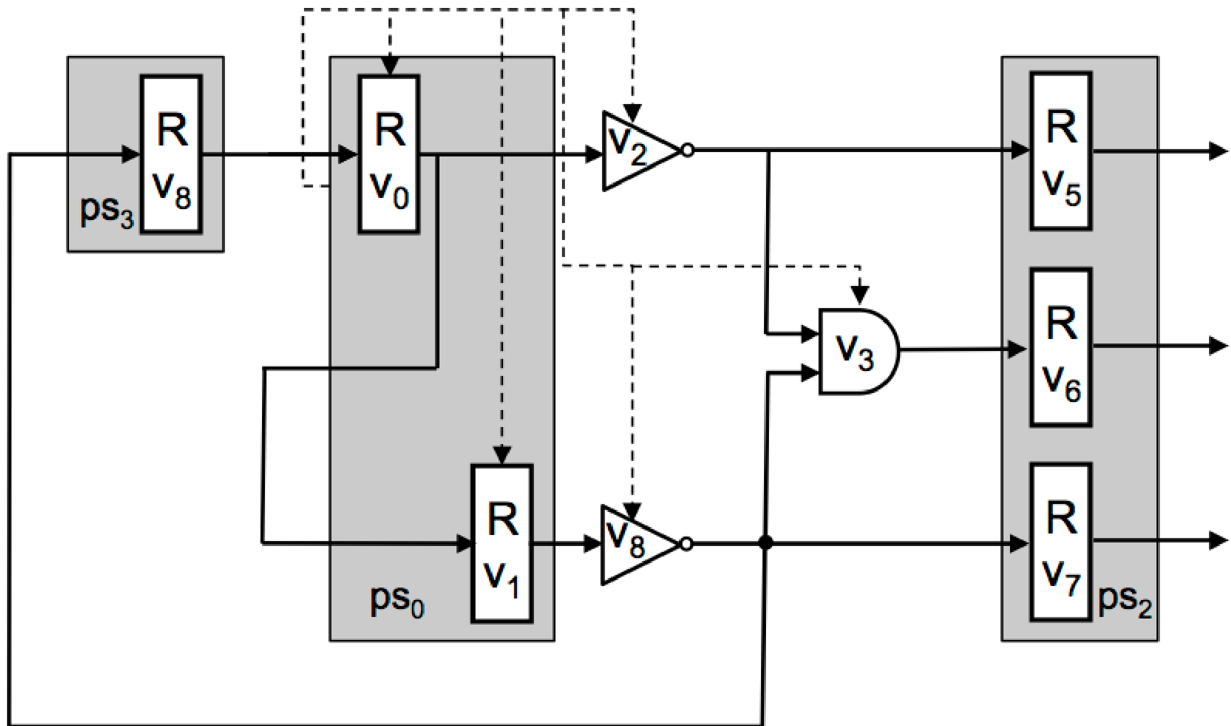

Figure 37: Sleep networks for abstract data path in Figure 36.

exists between the stages.

6 SCL Performance Analysis

Typically, simulation is used to determine the performance of an asynchronous circuit. However, the data-dependent delays make it difficult to determine worst-case performance using simulation. In synchronous designs, determining the performance of a synthesized netlist is straightforward; static timing analysis is used to determine the critical path and the maximum clock frequency. Only analyzing the paths between adjacent registers is needed to determine the critical path for synchronous circuits. However, determining the critical path of an asynchronous design is more difficult because the critical path may be through as few as two adjacent registers or as many as every register in the design. Using Petri nets is a common way of modeling the performance of asynchronous circuits, and can be used to determine the critical path, and hence, the worst-case performance.

The MG representation of an SCL circuit will be similar to a Signal Transition Graph (STG). In a STG each signal is represented by two transitions, one to represent the rising of the signal and one to represent the falling of the signal. The SCL MG will be more abstract, modeling the SCL circuit at the pipeline stage level. The components of the SCL pipeline stages in Figure 17 will be represented by a pair of transitions. Thus, each SCL pipeline stage i will be represented by eight transitions: $R_i^D, R_i^N, F_i^D, F_i^N, CD_i^D, CD_i^N, C_i^0, C_i^1$. Each transition is annotated with the delay expected from its circuit counterpart. To complete the SCL MG model, an ideal source and sink must be added. The ideal source provides a DATA/NULL token as soon as the input register requests DATA/NULL. The ideal sink requests DATA/NULL as soon as the output register generates a NULL/DATA token. A complete MG of a three-stage, reset-to-NULL SCL pipeline is

