Asynchronous Design for Low-Power

Sonal Khosla

Master of Telematics - Communication Networks and Networked Services (2
Submission date: August 2015
Supervisor: Rolv Bræk, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title:	Asynchronous Design for Low-Power
Student:	Sonal Khosla

Problem description:

Asynchronous logic has been shown to have many theoretical advantages, ranging from lower power consumption to faster operations. However, there are many practical hazards that need to be overcome – not least of which is the lack of tool support, which has discouraged extensive use in the industry. The overall goal is for the student to investigate the usability and/or power saving potential for selected circuits and if needed, implement macros for blocks with high power saving potential.

The student would start by designing asynchronous logic for reading from FLASH memory. Some initial goals are to:

1. Select the correct backend tool for synthesis.
2. Investigate issues for an asynchronous synthesis tool to be integrated into the Nordic Semiconductor's tool-chain.
3. Synthesis, timing analysis, netlist simulations.

If time permits, further potential goals could be:

1. Power analysis.
2. Layout, placement, route, macro-ify.
3. Investigate asynchronous design for writing to FLASH.

The following skills/knowledge would be helpful (in decreasing importance):

1. A very good grasp of Verilog/System-Verilog and digital design principles.
2. An understanding of asynchronous design techniques.
3. Some basic understanding of the process from RTL to ASIC.
4. Power analysis / power simulations.

Responsible professor:	Rolv Bræk, NTNU
Supervisor:	Omer Qadir & Arne W. Venås, Nordic Semiconductor ASA

Abstract

The last two decades have seen a sudden surge of renewing interest in asynchronous (or "clock-less") digital circuit design, along with its introduction into various consumer products in the industry. One major bottleneck to the further advancement of asynchronous design is the inability to integrate it with standard synchronous design. Furthermore, the ease of integration is primarily dictated by the effort spent on timing validation and analyses. It is easier to integrate an asynchronous design into a synchronous tool chain, if no or less timing analyses is required for the asynchronous part. This thesis 1) investigates several asynchronous tools to find an asynchronous tool that nullifies or minimises the timing validation to implement an asynchronous design; 2) uses "Balsa", the selected tool, to implement an asynchronous flash readout; 3) analyzes the delay insensitive nature of Balsa; and 4) integrates a simple asynchronous buffer design in Balsa together with a synchronous RTL design, and runs the combined design through Nordic Semiconductor's synchronous tool chain.

An asynchronous flash readout takes control of the flash and performs readouts from the flash. For this purpose, a Verilog netlist for both single rail and dual rail data encoding styles was generated. An analyses of the delay insensitive (DI) nature of Balsa was performed using a simple buffer design. It was found that for a Balsa generated netlist to be Delay Insensitive (DI), all the combinatorial loops must be controlled by handshaking signals and all components must be Quassi Delay Insensitive (QDI).

An attempt to integrate the Balsa Verilog netlist with Nordic Semiconductor's tool chain was made. The correct functionality of the combined design was verified before and after it was synthesized by Design Compiler (DC). One important feature is that after the synthesis of the combined design, no timing analyses is required to ensure the working of the asynchronous part. A manual inspection of the combined netlist revealed no changes in the functional behaviour for the Balsa part. Therefore, the integration of an asynchronous design written in the Balsa language is possible with Nordic Semiconductor's tool chain.

Preface

In the last decades, the market has been flooded with high performing, power efficient devices. 'Low power devices' has been one of the chief priorities for many progressive companies including Nordic Semiconductor. Research has shown that asynchronous design practices can achieve significant power savings, thus making asynchronous design a suitable choice for low power implementations. Currently, the industry for digital circuit design is dominated by synchronous design practices which are bound to stay for many years to come. Hence, for the successful adoption of asynchronous design practices, it is essential to be able to integrate asynchronous design with a standard synchronous design. This thesis aims at doing so. It integrates a simple asynchronous design in Balsa together with a synchronous RTL design, and runs the combined design through Nordic Semiconductor's synchronous tool chain.

This thesis has been submitted for the fulfillment of Masters in Telematics being pursued at the Norwegian University of Science an Technology (NTNU). It was undertaken over a course of 21 weeks. The assignment was given by Nordic Semiconductor and was performed under the supervision of Omer Qadir and Arne Wanvik Venås. As a part of masters degree, I did a specialization project on 'Asynchronous Design through Synchronous Tool Flow' preceding this master thesis in 2014. This research provided some really interesting results. We ran into some timing issues that could not be solved by this approach. Hence, it was decided to carry forward the research into asynchronous design by using another approach i.e. by using an asynchronous tool to implement an asynchronous design and then integrate it into a synchronous tool flow.

In the end, I would like to acknowledge the efforts of all those who have contributed towards the successful completion of this thesis. I would like to express my gratitude towards my co-supervisor, Arne Wanvik Venås, who has been a perennial source of guidance and motivation. His contribution has been significant in the various phases of this thesis, such as: design implementation, library generation, integration, etc. I would also like to thank my superviser, Omer Qadir, for all his endeavours to achieve the final goal. His contribution has been significant during the initial phase of the research into various asynchronous tools to find the correct asynchronous tool, and in the delay insensitivity analysis of Balsa. Moreover, he has played an important role towards ensuring the academic perspective of this thesis. I would also like to thank Frank Burns and Danil Sokolov at the University of Newcastle, Luciano Lavango at Politecnico di Torino, and the Balsa support at the University of Manchester for their quick and enriching responses during the initial research phase. My extended thanks to the Department of Telematics at the Norwegian University of Science and Technology (NTNU) for providing me the opportunity to pursue this thesis.

Contents

List of Figures

List of Tables

List of Acronyms

ACC Asynchronous Circuit Compiler.

CAD Computer-aided Design.

CHP Communicating Hardware Processes.

CPN Coloured Petri Net.

CSP Communicating Sequential Process.

CU Control Unit.

DC Design Compiler.

DCs David Cells.

DESI Decomposer Signal Transition Graph.

DI Delay Insensitive.

DP Data Path.

DUT Device Under Test.

EMI Electromagnetic Interference.

FPGA Field Programmable gate array.

FSM Finite State Machine.

GUI Graphical User Interface.

HDL Hardware Description Language.

IC Integrated Circuit.

LARD Language for Asynchronous Research and Development.

LPN Labelled Petri Net.

NTNU Norwegian University of Science and Technology.

PN Petri Net.

PNML Petri Net Markup Language.

PT Prime Time.

QDI Quassi Delay Insensitive.

RTL Register-Transfer Level.

SDF Standard Delay Format.

SG State Graph.

SI Speed Independent.

SoC System on Chip.

STG Signal Transition Graph.

TS Transition System.

VHDL VHSIC Hardware Description Language.

VLSI Very Large Scale Integration.

VSTGL Visual STG Lab.

Chapter 1 Introduction

1.1 Motivation

Advancements in semiconductor technology has continually reduced the size of transistors and wires which has inturn reduced chip size. As a consequence, a single IC chip can now support many more transistors and can perform extremely sophisticated tasks. [2] The market for these IC chips have thereby put more demands of better performance and lower power on digital circuits. Asynchronous circuits have some unique characteristics that can fulfil these demands. "While the synchronous approach to digital circuit design has led to dramatic progress in the advancement of modern computers" [10], asynchronous circuits offer the advantages of lower power, improved performance and low Electromagnetic Interference (EMI).

One alternative is to design asynchronous circuits using synchronous Hardware Description Language (HDL) Computer-aided Design (CAD) tools. [11] This approach, however, requires strenuous timing validation effort. Research has shown that by adopting this approach, it is most likely to run into timing problems with combinatorial loops. Moreover, it is almost impossible to constraint one path with respect to another. [11] The other alternative is to design asynchronous circuits using automatic asynchronous design tools. This is the approach that has been adopted in this thesis. An asynchronous design tool, generating a netlist where no or the least timing analysis is required, has been selected. A tool generating a Delay Insensitive (DI) netlist is good for this purpose. The selected asynchronous tool has been employed to implement an asynchronous flash readout. Currently, the flash readout at Nordic Semiconductor is performed synchronously. An asynchronous implementation of the same is driven by time saving and power saving interests.

Even though "synchronous design is likely to dominate for the foreseeable future" [10], there is no doubt that asynchronous design will play a significant role in the field of digital design in the coming years. One of the major bottlenecks in the widespread adoption of asynchronous design is the inability to integrate it with standard synchronous design. Synchronous design is bound to dominate the industry for many years. Hence, it is extremely essential to be able to integrate an asynchronous design with a synchronous one. Moreover, a synchronous implementation of a major part of the design, and an asynchronous implementation of some small stand alone parts where higher performance and lower power is required, saves effort and is the most optimal solution. The key limitation in the current industry or research integration practices is that a significant effort has to be spent on timing validation and analysis. Many researches that have aimed at integrating an asynchronous design with a synchronous design, land up using an asynchronous tool or method that leads to the generation of a Speed Independent (SI) netlist, thereby desiring alot more timing analysis. Hence, this thesis aims at choosing an asynchronous tool that nullifies or minimises the timing validation to implement an asynchronous design, and then tries to integrate it with some synchronous design in standard HDL by passing it through Nordic Semiconductor's synchronous tool chain.

1.2 Significance of the thesis

The novelty of this thesis lies in integrating a complete asynchronous design using Balsa together with a synchronous RTL design, and running the combined design through a normal synchronous tool chain. One important feature is that after the synthesis of the combined design, no timing analysis is required to ensure the working of the asynchronous part. Other than Balsa, no special tools are required for accomplishing this task. Standard synchronous HDL tools, only meant for synchronous design, have been used to handle a combined (asynchronous and synchronous) design.

The tool "Balsa" was used to implement an asynchronous flash readout. For this purpose, several tools have been investigated and analysed to find the most suitable tool for the task.

1.3 Report structure

The structure of the rest of the report is as follows: Chapter 2 presents the background and theory which is essential for understanding the following chapters. Chapter 3 describes the work done previously in the various categories of asynchronous tools/methods, and in the field of integrating an asynchronous design (using an asynchronous tool) with the existing synchronous tool chain. Chapter 4 lists all the tools used throughout the thesis and states the methodology followed. Chapter 5 presents the most interesting tools and methods that have been studied in depth for the purpose of this research, and shortlists the best suited tools based on certain criteria/parameters. Chapter 6 investigates the integration aspects, possibilities and issues for each of the short listed tools, with Nordic Semiconductor's design flow, in order to select a tool that integrates best with Nordic Semiconductor's tool chain. Chapter 7 explains some basic concepts of Balsa. Chapter 8 is entirely dedicated to the design implementation, synthesis and verification of the asynchronous flash readout written in Balsa. Chapter 9 describes the generation of a new Balsa technology for the purpose of this thesis. Chapter 10 presents an analysis of the delay insensitive (DI) nature of Balsa. Chapter 11 is solely dedicated to the integration of the Balsa output with Nordic Semiconductor's tool chain at the earliest possible stage of the design flow. Chapter 12 summarizes the work done in this thesis and the important results obtained. Chapter 13 lists all the possible works that could be done in the future as a continuation to this thesis.

Chapter 2
Background and Theory

This chapter presents the necessary background, theory and all the other information which is fundamental to understand the rest of the report. It sets out to explain concepts that form the basis for understanding asynchronous circuit design. It is noteworthy that the two initial sections i.e. Section 2.1 and Section 2.2 are summarized from the project report [11] submitted as a part of fulfillment of the third semester in M.Sc Telematics at Norwegian University of Science and Technology (NTNU). This is because the area of research of the thesis is another aspect of asynchronous design from what was researched while undertaking the project. Hence, sharing of some common background information between the project and thesis is natural. [11] should be referred for more details on basic concepts (section 2.1 of [11]) of synchronous and asynchronous design (Chapter 1 and 2 of [11]), benefits of asynchronous design over synchronous design (section 2.2 of [11]), challenges in asynchronous design (section 2.3 of [11]), basics of synthesizing ASIC (section 6.1 of [11]) and timing analysis and timing constraints (section 6.2 of [11]).

2.1 Synchronous and asynchronous design

Most of the prevalent digital design practices are based on a synchronous design approach, where the execution of all the functions in a circuit are dictated by the same periodic global signal, called the clock. For a hazard free operation, "the clock period is ensured to be large enough for the system to achieve a stable state before the next active clock edge." [11] Before the arrival of the clock event, all the inputs to the various latches in the circuit, must be stable. On the arrival of the clock signal all latches change simultaneously. The "clock is used to synchronize the data transferring between combinational logic blocks and filter out unexpected transient events (called glitches) before the circuit becomes stable." [12] Synchronous design has been the dominant design approach in the industry since the last many decades.

In contrast, there is no globally distributed clock in asynchronous circuits. Instead, the subsystems of an asynchronous system "communicate with each other at arbitrary times when they wish to exchange information." [11] Asynchronous design makes use of handshaking protocols to exchange data between functional blocks. Asynchronous design practices have various benefits over synchronous design. This has renewed the interest of researchers and designers in asynchronous design methodologies.

2.2 Benefits and challenges of asynchronous design

Asynchronous design offers several advantages over its synchronous counterpart, in terms of low power, improved performance and low EMI. [2] [1]

- **Low power**

  In synchronous systems, power is supplied to every part in a circuit even though that part of the circuit might not be involved in an ongoing computation, and might not be involved in any data processing at all. The continuous activity of a periodic global clock leads to increased power consumption in a system. From this perspective, asynchronous systems lead to significant power savings due to the lack of a global clock. Moreover, asynchronous circuits exhibit low standby power. This is because unlike synchronous design, they do not need to poll the external signals all the time and can respond simply whenever external stimulus is available. [11]

- **Improved performance**

  Asynchronous systems use handshaking to sense completion of a computation, and hence exhibit an average-case performance or delay. This is in contrast to the worst-case performance or delay of synchronous systems, where the system has to stabilize before the next clock tick. [11]

  Moreover, due to the lack of a global clock, clock skew is not an issue in asynchronous circuits. "Clock skew is the difference in arrival times of the clock signal at different parts of the circuit." [1] However, in synchronous circuits the clock skew is accommodated by making the clock period larger than the critical path. This produces slower circuits. [11]

- **Low Electromagnetic Interference**

  Synchronous circuits have a very narrow noise spectrum, and thus there is significant amount of electromagnetic noise in the bandwidth. The electromagnetic noise interferes with the neighbouring circuit signals. On the other hand, asynchronous circuits have a more distributed noise spectrum and thus display low EMI. [11]

Despite of all the above mentioned advantages, asynchronous design encounters certain challenges, in terms of design complexity, difficulty in testing and debugging and lack of CAD tool support. [2] [1]

- **Design complexity**

  Synchronous circuits can be easily designed in an ad-hoc fashion. This is however not the case for asynchronous circuits, where ordering of operations is a difficult task that needs to be ensured. "The use of flip-flops in synchronous design ensures the ordering of operations by making sure that all data is stable before the next active clock edge." [11] Due to the lack of the clocked operation of flip-flops, all signals in asynchronous control circuits react to a change in an input. This can cause glitches in the output signal. "A glitch is a momentary, undesired signal transition of short duration that occurs at the output of a logical circuit before a signal settles down to its intended state or value." [11] Hence, in order to achieve a hazard free operation (caused by glitches) in an asynchronous electronic circuit, ensuring the ordering of operations is a necessity.

- **Difficulty in testing and debugging**

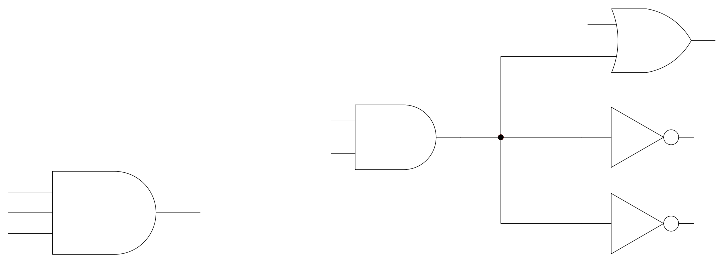
  Asynchronous design is more prone to deadlocks, due to the presence of combinatorial or timing loops that are not cut by flip-flops or latches. "Many of these loops must be cut with additional circuitry in order to achieve sufficient observability and controllability in the circuit for test purposes." [2] The availability of advanced CAD tools and the presence of latches and flip-flops makes it easier to test synchronous circuits as compared to asynchronous circuits. Furthermore, the high speed operation of asynchronous circuits makes debugging difficult. Often, delays have to be added to reduce the speed of the circuits, and make debugging possible. [11]

- **Lack of CAD tool support**

  There are very few CAD tools, synthesis techniques and commercially supported design flows available for asynchronous systems. Moreover, they are far less capable as compared to the readily available synchronous design tools. [11]

2.3 Fanin and fanout paths

In digital electronics, fanin paths are all the input paths to a logic gate. Figure 2.1a on the following page shows an AND gate with three fanin paths. Fanout paths on the contrary are the input paths to logic gates fed from the same logic gate output. Figure 2.1 on the next page represents a fanout where the output from the AND gate is connected to 3 other logic gates.



(a) ***Fanin of three*** **(b)** ***Fanout of three***

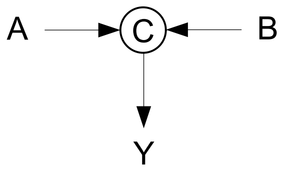
Figure 2.1: *Fanin and fanout*

2.4 Frontend and backend

In a digital design flow, frontend refers to carrying out the RTL design implementation and verification, while backend refers to carrying out the synthesis, layout, placement, routing and physical verification.

