ALGORITHMS AND METHODOLOGY TO DESIGN ASYNCHRONOUS CIRCUITS USING SYNCHRONOUS CAD TOOLS AND FLOWS

by

Vikas S. Vij

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

The University of Utah

December 2013

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

This dissertation of **Vikas S. Vij**

has been approved by the following supervisory committee members:

Kenneth S. Stevens, Chair **11/5/2013**
Date Approved

Chris Myers, Member **10/31/2013**
Date Approved

Erik Brunvand, Member **11/5/2013**
Date Approved

Priyank Kalla, Member **10/31/2013**
Date Approved

Christos Sotiriou, Member ________
Date Approved

and by **Gianluca Lazzi**, Chair of
the Department of **Electrical and Computer Engineering**

and by **David B. Kieda, Dean of the Graduate School.**

ABSTRACT

Asynchronous design has a very promising potential even though it has largely received a cold reception from industry. Part of this reluctance has been due to the necessity of custom design languages and *computer aided design* (CAD) flows to design, optimize, and validate asynchronous modules and systems. Next generation asynchronous flows should support modern programming languages (e.g., Verilog) and *application specific integrated circuits* (ASIC) CAD tools. They also have to support multifrequency designs with mixed synchronous (clocked) and asynchronous (unclocked) designs. This work presents a novel *relative timing* (RT) based methodology for generating multifrequency designs using synchronous CAD tools and flows.

Synchronous CAD tools must be constrained for them to work with asynchronous circuits. Identification of these constraints and characterization flow to automatically derive the constraints is presented. The effect of the constraints on the designs and the way they are handled by the synchronous CAD tools are analyzed and reported in this work.

The automation of the generation of asynchronous design templates and also the constraint generation is an important problem. Algorithms for automation of reset addition to asynchronous circuits and power and/or performance optimizations applied to the circuits using logical effort are explored thus filling an important hole in the automation flow. Constraints representing cyclic asynchronous circuits as *directed acyclic graphs* (DAGs) to the CAD tools is necessary for applying synchronous CAD optimizations like sizing, path delay optimizations and also using *static timing analysis* (STA) on these circuits. A thorough investigation for the requirements of cycle cutting while preserving timing paths is presented with an algorithm to automate the process of generating them.

A large set of designs for 4 phase handshake protocol circuit implementations with early and late data validity are characterized for area, power and performance. Benchmark circuits with automated scripts to generate various configurations for better understanding of the designs are proposed and analyzed. Extension to the methodology like addition of scan insertion using *automatic test pattern generation* (ATPG) tools to add testability of datapath in bundled data asynchronous circuit implementations and timing closure approaches are also described. Energy, area, and performance of purely asynchronous circuits and circuits with mixed synchronous and asynchronous blocks are explored. Results indicate the benefits that can be derived by generating circuits with asynchronous components using this methodology.

This dissertation is dedicated to my parents, my brother and his family who have always motivated me and supported me.

CONTENTS

CHAPTERS

APPENDICES

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. Ken Stevens for his guidance, understanding, patience, and most importantly, his assistance through my graduate studies at the University of Utah. His mentorship, expertise and humor provided me with the motivation to pursue my doctoral research. His support and encouragement were crucial through this whole experience. I would like to thank Dr. Chris Myers, Dr. Erik Brunvand, Dr. Priyank Kalla, and Dr. Christos Sotiriou for their constructive reviews of my work and for their constant assistance through my graduate studies.

I wish to thank my friends in the research group: William Lee, Shomit Das, Raghu Prasad Gudla, Krishnaji Desai, Dan Gebhardt and Junbok You for having all those constructive discussions and presentations to learn from and teach each other. Their reviews and critical comments on my manuscripts and work helped improve the quality immensely.

I would like to thank the Electrical and Computer Engineering Department at the University of Utah, especially Lori Sather for guiding me through the administrative stuff at the University and always being there to help. The long stay in Salt Lake City would not have been fun without the presence and help of my friends. Special thanks to Sai and Hemang for always being there to motivate me. Last but not the least my parents, my brother, my sister-in-law and my nephews for sticking by me through all this time and making this whole journey a fun filled one.

CHAPTER 1

INTRODUCTION

Integrated circuits continue to grow in performance and transistor count, with current designs exceeding a few billion transistors. Distributing the clock to the entire chip in such designs poses significant design effort and energy. Power consumed by the chip increases from generation to generation because total chip area remains constant or grows. This results in power hungry clock drivers that require high slew rates in order to distribute a high frequency clock with limited skew. This has resulted in designs where nearly 40 percent of the total on chip power consumption is due to the clock generation and distribution network [2, 3]. Earlier, performance and area were the metrics around which VLSI designers would build and optimize their chips, but in the last decade power has arguably become the most important metric.

The presence of a large number of transistors has also resulted in different approaches to design which involve different *intellectual property* (IP) blocks characterized and optimized for a specific frequency of operation. These IP blocks being integrated together to create a chip with multiple clock frequency domains is called multisynchronous design. The overhead of clock domain crossing and also the increase in design complexity of the system are becoming a big issue in terms of power and also performance.

Asynchronous circuits are a potential solution to all these problems, as they switch only to do useful work. The frequency of operation of asynchronous circuits is dynamic and is dependent on the amount of logic in the pipeline as well as on the operating frequency of the pipelines adjacent to it. Since there is no global clock and all the communication is local, there is no need for power hungry low skew drivers. These circuits are based on handshake protocols, which enhance the modularity and composability of the designs and thus assist in supporting multiple frequency designs without the need for synchronization. By operating the asynchronous system at frequencies that best optimize the power and performance of each individual asynchronous modules an overall better design is achieved. One of the better silicon examples of such an architecture is the Pentium Processor front end, which operates at three frequencies: 720MHz for instruction decode, 3.6GHz for instruction selection, and 900MHz for instruction steering and issue [4]. This design was fabricated in the same foundry as its commercial counterpart, and achieved an 17 fold improvement in $e\tau^2$ (energy delay squared product).

If the timing models employed in the clocked design style can be leveraged by general multifrequency design, the same tools, languages, and flows can be used with all methods of timing for design and architecture optimization. Relative timing (RT) does just that; it bridges the gap between the incompatible timing used by unclocked design styles by expressing the timing in a form used by commercial clocked *electronic design automation* (EDA) tools. Once timing compatibility is achieved, common design languages, standard cell libraries, and tool flows become common to all design styles. This compatibility enhances productivity, reduces the cost of adopting multifrequency design methodologies and results in power and performance advantages. A flow based on RT to generate multifrequency designs using commercial clocked EDA tools is described in this dissertation.

1.1 Related Work

The convergence of asynchronous design approaches and the synchronous (clocked) *computer aided design* (CAD) tools and flows has been an actively researched topic. Based on different timing models used for the asynchronous design, different approaches are proposed to use the synchronous CAD tools and flows either partially or fully.

Flows related to approaches that use *delay insensitive* (DI) encoding use only the clocked synthesis CAD tool like *Synopsys Design Compiler* (DC) [5]. The synthesized design output is then mapped to specific DI gate implementations, which preserve the hazard properties of the design. Since timing is inherent in the DI systems, there is no requirement to specify timing constraints for functional correctness. The *NULL Convention Logic* (NCL) designs by Theseus Logic ([6]), the Proteus flow by University of Southern California and Fulcrum Microsystems ([7]) and the phased logic ([8]) approach to design circuit using the *level encoded two-phase dual rail* (LEDR) encoding follow this partial use of synchronous CAD tools and flows.

Commercial companies like Silistix working on DI design also have attempted to address this problem for *network-on-chip* (NoC) designs with their toolflow, which requires a precharacterized technology library [9]. The library contains adapters for their IP interface protocols and hard macroblocks for the CHAIN interconnect ([10]), which is used to connect the system blocks. The toolflow is named CHAINarchitect and it converts the network specified in a custom language called the *connect specification language* (CSL) into an on-chip network implementation. The benefits of this toolflow are its completeness in terms of all the general CAD flow steps like place and route, testing and *static timing analysis* (STA). This flow applies only to NoC designs developed using the technology library precharacterized by Silistix. Hence, it is not a general design flow. Also, the synchronous CAD tool optimizations are not applied because of the use of precharacterized hard macroblocks.

Desynchronization approach is the most complete existing method for generating an asynchronous bundled data design using the synchronous CAD flows [11, 12, 13, 14, 15]. It also uses standard library cells and *hardware description languages* (HDLs) to specify the circuit. This approach is built on the marked graph theory and hence proves the liveness, safeness and flow equivalence properties of the circuits. It accomplishes a direct mapping of the synchronous design into an asynchronous equivalent by removing the clock network and replacing it with an asynchronous handshake network. Postsilicon numbers for the ASPIDA DLX processor and a DES core were published for comparison of the asynchronous design with its synchronous counterpart [11, 12, 13, 14, 15]. Desynchronization provides significant benefits in *electromagnetic interference* (EMI) improvements and shorter design cycles. However, design results show little or no power improvement over the initial clocked design. The asynchronous design operates at average case speed as compared to worst case speed for the synchronous design, thus resulting in performance benefits for the fabricated desynchronized DLX processor. The base of the timing constraint specification for this approach is first to divide each flip-flop into a pair of latches which are individually controlled by a handshake controller. Then, a virtual clock is created to enable each latch for timing. This approach restricts the design to the clock paradigm, thus preventing the application of asynchronous architectural and design optimizations, which are important to gain the benefits similar to the Pentium front-end example. The benefit of this approach is that the representation is completely like the clocked definition.

The application of synchronous synthesis tools for high-level timed asynchronous bundled-data design has also been investigated by using a channel-level VHDL code [16]. The circuit implementation was shown on a *field programmable gate array* (FPGA). The asynchronous control circuit for this implementation is derived using the ATACS tool [17], while the FPGA synthesis tool synthesizes the datapath. The benefit of this approach is the use of an HDL language to generate the asynchronous circuit and the concept of utilizing synchronous CAD tools to synthesize the combinational logic in the datapath. The timing and sizing algorithms of the synchronous CAD tools are not used to optimize the asynchronous design, hence delay elements are created by manually adding buffers based on the delay requirement.

Another approach that addresses generation of asynchronous bundled data as well as *quasi delay insensitive* (QDI) Micropipeline designs is the Weaver flow [18]. It modifies the library to make it compatible with DC, thus enabling synthesis of asynchronous circuits. The application of this approach is presented for deterministic as well as data dependent token propagation which enables its application to a large set of asynchronous circuits. The major drawback of this approach, however, is that it requires modifying the standard cell libraries to make it compatible with the flow. Hence, knowledge of library characterization and modification is required to derive the benefits.

A detailed study of the limitations of the synchronous CAD tools and flows with respect to applying them on an on-chip network is presented in [19]. These limitations are as follows:

- There is no mechanism to ensure that delays are matched on different paths in a circuit, thus making the use of bundled-data infeasible for large networks, since each path would require manual consideration. Due to this inability to specify relative delay constraints, there is no automated hazard (glitch) avoidance available.
- The CAD tools do not tolerate combinational feedback paths, hence they are unable to infer sequential circuits built from combinational gates. Because of this restriction, circuits that create sequential elements that are not explicitly defined in the standard cell library cannot be optimized using automatic gate sizing or repeater insertion.

- For automatic circuit optimization, each path must be referenced to a common global clock. If this reference cannot be made, the tool simply ignores the path and the delay on that path is not optimized.
- The circuit optimization tools are only designed to insert repeaters to manage wire delays. They do not have the ability to insert sequential elements.

The presence of these limitations resulted in the selection of DI circuit over bundled data implementation. Hence, if these problems can be addressed then bundled data designs could be developed using synchronous CAD tools.

Timing is the key to generation of complex asynchronous systems, which show improvements in terms of power, performance and even area. But to leverage timing in the CAD flows, there needs to be a representation that describes it in both the synchronous and asynchronous domains similarly. Relative timing is one such unifying approach. It defines the timing as a sequence of competing events that have a specific order of arrival at a point in the circuit [20]. Hence the problem of timing is simplified into a problem of ordering.

1.2 Motivation

Asynchronous circuits have a lot of benefits like modularity, composability, low power, and robustness against *process, voltage and temperature* (PVT) variations. For a long time, they have been proposed as the solution for most of the design challenges faced by synchronous designs, but this promise has not materialized commercially. Design examples like the Pentium front-end, the low power asynchronous ARM, the desynchronized DLX and many other examples have shown these benefits exist and can be derived [4, 21, 22, 23, 11, 12, 13, 14, 15]. The approaches to achieve these benefits are specific to a certain asynchronous design style which uses noncommercial research tools and custom specification languages. The flexibility in terms of design approaches and the vast choices of asynchronous design styles prevent us from achieving these benefits due to the lack of a unifying methodology.

The asynchronous protocol-based interfaces between design elements are one of the most efficient multifrequency designs, since this reactive design style requires no overhead to synchronize between frequency domains. Thus, one might expect the design of unclocked multifrequency architectures to exhibit reasonable design productivity due to their modularity; however, that has not been shown to be the case when compared with similar clocked design flows. The primary reason for this lack of productivity is related to the difference in the timing models between clocked and unclocked architectures and the clocked commercial CAD that has been developed to support design productivity.

The primary goal of this work is to present a multifrequency methodology that develops pure asynchronous designs or both synchronous and asynchronous designs together. The existing mature CAD tools and flows available for synchronous designs can be leveraged to generate asynchronous circuits, thus enhancing the acceptance of asynchronous circuits. This work also assists in an easy transition from the traditional synchronous designs with discrete-time domain into asynchronous design domain with continuous-time and reactive systems.

The new multifrequency methodology must facilitate in deriving the above mentioned benefits for any design style. It must give the designers the freedom necessary to choose any design style and easily develop a working system with either asynchronous or mixed asynchronous and synchronous timing domains.

1.3 Contributions of this Work

The goal of this research is to develop a multifrequency flow which addresses one of the major bottlenecks related to development and acceptance of asynchronous designs. Major contributions of this work are as follows:

- A methodology is presented to enable design of asynchronous circuits using synchronous CAD tools and flows. It facilitates an easy transition for any circuit designer/company to adopt asynchronous methodology.
- A flow for the characterization of asynchronous circuit templates has been defined, which enables their use with synchronous CAD tools and flows. This flow provides a systematic way of characterizing the asynchronous designs to be used with the overall methodology.
- Algorithms have been developed and presented for reset addition and cycle cutting for automatic characterization of asynchronous design templates to be used with synchronous CAD tools and flows. Understanding reset addition and a systematic approach for its addition for power/performance optimization of the overall design is necessary and is often a manual step in designing asynchronous circuits. The algorithm

presented in this dissertation addresses these issues. Similarly, the cycle cutting algorithm is an indispensable component of the methodology and flow presented here. It enables the use of timing and STA algorithms of synchronous CAD tools for sizing, validation and optimization of asynchronous circuits.

- Benchmark circuit templates have been developed and provided for rapid characterization of bundled-data asynchronous designs with respect to power, performance and area. Also, CAD has been developed to automate the generation of the characterization results. Comparison results for 4-phase protocols with early and late data validity, as well as asynchronous *first in first out* (FIFO) structures, with respect to energy, performance and area, are presented by using the tool flow presented in this dissertation. These results assist in quick selection of asynchronous templates based on the design requirements.
- The application of the tool flow and methodology on 16-point and 64-point *fast Fourier transform* (FFT) circuit are presented. Also, synchronous, asynchronous, *globally asynchronous locally synchronous* (GALS) and *locally asynchronous globally synchronous* (LAGS) implementations of the *open core protocol* (OCP) are implemented and compared. These case studies show that this methodology can be applied to large circuits as well as to circuits with mixed timing schemes i.e., synchronous and asynchronous.
- This dissertation presents approaches for timing closure for designing better asynchronous designs. The increase in the number of constraints result in an increase in the effort to achieve no negative slack for all the paths in the circuit. The timing closure approaches address a key manual step for rapid development of large asynchronous designs. Its applicability is shown on the 64-point FFT example. Wireload models are explored to understand their effects on timing and application of optimizations on asynchronous bundled-data circuits. Also, automatic scan insertion is applied using Tetramax to explore datapath testability and application of ATPG on bundled-data circuits.

1.4 Overview of this Dissertation

This dissertation describes a methodology to generate asynchronous circuits as well as circuits with both synchronous and asynchronous blocks using synchronous CAD tools and flows. The methodology consists of two parts: the first part develops the asynchronous circuit templates and characterizes them for timing and cycle cut constraints. These circuit templates and their constraints are then used in the second part with the synchronous CAD flows to apply timing driven optimization and sizing on the asynchronous design and synthesize the combinational logic for the datapath. The application of the methodology is described to design asynchronous circuit templates; additionally, some case studies to show the applicability of the methodology to derive asynchronous and mixed timed circuits are detailed.

Chapter 2 describes the concept of asynchronous circuits, RT, and synchronous CAD tool flow. Chapter 3 presents the approach and methodology of designing asynchronous circuits using synchronous CAD tools and flows. The generation of constraints and their applicability to allow timing driven sizing and optimizations on asynchronous designs are also described. The reset algorithm, which assists in automatic addition of the reset signal in an asynchronous circuit, is explained in Chapter 4. Additionally, the relationship between cycles and the addition of a reset signal as well as power or performance optimization is detailed. Use of STA algorithms and performing timing driven optimization on asynchronous designs requires the asynchronous circuit to be specified as a *directed acyclic graph* (DAG) to the synchronous CAD tools. Chapter 5 gives an algorithm to generate the cycle cuts constraints while preserving timing paths, thus representing asynchronous circuits as a DAG. Automatic generation of the cycle cut constraints and the addition of the reset signal enables automation of asynchronous circuit design and its characterization. Chapter 6 explains the application of the tool flow to generate results for a family of 4-phase handshake protocol with data valid at the falling edge of the request signal which are also known as 4-phase handshake protocols with late data validity. Any methodology and tool flow needs to be explored by generating circuits of varying complexity to show the merits of the approach. Chapter 7 describes the applicability of the tool flow on pure asynchronous designs like a simple 3-stage pipeline computing $x^2 + 3x$, 4 types of asynchronous FIFO configurations of varying depths and datapath widths, a 16-point and a 64-point asynchronous FFT design, and also synchronous, asynchronous OCP and OCP design with both clocked and unclocked domains. Chapter 8 concludes this work with proposals for future extensions.

CHAPTER 2

BACKGROUND

Asynchronous design is becoming increasingly attractive because of its potential benefits such as lower power, modularity, composability, ease of global timing issues, robustness against *process, voltage and temperature* (PVT) variations, and elasticity to name a few [24, 25]. This chapter gives an overview of asynchronous design approaches and an overview of the various terminologies used throughout this dissertation. A detailed understanding of asynchronous circuit design can be gained through following reference books: [26, 1, 27] . The basics of relative timing and an overview of the synchronous CAD (computer aided design) flows are also presented.

2.1 Asynchronous Circuit Basics

Asynchronous designs remove the discrete time assumption from the synchronous design approach and are continuous in terms of time. The presence of flexibility in terms of design style and approaches leads to a vast number of solutions to any given problem. They can be designed with or without any timing assumptions with respect to a specific delay model.

2.1.1 Asynchronous Circuit Classification

Different classes of asynchronous circuits can be defined based on the various assumptions they make on timing and environment. These assumptions are made with respect to how delays are modeled, i.e., as fixed, bounded and unbounded. The fixed delay model assumes the delay to have a fixed value. The bounded delay model considers delay to be bounded by a time interval. The unbounded delay model does not restrict the delay in any bounds and hence it can have any finite value. Based on these models, the following classification of circuits is defined.

- *Delay insensitive* (DI) - These circuits operate correctly irrespective of the delays on its gates and wires. An unbounded delay model is assumed while designing these circuits. These designs are the most conservative and are limited to a small set of circuits [28].

- *Quasi delay insensitive* (QDI) - The unbounded delay model for DI circuits is relaxed for a specific case with wire forks considered to have exactly the same delay, also known as "isochronic forks." This assumption leads to wider applicability of these circuits [29].
- *Speed independent* (SI) - These circuits assume arbitrary gate delays, but the wire delays are considered to be zero or negligible. Hence these circuits also consider wire forks to be isochronic.
- Timed circuits - These circuits are designed using the bounded delay model with specific minimum and maximum delays for each gate. The presence of defined bounds leads to more timing constraints and hence greater effort in timing verification. These circuits are generally faster, smaller, and consume less power [30].

The addition of timing assumptions can lead to performance benefit as well as simpler and lower power circuits, but this benefit is gained at the cost of robustness against PVT variation [31]. This occurs due to the reduction in functional redundancy in the circuits with the increase in the timing assumptions. It is helpful to keep these timing assumptions local so that they can be easily controlled. One of the design styles which exploits this is the bundled data design. Thus, to derive the maximum benefit in terms of power, performance and area, approaches to handle timing need to be explored.

2.1.2 2-Phase and 4-Phase Signaling Protocol

Asynchronous communication between two asynchronous entities is generally indicated by transitions on a request (*req*) and an acknowledge (*ack*) signal. These transitions indicate the initiation and the end of a valid transaction. The ordering of the events in a transaction is governed by the transitions on these signals, and this ordering results in the handshake protocol between two asynchronous entities. There are two signaling protocols that are commonly used, the 2-phase protocol and 4-phase protocol.

2.1.2.1 2-Phase Protocol

The 2-phase protocol, also known as *non-return to zero* (NRZ) protocol or *transition signaling* protocol, consists of two transitions for any valid transaction as shown in Fig. 2.1. One transition is on the request line, and it indicates the presence of a new data word and the other transition is on the acknowledge line, and it indicates the acceptance of the new data and also the end of the transaction. Each transition on the request and acknowledge line (i.e., from high to low and from low to high) indicates a new transaction [32]. 2-phase protocols are considered to be more efficient, as each transaction consists of only two transitions. The circuit implementation of the 2-phase protocols tend to be large due to the edge detection requirements of the protocol and at the register storage level. The family of untimed 2-phase protocols consists of a very small set of eight protocols [33].

2.1.2.2 4-Phase Protocol

The 4-phase protocol, also known as *return to zero* (RZ) protocol or *level signaling* protocol, consists of four transitions for any valid transaction as shown in Fig. 2.2. Since each transaction consists of four transitions, there is a greater flexibility in terms of selection of edges when the data word is valid, but there is greater overhead of the reset phase in the protocol which is a performance bottleneck. This results in lower throughput, which is one of the reason why asynchronous network-on-chip implementations use 2-phase protocols for long wires instead of 4-phase [34]. For designs whose delays are dominated by logic delays and not by wire delays, this protocol presents a wide variety of options with respect to data validity schemes employed (Sec. 2.1.3.2). The characterization of a family of untimed 4-phase early data validity protocols resulted in 131 different circuit implementations [35, 36]. Similarly, a family consisting of 32 untimed 4-phase late data validity protocols is reported in this dissertation in Chapter 6.

2.1.3 Bundled Data Protocols

A general 4-stage "bundled data" asynchronous linear pipeline design is shown in Fig. 2.3. Bundled data represents the data signals as normal Boolean levels, and separate request and acknowledge wires are required with the data signals to control the flow of data [1]. Hence, it consists of a datapath similar to a clock design and an asynchronous control

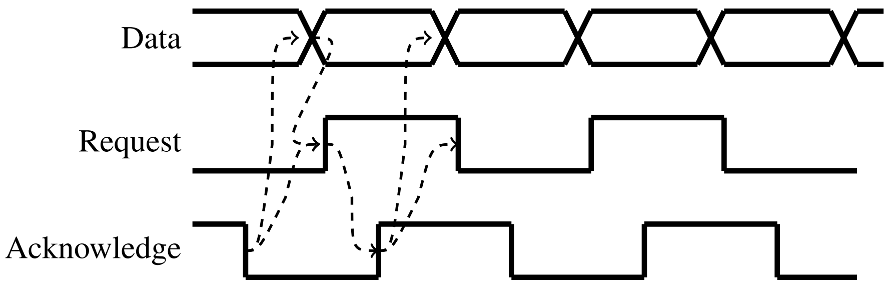

Figure 2.1: 2-Phase protocol.

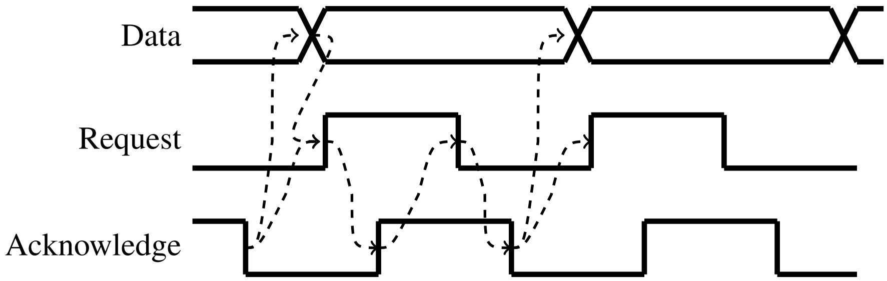

Figure 2.2: 4-Phase protocol.

path. The datapath contains acyclic *combinational logic* (*CL*) and registers (*L*). The control path consists of *linear control* (*LC*) blocks, which to ensure the functional correctness of a bundled data design, the setup and hold time constraints at the registers must be fulfilled. The setup time constraint is the time before the arrival of the clock signal at any register that the data must be stable. To fulfill the setup time constraint at the registers, the latency through the control logic must be greater than the maximum delay of the combinational logic. Thus, delay elements may be required between LC blocks. Similarly, the hold time constraint is the time after the arrival of the clock signal at any register that the data must be stable. To fulfill the hold time constraint at the registers, the reset phase of the protocol must be longer than the hold time requirements.

2.1.3.1 Handshake Channel Type

The channel on a handshake network generally consists of a request signal and an acknowledge signal. Different protocols can be derived by analyzing the sequence of transitions on a request and an acknowledge signal on a handshake channel. Handshake

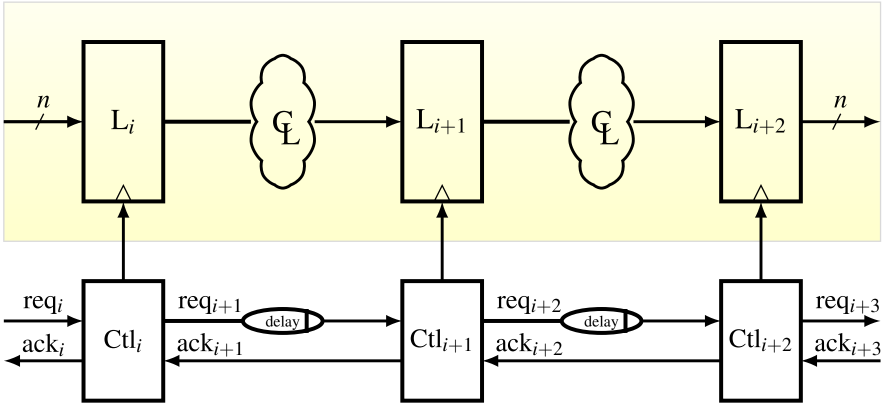

Figure 2.3: Timed (bundled data) handshake design. Delay sized by RT constraint $req_i\!\uparrow \mapsto L_{i+1}/d + s \prec L_{i+1}/clk\!\uparrow$.

channels can be characterized as *push* or *pull* channels based on whether the sender or the receiver initiates the handshake [37]. The side initiating the request is termed active while the passive side generally responds with an acknowledge signal .

- Push channel - Channel with a sender being active and the receiver being passive.
- Pull channel - Channel with a sender being passive and the receiver being active.

The handshake controllers interfacing between different channels considered in this dissertation are of a specific type. Any handshake controller interfaces two channels i.e., left and right channel. Both channels are of the push types with the handshake controller being the passive entity on the left channel and the active entity on the right channel.

2.1.3.2 Data Validity in Bundled Data Protocols

Data validity information in a protocol describes the specific transitions which initiate and end any transaction. Thus, it describes the band where the data remains stable and where it can change. 2-phase protocols have a fixed data validity scheme with the data valid between the transition on the request and the acknowledge. But 4-phase protocols have much more flexibility when using the data validity schemes to its benefit. Fig. 2.4 shows the different schemes which are described below [37]:

- Early data validity - A data word is valid before the rising edge of the request (*req*) signal and it has to stay stable until the rising edge of the acknowledge (*ack*) signal.
- Late data validity - A data word is valid before the falling edge of the request (*req*) signal and it has to stay stable until the falling edge of the acknowledge (*ack*) signal.

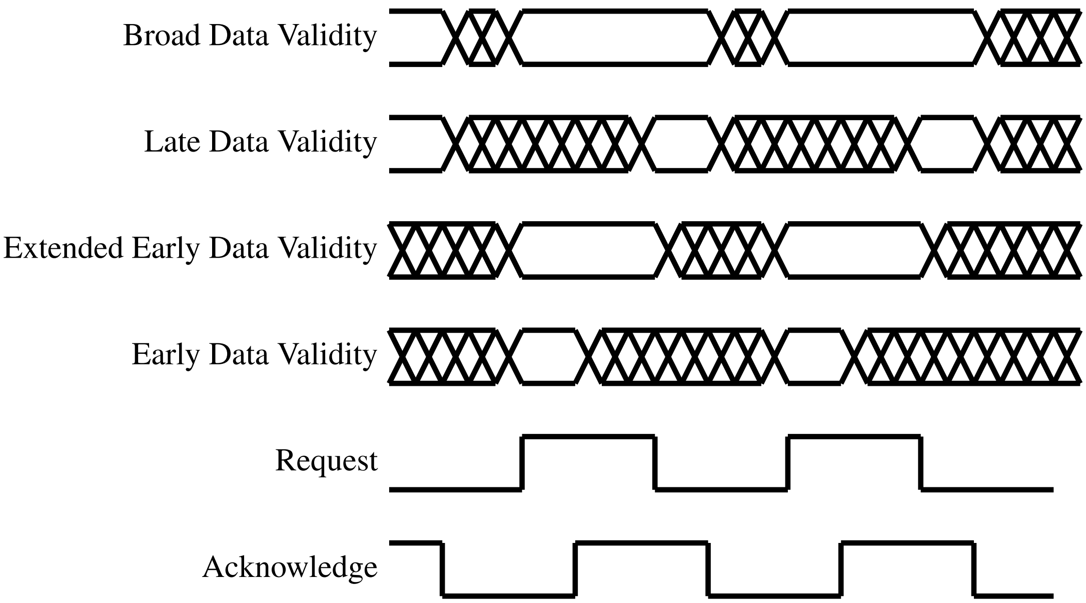

Figure 2.4: 4-phase protocol: data validity schemes[1] [1]

- Broad data validity - A data word is valid before the rising edge of the request (*req*) signal and it has to stay stable until the falling edge of the acknowledge (*ack*) signal.
- Extended early data validity - A data word is valid before the rising edge of the request (*req*) signal and it has to stay stable until the falling edge of the request (*req*) signal.

2.2 Relative Timing

Timing is the fundamental difference between clocked and asynchronous design flows. The effect of time on a system is to order and sequence events. In this work the relative timing (RT) concept is extended into a methodology that enables the representation of the sequencing that timing imposes on circuits [20]. A RT constraint consists of a common timing reference and a pair of events that are ordered in time for correct circuit operation. The common reference is called the *point-of-divergence* (pod), and each ordered event is called a *point-of-convergence* (poc). A constraint is represented as $\mathsf{pod} \mapsto \mathsf{poc}_1{+}m \prec \mathsf{poc}_0$, where poc_1 must occur in time before poc_0 with a margin of m. Hence, the maximum path delay from pod to poc_1 must be less than the minimum path delay from pod to poc_0.

[1] This figure was originally published by Springer and Kluwer Academic Publishers, 2001, page 117, "Chapter 7: Advanced 4-phase Bundled-data Protocols and Circuits," Jens Sparsø and Steve Furber, figure 7.2, © Springer and is used with kind permission from Springer Science and Business Media.

Fig. 2.5 shows a 3-stage clocked linear pipeline. For this clocked design, the relative timing constraint with respect to setup time (s) is expressed as $R_i/clk{\uparrow}_i \mapsto L_{i+1}/d+s \prec L_{i+1}/clk{\uparrow}_{i+1}$. The early arriving RT constraint path from pod clock generator to poc$_0$ L_{i+1}/d is expressed as a maximum delay constraint from the L_i/q output of latch L_i to L_{i+1}/d input of the latch L_{i+1}. The late arriving RT constraint path from pod clock generator to poc$_1$ L_{i+1}/clk can be expressed as a minimum delay from the clock generator to the next rising edge of the clock signal at the latch L_{i+1}. Bundled data pipelines (Fig. 2.3) have similar data timing requirements to clocked design (Fig. 2.5). This is expressed in the asynchronous pipeline as $LC_i/lr{\uparrow} \mapsto$ L_{i+1}/d+s $\prec L_{i+1}/clk{\uparrow}$. This constraint defines the pipeline frequency and setup time constraint at the register L_{i+1} with the requirement that the maximum delay path from $LC_i/lr{\uparrow}$ (pod) to L_{i+1}/d (poc$_0$) should be faster than the minimum delay path from $LC_i/lr{\uparrow}$ to $L_{i+1}/clk{\uparrow}$ (poc$_1$) by setup margin. For one RT constraint in Fig. 2.3, poc maps to LC_i/lr, poc$_0$ to L_{i+1}/d, and poc$_1$ to L_{i+1}/clk. The path from pod to poc$_0$ is [$LC_i/lr{\uparrow}$ $LC_i/clk{\uparrow}$ $L_i/clk{\uparrow}$ L_i/q L_{i+1}/d], and from pod to poc$_1$ is [$LC_i/lr{\uparrow}$ $LC_i/rr{\uparrow}$ $LC_{i+i}/lr{\uparrow}$ $LC_{i+1}/clk{\uparrow}$ $L_{i+1}/clk{\uparrow}$]. The RT constraint holds and the circuit operates correctly iff the minimum delay from pod to poc$_1$ is a setup time s more than the maximum delay from pod to poc$_0$.

2.3 Synchronous CAD Tool Flow

The synchronous design methodology and paradigm is ingrained in most of the design approaches today, thanks to the industry level CAD tools and flows available. A general

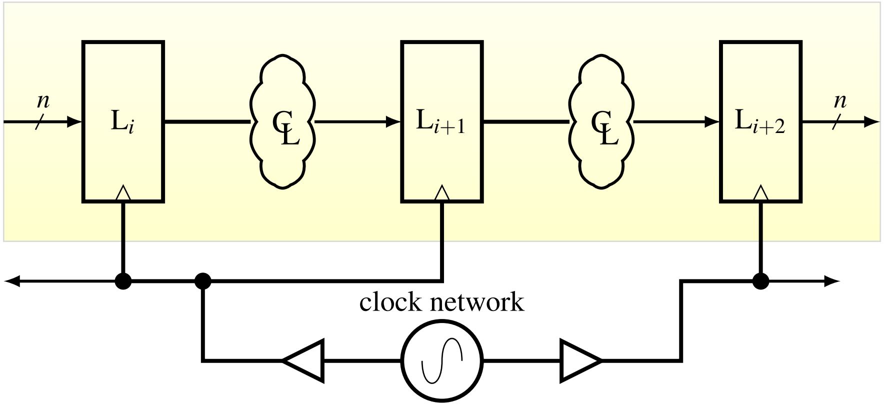

Figure 2.5: Frequency based (clocked) design. Clock frequency and datapath delay of first pipeline stage is constrained by $L_i/clk{\uparrow}_i \mapsto L_{i+1}/d+s \prec L_{i+1}/clk{\uparrow}_{i+1}$.

synchronous CAD tool flow (Fig. 2.6) starts with a specification, which is written as a behavioral description in an *hardware description language* (HDL). This description is then synthesized for a certain set of constraints like clock period, clock skew, input delay, output delay and variation assumptions. The output structural HDL circuit generated is then validated for timing violations and functionality. Timing information from the synthesis step is employed for this validation using *standard delay format* (SDF) back-annotation. The synthesized circuit is then routed using a physical design tool, after which another step of validation is performed for timing as well as functionality. The postroute timing validation also uses the SDF information. Timing path analysis and also estimation for power consumed by the circuit can be done using parasitic information (SPEF) file and also the *value change dump* (VCD) file, which represents the switching activity. If timing is met and the circuit functions correctly, then the circuit can be checked for placement errors and its logic verified against the specification before fabricating it. This is just a top level tool flow description and there can be other tools and steps, such as *design for test* (DFT) logic, timing closure, etc. that can be added to this flow.

The benefit of this flow is a fixed methodology, modularity of using tools from different companies at each step, presence of industry grade CAD tools and also availability of various algorithms for optimization at each step of the flow. Since the flow is defined, it becomes simpler for the circuit designers to design circuits. The limitation of this flow is that it has been restricted for synchronous design, and it cannot be used to generate systems based on different timing assumptions like asynchronous systems. The problem of generating multifrequency systems using synchronous CAD tools and flows is addressed in this dissertation by adding extra steps to generate designs with clocked and unclocked circuit blocks.

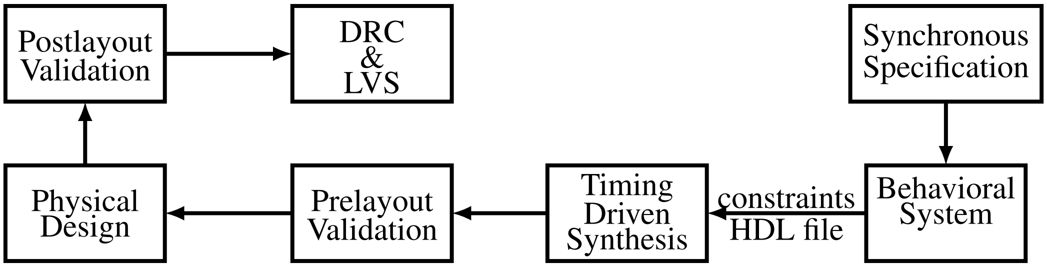

Figure 2.6: Synchronous CAD tool flow.

CHAPTER 3

ASYNCHRONOUS CAD TOOLS AND FLOWS

Multifrequency designs achieve improved power and performance by operating each circuit in a system at the power and performance point they are optimized for. But to gain these benefits, the timing models employed in the clocked design style must be leveraged by general multifrequency designs, thus enabling the reuse of the existing tools, languages, and flows with all methods of timing for design and architecture optimization as used for clocked (synchronous) designs. A transformative representation called *relative timing* (RT) does just that; it bridges the gap between the incompatible timing used in unclocked design styles by expressing the timing in a form used by commercial clocked EDA tools. Once timing compatibility is achieved, existing design languages, cell libraries, and tool flows become common to all design styles. This enhances productivity, increases the quality, and reduces the cost of adopting multifrequency design methodologies with their resultant power and performance advantages.

RT is robust because timing requirements are formally derived and proven correct and complete. RT is also general because it can be applied to the timing of all design approaches, including clocked design. A set of RT derived constraints are mapped into the clocked EDA tools enabling the existing tools and flows to be employed for both global clocked, multifrequency design, or mixed design modes. However, a number of additional tools are required to characterize the timing, represent the timing graphs of the sequential circuits as directed acyclic graphs, protect the sequential designs from logical modification through the tool flow, map the constraints onto an architecture, and enhance timing driven optimization and validation.

This chapter gives an insight into the complete tool flow with details regarding the asynchronous template characterization and constraint generation, as well as, using these templates with the traditional synchronous CAD tools and flows.

3.1 Key Contributions

The primary contribution is to define a methodology to develop asynchronous designs using synchronous CAD tools and flows. This methodology utilizes the existing tools for synthesis, verification, and automatic RT constraint generation in conjunction with synchronous CAD tools and flows to derive asynchronous circuits. Algorithms for automatic reset addition and generation of cycle cuts are developed, which enable automation, as well as the application of timing driven optimizations and *static timing analysis* (STA) algorithms of the synchronous CAD tools. Approaches and analysis of mapping the RT constraints to minimum and maximum delays is performed with different ways of achieving timing closure. The traditional clocked design flow remains unmodified except for the addition of these key technologies and algorithms deployed to characterize and integrate time incompatible design elements into the EDA tool flow.

3.2 Background

Timing is the source of the power, performance, noise, and area benefits that asynchronous circuits enjoys. It is also the primary impediment to commercial adoption. At the circuit and architectural level, self-timed/asynchronous design uses a continuous timing model, whereas clocked design uses a discrete model. Hence, the most challenging design to integrate with clocked tool flows are purely asynchronous design blocks. Thus, an asynchronous design is used as an example application to describe this methodology.

Instead of using examples of the more traditional delay insensitive circuit style, the bundled data design style shown in Fig. 2.3 is used as an illustrative example. The datapath of bundled data asynchronous design is specified, synthesized, and validated in the same way as for clock design when using an ASIC design flow. The key difference is that the clock is removed and replaced with handshaking protocols to implement performance, timing, sequencing, and flow control. The handshake protocol selected for the controller, as given in Fig. 3.1, is a *timed* burst-mode specification that requires protocol level timing constraints to function correctly. The protocols consists of inputs lr and ra and outputs la and rr with `c1` and `c2` being the synchronization points between the left channel and the right channel. This design choice is made to better illustrate two primary features of the tool methodology presented here: (i) the ability to support an arbitrary set of timing constraints; and (ii) to

LEFT-CHANNEL	=	$\underline{\texttt{lr}}.\texttt{c1}.\texttt{la}.\texttt{c2}.\underline{\texttt{lr}}.\texttt{la}$.LEFT-CHANNEL
RIGHT-CHANNEL	=	$\underline{\texttt{c1}}.\texttt{rr}.\underline{\texttt{c2}}.\underline{\texttt{ra}}.\texttt{rr}.\underline{\texttt{ra}}$.RIGHT-CHANNEL
SPEC	=	(LEFT-CHANNEL | RIGHT-CHANNEL) \\{ c1, c2 }

Figure 3.1: CCS specification of pipeline controller.

demonstrate the compatibility with current clocked EDA tools by showing how the method presented here can rely entirely upon the traditional EDA tools for design, optimization, and validation.

Timing has previously been measured and reported by value rather than by its effect on a system. The purpose of time on a system is to order and sequence events. RT models the sequencing imposed by circuit delays. Two core components are required to represent RT: (i) a common timing point of reference, called a *point-of-divergence* (pod); and (ii) two signal events that become ordered in time, called a *point-of-convergence* (poc). The RT equation $\mathsf{pod} \mapsto \mathsf{poc}_1 \prec \mathsf{poc}_0$ is the basic representation of how time affects a system. This equation is a logic expression, and can therefore be employed to define the behavior of a system independent of any specific time value. RT can also be used to represent both timing paths and frequencies. RT is used to constrain the logical behavior of a circuit due to system delays. RT can control delays in a system in order to achieve a desired signal ordering so as to verify that a system operates correctly in the time domain. When mapped to a physical design where the absolute values of time are important, the RT equation is annotated with delay and frequency values as well as with margins of separation between the poc. RT is completely general and represents time in a way that is natural for designers to understand. Thus, RT creates a new representation and method of thinking about design, and it also enables the the effects of time on a system.

3.3 Clock Compatible Multifrequency IC Flow

Extending the application of RT to the entire end-to-end design and synthesis flow is addressed. The flow and the necessary steps to seamlessly integrate this flow with current commercial clocked CAD tools are shown in Fig. 3.2

There are four main aspects in a basic design flow where additional steps are added to the traditional clocked flow: (i) the design and characterization of the RT design elements,

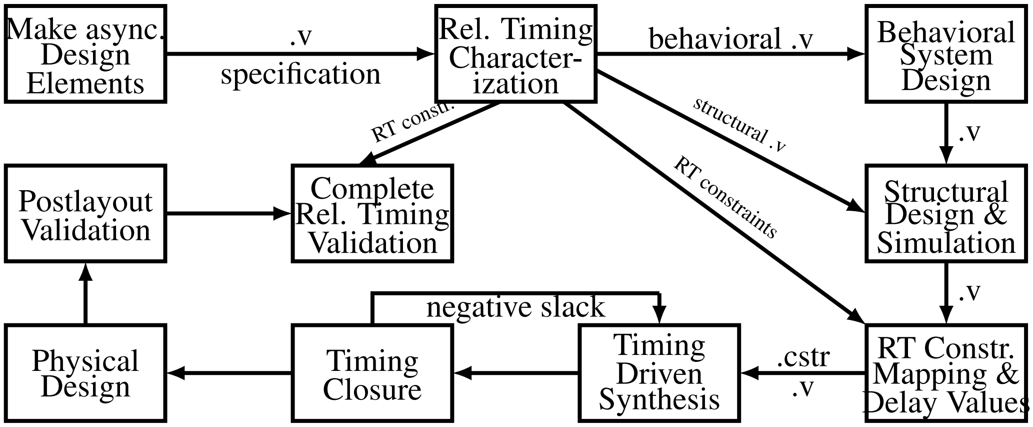

Figure 3.2: Simplified relative timing multifrequency design flow.

(ii) mapping of the RT constraints and timing values onto the physical architecture, (iii) perform timing closure on the timing targets supplied in the previous step, and (iv) perform complete postlayout validation of the RT constraints. Each of these areas require some additional CAD to support the algorithms of the corresponding aspects of the flow. Note that this design flow works well for both ASIC, as well as, full custom design flows, and is agnostic to the specification and design languages employed. It is also agnostic to the type of design methodology used. For example, the flow can be applied to *delay insensitive* (DI), *speed independent* (SI), or timed circuits in either the control or datapath or both. No specific gate primitives are required, and any cell library can be employed, including libraries with dynamic domino gates.

3.3.1 RT Element Design and Characterization

The first part of the RT flow creates and characterizes circuit element in a way that the traditional EDA tools can integrate them directly into their timing driven circuit design and optimization algorithms. Note that the design elements that are characterized may not be logically modified without changing the RT characterization information. Therefore, each of these modules are provided as a characterized structural design module. At the end of the flow, part of the constraint information includes control to protect the structural design while still allowing the tools to optimize the size of the cells for performance and power optimization. This portion is the fundamental foundation upon which the rest of the flow rests.

3.3.1.1 Element Design

The first aspect is to specify and design the elements. Fig. 3.3 shows the flow for designing asynchronous circuit elements. First, the designer must create a formal high-level specification of the protocol for the sequential block. This typically would be a controller specified as a Petri-Net, *communicating sequential processes* (CSP), burst-mode, or *calculus of communicating systems* (CCS). A CCS specification of a pipeline controller is provided in Fig. 3.1.

The next step is to create a design that has been fully mapped to the target cell library. If a design already exists for that specification, synthesis can be skipped. The controller can be designed by hand or synthesized from the specification to create hazard free logic equations. Various sequential circuit synthesis tools can be used for synthesis. These are all tools from academic institutions as no commercial tools currently exist for general sequential circuit synthesis. The design is then technology mapped to an implementation library generating the minimum number of hazards [38]. A valid design for the specification in Fig. 3.1, which has been technology mapped to the academic Artisan 65nm library, is provided in Fig. 3.4. This design is expressed in a high level design language such as Verilog using a *structural* description, shown in Fig. 3.5.

The protocols and designs that are characterized for RT are often sequential and specified as mealy state machines. This results in designs using combinational logic with feedback as can be seen in Fig. 3.4. Such designs need to be reset to the correct starting state. The next part of the element design consists of designing reset in a way that has the least impact on power, performance, and area of the design. Other methods of generating reset may be employed, such as through algorithms in the sequential synthesis tools or as stand alone applications.

The final circuit after reset addition can be deployed in the system level architecture. The characterized elements contain a behavioral representation for high-level design simulation. The structural representation is used for generating the RT constraints and in synthesis followed by the physical design of the overall system.

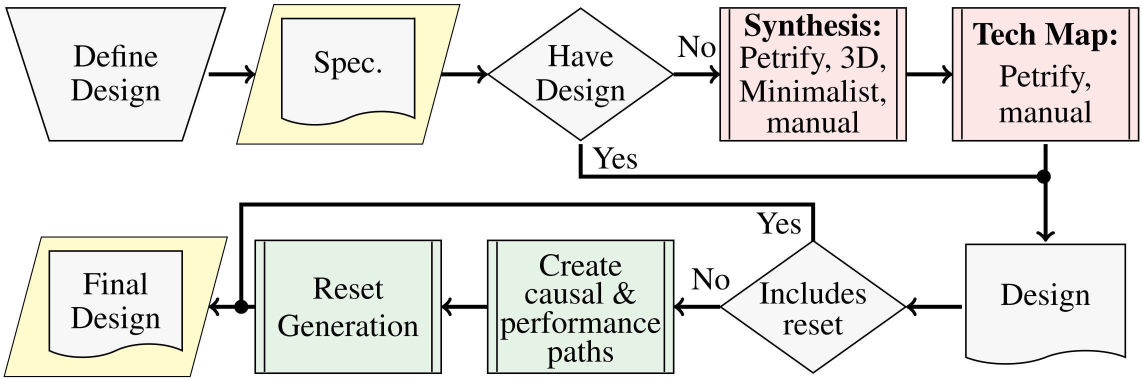
Define Design
Spec.
Have Design
No
Synthesis: Petrify, 3D, Minimalist, manual
Tech Map: Petrify, manual
Yes
Yes
Final Design
Reset Generation
Create causal & performance paths
No
Includes reset
Design

Figure 3.3: Design element creation flow.

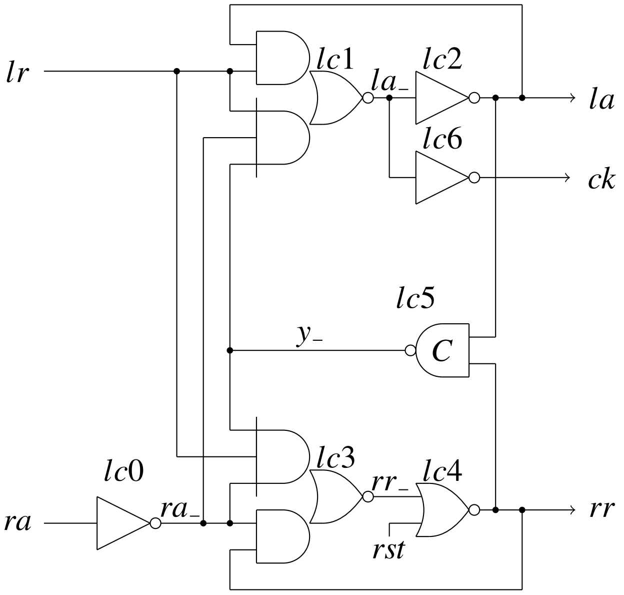

Figure 3.4: LC circuit implementation[1].

3.3.1.2 Automatic Sequential RT Characterization

This part of the flow is shown in Fig. 3.6 as a flowchart. It takes the design, specification, and architectural usage information and produces the constraints that are used by the EDA tools throughout the typical ASIC design flow.

The first step is to generate the RT constraints for the design. A tool called `verilog2ccs` translates the verilog module that is used in the design into a formal *calculus of communicating systems* (CCS) specification. A formal verification engine based on bisimulation

[1] This figure is the static combinational gate implementation for the RT controller specification in [39].

```
module pipe_ctl (lr, la, rr, ra, ck, rst);
  input            lr, ra, rst;
  output           la, rr, ck;
  INVX1A12TH       lc0  (.A(ra), .Y(ra_));
  AOI32X1A12TH     lc1  (.A0(lr), .A1(ra_), .A2(y_), .B0(lr), .B1(la), .Y(la_));
  INVX1A12TH       lc2  (.A(la_), .Y(la));
  AOI32X1A12TH     lc3  (.A0(ra_), .A1(lr), .A2(y_), .B0(ra_), .B1(rr), .Y(rr_));
  NOR2X1A12TH      lc4  (.A(rr_), .B(rst), .Y(rr));
  c_element_       lc5  (.A(la), .B(rr), .Y(y_));
  INVX1A12TH       lc6  (.A(la_), .Y(ck));
endmodule // pipe_ctl
```

Figure 3.5: Structural Verilog in 65nm Artisan library.