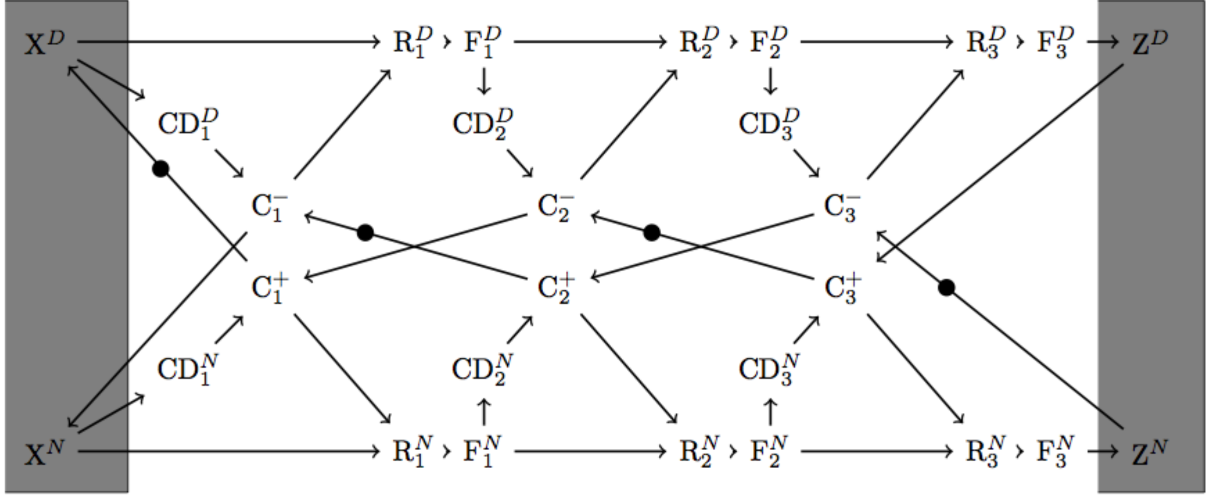

Figure 38: A three stage SCL pipeline represented as a MG.

shown in Figure 38.

A MG can be extracted from the register-graph discussed in Section 5.2. However, it is vital to initialize the tokens in the MG correctly to get an accurate model of the SCL pipeline. A MG for a three-stage SCL pipeline, which has the second stage, reset-to-DATA, is shown in Figure 39. If stage i is reset-to-NULL and drives a reset-to-NULL stage $i+1$, a token is initialized in the place between transitions C_{i+1}^{+} and C_i^{-}. If stage i is reset-to-NULL and drives a reset-to-DATA stage $i+1$, tokens are inserted on all places in the postset of transition F_i^N. Lastly, if a reset-to-DATA stage i drives a reset-to-NULL stage $i+1$, tokens are inserted on all places in the postset of transition F_i^D.

Linear SCL pipelines are not very interesting because they only contain the simple local cycles between pairs of adjacent stages shown in Figure 18. Hence, the performance of a linear SCL pipeline is determined by the slowest local cycle, of any pair of adjacent stages. However, SCL pipelines with cycles in the datapath can result in more complicated performance bottlenecks.

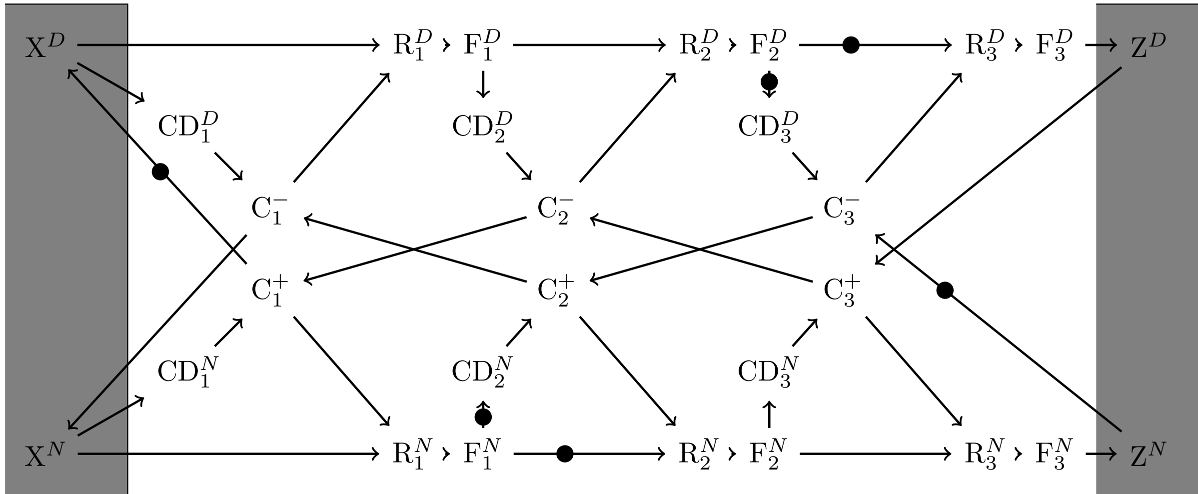

Figure 39: A three stage MG SCL with the second stage initialized to DATA.

A ring is simply a linear pipeline where the output of the last stage drives the input of the first stage. To avoid deadlock at least one stage in the ring must be reset-to-DATA. Ring structures are common to almost all practical designs; a finite state machine implemented in SCL is an example of a ring. Williams studied the performance of asynchronous rings extensively and found three different modes of operation: DATA-limited, bubble-limited, and cycle-limited [36]. If the ring has too few stages it will operate in the bubble-limited region until more stages are added and the ring reaches the cycle-limited mode. The cycle-limited mode will bound the performance of the ring until the slowest cycle is broken into multiple faster cycles. As the number of stages increases the ring eventually becomes performance bound by the lack of DATA tokens in the DATA-limited mode. Similar to the linear pipeline, the performance is bounded by the worst local cycle; however, a ring can also be bounded by loops in the datapath that result in DATA starvation or bubble starvation. The SCL MG model can be used to determine the performance bottleneck for any mode of operation, DATA-limited, bubble-limited, or cycle-limited.

