DESIGN OF ROBUST ASYNCHRONOUS RECONFIGURABLE CONTROLLERS FOR PARALLEL SYNCHRONIZATION USING EMBEDDED GRAPHS

JAMES SEBASTIAN GUIDO

Doctor of Philosophy(Ph.D.)
School of Electrical, Electronic, and Computer Engineering
Newcastle University

February 2015

DECLARATION

I, James Sebastian Guido, confirm that this thesis and the work presented in it are my own achievement.

I have read and understand the penalties associated with plagiarism.

Newcastle, February 2015

James Sebastian Guido

James Sebastian Guido

DEDICATION

Dedicated to the memory of my grandfather Joseph Nicholas Guido, who once found himself on a similar path to the one that I now walk. Though you left too early to share your insights with me, thanks for treading the ground first and making the trip feel less lonely.

April 5, 1923 – June 19, 1986

And to his brother, Carmelo Basil Guido, who was also walking the path as well, but had his trip cut short. I didn't know you, but I wish that I had.

May 10, 1915 – December 12, 1944

And to my grandmother Rosalia Anne Guido, who was always ready with words of encouragement and sage advice. You taught me to always look forward and not dwell too closely on the foibles and fallacies of the past, but rather to learn from them and make better decisions in the future. Thanks for being one of the major pillars in my life, and for helping to shape me into the man that I am today.

September 7, 1923 – December 1, 2014

ABSTRACT

Synchronization is a key System-on-Chip (SoC) design issue in modern technologies. As the number of operating points under consideration increases, specifications which are capable of altering key parameters such as the time available for synchronization and Mean Time Between Failures (MTBF) in response to input from the user/system become desirable. This thesis explores how a combination of parallelism and scheduling, referred to as wagging, can be utilized to construct schedulers for synchronizer designs which are capable of pooling the gain-bandwidth products of their composite devices, in order to satisfy this requirement.

In this work, we explore the ways in which the areas of graph theory and reconfigurable hardware design can be applied to generate both combinational and sequential scheduler designs, which satisfy the behavior requirement above. Further to this point, this work illustrates that such a scheduler is primarily comprised of an interrupt subsystem, and a reconfigurable token ring. This thesis explores how both of these components can be controlled in absence of a clock signal, as well as the design challenges inherent to each part.

The final noteworthy issue in this study is with regard to the flow control of data in a parallel synchronizer that incorporates a First-In First-Out (FIFO) buffer to decouple the reading and writing operations from each other. Such a structure incurs penalties if the data rates on both sides are not well matched. This work presents a method by which combinations of serial and parallel reading operations are used to minimize this mismatch.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

Guido, James Sebastian, and Alexandre Yakovlev. "Reconfigurable controllers for synchronization via wagging." In Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI, pp. 175-180. ACM, 2011.

Guido, James Sebastian, and Alexandre Yakovlev. "Design of Self-Timed Reconfigurable Controllers for Parallel Synchronization via Wagging." In IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 23(2):292-305, Feb. 2015 doi: 10.1109/TVLSI.2014.2306176

In my experience, finding design simplicity which copes with the demands of underlying complexity tends to yield the best rewards. I can also say that the quest for simplicity, is also usually anything but simple.

ACKNOWLEDGMENTS

It's been a lot of fun working at Newcastle University over the past few years. As my time here draws to a close, my thoughts turn to all of the people that have helped me over the years. First and foremost is my supervisor Alex Yakovlev, who helped me navigate from being a fresh out of college Masters student into a decent researcher. Ghaith Tarawneh assited me in developing a script which formed the front-end of my SPECTRE logical analysis experiments, and was generally beyond helpful during the course of my studies. Andrey Mokhov also deserves mention for his useful discussions which led to one optimization later in this thesis. Robin Emery also deserves mention, for reminding me to always try and see the big picture with regards to my work by asking me the all-important question"So what?"

On the more personal side of things, I'd like to thank my mother, Michaelanne, my father, James, my sister, Rochelle, and my niece and nephews, Magnus, Argyle, and Iona for being steadfast in their support. I'd also like to thank my good friend John Parry, for always being there to lend a sympathetic ear when times got rough or when things didn't quite go my way. More to that, I'd also like to thank his brother Richard, Anthony Pearson, and Thomas James for several years' worth of wonderful memories.

I could go on and on, but I'll always be the first to admit that I'm here because I had the encouragement and support of others. Thanks for making these some of the best years of my life.

CONTENTS

I PHD THESIS 1

1 INTRODUCTION 3
- 1.1 Motivation . . . 3
- 1.2 Contributions . . . 6
  - 1.2.1 Algorithms for Load Balancing in Token Rings via Distributed Embedded Graphs . . . 6
  - 1.2.2 Manipulation of Parallelism in a Self-Timed Reconfigurable Control Device via One-Hot Coding . . . 6
  - 1.2.3 Tolerance of Hard Faults in a Self-Timed Reconfigurable Control Device via Bypass Paths . . . 7
  - 1.2.4 Flow Control in a FIFO Synchronizer based on Wagging . . . 7
- 1.3 Organization of Thesis . . . 7

2 BACKGROUND LITERATURE & MODELS 9
- 2.1 Introduction(Asynchronous Circuits) . . . 9
  - 2.1.1 Properties of Asynchronous Circuits . . . 9
  - 2.1.2 Asynchronous Control Circuits . . . 11
  - 2.1.3 Asynchronous Circuit Primitives . . . 12
- 2.2 Petri Nets . . . 15
  - 2.2.1 Pre-sets and Post-sets . . . 16
  - 2.2.2 Enabling and Firing . . . 16
  - 2.2.3 Reachability . . . 17
  - 2.2.4 Other Petri Net Properties . . . 17
- 2.3 Signal Transition Graphs . . . 18
  - 2.3.1 Relation to Petri Nets . . . 18
  - 2.3.2 State Graphs and Reachability Graphs . . . 19
  - 2.3.3 Complete State Coding . . . 20
  - 2.3.4 Relative Timing in Signal Transition Graphs . . . 21
- 2.4 Metastability . . . 21
- 2.5 Embedded Cycle Graphs . . . 24
  - 2.5.1 Basic Graph Definitions . . . 24
  - 2.5.2 Applications of an Embedded Cycle Graph . . . 25

3 OVERVIEW OF WAGGING SYNCHRONIZATION 27
- 3.1 Introduction(Synchronization Methods) . . . 27
  - 3.1.1 Synchronization Overview . . . 27
  - 3.1.2 Cascaded Flip-Flop Synchronization . . . 28
  - 3.1.3 Mean-Time Between Failures in a Cascaded Flip-Flop Synchronizer . . . 29
  - 3.1.4 FIFO Synchronization . . . 31
- 3.2 Wagging Synchronization . . . 33
  - 3.2.1 Two-way Wagging Buffer . . . 35
  - 3.2.2 Wagging Synchronizer Concept . . . 35

3.2.3 Effect of Wagging on the Failure Rate of Cascaded Flip-Flop Synchronizers . . . 36
3.2.4 Impact of Incorporating Reconfigurable Hardware Capability into a Wagging Synchronizer Design . . . 38
3.2.5 Overheads of Incorporating Reconfigurable Hardware Capability into a Wagging Synchronizer Design . . . 40
3.2.6 Basic Operation of the Reconfigurable Control Device . . . 41
3.3 Topology of a Reconfigurable Token Ring . . . 43
3.3.1 Cyclic Behavior of Token Rings . . . 43
3.3.2 Sub-Optimal Distribution Algorithm Specification . . . 46
3.3.3 Optimal Distribution Algorithm Specification . . . 48
3.4 Token Ring Designs . . . 51
3.4.1 Ring Oscillators . . . 52
3.4.2 Fast David Cells . . . 52
3.4.3 Muller Pipeline . . . 55
3.4.4 Performance Comparison . . . 56
3.4.5 Distributed Token Ring Implementation . . . 57
4 PARALLELISM IN A RECONFIGURABLE CONTROLLER FOR WAGGING SYNCHRONIZATION 63
4.1 Introduction (Arbitration & Reconfiguration) . . . 63
4.2 Related Work (Overview of Arbitration) . . . 63
4.2.1 The Mutual Exclusion Element (MUTEX) . . . 64
4.2.2 Token Ring Arbiters . . . 64
4.2.3 Arbitration via Pausible Clocks . . . 65
4.2.4 Relation to a Reconfigurable Control Device for a Wagging Synchronizer . . . 66
4.3 Reconfiguration Protocol . . . 66
4.3.1 Mathematical Foundations (Assumptions) . . . 69
4.4 UCOM Threading . . . 70
4.4.1 Enforcing Firing Order in Cyclic Independent Loops . . . 70
4.4.2 Control of End Behavior in Cyclic Independent Loops . . . 71
4.5 Basic Controller Design . . . 71
4.5.1 Circuit Synthesis . . . 72
4.5.2 Performance Comparison . . . 75
4.6 Output Data Merging in Reconfigurable Controller . . . 77
4.6.1 Circuit Synthesis & Results Analysis . . . 77
5 ROBUSTNESS IN A RECONFIGURABLE CONTROLLER FOR WAGGING SYNCHRONIZATION 79
5.1 Introduction (Principle of Exclusion) . . . 79
5.2 Fault Model Definitions . . . 79
5.2.1 Fault types . . . 79
5.3 Related Work (Johnson Counter) . . . 80
5.4 Chordal Ring Networks (Algorithm Reinterpretation) . . . 81
5.4.1 Chordal Bypass Path Algorithm . . . 82
5.4.2 Results Analysis . . . 83
5.5 UCOM Thread Forwarding . . . 84

5.5.1 Theoretical Overview (Why Forward?) . . . 85
5.5.2 Applications to Robust Controller Design . . . 86
5.6 Advanced Controller Design (Hierarchy & Crossbar Plug-in) . . . 86
5.6.1 Theoretical Overview (System Hierarchy) . . . 86
5.6.2 Derivation of the Petri Net (PN) Model . . . 88
5.6.3 PN Model & Simulation . . . 90
5.7 Validating Configurations via a Nearest Neighbor Checking Algorithm . . . 92
5.7.1 Validation Algorithm . . . 92
6 FLOW CONTROL IN WAGGING SYNCHRONIZERS INCORPORATING FIFO BUFFERS 95
6.1 Introduction (Flow Control in FIFO Synchronizers) . . . 95
6.2 Related Work (STARI) . . . 95
6.2.1 Applications to Wagging Synchronization . . . 96
6.3 FIFO Testing Methodology . . . 98
6.3.1 Top-level Design Considerations . . . 98
6.3.2 FIFO I/O Considerations . . . 100
6.4 VHDL Experimental Setup . . . 101
6.4.1 VHDL Design Flow . . . 101
6.4.2 Test Vector Generation . . . 101
6.4.3 State Machine Logic for a Wagging Scheduler . . . 102
6.5 VHDL Simulations . . . 104
7 CONCLUSIONS 109
7.1 Main Contributions . . . 109
7.2 Future Work . . . 110

II APPENDIX 111
A APPENDIX 113
A.1 INTMUX Boolean Logic Minimization . . . 113
A.2 CADENCE PWL Waveform Generation Example . . . 114

BIBLIOGRAPHY 117

LIST OF FIGURES

Figure 1.1 Cascaded Two-Flop Synchronizer 4
Figure 1.2 Parallel Two-Flop Synchronizer 5
Figure 2.1 Behavioral representation of a D flip-flop 13
Figure 2.2 Behavioral representation of a Muller C-Element (MCE) . . . 13
Figure 2.3 Annotated Muller C-Element schematic with signal transitions representing 1 operating cycle 14
Figure 2.4 Annotated David Cell schematic with signal transitions representing 1 operating cycle 14
Figure 2.5 Petri Net Symbols 15
Figure 2.6 Petri Net of a 3-cell token ring 16
Figure 2.7 Reachability Graph of a 3-cell token ring 17
Figure 2.8 Annotated Signal Transition Graph (STG) of a 3-cell token ring 19
Figure 2.9 Annotated STG of a 3-cell token ring (CSC Conflict Resolved) 20
Figure 2.10 Abstract view of a butterfly curve. From [44] 22
Figure 3.1 Overview of clocking methods: (a) Synchronous, (b) Rationally related clocks, (c) Multiple clocks 28
Figure 3.2 Synchronizing interface. From [44] 29
Figure 3.3 Cascaded flip-flop synchronizer 30
Figure 3.4 FIFO synchronizer. From [44] 32
Figure 3.5 Conceptual diagram illustrating the wagging principle. . . . 33
Figure 3.6 Annotated diagram of a two-way wagging buffer. 34
Figure 3.7 Block diagram of an N-wagging synchronizer 37
Figure 3.8 Data flow from the transmitter end of a wagging synchronizer to the input of the FIFO in Fig. 3.7 with $j = 3$ and a 50% duty cycle. 38
Figure 3.9 Block diagram of a reconfigurable controller suitable for a wagging synchronizer. 42
Figure 3.10 Cycle graphs of a token ring 44
Figure 3.11 Results of the sub-optimal distribution algorithm 48
Figure 3.12 Optimized distributed feedback algorithm overview 51
Figure 3.13 Ring oscillator with $n_{gate} = 5$, and an active-low reset signal 52
Figure 3.14 Annotated fast David Cell schematic with signal transitions representing 1 operating cycle 53
Figure 3.15 Four-way sequential token ring based on fast David Cells (DCs) 54
Figure 3.16 4-way sequential token ring based on MCEs 55
Figure 3.17 8-way reconfigurable token ring implementation based on sequential logic (5 possible configurations) 58
Figure 3.18 Transient response of the self-timed reconfigurable token ring control circuit based on DCs illustrating the effect of increased parallelism on the time available for synchronization (path: CTR4) 58

Figure 3.19 Distributed graph which models the behavior of the 8-way reconfigurable token ring circuit. 59
Figure 3.20 Basic reconfigurable control circuit based on hardware replication and select signals. 59
Figure 3.21 Area comparison of a self-timed reconfigurable token ring control circuit based on fast DCs. $(cycle_{(min)}(even) = 4)$ . . 60
Figure 4.1 The mutual exclusion element 64
Figure 4.2 STGs of token ring arbiters 65
Figure 4.3 Pausible clock arbiter overview 66
Figure 4.4 Behavior graph of an 8-way reconfigurable controller with several cycles. 67
Figure 4.5 Interrupt block diagram. 67
Figure 4.6 Signal transition graph (FWD PATH). 68
Figure 4.7 Signal transition graph (REV PATH). 71
Figure 4.8 Top view of three 16-bit interrupt modules tied together via Unidirectional Communication (UCOM) threads. Solid lines represent the connectivity of the signal lines present in this section. Grayed out portions and dotted lines represent the connections and blocks which will be discussed in Section 4.6. 73
Figure 4.9 Internal view of an individual 16-bit interrupt module (i.e. module #1 in the example). 74
Figure 4.10 Transient response of the control signals for the subsystem of three interrupt devices, which demonstrates the firing behavior of the design. Reconfiguration data, UCOM, FLAG, and MEM signals have been omitted. 75
Figure 4.11 INTMUX boolean logic minimization 78
Figure 5.1 Eight-way ring network based on chordal bypass paths . . . 81
Figure 5.2 Chordal Bypass Path Algorithm Overview 82
Figure 5.3 Schematic of the self-timed reconfigurable token ring control circuit based on DCs illustrating the effect of exclusion on the time available for synchronization in the system (path: CTR6) 84
Figure 5.4 Transient response of the self-timed reconfigurable token ring control circuit based on DCs illustrating the effect of exclusion on the time available for synchronization in the system (path: CTR6) 85
Figure 5.5 Conceptual overview of the token ring based on chordal graphs. 87
Figure 5.6 Overview of the reconfigurable interrupt an XBAR select at vertex 0. 87
Figure 5.7 PN of building blocks for the reconfigurable control device. . 88
Figure 5.8 PN of the interrupt subsystem. 90
Figure 5.9 PN representation of chordal reconfiguration protocol. 91
Figure 6.1 Four-cell STARI interface schematic. From [12] 96
Figure 6.2 Data flow of a four-cell STARI interface, assuming that the transmitter and receiver clocks are in phase. From [12] . . . 96

Figure 6.3 Top view of a FIFO synchronizer incorporating wagging at the transmitter and receiver. 97
Figure 6.4 FIFO standard cell. From [82] 99
Figure 6.5 Serialized input from the transmitter to the input of the FIFO buffer. 100
Figure 6.6 Four-bit Pseudorandom Binary Sequence (PRBS) generator. . 102
Figure 6.7 Four-bit signature analyzer. 103
Figure 6.8 State machine for four-bit wagging control device. 104
Figure 6.9 VHDL Experimental Setup. 105
Figure 6.10 Transient response of the flow control in the complete self checking circuit. 106

LIST OF TABLES

Table 3.1 MTBF of a Cascaded Two-Flop Synchronizer. From [2] 29
Table 3.2 MTBF of a Fixed 3-Way Wagging Synchronizer. From [2] . . . 39
Table 3.3 Differences in the Cycle Lists of a Multi-cycle Token Ring with Distributed Edges vs. an Undistributed Ring (Maximum Cycle Length = 8 Vertices) 45
Table 3.4 Differences in the Cycle Lists of Two Different Multi-cycle Token Rings Implementations with Distributed Edges (Maximum Cycle Length = 8 Vertices) 50
Table 3.5 Average Power Consumption per Cell across Process Corners at V_{DD} = 1.0 V and t_{cycle} = 1 ns (TN/TP) 57
Table 3.6 Average Time Available for Synchronization across Process Corners at t_{cycle} = 1 ns (TN/TP) 57
Table 4.1 Duration of the Active Region of the Interrupt Subsystem for Different Configurable Modes at a TN-TP Corner with V_{DD} = 1.0 V 75
Table 4.2 Average Power Consumption of the Active Region of the Interrupt Subsystem for Different Configurable Modes at a TN-TP Corner with V_{DD} = 1.0 V 76
Table 5.1 Variations in the Time Available for Synchronization in the Reconfigurable Token Ring Control across 3 Major Process Corners @V_{DD} = 1.0 V and at a Temp = 27 °C 84
Table 5.2 Variations in the Time Available for Synchronization in the Reconfigurable Token Ring Control across Temperature Regions @V_{DD} = 1.0 V and a TN/TP Transistor Process 85
Table 5.3 XBAR Select Signals 92
Table 5.4 Nearest Neighbor Checking Algorithm 93
Table 6.1 Read Sequence Generated by Equation 6.1 per Cycle, where $i < j < 2i$ 104

Table 6.2 Effect of the Connectivity of the Serializing Multiplexer (MUX) on the Time Available for Synchronization in the Device . . 107

LIST OF ALGORITHMS

Figure 1 Sub-optimal Distribution Algorithm for a Static Token Ring (Part 1) 46
Figure 2 Sub-optimal Distribution Algorithm for a Static Token Ring (Part 2) 49

LISTINGS

Listing 1 MATLAB .PWL Cadence Vector File Generator Core 115
Listing 2 MATLAB .PWL Cadence Vector File Generator Sample Input 116

ACRONYMS

CSC	Complete State Code
DC	David Cell
DEMUX	Demultiplexer
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
GALS	Globally Asynchronous Locally Synchronous
LUT	Look-Up Table
MCE	Muller C-Element
MTBF	Mean Time Between Failures
MUTEX	Mutual Exclusion Element
MUX	Multiplexer

NoC	Network-on-Chip
PLL	Phase-Locked Loop
PN	Petri Net
PRBS	Pseudorandom Binary Sequence
RG	Reachability Graph
RO	Ring Oscillator
RTC	Relative Timing Constraint
SG	State Graph
SoC	System-on-Chip
STG	Signal Transition Graph
TS	Transition System
UCOM	Unidirectional Communication
VHDL	VHSIC Hardware Description Language
VCO	Voltage Controlled Oscillator
VTC	Voltage Transfer Characteristic

Part I

PHD THESIS

1 INTRODUCTION

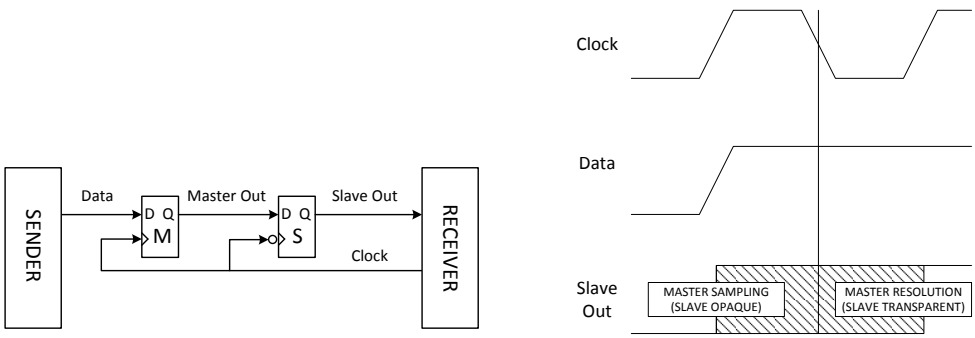
1.1 MOTIVATION

Synchronization is a prominent issue in the design of modern digital systems. As the diversity of components, clock frequencies, and voltages on a chip increases, so does the importance of synchronization in order to reliably pass data across the regions of variation. Both the methodologies of Globally Asynchronous Locally Synchronous (GALS) signaling, and Network-on-Chips (NoCs) require synchronization. The former case requires the presence of an asynchronous wrapper in order to reliably pass data between two clock regions, which must then be synchronized at each end of the transfer [10]. While in the latter case, the interconnection network is shared between devices via the use of packets, and synchronization is necessary in order to perform the handshakes between the network and the system resources [7].

A common method for performing synchronization across two clock domains involves the use of a cascaded flip-flop synchronizer. Let us begin with a brief exploration of such a synchronizer, while highlighting the limitations and shortcomings of the same to bring the primary issues discussed in this thesis into sharp relief. A basic two-flop synchronizer, as shown in Fig. 1.1(a), operates by using pair of flip-flops in a master-slave configuration to synchronize data items. A synchronization operation, as shown in 1.1(b) consists of two phases: sampling, and resolution.

During the sampling phase, when the *Clock* signal is high, the master latch samples *Data* items, while the slave latch remains opaque to changes. When the *Clock* signal transitions from high to low, the synchronizer enters the resolution phase, and master latch becomes opaque to changes, while the slave latch becomes transparent. During the resolution phase, the output signal from the master latch is pulled to either the supply voltage level or ground from an initial analog voltage level (i.e., it becomes a digital output signal). When the *Clock* signal transitions from low to high, the synchronizer re-enters the sampling phase and the output of the slave latch becomes opaque, thus retaining the resolved value from the previous sampling phase (i.e., it transmits the prior value from the master latch).

With the prior explanation in hand, we can now address the shortcomings of this design. When the *Data* input to the master latch changes state "sufficiently close" to the *Clock* transition marking the end of the sampling phase of the master latch, it results in longer than normal resolution times. If the output from the master latch does not resolve to a clear digital high or low before the end of the resolution phase (i.e., when the slave latch becomes opaque) the synchronization operation fails. The reliability of a two-flop synchronizer is thus directly related to the resolution time allotted to the synchronization operation, which is itself dependent on the duty cycle. Moreover, it will be shown in Chapters 2 and 3 of



(a) Schematic (b) Waveform

Figure 1.1: Cascaded Two-Flop Synchronizer

this thesis that the relationship between the failure rate of the synchronizer and the time available for synchronization is exponential.

Because of the aforementioned relationship, the reliability of the synchronizer can be impacted by variations in the duty cycle of *Clock* signal. Of specific note are:

1. Variations in the delays of the CMOS transistors which comprise the *Clock* signal, due to process, voltage, and temperature dependencies.

2. Variations in the duty cycle due to the presence of multiple operating modes.

Typically, a cascaded flip-flop synchronizer is designed to meet the requirements of the worst-case operating mode only (i.e., the minimum time available for synchronization). In most cases, additional flip-flops are placed in series with the original two-flop master-slave latch pair to accomplish this. However, as additional flops are added, modelling the reliability characteristics of the synchronizer becomes more difficult [42].

By integrating a combination of parallelism and task scheduling, known as wagging, into the design of a cascaded flip-flop synchronizer, as shown in Fig. 1.2, the worst case reliability requirements mentioned above can be met while using simpler two-flop equation models. Furthermore, if the parallelism and task scheduling in the cascaded flip-flop synchronizer of Fig. 1.2 can be altered by utilizing the practices of reconfigurable hardware design, the reliability characteristics of the synchronizer can be dynamically changed on an as needed basis via the use of control codes.

The parallel synchronizer of Fig. 1.2 operates by sharing the input data from the transmitter between the parallel master-slave flip-flop pairs, while the controller generates the Clock/Enable (CTR) signals used to schedule the sampling and resolution phases of each flip-flop pair. Resolved data items from the parallel master-slave flip-flop pairs are then merged back together and sent to the receiver. The tasks are shared evenly between master-slave flip-flop pairs, which results in an increase in the time available synchronization that is directly proportional to

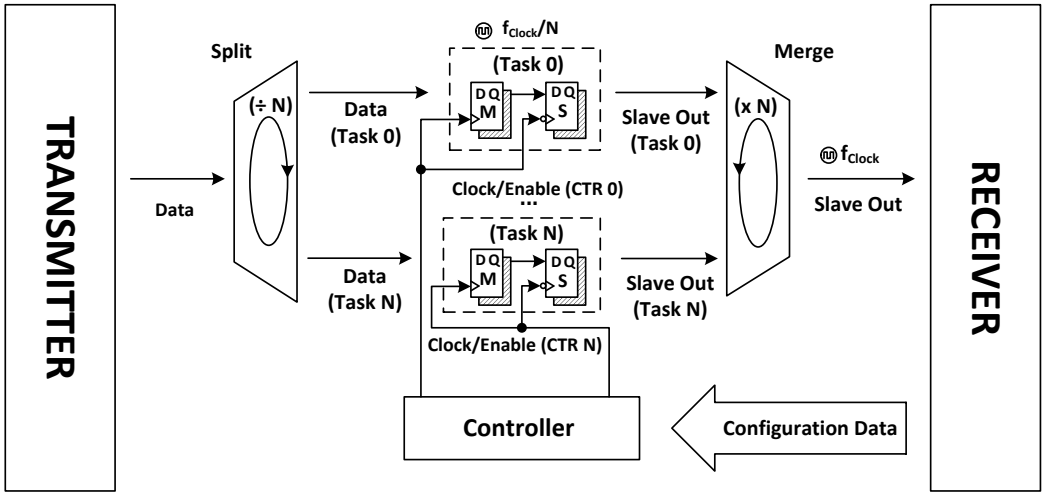


Figure 1.2: Parallel Two-Flop Synchronizer

the number of parallel flip-flop pairs used, which will be discussed in greater detail during Chapter 3. When comparing the synchronizer design of Fig. 1.2 to that of Fig. 1.1, the trade-offs to are divided into two categories:

1. Delays due to additional hardware components along the datapath from the transmitter to the receiver.

2. Power and area overheads due to the control hardware, replicated master-slave latch pairs, and additional hardware components along the datapath.

Achieving the aforementioned design goals of reconfigurability and parallel synchronization requires that the issues of scalability, reliability, and fault tolerance be addressed within the context of the control hardware. Scalability issues, within this context, stem from the sharing of devices within a multi-mode reconfigurable controller. More specifically, it is necessary to maintain a limit on the maximum fan-in or fan-out of the control hardware regardless of the number of configurable modes available in the system. Reliability issues, within this context, refer to the protocols by which configurable modes are changed and functional correctness is guaranteed. Fault tolerance issues, within this context, concern how to route around faulty devices within the controller, which is useful as devices may be shared between more than one configurable mode. Additionally, if the transmitter and receiver are decoupled from each other via the use of a First-In First-Out (FIFO) buffer, as in prior literature by Chelcea, then data flow issues, such as data accumulation of starvation within the buffer, must also be considered as well [15].

With that in mind, the central aspects of this thesis will focus on the following issues:

1. Limitation of capacitive loading within the devices which comprise a multi-mode reconfigurable controller (Chapter 3).

2. Development of an asynchronous reconfiguration protocol which guarantees the functional correctness of the reconfigurable controller (Chapter 4).

3. Methods for excluding faulty components within the reconfigurable controller (Chapter 5).

4. Amelioration of data accumulation/starvation in parallel synchronizer incorporating a FIFO buffer (Chapter 6).

1.2 CONTRIBUTIONS

This thesis presents and explores a framework for the design of reconfigurable control devices for parallel synchronizers through the use of distributed embedded graphs. Core to the work are the control of metastability within a parallel synchronizer through the application of one-hot codes, the issues of fault-tolerance within the parallel control structure via the use of bypass paths, as well as an of how the wagging paradigm affects the behavior of a FIFO synchronizer, and how to overcome the common pitfalls of data accumulation and reduced MTBF at higher clock frequencies.

It functions as a natural extension of work on parallel synchronization previously presented by Jex and Dike in 1995, Suk-Jin et. al in 2004, and work on wagging synchronization that was presented in 2010 by Alaskih [41] [43] [2]. Principle to this work are the following topics:

1.2.1 *Algorithms for Load Balancing in Token Rings via Distributed Embedded Graphs*

A basic algorithm for distributing the number of adjacent connections in an embedded cycle graph which models a multi-cycle token ring construct with several cycles is presented. Thereafter, It is optimized to improve the redundancy inherent to the graph and reduce overhead. Both the sub-optimal and optimized algorithms place concrete limitations on the maximum adjacency possible at a given vertex.

Such algorithms are useful in minimizing the capacitance and fan in/out of the connections of the resulting physical token ring implementation. As an example, embedded token rings based on ring oscillators, Muller pipelines, and David cells are simulated in UMC 90nm technology using CADENCE in Section 3.4. Abstract versions of the algorithms are also modeled in C++ in Section 3.3 [56] [21].

1.2.2 *Manipulation of Parallelism in a Self-Timed Reconfigurable Control Device via One-Hot Coding*

A design for a controller device is presented which is suitable for manipulating the parallelism contained within a token ring based on embedded graphs. Unidirectional Communication (UCOM) threading is shown to be useful in exerting control over asynchronous circuits based on a cyclic graph composed of independent loops, where the individual loops already possess Complete State Code (CSC). Merging of output data from multiple interrupt devices through the use of K-maps is shown to be essential in linking the interrupt devices together.

This method is useful in that it gives the designer direct control over the time available for synchronization inherent in a synchronizer composed of multiple par-

allel cascaded flip-flops at the cost of additional hardware. CADENCE simulations charting the performance of the control device across various process and temperature parameters in a UMC 90nm technology are used as a proof of concept.

1.2.3 *Tolerance of Hard Faults in a Self-Timed Reconfigurable Control Device via Bypass Paths*

The specification of the basic control device is modified by incorporating bypass paths into the underlying behavior graph. This allows faulty nodes to be excluded from the system, thereby facilitating recovery from hard faults at the cost of reduced performance. The concept of UCOM thread forwarding is introduced in order to ensure that the underlying Signal Transition Graph (STG) of the control device remains valid if a path is bypassed. The consistency of the graph is analyzed using a nearest neighbor checking algorithm and completion detection.

The utility of this method lies in overcoming the limitations of a Johnson counter, which tends to suffer either state-space locking in the case of stuck-at faults or unpredictable behavior in the case of intermittent hard faults [52]. This process is evaluated through simulations on both Petri Net (PN) and STG models of the system in Workcraft, as well as circuit simulations in CADENCE.

1.2.4 *Flow Control in a FIFO Synchronizer based on Wagging*

A VHDL analysis of a FIFO synchronizer based on wagging is tested in Xilinx using automated test methods such as PRBS input vectors, and signature analysis. It is shown that while the MTBF can be manipulated via the use of shift register control signals, care must be taken when the parallel lines are serialized or the designer might not see the anticipated MTBF gains of the wagging method.

1.3 ORGANIZATION OF THESIS

The thesis is organized as follows:

Chapter 1 "Introduction." Describes the motivation, contributions, and organization in the thesis.

Chapter 2 "Background Literature & Models." Presents a review of the relevant theory on asynchronous circuits and metastable events. Discusses conceptual models useful in understanding the content of subsequent thesis chapters such as PNs, STGs, and embedded cycle graphs.

Chapter 3 "Overview of Wagging Synchronization." Presents a brief review of prior synchronization literature. Introduces the concept of synchronization via wagging, and discusses its effects on relevant synchronization parameters. Introduces the abstraction of a token ring, and compares various combinational and sequential circuit topologies suitable for the realization of the same. Applications of embedded cycle graphs to arbitrary token rings are also discussed, which focus

on the algorithms for the balanced distribution of edges in the ring construct.

Chapter 4 "Parallelism in a Reconfigurable Controller for Wagging Synchronization." Presents a brief review of the concepts of arbitration and reconfigurability, as it pertains within the context of a token ring based on embedded graphs. Defines a protocol for reconfiguration based on STGs which uses both interrupts and one-hot codes to control the parallelism in a reconfigurable controller device via the selection of valid embedded configurations within the behavior graph. Illustrates how UCOM threads can be used to control both the firing order and end behavior of independent interrupt devices. Addresses issues such as the reachability of different embedded specifications, the process by which the output data from separate interrupt devices is merged together, and the optimization of the one-hot control data via encoding to limit the number of input signal lines to a logarithmic growth function. A basic controller design is simulated across 5 process corners in a CADENCE UMC 90nm technology, and the results are presented as a proof of concept.

Chapter 5 "Robustness in a Reconfigurable Controller for Wagging Synchronization." Explores how the fault tolerance of the parallel controller device in Chapter 4 can be improved by excluding faulty nodes from the embedded behavior graph, as well as touching on the limitations of prior work regarding Johnson Counters. The STG of Chapter 4 is extended through the incorporation of bypass paths, which are necessary for the exclusion process. The concept of UCOM thread forwarding is defined, which is used to maintain the validity of the underlying STG in the presence of such bypass paths. A PN model is used to simulate the reconfiguration and selection process of the system via Workcraft. A nearest neighbor checking algorithm is also presented, which is used to validate the consistency of the cycle graph produced via reconfiguration and chordal bypass paths.

Chapter 6 "Flow Control in Wagging Synchronizers incorporating FIFO Buffers." Analyzes the flow control issues present in a wagging synchronizer which utilizes a FIFO to decouple read and write operations from each other. Introduces key design considerations in the FIFO synchronizer architecture. Discusses the VHSIC Hardware Description Language (VHDL) design flow and test bench used to simulate and verify the flow control algorithm in the chapter.

Chapter 7 "Conclusions." The work in the thesis is summarized, and future research directions are discussed.

2 BACKGROUND LITERATURE & MODELS

2.1 INTRODUCTION(ASYNCHRONOUS CIRCUITS)

While the design and implementation of asynchronous circuits has been a topic of discourse for several years now and a complete overview is the subject of a textbook in and of itself, the work explored in the later body of this thesis is best served by providing a general overview of the main concepts which both motivate and underpin the concepts that will be discussed later, punctuated by a few well chosen examples. In the introductory section of this chapter the benefits, drawbacks, and differences between standard synchronous and asynchronous controller design are assessed. Major building blocks in asynchronous design used in the later portions of this thesis will also be presented. In Section 2.2, the theoretical concept of Petri nets will be presented with emphasis on how these nets can be used to model the behavior of asynchronous systems. Continuing onward, Section 2.3 will link the concept of a Petri net with that of a signal transition graph, which are used to further describe the internal workings of an asynchronous specification. Section 2.4 will discuss the concept of metastability, which is useful for describing the failure behavior of the synchonizer circuits covered in later chapters. Finally, Section 2.5 will cover the concept of an embedded cycle graph, which is a useful construct for handling the designs presented in the further chapters of this work.

2.1.1 *Properties of Asynchronous Circuits*

Asynchronous circuits function in the absence of a global time reference which is a hallmark of standard synchronous circuit design. In synchronous circuit designs, glitches are allowed at all times during circuit operation excepting the times at which a clock transition occurs and all signals are assumed to be stable. Only knowledge of the final system state and the time allotted to complete its operation are required. By contrast, asynchronous circuits require knowledge of not only the final state of the system *but also the partial states as well*, in addition to a requirement that all transitions be glitch-free during the entire time continuum of its operation. For this reason, asynchronous designs do not generally enjoy the ubiquity of their synchronous counterparts. However, asynchronous circuits do possess certain advantageous properties which merit consideration.

Modularity From the perspective of the individual circuit, having a global clock simplifies the design process. However, this requirement becomes restrictive as the number of transistors on a chip increases. In a synchronous system, both the phase and frequency of the global clock are assumed to be known quantities. However, the arrival time of the clock observed by separate components in the system varies from one end of the chip to the other, due to both the spatial distance of the components from the distribution network itself, as well as differences in both the

passive interconnect and active device parameters [22]. This variance is referred to as the *clock skew* of the system, and it places constraints on both the maximum and minimum permissible operating frequency [29].

Because of the aforementioned timing constraints and the tightly interconnected relationship between the clock distribution network and intellectual property (IP) blocks present in a large scale synchronous designs, frequency scalability and reuse are not inherent features. Initiatives in previous years dealt with how to increase the plug and play characteristics of synchronous designs, such as creating standardized interfaces for Virtual Components [1]. By contrast, modularity is an inherent property of asynchronous circuits. Asynchronous blocks communicate with each other through protocols called *handshakes*, which will be discussed in more detail in subsection 2.1.2. No timing assumptions exist between the blocks, and as a consequence the asynchronous blocks can be frequency scaled and reused elsewhere without loss of functionality.

In more recent years, asynchronous design practices have been adapted in efforts to improve the modularity between synchronous designs. In one case, the modularity of System-on-Chip (SoC) designs can be improved by constructing the system with several different voltage/frequency islands (each with their own local clock) and then *synchronizing* the data items between regions using handshakes. While the topic of synchronization will be covered in Chapter 3, this principle forms the basis of the GALS signaling paradigm. GALS requires the presence of an asynchronous wrapper in order to reliably pass data between two clock regions, which must then be synchronized at each end of the transfer [14].

Continuing forward on the point above, there has also been a drive to partially decouple the component blocks in a SoC from the clock distribution network, as in the NoC paradigm [20]. For reference, a NoC routes transmitted data items as packets (called flits) along the wires of a homogeneous interconnection network, and carries out data transfers between the interconnection network and the local components using a network interface circuit. This network interface handles synchronization and flow control at the endpoints, while arbitration placed at regular intervals along the interconnection network handle the routing and flow control of flits across the chip [20].

Both of these examples serve to illustrate that modularity is a major design concern, and as stated above asynchronous circuits and design methods are well suited to handling it.

Power The advantageous power considerations which factor into circuits built from asynchronous designs can be broken down into two general categories.

The first category is tied to the discussion on modularity above. More specifically, in systems where no global clock exists and each locally clocked module communicates system-wide via asynchronous handshakes (i.e. GALS) there is also no global power consumption attributed to a system wide clock distribution network. The clock distribution network can have a significant impact on the global power consumption of a system. As an example, 33% of the system power in the Intel Itanium 2 processor was consumed by the clock distribution network back in 2002, which serves to highlight the statement above [59]. More recently in 2005,

this emphasis on power consumption led to the development of an asynchronous NoC which demonstrated reduced power consumption in the clock distribution network when compared against a synchronous design [7]. The only drawback is the additional complexity incurred as a result of utilizing asynchronous design methods [53].

The second power consideration is the reduced electromagnetic emissions of asynchronous designs. To clarify, the switching tranitions of control and data signals in a synchronous system are often correlated to the clock itself, which lead to draws in the supply current at the frequency of the clock signal and related harmonics [78], [18]. These harmonics impact the frequency domain characteristics of the system and can prove detrimental in applications where such characteristics must be precise [78]. As asynchronous designs are event-driven systems, no such harmonics are present which lead to reduced emissions in the frequency domain.

It is the former category that has garnered more attention in recent years though, with self-timed microprocessors like the ARM AMULET series becoming prevalent in low power designs where asynchronous processing is prevalent [31], [33], [32]. However, it should be noted that the dynamic power consumption savings incurred as a result of asynchronous design methods, must be weighted against the increased standby power consumption resulting from the increased complexity of asynchronous implementations.