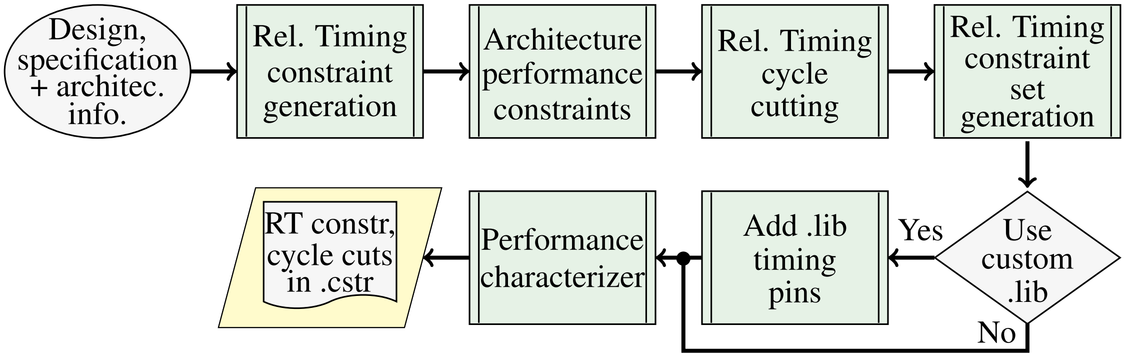
Design, specification + architec. info.
Rel. Timing constraint generation
Architecture performance constraints
Rel. Timing cycle cutting
Rel. Timing constraint set generation
Use custom .lib
Yes
No
Add .lib timing pins
Performance characterizer
RT constr, cycle cuts in .cstr

Figure 3.6: Automated sequential RT characterization flow.

semantics called ARTIST automatically generates all of the RT constraints necessary for the design to meet the specification [40]. It takes the specification and the formal representation of the design and the library cells used in the design (e.g., NAND gate shown in Fig. 3.7). The static library cells are represented as semimodular elements where inputs that disable the output are failures. RT constraints are generated to enforce a signal ordering that makes the Fail state unreachable in the design. For instance, in Fig. 3.7 if $\mathsf{c} \prec \mathsf{a}$ in state NAND0b0 the timing failure is avoided.

Constraints for the pipeline controller are shown in Fig. 3.8. The constraints on the first row are required for correct conformance between the implementation and specification. The constraints on the second row are due to the timed nature of the burst-mode specification. Additional performance and correctness constraints are added based on the intended

```
agent NAND001 = a.NANDa01 + b.NAND0b1 ;
agent NANDa01 = a.NAND001 + b.NANDab1 ;
agent NAND0b1 = a.NANDab1 + b.NAND001 ;
agent NANDab1 = a.Fail    + b.Fail    + 'c.NANDab0;
agent NANDab0 = a.NAND0b0 + b.NANDa00 ;
agent NAND0b0 = a.Fail    + b.NAND000 + 'c.NAND0b1;
agent NANDa00 = a.NAND000 + b.Fail    + 'c.NANDa01;
agent NAND000 = a.NANDa00 + b.NAND0b0 + 'c.NAND001;
```

Figure 3.7: The semimodular specification of a 2-input NAND gate. Inputs that would disable an output are not permitted.

$la\uparrow \mapsto y_{-}\downarrow \prec la\downarrow$	$rr\uparrow \mapsto y_{-}\downarrow \prec rr\downarrow$
$lr\uparrow \mapsto la\uparrow \prec ra\uparrow$	$lr\uparrow \mapsto rr\uparrow \prec lr\downarrow$

Figure 3.8: RT constraints for pipeline controller.

architecture of the design, such as the datapath constraint in Fig. 2.3 that applies to pipeline controller elements.

Reducing the number of constraints improves the run-time performance of the CAD tools. Further, some constraints cannot be jointly covered in the flows because they are incompatible with other constraints. Therefore, a subset of the RT constraints are selected to be employed in the timing driven optimization flow.

Current commercial CAD algorithms require that the timing graphs of a circuit are represented as a *directed acyclic graph* (DAG). Most asynchronous modules have combinational feedback loops that must be cut to create a DAG [41]. The selection of the correct set of constraints determines the ability to create a DAG and the optimizations that can be applied to gates in the design. The causal path from pod to poc cannot be cut for an RT constraint to be effective. In general, the full set of RT constraints cannot be used while modeling the timing graph as a DAG. Many constraint pairs are incompatible because they consist of a cyclic path and hence, cannot remain uncut. (Such constraint pairs cover a complete combinational cycle in the circuit.) Cycle cutting therefore plays a critical role in the characterization flow. It must preserve the chosen constraint paths while removing all combinational cycles in the design. Cycles exist both inside a sequential block, as well as, distributed across multiple sequentials due to channel handshaking protocols and system architecture. The interplay between the RT causal paths in the design and the combinational cycles that exist in a circuit makes the cycle cutting problem challenging. The causal paths of the subset of RT constraints are passed to a cycle cutting algorithm to produce a timing DAG. Cycles are cut by disabling timing paths through gates in the cell library with the set_disable_timing command.

The set_disable_timing command used to cut local and architectural cycles for the circuit in Fig. 3.4 include cutting the paths between the following input and outputs of the gates: $\{\ \mathsf{la} \not\rightarrow \mathsf{la\_}, \mathsf{y\_} \not\rightarrow \mathsf{la\_}, \mathsf{ra\_} \not\rightarrow \mathsf{la\_}, \mathsf{rr} \not\rightarrow \mathsf{rr\_}, \mathsf{ra\_} \not\rightarrow \mathsf{rr\_}, \mathsf{y\_} \not\rightarrow \mathsf{rr\_}\ \}$.

Some RT constraints are necessary to perform quality timing driven optimization of a design, others are not. Some constraints exist solely to ensure correct sequential behavior or make hazards unreachable. Many of these are not necessary in the optimization flow since they have a very loose margin, hence they can be ignored. However, correctness constraints that have tight margins may be required to be included in the flow to create functional designs under timing optimizations. RT constraints that are critical to cycle time optimization must be included. For example, $req_i\uparrow \mapsto L_{i+1}/d + s \prec L_{i+1}/clk\uparrow$ is necessary when employing bundled data and the 4-cycle handshake protocol of Fig. 2.3. Additional timing constraints may be added that are not automatically generated from formal verification but have important performance ramifications. Algorithms that employ timed separation of events or canopy graphs can be used to determine architectural level timing constraints [42, 43]. A method of selecting a subset of the constraints that, produce the best optimization and most efficient runtime for the tools, is employed. The full set of constraints are maintained for use later in the flow for postlayout validation.

The timing paths derived from the first row of RT constraints in Fig. 3.8 for Fig. 3.4 are: $\{\ \mathsf{lr} \rightarrow \mathsf{la\_} \rightarrow \mathsf{la}, \mathsf{lr} \rightarrow \mathsf{la\_} \rightarrow \mathsf{la} \rightarrow \mathsf{y\_}, \mathsf{lr} \rightarrow \mathsf{rr\_} \rightarrow \mathsf{rr}, \mathsf{lr} \rightarrow \mathsf{rr\_} \rightarrow \mathsf{rr} \rightarrow \mathsf{y\_}, \mathsf{ra} \rightarrow \mathsf{rr\_}\ \}$.

A set of `set_size_only` constraints are generated for most of the library cells in a characterized element to ensure that the CAD tools does not resynthesize their logic.

The characterized elements are evaluated for performance, power, and area from physical layout. The information includes the formal specification of the element including concurrency and synchronization, the performance values for the element including cycle time and forward and backward latencies, the energy consumption of the element, its area, information regarding the timing constraints, and other constraints on the design such as cycle cutting, sample timing targets, etc. This information is helpful for the designer to select the best part for a particular application. At this point, the sequential asynchronous circuit is fully characterized and ready for integration into the clocked tool flows.

3.3.2 Clocked Design Flow Using RT Sequentials

Once the elements have been characterized for RT, the traditional design flow ensues with some additional steps to integrate the constraints into the flow and to ensure that all constraints hold in the final design. A high-level behavioral description of the design may be created as in Fig. 3.2. It is validated using the behavioral equations from the bag of RT characterized elements. A designer writes traditional Verilog, but rather than expressing the pipeline stages as `always @(posedge clk)`, an instance to a pipeline controller is inserted. This design flow is effectively identical to connecting schematics together, where one is wiring up the handshake ports and data at each pipeline stage. For example, to create a pipeline stage with an adder with inputs a and b, the code shown in Fig. 3.9 could be employed, which stores the result in the register of pipeline stage l0 upon receiving a lr handshake.

3.3.3 Mapping RT Constraints Onto Design Instances

Once the high-level design has been created and validated, synthesis of the design occurs. This requires the structural version of the RT characterized elements that are included in the HDL. The rest of the traditional ASIC tool flow requires constraint information from the characterized circuits. This is provided by mapping the RT characterized circuit constraints onto instances in the design and placing these constraints in a .cstr constraint file.

Mapping of design constraints onto instances of an architecture requires that circuit connectivity be evaluated. A RT constraint for a single sequential instance may have several occurrences in a design. If a control path forks two ways, then at least two independent poc pairs may be required for the same pod. The result of the constraint mapping is to create a .sdc file for the entire design consisting of the critical subset of the RT constraints for each characterized instance in the architecture and their mapping onto pod and poc endpoints in the design [44]. The critical RT subset employed is selected for power/performance optimization and circuit correctness.

```
assign          sum = a + b;
linear_control  LC0   (.lr(lr), .la(la), .rr(rr), .ra(ra), .ck(ck0), .rst(rst));
latch32         l0    (.d(sum) .clk(ck0), .q(do));
```

Figure 3.9: Verilog code snippet for simple adder pipeline stage.

An example for the simple addition pipeline above is shown in Fig. 3.10. The controller is from Fig. 3.5. The values provide an adder delay target of 600ps. The constraint endpoints refer to both characterized controller I/O pins, as well as, pins in the cell library.

The timing targets are provided by the user, and can apply as the "clock frequency" for an entire hierarchical design block. They are expressed as the tcl variables d0_fdel, d0_fdel_margin and d0_bdel in the file, representing the forward delay, setup margin to the latch, and backward delay for pipelines in the design. The granularity of the timing may be as coarse or fine as the designer desires. The set_size_only constraints constrain the AOI and NOR gates in the pipeline controller to only allow drive strength modifications by the tool flow algorithms. The set_disable_timing constraints are used to create a directed acyclic timing graph of the design as required by the algorithms in the EDA tools. The set_max_delay and set_min_delay constraints drive the timing driven optimization and validation algorithms in the EDA tool flow for the synthesis and for the place and route engines.

The RT constraint $req_i\uparrow \mapsto L_{i+1}/d + s \prec L_{i+1}/clk\uparrow$ is demonstrated in the Fig. 3.10 SDC file example. The early arriving RT constraint path from pod lr to poc$_0$ l0/d is expressed as a maximum delay constraint from the a and b inputs to input of the latch. The late arriving RT constraint path from pod lr to poc$_1$ l0/clk is expressed as a minimum delay from lr to the clock pin of the latch. The RT constraint holds in the system if the maximum delay signal path plus the margin arrives before the minimum delay path.

3.3.4 Timing Closure

Timing driven synthesis and optimization of the architecture with traditional clocked EDA tools is enabled at this point due to the the constraint file. As long as the RT constraints hold, the circuit functions correctly. However, the timing targets as specified by the user and the mapping tools may provide aggressive max delay constraints so as to achieve a higher performance design. This often results in timing failures expressed as negative slack. Timing closure allows the synthesis tools to iterate on the architecture to converge to a design that

```
set d0_fdel 0.600
set d0_fdel_margin [expr $d0_fdel + 0.050]
set d0_bdel 0.060

set_size_only -all_instances [find -hier cell lc1]
set_size_only -all_instances [find -hier cell lc3]
set_size_only -all_instances [find -hier cell lc4]

set_disable_timing -from A2 -to Y [find -hier cell lc1]
set_disable_timing -from B1 -to Y [find -hier cell lc1]
set_disable_timing -from A2 -to Y [find -hier cell lc3]
set_disable_timing -from B1 -to Y [find -hier cell lc3]

set_max_delay $d0_fdel -from a -to l0/d
set_max_delay $d0_fdel -from b -to l0/d
set_min_delay $d0_fdel_margin -from lr -to l0/clk
set_max_delay $d0_bdel -from lr -to la
#margin 0.050 -from a -to l0/d -from lr -to l0/clk
#margin 0.050 -from b -to l0/d -from lr -to l0/clk
```

Figure 3.10: An SDC example for simple adder pipeline stage.

meets all of the path based timing requirements. If a negative slack is encountered, the timing closure tool adds delay to the related targets and then runs a new synthesis. This occurs until no negative slack exists in the design.

The "#margin" constraints placed in the .cstr file by the constraint mapping flow are used by the timing closure CAD. These constraints tie the two paths of a RT constraint together and ensure that the early path plus margin arrives before the late path. If an early path, expressed as a set_max_delay constraint, is increased, the late path, expressed as a set_min_delay constraint, must also be increased to maintain the required margin of separation.

Once all of the timing constraints have been met the design is ready for physical design and further performance evaluation. The optimized design can be simulated and validated against an architectural specification for performance and behavioral conformance. Iterations back to any part of the flow, including designing a new asynchronous sequential circuit, can occur to improve the architecture.

The physical design is created employing similar .sdc constraints used for synthesis. Additional information and algorithms are employed to improve the physical design quality.

For example, force directed placement can be enhanced, as well as, improving the optimization of minimum and maximum delay constraints can be improved. The physical design process may also iterate on timing closure to ensure that the final design meets all timing constraints.

3.3.5 Final RT Validation

After physical design, the complete set of RT constraints are validated against the design. This usually requires multiple RT constraint sets with different cycle cutting constraints to properly evaluate the design. At this point a .vcd activity file is created and power is evaluated based on postlayout parasitic extraction and node activities. *Engineering change order* (ECO) changes based on margins and yield are performed on the physical design, similar to the prelayout timing optimizations. The physical design is validated for behavioral and performance correctness and yield robustness, as is done with traditional products.

3.4 Results

The flow described in Fig. 3.2 has been applied to clocked and asynchronous multifrequency architectures. All designs reported here are specified behaviorally using Verilog. Structural pipeline control elements, as in Fig. 3.5, are used in the asynchronous circuits. ModelSim [45] is used to validate the architectures. The behavioral datapath and register banks are synthesized and optimized with Design Compiler [5]. The performance and power of the asynchronous control elements are likewise optimized with Design Compiler based on RT delay targets. Physical layout and parasitic extraction is performed with SoC Encounter [46]. Timing and power validation is performed with PrimeTime [5]. A number of custom tools and scripts are employed to characterize the asynchronous elements and integrate the RT constraints into the design flow. The design time of asynchronous architectures using this flow is similar to that of clocked design, assuming that all the asynchronous control elements have been predesigned and characterized.

A brief overview of all the design examples analyzed for this work is given below. The detailed analysis and results are described later in the dissertation. The quality of the results are compared against previous design work which did not have the benefit of the flow described in this dissertation. Two examples are used: a complete family of 4-phase asynchronous pipeline controllers and a large set of clocked and asynchronous FIFOs [35, 47]. The referenced projects used the same commercial tools, but without the support of RT. Due to a number of designs investigated by these comparisons, productivity demanded that ASIC flows be employed using high-level synthesis and validation. For these projects, the complete flow of Fig. 3.2 including flows in Fig. 3.3 and 3.6 along with creating the architectures is completely automated. Even though some steps in the flow can be improved with better algorithms and can be automated, a good comparison of the results at the element level is provided.

The FIFO circuits for linear, parallel, square and tree configurations showed an average improvement of 1.5×–2× in throughput, with less than half the area using this flow. The regularity and simplicity of the pipeline controllers allowed their performance and power to be manually optimized with relative ease in the reported paper [35]. After optimization, the controllers were evaluated using set_dont_touch constraints. Thus, little improvement is possible when using the RT flow resulting in the same average performance with 0.93× smaller area and it consumed 0.90× energy.

These small, homogeneous designs neither demonstrate multifrequency architectures nor they demonstrate jointly synthesized and placed and routed, clocked and asynchronous blocks. Applying this flow to large multifrequency implementation validates that the CAD flow scales, and that power and performance benefits can be derived on large design examples that contain a variety of datapath logic. It likewise validates the design efficiency scaling to large designs.

Clocked and asynchronous multirate 64-point FFT architectures with four distinct frequency domains were designed and compared against each other. The asynchronous design consists of 229K gates. When compared to a clocked design of the same architecture, the async design has a 2.4× improvement in energy per point, 2.4× reduction in area, and 2.0× the throughput for an $e\tau^2$ of 9.6×. Compared to a ultra low power single frequency design [48], the multifrequency design provides 6.8× reduction in energy per point and a 32.3× reduction in time to perform 1K samples, resulting in a 7k $e\tau^2$ improvement at the cost of a 2.1× increase in area.

Five different designs of *open core protocol* (OCP) are evaluated under a uniform test bench. These included designs with a single global clock, fully asynchronous, and multisynchronous designs. Designs with substantial asynchronous components are by far the best in terms of area, performance, and energy per transfer. The purely asynchronous design has 3× the performance and consumes approximately 1/9 the energy of its clocked counterpart. The GALS design also demonstrated almost 4× the throughput at less than 1/5 the energy per transaction.

This methodology allows the development of good asynchronous circuits. It also allows the circuit designer to derive energy, performance, and area benefits for designs with multiple frequencies of operation. The improvement derived from better asynchronous templates and also the quick selection of the templates based on the characterization results, helps in developing better circuits. The major benefits though are seen for designs where asynchronous architectural optimizations result in low costs and overhead in going from one frequency domain to another. It has been observed that the designs where multiple frequencies interact and need synchronization are the best candidates for asynchronous circuits, since there is no overhead in going from one asynchronous circuit block to another if the handshake protocol followed between the blocks is the same.

3.5 Summary

This chapter describes a transformative flow that enables high productivity and design quality in the commercialization of asynchronous and alternative design styles. The high productivity and design quality are achieved by integrating a proven complete set of timing requirements for the design to state of the art CAD. The flow unlocks the timing algorithms that exist in commercial CAD. The RT based flow is fully compatible with all design styles and methodologies. The flow characterizes the timing of sequential blocks which can then be integrated into an architecture with the timing mapped in a format that enables timing-driven optimization. The RT flow enables CAD to perform the same energy-delay optimizations on the asynchronous designs as is performed on the clocked circuits. A power advantage of 10× can be often derived for the asynchronous designs with respect to its synchronous counterpart at the same performance. The new flow enables the design and characterization of sequential circuits using RT in a form that is fully integrated into clocked CAD and tool flows. The characterized sequentials are then embedded into a design with the RT directives that enable their full CAD tool support. Applying the flow to a number of designs has demonstrated a significant improvement in energy, area, and performance.

CHAPTER 4

AUTOMATIC RESET ADDITION BASED ON LATENCY/POWER OPTIMIZATION [1]

The behavior of a sequential circuit cannot be determined solely by its *primary inputs* (PIs) because sequential logic can behave differently for identical input sequences based on the starting state. Thus, it is essential to initialize sequential logic to a specific state to ensure desired behavior.

The state-based behavior of sequential circuits is implemented with *state variables*. State variables are created with feedback cycles in the Boolean logic descriptions of sequential *asynchronous finite state machines* (AFSM). These feedback cycles are explicitly maintained in the circuit realization when the design is technology mapped to static logic gates. Other logic families, such as dynamic logic, can be used to implement AFSMs which would change the way the state variables are implemented. This work applies to designs mapped to static logic gate libraries, since they are the most commonly used logic family.

Initialization of a sequential AFSM is implemented with a reset signal that is asserted upon power up. This is usually a one-time event, but it can also dynamically occur during operation to reset a sequential circuit back to its starting state. This chapter shows how the the former case of power-up reset can be addressed.

Reset can have a significant impact on asynchronous logic design in several ways. The addition of reset signal has a direct influence on the power, performance, and area of a sequential circuit since it involves either addition of gates or adding extra reset input to existing gates. Hence, optimizing reset for power and/or performance can improve the overall design. Second, it is possible to change the hazard properties of an AFSM through

[1] This chapter is a revised version of the work in VLSI-SoC, 2013 [49]. © 2013 IEEE. Reprinted with permission, from Vikas S. Vij, Kenneth S. Stevens, “Automatic Addition of Reset in Asynchronous Sequential Control Circuits,” in 21st IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Oct., 2013.

the addition of reset. On the other hand, if reset is not fully automated, it poses a significant manual effort in the synthesis and characterization of asynchronous circuits.

The addition of reset to an AFSM can be performed at different stages of a design flow. Firstly, it can be added in the specification of the design and implemented during synthesis. Secondly, it can be added at the technology mapping phase of synthesis. Lastly, the addition of the reset signal can be performed during posttechnology mapping phase. Reset addition at the posttechnology mapping step in the design methodology is chosen because it allows the reset logic generation to be independent of the design or synthesis method used. Thus, the method and algorithms presented here can be employed for circuits designed by hand, or from synthesis tools such as MEAT, 3D, Petrify, ATACS or Minimalist [50, 51, 52, 53, 17, 54].

4.1 Key Contributions

This work has resulted in the following contributions:

- An algorithm is designed to generate an asynchronous circuit with reset logic resulting in an improvement in power, area, and performance as compared to the reset logic generated by other algorithms. This is primarily achieved with a three-step heuristic based on logical effort [55] to optimize the circuit either for performance or power.
- A relationship between reset and topological cycles in a circuit is shown when a design is exclusively implemented with static logic gates.
- The algorithm is agnostic of the asynchronous circuit design style used to generate the circuit since it works on a technology mapped circuit. Hence, it can be used with any of the synthesis engines as well as with hand designed circuits.
- It addresses one of the key steps in the AFSM design generation automation flow.

4.2 Background

Two significant holes currently exist in the CAD tools used for synthesis of sequential asynchronous circuit designs: technology mapping and reset generation. Both of these are interesting and related problems, as technology mapping can introduce hazards [38], whereas reset is dependent on the technology mapped circuit. Without automating these holes, synthesis and characterization of asynchronous circuits necessitates manual intervention.

Many tools and algorithms exist for the synthesis of AFSMs. The only one that includes integrated reset support is Petrify [53]. In Petrify, the addition of reset logic is performed after synthesis and is not technology mapped, requiring a final manual step to create a circuit. This manual step is being addressed by a tool named Petreset as an academic project. It analyzes the synthesis results and the design specification. Through simulation, Petreset determines which gate modifications can be performed to reset the design. The restriction of this tool is that it only applies to designs which are synthesized with Petrify; hence, it is not independent of design methodology. This work has not yet been published.

AFSM synthesis algorithms can theoretically be modified to automatically generate reset behavior jointly with the synthesis. However, no such work is reported in the literature. I am also not aware of any published work that presents an independent algorithm to add reset for AFSMs to posttechnology mapped designs.

4.3 Algorithm

An algorithm is described that automatically synthesizes reset logic for sequential AFSMs regardless of the specification style or method used to generate the circuit. The inputs to the algorithm include (a) the sequential circuit technology mapped to single output static gates, (b) the Boolean behavior of static gates available in the cell library, and (c) Boolean logic levels for all signals in the reset state. Two additional inputs may optionally be included: a set of performance critical paths, and a list of primary inputs that remain undefined upon application of system reset. No design specification information is required.

The algorithm presented here is based on the following observations: (a) State variables are implemented in a circuit that use static logic gates with topological feedback. (b) Feedback creates cycles in the circuit. (c) Cycles and undefined inputs are the only sources of undefined signals in a sequential circuit. (d) Any gate in a cycle can be used to reset the entire cycle.

This algorithm focuses on identifying combinational cycles in a circuit which needs to be explicitly reset, and then selecting an optimal location in such cycle to add reset. The cost of each solution is generated based on heuristics that employ logical effort to estimate performance and energy costs. The determination of which cycles need to be reset is performed by simulating the design and determining which nodes remain undefined.

Finding a reset configuration is not necessarily simple because circuit cycles may interact i.e., resetting one cycle, may automatically reset interacting cycles.

The algorithm consists of two main sections. The first section (Sec. 4.3.1) identifies and resets the *cycles* and the second one does the same for *paths* (Sec. 4.3.2). Multiple solutions are generated by adding reset signal to each gate of a cycle (path) as described in Sec. 4.3.3. Each generated solution is compared against the others to obtain the least-cost solution using optimization heuristics based on logical effort (Sec. 4.3.4).

4.3.1 Generate Cycles to Reset when PI's are Defined

A circuit is represented as a directed graph G where G is the pair (V,E). V is a finite set of vertices v, representing single output combinational gates, primary inputs (PIs), and primary outputs (POs) of a circuit. E is a set of edges e mapping $V \times V$ where e is an ordered pair (v_i, v_j), where v_i is the vertex output and v_j is an input to a vertex. P is a set of paths p where $p \in P$ is defined as an ordered sequence of vertices $\langle v_i, ..., v_j \rangle$, where $\forall v_k \in p$, no vertex $v_k \in p$ is repeated, $v_k \in V$, and where there is an edge $e_k \in E$ between each adjacent vertex in path p. Also, the path p is represented as $V_i \xrightarrow{p} V_j$. C is a set of cycles c where $c = p_i \in P$ and there exists an edge e_j that maps between the first and last element of path p_i.

Each path and cycle has two associated edge sets, internal edges E_{Int} and external edges E_{Ext}. E_{Int} is the set of edges between each vertex in a path or cycle, and E_{Ext} is the set of edges $e_i : (v_i, v_j)$ where $v_j \in p \wedge v_j \notin E_{Int}$. Note that external edges include fan-in but not fan-out connectivity.

Fig. 4.1 shows an example cycle consisting of the path $\langle v_1, v_2, v_3, v_4, v_5, v_6 \rangle$. The internal edge set E_{Int} equals $\{e_1, e_2, e_3, e_4, e_5, e_6\}$ and the external edge set E_{Ext} in the example is $\{e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}\}$.

Each vertex $v_i \in V$ is assigned a value in the set $\{0, 1, x\}$, where x is the logic level for an undefined vertex, while 0 and 1 are defined logic level. The value of each edge e_i is derived from the value of vertex v_i where $e_i : (v_i, v_j)$. The convention that a vertex (and its associated fanout edges) is *defined* when it has a Boolean value of either 0 or 1, otherwise it is said to be *undefined*, and is assigned the value x. Since gates (vertices) are single output,

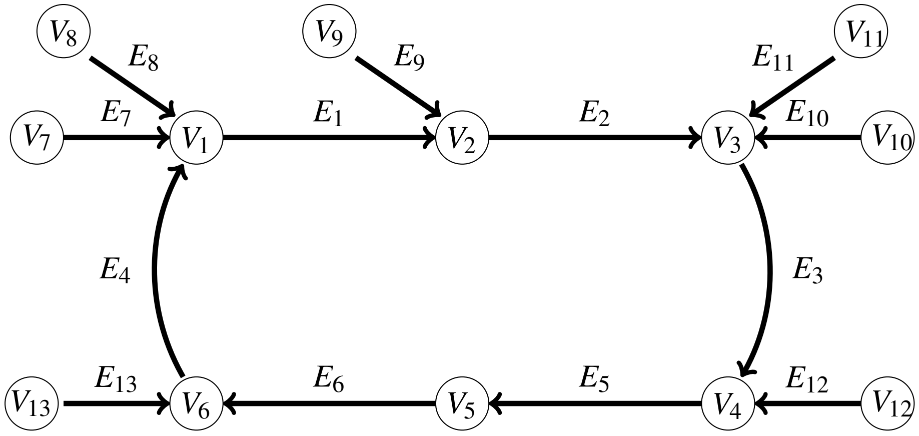

Figure 4.1: A cyclic directed graph example.

the state of all nets in the system are defined once all vertices are defined. One required input to the algorithm is the Boolean logic level for all vertices in the reset state.

Definition 1 An input of a static single output gate is said to have a *controlling value* if it uniquely determines the output of the gate independent of other gate inputs.

If an input to a gate does not uniquely determine the output of the gate, it is a *noncontrolling value*. If all inputs are noncontrolling, then a subset of the inputs must be defined to define the output.

Axiom 1 The output of a static combinational gate is defined if all the gate inputs are defined.

Lemma 1 For input set I, the output of a single output static combinational gate is uniquely controlled by input $i_i \in I$ when all other gate inputs $i_j \in I$ are assigned to noncontrolling values and the output remains undefined.

Proof: This holds due to Axiom 1. Since only one gate input is undefined, once that signal becomes defined all gate inputs are defined and the output must switch to a known value of 0 or 1 based on the combinational function.

Lemma 1 allows any complex static gate to be represented as a simple inverter or buffer based on the value of input i_i if, when all other gate inputs are defined, the output is still undefined. Fig. 4.2 shows examples of this representation. The NAND gate acts as a simple

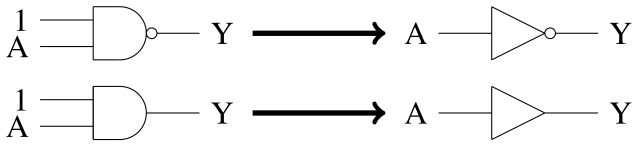

Figure 4.2: Gate conversion example for Lemma 1.

inverter when i_i is signal A since all other signals are at a high voltage. Similarly the AND gate can be modeled as a buffer.

Lemma 2 If all edges in E_{Ext} of a path (cycle) are set to logic 0 or 1, then the path (cycle) can be represented as a path (ring) consisting of inverters or buffers.

Proof: Follows Lemma 1.

Theorem 3 If all the edges in E_{Ext} for a cycle are defined then all the signals in E_{Int} are either defined or undefined.

Proof: Assume that the set of external inputs E_{Ext} for cycle c are set such that none of the vertices in c are defined. In this case, all internal edges E_{Int} of the cycle are undefined. On the other hand assume the case where a single edge $e_i \in E_{Ext}$ is modified such that $e_i = (v_i, v_j)$ is controlling or the value of vertex v_j becomes defined. This results in edge $e_j \in E_{Int}$ becoming defined. According to Axiom 1, this results in the vertex (gate) v_k becoming defined where $e_j = (v_j, v_k)$. This continues around the ring until all vertices (gates) become defined to a Boolean value.

Theorem 4 If the set of vertices in a cycle c_0 is a proper subset of the vertices in another cycle c_1, then the cycle c_0 contained in the bigger cycle c_1 must be reset to reset both the cycles.

Proof: Assume all edges in both the rings are undefined. Let V_0 and V_1 be the set of vertices in cycles c_0 and c_1, respectively, and let E_{Int_0}, E_{Ext_0} and E_{Int_1}, E_{Ext_1} be the internal and external edges. The smaller ring is the ring where $V_i = V_0 \cap V_1$. This results in a condition where Theorem 3 does not hold for a vertex v_k, since $e_i = (v_i, v_k) \in E_{Int_0}, E_{Ext_1}$ and $e_j = (v_j, v_k) \in E_{Int_1}, E_{Ext_0}$ are both undefined. Assume c_0 is the smaller cycle, and e_j

is the undefined external input to vertex v_k. Assuming that e_j is defined, cycle c_0 can be reset, which results in the edge e_i becoming defined. This results in all external edges in E_{Ext_1} becoming defined, so that Theorem 3 can hold on the larger cycle. Since c_0 is a proper subset of $c_1, \exists e_l = (v_i, v_l)$ where $v_l \in V_1 \wedge v_l \notin V_0$, the larger cycle c_1 automatically gets reset due to the smaller cycle.

Theorem 4 allows the number of vertices (gates) that require reset to be smaller than the number of cycles in a sequential circuits that are undefined without reset. Also note that sequential circuits may have many cycles that overlap each other in various ways, not just as proper subsets. Theorem 4 may also be extended to reset interacting cycles that are not nonproper subsets. However, the code developed here only applies optimization of multiple cycles according to this theorem, and thus may not generate the solution with the fewest number of reset vertices (gates). Such an extension is left for related work. Further, due to Theorem 4, this algorithm generates reset logic for cycles based on the smallest vector set cardinality first. This ensures that larger concentric cycles automatically get reset by their smaller cycles getting reset.

4.3.2 Generate Paths to Reset when PIs are Undefined

This section removes the initial condition which requires all the PIs to be defined during cycle reset generation. This is necessary to ensure that all signals in E_{Ext} are defined. The netlist generated in the previous section is used, since it guarantees all the cycles in the circuit are defined iff all the PIs are defined. The reset problem now becomes a path based rather than a cycle based problem.

Lemma 5 For output o and input set I of a single output static combinational gate, if o is undefined then at least one of the inputs in $i \in I$ is undefined.

Proof: Applying transposition to Axiom 1.

Definition 2 An undefined path is a path where $\forall e_i \in E_{Int}$, the value of e_i is undefined.

Lemma 6 Consider there are no undefined edges in a circuit when all the PIs are defined. If a PI is marked undefined, and this results in a set of POs of the circuit being undefined, then there exists at least one undefined path from the PI to each undefined POs.

Proof: The input netlist of this section considers that if all the PIs are defined then all the wires in a circuit including the POs are defined. Hence, if a PI is undefined which results in a subset of the POs being undefined, then there must be an undefined path from the PI to each undefined PO.

The path may be represented as a set of inverters, and resetting any vertex results in all downstream vertices becoming defined. This can be shown using a similar approach as is done for cycles. Therefore, to reset a path, any vertex in the path may be reset.

4.3.3 Gate Modifications for Reset Insertion

Each gate in a path (cycle) is a potential candidate for reset insertion. Therefore, every gate is evaluated for the cost and potential of adding reset to that gate. The reset signal must be inserted as a controlling value to the gate, and the resulting gate must be a member of the static gate library employed in the design.

Three separate transformation cases are given below which may be employed to insert a reset signal into a cycle. Out of these three cases only the first two transformation cases can be applied to paths. Selecting the appropriate case is based on the type of optimization being performed and the type of gate being modified for reset addition.

- **Case 1:** If the gate is an inverter (buffer), it gets converted into a NAND or NOR (AND or OR) gate depending on the required value of the output of the gate after reset. The asserted reset signal becomes the controlling value for the gate.
- **Case 2:** This is a generalized condition for case 1 that adds reset to any static single output gate. The input $e_i \in E_{Int}$ in the path (cycle) is identified. The behavior of the gate is represented in a sum-of-products format. If the output of the gate is inverting, and the desired output is 1, then an active low reset is ANDed with edge (signal) e_i. If the desired output is 0, then an active high reset signal is ORed with the full gate function. A similar transformation is performed for noninverting gates. The new gate is used as a possible solution if it is present in the cell library.
- **Case 3:** If the vertex (gate) v_i is an inverter, and the inverter drives an edge (gate) that is not an element of the path (cycle), this transformation can be employed. This transformation creates a duplicate inverter v_j, disconnects v_i from the cycle, and

applies the case 1 transformation to the new inverter v_j. This case is only applied to performance optimization of cycles requiring reset as defined in Sec. 4.3.1.

Fig. 4.3 shows a circuit implementation example before reset addition. Fig. 4.4 and 4.5 illustrates the application of these transformations on the circuit shown in Fig. 4.3. The case 1 example can be seen where the inverters U3 and U7 have been converted into NOR gates in Fig. 4.4 using an active high reset because the desired output values for these gates are 0. Case 2 is not directly illustrated because it results in an inferior solution according to logical effort. However, assume U2 is being evaluated. This is an inverting gate, and the desired output value of the gate is 1. Active low reset is ANDed with $rr \in E_{Int}$, changing U2 from an AOI21 gate into an AOI31 gate. Since this gate is present in our library, it is a valid transformation. However, because this solution is of higher cost than a case 1 transformation on gate U3 in this cycle, it is not used in the final solution. The case 3 transformation is illustrated with the new gate U11 added to the design in Fig. 4.5 when the performance path $lr \xrightarrow{p} rr$ is provided. The new gate although becoming a branching load to the performance path, adds more area to the design. Since the structure of the circuit is modified, it is possible that this transformation adds a hazard to the circuit. Hence, in our design flow a formal verification step is performed to ensure hazard fidelity of the design.

A special condition applies to all of these design cases when the input edge in a cycle passes through an inverter that is inside a gate. The case 2 transformation is applied as usual. Additionally, the gate is split into two gates with the inverter becoming an explicit external gate that is added to the cycle. Case 1 is then applied to the inverter, and case 2 is applied to the second gate. This is illustrated with the design shown in Fig. 4.6. The inverter bubble has been split into a separate inverter in Fig. 4.7 with the case 1 transformation applied.

4.3.4 Optimization Heuristics for Selecting the Best Solution

Power and performance optimizations are based on heuristics that use logical effort [55]. Logical effort theory provides a first-order approximation of the sizes (power) of the gates and the delay for a circuit path (performance). The optimization uses a priority based approach with delta logical effort having the highest priority and performance/power optimization having the lowest priority. If a heuristic solution is better than the previous best solution then no other solution costs are compared.

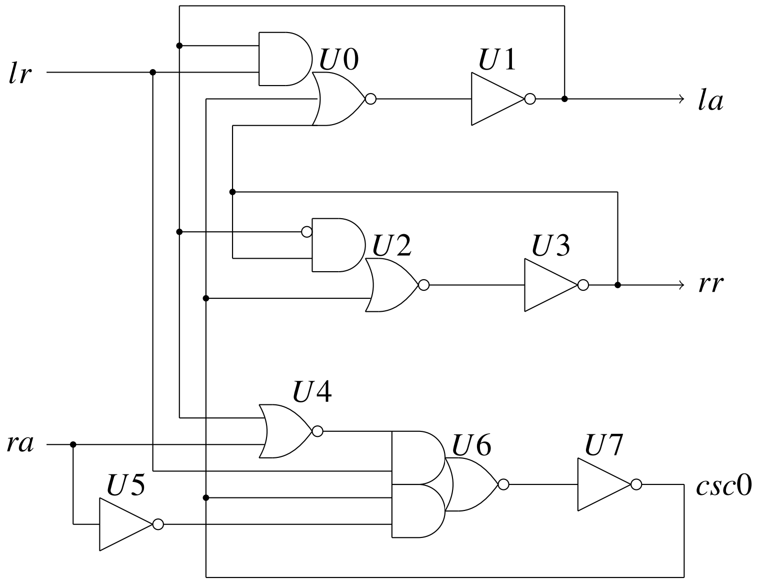

Figure 4.3: Example 1: Circuit implementation before reset.

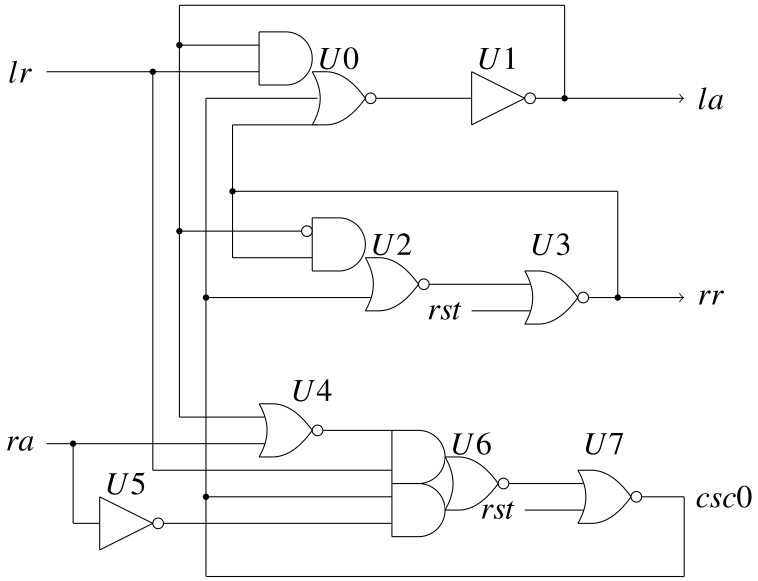

Figure 4.4: Example 1: Circuit implementation with power optimization.

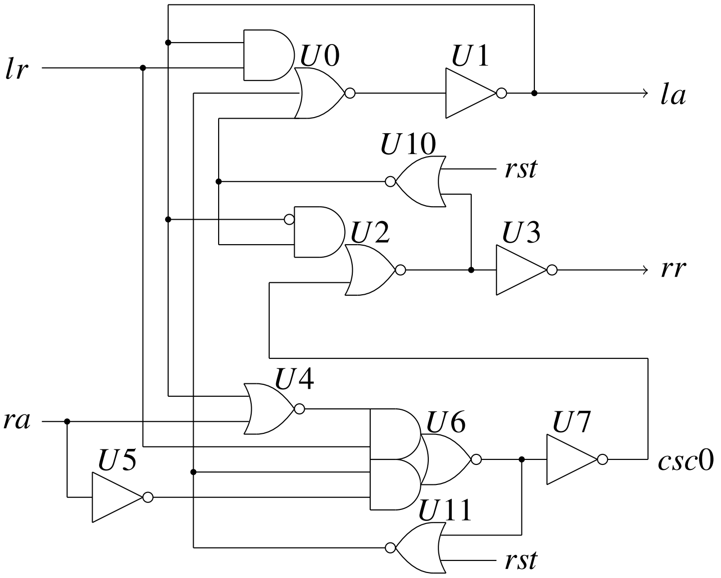

Figure 4.5: Example 1: Circuit implementation with performance optimization.

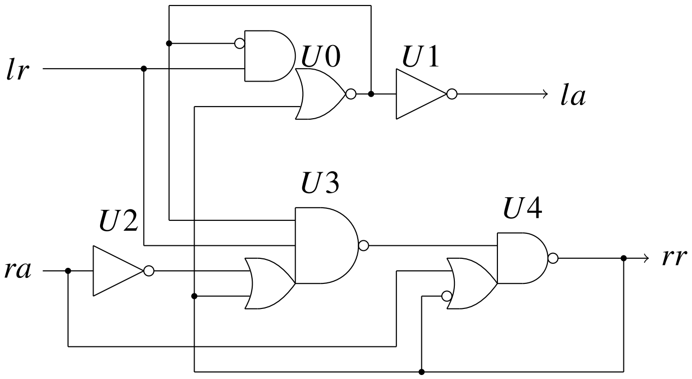

Figure 4.6: Example 2: Circuit implementation before reset.

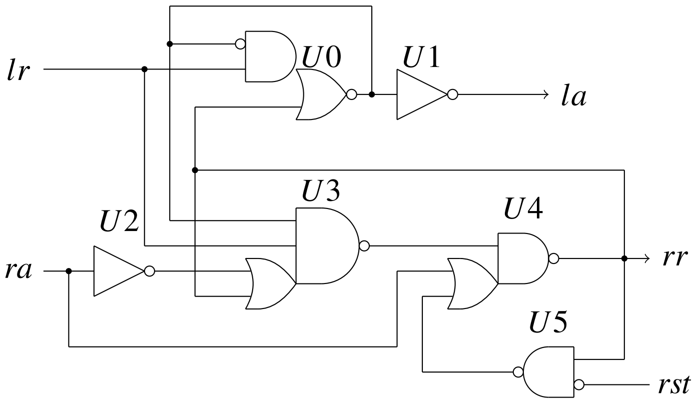

Figure 4.7: Example 2: Circuit implementation with power/performance optimization.

4.3.4.1 Delta Logical Effort

Logical effort often favors simpler gates over more complex gates due to their high cost. Hence, the first step of optimization looks at the relative increase in logical effort of modifying any gate which is named as delta logical effort (ΔLE) and is calculated as

$$\text{Cost} = \Delta\text{LE} = \text{New LE - Old LE.} \tag{4.1}$$

4.3.4.2 Relative Load on a Gate

Logical effort can be used to estimate the necessary drive strength (also referred to as size) of a gate by calculating the gate's output load. The load estimate is calculated by computing the sum of the logical effort of a gate and the logical effort of the inputs of all the successor gates to which the wire goes. This heuristic penalizes the modification of a gate which drives a big load and thus prefers simpler gates with small output load. The equation for this heuristic is

$$\text{Cost} = \text{LE of gate} + \text{LE load on gate output.} \tag{4.2}$$

4.3.4.3 Performance or Power Optimization

Logical effort theory defines path delay as

$$\text{Delay of N-stage path} = N * F^{1/N} + P$$

where $F = G * B * H$. Here, logical effort G is the product of the logical efforts of the logic gates along the path, B is the product of the branching effort at each stage along the path, P is the parasitic delay of the gate, and electrical effort H is the ratio of the capacitance loading the last stage of a path to the input capacitance of the first stage of the network.

The solution for this step is selected based on the optimization selected by the user. The heuristics to calculate the cost of the solution for each optimization are described below.

- *Performance optimization* - All three reset transformation cases are applied for performance optimization. However, case 3 is only applied on performance critical paths that are optionally supplied by the user. For the handshake controller examples, $lr \xrightarrow{p} rr$ and $lr \xrightarrow{p} la$ are provided as performance critical paths.

The quality of the solution for each cycle is the delay for each input to output path in the design. The total solution is the sum of the delays of the paths in the design. Hence, the final solution is selected based on the least overhead cost which is calculated as

$$\text{Performance cost} = \sum_{\text{all paths}} N * F^{1/N} + P. \tag{4.3}$$

This heuristic assumes electrical effort H to be 1. This can result in suboptimal results if the fanout load of the circuit output is large.

- *Power optimization* - Power consumption of a design depends on the total capacitance of the circuit that needs to be switched. The capacitance is approximated with the logical effort G of each gate, where a higher logical effort implies a larger input capacitance. Thus, the total solution for power optimization is calculated as

$$\text{Power cost} = \sum_{\text{all paths}} \sum_{0}^{i-1} \text{Avg. input LE for gate } V_i \text{ on a path.} \tag{4.4}$$

4.4 Examples

This section describes the application of the reset addition algorithm on two examples. The first example describes different gate modifications for power and performance optimizations, while the second example shows the reset addition problem applied to noninverting gates or gates with inverted inputs.

4.4.1 Example 1

The initial circuit for this example is shown in Fig. 4.3. In this example, lr, ra, U1, U3, U7 have a logic level 0 while U0, U2, U4, U5, U6 have a logic level 1 at reset state. It consists of six cycles: $\langle U0,U1\rangle$, $\langle U2,U3\rangle$, $\langle U6,U7\rangle$, $\langle U0,U1,U2,U3\rangle$, $\langle U0,U1,U6,U7\rangle$, and $\langle U0,U1,U6,U7,U2,U3\rangle$.

Cycle $\langle U0,U1\rangle$ does not need to be reset because it is defined by signals in the external signal set E_{Ext} = {lr, rr, csc0}. Of the other five cycles, only two need to be reset due to shared paths in the cycles. By reseting cycle $\langle U2,U3\rangle$ and $\langle U6,U7\rangle$, the (U2,U3) and (U6,U7) edges become defined, resetting the remainder of the cycles.

Fig. 4.4 shows the result of applying the power optimization heuristic. The case 1 optimization results in the best solution for both $\langle U2, U3\rangle$ and $\langle U6, U7\rangle$. This optimization modifies U3 and U7 from inverters to a NOR gates.

Results of performance optimization for the same circuit are shown in Fig. 4.5. The path from $lr \xrightarrow{p} rr$ ($\langle U6, U7, U2, U3\rangle$ and $\langle U0, U1, U2, U3\rangle$) and $lr \xrightarrow{p} la$ ($\langle U0, U1\rangle$) are defined as performance paths. Thus, both cycle 2 and cycle 3 are candidates for case 3 optimizations, and can push the added complexity of the reset gates off the critical path. Gates U3 and U7 are first duplicated to add U10 and U11 in the feedback of both these cycles. These duplicate gates are then converted to NOR gates that reset the cycles.

This example so far has assumed that the PIs are all defined. Consider the power optimization case when input lr is initially undefined. The algorithm then starts with the circuit of Fig. 4.4, marking lr as undefined. This results in the output of U0 and U1 being undefined resulting in la output being undefined. Applying the optimizations results in the gate U1 being changed into a NOR gate with reset as shown in Fig. 4.8. If the performance optimization solution is considered then the path $\langle U6, U7, U2, U3\rangle$ is also undefined resulting in gate U7 being converted into a NOR gate with reset. Note that this results in an inferior solution since there are two NOR gates performing the same task as shown in Fig. 4.9. Hence, the application of undefined input solution is the best for power optimization, but can result in an inferior solution for performance optimization in certain cases.

Petrify is used to apply reset to this sample circuit. Reset is achieved by using generic AND and OR gates as shown in Fig. 4.10. U10, U11 and U12 are added to initialize cycle 1, cycle 2, and cycle 3, respectively. Notice that gate U10 is not required, resulting in an inferior solution in terms of power and performance.

4.4.2 Example 2

The second example circuit is shown in Fig. 4.6. In this example, lr, ra, U1, U4 have a logic level 0 while U0, U2, U3 have a logic level 1 at reset state. It consists of four cycles $\langle U0\rangle$, $\langle U4\rangle$, $\langle U3, U4\rangle$ and $\langle U0, U3, U4\rangle$. Assuming the PIs are defined, only the $\langle U4\rangle$ cycle needs to be reset, because reset values for lr and rr define Gate U3. Fig. 4.7 and Fig. 4.11 show

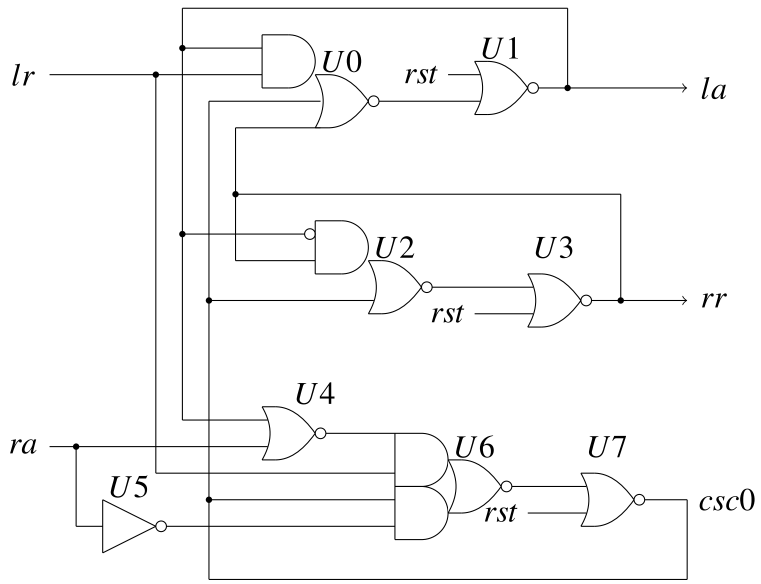

Figure 4.8: Example 1: Power optimization with undefined inputs.

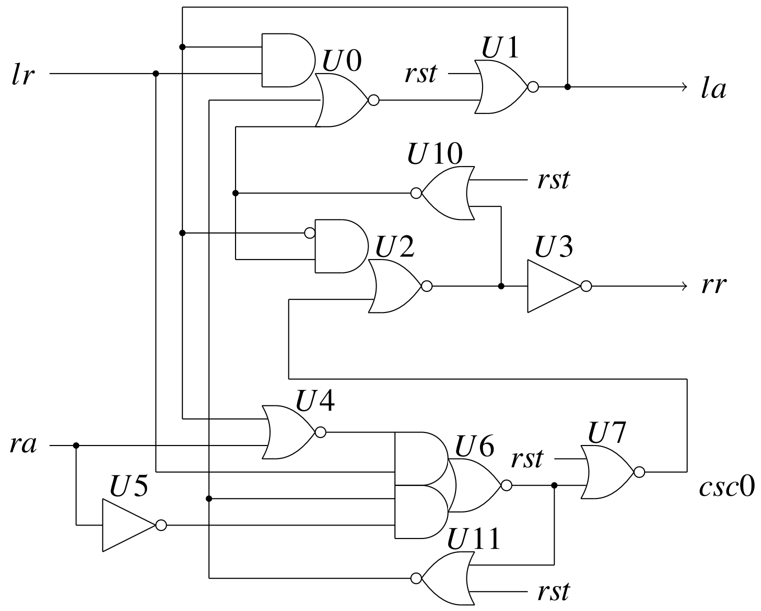

Figure 4.9: Example 1: Performance optimization with undefined inputs.

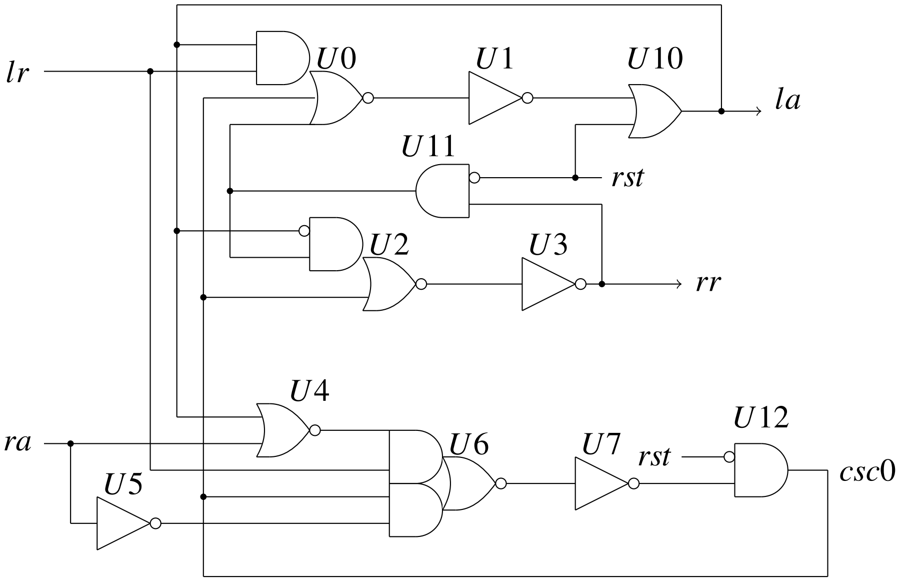

Figure 4.10: Example 1: Circuit implementation using Petrify.