7 SCL OPTIMIZATION TECHNIQUES

7.1 SCL Embedded Registration

The distinction between the SCL registers and SCL threshold gates was discussed in Section 4; the SCL registers are required to latch DATA, while their sleep pin is de-asserted, while the SCL threshold gates do not have hysteresis. The SCL latching behavior can be integrated into the SCL threshold gates [37], referred to here as SCL p1 threshold gates. If the threshold gates in F_i that directly receive input from stage $i-1$ are SCL p1 threshold gates, registers are not necessary. An SCL register in stage i can be eliminated if the output of the register only drives threshold gates in F_i. However, if the register in stage i directly drives a register in stage $i+1$, the register cannot be eliminated.

7.2 SCL Partially Slept Function Blocks

As discussed in Section 4, SCL achieves ultra-low standby power consumption by power-gating all the function blocks when idle. During operation, each SCL pipeline stage must wakeup and go to sleep for each DATA. The power-gating high-V_t PFET is on the critical path of the SCL threshold gates. Due to the sizing of the power-gating PFET transistor, the capacitance presented by the sleep input of the SCL threshold gates cannot be ignored. As a result of the aggregate sleep capacitance, for each pipeline stage, there is an energy overhead for waking and sleeping the pipeline stages. Thus, for applications with little to no idle time, the power overhead to wake and sleep the SCL pipeline stages can outweigh the reduction in standby power. This tradeoff between standby power savings and the cost of sleeping has been studied for synchronous circuits [11]; the

sleep capacitance of an SCL pipeline stage is functionally equivalent to the clock capacitance of a synchronous pipeline stage. The alternative to sleeping the whole SCL function block is to partition the threshold gates in each function block into gates that are slept each cycle and gates that remain powered on. By partially sleeping the SCL function blocks, the aggregate sleep capacitance can be reduced. However, any changes to the sleep mechanism require the timing assumptions from Section 4.7 to be analyzed again. Consider the case where only the SCL registers are slept, while the function blocks remain powered. Rather than the threshold gates in F_i being de-asserted due to the assertion of sleep, the gates will be de-asserted due to the propagation of NULL. Due to differences in path delays within F_i, $CD_{i+1} \uparrow$ can occur before all gates have been de-asserted. The delay of the slowest path from $R_i \downarrow$, to $g_j \downarrow$ is defined as $\max(T_{R_i\downarrow,g_j\downarrow})$. The delay of the fastest path from $R_i \downarrow$, to $CD_{i+1} \uparrow$ is defined as $\min(T_{R_i\downarrow,CD_{i+1}\uparrow})$. Note that RTA8 must still be satisfied, however RTA9 becomes

$$T_{C_i\uparrow,R_i\downarrow} + \max(T_{R_i\downarrow,g_j\downarrow}) < \max(\min(T_{C_i\uparrow,C_{i+1}\uparrow}, T_{C_i\uparrow,R_i\downarrow} + \min(T_{R_i\downarrow,CD_{i+1}\uparrow})) \tag{17}$$

It is not likely that RTA17 can be easily satisfied in practice. As a result, it is necessary to force a subset of the gates in F_i, which are on slow paths to transition to logical 0, using sleep, rather than waiting for NULL to propagate. The threshold gates in F_i are determined to be either a SCL threshold gate or a non-slept combinational gate. Assume that a threshold gate that is slept will become logical 0 after $T_{C_i\uparrow} + T_s$, which can be bounded. The pseudo-code for this approach is given in Figure 40.

Another approach is to partition the threshold gates in F_i based on interleaving slept and

```
1:  for g_i ∈ topo_sort(F_i) do
2:      max_reset(g_i) ←T_(C_i ↑, R_i ↓) + reset_delay(g_i)
3:      for g_j ∈ in_neighbors(g_i) do
4:          if max_reset(g_j) + reset_delay(g_i) > max_reset(g_i) then
5:              max_reset(g_i) = max_reset(g_j) + reset_delay(g_i)
6:          end if
7:      end for
8:      if max_reset(g_i) > max(min( T_{C_i↑,C_{i+1}↑}, T_{C_i↑,R_i↓} + min(T_{R_i↓,CD_{i+1}↑} )) then
9:          sleep(g_i) ← True
10:         max_reset(g_i) ← T_{C_i↑} + T_s
11:     else
12:         sleep(g_i) ← False
13:     end if
14: end for
```

Figure 40: Delay-dependent algorithm for partitioning slept and non-slept gates in F_i.

non-slept gates rather than analyzing the maximum delays of paths. In this approach the threshold gates in F_i are either SCL p1 threshold gates or non-slept threshold gates. Once again, all slept gates will become logical 0 after $T_{C_i\uparrow} + T_s$. If a non-slept gate g_i is only driven by slept gates, the output will become logical 0 after $T_{C_i\uparrow} + T_s + T_{g_i}$. This can be satisfied by first performing vertex coloring on the gates in F_i. Vertex coloring will assign each gate a color such that no gate will be driven by a gate with the same color. A single color can be assigned to be the non-slept class, while the remaining colors are assigned to the slept class; to minimize sleep capacitance the coloring with the largest number of gates is selected to be the non-slept coloring. The pseudo-code for this approach is given in Figure 41.