Latency and Throughput The final distinctive property that separates logical circuits implemented using synchronous methods from their asynchronous counterparts is the difference between the global latency and throughput characteristics of each respective implementation. To clarify, the modeling the latency and throughput behavior of a synchronous design relies on computing the *worst-case latency* of the system, and attaching worst-case maximum (and sometimes minimum) frequency requirements to the finished product, thereby specifying where the design can be applied. Asynchronous designs, by contrast, only rely on the *average-case latency* between their components. Due to the aforementioned modularity property, as long as the asynchronous communication protocols connecting the modules together are satisfied functional correctness of the system is maintained. To be succinct, the latency and throughput of synchronous design is limited by the slowest component in the system, while an asynchronous design is limited by the average delay of all the components in the system at the cost of possibly unbounded circuit delays. Appropriately, the differences between latency and throughput of synchronous and asynchronous designs are mirrored in the control methodology discussed in the next subsection.

2.1.2 *Asynchronous Control Circuits*

Conveying the distinctions between asynchronous control circuits and their synchronous counterparts, first requires a brief overview of the structural characteristics of a sequentials circuits composed using synchronous methods. The abstract model segregates the design into two parts, which are as follows:

• **data path** - refers to the individual blocks within a system that perform manipulations and transformations on the data they receive. These blocks can either perform arithmetic operations (e.g., adders, multipliers), or communication and storage operations (e.g., muliplexers, registers).

• **control** - refers to the signals that determine the data used by each block (e.g., multiplexer select signals), or the operations that each block performs (e.g. ALU operation codes).

In a synchronous design, the clock signal acts as a trigger for all of the control and data path operations that occur within a given clock period. As long as a final state is reached within a single clock cycle functional correctness is assured. From a control perspective, it can be said that the control within the system is centralized, and synchronization is implict across all blocks in the system. By contrast, in an asynchronous design control is distributed, as no global clock is present. Each block only synchronizes with its relevant neighbors, regardless of what other blocks in the system are doing. Because of this, the behavior of the entire system must be modeled with these concurrent executions in mind. The communication between neighboring blocks is accomplished via a handshaking protocol, which is composed of two parts:

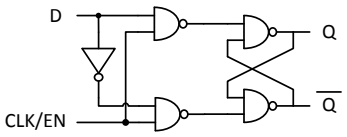
• **request** - refers to a signal issued from the environment to the input of a block indicating that data is ready at that device operation is requested.

• **acknowledge** - refers to a signal issued from the output of a block to the environment indicating that the device has finished its operations and that the data results are stable.

The weaknesses of this control method stems from the increased number of data lines used to implement completion detection, as is the case with delay-insensitive coding styles such as a *dual rail* code [70]. Alternatively, the delay of individual lines can be adjusted to ensure that the propagation delay of signals lines in the system exceeds the worst case combinational operation of any given block [18]. The later method is less robust because of the additional timing assumptions that hinder the modularity and portability of the design. The only other pitfall is the requirement that the signals are free of *hazards* (i.e., deviations from specified circuit behavior).

2.1.3 *Asynchronous Circuit Primitives*

Given the discussion above, we can now examine some of the fundamental sequential circuit primitives that will be used throughout the course of this work. Two major primitives common to asynchronous controller designs are readily apparent.

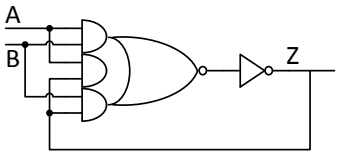


(a) Schematic

D	CLK/EN	Q	$\overline{Q}$	$Notes$
X	0	Q_{prev}	Q_{prev}	No change
0	1	0	1	Reset
1	1	1	0	Set

(b) Truth table

Figure 2.1: Behavioral representation of a D latch



(a) Schematic. From [51].

A	B	Z	$Notes$
0	0	0	Reset
0	1	Z_{n-1}	No change
1	0	Z_{n-1}	No change
1	1	1	Set

(b) Truth table.

Figure 2.2: Behavioral representation of a Muller C-Element (MCE)

2.1.3.1 *Latches and Flip-flops*

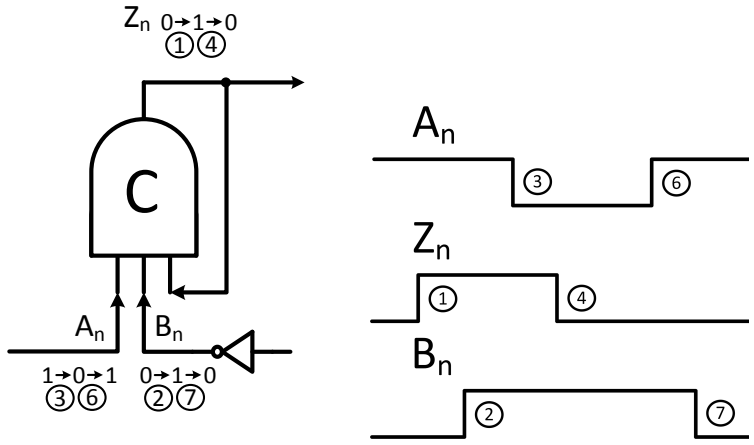
Latches A latch is defined as a level-sensitive circuit which is used to transmit and retain state information in a sequential process, as shown in Fig. 2.1(a). The state of the latch is altered when a differential voltage is applied to the inputs of the two cross-coupled NAND gates in the circuit. When the enable signal (CLK/EN) is high, the output signal of the latch (Q) follows the input signal (D). When the enable signal is low, the latch output retains its previous value. The truth table illustrating this process is shown in Fig. 2.1(b).

Flip-flops When two latches are placed in series, they form what is known as a *flip-flop*. The most prevalent example of a flip-flop is a D flip-flop (though others exist, such as the JK flip-flop), where the two latches forming the flip-flop are triggered on opposite edges of the clock in what is commonly known as a master-slave configuration [66]. Depending on how it is used it can either function as a data storage element, or as a synchronization element. In this work, we will focus on the latter application.

2.1.3.2 *Muller C-Elements*

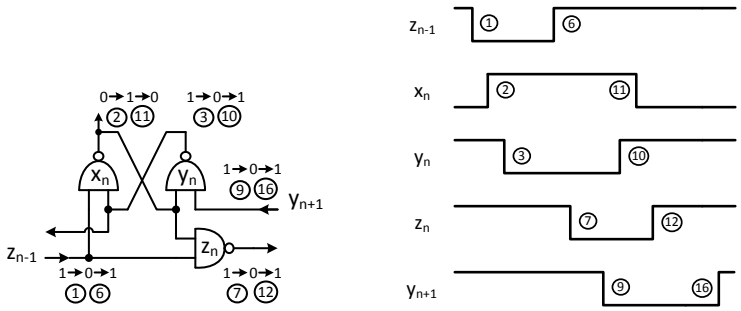
Muller C-Element (MCE) A MCE is a circuit that changes its state information only when all of its signal inputs (A, B, Z) are either logic high or logic low values, as in Fig. 2.2(a). All other input combinations result in the circuit outputting its last valid state. The truth table illustrating this process is shown in Fig. 2.2(b).

The operation of an individual cell is defined by both the output signal that the cell produces (Z_n), and the input signals to the cell ($A_n(Z_{n-1})$, $B_n(\overline{Z_{n+1}})$) which are generated by adjoining MCEs. Fig. 2.3(a) depicts a schematic of an individual MCE, where the circled numbers in the figure represent the individual transitions generated internal to the cell, starting at transition 1. External signals generated by



(a) Annotated Symbol. From [51]. (b) Timing Diagram.

Figure 2.3: Annotated Muller C-Element schematic with signal transitions representing 1 operating cycle



(a) Standard DC. From [21] (b) Timing Diagram.

Figure 2.4: Annotated David Cell schematic with signal transitions representing 1 operating cycle

the adjoining MCEs are shown in Fig. 2.3(a), and also appear in the timing diagram of Fig. 2.3(b).

2.1.3.3 *David Cells*

David Cell (DC) A DC, as shown in Fig. 2.4, can be thought of as a distributed circuit, where the state information or "token" is stored within the pair of complementary stable states formed by the cross-coupled NAND gates in the cell. If the CTR is high, then a token is present, and vice-versa if a token is absent.

The operation of an individual cell is defined by both the output signals that the cell produces (y_n, z_n), and the input signals to the cell (z_{n-1}, y_{n+1}) which are generated by adjoining DCs. Fig. 2.4(a) depicts a schematic of an individual DC, where the circled numbers in the figure represent the individual transitions generated internal to the cell, starting at transition 1. External signals generated by the adjoining DCs are shown in Fig. 2.4(a), and also appear in the timing diagram

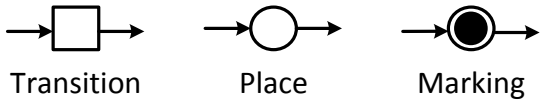


Figure 2.5: Petri Net Symbols

of Fig. 2.4(b). For reference purposes, in Fig. 2.4(a) these unmarked transitions are {4}, {5} for the previous DC in the token ring chain, and {8}, {13} for the next DC in the chain.

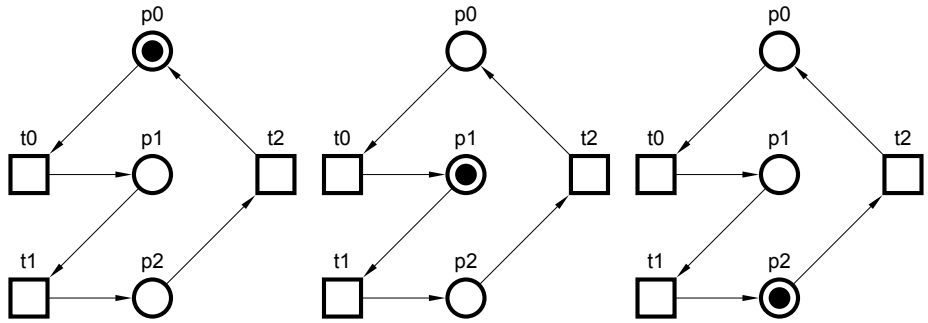
It should be noted that the "token" within the DC of Fig. 2.4(a) is atomic (i.e., the token is only "present" in a single cell at any given time). However, this does not have to be the case. In fact, the number of transitions which are required to complete the operation of an individual DC can be reduced if the atomicity of the token is violated and it is allowed to simultaneously exist in multiple DCs of the token ring. This concept has been referred to in prior literature as *token spreading* [73].

2.2 PETRI NETS

Accurately modeling and specifying the behavioral characteristics of an asynchronous circuit is of vital importance to the design process of the same. In a traditional synchronous design, the behavior of the system is only analyzed around the rising and falling edges of a clock signal. The periodicity between the successive rising or successive falling edges of the clock signal is referred to as the *duty cycle* of the circuit. The inputs and outputs of the system are allowed to vary at any point in between these edges, but are assumed to resolve prior these evaluation points. By contrast, in an asynchronous design, the behavior of the system must be known for the entire time continuum of the duty cycle. As a consequence, the design of an asynchronous circuit is often more complex than the design process of any synchronous counterpart.

As a consequence of the design challenges listed above, the design flow needed to visualize the system as a whole is best done using a formal model known as a *Petri Net (PN)*. This model was originally developed by Carl Adam Petri in 1962, where he depicted the behavior of a system in terms of concurrent events, the causalities between these events (called transitions), and the dynamic state of the net itself (called markings) [63], [58], [62]. Where a traditional *finite state machine* (FSM) depicts the behavior of a synchronous digital circuit using a single global state (i.e., all actions take place on the rising or falling edge of a global clock signal), Petri Nets define the behavior of an asynchronous system as a composition of the local states in the system which alter themselves via synchronization and communication.

Let us define the symbols used in this formal model, which are depicted in Fig. 2.5. Informally, the *transitions* in the PN represent the events in an asynchronous system. *Places* can be thought of as the conditions necessary for these events to occur. If a place contains a *token* (i.e., if it is *enabled*) it is said to have a *marking*.



(a) Initial state. (b) Firing transition t0. (c) Firing transition t1.

Figure 2.6: Petri Net of a 3-cell token ring

The global state of the asynchronous system is composed of all the marked places in the PN. With that in mind, the behavior of the PN can be thought of as a token game, where the markings change position (state) according to the enabling and firing rules in the net.

In the remainder of this Section, the formal definitions for the basic properties and behavior of a generic PN will be covered. Furthermore, the PN a 3-cell token ring, as shown in Fig. 2.6, will be used to provide additional insight.

Definition 2.1 A Petri Net is a quadruple, $PN = (P, T, F, m)$ where:
P is a finite set of places,
T is a finite set of transitions: $(T \cap P = \emptyset)$,
F: $(T \times P) \cup (P \times T) \to \mathbb{N}$ is a flow relation,
m: $P \to \mathbb{N}$ is the marking (or state) of the net.

These nets are governed by a series of rules which will be shortly codified in the following subsections.

2.2.1 *Pre-sets and Post-sets*

Definition 2.2 The *pre-set* of a transition t, as denoted by $\bullet t$, is the set of places $p \in \bullet t \subseteq P$ such that $p \in \bullet t \Rightarrow F(p,t) > 0$. Symmetrically, the pre-set of place p, denoted by $\bullet p$, is the set of transitions $t \in \bullet p \subseteq T$ such that $p \in \bullet t \Rightarrow F(t,p) > 0$. Similarly, the *post-set* of a transition (or place) x, denoted $x\bullet$, is the set of all places (or transitions) y, such that $y \in x\bullet \Rightarrow F(x,y) > 0$.

Speaking less formally, the post-set of a transition corresponds to all of its output places, while the pre-set corresponds to all of its input places via similar arguments. As an example, in Fig. 2.6(a) $\bullet p0 = t2$, and $p0\bullet = t0$

2.2.2 *Enabling and Firing*

Definition 2.3 A transition $t \in T$ is *enabled* at marking m_1 if for any place $p \in \bullet t$, $m_1(p) \geqslant F(p,t)$. This enabled transition may produce a new marking, m_2, by *firing*

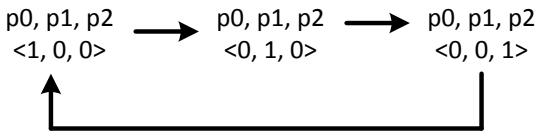


Figure 2.7: Reachability Graph of a 3-cell token ring

such that:

$$\forall p \in P : m_2(p) = m_1(p) - F(p,t) + F(t,p),$$

where $+$ and $-$ are defined component-wise, as denoted by $m_1 \xrightarrow{t} m_2$ or $m_1 \rightarrow m_2$. Thus for each $p \in {\bullet}t$, firing t subtracts $F(p,t)$ tokens from ${\bullet}t$, and adds $F(t,p)$ tokens to $t{\bullet}$ for each $p \in t{\bullet}$.

In the 3-cell token ring of Fig. 2.6(a), the only transition enabled in the Petri net is at t0 due to the presence of the token at place p0. After firing, the token marking is subtracted from place p0 and then added to place p1 using the flow relation specified above.

2.2.3 *Reachability*

Definition 2.4 The new marking m_2, produced from the firing of m_1, can enable further transitions. Consequently, we can now define sequences of transitions that are *reachable* from the firing of an initial marking, denoted m_0. These transition sequences are referred to as *traces*. The set of all markings reachable from marking m is denoted as $[m\rangle$. The set of all markings reachable from the initial marking of the PN, denoted $[m_0\rangle$, defines the *Reachability Graph (RG)* of the net, which consists of nodes corresponding to the reachable markings in the graph, and edges that correspond to the firing transitions between pairs of markings. Two nodes, m_i and m_j, in the graph are connected by an arc if $m_i \rightarrow m_j$ holds.

The RG of the 3-cell token ring depicted in the PN of Fig. 2.6, is illustrated in Fig. 2.7. The markings in the RG use vector encodings of the form $\langle p0, p1, p2\rangle$. From the initial marking $\langle 1,0,0\rangle$ the RG is traversed as follows $\langle 1,0,0\rangle \xrightarrow{t0} \langle 0,1,0\rangle \xrightarrow{t1} \langle 0,0,1\rangle \xrightarrow{t2} \langle 1,0,0\rangle$. When t2 fires, the PN returns to its initial marking. However, as there are no concurrent paths in the original PN, this particular RG is only useful in a pedagogical capacity.

2.2.4 *Other Petri Net Properties*

Boundedness

The behavior of a circuit specifcation must to be finite. A PN is called *bounded* if the net only accumulates a finite number of tokens at any given place. If the number of tokens accumuated is equal to k, then the PN is called *k-bounded*. If k is

equal to 1 then the PN is called *safe*. Due to the binary nature of signal transitions in many digital circuits, 1-safe nets are the nets that are the most applicable to logical synthesis.

Deadlocks If a firing pattern results in a marking where no PN transitions are enabled, it is referred to as a *deadlock state*. If no such reachable markings exist, the PN is called *deadlock-free*.

2.3 SIGNAL TRANSITION GRAPHS

While the description of a PN above yielded a basis for formal description of an asynchronous circuit, in the following sections another model for the description of an asynchronous circuit known as a Signal Transition Graph (STG) will be covered. While many of the same properties are shared between STGs and PNs, the following are of particular note [18]:

1. Rules for the enabling and firing of transitions.
2. Notions of reachable markings and traces.
3. Temporal relationships between transitions (i.e., choice, conflict, precedence, and concurrency).

However, the transitions in a STG are defined in terms of the rising and falling edges of a binary signal, rather than in terms of the presence or absence of a marking (token) in the pre-set or the post-set of a choice place of a PN. It is precisely this distinction that allows automated tools such as PETRIFY or Workcraft to derive complex gate equations necessary for logical synthesis from an asynchronous STG specification, but render them unable to do so when the same system is specified using a PN.[17] [18] [65]

First, Section 2.3.1. will provide a brief overview of the relationship between STGs and PNs, while Section 2.3.2 will provide a similar overview for State Graphs (SGs) and RGs. Next, Section 2.3.3. will cover the STG concept of *CSCs*. Finally, Section 2.3.4 will briefly touch on the concept of *Relative Timing Constraints (RTCs)*.

2.3.1 *Relation to Petri Nets*

Definition 2.5 A *Signal Transition Graph (STG)* is a triple $G = (PN, X, \lambda)$, where:
- PN is a Petri net $PN = (P, T, F, m)$,
- X is a finite set of binary signals, which generates a finite alphabet $A^X = X \times \{+, -\}$ of signal transitions,
- $\lambda : T \to A^X$ is a labeling function.

Let us briefly unpack this definition. The first and third bullet points that indicate that the transitions, T, in the PN are replaced using the labeling function, λ. The second bullet point indicates that each transition in the old PN is split into a positive (rising edge, +) transition and negative (falling edge, -) transition.

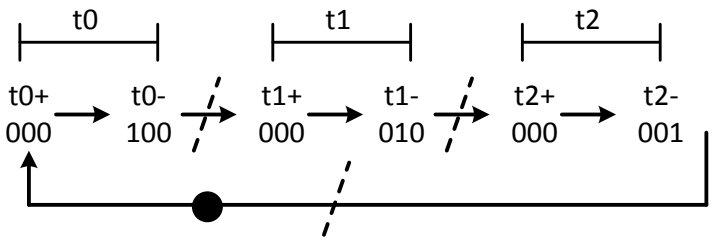


Figure 2.8: Annotated STG of a 3-cell token ring

Highlighting the differences between STGs and PNs is best illustrated with a concrete example, and the annotated STG of Fig. 2.8 will serve in this capacity. As with the PN of Fig. 2.6, the STG will represent the a 3-cell token ring. The *initial marking* of the STG is indicated by the black dot on Fig. 2.8, while dotted lines represent *CSC conflicts* (i.e., points of identical state encoding).

2.3.2 *State Graphs and Reachability Graphs*

A STG is also known as an interpreted PN, which is to say the transitions in the PN are labeled according to definition 2.5. Similarly, a SG is a RG with interpreted automata, which means that the markings and transitions in the RG have a 1-to-1 mapping with the (binary) states and events of a SG. To understand this relationship, we it must first note that a SG is defined by a Transition System (TS), rather than a Net like in a RG. A transition system is defined as follows [18]:

Definition 2.6 A TS is a quadruple $TS = (S, E, T, s_0)$, where:
- S is a non-empty set of states,
- E is a set of events (E and S must be disjoint sets),
- $T \subseteq S \times E \times S$ is a transition relation, and
- s_0 is an initial state.

Note: The elements of T are referred to as the transitions of TS, and will often be denoted by $s \xrightarrow{e} s'$ instead of (s, e, s').

If S and E are finite, then the TS is finite. With the definition of a TS in hand, we can now formally define a State Graph [18]:

Definition 2.7 A State Graph (SG) is a quadruple $SG = (TS, X, \lambda_S, \lambda_E)$, where:
- $TS = (S, E, T, s_0)$ is a transition system,
- $X = X_I \cup X_O$ is a set of binary signals $X = x_1, x_2, ..., x_n$,
- $\lambda_S : S \to \{0, 1\}^{|X|}$ is a state assignment function,
- $\lambda_E : E \to X \times \{+, -\}$ is an event assignment function.

Within the context of the individual states, each node $s \in S$ in the SG is labeled with a binary vector (s(1),s(2), ..., s(n)) by λ_S, where $s(i) \in \{0, 1\}$, in an order which corresponds to the original order of the signals in X. In other words, s(i) refers

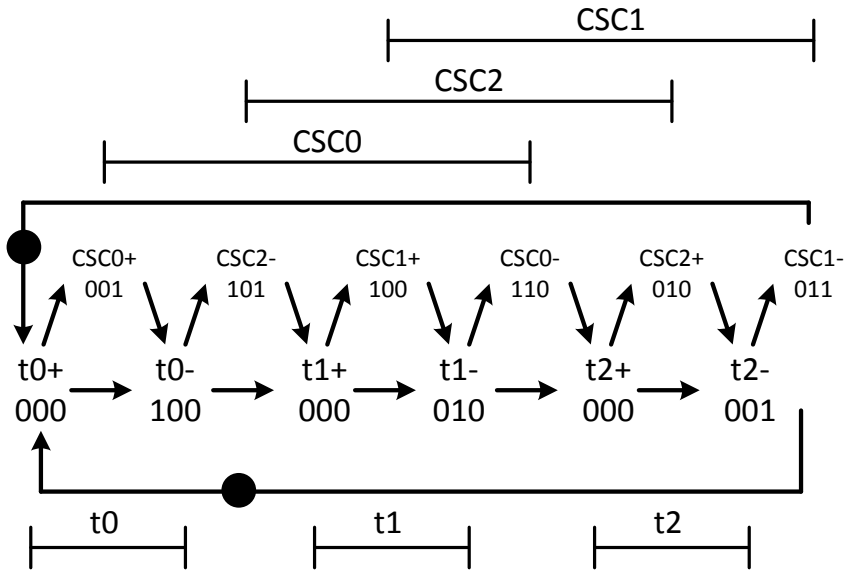


Figure 2.9: Annotated STG of a 3-cell token ring (CSC Conflict Resolved)

to the i-th component of s, which corresponds to the (binary) value of the signal $x_i \in X$. If all of the identifiers in the signals are unique, the notation $s(x) \in \{0, 1\}$ may be used to refer to the value of signal x in state$s$. Within the context of the transition arcs, let us note that $(s, x_i*, s') \in T$ refers to $(s, e, s') \in T$ and $\lambda_E(e) = x_i*$.

With those definitions in hand, we are now equipped to understand the relationship between a RG and a SG. A labeled RG of a particular STG $G = (N, m_0, X, \lambda)$, is formally a SG $SG = (TS, X, \lambda_S, \lambda_E)$, defined on the $TS = (S, E, T, s_0)$ that is generated by the full reachability analysis of Net N starting from the initial marking m_0. *Reachable markings of Net N* are used to obtain the *states* $s \in S$ *in SG*. *Transitions of N* that fire between corresponding markings are similarly used to obtain the *names of events* $e \in E$ *in SG*.

The state encodings of the 3-cell token ring are already listed in both Fig. 2.8, and Fig. 2.9, and all codes take the form $\langle s0, s1, s2 \rangle$. As will be shown in the following subsection, the state encoding of the STG will provide insight on hot to both detect and resolve CSC conflicts.

2.3.3 *Complete State Coding*

Definition 2.8 A state graph $SG = (TS, X, \lambda_S, \lambda_E)$, where $TS = (S, E, T, s_0)$, satisfies the *complete state coding* property if for every pair of states $s' \in S$ possessing the same binary code, $\lambda_S = \lambda_E$, the sets of enabled output signals are the same [16].

To paraphrase Definition 2.8, a SG contains CSC if there is no ambiguity as to the next-state behavior in the SG for any state. Observing the state encoding of the signals in the STG that comprise the 3-cell token ring in Fig. 2.8 provides us with insight into the nature of a *CSC conflict*. In the STG of Fig. 2.8, there are 3 separate instances where the state encoding not unique. Prior to transitions t0+, t1+, and

t2+, the binary state encoding ($\langle s0, s1, s2\rangle$) is $\langle 0, 0, 0\rangle$. As the state encoding for to+ ($\langle 1, 0, 0\rangle$), t1+ ($\langle 0, 1, 0\rangle$), t2+ ($\langle 0, 0, 1\rangle$) represent three *semantically different* (i.e., not equivalent) states which are reachable from the $\langle 0, 0, 0\rangle$ state encoding, there is an ambiguity in the next-state behavior of the SG. In other words, once the $\langle 0, 0, 0\rangle$ encoding is reached, the SG "can not decide" whether to fire t0+, t1+, or t2+, without additional information.

Because the encoding conflict at state $\langle 0, 0, 0\rangle$ stems from a lack of information about the next-state behavior of the SG, the conflict can be resolved by inserting new signals into the SG using bi-partition [18]. Let us examine Fig. 2.9 to get an understanding of how signal insertion can be used to resolve encoding conflicts within SG. As we can see, three distinct signal lines (CSC1, CSC2, and CSC3) have been added to the graph. In order to fire, the STG now requires information from the CSC signals. Note that at all three previously conflicting locations now have unique CSC signal combinations associated with them ($\langle 1, 0, 1\rangle$, $\langle 1, 1, 0\rangle$, and $\langle 0, 1, 1\rangle$). Also note that the state encoding of all CSC signals are unique as well (i.e., there are no CSC conflicts in the CSC resolution signals). Thus, we can say that the insertion of the CSC signals have resolved the encoding conflict in Fig. 2.9.

There is one more important point that bears mentioning: The interleaving of the CSC lines in Fig. 2.9 follows what is known as a *lock relation* (i.e., $A* \rightarrow B* \rightarrow A\bar{*} \rightarrow B\bar{*} \rightarrow A*$). It has been established if A and B are single-cycle signals, and a lock relation exists between A and B, then that is sufficient to guarantee Complete State Coding for A and B [49]. However, a because a lock relation is, by definition, a strongly connected sequence, it can also be used as a sequencer in between independent STGs which already possess Complete State Coding (i.e. asynchronous communication can be performed in the absence of a typical request/acknowledge handshake). Chapter 4.3 will utilize this principle to implement an asynchronous reconfiguration protocol.

2.3.4 *Relative Timing in Signal Transition Graphs*

One of the primary issues with STG synthesis, where all components in a system are assumed to have arbitrary delays, is the inefficiency of the resulting implementations. In an effort to ensure functional correctness, STG synthesis tools can factor in signal combinations into the final implementation which have no real physical meaning. In order to increase the efficiency of the resulting synthesis, timing assumptions can be employed which place limits on the reachability of certain firing patterns in the SG, thereby making sure the conditions of complete state coding in the STG are easier to satisfy [18], [73]. These assumptions are referred to as *RTCs*. Systems incorporating relative timing constraints are simpler in design than those which make no assumptions at all. However, care must be taken to ensure that such assumptions are reasonable.

2.4 METASTABILITY

In this section, we will discuss an asynchronous design concept known as *metastability*. At the most abstract level, a metastable event can be thought of as a con-

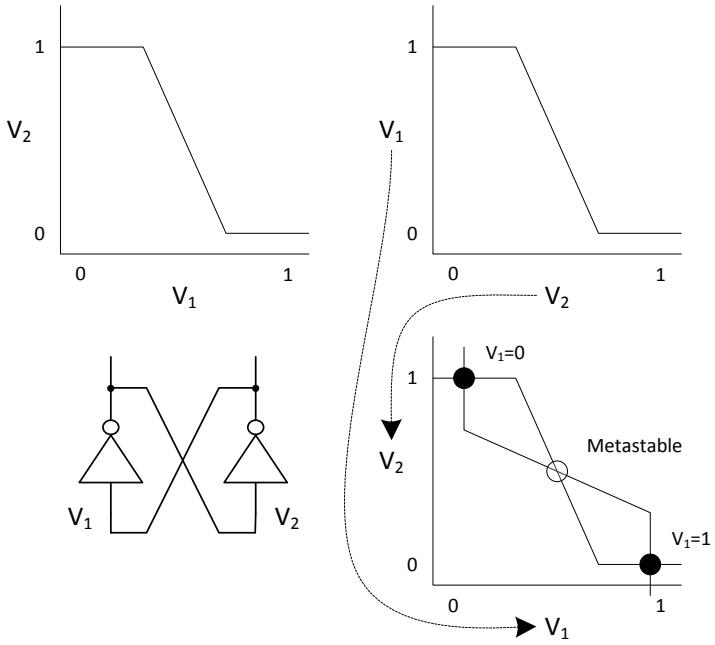


Figure 2.10: Abstract view of a butterfly curve. From [44]

tention between two equally valid choices. The classic example of this is a ball standing at the apex of a hill which can roll down either side when given a tiny external stimulus. From a digital circuit perspective, the metastable state is induced in a logic gate when the arrival times of two or more signals at the gate input lead to a longer than normal resolution time at the gate output. Analog logic can also exhibit metastability as well, but the trigger is sufficiently close differential voltage, rather than sufficiently close differential time as analog circuits are level-sensitive rather than edge sensitive. For the purposes of this work, we will restrict the review of this topic to the circuits discussed in section 2.1.3, as sequential logic elements (i.e. latches, flip-flops, MCEs).

In the case of a flip-flop, metastability manifests itself when the *butterfly curve* of the storage nodes reaches a crossing point. To form such a curve, the Voltage Transfer Characteristic (VTC) graph of one of the storage nodes is flipped over and superimposed on top of the VTC of the other node, as in Fig. 2.10. The resulting graph contains 2 stable states and one metastable state. At the metastable point, both of the latches comprising the flip-flop can be modeled as linear amplifiers using small signal analysis, which results in the KCL equations of (1):

$$-C_2\frac{dV_2}{dt} = G_2V_2 + AG_2V_1, \qquad -C_1\frac{dV_1}{dt} = G_1V_1 + AG_1V_2 \tag{1}$$

where G is the output gate conductance (i.e., 1/R) of the circuit, -A is the small-signal gain of the linear amplifiers, and the output time constants, τ are dependent on C/G as shown in (2) [80].

$$\tau_1 = \frac{C_1}{AG_1},\ \tau_2 = \frac{C_2}{AG_2} \tag{2}$$

Eliminating V_1 gives us a second order differential equation which has a solution (3) of the form:

$$V_1 = K_a e^{\frac{-t}{\tau_a}} + K_b e^{\frac{t}{\tau_b}} \tag{3}$$

Under the assumption that the inverters are identical to each other have a gain where A»1, thereby making $\tau_a = \tau_b = \sqrt{\tau_1 \tau_2}$. We are only interested in the metastable term in the equation, so the K_a term is discarded which gives us an output volage (4) of:

$$V_1 = K_b e^{\frac{t}{\tau_b}} \tag{4}$$

The initial term K_b depends on the differential between the arrival times of the clock and data signals at the latch input. If the signals are well separated, then K_b is large and the latch resolves quickly from the linear/metastable region. Conversely, if the time differential between the clock and data signals is small, then K_b will be close to zero and the latch remain in the metastable state for quite some time. Theoretically, it can be infinite time if K_b is exactly zero, but this doesn't occur in practice as there will always be some time differential between the arrival times of the input signals (though it can still lead to failures if it is sufficiently small). K_b is modeled by (5)

$$K_b = \Theta \Delta t_{in} \tag{5}$$

where Δt_{in} is the time differential from the $K_b = 0$ balance point, and Θ is a circuit parameter modeling the rate at which this differential converts to a voltage difference between the cross-coupled nodes of the latch. It is typically modeled as $V_{DD}/2\tau_s$, where τ_s is the RC delay constant of a capacitive load placed on the output nodes of the latch. The time taken to reach a stable exit voltage, V_e can be found by setting $V_1 = V_e$ in (4), yielding:

$$t = \tau \ln \left[\frac{V_e}{\Theta\ \Delta t_{in}} \right] \tag{6}$$

which will be used in the next chapter to help calculate the Mean Time Between Failures (MTBF) of a synchronizer circuit.

2.5 EMBEDDED CYCLE GRAPHS

This chapter will close with a discussion of the basic graphical terminology which will form The basis of what we will refer to throughout the remainder of this work as an *embedded cycle graph*, which is similar to (but not necessarily the same as) what has been referred to as a pancyclic graph or sub-tournament in prior work [38][54][3][9]. This graphical construct can be used to model the abstract connectivity of the controller specifications that will be presented in Chapter 3 and beyond. Section 2.5.1 will address the basic graph theoretic terms and definitions necessary to understand an embedded graph in the mathematical sense, while section 2.5.2 will place these graphs in a context which is useful for the remainder of this work.

2.5.1 *Basic Graph Definitions*

A *graph* is a mathematical construct composed of *vertices* and *edges*. Vertices are defined as the corner points of the graph, and formed via the intersection of edges. Edges refer to the set of *unordered pairs* (i.e., lines) which link together the vertices within the graph. If the edges are *ordered* (i.e., directed) they tend to be referred to as *arcs*. Formally,

Definition 2.9 A graph $G = (V,E)$, where V is the set of vertices, and E is the set of edges. Each edge is a pair (v,w) where $v,\ w \in V$.

Definition 2.10 A graph $G = (V,E)$, is referred to as a *directed graph* or *diagraph* if the edges are *ordered*.

Definition 2.11 The number of edges incident to a vertex v in the graph $G = (V,E)$, is referred to as the *degree* of the vertex, $\deg(v)$, where loops between the same vertex are counted twice. In this work, the *maximum degree* of the graph G will be denoted as $\Delta(G)$, and the *minimum degree* will be denoted as $\delta(G)$.

Definition 2.12 In a directed graph $G = (V,E)$, vertex w is *adjacent* to vertex v if and only if there exists an edge from v to w.

Definition 2.13 A *cycle* in a directed graph $G = (V,E)$ is defined as a *path* (i.e., a sequence of vertices connected by edges) that begins and ends at the same vertex, and contains at least one edge.

Definition 2.14 A graph $G = (V,E)$, is referred to as a *cycle graph* if the number of vertices and edges in the graph are equal $|V| = |E|$, every vertex has degree 2, and all vertices are connected via a single path.

Definition 2.15 A graph $G = (V,E)$, is referred to as a *chordal graph* if every cycle of length 4 or more contains a *chord*. A chord is defined as an edge which is not

part of the cycle, but which also connects two vertices of the cycle[23].

Definition 2.16 A graph *G = (V,E)*, is referred to as a *pancyclic graph* if the graph contains n vertices and for every k in the range $3 \leqslant k \leqslant n$, G contains a cycle of length k[9].

Definition 2.17 A graph *G = (V,E)*, is referred to as a *polycyclic graph* if the graph contains n vertices and there exist a finite number ($\geqslant 2$) of k in the range $3 \leqslant k \leqslant n$, where G contains a cycle of length k.

2.5.2 *Applications of an Embedded Cycle Graph*

So what is an embedded cycle graph? First, it must be noted that the terminology itself is almost certainly a point of contention. The term itself arose based on how the primary researcher viewed the graphs he worked on. Several cycles in the distributed multi-cycle token rings that he worked with, and which will feature prominently in Chapter 3, shared a large number of vertices between cycles. After speaking with other researchers, a more accurate terminology might be something along more along the lines of a *multi-cycle graph* or *polycyclic graph*.

On the surface, an embedded cycle graph, as it is used in this work, bears heavy resemblance to a pancyclic graph, as discussed in earlier work by Bondy [9]. However, unlike a pancyclic graph, which requires the possession of a cycle of every length from 3 to N, as per Definition 2.16, the embedded cycle graph described in Section 3.3.2 does not. That particular algorithm allows the user to specify whether to add or omit feedback paths based on user defined parameters (specifically the entries in a pair of configuration arrays). In a true pancyclic graph, there is no choice. On the other hand, from a hardware perspective, "omitting a path" might be the equivalent of just not using a route that might exist regardless. In short, it is easy to muddy this definition.

Chapter 5 does not make this task any easier. While referring to an embedded cycle graph in the context of a chordal ring network, we must qualify that the chordal ring network that was used as an example was a *symmetric chordal ring network*. By no means does this imply that symmetric chordal ring networks are equivalent pancyclic graphs. Having said that, Chapter 5 deals with configuration and bypass operations which act to both control and restrict the movement of a control token throughout the ring. Therefore, while a chordal ring network is not a pancyclic graph by definition, it can certainly be made to *emulate* the behavior of a pancyclic graph, as is demonstrated by comparing the results in Fig. 3.18, to those of Fig. 5.4 (both of which were obtained using CADENCE).

3 OVERVIEW OF WAGGING SYNCHRONIZATION

3.1 INTRODUCTION(SYNCHRONIZATION METHODS)

Continuing forward from the concepts discussed in Chapter 2, we can now present a discourse on synchronization. As synchronization plays a prominent role in the remainder of the thesis, we shall approach the issue from both a general perspective via a brief summary of prior work, and the more specific context of synchronization as it applies to a concept known as *wagging*.

An overview of prior work on synchronization will be discussed in Section 3.1. Section 3.2 will discuss the issue of wagging synchronization in detail. Section 3.3 will illustrate how to design a reconfigurable token ring which incorporates the material covered previously in Section 2.5. Finally, Section 3.4 will compare and contrast differing designs for both combinational and sequential token rings and lay the groundwork for the information contained in Chapters 4 and 5.

3.1.1 *Synchronization Overview*

As stated previously in Section 2.4, metastability can be thought of as a contention between two equally valid choices. It was also shown that this contention had real consequences when applied to circuit implementations. In one case, it arose because of the uncertainty in the arrival times of digital signals at the inputs to flip-flop, and the clock signal used to sample the input values. Synchronization can be thought of as a way to deal with this uncertainty by allowing the system to sample only the input at times where the input signal is assumed to be at a stable logic level.

Whether or not synchronization is necessary in a given scenario depends on the situation. We shall start by considering Figure 3.1(a). If A,B, C, and D are general processing elements, which pass data between each other in an electronic system, and all of the processing elements are driven by a single clock which possesses the same *phase* and *frequency* at all points, then synchronization is unnecessary as the entire system is *synchronous*.

If the system is driven by clocks that share rationally related frequencies with a limited phase variance at all points, such as with a phase-locked loop, then the system is referred to as *mesochronous*. This is the case in Figure 3.1(b). Synchronization might not be necessary provided that the data transfers occur at times which are mutually compatible for both the sender and receiver. If the clock frequencies are rationally related, but the phase drift is unbounded, then the system is referred to as *plesiochronous*. Since the clocks are rationally related the phase difference is to some extent predictable in advance, conflicts can be detected and resolved without the need for synchronization provided that the frequencies of the sender and receiver are not dissimilar enough that data is either missed on the receiver end (i.e. the receiver clock is much slower than the sender clock), or sampled more

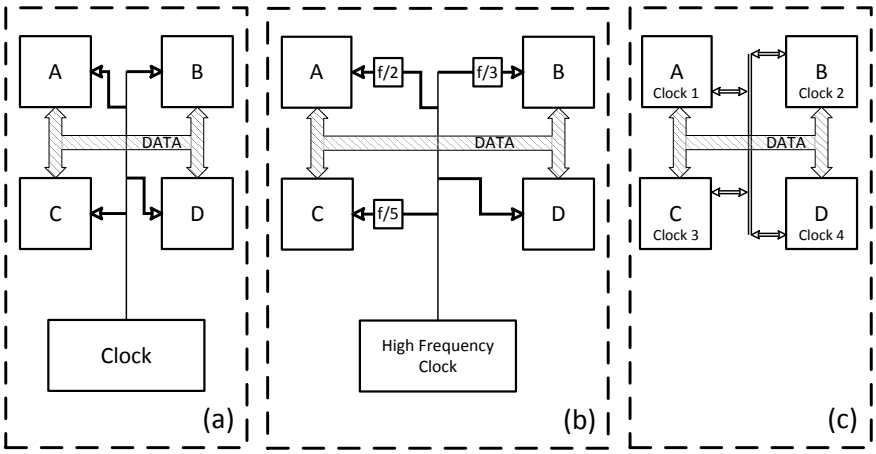


Figure 3.1: Overview of clocking methods: (a) Synchronous, (b) Rationally related clocks, (c) Multiple clocks

than once (i.e. is receiver clock is much faster than the sender clock) [28]. Otherwise, synchronization is necessary, as in Figure 3.1(c). If the clock domains have an unknown phase and frequency relationship with respect to each other, then the system is referred to as *heterochronous* and synchronization is required for every data transfer.