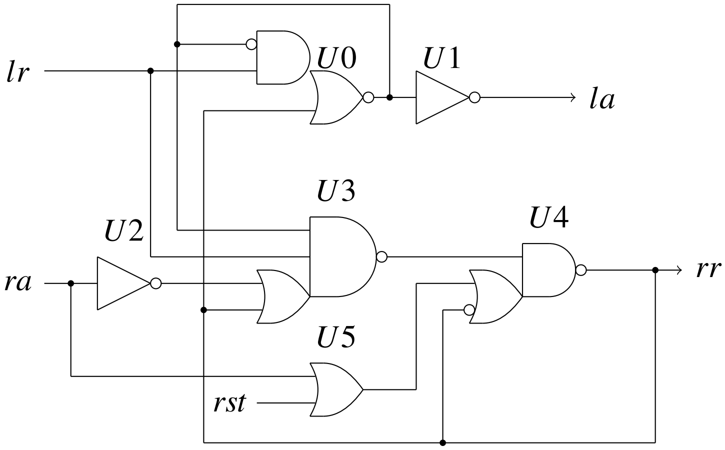

Figure 4.11: Example 2: Circuit implementation using Petrify.

the solution for this algorithm and Petrify, respectively. The optimized circuit generated by this reset algorithm is the same for both power and performance optimizations since the reset is not on a critical path. Petrify adds the OR gate U5. This increases the latency on the $ra \xrightarrow{p} rr$ resulting in a 10 percent increase in the backward latency and thus a 5 percent increase in the cycle time.

4.5 Results

The results of adding reset initialization with this algorithm are compared against Petrify. Benchmark circuits for GCD, PostOffice and PSCSI were employed as well as a set of 128 untimed 4-cycle handshake controllers generated by concurrency reduction [1, 56, 52, 35]. Each design in the controller set is tested as a four deep FIFO. All of these designs are synthesized and technology mapped with Petrify with and without reset addition. Our algorithm is applied to these circuits without reset. Petrify adds generic gates for reset addition, hence for comparison these gates are technology mapped using a script. The technology mapping is applied to the academic Artisan library for the IBM 65nm process.

This algorithm resulted in functionally correct circuits for all designs to which power optimization is applied, while application of performance optimization resulted in two circuits that failed due to hazards that were introduced. Petrify failed to generate a working circuit for one of the FIFO controllers since it assumed all the inputs to be defined at logic level 0 upon reset.

Performance, power, and area comparisons are performed by using timing driven optimization in commercial EDA tools. The flow is structured and automated in a way that produces results which are as fair as possible. The flow uses Design Compiler for sizing, SoC Encounter for place and route, and Modelsim and Primetime for performance and power evaluation using VCD and SPEF files.

The example set ranges in complexity from 4 to 71 gates, and up to 77 cycles. The maximum runtime for the algorithm is less than 3 seconds for the gcd example, which contains 11 inputs, 9 outputs, 71 gates, and 22 cycles. Critical paths from $lr \xrightarrow{p} rr$ and $lr \xrightarrow{p} la$ were provided for the 128 FIFO controllers.

Tables 4.1 and 4.2 show the average benefits for both optimizations with respect to Petrify. Performance optimization results in an improvement of 8 percent, 12 percent and

Table 4.1: Results comparison for benchmark circuits.

	Petrify			Power Opt.			Performance Opt.			Power Benefits			Performance Benefits		
Benchmark Circuit	Area (um^2)	Energy/ token (pJ)	Sim. Time (ns)	Area (um^2)	Energy/ token (pJ)	Sim. Time (ns)	Area (um^2)	Energy/ token (pJ)	Sim. Time (ns)	Area	Energy/ token	Sim. Time	Area	Energy/ token	Sim. Time
gcd	298.3	0.50	303.76	287.2	0.50	297.56	285.4	0.46	298.33	1.04	1.00	1.02	1.05	1.08	1.02
postoffice-rcv-setup	36.0	0.02	87.14	27.4	0.02	85.70	31.7	0.03	84.98	1.31	1.20	1.02	1.14	0.89	1.03
postoffice-sbuf-send-ctl	132.0	0.38	317.85	100.3	0.30	315.62	109.7	0.32	319.69	1.32	1.28	1.01	1.20	1.19	0.99
pscsi-isend	185.2	0.36	244.45	172.3	0.35	276.23	198.0	0.43	256.25	1.07	1.03	0.88	0.94	0.84	0.95
pscsi-trcv-bm	114.0	0.19	143.07	99.5	0.15	136.08	98.6	0.15	135.37	1.15	1.24	1.05	1.16	1.29	1.06
pscsi-tsend-bm	140.6	0.28	202.83	134.6	0.25	213.86	139.7	0.26	208.73	1.04	1.11	0.95	1.01	1.10	0.97
pscsi-tsend	147.5	0.24	203.12	145.7	0.25	191.28	145.7	0.25	191.28	1.01	0.99	1.06	1.01	0.99	1.06
Average Benefit										1.14	1.12	1.00	1.07	1.05	1.01

Table 4.2: Controller circuit comparison.

	Average Case		Best Case		Worst Case	
Optimization	Power	Performance	Power	Performance	Power	Performance
Forward Latency	1.00×	1.08×	2.33×	2.33×	0.63×	0.69×
Backward Latency	1.05×	1.12×	1.47×	1.72×	0.61×	0.71×
Cycle Time	1.03×	1.06×	1.39×	1.69×	0.54×	0.54×
Area	1.21×	1.12×	1.91×	1.66×	0.70×	0.69×
Energy/token	1.24×	1.12×	2.19×	1.84×	0.64×	0.64×

6 percent in forward latency, backward latency and cycle time for the 128 FIFO circuits. The benchmark circuits show only 1 percent improvement in performance (reported as simulation time – SimTime). A 12 percent reduction in area and energy/token for the FIFO controllers is observed, as compared to a 7 percent and 5 percent reduction in area and energy/token, respectively, for the benchmark circuits.

Power optimization results in no improvement in forward latency, and minor improvements in backward latency and cycle time for FIFO controllers as well no performance benefit (SimTime) for the benchmark circuits. However, there is a significant improvement in terms of area and energy. A 21 percent and 24 percent reduction in area and energy/token, respectively, are seen for the FIFO controllers, while a 14 percent and a 12 percent reduction is seen for the benchmark circuits.

Detailed results for application of the optimizations and equivalent numbers for 128 pipeline controllers generated by performance optimization, power optimization and Petrify are shown in Appendix A.

4.6 Summary

Sequential circuits require a reset signal to initialize them to their correct starting state. An algorithm is developed and implemented in C++ to generate reset logic for asynchronous finite state machines. The algorithm defines the relationship between reset and topological cycles in a circuit. The new algorithm also provides heuristics to optimize the reset logic for power or performance. It requires that the design has been technology mapped to the desired implementation library, and that the library consists of single output static logic gates. Inputs to the algorithm include the design netlist, the logic level of all circuit nets, and the behavior of the gates in the technology library. Optional inputs include a set of critical paths for performance optimization, and a set of inputs that may initially be undefined upon reset.

The algorithm is applied to a set of 7 large benchmark circuits and a set of 128 pipeline controllers that are configured into linear FIFOs. The designs range in complexity of up to 71 gates and 77 cycles. Maximum runtime for the tool is less than 3 seconds. Results are compared against Petrify. Performance heuristics show just a 1 percent performance improvement for the benchmark circuits. The FIFO designs show a 6 percent performance improvement and a 12 percent and 8 percent improvement for backward and forward latency. Power heuristics show an average improvement of 14 percent and 12 percent in area and energy per token for the benchmark circuits, and an average area and energy per token improvement of 21 percent and 24 percent for the FIFO controllers.

The algorithm is agnostic to how the circuit is implemented and technology mapped, so it can be used with any of the synthesis engines as well as with hand designed circuits. For the first time reset can now become part of any asynchronous finite state machine design automation flow.

CHAPTER 5

TIMING PATH DRIVEN CYCLE CUTTING FOR SEQUENTIAL CIRCUITS [1]

Asynchronous architectures and design methodologies are an excellent means of generating fast, low power circuit topologies. But these circuits have numerous topological feedback paths because they are sequential. The power and performance of such a design can be significantly improved by correctly sizing gates and transistors through timing driven optimization algorithms. For example, bundled data asynchronous controllers are very sensitive to timing driven optimizations and can easily show factors of improvement of $5\times$ in energy-delay product between well optimized and poorly optimized circuits.

The algorithms in current *application specific integrated circuit* (ASIC) *computer aided design* (CAD) tools do a good job of sizing gates to optimize the power and performance of a design. Timing driven optimizations are performed in the synthesis and place and route portions of the commercial ASIC design flow. However, such algorithms do not support cyclical timing dependencies and must operate on *directed acyclic graphs* (DAGs). Thus, as designed, commercial CAD does not support asynchronous circuits unless the sequential modules are precharacterized as DAGs and correct timing constraints are provided.

There are, therefore, two approaches to correctly size gates to optimize the power and delay and validate correct circuit timing for sequential asynchronous controllers. The first approach is to develop custom timing tools that operate on cyclic circuits. Such an approach normally unrolls the cycles using Shannon decomposition or other methods to produce time varying dependencies on the feedbacks to deal with the interdependencies of the cyclic graphs [58, 42]. These tools, while potentially very accurate, have large run times and are not directly supported by commercial ASIC CAD tool flows. Thus, a custom tool flow

[1] Patent No. 8,365,116 has been issued for this work [57].

would need to be developed for gate sizing and timing driven place and route using this method.

The second method is based on tool flows that employ traditional commercial CAD to optimize and validate power, performance, and timing correctness [41, 59, 31]. These flows generate timing paths through the sequential circuit that include delay targets necessary for correctness and for meeting performance requirements. A circuit cannot be properly optimized or validated if any timing path is cut. So the sequential circuit is preprocessed to generate the timing arcs[2] that must be cut to convert the cyclic timing graph to a DAG. All the timing paths are preserved while generating these cycle cuts, which are then passed as constraints to the ASIC CAD tools. This approach is less accurate since it does not allow optimization of circuits based on cycle time like the first method. Calculating cycle time of sequential circuits involves cycles, which get cut using this approach.

Cycle cutting that preserves timing paths is a key component in a high-level flow for characterizing, synthesizing, and validating sequential asynchronous designs using traditional commercial CAD tools. Part of this flow generates timing constraints that are required for correct circuit operation [40]. These timing constraints are passed to the algorithms in this tool that are then employed for timing driven synthesis, place and route, and postlayout validation of asynchronous designs.

5.1 Key Contributions

- An algorithm for generation of cycle cuts constraints while preserving timing paths in the circuit timing graph. Concept for identifying paths that must be cut in a design in order to remove cycles, along with its associated algorithm is also presented. This enables the use of STA and timing driven sizing and optimization algorithms of commercial CAD tools on sequential asynchronous circuits.
- Cycle cut constraints generated by the algorithm allows us to control the performance of each pipeline stage of the circuit and thus automatically generate bundled data delays. This preserves the correct ordering of signals required for proper functioning of an asynchronous circuit.

[2] Timing arc - Timing path from an input to an output of a gate.

- The generation of cycle cut constraints is performed automatically, thus enabling automation of asynchronous circuit design using commercial CAD tools and flows.

5.2 Related Work

Combinational cycles are generally associated with sequential circuit designs like asynchronous circuits. Cycles can also be present in combinational logic, and some cyclic combinational circuits have been shown to substantially reduce area [60]. Since CAD in EDA tools require acyclic timing graphs, the problem of finding cycles and analyzing the combinational nature of circuits with cycles has been investigated [61]. Algorithms that generate an equivalent acyclic combinational circuit which reproduces all the combinational behavior of the original cyclic circuit have been proposed [62, 63, 64]. These approaches cannot be applied to sequential circuits because they change the sequential behavior when state-holding feedback of a circuit are removed. In order to support general sequential circuits built as combinational logic with feedback, the cyclic circuit must be represented as a DAG without modifying its structure or behavior.

The work most closely related to this work applies cycle cutting to the testing of digital circuits with feedback [65]. The problem is formulated as a covering problem with the set of arcs forming the cycle and the cycles present, with the goal being to find the minimal number of arcs to cut all the cycles. The drawback of this approach is similar to the algorithms in current commercial CAD tools which also cut cycles. A set of cycle cuts, even if they are minimal, creates a DAG, but timing driven optimizations cannot be performed because timing paths are cut.

A novel aspect of this work is the ability to specify paths which *must* be cut in a sequential circuit. The must-cut path capability supports a modular system level design style where external cycles can be specified to be cut in particular manner in local handshake control modules. I am not aware of any published cycle cutting algorithm that supports embedded constrained paths in graphs that both cannot be cut and that must be cut.

5.3 Background

Asynchronous design has largely received a cold reception from industry. Part of this reluctance has been due to the requirement that custom design languages and CAD flows have been necessary to design, optimize, and validate asynchronous modules and systems. The asynchronous methodology and CAD tool flow described in Chapter 3 supports modern programming languages like Verilog and ASIC CAD tools. They also support mixed-clocked and asynchronous design. It exploits the benefits of asynchronous circuits and also uses the ASIC CAD tool flow which are meant to develop synchronous (clocked) circuits [41]. One of the key components to enable this is the generation of cycle cut constraints for the cyclic asynchronous circuits.

This section describes the concept of timing constraint paths and their derivation from *relative timing* (RT) constraints. Classification of cycles and the constraints required during generation of cycle cuts and the benefits of its applicability and generality is described on a 4-deep bundled data pipeline using the circuit realization of the timed burst-mode protocol shown in Fig. 5.1

5.3.1 Timing Constraint Paths

The flow in this dissertation is based on RT [20]. Each relative timing constraint pod and poc is mapped onto nodes in the circuit. Paths through the design which topologically connect a pod to a poc are created (e.g., for $lr\uparrow \mapsto la\uparrow$ in Fig. 5.1). The full set of paths, called *constraint paths*, form the set Φ. A subset of Φ, called Φ_c, contain the *performance critical* constraints. Constraints Φ_c include bundled data datapath constraints (Fig. 2.3), which determine pipeline frequency. The set $\Phi_n = \Phi - \Phi_c$ thus consists of the constraint paths which are not performance critical. A subset of Φ, usually Φ_c, is used for timing driven synthesis and place and route.

Bundled data pipelines (Fig. 2.3) have similar data timing requirements to clocked design (Fig. 2.5). Let $\Phi_{bd} = LC_i/lr\uparrow \mapsto R_{i+1}/D + margin \prec R_{i+1}/clk\uparrow$ be the general bundled data constraint where $\Phi_{bd} \in \Phi_c$.

Mapping the above constraint to the LC circuit implementation in Fig. 5.1 results in the constraint paths $[lr\uparrow\ la_{-}\downarrow\ ck\uparrow]$ for pod to poc$_0$ and $[lr\uparrow\ rr_{-}\downarrow\ rr\uparrow]$ for pod to poc$_1$. Guaranteeing that these paths exist uncut by the cycle cutting algorithm is essential for correct functionality and to correctly size and optimize the design for timing.

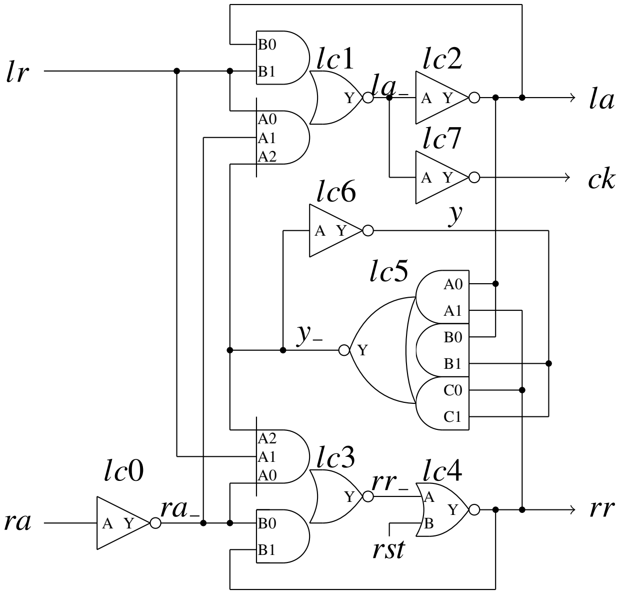

Figure 5.1: LC circuit implementation[3].

The specific timing constraint paths passed to each sequential block depends on the architecture and the protocols that are being used. Note that these paths are user inputs to the algorithm presented in this dissertation.

5.3.2 Classification of Cycles

There are two classes of cycles that exist in an asynchronous design. They are illustrated in Fig. 5.2:

1. **Local cycles:** Cycles which are present in a single design module. These cycles can be found by evaluating the connectivity of the module. Two of the local cycles of this circuit are $lc1$–$lc2$–$lc1$ and $lc3$–$lc4$–$lc3$.
2. **Architectural cycles:** Cycles which are present at a higher levels of hierarchy of a design through multiple design modules. LC_0/rr–LC_1/lr–LC_1/la–LC_0/ra–LC_0/rr is an example of an architectural cycle.

The request acknowledge handshake protocol on the channel between two LC blocks naturally creates architectural cycles. The number of these cycles can grow exponentially based on the architecture of the circuit. To prevent this, common cut points can be

[3] This figure is the static combinational gate implementation for the RT controller specification in [39].

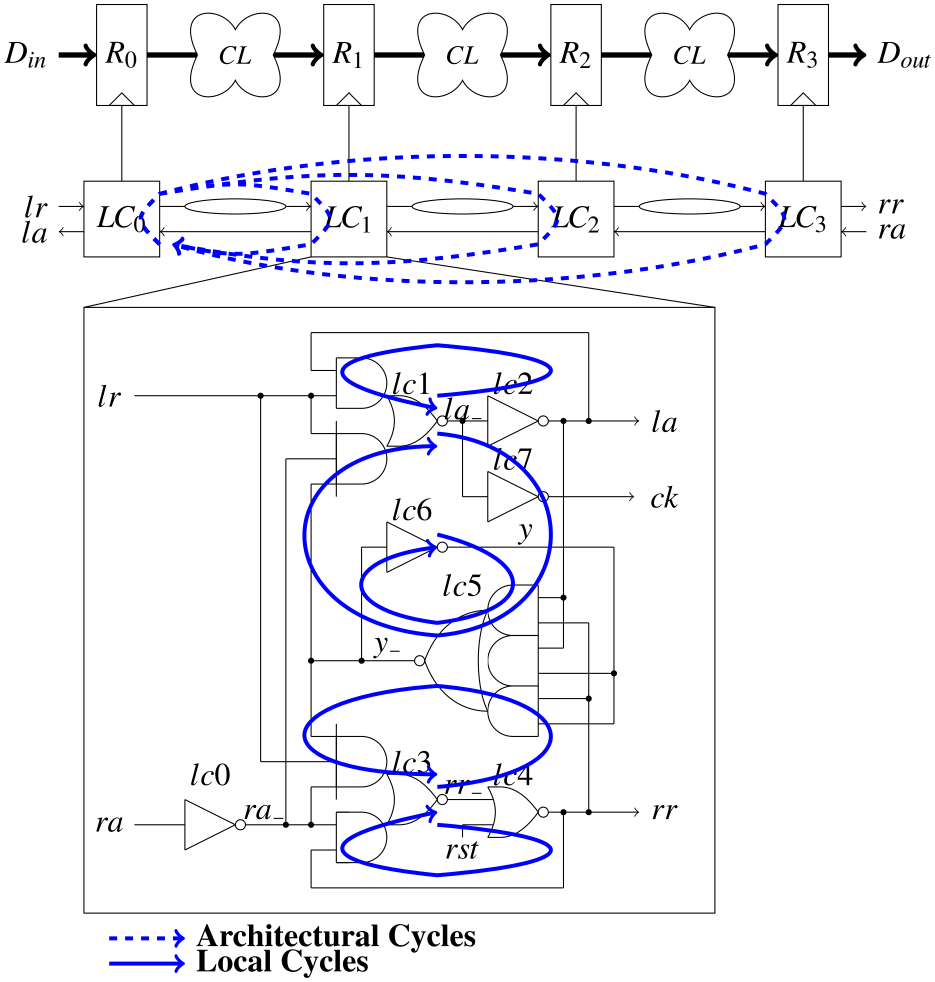

Figure 5.2: Classification of cycles example.

identified at the architectural level which enables cycle cuts to be generated at the module characterization level. This enables modules to be seamlessly connected without generating architectural cycles.

For example, Fig. 5.2 shows some architectural cycles created to the right of the LC_0 handshake controller. Similar cycles are present between connected sets of LC blocks. A common set of paths can be identified, which when guaranteed to be cut, lead to the removal of most of the architectural cycles. For this example, generating cuts for all the $ra \rightarrow rr$ paths for each LC module result in all the architectural cycles on the right of the LC_0 module being removed. Similarly, cutting the $lr \rightarrow la$ path in LC can get rid of all the architectural cycles on the left of a linear pipeline.

Architectural cycles can be specified to be cut locally in a sequential asynchronous control module. Localizing constraints this way simplifies the problem of cutting architectural cycles. Paths for architectural cycle cuts are the dual of the timing paths that cannot be cut. Architectural cycle cut paths are specified as **must-cut** path set Θ_m that the algorithm applies to a sequential module. These must-cut paths are represented in Θ_m as $A \nrightarrow B$, or $ra \nrightarrow rr$ in our LC example. Must cut paths may not be a subset of any path in the performance critical paths Φ_c, so $\forall p \in \Theta_m, q \in \Phi_c, p \nsubseteq q$. All architectural cycles of Fig. 5.2 are cut by making $ra \nrightarrow rr$ in module *LC* a must-cut path by placing it in set Θ_m.

5.3.3 Benefits of Correct Cycle Cutting

The LC circuit in Fig. 5.1 is used as an example to show the power, area, and performance benefits of applying timing driven cycle cutting using the flow in [41]. A 4-deep pipeline (Fig. 2.3) is implemented in this evaluation with the datapath removed; only the request and acknowledge control channels are connected.

Three local constraint paths are employed in the cycle cutting algorithm: $\{lr \rightarrow rr, lr \rightarrow la\} = \Phi_c$, and $\{\Phi_c, lr \rightarrow y\_\} = \Phi$. One must-cut path is provided: $\Theta_m = \{ra \nrightarrow rr\}$. The two constraint paths in Φ_c employ max_delay constraints. The optimization goal for these performance critical paths is to generate the fastest circuit (having the smallest max delay) without negative slack. The $lr \rightarrow y\_$ constraint is not performance critical but is necessary for correct functionality. It employs a min/max pair of constraints set to 1× and 1.7× the largest of the Φ_c constraints.

The circuits are then synthesized with *Design Compiler* (DC) employing optimization for power and performance to the Artisan 65nm academic library. The designs are then placed and routed with SoC Encounter to determine layout area and parasitics. The numbers for forward latency, backward latency, and cycle time are generated by simulating the postrouted design using *standard delay format* (SDF) back annotation. The step size of the sampling inputs and outputs for simulation is 10ps. A test set feeding 50 tokens to the design and then receiving it is used to generate a *value change dump* (VCD) file that reports node activity. The VCD file is used to generate power and simulation time numbers on the post-APR (automatic place and routed) design.

The design is evaluated under the following four scenarios. This demonstrates the importance of supplying paths from Φ_c that cannot be cut as well as paths from Θ_m that must be cut to create a DAG.

- **No constraints:** A commercial CAD tool cuts all cycles in the design.
- **Local constraints:** The local cycles are cut using the timing constraint path driven algorithms in this dissertation, but the commercial CAD tool creates architectural cycle cuts. (No must-cut paths are given to the algorithm.)
- **Architectural constraints:** Only the must-cut constraint path is provided to the algorithm in this dissertation. The commercial CAD tool creates the local cycle cuts.
- **Full constraints:** The algorithm is given all four constraints and performs all cycle cutting.

The max-delay targets for the Φ_c constraints differed in each scenario due to differences in the cycle cuts. The smallest max-delay target without negative slack is 130ps for no constraints, 125ps for architectural constraints, 105ps for local constraints, and 100ps for the full constraint set.

Table 5.1 shows that there is a substantial circuit quality improvement obtained by employing the timing path constrained algorithms in this dissertation to cut cycles when compared to a commercial CAD tool. Improvements for this circuit include 1.3× for cycle time, 2.5× for area, and 2.7× for energy per token when compared to using the algorithms in a commercial CAD tool for cycle cutting. The table also points out the importance of including both performance critical and must-cut constraint paths. If only the architectural must-cut constraints are included, the results are generally worse than having the commercial CAD tool perform all cycle cutting. Simply employing the performance critical constraint paths helps, as this reduces energy by 1.5× over a commercial CAD tool. However, there still remains a penalty of 1.8× in energy over our algorithm if one allows the commercial CAD tool to perform architectural cycle cutting.

5.3.4 Generality of Approach

The algorithm and flow applies to any asynchronous module. This is illustrated by applying this to a well known *quasi delay insensitive* (QDI) controller. Fig. 5.3 shows the implementation of a *weak-condition half-buffer* (WCHB) [66]. This design has been mapped

Table 5.1: Comparison of performance metrics using timing path cycle cutting versus the algorithm in a commercial CAD tool.

	No Constraints	Architectural Constraints	Local Constraints	All Constraints
Forward Latency(ps)	97.5	127.5	85.0	107.5
Backward Latency(ps)	327.5	347.5	305.0	232.5
Cycle Time(ps)	520	540	460	390
Area(um^2)	361.788	384.000	236.592	145.740
Power(mW)	2.296	2.307	1.723	1.013
SimTime(ns)	32.541	34.287	29.342	27.798
Energy/token(pJ)	0.374	0.395	0.253	0.141

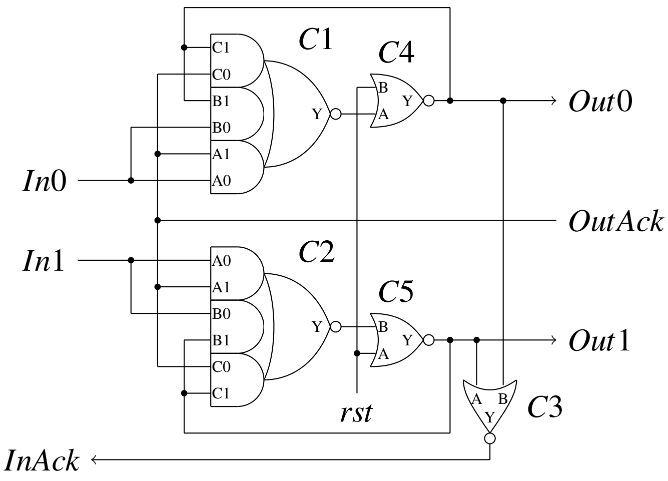

Figure 5.3: WCHB circuit.

to the same cell library and characterization flow described in the previous section. The constraint paths Φ_c for this circuit are $\{In0 \rightarrow Out0, In1 \rightarrow Out1, In0 \rightarrow InAck, In1 \rightarrow InAck\}$ while the must-cut paths are $\{OutAck \nrightarrow Out0, OutAck \nrightarrow Out1\} = \Theta_m$. The cycle cuts generated by the algorithm are shown graphically in Fig. 5.4 and the explicit constraints used are shown in Fig. 5.5

Postlayout results for a 4-deep pipeline are generated. Table 5.2 shows that using both local and architectural constraints clearly results in the best design. This timing optimized

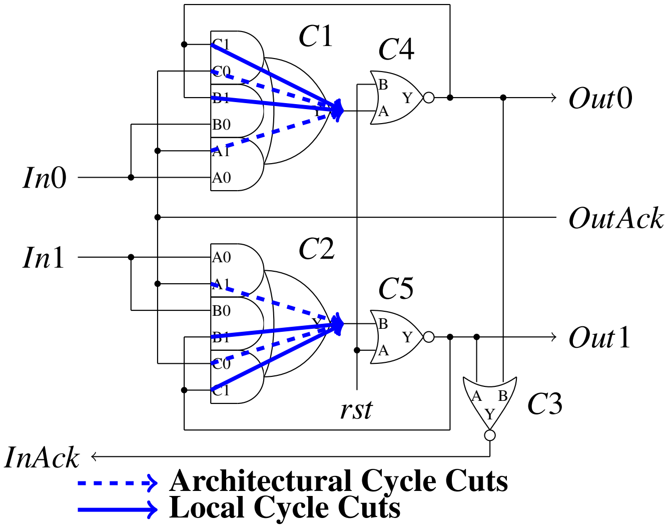

Figure 5.4: WCHB circuit with cycle cuts.

```
set_disable_timing -from B1 -to Y [find -hier cell *C1]
set_disable_timing -from C1 -to Y [find -hier cell *C1]
set_disable_timing -from B1 -to Y [find -hier cell *C2]
set_disable_timing -from C1 -to Y [find -hier cell *C2]
##Cuts for must-cut paths
set_disable_timing -from A1 -to Y [find -hier cell *C1]
set_disable_timing -from C0 -to Y [find -hier cell *C1]
set_disable_timing -from A1 -to Y [find -hier cell *C2]
set_disable_timing -from C0 -to Y [find -hier cell *C2]
```

Figure 5.5: Cycle cut constraints for WCHB circuit.

implementation has nearly a 2× improvement in forward and backward latency and cycle time, a 1.4× area advantage, and a 3.2× energy per token advantage.

5.4 Rules for Timing Path Driven Cycle Cutting

A set of rules are defined for creating DAGs from cyclic circuits based on timing paths. These rules are implemented in the algorithm described in the next section. The rules are illustrated based on the sequential circuit shown in Fig. 5.1. The same constraint paths and must-cut paths used in Sec. 5.3.3 are employed here.

Table 5.2: Comparison of performance metrics using timing path cycle cutting versus the algorithm in a commercial CAD tool for WCHB example.

	No Constraints	Architectural Constraints	Local Constraints	All Constraints
Forward Latency(ps)	162.5	160.0	152.5	82.5
Backward Latency(ps)	272.5	270.0	252.5	145.0
Cycle Time(ps)	510	520	460	270
Area(um^2)	1269.5	1214.3	1232.7	890.6
Power(mW)	0.717	0.607	0.646	0.344
SimTime(ns)	53.847	54.344	52.547	34.488
Energy/token(pJ)	0.770	0.659	0.678	0.237

5.4.1 Gate Sizing

To ensure performance and robustness of a design, each gate should have a delay target as part of a constraint path. See rule 1.

- **Rule 1** There must be at least one constraint path passing through a gate for it to be properly sized.

Delay targets in this flow are expressed as constraint paths derived from Φ. Assume gate G has no timing paths passing through it. The presence of no timing path through G occurs if no constraint paths are specified that pass through G, or all paths through G are cut to remove cycles and produce a DAG. Hence, there are no delay targets for sizing the gate.

A corollary to this rule is that no gate can have all input to output timing arcs cut. If that is the case, all timing arcs passing through the gate are cut and and the gate would not be properly sized.

5.4.2 Architectural Cycles are Cut

Must-cut paths are defined as the paths, which when cut, lead to the removal of the architectural cycles. See rule 2.

- **Rule 2** No path may exist from source to destination for any must-cut path in Θ_m.

Suppose there are n paths from source A to destination B of a must-cut path $A \not\twoheadrightarrow B$. If $n-1$ paths are cut, then there is only one remaining path from A to B of a must-cut path.

Likewise, if there are m must-cut paths, and only $m-1$ are cut, this leaves one must-cut path uncut. In either case, there is a path that results in an architectural cycle.

As an example, let us consider the must cut set $\Theta_m = \{ra \nrightarrow rr\}$ for the circuit of Fig. 5.1. There are two paths for the $ra \nrightarrow rr$ must-cut path, path 1: $[ra\ lc0\ lc3\ lc4]$ and path 2: $[ra\ lc0\ lc1\ lc2\ lc5\ lc3\ lc4]$. If only path 2 gets cut then there are still architectural cycles to the right of each handshake controller through path 1.

5.4.3 Timing Arc Fidelity

Each gate must lie on a timing path of a graph represented as a DAG to be properly sized according to rule 1. See rule 3.

- **Rule 3** At least one path from source to destination of every constraint path must remain uncut.

Consider a timing path $A \rightarrow B$ which has been cut into two segments at gate G. This results in uncut segments $A \rightarrow G$ and $G \rightarrow B$. Since timing constraints are expressed with respect to points A and B, and $A \rightarrow B$ is cut at G, no such path exists on the DAG and hence gate G cannot be optimized. If all such paths are cut, no timing paths exist from A to B.

5.4.4 Specifying the Correct Causal Path

The first three rules form the framework to create a DAG at the architectural level where every gate is sized. However, these are not sufficient to properly characterize and optimize a sequential circuit at the module level because the presence of cycles in a sequential circuit results in a plethora of timing paths. Some of these are causal paths that toggle gate outputs; others may in fact be noncausal paths that do not control signal transitions due to circuit logic. Therefore, rule 3 is in general too weak a constraint to ensure proper characterization of actual circuit delays with cycle cutting. *A timing path driven cycle cutting algorithm must cut noncausal paths but retain correct causal paths of each constraint path for the design to be properly timed and optimized.*

For example, constraint path $lr\uparrow \rightarrow ck\uparrow$ in Fig. 5.1 contains two paths: $[lc3\ lc4\ lc5\ lc1\ lc7]$ and $[lc1\ lc7]$. The shorter path is the causal path that must be selected for this constraint path.

A heuristic called the *greatest common path* (GCP) enables the tools to distinguish between the causal and noncausal paths in a sequential circuit. This heuristic makes use of the consideration that the shortest (least number of gates) path is usually the causal path in circuits which are designed to achieve the highest performance. The longer paths (having more gates in the path) are assumed to be noncausal paths. Noncausal paths are often part of the state holding logic that operates concurrently with the outputs in a timed asynchronous circuit.

Definition 3 A GCP is a minimal path from the input to output of a timing constraint. If the set of vertices of a shorter path is a strict subset of the same for a longer path where the order of the gates in the longer path is the same as the shorter path, the shorter path is a GCP.

The constraint path $lr\uparrow \rightarrow rr\uparrow$ in Fig. 5.1 contains two possible timing paths, path 1: $[lc3\ lc4]$ and path 2: $[lc1\ lc2\ lc5\ lc3\ lc4]$. Path 2 is not a GCP since it contains path 1. Path 2 is by default considered a noncausal path and removed from consideration by our algorithm. Verification and simulation of the circuit validates that path 2 is not causal because $lr\uparrow$, and not $y_{-}\downarrow$ causes $rr\uparrow$. Thus, the GCP is causal. See rule 4.

- **Rule 4** No timing arc of a GCP can be cut.

If a path exists from source to destination of a constraint path, the constraint path contains at least one GCP. If the algorithm does not cut arcs in a GCP, then at least one path is preserved for each constraint path.

Rule 4 now supersedes rule 3, since each constraint path contains at least one GCP. This rule also allows for multiple disjoint paths that exists on a timing path (Sec. 5.4.5). Thus, it guarantees that every path that can be used for timing is employed. It allows the removal of redundant paths in a design and the ability to define which paths are causal in a sequential circuit.

The GCP heuristic has several advantages in removing noncausal paths. GCP is a conservative constraint for min-delay paths that always result in sufficient clock margins for the latch/flop bank of the bundled data pipeline of Fig. 2.3. Assume the longer path for constraint path $lr\uparrow \rightarrow rr\uparrow$ that is not a GCP is the correct causal path. The longer path is cut, forcing the EDA tools to add delay to the shorter GCP. This results in larger delays added to the design than are required to meet the min-delay margin for the control path. The larger delay increases latency to the clock signal, and increase the cycle time, by providing extra margin and robustness to the design.

This heuristic also works well for high performance timed asynchronous circuits. For example, in burst-mode designs the state variables change concurrently with the outputs, increasing performance [50]. The state variable feedback signals create local cycles. Since feedback holds state, changes in these signals generally do not cause a change in the outputs of a high performance sequential circuit. Therefore, cycles can usually be cut in the state holding feedback logic. Thus, GCP facilitates cycle cutting of high performance designs by keeping the performance critical paths Φ_c uncut, focusing the arc cutting on the noncausal paths of the state feedback signals. This is the case in the burst-mode circuit of Fig. 5.1, where the state variable y_- switches concurrently with the outputs, making the GCPs the correct causal timing path constraints.

5.4.5 Defining Causal Paths that are Not GCPs

In some asynchronous circuits, the GCP may not be the causal path. This is more common with untimed asynchronous designs such as *speed-independent* (SI) implementations. Such designs reduce the number of relative timing constraints by sequencing state changes in such a way that output changes are delayed until the state variables have settled. This results in slow controller response time. For such designs, the causal timing paths must be defined such that they pass through the feedback logic of the gates. This ensures that these paths are not cut by rule 3.

The longer non-GCP path can remain uncut by breaking the original timing constraint into a composition of multiple GCP timing constraints. In Fig. 5.6 the timing paths for constraint path $lr\uparrow \rightarrow la\uparrow$ are path 1: $[lc1\ lc2]$, path 2: $[lc4\ lc5\ lc8\ lc9\ lc1\ lc2]$, and path 3: $[lc4\ lc5\ lc6\ lc7\ lc8\ lc9\ lc1\ lc2]$. Path 1 is the GCP for the three paths, but is not the causal path for $la\uparrow$. Note that la starts at logic zero, so $lr\uparrow$ cannot assert $la\uparrow$ due to the NAND gate in $lc1$ that is disabled by la. Thus, $la\uparrow$ gets set via the non-GCP path 2. The causal path first asserts $rr\uparrow$, then the state variable $csc2\uparrow$, which then asserts output $la\uparrow$. The algorithm can use rule 4 to ensure the path, considered as noncausal, from being cut by providing three constraint paths: $lr\uparrow \rightarrow rr\uparrow$, $rr\uparrow \rightarrow csc2\uparrow$, and $csc2\uparrow \rightarrow la\uparrow$. These three

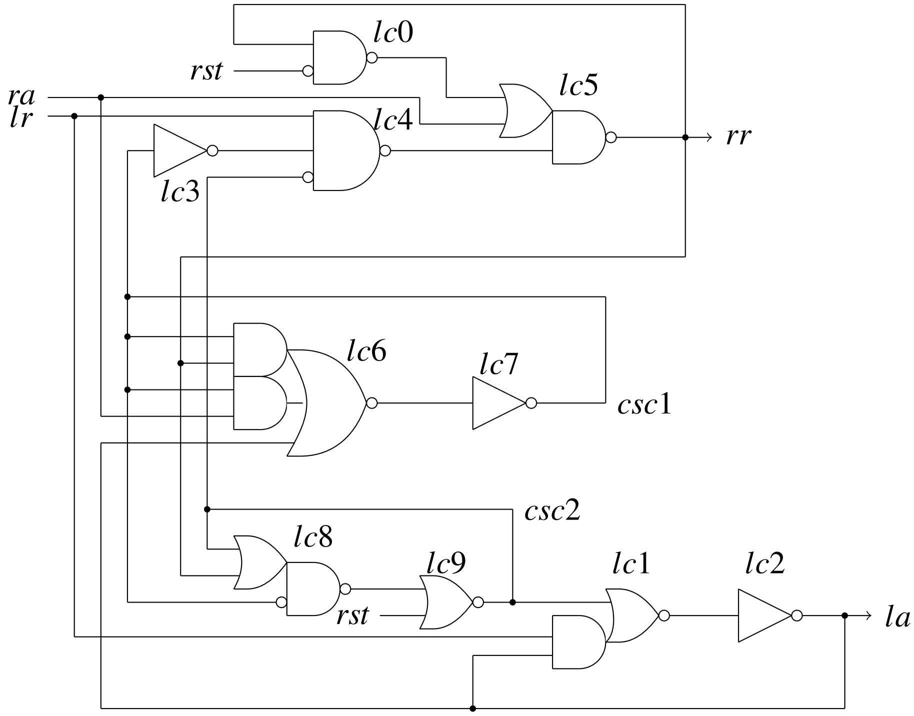

Figure 5.6: L0000_R0044 circuit implementation.

constraint paths replace the $lr\uparrow \rightarrow la\uparrow$ constraint path that they cover. This set of constraint paths define the shortest and longest paths as noncausal paths.

5.5 Algorithm

An algorithmic process is developed which automates the process of generating cycle cuts for sequential circuits. It obeys the four rules described in Sec. 5.4. This is a simple vectorless algorithm that deals with the circuit structure, without considering the exact logic function or delays of the gates.

The algorithm takes as input a structural Verilog description for a cyclic circuit implementation, a set of timing paths Φ_t and a set of must-cut paths Θ_m. It finds all the cycles present in the circuit module, and based on the timing paths specified, outputs a set of cycle cut constraints in the SDC format. These constraints can then be passed through the synthesis and place and route flows to allow the circuits to be automatically power and performance optimized. The algorithm is divided into four parts, explained below.

5.5.1 Adjacency List Creation

The circuit description is parsed and stored as an adjacency list $G = (V, E)$, where V is the list of vertices and E the edges of the circuit. Each vertex V_i is a tuple $\{C, N, I, O, P\}$, where C is the instance name, a vertex number N assigned to this vertex, a list of inputs I and outputs O and a list of pointers P to an entry E_i in the edges structure. The edge structure defines connectivity between gates in the design. This is achieved by defining a specific input pin on a gate instance and pointers to link multiple pins together due to fanout. Each edge E_i connects the output of vertex V_i with an input of a successor vertex V_{i+1}. Edge E_i is a tuple $\{C, N, M, P\}$, where C is the successor vertex (V_{i+1}) instance name, N is the vertex number of V_{i+1} in V, M is the input number on V_{i+1}, and P points to connected E_i structures on the output O of vertex V_i. There are also data structures *timing_path* (Φ_t), *constraint_path*, *mustcut_path* (Θ_m), and *cycles* that store all the $A \rightarrow B$ timing paths specified by the user, the GCPs, the must-cut paths and cycles present in the circuit, respectively.

A vertex V_i is created for each primary input and gate present in a structural Verilog module. There is an edge structure E_i created for each input pin on every vertex instance. Fig. 5.7 shows the adjacency list for the LC circuit example of Fig. 5.1.

5.5.2 Finding All the Cycles Present in the Circuit

A brute force algorithm with worst case complexity of $O(|V| * (|V| + |E|))$ is implemented to find all the local cycles in a circuit. These structural cycles are independent of the timing constraints.

A *depth first search* (DFS) is performed for each vertex V_i in the adjacency list G to find paths that return to the vertex V_i. If such a path exists then the stack which stores the trace is recorded as a cycle in the *cycles* data structure. The LC controller shown in Fig. 5.1 has eight cycles present as shown below.

Cycle 1	$[lc1\ lc2\ lc1]$
Cycle 2	$[lc1\ lc2\ lc5\ lc1]$
Cycle 3	$[lc1\ lc2\ lc5\ lc1]$
Cycle 4	$[lc3\ lc4\ lc3]$
Cycle 5	$[lc3\ lc4\ lc5\ lc3]$
Cycle 6	$[lc3\ lc4\ lc5\ lc3]$
Cycle 7	$[lc5\ lc6\ lc5]$
Cycle 8	$[lc5\ lc6\ lc5]$

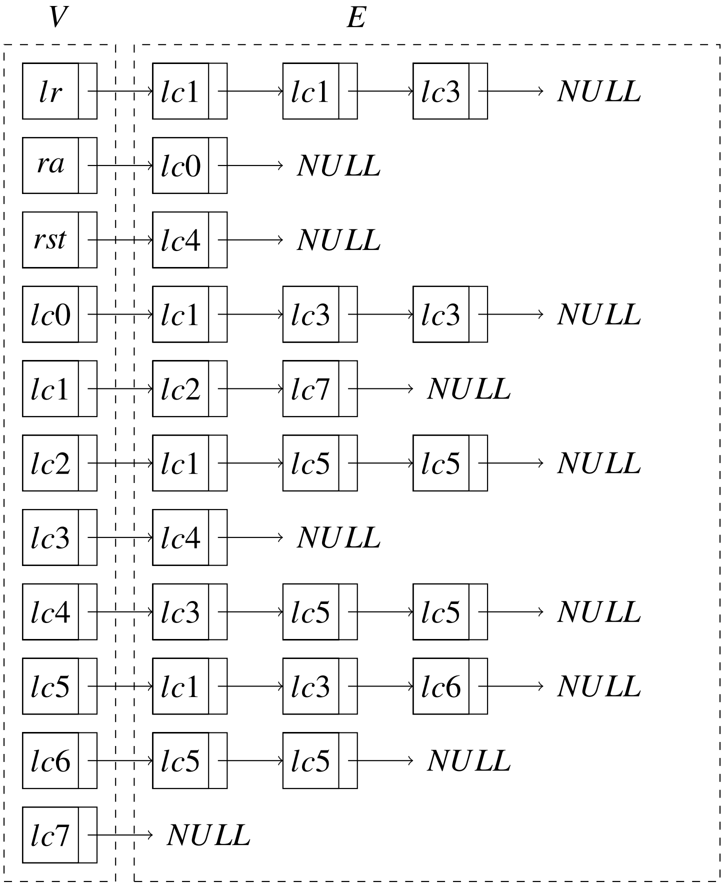
V
E
lr → lc1 → lc1 → lc3 → NULL
ra → lc0 → NULL
rst → lc4 → NULL
lc0 → lc1 → lc3 → lc3 → NULL
lc1 → lc2 → lc7 → NULL
lc2 → lc1 → lc5 → lc5 → NULL
lc3 → lc4 → NULL
lc4 → lc3 → lc5 → lc5 → NULL
lc5 → lc1 → lc3 → lc6 → NULL
lc6 → lc5 → lc5 → NULL
lc7 → NULL

Figure 5.7: Adjacency list for LC circuit of Fig 5.1.

Note cycle 2 and cycle 3 have the same set of gate names because the output of gate $lc2$ goes into two separate inputs of gate $lc5$. This can be verified from the adjacency list with gate $lc5$ appearing twice as successor of gate $lc2$. Similarly, cycle 5 and cycle 6 and also cycle 7 and cycle 8 have the same set of gate names but are two separate cycles. They might require two set_disable_timing constraints to cut the cycles.

Any circuit can be represented as a directed multigraph if the output of a gate goes into multiple inputs of any other gate. The brute force approach to find all the cycles in a multigraph results in an exponential algorithm. But the number of multiple edges going from a source node to the same destination in a sequential circuit node are limited and rare. This primarily occurs when using a gate library with very few complex gate functions, such as the academic Artisan library employed in this work. In a sufficiently rich library, such connectivity would not exist. Even so, working on a multigraph representation of these circuits at the module level does not lead to many additional cycles and thus the exponential runtime is avoided.[4]

The creation of a multigraph can be avoided in complex gates by converting them into two or more gates with one edge going into each gate. This adds two or more vertices to the graph and converts it into a simple directed graph. An example for this conversion is to replace $lc1$ in Fig. 5.1 into an AND3 gate and an AOI21 gate, with lr input going to both these gates. Another approach can be to replace the multiple edges from source vertex to destination vertex with one edge. Hence, while cutting the cycles, all the edges from source to destination vertex need to be cut. Thus, both these approaches can bound the number of edges by $|V|^2$ and the number of cycles is polynomial in terms of the number of vertices. But this directed multigraph can be converted into a directed graph by keeping only one directed edge between any source and destination vertex. A scan through the adjacency list is performed to remove all the duplicate edges. This results in the number of edges in the circuit getting bound by $|V|^2$ and the complexity defined above.

5.5.3 Timing Constraint Paths with Noncausal Path Removal

A DFS is performed to generate all paths between the nodes A and B, which are given as timing paths specified by the user. The complete list is pruned to the GCPs for noncausal path removal.

The following paths are returned as possible for timing constraint path $lr\uparrow \rightarrow la\uparrow$ for the LC controller shown in Fig. 5.1. These are pruned to the GCP, removing the second path.

$$\begin{array}{l} [lc1\ lc2] \\ [lc3\ lc4\ lc5\ lc1\ lc2] \end{array}$$

The complete set of GCPs for the following two timing constraint paths are shown here:

$$\begin{array}{ll} lr\uparrow \rightarrow la\uparrow & [lc1\ lc2] \\ lr\uparrow \rightarrow rr\uparrow & [lc3\ lc4] \end{array}$$

GCPs in the timing paths remain connected throughout the cycle cutting algorithm. Each path from the list of GCPs is traversed and the inputs of the gates present on these paths are marked as constrained to prevent them from being cut.

[4] Analysis of the set of 131 controllers in our test bench reveals that complex gates with more than 3 gate inputs have at most 2 edges coming into them from the same predecessor.

5.5.4 Generating Cycle Cuts

Two algorithms have been implemented to generate cycle cuts:

- **V1:** This is a polynomial time greedy approach. The solution is created by cutting maximum occurring edges.
- **V2:** This is an exponential time approach that searches through the complete list of solutions possible to find the highest quality solution.

Quality metrics for the tool flow report the status whether all local and architectural cycles are cut, and if there are any gates without a timing path passing through them. There is no need to create a minimal set of cuts, what matters is that the "right" set is created.

The base of both the approaches is the same. Up to this point all cycles and constraint paths have been defined. The problem of generating the cycle cuts is converted into a covering problem for which a covering table is generated with the cycles as the rows and the edges as columns. Only the edges present in the cycles which can be cut are considered. All the edges which are present on a GCP are excluded since they cannot be cut (rule 4).

Edges that have the same source and destination gates are combined into a single column even though it might generate multiple set_disable_timing constraints. The subscript used with edges are to distinguish between an edge forking into different paths. la_0, la_1, $y_{\_0}$, $y_{\_1}$, $y_{\_2}$, rr_0 and rr_1 are the edges $lc2-lc1$, $lc2-lc5$, $lc5-lc1$, $lc5-lc3$, $lc5-lc6$, $lc4-lc3$ and $lc4-lc5$, respectively.

Fig. 5.8 shows the covering table for the circuit shown in Fig. 5.1. After the generation of the table, the V1 algorithm selects the edge that cuts the maximum number of cycles. Each selected edge is removed from future consideration by removing that column. One or more set_disable_timing cycle cut constraints are written out, and all rows representing cycles which get cut by this edge are removed. The algorithm iterates through the table to find a solution by repeatedly selecting the next maximum occurring edge and updating the table. The generation of the local cycle cuts end when there are either no more edges (some cycles are not cut), or there are no more rows (all cycles have been cut) in the table.

There are six edges which can result in cutting two cycles for the covering table of the circuit of Fig. 5.1. The first is selected, i.e., la_1. Cycle 2 and cycle 3 are cut by removing this edge. This leads to the cycle count for the $y_{\_0}$ edge to become 0, and hence that column

	la_0	la_1	rr_0	rr_1	y_{-0}	y_{-1}	y_{-2}	y
$[lc1\ lc2\ lc1]$	√							
$[lc1\ lc2\ lc5\ lc1]$		√			√			
$[lc1\ lc2\ lc5\ lc1]$		√			√			
$[lc3\ lc4\ lc3]$			√					
$[lc3\ lc4\ lc5\ lc3]$				√		√		
$[lc3\ lc4\ lc5\ lc3]$				√		√		
$[lc5\ lc6\ lc5]$							√	√
$[lc5\ lc6\ lc5]$							√	√

Figure 5.8: Covering table for LC circuit.

is also removed. Continuing this process leads to the cycle cut set shown in Fig. 5.9. It removes all the cycles with seven cuts, graphically shown in Fig. 5.10.

V2 algorithm again creates the covering table, but goes one step further. It generates the complete list of solutions. The solution cost is employed to select the best solution. After generating each new solution, its cost is calculated and compared against the previous best solution. The solution with the minimum cost is selected as the best solution. The cost heuristics considers the number of uncut cycles and orphaned gates[5] and is calculated as: $\mathit{Cost} = 3 \times \mathit{number\ of\ uncut\ cycles} + \mathit{number\ of\ orphaned\ gates}$.

This exhaustive search has $O(2^n)$ complexity, where n is the number of edges in the covering table. Thus, the worst case complexity for V2 is $2^{|V|^2}$ because n has a complexity of $|V|^2$, which makes it very slow as compared to the $|V|^3$ complexity for the V1 algorithm, but the best solution can be found. The search ends when the first solution with zero cost is found, or when all the solutions have been generated. In the latter case the solution with the lowest cost is returned.