2.5 Muller C element

Asynchronous design very often comprises of a special library comprising of various library elements to which it is mapped to. The Muller C gate is a well known asynchronous library element. A 2 input Muller C gate along with its associated truth table has been shown in Figure 2.2. A Muller C gate can be defined as an element that outputs a 1 when all its inputs are 1, outputs a 0 when all its inputs are 0 and holds the same state as before otherwise. [1] In other words, it can be said that it is an element which propagates an event only when there has been an event on all its inputs. [11]



A	B	Y
0	0	0
0	1	Y_{n-1}
1	0	Y_{n-1}
1	1	1

Figure 2.2: *Muller C element and truth table*

2.6 The data path and control path

An asynchronous system is a combination of a number of modules or components communicating with each other. It can be visualized as a partitioning of two main parts: the data path and the control path. The data path includes all the logical units and blocks required for processing data, such as, combinational logic blocks, adders, data encoders, etc. It also includes all the units required for the communication and storage of data signals, such as, registers. On the other hand, the control path usually includes signals that control, monitor and ensure the proper functioning of the data path. The control path most commonly comprises of handshaking signals, such as, request and acknowledgement. These handshake signals coordinate the communication between the different parts of an asynchronous circuit to ensure its correct operation. [12]

2.7 Handshaking protocols for asynchronous control communication

Typically, asynchronous control communication is based on handshaking protocols employed between a sender and a receiver. Such protocols are initiated by a request from the sender to the receiver indicating an action initialization, as a result of which corresponding acknowledgements are sent back from the receiver to the sender indicating the action completion. [10] Two phase handshaking and four phase handshaking are the most widely used handshaking protocols.

2.7.1 Two-phase handshaking

A two phase handshaking protocol involves two signal transitions: a request and an acknowledgement. The handshake is initiated by a request and is completed by an acknowledgement. In two phase handshaking, every toggle on the request wire corresponds to a new handshake and a new data transfer. Figure 2.3 on the next page represents two phase handshaking.

As an example, consider a two phase handshaking protocol employed between two modules. The transition sequence is given by: req+, ack+, signifying the first handshake and data transfer. The next handshake and data transfer between these two modules will be given by: req-, ack-. The third handshake and data transfer will again be given by: req+, ack+, and so on. [10]

2.7.2 Four-phase handshaking

A four phase handshaking protocol involves four signal transitions: two requests and two acknowledgements.[10] The handshake is initiated by asserting a request, followed

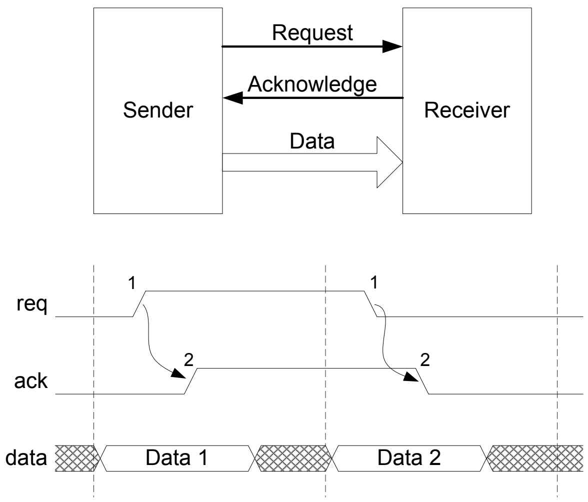


Figure 2.3: *Two-phase handshake protocol*

by asserting an acknowledgement, which in turn is followed by deasserting the request followed by deasserting the acknowledgement. This completes the handshake. This implies that in four phase handshaking, every alternate assertion on the request wire corresponds to a new handshake and a new data transfer. Figure 2.4 represents four phase handshaking.

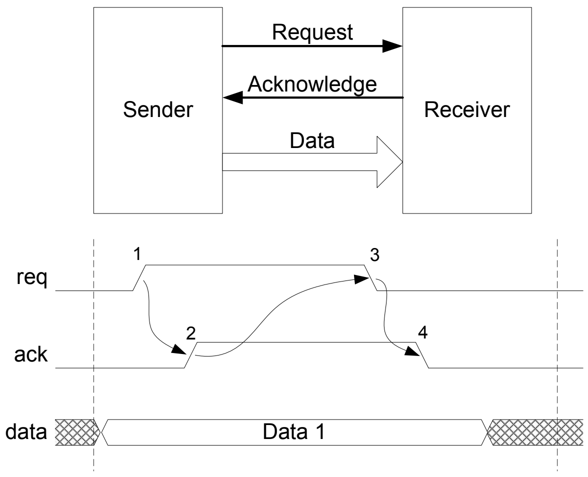


Figure 2.4: *Four-phase handshake protocol*

As an example, consider a four phase handshaking protocol employed between two modules. The transition sequence is given by: req+, ack+, req-, ack-, signifying the first handshake and data transfer. The next handshake and data transfer between these two modules will also be given by the same sequence of transitions. Four phase handshaking offers the advantage of simpler logic implementations as compared to two phase handshaking. This is because all wires are eventually reset to the same initial state. The only disadvantage is that more number of transitions as compared to two phase handshaking are required in order to complete a single communication. [10]

2.8 Data encoding protocols for asynchronous datapath

In order to communicate data between modules, some data encoding techniques are required in addition to the various control signals. There exist several data encoding styles. Single rail encoding, dual rail encoding and bundled data encoding techniques are the most commonly used styles.

2.8.1 Single rail encoding

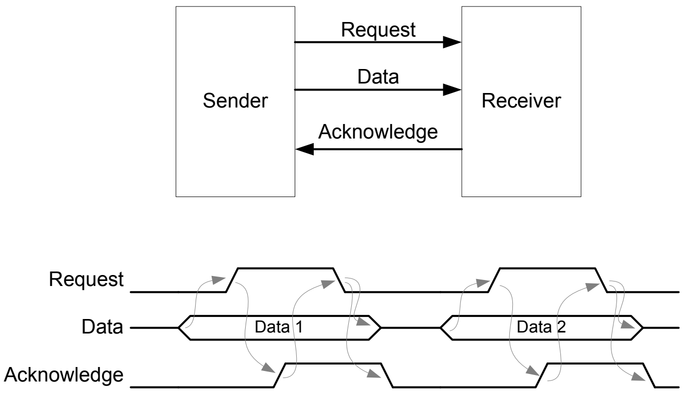


Figure 2.5: *Single rail encoding*

As indicated in Figure 2.5, single rail encoding scheme makes use of a single wire for each bit. In this encoding style, the validity of data is indicated by issuing a request signal once all the data bits are valid. However, "the request signal must reach the receiver only after all data bits are valid. This represents a local one-sided timing constraint, called a bundling constraint." [10] Besides the request, an acknowledgement is issued from the receiver to the sender to indicate that the data was received successfully.

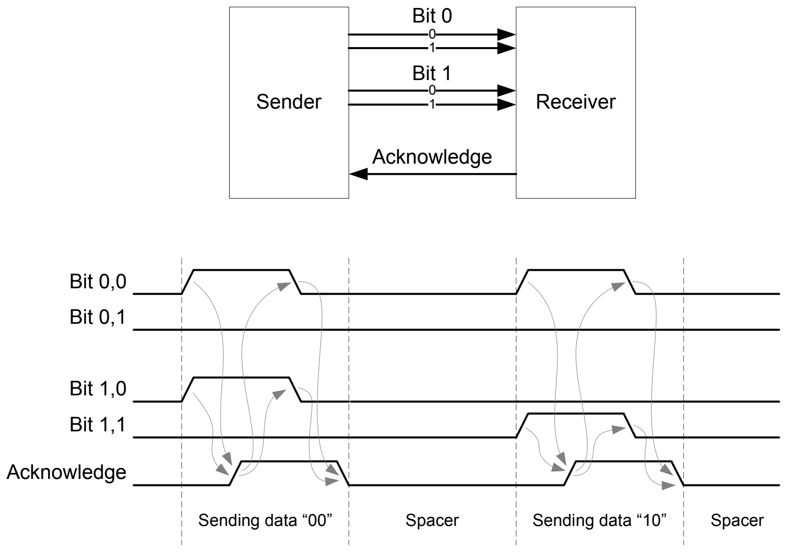


Figure 2.6: *Dual rail encoding*

2.8.2 Dual rail encoding

As indicated in Figure 2.6, dual rail encoding scheme makes use of two wires for each bit. "The code 01 on the two wires represents the data value 1 and the code 10 represents the data value 0. Code 00 is used to separate data values and called a spacer. Code 11 is not used." [10]

In this scheme, there is no separate request being sent from the sender to the receiver. However, a separate acknowledgement from the receiver to the sender is necessary. Initially, when there is no data to be transferred, all the wires display the spacer code (00) representing invalid data. Then, when the sender has valid data, the wires switch from the spacer code to display the data bits to be sent. The receiver detects the receiving of data by verifying the lack of presence of spacer code on any of the wire pair, and acknowledges this with an ack+ to he sender. On receiving an acknowledgement, the sender resets all the wire pairs to the spacer code. Once the receiver detects the resetting, it sends an ack- to the sender. [10]

2.8.3 Bundled data encoding

Bundled data encoding scheme assumes a bounded delay model. It implements single wires for each data bit and an additional control line for each data word. Figure 2.7 on the next page illustrates a bundled data encoding scheme. This scheme guarantees that the delay in the additional control wire is longer than the delay experienced in each of the data bit wires. [1] Therefore, when data is valid i.e. the "data values are set on the single wires and a transition is sent on the control wire, the receiver interprets the received transitions as the arrival of all the data bit values sent on the other wires." [11]

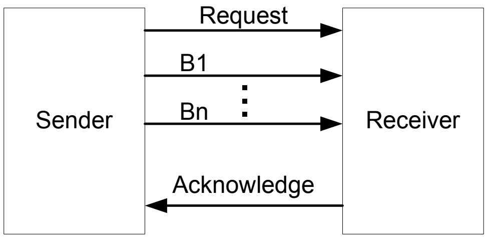


Figure 2.7: *Bundled data*
Based on a figure in [1]

2.8.4 1-of-N data encoding

A 1-of-N data encoding scheme sends log_2N bits data bits over N data wires. It is normally designed as a four phase protocol. In this scheme, the assertion of a single data wire is followed by the assertion of the acknowledgement wire. This indicates that the receiver has consumed the data and the that sender can reset the asserted data wire. Once the asserted data wire is reset by the sender, the receiver also resets the acknowledge wire, thus completing the four phase protocol. [2] Figure 2.8 illustrates a 1-of-4 data encoding scheme to encode 2 data bits over 4 data wires.

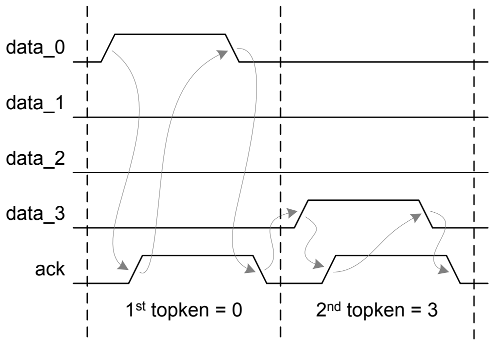


Figure 2.8: *1-of-N data encoding*
Based on a figure in [2]

2.9 Delay models

Asynchronous circuits are often classified on the basis of their behaviour with respect to the delays they experience within a circuit. This is referred to as delay modelling. The delay model primarily dictates the assumptions made about the gate delays and wire delays in an asynchronous circuit during the design process. In general, the lesser restrictions posed by these delay assumptions onto a circuit, the more robust is the design to several varying environmental factors. There can be various environmental factors subject to change over a period of time which can cause delay variations within a circuit. Some of them are: crosstalk noise, variations in the manufacturing process and unpredictable wire lengths. However, these more robust circuits (with less restrictive delay assumptions), often pay a price in terms of larger area, higher power and/or lower performance. [2] Most of the asynchronous tool and methods discussed in Chapter 5 on page 27 implement one of the first three delay models described below.

2.9.1 Delay Insensitive circuits (DI)

According to [5], Delay insensitive (DI) circuit models are extremely robust to variations and are infact the most robust amongst all asynchronous circuit delay models. They make no assumptions on the gate delays or wire delays i.e. the delays in both the gates as well as the wires are unbounded and can range anywhere between zero and infinity.

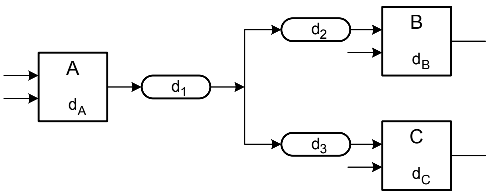


Figure 2.9: *Circuit with gate and wire delay*
Based on a figure in [3]

In order to understand the concept of delay insensitivity in circuits, consider the Figure 2.9. In this figure, d_A, d_B, d_C represent the gate delays, while d_1, d_2, d_3 represent the wire delays. Moreover, the output from the logc gate A forks to the inputs of logic gates B and C. When this circuit follows a delay insensitive model, d_A, d_B, d_C, d_1, d_2 and d_3 i.e. all the gate delays and wire delays are arbitrary. They are unknown, unbounded and are not subjected to any delay assumption or timing restriction. [3] As a result, a much less elaborate timing analysis is required to verify the correctness of DI circuits after physical design. This reduces effort and time spent on setting timing constraints and verifying their execution.

Despite of their robustness, DI circuits are limited in their practical aspects. Alain J. Martin showed in his paper [13] that no practical implementations of single output gate-level delay insensitive circuits are possible. Furthermore, [1] showed that the only practical implementation for these circuits is mainly based on the fact that all ends of a fork have to be sensed in the correct order so as to avoid any hazards.": if an autonomous DI circuit is built of single output gates, then all gates must only be C-elements, inverters, buffers or wires. Moreover, the C-element itself does not have a DI implementation built of basic gates." [12]

2.9.2 Quasi-Delay Insensitive circuits (QDI)

Alain J. Martin in his paper [14] proposed a delay model with a slight compromise to the delay insensitivity of a gate level design which was termed as Quasi-Delay Insensitive design. Quassi Delay Insensitive (QDI) circuits adopt similar assumptions to the DI circuits, but augment this with isochronic paths or forks. [1]. Like a DI circuit, all gate delays and wire delays in a QDI circuit can have unbounded arbitrary delays. However, a designated set of 'wire forks' and 'reconverging fanout combinatorial logic' both of which will lead to hazards in a circuit are excluded from the arbitrary delay assumption and are labelled as isochronic. Being labelled as isochronic, they are subjected to an additional constraint of having the same delay to the different ends of the fork. Signal transitions occur at the same time at the various end points of such wire forks. Typically, isochronic wire forks or isochronic reconverging paths are prevalent in "gate level implementations of basic building blocks where the designer can control all the wire delays." [3] For the circuit represented in Figure 2.9 on the preceding page, d_2 and d_3 are the same i.e. $d_2 = d_3$, when the circuit is implemented as a QDI model where d_2 and d_3 form a wire fork which is labelled as isochronic to avoid hazards . However, d_1 (not being part of the isochronic wire fork) principally has an unbounded and arbitrary wire delay. The gate delays i.e. d_A, d_B and d_C also have unbounded and arbitrary delay values.

"This strict and unrealistic definition is often interpreted to mean that the difference in times at which the signal arrives at the ends of an isochronic fork must be less than the minimum gate delay." [2] This slight modification in the meaning makes the definition of QDI circuits less restrictive than what was originally proposed. The main idea behind this assumption on isochronic forks for QDI circuits is to ensure the ordering of transitions of all the inputs to a logic gate, so as to guarantee a hazard free operating circuit. The concept behind QDI circuits can be better understood by considering a reconvergent circuit illustrated in Figure 2.10 on the following page. In this figure, 'F' denotes a wire fork. If this circuit is a QDI circuit adhering the isochronic fork assumption, a rising transition at 'B' is guaranteed to arrive before a falling transition at A. This behaviour ensures the presence of a glitch free output on C. [2] It can clearly be noticed that the circuit is logically designed to maintain a 1 at the output C. Let us say that the isochronic fork assumption is not adhered to and the transition from 0 to 1 on B happens to be slower than the transition from 1 to 0 on A. In this case there is a situation where A has already become 0 (being a faster path) while B is still 0 since the new transition to 1 on B has not reached it yet (being a slower path). Then, for a short interval both A and B will be 0 resulting output C to be 0 in between. This is not how the circuit was logically intended to work. Thus, it can be seen that in the absence of this assumption, an occurrence of a glich at C is possible which would make the circuit hazardous.

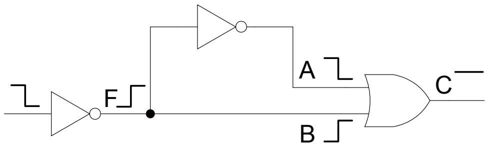


Figure 2.10: *Isochronic fork assumption*
Based on a figure in [2]

An interesting feature of circuits following a QDI model is that the primary inputs to the design are unordered. "The reason is that even if the specification indicates they are ordered, because of the unbounded wire delays to the gates that they drive the ordering is not guaranteed at these gates." [2]

2.9.3 Speed Independent circuits (SI)

A large number of circuits belong to the class of Speed Independent circits. These circuits assume all the gate delays to be unbounded and arbitrarily large and all the wire delays are ideally zero. Referring to Figure 2.9 on page 14 it implies that $d_1 = d_2 = d_3 = 0$ and d_A, d_B, d_C are arbitrary. However, practically this assumption is reduced to all gate delays being arbitrarily large and wire delays being negligible i.e. less than the minimum gate delay.