3.1.2 *Cascaded Flip-Flop Synchronization*

To transact communication reliably between components forming a sender/receiver pair on a SoC where synchronization is required, an interface must be constructed which synchronizes the write data value from the clock domain of the sender to that of the receiver, and thereafter synchronize the acknowledgement from the receiver back to the sender indicating that the sender is free to write the next data value (if any exist), as depicted in Figure 3.2. The most basic circuit which provides the synchronization functionality within this interface is commonly referred to as a *cascaded flip-flop synchronizer*, which is depicted in Figure 3.3(a). In a cascaded flip-flop synchronizer, synchronization is performed using two or more flip-flops in a master-slave configuration as depicted in Figure 3.3, which are situated at each end of the a transmitter/receiver device pair, as in Figure 3.2. When the master device is transparent, the slave is opaque, and vice versa.

Using the example of Figure 3.3(b), the master latch becomes opaque on the rising edge of the clock signal, *Clock*, at which point it stops sampling the instantaneous value of the input voltage from the data line, *Data*, and begins its resolution phase, while the slave latch begins sampling the instantaneous voltage of the master latch. On the falling edge of the clock the slave latch becomes opaque thereby transferring the instantaneous voltage value from the master latch, while the master latch becomes transparent and begins sampling voltage values from the input data line again. If the voltage values of the slave latch are stable when it becomes

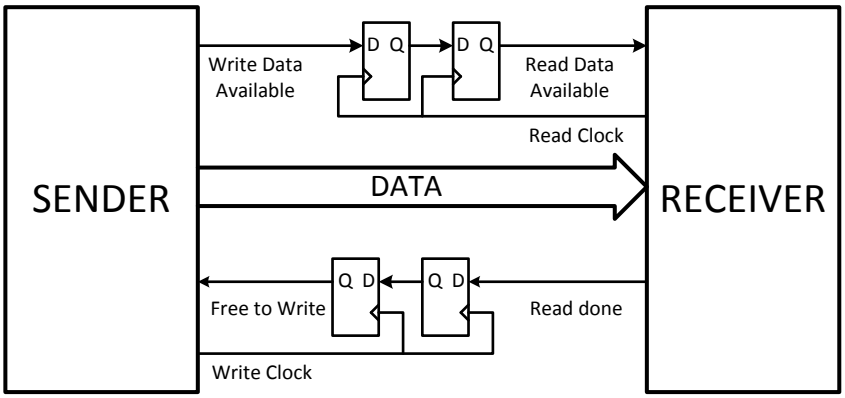


Figure 3.2: Synchronizing interface. From [44]

Table 3.1: MTBF of a Cascaded Two-Flop Synchronizer. From [2]

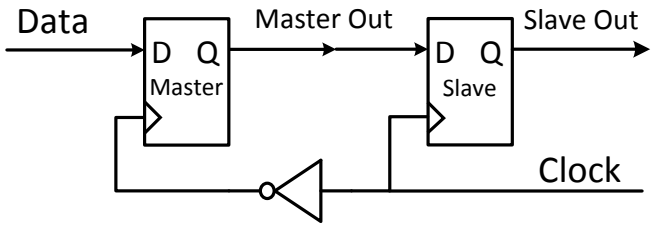
Parameters	Nominal	Low V_{DD}	Worst Case	Worst Case & Low V_{DD}
V_{DD}	1.0 V	0.3 V	1.0 V	0.3 V
$f_c = f_d$	1 GHz	5 MHz	1 GHz	5 MHz
τ	8.9 ps	1.95 ns	13.76 ps	4.12 ns
T_W	10 ps	50 ns	10 ps	50 ns
t	846.4 ps	80.2 ns	825.6 ps	44 ns
MTBF	6.5×10^{26} years	2.1×10^{4} years	3.6×10^{11} years	38 ms days

opaque, the data has been synchronized to the clock signal. If not, the master-slave flip-flop suffers a synchronization failure.

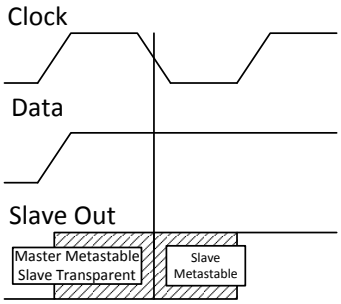
3.1.3 *Mean-Time Between Failures in a Cascaded Flip-Flop Synchronizer*

Having just mentioned synchronization failures, we are left with a query: *How often do such failures occur?* In general, this query is expressed in the form of a design concept known as the Mean Time Between Failures (MTBF). This concept is broadly defined as follows:

Definition 3.1: The Mean Time Between Failures (MTBF) of a system is the predicted time elapsed between inherent failures of a system during normal operation. In the context of a synchronizer, the MTBF is defined as the probability that the arrival of concurrent transitions on the clock and data lines fall within a *"window of vulnerability"* such that the synchronizer does not resolve to a logical high



(a) Schematic



(b) Operational waveform

Figure 3.3: Cascaded flip-flop synchronizer

or low value in the time allotted to it by the system.

So, again, how often does failure occur? It is actually dependent on several factors, however it is most strongly impacted by the supply voltage. Using data that was provided by Alshaikh et. al, and has been reproduced in Table 3.1, we can see that the MTBF for a cascaded two-flop synchronizer in can vary from 6.5×10^{26} *years* all the way down to 38 *milliseconds* [2].

Now that the MTBF has been expressed in specific terms above, let us illustrate how to characterize and derive it. In Section 2.4, an equation for the time necessary to reach an exit voltage,V_e, from the metastable region of operation within a latch was defined. Using the same concepts, we can define an equation to ascertain the MTBF for the cascaded flip-flop synchronizer that was just discussed. First, we must define terms to represent the data rates of the sender, f_d, and receiver, f_c, circuits. If the frequency relationship between the two regions is unknown, then the distribution of the input arrival times is assumed to be uniform between 0 and $1/f_c$. Then we must define a window of vulnerability, Δt_{in}, during which concurrent transitions between the data and clock (or enable) signals lead to longer than normal resolution times in the latches of the synchronizer. Rewriting (6) in Section 2.4 yields the following for Δt_{in}:

$$\Delta t_{in} = T_w e^{\frac{-t}{\tau}},\ T_w = \frac{V_e}{\theta} \tag{7}$$

where t is the time available for synchronization allotted for recovery from metastability (also referred to hereafter as t_{MSR}), T_w is a parameter known as the metastability window, τ is the metastability time constant, and θ was defined previously in Section 2.4, all of which are dependent on various circuit parameters. Within a given time T, $f_d T$ data values will be transmitted, and of those values $f_c \Delta t_{in}$ will have input arrival times at or less than Δt_{in}, which will result in synchronization failures. Thus, the MTBF for a latch based synchronizer is as follows:

$$\text{MTBF} = \frac{1}{f_d f_c \Delta t_{in}} = \frac{e^{\frac{t}{\tau}}}{f_d f_c T_w} \tag{8}$$

However in an cascaded flip-flop synchronizer, the time available for synchronization t is changed to $t/2$, as only half of the clock cycle is alloted for resolution, which changes the MTBF equation. Thus (7) becomes the following:

$$\Delta t_{in} = T_w e^{\frac{-t}{2\tau}} \tag{9}$$

The MTBF of a cascaded flip-flop synchronizer can be found by counting the total number of times that the output of the slave latch resolves from metastable behavior within a given δt over the course of t events, in a given time interval T.

$$\frac{\delta t}{\tau}\left[f_d f_c T \Delta t_{in}\right] = \frac{\delta t}{\tau}\left[f_d f_c T T_w e^{\frac{-t}{2\tau}}\right] \tag{10}$$

In order to find the total number of events that lead to synchronizer failures, δt must be set equal to $\Delta t_{in} = T_w e^{\frac{-t}{2\tau}}$.

$$\frac{T_w e^{\frac{-t}{2\tau}}}{\tau}\left[f_d f_c T T_w e^{\frac{-t}{2\tau}}\right] = \frac{T_w}{\tau} f_d f_c T T_w e^{\frac{-t}{\tau}} = \frac{T}{\text{MTBF}} \tag{11}$$

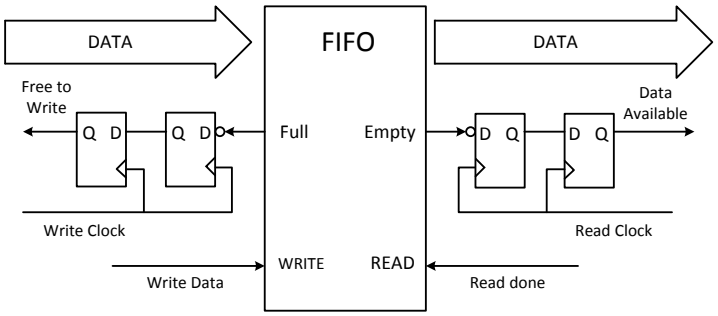
As a result, the MTBF equation in (8) becomes

$$\text{MTBF} = \frac{e^{\frac{t}{\tau}}}{f_d f_c T_w}\frac{\tau}{T_w} \tag{12}$$

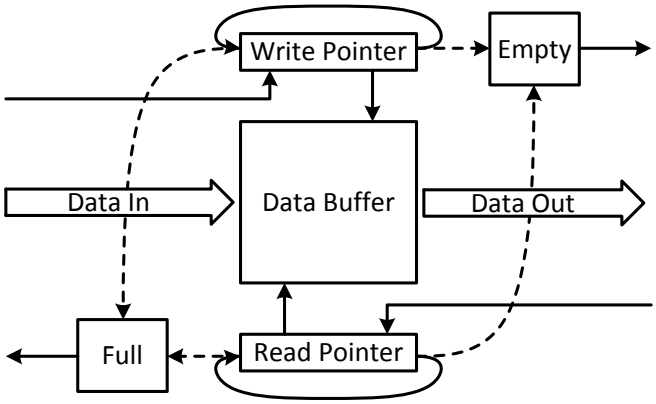
for a cascaded flip-flop synchronizer. Thus, the T_w term in (8), is replaced by T_w^2/τ in (12).

3.1.4 *FIFO Synchronization*

One limitation in interfaces utilizing cascaded flip-flop synchronization is that the throughput of any data transfer is governed by the roundtrip delay through both the transmitter and receiver ends of the synchronizer [24]. However, by incorporating a FIFO buffer in between the sender and receiver device pair in Figure 3.2, the reading and wring operations can be decoupled from each other, thereby allowing both operations take place as soon as the system is ready [15], as depicted



(a) Schematic



(b) Abstract behavior model

Figure 3.4: FIFO synchronizer. From [44]

in Figure 3.4(a). The read operation can take place as long as there is valid data stored in the FIFO, while the write operation can be performed as long as the FIFO has space available in the buffer.

A basic FIFO buffer suitable for synchronization applications is composed of a set of registers, which are referenced via read and write pointers, as shown in Figure 3.4(b). The write pointer points to a free location within the buffer, and is advanced by one after each new data item is written. Similarly, the read pointer points to previously written valid data items, and is advanced by one upon a successful read operation. To guarantee that the writing function in the buffer occurs without error the write pointer must both point to location that is presently free (i.e. writeable), and also ensure that the next location is free as well, so that the write pointer can be advanced by one when the operation is completed. By a similar argument, the read pointer must not only point to data item that is presently readable (i.e. previously written with valid data), but also ensure that the next data location is readable as well. For this reason, the write pointer must always lead the read pointer by two data locations.

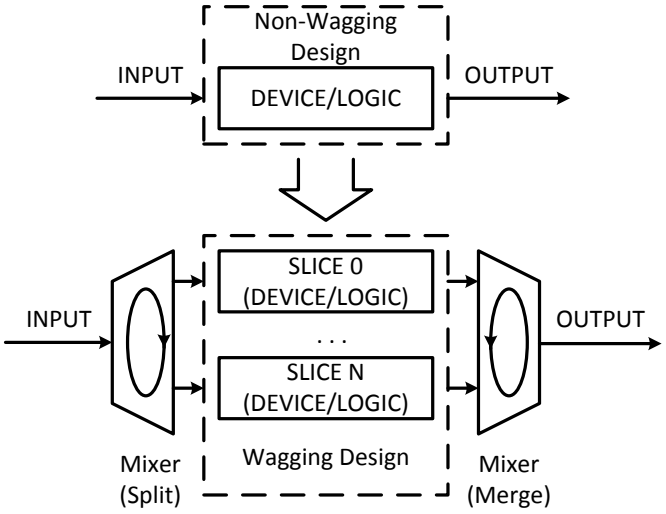


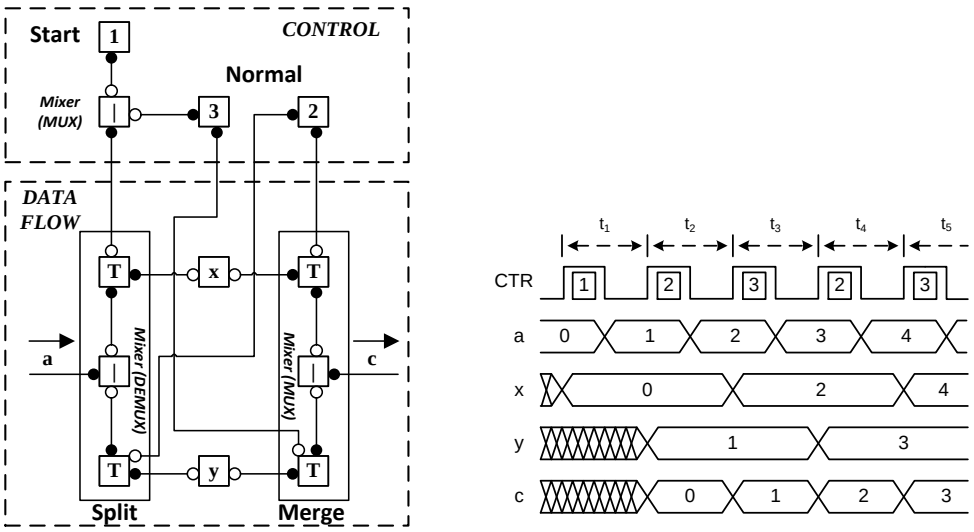
Figure 3.5: Conceptual diagram illustrating the wagging principle.

The buffer in Figure 3.4(b) also contains flags *Empty* and *Full*, which indicate if the FIFO buffer has either an unacceptable absence or abundance of data items by comparing the position of the read and write pointers. *Full* is set to true if there is not sufficient space within the buffer to accept incoming data items that might be written before the sender can be halted. Similarly, *Empty* is set to true if there are an insufficient number of valid data items in the buffer which might be read before the receiver can be halted. While the read and write pointers are synchronized to the clocks of the receiver and sender, respectively, neither the *Full* or *Empty* flags are synchronized at all. This necessitates that the *Full* and *Empty* signals, be synchronized to the clocks of the sender and receiver, respectively.

Thus, there must always be at least n to $n+1$ valid data items in the FIFO buffer before reading operations can commence, where n represents number of cycles necessary to synchronize the *Empty* signal to the receiver clock. By similar arguments, there must always be between k and $k+1$ spaces in the buffer for writing operations to occur, where k represents number of cycles necessary to synchronize the *Full* signal to the sender clock. Further to that point, if the clock frequencies of the sender and receiver are vastly different, then the throughput of a synchronizer which incorporates a FIFO buffer is actually worse than using a simple cascaded flip-flop synchronizer due to the extra communication latency incurred from buffer *underflowing* if the receiver clock is faster than the sender, or *overflowing* if the sender clock is faster than the receiver. Thus, both synchronization methods carry various benefits and tradeoffs.

3.2 WAGGING SYNCHRONIZATION

With the previous discourse on the concept of MTBF in Section 3.1 in hand, we can now discuss refinements that others have made in an effort to motivate the need for the primary synchronization method discussed in this thesis. Jones, Yang, and Greenstreet refined the basic equations in (9) and (12) to incorporate the effects of



(a) Schematic

(b) Timing diagram

Figure 3.6: Annotated diagram of a two-way wagging buffer.

a sizing mismatch between the latching elements within a cascaded flip-flip synchronizer [42]. Ginosar et al. showed that the metastability time constant, τ, fails to track the FO_4 delay at low feature sizes [6]. Perhaps most importantly though, Zhou demonstrated how τ degrades (becomes larger) with reduced supply voltages in an unmodified jamb latch [83].

While many solutions exist to resolve the issues of data synchronization, the solutions which employ parallelism (i.e. using many components to perform one specific task) remain of particular interest to this work. Employing parallelism in the context of a synchronizer circuit has two useful properties:

1. With appropriate scheduling of parallel tasks, mismatches between the transmitter and receiver ends of a synchronizer can be minimized.

2. The *Mean Time Between Failures (MTBF)* can be manipulated by adjusting the degree of parallelism employed.

These two properties have appeared in other bodies of literature in one form or another. Of specific interest is prior work by Kees van Berkel, on handshaking circuits where he defined a concept known as *wagging* (i.e. employing parallelism, in tandem with the scheduling of tasks via time division) [77]. Ebergen also experimented with the scheduling of tasks in parallel compositions of finite state machines in his own work [25]. More recently, Brej used the concept of wagging to compose a system of parallel logic wherein he used the phrase *wagging level* to denote the number of replicated parallel components in the system to which the input data was given. Each parallel component, called a *slice*, was then assigned a *slice number* for the purposes of task scheduling [11].

A design incorporating wagging will always contain two properties, as in Fig. 3.5, regardless of whether or not the design is controlled via synchronous control signals or asynchronous communication signals called *handshakes*.

1. Usage of parallel components to share the workload of a task.
2. Scheduling of said tasks via time division.

In Section 3.2.1 we will explore an example of a 2-way wagging buffer, then extend the discussion to a cascaded flip-flop synchronizer in Section 3.2.2. Thereafter, Section 3.2.3 examination of the effects of wagging on the failure rates in a cascaded flip-flop synchronizer. Next, Section 3.2.4 and 3.2.5 will comment on the impacts and overheads of incorporating reconfigurable hardware design methods into a wagging synchronizer design. Finally, Section 3.2.6 will conclude with an overview of a reconfigurable control device suitable for use in a wagging synchronizer.

3.2.1 *Two-way Wagging Buffer*

To understand the concept of wagging we must look to prior literature on the subject. In 1992, Kees van Berkel presented work on asynchronous handshaking circuits, including work on a two-way buffer which he referred to as a *wagging buffer*, reproduced in Fig. 3.6 [77]. Active ports in Fig. 3.6(a) are indicated by black dots, while passive ports are indicated by white dots.

The data flow aspect of the buffer is composed of *mixers* (|), *transferrers* (T), and variables x, and y which function as memory. Transferrers are components which pass values through their active ports when triggered along their passive ports, while mixers are components which pass handshakes from their passive ports to their active ports, and can either act as demultiplexers (DEMUX) or multiplexers (MUX).

Let us explore the operation of the circuit with a concrete example, as in Fig. 3.6(b). Let *a* be the data input of the 2-way wagging buffer, *c* be the data output of the buffer, and *CTR* be the asynchronous control/enable signals used to sequence the operations in the buffer. Furthermore, let t_i be the time intervals upon which the the specific control operations (represented by the squared numbers) are performed, where $1 \leqslant i \leqslant 5$.

The operation of the circuit is as follows.

1. $x \leftarrow a\ @t_1$ (x is written with the value of a at time t_1).
2. $c \leftarrow x$ (i.e., $a\ @t_1$ is passed to the output of the buffer), and $y \leftarrow a\ @t_2$ (y gets the value of a at time t_2).
3. $c \leftarrow y$ (i.e., $a\ @t_2$), and $x \leftarrow a\ @t_3$

Step 1 only occurs at the start-up of the circuit, in order to place valid data on x prior to its read out during the next step. Thereafter, only steps 2 and 3 are executed. Thus, even though x and y in Fig. 3.6 are placed in parallel, functionally they act as if they were placed in series due to the scheduling of tasks[77].

3.2.2 *Wagging Synchronizer Concept*

With that discussion in hand, we can now apply the principle of wagging to the cascaded flip-flop synchronizer discussed earlier. In Fig. 3.7, we explore a synchro-

nizer example in which the path from the transmitter to the receiver incorporates wagging, and where the reverse path does not. By juxtaposing the two halves of the synchronizer circuit in this manner, it is easier to illustrate the impacts of wagging.

The wagging portion of the circuit divides the workload across N slices, where each slice represents a pair master-slave latches. Each of these slices, operates at a frequency of $f_{clk(READ)}/N$. The slices are scheduled by the wagging controller, which is also responsible for configuring the mixers which encapsulate the parallel data paths used for wagging. The controller also contains an interrupt subsystem which is used during reconfiguration operations, which will be discussed further in the next subsection. For now, let us just assume that N is fixed and configuration operations are unnecessary.

By contrast, the reverse path only consists of a single pair of master-slave latches, which operate at a frequency of $f_{clk(WRITE)}$. If we assume the simplest plesiochronous case where $f_{clk(READ)}$ and $f_{clk(WRITE)}$ are equal in frequency with an unbounded phase drift between the arrival times, then the time available for synchronization between the two halves of the circuit can be directly compared.

As the data slices in the wagging portion of the circuit operate at a frequency of $f_{clk(READ)}/N$, the time available for synchronization is N times greater than in the half of the circuit in which wagging is not employed. Recall, that in (6) the time it takes for a latch to exit from the metastable state is dependent on the temporal difference, Δt_{in} between the arrival times of the clock and data signals at the gate inputs. From (6) we can see that a linear increase in the time allotted for metastability has an exponential impact on the V_e necessary to remain in a metastable state beyond the time given. More accurately, V_e must be exponentially smaller to cause the output to remain metastable beyond the new time available for synchronization. Thus, we can say that wagging has a net effect of reducing the probability that a synchronization operation will fail, thereby resulting in a more reliable circuit.

Using parallel hardware components to increase the throughput and metastability characteristics of a synchronizer circuit was previously put forth by Jex and Dike in 1995, and further characterized by Suk-Jin et. al in 2004, but there were still design concerns which were not addressed [41], [43]. One such concern was the limited nature of the ring counter used in the original works which acted as a control device for the interconnect lines. More specifically, in both works, the parallelism in the simulations and resulting circuits was fixed, and the could not be dynamically adjusted at run-time. This was also true in more directly related work on the subject by Alshaikh et. al, in 2010 on a three-phase synchronizer design, which also incorporated the wagging concept [2].

3.2.3 *Effect of Wagging on the Failure Rate of Cascaded Flip-Flop Synchronizers*

In the last subsection, we touched briefly how the time available for synchronization in a cascaded flip-flop synchronizer circuit was affected, or more specifically extended, by incorporating wagging of a fixed degree of N parallel components. Now, we will explore both how this change in the time available for synchroniza-

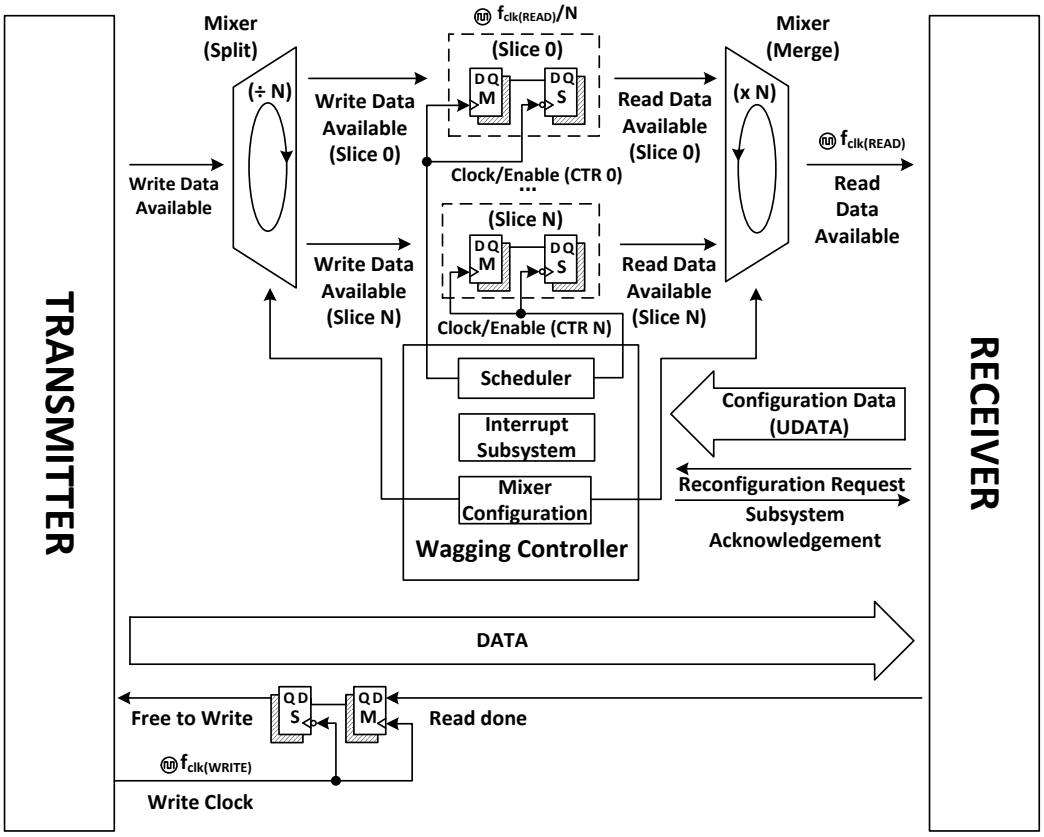


Figure 3.7: Block diagram of an N-wagging synchronizer

tion affects the failure rate in a cascaded flip-flop synchronizer, and also discuss the benefits and trade-offs of incorporating reconfigurable hardware capability into the circuit. While reconfigurable hardware design is a broad topic, in this context reconfiguration will refer to the ability of a circuit to modify itself around an external stimulus. In the case of the controller in Fig. 3.7, this stimulus takes the form of an asynchronous handshake and associated configuration data, and the modification will refer to the change in the number of parallel components (i.e., slices) which contribute to the synchronization operation.

Let us briefly touch on the first topic. We can modify the failure equation for a cascaded flip-flop synchronizer shown in (12) to account for the effects of increased parallelism. The number of slices, N, was fixed in the previous example, but now let us assume that the number of slices can be varied. Let j be an integer $1 \leqslant j \leqslant N$, where j represents number of slices (i.e., master-slave latch pairs) which are active in a given configuration. Parallelism functions to linearly increase the value of t (i.e. the time available for synchronization) by splitting the synchronization "workload" across j slices, as in Fig. 3.8. Thus, j affects the numerator in the exponential portion of (12) as shown in (13).

$$\mathrm{MTBF}_{(parallel)} = \frac{e^{\frac{jt}{\tau}}}{f_d f_c T_w} \frac{\tau}{T_w} \tag{13}$$

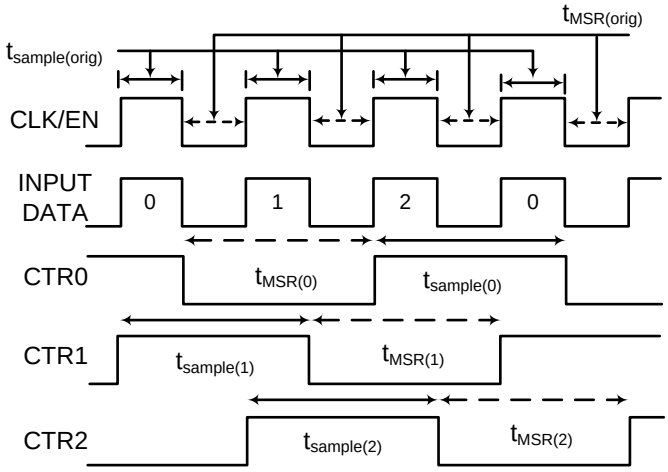


Figure 3.8: Data flow from the transmitter end of a wagging synchronizer to the input of the FIFO in Fig. 3.7 with $j = 3$ and a 50% duty cycle.

A linear increase in the time available for synchronization results in an *exponential* improvement in the MTBF of each parallel cascaded flip-flop synchronizer component in the system. Consequently, the master-slave latch pairs which comprise each slice are less likely to suffer a synchronization failure. Assuming that the scheduling of the slices is uniform, the MTBF of the wagging portion of the synchronizer circuit is the same as the MTBF of the individual slices. However, the denominator in the exponential portion of (13) (i.e. τ) exponentially increases with a linear decrease in voltage[83]. Thus, while the impact of parallelism on the exponential portion of (13) is *linear*, the impact of voltage on the exponential portion of (13) is *exponential* (at low voltages). For this reason, robust latches such as those defined by Zhou et al. can be used instead of regular jamb latches in the design of a wagging synchronizer if low voltage operation is desired, but this is by no means a design constraint [83].

Once again, we can use a data set that was provided by Alshaikh et. al, which has been reproduced in Table 3.2, to support our case [2]. In that study, he compared the MTBF of a (fixed) 3-way wagging synchronizer to that of a cascaded two-flop synchronizer (the results of which were shown earlier in Table 3.1). Note that the parameters of interest are identical, except for the time available for synchronization which was extended via the use of the wagging paradigm. By comparing the two tables, we can see that the MTBF of the $WorstCase$, Low V_{DD} has improved from *38 milliseconds* in the cascaded two-flop synchronizer to *72 days* 3-way wagging synchronizer design.

3.2.4 *Impact of Incorporating Reconfigurable Hardware Capability into a Wagging Synchronizer Design*

Moving forward, it is now possible to discuss the practical motivations for including reconfigurable hardware capability into a parallel synchronizer design.

Table 3.2: MTBF of a Fixed 3-Way Wagging Synchronizer. From [2]

Parameters	Nominal	Low V_{DD}	Worst Case	Worst Case & Low V_{DD}
V_{DD}	1.0 V	0.3 V	1.0 V	0.3 V
$f_c = f_d$	1 GHz	5 MHz	1 GHz	5 MHz
τ	8.9 ps	1.95 ns	13.76 ps	4.12 ns
T_W	10 ps	50 ns	10 ps	50 ns
t	923.2 ps	140.1 ns	912.8 ps	122.2 ns
MTBF	3.7×10^{30} years	4.9×10^{17} years	2.0×10^{14} years	72 days

Conceptually, the strongest reason for incorporating such functionality lies in the treatment of the MTBF itself. More specifically, we can alter the context in which the MTBF is addressed. In general, the MTBF of the system is fixed at design time (i.e., it is a static parameter). By allowing the designer to control the MTBF by modifying the degree of wagging in the system on-the-fly, the MTBF can be treated as a variable parameter (i.e., the system can alter itself to suit the MTBF needs of a given situation).

There are two scenarios in which wagging synchronization can be usefully employed in tandem with reconfiguration:

1. Adjusting for subtle timing variations introduced via processing parameters, which are unknown at tape out.

2. Enabling the synchronizer to maintain a pre-defined MTBF value when the duty cycle of either the write clock (transmitter) or read clock (receiver) are altered in a way that would otherwise negatively impact the MTBF.

In the former case, the process parameters of CMOS transistors in a region local to the wagging synchronizer can be deduced through the use of a clock recovery circuit, such as Costas loop [19]. The input to the recovery circuit is the oscillation generated by one of the Clock/Enable (CTR) outputs in the scheduler potion of the Wagging controller of Fig. 3.7, provided that the output signal selected is actively participating in the present configurable mode. If the number of slices in the synchronizer, N, is set to a fixed reference value, R, then the operating frequency of the scheduler portion of the Wagging controller can be tested against a reference frequency, which is equal to the operating frequency of the scheduler under the typical-typical process corner.

If the Phase-Locked Loop (PLL) achieves lock (i.e., if the phase detector reaches its minimum value), then the output frequency of the Voltage Controlled Oscillator (VCO), and by proxy the scheduler portion of the Wagging controller, can be found using either a Look-Up Table (LUT) of the VCO output frequency vs. VCO control voltages, or a suitable approximation [47]. This output frequency can then

be directly compared to the reference frequency to ascertain whether or not the transistors which comprise the wagging synchronizer are faster or slower than in the typical case, and by how many standard deviations, thus yielding the process parameters for the Wagging controller and the transistors in the area local to it. This information can then be stored in memory, and later used to adjust the number of active nodes in the control device at run-time based on internal conditions within the die. The objective is to minimize the timing variations and phase drift between the transmitter and receiver ends of the synchronizer under varying conditions.

In the latter case, when the clock frequency of the write clock (transmitter) or read clock (receiver) in a cascaded flip-flop synchronizer is increased (i.e., when the duty cycle is reduced) it has a negative impact on the MTBF, as illustrated in the last subsection. For example, if the frequency of the read clock in Fig. 3.7 is increased by a factor of two (effectively halving the time available for synchronization) , then the MTBF of the synchronization path from the transmitter to the receiver degrades exponentially, as specified by (19), assuming that the number of slices, j, is constant. However, if the number of parallel components (i.e., slices) which contribute to the synchronization operation is increased by a factor of two, the MTBF of the synchronization path from the transmitter to the receiver returns to its previous value. Thus, by employing reconfiguration in tandem with wagging synchronization, the designer can select the amount of parallelism that they require to maintain a specific MTBF value as the clock domains of the transmitter and receiver are varied.

3.2.5 *Overheads of Incorporating Reconfigurable Hardware Capability into a Wagging Synchronizer Design*

The benefits of utilizing reconfiguration in tandem with wagging synchronization must be balanced against the costs of employing such techniques. These costs come in the come in two forms:

1. Additional delays along the critical path due to the addition of the mixers necessary for the wagging functionality
2. Additional power and area overheads due to the aforementioned mixers, control hardware, and replicated master-slave latch pairs.

While the delay component of the reconfiguration hardware is solely defined by the delay across the mixer elements in Fig 3.7, the power overheads can be characterized in terms of the dynamic and standby leakage power of the additional components in the wagging synchronizer. For reference purposes, the dynamic and standby leakage power, P_{DYN} and P_{LEAK} respectively, of the individual transistors are characterized by the following two well-known equations[66] [69]

$$P_{DYN} = \alpha f C {V_{DD}}^2 \tag{14}$$

$$P_{LEAK} = I_{LEAK} \times V_{DD} \tag{15}$$

where α represents the switching factor, f is the clock frequency of the circuit, C is the load capacitance, and V_{DD} is the supply voltage in (14). In (15), I_{LEAK} represents the standby leakage current of the transistor, and V_{DD} is the supply voltage. When looking at the power in those terms, the excess dynamic and standby power overheads, $P_{DYN(OVER)}$ and $P_{LEAK(OVER)}$ respectively, incurred as a result of incorporating parallelism and reconfigurability into the synchronizer design, as in Fig. 3.7, can be characterized as follows:

$$P_{DYN(OVER)} = P_{DYN(MIXER)} + P_{DYN(CTRL)} \tag{16}$$

$$P_{LEAK(OVER)} = (N-1)P_{LEAK(M/S)} + P_{LEAK(MIXER)} + P_{LEAK(CTRL)} \tag{17}$$

where $P_{DYN(MIXER)}$ and $P_{DYN(CTRL)}$ in (16) are the dynamic power contributions of the mixer and control hardware, respectively. When looking at the leakage power overhead, $P_{LEAK(OVER)}$ in (17), the contributing factors are the leakage power of the master-slave latch pair, $P_{LEAK(M/S)}$, the mixer, $P_{LEAK(MIXER)}$, and the control hardware, $P_{LEAK(CTRL)}$, and where N is the maximum number of slices in the synchronizer.

Because of these overheads, incorporating both wagging and reconfigurable hardware design into the synchronizer specification becomes impractical when the bit-width of the original datapath is large. However, in the cases where the bit-width of the original datapath is small, the reliability benefits of the extra hardware may outweigh the aforementioned power and area costs.

3.2.6 *Basic Operation of the Reconfigurable Control Device*

Having covered the concepts of wagging, reconfigurability, and reliability within the context of a cascaded flip-flop synchronizer in preceding subsections, we will now explore how to design a control circuit which achieves these three design criteria. Such a control circuit is shown in Fig. 3.9.

The control circuit is divided into two parts, one being a token ring composed of several embedded cycles which selects its cycle length based on a control code (RDATA) generated by the interrupt subsystem. In this chapter, a one-hot code was used, though others could conceivably be used as well. The ring determines the number of parallel master-slave latches present in the synchronizer, and outputs the delayed clock/enable (CTR) signals to each of the flip-flops in the synchronizer, thereby partitioning the input data into slices [11]. The second part is an interrupt subsystem responsible for halting the operation of the token ring (and by proxy the synchronizer) while the system undergoes reconfiguration, ensuring the functional correctness of the synchronizer by preventing loss of the control token. The control circuit also outputs the configuration information for the mixer, which is also derived from the RDATA signal.

The conditions for initiating a reconfiguration operation were mentioned in Section 3.2.4. To review, if the time available for synchronization is reduced below a (user-defined) threshold as a result of either local PVT variations, or via a deliberate change in the operating frequency at the transmitter or receiver ends the

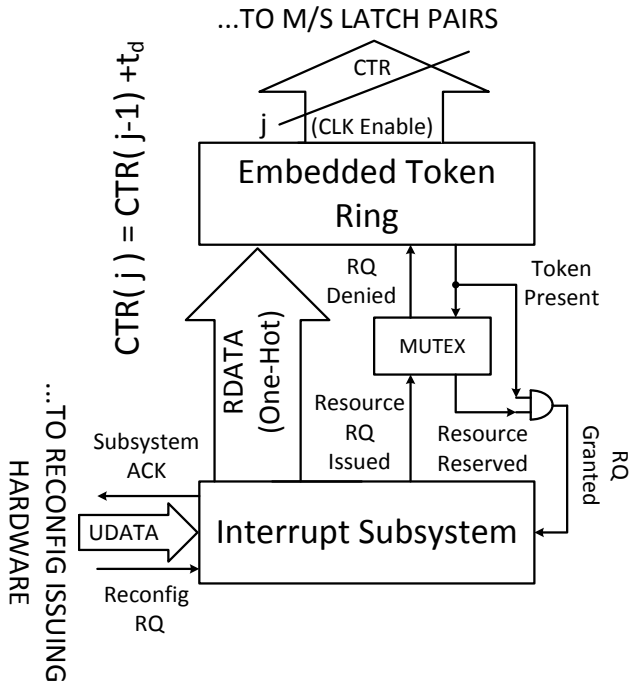


Figure 3.9: Block diagram of a reconfigurable controller suitable for a wagging synchronizer.

synchronizer, then a reconfiguration operation is required. Generally, this requires some *a priori* knowledge about the system itself (i.e., operating voltages, temperature, relative phase/frequency relationships between two clock domains). Regardless, once the need for reconfiguration has been identified via an external stimulus from the environment (i.e., changing PVT or frequency parameters), then a reconfiguration request (RQ) is issued to the interrupt subsystem of the controller bundled with the relevant one-hot control data (UDATA) generated external to the controller, which will be used in the ensuing reconfiguration operation. This request is then fed to a Mutual Exclusion Element (MUTEX) that determines whether or not the token ring can be safely halted, issuing a grant when successful. Once the request has been granted, reconfiguration proceeds. The subsystem then alters the one-hot code, RDATA, currently being sent to both the embedded token ring and the mixer, issuing an acknowledgement (ACK) signal when the process is complete, whereupon the MUTEX is released and the synchronizer resumes its operation. Due to the *bundled data assumption*, which states that the causality of asynchronous signals is to be enforced at the physical level during circuit layout (i.e., the RQ and UDATA are assumed to arrive together), the RQ and UDATA signals are prevented from becoming metastable due to a *hazard* (i.e., a race between two asynchronous signals) [18]. In theory, it would also be possible to employ dual-rail coding in tandem with completion detection to avoid a hazard between the RQ and UDATA signals, but such a method would require additional hardware and is not the focus of this work [44].