7.3 SCL Pipeline Standby Detection

Using partially slept function blocks results in a reduction of aggregate sleep capacitance but the SCL threshold gates partitioned into the non-slept class are not power-gated. Since the non-slept gates are not power-gated, they remain powered on regardless of pipeline activity. However,

```
1:  for g_i ∈ reversed(topo_sort(F_i)) do
2:      color(g_i) ← 0
3:      count ← 1
4:      conflict ← False
5:      while color(g_i) = 0 do
6:          for g_j ∈ out_neighbors(g_i) do
7:              if  color(g_j) = count then
8:                  conflict ← True
9:              end if
10:         end for
11:         if conflict = False then
12:             color(g_i) ← count
13:         else
14:             count ← count + 1
15:             conflict ← False
16:         end if
17:     end while
18: end for
```

Figure 41: Greedy vertex coloring for partitioning slept and non-slept gates in F_i.

the gates partitioned in the non-slept class can still be power-gated during periods of no activity by detecting when the SCL pipeline is in the standby state. Consider a n-stage linear SCL pipeline, which is known to contain no DATA during periods of no activity. This standby state can be easily detected by logically ANDing the outputs of the final completion gates. As DATA propagates through the stages, at least one final C-element will be logical 0. A three-stage SCL pipeline with standby-detection is shown in Figure 42. This standby detection signal, s0, can be used to power-gate the SCL threshold gates partitioned into the non-slept group. Thus, during periods of no activity, all of the SCL threshold gates will be power-gated. Determining the standby state of an SCL pipeline is application dependent; however, every pipeline's standby state can be characterized by some combination of values for the final C-elements of each stage.

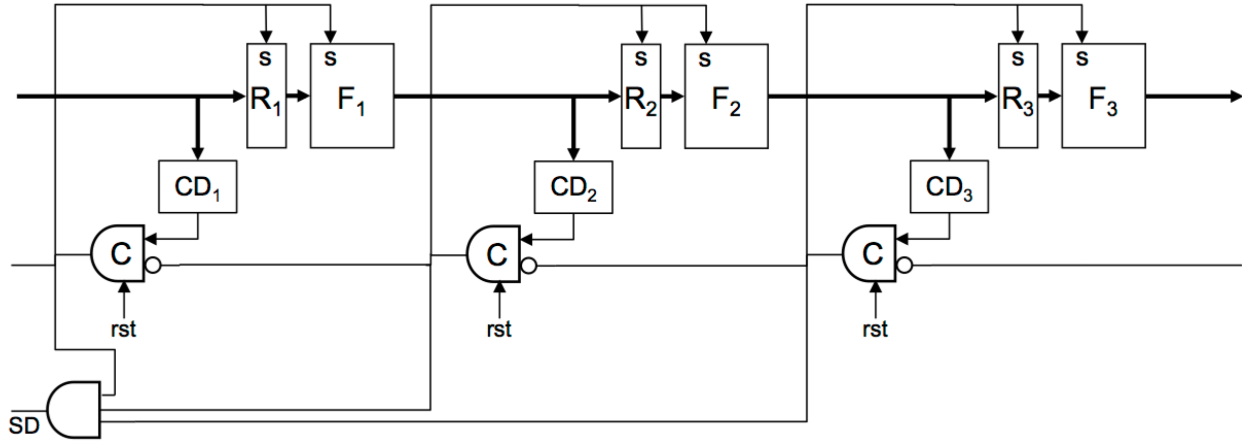

Figure 42: A three-stage SCL pipeline with standby-detection logic.

7.4 SCL Pipelining

As asynchronous circuits are slack elastic, it is possible to insert additional pipeline stages to increase performance without altering the observable behavior of the circuit. This is a useful property that allows designers to trade increased area for increased performance at a late stage in the design process. However, the complexity of determining the minimum pipelining needed to achieve a given performance for an asynchronous circuit has been shown to be NP-complete [13]. Due to the high complexity of solving for the optimum pipelining, an efficient algorithm for pipelining linear NCL pipelines was presented in [28]. Each pipeline stage of a NCL circuit typically has symmetric to-DATA and to-NULL delays, which means the performance of a linear pipeline can be determined by the performance of the single slowest stage [33]. However, the performance of linear SCL pipelines is always a function of adjacent pipeline stages as discussed in Section 4.7.

The pipelining procedure begins with a combinational block bounded by registers on the inputs and outputs. The goal is to partition this initial single pipeline stage into multiple smaller

pipeline stages, such that the new pipelined configuration satisfies a target performance TDD_{target}, where TDD is defined as the period between successive DATA. The combinational block F_i is first partitioned into levels, in a top-down fashion. Specifically, each gate is assigned a level l, such that it can only receive signals generated by gates in level k, where $k < l$. If a combinational block is partitioned into n-levels, the inputs are registered in level 0 and the outputs are registered in level n. For example, an unsigned 4x4 multiplier results in the following partitioning given in Table 1.