"The DI circuits are quite limited and most practical specifications do not have DI implementations. The SI circuits are usually adopted to enlarge specifications that could be synthesized." [12] However, in the real world wire delays can amount to be a significantly large fraction of the total delay. Thus, it is very difficult to guarantee the SI delay model for large circuits. SI circuits pose to be very restrictive and stricter circuits which require very elaborate timing analysis. Myers in [15] describes in detail the synthesis methods for SI control asynchronous circuits. These models need to account for almost zero or negligible wire delays and therefore need to satisfy timing constraints for many interfaces. From the above discussion it can be concluded that SI circuits might be a practical model to achieve from a synthesis perspective, but only "as long as a more detailed analysis of the underlying timing assumptions is used to verify the correctness after physical design." [2] [11] shows that the procedure for timing analysis by setting constraints is quite intensive and can get complicated when timing constraints have to be set manually to set the delay of one path with respect to the other. DI circuits or QDI circuits are therefore preferred over SI circuits for the purpose of this thesis. "The more relaxed these timing constraints are, the more likely it is that the design will work or can be modified to work." [2]

SI circuits are quite similar to QDI circuits. The zero wire delay assumption made for SI circuits is equivalent to the isochronic fork assumption made for QDI circuits. This is because the delay in the isochronic fork can be thought of as an extra delay present within the gate which is driving the fork as seen from Figure 2.11. [4] The main difference is that in a SI circuit model, the transitions on the primary inputs to the circuit may be ordered since all wires are considered to be almost instantaneous. Ordering of operations, however, does not make much sense for QDI implementations. [2]

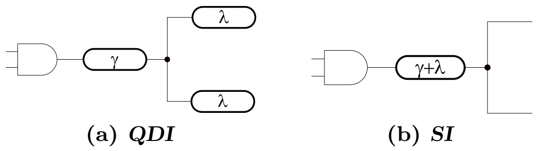


Figure 2.11: *QDI and SI assumption equivalence Based on a figure in [4]*

2.9.4 Bounded delay models

Bounded delay models assume that the delay in all circuit elements and wires is bounded or known. Huffman circuits, both in fundamental and non-fundamental mode are classified as bounded delay models.

2.10 Petri Net (PN)

Petri Nets derive their name from the person who first formally introduced Petri Nets and was called Carl Adam Petri. He described Petri Nets as a language that could be used for modelling discrete distributed concurrent systems. Petri Nets have strict semantics, a mathematical definition and a visual graphic representation approach. [12] Petri Nets evolved to be extensively used for modelling asynchronous system specifications. Several modelling and graphical representation techniques were then developed from Petri Nets. Some of the most widely used asynchronous modelling techniques besides Petri Nets such as Signal Transition graphs, Change Diagrams, Marked Graphs, Transition Systems and State Graphs, have all been derived from Petri Nets. All these approaches formulate a subset of Petri Nets.

A Petri Net comprises of a quadruple given by $N = (P, T, F, m_0)$, where

- P consists of a finite set of places denoted by p_i,
- T consists of a finite set of transitions denoted by t_i,
- F is a flow relation given by $F \subseteq (P \times T) \cup (T \times P)$
- and $m_0 \in N^{|P|}$ is the initial marking where N is a set of natural numbers.

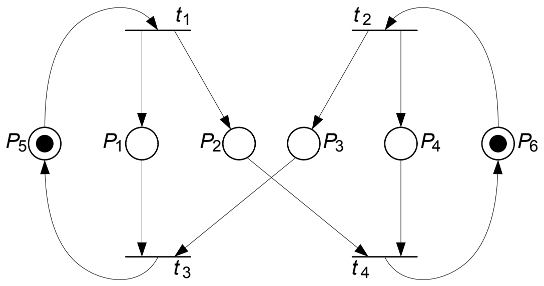


Figure 2.12: *Example of a Petri net*

Figure 2.12 is a Petri Net example taken from [2] to explain the basic concept and working of a Petri Net. As seen from the figure, a Petri Net is commonly represented in the form of a bipartite graph consisting of two types of nodes: places (p_i) denoted by circles (◯) and transitions (t_i) denoted by bars ($-$). Places and transitions are connected using directed arcs denoted by an arrow ($\rightarrow$). An arc can run from a place to a transition or from a transition to a place, but never between two places or two transitions. A marking (m) represents the current state of the system dictated by the assignment of tokens to the various places in the system. Therefore, initial marking (m_0) represents the initial state of the system. Formally, a marking is an increasing array vector of the various places in the system. The value of a place in the marking can either range anywhere between $0..N$, depending on the number of tokens in the place p at that particular marking. Graphically, the presence of a token at a particular place is indicated by a filled black dot inside the circle that denotes a place ($\odot$). [2]

"In this example,

- $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$,

- $T = \{t_1, t_2, t_3, t_4\}$,
- $F = \{(t_1, p_1), (t_1, p_2), (p_1, t_3), (p_2, t_4), (t_2, p_3), (t_2, p_4), (p_3, t_3), (p_4, t_4), (t_3, p_5), (p_5, t_1), (t_4, p_6), (p_6, t_2)\}$
- and $m_0 = \{$tokens at p_1, tokens at p_2, tokens at p_3, tokens at p_4, tokens at p_5, tokens at $p_6\} = [0, 0, 0, 0, 1, 1]$" [2]

A transition t is said to be enabled at a marking m, if there is at least one token at each place p from which an arc enters into t (called the preset of t). [2] On the other hand, the postset of t includes the places p into which arcs enter on leaving from t. On a transition being enabled, "it can fire by removing one token from each place in its preset, and adding one token to each place in its postset." [2] In the example in Figure 2.12 on the facing page, the transition t_1 can fire by removing the token in its preset i.e. p_5 and by adding it to one of the places in its postset i.e. p_1 or p_2.

2.11 Signal Transition Graph (STG)

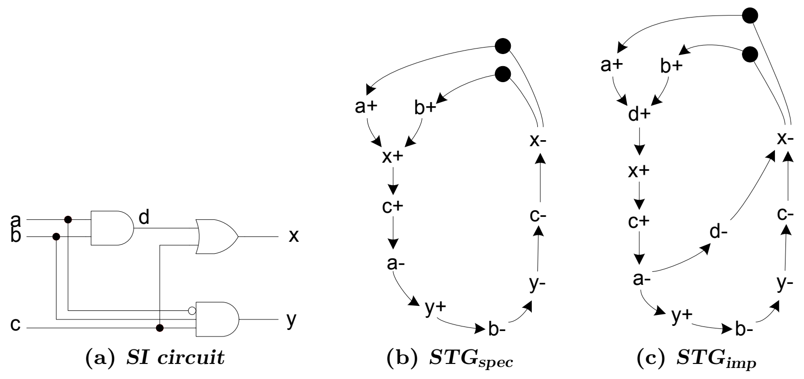
Asynchronous control circuits can be specified using an alternative approach called Signal Transition graphs. They were "first introduced by Chu in 1985" [2] as a high level description approach for asynchronous control circuits. They inherit all the semantics of a Petri Net in addition to satisfying some other requirements.

Signal Transition Graph (STG)'s are often known as interpreted Petri Nets. Every transition in a Petri Net is represented by a "rising or falling transition of an input or output signal." [2] A STG consists of a triplet given by $G = (N, A, \lambda)$ where

- N denotes the underlying Petri Net,
- A is a finite set comprising of signals and
- λ is a function which labels the transitions in N into $A \times \{+, -\}$

Figure 2.13 on the next page is a STG example taken from [12] to explain the basic concept of a STG. In this figure, $a+$ denotes a rising transition i.e. from low to high on a signal a while $a-$ denotes a falling transition i.e. high to low on a signal a, for all $a \in A$. Besides these notation, $a*$ is used to denote either $a+$ or $a-$. Moreover, multiple occurences of the same signal can be denoted using an index i such as: $a + /i$ or $a * /i$. [12]

An STG exhibits high concurrency as it operates on the input output mode where no waiting is required for the control circuit to stabilize before a new transition



(a) ***SI circuit*** (b) ***STG$_{spec}$*** (c) ***STG$_{imp}$***

Figure 2.13: *An SI circuit with its STG$_{spec}$ and STG$_{imp}$*

can take place at the input. Instead, an input transition can occur right after the occurrence of other specified input and/or output transitions. "The basic requirements for STG specifications to be implementable is that transitions on every signal alternate between rising and falling and the underlying Petri Net be safe." [2]

2.12 Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) is a formal language for modelling channel based communication in distributed concurrent systems. CSP was highly influential in the design of several programming languages used for modelling asynchronous systems, such as Tangram, Occam, Communicating Hardware Processes (CHP), etc.

The main underlying idea behind Communicating Sequential Process (CSP) is "the parallel composition of a fixed number of sequential processes communicating with each other strictly through synchronous message-passing" [16] via point to point channels. The former part implies that each process is a program comprising of statements within executing in a sequence. Each statement is separated by a semicolon (;) used to denote sequential execution. However, the processes execute in parallel with respect to each other. The parallel operator (||) denotes parallel composition of the processes. The latter part explains more about how the communication takes place between processes. In general, each process is given an explicit name, and the "source or destination of a message to be sent is defined by specifying the name of the intended sending or receiving process." [16] Furthermore, the sending action on a port is denoted by an exclamation mark (!) and the receiving action on a port is denoted by a question mark (?).

As an example, consider two processes P_1 and P_2 connected to each other through a channel C. If the value of a variable x has to be sent from P_1 to a variable y in P_2 over the channel C, the sending action from P_1 on the port C will be denoted by $C!x$ and the receiving action in P_2 on the port C will be denoted by $C?y$. [3] It is important to note that in CSP "the channel is memoryless and the transfer of the value of variable x in P_1 into variable y in P_2" [3] is an instantaneous action. This syncronizes P_1 and P_2. Basically, whichever party initiates the sending or receiving first has to wait for the other party (by temporary suspension) to complete sending or receiving on the other end of the channel. The temorary suspension causes blocking which might not always be desirable. As a work around, "Martin has extended CSP with a probe construct which allows the process at the passive end of a channel to probe whether or not a communication is pending on the channel." [3]

2.13 David cells

David Cells, first introduced by Rene David in [17], form a distributed control circuit. Quite often, they are used to mimic the token flow of an STG or a Petri Net (PN). [18] Each David Cell has an elementary two state automation. An overall system made of David Cells is thus a product of such automations. In general, David cell implementations are basic structures made up of three logic blocks and three NAND gates, as seen in Figure 2.14. However, it is possible to use other implementations as well. [19]

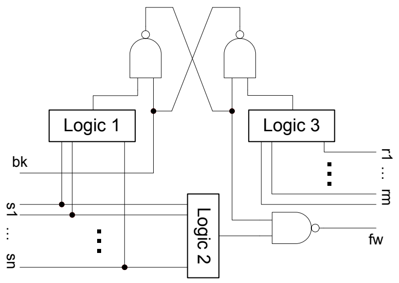


Figure 2.14: *Definition of David cells*
Based on a figure in [5]

Chapter 3

Previous Work

This chapter presents some of the most significant work done previously in the various categories of asynchronous tools/methods. In the world of asynchronous design, the ease of integration of an asynchronous design with the conventional synchronous design is primarily dictated by the delay model of the generated circuit or netlist and the input specification language. This chapter describes the most remarkable efforts made in the field of integrating an asynchronous design (using an asynchronous tool) with the existing synchronous tool chain.

[20] provides a vivid evaluation of the various available asynchronous tools and methods for designing asynchronous digital VLSI systems. Most of the asynchronous tools are classified into one of the following categories: text based specification approaches, graph based specification approaches and control-data flow graph based methods. [10] has described some of the remarkable works done in these categories. Text based specification approaches are basically syntax directed translation approaches based on CSP. One of the earliest works in this field was initiated by Martin in CHP [21]. Another popular translation based method developed by Philips is Tangram. Tangram has been successfully used to design and fabricarte several IC's, such as the asynchronous 80C51 microcontrollers [22], which have been employed in various consumer electronic products such as cell phones and pagers. [10] However, the most pioneering work in this field has been done by Balsa [23] which was developed by the University of Manchester. Balsa generated circuits are DI. [24] Balsa has been used to implement AMULET3 [25] which is a 32-bit fully asynchronous processor core compatible with clocked ARM cores. AMULET3 has been used on a commercially available chip called DRACO. Furthermore, tools such as Teak [26] and Balsa-CUBE [27] were developed as extensions to Balsa in order to improve its performance. Lard [28] and Occam [29] were some other tools developed in this category.

Graph based formalisms specify an asynchronous design graphically in the form of PNs or STGs. The most significant work in this category was done by the Universitat

Politècnica de Catalunya by developing Petrify [30]. Since then Petrify has been used actively for synthesizing several asynchronous circuits for various academic and research projects. [31] [7] [32] [33] [34] Petrify generates SI circuits. [35] Workcraft [36] is another tool in this category.

Control-data flow graph based methods split the initial design specification into a control path and a datapath and synthesize them separately. The control path is synthesized asynchronously while the datapath is synthesized using standard synchronous tools. Lavagno and Blunno made use of this technique to synthesize micro-pipelines from behavioral Verilog HDL. [9] Later, they used this approach to present a tool called Pipefitter to design asynchronous microcontrollers. [37]. Verisyn [8] was another tool based on this approach. It was developed at the University of Newcastle, and just like Pipefitter, it used Verilog as the input specification language. Both Pipefitter [38] and Verisyn [39] developers have done significant work in trying to integrate an asynchronous design with synchronous design. This is mainly because both these tools use Verilog as the input specification language and attempt to combine the netlist from the synchronous and asynchronous path towards the end. Since both these tools employ Petrify for synthesizing the asynchronous path, they lead to the generation of SI circuits. Several other works, such as [40], [41] and [42] were based on control-data flow graph based methods.

The delay model of the final netlist of an asynchronous circuit plays an important role in determining the effort spent on timing analysis and validation. The lesser the timing validation effort, the easier it is to integrate asynchronous logic with synchronous logic. [1], [2], [3], [15], [43], [12], [4] provides detailed explanations on the various delay models for asynchronous circuits. SI circuits require more timing validation as compared to DI circuits where almost no timing validation is required. [12] proposes a method to ensure the correct working of a circuit by relaxing the isochronic fork timing assumption. [4] proposes "a new combinational logic synthesis technique in which QDI/SI boolean functions can be synthesised using a small set of standard cells" [4] and also makes an analysis of the QDI implementations of circuits using Balsa.

Chapter 4
Tools and Methodology

4.1 Tools

Table 4.1 on the next page shows the different tools used in this thesis. The main tool used for asynchronous design was Balsa, developed at the University of Manchester. The Balsa tool uses the Balsa language for specifying the design. Besides Balsa, several other asynchronous tools were tested and are provided in Table 4.1. System Verilog was the HDL language used for specifying the synchronous RTL parts of the design. All testbenches were written in System Verilog. *Eclipse* was used as the main text editor. All simulations were done using Mentor Graphics *QuestaSim*. Synopsys *Design Compiler (DC)* was used to synthesize the combined (asynchronous plus synchronous) design and Synopsys *Prime Time (PT)* was used to generate the needed timing information i.e. the Standard Delay Format (SDF) file for the netlist generated by DC.

4.2 Methodology

Initially, the task of selecting the most suitable asynchronous tool was started by investigating and analysing fifteen different asynchronous design tools. Based on certain criteria or parameters, five tools were shortlisted for further investigation. Then, the integration aspects and issues of these shortlisted tools, with Nordic Semiconductor's tool chain, were examined, and one out of the five shortlisted asynchronous tools was selected. Next, an asynchronous flash readout was implemented using the selected asynchronous tool, and a Verilog netlist for both single rail and dual rail data encoding styles was generated. Then, the delay insensitivity nature of Balsa was analyzed by using a simple buffer design written in Balsa. In order to perform this analysis, a new technology matching the library used at Nordic Semiconductor was developed. Lastly, an attempt to integrate the Balsa Verilog netlist with Nordic Semiconductor's tool chain was made. The Balsa Verilog netlist generated from a buffer design was combined with a normal synchronous Verilog design and run

Table 4.1: *Tool list*

Tool		Version	Area of Application
Eclipse	[44]	3.8.1	IDE for writing the RTL code and the testbench
Mentor QuestaSim	[45]	10.2d	Simulator for RTL and netlist
Synopsys Design Compiler	[46]	J-2014.09	Compiling the RTL into netlist
Synopsys PrimeTime	[47]	J-2014.06-SP2	Timing Signoff tool. Used to generate timing information for the netlist
Balsa	[23]	4.0	Asynchronous tool
Petrify	[30]	4.2	Asynchronous tool
Workcraft	[36]	3.0.3	Asynchronous tool
Verisyn	[8]	September 2004 [a]	Asynchronous tool
Pipefitter	[48]	1.0	Asynchronous tool

[a]New version compiled for this thesis March 2015

through Synopsys DC.

Chapter 5 Asynchronous tools and methods

There are various tools and methods available for designing asynchronous digital VLSI systems. Some of them are freely available open source tools, while a few others are commercially available CAD tools for implementing asynchronous circuits. Some of them have been developed continuously, while others have either been abandoned or their development has halted a while ago. The primary objective of this chapter is to investigate and present the most interesting tools and methods that have been studied in depth for the purpose of this research, and shortlist the best suited tools based on certain criteria/parameters at the end of this investigation. The shortlisted tools from this analysis are the results obtained from this chapter, which have been presented as a part of this chapter itself. For the purpose of clarity of the reader, this was more natural than placing them in a separate results chapter. For each of the tools and methods, this chapter provides an overview of the tool, followed by it's pro's and con's, and a discussion to conclude whether the particular tool has been shortlisted or not, and based on which criteria/parameters. Furthermore, the chapter briefly skims the integration issues and possibilities of the tools with the standard commercial synchronous design tools and flow. However, a more detailed discussion about tool integration is carried out in Chapter 6 on page 59 for the tools short listed by the end of this chapter.