3.3 TOPOLOGY OF A RECONFIGURABLE TOKEN RING

In Section 3.2 it was mentioned that while the concepts of parallelism or even wagging are not unheard of in the context of synchronizer design, incorporating the principles of reconfigurable hardware design in tandem with wagging is [41], [43], [2]. It was shown that a combination of wagging and reconfigurable hardware design could be used to augment the MTBF characteristics of a cascaded flip-flop synchronizer. Furthermore, an overview of a controller suitable for accomplishing this task was presented in Section 3.2.6. The controller was divided into two parts:

1. A reconfigurable token ring, which acts as a scheduler for the parallel master-slave latch pairs in the wagging synchronizer.

2. An interrupt subsystem responsible for halting the operations in the synchronizer, which guarantees that the reconfiguration operations in the wagging synchronizer occur without error.

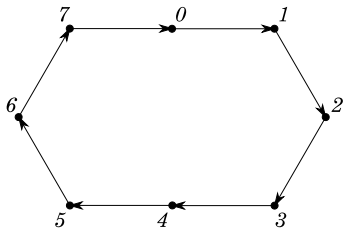
While Chapter 4 will deal with the latter topic, the former topic will be discussed throughout the remainder of this chapter.

In this Section, we will explore the design considerations of the scheduler, with emphasis on the topology of the reconfigurable token ring responsible for the generation of the *CTR* signals in Fig. 3.8(b). Of primary interest is the issue of the capacitive loading on the individual devices which comprise the token ring. More specifically, the maximum number of fan-in and/or fan-out connections for each device in the token ring comprising the scheduler must be limited to a small number (i.e., less than 5) regardless of the length of the token ring or the number of configurable modes. In basic terms if the number of fan-in or fan-out connections on a single device becomes too large, then it can lead to both uneven delays relative to the other devices in the token ring due to capacitive differences, as well as problems with device fabrication when using standard-cell libraries [76].

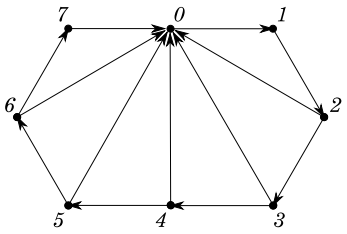
To that end, we will explore how the vertices in the embedded cycle graph composing the reconfigurable token ring can be distributed to support several configurable modes while simultaneously limiting the maximum degree of input and output connections present at each vertex. First, Section 3.3.1 will cover the... Second, Section 3.3.2 will illustrate a distribution algorithm which generates an interesting, yet sub-optimal, token ring topology that will recur again in Chapter 4.3. Finally, Section 3.3.3 will present a distribution algorithm which generates an optimized variant of the token ring topology in Section 3.3.2.

3.3.1 *Cyclic Behavior of Token Rings*

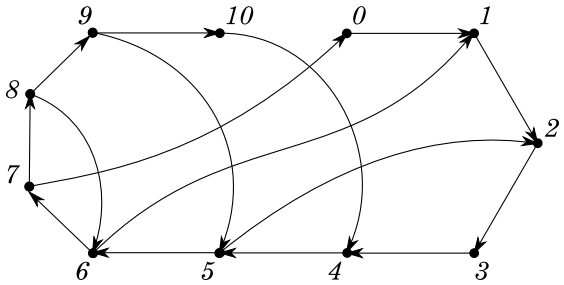
Cycles were discussed briefly in Chapter 2. Recall, cycle graphs are composed from a closed set of vertices linked together by edges. The connectivity and directionality of the edges in the cycle, are defined by the adjacency of the vertices in the graph. If the cycle forms a closed loop where each element has a single input and a single output, then it is referred to as a *ring*. If the vertices in the ring are traversed via the use of a specific bit pattern, called a token, then it is referred to as a *token*



(a) with a single cycle of 8 vertices



(b) with several cycles and a maximum cycle length of 8 vertices



(c) with several cycles, a maximum degree equal to 4, and a maximum cycle length of 8 vertices

Figure 3.10: Cycle graphs of a token ring.

ring. Via the use of token ring cycle graphs, we can create a visual representation of the configurable modes in the control device of Fig. 3.8(b).

A cycle graph of a token ring with eight vertices is depicted by Fig. 3.10(a). In Fig. 3.10(a), vertices 0 and 1 are adjacent since the token travels from 0 to 1 as it travels around the loop. The path from vertex 0 to 7 within the token ring constitutes a *cycle*. However the token ring in Figure 3.10(a) only contains a single cycle. If a single token ring contains more than one cycle, the cycles are said to be *embedded* within the token ring. Figure 3.10(b) shows the same token ring as Figure 3.10(a), but now instead of having just one cycle of length 8, it now contains cycles of length 3, 4, 5, 6, 7, and 8. As vertex 0 exists as a part of all the cycles in Fig. 3.10(b), we can say that 0 is a *shared vertex* between all of the configurable modes in the cycle graph of Fig. 3.10(b).

However, in Fig. 3.10(b) vertex 0 also accumulates a disproportionately high number of input connections relative to the other vertices in the cycle graph. Furthermore, because the number of cycles necessary to implement reconfiguration increases with the number of configurable modes desired in the scheduler, we can say that both the capacitive loading and delay associated with vertex 0 will be negatively impacted as the number of cycles and configurable modes in the cycle graph of Fig. 3.10(b) grows larger [76]. Thus, the cycle graph of Fig. 3.10(b) is insufficient for the reconfiguration needs of the wagging synchronizer defined in Section 3.2, due to the aforementioned scalability issue. In order to overcome this issue, another cycle graph must be defined in which the number of incoming and

Table 3.3: Differences in the Cycle Lists of a Multi-cycle Token Ring with Distributed Edges vs. an Undistributed Ring (Maximum Cycle Length = 8 Vertices)

	Simple		Distributed (sub-optimal)
Cycles	$\overline{0},\overline{1},\overline{2} \rightarrow 3$ $\overline{0},\overline{1},\overline{2},3 \rightarrow 4$ $\overline{0},\overline{1},\overline{2},3,4 \rightarrow 5$ $\overline{0},\overline{1},\overline{2},3,4,5 \rightarrow 6$ $\overline{0},\overline{1},\overline{2},3,4,5,6 \rightarrow 7$ $\overline{0},\overline{1},\overline{2},3,4,5,6,7 \rightarrow 8$	Odd Cycles	$\overline{6},\overline{7},\overline{8} \rightarrow 3$ $5,\overline{6},\overline{7},\overline{8},9 \rightarrow 5$ $4,5,\overline{6},\overline{7},\overline{8},9,10 \rightarrow 7$
		Even Cycles	$\overline{2},\overline{3},\overline{4},\overline{5} \rightarrow 4$ $1,\overline{2},\overline{3},\overline{4},\overline{5},6 \rightarrow 6$ $0,1,\overline{2},\overline{3},\overline{4},\overline{5},6,7 \rightarrow 8$
Shared Vertices	0, 1, 2		None

outgoing connections (i.e., the maximum degree) at any given vertex is limited regardless of how large the cycle graph grows. In other words, we must create an embedded cycle graph with *distributed connections*. An example of such a cycle graph is shown in Fig. 3.10(c).

Before proceeding further, let us examine the properties of the cycle graph of Fig. 3.10(c). First, let us note that the maximum number of edges at any given vertex in Fig. 3.10(c) is equal to 4 (though it might be less than that). Second, let us note that by distributing the number of edges within the cycle graph of the token ring in Fig. 3.10(c), while preserving the same number of embedded cycles as in Fig. 3.10(b), we incur additional overhead in terms of the total number of vertices necessary to realize the cycle graph of the token ring (i.e., 11 vertices in Fig. 3.10(c) vs. 8 in Fig. 3.10(b)).

Let us further examine the differences between Fig. 3.10(b) and Fig. 3.10(c) via the use of Table 3.3, which compares the cycle lists of each graph to one and another. In Table 3.3, we will refer to the entries regarding Fig. 3.10(b) as the *Simple* case, while the entries regarding Fig. 3.10(c) will be referred to the *Distributed (sub-optimal)* case. The term *sub-optimal* is used because, as will be shown in Section 3.3.3, there is a more efficient way to construct a distributed cycle graph than shown in Fig. 3.10(c). The overbars in Table 3.3, represent the *shared vertices* between multiple cycles. This point bears mentioning because if there are configurable modes in the cycle graph of Fig. 3.10(c) which are *disjoint* from each other (i.e., there exist modes which share no common vertices), then it poses an interesting problem. Namely, how can a control token reach disjoint configurable modes in the absence of a shared vertex? The mathematical properties of the cycle graph in Fig. 3.10(c) garner some insight into this issue:

1. If graph, G, is a multi-cycle token ring, then for all vertices ΔG (i.e., $\lceil \deg(G) \rceil$) = 4.

Algorithm 1 Sub-optimal Distribution Algorithm for a Static Token Ring (Part 1)

Input: $X := \left|cycle_{(max)}(odd)\right|$, $L := \left|cycle_{(max)}(even)\right|$,
$\boldsymbol{E} := [a_n]$, a_0 to a_n are binary, $|\boldsymbol{E}| := L/2$
$\boldsymbol{O} := [b_m]$, b_0 to b_m are binary, $|\boldsymbol{O}| :=$ **floor** $(X/2)$
Let L be even, $L > X$

Output: Collection of objects (memory addresses) whose connections model a distributed token ring of length $X+L/2$, with many cycles, and maximum adjacency $k = 4$

```
1:  object *list, *current
2:  list ← new object // get a pointer to the first object in the list
3:  for i = 0 to L − 1 do
4:      current → index ← i
5:      current → norm_tail ← new object
6:      if i ⩾ L/2 then
7:          if E[i − (L/2)] ← true then
8:              current → even_fb_tail ← find_index (list, L − i)
9:          else
10:             do nothing
11:     else
12:         do nothing // no feedback paths prior to L/2 − 1
13:     current ← current → norm_tail // advance to the next object in the list
14:     increment i
```

2. If graph, G, is a multi-cycle token ring, then the intersection of the set of vertices $cycle_{min(even)}$ and $cycle_{min(odd)}$ may be equal to the null set. (i.e., some cycles have independent sets of vertices)

3. If graph, G, is a multi-cycle token ring, then the intersection of the set of vertices $cycle_{min(even)}$ and $cycle_{max(odd)}$ can not be equal to the null set, and vice versa. (i.e., there will always be a route between these two sets).

While property 2 states that there *may not* be a direct path between the minimum length odd ($cycle_{min(odd)}$) and even ($cycle_{min(even)}$) cycles in Fig. 3.10(c), property 3 indicates that, in this topology, there *will always* be an indirect path between them via the maximum length odd ($cycle_{max(odd)}$) and even ($cycle_{max(even)}$) cycles. Chapter 4.3 will discuss how to construct a reconfiguration protocol which makes use of this observation.

3.3.2 *Sub-Optimal Distribution Algorithm Specification*

In Section 3.3.1, we explored the properties of a (sub-optimal) embedded cycle graph for a distributed multi-cycle token ring which satisfies the requirements for the scheduler aspect of the wagging controller discussed in Section 3.2.6. As stated earlier, distributing the edges in a multi-cycle token ring, serves two core purposes. By limiting the maximum number of vertices in the graph, we can:

1. Reduce the capacitance of the fan-in and fan-outs of the resulting circuit implementations.

2. Equalize the delays between the devices/nodes of the resulting circuit.

The algorithms for generating the embedded cycle graph for a distributed multi-cycle token ring, in the same vein as the specific example depicted in Fig. 3.10(c), are outlined below. Both of them require four pieces of information a priori:

1. The cycle length containing the maximum number of even vertices (L).

2. The cycle length containing the maximum number of odd vertices (X).

3. An array of length L/2 containing the information used to construct the distributed feedback paths for the even cycles (E).

4. An array of length $\lfloor X/2 \rfloor$ containing the information used to construct the distributed feedback paths for the odd cycles (O)

E and *O* contain information of a binary type, indicating either the presence (true) or absence (false) of a feedback path at that index location. Using this information, it is possible to dynamically construct a linked list of objects, where the connectivity between the memory addresses in the list models the connectivity of a distributed token ring. Figure 3.10(c) used the following parameters: L=8, X=7, E = [0,1,1,1], O = [1,1,1].

The objects themselves are data structures which contain the following 4 pieces of information:

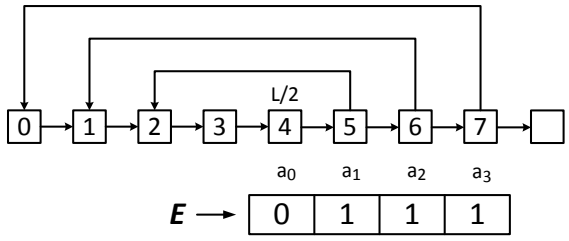
1. index: integer data type.

2. norm_tail: pointer data type

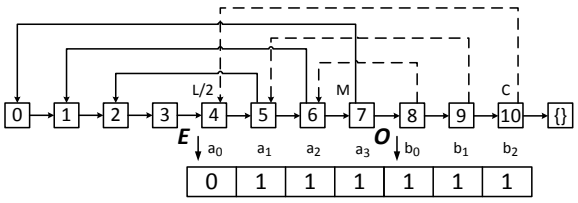
3. even_fb_tail: pointer data type

4. odd_fb_tail: pointer data type

Part 1 of the algorithm deals with the construction of the even cycles within the distributed token ring, where it is assumed that $L > X$. The algorithm begins by getting a pointer to the first object in the list, and then enters the initial **for** loop. The index value of the current data object is then assigned a unique number, which is equal to the present value of the loop counter *i*. Additionally, the norm_tail (normal tail) data field of the present object is linked to a new object of the same type for later use when the loop is incremented. If the current value of the loop counter is greater than or equal to *L/2*, then the data entry at location *i-(L/2)* in the E array is checked for the presence of a feedback path. If one exists, then a linear search function `find_index()` is called which accepts a pointer to the head node of the list, and a location (*L-i* in this case) as its inputs. The return value of the function is a pointer to the location in the list which contains an index value equal to *L-i*. This result is then assigned to data field even_fb_tail (even feedback tail) in the present object. The loop then advances the pointer to the next object in the list and increments *i* by 1. It is worth noting, that checking does not occur



(a) after building the even cycles in the token ring (L=8, X=7, E=(0,1,1,1), O(1,1,1))



(b) after building the odd cycles and linking them to the list of previously created objects

Figure 3.11: Results of the sub-optimal distribution algorithm

before i = $L/2$-1, because no feedback paths can exist prior to that point. Figure 3.11(a) depicts the results of employing this algorithm with the same parameters as Figure 3.10(c).

Continuing onward, the second half of this algorithm incorporates some additional constants. C is a control invariant used to indicate whether or not the pointer has reached the "end of the list", while M represents the midpoint of the odd cycle list (serving as an equivalent to the $L/2$ control term in part 1). Lines 5-11 in part 2 are equivalent to lines 4 & 5 in the above algorithm. The test as to whether or not j has exceeded L is intended to prevent aliasing with previously created objects, while the test for $j < C$ is used to decide whether to create a new object, or to set the next pointer location to **null** (if the end of the list has been reached). Similarly, line 13 & 14 in Algorithm 2 are equivalent to line 7 & 8 in algorithm 1. Array O is checked for presence of feedback paths now, the odd_fb_tail is being used instead of the even_fb_tail, and the find_index() function searches for an index value of C-j+L/2 instead of L-i. The resulting distributed token ring is shown in Figure 3.11(b), which matches the results of Section 3.3.1. The overhead associated with this method is the addition of an extra (X+L/2) - L new vertices to the maximum cycle length ($cycle_{(max)}(even)$) of the original behavior graph.

3.3.3 *Optimal Distribution Algorithm Specification*

While the algorithms in Section 3.2.2 create balanced distributed feedback paths as described, they do not form the basis of an optimal solution. It was Dr. Andrey Mokhov who suggested that it is also possible to create an embedded cycle graph based on pancyclic rings, which possesses the same end behavior as the embedded

Algorithm 2 Sub-optimal Distribution Algorithm for a Static Token Ring (Part 2)

```
1:  C := X + L/2 − 1 // control invariant used in second for loop
2:  M := C/2 + L/4 // conditional control invariant used in midpoint calculation
3:  current ← find_index (list, L/2) // set current index to L/2
4:  for j = L/2 to C do
5:    if j ⩾ L then
6:      current → index ← j
7:      if j < C then
8:        current → norm_tail ← new object
9:      else
10:       current → norm_tail ← null // at end of list
11:   else
12:     do nothing // prevent aliasing with previously created objects
13:   if j > M then
14:     if O[j − (M + 1)] ← true then
15:       current → odd_fb_tail ← find_index (list, [C − j + (L/2)])
16:     else
17:       do nothing
18:   else
19:     do nothing // no feedback paths prior to M
20:   if j < C then
21:     current ← current → norm_tail // advance to the next object in the list
22:   else
23:     do nothing // at end of list
24:   increment j
```

cycle graph in Section 3.3.2, but which lack the drawback of not having a set of shared vertices as the cycle graph grows an arbitrary size with an arbitrary number of configurable modes. As these shared vertices can be used as interrupt points for the reconfiguration protocol discussed in Section 4.3, this optimized algorithm allows reconfiguration to be performed in a single configuration operation, without the need for intermediate configuration steps.

As before, we can compare the differences in the embedded cycle graphs produced by the two algorithms by comparing their cycle lists via the use of Table 3.4. In Table 3.4, we will refer to the entries regarding Fig. 3.12(b) as the *Distributed (optimal)* case, while the entries regarding Fig. 3.11(b) (or alternatively 3.10(c), as they are the same) will be referred to the *Distributed (sub-optimal)* case. As before, the overbars in Table 3.3, represent the *shared vertices* between multiple cycles.

The properties of the *sub-optimal* case of Fig. 3.11(b) (Fig. 3.10(c)) in Table 3.4 are the same as they were in Table 3.3 in Section 3.3.1. However, when taking a look at the *optimal* case of Fig. 3.12(b) it can be observed that vertices 0, 2, 3, and 7 are shared between all configurable modes. . As a result, properties 2 and 3 of the embedded cycle graph of Fig. 3.11(b) (i.e., Fig. 3.10(c)) are no longer applicable to this algorithm. Thus, the mathematical properties of the embedded cycle graph in Fig. 3.12(b) are as follows:

Table 3.4: Differences in the Cycle Lists of Two Different Multi-cycle Token Rings Implementations with Distributed Edges (Maximum Cycle Length = 8 Vertices)

	Distributed (optimal)		Distributed (sub-optimal)
Cycles	Does Not Exist	Odd Cycles	$\overline{6}, \overline{7}, \overline{8} \rightarrow 3$
	$\overline{0}, \overline{2}, \overline{3}, \overline{7} \rightarrow 4$		$5, \overline{6}, \overline{7}, \overline{8}, 9 \rightarrow 5$
	$\overline{0}, 1, \overline{2}, \overline{3}, \overline{7} \rightarrow 5$		$4, 5, \overline{6}, \overline{7}, \overline{8}, 9, 10 \rightarrow 7$
	$\overline{0}, \overline{2}, \overline{3}, 5, 6, \overline{7} \rightarrow 6$	Even Cycles	$\overline{2}, \overline{3}, \overline{4}, \overline{5} \rightarrow 4$
	$\overline{0}, 1, \overline{2}, \overline{3}, 5, 6, \overline{7} \rightarrow 7$		$1, \overline{2}, \overline{3}, \overline{4}, \overline{5}, 6 \rightarrow 6$
	$\overline{0}, 1, \overline{2}, \overline{3}, 4, 5, 6, \overline{7} \rightarrow 8$		$0, 1, \overline{2}, \overline{3}, \overline{4}, \overline{5}, 6, 7 \rightarrow 8$
Shared Vertices	0, 2, 3, 7		None

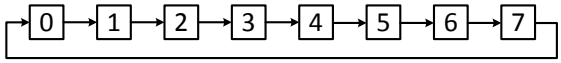
1. If graph, G, is a multi-cycle token ring, then for all vertices ΔG (i.e., $\lceil deg(G) \rceil$) = 3.

Comparing this property to property 1 in Section 3.3.1, it can be stated that the maximum degree of any vertex in the embedded graph of the multi-cycle token ring of Fig. 3.12(b) is less than that of 3.11(b) (i.e., Fig. 3.10(c)). Hence the algorithm in Section 3.3.3. is optimized in terms of capacitive loading on the vertices of the embedded cycle graph (i.e., the capacitive load is reduced).

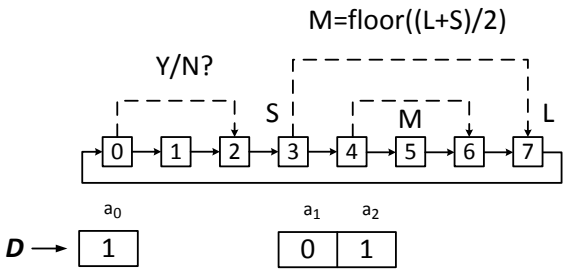
The optimized distribution algorithm in this section can be modeled by taking a basic linked list of objects, as shown in Figure 3.12(a) and adding a few parameters. The objects themselves are data structures which contain the following 3 pieces of information:

1. index: Integer data type. Used to order the entries in the linked list.
2. norm_tail: Pointer data type. Contains the *initial* path to the next object in the list.
3. jump_tail: Pointer data type. Contains the *secondary* path to the next object in the list.

On the first pass, the objects in in the list from 0 to L will be created and linked along the *normal tails*. When the final entry is created, the norm_tail of this entry will be given a pointer to the head of the list, as in Figure 3.12(a). Using this linked list, the *jump tails* can be constructed via the application of a few observations. When the second pass begins, index entry 0 is given a pointer to object 2. This is a property of the graph. This tiny portion of the graph functions almost like a query, with the pertinent question being "odd or even?" As for the subsequent jump tails, the index values of the resulting connections can be computed as follows. If we consider object 3 to be the starting point of the graph after the omission of the aforementioned query, which will hereafter be referred to as S, then it turns out



(a) Basic linked list



(b) Anotated diagram outlining the methodology for creating abstract distributed feedback loops

Figure 3.12: Optimized distributed feedback algorithm overview

that the values of the target indexes for the jump tails can be statically calculated as follows.

1. Add S to the highest index value and divide by 2 (take the floor function of the result). This will yield M (the midpoint of the graph).
2. Since M is located in the middle of the graph and the feedback (feedforward) path target is located a symmetric distance away on the other side of the midpoint, the target index, T, at any given moment can be calculated as T = M + (M-i) where i represents the current position of the list pointer.

In this manner, all of the paths can be constructed in a total of two complete iterations. Further to the above information, one point bears mentioning. Namely, if the total number of vertices in the system is odd the graph will contain two possible midpoints. However, since M is defined by the floor function of the calculation in step 1 of the above procedure, the path target ,T, defined in step 2 can be adjusted by adding substituting *M* in the first operand with *M+1* (i.e. T = (M+1) + (M-i)).

We also require one more piece of information to complete the algorithm. More specifically, we require an array variable $D[a_n]$ to store the state of the distributed feedback paths in the system, where n ranges from 0 to *M-S*. *D* controls whether or not a secondary path is active (if D = 1) after the initial creation of the graph, as in Figure 3.12(b). $D[a_0]$ is always linked to the object with an index value 0, whereas entries $D[a_1]$ to $D[a_n]$ are linked to objects with index values of *S* to *M-1*.

3.4 TOKEN RING DESIGNS

While Section 3.3.2 dealt with the construction of a distributed token ring through the use of a linked list of objects, where the directed connections between the memory addresses modeled the connectivity of the underlying embedded behavior

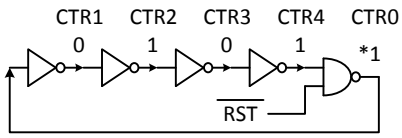


Figure 3.13: Ring oscillator with $n_{gate} = 5$, and an active-low reset signal

graph, the following subsection illustrates how the same token ring is constructed using both combinational and sequential circuit elements. As stated in the introduction, each of the implementations shall be compared and contrasted against each other in terms of both power and latency, to illustrate the tradeoffs present between them.

3.4.1 *Ring Oscillators*

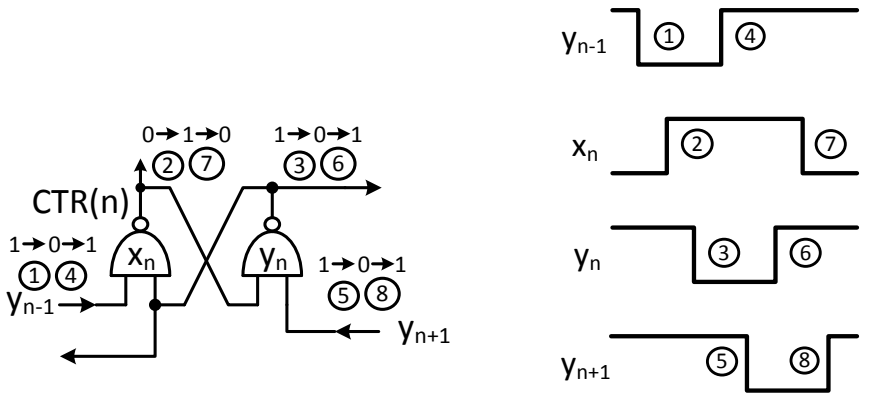
The first token ring implementation under study is a *ring oscillator*. A ring oscillator is token ring which is constructed from an odd number of inverting elements (the minimum number is 3). All of these elements are built using *combinational logic* , which means that the outputs from the inverting gates of the token ring are a function of the present inputs only (i.e. the circuit is *memoryless*). Consequently, the resulting token ring circuit is simpler than its sequential counterparts and uses less area on the die [66]. As an example, Figure 3.13 depicts a ring oscillator where the number of gates, n_{gate}, in the inverter chain is equal to 5. The ring oscillator has a cycle time, t_{cycle} , which is governed as follows:

$$t_{cycle} = n \times \left(\frac{t_{pLH} + t_{pHL}}{2} \right) \tag{18}$$

in which t_{pLH} and t_{pHL} are the low-to-high and high-to-low propagation delays of each inverter, respectively. The duty cycle of this token ring design is 50% of the clock period, which allows only half of the cycle time to be used for synchronization. The oscillator is set by forcing a zero on the reset line, *RST*, which determines the default values at each of the other gates as shown in Figure 3.13. The "token" in this ring of inverters is represented by the [1, 1] output signal pair which begins to propagate through the chain of devices once RST goes high.

3.4.2 *Fast David Cells*

We briefly touched on David Cells back in Section 2.1.3, and now we will continue the discussion [21]. In contrast to the combinational token ring implementation based on ring oscillators from Section 3.4.1, token ring control circuits based on *sequential logic* use both the present and prior inputs of the system to determine their output values. While such systems are more difficult to design and verify, they also offer additional time available for synchronization when compared to



(a) Fast DC. From [73].

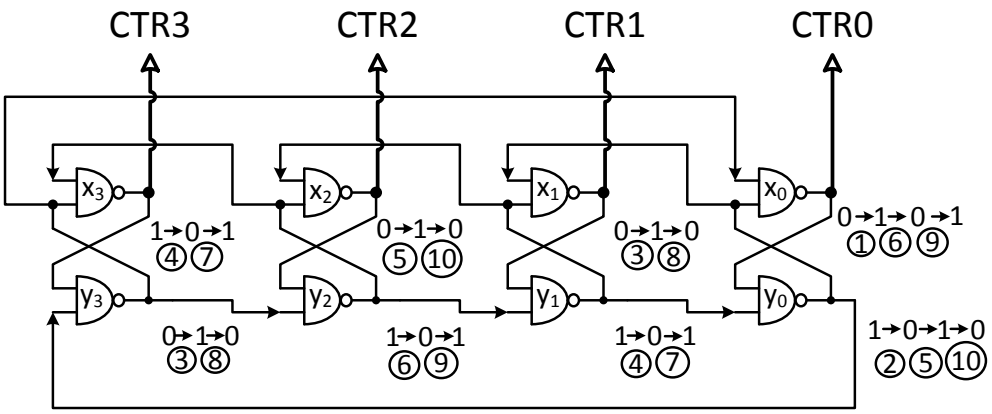
(b) Timing diagram.

Figure 3.14: Annotated fast David Cells schematic with signal transitions representing 1 operating cycle

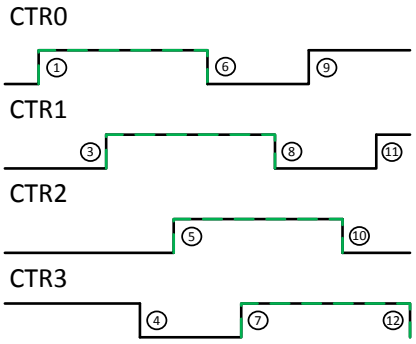
their combinational counterparts. Primarily, this is due to the sampling time of the master latch, t_{sample}, being governed by the so-called *mark to space ratio* as defined by the underlying STG. Figure 2.4(a) shows the progression of the signal transitions in a *standard* DC, while Fig. 2.4(b) shows its timing diagram. As stated previously, the circled numbers in Fig. 2.4(b) represent the individual transitions generated internal to the cell, starting at transition 1. The "mark" is represented as the number of transitions during which the *CTR* signal is high, while the "space" corresponds to the time where it is low. Using Figure 2.4(a) as a reference, we can see that the mark lasts for 9 transitions (i.e. from {2} to {11}), while the space is a variable length. As it takes 7 transitions for the signal in {1} to reach the same relative location in the next DC (at transition{7}), the space can be expressed as 7 * $(L-1)$ where L is equal to the number of DCs in the token ring. It is useful to note that the mark is actually invariant to changes in the cycle length of the token ring, and this property holds for all the sequential token ring implementations. Consequently, the time allotted to recover from metastability (i.e. the time available for synchronization) increases as the number of devices in the chain increases, assuming that the system samples on the mark, and recovers on the space.

As it turns out, the latency of the DCs in the token ring can be improved by incorporating *relative timing constraints* (as previously mentioned in Section 2.3.4) into the specification. If the lower NAND gate in Figure 2.4(a) is removed, it allows a token to propagate forward to the next DC in the system concurrently with the reset phase of the current DC (i.e. transitions {4} to {8}. This circuit structure is commonly referred to as a *fast DC*, and is depicted in Fig. 3.14(a), while its timing diagram is shown in 3.14(b) [73]. The *set* phase of a fast DC occurs during transitions {1} to {3}, while the reset phase occurs during transitions {4} to {8}.

Thus, two additional fast DCs are required in order to provide the 5 transitions necessary for the CTR signal to reset [73]. This latency requirement has been referred to as *token spreading* in prior literature. This is somewhat misleading though



(a) Annotated schematic with 10 signal transitions

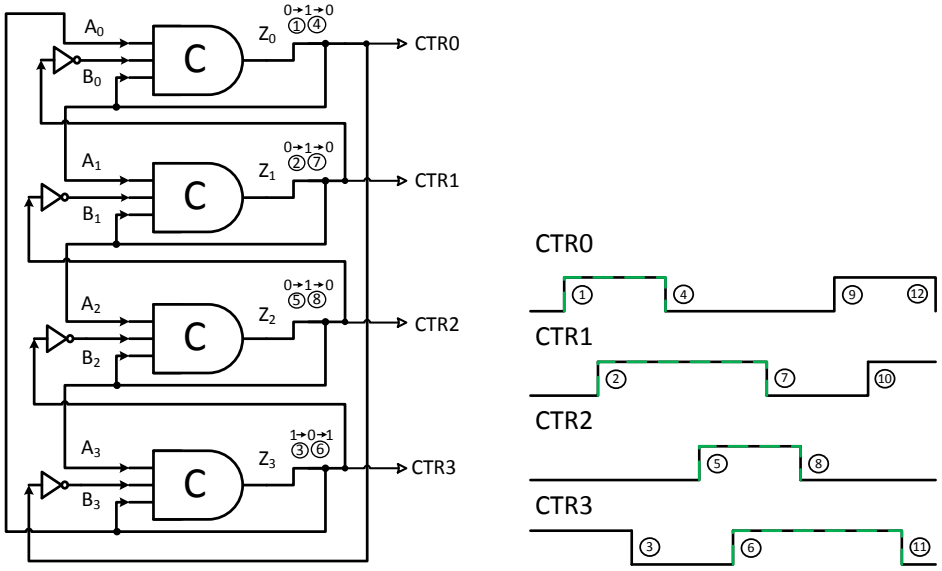


(b) Waveform illustrating the output (CTR) signals

Figure 3.15: Four-way sequential token ring based on fast DCs

because a fast DC takes 8 transitions to completely stabilize from an initial perturbation when placed in a token ring configuration. Consider the diagram in Figure 3.15, which depicts a token ring based on fast DCs where L=4. One possible path through the system is [x0+,y0-,x1+,y1-,x2+,y2-/x0-,x3+,y3-] which takes 8 transitions to complete before returning to position x0+ on transition 9. Another (reverse) path through the system is [x0+,y0-,x1+,y1-,x2+,y2-,y1+] which takes 7 transitions, and is invariant to changes in the length of the token ring. If the token ring only had L=3 it would return to x0+ on transition 7, where both x0+ and y1+ act as inputs to the y0 NAND gate on its falling (y0-) transition. Also note that an input vector of [1 1] causes the state of the NAND gate to change from 1 to 0. In short, a token ring using fast DCs with L=3 results in a race condition, which is made worse by the symmetry inherent to the system. Simply put, if this race condition occurs during the reset phase of y0, then is also *guaranteed* to occur in y1, y2, and y3 during the same portion of their reset phase. Furthermore, because the token ring is cyclic this race condition will *always happen*.

By contrast, since the number of transitions necessary to traverse the token ring of length L=4 in the forward direction exceeds 8, the race condition de-



(a) Annotated schematic with eight signal transitions

(b) Waveform illustrating the output (CTR) signals

Figure 3.16: Four-way sequential token ring based on MCEs

scribed above can not happen, as illustrated in Figure 3.15. The fast DC has a few advantages over its standard counterpart. Reduction in the latency of the sampling interval (mark) by 44% is one such advantage, though more will be discussed in section 3.4.4. Initializing the token ring in Figure 3.15 requires that the two-input NAND gates labeled [yo,y1,y2, and x3] be replaced with three-input NAND gates so that the token ring can be initialized and driven by and active-low control signal. The circuit is initialized with the following data from $\langle x0, y0\rangle$ to $\langle x3, y3\rangle$: $\langle 0,1\rangle$ $\langle 0,1\rangle$ $\langle 0,1\rangle$ $\langle 1,0\rangle$

3.4.3 *Muller Pipeline*

In Chapter 2.1.3, the MCE was introduced. [56]. In this subsection, we shall explore the utility of a sequential token ring controller that is constructed from MCEs. With that in mind, Figure 3.16(a) depicts a 4-way sequential token ring composed of MCEs, while Fig. 3.16(b) represents its timing diagram. As before with Fig. 2.3(b) in Section 2.1.3, the circled numbers in Fig. 3.16(b) represent the individual transitions generated internal to the cell, starting at transition 1. The one marked difference between this sequential implementation and that of 3.4.2 is that the "mark" in a MCE token ring isn't a precise value. Using the example of Figure 3.16(b), the duration of time where the control signals are high varies from 3 transitions, to 5 transitions. To characterize it more plainly, the average time available for synchronization in a tends to converge to a mean value rather than being cast in stone like in sections 3.4.1, and 3.4.2 The control signals in the token ring are tapped from the inverter outputs (i.e. Z_0 to Z_3). The default initializations, values for the control lines from Z_0 to Z_3 are [0001], respectively.

3.4.4 *Performance Comparison*

Before comparing the token ring designs to each other it is necessary to specify the objective of these tests. In Section 3.2.4, it was mentioned that one of the applications of a wagging synchronizer that incorporates reconfigurable hardware design is to adjust for subtle variations in timing variations introduced by PVT parameters. However, because the circuit structures which comprise the cells in the token ring chains all have differing propagation delays, the number of cells necessary to approximate a target duty cycle is different for each design style.

In order to fairly compare the combinational and sequential token rings to each other, each token ring design must:

1. Target a fixed duty cycle which will act as frequency reference for every token ring implementation.
2. Operate at a fixed supply voltage.
3. Operate at a fixed temperature.

If those three conditions are met, then a corner-analysis can be used to observe the impacts of processing parameters on both the power consumption and the time available for synchronization in the scheduler portion of the wagging controller introduced in Section 3.2.6. The examples in Table 3.5 and 3.6 used the following parameters:

1. Target duty cycle = 1 ns (Typical-Typical Case, UMC 90nm SP Technology)
2. Supply voltage = 1.0V
3. Temperature = 27 °C

All of the designs employ optimal sizing rules and are simulated via CADENCE. The *numerals* at the front of each design entry correspond to the *number of cells* in the token ring chain (i.e. 12, 16, 19, etc.) required to meet the 1 ns target frequency in the Typical-Typical case, followed by the *design style* under consideration (i.e. Ring Oscillator (RO), Muller Pipeline (MP), and fast DCs (DC)). The number of cells impacts the natural frequency of the physical implementation. For example, in order to approximate the 1 ns target frequency the ring oscillator requires 37 cells in the device chain, while the sequential implementations based on the Muller Pipeline and fast DCs require 12 and 19 cells, respectively. However, these results do not take capacitive loading into account, and therefore the number of cells required in the actual token ring to approximate the target frequency of interest can be lower in practice than stated above. As shown in Table 3.5, the combinational logic implementation of the token ring has the lowest average power consumption per cell, due to the simplicity of the underlying physical implementation. The average power values in Table 3.5 were measured over a time interval of 1 ns. Data values in Table 3.5 can be extended to a token ring of arbitrary length by multiplying the average power per cell by the number of cells in the token ring configuration of interest. Therefore, the power consumption is proportional to the number of devices in the token ring chain.

Table 3.5: Average Power Consumption per Cell across Process Corners at V_{DD} = 1.0 V and t_{cycle} = 1 ns (TN/TP)

Design	SN/SP (μW)	FN/SP (μW)	TN/TP (μW)	SN/FP (μW)	FN/FP (μW)
12MP	5.049	7.143	7.493	7.765	10.182
19DC	1.869	3.050	2.744	2.785	3.817
37RO	0.675	1.397	1.326	1.329	1.719

Table 3.6: Average Time Available for Synchronization across Process Corners at t_{cycle} = 1 ns (TN/TP)

Design	SN/SP (ps)	FN/SP (ps)	TN/TP (ps)	SN/FP (ps)	FN/FP (ps)
12MP	1046.0	841.65	784.64	729.57	618.03
19DC	1,132.4	862.94	842.30	836.81	677.27
37RO	691.58	510.82	502.65	503.50	396.71

On the other hand, the values in Table 3.6 illustrate that the sequential token ring implementations exhibit additional time available for synchronization when compared to their combinational counterparts. The fast DC design outperforms the Muller Pipeline in terms of time available for synchronization and power consumption, due to the simplicity of the former design. It should be noted that the above results in Table 3.6 do not take into account the setup and hold times of the slave latch, which must be deducted from the time available for synchronization above in order to ensure that the design satisfies the required MTBF.