Now that the partitioning for a combinational block has been determined, each pipelining configuration can be represented by a binary vector. The un-pipelined multiplier shown in Table 1 can be represented by the binary vector $[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]$, where a 1 indicates that the outputs of a level are registered; the number of registers required ranges from 16 to 110. For a given pipelining configuration we must also determine the resulting TDD. As was discussed in Section 4.7, the performance of an SCL linear pipeline is the slowest cycle of any pair of adjacent pipeline stages. Therefore, the performance of a pipeline configuration can be found by evaluating every pair of adjacent stages and applying the equation

$$T_{cycle} = 4 * T_{C-element} + T_{R_i} + T_{F_i} + T_{CD_i} + T_{R_i} + T_{F_{i+1}} + T_{CD_{i+1}} \tag{18}$$

where T_{CD_i} is $T_{C-element} * \log_4(width(i))$. Using this sliding-window approach, the performance of each pipelining configuration can be found exhaustively. The performance and number of registers required of each pipeline configuration for the unsigned 4x4 multiplier is illustrated in Figure 43. Despite the large number of possible configurations, the only valuable ones are the pareto-optimal configurations. For example, if TDD_{target} = 25, the configuration requiring 78 registers

Table 1: Pipelining partitions for 4x4 unsigned multiplier.

Level	Width	Delay
0	8	0
1	22	2
2	16	2
3	13	4
4	12	4
5	11	4
6	10	4
7	10	2
8	8	2
9	8	0

and providing a TDD of 22 would be selected, since the other seven feasible configurations either require more registers or result in lower performance.

As the number of pipelining configurations is 2^n for a n-level partitioning, enumerating all possible configurations is impractical for large designs. A variation on the approach in [32] is used, which starts with the pipeline configuration with all levels registered, then attempts to remove any registered level that does not cause the TDD to become larger than TDD_{target}. The pseudo-code for the pipelining algorithm is given in Figure 44. Note that while the previous approaches for NCL [28][32] considered a single stage, this approach for SCL considers pairs of adjacent stages.

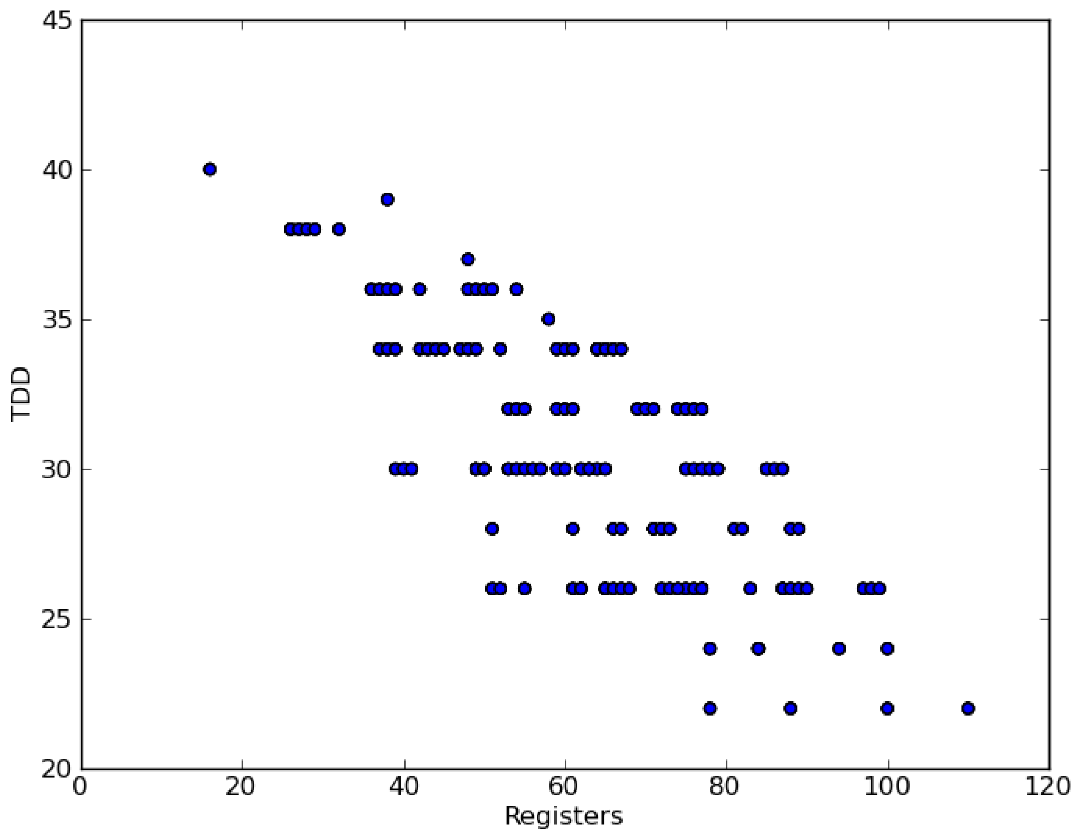

Figure 43: Pipeline configurations for 4x4 unsigned multiplier.

1: **for** $l_i \in$ levels **do**
2: level_registered(l_i) $\leftarrow$ True
3: **end for**
4: **for** $l_i \in$ levels **do**
5: **if** level_registered(l_i) = True **then**
6: level_registered(l_i) $\leftarrow$ False
7: new_TDD $\leftarrow$ estimate_TDD(level_registered)
8: **if** new_TDD $>$ target_TDD **then**
9: level_registered(l_i) $\leftarrow$ True
10: **end if**
11: **end if**
12: **end for**

Figure 44: Efficient pipelining algorithm for linear SCL pipeline.

8 AUTOMATED SCL CONVERSION FLOW

The SCL design flow consists of three main stages, illustrated in Figure 45.