The criteria/parameters based on which the tools are shortlisted, can be listed and defined as follows:

– **Longevity:** Longevity can be defined as a combined measure of how actively the tool has been developed since it was created, recentness of the last paper read or found, when was the last version of the tool released and how stable is the latest version.

– **Tool complexity:** It can be defined as a combined measure of the number of new things that need to be learnt and the difficulty in understanding the background or the basic concepts the tool is based on. Optionally, it can be measured as the initial estimate of the tool complexity in the application domain. This is an optional consideration since it might be impossible to comment on this aspect of the tools before they are used.

- **Cost:** It is a measure of the cost associated with procuring the tool. Another cost involved is the cost of the resources involved, such as specific training for using the tool. However, at this stage the latter is not considered for the cost evaluation since all of the studied tools are new for the developers at Nordic Semiconductor and will require training.

- **Estimate of performance of corresponding circuits:** It is a measure of the performance estimation of the circuits developed using the tool, as stated by findings in academic or research based papers. Performance findings can be in terms of speed, power, overhead and area.

- **Commercial or Non-commercial implementations:** It is the measure of how actively, at what scale and to which extent has the tool has been used. Commercial implementations, i.e. those where chips have been taped out, are extremely favourable.

- **Delay model used by the final netlist:** It is the measure of the ease of timing analysis based on the delay model of the final netlist. The beneficial order of favour for this thesis is: Delay insensitive → Quasi-Delay Insensitive → Speed Independent.

- **Support:** It is the measure of the quickness and the quality of the support available for the tool.

- **Integration with Nordic Semiconductor's design flow:** It is the measure of the initial estimate of the ease of integrating the tools design flow with Nordic Semiconductor's design flow.

For each tool, most of these parameters have been mentioned as a part of either the pro's or the con's, depending upon whether it works be in the favour of the tool or not respectively. It is noteworthy that based on the literature study done for this thesis, an attempt has been made to comment on the various parameters for every tool. However, there are some tools for which certain parameter specifics are unknown or difficult to comment on.

"Many ways have been proposed to specify the behaviour of asynchronous circuits."[43] The specification type depends on the tool or method being used, the delay assumption and the synthesis approach or algorithm being employed. In general, most of tools/methods for asynchronous circuits can be classified into one of the five categories: text based specification approaches, graph based specification approaches, control-data flow graph based methods, HDL based methods and miscellaneous.

5.1 Tools/Methods based on text based specification approaches

Text based specification approaches use a particular formal language to encode the proposed circuit behaviour. They are very often referred to as the syntax-driven translation based approaches, wherein a circuit's behavioral description is expressed by a formal programming language such as Balsa, Tangram or CSP and is "translated using syntax-driven translation into an intermediate form based on handshake circuits, which represent abstract asynchronous blocks."[2] Text based specifications tend to produce stricter delay models such as DI, bundled data or QDI. This section essentially looks into tools/methods employing text based specification approaches.

5.1.1 Balsa

Amongst the various research tools that have been proposed to automate the process of asynchronous design, Balsa is a comprehensive tool with an open source environment. The tool was designed and is maintained by the Advanced Processor Technologies Group of the School of Computer Science at The University of Manchester. Balsa has been under continuous development since 2003-2010 with a number of improved versions over the years. The latest version of the Balsa framework i.e. Balsa 4.0 was released in June, 2010. [6] [23]

5.1.1.1 Overview

"Balsa is both a language to describe asynchronous circuits and a framework to simulate and synthesize such circuits." [49] It is used to create purely asynchronous macro modules from CSP like descriptions which closely mimic the descriptions written using the Tangram tool from Philips. In fact, Balsa was developed as a tool replacing the Philips Tangram system. The Balsa language is basically an extension of the Tangram language. "The approach adopted is that of syntax-directed compilation into communicating handshaking components and closely follows the Tangram system." [3] Syntax-directed translation is a technique by which the designers can control the resulting circuit by altering the high-level specification language. The Balsa compiler i.e. balsa-c is an integral part of the Balsa framework front end that compiles Balsa descriptions into the Breeze format.The specific design description is written in the proprietary Balsa language and is synthesized into a communicating network of handshake components similarly to the Tangram compiler. This yields a Breeze netlist. "Breeze is the target format for the Balsa compiler and is simply a netlist format for handshake circuits." [50] Once the Balsa descriptions have been translated into a network of handshaking components or the intermediate Breeze format netlist, various technologies can be employed for mapping it into a circuit. The Breeze netlist must then be translated (using the option Breeze-netlist) into a gate level netlist format supported by the backend technology planned to be used. Balsa descriptions can be synthesized using different technologies that generate Field Programmable gate array (FPGA) or silicon gate level netlist implementations which can thereafter be imported to commercially available backend CAD systems for physical implementation. Figure 5.1 shows an overview of the Balsa design flow.

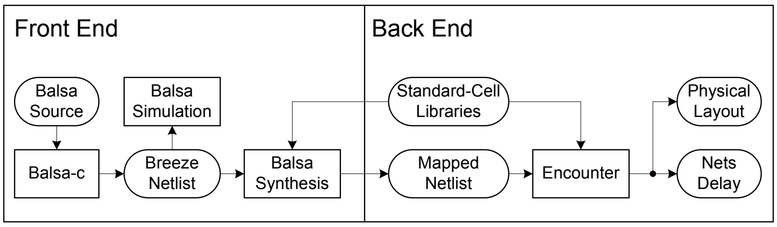


Figure 5.1: *Overview of the Balsa design flow. Based on a figure from [6]*

The direct mapping of Balsa language constructs into handshake circuits makes the compilation of circuits transparent. This makes visualization of the design specified by Balsa descriptions relatively simpler. There is a one to one mapping between the language constructs and the corresponding resulting circuit. Hence, modifications in the description language "reflect predictable changes in the resulting circuit, which means that the designer has a clear control of the generated hardware." [6]

Since Balsa supports different technologies, it can also generate netlists in different formats. It can produce a netlist in Verilog or EDIF depending on whichever is supported by the backend technology being used. Moreover, Balsa supports various asynchronous implementation design styles such as "a bundled data scheme using a four-phase-broad/reduced-broad signalling protocol, a dual-rail delay insensitive scheme and a a one-of-four encodings delay-insensitive scheme." [24] The former needs a more careful and strict post-layout timing validation checks. On the other hand the latter two are more robust to variations in the layout and are quite good to be sent for fabrication.

According to [20] and [49], balsa generated netlists fully integrate with many backend CAD systems such as ARM cell library and design rules (as used in the implementation of AMULET3), Cadence design framework and Xilinx FPGA design tools. The Balsa GUI i.e. balsa-mgr forms the GUI front end of the Balsa framework design flow. It allows for the easy compilation and simulation of the design descriptions in Balsa. The Balsa system also comprises of a package called balsa-verilog-sim "which makes Verilog simulation of Balsa descriptions easier by providing wrapper scripts for common simulators and by supporting user written built in functions which can be called from Balsa. Thus, Balsa provides an interface to several commercial Verilog simulators for functional simulation" [20]

The Balsa language provides for language operators to describe sequential and concurrent communication between handshake components. The communication between handshake components can be exchange of data or control. However, the circuit behaviour only needs to be modelled using a data-flow approach. The tool automatically generates the control circuit and the network of handshake components.[6]

Balsa makes VLSI programming more intuitive for the designer since it is very similar to familiar programming languages. It has thus been a choice for many research and commercial implementations. "Balsa has been used to generate the DMA controller used in AMULET3i, an integrated asynchronous microprocessor design for embedded systems." [51] It was developed at the University of Manchester. According to [25], AMULET3 is a 32-bit fully asynchronous processor core compatible with clocked ARM cores which has been used on a commercially available chip called DRACO. This chip is itself based on synchronous design, however, uses an asynchronous AMULET processor implemented using BALSA. Moreover, [6] has used the Balsa framework design flow to implement a pair of power efficient routers. Furthermore, several researchers have made an attempt at creating an optimized and more efficient backend for Balsa which results in circuits with greater performance. Balsa-Columbia University Back-End (Balsa - CUBE) [27] and a burst mode oriented back-end for the Balsa Synthesis System [52] are good examples demonstrating this approach. In general, this approach is based on two types of optimizations: peephole optimizations and resynthesis. Peephole optimization replaces a set of components by an optimized set of existing components. "In contrast, resynthesis methods manipulate one or more components and produce new specifications which do not correspond to existing components; these specifications are then directly synthesized." [52]

5.1.1.2 Pro's

The various advantages of Balsa are as follows:

- **Longevity:** Balsa was developed in 2003 and has been under active development until 2010.[23] Balsa version 4.0, released in 2010, is the latest available version. From the various available research papers based on the Balsa framework, it appears that Balsa has been actively used. The last research paper [6] found is quite recent and dates from 2013.

- **Cost:** Balsa is an free and open source tool available from [23].
- **Commercial or Non-commercial implementations:** Commercial implementations of Balsa exist, as with Amulet3 processor used on the commercially available DRACO chip. Besides this, several non-commercial research based implementations using the Balsa framework have also been carried out over the years.
- **Delay model used by the final netlist:** The netlist produced by Balsa is Delay Insensitive [24] which is the best suited for our requirement. A DI netlist will require lesser effort during timing analysis as compared to a QDI and SI netlist.
- **Support:** The University of Manchester has a separate webpage [23] for Balsa, with all the different versions and their corresponding bug fixes, the latest Balsa version available for download and a contact email address to put forward queries. As mentioned before, an email was sent to this contact address regarding setting up a technology for Balsa, and a response was obtained within a couple of hours.
- **Integration with Nordic Semiconductor's design flow:** As discussed earlier, the support for various implementation design styles and various backend technologies makes Balsa an extremely favourable option. Moreover, Balsa is able to generate a Verilog netlist.
- The Balsa language constructs make it possible to express sequential and parallel execution clearly and explicitly. Moreover, several other language features such as '`select`', '`case`', '`arrays`', port read/write operations and powerful parameterization options are supported by Balsa which makes design description easier for the designer. [53] Also, Balsa somewhat resembles other VLSI languages which makes it simpler to learn and understand.
- "Another advantage of using Balsa descriptions is that the Balsa synthesis system can be applied to obtain actual circuits, which makes it easier to check the specification and estimate the circuit performance." [53] This clearly implies that physical implementation of Balsa descriptions is actually possible i.e. approaches from netlist to fabrication exist.

5.1.1.3 Con's

Despite of all the advantages, Balsa suffers from certain drawbacks which are as follows:

- **Tool complexity:** Using Balsa involves learning the Balsa language, understanding the basic handshaking concept Balsa is based on and understanding the Balsa framework and design flow. However, the Balsa package includes a comprehensive tutorial.
- **Estimate of performance of corresponding circuits:** According to [54], Balsa is good for implementing moderate speed peripheral designs. However, Balsa might incur some performance overhead. To reduce this a new backend synthesis system, Balsa-Cube, has been proposed as an extension to Balsa. [27] [52] However, since Balsa-Cube is an extension to Balsa, it will not be tested in this thesis in case Balsa is selected.
- "Timing validation of Balsa generated implementations is not yet implemented as part of the Balsa design flow. This form of validation is currently undertaken through simulation although the use of existing synchronous static timing analysis tools is to be investigated as part of further development of Balsa." [54]

5.1.1.4 Balsa discussion

Based on Balsa's pro's and con's, Balsa is one amongst the selected shortlisted tools mainly because the possibility of integrating Balsa with Nordic Semiconductor's tool flow looks promising. The other factors contributing to Balsa's selection are longevity, commercial implementations, delay model of the final netlist and support.

5.1.2 Tangram

Tangram was developed by Handshake Solutions, Philips and was mostly meant to be used within the organization for application specific asynchronous design. Tangram is quite similar to Balsa in many of its features and the way it functions. As mentioned before, Balsa was basically an open source and improved version of Tangram. Tangram was inspired by CSP. It is based on an intermediate handshaking circuit representation.

5.1.2.1 Overview

Tangram supports all the imperative program and language constructs such as assignments, parallelism, sequencing, loops, variables, channels and their communication, iteration and conditional operators, etc. [20] It provides more flexibility in dealing with external interfaces which are non-delay insensitive. Moreover, the Tangram package contains a useful power performance analyser tool to analyze the power-performance trade off of the implemented design. [3] Philips mostly used Tangram in applications where performance was of lesser importance than power consumption, simplicity of integration and EMI. "Current applications include several products in the wireless communication area, smart cards, and in-vehicle networks for automotive." [20] The Tangram design flow has been used to obtain several IC's that are available on the market. The Tangram design flow integrates well with many standard commercial tools. Many commercial synchronous tools are used in the backend of the design flow. Tangram can be used to design either two-phase or four-phase implementations.

5.1.2.2 Pro's

The various advantages of Tangram are as follows:

- **Commercial or Non-commercial implementations:** Tangram has been used by Philips for successful IC deployment being used for several applications.
- **Integration with Nordic Semiconductor's design flow:** The Tangram design flow integrates well with many standard commercial tools. Tangram aims at minimizing "the development of new dedicated tools, and to re-use common synchronous tools where-ever possible." [20]
- Just like Balsa, Tangram is a highly transparent language.
- The Tangram language constructs make it possible to express sequential and parallel execution clearly and explicitly. Moreover, support for several other language features is provided by Tangram which makes design description easier for the designer.

5.1.2.3 Con's

Tangram suffers from certain drawbacks which are as follows:

- **Longevity:** Tangram was replaced by Balsa, see Section 5.1.1 on page 29, and has not been developed or patch fixed in a decade.
- **Tool complexity:** Since Balsa is an improved version of Tangram, it is possible to say that the tool complexity of Tangram is more or less similar to Balsa. Using Tangram involves learning the Tangram language, understanding the basic handshaking concept Tangram is based on and understanding the Tangram framework and design flow.
- **Cost:** Tangram is not freely available but the cost is unknown. It forms a closed framework mostly developed to be used inside Philips.
- **Estimate of performance of corresponding circuits:** According to [43], Tangram can produce inefficient circuits due to its focus on composability and handshaking.

- **Delay model used by the final netlist:** The delay model of the final netlist is not specified in the papers read. Hence, it is unknown.

- **Support:** Tangram was developed by Philips. Hence, for further information on the tool Philips can be contacted. Since Tangram was mostly developed for use within Philips, it might not offer good support for external environments. However, since Tangram was not contacted, the quality of support available for Tangram is unknown.

5.1.2.4 Tangram discussion

Tangram was eliminated and was not looked into any further. This was mainly because Balsa was a freely available and improved version of Tangram that was more recently developed and used. Hence, it was chosen over Tangram. Moreover, Tangram lacked on the longevity front. Also, many parameters such as the used delay model, cost, support and performance for the tool were unknown and this was a hindrance to the goal.

5.1.3 Teak

Teak is a second open-source backend tool system developed at the University of Manchester for synthesizing Balsa descriptions. It is a tool that creates asynchronous implementations of circuits described using the Balsa language.

5.1.3.1 Overview

Just like Balsa, Teak produces an intermediate representation which is basically a Teak network composed of Teak components which are connected to each other using handshaking channels. The Teak components are nothing but a set of new parameterizable components different from the Balsa components and belong to the target library used by the Teak synthesizer for synthesis. The Teak system differs from the traditional Balsa system design flow by employing a synthesis scheme which leads to improvement in the cost and performance of the circuits described using Balsa. "The tool optimizes Balsa descriptions synthesis by replacing data-less activation channels with separate control channels." [55] "This new scheme removes the reliance on precise handshake interleaving and enclosure by separating out control ‘go’ and ‘done’ signalling into separate channels rather than using different phases of the asynchronous handshake." [56] This leads to optimized circuits in which the control and data channels are separated/merged by the insertion of handshake-decoupling latches and data buffering. Teak has been used to create elastic pipelined asynchronous systems from Balsa descriptions. "The teak tool currently consists of:

- A synthesiser from Balsa to Teak component networks

– A mechanism to plot those networks

– A language-level simulator for Balsa

– A programmable peephole optimiser for component networks

– A GUI to drive and visualise optimisation choices

– A prototype 'back end' to generate Verilog gate-level implementations of Teak components " [26]

5.1.3.2 Pro's

The advantages of Teak are as follows:

– **Cost:** Just like Balsa, Teak is a free and open source tool available from [26].

– **Estimate of performance of corresponding circuits:** Using the Teak backend system for Balsa descriptions leads to performance improvement, cost improvement and mitigation of control overhead.

5.1.3.3 Con's

The disadvantages of Teak are as follows:

– **Longevity:** Teak is another tool (besides Balsa) which has been developed at the University of Manchester. Even though it is a much newer tool compared to Balsa, the last research paper [56] found dates from 2009. It seems like Teak is much less actively developed than Balsa. Moreover, no improved versions of teak has been available since 2009. There is only one available version for Teak since it was produced i.e. version 0.4.

– **Tool complexity:** Teak is basically developed on top of Balsa. It uses Balsa descriptions as a starting point. The only difference is that Teak provides a better backend synthesis system which was an improvement to the existing Balsa tool for achieving better performance. Hence, the tool complexity of Teak is very similar to Balsa. The main factors contributing to the tool complexity here are learning the Balsa language and understanding the more advanced backend synthesis system.

– **Commercial or Non-commercial implementations:** Unlike Balsa, there are no commercial implementations using Teak. However, there are a few academic research based implementations available.

– **Delay model used by the final netlist:** Teak implementations are usually limited to QDI four-phase dual rail asynchronous circuits.