After generating constraints for all the local cycles, the global architectural cycles are removed. These cuts are generated from must-cut constraints. A DFS is performed to find all the must-cut paths in the circuit. This results in the following paths for Fig. 5.1 when the $ra\uparrow \nrightarrow rr\uparrow$ path is marked as must-cut.

Path 1	$[ra\ lc0\ lc3\ lc4]$
Path 2	$[ra\ lc0\ lc3\ lc4]$

[5] Gates having all timing paths passing through them getting cut.

```
set_disable_timing -from A1 -to Y [find -hier cell *lc5]
set_disable_timing -from B0 -to Y [find -hier cell *lc5]
set_disable_timing -from A0 -to Y [find -hier cell *lc5]
set_disable_timing -from C0 -to Y [find -hier cell *lc5]
set_disable_timing -from A -to Y [find -hier cell *lc6]
set_disable_timing -from B1 -to Y [find -hier cell *lc1]
set_disable_timing -from B1 -to Y [find -hier cell *lc3]
```

Figure 5.9: Local cycle cut constraints for LC circuit of Fig 5.1.

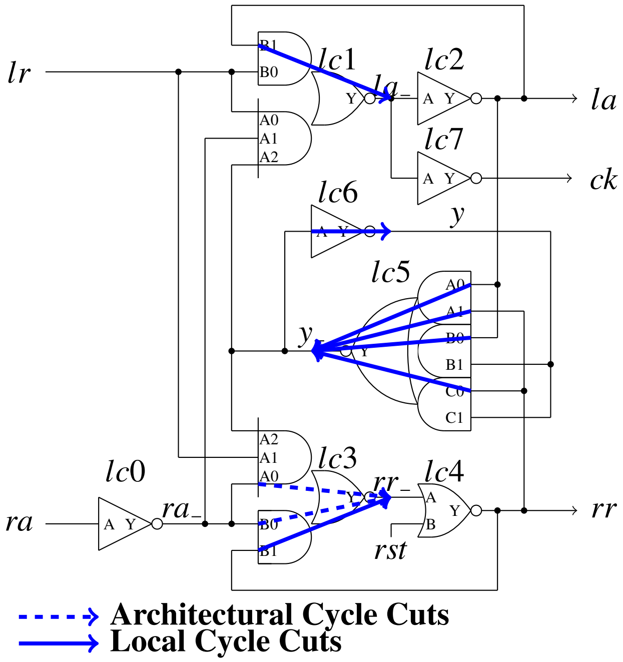

Figure 5.10: LC circuit with cycle cuts.

A covering table is constructed where the rows define must-cut paths and columns define the edges on those paths. Similar to local cycle cutting, edges that are on a timing constraint path cannot be added to the table. The V1 algorithm, that eagerly selects cuts based on the number of paths cut, is employed. A set_disable_timing cycle cut constraint is written out, and all rows representing must-cut paths by this edge are removed.

The algorithm ends when there are either no more edges (some must-cut paths could not be cut), or there are no more rows (all must-cut paths have been cut) in the table.

Following is the covering table for the must-cut constraint $ra \nrightarrow rr$ applied to Fig. 5.1.

	ra_-	ra
[*ra lc*0 *lc*3 *lc*4]	√	√
[*ra lc*0 *lc*3 *lc*4]	√	√

Both paths can be cut by selecting the ra_- edge. The architectural cycle cuts for the LC circuit of Fig. 5.1 are graphically shown by dashed line in Fig. 5.10 and the constraints are shown in Fig. 5.11.

5.5.5 Uncut Cycles

It is possible to leave uncut cycles in the timing graph due to rule 4. When cycles are uncut, it is not possible to optimize the circuit in a single pass with the given set of timing constraints. The user must either define a different set of constraints that can do timing optimization with a single pass, or run multipass optimization on the design. A new set of timing constraints may be possible by analyzing the GCPs and modifying the timing paths provided as inputs so as to relax the constraints a bit and thus reduce the number of GCPs for the circuit. This results in an increase in the number of cycle cut candidates and thus all the uncut cycles can be cut accordingly. This may also lead to more gates being unsized due to the reduction in timing paths.

5.6 Results

The results for applying the V1 and V2 algorithms to a large set of handshake controllers and a few benchmark circuits are reported. The design and evaluation flow described in Chapter 3 are employed to generate the following results.

5.6.1 Four-cycle Handshake Controllers

This example set consists of the complete family of 131 untimed 4-cycle handshake controllers with data valid at the rising edge of request (*lr*). They are generated from concurrency reduction rules [35, 67]. This creates a rich set of protocols with various properties, such as half and full data buffered pipelines. The concurrency reduction rules are applied to the most concurrent protocol (top left corner) to generate the complete set of *untimed* (speed independent and delay insensitive) protocols. Results are shown in a tabular format from the most concurrent controller in the top left corner of the tables to the least concurrent in the bottom right. Here, the number LXXXX represents increased

```
set_disable_timing -from A0 -to Y [find -hier cell *lc3]
set_disable_timing -from B0 -to Y [find -hier cell *lc3]
```

Figure 5.11: Architectural cycle cut constraints for LC circuit of Fig. 5.1.

concurrency reduction on the downstream (*rr*, *ra*) channel. Likewise the RXXXX values represent orthogonal concurrency reduction on the upstream channel (*lr*, *la*). Some of the handshake controllers did not have an implementation and are marked as '–' in the tables. Those that deadlock due to too much concurrency reduction are marked with '.'.

Some concurrency reduction rules create protocols that implement speed independent protocols employing *output ordering*. This forces one output (or state variable) to switch before another. For these cases, GCPs from a primary input to the primary output, as employed for the example set, but are not the causal path. Hence, each design is manually analyzed to find the correct causal path. These paths are specified with several related constraint paths (Sec. 5.4.5).

The algorithm described in this dissertation is written in C++. The results are run on a Core i7 processor with 4GB memory. Most sequential control circuits are relatively small and so this problem is not constrained by run time or memory. The run time for the whole set of handshake controller circuits is 2.43s and 6.60s for the V1 and V2 algorithm, respectively.

Table 5.3 shows the total number of cycles in these controllers. The amount of concurrency in a design is directly proportional to the number of state variables required [67]. As expected, the most concurrent protocols contain the largest number of cycles due to more state holding feedback signals. The complexity of the circuits and the number of cycles in the circuits decrease with concurrency reduction.

Two generic constraint paths, $lr{\rightarrow}rr$ and $lr{\rightarrow}la$, have been applied to each circuit module. Both are performance critical constraints in Φ_c for all designs. The must-cut constraint path $ra{\nrightarrow}rr$ is employed. The quality metrics for the algorithm can be directly derived from rules 1, 2, and 4. The number of uncut cycles, number of must-cut paths, and gates that do not have a timing path passing through them are reported for each circuit.

Table 5.4 shows the number of cycles left uncut. This shows that 127 of the 131 test cases are able to remove all cycles employing the given constraint paths. The four circuits with cycles left uncut have one gate repeated twice in the causal path. This creates a causal

Table 5.3: Total number of cycles found.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	18	35	–	–	11	24	7	5
R0020	38	–	17	14	11	14	9	13	9	8
R0040	14	23	15	14	44	19	8	5	8	10
R0022	25	50	7	10	19	8	7	3	4	4
R0042	39	13	14	–	16	35	7	10	6	5
R2022	22	30	44	7	12	12	8	6	4	.
R2042	50	20	10	5	7	8	6	4	4	.
R0044	10	7	10	4	5	10	4	6	3	3
R2044	7	9	7	4	6	6	4	2	3	.
R4044	18	7	.	3	.	5	.	.	.	.
R2222	19	7	5	3	5	5	4	4	2	.
R2242	17	9	8	7	6	5	4	3	3	.
R2262	7	7	10	4	5	.	.	3	.	.
R2244	3	4	4	1	1	1	1	1	1	.
R2264	5	6	5	1	2	.	.	1	.	.
R4244	5	6	.	1	.	2	.	.	.	.
R4264	4	4	.	1	.	.	.	.	.	.

cycle with GCPs which cannot be cut. For example, the causal path from $lr\uparrow$ to $rr\uparrow$ in the $L1111\_R0022$ protocol is $[lr\uparrow\ U3\downarrow\ la\uparrow\ U6\downarrow\ U7\uparrow\ U2\downarrow\ U3\uparrow\ U1\downarrow\ rr\uparrow]$. The $U3\downarrow$ transition first sets the controlling value to one of the inputs of gate $U1$ to prevent it from switching $rr\uparrow$, since the protocol requires the $la\uparrow$ output to switch before rr. After this it sets the internal state variable and then switches rr via $U3\uparrow$. The must-cut path is correctly cut for all the 131 test cases.

Table 5.5 shows that a number of gates are left unsized. Gates are left unsized for two reasons: first, there is no timing path through the gate, and second, all input to output timing arcs get cut. Table 5.5 does not identify the reason a gate is unsized. Further investigation reveals that the V1 algorithm cuts all input to output timing arcs on 59 gates. This number is

Table 5.4: Cycles left uncut for V1 algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	2	0	–	–	0	0	0	0
R0020	0	–	2	0	0	0	0	0	0	0
R0040	0	0	0	0	0	0	0	0	0	0
R0022	0	0	2	0	0	0	0	0	0	0
R0042	0	0	0	–	0	0	0	0	0	0
R2022	0	0	0	0	0	0	0	0	0	.
R2042	0	0	0	0	0	0	0	0	0	.
R0044	0	0	0	0	0	0	0	0	0	0
R2044	0	0	0	0	0	0	0	0	0	.
R4044	0	0	.	0	.	0	.	.	.	.
R2222	0	0	0	0	0	0	0	0	0	.
R2242	0	2	0	0	0	0	0	0	0	.
R2262	0	0	0	0	0	.	.	0	.	.
R2244	0	0	0	0	0	0	0	0	0	.
R2264	0	0	0	0	0	.	.	0	.	.
R4244	0	0	.	0	.	0	.	.	.	.
R4264	0	0	.	0	.	.	.	.	.	.

improved in the V2 algorithm resulting in only five gates having all input to output paths cut. The remainder of the unsized gates have no timing constraint path passing through them. These gates are primarily associated with the local state variable logic. Thus, these gates can be sized by applying constraint paths that are specific to the state logic of each design.

Table 5.6 shows the energy delay product comparing cycle cutting being performed by the V1 algorithm and a commercial CAD tool. The benefit of the V1 algorithm ranges from an improvement of 11.87× to 1.66× over a commercial CAD tool. Table 5.7 summarizes the performance results. It presents a comparison of the performance values employing cycle cutting done by a commercial CAD tool and by V1 algorithm for latency, cycle time, area, power, and energy. The average aggregate improvement of forward latency × area ×

Table 5.5: Unsized gates for V1 algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	2	6	–	–	2	12	1	4
R0020	6	–	3	5	2	3	4	3	2	2
R0040	4	4	6	5	7	7	3	3	3	6
R0022	8	7	2	4	3	4	4	3	2	3
R0042	11	5	7	–	7	6	4	5	4	4
R2022	9	6	6	6	6	2	3	3	2	.
R2042	10	3	4	3	3	4	4	3	3	.
R0044	5	2	4	2	2	5	2	5	4	3
R2044	3	4	5	4	5	2	2	0	5	.
R4044	8	3	.	1	.	2	.	.	.	.
R2222	6	2	1	1	2	1	2	3	2	.
R2242	6	2	3	3	3	4	2	2	4	.
R2262	4	3	2	2	3	.	.	4	.	.
R2244	3	4	2	2	1	2	2	0	0	.
R2264	1	2	2	2	2	.	.	0	.	.
R4244	4	2	.	1	.	1	.	.	.	.
R4264	3	4	.	3	.	.	.	.	.	.

$e\tau$/token results in a 25.8× improvement over a commercial CAD tool across the protocol set.

Appendix B shows the detailed comparison for forward latency, backward latency and cycle time of each template with cycle cutting performed by a commercial CAD tool and our timing path driven cycle cutting. A comparison of these numbers show that the commercial CAD tool typically generates a slower circuit except for a few cases where it used big gates that improved performance but wastes a lot of energy. An example for this is the L3333_R0040 circuit. The commercial CAD tool generates cuts which enable a 20 percent faster implementation than that with cuts generated by V1 algorithm. But the area of the design is 2.32× larger and the energy consumed is 2.75× higher.

Table 5.6: $e\tau$ ratio of cycle cutting done by the commercial CAD tool and V1 algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	2.14	4.79	–	–	2.72	4.66	7.72	8.92
R0020	6.45	–	2.07	3.01	5.79	6.66	9.25	8.67	6.16	8.36
R0040	8.23	8.58	8.81	2.46	4.09	7.89	7.45	6.86	1.98	2.25
R0022	5.04	5.93	1.68	9.45	10.47	6.70	4.28	9.30	7.53	6.22
R0042	4.20	5.55	8.16	–	11.87	8.06	4.20	3.96	8.79	5.96
R2022	5.32	4.57	4.43	6.19	5.28	4.47	8.22	5.40	9.50	.
R2042	8.35	7.44	9.45	6.39	5.26	4.92	9.56	3.34	3.96	.
R0044	5.13	7.52	6.73	6.16	4.81	5.07	2.96	4.04	3.59	6.66
R2044	7.37	5.02	4.30	3.24	3.26	4.01	3.60	5.28	2.67	.
R4044	7.07	6.27	.	4.11	.	3.53	.	.	.	.
R2222	7.71	4.48	9.08	8.48	5.43	8.47	7.09	9.06	6.62	.
R2242	7.04	1.66	5.06	5.32	4.53	5.87	8.03	4.47	2.46	.
R2262	5.85	6.16	6.07	7.14	4.13	.	.	3.76	.	.
R2244	7.93	5.80	6.19	4.75	6.24	6.35	5.62	4.14	5.26	.
R2264	5.73	8.69	7.93	4.82	5.53	.	.	4.35	.	.
R4244	5.72	6.42	.	4.59	.	5.62	.	.	.	.
R4264	6.14	6.10	.	5.28	.	.	.	.	.	.

Appendix B show the routed core area for circuits with cycle cuts generated by my algorithm and by commercial CAD tool, respectively. For this set of examples, the commercial CAD tool always over-sizes the gates, if it performs cycle cutting. Four of the five cases with area 1.9× times or less than the circuits optimized with timing are for the circuits with uncut cycles. In these five cases, the commercial CAD tool performed cycle cuts after the timing driven algorithms are employed.

The same conclusion can be made by comparing the power consumption and simulation time numbers reported in Appendix B. There are five designs that have faster simulation time when the commercial CAD tool does cycle cutting, but the performance improvement

Table 5.7: Comparison of performance metrics using the algorithm in a commercial CAD tool versus timing path based cycle cutting (V1) (Commercial CAD tool number/V1 number).

	Minimum Value	Maximum Value	Average
Forward Latency	0.76×	3.56×	1.60×
Backward Latency	0.33×	2.92×	1.55×
Cycle Time	0.77×	2.32×	1.52×
Area	1.46×	4.49×	2.96×
Power	1.00×	6.19×	2.81×
SimTime	0.81×	2.16×	1.42×
Energy/token	1.43×	6.34×	3.84×
eτ/token	1.66×	11.87×	5.88×

comes at the cost of wasting a lot of power and thus the optimization results in poor energy values.

Energy per token comparison is also shown in Appendix B. The Energy numbers for the pipeline design are 3.84× larger on an average for the case when the commercial CAD tool does cycle cutting.

Results of the V1 and V2 algorithms are compared for forward latency, backward latency and cycle time. The average variation is found to be 0.3 percent. Designs using the V2 algorithm are generally faster except for 7 cases with more than ±10 percent variation. Similarly in terms of power consumption and simulation time, the results are pretty much the same except for 6 cases where the variation is more than ±10 percent and ±5 percent, respectively. Similarly, the average energy variation is around 0 percent, but four of the previous outlying designs show results with more than ±10 percent variation.

5.6.2 Benchmark Circuits

The analysis of the V1 and V2 algorithms is done on benchmark circuits of varying complexity. Benchmark designs like GCD and modules of postoffice and PSCSI are synthesized using Petrify to generate a gate level netlist to which reset is added by hand [56, 68]. The causal relationship resulting in the outputs switching based on the inputs is analyzed from the state graph. This inspection led to selecting the input to output paths as the timing constraint paths for the algorithm. Manual analysis of these paths led to selecting the actual causal paths taken when the inputs change as described in Sec. 5.4.5. The architectural connectivity of these designs is ignored hence no must-cut paths are specified alongwith the timing constraint paths. Using this information, the cycle cuts are generated for these circuits using the V1 and V2 algorithms.

Table 5.8 shows the design complexity of each of these design and a comparison of generating the cycle cuts using V1 and V2 algorithms. It compares the two algorithms based on the number of unsized gates and algorithm runtime. A list of the number of causal path segments present in the designs and also the number of GCPs which they result in are also shown to give a comparison between the complexity of the designs and the algorithm performance. Analysis of the pscsi-isend design gives a better picture of the exponential nature of the exahaustive V2 algorithm. The presence of lots of cut point candidates because of the lack of GCPs, which is evident with the unsized gates list, results in the exhaustive search of going through the whole search place for a zero cost solution which is not present, but it consumes a very long run time.

Table 5.9 gives a comparison for the designs generated by the algorithms with respect to a commercial CAD tool in terms of area, energy/token, simulation time and $e\tau$. These numbers give a comparison on the effectiveness of applying the greedy approach (V1) and the exhaustive approach (V2) to finding the cycle cut points in a circuit. The generation of these numbers is done as described for the examples in Sec. 5.3.3. The designs generated using the V1 algorithm cycle cuts at an average are 2/3 the size, with 5 percent performance improvement at half the energy, thus overall resulting in a 2.08× benefit in terms of $e\tau$. The benefits for the V2 algorithm are at par with the V1 algorithm with the only difference being that the designs are lower energy and are slightly smaller.

5.7 Summary

Timing arcs must be cut to represent the timing graphs of sequential circuits as DAGs in the current state-of-the-art CAD tools. An algorithmic approach is presented for automating the timing path driven generation of these cycle cuts so that the CAD tools can perform proper gate sizing for performance, area, and energy optimizations on sequential circuits

Table 5.8: Benchmark circuits design comparison (*Number of gates with all the input to output paths cut).

	Number of Cycles	Gate Count	Unsized Gates		Algorithm Runtime (s)			
			V1	V2	V1	V2	Paths	GCPs
gcd	22	72	34(0*)	34(0*)	<0.01	0.01	175	43
postoffice-rcv-setup	1	8	2(0*)	2(0*)	<0.01	<0.01	4	4
postoffice-sbuf-send-ctl	12	28	16(1*)	16(0*)	<0.01	0.01	120	14
pscsi-isend	325	43	19(2*)	19(0*)	0.17	49710	6122	64
pscsi-trcv-bm	6	26	11(1*)	11(0*)	<0.01	<0.01	32	21
pscsi-tsend-bm	10	33	7(2*)	7(0*)	0.02	0.03	377	114
pscsi-tsend	10	35	7(2*)	7(0*)	0.04	0.12	1819	108

without modifying the underlying netlist. Timing is specified as timing *constraint paths* to the algorithm. The timing *constraint paths* are of two forms: those that cannot be cut to preserve necessary timing paths, and those that must be cut to prevent architectural cycles. A method is provided for specifying the correct causal timing paths in the sequential circuits based on constraint path composition. The CAD tool reports on the quality metrics of the results, consisting of the number of cycles left uncut and the number of gates that do not have a timing path passing through them. Two versions of the algorithm are presented: a faster greedy search as well as an exhaustive algorithm that returns a result of the highest quality.

The CAD tool developed generates cycle cutting constraints in the SDC format. This timing path driven cycle cutting algorithm is a key component of a CAD flow that enables asynchronous design to be synthesized, placed and routed, power and performance optimized, and validated for postlayout timing correctness using commercial CAD tools that are intended for combinational logic using a clocked timing paradigm.

The algorithms are general to any sequential circuit. The algorithm is demonstrated on a test bench of 131 4-cycle bundled data asynchronous controllers, one delay insensitive design and a set of benchmark circuits. Circuits in this example set have as many as 325

Table 5.9: Results comparison for benchmark circuits.

	Commercial CAD tool			V1 Algorithm			V2 Algorithm			V1 Benefits			V2 Benefits		
Benchmark Circuit	Area (um^2)	Energy/ token (pJ)	Sim. Time (ns)	Area (um^2)	Energy/ token (pJ)	Sim. Time (ns)	Area (um^2)	Energy/ token (pJ)	Sim. Time (ns)	Area	Energy/ token	Sim. Time	Area	Energy/ token	Sim. Time
gcd	647.2	1.25	314.46	428.6	0.77	302.74	428.6	0.77	302.74	1.51	1.63	1.04	1.51	1.63	1.04
postoffice-rcv-setup	68.6	0.05	86.78	54.9	0.03	87.86	54.9	0.03	87.86	1.25	1.25	0.99	1.25	1.35	0.99
postoffice-sbuf-send-ctl	224.6	0.78	316.81	132.0	0.36	317.85	122.6	0.34	329.45	1.70	2.17	1.00	1.83	2.31	0.96
pscsi-isend	495.5	0.98	296.47	237.5	0.46	261.39	261.4	0.54	277.51	2.09	2.13	1.13	1.90	1.82	1.07
pscsi-trcv-bm	309.5	0.66	143.91	149.2	0.24	141.17	152.6	0.25	139.69	2.07	2.72	1.02	2.03	2.60	1.03
pscsi-tsend-bm	347.2	0.79	223.41	208.3	0.36	213.44	197.2	0.32	234.11	1.67	2.18	1.05	1.76	2.44	0.95
pscsi-tsend	309.5	0.61	219.42	210.0	0.37	198.36	193.7	0.30	213.77	1.47	1.68	1.11	1.60	2.06	1.03
Average Benefit										1.68	1.98	1.05	1.70	2.03	1.01

cycles in the implementation. The general run time for each example is very small with the worst case for a complex benchmark design like pscsi-isend being 0.17s for the greedy algorithm, and 49710s for the exhaustive algorithm. Results are reported with an identical set of timing paths on these examples. The number of cycles left uncut, the number of must-cut paths left uncut and the number of gates left orphaned are reported. It is shown that leaving any cycle uncut leads to issues in timing optimization as validation, as well as producing inferior circuits. The circuits generated when cycle cutting is performed by a commercial CAD tool are on average 2.96× larger, operate 1.42× slower, have a forward latency 1.60× greater and consume 3.84× more energy. The average aggregate benefit of this approach is a 25.8× improvement using these metrics.

CHAPTER 6

CHARACTERIZATION OF FAMILY OF 4-PHASE LATE PROTOCOLS

Power, performance, and area trade-offs are necessary considerations during any digital design. The selection of any good design is generally for a specific set of requirements of the system based on the cost-benefit analysis through optimum trade-off in these three parameters. To ease this trade-off selection, prior information about the characteristics for a slew of circuit choices is required. In case of bundled data systems, there can be huge differences in the final circuit based on the choice of the handshake controller design used. Hence, an understanding of protocol functionality and optimizations that can be performed is necessary to develop optimized systems. As an example, if performance requirements guides the protocol selection then for a high performance design, a low cycle time protocol must be selected. Similarly, for a very low performance design a slow, sequential protocol with less concurrency is preferred. Additional parameters like area and energy are also important and need to be considered along with performance. The process of selection of the best available circuit can be simplified if an easy method of characterizing the designs is present or tables for precharacterized designs are generated for the circuit implementations in the particular technology library.

This chapter explores the family of 4-phase handshake protocols with data valid at the falling edge of the request signal, also known as late data validity protocols. It considers the work in [35, 36, 67] which explains the theory for deriving various protocols based on concurrency reduction and also describes the characterization results for 4-phase protocols with data valid at the rising edge of the request, i.e., early data validity protocols. The theory and the approach for these early data validity protocols is taken as the base and the theory for the 4-phase late data validity protocols is described. Two sets of designs are generated based on the way the reset signal gets added to them and compared against each other. Comparison between these designs lead to detailed tables for forward latency, backward latency, cycle time, buffering depth, area, and energy. These tables assist in easing the trade-off based selection while using these designs to make a good bundled data system.

6.1 Key Contributions

- Concurrency reduction approaches for a family of untimed 4-phase late data validity protocols based on [67] and [35, 36].
- Characterization of all the protocols in this family of protocols for energy, performance and area.
- Design of a hand-optimized controller and its comparison with equivalent circuit generated by Petrify.

6.2 Background

A family of 4-phase handshake protocols with data valid at the rising edge of a request signal coming into the circuit block is presented in [67] and [35, 36]. The earlier publication looks at the theory for concurrency reduction and composition of these protocols in linear and parallel pipeline structure. The latter publications analyze the circuit implementations for these protocols. The systematic study presented for the early data validity protocols resulted in 131 handshake controller designs. Detailed analysis of the effects of concurrency reduction on circuit parameters like forward latency, backward latency, cycle time, buffering depth, area and energy is presented.

Analysis of the early data validity protocol family starts by specifying the most concurrent protocol, also known as the *max* protocol. The *calculus of communicating systems* (CCS) specification for this *max* protocol is shown below.

$$\begin{aligned}
L &= lr\uparrow.gS.\overline{rEn}\uparrow.aEn\uparrow.\overline{rEn}\downarrow.aEn\downarrow.pV.\overline{la}\uparrow.lr\downarrow.\overline{la}\downarrow.L \\
R &= gV.\overline{rr}\uparrow.ra\uparrow.pS.\overline{rr}\downarrow.ra\downarrow.R \\
S &= \overline{gS}.\overline{pS}.S \qquad\qquad\qquad\qquad V = \overline{pV}.\overline{gV}.V \\
LC &= (L \mid R \mid S \mid V)\setminus\{gS, pS, gV, pV\} \\
LATCH &= rEn\uparrow.\mathsf{open}.\overline{aEn}\uparrow.rEn\downarrow.\mathsf{closed}.\overline{aEn}\downarrow.LATCH \\
max &= (LC \mid LATCH)\setminus\{rEn, aEn\} \qquad (6.1)
\end{aligned}$$

Fig. 6.1 shows the minimized state graph representation of the *max* protocol which can be represented as a simple STG as shown in Fig. 6.2. These representations help in visualizing the protocols. A less cluttered shape, shown below, for *max* was proposed and it removes the arcs.

```
o o o ● o o o o o
        o o o o o
        o o o o o o o o o
        o o o o o o o o o
```

Concurrency reduction rules with left ($\mathscr{L}$) and right ($\mathscr{R}$) cuts were then derived and applied on the *max* protocol to derive less concurrent designs. This systematic approach leads to easily characterizing the family of early data validity protocols.

Late data validity protocols, as defined in Sec. 2.1.3.2, are the protocols where the data is valid from the falling edge of the request signal until the falling edge of the acknowledge signal. The family of late data validity protocols is derived using the same systematic approach as described above. Concurrency reduction techniques and rules similar to the early data validity protocols are then derived for these circuits.

The drawback of bundled data system is the area, performance and power/energy penalty of the delays that are added between pipeline stages based on the amount of logic in the datapath. In the case of 4-phase protocols, the delay between two pipeline stages is traversed through twice. For the early data validity protocols, this results in the operational frequency of the pipeline stage in being half of that of the amount of logic in the datapath. Various ways to hide this overhead in the protocols, like acknowledging early any request signal in parallel with enabling the clock signal at the successor pipeline stage a bit late, reduces the size of the delay. But, still this penalty is pretty big for datapath with large combinational circuits. Another approach is to have nonsymmetric delays have a faster reset phase of the protocol than the set phase.

These drawbacks get addressed by the late data validity protocols. This protocol theoretically allows the delay element to be half of what is required for the early data validity protocols, thus reducing its impact on the area of the design as well as the power/energy consumption. Also, the reset phase of the 4-phase protocol is shortened to falling edge of the acknowledge signal as compared to early data validity protocols for which the reset phase consist of the rising and falling edge of the acknowledge signal and the falling edge of

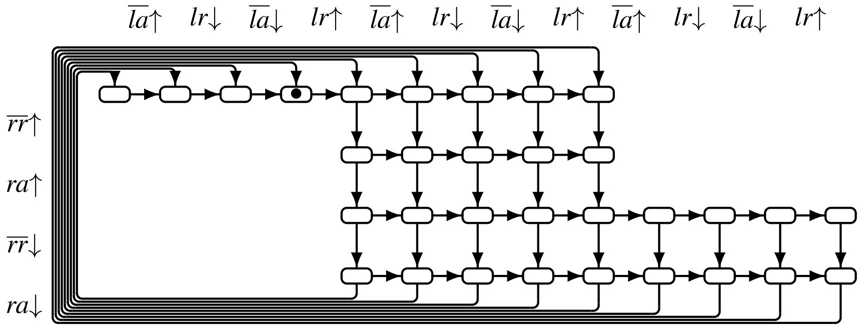

Figure 6.1: Minimized state graph of *max*, configured as a shape for early data validity protocols.

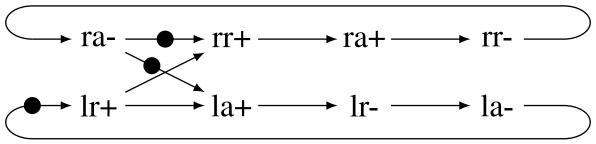

Figure 6.2: STG for the abstracted *max* protocol for early data validity protocols.

the request signal. Hence, late data validity protocols can lead to performance similar to the amount of logic present in a pipeline stage. But this also creates a drawback for these circuits being slow and not useful when the target design requires very small forward latency. In light of these motivations, exploring the family of untimed late data validity protocols to gain a better understanding of its impact on various design parameters is necessary. Results are presented for these circuits for area, energy and performance. A hand-optimized controller is also described to show the optimizations that can be achieved.

6.3 Late Data Validity Protocols

The CCS specification for the *max* protocol for late data validity protocol family is as shown below. Since the data word is valid between $lr\downarrow$ and $la\downarrow$, the synchronization points in the CCS specifications, i.e., gS and pV, and also the latch enable with its open and close operation are present between these transitions.

$$\begin{aligned}
L &= lr\uparrow.\overline{la}\uparrow.lr\downarrow.gS.\overline{rEn}\uparrow.aEn\uparrow.\overline{rEn}\downarrow.aEn\downarrow.pV.\overline{la}\downarrow.L \\
R &= gV.\overline{rr}\uparrow.ra\uparrow.\overline{rr}\downarrow.ra\downarrow.pS.R \\
S &= \overline{gS}.\overline{pS}.S \qquad\qquad\qquad\qquad V = \overline{pV}.\overline{gV}.V \\
LC &= (L \mid R \mid S \mid V) \setminus \{gS, pS, gV, pV\}
\end{aligned}$$

$$\begin{aligned} LATCH &= rEn\uparrow.\mathsf{open}.\overline{aEn}\uparrow.rEn\downarrow.\mathsf{closed}.\overline{aEn}\downarrow.LATCH \\ max &= (LC \mid LATCH)\setminus\{rEn,aEn\} \end{aligned} \tag{6.2}$$

A minimized state graph and an STG for this *max* protocol are shown in Fig. 6.3 and Fig. 6.4, respectively. For simpler representation, the minimized state-graph for the late data validity protocols can be represented in a less cluttered form as shown below.

```
o ● o o o o o o o
        o o o o o
        o o o o o
        o o o o o
```

6.3.1 Concurrency Reduction on MAX with Cuts

Concurrency reduction approach for any protocol starts by removing states from a concurrent protocol. The final protocol derived becomes more sequential and can be simpler, faster, and can consume less power and area. Since *max* is the most concurrent protocol, it is used as the base to start reducing the concurrency and derive other protocols in the late data validity protocol. Various less concurrent protocols can be generated by removing states from *max* in a systematic way. The removal of states is shown by replacing the ○ with a **.** if the state is unreachable (cutaway) in a particular protocol. The concurrency reduction is possible $\mathscr{L}$ on the left and $\mathscr{R}$ on the right side of the max protocol. Hence, using the convention of the early protocols, the concurrency reduction rules generate different protocols based on the *left cuts* $\mathscr{L}$ and *right cuts* $\mathscr{R}$ from the *max* shape. An overview of the cut representation is shown for the left and right cuts below.

6.3.1.1 Concurrency Reduction from Right Cuts $\mathscr{R}$

The states removed in a right cut are denoted as *Rabcd* as shown in Fig. 6.5. Rabcd denotes the removal from *max* of *a* states from the right end of row 1, *b* from row 2, *c* from row 3, and *d* states from the right end of row 4. The maximal cutaway per row is 4 for all the rows and is shown by the dashed box in Fig. 6.5. The result of cut R2200 is depicted in Fig. 6.6. The family of all $\mathscr{R}$ cuts is generated by the constraints:

$$\begin{gathered} 0 \leq a,b,c,d \leq 4 \\ a \geq b \ \wedge \ b \geq c \ \wedge \ c \geq d \end{gathered} \tag{6.3}$$

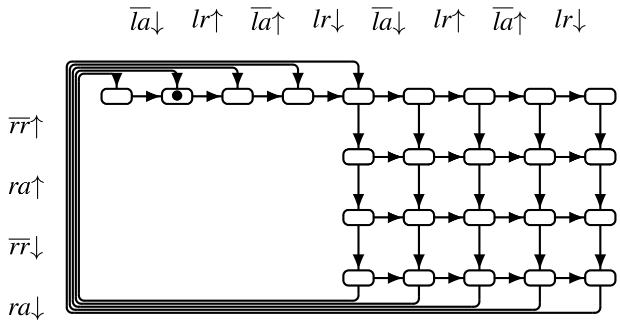

Figure 6.3: Minimized state graph of *max*, configured as a shape for late data validity protocols.

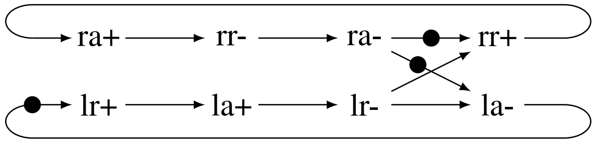

Figure 6.4: STG for the abstracted *max* protocol for late data validity protocols.

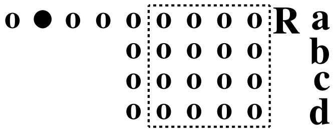

Figure 6.5: Right cut $\mathscr{R}$ denotation and range for late data validity protocols.

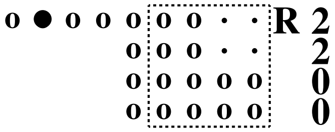

Figure 6.6: The shape resulting from cutaway R2200.

6.3.1.2 Concurrency Reduction from Left Cuts $\mathscr{L}$

Similar to the right cuts, the left cuts are denoted as *Labcd*. This cut removes from *max* *a* states from the left of row 2, *b* from row 3, *c* from row 4, and *d* from the left of row 1. Note that the *d* cut represents the number of states that are removed from the row 1 but for representation an extra row is added which is simply a duplicate of the row 1 (Fig. 6.7). Since the initial state has to be present and reachable for liveness, only one state can be removed from each row resulting in the potential candidates for a left cut now lie in a 1×4

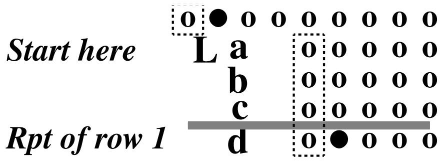

Figure 6.7: Left cut $\mathscr{L}$ denotation and range. The top row is duplicated at the bottom of the shape to more easily show the left cut ordering.

block of states in the dashed boxed of Fig. 6.7. The result of cut L0011 is depicted in Fig. 6.8. The family of all $\mathscr{L}$ cuts is generated by the constraints:

$$0 \leq a, b, c, d \leq 1$$
$$a \leq b \ \wedge \ b \leq c \ \wedge \ c \leq d \tag{6.4}$$

6.3.1.3 Untimed Protocol Family

The complete family of protocol shapes is generated by applying concurrency reduction using left cuts and right cuts to *max*. Each of these protocols are called untimed since they can only control the outputs of the circuit i.e., *rr* and *la* and have no control over the incoming inputs i.e., *lr* and *ra*. Hence, there are no timed protocol considered since they require control over the environment. Similar to the early protocol exploration, only untimed (*delay insensitive* (DI) and *speed independent* (SI)) protocols are considered.

L0000$\circ$R4422 representation is used to distinguish between the different protocols based on the left cut and the right cut used to generate them.

Not all these shapes are valid: for example the shape L1111$\circ$R4400 is not live since it deletes all the states from row 2 of the shape. Using an obvious cut indexing, shape Labcd$\circ$Rabcd is live iff Eqn. 6.5 holds. Thus the liveness of a shape can be calculated directly from its cuts from *max*.

$$La + Rb < 5 \ \wedge \ Lb + Rc < 5 \ \wedge \ Lc + Rd < 5 \ \wedge$$
$$La + Ra < 5 \ \wedge \ Lb + Rb < 5 \ \wedge \ Lc + Rc < 5 \ \wedge \ Ld + Rd < 5 \tag{6.5}$$

The right and left cut constraints in Eqn. 6.3 and 6.4 express all cuts, including the burst-mode and relative timed protocols. Constraint rule R1 must additionally hold for all untimed protocols. Delay insensitive protocols must also obey constraint R2.

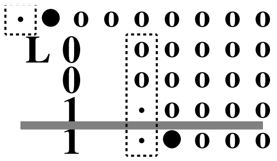

Figure 6.8: The shape (above the duplicated line) resulting from cutaway L0011.

1. **R1:** input signals *lr* and *ra* must always be accepted
2. **R2:** output signals may be delayed only by inputs

- **The SI family:** When rule R1 is added to the cut and liveness constraints of Eqn. 6.3, 6.4 and 6.5 we obtain the speed independent family of cuts. States must be removed in 1×2 pairs by left cuts and 2×1 pairs by right cuts. For example, referring to Fig. 6.3, one cannot remove just the rightmost state in any row (e.g., cut R1100) or the input $lr\downarrow$ is delayed (resulting in a timed design). The state to the left must also be removed (giving cut R2200). R1 is enforced by the following cut equation.

$$\begin{aligned} \mathscr{R}: &\quad a,b,c,d \text{ are even} \\ \mathscr{L}: &\quad a=b \wedge c=d \end{aligned} \qquad (6.6)$$

- **The DI family:** Adding rules R1 and R2 to our base cut constraints creates the delay insensitive protocols. This is more restrictive than the SI cuts, requiring states to be removed in 2×2 blocks. Thus it enables both the outputs to occur concurrently and removes the “output ordering” in the protocol. For example, referring to Fig. 6.3, if the rightmost two states are cut in the top row, output $\overline{la}\uparrow$ is delayed, producing cut R2000. To obey R2 and prevent output $\overline{la}\uparrow$ being delayed by output $\overline{rr}\uparrow$, the right two states must also be removed in the second row. This produces the DI cut R2200. R1 and R2 are enforced by the following equation on both $\mathscr{L}$ and $\mathscr{R}$ cuts. There is only one DI $\mathscr{L}$ cut present for this family of protocol and it is L0000.

$$\mathscr{R},\mathscr{L}: \quad a,b,c,d \text{ are even} \wedge a=b \wedge c=d \qquad (6.7)$$

6.3.1.4 $\mathscr{L}$ and $\mathscr{R}$ Cut Lattices

The DI definition is a subset of the SI family. Rather than subtracting them out, we prefer to keep them all and refer to it as the DI/SI family. This results in 3 left cuts and 15 right cuts. Composing members of the $\mathscr{L}$ and $\mathscr{R}$ cuts to create protocol shapes gives 45 possible protocols, where 13 are not live as their cuts violate the liveness constraint Eqn. 6.5.

Similar to the early data validity protocols, this family of protocols also forms symmetric lattices for both sets of cuts and each cut has a complement. The left and the right lattices are shown in Fig. 6.9. The lattice of left cuts has three members and is symmetric about an axis through cuts L0011. Each cut Labcd has a complement given by L(1-d)(1-c)(1-b)(1-a). The cuts on the axis are self-complementary. The lattice of right cuts has 15 members and is symmetric about an axis through R2222, R4220, R4400 (which are self-complementary). Each cut Rabcd has a complement given by R(4-b)(4-a)(4-d)(4-c). Thus, these sets of protocol lattice look similar to those shown for the early data validity protocols and they also show the same self-complementary left and right cut sets.

6.4 Results

The complete set of valid late data validity protocols were characterized using the flow described in Chapter 3. Each protocol is specified in CCS and synthesized and technology mapped using Petrify ([53]). Reset is added to the technology mapped circuit using the reset addition algorithm from Chapter 4. The analysis of the circuits is manually done to find the actual GCPs as done for all the early data validity protocols in Chapter 5. $lr \rightarrow rr$ and $lr \rightarrow la$ timing paths were considered for these controllers as for the early data validity protocols. Each controller is then mapped to the handshake controller of a 4-deep linear FIFO (first in first out) structure with the datapath abstracted out and run through the synchronous CAD tools. Numbers are reported for these circuit implementations for forward latency, backward latency, cycle time, buffering depth, area, and energy.

Tables 6.1 and 6.2 describe the complexity of the circuit implementations for the late data validity protocols by detailing the number of gates and cycles present. The largest circuit for this protocol family consists of 11 gates before the addition of reset for power or performance optimization as shown in Chapter 4. The number of cycles present in the circuit ranges from 1 to 15. The *max* protocol is the most complex among the set of circuit implementation.

Two sets of designs were generated based on the optimization selected during the reset addition phase of the tool flow. The forward latency comparison for designs generated

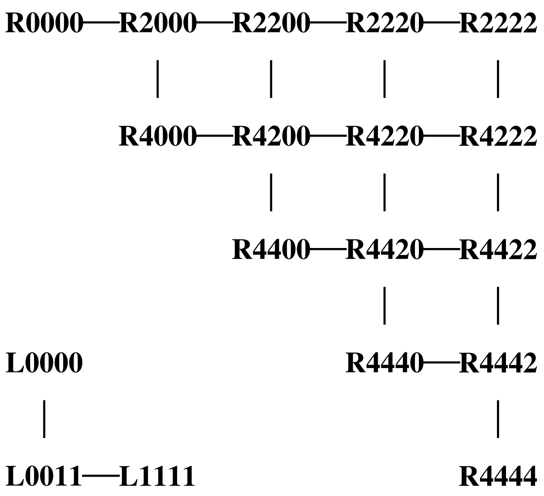

Figure 6.9: The symmetric lattices of untimed DI/SI left and right cuts.

using power and performance optimization is shown in Tables 6.3 and 6.4, respectively. Similarly, backward latency comparison using power and performance optimization is shown in Tables 6.5 and 6.6, respectively, and cycle time comparison is shown in Tables 6.7 and 6.8, respectively. Overall the trend is towards better performing circuits for the performance optimization with a few outliers. On analysis it is seen that a few big complex gates like AOI32 have bigger load to drive resulting in them being slower and hence, leading to reduction in performance.

Table 6.9 show the buffering depth for each protocol and the reduction in concurrency leads to fewer tokens which can be present when the pipeline is stalled. Note that the L0000$\circ$R4444 has a buffering depth of 0 since it is an unpipelined controller. Its definition says that the right hand side handshake must complete before the $la\downarrow$ signal is generated. Behaviorally, this protocol is similar to just a wire connecting lr to rr and la to ra. Hence, the cycle time of the protocol is very high but due to highly sequential nature, forward latency and backward latency numbers are very small.

Tables 6.10 and 6.11 show the area comparison between the power and performance optimization based reset addition. The general trend is that performance optimization duplicates the logic gate from a latency path to the feedback and thus results in increase in the area. There are a few outliers in this example set which on further analysis showed the different sizing for the gates employed by *Design Compiler* (DC) for different fanout

Table 6.1: Number of gates before adding reset.

LoR	L0000	L0011	L1111
R0000	11	8	8
R2000	9	9	7
R4000	9	7	–
R2200	7	8	5
R4200	8	7	–
R2220	9	8	6
R4220	7	6	–
R2222	9	7	7
R4222	5	6	–
R4400	7	3	–
R4420	6	6	–
R4440	7	–	–
R4422	4	3	–
R4442	4	–	–
R4444	4	–	–

Table 6.2: Total number of cycles found.

LoR	L0000	L0011	L1111
R0000	15	7	10
R2000	7	7	7
R4000	10	7	–
R2200	3	4	4
R4200	6	4	–
R2220	10	6	7
R4220	4	4	–
R2222	6	3	3
R4222	2	3	–
R4400	4	2	–
R4420	3	3	–
R4440	3	–	–
R4422	1	1	–
R4442	1	–	–
R4444	1	–	–

loads in the circuit. For these outliers, the distribution of the fanout load into two separate branches resulted in smaller sized gates and consequently, reduction in area.

The same conclusion can be made by comparing the power consumption (Tables 6.12 and 6.13), simulation time (Tables 6.14 and 6.15) and energy numbers (Tables 6.16 and 6.17) for the circuit implementations generated using power and performance optimizations. The presence of undersized gates for certain cases like L0000$\circ$R0000 for performance optimization results in longer simulation time but also smaller and low power circuit even though extra logic gets added. The presence of few extra timing constraint paths over and above the two timing paths can result in improvements in performance for paths which are currently not optimized automatically by the CAD tools.

Table 6.3: Forward latency ($ps/pipestage$) (Power optimization).

LoR	L0000	L0011	L1111
R0000	359.50	425.25	433.50
R2000	392.75	349.25	413.50
R4000	264.75	346.25	–
R2200	383.00	310.50	369.75
R4200	349.00	384.00	–
R2220	356.50	342.25	418.25
R4220	290.00	286.00	–
R2222	381.00	329.75	396.50
R4222	332.75	278.75	–
R4400	283.25	331.75	–
R4420	319.75	311.50	–
R4440	309.25	–	–
R4422	287.50	253.75	–
R4442	306.00	–	–
R4444	264.50	–	–

Table 6.4: Forward latency ($ps/pipestage$) (Performance optimization).

LoR	L0000	L0011	L1111
R0000	387.75	376.75	469.25
R2000	392.75	324.75	406.50
R4000	299.25	338.50	–
R2200	370.25	278.25	316.50
R4200	283.50	342.75	–
R2220	399.75	342.25	367.50
R4220	268.00	274.00	–
R2222	334.00	329.75	342.00
R4222	325.00	278.75	–
R4400	255.50	331.75	–
R4420	290.75	311.50	–
R4440	309.25	–	–
R4422	289.00	270.75	–
R4442	327.25	–	–
R4444	255.50	–	–

6.5 Hand-optimized Handshake Controller

Analysis of the results in Sec. 6.4 for the late data validity protocol implementations resulted in the selection of L0011 $\circ$ R4220 protocol for hand optimization. L0011 $\circ$ R4220 protocol is one of the best performing circuits and also has reasonable concurrency reduction applied to it. The technology mapped implementation for this circuit generated by Petrify is shown in Fig. 6.10. Analysis of the protocol and the EQN file generated by Petrify were then manually technology mapped to the implementation shown in Fig. 6.11. The shifting of the inversion at the inputs of the protocol implementation led to a good gate count reduction from 8 to 5 for the design.

The data for late data validity protocol gets latched before the $la\downarrow$, and this latching operation has to be delayed until $lr\downarrow$ since the data word is valid only at the falling edge of

Table 6.5: Backward latency ($ps/pipestage$) (Power optimization).

LoR	L0000	L0011	L1111
R0000	372.00	212.75	99.75
R2000	144.75	178.25	162.75
R4000	230.75	216.50	–
R2200	148.00	141.00	136.75
R4200	206.75	194.50	–
R2220	118.00	137.25	148.25
R4220	210.50	173.00	–
R2222	229.50	262.25	313.75
R4222	302.50	258.50	–
R4400	296.50	234.50	–
R4420	269.50	243.75	–
R4440	322.75	–	–
R4422	89.75	67.00	–
R4442	128.25	–	–
R4444	47.50	–	–

Table 6.6: Backward latency ($ps/$ $pipestage$) (Performance optimization).

LoR	L0000	L0011	L1111
R0000	369.75	254.25	85.50
R2000	144.75	168.75	145.25
R4000	226.75	198.00	–
R2200	131.25	114.00	120.75
R4200	190.25	174.75	–
R2220	123.00	137.25	125.75
R4220	163.25	146.00	–
R2222	208.25	262.25	261.50
R4222	291.50	258.50	–
R4400	270.00	234.50	–
R4420	222.75	243.75	–
R4440	322.75	–	–
R4422	79.50	55.75	–
R4442	138.75	–	–
R4444	48.00	–	–

request signal. This results in a very small duration for the clock signal to be enabled. The clock signal for the implementation shown in Fig. 6.11 is generated as a pulse, and the pulse width is controlled by the delay of the path $[lr\downarrow U2\ U1\ U6]$. Hence, a min delay constraint is required to fix this pulse width otherwise the simulations with inertial model of delay results in no latching of data at the register element.

The hand optimization of this design resulted in a forward latency, backward latency and cycle time of 267ps, 119.25ps and 430ps, respectively, as compared to 286ps, 173ps, and 483 ps for the initial design generated by Petrify with reset addition for power optimization. The area and energy numbers for the hand optimized design are $65.160um^2$ and 0.227pJ/token. The hand-optimized design is 47 percent smaller, 12 percent faster and consumes 52 percent less energy as compared to the design generated by Petrify.

Table 6.7: Cycle time (*ps*) (Power optimization).

LoR	L0000	L0011	L1111
R0000	714	572	513
R2000	510	506	585
R4000	493	555	–
R2200	547	477	513
R4200	568	539	–
R2220	609	528	568
R4220	568	483	–
R2222	614	581	652
R4222	646	535	–
R4400	555	568	–
R4420	614	551	–
R4440	675	–	–
R4422	662	562	–
R4442	765	–	–
R4444	1500	–	–

Table 6.8: Cycle time (*ps*) (Performance optimization).

LoR	L0000	L0011	L1111
R0000	714	555	559
R2000	510	480	576
R4000	477	551	–
R2200	520	428	490
R4200	477	539	–
R2220	614	528	506
R4220	474	436	–
R2222	547	581	635
R4222	646	535	–
R4400	513	568	–
R4420	539	551	–
R4440	675	–	–
R4422	650	576	–
R4442	842	–	–
R4444	1470	–	–

6.6 Summary

The systematic approach of starting with a maximum concurrency protocol and then reducing the concurrency with a set of left and right cuts is extended to the late data validity protocols. Rules similar to those generated for early protocols in [67] and [35] are derived for late data validity protocols.

A complete set of untimed late data validity protocols has been characterized for forward latency, backward latency, cycle time, area, and energy. The data give an insight on the effects of concurrency reduction on various design parameters, thus assisting in selection of the best design for any specification. The reduction in concurrency can result in simpler and faster handshake controller, thus resulting in high throughput design with fast cycle time. Also, a large area and energy benefit is gained. The benefits of concurrency reduction are

Table 6.9: Buffering depth for circuits.

LoR	L0000	L0011	L1111
R0000	4	4	4
R2000	4	4	4
R4000	4	4	–
R2200	4	4	4
R4200	4	4	–
R2220	4	4	4
R4220	4	4	–
R2222	4	4	4
R4222	4	4	–
R4400	4	4	–
R4420	4	4	–
R4440	4	–	–
R4422	2	2	–
R4442	2	–	–
R4444	0	–	–

demonstrated by comparing the Petrify circuit implementation with power optimization for L0011$\circ$R4220 cut with the *max* protocol. The *max* protocol has 48 percent slower cycle time, 111 percent larger area and consumes 105 percent more energy with respect to the L0011$\circ$R4220 cut circuit implementation.

The data word is valid at the falling edge for the late protocols, leading to efficient implementation for design where the datapath delay is large. As compared to early data validity protocols, the size of the delay elements between pipeline stages can be nearly halved on the control path, thus resulting in a smaller area and lower energy consumption. Going twice through the delay element for the set phase of a data transfer can be too restrictive and can lead to performance penalty, especially for designs requiring very fast forward latency.

Table 6.10: Routed core area (um^2) (Power optimization).

LoR	L0000	L0011	L1111
R0000	258.912	216.864	284.592
R2000	200.580	162.000	147.480
R4000	188.580	172.320	–
R2200	148.320	113.184	119.184
R4200	147.480	116.592	–
R2220	193.740	150.900	132.000
R4220	132.864	122.592	–
R2222	136.320	102.864	99.456
R4222	78.876	85.716	–
R4400	102.864	82.296	–
R4420	109.728	82.296	–
R4440	140.592	–	–
R4422	69.444	48.000	–
R4442	61.740	–	–
R4444	61.740	–	–

Table 6.11: Routed core area (um^2) (Performance optimization).