8.1 Generating the Single-Rail Netlist

The first stage begins very similar to the synthesis of any synchronous design. Synopsys Design Compiler® is used to synthesize the original synchronous RTL (VHDL/Verilog). There are some restrictions on the synchronous design: the design must have a single clock domain and no gating logic can be used in the clock or asynchronous reset paths. The target library used for synthesis is a virtual library containing typical standard cell components. This virtual library contains single-rail versions of the dual-rail library components used in the following stage. The area, delay, and power models for the virtual library components should reflect their dual-rail counterparts to aid the synthesis tool during technology mapping. In this work the combinational components of the virtual library consist of basic gates, a full adder, and a 2:1 multiplexer. However, any hand-designed components can be easily added. Both asynchronous resettable and non-resettable D flip-flops are included in the virtual library.

8.2 Generating the Dual-Rail Netlist

Once the synchronous netlist of virtual cells has been generated, the tools extract a netlist-graph as discussed in Section 5.2. The virtual flip-flops are now replaced with the corresponding reset-to-DATA SCL register; a flip-flop with no reset becomes a reset-to-DATA0 SCL register, while a flip-flop with an asynchronous reset becomes a reset-to-DATA0/1 SCL register. The clock

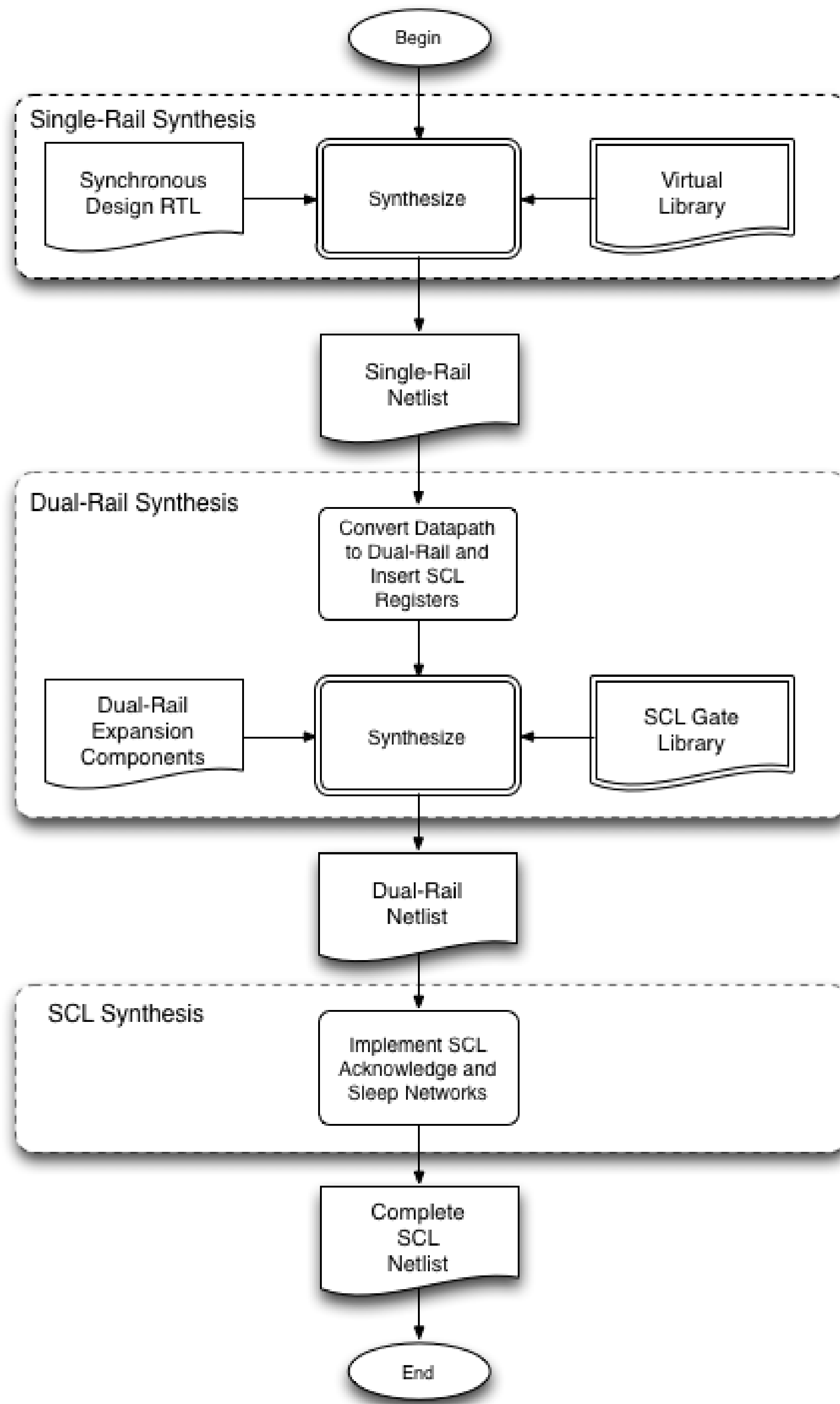

Figure 45: The flow for automated synchronous to SCL conversion.

network is removed. A register-graph is then extracted from the netlist-graph, as discussed in Section 5.2. Reset-to-NULL SCL registers are then inserted to satisfy the conditions discussed in Section 5.2. Each SCL register is first grouped into a unique pipeline stage and then iteratively merged until no more stages can be merged. The acknowledge networks are then extracted from the register-graph as discussed in Section 5.2. Finally, the remaining virtual cells are replaced with their threshold gate counterparts and the single-rail signals are expanded to dual-rail signals. Several pieces of information are stored for each stage, including the register grouping $R(ps_i)$, the acknowledge set $ACK(ps_i)$, and the sleep set $SLEEP(ps_i)$. In addition, the SCL MG model discussed in Section 6 is extracted for the design, and the performance bottleneck is stored.