- **Support:** Just like Balsa, the University of Manchester has a separate webpage [26] for Teak, with version 0.4 available for download and a contact email address to put forward queries. However, this tool was not further investigated. Therefore, the support available for Teak is unknown.

- **Integration with Nordic Semiconductor's design flow:** Teak limits the choice of the data encoding styles and the protocols used. Delay insensitive circuits cannot be obtained using a Teak synthesis system.

5.1.3.4 Teak discussion

Teak was eliminated and was not looked into any further. Even though it theoretically states to provide performance benefits over Balsa, there are not any known commercial implementations using Teak that have a better performance over Balsa implementations. Therefore, Balsa seems to be a safer option. The other main parameters on the basis of which Teak was eliminated from further investigation were longevity, tool complexity, integration with Nordic Semiconductor's design flow, and the used delay model.

5.1.4 Communicating Hardware Processes

Communicating Hardware Processes (CHP) was proposed by the Caltech University and is based on a subset of CSP.

5.1.4.1 Overview

CHP comprises of a number of parallelly operating processes that are connected to each other by channels. All the data transfer and synchronization takes place over these channels. "In particular, it is an almost ideal formalism for the decomposition of a large sequential code into a collection of fine-grain processes." [57] In comparison to CSP, CHP has certain restrictions and extensions. The restrictions involve the inability to create resources dynamically, inability to create physical resources during program execution and support for finite data ranges and boolean data types only. The extensions are mostly related to efficient communication and the ability to describe complex process structures. [43] [57] According to [57], CHP has been used in several asynchronous VLSI design projects at the Caltech University.

5.1.4.2 Pro's

The main advantages of CHP are as follows:

- **Estimate of performance of corresponding circuits:** According to [43], CHP is good for the production of faster and smaller circuits.

- **Commercial or Non-commercial implementations:** CHP has been used for some commercial implementations earlier such as the asynchronous Caltech MiniMIPS microprocessor in 1998. [57] Moreover, according to [57], CHP has been used in several asynchronous VLSI design projects at the Caltech University. It is used in the industry by companies such as France Telecom, ST and CEA/LETI.

5.1.4.3 Con's

The disadvantages of CHP are as follows:

- **Longevity:** CHP was introduced at least 2 decades ago. The exact year is unknown. Even though the last research paper [57] found, dated 2012, mentions that CHP has gone through many modifications since it was created, no records of latest development or fixes on the tool were found. It discusses CHP as a language, the various CHP constructs and the CHP simulator. However, no recent academic research papers using CHP for practical implementations were found. Moreover, the last commercial implementation using CHP is old and dates back to 1998. Furthermore, several newer languages such as Balsa and Tangram were developed using the same principle as CHP. Hence, it seems like CHP has not been employed or developed in a long time.

- **Tool complexity:** Using CHP involves learning the CHP language and understanding the process decomposition in CHP. CHP appears to be a complicated language with complex formalisms. There are a lot of inadequacies and restrictions in the language.

- Many important parameters for CHP, such as, cost, delay model used by the final netlist, integration with Nordic Semiconductor's design flow and support are unknown.

5.1.4.4 CHP discussion

CHP was eliminated and was not looked into any further. It was mainly because of the tool complexity and so many unknown parameters. Moreover, since CHP had not been developed or used in a while, newer tools like Balsa were considered as a better option.

5.1.5 Occam

Occam is a concurrent hardware description language which was developed at Inmos Limited in Great Britain. It is a parallel programming language based on CSP.

5.1.5.1 Overview

"Occam is an executable programming language with well defined syntax and semantics." [29] It is commercially supported and is widely used. A large range of hardware platforms provide support for Occam. It is a practical realization of CSP and allows explicit description of sequential and parallel processes. Occam can be used for the purpose of distributed simulation.

One of the attractive features of Occam is its simplicity. The support for parallel processes avoids the need for complicated shared variables. Moreover, concurrency can be expressed explicitly at the statement level using Occam. Most programming languages supporting parallel processes can only be used to express concurrency implicitly at the procedure level. Occam also provides security by not including a lot of common language features such as pointers, recursive functions and dynamic process and member allocation. [58]

5.1.5.2 Pro's

The main advantages of Occam are as follows:

- **Estimate of performance of corresponding circuits:** Owing to its parallel nature, "Occam may be executed multiprocessor systems and thus has the potential for high performance." [29] The parallel, concurrent and distributed nature of Occam serves well for asynchronous design where a global state ceases to exist. Moreover, a parallel approach to design and simulation can significantly reduce the cost and duration of the design cycle that brings about performance improvements.

- **Commercial or Non-commercial implementations:** "From the Occam model, Inmos developed a hardware chip to support their concurrency model. This hardware is in the form of a Very Large Scale Integration (VLSI) Integrated Circuit (IC) called the Transputer." [58]

- The level at which Occam describes asynchronous control circuits is quite close to their implementation, "consequently it may provide guidance for the realization of the design (e.g. an IF statement will correspond to a Select block, a PAR of input commands will be implemented using a Muller-C block etc)." [29] As a result of this characteristic, Occam specifications can also be use for the automatic derivation of asynchronous circuits.

5.1.5.3 Con's

The disadvantages of Occam are as follows:

- **Longevity:** Occam was developed by David May in 1983. [58] The last available version is Occam 2 which dates back to 1986 and the last research paper [29] found dates from 1997. Hence, it is very clear that Occam has not been used or developed since a very long time.
- **Tool complexity:** Being a new language, time has to be spent learning Occam. It is not a very advanced language to describe hardware. It lacks many useful data structures, processes and protocols that are extremely useful for the description and simulation of hardware. Moreover, "rigid and verbose layout format and the semantic significance of indentation which makes both the development and debugging of programs time consuming and frustrating tasks." [29]
- Many important parameters for CHP, such as, cost, delay model used by the final netlist, integration with Nordic Semiconductor's design flow and support are unknown.

5.1.5.4 Occam discussion

Occam was eliminated and was not looked into any further. It was mainly because of the longevity, tool complexity and so many unknown parameters. Moreover, the only known commercial implementation using Occam i.e. Transputer is very old and dates back to the 1980's.

5.1.6 LARD

Language for Asynchronous Research and Development (LARD) is a hardware description language used for describing the behaviour of asynchronous VLSI systems. It was developed at the University of Manchester and uses CSP-like channel based communication. It is a freely available tool.

5.1.6.1 Overview

"In asynchronous systems, timing information about the validity of data signals is provided by local timing signals, often in the form of request and acknowledge signals." [28] LARD provides communication abstraction by using send-receive channel based communication and hiding some complex details. Moreover, LARD provides fine grained concurrency by allowing for statements in the description to be composed concurrently as well as sequentially. LARD has a comprehensive system providing facilities expected for structured programming. [59]

Furthermore, simulations can be performed using the LARD toolkit. They are useful for performance analysis, debugging and for the validation of the behavioural model of a design against its corresponding gate level netlist or schematic. The

LARD toolkit has been implemented in an extremely flexible fashion. This makes expansion of functionality and the user interface simple. This flexibility also provides for easy adaptation and expansion to other tasks and application domains. LARD has been used for the behavioral modelling of AMULET3 and has shown to have significant performance improvements. [28]

5.1.6.2 Pro's

The advantages of LARD are as follows:

- **Cost:** LARD is available free of cost.
- **Estimate of performance of corresponding circuits:** According to [28], LARD has shown significant performance improvements in the behavioural modelling of AMULET3.
- **Commercial or Non-commercial implementations:** As stated above, LARD has been used for the behavioural modelling of AMULET3.
- The other advantages of LARD include flexibility, support for concurrency, extensive data types and structured language.

5.1.6.3 Con's

The disadvantages of LARD are as follows:

- **Longevity:** LARD was basically developed to provide a more advanced modelling language and simulation environment for AMULET3 as compared to Balsa. The last academic research paper [28] found dates from 1998. It explains the main features of LARD and the simulation toolkit it provides. However, no academic papers exhibiting LARD implementations were found. Moreover, the link to the LARD webpage provided in [59] is no longer active. It seems like LARD has not been used or developed after it was used for the behavioural modelling of AMULET 3.
- **Tool complexity:** Despite of a simple user interface, time has to be spent on learning the new LARD language, understanding the LARD design flow and understanding the important features of the LARD toolkit and the simulation environment it provides.
- **Support:** Since the link to the LARD webpage provided in [59] is no longer active, it seems like the University of Manchester does not offer much support for LARD anymore.

– It is difficult to comment on the delay model used by the final netlist and the integration with Nordic Semiconductor's design flow, since it was hard to find information pointing towards these parameters.

5.1.6.4 LARD discussion

LARD was eliminated and was not looked into any further. It was mainly because of the longevity and support. Moreover, the ambiguity in the delay model and integration aspect of LARD contributed to its rejection. LARD was developed as an alternative to Balsa. At this stage, Balsa was considered to be a more simple and stable option.

5.2 Tools/Methods based on graph based specification approaches

Graph based specification approaches describe the behaviour of an asynchronous system graphically in the form of partially ordered sequence of events. They are often used at a low conceptual level and consist of states and transitions. Their semantics are "defined using additional entities, e.g. tokens or node/arc states, which in turn form the overall state of the system." [60] Owing to their complexity, they can be a pain for the designer but they produce faster and more efficient synthesized circuits. These specifications are often based on formalisms such as Petri Nets (PN), Signal Transition Graphs (STG), State Graphs (SG) or Transition Systems (TS). This section essentially looks into tools/methods employing graph based specification approaches.

5.2.1 Petrify

Petrify is one of the most popular academic research tools for the synthesis of asynchronous circuits. The tool uses graphical based specifications in the form of PN, STG, SG or TS as an input for synthesis. It is freely available on the web. Petri Net Markup Language (PNML) [61] is a standard format for the specification of Petri Nets.

5.2.1.1 Overview

Petrify is a well known tool for the manipulation of concurrent specifications and for synthesizing and optimizing asynchronous control circuits. "Given a Petri Net (PN), a Signal Transition Graph (STG), or a Transition System (TS) it generates another PN or STG which is simpler than the original description and produces an optimized net-list of an asynchronous controller in the target gate library while preserving the specified input-output behavior." [7] In short, from a given specification as input, Petrify outputs an optimized netlist of the asynchronous circuit and a simplified PN showing the events and the various transitions between them. The latter provides the ability to back-annotate to the original specification.

Petrify transforms a specification by performing a token flow analysis of the original PN and thereby producing a TS. Initially, all the transitions having the same label are marked as one event. Then, the transitions are relabelled for fulfilling the requirements in order to obtain a safe irredundant PN and the TS is transformed. As a part of synthesis, Petrify solves various logic synthesis problems such as logic decomposition, state encoding and technology mapping onto a gate library in order to generate a netlist. The final netlist is a speed independent circuit which means that the circuit is guaranteed to be hazard free irrespective of the gate delays and changes in multiple inputs which satisfy the initial specification. Figure 5.2 shows the Petrify framework. "Petrify can also synthesize circuit under timing assumptions specified by the designer or automatically generated by the tool" [62] Petrify is a well equipped tool that can be used for PN composition, PN synthesis and re-synthesis of asynchronous control circuits and other related areas dealing with concurrent asynchronous specifications or programs. [7]

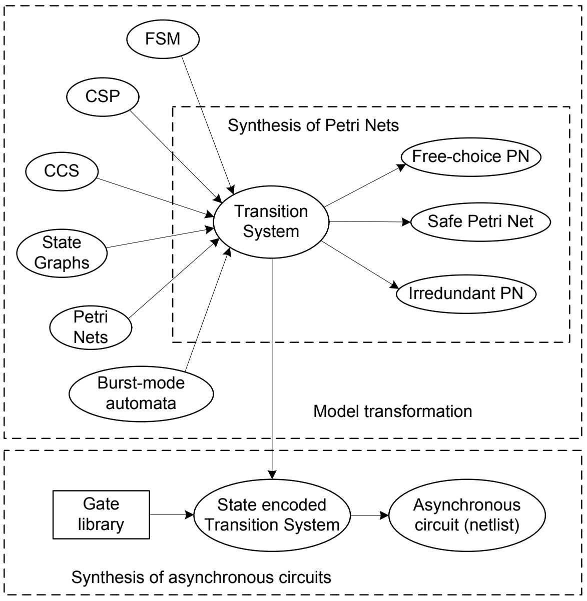


Figure 5.2: *Block diagram of the Petrify synthesis framework. Based on a figure in [7]*

Petrify implements a method in which a safe PN with a reachability graph similar to the original PN or TS is synthesized from an originally given PN or TS. The new

PN formed is nothing but a minimized version of the original PN. The reachability graph is said to be isomorphic to either the original PN or the minimized PN. The synthesized PN exhibits place irredundancy, which means that removing a place from the net would change its behaviour. [7]

Petrify can also be used to generate netlists in Verilog, EQN or BLIF. The process from an initial PN to the generation of a circuit netlist is completely automated. Petrify has been used for the synthesis of asynchronous controllers such as AMULET microprocessor, circuits from RAPPID by Intel Corporation and controllers based on theseus logic [20]. [32] describes a reverse engineering methodology of synthesizing PNs from state-based models such as Finite State Machine (FSM) or TS. [33] presents a method for the automated synthesis of asynchronous circuits from specifications based on process algebra. It combines PNs and process algebra for the specification and synthesis of the given asynchronous circuit.

5.2.1.2 Pro’s

Petrify has the following advantages:

- **Longevity:** Petrify was developed in the the late 90’s. The latest version of Petrify available on the Petrify webpage is from 1999. There is no improved or upgraded version since then. Even though, it seems like Petrify is not actively developed, the 1999 version of Petrify seems to be extremely stable since it has been the most used tool for academic research on asynchronous design. The last academic research paper [63] and book [64] found for Petrify are from 2000. However, the latest version of PNML available is from 2009 [61].
- **Cost:** Petrify is freely available from the Petrify webpage [30].
- **Commercial or Non-commercial implementations:** Even though there are no commercial implementations of Petrify, it has been the most widely used tool for academic projects in the field of asynchronous design. Petrify has been used for the synthesis of asynchronous control circuits in several projects.
- **Integration with Nordic Semiconductor’s design flow:** Petrify can generate a netlist in Verilog which makes integration with Nordic Semiconductor’s tool chain more feasible since Verilog is the HDL being used in the Nordic Semiconductor’s design flow.
- Petrify exhibits a property of back-annotation which helps the designer to execute more control over the design process.

5.2.1.3 Con’s

Petrify has the following disadvantages:

- **Tool complexity:** Specification of the design in PN format is cumbersome. An in-depth understanding of PNs and PNML i.e. the language to describe PNs is required for this purpose. Understanding PNs can be time consuming and complicated.
- **Estimate of performance of corresponding circuits:** State space explosion can be a problem with Petrify for STGs involving many variables which might consume hours of CPU time. Even though Petrify is a powerful tool for logic synthesis, a more realistic approach to where it can be used efficiently is required while using it. [18]
- **Delay model used by the final netlist:** Petrify generates a speed-independent netlist. A speed independent netlist requires timing analysis for isochronic forks. This increases the timing validation effort.
- **Support:** Unknown
- For rapid implementation and synthesis using Petrify, a way has to be figured out for the conversion from standard HDL (Verilog) to PN or STG which is yet unknown. [64] has made an attempt to describe a technique doing so. However, as of now there is no specific known method or standard to convert Verilog to PN or STG.

5.2.1.4 Petrify discussion

Petrify is one amongst the selected shortlisted tools mainly because of its wide use and acknowledgement for academic research based projects in the field of asynchronous design. Moreover, some papers suggest that Petrify can generate a Verilog netlist, which was ideal for the thesis. Hence, trying out Petrify for further investigation was thought to be a good idea.

5.2.2 DESI

DESI is an acronym for Decomposer Signal Transition Graph. It is a tool designed for free use with academic tools such as Petrify, CASCADE and 3D. DESI is mostly used for the synthesis of asynchronous circuits with complex STG specifications. It mainly revolves around the principle of STG decomposition. It involves the decomposition of an initial Signal Transition Graph (N) into many small STG components (C_i). Each C_i is then synthesized into a separate module. Finally, all the modules are composed together to reach a modular circuit. [20]

5.2.2.1 Overview

DESI is a tool that aims at decomposing a complex STG into smaller STG components to ease the synthesis problem at hand by avoiding the state explosion problem. When an initial specification STG (N) is provided as an input to a tool like Petrify for logic synthesis, a reachability graph for the circuit at hand is constructed by the tool. However, owing to the complexity of circuits mimicked by their corresponding STG, a problem of state explosion might occur during the construction of the reachability graph. The number of reachable states (r) might become extremely large to be handled as a result of very long CPU times or insufficient storage space. In order to avoid the state explosion problem, decomposing the initial STG (N) into STG components called C_i is a viable solution. Each circuit component is a separate module. Hence, a circuit is decomposed into several module components using this approach. To a great extent, this helps to solve the problem, since "the reachability graphs of the C_i, taken together, can be much smaller than r since r might be the product of their sizes." [20] The approach adopted by DESI, significantly reduces design effort and saves circuit area. [20]

"Decomposition can also be useful aside from size considerations: there are examples where N cannot be handled by a certain synthesis method, while its C_i can (e.g. deriving a set of XBM machines from an STG)" [20] The algorithm for DESI performs decomposition automatically. However, the output partitions and the output file types need to be specified manually.

5.2.2.2 Pro's

DESI has the following advantages:

- **Cost:** DESI is free for academic use. [20]
- **Estimate of performance of corresponding circuits:** DESI avoids the problem of state explosion. It helps to save the circuit area.
- It helps in reducing design effort.
- It makes extraction of library elements easier.