3.4.5 *Distributed Token Ring Implementation*

Given the prior discussion on token ring topologies, let us now examine a practical example of how an embedded token ring controller changes its synchronization parameters based on application of various one-hot codes. An implementation of an 8-way reconfigurable token ring controller based on fast DCs with 5 configurations is shown in Figure 3.17. The output behavior of the controller is modeled by the graph in Figure 3.19. Signals CTRLA, CTRLB, CTRLC, CTRLD, and CTRLE relate to the 4, 5, 6, 7, and 8 cell token ring configurations, respectively. As an example, the outputs of the 7 cell configuration are [CTR3, CTR4, CTR5, ..., CTR9] while the outputs of the 8 cell configuration are [CTR0, CTR1, CTR2, ..., CTR7]. The feedback paths within the circuit are controlled via one-hot codes (RDATA) which are generated via an INTMUX element, which will be discussed in Section 4.5. The operation of the system is also similar to the graphs shown in Figures 3.10(c) and 3.11(b), with a few exceptions. Due to the relative timing constraints imposed

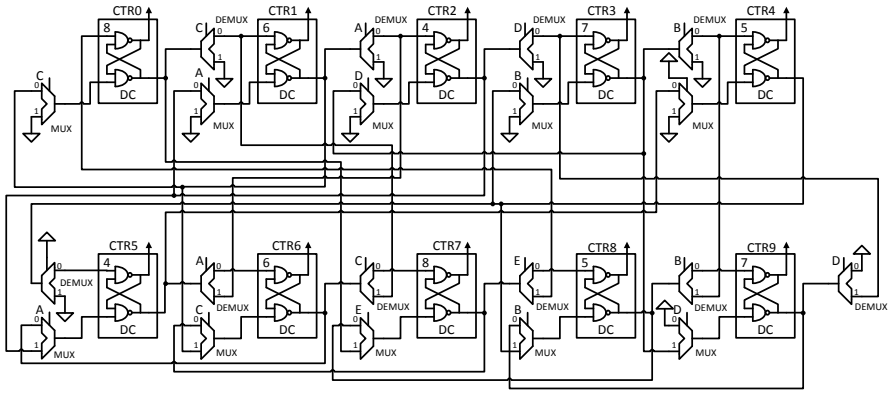


Figure 3.17: 8-way reconfigurable token ring implementation based on sequential logic (5 possible configurations)

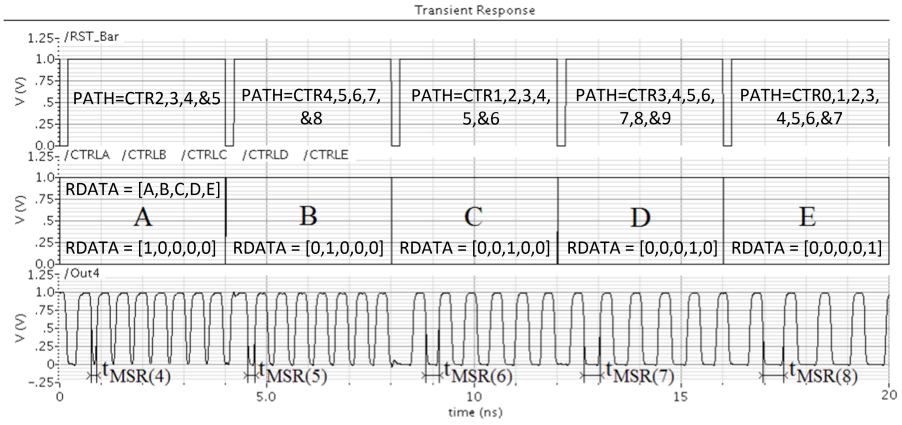


Figure 3.18: Transient response of the self-timed reconfigurable token ring control circuit based on DCs illustrating the effect of increased parallelism on the time available for synchronization (path: CTR4)

on token ring circuits utilizing fast DCs, the 3-cell path (CTRL5, CTRL6, CTRL7) does not exist (i.e. $b_0 = 0$ in Figure 3.11(b)). Recall that token rings constructed using fast DCs require a minimum of 4 elements in order to both satisfy the requirements of the underlying STG, and also to avoid the data racing conditions outlined in Section 3.4.2 [73]. Furthermore, the graph in this subsection differs from those discussed in section 3.3, by using the absent location present in the graphs of Figures 3.10(c) and 3.11(b) to reduce the overhead associated with the process of distribution. This is a topic which will merit further discussion later on in the thesis. In all cases, the average distance between successive output elements is uniform and equal to 1.

Figure 3.18 depicts the transient response of a single output (CTR4) of the token ring over a time interval of 20 ns, where each one-hot code is active for a time of 4 ns. As the one-hot codes move from CTRLA to CTRLE the distance between the falling edge of Out4 (CTR4) and its next rising edge increases. This region

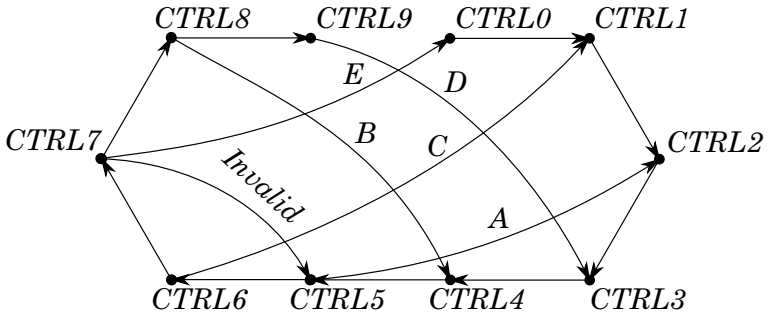


Figure 3.19: Distributed graph which models the behavior of the 8-way reconfigurable token ring circuit.

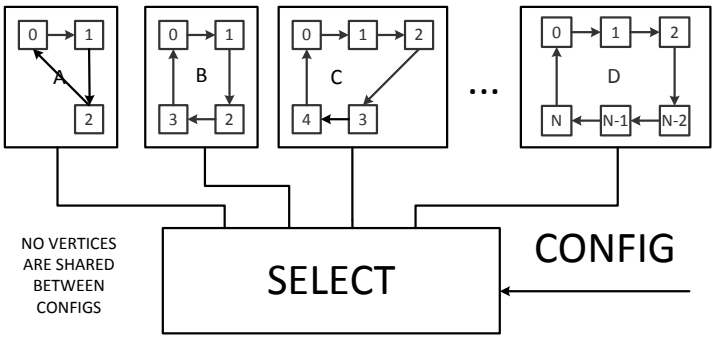


Figure 3.20: Basic reconfigurable control circuit based on hardware replication and select signals.

($t_{MSR}(4)$, $t_{MSR}(5)$, $t_{MSR}(6)$, etc.) represents the time available for synchronization of the master latch in a cascaded flip-flip synchronizer, neglecting the setup and hold times of the slave devices. The interrupt module was tested separately from the token ring in these tests. In lieu of generating the control signals from the interrupt module, a reset signal (RST_Bar) is triggered directly after performing reconfiguration. This process flushes the old control token out of the system, and initializes a new control token for use along the new configuration. This reset signal is unnecessary when the token ring is connected to the interrupt modules, but it is useful for testing purposes. The time available for synchronization of each master latch increases with an increasing number of devices in the chain and Figure 3.18 still demonstrates that t_{MSR} can be controlled via the use of one hot codes. Thus, the foundation of the design is sound. Furthermore, though a combinational implementation of the token ring has not been discussed, the operation is similar to the sequential design. The only noticeable difference is that there are no even configurations in the graph of the system as defined by Figures 3.10(c) and 3.11(b), due to ring oscillators requiring an odd number of elements to function.

This Chapter will close with a discussion on the general area costs of the of the distributed token ring controller described in Section 3.3.2. The multiplexer (MUX) and de-multiplexer (DEMUX) elements,constitute the extra hardware overhead along the critical path when using this structure. To put this in perspective, the

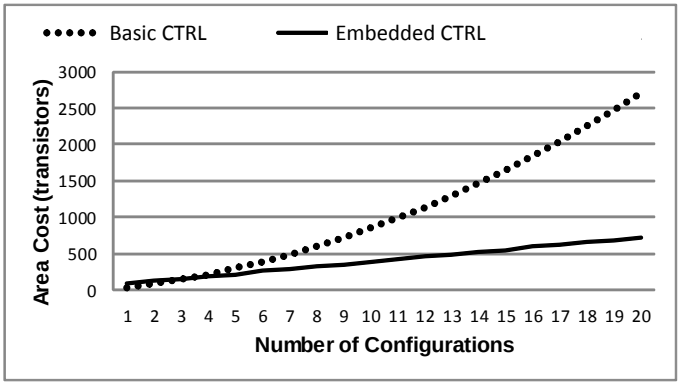


Figure 3.21: Area comparison of a self-timed reconfigurable token ring control circuit based on fast DCs. $(cycle_{(min)}(even) = 4)$

overhead costs of using a distributed token ring can be compared against the costs of implementing the reconfigurable control circuit by using separate token rings for each configurable mode and using a control signal to switch between them, as in Fig. 3.20. The cost function of this method in terms of total vertices (devices), V_{total}, is defined as follows:

$$\begin{aligned} V_{total(Basic)} &= V_{config(k-1)} + V_{config(k-2)} + V_{config(k-1)} \\ &+ \ldots + V_{config(k-i)} + \ldots + V_{config(1)} V_{config(0)} \end{aligned} \tag{19}$$

where $0 \leqslant i < k$, k is the total number of possible configurations, and $V_{config(k-1)}$ and $V_{config(0)}$ are the number of vertices in the maximum and minimum length configurations, respectively. As this is a cumulative summation, the solution will have the following form [75]:

$$V_{total} = \frac{k(k+1)}{2} \tag{20}$$

By contrast, when using embedded cycle graphs, such as the sub-optimal algorithm in Section 3.3.2, vertices are shared between many configurations. The total number of vertices when using the algorithm described in Section 3.3.2, $V_{total(Embedded)}$ is defined as follows:

$$V_{total(Embedded)} = V_{config(k-1)} + \left(\left\lfloor \frac{1}{2} (num_{config(total)}) \right\rfloor - 1 \right) \tag{21}$$

where $V_{config(k-1)}$ is, as above, equal to the number of vertices present in the maximum length controller configuration, and $num_{config(total)}$ is the total number of configurations in the system. Fig. 3.21 quantifies the area cost in transistors of both methods as the number of available configurations in the controller increases 1 to 20.

One conclusion that can be drawn from Fig. 3.21 is that while building each configurable mode separately might be the best choice if the number of configurable

modes is small (i.e., less than 3) the benefits of sharing hardware (vertices) become apparent in short order. Because of the limit equation in (20), we can also say that the complexity of the *Basic* control device is $O(n^2)$ while it is O(n) for an *Embedded* control device.

4 PARALLELISM IN A RECONFIGURABLE CONTROLLER FOR WAGGING SYNCHRONIZATION

4.1 INTRODUCTION (ARBITRATION & RECONFIGURATION)

With the aforementioned discussion on token ring topologies in the previous chapter, we can now pursue a discourse regarding the protocols which govern the reconfiguration process as well as the design and simulation of circuits that enforce them.

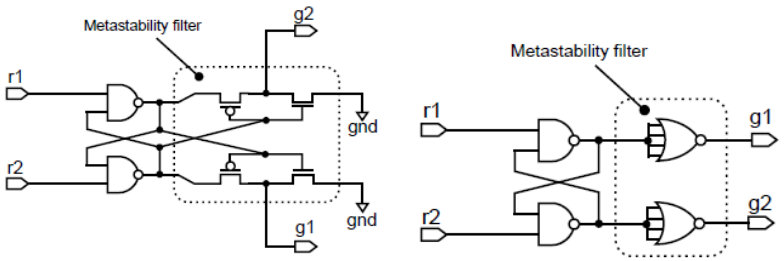
This chapter will begin with a brief overview of a class of circuits known as *arbiters* in Section 4.2, which serve as a useful foundation from which to render our examination ofthe main body of work contained herein. Thereafter, Section 4.3 will examine the mathematical foundations of the reconfiguration protocol in detail. Next, the control concept of *UCOM threading* will be introduced and examined in Section 4.4. Section 4.5 will examine the design and synthesis of a basic reconfigurable control design which incorporates the concepts in the aforementioned sections of the chapter. Finally, Section 4.6 will examine methods to ensure that the output data of the reconfiguable controller to the token ring is consistent in the presence of multiple data inputs from the attached interrupt devices.

4.2 RELATED WORK (OVERVIEW OF ARBITRATION)

Before we can touch on the design of a subsystem which controls the distributed token rings discussed in the previous chapter, we must first introduce the concept of *arbitration*. Arbitration refers to the process by which resources are shared in an asynchronous system via the use of *requests* and *grants* issued to and from their clients. This is done via handshaking protocols such as those discussed in Section 2.1.2. In short, a client issues a request to the arbiter, which is then granted sometime later. After the operation is completed the resource is released, and an acknowledgement signal is sent back to the client.

Modeling these types of systems requires us to employ the some of the concepts that were introduced back in Chapter 2. Specifically, it requires the use of STGs and PNs. Petri nets can be used to model aspects of the design abstraction related to functionality and behavior, such as concurrent events, non-deterministic choices, and unbounded component delays. Signal Transition Graphs are useful in translating these functional models into a set of signal transitions which allow the internal logic equations to be derived via synthesis tools like Petrify, and Workcraft [17], [65].

However, before the topic of arbitration can be properly addressed, we must first introduce one of the key circuit designs typically utilized in the creation of asynchronous arbiters. The component to which we allude is referred to in prior literature as a *mutual exclusion element*, or MUTEX. And while a full review of arbiters would be cumbersome and unwieldy, it is instructive to analyze three



(a) MUTEX element. From [51].

(b) MUTEX element using standard FPGA gates. From [44].

Figure 4.1: The mutual exclusion element

specific methods of arbitration due to their relationship with the material in the rest of the chapter. Namely, arbitration based on *busy and lazy token rings* and *pausible clocks*, respectively.

4.2.1 *The Mutual Exclusion Element (MUTEX)*

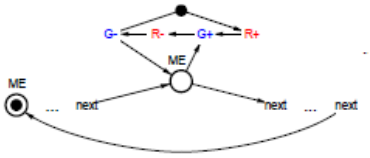
A MUTEX, as shown in Fig. 4.1(a), is a circuit which satifies the client-resource relationship stated above. In a MUTEX, if access is granted to a single request, then all other requests are disabled until the operation is completed and the resource is released.

This component has gone through several iterations over the years. An early attempt by Plummer was made to solve the issue of mutual exclusion in asynchronous requests [64]. However, that design could still fall into metastability if the requests occurred too close together, which is a possibility if the distribution of request arrival times is assumed to be uniform [45], [13]. Shortly thereafter Patil proposed a MUTEX based on threshold filters [61]. Seitz also proposed his own variation in 1980 based on an NMOS analog diffence circuit, but was later contested by several others including A.J. Martin who proposed his own version based on CMOS in 1985 [71], [50].

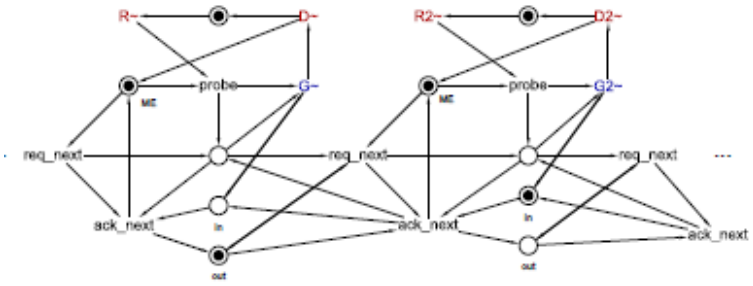
The circuit representation of a MUTEX generally consists of an S-R latch tied to a *metastability filter*, composed of low threshold inverters, which prevents changes on the output from appearing until the voltage differential between the cross coupled nodes is sufficiently different. Fig. 4.1(b) represents a variation of this design which is useful in FPGA environments where low-threshold components might not be available [44]. The only difference is that the circuit in Fig. 4.1(b) exhibits a higher latency.

4.2.2 *Token Ring Arbiters*

In the case of a busy ring arbiter, the term "busy" is used to denote the tendency of a *token ring arbiter*, composed of cells similar to those Fig. 4.2(a) to move the control token to its nearest neighbor as long as there is no associated client request. If the client request "wins" the arbitration, then the token is halted as soon as it arrives



(a) Busy Ring Arbiter. From [34].



(b) Lazy Ring Arbiter. From [34].

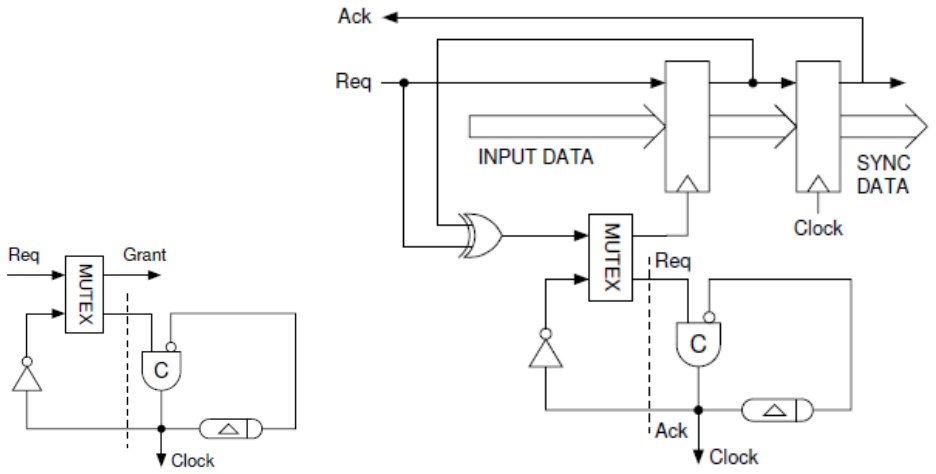
Figure 4.2: STGss of token ring arbiters

at the MUTEX input. If the client request loses the arbitration, then a round-trip penalty of one cycle is incurred until the token arrives back at the MUTEX resource again (where the client wins the arbitration automatically).

By contrast, a lazy ring arbiter composed of cells similar to those in Fig. 4.2(b) moves the control token along the cells in the ring *only when a request from the client is detected*. If such a request is issued, the request propagates through the ring until it reaches its destination (i.e. the cell holding the token), after which the token is "grabbed" and moved backwards through the ring until it reaches its source (i.e. the cell that issued the request).

4.2.3 *Arbitration via Pausible Clocks*

A *pausible clock* refers to a time reference that can be halted via the addition of MUTEX element, as shown in Fig. 4.3(a), but which will otherwise oscillate normally interrupted by a data request (Req) [55], [57]. Fig. 4.3(b) shows a receiver interface published by Mullins, based on the principle of pausible clocks [55]. On the rising edge of the data request, an attempt is made to reserve the MUTEX resource in the pausible clock portion ot the circuit. The *grant* signal of MUTEX serves as an enable signal to the first register input. When the grant signal goes high, the local clock is paused while both the *data present at the input of the register* (DATA), and the asynchronous request, are passed to the register output. The request signal at this point is fed back to the MUTEX via an *exclusive or* (XOR) gate that deasserts the request input to the MUTEX, releasing the resource and causing the local clock to resume its operation. The local clock then serves as an enable to the second register, which synchronizes the data to the local clock region and transmits it at the output of the second register. At this point the synchronized data is passed



(a) Pausible Clock Circuit. From [57].

(b) Data-driven Rx interface. From [55].

Figure 4.3: Pausible clock arbiter overview

to the output of the second register along with the *data acknowledge signal* (Ack), which is just a tap of the data request signal at that juncture.

4.2.4 *Relation to a Reconfigurable Control Device for a Wagging Synchronizer*

With the above discourse on arbiters in hand, we can now explore the association between the arbitration methods mentioned above, the material covered thus far in prior chapters, and the material covered in the rest of this chapter. In the previous chapter rules were established outlining the connectivity of an embedded token ring suitable for use in the control hardware of a parallel synchronizer based on wagging. However, that connectivity alone is insufficient to produce a meaningful control device. Because there are several ways that the embedded token ring might be implemented, including the ones presented in Section 3.3 in which the connectivities of certain embedded token rings are implied to contain a increasing number of mathematically disjoint configurations as the number of nodes and configurations in the token ring increases, arbitration is necessary to link the disjoint configurations together via a series of intermediate steps, which will be discussed in depth in Section 4.3.

4.3 RECONFIGURATION PROTOCOL

We can now move onto an examination of the protocols governing the reconfiguration process. The process is contingent on the ability of the circuit to direct the flow of a control token as it passes between the graphs of two separate configurable modes, and to ensure that the operation occurs without failure. Fig. 4.4 depicts the graph of an 8-way reconfigurable controller with several cycles, as in Fig. 3.19. The vertices of the graph are represented by the *control (CTR)* signals, while the *token request (RQ)* and *token acknowledge (ACK)* signals represent the edges. The RQ sig-

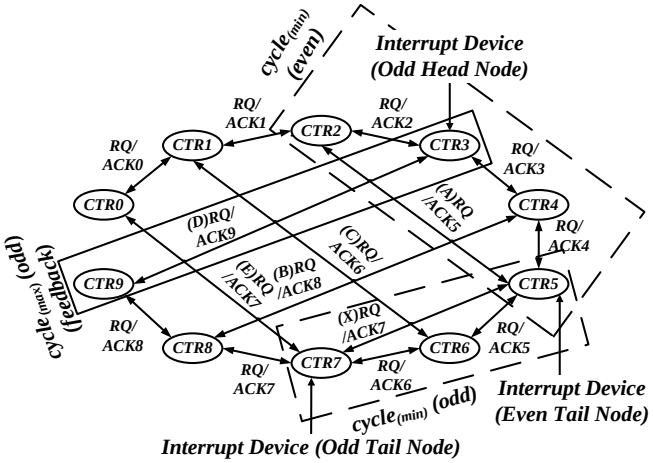


Figure 4.4: Behavior graph of an 8-way reconfigurable controller with several cycles.

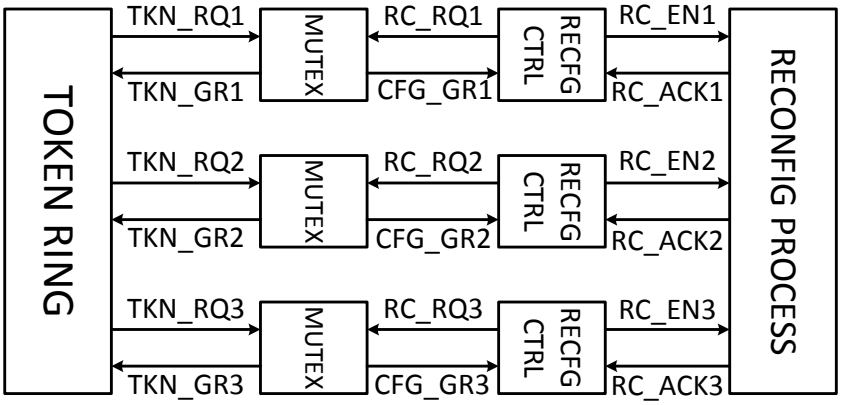


Figure 4.5: Interrupt block diagram.

nals of the token ring move clockwise around the graph, while the ACK signals move in the opposite direction. The paths denoted A, B, C, D, and E represent the valid feedback paths in the system, while X denotes an invalid path, as previously discussed in Section 3.4.5. Unfortunately, the exact position of the control token is unknown at the time of reconfiguration. If the system reconfigures when the token is located in a region of the present configuration which is not covered by the graph of the subsequent configuration, then the token will be lost.

Arbitration is therefore necessary to ensure that the token is passed between configurations without incident. This is similar to prior work used to arbitrate between configurations in a system with pausible clocks [57]. It also shares commonalities with prior work on lazy-ring arbiters [51]. However, while a lazy ring arbiter stalls the system and sends the token backward to the point of an initial request signal, the proposed design stalls the control circuit and waits for the token to traverse forward through ring of vertices until it arrives at the request point. In

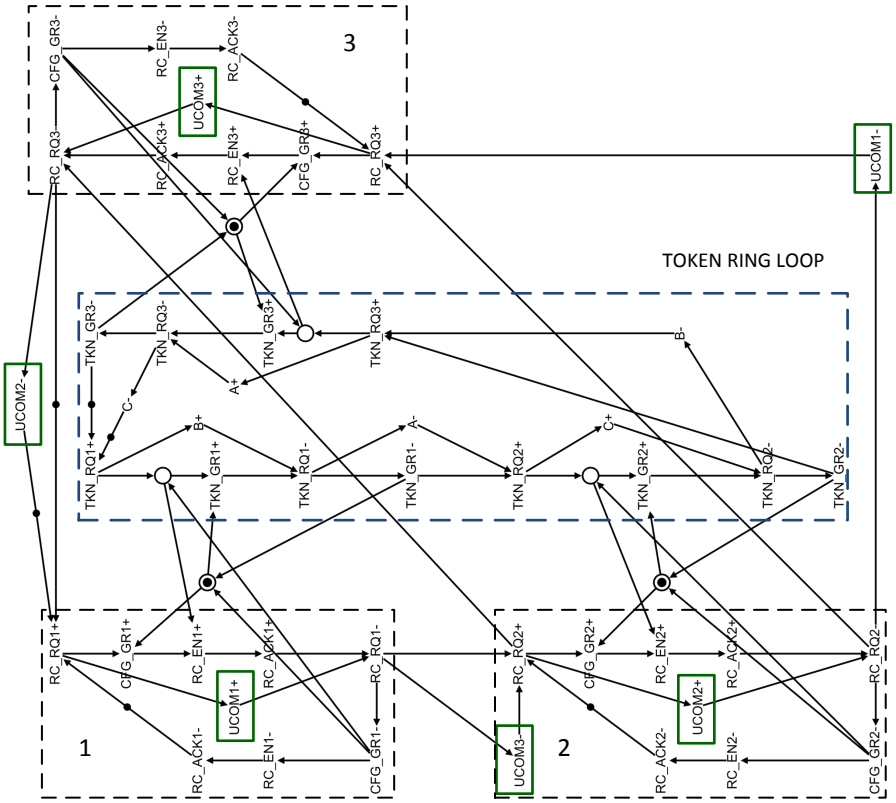


Figure 4.6: Signal transition graph (FWD PATH).

short, the control token can only flow in one direction. Stalling the control device requires the use of a MUTEX element, as shown in Fig. 4.5. A MUTEX has the property that if access is granted to a single request, then all other requests are disabled until the operation is completed and the resource is released. Therefore, if such a device is inserted into the token ring at a location which is common to both the old configuration and the new one, then the token can be stalled until the reconfiguration operation has finished and then released along the new configuration without error. The protocol itself functions as follows, denoted by the black dotted portion of Fig. 4.6:

1. The reconfiguration request arrives at RC_RQ1 (RC_RQ1 goes high), and the MUTEX resource is reserved (CFG_GR1 goes high).

2. The control token arrives at the MUTEX element along TKN_RQ1, and is halted from continuing.

3. System reconfiguration is performed. (RC_EN1 goes high)

4. An acknowledge signal is generated by the interrupt hardware.* (RC_ACK1 goes high)

5. The reconfiguration acknowledgement arrives, the MUTEX resource is released (RC_RQ1 goes low), and the token is allowed to continue along TKN_GR1. (All other signals become low thereafter).

* If necessary, intermediate reconfiguration is used to pass the token to the correct configuration of interest.

4.3.1 *Mathematical Foundations (Assumptions)*

Intermediate reconfiguration is not required if the protocol is implemented across systems which contain either an *exclusively odd* or an *exclusively even* number of vertices, as defined by the embedded behavior graph of the token ring (similar to the one shown in Fig. 3.19 and Fig. 4.4). Assume $\mathtt{cycle}_{(g)}(even)$ is a valid configuration (set of vertices) where g is even. The embedded nature of our ring construct is such that for all valid configurations with an even number of vertices the following holds:

$$\mathtt{cycle}_{(g)}(even) \subset \mathtt{cycle}_{(h)}(even) \tag{22}$$

where $g < h$. As a consequence, only one interrupt device is required in the above case, which is to be placed at the tail vertex of the minimal even ($\mathtt{cycle}_{(min)}(even)$) configuration of vertices. The argument follows similarly for the valid configurations in the system with an odd number of vertices. Conversely, intermediate reconfiguration is mandatory if any two valid configurations are mathematically disjoint from each other. If (23) holds

$$\begin{aligned} \mathtt{num}_{\mathtt{config}(odd)} &\geqslant \left|\mathtt{cycle}_{(min)}(even)\right|, \text{ or} \\ \mathtt{num}_{\mathtt{config}(even)} &\geqslant \left|\mathtt{cycle}_{(min)}(odd)\right| \end{aligned} \tag{23}$$

where $\mathtt{num}_{\mathtt{config}(odd)}$ ($\mathtt{num}_{\mathtt{config}(even)}$) is equal to the total number of valid odd (even) configurations in the behavior graph, then the intersection of the elements in the sets of $\mathtt{cycle}_{(min)}(even)$) and $\mathtt{cycle}_{(min)}(odd)$) are equal to the empty set (i.e. the two configurations are disjoint). *Thus, a series of intermediate steps are required to link these configurations together.*

In order to minimize the hardware overhead in the final control device, care must be taken when placing the interrupt devices in the system. Interrupt devices should be placed at the tail vertices of the minimal odd and even configurations, while an interrupt device must also be placed at the head vertex of either the maximal length odd or even configuration, as shown in Fig. 4.4. Mathematically, (24) must be true

$$\begin{aligned} \mathtt{cycle}_{(max)}(odd) \cap \mathtt{cycle}_{(min)}(even) &\neq \{\} \\ \mathtt{cycle}_{(max)}(even) \cap \mathtt{cycle}_{(min)}(odd) &\neq \{\} \end{aligned} \tag{24}$$

where $\mathtt{cycle}_{(max)}(odd)$ ($\mathtt{cycle}_{(max)}(even)$) is the valid configuration which contains the maximum number of odd (even) elements, and $\mathtt{cycle}_{(min)}(odd)$ ($\mathtt{cycle}_{(min)}(even)$) represents the valid configuration which contains the minimum number of even (odd) elements. Thus, the vertices within the graph of the largest odd (even) configuration and the smallest even (odd) configuration are not

mutually exclusive, as shown in Fig. 3.10(c) and Fig. 4.4. Because of (24) and the containment property in (22), the control token can now be successfully passed between any valid configuration (set of vertices) in the behavior graph through the use of only three interrupt devices. In this manner, the control token is contained when switching between even and odd configurations. This will be true for control circuits with an arbitrary number of vertices (devices). *Therefore, it is possible to create an interrupt system which covers a complete range of cases with only three interrupt devices.*

4.4 UCOM THREADING

With the above discussion in hand, we can now discuss the concept of *UCOM threading*, which is central Chapters 4 and 5 of this thesis. Recall, that in Section 2.3.3. we touched on the concept of a *lock relation*, which was a relationship between two or more single-cycle signals, where the order of signals takes the form of $(A* \rightarrow B* \rightarrow A\bar{*} \rightarrow B\bar{*} \rightarrow A*)$. Such a relationship has been used in the past to satisfy the CSC condition in asynchronous circuits. However, the locking relationship can be used for more than just resolving state encoding ambiguities in the SG of an asynchronous STG.

In this section we will explore how signals that employ the locking relation can function as *UCOM threads*, and be used to sequence the operations in an STGs that *already satisfies the CSC property*. Furthermore, we will utilize these UCOM threads as a core aspect of the control theory in the logic of the interrupt subsystem which implements the reconfiguration protocol outlined in Section 4.3 on the embedded token rings described in Chapter 3.

4.4.1 *Enforcing Firing Order in Cyclic Independent Loops*

As stated above, when a the STG representation of a circuit is composed of independent loops, a secondary loop of UCOM signals can be used to link them together and enforce the firing order of the device. Fig. 4.6 and 4.7 illustrate the STGs (both forward and reverse) of a circuit which implements the reconfiguration protocol in Section 4.2. They are both composed of one *token propagation loop* (blue dotted lines), three interrupt module loops (black dotted lines), and one *UCOM control loop*. The UCOM control loop formed from the rising and falling edges of the threads (signals) UCOM1, UCOM2, and UCOM3, respectively, which are encapsulated within the green boxes in both figures.

The rising edges of the UCOM threads are triggered by the rising edges of the request lines in each interrupt module (RC_RQ1, RC_RQ2, and RC_RQ3), while the falling edges of the threads are triggered by these same signals. However, the request lines in each module act along different UCOM threads. RC_RQ1 controls UCOM1+ and UCOM3- transitions in the forward direction, and UCOM1+ and UCOM2- in the reverse. Similarly, RC_RQ2 and RC_RQ3 control the other UCOM threads in the loop, thereby interleaving them into a secondary cyclic graph. As the STGs of each interrupt module are dependent on the states of the UCOM threads

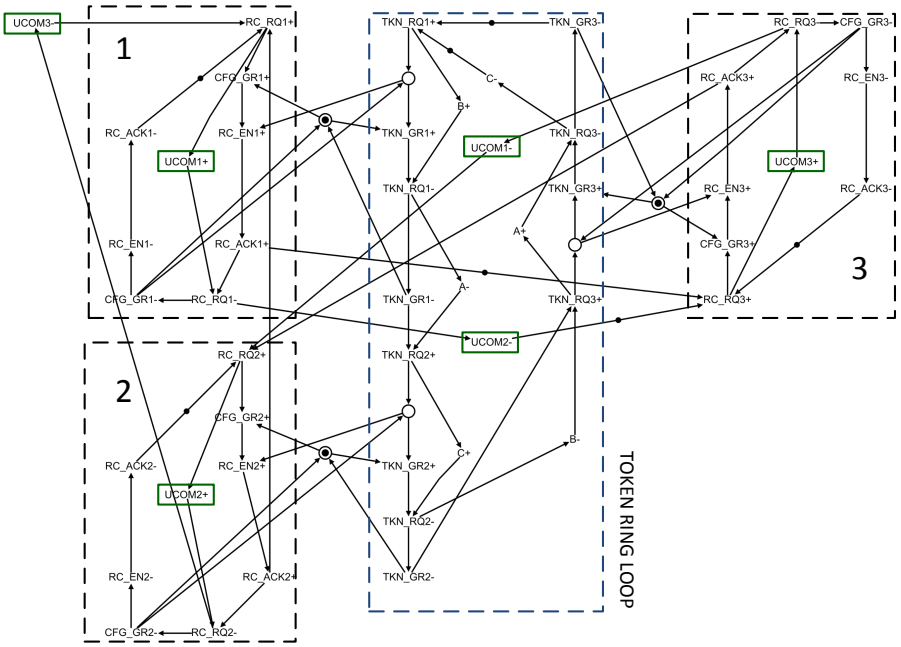


Figure 4.7: Signal transition graph (REV PATH).

in the control loop, we can say that the firing order of each module is both defined and enforced by these threads.

4.4.2 *Control of End Behavior in Cyclic Independent Loops*

While the information in the prior subsection was sufficient to illustrate that the control loop of UCOM threads is sufficient to enforce the firing order of the signal transitions in the STG representations of the interrupt subsystem, it should be noted that the control loop is itself *also a cyclic graph*. As such, a termination condition can only be invoked by manipulating the UCOM signals in the control loop. This is accomplished by logically masking the UCOM signals generated by the final interrupt module from the other modules in the loop, thereby making the other interrupt modules in the system "think" that the final module has not completed its operation when it finally does finish. This renders the UCOM control loop inactive until the system decides to reactivate the control loop by removing the logical mask placed on the signals.

4.5 BASIC CONTROLLER DESIGN

Continuing forward from the prior discussion, the underlying implementation of the interrupt subsystem in this work can now be explored. As stated previously, the reconfiguration system acts as a set of individual modules linked together via UCOM threads. Each interrupt module uses a MUTEX to stall the token ring while reconfiguration is performed, in accordance with the protocol of Section 4.3. For each module added to the system, a delay equal to the propagation time across a single MUTEX element is added to the critical path of the embedded token ring.

When an external reconfiguration request arrives at the controller bundled with reconfiguration data, the interrupt subsystem issues a resource request to the relevant MUTEX element, and waits for a grant. The subsystem also waits for the head of control token to arrive at the other MUTEX input (indicating that the embedded token ring has been halted). Once both conditions are met, the reconfiguration process begins. Old reconfiguration data is then flushed from the interrupt hardware and updated with new data, which is then fed to the Multiplexer (MUX) inputs of the embedded token ring. When the new data has finished settling, an acknowledgement signal is generated, indicating and the subsystem is free to proceed to the next stage of the reconfiguration process (if any exist). The cost of the subsystem is proportional to the number of MUX elements present in the system, which also depends on the total number of configurations in the system.

4.5.1 *Circuit Synthesis*

The following example documents the operation of three 16-way reconfigurable interrupt devices linked together via UCOM threading as discussed previously, albeit with some notable differences. MUX elements have been inserted into the UCOM threads, which make it possible to *reverse the directionality of the reconfiguration* with a single signal line, *FLAG*, as shown in Fig. 4.8. When the FLAG signal is high, the token traverses the interrupts in order from module 1 to 3 (FWD direction), and vice versa when FLAG is low (REV direction). If the *FLAG* and *MEM* signals have *initial logic values which are different*, then the token *traverses all three interrupt devices* (multi-mode operation), whereas only one interrupt module is activated if the initial values are the same (single-mode operation). Perhaps one of the most interesting points in Fig. 4.8 deals with the activation and termination of the interrupt subsystem. The STG of Fig. 4.6 is *cyclic* (i.e. the first and final states are identical), which means that forcing the interrupt modules to halt their operation and terminate is problematic. In order to force the system to stop operating, a specific UCOM line must be de-asserted (*UCOM3* in the FWD mode, and *UCOM1* in REV mode).

In Fig. 4.8, *RC_ACK2* acts as an enable to the *FLAG MEMORY* block, and copies the current value of the FLAG into MEM *if the initial values the two signals were different*. Once the current value of FLAG has been copied, an XOR gate is triggered which causes either the *UCOM3* (FWD) or *UCOM1* (REV) signal observed at interrupt module 2 to be forced low, shutting down the system. The only signals which persist after the reconfiguration process is complete are the RDATA outputs and the FLAG signal. It should be noted that the FLAG signal (whether '0' or '1') *must persist after the reconfiguration operation is complete in order to guarantee that the system remains off*.

Fig. 4.9 depicts the circuit of an individual interrupt module. In the following example, it is responsible for the signals *UCOM1*, *RC_RQ1*, and the reconfiguration data, RDATA, associated with the leftmost interrupt module in Fig. 4.8. The UDATA arrow represents the unencoded 16-bit reconfiguration input data, while the *UCOM_OUT* (UCOM1) signal depicts the UCOM thread which is generated inside the module by the *UCOM GENERATE* block. It is a function of itself,

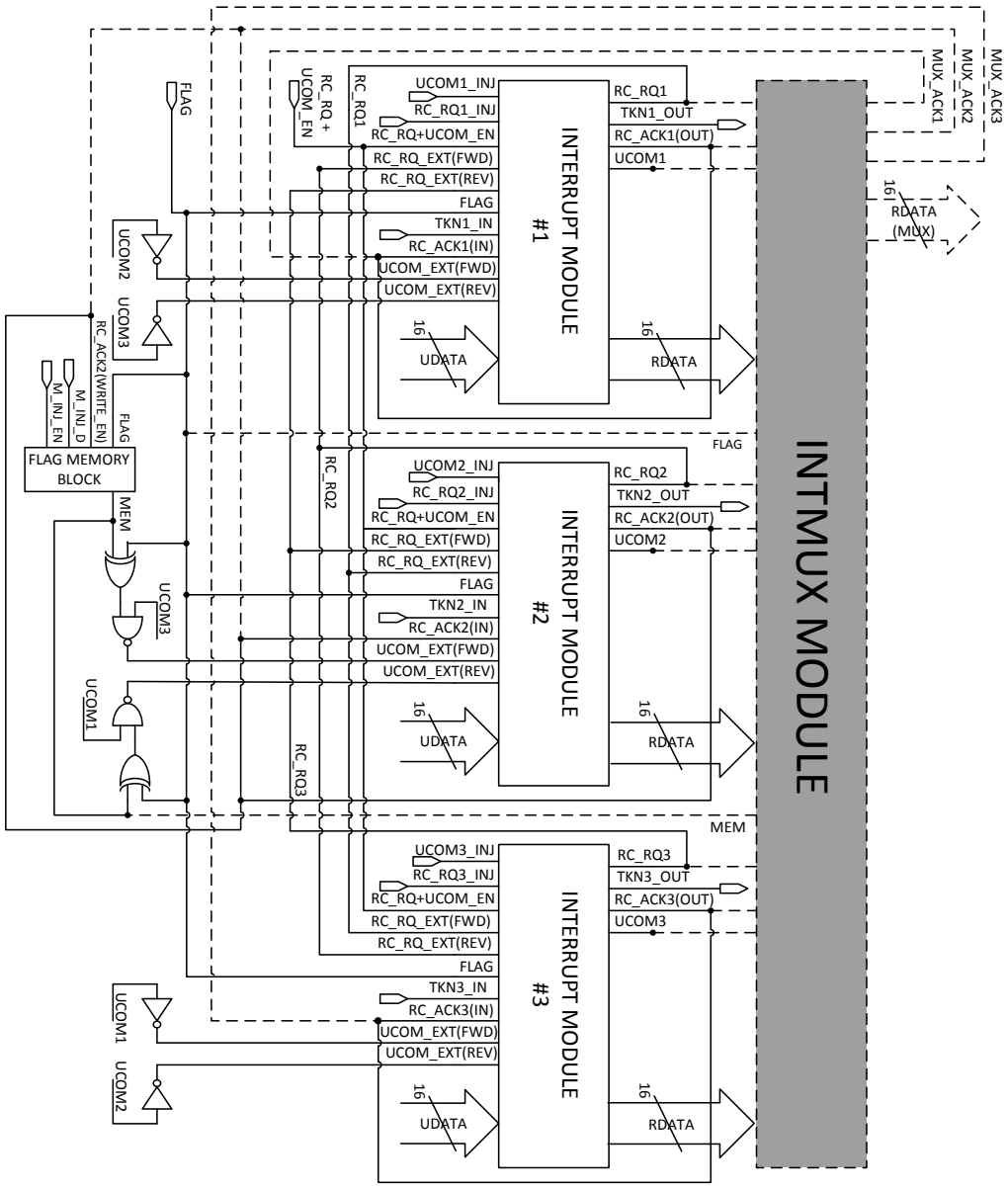


Figure 4.8: Top view of three 16-bit interrupt modules tied together via UCOM threads. Solid lines represent the connectivity of the signal lines present in this section. Grayed out portions and dotted lines represent the connections and blocks which will be discussed in Section 4.6.