LoR	L0000	L0011	L1111
R0000	235.728	155.160	243.432
R2000	200.580	204.900	186.900
R4000	200.580	144.900	–
R2200	128.592	144.900	98.592
R4200	168.900	133.728	–
R2220	208.296	150.900	120.000
R4220	159.480	138.000	–
R2222	153.480	102.864	127.728
R4222	89.172	85.716	–
R4400	113.184	82.296	–
R4420	130.320	82.296	–
R4440	140.592	–	–
R4422	72.864	58.284	–
R4442	72.000	–	–
R4444	68.580	–	–

Based on the results for the concurrency reduction, the L0011 $\circ$ R4220 cut is identified as one of the best in terms of performance and concurrency reduction. A hand optimized circuit is developed and details of the optimization are presented. The results of comparing the hand optimized circuit with the circuit generated by Petrify are reported. The hand optimized circuit is 47 percent smaller in area, has 12 percent faster cycle time and consumes 52 percent lower energy as compared to the one generated by Petrify.

Table 6.12: Power consumed (*mW*) (Power optimization).

LoR	L0000	L0011	L1111
R0000	1.310	1.586	1.952
R2000	1.608	1.073	0.857
R4000	1.378	1.127	–
R2200	0.859	0.858	0.738
R4200	0.899	0.715	–
R2220	1.163	0.948	0.767
R4220	0.869	0.929	–
R2222	0.762	0.573	0.542
R4222	0.353	0.479	–
R4400	0.659	0.375	–
R4420	0.700	0.553	–
R4440	0.774	–	–
R4422	0.291	0.230	–
R4442	0.220	–	–
R4444	0.120	–	–

Table 6.13: Power consumed (*mW*) (Performance optimization).

LoR	L0000	L0011	L1111
R0000	1.222	0.936	1.442
R2000	1.608	1.352	1.189
R4000	1.537	0.967	–
R2200	0.792	1.244	0.695
R4200	1.276	0.891	–
R2220	1.242	0.948	0.893
R4220	1.254	1.179	–
R2222	1.028	0.573	0.744
R4222	0.429	0.479	–
R4400	0.796	0.375	–
R4420	0.913	0.553	–
R4440	0.774	–	–
R4422	0.333	0.283	–
R4442	0.236	–	–
R4444	0.149	–	–

Table 6.14: Simulation time (*ns*) (Power optimization).

LoR	L0000	L0011	L1111
R0000	187.910	151.440	136.760
R2000	135.550	135.140	154.730
R4000	131.500	146.860	–
R2200	144.910	127.310	136.710
R4200	150.400	143.420	–
R2220	161.440	139.650	150.900
R4220	151.490	128.770	–
R2222	162.040	150.850	171.260
R4222	169.190	141.490	–
R4400	149.620	150.320	–
R4420	163.290	145.900	–
R4440	179.330	–	–
R4422	174.360	148.730	–
R4442	200.960	–	–
R4444	386.980	–	–

Table 6.15: Simulation time (*ns*) (Performance optimization).

LoR	L0000	L0011	L1111
R0000	188.020	147.440	148.610
R2000	135.550	127.650	152.660
R4000	127.290	146.830	–
R2200	137.730	114.440	130.390
R4200	127.200	143.000	–
R2220	162.620	139.650	134.160
R4220	126.920	116.490	–
R2222	144.760	150.850	167.190
R4222	170.170	141.490	–
R4400	138.050	150.320	–
R4420	143.290	145.900	–
R4440	179.330	–	–
R4422	171.280	152.600	–
R4442	220.390	–	–
R4444	378.260	–	–

Table 6.16: Energy consumed ($pJ/token$) (Power optimization).

LoR	L0000	L0011	L1111
R0000	0.962	0.938	1.043
R2000	0.852	0.566	0.518
R4000	0.708	0.647	–
R2200	0.486	0.427	0.394
R4200	0.528	0.401	–
R2220	0.733	0.517	0.452
R4220	0.514	0.468	–
R2222	0.482	0.337	0.363
R4222	0.233	0.265	–
R4400	0.385	0.220	–
R4420	0.446	0.315	–
R4440	0.542	–	–
R4422	0.198	0.134	–
R4442	0.173	–	–
R4444	0.181	–	–

Table 6.17: Energy consumed ($pJ/token$) (Performance optimization).

LoR	L0000	L0011	L1111
R0000	0.897	0.539	0.837
R2000	0.852	0.674	0.709
R4000	0.764	0.555	–
R2200	0.426	0.556	0.354
R4200	0.634	0.498	–
R2220	0.789	0.517	0.468
R4220	0.622	0.537	–
R2222	0.581	0.337	0.486
R4222	0.285	0.265	–
R4400	0.429	0.220	–
R4420	0.511	0.315	–
R4440	0.542	–	–
R4422	0.223	0.169	–
R4442	0.203	–	–
R4444	0.221	–	–

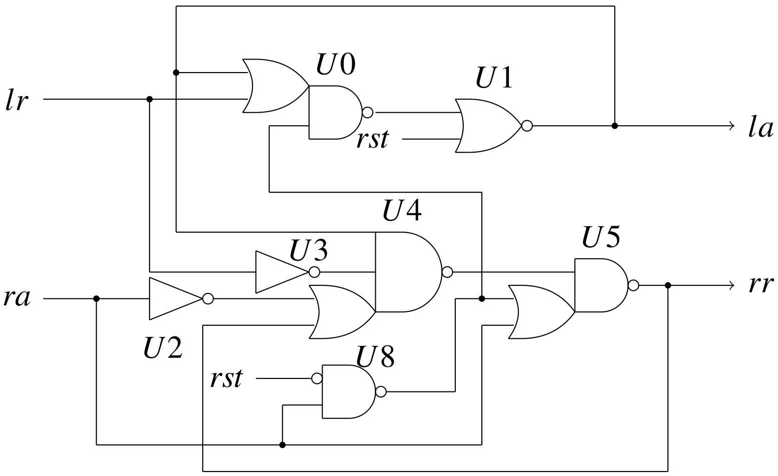

Figure 6.10: L0011 ◦ R4220 circuit implementation using Petrify.

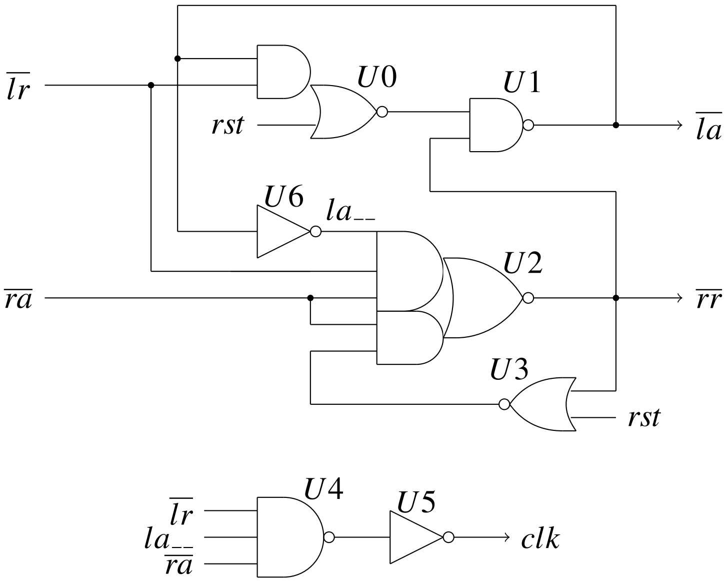

Figure 6.11: Hand-optimized L0011 ◦ R4220 circuit implementation.

CHAPTER 7

CASE STUDIES

Application of any design flow needs to be validated on designs of varying sizes. Also, templates for characterization of designs is necessary for the comparison and adoption of any circuit. This chapter describes the exploration of various circuits and characterization benchmarks that have been developed.

7.1 Key Contributions

- Application of *relative timing* (RT) based methodology and flow described in Chapter 3 on a large design.
- Addressing important methodology issues like scan insertion and timing closure.
- Automation of template characterization with some benchmark circuits.
- Application of the methodology on designs with purely asynchronous and mixed synchronous and asynchronous designs.

7.2 Toy Example

The generation of asynchronous templates requires a few different benchmark circuits to characterize the circuit implementations. One such benchmark circuit is demonstrated in this case study with respect to the linear pipeline controller LC shown in Fig. 3.4. This template is named the toy example based on its simplicity as shown in Fig. 7.1. There are only two asynchronous templates in this design, the *linear controller* (LC) and the Fork Join template (F/J). The logic level implementation of this design consists of a three stage pipeline which calculates the function x^2+3x.

This example is used initially to describe the two ways constraints can be specified for a system and their advantages and disadvantages. The controllability of delays and use of different register elements in conjunction to the design are also shown. The wire-load model exploration for the Artisan 65nm library is detailed with proposed improvements to it, thus

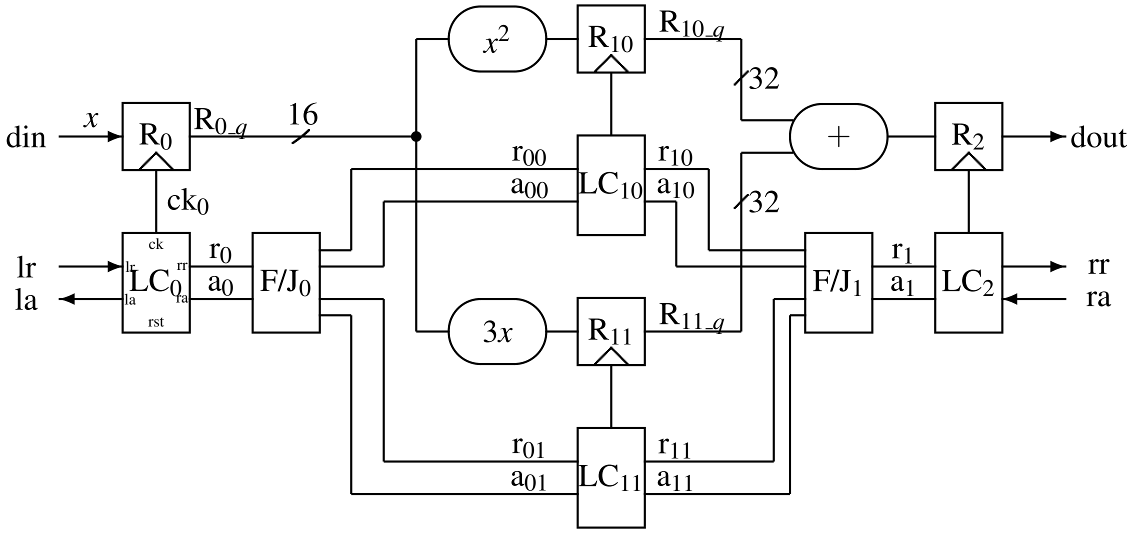

Figure 7.1: Example design: a simple ASIC mathematical pipeline segment computing dout $= x^2 + 3x$.

allowing better estimation of the delays at the *Design Compiler* (DC) synthesis step. Lastly, the application of scan insertion to implement testability of the datapath using Tetramax is shown.

7.2.1 Constraint Specification

The asynchronous designs developed using the methodology described in this dissertation require constraints like set_size_only, set_dont_touch, set_disable_timing to be specified with respect to the circuit implementation. These constraints prevent the synchronous CAD tools to modify the asynchronous design structurally and also enable presenting the timing graph of these circuits as a DAG to the CAD tools for timing driven sizing and optimizations.

The performance targets and method of specifying these constraints have two different approaches, with their associated advantages and disadvantages. The toy example is used as the reference to show these approaches. The performance of any pipeline stage in a design is represented as the cycle time of the handshake channel between their pipeline controllers. The cycle time of these adjacent controllers can be controlled in different ways based on the paths that are specified to perform timing on the design. Considering the *LC* block in the toy example being the protocol with data valid at rising edge (Fig. 3.4), the different paths for the setup constraint between LC_0 and LC_{10} are as follows. The max delay path is from lr input to the datapath input of R_{10} via R_0 while the min delay path is from lr input to the clock pin of R_{10} via F/J_0 and LC_{10}. For the max delay path, there is a clear point where the path gets divided and hence the constraints from $lr \rightarrow R_0/clk$ and $R_0 \rightarrow R_{10}/D$ are specified. The performance of this pipeline is controlled by the min delay path and there are two different approaches to specifying the timing paths.

The first approach keeps all the paths local to a template and all the global paths start and end at the input of the template; while the second approach tries to specify as long a path as possible such that the number of constraints is reduced. As an example, let us consider the path from $LC_0/lr \rightarrow LC_{10}/lr$. This path can either be specified as it is or it can be divided into 2 segments, i.e., $LC_0/lr \rightarrow LC_0/rr$ and $LC_0/rr \rightarrow LC_{10}/lr$. It can be seen that although there are more constraints in the second case, it has a clear division between controllers in term of the paths. Hence, these approaches are named based on the path as $lr \rightarrow lr$ and $rr \rightarrow lr$.

- $lr \rightarrow lr$ approach - The advantages of this approach are the reduction in the number of constraint paths. It is more robust against delay variations since longer paths bring in the averaging effect for variations; this approach can achieve an overall better design since there is more margin for the tools to perform sizing gates and adding new ones for the delays, if required.
- $rr \rightarrow lr$ approach - The advantages of this approach are the easy computation of the size of the delay element required. If $lr \rightarrow rr$ path delay equals the margin for the min/max variations on the $lr \rightarrow clk$ path for a controller, then $rr \rightarrow lr$ can be equal to the combinational logic delay for the pipeline stage. Secondly, there is a clear separation between the hierarchy in terms of constraint specification. Thirdly, if constraints are specified for any controllers then the controller along with the constraints can be easily inserted in any design. The disadvantage of this approach is the increase in the number of constraints.

Based on the merits and demerits of the two approaches, the $lr \rightarrow lr$ approach is selected to be used for the rest of the work.

7.2.2 Delay Controllability in Pipelines with Variable Frequency of Operation [2]

Twelve different versions of our example are synthesized, simulated and evaluated in order to demonstrate the flexibility and advantages of the methodology described in the dissertation. The different versions are derived (i) mapping the design to latches or flops, (ii) using an incomplete set of constraints, (iii) having various frequencies for each pipeline stage, and (iv) applying time borrowing to the latch design. All designs started with the same behavioral module of Fig. 7.2 with one exception – the flop based designs required replacing the latch_active_high module with a structural flop bank. All designs are synthesized, physically placed and routed, and simulated using postlayout parasitics to generate delay and power results.

The reported results used the Artisan library for the IBM 65nm HVT (High threshold) 10sf process using full layout and parasitic extraction. DC is used for synthesis, Modelsim is used for simulation, and SoC Encounter is used for place, route, and parasitic extraction. The power and delay numbers used sdf parasitic back annotation into the Modelsim. The power numbers are generated using parasitic extraction and activity factors from a simulation run by importing a vcd file from Modelsim into SoC Encounter. The simulations are run exhaustively executing all input values from zero to 256 while also validating functionality. Postlayout timing is validated using the full set of constraints, including the DI wire constraints, using PrimeTime with extracted parasitics.

Two delays are critical in these designs for timing driven synthesis and place and route: the delay of the combinational logic and the delay of the control logic to ensure proper storing of the data. Each of these delays can be independently set for each pipeline stage. For all comparable designs, the combinational logic between flops or latches have the same target delay. However, the delay element between control logic may be sized differently based on the efficiency of synthesizing the control logic as described below.

It is ensured that the data word is valid before the rising edge of lr into the control logic for the LC protocol employed; and latches are operated in normally closed mode in the

[2] This section has been published in ASYNC, 2009 [41]. © 2009 IEEE. Reprinted with permission from Kenneth S. Stevens, Yang Xu, Vikas Vij "Characterization of Asynchronous Templates for Integration into Clocked CAD Flows," in 15th IEEE International Symposium on Asynchronous Circuits and Systems, May, 2009.

```
module toy (din, dout, lr, la, rr, ra, rst);
   input lr, ra, rst; output la, rr; input [15:0] din; output [31:0] dout;
   reg [31:0] R0, R10, R11, R2;
   ...
   assign dout = R2_q;

   always @(*)   R0 = din;
   linear_control      lc0        (.ck(ck0), .lr(lr), .la(la), .rr(r0), .ra(a0), .rst(rst));
   latch_active_high   R0_reg     (.d(R0), .clk( ck0), .q(R0_q));
   bcast_fork          bcf0       (.bi(r0),.bo0(r00),.bo1(r01),.ji0(a00),.ji1(a01),.jo(a0));
   always @(*)   R10 = R0_q * R0_q;
   linear_control      lc10       (.ck(ck10), .lr(r00),.la(a00),.rr(r10),.ra(a10),.rst(rst));
   latch_active_high   R10_reg    (.d(R10), .clk( ck10), .q(R10_q));
   always @(*)   R11 = R0_q * 3;
   linear_control      lc11       (.ck(ck11), .lr(r01),.la(a01),.rr(r11),.ra(a11),.rst(rst));
   latch_active_high   R11_reg    (.d(R11), .clk( ck11), .q(R11_q));
   bcast_fork          bcm0       (.bi(a1),.bo0(a10),.bo1(a11),.ji0(r10),.ji1(r11),.jo(r1));
   always @(*)   R2 = R10_q + R11_q;
   linear_control      lc2        (.ck(ck2), .lr(r1), .la(a1), .rr(rr), .ra(ra), .rst(rst));
   latch_active_high   R2_reg     (.d(R2), .clk( ck2), .q(R2_q));
endmodule // toy
```

Figure 7.2: The synthesized arithmetic Verilog for our toy example.

design. This results in an ability to time borrow based on the delay between la asserting and deasserting because new data do not need to be propagated forward until la lowers (see Fig. 3.1 and Fig. 7.3). Note that for efficient operation, a *unidirectional* delay between rr and lr in the pipeline is desired, where the rising delay is large and the falling delay is as small as possible. However, the scripts used result in the clocked CAD tools generating bidirectional delays. Unfortunately, bidirectional delays result in over a 100 percent delay overhead for protocols where data word is valid on the rising edge of lr. Efficient designs must employ different protocols like the ones described in Chapter 6 or unidirectional delays. However, this protocol works well for our example pipeline because it provides an ample time borrowing window. For the implemented design (see Fig. 7.1), the 16-bit multipliers of the second pipeline stage are much larger than the 32-bit adder delay in the final stage. This allows the stages previous to the adder stage to borrow some of its cycle time.

One of the primary examples of this tool flow is to evaluate the effectiveness of timing driven synthesis and place and route of the asynchronous templates. This is demonstrated

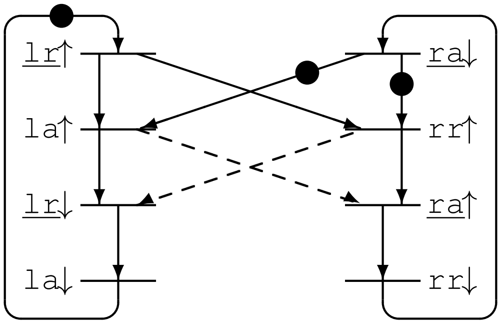

Figure 7.3: Petri net specification of linear control.

by utilizing an *incomplete constraint set* (ICS) from the template characterization, as well as the *full constraint set* (FCS) for each version of the design. The incomplete constraint set utilizes all of the relative-timing generated constraints, but allows the clocked CAD tools to utilize their internal cycle cutting algorithms to generate the timing DAGs. Thus, the incomplete constraint set leaves out the loop breaking constraints in the flow, as shown in Fig. 7.4.

Table 7.1 shows four designs synthesized to compare the pipeline using flops versus latches in the datapath. Comparing the flopped pipeline versus a latch pipeline gives the expected results: the latch design is more energy efficient (12 percent and 18 percent, respectively, for ICS and FCS) and smaller ($\approx$ 12 percent for both). The full constraint set (FCS) shows a large improvement in power and performance, and minor area reduction. The timing optimized design resulted in 35 percent and 40 percent reduction in energy for the flop and latch designs, respectively. Inspecting the postlayout netlist reveals that the ICS design substantially oversized many gates. For example, the AOI32 gate in Fig. 3.5 is sized 6× bigger for ICS versions as compared to the FCS versions of the design. This larger gate, while very energy inefficient, leads to a faster circuit that required larger delay elements for the flop circuit to operate correctly. This required creating a target control clock period 20 percent slower than the datapath period for the flopped design. Due to the speed-up of the control logic, the ICS design ran 11 percent slower. However, for the latch design, the same control target frequency as the FCS version can be used due to time borrowing that occurs. This results in a design that is 4 percent faster for the ICS as against FCS design.

```
breaking local cycles:
set_disable_timing -from A3 -to Y [find -hier cell *lc0]
set_disable_timing -from B2 -to Y [find -hier cell *lc0]
set_disable_timing -from A3 -to Y [find -hier cell *lc1]
set_disable_timing -from B2 -to Y [find -hier cell *lc1]
breaking handshake protocol cycles:
set_disable_timing -from A2 -to Y [find -hier cell *lc0]
set_disable_timing -from A2 -to Y [find -hier cell *lc1]
set_disable_timing -from B1 -to Y [find -hier cell *lc1]
```

Figure 7.4: Loop breaking constraints.

Table 7.1: Example comparing flop and latch based design with identical pipeline frequency. The ICS column uses an incomplete constraint set. Energy reported in pJ per token, clock period in nsec.

	Flip-Flops		Latches	
	ICS	FCS	ICS	FCS
Run Time (μsec)	1.519	1.358	1.333	1.391
Avg. energy (nJ)	0.762	0.493	0.673	0.406
Avg. sw. energy	0.673	0.158	0.305	0.169
Avg. intrnl energy	0.440	0.308	0.343	0.212
Avg. leakge enrgy	0.031	0.028	0.025	0.025
Area (mm^2)	12,724	12,294	11,215	10,770
Datapath clk per.	2.0	2.0	2.0	2.0
Control clk per.	2.5	2.0	2.0	2.0

Table 7.2 shows four new designs where the pipeline stages independently assigned delays to optimize the power-delay product for each pipeline function. The 16-bit multipliers are given a target frequency of 2.0ns, and the 32-bit adder a frequency of 1.4ns. This example shows that even with traditional clocked tools, our flows are able to directly synthesize and validate multifrequency pipelined designs. As is the case with a single frequency, the full constraint set results in lower area and power than the unconstrained set, as well as a faster design (ignoring time borrowing that occurs for the latched ICS version).

The final four designs show how this flow can be used to exploit time borrowing between pipeline stages in the clocked CAD with the same set of constraints by assigning different

Table 7.2: Version with variable pipeline frequencies.

	Flip-Flops		Latches	
	ICS	FCS	ICS	FCS
Run Time (μsec)	1.375	1.377	1.056	1.415
Avg. energy (nJ) Avg. sw. energy Avg. intrnl energy Avg. leakge enrgy	0.752 0.285 0.439 0.028	0.492 0.159 0.306 0.027	0.677 0.308 0.349 0.021	0.398 0.167 0.206 0.025
Area (mm^2)	12,878	12,258	11,516	10,887
Datapath clk per. multipliers adder	 2.0 1.4	 2.0 1.4	 2.0 1.4	 2.0 1.4
Control clk per. multipliers adder	 3.2 1.5	 2.0 1.4	 2.0 1.4	 2.0 1.4

delay values to the control path. The first two versions of the design, shown in Table 7.3, use a fixed frequency for all datapath pipeline stages. The last two versions use different frequencies for the multiplier and adder stages. The primary difference between the fixed and multifrequency designs is that the multifrequency design slightly constrains the worst case adder path, which results in a very small reduction in overall run-time (1.3 percent) and energy (1.2 percent). The most significant observation from these designs is the ability for time borrowing to mitigate variations in the design, whether the source is from poor frequency or design optimization (as can be seen by the energy difference of 44 percent).

All relative-timing constraints, including the delay insensitive constraints, are used to validate postlayout timing (using extracted layout parasitics imported as standard delay file) in Primetime. The timing report validated all the constraints used for timing driven synthesis and place and route to be correct with positive slack. In latch based pipeline implementation, the multiplication latch stages can use time borrowing from the next stage. Tables 7.4 and 7.5 show a brief summary of the timing reports.

Table 7.3: Latch based time borrowing versions with and without variable pipeline frequencies using incomplete and complete timing path constraints.

	ICS	FCS	ICS	FCS
Run Time (μsec)	0.922	0.929	0.922	0.917
Avg. energy (nJ)	0.670	0.378	0.670	0.377
Avg. sw. energy	0.309	0.160	0.309	0.158
Avg. intrnl energy	0.343	0.201	0.343	0.203
Avg. leakge enrgy	0.017	0.016	0.017	0.017
Area (mm^2)	11,264	10,739	11,258	10,937
Datapath clk per.				
multiplier	2.0	2.0	2.0	2.0
adder	2.0	2.0	1.1	1.1
Control clk per.				
multiplier	1.2	1.1	1.2	1.1
adder	1.2	1.1	1.1	1.1

Table 7.4: Data check timing report summary on some RT constraints. Listed slacks are all worst case.

RT Constraints	Setup (ns)	Slack (ns)
$lr\uparrow \mapsto rr\uparrow \prec y_{-}\downarrow$	0.05	0.16
$lr\uparrow \mapsto la\uparrow \prec y_{-}\downarrow$	0.05	0.12
$lr\uparrow \mapsto la\uparrow \prec ra_{-}\downarrow$	0.00	0.92
$lr\uparrow \mapsto rr\uparrow \prec lr\downarrow$	0.00	0.80

7.2.3 Wireload Models and its Impact on RT Methodology

The presence of *wireload models* (WLM) in a library helps in estimating the wire delays during synthesis [69]. For a wire with a given fanout, the wireload model specifies the capacitance, resistance, and area of the wire. An understanding of how these models work for this methodology needs to be investigated. The Artisan 65nm RVT (regular threshold) library is used to generate these results. The library consists of coarse wireload models with the base starting at WLM10 and going upto WLM50 with a step size of 10.

This study tries to generate the fastest design for the toy example selecting each wireload model specified. Any design is said to work if DC does not show any negative slack and goes through the synchronous CAD flows with design validation using SDF back-annotation.

Table 7.5: Timing report summary for constraints between pipeline stages. The latches in datapath borrow time from the next stages with LSup (library setup time), MxTB (maximum time borrowing) and TB (real time borrowing) listed. All the numbers are in nanoseconds.

PathType	From	To	Constr.	LSup	MxTB	TB/Slk
DataPath	R0	R10	max 1.70	0.20	0.65	0.25
DataPath	R10	R2	max 1.08	0.17	0.68	0.20
DataPath	R0	R11	max 1.70	0.17	0.68	0.01
DataPath	R11	R2	max 1.08	0.20	0.65	0.20
CtrlPath	tk0/lr	tk10/lr	min 1.19	N/A	N/A	0.12
CtrlPath	tk10/lr	tk2/lr	min 1.08	N/A	N/A	0.13
CtrlPath	tk0/lr	tk11/lr	min 1.19	N/A	N/A	0.12
CtrlPath	tk11/lr	tk2/lr	min 1.08	N/A	N/A	0.11

The final timing path validation step uses PrimeTime PX to validate the delays of all the specified min and max timing constraint paths. The maximum negative slack on the control path and the datapath are found based on the report from PrimeTime with the number of violating paths.

Table 7.6 shows a detailed comparison of the results of different WLM models. It is observed that the existing wireload models in the library are too pessimistic; and for a better estimate for these asynchronous designs which have local timing paths, better models need to be included. Hence, custom wireload models (WLM2, WLM2p5, and WLM5) are added for investigating their effects on timing and sizing the circuits during synthesis.

The results generated by this study show a transition point from where the types of the violating paths switch completely. The no wireload case resulted in all the violations reported for max delay paths by PrimeTime for place and routed design. While for the WLM5, all the violating paths are of the min delay type. As observed in Table 7.6 there is a sweet spot where the transition from max delay violators to min delay violators takes place, which for this study, is between WLM2 and WLM2p5. Hence, selecting either WLM2 or WLM2p5 can be very helpful for getting better designs and quicker timing closure. The addition of more pessimism in the estimation of the wireload leads to an increase in the negative slack of the violating paths. The results also show an improvement in the performance of the design until the design is simplified to such an extent that the extra

Table 7.6: Results for toy example using different wireload models.

	No WLM	WLM2	WLM2p5	WLM5	WLM10	WLM20	WLM30	WLM40
Datapath delay (*ps*)	600	620	628	730	905	1060	1220	1200
$lr \rightarrow lr$ min delay (*ps*)	600	620	628	730	905	1060	1220	1200
$lr \rightarrow lr$ max delay (*ps*)	630	651	659	767	950	1113	1281	1260
$lr \rightarrow clk$ min delay (*ps*)	90	50	50	80	150	250	350	350
$lr \rightarrow clk$ max delay (*ps*)	120	180	195	220	350	410	500	550
$lr \rightarrow y_-$ max delay (*ps*)	180	215	225	255	350	410	500	600
Post Place and Route Numbers								
Forward Latency (*ps*)	1590	1430	1350	1230	2100	2460	2930	1950
Backward Latency (*ps*)	2420	2390	2170	2020	4060	5140	6330	3120
Cycle Time (*ps*)	1790	1740	1490	1470	1330	1280	1340	2980
SimTime (*ns*)	343	297	285	281	255	245	257	559
Power (*mW*)	1.47	1.92	2.03	2.34	2.81	4.69	6.81	3.02
Negative Slack Path Type	max	max	min	min	min	min	min	min
Max Negative Slack on Control path (*ps*)	110	2	110	231	374	684	753	659
Max Negative Slack on Datapath (*ps*)	93	0	0	0	0	0	0	0
Number of Violating Paths	28	1	3	3	7	5	5	5
Area of Core (um^2)	4305.64	5523.48	6051.46	6938.04	8479.78	11324.02	17727.60	19374.98

delays added due to wireload model start to dominate. Also, the area of the core without the pad rings increases with the added pessimism.

7.2.4 Automatic Scan Insertion using Tetramax

Testability of any circuit postfabrication is a critical factor for any methodology and it needs to be explored for the proposed RT based methodology too. Since the methodology described in this dissertation adds the pipeline stages manually and the datapath logic is synthesized as a combinational block, it is necessary to see if the pipeline stages are seen correctly by the synchronous CAD tools and the register elements are replaced by scan elements. No change is required on the base Verilog for the toy example except for the addition of the scan clock, scan enable, scan in, scan out and test mode pins. Also, since the asynchronous control network is event driven, it would not be activated if all its control network inputs remain at logic level 0. This results in a simple OR gate with the scan clock and local clock generated by asynchronous handshake controller being used to select the clock at each pipeline register bank.

There are no modifications to the scripts for DC synthesis except for the addition of scan insertion based commands which are the same as in the case of synchronous design. An extra script for *automatic test pattern generation* (ATPG) using Tetramax is also developed. This script reads the scan insertion definition and test setup generated by DC, and it generates a set of vectors which can automatically test the datapath logic. All three Tetramax scan modes, i.e., basic scan, fast sequential scan and full sequential scan mode are validated.

The presence of the asynchronous inputs and also the DAG representation of the asynchronous circuits result in Tetramax recognizing them as combinational blocks and hence, it generates the test patterns for their validation. The testability of the asynchronous control network needs to be separate from that of the datapath. Therefore, the asynchronous control network blocks are marked as black boxes and the primary inputs to them are tied to their reset logic level, which is low for this implementation. Table 7.7 shows a comparison of the results of scan insertion for stuck-at-fault testing for the synchronous and asynchronous toy example implementation. The timing targets for the asynchronous implementation used the approach described in Chapter 3, while the scan clock with a slower frequency of operation is selected for the synchronous design. Test coverage of 98.18 percent is achieved

Table 7.7: Scan insertion coverage and stuck at fault summary.

Design		Asynchronous	Synchronous
Fault Class	Code	#faults	#faults
Detected	DT	7124	3782
detected_by_simulation	DS	(6253)	(2970)
detected_by_implication	DI	(871)	(812)
Possibly detected	PT	4	0
atpg_untestable-pos_detected	AP	(4)	
Undetectable	UD	29	5
undetectable-tied	UT	(16)	(5)
undetectable-redundant	UR	(13)	
ATPG untestable	AU	58	5
atpg_untestable-not_detected	AN	(58)	(5)
Not detected	ND	21	0
not-observed	NO	(21)	
total	faults	7236	3792
test	coverage	98.85%	99.87%
fault	coverage	98.45%	99.74%
ATPG	effectiveness	99.71%	100.00%

for the stuck at fault testing of the datapath of the toy example as compared to 99.87 percent for the synchronous implementation. The number of test patterns generated are 2.78× more for the asynchronous case with 50 patterns as against 18 patterns for the synchronous design. It is observed that in order to be sure of the controllability of the clock input based on the test mode, DC added an extra logic on the clock path which is gauranteed to be controlled by the scan mode, instead of the simple OR gate which is added manually in the asynchronous design before synthesis. All the *untestable faults* (AU) in the asynchronous design are due to this extra logic addition and also the asynchronous inputs as per the fault reports generated by Tetramax.

This study looks at using synchronous CAD tools to do datapath combinational logic scan testing. The testability of the asynchronous control network and applicability of scan testing approaches suggested by Beest, Roncken and Hazelwindus needs be explored in the future [70, 71, 72].

7.3 FIFO Design Automation for Template Characterization

Generation of FIFOs for various configurations is a good benchmark for characterization of asynchronous handshake controllers as well as templates to steer data. The general set of templates used for complete characterization involved select fork, select join, and bundled data handshake controllers. The goal of this case study is to generate a scripted flow to develop FIFOs of various configuration and of varying depth and datapath widths. The approach for FIFO characterization must be general enough to feed any set of constraints for the templates which can be used automatically with the FIFO designs and compare them against other designs.

There are four types of FIFOs that are examined in this study – linear, parallel, square and tree. The start point for each FIFO structure is from the work done by Hosuk Han for his dissertation work [73, 47]. Each of the designs done in [73] were manually generated and gate sizing information is generated by hand. This is a very tedious process which has been simplified drastically by automating the whole process with scripts to generate various FIFO structures with insertion of the constraints, so as to allow rapid comparison of different designs. It can assist in characterizing both the handshake controllers as well as the data and control steering blocks like select forks, select joins, etc.

This work is used to generate FIFO designs for conceptualizing the *source asynchronous signaling* (SAS) protocols [74]. The methodology presented in this dissertation along with the automation to rapidly generate comparisons for different FIFOs of varying depth with and without datapath, allowed the investigation of different SAS designs as has been reported in [74].

The overall automation process involved specifying an asynchronous handshake controller with all its cycle cut and relative timing constraints. The following performance paths specified to optimize and time the circuit include: $LC_i/lr \rightarrow R_i/clk$, $R_i/q \rightarrow R_{i+1}/d$, $LC_i/lr \rightarrow LC_{i+1}/lr$, $LC_{i+1}/lr \rightarrow LC_i/ra$. $LC_i/ra \rightarrow LC_i/rr$ and $LC_i/ra \rightarrow LC_i/la$ must-cut paths are used to automatically generate the cycle cut constraints for this design as explained in

Chapter 5. Cutting these paths resulted in the removal of the architectural cycles between the handshake controllers.

All the performance paths specified above have a max delay constraint associated to them, while the $LC_i/lr \rightarrow R_i/clk$ and the $LC_i/lr \rightarrow LC_{i+1}/lr$ constraints also have an associated min delay constraint. The goal for each synthesis operation is to achieve no negative slack for all the paths specified for a design. The delay values for each path are changed equal to the negative slack reported by DC after each step of its use. Margins are specified for competing paths. Based on the margin requirements, the min delays for the paths are increased automatically if the competing max delay path value is also increased. This approach is the key to achieving a functioning circuit.

The absence of a reference clock in asynchronous designs creates a problem with respect to balancing the $LC_i/lr \rightarrow R_i/clk$ path for each pipeline stage. If the delays of the $LC_i/lr \rightarrow R_i/clk$ path for each pipeline stage are not similar, then huge margins need to be added on the $LC_i/lr \rightarrow LC_{i+1}/lr$ path to compensate for it. To solve this problem, an extra run of DC is necessary. The constraint path $LC_i/lr \rightarrow R_i/clk$ is local and is unrelated to a global clock signals. Hence, the first run for achieving no negative slack for the circuit implementation resulted in very large delay for the $LC_i/lr \rightarrow R_i/clk$ path. This leads to a very inferior design and it can also result in setup violations. Hence, after the first run, the min and max delays of $LC_i/lr \rightarrow R_i/clk$ path are reset to a value which is deemed to be adequate for the controller being used. This value is calculated by characterizing the controller once by hand. After this, DC is rerun on the circuits to achieve no negative slack. The addition of this last step resulted in not only improving the overall quality of the designs, but it also resulted in resolving all the setup violations because of the balancing of the $LC_i/lr \rightarrow R_i/clk$ path for each pipeline stage.

7.3.1 Linear FIFO

The simplest of the four FIFOs is the linear FIFO. It does not involve any type of data steering logic and hence is simply used to characterize the handshake controllers for *forward latency* (FL), *backward latency* (BL) and *cycle time* (CT). This base setup is used to characterize a 4-deep bundled data pipeline for results shown in the Chapters 4, 5 and

6. The presence of the script resulted in automatically generating FIFO of any depth and datapath width.

Linear FIFO designs of depth 1 to 48 with datapath widths of 0, 8, 32 and 64-bits, are synthesized, place and route and validated for correct functioning using the methodology described in Chapter 3 of this dissertation. Fig. 7.5 show the concise results obtained for linear FIFOs with respect to the datapath widths. A range for the min, max and average values for various parameters such as forward latency, backward latency, cycle time, area, power, simulation time and energy are reported. Except for cycle time and simulation time, all the other parameters are divided by the buffering depth of the FIFO. Thus, the graph values for each datapath width gives a comparison of per pipeline stage variation of each parameter when the FIFO depth is changed.

The increase in width of the datapath results in an increased load on the clock pin for the handshake controllers. Hence, an extra margin of 30ps is added to the control network to prevent setup violations. This results in an increase in forward latency, cycle time and also simulation time. Overall the quality of results are pretty consistent for all the linear FIFO designs. The big variations seen between the average and the max values for the backward latency are due to the linear FIFO of depth 1, which results in very slow backward latency numbers. Overall, the addition of extra pipeline stages to the linear FIFO results in a linear increase in the various parameters considered for design comparisons and the effectiveness of this methodology.

7.3.2 Parallel FIFO

The parallel FIFO structure consists of a decimator and an expander modules to steer the request and data to and from a specific channel. A 1-to-n decimator directs the incoming request to any one of the *n* outputs. The order of the selection is done by a one-hot shift register. The decimator shift register switches to a new value on the falling edge of the incoming request signal from the upstream handshake controller. Similarly, an *n*-to-1 expander steers the data and incoming request from *n* input channels to one output channel based on the one-hot shift register. The expander shift register values are changed on the falling edge of the incoming acknowledge signal from the downstream handshake controller.

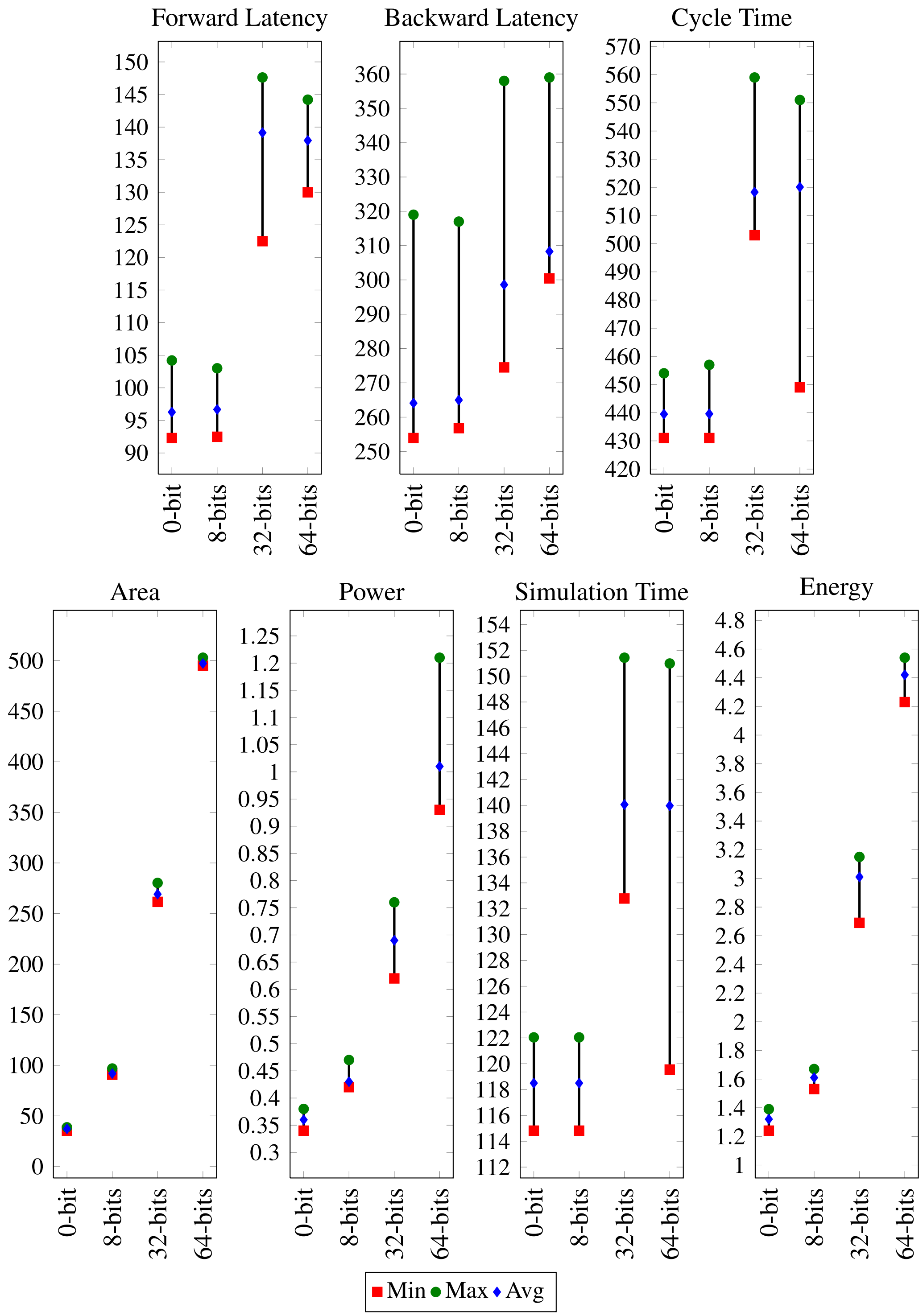

Figure 7.5: Comparison of results for linear FIFO. All reported numbers are averages with respect to the buffering depth of FIFO except for cycle time and simulation time.

Results are generated for a FIFO of channel widths from 2 to 7 with each channel depth being from 1 to 7. The representation used for the FIFO is $n \times m$ where n is the depth of the FIFO and m is the width of the FIFO, i.e., either the output channels of the decimator or the input channels of the expander.

Fig. 7.6 shows a comparison of the results generated for 42 different designs generated for four different datapath widths. The total forward and the backward latency of the parallel FIFO is less than that of the linear FIFO because of fewer pipeline stages, but per stage forward and backward latency numbers are larger because of the presence of decimator and expander. Similarly, the cycle time reported for this FIFO is also larger than the linear FIFO.

The increase in the datapath width resulted in setup violations at the expander interface because of the increase in the expander datapath delay. Hence, a larger min delay is added on the control network to generate a working circuit. This additional min delay results in a sudden increase in the forward latency, the cycle time and the simulation time for the 32-bit and 64-bit datapath designs. The decimator and expanders with $m \geq 5$ channels require min delay margins of at least 60ps more than datapath delay to derive a functioning circuit.

The parallel FIFO has only one path active at a time for any token, hence most of the circuit generally remains idle unless there is valid data to be transferred. This results in smaller energy numbers for parallel FIFOs as compared to linear FIFOs of the same buffering depth. The presence of extra logic for decimator and expander results in an area penalty and also larger cycle time for the parallel FIFO as opposed to linear FIFOs of same buffering depth.

7.3.3 Tree FIFO

The tree FIFO is similar to a binary tree wherein the data and control information is steered alternately via a select fork for each data transfer. Hence, incoming data word and request are steered to the n base nodes, where $n = 2^{depth}$. The inverse combination from the n base nodes to the output channel is then done using select join modules to alternately steer the incoming data and requests to the output channel.

There are $2^{depth} - 1$ pipeline stages for steering the data forward and the same number of pipeline stages combine the channels at the output. The total buffering depth for these FIFOs is $2^{depth+1} - 2$, because of the use of a fully buffered protocol for the handshake

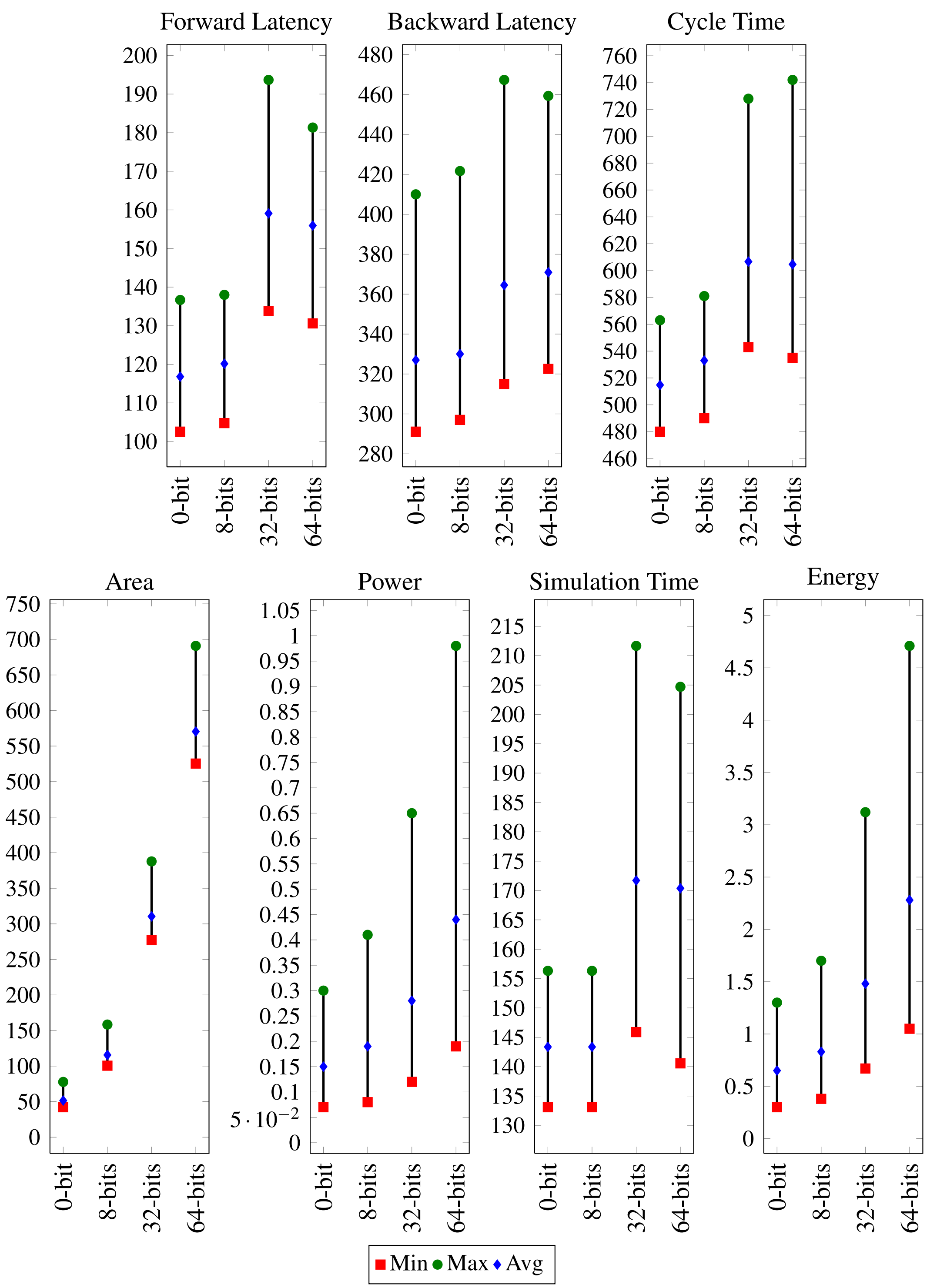

Figure 7.6: Comparison of results for parallel FIFO. All reported numbers are averages with respect to the buffering depth of FIFO except for cycle time and simulation time.

controllers. The results of comparison for tree FIFO with depth ranging from 2 to 5 and datapath widths 0, 8, 32 and 64-bits are reported in Fig. 7.7.

7.3.4 Square FIFO

The square FIFO consists of a different behavior for the data and control steering templates that are used. The steering for the channels is based on the placement of the module in the design. Hence, based on the position it can have a behavior of sending two tokens on one channel and one token on the other or three tokens on one and two on other. Thus, this FIFO can be used to characterize templates with data steering templates which are ordered but different from the select fork or the decimator or the expander.

The buffering depth for these FIFO is $depth^2$ and because of the exponential nature of growth of the buffering depth, there is a limited use of them. Fig. 7.8 shows the results for square FIFOs with depth 2 to 7 for datapath width of 0, 8, 32 and 64-bits.

7.4 OCP-IP Case Study for Synchronous and Asynchronous Domain Interfacing [4]

The complexity of current integrated circuits have resulted in a *system-on-chip* (SoC) revolution. Time-to-market, the need to limit design engineering costs, as well as other factors have resulted in the need for modularity and design reuse. Various system building blocks, called *intellectual property* (IP) blocks, are designed once and then used in multiple different systems. Such SoC designs contain IP blocks such as memory, general purpose processors, communication blocks such as busses or *network-on-chip* (NoC), and other specific function units and coprocessors.

There are number of technical challenges to creating modular IP blocks that can be reliably and rapidly interconnected to form SoC designs that span multiple applications and process technology nodes. The size and diversity of operation of the function blocks result in many of the IP blocks requiring independent operating frequencies. Traditional clocked methodologies encourage a single operating frequency for the entire chip. Integration of

[4] This work has been selected for publication in VLSI Design, 2014 [75]. © 2014 IEEE. Reprinted with permission, from Vikas S. Vij, Raghu Prasad Gudla, Kenneth S. Stevens, “Interfacing Synchronous and Asynchronous Domains for Open Core Protocol,” to appear in 27th International Conference on VLSI Design (VLSI-Design), Jan., 2014.