5.2.2.3 Con's

Almost all disadvantages of DESI are associated to the requirements that need to be fulfilled by the initial STG to be decomposed. They are as follows:

- **Tool complexity:** DESI is meant to be used with some other tool such as Petrify. Hence, all the parameters adding to the complexity of Petrify are applicable for DESI as well. An understanding of STGs and their decomposition is required. Some things that need to be ensured are as follows:
  1. The initial STG should be free from any internal transitions.

2. There should be no structural conflicts or conflicts between the inputs and outputs.
3. There should be no automatically generated concurrency.

5.2.2.4 DESI discussion

DESI was eliminated and was not looked into any further. This is because DESI is a tool to be used in combination with other tools. It is not a self equipped tool that needs to be investigated separately.

5.2.3 VSTGL

VSTGL is an acronym for Visual STG Lab. It is a public domain tool that was developed at the Technical University of Denmark (DTU). It is a graphical tool used to create, capture and simulate Signal Transition Graphs.

5.2.3.1 Overview

STG's are very often used to describe asynchronous control circuits which are then synthesized using tools such as Petrify which makes of a textual description of the STG as an input format. "Visual STG Lab (VSTGL) is a graphics editor and test environment for creating STGs, and it can be used as a front-end to Petrify." [20] Using VSTGL over the normal Petrify design flow, results in a design flow which is less error prone and much faster. [20]

5.2.3.2 Pro's

VSTGL has the following advantages:

- **Cost:** It is a public domain tool and hence freely available. [20]
- VSTGL provides a platform for the graphical entry of a STG and checking its structural properties. [20]
- It allows for the simulation of the STG prior to synthesis.
- "VSTGL outputs the STG for documentation (.eps file) exactly as the designer entered it. This sounds trivial, but as the designer normally expresses key ideas about the design in the topological structure of the STG this is important. Petrify uses the program dot, and normally it produces drawings that bear little resemblance with what the designer had in mind." [20]
- It can generate an input file for Petrify.

5.2.3.3 Con's

VSTGL has the following disadvantages:

- **Longevity:** VSTGL was developed in the late 90's. It has not been used much ever since.
- **Tool complexity:** The tool complexity is mostly related to understanding how the tool functions and learning how to make an STG entry.

5.2.3.4 VSTGL discussion

VSTGL was eliminated and was not looked into any further. This is because it provides a front end to Petrify. Since, it is used in combination with another tool like Petrify, it does not need to be investigated separately.

5.2.4 Workcraft

Workcraft is a tool kit that provides a common and flexible framework for the development of graph based specification techniques/models. It provides as an excellent front-end as well as back-end system for creating, visual editing, simulation, synthesis and timing analysis of graph based models. It is a platform independent tool which is freely available for academic purpose or usage. Workcraft can be highly customized by the designer with the addition of plug-ins.

5.2.4.1 Overview

Workcraft is a tool set for capturing, simulating, verifying and synthesizing interpreted graph models. Interpreted graph models can be defined as specification that have an underlying static graph structure, such as PNs, STGs, etc. "Workcraft provides a cross-platform front-end to established synthesis and verification back-end tools. It sets a plugin-based framework for new graph formalisms and their analysis tools." [36] This can either be performed directly or by mapping a model into a different type of behaviourally equivalent model (generally a PN). "Hence the user can design a system using the most appropriate formalism (or even different formalisms for the subsystems), while still utilising the power of PN analysis techniques." [60] Workcraft is being actively developed as an important project at the Newcastle University.

5.2.4.2 Pro's

Workcraft has the following advantages:

- **Longevity:** Workcraft is a recent toolset being actively developed. Several versions of Workcraft (for both Windows and Linux based systems) have been introduced since the first one. It has been under development until recently in 2015. The website [36] was last updated in 2015.

- **Cost:** Workcraft is available for free download on the Workcraft website [36].
- **Support:** Workcraft provides an excellent support system. An email was sent to the Workcraft developers and a quick response was received.
- Workcraft provides a complete system from designing to synthesizing different graph based models.
- It provides a very user friendly and intuitive front-end interface.
- It allows for the simulation of the graph based models prior to synthesis.
- It can generate an input file for Petrify.

5.2.4.3 Con's

Workcraft has the following disadvantages:

- **Tool complexity:** Workcraft provides a front end to well established backend tools such a Petrify. Hence, all the parameters adding to the complexity of Petrify are applicable for Workcraft as well. Moreover, the user needs to learn how to use the toolset and the various functions available in the toolset.
- **Delay model used by the final netlist:** If Workcraft uses Petrify as the backend tool, it will generate a speed independent netlist.

5.2.4.4 Workcraft discussion

Since, Workcraft uses Petrify as a backend tool, it has not been investigated separately. It was looked into with Petrify, as a front end to Petrify for designing STGs.

5.3 Control-data flow graph based methods

These approaches split the initial design specification into a control path and a datapath and synthesize them separately. The control path is synthesized asynchronously while the datapath is synthesized using standard synchronous tools.

5.3.1 Verisyn

Verisyn is a tool developed at the University of Newcastle which operates as a front end to a behavioural synthesis system. It employs a global direct mapping approach at all levels of the synthesis flow for specifications in commercial input language such as Verilog. [8]

5.3.1.1 Overview

Verisyn uses high level behavioural specifications written in Verilog as an input, compiles it, optimizes and schedules it and then converts it into an intermediate PN format. The intermediate format is subsequently synthesized by optimization and mapping tools into control circuits and datapath circuits using direct mapping techniques like David Cells (DCs). This leads to the generation of Speed Independent (SI) circuits. [65]

Figure 5.3 shows the synthesis flow adopted by Verisyn. The intermediate PN format represents a multi-Petri Net format which comprises of two types of nets: control nets which are based on Labelled Petri Nets (LPNs) and datapath nets which are based on Coloured Petri Nets (CPNs). For the purpose of mapping, control nets are further split into local control nets which are used for mapping to simple control gates and global control nets for direct mapping to DCs. [65] [39]

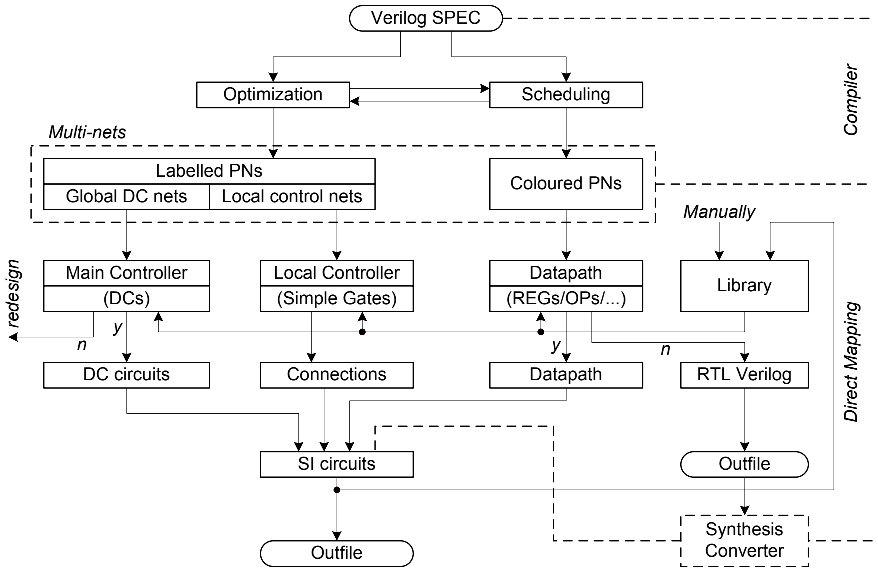


Figure 5.3: *Block diagram of Verisyn synthesis flow. Based on a figure in [8]*

5.3.1.2 Pro's

Verisyn has the following advantages:

– **Cost:** Verisyn is freely available from the Verisyn webpage [8].

- **Estimate of performance of corresponding circuits:** Verisyn adopts a direct mapping approach by employing "handshake specifications which describe the control and datapath together." [8] This technique helps to avoid state explosion problems prevalent in tools such as Petrify.

- **Support:** Verisyn offers excellent support. The developers of Verisyn at the University of Newcastle were contacted and a quick reply was received.

- **Integration with Nordic Semiconductor's design flow:** Verisyn uses a standard commercial HDL such as Verilog as an input language which makes it easier to integrate with Nordic Semiconductor's design flow. Moreover, it avoids the need to learn new specification languages like PNML as it uses Verilog to automatically generate an intermediate PN format.

5.3.1.3 Con's

Verisyn has the following disadvantages:

- **Longevity:** Verisyn was developed in 2004. The last research paper [65] found for Verisyn dates from 2004. Verisyn was not developed ever since.

- **Tool complexity:** The complexity of Verisyn lies in understanding the synthesis flow and net generation process, which can be tricky and time consuming.

- **Commercial or Non-commercial implementations:** No commercial or non-commercial implementations were found.

- **Delay model used by the final netlist:** Verisyn generates SI circuits, which is an advantage to a great extent since it produces hazard free circuits under distributed gate delays. However, SI circuits still need to be accounted for wire delays. Verification of timing assumptions for isochronic forks needs to be taken care of. In contrast to SI implementations, Delay Insensitive (DI) implementations serve better since they produce hazard free circuits under any gate delay and wire delay distribution. Therefore, a more careful timing analysis is required for SI circuits as compared to DI circuits.

5.3.1.4 Verisyn discussion

Verisyn is one amongst the selected shortlisted tools mainly because of better integration possibilities with Nordic Semiconductor's design flow over Petrify. Verisyn was based on PNs (like Petrify), but offered many advantages over Petrify in terms of performance benefits, the possibility to use Verilog as input and much better support. Hence, it was decided to try Verisyn for further investigation.

5.3.2 Pipefitter

Pipefitter was developed by the Microelectronics Group at Politecnico di Torino (Italy). [38] It is a "tool chain that implements a fully automated synthesis flow for asynchronous circuits" [37] and "can be integrated inside any standard design flow in order to implement a fully asynchronous design flow." [38] It has been used for the automated synthesis of asynchronous microcontrollers and micro-pipelined asynchronous circuits. Pipefitter allows description of an asynchronous circuit specification using a subset of standard HDL Verilog and automatically produces an asynchronous Control Unit (CU) and a synchronous Data Path (DP) of that circuit. The CU is converted into a STG and can be synthesized using Petrify while the DP is "synthesized into standard cells by an appropriate RTL/logic synthesis tool, such as the Synopsis Design Compiler or Cadence RC or Mentor Leonardo." [20]

5.3.2.1 Overview

Pipefitter uses Verilog for the initial specification as well as the output format for the intermediate representation of the CU and DP. The complete design flow for the tool is shown in Figure 5.4 on the next page. To begin with, Pipefitter optimizes the initial Verilog specification and splits it into a CU and a DP using a splitter. The splitter outputs a STG describing the behaviour of the CU, "a set of files, described in synthesizable HDL (synthesizable Verilog in this case), one for each register and its input combinational logic in the Data Path" [9] and a bunch of scripts, required to perform the logic synthesis and timing analysis of the DP blocks.

Logic synthesis and state encoding for the CU can be performed using an asynchronous tool such as Petrify. Eventually a Verilog control netlist is produced for the CU which is used by standard synthesis tool for technology mapping. Standard logic synthesis tools can also be employed for synthesizing the DP. As a result of the DP synthesis, Pipefitter automatically generates a DP netlist and a netlist of matched delays for each unit in the DP. "Timing analysis must then be performed on the DP, in order to determine the worst case delays. Its result can be used to modify the automatically generated matched delay block for each register and DP block, in order to generate the proper acknowledge signals for the control unit." [20]

At last, the automatically generated top-level netlists i.e. the control netlist, the DP netlist and the netlist of matched delays are combined in the merger to generate a final standard cell netlist of the complete circuit. [37] It is "the merger that reads the timing analysis results, and sizes the delay lines that implement the acknowledgement signals from the DP to the CU." [9] Subsequently, placement and routing can be performed so as to obtain the final layout.

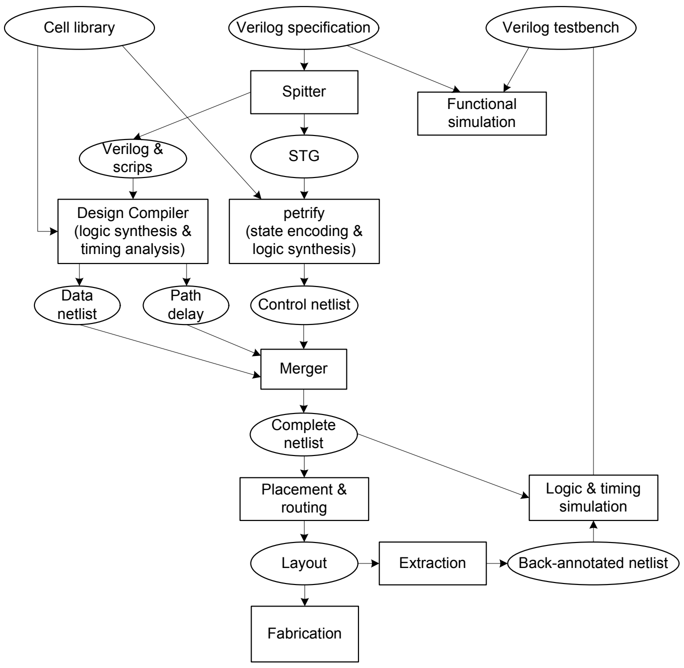


Figure 5.4: *Block diagram of Pipefitter synthesis flow. Based on a figure in [9]*

5.3.2.2 Pro's

Pipefitter has the following advantages:

- **Cost:** Pipefitter is a freely available tool.
- **Integration with Nordic Semiconductor's design flow:** The use of Verilog makes it possible to use the existing EDA commercial tools for almost all design phases such as simulation, synthesis and layout. Moreover, Pipefitter uses the standard synchronous HDL tools for synthesizing the DP.
- **Support:** Pipefitter offers excellent support. The developers of Pipefitter were contacted and a quick reply was received.

5.3.2.3 Con's

Pipefitter has the following disadvantages:

- **Longevity:** Pipefitter was developed in 2004. The last research paper [37] found for Pipefitter dates from 2002. Pipefitter was not developed after the initial version was introduced in 2004.
- **Tool complexity:** Pipefitter uses an asynchronous tool, such as petrify to synthesize the CU. Hence, all the parameters adding to the complexity of Petrify are applicable for Pipefitter as well. Unlike the splitter, the merging unit of Pipefitter was not implemented by the developers and has to be implemented. Furthermore, the synthesis process is only partially automated, for example the constraints for the timing analysis and layout tools cannot be generated automatically.
- **Estimate of performance of corresponding circuits:** Unknown.
- **Commercial or Non-commercial implementations:** There are no known commercial implementations using Pipefitter. However, it has been used for the automated synthesis of asynchronous microcontrollers and micro-pipelined asynchronous circuits in academic research based projects. Furthermore, several newer tools have been proposed based on Pipefitter, such as the PAid tool in 2012 [42].

5.3.2.4 Pipefitter discussion

Pipefitter is one amongst the selected shortlisted tools because of excellent integration with Nordic Semiconductor's design flow.

5.4 HDL based specification approaches

HDL based specification methods use a standard HDL such as Verilog as an input to the tool.

5.4.1 Chainworks

Chainworks was developed by Silistix Ltd, UK. It is a tool suite comprising of various software tools that help to design and synthesize a customized on-chip interconnect fabric between various IP blocks using self-timed (clockless) circuits. [66]

5.4.1.1 Overview

"The CHAINworks tool suite takes a description of the initiator and target ports of an System on Chip (SoC) design and synthesizes a structural netlist for a CHAIN interconnect. The suite comprises three tools: CHAINdesigner™ for design exploration, CHAINcompiler™ for CHAIN fabric synthesis, and CHAINlibrary™, which is an interconnect component library." [67] CHAINdesigner uses a description of the initiator and the target ports of the design as an input to perform synthesis and generate a structural netlist for the interconnect fabric. It is CHAINcompiler which then uses this structured netlist and components from the CHAINlibrary to perform logic synthesis and map to standard cells for layout. [68]

Furthermore, the tools used by Chainworks integrate well with the existing conventional design flow. Design entry as the input to CHAINcompiler can be done using Schematic Capture or System Verilog, which in turn generates a netlist in structural Verilog. Existing tool setup can then be employed to perform synthesis and timing analysis using the scripts generated by Chainworks. [69]

5.4.1.2 Pro's

Chainworks has the following advantages:

– **Estimate of performance of corresponding circuits:** Silistix claims that Chainworks leads to significant power savings compared to a conventional design flow. However, replacing the clock with handshaking signals might lead to some area overhead.

– **Commercial or Non-commercial implementations:** Chainworks has been used to implement "a smart card test IC built in 180-nanometer CMOS technology." [70]

– **Integration with Nordic Semiconductor's design flow:** According to [67], Chainworks can easily integrate into the existing synchronous design flow. It can use Verilog as an input and generate a structural netlist in Verilog.

5.4.1.3 Con's

Chainworks has the following disadvantages:

– **Longevity:** Chainworks was introduced by Silistix in 2006. However, since it is not freely available, no open information or reasearch papers are available for Chainworks. Hence, it is difficult to comment on the longevity of this tool.

– **Tool complexity:** It will take sometime for the designers to educate themselves on how the new tool suite functions.

– **Cost:** It is not a freely available tool and thus there is a cost associated with this tool. Chainworks is a commercially available tool which is approximately priced the same as Synopsis Design Compiler, which is a tool for synthesizing and performing the timing analysis for synchronous design.