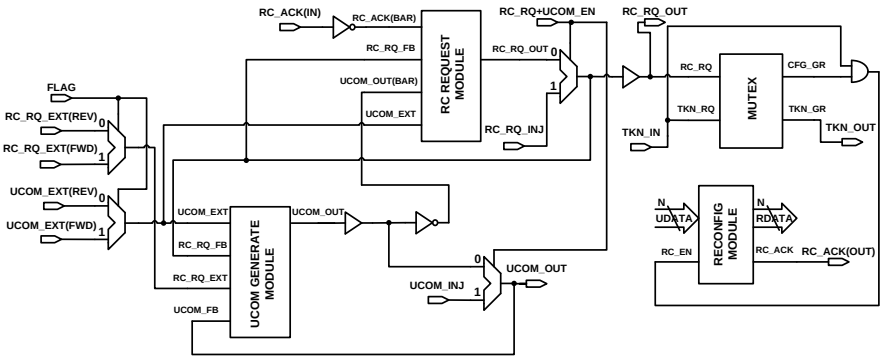


Figure 4.9: Internal view of an individual 16-bit interrupt module (i.e. module #1 in the example).

UCOM_FB, the request signal specific to that module, *RC_RQ_FB* (RC_RQ1), the external UCOM thread, *UCOM_EXT*, and the external request line, *RC_RQ_EXT*, as defined by Fig. 4.8 (which are selected via the FLAG signal). The output of the UCOM generation module *for the signal UCOM1* is defined in (25).

$$\begin{aligned} \text{UCOM1} &= (\text{UCOM1} * \overline{\text{UCOM2}}) + \text{RC\_RQ1} + \text{RC\_RQ2}\ (\text{FWD}) \\ \text{UCOM1} &= (\text{UCOM1} * \overline{\text{UCOM3}}) + \text{RC\_RQ1} + \text{RC\_RQ3}\ (\text{REV}) \end{aligned} \tag{25}$$

The request signal specific to the module *RC_RQ_OUT* (RC_RQ1) is generated from the *RC REQUEST* block. The block uses the acknowledgement signal generated from the output of the interrupt module (RC_ACK1), the internally generated UCOM signal UCOM_OUT (UCOM1), the externally generated UCOM signal used above (UCOM_EXT), and its own feedback signal (RC_RQ_FB). The output of the request module *for the signal RC_RQ1* is characterized in (26).

$$\begin{aligned} \text{RC\_RQ1} &= \overline{\text{RC\_ACK1}}(\overline{\text{UCOM1}} * \overline{\text{UCOM2}} + \text{RC\_RQ1}) \\ &+ (\text{RC\_RQ1} * \overline{\text{UCOM1}})(\text{FWD}) \\ \text{RC\_RQ1} &= \overline{\text{RC\_ACK1}}(\overline{\text{UCOM1}} * \overline{\text{UCOM3}} + \text{RC\_RQ1}) \\ &+ (\text{RC\_RQ1} * \overline{\text{UCOM1}})(\text{REV}) \end{aligned} \tag{26}$$

The rest of the system follows the reconfiguration protocol defined in Section 4.3. The request and grant signals *RC_RQ* and *CFG_GR* represent the path of the control token taken during reconfiguration, while the signals *TKN_RQ* and *TKN_GR* define the path taken during normal system operation. The signal *RCMOD16_EN* (RC_EN1) represents the enable signal used to update the 16-bit reconfiguration data, RDATA, of the module while *RC_ACK* (RC_ACK1) represents acknowledgement signal used to indicate when the update operation has completed. Interrupt modules 2 and 3 are constructed similarly.

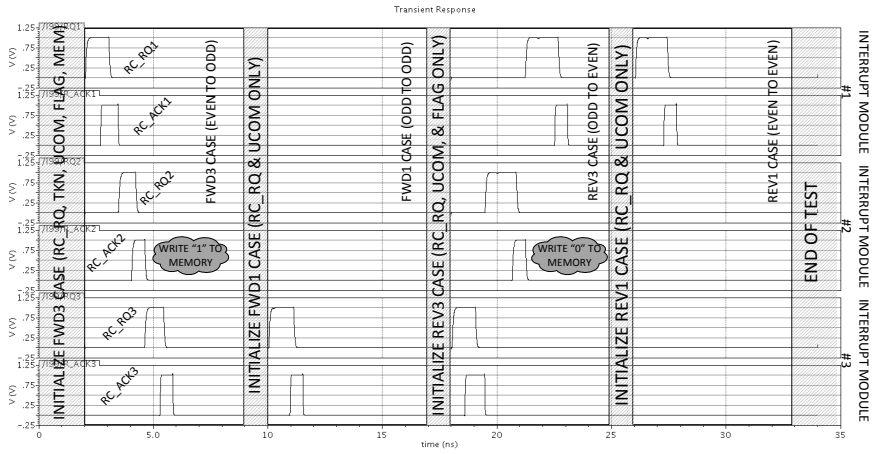


Figure 4.10: Transient response of the control signals for the subsystem of three interrupt devices, which demonstrates the firing behavior of the design. Reconfiguration data, UCOM, FLAG, and MEM signals have been omitted.

Table 4.1: Duration of the Active Region of the Interrupt Subsystem for Different Configurable Modes at a TN-TP Corner with V_{DD} = 1.0 V

Temp. (°C)	FWD3 (ns)	FWD1 (ns)	REV3 (ns)	REV1 (ns)
0	3.669	1.391	4.822	1.686
27	3.817	1.479	5.009	1.775
100	4.068	1.657	5.473	1.982

4.5.2 *Performance Comparison*

Fig. 4.10 shows the transient response of the control signals in the circuit when the three interrupt modules are connected together in a ring of 15 inverting elements, via their *TKN_IN/TKN_OUT* ports, with each interrupt module being placed a uniform distance of 5 inverting elements apart from each other. It illustrates how the request and acknowledgement signals in the interrupt devices can be controlled to *both fire in a specific order and then terminate their operations*, as discussed previously. The duration of the simulation is 33 ns, testing 4 cases. The initialization time is 2 ns for the first configuration and 1 ns for each configuration thereafter. Cases *FWD3* (FLAG = '1', UCOM = '110', MEM = '0') and *REV3* (FLAG = '0', UCOM = '011') simulate the operation of the system in the forward and reverse directions *where intermediate reconfiguration is necessary*. Similarly, cases *FWD1* (FLAG = '1', UCOM = '011') and *REV1* (FLAG = '0', UCOM = '110') simulate the operation of the system in the forward and reverse directions where it is not. The duration of each reconfiguration operation is characterized in Table 4.1 while the energy consumed is depicted in Table 4.2.

Table 4.2: Average Power Consumption of the Active Region of the Interrupt Subsystem for Different Configurable Modes at a TN-TP Corner with V_{DD} = 1.0 V

Temp. (°C)	FWD3 (μW)	FWD1 (μW)	REV3 (μW)	REV1 (μW)
0	291.93	267.64	232.19	223.72
27	290.39	260.12	227.88	214.10
100	281.04	245.86	226.40	211.82

However, it should be noted that since the interrupts are connected via a ring of inverting elements, intermediate reconfiguration is not necessary. The ring of inverters only serves to tie the interrupt modules together. The placement of the devices is not the same as the protocol in Section 4.3 either. Fortunately, the placement of the devices does not affect the functionality of the control device, as that is defined only by the STGs of Figs. 4.6 and 4.7. While placement of the interrupt modules might have consequences on the duration of each reconfiguration operation as shown in Table 4.1, it has no impact on the power consumption of the interrupt devices for reasons which will be discussed shortly.

As shown in Table 4.1, the duration of the reconfiguration operation varies based on the number of interrupt devices required to complete the reconfiguration process as well as whether the directionality of the control token opposes the flow of the UCOM threading in the system. Table 4.1 assumes that the forward (i.e. FWD1, FWD3) direction corresponds to when the flow of both the UCOM threads and the control token *operate in the same direction*, and vice versa in the reverse (i.e. REV1, REV3) direction. Consequently, the control token always "hits" in the forward direction (the token arrives after the request has been issued to the next interrupt device), while in the opposite case it always "misses" (it arrives before), resulting in higher latencies in the reverse direction.

Table 4.2 characterizes the average power consumption of the system during the active portions of the reconfiguration process across varying temperature parameters. The system consumes an average power which varies between 212 and 292 µW with a maximum variance of 72.5% when tested using a corner analysis. The worst case corner was the fast-fast case at a temperature of 100°C. Durations similar to those supplied in Table 4.1 were used in order to adjust the intervals over which the average power consumption was measured in order to ensure the fairness of the testing procedure. However, these results are confounded by two factors.

First, the power consumption of the 15 inverting elements which form the token ring used for the test bench need to be considered. In order to ascertain the power of the interrupt subsystem alone, the values in Table 4.2 must be adjusted. This is done by taking the average power consumption values in Table 3.5, and multiplying values by the number of cells in a given configuration, and then subtracting this estimate from the appropriate values in Table 4.2. In this case, we must subtract the power consumption of a 15 cell ring oscillator from the results in Table

4.2. After adjustment, the average power consumption of the interrupt subsystem alone during active operation is found to range from 194 to 270 µW with a variance of 48.6% across process corners at a nominal temperature of 27°C.

Second, the interrupt subsystem used in the test bench of Fig. 10 requires the interaction of three separate interrupt devices. Therefore, the power consumption of the subsystem is spread across every interrupt device, even in single mode operation. Therefore the adjusted values listed above need to be divided by three in order to find the power consumption for each individual interrupt module. Using the above method, the average power consumption of each interrupt module during active operation is found to range from 64.7 to 90.0 µW across process corners at a nominal temperature of 27°C. The variance remains unchanged from above.

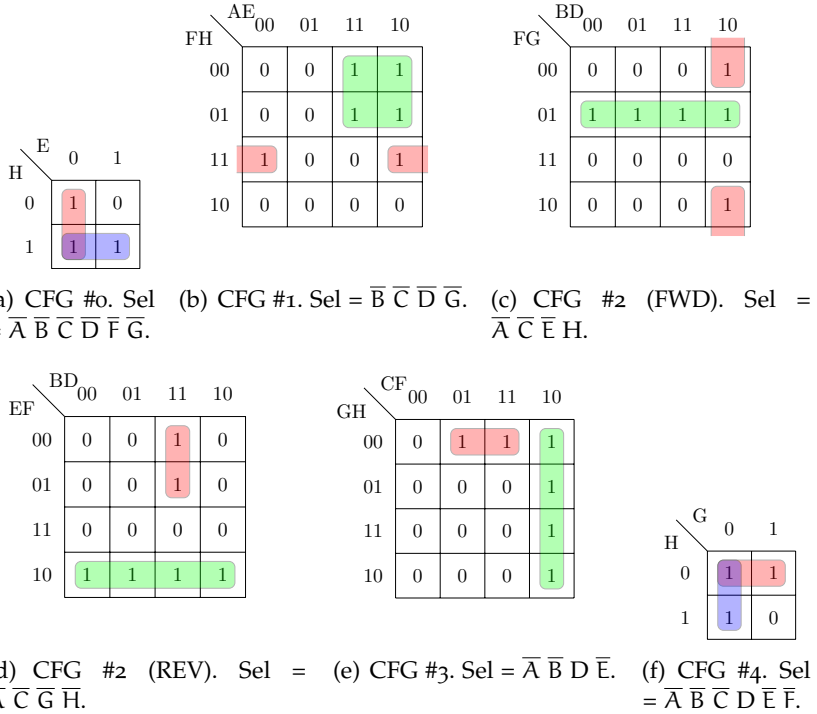
When the system is inactive (i.e. not receiving requests) the interrupt devices only consume an average power equal to the cumulative summation of the standby leakage currents across the individual transistors in the module multiplied by the supply voltage applied to them. Thus, the power consumption of the interrupt subsystem will vary in proportion to the frequency of reconfiguration requests received. As stated previously, it should be noted that the token ring control and interrupt devices were tested separately. Thus, interactions between token rings of varying length and the power consumptions of the interrupt modules have yet to be documented.

4.6 OUTPUT DATA MERGING IN RECONFIGURABLE CONTROLLER

As the interrupt modules are tied together via UCOM threads and each module generates its own output signals based on the reconfiguration data provided by its bundled data lines, it is necessary to both select the appropriate output signals during the reconfiguration process and also ensure that the final configuration persists once the interrupt device has powered down. To accomplish this a different multiplexer element, hereafter referred to as *INTMUX*, has to be constructed which not only provides a single output signal to the MUX elements in the token ring but also produces separate acknowledgement signals specific to each interrupt module (in the event of multiple configuration steps).

4.6.1 *Circuit Synthesis & Results Analysis*

The INTMUX must be programmed with the control signal combinations which correspond to each respective configuration (CFG) so that acknowledgement signals are generated which enforce the behavior specified in Fig. 4.6 and Fig. 4.7. Furthermore, the INTMUX must also contain redundant configurations that store the last known system state and persist once the system operation has ceased. Thus, the INTMUX module also requires memory which allows the device to "remember" the interrupt module which contains the last known output signals. In this way the behavior outlined by Fig. 4.6 and Fig. 4.7 is maintained, and the system does not lose its output signals when the interrupt module completes its operation.



(a) CFG #0. Sel = $\overline{A}\ \overline{B}\ \overline{C}\ \overline{D}\ \overline{F}\ \overline{G}$. (b) CFG #1. Sel = $\overline{B}\ \overline{C}\ \overline{D}\ \overline{G}$. (c) CFG #2 (FWD). Sel = $\overline{A}\ \overline{C}\ \overline{E}\ H$.

(d) CFG #2 (REV). Sel = $\overline{A}\ \overline{C}\ \overline{G}\ \overline{H}$. (e) CFG #3. Sel = $\overline{A}\ \overline{B}\ D\ \overline{E}$. (f) CFG #4. Sel = $\overline{A}\ \overline{B}\ \overline{C}\ D\ \overline{E}\ \overline{F}$.

Figure 4.11: INTMUX boolean logic minimization

The complete definition of the transient input signal combinations for which the INTMUX module is sensitized is elaborated in Appendix A, which also lists the associated Boolean logic minimization used in the circuit synthesis. Once the INTMUX module had been programmed with the values listed therein it was tested using a control sequence generated from an external waveform editing program written by both myself and Ghaith Tarawneh, also elaborated in Appendix A. He handled the file open commands, while I got it to generate the timing information for the SPECTRE waveforms. Manual synthesis was used due to the possibility of hazards being introduced into the circuit via automated synthesis tools. In a synchronous system, glitches do not affect the correct operation of the circuit, but as this was an asynchronous design glitch-free operation is a requirement [18]. The signal sets for which the outputs of MUX_ACK1, MUX_ACK2, and MUX_ACK3 are asserted or de-asserted in Fig. 4.8, directly correspond to the locations where RC_ACK1, RC_ACK2, and RC_ACK3 are asserted or de-asserted in the transient simulation of Fig. 4.10, respectively.

However, in order to fully validate the INTMUX module an exhaustive search of all the possible signal combinations was also performed. The number of anticipated sensitized control outputs in the partial sequence was then compared to the number of sensitized outputs in the exhaustive search, to see if they matched. While the outputs did match the expectation of the truth table in Fig. 4.11 when the module was tested independently, the module failed when it was connected to the entire system. An attempt was then made to remedy this by extending the INTMUX module by incorporating the UCOM control signals into the truth table of the INTMUX as well. Unfortunately, the INTMUX remained inoperative.

5 ROBUSTNESS IN A RECONFIGURABLE CONTROLLER FOR WAGGING SYNCHRONIZATION

5.1 INTRODUCTION (PRINCIPLE OF EXCLUSION)

While the material in the Chapter 4 illustrated how an interrupt subsystem based on STGs could be utilized to control an embedded token ring, such as those defined in Chapter 3, a major limiting factor in the design is the sensitivity of the controller to permanent failures. In this chapter, we will explore how the robustness of the solutions presented in prior chapters of this work can be improved by incorporating the ability to exclude failed nodes and vertices from the connectivity graph of the token ring control device.

Section 5.2 will define the fault models used in this chapter and briefly explore the concept of exclusion and related work in circuit theory.

Section 5.3 will expand on the concept of UCOM threading presented in Section 4.4 to account for the existence of multiple paths in the STG when exclusion is utilized. Section 5.4 will apply the above the principles discussed to the token ring controller presented in Chapter 4.

5.2 FAULT MODEL DEFINITIONS

Before we can properly address how the principle of exclusion can be used to ameliorate faults and failures in the controller presented in Chapter 4, we must first *define* what is meant by faults and failures. For the purposes of this review, we will restrict the discussion to those faults which are most commonly encountered.

5.2.1 *Fault types*

There are generally three classes of faults commonly seen in digtial circuits, regardless of whether or not they are constructed using synchronous or asynchronous logic.

1. Transient faults
2. Permanent faults
3. Intermittent faults

Transient faults occur when radioactive particles strike the inputs of a gate with sufficient energy to momentarily change the state of a line from logic '0' to logic '1', or when there is a momentary drop in the supply voltage the gate of a causes the signal inputs/outputs temporarily to dip below a certain logical threshold (which sometimes referred to as a "brown out"), or when signal lines are placed close enough together that they erroneously communicate (i.e. *crosstalk*) via capacitive coupling during signal transitions [72].

Permanent faults occur as a consequence of either defects in the circuit manufacturing process, or as a consequence of aging within the component. In practice the latter is more important, as commercial chips are heated at the time of manufacturing to prevent *infant mortality* (i.e. when a fabricated chip fails after a short period of time into the product life cycle due to processing defects). *Electromigration* is the most common cause of transistor failures related to age, and it occurs when prolonged application of direct currents to the signal lines in a circuit results in the diffusion of atomic material (more specifically the interconnect metal) via momentum transfer from the conducting electrons [39]. Historically, *hot-carrier injection* into the gate-oxide of CMOS transistors was also an issue, where "sufficiently energetic" electrons or holes traveling between the source and drain regions of the transistor could embed themselves into the gate-oxide thereby causing permanent damage by altering the characteristics of the threshold voltage in the transistor [27], [26]. Nowadays, gate-oxides are so thin that quantum tunneling is of greater concern, necessitating the use of high-k dielectrics in the gate [68].

Intermittent faults, as with permanent ones, can be caused by malformed CMOS transistor devices or interconnect lines, and may even evolve into permanent faults given time. However, as the name suggests, the errors an induced by an intermittent fault share the properties of unpredictability, as with transient faults, while remaining systematic in nature, as with permanent faults. Typically such faults manifest themselves in error bursts which are localized at a specific point. *IDDQ tests* are typically employed to find such faults, which are conducted by powering down a circuit after it has remained on for a "sufficiently long time" and then analyzing the circuit for nodes which remain highly capacitive [48], [67].

5.3 RELATED WORK (JOHNSON COUNTER)

To set this work in the proper context, we must first compare it to other architectures that possess similar functionality. The most notable example of a cyclic counter that possesses the capacity for error recovery is that of a *twisted ring counter*, which is also sometimes referred to as a Johnson counter or a Moebius counter. Rather than counting in a one-hot manner, as does a normal ring counter, it cycles between $2n$ different states where n is the number of bits in the counter. A standard ring counter only possesses n different states, therefore a Johnson counter only uses half the number of flops to implement the same number of states, though the states require decoding in order to map them to their associated state number [46].

However, the source of the error recovery capabilities within the counter is derived from the fact that the counter changes its states in a circular manner, and only one bit changes at a time. Because of this, it is possible to use a binary reflected code, more commonly referred to as a Gray code, to implement forward error correction to recover from single bit errors in the system [35]. This is commonly used to recover from transient errors induced within the counter. While this functionality is no doubt useful, in the case of a permanent fault the error correcting hardware will be activated on every iteration of the system, simply to ensure normal operation. And if a transient error then occurs at that point, the

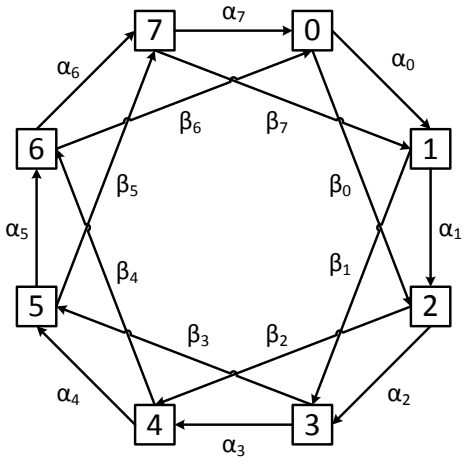


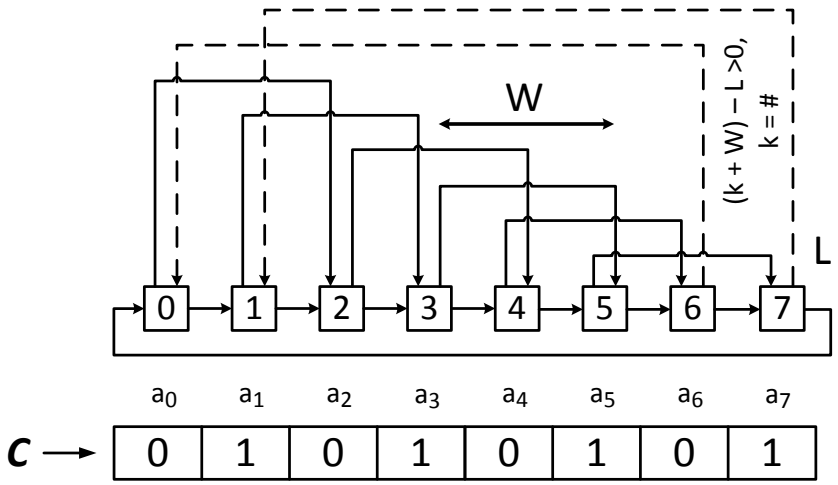
Figure 5.1: Eight-way ring network based on chordal bypass paths

system will fail. As subsequent sections will show, by excluding faulty nodes from the system it is possible to recover from such a malady at the cost of permanently reducing the number of states in the system.

5.4 CHORDAL RING NETWORKS (ALGORITHM REINTERPRETATION)

In Chapter 3, we dealt with how an embedded cycle graph based on token rings controlled its adjacency and parallelism via *distributed feedback paths*. However, that method is not without some limitations. Most prominent among those limitations is that the system is still very susceptible to permanent faults. This particular issue also not unique to the design. Johnson counters also get permanently locked into a 'subset of their entire state space if a hard fault occurs on one of the signal lines, assuming no forward error correction is employed, but even if it is a permanent failure will remove the capability of the Johnson counter to recover from transient SEUs. In that case, it is because a Johnson counter is designed to recover from transient faults and is incapable of ascertaining whether the prior fault was caused by a transient α-particle strike, or whether it is because electromigration has eroded the line, or whether negative biasing has just made the PMOS transistors incapable of switching. Therefore it is desirable to build a token ring controllers that can adapt to the non-idealities present in modern technology nodes. With that in mind, this subsection will introduce a method of robust design known as a *chordal bypass path*. The name is derived from a 1981 paper by Arden where he explored how to connect LAN microcomputers together using a ring of degree 3 [5]. In his original paper α-moves and β-moves were defined as routing movements in the clockwise and counter-clockwise directions, respectively. Since a token ring can be thought of as a diagraph (directed graph), these definitions aren't useful in this context.

For the purposes of this work we will define an α-move as the *absence* of a chordal jump when the token traverses through a vertex in the embedded cycle



(a) Annotated diagram outlining the methodology for creating abstract chordal bypass paths

Figure 5.2: Chordal Bypass Path Algorithm Overview

graph, and a β-move as the *presence* of a jump when traversing through a vertex, as in Fig. 5.1. The number of spaces that the signal "jumps over" when traveling through the bypass path is referred to as the *width* of the chord. In Fig. 5.1 the width, W, is equal to 2. It should be noted that W-1 represents the maximum number of subsequent faulty nodes that a system can handle before failure. In other words, Fig. 5.1 is tolerant of single bit faults, but if the system suffers from 2 permanent faults in a row at any point, then the faults can no longer be bypassed and the circuit fails There is also a minimum limit on the number of nodes which must be present in the graph of the system in order for it to function. This limit is equal to the cardinality of the system in the absence on any β paths (i.e. only α paths are present), divided by the width of the bypass chords plus the modulo result (i.e. the remainder) resulting from the division. Assume L = $|cycle_{(max)}even|$, where $cycle_{(max)}even$ is the valid configuration which contains the maximum number of even elements. In the case of Figure 5.1, L=8 so 8/2 = 4. In terms of the underlying graph if the number of nodes in the system is fewer than the results of the calculation outlined above, it isn't possible to traverse the system. The system *requires* that both the source and destination nodes in the system must presently exist in the graph for it to function properly.

5.4.1 *Chordal Bypass Path Algorithm*

Similar to the above subsection if one begins with the basic figure for a linked list, as in 3.12(a), it can be modified to incorporate the characteristics of an embedded cycle graph based on chordal bypass paths. In order to extend the functionality of this list, a few pieces of information are required. Additionally each object in the array must have the following properties.

1. int index: Integer data type. Used to order the entries in the linked list.
2. *pointer α_tail: Pointer data type. Contains the *initial* path to the next object in the list.
3. *pointer β_tail: Pointer data type. Contains the *secondary* path to the next object in the list.
4. $C[a_k]$: Array variable used to store the state of the bypass paths in the system, which must be large enough to contain a unique position for each entry in the list (i.e. $|C[a_k]| = L$).

As with the distributed feedback algorithm of Section 3.3.3, the creation of this graph is process is simpler to understand if it is performed via iteration. On the first pass the objects in in the list from 0 to L will be created and linked along the α *tails*. When the final entry is created, the α_tail of this entry will be given a pointer to the head of the list, as in Figure 3.12(a) (effectively creating the α paths after one full iteration). On the second pass, the β paths can be constructed by utilizing *a priori* knowledge of the width, W, of the β chords. The β paths are created by linking the β_tail of the current object (index value = k) to its target object (index value = $k + W$). The only complication in this process is ascertaining when to "jump backwards" (i.e. when the next β jump will loop the target index value back to the beginning of the list, or near it). Fortunately, these locations can be deduced by adding the width of the β chord to the value of the index at the present location, and then subtracting the highest index value in the linked list from the result. If the result is *not strictly greater than zero*, then there is still room to "move forward." On the other hand, if the check fails, then the destination index can be found by subtracting 1 from the result of the check (assuming that the list is counting in ascending order, and also by 1). The result should be the graph shown in Fig. 5.2. The array variable C can be used to control the path selection in the linked list in a manner similar to D in the previous subsection. The only caveat is that the destination index must not have been bypassed by a previous β chord. This is a condition for the resulting graph to be consistent.

5.4.2 *Results Analysis*

Let us now examine the testing results of a chordal ring network of Fig. 5.3 which was generated using the methods above, in a CADENCE UMC90nm technology. The token ring is composed of fast DCs. RST_BAR is an active-low signal used to inject a control token at Out6 by inducing a $\langle 1, 0 \rangle$ vector pair on signals x_6 and y_6 and the reverse on all of the on and also flush any remnants of the prior token in the system. It should also be noted that signals $\langle$BYP0, BYP1, ..., ByP7$\rangle$ are actually $\langle \beta 7, \beta 0, ..., \beta 6 \rangle$, which made it phasing out nodes more intuitive because nodes are excluded from the system by the decisions of their "previous" neighbor.

The transient response of the system is depicted in Figure5.4. It ran for a duration of 24 ns in which the number of fast DCs present in the system was gradually reduced from 8 cells to 4 (the absolute minimum as discussed in the beginning of this section). The tests were repeatedly run across the SN/SP, TN/TP, and FN/FP

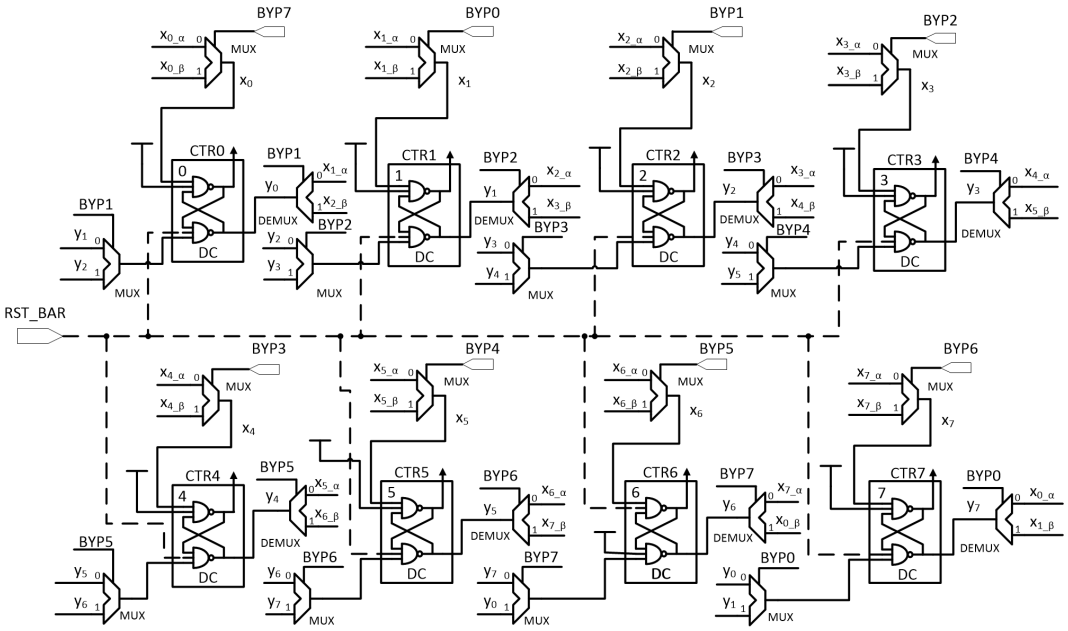


Figure 5.3: Schematic of the self-timed reconfigurable token ring control circuit based on DCs illustrating the effect of exclusion on the time available for synchronization in the system (path: CTR6)

Table 5.1: Variations in the Time Available for Synchronization in the Reconfigurable Token Ring Control across 3 Major Process Corners @V_{DD} = 1.0 V and at a Temp = 27 °C

ProcessCorner @27 °C	L = 8 (ps)	L = 7 (ps)	L = 6 (ps)	L = 5 (ps)	L = 4 (ps)
SN/SP	931.1	775.1	609.6	383.8	213.7
TN/TP	692.4	564.8	447.5	270.4	159.1
FN/FP	553.1	455.1	364.7	242.3	116.0

process corners at a constant temperature of 27°C as well as additional tests in which the temperature was varied from 0 to 100°C while holding the process corner at TN/TP. All results indicated that the time available for synchronization (t_{MSR}) decreased as the number of fast DCs were excluded via induced control signals (which was expected). The control signals were auto-generated using a MATLAB front-end and fed into cadence via a series of *vpwlf* (piecewise linear voltage source) elements. The simulated rise and fall times of these test signals was 5ps, and each vector had a duration of 1ns.

5.5 UCOM THREAD FORWARDING

In the previous chapter, UCOM threads were used to demonstrate how an interrupt subsystem could be controlled by a cyclic loop composed of interleaved UCOM

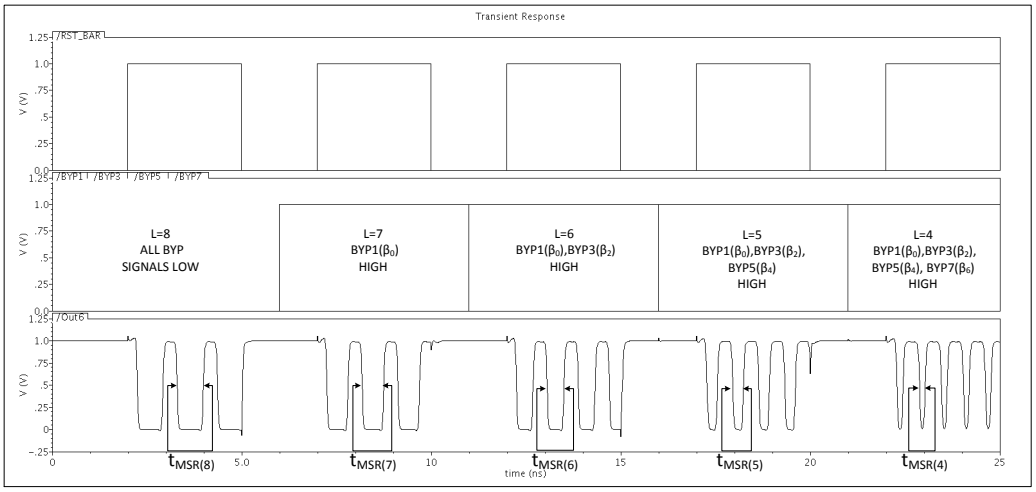


Figure 5.4: Transient response of the self-timed reconfigurable token ring control circuit based on DCs illustrating the effect of exclusion on the time available for synchronization in the system (path: CTR6)

Table 5.2: Variations in the Time Available for Synchronization in the Reconfigurable Token Ring Control across Temperature Regions @V_{DD} = 1.0 V and a TN/TP Transistor Process

Temperature (°C)	L = 8 (ps)	L = 7 (ps)	L = 6 (ps)	L = 5 (ps)	L = 4 (ps)
0	636.5	530.5	414.5	258.7	137.3
27	692.4	564.8	447.5	270.4	159.1
100	790.8	663.1	515.2	337.9	182.4

signals. However, in that chapter, all of the nodes were visited in a deterministic manner. In order to achieve UCOM locking and control in a reconfigurable token loop incorporating chordal jumps as discussed above, it is also necessary to utilize UCOM signals within the token loop in a reconfigurable manner. That is to say, the assertion and de-assertion of UCOM signals must also be contingent on the path taken through the token ring.

5.5.1 *Theoretical Overview (Why Forward?)*

As discussed in Chapter 2, adding CSC signals to a STG is required when the progression of signal transitions within the behavior graph of the a circuit leads to state encodings which are not unique. CSC signals also require conditions for their assertion and de-assertion within the system, as do the other signals the STG representation of an asynchronous circuit. In a graph where all paths are deterministic in nature, these conditions are easily met even when the encodings are interleaved as they were in the previous chapter.

However, when the circuit is constructed as in Fig. 5.1, the situation becomes more complex. The interleaved nature of the UCOM signals within our token ring construct implies that it is possible when bypassing vertices within a STG using chordal jumps, some of the interleaved UCOM transitions will also be bypassed. Therefore the STG of the system needs to be modified to do either one of two things.

1. Replace the UCOM signals bypassed through exclusion via chordal jumps with an alternate set of UCOM signals depending on the path taken through the STG.
2. Forward the UCOM signals bypassed through exclusion via chordal jumps to their appropriate destinations in the STG, to ensure consistency.

In this work we will focus on the latter.

5.5.2 *Applications to Robust Controller Design*

The above discussion only pertains to the connectivity and implementability of the token ring itself. As stressed in the prior chapter, the reconfiguration protocol defined earlier requires that the control token traversing the ring be halted at a known location when a reconfiguration action is performed. This is necessary to both ensure that the device starts and stops at known locations, as well as ensuring that the control token is not lost during reconfiguration. Token spreading as discussed in Chapter 3, further exacerbates this problem, as it places additional restrictions on the reconfigurability of the system by requiring that the locations occupied by the control token remain untouched during a reconfiguration operation. Thus, more than one reconfiguration operation will be required to configure all of the paths in the system graph, even if the optimization in Section 3.3.3 is used.

5.6 ADVANCED CONTROLLER DESIGN (HIERARCHY & CROSSBAR PLUG-IN)

With the above discussion in hand, we can now present a new design for a reconfigurable control device which addresses the concerns raised earlier. In order to accomplish this, we will first discuss the organization of the controller from a theoretical standpoint in Section 5.3.1. Thereafter, Section 5.3.2 will provide a derivation of the building blocks that feature prominently in the PN model of Section 5.3.3. Finally, Section 5.3.3. will examine the operation of a PN which behaves in the manner that we desire.

5.6.1 *Theoretical Overview (System Hierarchy)*

On an abstract level the token ring of Fig. 5.1 can be viewed in a manner analogous to a one-way train track formed by three concentric circles with differing radii, as in Fig. 5.5. At each station (i.e., vertex) a decision is made as to whether or not to continue along the outer or the inner track, depending on which track the train

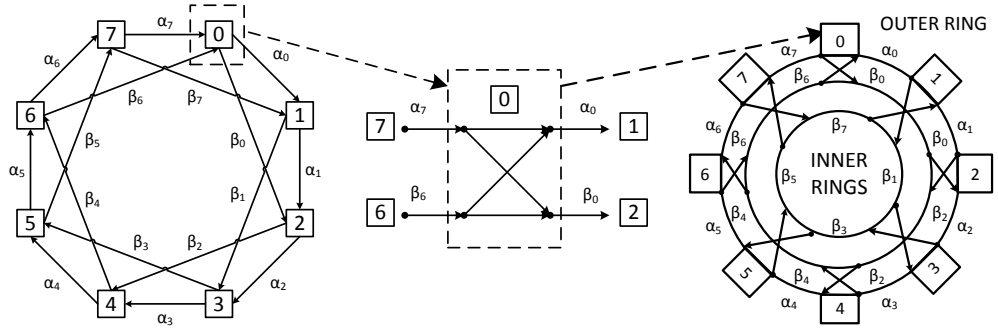


Figure 5.5: Conceptual overview of the token ring based on chordal graphs.