8.3 Optimizing the Dual-Rail Netlist

After the dual-rail SCL netlist has been produced, a second technology-mapping pass takes place. While NCL requires that dual-rail functions satisfy input-completeness and observability criteria, the dual-rail functions in SCL can be logically optimized by Design Compiler® to reduce area, power and delay, since SCL does not require input-completeness or observability [32]. The *dont_touch* attribute is enabled for SCL register instances to prevent DC from modifying them. For this pass, a Liberty file with data for the SCL threshold gates is utilized; however, the cells described in the Liberty file lack the sleep input. The resulting dual-rail SCL netlist still lacks the completion detectors, final C-elements, and acknowledge and sleep networks.

8.4 Completing the SCL Netlist

In this final stage, the SCL pipeline stages are completed and the necessary handshaking signals are generated. While the acknowledge and sleep network information was extracted earlier, the acknowledge and sleep networks are now implemented in the SCL netlist. For each SCL pipeline stage a completion detector network and a final C-element are generated, as shown in Figure 29. The resulting netlist represents a complete and functional SCL design.

8.5 Validating the SCL Netlist Equivalance

A testbench is now automatically generated for the SCL netlist. The testbench instantiates both the original synchronous design as well as the SCL design. Both the synchronous and SCL blocks are provided with the same random input vectors, and the outputs are verified to be consistent, as discussed in Section 5.1.

8.6 Experimental Results

The procedure presented in this work was implemented in a combination of Python, C++ and TCL scripts. The implemented flow was tested against sequential designs from the ISCAS'89 benchmark suite. The approaches using both triplet register replacement and intelligent register replacement were considered, as well as un-merged and merged acknowledgement networks. The resulting areas are shown in Table 2. The merging of acknowledgment networks clearly results in significant area reductions for the ISCAS benchmark designs. Recall that intelligent register replacement seeks to reduce the number of registers needed, which can be as low as two-thirds

Table 2: Area of ISCAS'89 Designs

	Triplet Register Replacement			Intelligent Register Replacement		
	Un-Merged	Merged		Un-Merged	Merged	
Design	Area (um^2)	Area (um^2)	Reduction (%)	Area (um^2)	Area (um^2)	Reduction (%)
s208_1	1292.4	713.9	45	1292.4	713.9	45
s27	354.2	251.3	29	354.2	251.3	29
s298	1740.6	1077.1	38	1740.6	1077.1	38
s344	2307.2	1331.3	42	2307.2	1331.3	42
s349	2308.3	1332.4	42	2308.3	1332.4	42
s382	2596.7	1468.1	43	2495.2	1401.1	44
s386	1807.9	942.5	48	1807.9	942.5	48
s400	2635.9	1472.4	44	2534.4	1405.4	45
s420_1	3373.9	1354	60	3373.9	1354	60
s444	2541.6	1458.7	43	2440.1	1391.8	43
s510	2885.4	1566	46	2885.4	1566	46
s526	2770.6	1508.8	46	2770.6	1508.8	46
s526n	2762.3	1507.3	45	2762.3	1507.3	45
s641	4037.8	1897.6	53	4003.9	1863.7	53
s713	4039.2	1899	53	4005.4	1865.2	53
s820	4045.3	1841.8	54	4045.3	1841.8	54
s832	4045.3	1841.8	54	4045.3	1841.8	54
s838_1	9756	2647.8	73	9756	2647.8	73
s1238	7045.9	3288.6	53	6741.4	3166.6	53
s1488	5897.5	3164	46	5897.5	3164	46
s1494	6287	3224.5	49	6287	3224.5	49
s1423	21591.4	5121.4	76	21540.6	5123.5	76

the number of registers compared to triplet register replacement. However, the use of intelligent register replacement offers little area reduction for the ISCAS sequential benchmarks; this is due to the fact that many of the designs are control-oriented and have feedback paths from one pipeline stage to a previous stage. As a result of these feedback paths, the tools must insert a third register on these paths bringing the total number of registers up to parity with the triplet register replacement method.

9 CONCLUSION

In this dissertation, a flow utilizing commercial synchronous design software and custom tools to convert synchronous RTL to an SCL netlist was presented. The performance and timing assumptions were derived for SCL in Chapter 4. The detailed procedure for converting a synchronous design to SCL while maintaining equivalence was discussed in Chapter 5. A method to determine the bottleneck in any SCL design was presented in Chapter 6. A number of SCL optimizations, including embedded registration, partially slept function blocks, standby detection and automated constraint-driven pipelining were presented in Chapter 7. The integration of the flow with Synopsys Design Compiler and experimental results for the ISCAS '89 benchmark designs was discussed in Chapter 8.

10 REFERENCES

[1] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, April 1997.

[2] A. Bardsley and D. A. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000.