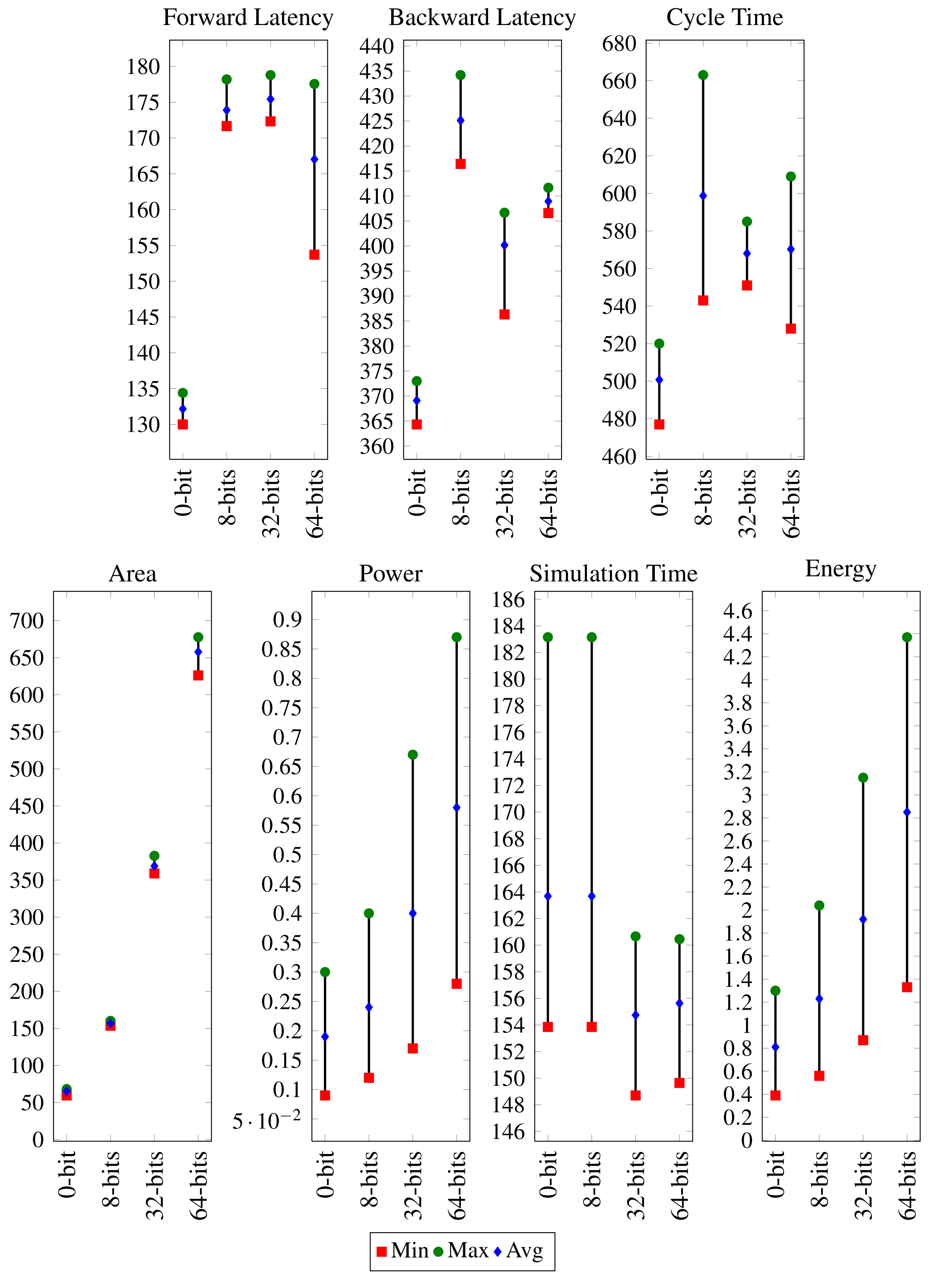

Figure 7.7: Comparison of results for tree FIFO. All reported numbers are averages with respect to the buffering depth of FIFO except for cycle time and simulation time.

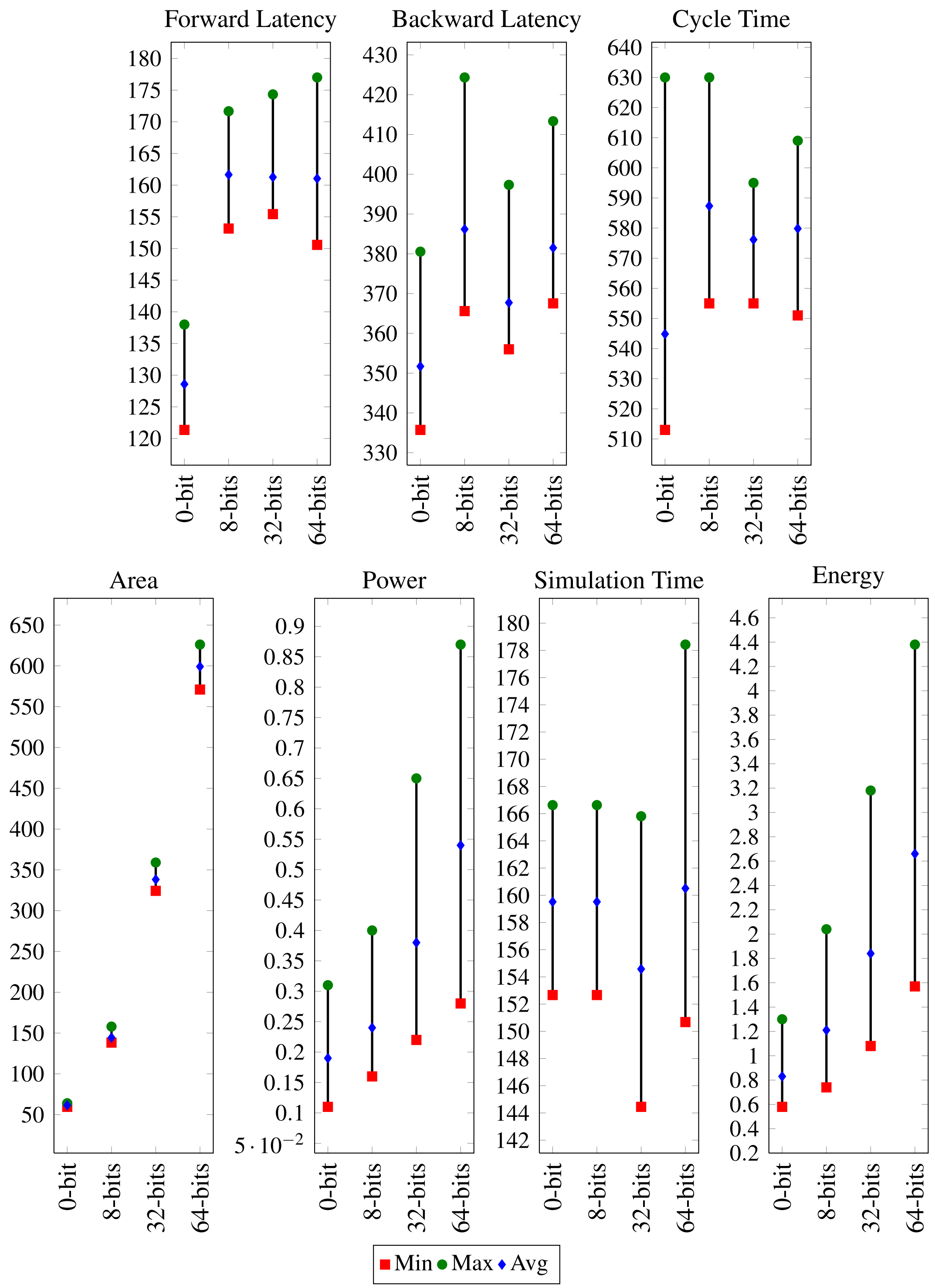

Figure 7.8: Comparison of results for square FIFO. All reported numbers are averages with respect to the buffering depth of FIFO except for cycle time and simulation time.

many IP blocks into an SoC is efficiently performed by interconnecting the design blocks with a bus or NoC. Due to design complexity and wire latencies, modern bus and NoC interfaces result in nondeterministic response delays. This often requires a complex interface to clocked systems which traditionally require events to occur on specific clock cycles.

Several common bus interface protocols have been created to enhance modularity and composability of IP blocks. These include protocols such as the *advanced microcontroller bus architecture* (AMBA), Wishbone, and *open core protocol* (OCP). If IP blocks and a bus (or NoC) have been enhanced to support such a protocol, then the IP blocks can be directly connected to communicate across the bus (or NoC). Such designs can be directly reused in any system that supports the protocol. An implementation of a large subset of the full OCP protocol is done for this dissertation [76]. The implementation is enhanced to facilitate multiple timing methodologies, simplify buffering, and to reduce the integration effort required to make an IP block OCP compliant.

Gudla implemented clocked OCP for IP cores and NoC fabric by introducing the concept of *domain interface* (DI) [77]. Purely synchronous implementation and synchronous implementation with asynchronous design is demonstrated and the corresponding numbers are reported. There are a few drawbacks in the OCP implementation performed in Gudla's work. Firstly, there is no mode selection present and the OCP protocol can operate in only one fixed mode. Secondly, it only covers the synchronous design combinations.

As a case study, this work implements a more general OCP implementation with mode selection and stall mechanism which allows wider scope of its applications for various implementations. It can operate in the basic OCP mode, with selections for burst and tag extensions. The selection of a select word of data bytes has also been enabled which is missing in [77]. A new state machine based circuit is implemented from scratch to add-in these features and make it more general. An additional asynchronous OCP implementation similar to the synchronous one is implemented to enable systems with different timing. This allowed the timing to be completely general so as to enable a *globally asynchronous locally synchronous* (GALS) system implementation, as well as to enable the use of asynchronous IP blocks in the design. The choice of timing and functionality of each IP block in an SoC needs to be carefully considered based on its specific power and performance targets. Without a modular design integration methodology this can lead to increased redesign effort, particularly for clockless asynchronous design or a new GALS architecture. Thus, this method enables the use of clocked or asynchronous IP blocks, including asynchronous NoC designs.

The contributions of this case study are as follows:

- The concept of a DI introduced by Gudla ([77]) is enhanced further for clockless domains, thus enabling interaction of different IPs with GALS, *locally asynchronous globally synchronous* (LAGS) and purely asynchronous network interfaces.
- A more general OCP circuit realization is achieved.
- An asynchronous OCP circuit implementation similar to the synchronous OCP is proposed.
- The modularity across all designs and even *network-on-chip* (NoC) components is increased as the OCP master and slave front end blocks can be reused. Also, the design of specific IP back-ends is simplified thanks to the options of interfaces available.
- This case study demonstrates the applicability of the methodology and tool flow described in Chapter 3 to asynchronous and synchronous designs simultaneously. The constraints and design blocks for both clocked and asynchronous designs are specified together.

7.4.1 Open Core Protocol (OCP) Background

OCP is a nonproprietary, open standard, core-centric protocol addressing IP core system-level integration requirements [76]. It is defined as a clocked system with unidirectional data transfer which assists in simplified core implementation, integration and timing analysis. OCP enables the design of IP cores independent of the other cores, thus enabling the reuse of IP. The goal of this protocol is to enhance the modularity and composability of the IPs without requiring redesign.

Fig. 7.9 shows a block diagram of the basic OCP implementation between a master IP core and a slave IP core communicating across an NoC. The IP cores have an OCP master/slave component directly integrated into their design.

The basic OCP implementation consists of two channels, a request channel and a response channel as shown in Fig. 7.10. Any read command issued by the master IP core results in a transfer on the request channel including the address associated with the read

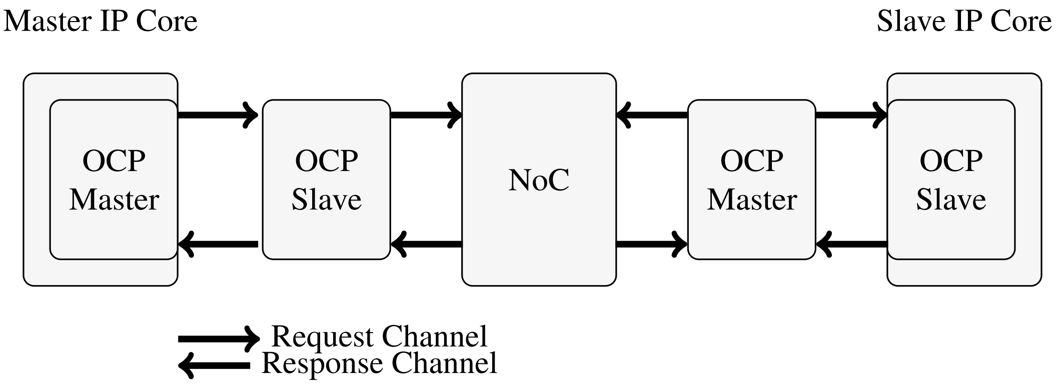

Figure 7.9: OCP implementation block diagram with native OCP master and slave.

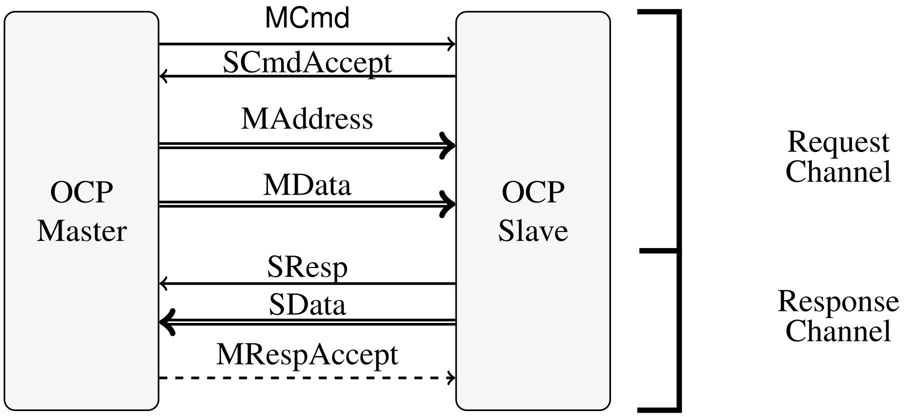

Figure 7.10: Basic OCP master and slave interface.

command. The slave IP core responds with data on the response channel for the master IP core. Similarly, write commands are issued across the request channel with the associated data and address. Various extensions are added in different versions of the OCP protocol such as the transfer of a burst of data, out-of-order responses, data handshake extensions, test control extension and a few more.

Some additional details of the OCP handshake protocol are introduced to help understand how traditional asynchronous handshakes can be mapped onto OCP[5]. The write command generated by the master IP core results in the OCP master interface defining a write command on the MCmd line, along with the Data and the Address information. The OCP slave block acknowledges the transaction by asserting the SCmdAccept signal on the request channel. Similarly for the read command, the master IP core sets the MCmd line to a read command

[5] Refer to the OCP manuals for full protocol information.

and asserts the Address. The OCP slave acknowledges this read command with SCmdAccept, thus completing the handshake with the NoC slave interface. For a normal read, the OCP master then waits for an acknowledgment on the response channel before initiating a new command. If out-of-order reads are employed the OCP master does not need to wait for responses and may immediately send the next command.

The basic handshake protocol for any request on the OCP channel involves an operation wherein the OCP master sets the MCmd wires on a request which is acknowledged by the SCmdAccept from the OCP slave. Similarly, there are other optional handshake signals that are asserted based on which OCP extensions are being employed. For example, if the write data handshake extension is enabled, an extra SDataAccept handshake signal is used to indicate an acceptance of Data by the OCP slave. Similarly for the response channel, there is an optional MRespAccept signal generated by the OCP master for any response acknowledgment. Thus, flow control is implemented using OCP handshake protocols.

The asynchronous designs replace the clock by embedding request / acknowledgment handshake signals across the OCP master-slave interfaces. The OCP signals are considered bundled data signals in relation to the request. The SCmdAccept and MRespAccept signals can be entirely replaced with the asynchronous acknowledgment signal.

7.4.2 Designs

A subset of the OCP protocol is implemented from the ground up to first develop a base synchronous implementation. The OCP subset used implemented normal read and write with extensions for burst mode as well as tags which enable out-of-order responses. Hence, there are four modes that are possible for this simple OCP implementation which include normal read/write, burst read/write, normal read/write with out-of-order responses and burst read/write with out-of-order responses. By default, we have enabled the data byte extension, which allows the selection of a certain group of 8-bits of datapath width. All these modes and selections are parameterized and can be changed easily during synthesis of the circuits.

Fig. 7.9 shows a block level implementation showing the communication network between a master and a slave IP with an OCP master, OCP slave and NoC. This standard implementation requires the OCP master and slave to integrate the OCP protocol into the IP and does not offer clear regions for *clock domain crossing* (CDC). Hence, changes to an IP block can lead to redesign of other design blocks if the CDC is moved into an adjacent block. To improve modularity, a DI is introduced by Gudla to interface between clock domains of different frequency [77]. The applicability of the DI is extended in this case study to cover domain crossing from and to asynchronous (unclocked) domains as well as for purely asynchronous implementations. This block contains a simplified handshake protocol and confines domain crossings to this block. It separates OCP design integration into a custom back-end specific to the IP block, and an OCP master and slave block. This results in full reuse of a single OCP master and slave block for all IP designs. The DI also provides a good location for synchronization and buffering, when needed.

Fig. 7.11 shows the explicit partition created by the domain interface. *Back-end* (BE) modules interface between the cores or NoCs and the domain interface. This architecture allows the BE blocks to take care of the interface between the DI and the IP operation without worrying about domain crossing. Thus, back-end modules can be directly integrated into the IP core if advantageous. The DI is where frequency domains are synchronized. Any blocks, including IP cores and NoC, can now be implemented independently and then interfaced with the DI or any similar interface using a back-end module. The correct domain crossings and buffering are implemented directly into the DI.

7.4.3 Asynchronous OCP Implementation

The OCP protocol implements a handshake between the OCP master and OCP slave blocks, which prevents overwriting of the data and allows for flow control. Response times can arbitrarily vary, e.g., while communicating across a NoC due to traffic and congestion. Therefore, a stall capability needs to be implemented into the OCP to IP back-end, so as to enable pausing of the cores to prevent overwriting of data when required by the OCP interface. The stall operations may require data buffering, which can result in significant complexity for traditional clocked IP blocks. Alternatively, asynchronous handshake modules natively allow for arbitrary stalls. Therefore, the design of asynchronous back-ends are normally much easier to build, are smaller, and consume lower power than clocked back-ends. Further, certain OCP handshake signals, such as SCmdAccept are directly implemented with the asynchronous handshake signals. Thus, OCP handshake signals can be removed from the asynchronous implementation.

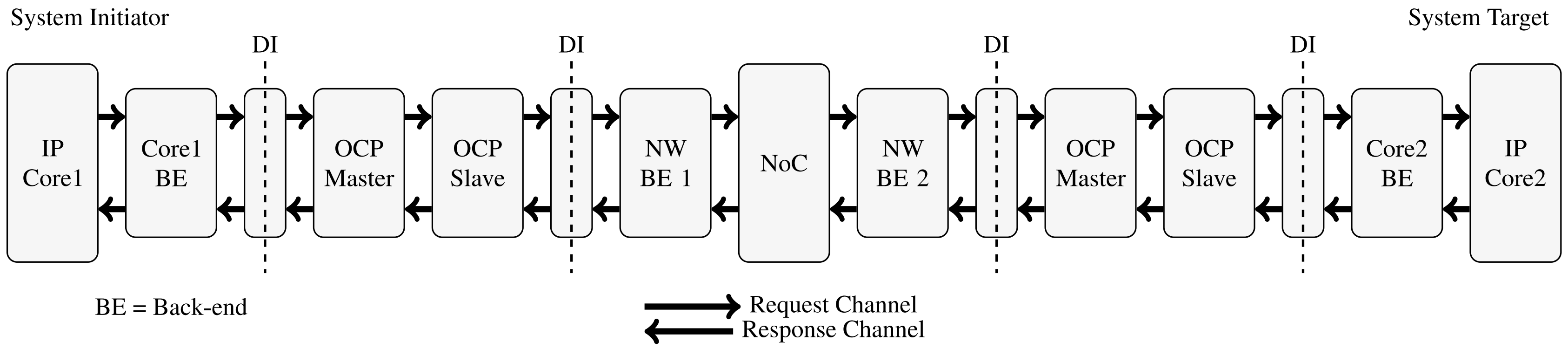

Figure 7.11: OCP implementation block diagram with domain interface (DI) and back-end (BE) modules.

The initial clocked OCP master, OCP slave, and domain interface designs were converted into fully asynchronous designs. This resulted in an asynchronous implementation similar to the synchronous designs. The conversion of the synchronous OCP master and OCP slave to an asynchronous circuit required finding the point of interaction between the request and the response channel from the synchronous implementation. Only the OCP master interacts and stalls the datapath based on the response channel. This stall is performed only during the read command and during the modes without out-of-order response like the basic (normal) or the burst mode. In the synchronous circuit, the absence of any data validity information leads to implementation of the stall mechanism in the datapath. For the asynchronous design, the data validity information is associated to the asynchronous handshake network. Hence, the asynchronous control network needs to be stalled instead of the datapath.

The asynchronous circuit consists of two separately developed circuits, the first one mimics the state machine of the synchronous OCP implementation, and the other one stalls the asynchronous control network. Fig. 7.12 shows the implementation of the control network of the asynchronous OCP master state machine, which is designed by direct conversion of the synchronous state machine. The state register flip-flops are divided into two latches and each latch is individually controlled by a *pipeline controller* (LC). On reset, the $LC1$ controller is initialized in a state indicating a token it wants to send to $LC0$, while $LC0$ controller is in the receiving state. The change of the state is dependent on any incoming request (lr1) from its predecessor block. Hence, if a request arrives, the join block results in a valid token to be sent to $LC0$ thereby initiating the asynchronous handshake. The data as well as the state are latched by $LC0$ and a request is sent to the upstream module as well as to $LC1$. The same implementation is also used for the state machine in OCP slave.

Fig. 7.13 shows the realization of the asynchronous OCP master module with the stalling and steering logic for the asynchronous handshake network, which is implemented using the select fork and join block. The select fork module does not steer the data; it only steers the request and the acknowledge handshake signals of the asynchronous control network. The data word from the output of $LC0$ is directly sent to the upstream module. For the control network, the select fork directs the request on both the channels, i.e., rr1 and the FIFO input channel, or only the upstream channel, i.e., rr1, based on the select input. The select input is generated by sampling the MCmd bits when rr1 for the state machine design block switches

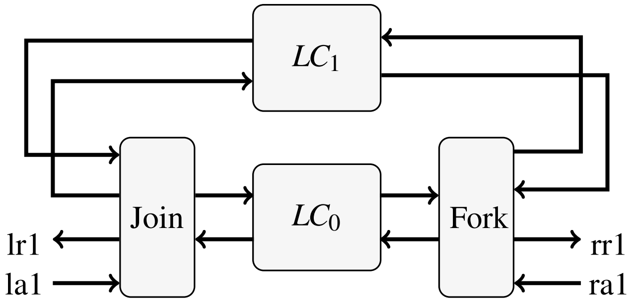

Figure 7.12: State machine control network.

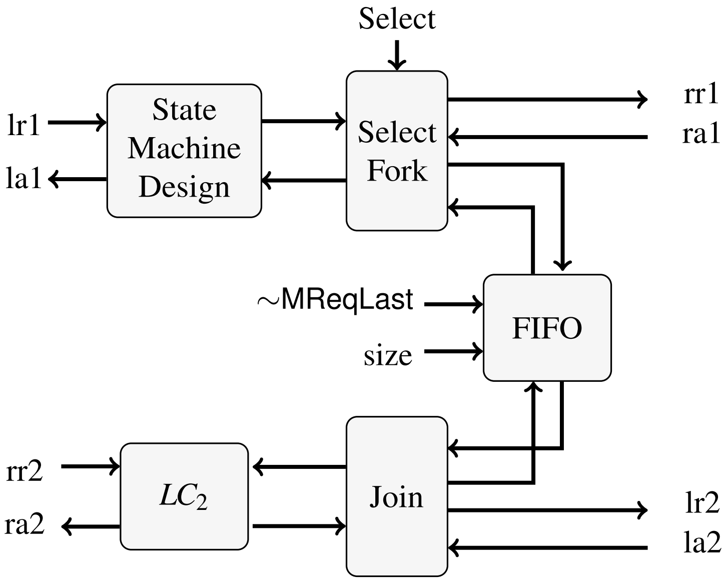

Figure 7.13: OCP master control network with steering and stalling block.

from logic level 0 to 1, thereby indicating that the OCP master has latched the incoming data. Both the output channels for the select fork are enabled only for the read command, while only the upstream channel is enabled for the write command, since no responses are generated for the write command for this OCP implementation. If the mode of operation is burst, only a FIFO is included between the select fork and join. For the normal mode of operation, the FIFO is replaced by just a wire connection since only one read operation can occur at any given moment. Hence, with the same implementation, both the modes of operation, where the request and response channels interact, are taken care of. The use of join enables the stalling of the request channel until a request signal (lr2) for a valid response arrives.

The implementation for the FIFO has the following requirements. Firstly, it should be of dynamic size and the size required is set based on the MBurstLength signal, which indicates the size of the burst operation. Secondly, the FIFO needs to stall when MReqLast signal is generated, thus indicating the last data token of the burst. Thirdly, it should support both precise and imprecise burst. In the precise mode, the number of data tokens to be transferred is known at the start of the burst transaction, while in imprecise mode, the end of the burst transaction is indicated by a token with MBurstLength signal set to 1. Hence, an input for the size of the burst and the MReqLast signal are required, as shown in Fig. 7.13. This circuit fails if the dynamic size of the burst is not stored and updated while transferring any read burst, since the asynchronous handshake network stalling is dependent on it. The stalling of the request channel is enabled by preventing the last acknowledge to reach the select fork. This acknowledgment signal is only generated when the last response for that burst has been received by the response channel. The dependency on the datapath for selecting the output channel of the Select Fork, and on the operation of the FIFO results in a min delay constraint on the request signal coming into the Select Fork module as well as on the module that samples the MCmd data bits. This request signal needs to be slow enough for the MCmd bits to be resolved before it is sampled at the select fork. This bottleneck affects the performance of the design making the handshake between OCP master and OCP slave modules the critical stage, thus resulting in power and performance penalty. Better protocols with data steering logic templates need to be developed to reduce this min delay overhead. One of the possible solutions would be to use the late data validity protocols described in Chapter 6, but it is left for future work.

These independent clocked and asynchronous blocks resulted in various configurations which can be evaluated for power, performance, area, and flexibility of choosing the IPs as well as the NoC. This enables using either a synchronous or asynchronous component at any side of the DI and hence opening up many architectural options.

There are five different implementations which are investigated in this work. First is a purely synchronous design with a single clock frequency controlling the operation of the IPs and the NoCs. Second is also a purely synchronous design but with IPs and NoCs operating at different frequencies. The third design is an unclocked asynchronous design. The fourth design has IPs which are synchronous, while the OCP implementation and the NoCs are unclocked. The last design has a synchronous OCP implementation with the NoC operating at a fixed frequency combined with asynchronous IP blocks.

7.4.4 Domain Interface Designs

Different domain interface designs are required to implement corresponding implementations. Five different domain interface designs corresponding to five different implementations are presented in this section. The DI for the purely synchronous and asynchronous design is just combinational. If needed, FIFO buffering can be added at the DI boundaries in these cases. For the synchronous design, the DI steers the data forward without storing it. In case of the asynchronous design, the DI also needs to forward the request and acknowledge the signal with the datapath signals. The complexity of the DI increases for the three other cases where different timing domains interact.

The domain interface, when CDC occurs, is defined in terms of an OCP request and response channel (Fig. 7.10). Hence, for every DI implementation, there are two places where CDC can occur, one for the request channel and the other one for the response channel. We use *sync* for synchronous or clocked interface and *async* for the asynchronous or unclocked interface. Thus, synchronization uses the naming convention of a timing domain pair to identify the two synchronized domains.

There are three different cases for the DI design where different timing domains interact. They are the *sync-sync*, *async-sync* and *sync-async* domain interfaces. The *sync-async* DI requires a *sync-async* request channel synchronization and an *async-sync* response channel synchronization. For a *async-sync* DI, it is the other way round, with the request channel having a *async-sync* synchronization and the response channel performing the *sync-async* synchronization.

Timing domain crossings, which occur in the DI are now confined to three different designs: the *sync-sync*, *sync-async* and *async-sync* DI interfaces.

7.4.4.1 The *sync-sync* Domain Interface FIFO

The core of this interface is a head and tail pointer FIFO that interfaces between two different clock domains. Fig. 7.14 shows an implementation of this design, which is similar to other work [78, 79]. The benefit of this design is an easy synchronization and generation

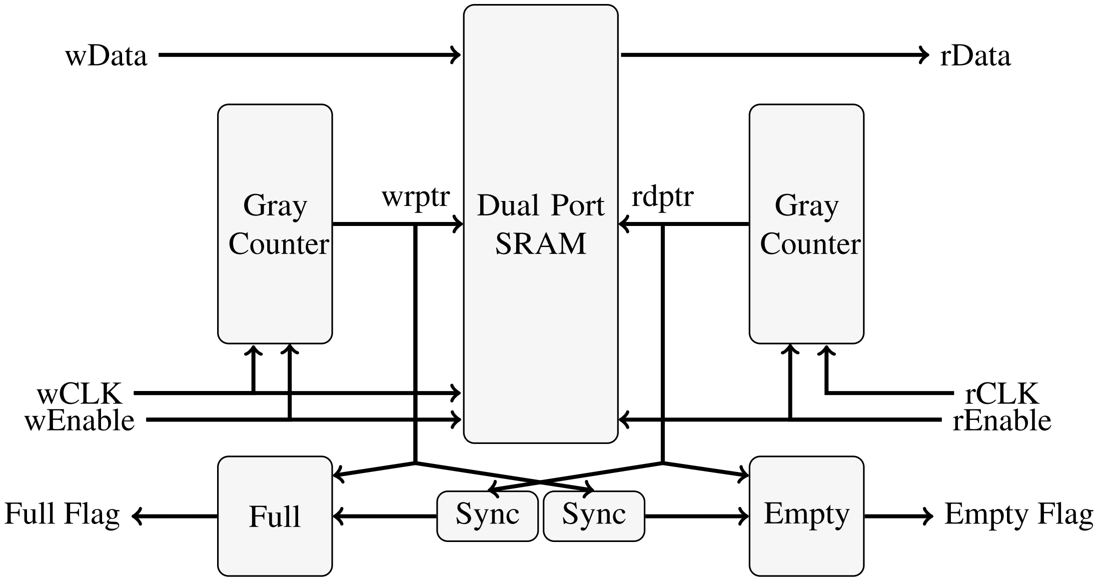

Figure 7.14: *sync-sync* domain interface FIFO.

of the full and empty statuses before the arrival of the next rising edge of its domain clock signal. Latency through this FIFO is dependent on the synchronizers, which sample the empty and full status based on the read (rdptr) and write (wrptr) pointers. Reading and writing the FIFO can be done at each clock edge of their respective domain until the Empty or Full flag is set. A two flop synchronizer is used (the sync block in Fig. 7.14); therefore, it takes approximately two clock cycles after the next write or read to update the Empty or Full flag status, respectively.

The domain interfaces specify Stall and Valid signals in order to indicate data validity (Stall and Valid signals are not shown but can be directly derived). Their behavior is similar to elastic systems [80]. The Stall signal on the write port is derived from the FIFO Full flag and the write Valid signal. Similarly, the Valid signal on the DI read port is derived from the FIFO Empty flag and the Stall signal on the read port. The wEnable signal is used to control the writing in the FIFO and the incrementing of the gray counter. It is derived from the write port Valid and stall signals. The rEnable signal is similarly derived from the Empty flag and the read port Stall signal.

7.4.4.2 The *async-sync* Domain Interface FIFO

The FIFO for the *async-sync* FIFO is shown in Fig. 7.15. The read port in this design is identical to that of the *sync-sync* FIFO. The write domain interface Valid signal is directly

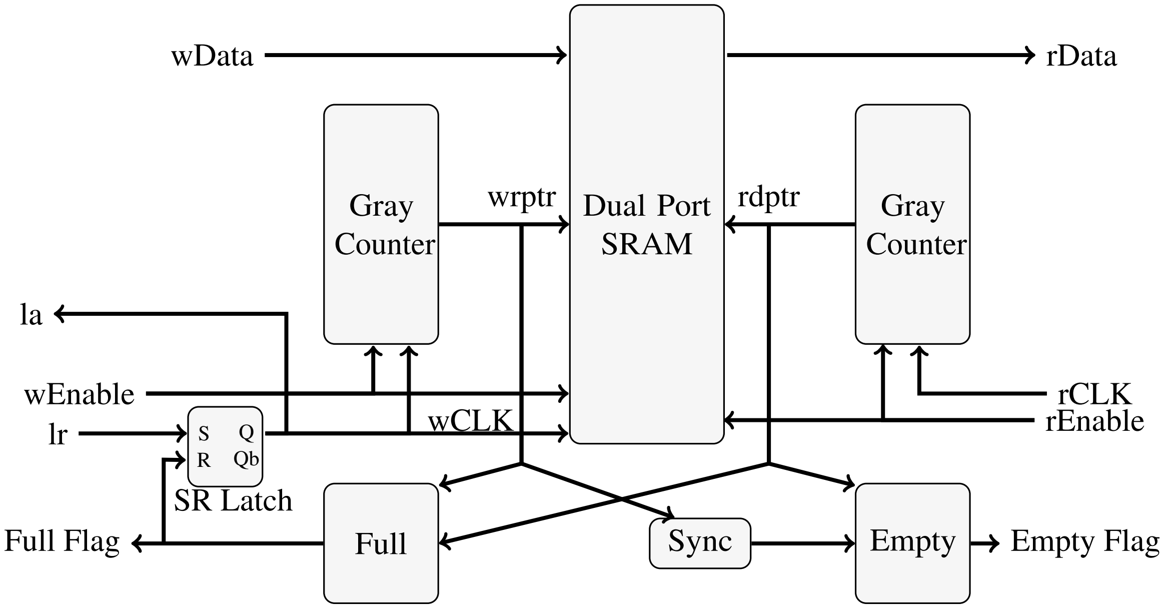

Figure 7.15: *async-sync* domain interface FIFO.

mapped to the asynchronous request signal (lr), and the Stall signal is directly mapped to the acknowledge (la) signal. (The Valid and Stall signals are not shown but can be directly derived.) The FIFO write port has been modified to generate the write clock wCLK from the write port Valid (lr) signal, and to directly generate the Stall (la) handshake signal.

The FIFO Full flag is causally generated from the assertion of the lr signal. A relative timing constraint is required to ensure proper operation. The Full flag update time is the sum of the delay to shift the gray counter, which updates the write pointer wrptr and propagates its value through the full flag logic.

No synchronization is required from the clocked to the asynchronous port. Write operations to the FIFO are deferred while the FIFO is full by blocking the assertion of the wCLK signal with the SR latch. If the FIFO is full and a pending data request exists, the lr signal does not propagate to the write clock signal wCLK so long as the full flag is asserted. As soon as the data word is read from a memory slot, the read pointer rdptr updates its value. This results in the Full Flag becoming unasserted. The SR latch is then released, allowing the lr signal to generate the write clock signal wCLK and acknowledge a data write on la.

7.4.4.3 The *sync-async* Domain Interface FIFO

The *sync-async* FIFO is similar to the *asyc-sync* FIFO as shown in Fig. 7.16. The design uses a SR latch on the read port to generate the domain interface Valid signal. There is no synchronization that is needed to generate the read port Valid signal. Thus, asynchronous implementation has half of the synchronization delay overhead compared to clocked designs.

7.4.5 Results

Fig. 7.11 shows the top level implementation for the test structure used to test the various designs explained in this chapter. The IP cores are abstracted out from the test setup. The test simulation directly drives signals into the IP back-end as though an IP were designed and directly connected. The NoC used in the evaluation performed here is simply a point-to-point connection. It is implemented as a wired connection between NW_BE1 and NW_BE2. This is done to provide better area and power comparisons of the OCP blocks themselves by not adding in extra IP logic. All blocks are placed in close proximity for this evaluation. The IPs were considered to be on the same domain, i.e., either clocked at the same frequency, or unclocked asynchronous IP. For all the designs with multiple domains, the leftmost and rightmost DI in Fig. 7.11 were the domain crossing blocks.

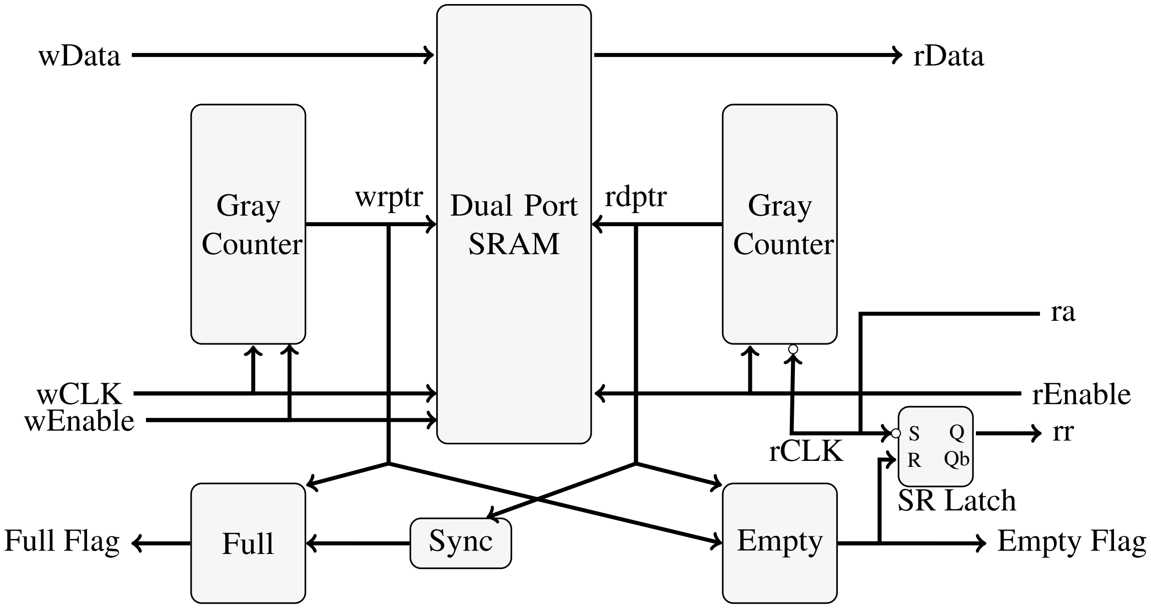

Figure 7.16: *sync-async* domain interface FIFO.

There are five different cases implemented, which include: a clocked system with a single clock domain (*sync*), clocked circuit with a different clock frequency used for the NoC (*sync-sync-sync*), asynchronous unclocked circuit (*async*), and mixed clocked and asynchronous designs *async-sync-async* and *sync-async-sync* (a GALS architecture). For each individual case, the operation of the design is validated for four OCP design modes: normal read/write, burst read/write, read/write with out-of-order response and burst read/write with out-of-order response.

The reported results use the Artisan RVT (regular threshold) library for the IBM 65nm 10sf process using full layout and parasitic extraction. The toolflow used is the same as the one described in Chapter 3 with the only difference being the presence of both a clock signal and the asynchronous constraints related to the asynchronous designs. DC is used for synthesis, Modelsim is used for simulation, and SoC Encounter is used for place, route, and parasitic extraction. The power and delay numbers used *standard delay format* (SDF) parasitic back annotation into the Modelsim. The power numbers were generated using parasitic extraction and activity factors from a simulation run by importing a *value change dump* (VCD) file from Modelsim into SoC Encounter. The simulation runs a set of read and write commands to validate the functioning of the design. Postlayout timing is validated using PrimeTime with extracted parasitics.

The operating frequency of all the designs with a single clock domain, i.e., *sync*, *async-sync-async* and *sync-async-sync* is 667MHz. For the multiply clock domain design, i.e., *sync-sync-sync*, the operating frequency of the OCP and NoC domain is 570MHz, while each IP back-end operated at 667MHz. For the asynchronous blocks, the constraints for the handshake controllers were specified based on the amount of logic in each pipeline stage as described in [41]. The simulation testbench performs 22 transactions consisting of 12 writes and 10 reads.

Tables 7.8 and 7.9 show the results for designs with and without CDC in the domain interfaces. Performance is based on the simulation time for the testbench. Energy numbers represent the average energy consumed per transaction.

Designs which did not include extended capability, did not have those features compiled into the logic. It can be seen that the addition of extra logic for the burst, tag or both burst and tag, results in an increase in the area and power consumed by each design as against the

Table 7.8: Energy, performance and area comparison for design with no domain crossing.

	Area	Simulation	Energy/trans.	Area	Performance	Energy
	(um^2)	Time (ns)	(pJ)	Benefit	Benefit	Benefit
sync Design						
Normal	15,822.0	309.75	94.43	1.00×	1.00×	1.00×
Burst	17,422.8	210.75	74.84	1.00×	1.00×	1.00×
Normal + Tag	16,432.8	144.75	52.40	1.00×	1.00×	1.00×
Burst + Tag	18,337.8	144.75	58.06	1.00×	1.00×	1.00×
async Design						
Normal	11,572.2	100.65	10.63	1.37×	3.08×	8.88×
Burst	14,218.8	81.36	13.09	1.23×	2.59×	5.72×
Normal + Tag	11,955.0	53.56	11.04	1.37×	2.70×	4.74×
Burst + Tag	13,518.0	52.62	12.49	1.36×	2.75×	4.65×

base design without any extensions. Respective areas and power in Table 7.9 are over 3× larger than those in Table 7.8. The overhead of domain crossing is also increased due to the addition of extra buffering using FIFO structures. These DI FIFOs are all eight words deep.

Table 7.8 allows us to compare purely clocked and asynchronous designs. Designs that only use asynchronous components, are substantially better than those which use only clocked components in terms of area, performance and power. Results in the table are all relative to the purely clocked design. The pure asynchronous design shows up to a 8.9× improvement in power, 3.1× improvement in performance, and 1.4× improvement in area over the purely clocked design.

Table 7.9 compares designs with clock domain crossings. The bulk of the OCP logic lies in the network domain, since the CDC boundaries are in the far left and right domain interfaces. Therefore, these result are largely dominated by the network clocking methodology. The GALS design, the *sync-async-sync* design, is far superior to the multisynchronous clocked design and the LAGS (*async-sync-async* – locally asynchronous globally synchronous) design. One of the primary benefits of asynchronous design is that it does not require synchronization when signals move into an asynchronous domain. Also, since

Table 7.9: Energy, performance and area comparison for design with domain crossing.

	Area	Simulation	Energy/trans.	Area	Performance	Energy
	(um^2)	Time (ns)	(pJ)	Benefit	Benefit	Benefit
sync-sync-sync Design						
Normal	59,419.8	401.25	414.02	1.00×	1.00×	1.00×
Burst	63,574.8	242.25	275.28	1.00×	1.00×	1.00×
Normal + Tag	61,737.6	138.75	160.19	1.00×	1.00×	1.00×
Burst + Tag	66,683.4	138.75	182.27	1.00×	1.00×	1.00×
sync-async-sync Design						
Normal	50,866.2	102.75	75.19	1.17×	3.91×	5.51×
Burst	55,956.0	98.25	79.05	1.14×	2.47×	3.48×
Normal + Tag	53,438.4	78.75	71.59	1.16×	1.76×	2.24×
Burst + Tag	57,557.4	78.75	81.97	1.16×	1.76×	2.22×
async-sync-async Design						
Normal	51,586.8	332.70	226.84	1.15×	1.21×	1.83×
Burst	55,584.0	235.58	186.32	1.14×	1.03×	1.48×
Normal + Tag	54,436.8	178.58	144.49	1.13×	0.78×	1.11×
Burst + Tag	58,363.8	178.59	155.96	1.14×	0.78×	1.17×

the pipeline implementation is elastic and stalls can be easily handled and resolved, the asynchronous design works better than the synchronous design where elasticity is added. There is a large overhead in terms of area, energy and performance for adding elasticity in synchronous circuits. This results in designs with asynchronous domains, such as the GALS or LAGS designs, providing better results when compared to synchronous multifrequency designs.

The performance penalty for the synchronous system presented in this work is due to the necessity to synchronize when moving into a new clock domain. However, it is also partly due to the implementation of elasticity and its stall mechanism. Hence, analysis of better stall protocols might make the results better for designs that are partly or fully synchronous, but that is left for future work.

7.5 Asynchronous 64-point FFT Example [7]

Application of the multifrequency approach on large designs for validation of the power and performance benefit comparisons with respect to an equivalent synchronous design needs to be performed. An FFT architecture with concurrent data stream computation is selected. Asynchronous and synchronous implementations for a 16-point and a 64-point FFT circuit were designed and compared for energy, performance and area. Both versions are structurally similar. The asynchronous circuit is developed using the multifrequency CAD tools and flows (Chapter 3), which is similar to any synchronous CAD tools and flows. The asynchronous design shows a benefit of 2.4×, 2.4× and 3.2× in terms of area, energy and performance, respectively, over its synchronous counterpart. The circuit is further compared with other published designs and shows 0.4×, 4.8× and 32.4× benefit with respect to area, energy and performance.

7.5.1 FFT Architecture

The FFT is an algorithm that requires global dependencies, but it can be derived in a multirate form that allows a hierarchical representation as shown in Eqn. 7.1 [82]. This multirate architecture exploits performance from concurrency by allowing parallel computations to occur at reduced frequencies. The equation represents N_2 FFTs using N_1 values as the inner summation, which are scaled and then used to produce N_1 FFTs of N_2 values. This representation has the advantage that it takes a high frequency stream and decimates it so that each of the internal FFTs operate at a lower decimated data stream frequency. It allows the architecture to simultaneously have lower energy and higher performance.

$$X_{m_1}(m_2) = \sum_{n_2=0}^{N_2-1} \left[W_N^{m_1 n_2} \sum_{n_1=0}^{N_1-1} x_{n_2}(n_1) W_{N_1}^{m_1 n_1} \right] W_{N_2}^{m_2 n_2} \tag{7.1}$$

[7] This section has been published in DATE, 2013 [81]. © 2013 IEEE. Reprinted with permission, from William Lee, Vikas S. Vij, Anthony R. Thatcher, Kenneth S. Stevens, “Design of Low Energy, High Performance Synchronous and Asynchronous 64-Point FFT,” in Proceedings of the Conference on Design, Automation and Test in Europe, Mar., 2013.

The general architecture derived from Eqn. 7.1 is shown in Fig. 7.17. There are three architectural control structures: a decimator, expander, and crossbar block. Each of the N_i blocks can be a hierarchical instance of the design where i is the size of the FFT performed in that block. The values of $N_1 \times N_2$ equals N_1 or N_2 at the higher level in the hierarchy.

The decimator block down-samples the input stream [83]. For a sampled signal $x(n)$, the output of the M-fold decimator is given by $y(Mn)$. The sampling of the N_2 decimator is arranged in a regular repeating fashion where the first sample is steered to the first output stream, the second to the second stream, and so on. The M^{th} sample is steered back to the first stream. This effectively produces M parallel streams operating at $1/M$ the frequency of the input.

The expander block is the dual of the decimator block. They take M low-frequency streams and up-sample by combining them into a stream that has an M-fold higher frequency. In the FFT architecture, the expander operates on a stream of data $x_0(m_2),\ldots,x_{N-1}(m_2)$ reproducing a stream at the original frequency and in the correct functional order for the algorithm.

Product blocks multiply a stream of results coming from the N_1 point FFT units by a set of constant values. Both constants and results are complex numbers, requiring four multiplications and two additions per sample. The constants are calculated by $W_N^{m_1 n_2}$, where $m_1 = 0,\ldots,N_1 - 1$ and $n_2 = 0,\ldots,N_2 - 1$.

The crossbar switch maps results from the product block to the N_2 FFT units. The N_2 FFT units take a transform of time displaced Fourier transform samples. Each N_1-point FFT provides one data sample to each of the N_2-point FFT units. The first row of the N_2 FFT units takes the first sample from each of the N_1 rows, the second row similarly takes the second sample, and so on. This is implemented by performing an N_2 up-sampling followed by a N_1 down-sampling. Another solution is to steer the data to N_2 N_1-way decimators, followed by N_1 N_2-way expanders. Decimator sequencing here is different than that of the top level block because it steers the first N_2 samples to each row before moving onto the next row.

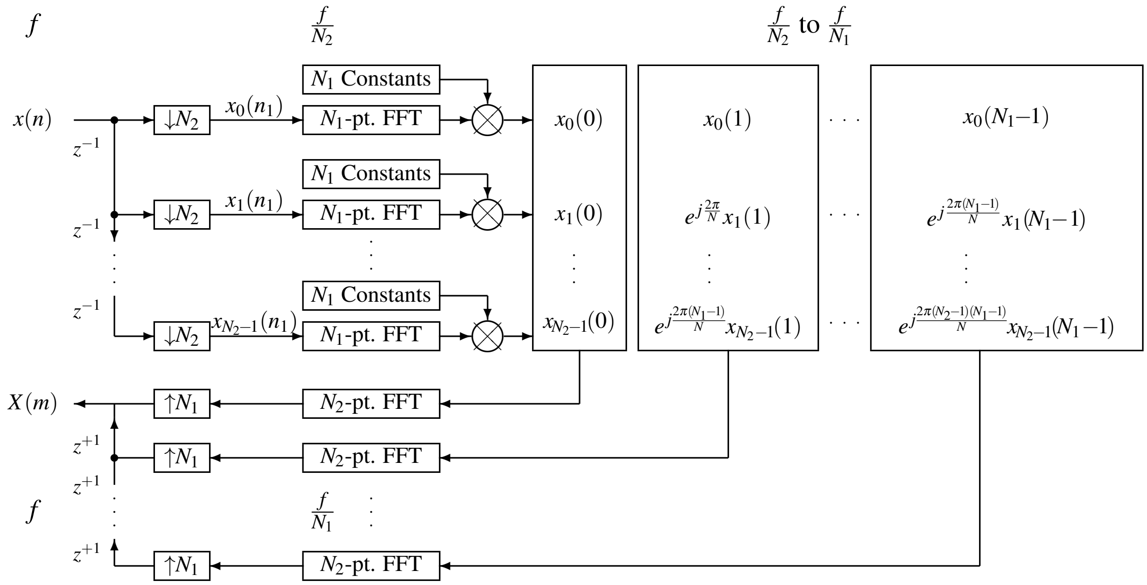

Figure 7.17: Multirate FFT architecture.

7.5.2 FFT Design

Multifrequency asynchronous and clocked 64-point FFT designs are implemented from the architecture block diagram shown in Fig. 7.17. Both designs are hierarchically decomposed at the top level such that $N_1 = 16$ and $N_2 = 4$. The 16-point FFT implementations are also hierarchically decomposed with $N_1 = N_2 = 4$. The terminal hierarchical nodes in the design are the 4-point FFT block since it can be implemented with simple add and subtract operations due to the value of the constant data values. There are four frequency domains in this design. The frequency of the incoming data is f, which gets decimated to derive $f/4$, $f/16$ and $f/64$ frequencies.

The datapath for all the designs are specified behaviorally with the control being the only differentiating point. The asynchronous design is implemented as a bundled data pipeline (Fig. 2.3). The LC block, that controls timing and sequencing, is a 4-phase handshake protocol similar to that in Fig. 3.4. This cell generates a local clock signal to control the pipeline stage based on the handshake with the adjacent handshake controllers.

These designs operate on fixed-point data. The input and output are 32 bits wide, with the upper 16 bits representing the real value and the lower 16 representing the imaginary value. The fields use two's complement representation of signed numbers that are decimal values less than or equal to plus or minus 1. The first four bits are used for the whole part of the number and the rest 12 bits for the fractional part.

7.5.2.1 Asynchronous Design

The first step in an RT asynchronous design is to create and characterize the handshake elements. This design uses four circuit elements: a linear pipeline controller (LC) (Fig. 3.4), a 2-input fork/join element, the decimator and expander. The LC circuit interfaces two pipeline stages by controlling the protocol between the stages and storing one data word (Fig. 2.3). The fork (Fig. 7.18) broadcasts a request from a sender to two receivers. The ack from the two receivers is synchronized with a C-element before being passed on to the sender [84]. The join element contains the same logic and is the dual of the fork. Requests from two senders are first synchronized before being sent to a receiver, while the ack signal from the receiver is broadcasted to both the senders.

Fig. 7.19 and Fig. 7.20 show the design of the four-way synchronous and asynchronous decimator, respectively. The asynchronous decimator consists of a ring connected shift register with one bit asserted to steer the requests to four different pipelines based on the value in the shift register. The req and the ack signals are active high. Since only one acknowledgment is active at a time, the four ack signals are passed through an OR gate. Values in the shift register get updated when the input request goes low. The circuit is characterized for its timing constraints. As long as the shift register can change in one half cycle time (before the next req occurs), this logic operates correctly. Note that this block adds a 2-input AND gate delay on the request path and a 4-input OR gate delay on the acknowledge path. This is the only overhead of the decimator, and it adds approximately 8 gate delays to the cycle time of the architecture, allowing it to operate at approximately a 16 gate delay cycle time. This resulted in a frequency that is close to 1.3 GHz, which is a sufficiently fast performance target.

The design of the synchronous and asynchronous expander in Fig. 7.21 and Fig. 7.22 is similar to the decimator. The asynchronous expander includes an N_i-bit ring connected shift register and some combinational gates to select the data and control signals to be driven to the output channel.

Once these blocks were designed, the top level asynchronous architecture is built by simply composing the pipeline control and datapaths together. A hierarchical structural design style is employed, which is almost identical to drawing and connecting block level schematics for the design. In this method a functionally correct design is hierarchically

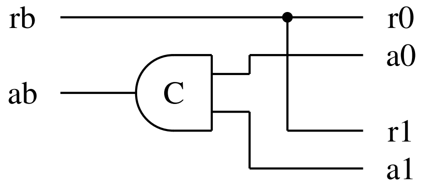

Figure 7.18: Fork/Join template.