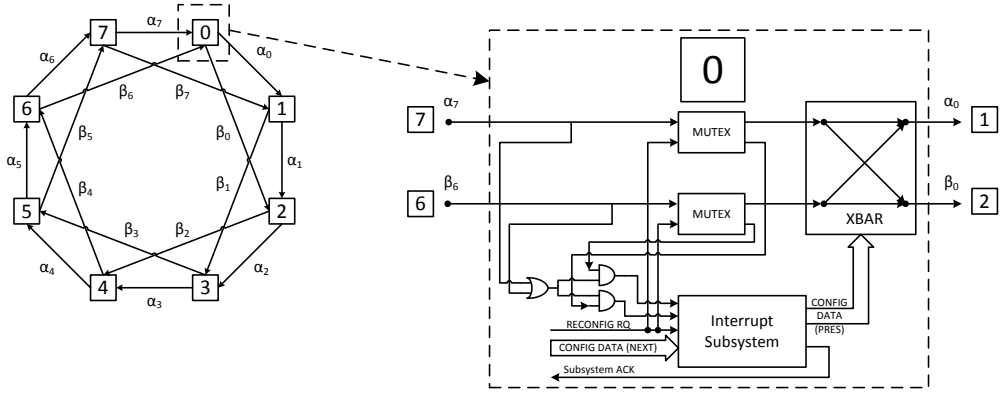
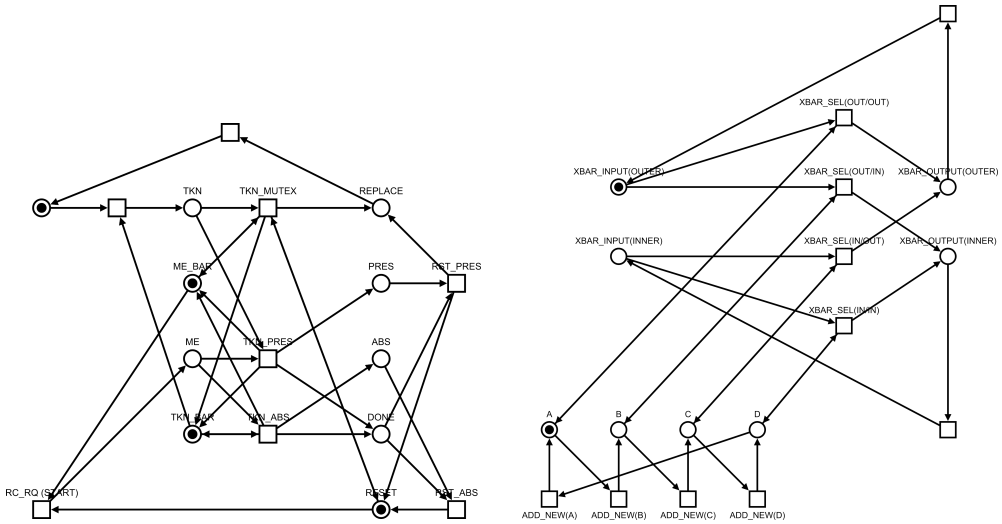


Figure 5.6: Overview of the reconfigurable interrupt an XBAR select at vertex 0.

(i.e., control token) arrived on, or whether to switch tracks using a junction box (i.e., crossbar). While the train runs uninterrupted through the track (i.e., token ring) the path traveled through the junctions is not allowed to change. Otherwise the train might derail (i.e. the control token will be lost).

Continuing with that analogy, the outer and inner tracks have different properties. The outer track visits the greatest number of stations, but has a longer path as a result. By contrast, each of the two inner tracks only visit a subset of the stations that the outer track does, but arrive at their destination faster as a result, because they both bypasses certain stations (vertices). The arrival times (i.e. duty cycle) of the train depend on the number of stations visited. However, if the stations on the outer track go out of service (i.e. suffer a permanent fault), the inner track (i.e. the bypass path) must be used to ensure that the train can still visit the other stations (i.e. the vertices from the non-faulty devices must maintain their reachablility). However, the arrival time (duty cycle) of the train (control token) will be permanently altered as a result.

The following sections will be concerned with the construction of the stations (vertices) in the behavior graph of the system.



(a) MUTEX element (b) 2x2 XBAR element

Figure 5.7: PN of building blocks for the reconfigurable control device.

5.6.2 *Derivation of the PN Model*

Moving away from analogies, let us explore the practicalities of what a circuit satisfying the criteria mentioned above entails. Fortunately, a fair bit of insight can be gleaned from the material listed in prior chapters of this work. In order to safely perform a reconfiguration operation, the progress of the control token must be halted at a known location, just as it was in Chapter 4. The path that the control token takes throughout the ring must be a function of the configuration data output from the interrupt subsystem. This path data took the form of a (one-hot) control code being input to the MUX and DEMUX elements of an embedded cycle graph in Chapter 3. Now that we are dealing with chordal graphs in this chapter, this control data will take the form of a crossbar selection signal, as in Fig. 5.6. From Fig. 5.6 it can be observed that the only appreciable difference between this control device and that of Fig. 3.9 is with regard to the handling of the requests and grants from multiple MUTEX elements, instead of a single one as in Fig. 3.9.

While the next subsection will present a PN which models the behavior of reconfigurable control device suitable for use in a chordal graph, such a Net is unwieldy to examine in isolation. The model can be better understood if the PNs for each module in the control device are discussed individually beforehand. To that end, in this subsection we will review the operation of the PNs for a MUTEX, a 2x2 crossbar, and the interrupt subsystem of the control device.

5.6.2.1 *Petri Nets for the MUTEX and Crossbar*

The PN of the MUTEX element in Fig. 5.7(a) can be analyzed as follows:

Before delving into the progression of the transitions in the PN MUTEX, we should first define the important labels in the Net. The places ME and ME_BAR are in-

verses which represent whether the interrupt (ME = 1) or token ring (ME_BAR =1) portions of the PN are enabled. Similarly, the places TKN and TKN_BAR represent whether a control token is present (TKN=1) at the input to the MUTEX (TKN_MUTEX) or not (TKN_BAR=1). The final set of paired inverse places are PRES and ABS, which indicate whether or not a token was present at place TKN and consumed during the interrupt operation. PRES and ABS do not have much meaning in this context, however they become important when considered in tandem with Fig. 5.8 and Fig. 5.9 where they act as control places in the PN.

Let us now walk through a MUTEX operation. When RC_RQ fires, a control token is removed from the read arcs of both ME_BAR and RESET, and the token ring transition at TKN_MUTEX is disabled. If a token is present at place TKN then transition TKN_PRES is enabled, otherwise transition TKN_ABS is enabled. If TKN_PRES (TKN_ABS) fires, then a token is placed on PRES (ABS) and a token is also placed on DONE. If PRES (ABS) is high then, RST_PRES (RST_ABS) is enabled, which consumes the token at DONE and places a token at REPLACE (RESET) thus re-enabling token ring.

The PN of the 2x2 crossbar in Fig. 5.7(b) can be analyzed as follows:

The XBAR_INPUT (INNER) and XBAR_INPUT (OUTER) represent the inputs to the crossbar along the inner and outer token rings, respectively, while XBAR _OUTPUT (INNER) and XBAR_OUTPUT (OUTER) similarly represent the outputs. The select signals XBAR_SEL(OUT/OUT), (OUT/IN), (IN/OUT), (IN/IN) are controlled, respectively, by the read arcs of A, B, C, and D. In the example of Fig. 5.7(b) read arcs A, B, C, and D are altered by firing the transitions ADD_NEW (B), (C), (D), and (A) respectively. In short, adding a token to A removes one from D, adding a token to B removes one from A, etc. The only other points which bear mentioning are that the inner ring can only be accessed via the outer ring while B is asserted, and vice versa for C.

5.6.2.2 *PN for the Interrupt Subsystem*

The PN of the interrupt subsystem in Fig. 5.8 can be analyzed as follows:

CFG_EN (START) represents the configuration grant from one of the MUTEX elements in the inner and outer ring (i.e., it is assumed that only one control token exists, therefore only one MUTEX issue the grant). CFG_FLUSH enables one of the RM_OLD (A), (B), (C), (D), or (E) transitions, depending on whether the token is at location A (OUT/OUT), B (OUT/IN), C (IN/OUT), D (IN/IN), or E (ABS/ABS). Once the RM_OLD transition fires a token is removed from A, B, C, D, or E, and is placed at the location CFG_STORE. The transitions enabled from CFG_STORE depend on the whether the token is at the outer ring (PRES(OUT) = 1, ABS(IN) = 1), the inner ring ((PRES(IN) = 1, ABS(OUT) = 1), or neither (ABS(OUT) = 1, ABS(IN) = 1). If the token is at the outer ring, then ADD_NEW (A) and (B) are enabled. Similarly, if token is at the inner ring, then ADD_NEW (C) and (D) are enabled. Finally, if the token is at neither the outer ring, nor the inner ring, then ADD_NEW (E) is enabled. Once ADD_NEW (A) (B) (C) (D) or (E) fires, a new token is placed at

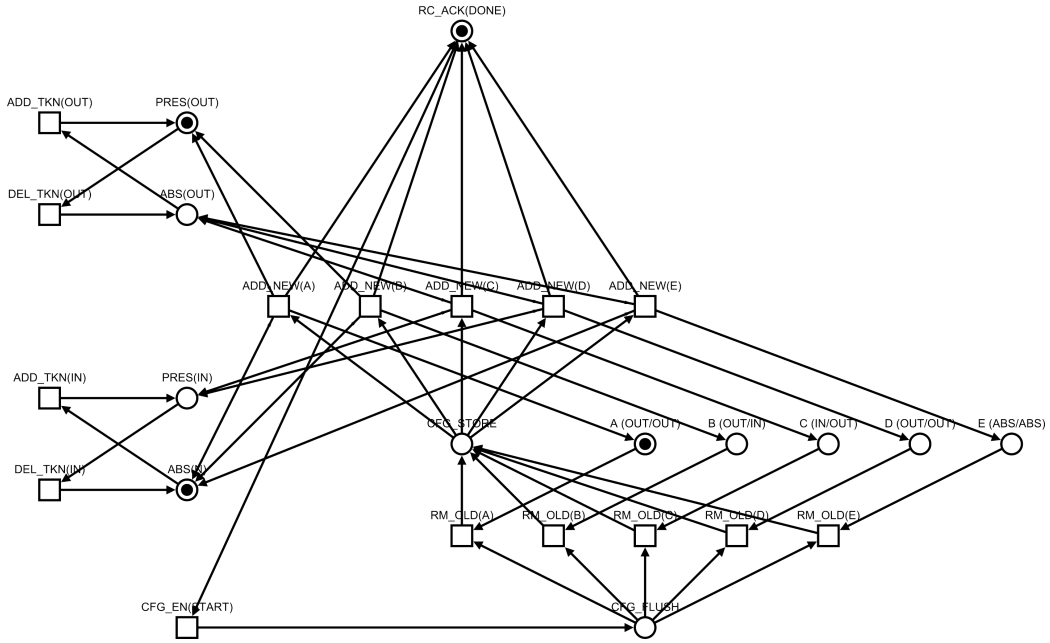


Figure 5.8: PN of the interrupt subsystem.

the corresponding location A (OUT/OUT), B (OUT/IN), C (IN/OUT), D (IN/IN), or E (ABS/ABS) and a token is placed at RC_ACK (DONE), which indicates that the reconfiguration operation has concluded.

5.6.3 *PN Model & Simulation*

With the building blocks in Section 5.6.2 in hand, the PN of Fig. 5.9 can now be discussed in detail. Due to the modularity property mentioned during Chapter 2, the PN of Fig. 5.9 inherits the characteristics of the modules which compose it. All of the individual modules in Figs. 5.7(a) 5.7(b) 5.8 had PNs that were both 1-safe, and deadlock-free. Thus, the PN of Fig. 5.9 must also be 1-safe, and deadlock-free as well. It should be noted that the RESET read arc feeding into TKN_MUTEX in Fig. 5.7(a) has been substituted with a read arc from RC_ACK which feeds into both TKN_MUTEX (OUT) and TKN_MUTEX (IN) in Fig. 5.9. However as this is a functionally equivalent operation, no behavior or PN properties in Fig. 5.9 are altered.

As the individual behavior of each module that composes the PN of Fig. 5.9 was just discussed in Section 5.6.2, there is no need to re-iterate it again. Looking at this PN in more detail, we can see that the underlying circuit consists of two MUTEX elements, one for the outer ring (α-path) and one for the inner one (β-path), a reconfigurable interrupt subsystem used to switch between and "remember" configurations, and a crossbar which is responsible for routing the control token and its UCOM signals through the chordal ring construct. Because the location of the token is unknown at the time of reconfiguration, all of the MUTEX elements in the token ring need to poll for the presence or the absence of the control token. As stated earlier, if the control token is present at a location (or locations if token spread-

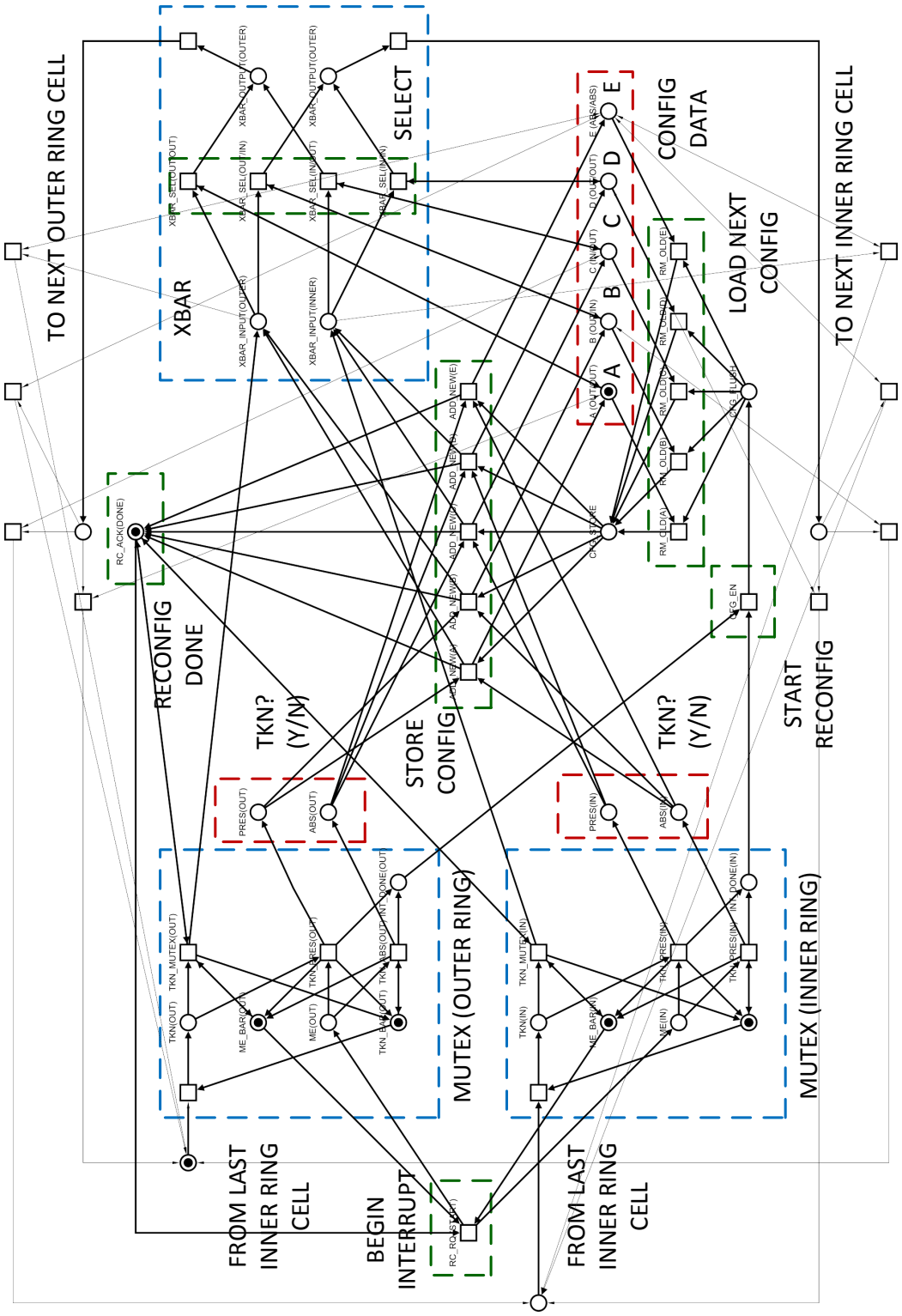


Figure 5.9: PN representation of chordal reconfiguration protocol.

Table 5.3: XBAR Select Signals

CODE	PATH
0001	INNER/INNER
0010	INNER/OUTER
0100	OUTER/INNER
1000	OUTER/OUTER

ing is involved), that locale is forbidden from participating in the reconfiguration process.

If it is desirable for the control token to start or stop from a specific vertex in the token ring, the interrupt subsystem can be designed to give priority to that vertex by manipulating the firing order of the interrupt devices in the control subsystem using UCOM threads, as in the last chapter. By doing so it is possible to implement reconfiguration on the entire token ring without having to poll every vertex in the token ring as stated above, but it comes at the cost of introducing a set failure point into the circuit.

5.7 VALIDATING CONFIGURATIONS VIA A NEAREST NEIGHBOR CHECKING ALGORITHM

A final aspect of the controller design which bears mentioning is the verification that reconfiguration operations result in no discontinuities in the path traversed by the control token through the ring. The purpose of this check is to test whether of not the new configuration is valid prior to releasing the control token along the ring.

5.7.1 *Validation Algorithm*

The algorithm that checks the validity of the configurations asserted on the token ring functions as follows. After a reconfiguration operation is complete, each vertex checks the configurations induced on either its left or right hand neighbors in a round-robin fashion. When the check is complete, a completion signal is sent to a MCE along with a dual-rail code indicating whether or not the logical check evaluated to a high (valid) or low (invalid) result. When all of the vertices have reported (i.e. when the MCEs go high), the results of the logical check are evaluated. If all of the results returned valid (high), the configuration is accepted, and the control token is released to traverse along its new path. If the results of the check returned false, the a flag is sent back to the system indicating that the configuration information was invalid, and the prior configuration will have to be reinstated before the control token can be released.

We use one hot-state assignment to give unique values to the four possible outputs of the 2 x 2 crossbar in Fig. 5.9, as in Table 5.3. The entries in Table 5.4 are used to perform the checks on the nearest left-hand (LEFT) or right-hand (RIGHT)

Table 5.4: Nearest Neighbor Checking Algorithm

PRES	LEFT	FUNC
0001	0001	1
0001	0010	0
0001	0100	1
0001	1000	0
0010	0001	1
0010	0010	0
0010	0100	1
0010	1000	0
0100	0001	0
0100	0010	1
0100	0100	0
0100	1000	1
1000	0001	0
1000	0010	1
1000	0100	0
1000	1000	1

PRES	RIGHT	FUNC
0001	0001	1
0001	0010	1
0001	0100	0
0001	1000	0
0010	0001	0
0010	0010	0
0010	0100	1
0010	1000	1
0100	0001	1
0100	0010	1
0100	0100	0
0100	1000	0
1000	0001	0
1000	0010	0
1000	0100	1
1000	1000	1

neighbors, using the entries in Table 5.3 as a key. After reduction they yield the following equations, where [A,B,C,D] correspond to the [0,1,2,3] vector entries of the XBAR select signal at the present node (PRES), and [E,F,G,H] correspond to the [0,1,2,3] vectors of either the nearest left-hand (LEFT) or right-hand (RIGHT) neighbors.

$$\begin{aligned} VRHS &= A'C'E'F'(GH' + G'H)(BD' + B'D) \\ &+ B'D'G'H'(EF' + E'F)(AC' + A'C) \end{aligned} \tag{27}$$

$$\begin{aligned} VLHS &= A'B'E'G'(FH' + F'H)(CD' + C'D) \\ &+ C'D'F'H'(EG' + E'G)(AB' + A'B) \end{aligned} \tag{28}$$

6 FLOW CONTROL IN WAGGING SYNCHRONIZERS INCORPORATING FIFO BUFFERS

6.1 INTRODUCTION (FLOW CONTROL IN FIFO SYNCHRONIZERS)

In the final chapter of this work we will address the issue of flow control in parallel synchronizers that incorporate FIFO buffers to decouple their read and write operations. As stated in section 3.1.3, FIFO synchronizers decouple read and write operations from each other by using a FIFO buffer. From the perspective of a write operation, the FIFO only exists as a black box to sink data items, ideally without limit. Conversely, the read operation only sees the buffer as a data source from which values can be read out, also without limit in an ideal scenario. The complications as stated in earlier in this thesis arise from the fact that the memory storage space is finite, and that mismatches between the transmitter and receiver ends of the synchronizer eventually lead to either *data overflow* if the transmitter writes data into the buffer faster than the receiver can read it out, or *data starvation* if the situation is the reverse [15].

Section 6.2 will introduce the issue of data accumulation and starvation and provide a brief review of a prior FIFO synchronizer design that has attempted to address this issue in the past. Continuing forward, Section 6.3 will discuss the relevant methods for analyzing the flow control in a FIFO synchronizer, and also comment on the quality of the VHDL tests that follow later in the chapter. Finally, Section 6.4 will define the experimental setup that will be used to test the VHDL design in question, while Section 6.5 and analyze the results of the final VHDL design and simulation.

6.2 RELATED WORK (STARI)

Managing the data flow in a FIFO synchronizer was the topic of prior discourses in 1993 and 1995 when Greenstreet implemented a mesochronous interface that was Self-Timed At the Receiver Input (STARI), as shown in Fig. 6.1 [36], [37].

This interface mandates that the FIFO be initialized in a half-full state, whereafter the reset signal is released and the read and write pointers both count from 0 to 2 offset from each other by two entries, as shown in 6.2. If the clock frequency of both sides matches exactly and is in phase, then the Four-cell FIFO buffer never starves or gets full. It can even tolerate the case where the transmitter and receiver clocks fall out of phase with one and another, contingent on the requirement that the read and write pointers do not overlap. Using this approach also avoids the necessity of synchronizing the full and empty signals of the FIFO as in other designs [15].

While this design is simple and has been used before in architectures like the Intel TeraFLOPS chip, it does have some inherent limitations [79]. First, the clocks at the transmitter and receiver ends of the system must match (otherwise overflow and underflow can occur), though the phase relationship of each end may vary.

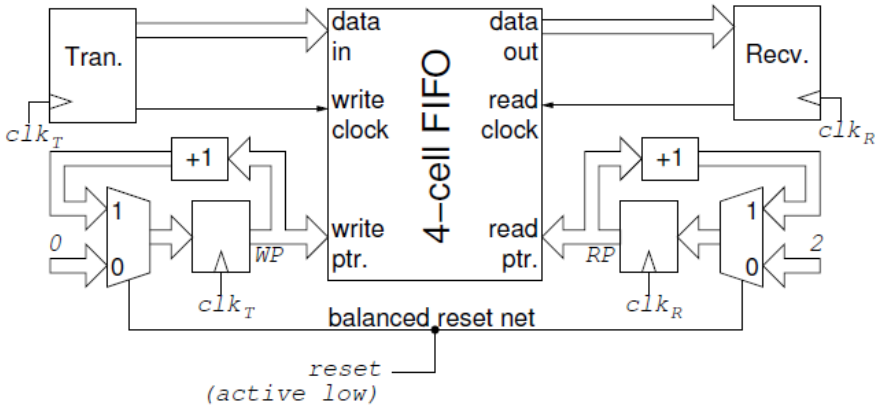


Figure 6.1: Four-cell STARI interface schematic. From [12]

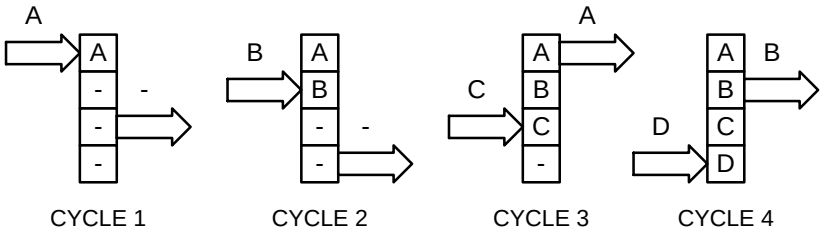


Figure 6.2: Data flow of a four-cell STARI interface, assuming that the transmitter and receiver clocks are in phase. From [12]

As a mesochronous design, this is to be expected. Second, the transmitter requires that the transmitter use self-timed (i.e., dual-rail) codes at its output, presumably so that the signals arriving at the input of the receiver are atomic.

6.2.1 *Applications to Wagging Synchronization*

Let us take a look at a FIFO synchronizer which incorporates the principles of the prior subsection, as in Fig. 6.3. Central to this design are two major points. First, is the maximization of the synchronization performance from the input data channel to the input of the FIFO by pooling the gain-bandwidth products of the parallel master-slave flip-flops, as illustrated in prior work by Horstmann [40]. Second is the manipulation of the data flow mismatch between the transmitter (Tx) and receiver (Rx) ends of the synchronizer. However, before proceeding further it must be understood that the example used throughout this chapter is only a single case of a larger problem, rather than a comprehensive solution. With that in mind, Fig. 6.3 provides a general overview of the synchronizer that will be used throughout the remainder of the chapter.

First, let us assume the FIFO is asynchronous, and that reads and writes occur independently of each other, where $f_{clk(Tx)}$ and $f_{clk(Rx)}$ represent the local data rates of the transmitter and receiver regions of the synchronizer, respectively. Let

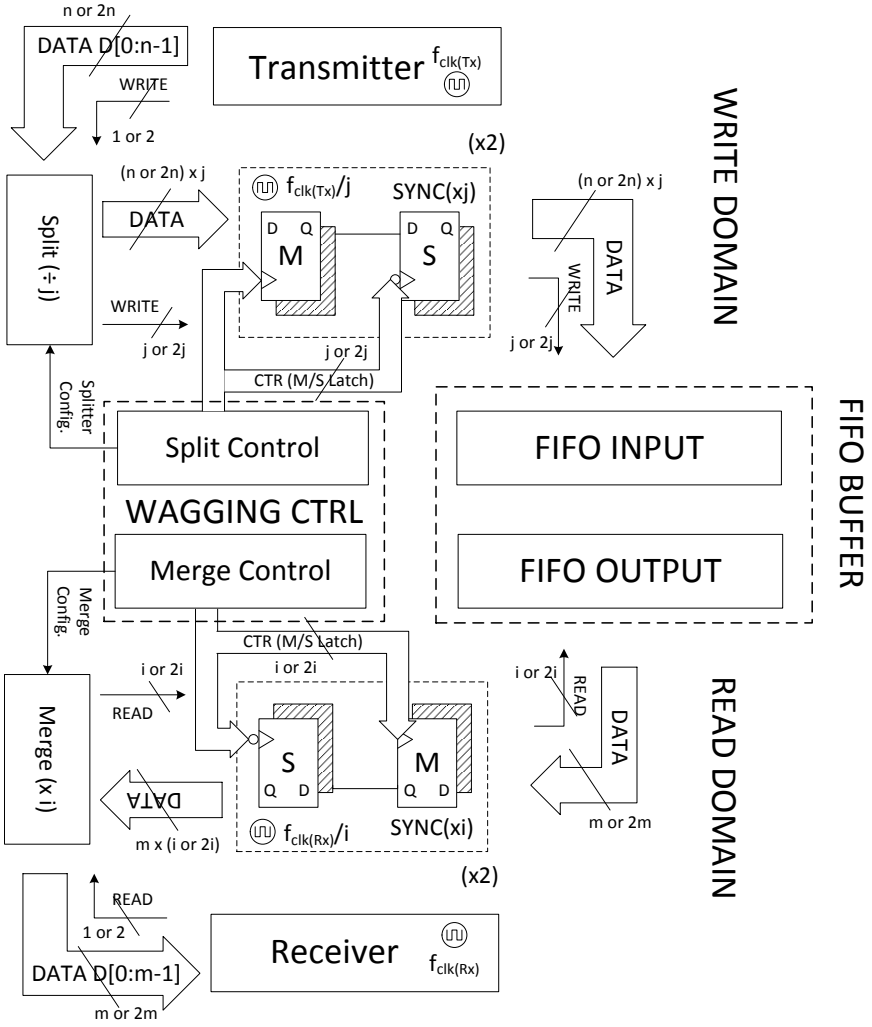


Figure 6.3: Top view of a FIFO synchronizer incorporating wagging at the transmitter and receiver.

us further assume that i and j are multiples of a common base frequency f_0, and that $i < j < 2i$ (i.e., the transmitter is faster than the receiver, but not by more than a factor of 2).

With those assumptions in hand we can step through the operation of Fig. 6.3 as follows. The serial input data and write validation signals (DATA, WRITE) from the transmitter arrive at the splitter module at a rate of $j \times f_0$ where they are both split into j identical signals using mixers. These signals are then broken into j tasks (slices), through sampling via the use of j parallel master-slave flip-flops which act as synchronization (SYNC) elements. Thereafter, these SYNC elements are triggered using control signals which all operate at a base frequency of f_0, but are offset from each other by j divisions as shown in Fig. 6.3 (though only DATA is shown). Finally, the DATA and WRITE signals arrive at the FIFO input in j parallel lines operating at rate of f_0. Subsection B will illustrate the impact of this process on the synchronizer MTBF.

Continuing forward, we can now examine the operation of the synchronizer from the output of the FIFO to the input of the receiver. Because $i < j < 2i$, the FIFO is still subject to data accumulation. In order to minimize such accumulation, the receiver needs to be designed to allow for the data being read out from the FIFO to temporarily exceed the amount of data being written in when certain conditions are met. This is accomplished by allowing the read operations to be done either serially, or in parallel.

Both read operations can use the same hardware. During a serial read, i parallel data lines from the FIFO are sent to a mixer that recombines them along $data_{prime}$ at a rate of $i \times f_0$, in a manner which is identical to the recombination of the signals in the wagging buffer discussed earlier. A parallel read operation functions similarly, except that data is simultaneously read out along both $data_{prime}$ and $data_{sec}$.

Whether or not a serial or parallel data read is necessary depends on the present memory differential between the transmitter and receiver ends of the FIFO synchronizer, $(j-i)_{cur}$ as defined by (29).

$$\begin{aligned} &\text{If } (j-i) + \textstyle\sum (j-i)_{cur} < i, \text{ then increase} \\ &\textstyle\sum (j-i)_{cur} \text{ by } (j-i), \text{ (serial read)} \\ &\text{If } (j-i) + \textstyle\sum (j-i)_{cur} \geqslant i, \text{ then decrease} \\ &\textstyle\sum (j-i)_{cur} \text{ by } i-(j-i), \text{ (parallel read)} \end{aligned} \tag{29}$$

When the conditions for a parallel read are met, the system continues parallel read operations until the memory differential reaches 0, whereafter it resumes serial operation. It should also be noted that read acknowledgement signal (READ) is processed in vectors of length i or $2i$ depending on whether a serial or parallel read operation was last performed.

6.3 FIFO TESTING METHODOLOGY

This subsection will provide an overview of the theory which underlies the VHDL tests performed later in this chapter. Section 6.3.1 will discuss the relevant top-level design considerations of the parallel FIFO synchronizer introduced earlier. Thereafter, Section 6.3.2 will address design concerns in the FIFO I/O ports.

6.3.1 *Top-level Design Considerations*

With the discussion of the last section in hand, we can now discuss some important top-level issues in the synchronizer design outlined above.

First, we must examine the priority that the reconfiguration request takes in the system and behavioral requirements which surround it. When a reconfiguration request is issued there are a few prerequisites that must be satisfied. If the transmitter in the system is still sending data, then it must be asked to stop doing so via a handshake before reconfiguration is performed. Concurrently with the request to stop sending data, the FIFO buffer must be allowed to empty itself of data. Both of these measures are intended to prevent data loss as the system un-

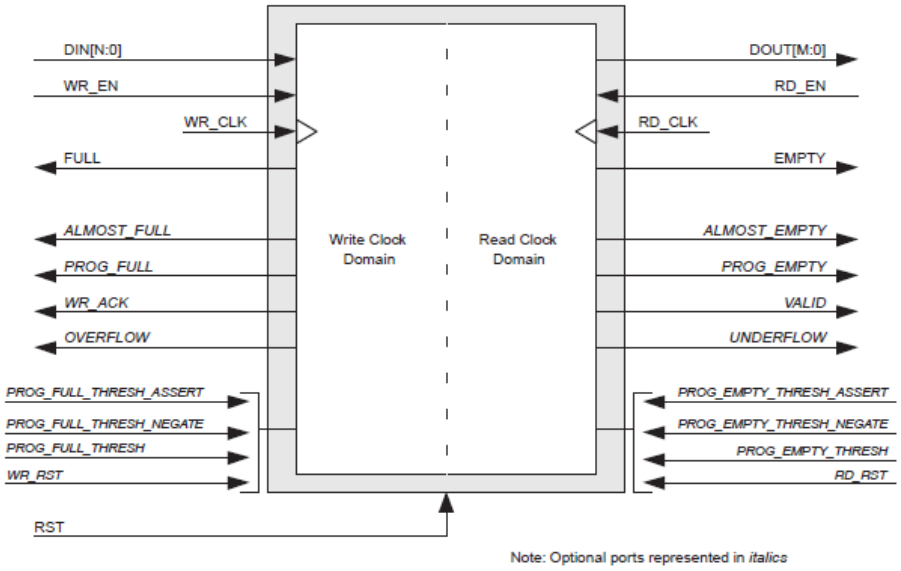


Figure 6.4: FIFO standard cell. From [82]

dergoes the reconfiguration process. After the EMPTY signal is asserted, then the reconfiguration request can proceed as normal. Though these assumptions lead to a communication penalty upon reconfiguration that is greater than the penalty incurred on the start-up of a baseline FIFO synchronizer, if reconfiguration operations are sufficiently infrequent then the total communication penalties incurred over a prolonged period of time will be less than in the baseline design, assuming that the sender and receiver clocks are not well matched.

Second and finally, the FIFO synchronizer requires a method for deciding when to minimize data accumulation and starvation through the application of serial and parallel writes discussed in Section 6.2.1. Fortunately, there are standard cell designs for FIFOs which not only generate the FULL and EMPTY signals necessary for proper implementation, but also allow for the definition of optional user defined signals ALMOSTFULL and ALMOSTEMPTY, as in Fig. 6.4 [81]. The vector combination of these four signals can be used to ascertain how full or empty the FIFO buffer is at any given moment. If we assume that the signals are all active high and that the threshold for ALMOSTFULL and ALMOSTEMPTY are set to 60% and 40% of the buffer's capacity, then the vector combination [0, 0, 0, 0] serves the same function as initialization did in the STARI interface (i.e., ensuring that the buffer starts close to half-full) [37]. Once the [0, 0, 0, 0] state is reached, the parallel read and write operations proceed as defined by (29).

If the buffer is almost full [0, 0, 1, 0], then only parallel reads are enabled until the almost empty flag is triggered [0, 1, 0, 0] where after serial reads are performed until almost empty flag is de-asserted again. Once again, this assumes that the transmitter is faster than the receiver.

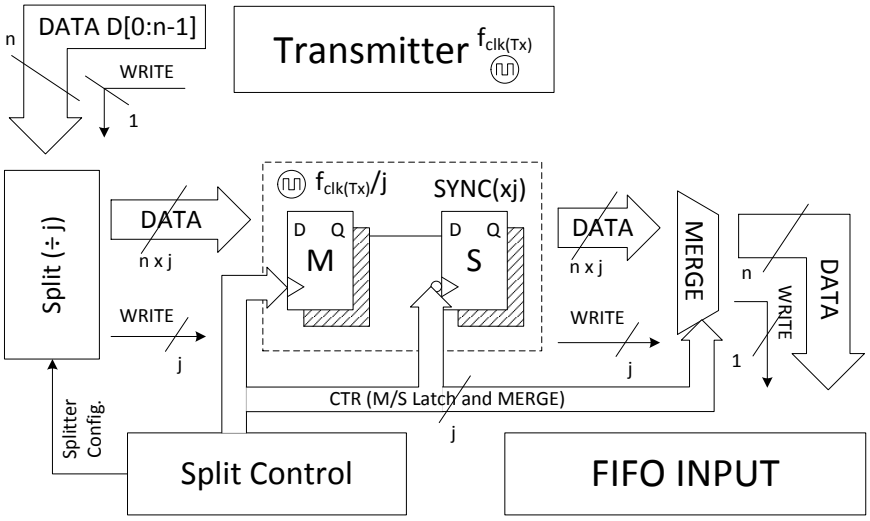


Figure 6.5: Serialized input from the transmitter to the input of the FIFO buffer.

6.3.2 *FIFO I/O Considerations*

The top-level specification of the FIFO buffer in Fig. 6.3 requires that the data width of the input and output ports attached to the attached memory space be freely adjustable. In practice, there is a finite limit to this sort of configurability. As the simplest FIFOs possess serial data inputs and outputs, it is useful to explore methods by which these parallel inputs and outputs may be serialized to aid in synthesis using simpler devices.

From the output of the transmitter to the input of the FIFO, the parallel data input (DATA) and valid (WRITE) signals can both be serialized through the use of MUX elements immediately prior to the FIFO inputs, as shown in Fig. 6.5. The MUX can be controlled using atomic signals generated by the vector combination of the CTR signals which act as enable signals for the SYNC elements. Whether combinational or sequential, the atomicity of these signals is guaranteed by consequence of the underlying token ring structure. The only caveat to this method is that the latency of the MUX becomes part of the specification itself and must be accounted for, to ensure that it samples data from the slave latches of the SYNC elements during the stable portion of their duty cycle (i.e., at the closest transition prior to the next sampling period of the slave latch). The mapping function of the inputs and outputs of the MUX also changes depending on the type of code used, however modern synthesis tools render the issue pedantic.

Satisfying the requirements for serial and parallel operations from the output of the FIFO to the input of the receiver is more challenging. Because parallel operation is necessary, the memory space needs to be split between two independent serial-output FIFO devices.The serial output of the secondary FIFO must be sent to a Demultiplexer (DEMUX) element, where one path leads to the input of a MUX that ties together the serial outputs of both FIFOs, and the other path leads to the secondary SYNC element. During serial operations, the MUX acts as a toggle, while during parallel operations it functions as a straight path. During parallel opera-

tions the output DEMUX acts as a straight path to the SYNC element, while during serial operations it leads to the MUX.

If the FIFOs are split as described above, then a DEMUX needs to be added to the FIFO input to toggle the DATA and WRITE signals between themThe last point of note is that the read acknowledgement signal also needs to be toggled between the two FIFOs during serial operation, and issued to both FIFOs simultaneously during parallel operations.

6.4 VHDL EXPERIMENTAL SETUP

In this section we will explore a VHDL implementation of the FIFO design presented earlier. Section 6.4.1 will touch on the VHDL design process. Thereafter, Section 6.4.2 will address the concept of automatic test pattern generation. Next, Section 6.4.3 will examine state machine logic which underlies the VHDL synthesis of the design. Finally, Section 6.4.4 will discuss and analyze the final VHDL simulation results.

6.4.1 *VHDL Design Flow*

VHSIC Hardware Description Language is a programming language developed in 1988, which is used to generate gate-level net lists of circuit specifications [74]. At the abstract level, every VHDL circuit description contains an ENTITY statement which defines the I/O ports of the device, and an ARCHITECTURE statement which defines its behavior. Blocks are then connected together using PORT MAP statements. From these basic principles, complex hierarchies can be constructed. A net list is generated upon successful program compilation. These net lists can either be functionally simulated, imported into other tools for further optimization, or placed and routed on a field programmable gate array (FPGA).

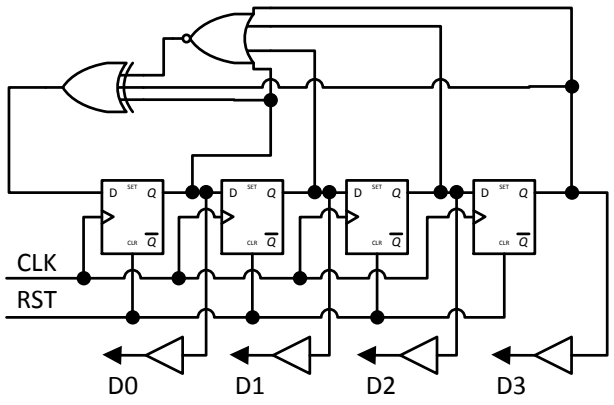
6.4.2 *Test Vector Generation*

When testing systems in VHDL, it is necessary to generate appropriate input stimuli. While it is feasible to produce input signals using a series of successively divided clock signals thereby approximating the sequential input combinations of a basic truth table, it is more useful from a testing perspective to automatically generate repeating pseudo-random patterns at the input of a circuit which can then be automatically checked and verified at the at the output to ensure correctness.