[3] P.A. Beerel, G.D. Dimou, and A.M. Lines. Proteus: An asic flow for ghz asynchronous designs. *Design Test of Computers, IEEE*, 28(5):36–51, Sept.-Oct.

[4] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(10):1437–1455, Oct.

[5] J. Cortadella, A. Kondratyev, L. Lavagno, and C.P. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(10):1904–1921, Oct.

[6] W.M. Elgharbawy and M.A. Bayoumi. Leakage sources and possible solutions in nanometer cmos technologies. *Circuits and Systems Magazine, IEEE*, 5(4):6–17, Quarter.

[7] Karl Fant. *Logically Determined Design: Clockless System Design with NULL Convention Logic*. Wiley-Interscience, 2005.

[8] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 4(2):247–253, June.

[9] J.D. Garside, W.J. Bainbridge, A. Bardsley, D.M. Clark, D.A. Edwards, S.B. Furber, J. Liu, D.W. Lloyd, S. Mohammadi, J.S. Pepper, O. Petlin, S. Temple, and J.V. Woods. Amulet3i-an asynchronous system-on-chip. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 162–175.

[10] R. Ginosar. Metastability and synchronizers: A tutorial. *Design Test of Computers, IEEE*, 28(5):23–35, Sept.-Oct.

[11] Zhigang Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 32–37, 2004.

[12] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, Dec.

[13] Sangyun Kim and P.A. Beerel. Pipeline optimization for asynchronous circuits: complexity analysis and an efficient optimal algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(3):389–402, March.

[14] A. Kondratyev and K. Lwin. Design of asynchronous circuits using synchronous cad tools. *Design Test of Computers, IEEE*, 19(4):107–117, Jul/Aug.

[15] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev. Asynchronous design using commercial hdl synthesis tools. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 114–125.

[16] D.H. Linder and J.C. Harden. Phased logic: supporting the synchronous design paradigm with delay-insensitive circuitry. *Computers, IEEE Transactions on*, 45(9):1031–1044, 1996.

[17] Rajit Manohar and Alain J. Martin. Quasi-delay-insensitive circuits are turing-complete. Technical report, Pasadena, CA, USA, 1995.

[18] Alain J. Martin. Compiling communicating processes into delay-insensitive vlsi circuits. *Distributed Computing*, 1(4):226–234, 1986.

[19] Alain J. Martin. The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation. In *Hardware Specification, Verification and Synthesis*, pages 244–259, 1989.

[20] Alain J. Martin. A program transformation approach to asynchronous vlsi design. In *NATO ASI DPD*, pages 441–467, 1996.

[21] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystrom, Paul Penzes, Robert Southworth, Uri Cummings, and Tak Kwan Lee. The design of an asynchronous mips r3000 microprocessor. In *in Advanced Research in VLSI*, pages 164–181, 1997.

[22] S. Masteller and L. Sorenson. Cycle decomposition in ncl. *Design Test of Computers, IEEE*, 20(6):38–43, Nov.-Dec.

[23] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. *Theory of Switching, Proceedings of the International Symposium*, pages 204–243, 1959.

[24] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr.

[25] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada. 1-v power supply high-speed digital circuit technology with multithreshold-voltage cmos. *Solid-State Circuits, IEEE Journal of*, 30(8):847–854, Aug.

[26] R.O. Ozdag and P.A. Beerel. High-speed qdi asynchronous pipelines. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 13–22, April.

[27] R.B. Reese, S.C. Smith, and M.A. Thornton. Uncle - an rtl approach to asynchronous design. In *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, pages 65–72, May.

[28] V. Satagopan. *Automated Pipelining Optimization, Energy Estimation, and DFT Techniques for Asynchronous NULL Convention Circuits using Industry-Standard CAD Tools*. Ph.D. dissertation, University of Missouri-Rolla, 2007.

[29] A. Smirnov, A. Taubin, Ming Su, and M. Karpovsky. An automated fine-grain pipelining using domino style asynchronous library. In *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on*, pages 68–76, June.

[30] R. Smith and M. Ligthart. High-level design for asynchronous logic. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 431–436.

[31] S.C. Smith. Speedup of self-timed digital systems using early completion. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 98–104.

[32] Scott Smith and Jia Di. *Designing Asynchronous Circuits using NULL Convention Logic (NCL)*. Morgan & Claypool, 2009.

[33] Scott C. Smith, Ronald F. DeMara, Jiann S. Yuan, M. Hagedorn, and D. Ferguson. Delay-insensitive gate-level pipelining. *Integration*, 30(2):103–131, 2001.

[34] K.S. Stevens, Yang Xu, and V. Vij. Characterization of asynchronous templates for integration into clocked cad flows. In *Asynchronous Circuits and Systems, 2009. ASYNC '09. 15th IEEE Symposium on*, pages 151–161, May.

[35] S. Tugsiriavisut and P.A. Beerel. Control circuit templates for asynchronous bundled-data pipelines. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 1098–.

[36] Ted E. Williams. Performance of iterative computation in self-timed rings. *J. VLSI Signal Process. Syst.*, 7(1-2):17–31, February 1994.

[37] Liang Zhou. *ULTRA-LOW POWER AND RADIATION HARDENED ASYNCHRONOUS CIRCUIT DESIGN*. Ph.D. dissertation, University of Arkansas, 2012.