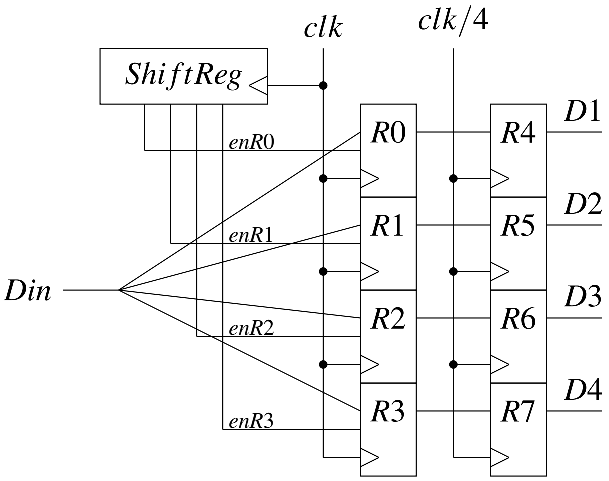

Figure 7.19: Synchronous decimator.

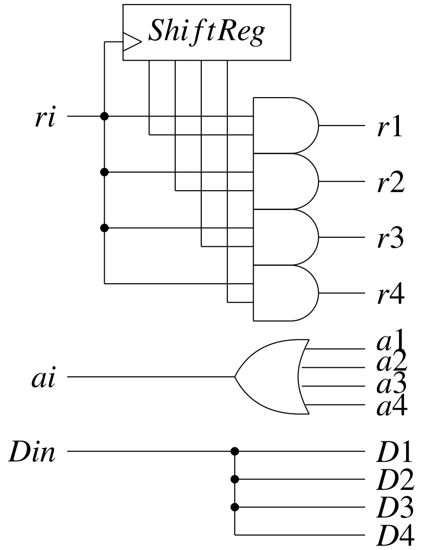

Figure 7.20: Asynchronous decimator.

designed and validated for performance and correctness. First, a simple 4-point FFT is built, which is used to build a 16-point FFT, and then these components were integrated into the 64-point FFT. The dataflow graph of a 4-point FFT is shown in Fig. 7.23. The pipelined asynchronous control logic for that design is shown in Fig. 7.24. The design of the pipeline is almost as simple as drawing a schematic on a paper, where the butterfly network and first set of adders are between stages LC1 and LC2, the second butterfly network and adders are between stages LC2 and LC3, and the last network convolution is between stages LC3 and LC4. Fig. 7.25 shows a code snippet from the design to give a flavor of the RTL. Some liberty is taken in the syntax to compress the example. This shows a pipeline stage at the input of the design that feeds into the next stage of 16-point FFTs. Each pipeline stage and structural block are similarly designed.

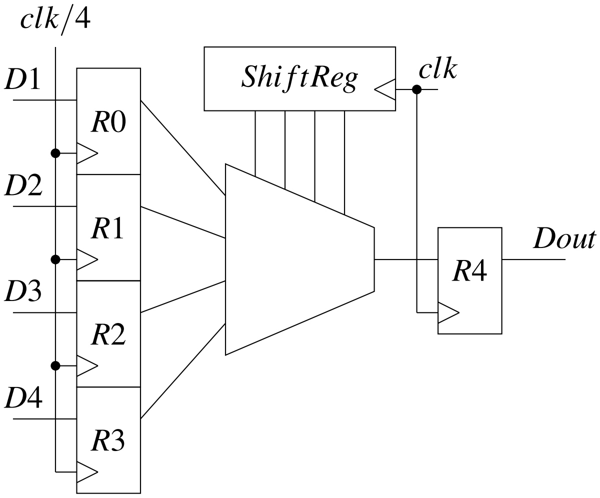

Figure 7.21: Synchronous expander.

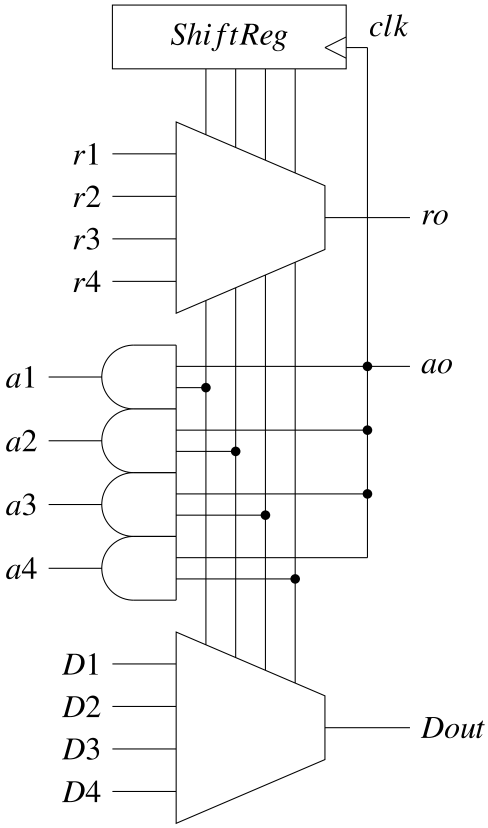

Figure 7.22: Asynchronous expander.

Performance and functionality optimizations in an asynchronous design are somewhat independent operations. A design that is functionally correct can be created relatively quickly. However, some effort is needed to balance the cycle times and pipelining to optimize performance, particularly for multifrequency designs. This is very different from clocked design, where performance and pipelining are essential for correct functionality, and part of the initial specification.

A primary aspect of optimizing performance of an asynchronous architecture is to calculate the critical paths and focus on those. Experimenting with the power-performance tradeoffs allowed us to quickly identify the critical paths in the asynchronous design. Due to the multirate architecture, it is not the complex multipliers or adders that operate at 1/4, 1/16, or 1/64 of the input frequency. Rather, the top level decimators and expanders limit the operating frequency of the design. We, therefore, focused on designing high throughput decimators and expanders.

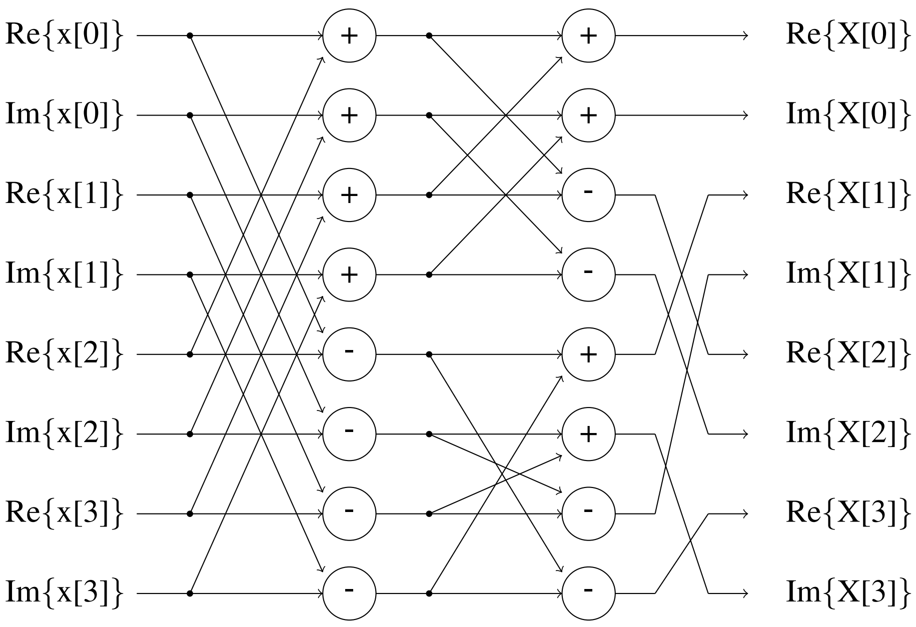

Figure 7.23: Data flow graph of 4-point FFT calculation.

The performance optimizations are illustrated with the 4-point FFT pipeline, shown in Fig. 7.23 and 7.24. From a correctness perspective, the data through the expander could pass straight through the expander through the butterfly network to the adders. However, this would create too long a cycle time at the decimators. Increased performance is obtained by adding pipeline stages before and after the decimators and expanders since they are the critical paths in the design.

The next power-performance optimization of the asynchronous 4-point FFT design is to determine the frequency target for the smallest area and lowest power adder in the given technology. A 16-bit ripple carry adder needed about 860ps in this technology, so that became the performance target of the 4-point FFT design. This is less than the time available for the computation (3ns in the top level 64-point block and 12ns in the 16-point blocks at 1.3 GHz operating frequency). However, slowing the operation down beyond 860ps simply adds more area, energy, and latency to the control path.

An additional performance critical aspect of a design is due to pipeline synchronizations. Adding or removing pipeline stages in an asynchronous design can be employed to remove forward and backward stalls in an architecture. This has been referred to as “slack matching” in the asynchronous literature [66]. Therefore, a version of the performance critical four-way

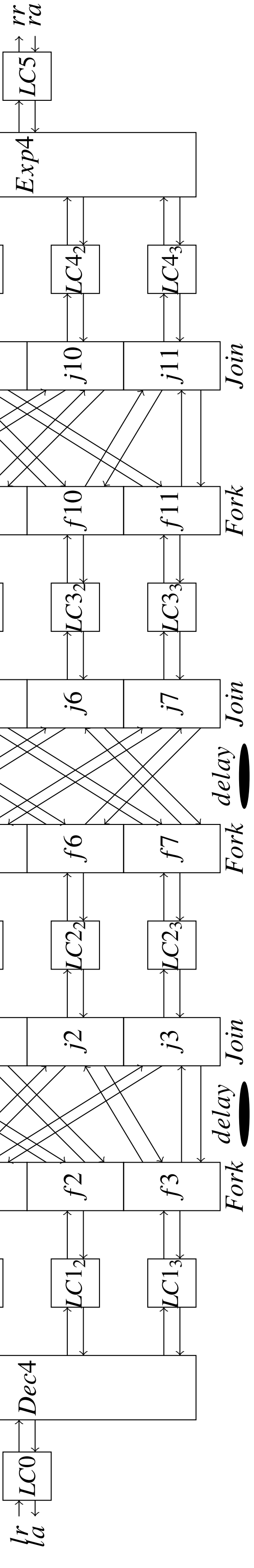

Figure 7.24: 4-point FFT design.

```
module FFT_64 (ri, ai, DI, ro, ao, DO, rst);
   input [‘WORD_SIZE-1:0] DI; ...

   // input pipeline
   linear_control  LC0     (.lr(ri), .la(ai), .rr(p0r), .ra(p0a), .ck(ck0), .rst(rst));
   latch           P0      (.d(DI), .clk(ck0), .q(P0D0));

   // 1-to-4 Decimator
   decimator_4     D4_0    (.DI(P0D0), .D1(P0DT1), .D2(P0DT2), .D3(P0DT3), .D4(P0DT4),
                           .ri(p0r), .ai(p0a), .rst(rst), .r1(p0rt1), .r2(p0rt2), .r3(p0rt3), .r4(p0rt4),
                           .a1(p0at1), .a2(p0at2), .a3(p0at3), .a4(p0at4));

   // The FFT_16 modules.
   FFT_16          F16_0   (.ri(p0rt1), .ai(p0at1), .ro(p1rt1), .ao(p1at1),
                           .DI(P0DT1), .DO(P1DT1), .rst(rst));
   FFT_16          F16_1   (.ri(p0rt2), .ai(p0at2), .ro(p1rt2), .ao(p1at2),
                           .DI(P0DT2), .DO(P1DT2), .rst(rst));
   FFT_16          F16_2   (.ri(p0rt3), .ai(p0at3), .ro(p1rt3), .ao(p1at3),
                           .DI(P0DT3), .DO(P1DT3), .rst(rst));
   FFT_16          F16_3   (.ri(p0rt4), .ai(p0at4), .ro(p1rt4), .ao(p1at4),
                           .DI(P0DT4), .DO(P1DT4), .rst(rst));

   linear_control  LC2_0   (.lr_(p1rt1), .la_(p1at1), .rr_(p2rt1), .ra_(p2at1), .ck(ck1_0), .rst(rst));
   latch           P2_0    (.d(P1DT1), .clk(ck1_0), .q(P2DT1));

   // Constant Block and Complex Multiplier.
   CB64_1          CB_1    (.update(p1at2), .DO(CDT2), .en(endt2), .rst(rst));
   comp_mult       CM_1    (.A(P1DT2), .B(CDT2), .P(CP2), .en(endt2));
```

Figure 7.25: Verilog code snippet for FFT_64.

decimator is designed as a 2×2 pipelined decimator to increase throughput and reduce sensitivity to backward stalls. Likewise, the crossbar and 16-way expander in the 64-point design of Fig. 7.26 have been pipelined.

The asynchronous design is built using “natural” pipelining for each block with pipeline performance targets based on the top level architecture. For example, pipeline stages exist between the adders of the design, whereas they can be removed from a performance perspective. A few modifications to the original pipeline structure have been made to improve area in the “async-opt” design.

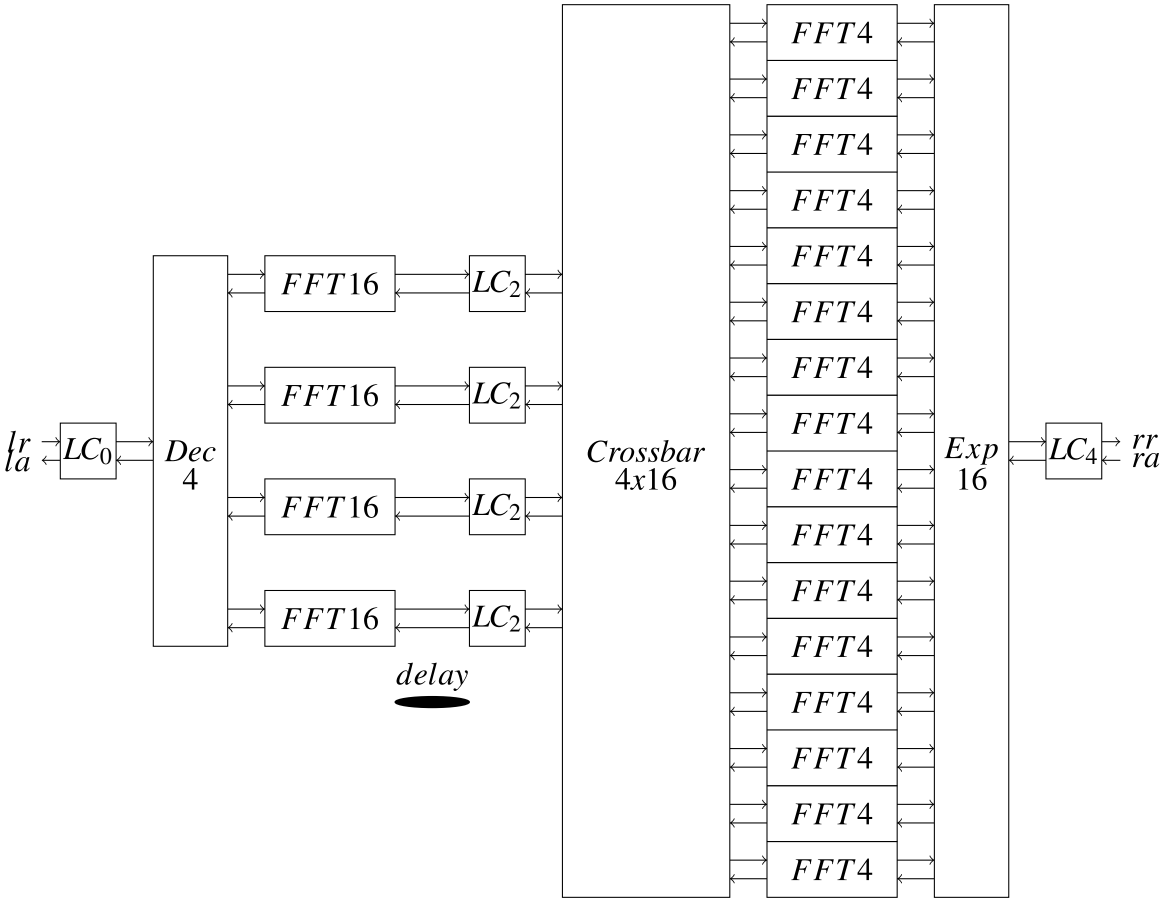

Figure 7.26: 64-point FFT design.

7.5.2.2 Synchronous Design

The synchronous FFT used for comparison has the same architecture as shown in Fig. 7.17. It consists of clocked decimator and expander (Fig. 7.19 and 7.21). The 4-point synchronous FFT design is a 6-deep pipeline.

Fig. 7.19 shows the clocked four-way decimator. The design consists of a high frequency register bank and a low frequency register bank, a clock divider, and a shift register to track the relationship between the two clocks. The shift register must be properly initialized in relation to the global state of the circuit based on data arrival to ensure proper data steering. The data word is incrementally latched into the high frequency register bank. At the low frequency clock, the data word is then shifted into the low frequency register bank, where it is sampled at a $1/N_2$ frequency. The expander in Fig. 7.21 is the dual of the decimator.

The parallel data stored into a low frequency register are streamed and stored in the output register based on the higher frequency clock. The channel selection is dependent on the shift register and requires proper initialization similar to the decimator.

7.5.3 Results

These circuits use the Artisan academic library in IBM's 65nm 10sf process. The circuits were designed in behavioral Verilog, synthesized using DC, and place and routed using SoC Encounter. Circuits were simulated for timing and functional correctness using Modelsim with postlayout parasitics backannotated. Testing is performed using predefined input vectors which included 1024 random numbers. Both 64-point FFT circuits have less than ±0.3 percent variation as compared to MATLAB FFT computation. Various performance parameters including forward latency, cycle time, and throughput were also generated from the simulation along with VCD file. The simulation VCD file along with the parasitics of the place and routed design is used to calculate the power numbers for each design by PrimeTime PX.

Tables 7.10 and 7.11 summarize these multirate designs against several other designs. These 16-point implementations are compared against a design that is similar in architecture [85]. The 64-point benchmark is a low power Texas Instruments design [48]. Performance is measured as the time to completely process 1024 samples.

The simplicity of making architectural and performance modifications to the asynchronous design allowed us to quickly explore a simple area improvement to our asynchronous architecture. The 64-point architecture contains four 16-point FFTs. Each of these contain three complex multipliers operating at 1/16 the top level frequency. At this frequency, the multipliers could be shared, removing eight complex multipliers from the design (Async-opt), resulting in an overall 18 percent area reduction. This modification has a minor positive affect on performance and negative affect on latency and energy per point. Other modifications that reduce area at little or no energy and performance cost can also be explored along with other optimizations based on target versus required frequencies. Asynchronous designs are particularly amenable to such architectural explorations.

Table 7.10: The 16-point FFT comparison result (* constant field scaled to 65 nm technology).

Design	Tech.	Points –	Word	Clock	1K-point Exec.	Power	Energy/point	Area	Exec.Time	Energy	Area
	nm	Samples	*bits*	*MHz*	Time μs	*mW*	*pJ*	Kgates	Benefit	Benefit	Benefit
This Design (Async)	65	16-1024	16	1,274	0.83	30.9	25.05	54	8.32	3.93	2.73
This Design (clock)	65	16-1024	16	588	1.73	24.7	41.83	71	3.98	2.35	2.07
Guan [85]	130	16-1024	16	653*	6.91*	14.6*	98.33*	147	1.00	1.00	1.00

Table 7.11: The 64-point FFT comparison result (* constant field scaled to 65 nm technology, + nominal process voltage).

Design	Tech.	Points –	Word	Clock	1K-point Exec.	Power	Energy/point	Area	Exec.Time	Energy	Area
	nm	Samples	*bits*	*MHz*	Time μs	*mW*	*pJ*	μm^2	Benefit	Benefit	Benefit
This Design (Async-opt)	65	64-1024	16	1,357	0.87	69.4	59.23	395	32.24	4.51	0.47
This Design (Async)	65	64-1024	16	1,316	0.87	65.5	55.65	479	32.39	4.80	0.39
This Design (Clock)	65	64-1024	16	667	2.76	50.2	135.30	1,160	10.21	1.97	0.16
Baireddy [48]	90	64-4096	–	514*	28.18*	9.7*	266.95*	186*	1.00	1.00	1.00
Chong (1.1V) (Async) [86]	350	128-128	16	–	1,633.64*	–	4.45*	45*	0.02	59.98	4.12
Chong (3.5V) (Async) [86]	350	128-128	16	–	513.43*	–	45.06*	45*	0.05	5.92	4.12
Baas (3.3V)[87]	600	1024-1024	20	1,470*	3.53*	11.7*	40.31*	679*	7.98	6.62	0.27
Baas (5V)[87]	600	1024-1024	20	2,228*	2.33*	40.7*	92.55*	679*	12.10	2.88	0.27

For comparison, results in these tables are optimistically scaled to an equivalent for 65nm technology node by using theoretical constant-field scaling assuming the scaling factor $\kappa = 1.43$ per node (let $s = 1/\kappa = 0.7$) [88]. This results in delays in the tables multiplied by s, s^2, and s^6 for the 90nm, 130nm, and 600nm nodes, respectively. Energy values are scaled by s^3, s^6, and s^{18}. Area reduces by s^2 per generation.

The biggest advantage of this multifrequency architecture against the other architectures comes in the form of throughput. These designs can sustain a rate of 1 data point per clock cycle, at a relatively constant frequency regardless of the point size. The asynchronous design also provides a substantial reduction in latency. From an idle start, the asynchronous 16-point and 64-point designs can complete processing 1024 samples over 8× and 32× faster, respectively, than the benchmark designs. Multifrequency design also shines in energy per sample. The asynchronous designs consume approximately 1/4 the energy per sample as that of the competitors. This 16-point pipelined design is less than half the size of this comparable clocked hierarchical pipelined design. When comparing this design against the low power 64-point design from Texas Instruments, the clocked design and area optimized asynchronous designs consume 6× and 2× the area, respectively. This points out very different design targets and architecture styles. Their architecture shares the computation units for area efficiency at a cost of higher energy and much lower performance.

The Async-opt design is significantly better than the clocked design of the same architecture. The 64-point design shows an improvement of 2.28× the energy per data point and 3.16× the performance while costing only 1/3 the area.

Accurately comparing FFT designs with different point sizes, technology nodes, and architectures is challenging. Table 7.12, therefore, provides a design comparison based on three metrics - *Benefit product* [87] and $e\tau^2$ using Baireddy as the reference, and *Normalized FFTs per energy* [86]. Benefit product is the product of the area, energy, and execution time. Baas and Chong employ voltage scaling to quadratically reduce energy. Energy times square of the execution time ($e\tau^2$) provides a reference that is independent of voltage scaling. The normalized FFTs per energy metric largely disregards performance. Those results are normalized to the 350nm node to produce the same values as reported in [86].

Table 7.12: Design comparisons (+ The nominal process voltage).

Design	$e\tau^2$ Advantage	Normalized FFTs per Energy [86]	Benefit Prod. [87]
This Design (Async-opt)	4,683.38	17.35	68.54
This Design (Async)	5,031.00	18.47	60.37
This Design (Clock)	205.60	7.60	3.23
Baireddy [48]	1.00	–	1.00
Chong (Async-1.1V)	0.02	8.33	4.26
Chong (Async-3.5V)$^{+}$	0.02	17.01	1.34
Baas (3.3V) [87]	421.98	8.44	14.48
Baas (5V)$^{+}$[87]	421.98	3.31	9.56

7.5.4 Timing Closure Approaches for Asynchronous Circuits

A design consists of timing constraints that need to be met for functionality and correctness. In case of synchronous designs, these are represented as setup and hold time constraints. The optimization process that modifies the design so as to meet the constraint requirements is called timing closure. For the flow described in Chapter 3, the timing information of any design is represented as min and max delay constraints. Each correctness and functionality *relative timing* (RT) constraint gets mapped to a min and a max timing path. The timing path sometimes have to be divided into small path segments in the presence of overlap between multiple paths. These path segments are specified to the CAD tools as set_min_delay/set_max_delay timing constraints.

The validation of all the paths, such that no negative slack paths are present, becomes critical for proper functioning of any circuit. The absence of a reference clock in an asynchronous circuit, results in nonapplicability of the synchronous timing closure approaches because of the need for the constraints to be validated with respect to another path. For example, the RT constraint for a bundled data pipeline, as shown in Fig. 2.3, is $req_i\uparrow \mapsto L_{i+1}/d + setup \prec L_{i+1}/clk\uparrow$. It consists of a max path from req_i to L_{i+1}/d via L_i/clk and a competing min path from req_i to L_{i+1}/clk via req_{i+1} with a *setup* margin between them. The max path is specified in the tools by dividing it into two segments from req_i to L_i/clk and L_{i+1}/d to L_{i+1}/q, since *clk* to *d* path of any register is cut by the tools to avoid cycles. Similarly, the min path is divided into two segments req_i to req_{i+1} and req_{i+1} to L_{i+1}/clk.

The presence of these competing paths results in a different set of problems which are not seen for synchronous designs. Due to the ordering that must be maintained at the point of convergence of any RT constraint, any modification to the delay of one path needs to be matched with similar modifications on another competing path. This is explained by an example. If the max delay path for the datapath of a bundled data pipeline needs to be slowed down, then the corresponding min delay constraint on the control path also needs to be increased such that the setup margin between the two paths is maintained. Dividing the paths into small segments results in an extra step for complete path validation which needs to be performed at a later stage. The current validation framework addresses this by simulating the circuit using SDF backannotation.

Consider the design synthesis step with all the constraints specified for an asynchronous circuit. If the circuit meets timing, then no violating paths with negative slack are reported, else the violating timing paths reported. Consider any bundled data design with a control network comprised of handshake controllers and datapath: if violating paths are reported by the synthesis tool then achieving timing closure on these paths can be approached in two ways. The first approach relaxes the timing constraint on the violating paths based on the negative slack reported and it resynthesizes the circuit with the new relaxed constraints. A similar approach is taken for synthesizing any synchronous circuit too by increasing the clock cycle period to accommodate a slow path. For timing path driven synthesis with lots of min and max paths, increasing the max delays for the violating max paths requires an increase in the competing min paths delays. But, for designs with a lot of competing paths, this approach leads to an inferior circuit, and hence requires a different method if performance is critical. The second approach divides the timing closure problem into two steps. The first step involves only setting the datapath timing constraints and achieving timing closure. This enables the circuit designer to know the operating frequency of each pipeline stage in a circuit. In the second step, the constraints corresponding to the control path are also specified with the constraints of the first step. It is observed that DC stops optimizing the design if any one violating path cannot be resolved. This results in violations on the paths, which were earlier reported as having a positive slack. Hence, various paths from the datapath can be reported as having negative slack, and consequently, shown as violating paths. Such datapath violations are ignored and only the control path related violaters are modified, like in the first step, to achieve the timing closure.

Both of these approaches have advantages and disadvantages, which need to be analyzed to select the best approach for timing closure. The first approach, of using all the constraints together, results in faster convergence for timing closure, since the convergence is not divided into two steps. But, the control over the performance of the circuit is reduced. The second approach though, gives more control over performance of the design, can lead to large gates and results in a power hungry circuit. Reduction of this overhead, before the second step by increasing the datapath max delays by some approximate margin, can lead to achieving an overall better circuit. Also, the two step approach leads to dividing the set of constraints into two sets, which is very tedious.

The 64-point FFT design, described earlier, is selected as a start point, and both the timing closure approaches were applied. Individual designs generated for each timing closure approach were placed and routed and validated for correct functioning. Table 7.13 reports the result for each design by performing only timing closure on them. No optimization to resolve the design bottlenecks is performed. The latency and cycle time results for the second approach (datapath first) are larger due to the addition of 10 percent margin to all the datapath delays after the first step. The analysis of the designs show that the bottleneck for the second approach is at the interfaces, that are the decimator and expander, while the bottleneck for the first approach is at the complex multipliers. This results in similar simulation times for both the circuits. It is observed that the addition of the extra margin resulted in the improvement of the overall design, thus resulting in better energy numbers.

The approaches given above present an overview of the problems related to achieving timing closure for the methodology presented in this dissertation. Two unique solutions are presented to provide an algorithmic approach to solving the timing closure problem for bundled-data asynchronous circuits. The benefit of this systematic approach is the amount of time saved in addition to giving direction to achieve a working circuit. Before these approaches were derived, the 64-point FFT design timing closure problem took weeks to arrive at a solution, which has been reduced to days.

Table 7.13: Comparison of Timing Closure Approaches on 64-point FFT.

	Forward Latency (*ns*)	Backward Latency (*ns*)	Cycle Time (*ns*)	Area (um^2)	SimTime (*ns*)	Power (*mW*)	Energy/point (*pJ*)
All Constraints	68080	32390	606	527071.8	876.08	104.5	89.40
Datapath First	78850	35560	653	524556.0	956.96	95.3	89.04

7.6 Summary

Design of any circuit needs different modules, which need to be characterized for energy, performance and area. Templates for rapid characterization of the handshake circuits, and of the data and control steering blocks are described and their results reported for one specific handshake controller. Similar extensions can be done for any design, and numbers generated for better understanding of the design and its interaction with other design modules. These template characterization examples can also be used to implement new methodology steps, like addition of scan test to the datapath of the bundled data circuit using Tetramax, and also for analyzing the effects, like delay generation and min and max violations, for wire-load model evaluation.

Multirate asynchronous designs can lead to huge benefits in terms of energy, performance and area. This is validated by applying the multifrequency flows on 16-point and 64-point FFT circuits. Case studies presented in this chapter demonstrate that the flow can be efficiently applied to a large asynchronous design. The relative cost of development of an asynchronous circuit with the new flow is similar to its synchronous counterpart for development of these multirate designs. The FFT circuit operates at 1.4GHz and consumes 59.2pJ of energy per data point. A 2.4×, 2.4× and 3.2× benefit in terms of area, energy and throughput, respectively, over its synchronous counterpart is achieved. Also, a 0.48×, 4.5× and 32.20× benefit over a low power 64-point FFT design by Texas Instruments [48], as well as a 2.77×, 8.01× and 8.32× benefit over a similar 16-point FFT architecture [85] are reported, respectively, for area, energy and throughput.

The benefits and the overall application of this flow to mixed synchronous and asynchronous designs are shown to work on an OCP with different clocked and unclocked domains. Five different designs were evaluated under a uniform testbench. These included designs with a single global clock, fully asynchronous, and multifrequency designs. Designs with substantial asynchronous components are by far the best in terms of area, performance, and energy per transfer. The purely asynchronous design has 3× the performance and it consumes approximately 1/9 the energy of the clocked design. The GALS design also demonstrated almost 4× the throughput at less than 1/5 the energy per transaction. Much of the performance is due to the decreased area and performance penalty for synchronization, as well as the lower latency for asynchronous designs.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

A multifrequency methodology for generating purely asynchronous designs, as well as, designs with asynchronous and synchronous blocks using synchronous *computer aided design* (CAD) tools and flows using a unifying timing representation called *relative timing* (RT) are presented in this dissertation. The methodology is divided into two parts: the first part addresses the characterization of asynchronous design templates and describes the generation of constraints required to enable the use of synchronous CAD tools and flows on these designs. The second part uses the constraints to generate a working asynchronous/multifrequency circuit using the synchronous CAD tools. Custom algorithms required to enable the asynchronous template characterization with its automation are also presented.

The existing design flow for asynchronous circuits consists of good synthesis algorithms, but addition of reset to these circuits is an important manual step that needs to be automated. This work contibutes an algorithm to add a reset signal based on power/performance optimization. The theory of finding reset addition candidates based on the topology of the circuit, with and without the inputs of the circuit being defined is derived. Details about optimizations based on logical effort to reduce the impact of reset addition in terms of power and performance penalty for any design is also presented. The benefits of this algorithm are justified by a 21 percent and 24 percent reduction in area and energy/token, respectively, for the asynchronous FIFO controllers as compared to reset addition done by Petrify. A 14 percent and a 12 percent average reduction in area and energy/token is seen for the three asynchronous benchmark circuits.

The key to deriving the maximum benefits from the synchronous CAD tools and flows is to use their timing driven optimization and sizing algorithms. Because the asynchronous circuits are cyclic and sequential circuits, they have to be represented as *directed acyclic graphs* (DAGs) to apply these algorithms. This work presents an algorithm that automatically generates the cycle cut constraints to represent the timing graphs of asynchronous circuits as DAGs. The algorithm preserves the timing paths from being cut, thus enabling the use of *static timing analysis* (STA), timing driven optimization and sizing algorithms of the synchronous CAD tools to optimize asynchronous circuits. This algorithm preserves the timing paths, thus guaranteeing the applicability of the RT constraints to derive a functioning circuit. It also allows full control over the delays required to generate the best asynchronous circuits. The circuits generated by using this cycle cutting algorithm are 1/3 the size, consume 1/3 the energy and are 50 percent faster that those derived by the synchronous CAD tools.

The characterization of asynchronous circuits and the constraint generation is applied to derive a set of asynchronous templates. A family of 4-phase handshake protocols with data valid at the falling edge of request, also known as late data validity protocols are characterized for area, latency and energy. The tabulated results for the late data validity protocols enable the quick selection of the best asynchronous handshake protocols for any application.

General templates for steering data and control information were identified and circuit structures were implemented to characterize them. A simple toy example and four different FIFO structures were generated and automated for characterizing these templates. Detailed comparison of results for area, latency and energy of these designs is presented. These automated flows for characterization and benchmarking new circuit templates assists in quick development and comparison of new designs, thus facilitating in the design, analysis and optimization required for deriving better circuits.

The application of the novel RT based methodology to large and complex designs is shown in a few case studies. A 225k gate 64-point FFT circuit is designed and compared to a synchronous equivalent. The benefit of asynchronous designs for multifrequency applications is demonstrated with the asynchronous FFT circuit showing a benefit of 2.4×, 2.4× and 3.2× in terms of area, energy and throughput, respectively, over its synchronous counterpart.

The applicability of this methodology on design combination of both synchronous and asynchronous circuits is explored. A subset of the *open core protocol* (OCP) is implemented and the *domain interface* (DI) circuit concept is extended to circuits consisting of asynchronous circuit blocks. Detailed circuit implementations of the DI for asynchronous to synchronous domain crossing and vice versa are developed, and these designs are compared against the purely synchronous, asynchronous, and synchronous design with two different asynchronous clock domains. The purely asynchronous design has 3× the performance and approximately 1/9 the energy of the clocked design. The GALS design also demonstrated almost 4× the throughput at less than 1/5 the energy per transaction.

The utility and impact of this research work can be summed up as follows: This methodology enables the industry to transition from purely synchronous design approaches to asynchronous designs by exploring various asynchronous circuit design styles. It also allows the circuit designers to choose the best circuit solution for any specification. Thus, this work not only allows designers to create better designs, but it also opens up a host of optimization and algorithmic approaches that can be explored.

8.2 Future Work

The work presented in this dissertation addresses the overall methodology issues and analyzes its applicability to different circuit examples. The adoption of this research work can be enhanced by addressing certain automation steps, thus resulting in improved productivity. The proposed future work includes the following:

- The constraints used in the tool flow are generated by `ARTIST` tool and are mapped onto the circuit [40]. The presence of the timing paths and the application of the constraint on these paths is currently validated by hand after the synthesis. The same manual validation approach is followed after the physical place and route step to ascertain the validity of the constraints. Automation of this manual step facilitates in confirming the mapping of the constraints in the design. It also assists in confirming whether all the timing paths exist uncut in the design or not. This verification step results in increased productivity of the functional and timing correctness validation of the design after each step.

- The current implementation maps the RT constraints to the set_min_delay and set_max_delay timing constraints by hand based on the architecture of the overall system. The mapping of the constraints to timing paths differ based on the system level design. Hence, an automation tool, that can extract the connectivity between modules at the system level is required to map the local template level and intertemplate level constraints onto the system.
- The requirements of causal timing paths in the cycle cutting and the automatic reset generation algorithm can change the results of the output circuit and constraints considerably. Methods to automate the generation of causal path using a specific set of RT constraints need to be investigated.
- A subset of RT constraints is used at the initial steps of the flow and this subset derivation is based on intuition. Better algorithms to automatically generate the required RT constraints at each step of the flow need to be explored.
- The constraints for a system can be specified in various ways and the merits and demerits of the different approaches need to be explored with respect to different design styles.

APPENDIX A

RESULTS FOR CHAPTER 4

Table A.1: Forward latency ($ps/pipestage$) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Performance optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	462.50	87.50	–	102.50		367.50	357.50	352.50
R0020	257.50	–	357.50	87.50	252.50	92.50	152.50	430.00	515.00	337.50
R0040	240.00	97.50	242.50	175.00	327.50	165.00	205.00	285.00	337.50	435.00
R0022	102.50	297.50	260.00	87.50	435.00	177.50	172.50	300.00	255.00	280.00
R0042	132.50	247.50	167.50	–	285.00	112.50	177.50	332.50	285.00	332.50
R2022	90.00	105.00	325.00	120.00	152.50	90.00	147.50	282.50	–	.
R2042	82.50	205.00	175.00	–	167.50	117.50	95.00	235.00	230.00	.
R0044	85.00	155.00	192.50	125.00	315.00	92.50	167.50	335.00	282.50	285.00
R2044	140.00	160.00	150.00	–	217.50	127.50	150.00	257.50	232.50	.
R4044	75.00	125.00	.	125.00	.	115.00	.	.	.	.
R2222	257.50	207.50	190.00	182.50	197.50	252.50	220.00	257.50	–	.
R2242	192.50	217.50	275.00	225.00	305.00	230.00	212.50	295.00	385.00	.
R2262	137.50	222.50	285.00	160.00	250.00	.	.	332.50	.	.
R2244	142.50	107.50	127.50	80.00	132.50	85.00	85.00	220.00	182.50	.
R2264	105.00	100.00	167.50	92.50	162.50	.	.	247.50	.	.
R4244	95.00	105.00	.	102.50	.	90.00	.	.	.	.
R4264	125.00	105.00	.	47.50	.	.	.	.	.	.

Table A.2: Backward latency ($ps/pipestage$) for circuits with $lr\rightarrow la$ and $lr\rightarrow rr$ constraints (Performance optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	120.00	485.00	–	247.50	–	170.00	120.00	135.00
R0020	245.00	–	152.50	485.00	212.50	162.50	157.50	207.50	187.50	107.50
R0040	237.50	410.00	210.00	215.00	292.50	192.50	210.00	137.50	187.50	182.50
R0022	177.50	117.50	157.50	485.00	225.00	170.00	205.00	87.50	112.50	120.00
R0042	345.00	285.00	195.00	–	267.50	237.50	217.50	170.00	137.50	157.50
R2022	182.50	92.50	142.50	172.50	137.50	137.50	100.00	140.00	–	.
R2042	267.50	277.50	210.00	–	162.50	185.00	145.00	147.50	120.00	.
R0044	437.50	342.50	220.00	300.00	220.00	362.50	260.00	200.00	260.00	247.50
R2044	330.00	345.00	297.50	–	290.00	310.00	270.00	285.00	247.50	.
R4044	515.00	307.50	.	317.50	.	325.00	.	.	.	.
R2222	470.00	355.00	307.50	297.50	292.50	292.50	250.00	292.50	–	.
R2242	330.00	360.00	322.50	280.00	367.50	307.50	260.00	247.50	305.00	.
R2262	355.00	412.50	430.00	300.00	372.50	.	.	342.50	.	.
R2244	135.00	120.00	115.00	97.50	92.50	97.50	70.00	75.00	55.00	.
R2264	137.50	145.00	142.50	142.50	105.00	.	.	120.00	.	.
R4244	127.50	147.50	.	112.50	.	117.50	.	.	.	.
R4264	142.50	137.50	.	97.50	.	.	.	.	.	.

Table A.3: Cycle time (*ps*) for circuits with *lr*→*la* and *lr*→*rr* constraints (Performance optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	600	688	–	609	–	568	568	572
R0020	600	–	609	688	483	600	477	773	765	487
R0040	517	657	463	480	625	503	431	449	600	635
R0022	483	572	528	688	824	454	454	517	471	513
R0042	641	539	449	–	577	614	474	539	528	563
R2022	609	581	701	463	477	524	477	513		.
R2042	539	563	428	–	423	477	403	493	443	.
R0044	524	543	463	477	568	609	496	600	590	586
R2044	543	543	543	–	524	503	496	625	517	.
R4044	641	528	.	513	.	531	.	.	.	.
R2222	694	551	496	474	483	563	510	539		.
R2242	543	577	551	503	635	543	480	572	663	.
R2262	524	586	657	474	614	.	.	714	.	.
R2244	557	464	524	461	471	565	464	615	522	.
R2264	540	517	646	615	541	.	.	750	.	.
R4244	513	496	.	470	.	582	.	.	.	.
R4264	546	481	.	467	.	.	.	.	.	.

Table A.4: Routed core area (um^2) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Performance optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	200.58	324.91	–	326.59	–	335.16	178.32	176.58
R0020	343.73	–	276.05	324.91	218.59	246.89	191.16	200.58	218.59	162.00
R0040	234.86	274.32	211.75	218.59	246.02	272.59	183.48	204.00	185.16	137.18
R0022	288.88	293.16	200.58	324.91	273.46	213.48	190.32	179.16	130.32	121.73
R0042	327.43	221.18	195.48	–	190.32	307.78	182.58	229.75	150.90	116.59
R2022	246.02	328.36	206.58	204.90	261.43	216.00	165.48	187.74	–	.
R2042	232.34	224.57	195.48	–	183.48	191.16	155.16	132.86	114.00	.
R0044	193.74	138.86	162.90	140.59	222.05	162.00	161.16	163.74	106.32	147.48
R2044	161.16	192.00	133.73	–	144.00	160.32	120.00	95.18	100.32	.
R4044	162.90	119.18	.	129.46	.	155.16	.	.	.	.
R2222	280.30	149.16	144.00	110.59	154.32	173.16	150.90	123.46	–	.
R2242	331.72	173.16	152.58	178.32	132.86	156.90	119.18	106.32	82.30	.
R2262	172.32	170.58	156.00	137.18	129.46	.	.	150.90	.	.
R2244	106.32	151.74	115.73	89.17	115.73	96.00	76.28	80.57	62.57	.
R2264	165.48	160.32	102.86	102.86	56.59	.	.	78.88	.	.
R4244	151.74	120.86	.	98.59	.	82.30	.	.	.	.
R4264	92.59	99.46	.	82.30	.	.	.	.	.	.

Table A.5: Power consumed (*mW*) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Performance optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	1.19	1.74	–	2.01	–	2.08	1.19	1.08
R0020	2.07	–	1.75	1.74	1.62	1.43	1.50	0.97	1.07	1.22
R0040	1.68	1.62	1.69	1.57	1.51	2.13	1.41	1.68	1.17	0.78
R0022	2.26	1.84	1.32	1.74	1.22	1.60	1.41	1.21	0.95	0.77
R0042	1.84	1.43	1.43	–	1.21	1.95	1.33	1.57	0.95	0.68
R2022	1.43	2.14	1.03	1.71	2.05	1.52	1.22	1.29	–	.
R2042	1.36	1.40	1.55	–	1.53	1.40	1.28	0.95	0.86	.
R0044	1.29	0.96	1.20	0.96	1.43	0.97	1.14	0.92	0.58	0.91
R2044	1.14	1.42	0.97	–	0.98	1.19	0.91	0.51	0.65	.
R4044	0.83	0.85	.	0.88	.	1.13	.	.	.	.
R2222	1.45	0.96	0.99	0.87	1.23	1.22	0.99	0.83	–	.
R2242	2.22	1.02	1.06	1.17	0.80	0.99	0.86	0.69	0.49	.
R2262	1.25	0.90	0.87	0.90	0.76	.	.	0.82	.	.
R2244	0.68	1.21	0.73	0.65	0.95	0.55	0.53	0.41	0.35	.
R2264	1.11	1.02	0.51	0.54	0.29	.	.	0.30	.	.
R4244	1.16	0.89	.	0.69	.	0.37	.	.	.	.
R4264	0.61	0.75	.	0.62	.	.	.	.	.	.

Table A.6: Simulation time (Post-APR with SDF back-annotation) (*ns*) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Performance optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	158.72	180.59	–	160.85	–	150.68	149.38	150.89
R0020	155.93	–	169.91	180.59	128.04	157.95	126.25	201.18	200.25	129.66
R0040	136.92	172.12	122.91	127.99	164.30	133.55	114.86	120.77	157.46	172.11
R0022	127.45	150.93	139.56	180.59	215.48	121.47	120.66	136.65	125.25	136.06
R0042	168.88	142.56	119.81	–	152.40	162.50	125.22	142.91	139.65	149.02
R2022	159.53	152.70	183.42	123.53	128.70	138.54	125.91	136.08	–	.
R2042	140.47	149.02	118.56	–	112.92	126.96	119.37	131.29	118.01	.
R0044	138.09	143.73	122.96	126.32	149.97	160.19	131.28	158.98	155.45	154.19
R2044	142.38	142.98	142.43	–	138.11	130.28	131.21	163.26	136.64	.
R4044	167.89	140.04	.	135.44	.	139.91	.	.	.	.
R2222	181.43	146.24	131.65	125.99	128.08	148.64	134.78	142.38	–	.
R2242	143.96	152.63	145.25	133.26	167.55	142.96	127.62	151.73	174.50	.
R2262	138.89	154.69	173.29	126.44	161.73	.	.	189.36	.	.
R2244	145.96	122.37	137.49	124.44	124.25	148.25	122.43	161.57	137.71	.
R2264	141.23	136.62	168.51	160.65	141.97	.	.	195.86	.	.
R4244	135.84	130.01	.	123.63	.	152.33	.	.	.	.
R4264	143.09	126.18	.	124.04	.	.	.	.	.	.

Table A.7: Energy consumed ($pJ/token$) for circuits with $lr{\rightarrow}la$ and $lr{\rightarrow}rr$ constraints (Performance optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	0.74	1.23	–	1.26	–	1.22	0.70	0.64
R0020	1.26	–	1.16	1.23	0.81	0.88	0.74	0.76	0.84	0.62
R0040	0.90	1.09	0.81	0.79	0.97	1.11	0.63	0.79	0.72	0.52
R0022	1.13	1.09	0.72	1.23	1.03	0.76	0.66	0.64	0.46	0.41
R0042	1.21	0.80	0.67	–	0.72	1.24	0.65	0.88	0.52	0.40
R2022	0.89	1.28	0.74	0.83	1.03	0.83	0.60	0.69	–	.
R2042	0.75	0.82	0.72	–	0.67	0.69	0.59	0.49	0.39	.
R0044	0.70	0.54	0.58	0.47	0.84	0.61	0.59	0.57	0.35	0.55
R2044	0.63	0.79	0.54	–	0.53	0.61	0.47	0.32	0.35	.
R4044	0.55	0.46	.	0.47	.	0.62	.	.	.	.
R2222	1.03	0.55	0.51	0.43	0.61	0.71	0.52	0.46	–	.
R2242	1.25	0.61	0.60	0.61	0.52	0.55	0.43	0.41	0.33	.
R2262	0.68	0.54	0.59	0.44	0.48	.	.	0.61	.	.
R2244	0.39	0.58	0.39	0.32	0.46	0.32	0.25	0.26	0.19	.
R2264	0.61	0.55	0.34	0.34	0.16	.	.	0.23	.	.
R4244	0.61	0.45	.	0.33	.	0.22	.	.	.	.
R4264	0.34	0.37	.	0.30	.	.	.	.	.	.

Table A.8: Forward latency ($ps/pipestage$) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Power optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	442.50	90.00	–	135.00	–	412.50	375.00	337.50
R0020	317.50	–	407.50	90.00	247.50	90.00	195.00	430.00	515.00	375.00
R0040	270.00	102.50	255.00	177.50	342.50	190.00	242.50	297.50	367.50	465.00
R0022	107.50	310.00	260.00	90.00	415.00	205.00	197.50	325.00	277.50	357.50
R0042	125.00	252.50	235.00	–	285.00	112.50	205.00	315.00	285.00	332.50
R2022	82.50	122.50	347.50	145.00	210.00	62.50	185.00	275.00	317.50	.
R2042	85.00	212.50	195.00	–	190.00	157.50	150.00	255.00	250.00	.
R0044	85.00	155.00	217.50	125.00	315.00	117.50	187.50	332.50	282.50	352.50
R2044	162.50	185.00	167.50	–	217.50	137.50	175.00	275.00	232.50	.
R4044	87.50	125.00	.	125.00	.	140.00	.	.	.	.
R2222	252.50	230.00	207.50	182.50	207.50	260.00	235.00	267.50	–	.
R2242	227.50	235.00	300.00	225.00	305.00	230.00	220.00	320.00	385.00	.
R2262	150.00	222.50	285.00	170.00	250.00	.	.	332.50	.	.
R2244	142.50	107.50	145.00	90.00	145.00	102.50	95.00	230.00	197.50	.
R2264	137.50	140.00	167.50	100.00	157.50	.	.	250.00	.	.
R4244	87.50	112.50	.	107.50	.	75.00	.	.	.	.
R4264	125.00	105.00	.	62.50	.	.	.	.	.	.

Table A.9: Backward latency ($ps/pipestage$) for circuits with $lr{\rightarrow}la$ and $lr{\rightarrow}rr$ constraints (Power optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	125.00	482.50	–	285.00	–	195.00	132.50	137.50
R0020	245.00	–	142.50	482.50	185.00	170.00	180.00	207.50	187.50	135.00
R0040	235.00	427.50	207.50	220.00	317.50	212.50	197.50	160.00	190.00	207.50
R0022	190.00	137.50	182.50	482.50	222.50	202.50	212.50	107.50	145.00	140.00
R0042	340.00	305.00	220.00	–	267.50	235.00	230.00	195.00	137.50	162.50
R2022	187.50	127.50	130.00	225.00	135.00	140.00	127.50	147.50	112.50	.
R2042	235.00	302.50	230.00	–	200.00	202.50	182.50	172.50	145.00	.
R0044	437.50	342.50	240.00	300.00	220.00	385.00	282.50	207.50	260.00	300.00
R2044	355.00	377.50	322.50	–	290.00	342.50	285.00	302.50	247.50	.
R4044	517.50	307.50	.	340.00	.	375.00	.	.	.	.
R2222	485.00	420.00	340.00	295.00	317.50	287.50	295.00	280.00	–	.
R2242	325.00	395.00	350.00	280.00	367.50	307.50	275.00	270.00	305.00	.
R2262	340.00	412.50	430.00	282.50	372.50	.	.	342.50	.	.
R2244	135.00	120.00	132.50	105.00	95.00	110.00	77.50	90.00	65.00	.
R2264	152.50	170.00	142.50	140.00	102.50	.	.	122.50	.	.
R4244	127.50	165.00	.	120.00	.	115.00	.	.	.	.
R4264	142.50	137.50	.	107.50	.	.	.	.	.	.

Table A.10: Cycle time (ps) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Power optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	572	694	–	714	–	657	590	528
R0020	600	–	590	694	468	600	496	773	765	539
R0040	513	663	471	513	681	547	477	520	635	694
R0022	493	577	528	694	797	510	483	547	474	535
R0042	609	577	490	–	577	721	477	572	528	547
R2022	604	663	714	503	555	531	463	520	563	.
R2042	471	563	433	–	457	506	433	568	493	.
R0044	524	543	487	477	568	646	528	619	590	675
R2044	531	595	563	–	524	551	500	646	517	.
R4044	641	528	.	500	.	568	.	.	.	.
R2222	701	595	528	480	503	551	503	539	–	.
R2242	551	630	600	503	635	543	490	614	663	.
R2262	460	586	657	513	614	.	.	714	.	.
R2244	557	464	569	480	500	548	477	665	565	.
R2264	591	628	646	592	531	.	.	773	.	.
R4244	531	618	.	484	.	543	.	.	.	.
R4264	546	481	.	497	.	.	.	.	.	.

Table A.11: Routed core area (um^2) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Power optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	216.00	284.59	–	291.48	–	376.32	149.16	116.59
R0020	282.91	–	265.75	284.59	179.16	214.34	164.58	200.58	218.59	167.16
R0040	226.30	263.16	198.90	198.90	230.62	275.18	168.00	150.90	169.74	133.73
R0022	233.14	327.43	149.16	284.59	250.34	168.00	190.32	148.32	124.32	101.18
R0042	319.79	192.90	180.90	–	190.32	295.76	166.32	176.58	150.90	109.73
R2022	220.32	320.63	194.58	187.74	200.58	308.62	135.46	254.59	114.86	.
R2042	186.00	219.46	160.32	–	140.59	198.90	138.00	120.00	121.73	.
R0044	193.74	138.86	141.46	140.59	222.05	171.48	126.86	172.32	106.32	96.00
R2044	158.58	198.90	132.00	–	144.00	157.74	114.86	78.88	100.32	.
R4044	196.32	119.18	.	133.73	.	137.18	.	.	.	.
R2222	253.73	183.48	143.18	113.18	137.18	108.86	137.18	120.00	–	.
R2242	216.00	152.58	120.00	178.32	132.86	156.90	100.32	109.73	82.30	.
R2262	153.48	170.58	156.00	130.32	129.46	.	.	150.90	.	.
R2244	106.32	151.74	102.00	85.72	127.73	82.30	74.59	69.44	48.00	.
R2264	148.32	140.59	102.86	90.86	61.74	.	.	61.74	.	.
R4244	136.32	134.59	.	90.86	.	82.30	.	.	.	.
R4264	92.59	99.46	.	72.00	.	.	.	.	.	.