A four-bit PRBS generator, as shown in Fig. 6.6(a), was used to generate the input test patterns. It produces $2^n - 1$ combinations before repeating , with $n = 4$ in this case. The test sequence is depicted by Figure 6.6(b). It has a characteristic equation depicted by (29)

$$1 + X_0{}^1 + X_1^2 + X_2^3 + X_3{}^4, \text{ where } X_n = D(n) \tag{30}$$

The terms present in the equation are determined by the connectivity of Fig. 6.6(a), which makes the characteristic equation $1 + D0^1 + D3^4$. The NOR logic



(a) Schematic

$D0$	0*	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0
$D1$	0*	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0
$D2$	0*	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0
$D3$	0*	0	0	0	1	1	1	1	0	1	0	1	1	0	0	1

(b) Truth table. *-only happens at start-up

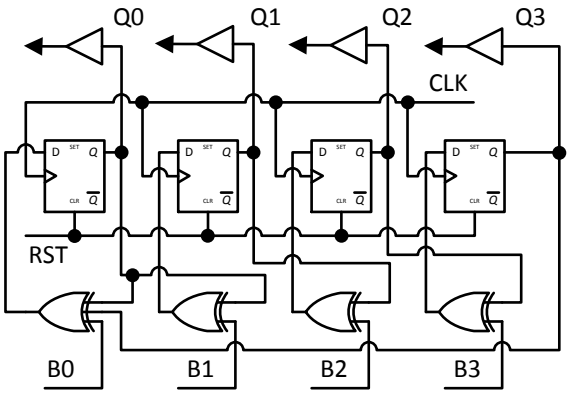
Figure 6.6: Four-bit PRBS generator.

associated with the final term has the net effect of adding the code $\langle 0, 0, 0, 0 \rangle$ to the sequence of binary numbers, which allows the PRBS generator to operate without external input stimuli aside from the clock (CLK) and reset (RST) signals.

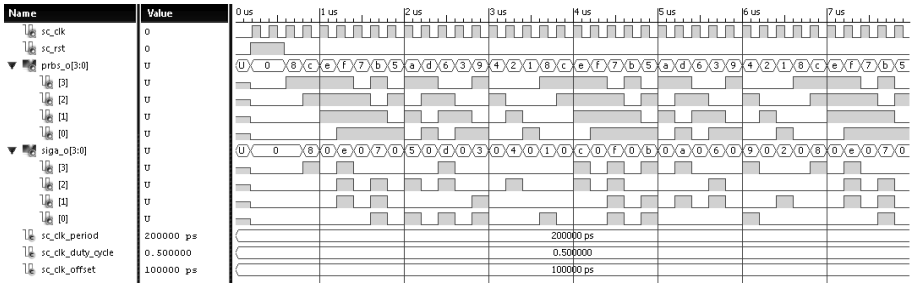
At the opposite end, the outputs from the PRBS generator (Do to D3) are fed to in parallel to the inputs of a four-bit signature analyzer (Bo to B3), as shown in Fig. 6.7(a). When connected in this fashion the signature analyzer divides the gate inputs by the characteristic polynomial above resulting in the output values shown in Fig. 6.7(b) [30][8]. A FIFO is then placed in between the PRBS generator and signature analyzer components. If the FIFO overflows or starves (resulting in a communication penalty), the I/O sequence contained within the buffer is disturbed. As a consequence, the outputs of the signature analyzer will change in response when the offending portion of the sequence is read out on the next cycle (if the buffer is empty) or $f_{CLK(Tx)} * FIFO_{WIDTH}$ cycles later (if the buffer is full), where $FIFO_{WIDTH}$ is the maximum number of words that the buffer can store. As such, this is a useful construct for verifying flow control experiments within a FIFO.

6.4.3 *State Machine Logic for a Wagging Scheduler*

The controller responsible for scheduling the tasks in the parallel synchronizer can be modeled in VHDL via the use of finite state machine logic. The control device will have j states, where j is equivalent to the number of tasks, or wagging



(a) Schematic



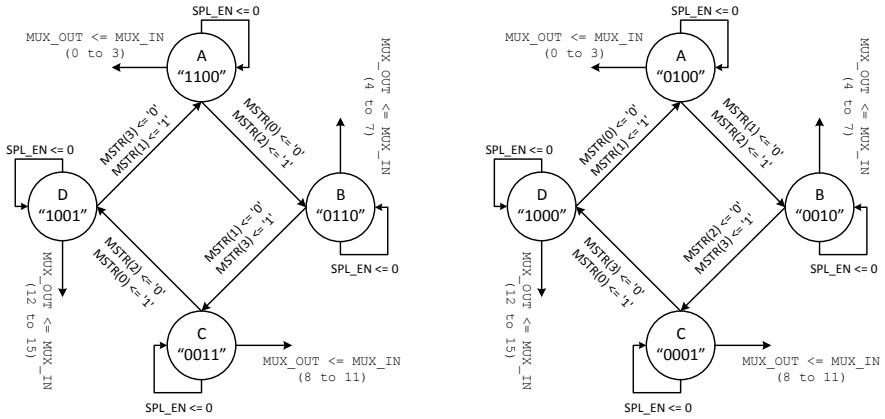
(b) Output response to PRBS test patterns

Figure 6.7: Four-bit signature analyzer.

level, desired in the synchronizer. The transitions at the CLK/EN inputs of the master and slave latches in the parallel array of SYNC elements can be modeled via binary state assignment, with logic 1 and 0 corresponding to the positive and negative transitions of each device.

The state table of the transitions in present at the CLK/EN inputs of the flip-flop array correspond to the entries in the truth table of either a j-bit Overbeck counter (i.e. a straight ring counter) when modeling control device which uses a one-hot code, or a j-bit Johnson (i.e., twisted ring counter) if modeling a synchronous clock signal with a 50% duty cycle (neglecting the $\langle 0,0,0,0\rangle$ state [60]. In the latter case, it should be noted that even though the $\langle 0,0,0,0\rangle$ state has no physical meaning it still needs to point to the initial state in the event of it being induced as a result of transient errors. Given that ring counters can be used to synthesize basic finite state machines on a Field Programmable Gate Array (FPGA) via the selective mapping of the ring outputs to a given logic function, and that the counters are also shift registers too, this is hardly surprising.

The state mapping of the four-bit wagging controller used in the subsequent experiments in of the chapter is depicted in Fig. 6.8(a) where C is the default state. MSTR(0 to 3) represents the transitions of the master latches in the SYNC element array. MUX_OUT indicates which n-bit pair is currently being passed



(a) Model for a synchronous control signal. (b) Model for a one-hot control signal.

Figure 6.8: State machine for four-bit wagging control device.

Table 6.1: Read Sequence Generated by Equation 6.1 per Cycle, where $i < j < 2i$

Cycle	$(j-i)_{cur}$	Check	ReadType;
1*	0	Pass(+1)	Serial
2	1	Fail(+0)	Parallel
3	1	Pass(+1)	Parallel
4	0	Fail(+0)	Serial

from the output of the serializing MUX depicted in Fig. 6.5 to the input of the FIFO buffer. Finally, SPL_EN acts an enable line for the controller. As for state transitions of the slave latches in Fig. 6.8(a), they are simply the inverse of the master latch transitions. If the code is one-hot, as in 6.8(b), the situation becomes more complex. At that point it is the decision of the designer whether to latch the slave device earlier or later in the duty cycle. To match the performance above, the slave latch should be take its result $\lfloor j/2 \rfloor$ cycles later than the master latch. However, to maximize the time available for synchronization of the individual slice and by proxy the MTBF of the synchronizer, the slave device should be latched 1 cycle after the master latch takes its sample, and the serializing MUX at the FIFO input should take its result 1 cycle before the master latch samples its next result.

6.5 VHDL SIMULATIONS

Having covered the topics of test pattern generation and state machine logic as it applies to this work, we can now carry out a behavioral analysis of the parallel FIFO synchronizer circuit which will shed some light on a couple of important properties. First, we will analyze the general behavior of the circuit itself using the

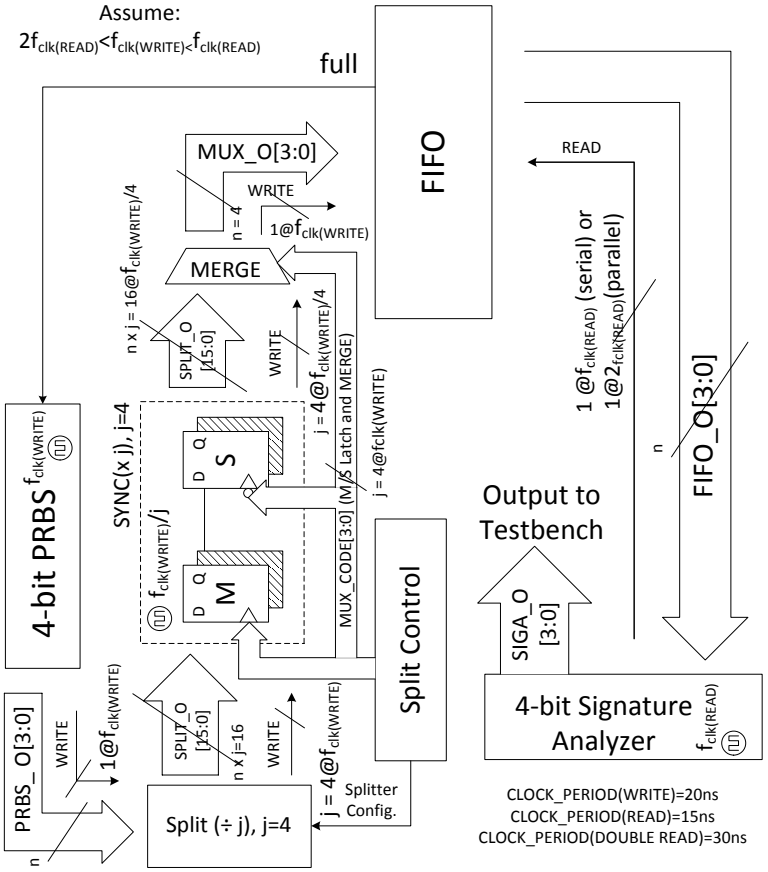


Figure 6.9: VHDL Experimental Setup.

test bench shown in Fig. 6.9. There are a few key points in this test bench that are worthy of note.

First, serial and parallel read operations at the output can be performed using an appropriately chosen clock signal, $f_{clk(READ}$, in lieu of either parallel memory access operations or serialization of the FIFO output. This is acceptable due to the established relationship between throughput and latency, as outlined by others such as Gene Ahmdahl in his paper discussing the effect of multiprocessing on a sequential computer program [4]. In terms of data processed over a given period of time, a serial circuit with a single output port operating at a given frequency processes the same amount of data as a parallel implementation with two output ports operating a half that frequency, assuming that data is always available and that the circuit can be completely parallelized [4]. Verification of the flow control algorithm stated in (29) also become easier as well due to fewer required test components.

Second, the number of data copies present in the parallel synchronizer (i.e., the wagging level of the synchronizer) remains fixed at j=4 over the course of the testing process, rather than setting j as a variable, as it was in the one-hot code of Section 3.4.5. However, as the control device present in Fig. 6.9 can be modeled as a special case of a four-bit one-hot counter, where the delays between the rising

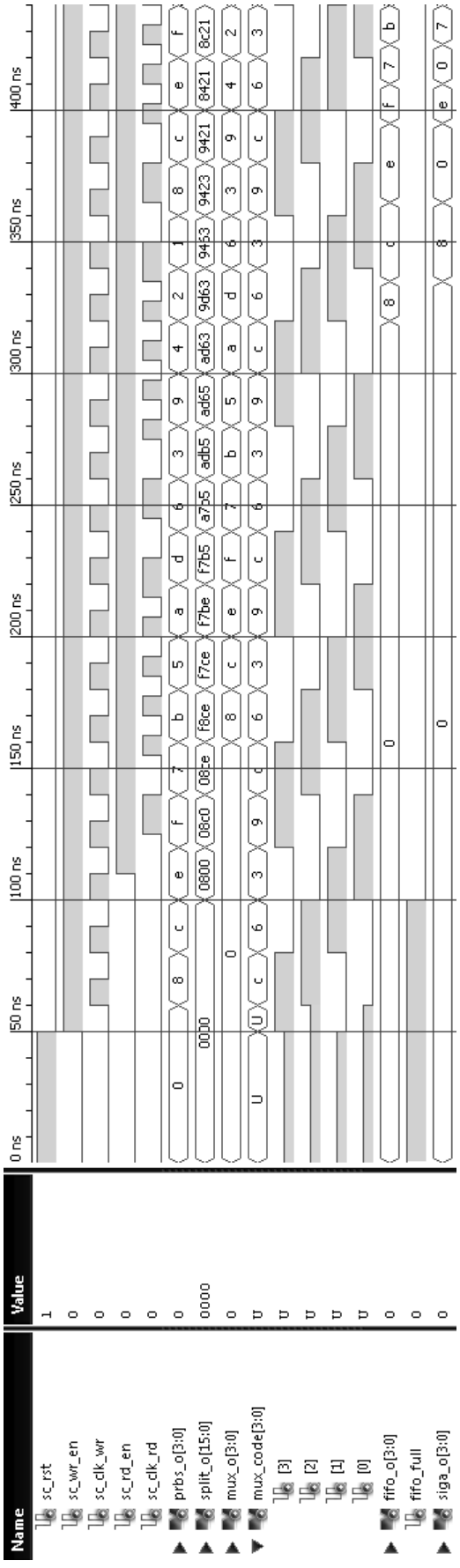


Figure 6.10: Transient response of the flow control in the complete self checking circuit.

Table 6.2: Effect of the Connectivity of the Serializing MUX on the Time Available for Synchronization in the Device

State $\langle$(MSB)3, 2, 1, 0(LSB)$\rangle$	Δt_{MSR} = 3 cycles	Δt_{MSR} = 2 cycles	Δt_{MSR} = 1 cycles	Δt_{MSR} = 0 cycles
A(1100)	0 to 3	4 to 7	8 to 11	12 to 15
B(0110)	4 to 7	8 to 11	12 to 15	0 to 3
(default) C(0011)	8 to 11	12 to 15	0 to 3	4 to 7
D(1001)	12 to 15	0 to 3	4 to 7	8 to 11

edges of successive ring outputs are merely scaled versions of the single gate delays seen previously, this is also not a major concern.

Our primary interest is assessing whether or not the flow control algorithm presented in section 6.2.1. prevents data accumulation from occurring within the FIFO buffer under ideal conditions, thereby minimizing the accumulation or starvation under non-ideal scenarios, and also to provide a commentary on the effects of wagging as it pertains a FIFO synchronizer. The clock periods chosen for the test bench were 30 ns and 20 ns for the read and write domains, respectively.The FIFO used for testing was automatically generated by the Xilinx LogiCORE FIFO generator [82]. It had a BUFFER WIDTH = 32 words and a WORD LENGTH = 4, and ISim was used to simulate the behavior of the VHDL netlist. Because $f_{clk(READ)} < f_{clk(WRITE)} < 2 * f_{clk(READ)}$, the basic conditions (29) are satisfied and the progression of the corresponding sequence of serial reads and writes asserted by $f_{clk(READ)}$ are as shown in in Table 6.1. The resulting transient analysis of the test bench, depicted in Fig. 6.10, reveals a couple useful results.

One, the signature patterns are correctly reproduced, and the buffer neither starves, nor gets full using the flow control described above. This was verified by checking the test patterns at the output of the signature analyzer (SIG_O). The test patterns remain the same under ideal conditions, even long after (50µs) the unmodified FIFO has become full and changed its test patterns. For reference purposes, the unmodified FIFO becomes full at around 3.3 µs.

Two, if a MUX is used to serialize the inputs of the FIFO using the state diagram of Fig. 6.8(a), then the connectivity of the *j* n-bit inputs to the MUX is very important, where both *j* and *n* are equal to 4 in this case. Because the sampling of the MUX outputs MUX_O is controlled via the state machine in 6.8(a), the amount of time available for synchronization gained, Δt_{MSR}, is dependent on the connectivity of these lines, as depicted in Table 6.2. To put it more simply, if $\Delta t_{MSR} = 0$ then nothing has been gained through the use of wagging, whereas if $\Delta t_{MSR} = j - 1$ then *j-1* extra cycles have been gained for metastability recovery.

7 CONCLUSIONS

In closing, synchronization remains a key SoC design issue in modern technologies. As the number of operating points under consideration increases, specifications which are capable of altering key parameters such as the time available for synchronization (and MTBF) in response to input from the user/system become desirable. If a combination of parallelism and scheduling, referred to as wagging, is utilized, then schedulers can be constructed for synchronizer designs which are capable of pooling the gain-bandwidth products of their composite devices.

In this work, we explore the ways in which the areas of graph theory and reconfigurable hardware design can be applied to generate both combinational and sequential scheduler designs, which satisfy the behavior requirement above. Further to this point, this work illustrates that such a scheduler is primarily comprised of an interrupt subsystem, and a reconfigurable token ring. This thesis explores how both of these components can be controlled in absence of a clock signal, as well as the design challenges inherent to each part.

Synchronizers may also require flow control, especially if the reading and writing operations are decoupled from each other via the use of a FIFO buffer. Such structures incur penalties if the data rates of both sides are not well matched [15]. As this can often be the case in designs which incorporate both reconfiguration and parallelism, it is sometimes necessary to implement aspects of both serial and parallel data transmission to minimize this mismatch.

7.1 MAIN CONTRIBUTIONS

• An interrupt subsystem for a reconfigurable control circuit suitable for use in a wagging synchronizer was simulated in a UMC 90nm technology via CADENCE, which demonstrated a power consumption of for each interrupt module, which was found during active operation to range from 64.7 to 90μW across process corners at a temperature of 27°C.

• Token ring designs suitable for use as schedulers in the reconfigurable control circuit mentioned above were explored in Chapters 3 and 5, and simulated using CADENCE with the following results:

o At a length of 19 elements (duty cycle = 1ns), a token ring based on fast David Cells was found to have 84.23% of the cycle time available for synchronization.

o At a length of 12 elements (duty cycle = 1ns), a token ring based on Muller Pipeline was found to have 78.46% of the cycle time available for synchronization.

o At a length of 37 elements (duty cycle = 1ns), a token ring based on Ring Oscillator was found to have 50.26% of the cycle time available for synchronization.

o At a length of 8 elements, a token ring based on Chordal graphs was found to have 692.4 ps available for synchronization.

• A VHDL flow control experiment for a 4-way wagging synchronizer was performed in chapter 6 using a Xilinx ISE test bench, and the time available for synchronization (t_{MSR}) was shown to be variable from 0 to 3 cycles. Additionally, the FIFO buffer did not accumulate excess data values or starve, as this was an ideal case (in real life data accumulation or starvation would occur, but the rate would be minimized).

7.2 FUTURE WORK

This work leaves many open venues. First and foremost, among them relates to whether or not the parallel synchronization methods used throughout the body of this work can be adapted to further enhance other synchronization methods. Prior work on predictive synchronizers indicated that there is an upper limit to the applicability of the method based on the fundamental frequency of the conflict detector circuit [28]. It is within reason that if parallelism/wagging is employed, this fundamental limit can be overcome at the cost of power, area, and complexity.

On a more abstract level, the concept of utilizing cyclic embedded graphs to create more flexible specifications (i.e., specifications which encompass a range of parameters rather than just one) and using interleaved CSC signals as a fundamental aspect of the control theory in the STG representation of the same, has many potential applications. Perhaps specifications which schedule events using graph theory can yield benefits over synchronous methods relying on local clocks and watchdog timers to handle event-driven transmissions under certain conditions.

Further to the point of scheduling, creating graphical specifications where the directedness of each node can be controlled based on externally generated user stimuli can be used to meet operating points which fall outside the traditional power, speed, and area metrics which are common to many designs. Case in point, 3D integrated circuit designs have issues with thermal extraction. Creating specifications as listed above can be used to construct protocols, which could be useful in meeting thermal constraints, given the development of proper tools to analyze such behavior. And finally, using parallelism to assist in the creation of interfaces which link together technologies with vastly different gain-bandwidth products remains a promising avenue of research.

Part II

APPENDIX

A APPENDIX

A.1 INTMUX BOOLEAN LOGIC MINIMIZATION

FORWARD DIRECTION

1. $\langle 0,0,0,0,0,0,0,1\rangle$, INTERRUPT IS ACTIVATED (FWD3) (↑ FLAG)
2. $\langle 0,0,0,0,1,0,0,1\rangle$, TOKEN IS STOPPED AT TKN_RQ1 (↑ RC_RQ1)
3. $\langle 1,0,0,0,1,0,0,1\rangle$, CHANGE FROM CFG#0 to CFG#1 (↑ RC_ACK1)
4. $\langle 1,0,0,0,0,0,0,1\rangle$, TOKEN IS GRANTED ALONG TKN_GR1 (↓ RC_RQ1)
5. $\langle 1,0,0,0,0,1,0,1\rangle$, TOKEN IS STOPPED AT TKN_RQ2 (↑ RC_RQ2)
6. $\langle 0,0,0,0,0,1,0,1\rangle$, ACK FROM INTERRUPT #1 COMPLETES (↓ RC_ACK1)
7. $\langle 0,1,0,0,0,1,0,1\rangle$, CHANGE FROM CFG#1 to CFG#2 (↑ RC_ACK2)
8. $\langle 0,1,0,0,0,0,0,1\rangle$, TOKEN IS GRANTED ALONG TKN_GR2 (↓ RC_RQ2)
9. $\langle 0,1,0,0,0,0,1,1\rangle$, TOKEN IS STOPPED AT TKN_RQ3* (↑ RC_RQ3)
10. $\langle 0,0,0,0,0,0,1,1\rangle$, ACK FROM INTERRUPT #2 COMPLETES (↓ RC_ACK2)
11. $\langle 0,0,0,1,0,0,1,1\rangle$, WRITE "1" TO MEM (↑ MEM)
12. $\langle 0,0,1,1,0,0,1,1\rangle$, CHANGE FROM CFG#2 to CFG#3 (↑ RC_ACK3)
13. $\langle 0,0,1,1,0,0,0,1\rangle$, MEMORY BECOMES OPAQUE** (↓ RC_RQ3)
14. $\langle 0,0,0,1,0,0,0,1\rangle$, INTERRUPT SHUTS DOWN (CHANGE TO CFG#4) (↓ RC_ACK3)

REVERSE DIRECTION

1. $\langle 0,0,0,1,0,0,0,0\rangle$, INTERRUPT IS ACTIVATED (REV3) (↓ FLAG)
2. $\langle 0,0,0,1,0,0,1,0\rangle$, TOKEN IS STOPPED AT TKN_RQ3 (↑ RC_RQ3)
3. $\langle 0,0,1,1,0,0,1,0\rangle$, CHANGE FROM CFG#4 to CFG#3 (↑ RC_ACK3)
4. $\langle 0,0,1,1,0,0,0,0\rangle$, TOKEN IS GRANTED ALONG TKN_GR3 (↓ RC_RQ3)

(5) ⟨0,0,1,1,0,1,0,0⟩, TOKEN IS STOPPED AT TKN_RQ2 (↑ RC_RQ2)

(6) ⟨0,0,0,1,0,1,0,0⟩, ACK FROM INTERRUPT #3 COMPLETES (↓ RC_ACK3)

(7) ⟨0,1,0,1,0,1,0,0⟩, CHANGE FROM CFG#3 to CFG#2 (↑ RC_ACK2)

(8) ⟨0,1,0,1,0,0,0,0⟩, TOKEN IS GRANTED ALONG TKN_GR2 (↓ RC_RQ2)

(9) ⟨0,1,0,1,1,0,0,0⟩, TOKEN IS STOPPED AT TKN_RQ3* (↑ RC_RQ1)

(10) ⟨0,0,0,1,1,0,0,0⟩, ACK FROM INTERRUPT #2 COMPLETES (↓ RC_ACK2)

(11) ⟨0,0,0,0,1,0,0,0⟩, WRITE "0" TO MEM (↓ MEM)

(12) ⟨1,0,0,0,1,0,0,0⟩, CHANGE FROM CFG#2 to CFG#1 (↑ RC_ACK1)

(13) ⟨1,0,0,0,0,0,0,0⟩, MEMORY BECOMES OPAQUE** (↓ RC_RQ1)

(14) ⟨0,0,0,0,0,0,0,0⟩, INTERRUPT SHUTS DOWN (CHANGE TO CFG#0) (↓ RC_ACK1)

A.2 CADENCE PWL WAVEFORM GENERATION EXAMPLE

Listing 1: MATLAB .PWL Cadence Vector File Generator Core

```
%MATLAB .PWL Cadence Vector File Generator
%Created by Ghaith Tarawneh & Jim Guido
%Last modified on 2/9/2011
%Version 2.0
%This program is designed to generate valid SPECTRE input vector files for
%independent piecewise linear voltage sources.

max_pulse_width = 1e-9;
pulse_width_var = 1e-10;
num_pulse_steps = max_pulse_width/pulse_width_var; %MUST BE RATIONAL INTEGER
    RESULT
rise_time = 5e-12;
fall_time = 5e-12;
file_template = 'bit%i_%ips_ud1.pwl';
high_voltage = 1;
low_voltage = 0;

DATA GOES HERE

for bit=1:columns
    for pulse_step=1:num_pulse_steps
        pulse_width = (max_pulse_width) - (pulse_width_var * (pulse_step-1));
        filename_pulse = round(pulse_width * 1e12);
        filename = sprintf(file_template, bit, filename_pulse);
        fid = fopen(filename, 'w');
        t = 0;
        for i=1:rows
            v = sequence(i, bit);
            if (v)
                fprintf(fid, '%1.6e %1.3e\n', t, high_voltage);
                t = t + (pulse_width - rise_time - fall_time) + rise_time;
                fprintf(fid, '%1.6e %1.3e\n', t, high_voltage);
                t = t + fall_time;
            else
                fprintf(fid, '%1.6e %1.3e\n', t, low_voltage);
                t = t + (pulse_width - rise_time - fall_time) + rise_time;
                fprintf(fid, '%1.6e %1.3e\n', t, low_voltage);
                t = t + fall_time;
            end
        end
        fclose(fid);
    end
end
```

Listing 2: MATLAB .PWL Cadence Vector File Generator Sample Input

```
% bit1 = UDATA0(1) (LSB)
% bit2 = UDATA1(1)
% bit3 = UDATA2(1)
% bit4 = UDATA3(1)
% bit5 = UDATA4(1)
% bit6 = UDATA5(1)
% bit7 = UDATA6(1)
% bit8 = UDATA7(1)
% bit9 = UDATA8(1)
%bit10 = UDATA9(1)
%bit11 = UDATA10(1)
%bit12 = UDATA11(1)
%bit13 = UDATA12(1)
%bit14 = UDATA13(1)
%bit15 = UDATA14(1)
%bit16 = UDATA15(1) (MSB)

%BLANK DATA FOR MEM_INJECT

 f1 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];
 f2 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];

%UNENCODED RECONFIG DATA (EVEN DEVICE)

 f3 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];
 f4 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];
 f5 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];
 f6 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];
 f7 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];
 f8 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];
 f9 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];
f10 = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0];

f11 = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0];
f12 = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0];
f13 = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0];
f14 = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0];
f15 = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0];
f16 = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0];
f17 = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0];
f18 = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0];

sequence = [ f1;  f2;  f3;  f4;  f5;  f6;  f7;  f8;  f9; f10;
            f11; f12; f13; f14; f15; f16; f17; f18

[rows,columns] = size(sequence);
```

BIBLIOGRAPHY

[1] V.S.I. Alliance. System-Level Interface Behavioral Documentation Standard (SLD 1 1.0), 2000. URL http://www.vsi.org.

[2] Mohammed Alshaikh, David Kinniment, and Alexandre Yakovlev. A synchronizer design based on wagging. *IEEE International Conference on Microelectronics (ICM '10)*, pages 415–418, December 2010.

[3] Brian Alspach. Cycles of each length in regular tournaments. *Canad. Math. Bull*, 10(2), 1967.

[4] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[5] Bruce W. Arden and Hikyu Lee. Analysis of chordal ring network. *IEEE Transactions on Computers*, 100(4):291–295, April 1981.

[6] Salomon Beer, Ran Ginosar, Michael Priel, R. Dobkin, and Avinoam Kolodny. The Devolution of synchronizers. *IEEE Symposium on Asynchronous Circuits and Systems (ASYNC '10)*, pages 94–103, May 2010.

[7] Edith Beigne, Fabien Clermidy, Pascal Vivet, Alain Clouard, and Marc Renaudin. An asynchronous NOC architecture providing low latency service and its multi-level design framework. *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC '05)*, pages 54–63, March 2005.

[8] Dilip K Bhavsar. System for polynomial division self-testing of digital networks, February 1985. US Patent 4,498,172.

[9] J.A Bondy. Pancyclic graphs i. *Journal of Combinatorial Theory, Series B*, 11(1):80 – 84, 1971. ISSN 0095-8956. doi: http://dx.doi.org/10.1016/0095-8956(71)90016-5. URL http://www.sciencedirect.com/science/article/pii/0095895671900165.

[10] David S. Bormann and Peter Y.K. Cheung. Asynchronous wrapper for heterogeneous systems. *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '97)*, pages 307–314, October 1997.

[11] Charlie Brej. Wagging Logic: Implicit Parallelism Extraction Using Asynchronous Methodologies. *IEEE International Conference on Application of Concurrency to System Design (ACSD '10)*, pages 35–44, June 2010.

[12] Jean-Michel Chabloz. *Globally-Ratiochronous, Locally-Synchronous Systems*. PhD thesis, KTH, School of Information and Communication Technology, 2012.

[13] Thomas J. Chaney and Charles E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, 100(4):421–422, April 1973.

[14] Daniel M. Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, Dept. of Computer Science, 1984.

[15] Tiberiu Chelcea and Steven M. Nowick. Robust interfaces for mixed-timing systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(8): 857–873, August 2004.

[16] Tam-Anh Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1987.

[17] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, 80(3):315–325, 1997.

[18] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*, volume 8 of *Springer Series in Advanced Microelectronics*. Springer, Berlin, DE, 2002.

[19] J.P. Costas. Synchronous Communications. *Proceedings of the IRE*, 44(12):1713–1718, Dec 1956. ISSN 0096-8390. doi: 10.1109/JRPROC.1956.275063.

[20] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. *Proceedings of the IEEE Design Automation Conference (DAC '01)*, pages 684–689, 2001.

[21] Rene David. Modular Design of Asynchronous Circuits Defined by Graphs. *IEEE Transactions on Computers*, C-26(8):727–737, Aug 1977.

[22] Sanjay Dhar, Mark A. Franklin, and D.F. Wann. Reduction of clock delays in VLSI structures. *IEEE International Conference on Computer Design*, pages 778–783, 1984.

[23] G.A. Dirac. On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der UniversitÃ€t Hamburg*, 25(1-2):71–76, 1961. ISSN 0025-5858. doi: 10.1007/BF02992776. URL http://dx.doi.org/10.1007/BF02992776.

[24] Rastislav Dobkin and Ran Ginosar. Zero latency synchronizers using four and two phase protocols. Technical Report CCIT TR642, VLSI Systems Research Center, Technion, IL, October 2007.

[25] Jo Ebergen. Squaring the fifo in gasp. *IEEE International Symposium on Asynchronus Circuits and Systems (ASYNC '01)*, pages 194–205, March 2001.

[26] Boaz Eitan and D. Frohman-Bentchkowsky. Hot-electron injection into the oxide in n-channel MOS devices. *IEEE Transactions on Electron Devices*, 28(3): 328–340, March 1981.

[27] Fausto Fantini. Reliability problems with VLSI. *Microelectronics Reliability*, 24 (2):275–296, 1984.

[28] Uri Frank, Tsachy Kapshitz, and Ran Ginosar. A predictive synchronizer for periodic clock domains. *Formal Methods in System Design*, 28(2):171–186, March 2006.

[29] Eby G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5):665–692, May 2001.

[30] Robert A. Frohwerk. Signature analysis: A new digital field service method. *Hewlett-Packard Journal*, 28(9):2–8, 1977.

[31] Stephen B. Furber, Paul Day, Jim D. Garside, Nigel C. Paver, and John V. Woods. AMULET1: a micropipelined ARM. *Compcon Spring'94, Digest of Papers*, pages 476–485, February/March 1994.

[32] Stephen B. Furber, Jim D. Garside, and David A. Gilbert. AMULET3: A high-performance self-timed ARM microprocessor. *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '98)*, pages 247–252, October 1998.

[33] Stephen B. Furber, James D. Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, and Nigel C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, February 1999.

[34] Stanislavs Golubcovs. *Multi-resource approach to asynchronous SoC: design and tool support*. PhD thesis, University of Newcastle Upon Tyne, 2011.

[35] Frank Gray. Pulse code communication, March 1953. US Patent 2,632,058.

[36] Mark R. Greenstreet. Implementing a STARI chip. *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '95)*, pages 38–43, October 1995.

[37] Mark Russell Greenstreet. *Stari: a technique for high-bandwidth communication*. PhD thesis, Princeton University, Dept. of Computer Science, 1993.

[38] Frank Harary and Leo Moser. The theory of round robin tournaments. *The American Mathematical Monthly*, 73(3):pp. 231–246, 1966. ISSN 00029890. URL http://www.jstor.org/stable/2315334.

[39] C.S. Hau-Riege and C.V. Thompson. Electromigration in Cu interconnects with very different grain structures. *Applied Physics Letters*, 78(22):3451–3453, January 2001.

[40] Jens U. Horstmann, Hans W. Eichel, and Robert L. Coates. Metastability behavior of CMOS ASIC flip-flops in theory and test. *IEEE Journal of Solid-State Circuits*, 24(1):146–157, February 1989.

[41] Jerry Jex and Charles Dike. A fast resolving BiNMOS synchronizer for parallel processor interconnect. *IEEE Journal of Solid-State Circuits*, 30(2):133–139, February 1995.

[42] Ian W. Jones, Suwen Yang, and Mark Greenstreet. Synchronizer behavior and analysis. *IEEE Symposium on Asynchronous Circuits and Systems (ASYNC '09)*, pages 117–126, May 2009.

[43] Suk-Jin Kim, Jeong-Gun Lee, and Kiseon Kim. A parallel flop synchronizer for bridging asynchronous clock domains. *IEEE Asia-Pacific Conference on Advanced System Integrated Circuits*, pages 184–187, August 2004.

[44] David J Kinniment. *Synchronization and Arbitration in Digital Systems*. John Wiley & Sons, 2008.

[45] David J. Kinniment and D.B.G. Edwards. Circuit technology in a large computer system. *Radio and Electronic Engineer*, 43(7):435–441, July 1973.

[46] Gideon Langholz, Abraham Kandel, and Joe L. Mott. *Foundations of digital logic design*, volume 1. World Scientific, 1998.

[47] Thomas H Lee. *The design of CMOS radio-frequency integrated circuits*. Cambridge University Press, 2004.

[48] M.W. Levi. CMOS is most testable. *International Test Conference*, pages 217–220, 1981.

[49] Kuan-Jen Lin, Jih-Wen Kuo, and Chen-Shang Lin. Direct synthesis of hazard-free asynchronous circuits from STGs based on lock relation and MG-decomposition approach. In *Proceedings of 1994 European Design and Test Conference EDAC-ETC-EUROASIC*, pages 178–183, Feb 1994. doi: 10.1109/EDTC.1994.326879.

[50] Alain J. Martin. On Seitz' Arbiter. Technical Report 5212:TR:86, California Institute of Technology, Dept. of Computer Science, Pasadena, CA, USA, 1986.

[51] Alain J. Martin. Synthesis of asynchronous VLSI circuits. Technical Report CALTECH-CS-TR-93-28, California Institute of Technology, Dept. of Computer Science, Pasadena, CA, USA, March 2000.

[52] K. Meena. *Principles of Digital Electronics*. Prentice-Hall Of India Pvt. Limited, Connaught Circus, New Delhi, IN, 2009.

[53] Ivan Miro-Panades, Fabien Clermidy, Pascal Vivet, and Alain Greiner. Physical Implementation of the DSPIN Network-on-Chip in the FAUST Architecture. *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 139–148, April 2008.

[54] JW Moon. On subtournaments of a tournament. *Canad. Math. Bull*, 9(3):297–301, 1966.

[55] Simon Moore, George Taylor, Robert Mullins, and Peter Robinson. Point to point GALS interconnect. *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC '02)*, pages 69–75, April 2002.

[56] David E. Muller and W. Scott Bartkey. A theory of asynchronous circuits. *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243, 1959.

[57] Robert Mullins and Simon Moore. Demystifying data-driven and pausible clocking schemes. *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC '07)*, pages 175–185, March 2007.

[58] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[59] Samuel D. Naffziger, Glenn Colon-Bonet, Timothy Fischer, Reid Riedlinger, Thomas J. Sullivan, and Tom Grutkowski. The implementation of the Itanium 2 microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1448–1460, November 2002.

[60] Wilcox P Overbeck. Vacuum tube cycle counter, February 1939. US Patent 2,147,918.

[61] Suhas Shrikrishna Patil. *Synchronizers and arbiters*. Massachusetts Institute of Technology, project MAC, 1973.

[62] James L. Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, September 1977.

[63] Carl A. Petri. *Communication with Automata, Supplement 1 to Technical Report RADC-TR-65-337, NY, 1965. Translation by CF Greene of Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.

[64] William Wharton Plummer. Asynchronous arbiters. *IEEE Transactions on Computers*, 100(1):37–42, January 1972.

[65] Ivan Poliakov, Danil Sokolov, and Andrey Mokhov. Workcraft: A Static Data Flow Structure Editing, Visualisation and Analysis Tool. *International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN '07)*, pages 505–514, 2007.

[66] Jan M. Rabaey, Anantha P. Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, Upper Saddle River, NJ, USA, 2003.

[67] Rochit Rajsuman. Iddq testing for CMOS VLSI. *Proceedings of the IEEE*, 88(4): 544–568, April 2000.

[68] G. Ribes, J. Mitard, M. Denais, S. Bruyere, F. Monsieur, C. Parthasarathy, E. Vincent, and Ge Ghibaudo. Review on high-k dielectrics reliability issues. *IEEE Transactions on Device and Materials Reliability*, 5(1):5–19, March 2005.

[69] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits. *Proceedings of the IEEE*, 91(2):305–327, Feb 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.808156.

[70] Charles L. Seitz. System timing. *Introduction to VLSI systems*, pages 218–262, 1980.

[71] Charles L. Seitz. Ideas about arbiters. *Lambda*, 1(1):10–14, 1980.

[72] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. *IEEE International Conference on Dependable Systems and Networks (DSN '02)*, pages 389–398, 2002.

[73] Danil Sokolov. *Automated synthesis of asynchronous circuits using direct mapping for control and data paths*. PhD thesis, University of Newcastle Upon Tyne, Dept. of Electrical, Electronic and Computer Engineering, 2006.

[74] IEEE Standard. VHDL Language Reference Manual. *IEEE Std*, pages 1076–1987, 1988.

[75] James Stewart. *Calculus: Early Transcendentals*. Brooks/Cole, Belmont, CA, USA, 5th edition, 2003.

[76] Ivan E. Sutherland and Robert F. Sproull. Logical effort: Designing for speed on the back of an envelope. In *Proceedings of the University California/Santa Cruz Conference on Advanced Research in VLSI*, pages 1–16. MIT Press, 1991.

[77] Kees van Berkel. *Handshake circuits: An intermediary between communicating processes and VLSI*. PhD thesis, Technische Univ., Eindhoven (Netherlands)., 1992.

[78] Kees van Berkel, Mark B. Josephs, and Steven M. Nowick. Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.

[79] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, et al. An 80-Tile Sub-100-W teraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, January 2008.

[80] Harry JM Veendrick. The behaviour of flip-flops used as synchronizers and prediction of their failure rate. *IEEE Journal of Solid-State Circuits*, 15(2):169–176, 1980.

[81] Xilinx. UG070 Virtex-4 FPGA User Guide, December 2008. URL http://www.xilinx.com/support/documentation/user_guides/ug070.pdf.

[82] Xilinx. LogiCORE IP FIFO Generator v9.3 Product Guide, December 2012. URL http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v9_3/pg057-fifo-generator.pdf.

[83] Jun Zhou, David Kinniment, Gordon Russell, and Alexandre Yakovlev. A robust synchronizer. *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI '06)*, pages 442–443, March 2006.