- **Delay model used by the final netlist:** This information is not available openly, and since Silistix was not contacted, it is difficult to comment on the nature of the netlist obtained from the Chainworks tool suite. However, it seems like Chainworks is following a desynchronization approach, see Section 5.5.1 on the facing page.

- **Support:** No separate webpage for Chainworks was found. Moreover, Chainworks was not found on the webpage for Silistix [71].

5.4.1.4 Chainworks discussion

Chainworks was eliminated and was not looked into any further. It was mainly because of the high cost and the desynchronization approach used. Desynchronization involves replacing clock domains by local handshaking circuitry. This leads to alot of overhead. It is not a purely asynchronous technique and involves conversion from a synchronous design to an asynchronous one, which is not the most efficient way available. Moreover, the aim of this thesis is to find a purely asynchronous tool.

5.4.2 ACC

ACC stands for Asynchronous Circuit Compiler. It is a commercial synthesis tool by Tiempo, that generates DI asynchronous circuits from input specifications in standard HDL such as Verilog or VHSIC Hardware Description Language (VHDL). According to Tiempo's website [72], the licence for ACC can be purchased.

5.4.2.1 Overview

ACC takes design descriptions in Verilog/System Verilog, and generates a standard Verilog gate level netlist as the output. "ACC can be inserted in any standard design flow, allowing designer to verify asynchronous and mixed asynchronous/synchronous circuits using any industry-standard simulation tools. The generated Verilog netlist can then be placed-and-routed using any standard back-end tool and verified with any electrical simulation tool." [72]

5.4.2.2 Pro's

ACC has the following advantages:

- **Delay model used by the final netlist:** The website [72] claims to generate a DI netlist. Owing to less strict timing analysis requirements, a delay DI is the most suited for the purpose of this thesis.

- **Integration with Nordic Semiconductor's design flow:** ACC integrates well with the standard synchronous design tools and flow. It uses standard HDL such as Verilog/System Verilog as the input specification language. Moreover, it produces a gate level netlist in standard Verilog format as an output.

– ACC is a fully automatic synthesis tool.

5.4.2.3 Con's

ACC has the following disadvantages:

– **Cost:** Being a commercial tool, there is a certain cost associated with the purchase of the ACC license.

– The website [72] seems to provide extremely general information without getting into any ACC details or specifics. Hence, a lot of parameters such as longevity, tool complexity, performance, commercial implementations and support are unknown.

5.4.2.4 ACC discussion

Based on ACC's pro's and con's, ACC is one amongst the selected shortlisted tools mainly because of the DI nature of the netlist obtained and the possibility for easy integration into Nordic Semiconductor's tool flow. Using a Verilog input and outputting a Verilog netlist is ideal for this thesis. Even though ACC had an associated cost, it appeared to be exactly something that was required. It was thought that in the next integration phase, Tiempo could be contacted for ACC, and hence, the unknown parameters could also be figured out.

5.5 Miscellaneous

This section comprises of tools/methods that cannot be placed in any of the above categories.

5.5.1 Desynchronization

In its simplest form, desynchronization is the method of converting a synchronous circuit into an asynchronous circuit by replacing the functions originally handled by a clock signal with local handshaking circuitry. [73]

5.5.1.1 Overview

Synchronous design has been a long lived practice in the industry in the field of digital circuit design. The operation cycles of the clock have been known to ease design practices for decades now. On the other hand, asynchronous design implementations have proven their usefulness on various fronts such as low power consumption, low

EMI, ease of integration, etc. Desynchronization is a technique that reconciles both these conflicting design practices. "Desynchronization is a new paradigm to automate the design of asynchronous circuits from synchronous specifications, thus, permitting widespread adoption of asynchronicity without requiring special design skills or tools." [74] Given a synchronous synthesized circuit, the essential idea is to "replace directly the global clock network with a set of local handshaking circuits." [74] The circuit can then be implemented using the standard synchronous design tools and flow. The only difference is the algorithm responsible for the generation of the clock tree which is replaced with handshaking signals.

5.5.1.2 Pro's

Desynchronization has the following advantages:

- **Tool complexity:** It is a relatively simple approach which can be followed by experienced designers almost instantaneously and without much risk. It can be used to achieve asynchronicity without requiring any special tools.
- **Commercial or Non-commercial implementations:** Several designers and researchers, for example [75] have used this approach before to achieve the desired asynchronous implementation starting from a corresponding synchronous implementation. In fact, there exists a valid patent given by [76] for this approach or method.

5.5.1.3 Con's

Parameters such as longevity, performance and integration with Nordic Semiconductor's design flow are unknown. Parameters such as cost and support are not applicable here, since desynchronization is a method and not a tool.

5.5.1.4 Desynchronization discussion

Desynchronization was eliminated mainly because it is a method and not a tool. A purely asynchronous tool is preferred for this thesis.

Chapter 6

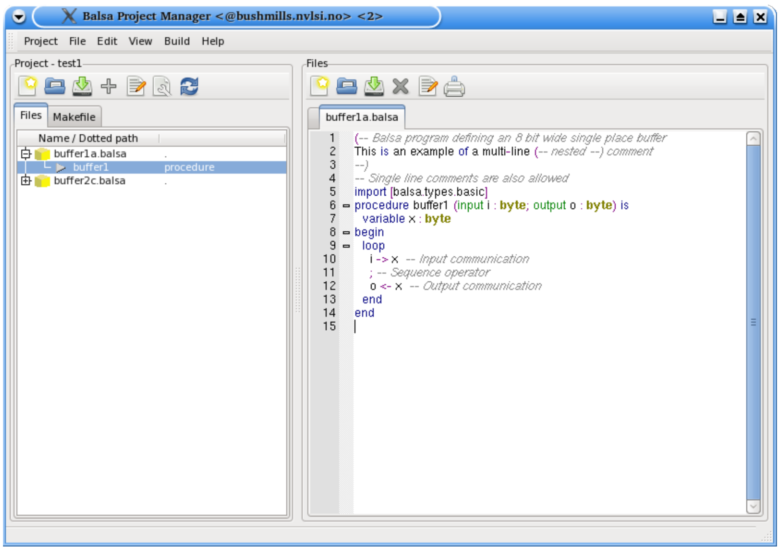
Tool integration aspects for the shortlisted tools

The preceding Chapter 5 carried out a detailed discussion about the most significant asynchronous tools and methods that have been employed for asynchronous design and synthesis in the industry and academia. The result from Chapter 5 comprises of a shortlisted selection of the most desired asynchronous tools/methods for fulfilling the requirement of this thesis. The list of the selected tools is: Balsa, ACC, Petrify, Verisyn and Pipefitter.

The goal for this chapter is to find the most suitable tool (out of the five above mentioned tools) that integrates best with Nordic Semiconductor's tool chain. An investigation of the integration aspects, possibilities and issues for each of the above mentioned tools, with Nordic Semiconductor's design flow, has been been conducted. Each of the tools was installed and tried out. The instructions in the manual/tutorial guide available with the tool, were followed to synthesize a simple design given in the documentation. The selected tool and the corresponding discussion so as to why the tool was selected formulates the results from this chapter. The end of the chapter presents a discussion based on a single tool selection out of the lot of 5 tools. For the purpose of clarity of the reader, it was more natural to have these results in this chapter rather than placing them in a separate results chapter.

6.1 Balsa Integration

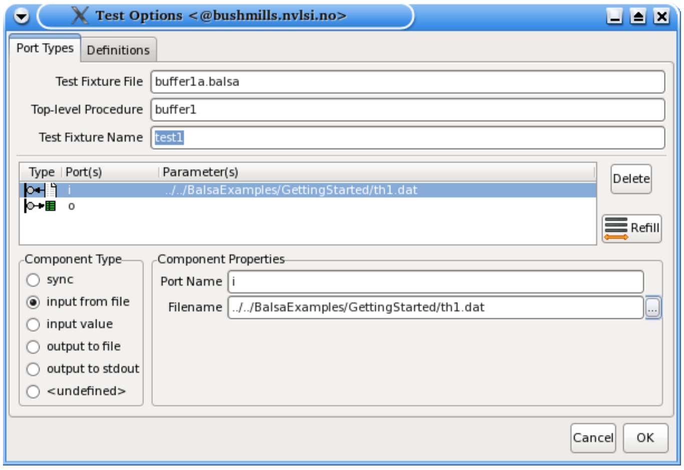
Balsa version 4.0, June 2010 release was downloaded from [23]. Several plugins like different backend technologies and simulator support for QuestaSim were also downloaded and installed. Balsa also offers a comprehensive tutorial [24] that includes the theory revolving around Balsa, example code samples, tool usage, procedure for simulation and synthesis, etc. Initially, a simple example of a buffer available with the Balsa package was tried. This example involved setting up a project, loading the initial design into Balsa, generating an automatic testbench, simulation and synthesis of a Verilog netlist.


Balsa Project Manager <@bushmills.nvlsi.no> <2>
Project File Edit View Build Help
Project - test1
Files
Files
Makefile
buffer1a.balsa

Name / Dotted path	
buffer1a.balsa	.
buffer1	procedure
buffer2c.balsa	.

```
(-- Balsa program defining an 8 bit wide single place buffer
This is an example of a multi-line (-- nested --) comment
--)
-- Single line comments are also allowed
import [balsa.types.basic]
procedure buffer1 (input i : byte; output o : byte) is
  variable x : byte
begin
  loop
    i -> x  -- Input communication
    ; -- Sequence operator
    o <- x  -- Output communication
  end
end
```

Figure 6.1: *balsa-mbr, the balsa project manager environment*



Test Fixture File: buffer1a.balsa
Top-level Procedure: buffer1
Test Fixture Name: test1

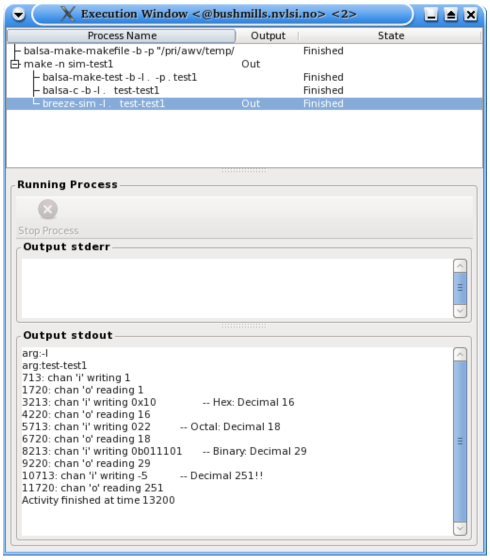
Type	Port(s)	Parameter(s)
	i	../../BalsaExamples/GettingStarted/th1.dat
	o	

Component Type: ( ) sync [✓ input from file] ( ) input value ( ) output to file ( ) output to stdout ( ) <undefined>

Component Properties
Port Name: i
Filename: ../../BalsaExamples/GettingStarted/th1.dat

Figure 6.2: *Adding a test fixture to the design*

Figure 6.1 shows the main window for Balsa with the initial code for the buffer loaded. The left side shows the files in the given project and the right side shows the example code. The first step was to compile and generate a technology independent breeze netlist of handshaking components. The next step was increasing the complexity of the example code by adding several buffers. A new compilation was performed and a new breeze netlist was generated. Then, a simulation of the breeze netlist was conducted. Balsa has a built in simulator which is able to simulate both a technology independent breeze netlist and a Verilog netlist. In order to perform a simulation, a testbench needs to be generated. In Balsa this process is called 'adding a text fixture' as shown in Figure 6.2 on the facing page. When setting up the test fixture, the handling of the inputs and outputs is defined. For example, the inputs can be set to a constant or given values from a file. Similarly, the outputs can be written to a file or shown in the simulator transcript. In the example code for the buffer, the inputs are given values from a file while the outputs are written directly to the transcript. Balsa has two options for simulation. One is the text-only simulation as seen in Figure 6.3 and the other is the Graphical User Interface (GUI) mode simulation as seen in Figure 6.4 on the following page. Some of the benefits of using the GUI mode simulation include the possibility to display the signals in the waveform viewer as seen in Figure 6.4 and displaying the handshake circuit graph as seen in Figure 6.5 on the next page. The handshake circuit graph is giving an graphical representation of the breeze components used in the design. Both Figure 6.4 and Figure 6.5 are generated as a result of performing the simulation of the example in the Balsa manual. To be able to use the GUI mode simulation, GTKWave [77] had to be installed.

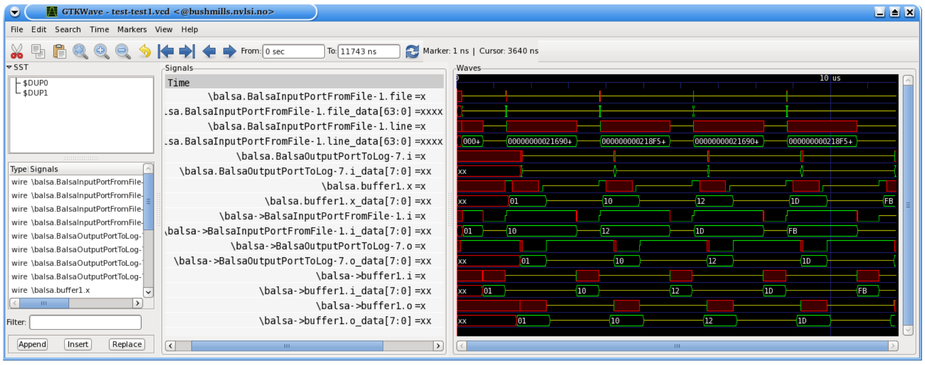


Process Name	Output	State
balsa-make-makefile -b -p "/pri/awv/temp/		Finished
make -n sim-test1	Out	
balsa-make-test -b -I . -p . test1		Finished
balsa-c -b -I . test-test1		Finished
breeze-sim -I . test-test1	Out	Finished

```
arg:-I
arg:test-test1
713: chan 'i' writing 1
1720: chan 'o' reading 1
3213: chan 'i' writing 0x10          -- Hex: Decimal 16
4220: chan 'o' reading 16
5713: chan 'i' writing 022       -- Octal: Decimal 18
6720: chan 'o' reading 18
8213: chan 'i' writing 0b011101      -- Binary: Decimal 29
9220: chan 'o' reading 29
10713: chan 'i' writing -5       -- Decimal 251!!
11720: chan 'o' reading 251
Activity finished at time 13200
```

Figure 6.3: *Text only simulation*

The next step was to see if Balsa could generate a netlist that could be used in Nordic Semiconductor's design flow. Different technologies and styles can be selected for performing synthesis and generating a netlist as shown in Figure 6.6 on page 63. The different technologies and styles are not discussed at this point.


GTKWave - test-test1.vcd <@bushmills.nvlsi.no>
File Edit Search Time Markers View Help
From: 0 sec
To: 11743 ns
Marker: 1 ns | Cursor: 3640 ns
SST
$DUP0
$DUP1
Type Signals
Filter:
Append Insert Replace
Signals
Waves
10 us

```
Time
\balsa.BalsaInputPortFromFile-1.file =x
.sa.BalsaInputPortFromFile-1.file_data[63:0] =xxxx
\balsa.BalsaInputPortFromFile-1.line =x
.sa.BalsaInputPortFromFile-1.line_data[63:0] =xxxx
\balsa.BalsaOutputPortToLog-7.i =x
\balsa.BalsaOutputPortToLog-7.i_data[7:0] =xx
\balsa.buffer1.x =x
\balsa.buffer1.x_data[7:0] =xx
\balsa->BalsaInputPortFromFile-1.i =x
\balsa->BalsaInputPortFromFile-1.i_data[7:0] =xx
\balsa->BalsaOutputPortToLog-7.o =x
\balsa->BalsaOutputPortToLog-7.o_data[7:0] =xx
\balsa->buffer1.i =x
\balsa->buffer1.i_data[7:0] =xx
\balsa->buffer1.o =x
\balsa->buffer1.o_data[7:0] =xx
```

```
wire \balsa.BalsaInputPortFromFile-
wire \balsa.BalsaInputPortFromFile-
wire \balsa.BalsaInputPortFromFile-
wire \balsa.BalsaInputPortFromFile-
wire \balsa.BalsaOutputPortToLog-
wire \balsa.BalsaOutputPortToLog-
wire \balsa.BalsaOutputPortToLog-
wire \balsa.BalsaOutputPortToLog-
wire \balsa.buffer1.x
```

Figure 6.4: *GUI simulation*

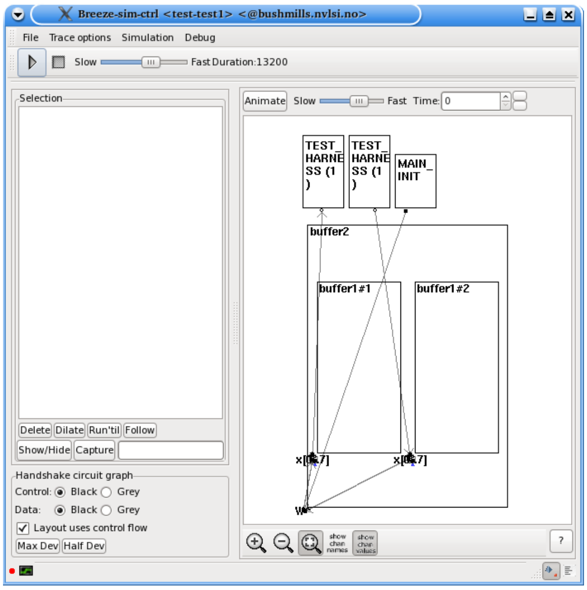
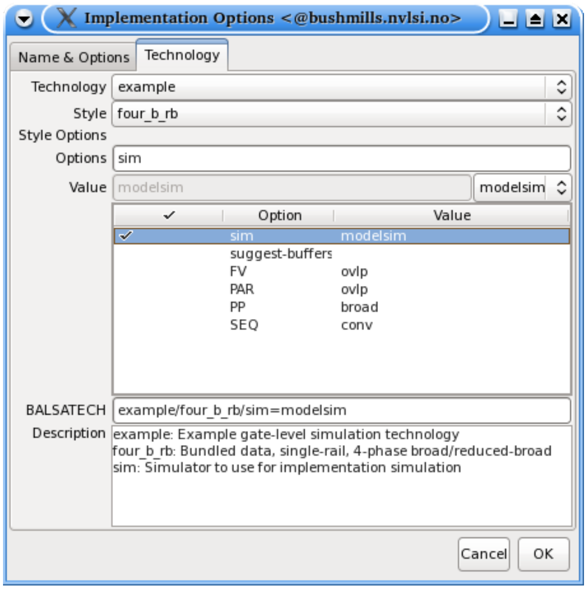


Figure 6.5: *GUI simulation with State graph*

For the initial testing at this stage a technology named balsa-tech-example was chosen since it generated a Verilog netlist, which is a requirement for integration into Nordic Semiconductor's design flow. Also, a design style named four_b_rb was chosen since it in theory produces the simplest netlist. This was considered to make debugging of the netlist easier. Finally, the synthesis was performed and a Verilog netlist was successfully generated. Moreover, this netlist was loaded and simulated in QuestaSim. This proves that a valid netlist has been generated and should fit into

Nordic Semiconductor's design flow.