Table A.12: Power consumed (*mW*) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Power optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	1.36	1.46	–	1.45	–	2.11	0.87	0.75
R0020	1.68	–	1.49	1.46	1.28	1.28	1.09	0.97	1.07	1.11
R0040	1.74	1.35	1.62	1.42	1.21	1.89	1.14	0.98	1.01	0.69
R0022	1.67	2.09	1.02	1.46	1.12	1.07	1.21	0.86	0.87	0.49
R0042	1.89	1.12	1.22	–	1.21	1.54	1.09	1.08	0.95	0.59
R2022	1.25	1.85	0.98	1.37	1.36	2.10	0.99	1.81	0.65	.
R2042	1.35	1.21	1.22	–	1.03	1.40	1.13	0.77	0.91	.
R0044	1.29	0.96	0.92	0.96	1.43	0.85	0.79	0.96	0.58	0.50
R2044	1.13	1.27	0.88	–	0.98	1.03	0.72	0.35	0.65	.
R4044	1.06	0.85	.	0.99	.	0.96	.	.	.	.
R2222	1.28	1.06	0.93	0.85	1.00	0.72	0.84	0.81	–	.
R2242	1.38	0.72	0.79	1.17	0.80	0.99	0.67	0.70	0.49	.
R2262	1.24	0.90	0.87	0.88	0.76	.	.	0.82	.	.
R2244	0.68	1.21	0.60	0.62	0.94	0.49	0.52	0.29	0.23	.
R2264	0.88	0.80	0.51	0.48	0.30	.	.	0.22	.	.
R4244	0.96	0.75	.	0.60	.	0.47	.	.	.	.
R4264	0.61	0.75	.	0.51	.	.	.	.	.	.

Table A.13: Simulation time (Post-APR with SDF back-annotation) (*ns*) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Power optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	150.99	182.62	–	187.29	–	173.31	155.57	139.86
R0020	158.14	–	156.21	182.62	124.21	158.16	131.68	201.18	200.25	143.08
R0040	135.51	174.34	125.26	136.16	178.39	144.63	126.46	138.43	167.27	181.68
R0022	130.02	151.99	139.81	182.62	209.28	130.01	128.20	144.91	126.11	141.73
R0042	160.17	152.28	130.02	–	152.40	190.79	129.75	150.73	139.65	145.20
R2022	158.97	174.19	187.58	133.09	146.75	140.35	122.95	138.08	149.47	.
R2042	124.58	145.98	115.07	–	121.94	134.88	115.09	150.75	131.36	.
R0044	138.09	143.73	128.67	126.32	149.97	169.14	137.23	163.30	155.45	177.66
R2044	139.67	156.08	147.85	–	138.11	145.06	134.62	169.19	136.64	.
R4044	166.91	140.04	.	131.62	.	149.64	.	.	.	.
R2222	184.47	157.03	139.37	127.53	133.22	145.86	133.28	143.18	–	.
R2242	145.86	166.20	157.85	133.26	167.55	142.96	129.42	163.29	174.50	.
R2262	121.59	154.69	173.29	136.91	161.73	.	.	189.36	.	.
R2244	145.96	122.37	149.03	126.25	131.70	143.96	125.74	174.36	148.73	.
R2264	154.36	163.56	168.51	154.55	139.65	.	.	200.97	.	.
R4244	140.40	162.68	.	127.21	.	142.36	.	.	.	.
R4264	143.09	126.18	.	131.70	.	.	.	.	.	.

Table A.14: Energy consumed ($pJ/token$) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Power optimization).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	0.80	1.04	–	1.06	–	1.43	0.53	0.41
R0020	1.04	–	0.91	1.04	0.62	0.79	0.56	0.76	0.84	0.62
R0040	0.92	0.92	0.80	0.76	0.84	1.07	0.56	0.53	0.66	0.49
R0022	0.85	1.24	0.56	1.04	0.92	0.55	0.61	0.49	0.43	0.27
R0042	1.18	0.67	0.62	–	0.72	1.15	0.55	0.64	0.52	0.34
R2022	0.78	1.26	0.72	0.71	0.78	1.15	0.48	0.98	0.38	.
R2042	0.66	0.69	0.55	–	0.49	0.74	0.51	0.45	0.47	.
R0044	0.70	0.54	0.46	0.47	0.84	0.56	0.42	0.61	0.35	0.35
R2044	0.62	0.78	0.51	–	0.53	0.59	0.38	0.23	0.35	.
R4044	0.69	0.46	.	0.51	.	0.56	.	.	.	.
R2222	0.92	0.65	0.51	0.42	0.52	0.41	0.44	0.46	–	.
R2242	0.79	0.47	0.49	0.61	0.52	0.55	0.34	0.45	0.33	.
R2262	0.59	0.54	0.59	0.47	0.48	.	.	0.61	.	.
R2244	0.39	0.58	0.35	0.30	0.48	0.28	0.26	0.20	0.13	.
R2264	0.53	0.51	0.34	0.29	0.17	.	.	0.17	.	.
R4244	0.53	0.47	.	0.30	.	0.26	.	.	.	.
R4264	0.34	0.37	.	0.26	.	.	.	.	.	.

Table A.15: Forward latency ($ps/pipestage$) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Petrify).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	470.00	87.50	–	105.00	–	457.50	362.50	392.50
R0020	295.00	–	345.00	87.50	250.00	92.50	175.00	1002.50	462.50	412.50
R0040	255.00	102.50	255.00	195.00	340.00	195.00	230.00	292.50	355.00	432.50
R0022	100.00	315.00	230.00	87.50	457.50	207.50	170.00	335.00	295.00	252.50
R0042	115.00	235.00	247.50	–	250.00	120.00	170.00	355.00	275.00	320.00
R2022	87.50	132.50	357.50	112.50	185.00	62.50	172.50	257.50	295.00	.
R2042	90.00	227.50	197.50	–	220.00	152.50	125.00	307.50	235.00	.
R0044	85.00	140.00	210.00	130.00	352.50	92.50	175.00	290.00	260.00	367.50
R2044	140.00	145.00	135.00	–	200.00	160.00	110.00	307.50	270.00	.
R4044	102.50	147.50	.	112.50	.	115.00	.	.	.	.
R2222	295.00	190.00	222.50	172.50	235.00	277.50	267.50	292.50	–	.
R2242	200.00	210.00	272.50	222.50	245.00	212.50	230.00	292.50	297.50	.
R2262	177.50	212.50	272.50	187.50	225.00	.	.	322.50	.	.
R2244	140.00	92.50	122.50	110.00	127.50	120.00	105.00	270.00	177.50	.
R2264	137.50	177.50	187.50	110.00	137.50	.	.	285.00	.	.
R4244	60.00	127.50	.	130.00	.	80.00	.	.	.	.
R4264	90.00	82.50	.	85.00	.	.	.	.	.	.

Table A.16: Backward latency (*ps*/*pipestage*) for circuits with *lr*→*la* and *lr*→*rr* constraints (Petrify).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	147.50	465.00	–	310.00	–	162.50	150.00	177.50
R0020	250.00	–	127.50	465.00	175.00	210.00	232.50	267.50	240.00	147.50
R0040	260.00	437.50	227.50	235.00	337.50	210.00	195.00	150.00	217.50	197.50
R0022	175.00	112.50	170.00	465.00	287.50	195.00	245.00	120.00	182.50	85.00
R0042	330.00	307.50	227.50	–	210.00	247.50	257.50	165.00	132.50	147.50
R2022	222.50	127.50	137.50	230.00	150.00	170.00	137.50	117.50	140.00	.
R2042	335.00	367.50	240.00	–	225.00	187.50	225.00	202.50	155.00	.
R0044	407.50	310.00	260.00	330.00	247.50	397.50	267.50	177.50	240.00	342.50
R2044	400.00	407.50	352.50	–	270.00	387.50	255.00	342.50	280.00	.
R4044	517.50	337.50	.	342.50	.	345.00	.	.	.	.
R2222	525.00	347.50	370.00	282.50	365.00	315.00	322.50	332.50	–	.
R2242	420.00	345.00	337.50	232.50	282.50	275.00	295.00	242.50	235.00	.
R2262	367.50	407.50	442.50	320.00	347.50	.	.	342.50	.	.
R2244	145.00	127.50	112.50	130.00	107.50	132.50	110.00	110.00	52.50	.
R2264	180.00	250.00	160.00	160.00	135.00	.	.	140.00	.	.
R4244	272.50	167.50	.	140.00	.	122.50	.	.	.	.
R4264	140.00	137.50	.	132.50	.	.	.	.	.	.

Table A.17: Cycle time (*ps*) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Petrify).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	641	675	–	735	–	625	568	619
R0020	663	–	641	675	451	669	563	824	773	595
R0040	517	663	503	539	688	535	480	443	652	641
R0022	517	619	590	675	903	555	490	563	517	446
R0042	595	563	555	614	490	604	483	581	524	520
R2022	551	728	806	555	568	535	496	503	524	.
R2042	547	604	490	–	487	520	513	600	449	.
R0044	572	528	513	506	586	652	493	513	320	728
R2044	600	590	595	–	483	625	474	714	590	.
R4044	652	535	.	487	.	513	.	.	.	.
R2222	773	539	568	449	559	559	563	595	–	.
R2242	652	539	604	513	477	510	517	563	520	.
R2262	531	586	657	535	551	.	.	646	.	.
R2244	576	490	518	552	505	698	559	781	519	.
R2264	649	872	707	664	573	.	.	872	.	.
R4244	663	580	.	543	.	550	.	.	.	.
R4264	506	486	.	529	.	.	.	.	.	.

Table A.18: Routed core area (um^2) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Petrify).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	246.02	326.59	–	291.48	–	380.64	159.48	162.90
R0020	342.05	–	261.43	326.59	219.46	246.89	203.16	244.30	252.00	171.48
R0040	256.32	282.02	278.57	222.05	282.91	340.28	233.14	187.74	202.32	177.48
R0022	272.59	407.14	186.90	326.59	295.76	204.00	198.00	178.32	147.48	144.00
R0042	425.18	367.75	256.32	251.14	260.57	267.48	210.89	256.32	185.16	164.58
R2022	263.16	361.79	234.00	224.57	253.73	272.59	166.32	178.32	219.46	.
R2042	238.32	249.48	202.32	–	224.57	202.32	169.74	185.16	149.16	.
R0044	222.91	171.48	200.58	120.00	224.57	200.58	136.32	217.73	148.32	138.86
R2044	161.16	181.74	146.58	–	195.48	168.00	167.16	95.18	126.86	.
R4044	237.46	162.00	.	151.74	.	177.48	.	.	.	.
R2222	278.57	171.48	136.32	113.18	141.46	120.00	126.86	142.32	–	.
R2242	360.02	162.90	201.48	178.32	175.74	159.48	121.73	102.86	113.18	.
R2262	257.18	201.48	171.48	155.16	130.32	.	.	150.00	.	.
R2244	114.00	159.48	132.86	100.32	103.73	108.86	91.73	85.72	84.02	.
R2264	220.32	216.00	130.32	115.73	83.16	.	.	84.89	.	.
R4244	294.92	157.74	.	117.46	.	89.17	.	.	.	.
R4264	110.59	103.73	.	85.72	.	.	.	.	.	.

Table A.19: Power consumed (mW) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Petrify).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	1.41	1.81	–	1.42	–	2.37	0.99	1.02
R0020	1.82	–	1.42	1.81	1.72	1.35	1.21	1.03	1.23	1.02
R0040	1.94	1.60	2.32	1.29	1.58	2.44	1.58	1.44	1.14	1.04
R0022	1.84	2.44	1.13	1.81	1.07	1.31	1.33	1.05	1.04	1.01
R0042	2.60	2.52	1.53	1.26	1.84	1.65	1.38	1.69	1.17	1.08
R2022	1.41	1.79	1.04	1.37	1.50	1.94	1.09	1.20	1.48	.
R2042	1.38	1.42	1.32	–	1.59	1.26	1.08	1.16	1.11	.
R0044	1.46	1.20	1.24	0.78	1.25	1.11	1.03	1.46	0.78	0.72
R2044	0.87	1.07	0.78	–	1.32	0.99	1.12	0.42	0.83	.
R4044	1.15	1.19	.	1.17	.	1.24	.	.	.	.
R2222	1.31	1.14	0.81	0.95	0.97	0.78	0.83	0.87	–	.
R2242	2.07	1.07	1.27	1.25	1.41	1.02	0.82	0.71	0.75	.
R2262	1.83	1.30	1.01	0.84	0.80	.	.	0.92	.	.
R2244	0.72	1.35	0.84	0.68	0.79	0.57	0.58	0.35	0.44	.
R2264	1.16	0.81	0.72	0.65	0.50	.	.	0.31	.	.
R4244	1.52	0.95	.	0.72	.	0.42	.	.	.	.
R4264	0.92	0.82	.	0.59	.	.	.	.	.	.

Table A.20: Simulation time (Post-APR with SDF back-annotation) (*ns*) for circuits with *lr*→*la* and *lr*→*rr* constraints (Petrify).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	169.47	178.13	–	192.21	–	165.07	150.17	162.52
R0020	169.53	–	168.45	178.13	119.88	176.39	148.37	213.41	202.58	157.25
R0040	136.73	173.20	133.15	142.87	179.66	141.97	127.71	123.34	171.55	169.30
R0022	135.85	162.71	155.24	178.13	236.73	147.10	129.39	148.78	136.89	118.87
R0042	160.38	148.64	146.14	162.53	129.57	159.17	130.88	153.45	138.85	137.75
R2022	145.11	190.81	211.34	146.80	149.96	141.44	131.40	133.42	139.18	.
R2042	144.03	159.06	129.61	–	129.45	138.13	137.40	158.10	119.82	.
R0044	150.32	139.84	135.53	131.89	154.71	170.89	130.04	136.61	86.30	192.51
R2044	157.69	154.64	156.39	–	127.84	164.58	125.43	185.89	155.14	.
R4044	170.54	141.14	.	127.99	.	133.52	.	.	.	.
R2222	202.20	143.14	149.89	119.34	144.31	148.22	148.68	162.41	–	.
R2242	172.22	142.40	159.76	136.27	127.00	134.77	136.32	149.68	135.16	.
R2262	140.80	153.84	172.96	142.07	143.28	.	.	170.70	.	.
R2244	150.55	128.42	135.95	144.70	132.90	181.93	146.50	203.86	137.18	.
R2264	168.90	226.18	184.14	172.96	149.78	.	.	227.11	.	.
R4244	173.18	151.52	.	142.35	.	141.59	.	.	.	.
R4264	133.78	128.64	.	138.48	.	.	.	.	.	.

Table A.21: Energy consumed ($pJ/token$) for circuits with $lr \rightarrow la$ and $lr \rightarrow rr$ constraints (Petrify).

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	0.93	1.26	–	1.07	–	1.53	0.58	0.65
R0020	1.21	–	0.93	1.26	0.80	0.93	0.70	0.86	0.97	0.62
R0040	1.04	1.09	1.21	0.72	1.11	1.35	0.79	0.70	0.76	0.69
R0022	0.98	1.55	0.69	1.26	0.99	0.75	0.67	0.61	0.56	0.47
R0042	1.63	1.46	0.88	0.80	0.93	1.02	0.71	1.02	0.64	0.58
R2022	0.80	1.33	0.86	0.79	0.88	1.07	0.56	0.63	0.81	.
R2042	0.78	0.88	0.67	–	0.81	0.68	0.58	0.71	0.52	.
R0044	0.86	0.66	0.65	0.40	0.76	0.74	0.52	0.78	0.26	0.54
R2044	0.54	0.65	0.48	–	0.66	0.64	0.55	0.31	0.50	.
R4044	0.77	0.66	.	0.59	.	0.64	.	.	.	.
R2222	1.04	0.64	0.48	0.44	0.55	0.45	0.48	0.55	–	.
R2242	1.39	0.59	0.79	0.67	0.70	0.54	0.44	0.41	0.40	.
R2262	1.01	0.78	0.68	0.46	0.45	.	.	0.61	.	.
R2244	0.43	0.68	0.45	0.38	0.41	0.40	0.33	0.28	0.24	.
R2264	0.77	0.71	0.52	0.44	0.29	.	.	0.28	.	.
R4244	1.03	0.56	.	0.40	.	0.23	.	.	.	.
R4264	0.48	0.41	.	0.32	.	.	.	.	.	.

APPENDIX B

RESULTS FOR CHAPTER 5

Table B.1: Forward latency (*ps*) for circuits with cycle cuts generated by my algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	540	1910	–	–	920	840	630	510
R0020	1090	–	710	1410	750	730	700	770	750	560
R0040	890	2120	1020	1190	1610	950	770	640	1090	990
R0022	880	460	1160	890	850	860	880	470	620	540
R0042	1460	1260	840	–	1070	1150	940	820	730	650
R2022	910	520	620	610	590	780	580	490	510	.
R2042	1170	1580	890	790	800	870	700	640	610	.
R0044	1580	1350	900	1100	1170	1480	1060	720	940	1110
R2044	1510	1420	1290	1140	1100	1170	1120	1060	1080	.
R4044	2120	1460	.	1350	.	1390	.	.	.	.
R2222	1720	1340	1030	1050	1160	880	910	900	580	.
R2242	1130	1680	1320	880	1170	930	850	880	770	.
R2262	1030	1490	1350	1130	1320	.	.	1160	.	.
R2244	560	420	390	400	370	410	350	350	260	.
R2264	590	700	590	500	420	.	.	460	.	.
R4244	510	610	.	480	.	440	.	.	.	.
R4264	480	470	.	520	.	.	.	.	.	.

Table B.2: Forward latency (*ps*) for circuits with cycle cuts generated by commercial CAD tool.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	860	2690	–	–	1440	860	1440	1130
R0020	1450	–	860	2250	1430	1570	1660	1550	1220	1400
R0040	2130	3400	2120	1900	2160	2330	1080	1120	1470	840
R0022	1180	600	1050	1870	3030	1440	880	1180	1090	1070
R0042	1980	1610	1810	–	2370	2200	1040	940	1200	770
R2022	1850	1150	1010	790	480	1300	1420	880	1050	.
R2042	2130	3330	1870	1600	1570	1240	1660	710	1080	.
R0044	2500	2310	1250	1770	1780	1620	1100	890	980	2190
R2044	2310	1360	1370	1320	1060	1790	1260	1470	820	.
R4044	3180	2290	.	1980	.	2100	.	.	.	.
R2222	3400	1310	1410	2460	1540	1620	1710	1740	560	.
R2242	2090	1900	1600	1760	1000	1520	1630	1330	1060	.
R2262	1580	2320	2360	1540	1290	.	.	1030	.	.
R2244	1200	530	440	610	820	590	550	480	360	.
R2264	860	1310	980	810	500	.	.	580	.	.
R4244	710	1020	.	770	.	500	.	.	.	.
R4264	840	920	.	880	.	.	.	.	.	.

Table B.3: Backward latency (*ps*) for circuits with cycle cuts generated by my algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	1900	540	–	–	90	1880	1450	1300
R0020	1400	–	2540	670	1030	340	730	1480	1380	1650
R0040	1080	200	1240	510	1760	740	950	1130	1790	1810
R0022	400	1600	1780	940	1540	760	920	1190	980	1410
R0042	540	1060	920	–	1070	590	790	1400	1250	1240
R2022	330	580	1320	720	1000	360	820	1010	1170	.
R2042	880	1140	940	710	800	630	680	870	880	.
R0044	420	560	950	530	910	430	750	1170	1010	1270
R2044	850	730	820	860	870	570	720	940	900	.
R4044	280	600	.	570	.	500	.	.	.	.
R2222	860	680	630	580	590	330	590	800	790	.
R2242	780	910	1090	490	900	260	490	1050	950	.
R2262	420	650	760	500	700	.	.	1040	.	.
R2244	630	400	540	310	620	360	410	750	660	.
R2264	620	480	600	320	500	.	.	830	.	.
R4244	360	490	.	440	.	220	.	.	.	.
R4264	300	310	.	310	.	.	.	.	.	.

Table B.4: Backward latency (*ps*) for circuits with cycle cuts generated by commercial CAD tool.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	2750	560	–	–	30	2540	2680	2240
R0020	2800	–	4090	500	1680	390	2100	2530	2860	3220
R0040	2150	260	2880	390	2420	1130	1480	1680	3000	1600
R0022	430	2800	1660	2290	3510	1290	1180	2530	1590	2170
R0042	960	1430	1850	–	2530	1010	1130	1510	2200	1520
R2022	340	950	2210	970	1520	340	1940	1830	2550	.
R2042	1630	2150	2290	1460	1440	910	1800	950	1440	.
R0044	650	670	1170	810	1390	450	880	1310	1120	2340
R2044	1310	710	1160	1140	820	900	910	1320	700	.
R4044	360	700	.	900	.	820	.	.	.	.
R2222	1750	620	1000	1290	910	340	1250	1400	740	.
R2242	1600	1060	1230	790	790	230	1130	1430	1170	.
R2262	400	1030	1710	740	760	.	.	980	.	.
R2244	1250	670	590	650	1130	440	820	990	910	.
R2264	1440	1400	1150	640	540	.	.	1040	.	.
R4244	360	840	.	660	.	190	.	.	.	.
R4264	630	690	.	710	.	.	.	.	.	.

Table B.5: Cycle time (*ps*) for circuits with cycle cuts generated by my algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	710	710	–	–	21140	760	620	540
R0020	780	–	980	730	530	710	520	680	650	660
R0040	570	800	660	600	920	620	510	520	910	870
R0022	600	750	810	540	730	500	510	510	480	600
R0042	650	650	530	–	650	690	520	640	590	580
R2022	660	780	770	470	700	700	520	560	570	.
R2042	750	760	540	440	460	530	450	490	470	.
R0044	570	560	540	480	600	670	540	560	580	710
R2044	690	620	600	600	600	530	540	580	610	.
R4044	680	620	.	560	.	630	.	.	.	.
R2222	850	610	510	510	540	500	510	530	450	.
R2242	630	800	770	520	660	550	460	630	580	.
R2262	480	680	630	540	620	.	.	700	.	.
R2244	660	460	530	510	560	590	530	630	550	.
R2264	690	690	650	560	520	.	.	730	.	.
R4244	510	600	.	520	.	570	.	.	.	.
R4264	460	450	.	630	.	.	.	.	.	.

Table B.6: Cycle time (*ps*)for circuits with cycle cuts generated by commercial CAD tool.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	1040	930	–	–	21720	930	1170	970
R0020	1190	–	1420	950	920	1160	1040	1250	1150	1300
R0040	1190	1280	1370	630	1310	1200	780	830	1190	710
R0022	640	1260	830	1170	1690	790	630	1060	770	920
R0042	900	830	1060	–	1450	1170	620	770	980	710
R2022	870	1110	1260	650	1050	1000	990	970	1150	.
R2042	1440	1450	1170	860	830	650	990	520	710	.
R0044	870	890	710	770	850	780	590	660	580	1300
R2044	1070	590	720	730	580	800	670	770	470	.
R4044	1070	890	.	820	.	910	.	.	.	.
R2222	1560	610	730	1120	770	780	900	940	470	.
R2242	1160	890	870	920	640	790	840	870	710	.
R2262	720	1140	1200	820	670	.	.	650	.	.
R2244	1300	710	570	700	1050	680	800	810	710	.
R2264	1270	1480	1120	820	610	.	.	880	.	.
R4244	670	1010	.	820	.	560	.	.	.	.
R4264	800	890	.	970	.	.	.	.	.	.

Table B.7: Routed core area (um^2) for circuits with cycle cuts generated by my algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	442.4	286.4	–	–	155.2	265.8	170.6	128.6
R0020	258.9	–	351.5	235.7	166.3	282.9	152.6	145.7	162.9	153.5
R0040	180.0	238.3	192.0	200.6	348.0	214.3	152.6	142.3	175.7	198.9
R0022	231.5	286.4	272.6	137.2	256.3	170.6	179.2	114.9	104.6	113.2
R0042	285.4	180.0	162.9	–	202.3	262.3	179.2	168.9	121.7	114.9
R2022	210.9	245.2	247.8	147.5	233.1	266.6	160.3	137.2	123.5	.
R2042	269.1	178.3	135.5	149.2	188.6	146.6	111.5	125.2	111.5	.
R0044	173.2	145.7	149.2	118.3	145.7	156.0	134.6	145.7	118.3	104.6
R2044	138.9	150.9	142.3	147.5	156.0	156.0	132.0	87.4	114.9	.
R4044	186.9	142.3	.	135.5	.	139.7	.	.	.	.
R2222	248.6	142.3	125.2	111.5	111.5	108.0	111.5	111.5	80.6	.
R2242	183.5	304.3	131.2	142.3	118.3	138.9	94.3	94.3	94.3	.
R2262	152.6	147.5	135.5	111.5	111.5	.	.	114.9	.	.
R2244	124.3	120.9	84.0	94.3	102.9	94.3	73.7	70.3	56.6	.
R2264	133.7	118.3	97.7	94.3	90.9	.	.	70.3	.	.
R4244	128.6	121.7	.	97.7	.	80.6	.	.	.	.
R4264	101.2	104.6	.	94.3	.	.	.	.	.	.

Table B.8: Routed core area (um^2) for circuits with cycle cuts generated by commercial CAD tool.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	643.8	744.1	–	–	420.0	795.4	469.8	464.6
R0020	790.3	–	643.0	637.8	406.4	629.2	542.6	529.7	399.5	416.6
R0040	560.6	882.9	636.0	622.3	559.8	704.6	538.4	340.3	481.8	461.2
R0022	829.0	870.9	512.6	519.5	771.5	518.6	404.6	365.1	375.5	324.1
R0042	817.7	608.6	546.9	–	744.1	771.5	404.6	482.7	474.0	450.0
R2022	673.8	701.2	570.0	483.5	622.3	560.6	451.8	458.6	423.5	.
R2042	852.0	553.7	519.5	415.8	390.0	484.4	422.6	262.3	248.6	.
R0044	457.7	531.5	560.6	330.9	395.1	499.7	245.2	463.8	304.3	293.2
R2044	437.2	525.4	389.2	312.1	380.6	396.0	255.5	214.3	313.7	.
R4044	612.0	486.9	.	330.9	.	327.4	.	.	.	.
R2222	783.4	469.8	526.3	344.6	384.0	474.9	348.9	406.4	361.8	.
R2242	564.1	488.6	430.4	378.9	432.9	454.4	335.2	231.5	183.5	.
R2262	424.3	397.7	384.0	366.9	350.6	.	.	310.3	.	.
R2244	375.5	314.6	313.7	246.0	307.8	400.3	230.6	166.3	145.7	.
R2264	327.4	368.6	368.6	246.0	311.1	.	.	162.0	.	.
R4244	478.3	396.0	.	256.3	.	272.6	.	.	.	.
R4264	282.9	282.9	.	251.1	.	.	.	.	.	.

Table B.9: Power consumed (mW) for circuits with cycle cuts generated by my algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	2.441	1.098	–	–	0.795	1.004	0.668	0.641
R0020	0.891	–	2.011	1.115	0.739	0.855	0.838	0.573	0.650	0.544
R0040	0.918	0.815	0.900	1.328	0.781	1.041	0.760	0.652	1.412	0.617
R0022	1.094	1.255	1.485	0.775	0.918	0.830	0.755	0.541	0.589	0.463
R0042	1.506	0.780	0.840	–	0.794	0.906	0.766	0.778	0.511	0.536
R2022	0.989	1.056	0.993	0.796	0.944	0.895	0.692	0.863	0.565	.
R2042	1.066	0.691	0.775	0.885	0.841	0.741	0.713	0.704	0.611	.
R0044	0.864	0.706	0.718	0.508	0.692	0.626	0.548	0.684	0.499	0.427
R2044	0.513	0.596	0.645	0.685	0.637	0.968	0.545	0.354	0.496	.
R4044	0.724	0.619	.	0.723	.	0.782	.	.	.	.
R2222	0.886	0.638	0.576	0.600	0.536	0.610	0.586	0.594	0.383	.
R2242	0.861	1.392	0.478	0.791	0.520	0.654	0.566	0.454	0.490	.
R2262	0.862	0.529	0.732	0.473	0.499	.	.	0.426	.	.
R2244	0.528	0.663	0.349	0.502	0.467	0.441	0.382	0.259	0.211	.
R2264	0.586	0.494	0.445	0.465	0.361	.	.	0.224	.	.
R4244	0.684	0.583	.	0.496	.	0.298	.	.	.	.
R4264	0.602	0.637	.	0.412	.	.	.	.	.	.

Table B.10: Power consumed (*mW*) for circuits with cycle cuts generated by commercial CAD tool.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	2.566	3.222	–	–	1.591	3.245	1.575	1.936
R0020	2.554	–	2.016	2.076	1.605	2.245	2.089	1.632	1.350	1.264
R0040	1.952	2.871	1.984	2.904	1.651	2.447	2.744	1.921	1.577	2.010
R0022	4.843	2.844	2.401	1.717	1.918	2.376	2.214	1.300	1.917	1.308
R0042	3.349	2.770	1.938	–	2.123	2.733	2.333	2.207	1.726	2.169
R2022	3.070	2.400	1.729	2.891	2.432	2.063	1.701	1.703	1.452	.
R2042	2.531	1.510	1.717	1.696	1.506	2.586	1.576	2.099	1.147	.
R0044	2.046	2.269	2.960	1.377	1.716	2.350	1.406	2.067	1.762	0.911
R2044	1.724	3.298	1.978	1.558	2.216	1.814	1.341	1.112	2.118	.
R4044	2.263	2.006	.	1.479	.	1.393	.	.	.	.
R2222	2.085	2.893	2.746	1.196	1.591	2.321	1.456	1.861	2.372	.
R2242	1.935	1.889	1.928	1.528	2.618	1.978	1.511	1.127	0.826	.
R2262	2.543	1.238	1.306	1.587	1.875	.	.	1.836	.	.
R2244	1.186	1.792	1.885	1.314	0.916	2.269	1.006	0.682	0.700	.
R2264	1.067	0.997	1.290	1.122	1.557	.	.	0.693	.	.
R4244	2.528	1.451	.	0.998	.	1.708	.	.	.	.
R4264	1.355	1.136	.	0.961	.	.	.	.	.	.

Table B.11: Simulation time (Post-APR with SDF back-annotation) (*ns*) for circuits with cycle cuts generated by my algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	42.99	43.35	–	–	37.45	47.05	39.49	35.25
R0020	46.25	–	56.90	44.19	33.94	43.65	33.50	42.00	40.28	41.38
R0040	36.70	47.30	40.50	36.10	54.50	38.20	33.30	34.50	51.65	52.77
R0022	36.95	45.80	49.04	34.70	44.80	32.20	33.75	33.90	32.69	38.65
R0042	39.79	41.08	34.00	–	39.25	41.89	33.69	39.95	38.10	37.54
R2022	39.40	44.94	46.54	31.05	43.45	43.34	33.80	35.09	36.89	.
R2042	44.45	45.90	34.70	30.00	31.04	34.33	30.04	31.80	31.30	.
R0044	35.80	35.59	34.75	31.80	37.89	41.20	34.74	35.45	37.80	43.95
R2044	41.59	39.10	37.50	37.45	37.90	34.85	34.83	37.63	38.34	.
R4044	41.25	37.99	.	36.30	.	39.64	.	.	.	.
R2222	48.39	38.75	33.94	33.44	35.43	33.73	34.04	35.15	30.75	.
R2242	38.70	48.05	46.55	33.70	41.80	34.84	31.35	39.90	37.80	.
R2262	31.19	41.00	39.50	34.22	39.70	.	.	43.05	.	.
R2244	40.50	30.94	34.40	32.90	35.89	37.54	34.60	40.20	36.09	.
R2264	41.90	41.34	41.19	35.74	33.45	.	.	45.25	.	.
R4244	32.55	38.10	.	33.80	.	35.70	.	.	.	.
R4264	30.85	30.20	.	36.14	.	.	.	.	.	.

Table B.12: Simulation time (Post-APR with SDF back-annotation) (*ns*) for circuits with cycle cuts generated by commercial CAD tool.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	59.59	54.05	–	–	49.60	55.40	68.60	58.00
R0020	68.16	–	80.95	54.94	52.14	67.71	62.12	69.64	67.53	75.54
R0040	68.05	71.99	77.99	38.75	74.14	66.30	44.95	50.30	70.10	44.62
R0022	39.40	71.34	49.65	68.34	96.94	47.70	39.89	63.17	47.15	55.53
R0042	54.24	50.25	60.10	–	78.09	65.98	39.00	46.40	59.63	45.15
R2022	51.20	63.50	72.35	38.24	59.39	58.90	59.39	55.39	67.64	.
R2042	81.30	81.90	68.34	51.14	50.55	39.50	58.99	33.54	43.62	.
R0044	50.90	52.43	43.15	45.04	51.79	47.80	36.55	40.20	38.35	74.85
R2044	58.83	37.29	43.85	43.89	36.74	49.50	41.10	47.74	31.05	.
R4044	59.34	51.18	.	49.83	.	54.35	.	.	.	.
R2222	86.30	38.24	45.15	64.88	45.45	48.20	55.04	57.25	31.54	.
R2242	65.89	52.95	51.70	52.50	38.84	47.09	51.68	52.05	45.00	.
R2262	41.25	64.22	70.53	48.00	40.33	.	.	40.50	.	.
R2244	72.67	42.95	36.64	43.49	60.84	40.19	49.10	49.14	44.50	.
R2264	71.68	83.10	65.45	48.64	36.65	.	.	52.89	.	.
R4244	38.30	58.40	.	48.90	.	35.65	.	.	.	.
R4264	48.30	52.15	.	53.09	.	.	.	.	.	.

Table B.13: Energy consumed ($pJ/token$) for circuits with cycle cuts generated by my algorithm.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	2.098	0.952	–	–	0.595	0.945	0.528	0.452
R0020	0.824	–	2.288	0.985	0.502	0.746	0.561	0.482	0.524	0.450
R0040	0.674	0.771	0.729	0.959	0.852	0.796	0.506	0.450	1.459	0.652
R0022	0.808	1.149	1.457	0.538	0.822	0.534	0.510	0.367	0.385	0.358
R0042	1.198	0.641	0.571	–	0.623	0.759	0.516	0.622	0.389	0.402
R2022	0.779	0.949	0.924	0.494	0.820	0.776	0.468	0.605	0.417	.
R2042	0.947	0.634	0.538	0.531	0.522	0.509	0.428	0.448	0.382	.
R0044	0.619	0.503	0.499	0.323	0.524	0.516	0.380	0.485	0.377	0.375
R2044	0.427	0.466	0.484	0.513	0.483	0.675	0.380	0.267	0.380	.
R4044	0.598	0.470	.	0.525	.	0.620	.	.	.	.
R2222	0.857	0.494	0.391	0.402	0.380	0.412	0.399	0.417	0.236	.
R2242	0.667	1.338	0.445	0.533	0.435	0.456	0.355	0.362	0.370	.
R2262	0.538	0.433	0.578	0.324	0.396	.	.	0.367	.	.
R2244	0.428	0.410	0.240	0.330	0.335	0.331	0.265	0.208	0.153	.
R2264	0.491	0.409	0.367	0.332	0.242	.	.	0.203	.	.
R4244	0.445	0.444	.	0.335	.	0.213	.	.	.	.
R4264	0.371	0.384	.	0.298	.	.	.	.	.	.

Table B.14: Energy consumed ($pJ/token$) for circuits with cycle cuts generated by commercial CAD tool.

LoR	L0000	L0011	L1111	L0022	L1122	L0033	L1133	L2222	L2233	L3333
R0000	–	–	3.058	3.483	–	–	1.578	3.595	2.161	2.245
R0020	3.481	–	3.264	2.282	1.674	3.040	2.595	2.273	1.823	1.910
R0040	2.657	4.133	3.095	2.250	2.448	3.245	2.466	1.933	2.210	1.794
R0022	3.817	4.058	2.384	2.346	3.719	2.266	1.767	1.642	1.808	1.453
R0042	3.632	2.784	2.329	–	3.316	3.606	1.819	2.048	2.058	1.958
R2022	3.144	3.047	2.502	2.211	2.889	2.430	2.020	1.886	1.964	.
R2042	4.116	2.473	2.346	1.735	1.523	2.043	1.859	1.408	1.001	.
R0044	2.082	2.379	2.554	1.240	1.778	2.247	1.028	1.661	1.352	1.363
R2044	2.028	2.460	1.735	1.368	1.628	1.795	1.102	1.062	1.315	.
R4044	2.686	2.053	.	1.474	.	1.514	.	.	.	.
R2222	3.599	2.213	2.479	1.552	1.446	2.237	1.602	2.131	1.496	.
R2242	2.551	2.000	1.994	1.604	2.034	1.863	1.562	1.173	0.744	.
R2262	2.098	1.591	1.842	1.523	1.512	.	.	1.487	.	.
R2244	1.724	1.540	1.381	1.143	1.114	1.824	0.987	0.670	0.623	.
R2264	1.529	1.657	1.688	1.092	1.141	.	.	0.733	.	.
R4244	1.936	1.694	.	0.976	.	1.218	.	.	.	.
R4264	1.309	1.185	.	1.021	.	.	.	.	.	.

REFERENCES

[1] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design – A Systems Perspective*. Kluwer Academic Publishers, 2001.

[2] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing Power in High-Performance Microprocessors," in *35th Design Automation Conference (DAC'98)*, June 1998, pp. 732–737.

[3] P. E. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowan, and R. L. Allmon, "High-Performance Microprocessor Design," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 5, pp. 676–686, May 1998.

[4] K. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, and M. Roncken, "An Asynchronous Instruction Length Decoder," *IEEE Journal of Solid State Circuits*, vol. 36, no. 2, pp. 217–228, Feb. 2001.

[5] Synopsys Inc., Available at - http://www.synopsys.com.

[6] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous Design Using Commercial HDL Synthesis Tools," in *6th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr 2000, pp. 114–125.

[7] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC Flow for GHz Asynchronous Designs," *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.

[8] D. H. Linder and J. C. Harden, "Phased Logic: Supporting the Synchronous Design Paradigm with Delay–Insensitive Circuitry," *Computers, IEEE Transactions on*, vol. 45, no. 9, pp. 1031–1044, Sep 1996.

[9] *CHAINworks*, Silistix, Available at - http://www.silistix.com/.

[10] J. Bainbridge and S. Furber, "Chain: A Delay-Insensitive Chip Area Interconnect," *IEEE Micro*, vol. 22, no. 5, pp. 16–23, 2002.

[11] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "Handshake Protocols for De-synchronization," in *10th IEEE International Symposium on Asynchronous Circuits and Systems*, Apr 2004, pp. 149–158.

[12] J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "From Synchronous to Asynchronous: An Automatic Approach," in *Proceedings of the Conference on Design, Automation and Test in Europe*, vol. 2. IEEE, 2004.

[13] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "A concurrent model for de-synchronization," in *Proc. Intl. Workshop on Logic Synthesis*, 2003, pp. 294–301.

[14] N. Andrikos, L. Lavagno, D. Pandini, and C. P. Sotiriou, “A Fully-Automated Desynchronization Flow for Synchronous Circuits,” in *ACM/IEEE Design Automation Conference*, June 2007, pp. 982–985.

[15] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou, “Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1904–1921, Oct 2006.

[16] Y. Zhao, “Application of Synchronous Synthesis Tools for High-level Asynchronous Design,” Master’s thesis, The University of Utah, 2004.

[17] C. J. Myers and T. H. Meng, “Synthesis of Timed Asynchronous Circuits,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 1, no. 2, pp. 106–119, 1993.

[18] A. Smirnov and A. Taubin, “Synthesizing Asynchronous Micropipelines with Design Compiler,” *SNUG 2006 (Synopsys Users Group Conference, Boston, MA, 2006) User Papers*, 2006, Available at - http://asynceda.com/publications/snug06.pdf.

[19] B. R. Quinton, M. R. Greenstreet, and S. J. Wilton, “Asynchronous IC Interconnect Network Design and Implementation Using a Standard ASIC Flow,” in *International Conference on Compulter Design: VLSI in Computers and Processors*. IEEE, Oct 2005, pp. 267–274.

[20] K. S. Stevens, R. Ginosar, and S. Rotem, “Relative Timing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 11, pp. 129–140, feb 2003.

[21] S. B. Furber, J. D. Garside, and D. A. Gilbert, “AMULET3: A High-Performance Self-Timed ARM Microprocessor,” in *International Conference on Computer Design (ICCD)*, 1998, pp. 247–252.

[22] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver, “AMULET2e: An Asynchronous Embedded Controller,” *Proceedings of the IEEE*, vol. 87, no. 2, pp. 243–256, 1999.

[23] J. Woods, P. Day, S. Furber, J. Garside, N. Paver, and S. Temple, “AMULET1: An Asynchronous ARM Microprocessor,” *IEEE Transactions on Computers*, pp. 385–398, 1997.

[24] S. Hauck, “Asynchronous Design Methodologies: An Overview,” *Proceedings of the IEEE*, vol. 83, no. 1, pp. 69–93, 1995.

[25] A. Davis and S. M. Nowick, “An Introduction to Asynchronous Circuit Design,” *The University of Utah Technical Report, UUCS-97-013*, 1997.

[26] C. J. Myers, *Asynchronous Circuit Design*. Wiley, 2004.

[27] S. H. Unger, *Asynchronous Sequential Switching Circuit*. Krieger Publishing Co., Inc., 1983.

[28] A. J. Martin, “The Limitations to Delay-Insensitivity in Asynchronous Circuits,” in *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, W. Dally, Editor. MIT Press, 1990, pp. 263–278.

[29] A. J. Martin, “Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits,” in *Developments in Concurrency and Communication, UT Year of Programming Series*, C.A.R. Hoare, Editor. Addison-Wesley, 1990, pp. 1–64.

[30] C. J. Myers, W. Belluomini, K. Killpack, E. Peskin, and H. Zheng, “Timed Circuits: A New Paradigm for High-Speed Design,” in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. ACM, 2001, pp. 335–340.

[31] A. Taubin, J. Cortadella, L. Lavagno, A. Kondratyev, and A. Peeters, “Design Automation of Real-Life Asynchronous Devices and Systems,” *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 1, pp. 1–133, 2007.

[32] I. E. Sutherland, “Micropipelines,” *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[33] G. Birtwistle and K. S. Stevens, “A Design Space and its Patterns: Modelling 2phase Asynchronous Pipelines,” *Howard Barringer Festschrift*, 2013.

[34] J. You, “Design and Optimization of Asynchronous Network-on-chip,” Ph.D. dissertation, The University of Utah, 2011.

[35] S. Nagasai, K. S. Stevens, and G. Birtwistle, “Concurrency Reduction of Untimed Latch Protocols – Theory and Practice,” in *16th IEEE International Symposium on Asynchronous Circuits and Systems*, May 2010, pp. 26–37.

[36] S. N. Varanasi, “Performance Analysis of Four-Phase Untimed Asynchronous Handshake Protocols,” Master’s thesis, The University of Utah, May 2009.

[37] A. M. G. Peeters, “Single-Rail Handshake Circuits,” Ph.D. dissertation, Eindhoven University of Technology, 1996.

[38] S. M. Burns, “General Conditions for the Decomposition of State Holding Elements,” in *2nd IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996, pp. 48–57.

[39] J. Cortadella, M. Kishinevsky, S. M. Burns, and K. Stevens, “Synthesis of Asynchronous Control Circuits with Automatically Generated Relative Timing Assumptions,” in *Proc. International Conf. Computer-Aided Design (ICCAD)*, November 1999, pp. 324–331.

[40] Y. Xu and K. S. Stevens, “Automatic Synthesis of Computation Interference Constraints for Relative Timing,” in *26th International Conference on Computer Design*. IEEE, Oct. 2009, pp. 16–22.

[41] K. S. Stevens, Y. Xu, and V. Vij, “Characterization of Asynchronous Templates for Integration into Clocked CAD Flows,” in *15th IEEE International Symposium on Asynchronous Circuits and Systems*. IEEE, May 2009, pp. 151–161.

[42] H. Hulgaard, “Timing Analysis and Verification of Timed Asynchronous Circuits,” Ph.D. dissertation, University of Washington, 1995.

[43] G. Gill and M. Singh, “Bottleneck Analysis and Alleviation in Pipelined Systems: A Fast Hierarchical Approach,” in *15th IEEE International Symposium on Asynchronous Circuits and Systems*, 2009, pp. 195–205.

[44] E. Quist and P. Beerel, “Automated Path Specification for Static Timing Analysis of Relative Timing Designs,” in *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU Workshop)*. ACM, March 2010.

[45] Mentor Graphics, Available at - http://www.mentor.com.

[46] Cadence Design Systems Inc., Available at - http://www.cadence.com.

[47] H. Han and K. S. Stevens, “Clocked and Asynchronous FIFO Characterization and Comparison,” in *17th IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2009, pp. 101–108.

[48] V. Baireddy, H. Khasnis, and R. Mundhada, “A 64-4096 point FFT/IFFT/Windowing Processor for MultiStandard ADSL/VDSL Applications,” in *IEEE International Symposium on Signals, Systems and Electronics, ISSSE’07*, 2007, pp. 403–405.

[49] V. S. Vij and K. S. Stevens, “Automatic Addition of Reset in Asynchronous Sequential Control Circuits,” in *To be published in 21st IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2013.

[50] W. S. Coates, A. L. Davis, and K. S. Stevens, “Automatic Synthesis of Fast Compact Self-Timed Control Circuits,” in *IFIP Working Conference on Design Methodologies*, April 1993, pp. 193–208.

[51] K. Y. Yun and D. L. Dill, “Automatic Synthesis of Extended Burst-Mode Circuits: Part I (Specification and Hazard-Free Implementation),” *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 2, pp. 101–117, Feb 1999.

[52] K. Y. Yun and D. L. Dill, “Automatic Synthesis of Extended Burst-Mode Circuits: Part II (Automatic Synthesis),” *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 2, pp. 118–132, Feb 1999.

[53] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, “Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers,” *IEICE Transactions on Information and Systems*, vol. E80-D, no. 3, pp. 315–325, 1997.

[54] R. M. Fuhrer and S. M. Nowick, *Sequential Optimization of Asynchronous and Synchronous Fininte State Machines: Algorithms and Tools*. Kluwer Academic, 2001.

[55] I. Sutherland, B. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, 1999.

[56] K. S. Stevens, S. V. Robison, and A. Davis, "The Post Office – Communication Support for Distributed Ensemble Architectures," in *Proceedings of 6th International Conference on Distributed Computing Systems*, May 1986, pp. 160 – 166.

[57] K. S. Stevens and V. Vij, "Cycle cutting with timing path analysis," Jan 2013, US Patent 8,365,116.

[58] H. Hulgaard and S. M. Burns, "Bounded Delay Timing Analysis of a Class of CSP Programs with Choice," in *1st IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems*, November 1994, pp. 2–11.

[59] A. Smirnov, "Asynchronous Micropipeline Synthesis System," Ph.D. dissertation, Boston University, 2009.

[60] M. D. Riedel and J. Bruck, "The Synthesis of Cyclic Combinational Circuits," in *Design Automation Conference*. ACM/IEEE, 2003, pp. 163–168.

[61] S. Malik, "Analysis of Cyclic Combinational Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 7, pp. 950–956, 1994.

[62] S. A. Edwards, "Making Cyclic Circuits Acyclic," in *Proceedings of the 40th Conference on Design Automation*. ACM, 2003, pp. 159–162.

[63] O. Neiroukh, S. A. Edwards, and X. Song, "Transforming Cyclic Circuits Into Acyclic Equivalents," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1775–1787, 2008.

[64] T. R. Shiple, V. Singhal, R. K. Brayton, and A. L. Sangiouanni-Vincentelli, "Analysis of Combinational Cycles in Sequential Circuits," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, 1996.

[65] V. V. Filippovich, "Transforming a Cyclic Directed Graph Into an Acyclic Graph," *Cybernetics and Systems Analysis*, vol. 9, no. 2, pp. 348–351, March 1973.

[66] A. M. Lines, "Pipelined Asynchronous Circuits," Master's thesis, California Institute of Technology, Pasadena, CA, 1998.

[67] G. Birtwistle and K. S. Stevens, "The Family of 4-phase Latch Protocols," in *14th IEEE International Symposium on Asynchronous Circuits and Systems*, April 2008, pp. 71–82.

[68] K. Y. Yun and D. L. Dill, "A High-Performance Asynchronous SCSI Controller," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, 1995, pp. 44–49.

[69] S. Golson, "Resistance is Futile! Building Better Wireload Models," in *SNUG 1999 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, 1999, Available at - http://ns1.stevegolson.name/pdf/golson_snug99.pdf.

[70] F. J. te Beest, "Full Scan Testing of Handshake Circuits," Ph.D. dissertation, University of Twente, 2003.

[71] M. Roncken, “Partial Scan Test for Asynchronous Circuits Illustrated on a DCC Error Corrector,” in *1st IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, 1994, pp. 247–256.

[72] P. J. Hazewindus, “Testing Delay-Insensitive Circuits,” Ph.D. dissertation, California Institute of Technology, 1992.

[73] H. Han, “The Characterization and Evaluation of Clocked and Unclocked First-In-First-Out,” Master’s thesis, The University of Utah, 2008.

[74] S. Das, V. Vij, and K. S. Stevens, “SAS: Source Asynchronous Signaling Protocol for Asynchronous Handshake Communication Free From Wire Delay Overhead,” in *19th IEEE International Symposium on Asynchronous Circuits and Systems*, 2013, pp. 107–114.

[75] V. S. Vij, R. P. Gudla, and K. S. Stevens, “Interfacing Synchronous and Asynchronous Domains for Open Core Protocol,” in *To be published in 27th International Conference on VLSI Design (VLSI-Design)*, 2014.

[76] *Open Core Protocol Specification Ver.3*, Open Core Protocol, Available at - http://www.ocpip.org/.

[77] R. P. Gudla, “Design and Implementation of Clocked Open Core Protocol Interfaces for Intellectual Property Cores and On-chip Network Fabric,” Master’s thesis, The University of Utah, 2011.

[78] C. E. Cummings, “Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs,” in *SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers*, 2001, Available at - http://www.sunburst-design.com/papers/CummingsSNUG2001SJ_AsyncClk.pdf.

[79] C. E. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design,” in *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, 2002, Available at - http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf.

[80] J. Cortadella, M. Kishinevsky, and B. Grundmann, “Synthesis of Synchronous Elastic Architectures,” in *Proceedings of the Digital Automation Conference (DAC06)*. IEEE, July 2006, pp. 657–662.

[81] W. Lee, V. S. Vij, A. R. Thatcher, and K. S. Stevens, “Design of Low Energy, High Performance Synchronous and Asynchronous 64-Point FFT,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013, pp. 242–247.

[82] B. W. Suter and K. S. Stevens, “Low Power, High Performance FFT Design,” in *Proceedings of IMACS World Congress on Scientific Computation, Modeling, and Applied Mathematics*, no. 1, 1997, pp. 99–104.

[83] B. W. Suter, *Multirate and Wavelet Signal Processing*. Academic Press, 1997.

[84] R. Miller, *Switching Theory*. Wiley, 1965, vol. 1.

[85] X. Guan, Y. Fei, and H. Lin, “Hierarchical Design of an Application-Specific Instruction Set Processor for High-Throughput and Scalable FFT Processing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 3, pp. 551–563, March 2012.

[86] K.-S. Chong, B.-H. Gwee, and J. S. Chang, “Energy-Efficient Synchronous-Logic and Asynchronous-Logic FFT/IFFT Processors,” *IEEE Journal of Solid-State Circuits*, vol. 42, no. 9, pp. 2034–2045, 2007.

[87] B. M. Baas, “A Low-Power, High-Performance, 1024-Point FFT Processor,” *IEEE Journal of Solid-State Circuits*, vol. 34, no. 3, pp. 380–387, 1999.

[88] A. Chandrakasan, W. J. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*. Wiley-IEEE Press, 2000.