Technology: example
Style: four_b_rb
Style Options
Options: sim
Value: modelsim | modelsim

✓	Option	Value
✓	sim	modelsim
	suggest-buffers	
	FV	ovlp
	PAR	ovlp
	PP	broad
	SEQ	conv

BALSATECH: example/four_b_rb/sim=modelsim

Description:
example: Example gate-level simulation technology
four_b_rb: Bundled data, single-rail, 4-phase broad/reduced-broad
sim: Simulator to use for implementation simulation

Figure 6.6: *Adding an implementation*

6.2 Petrify Integration

Petrify version 4.2 was downloaded from the webpage [30]. This is the last available version of Petrify and dates back to 1999. Therefore, it seems like Petrify has not been under active development for a long time. A comprehensive tutorial was included in the installed Petrify package. The aim of the tutorial is to use simple examples to show the main features of Petrify.

An example code is provided in the Petrify tutorial. This code is shown below. In comparison to Balsa, Petrify does not incorporate a GUI environment. Instead, the code must be written in a separate text editor and then different Petrify functions are run from within the command line on the various code files.

```
.model pn_synthesis

# Declaration of signals
.inputs a
.outputs b e f
.internal c
.dummy d
```

```
# Petri net
.graph
a/1 b c
b d
c d
d a/2
a/2 e f
e d/1
f d/1
d/1 a/1

# Initial marking
.marking { <d/1,a/1> }
.end
```

To visualize the Petri Net (PN) described in the code, Petrify is providing the function '`draw_astg`'.This generates a graphical representation of the PN being described. Running the command:

```
draw_astg -nofold -bw pn_syn.g -o pn_syn.g.ps
```

where *pn_syn.g* is the file containing the example code, produced the PN shown in Figure 6.7

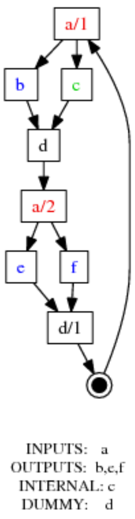


Figure 6.7: *Petri Net for the example code*

The next step was to generate a State Graph (SG) for the PN in Figure 6.7. The command:

```
write_sg pn_syn.g | draw_astg -sg -noinfo -o pn_syn.sg.ps
```

produces the SG shown in Figure 6.8.

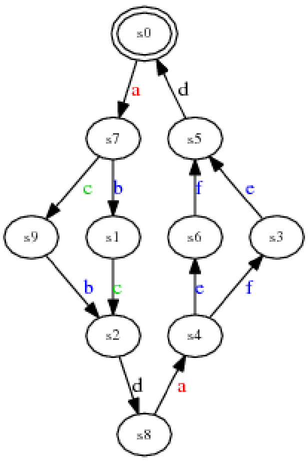


Figure 6.8: *State graph for the example code*

There are several options while synthesizing PNs. The two commonly used options are: synthesizing into a simpler PN and synthesizing to generate an equation. Both the mentioned options were tried one by one.

First, the original PN was synthesized to generate another simpler PN. The command:

```
petrify pn_syn.g -o pn_syn2.g
```

was used to run the synthesis. The newly derived PN is shown in Figure 6.9 on the next page. In Petrify, synthesis of an PN or SG involves performing a token flow analysis of the initial PN and generates another simpler PN which is an optimized version of the initial PN or SG. 'Simpler' in this context refers to a PN where it is easier to understand the underlying functionality of the design. It does not necessarily imply that the derived PN is visually simpler. The newly generated simpler PN is functionally equivalent (bisimialar) to the initial one with the relabelling of all transitions. Also, the correctness properties of the derived PN can be verified to prove equivalence of both the PNs.

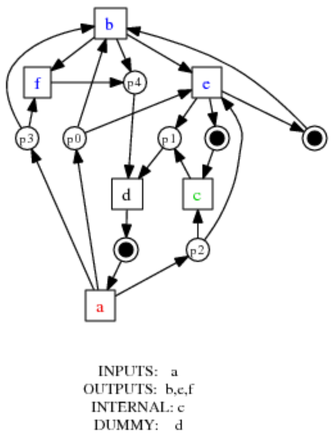


Figure 6.9: *Petri net synthesized by petrify*

Then, for synthesizing into an equation, a different example code (from the tutorial) with its corresponding SG was used. The synthesis of PNs using the complex gate synthesis approach of petrify, produces a file with a DPean equation expression. Given below is the code, and the SG is shown in Figure 6.10 on the facing page

```
.inputs x
.outputs y z
.graph
x+ y+ z+
z+ x-
y+ z-
x- z-
z- y-
y- x+
.marking{<y-,x+>}
.end
```

The command for synthesizing into an equation is:

```
petrify xyz.g -cg -eqn xyz.eqn -no
```

Using this command, the following equation was generated:

```
# EQN file for model xyz
# Generated by ../bin/petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 5.00
```

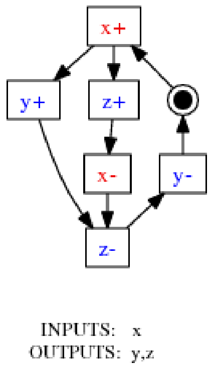


Figure 6.10: *State graph for the given code*

```
INORDER = x y z;
OUTORDER = [y] [z];
[y] = z + x;
[z] = y' z + x;

# Set/reset pins: reset(z)
```

The next step would be to transform the obtained Boolean equation or PN into a Verilog gate level netlist. However, this process was not documented in the Petrify tutorial or any other research paper read regarding Petrify. [78] showed the manual mapping of a Boolean equation into a Verilog netlist to verify the circuit. However, a manual mapping is not feasible for a larger design. Hence, automatic mapping is a requirement for using Petrify in Nordic Semiconductor's design flow. It was then considered to contact the developer of Petrify and clarify the possibilities to convert the Petrify output into a Verilog gate level netlist. However, Petrify is not under development anymore. Hence, it was considered to contact the developers of Workcraft [36] instead. Workcraft (Section 5.2.4 on page 48)is a new toolset being developed at the University of Newcastle. It employs Petrify as one of the possible backend tools and therefore its developers should have extensive knowledge about the practical aspects of Petrify.

Danil Sokolov from the University of Newcastle was contacted. He is one of the main developers of Workcraft. The query put forward was how to produce a Verilog netlist from the Boolean equation or PN generated by Petrify. According to Danil Sokolov, there is no straightforward (well documented) way to proceed from a Petrify generated netlist to fabrication. After some discussion with Danil Sokolov, it was concluded that Petrify can do technology mapping into the gates specified in the Petrify library i.e. petrify.lib. Basically, if petrify.lib and a PN specification are present in the same directory, then the following command would generate a Verilog netlist:

```
petrify -lib petrify.lib -tm -vl circuit_netlist.v pn_spec.g
```

This is producing a Speed Independent (SI) netlist. Furthermore, Petrify is only providing a generic library. However, a specific library targeting the desired technology to be used for layout in the backend, must be generated. Making a specific library has not been done at this stage and will only be executed for the final selected tool.

6.3 Verisyn Integration

Verisyn was downloaded from [8]. The only available version was from September, 2004. Verisyn was precompiled and the source code was not available. A tutorial was included in the downloaded Verisyn package.

Verisyn was started and a project called ’test’ was created. From the various available files in the Verisyn package, an example file was loaded into the project. Then, the project was compiled using the ’compile’ option in the window. When starting the compilation of the project, the program crashed with a segmentation fault. The error message was:

```
[2]    1596 segmentation fault  ./verisyn
```

Since, the source code of Verisyn was not available, the chief developer of Verisyn, Frank Burns, was contacted for a solution. According to him, Verisyn was developed some years back and there will be problems running it on current systems as it is dependent on old libraries and graphical software. Frank Burns offered to compile a new version of Verisyn using updated libraries. Meanwhile, he suggested to look more into Workcraft (Section 5.2.4 on page 48) since it was a newer tool being actively developed at the University of Newcastle. It took him several weeks to compile and send a new version. Verisyn is using another program in the background, iverilog [79], for all Verilog related handling. According to him, due to changes in the IEEE specification of Verilog, the current iverilog was not compatible with the older Verisyn version that he was using. Hence, he had to update Verisyn to comply with these changes. However, by then, the final tool had already been selected due to the strict time frame and no more testing was carried out with Verisyn.

6.4 Pipefitter Integration

The webpage for Pipefitter [48] was not up-to-date. Moreover, the links for installing Pipefitter were broken. The author of Pipefitter, Luciano Lavango, was contacted with a request for the tool. He sent the Pipefitter version 1.0 from November 2003, but also mentioned that his research team had abandoned Pipefitter a long time ago. This is because they found better ways to use Verilog to specify asynchronous circuits. Moreover, Pipefitter failed to have commercial success. Hence, they stopped developing Pipefitter.

The Pipefitter version received was not stable in Nordic Semiconductor's environment. It gave both compilation and runtime errors. In order to use Pipefitter, the code was debugged and errors were corrected. Thereby, Pipefitter was successfully installed.

Pipefitter included a tutorial that was followed. An example for an asynchronous register written in Verilog was provided in the tutorial, as shown below:

```
// Your first pipefitter specification!!!
module async_reg(Data_in, Rin, Ain, Data_out, Rout, Aout);
        input [7:0] Data_in;
        input Rin;
        output Ain;
        reg Ain;
        output [7:0] Data_out;
        reg [7:0] Data_out;
        output Rout;
        reg Rout;
        input Aout;

        always begin
                wait(Rin);
                Data_out = Data_in;
                Ain = 1;
                wait(!Rin);
                Ain = 0;
                Rout = 1;
                wait(Aout);
                Rout = 0;
                wait(!Aout);
        end

endmodule
```

The code was compiled using the command:

```
pipefitter  register.v
```

which gave the output message:

```
<register.v> successfully compiled
```

As mentioned before in section 5.3.2 on page 52, Pipefitter splits the asynchronous input specification into a Data Path (DP) and a Control Unit (CU). Therefore, the next step was to generate a DP for this example code. The command:

```
pipefitter register.v -dp data_path.v
```

The DP was written in the file data_path.v and was a synthesizable Verilog netlist.

Then, a CU was generated for the example code using the command:

```
pipefitter register.v -cntl control_unit.v
```

The CU was written in the file control_unit.v and was a synthesizable Verilog netlist of David Cells.

Furthermore, Pipefitter generates a file with all the delay elements i.e. propagation delays and logic delays, for the example circuit using the command:

```
pipefitter register.v -del del.v
```

The delay elements was written in the file del.v. This file is only used for behavioural simulation.

Even though Pipefitter is able to generate synthesizable Verilog netlists for both the DP and the CU, it is lacking the functionality to perform the merging of the two netlists into a single netlist that can be used for timing analysis, placement and layout.

6.5 ACC Integration

ACC is a commercial tool developed by Tiempo [80]. It is not available for free download. Therefore, Tiempo was contacted with the request for the ACC tool. However, even after several attempts to contact them, no answer was received. It seems like the company does not exist anymore. Hence, no further evaluation of ACC was possible.

6.6 Discussion for a single tool selection

ACC is eliminated as no response was received from Tiempo, even after trying to contact them several times. Pipefitter is able to generate a synthesizable Verilog netlist for both the DP and the CU. However, it lacks the source code for the functionality performing the merging, which has to be implemented if Pipefitter is to be used. Pipefitter's integration with Nordic Semiconductor's design flow depends on the merger, which combines the CU Verilog netlist obtained from the asynchronous tool with the DP Verilog netlist obtained from the conventional synchronous tools. The timing analysis is then performed on the merged Verilog netlist. Implementing the merger seems to be a lot of work for a tool to begin with. Hence, Pipefitter is eliminated and not investigated any further. Verisyn was developed many years back and there is a problem running it on current systems as it is dependent on old libraries and graphical software. As a result, the Verisyn program crashed with a segmentation fault error. The developer of Verisyn offered to solve this problem and send a version for the tool that could be successfully compiled. However, before an error free version of the tool could be produced, a tool had to be selected due to timing limitations. Hence, Verisyn is eliminated as well. There was a close tie between Petrify and Balsa. Both of them are able to successfully generate Verilog netlists and show a good integration possibility with Nordic Semiconductor's design flow. Moreover, Balsa is even able to load the generated Verilog netlist in QuestaSim. The major difference, however, is that Petrify generates a SI Verilog netlist while Balsa generates a DI Verilog netlist. As mentioned earlier, for the purpose of this thesis, a DI netlist is preferred over a SI netlist due to less restrictive timing assumptions. A more elaborate timing analysis (by constraining the isochronic forks) is required for SI circuits as compared to DI circuits. Moreover, there is no straightforward (well documented) way to proceed from a Petrify generated netlist to fabrication. This however is not the case for Balsa. Balsa includes a comprehensive manual which provides a good explanation of taking a Balsa generated netlist to fabrication. It also explains the procedure to generate a new technology for Balsa. Hence, Balsa was selected over Petrify.

Therefore, Balsa is the selected tool that will be further investigated and used to implement a design.

Chapter 7 An introduction to the Balsa system

This chapter describes briefly some of the basic concepts of Balsa which are a prerequisite for understanding the remaining chapters of the report. It sets out to explain the basic handshaking principle on which Balsa is based, the Balsa tool set and design flow.

7.1 Basic Concepts

7.1.1 Handshaking principle

Balsa compiles circuits into a network of handshaking components that communicate by means of channels connecting them over which handshakes take place. Balsa employs the principle of four phase handshaking over its channels. The channels can be associated to datapaths or control paths. A channel dedicated to datapath represents a handshake involving the transfer of data. On the contrary, a control path represents a handshake signifying synchronization or a rendezvous point. [3]

"Each channel connects exactly one passive port of a handshake component to one active component of another handshake component." [24] An active port can be defined as the one initiating the communication while a passive port is the one that responds by means of an acknowledgement to the request from the active port whenever it is ready.

Furthermore, data channels can be categorized as push channels and pull channels. Data transfer in push channels takes place from the active port to the passive port. A request signals that the data is valid and the data is released on receiving an acknowledgement. On the other hand, the data transfer in pull channels takes place from the passive port to the active port. The data transfer is requested by the active port to which the passive port responds with an acknowledgement signalling that the data is valid. [24]

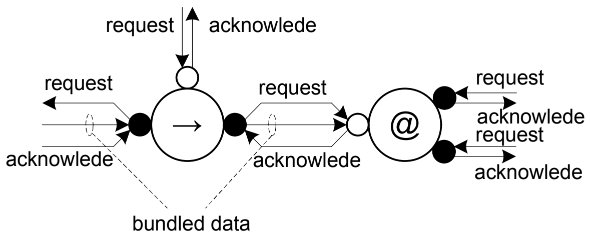


Figure 7.1: *Two connected handshake components*

The basic principle of handshaking between handshake components in Balsa is shown in Figure 7.1 which is sourced from [24]. It composes of two handshake components namely *Transferrer* (denoted by →) and *Case* (denoted by @). In this figure, active ports are denoted by filled circles while the passive ports are denoted by empty circles. The entire communication process is initiated by a request to the *Transferrer* at its passive port. Upon receiving this request, the first round of circuit action is activated and the *Transferrer* issues a request demanding data from its environment on the left. The environment then supplies the data to the *Transferrer* by indicating data validity using an acknowledgement signal. The *Transferrer* then "presents a handshake request and data to the *Case* component using its active port which the *Case* component receives on its passive port." [24] Depending on the value of data received, the *Case* component issues a handshake request towards the environment on its right on either the top port or the bottom port. Finally, when the *Case* component receives an acknowledgement from its environment, the acknowledgement is returned to the *Transferrer* along the original channel in the direction opposite to which the handshake request and data were sent. This terminates the handshake between the *Transferrer* and the *Case* component and the circuit is ready for another communication round. [3]

7.1.2 Tool set and design flow

Given below is a list of the most useful components included in the Balsa tool set. These components along with their corresponding functions have been provided to make the reader more aware about the most important features of the Balsa tool set. More information about these or any other components can be found in the Balsa manual [24]. The components are as follows:

- **balsa-c:** It is the Balsa language compiler which produces an intermediate breeze netlist.

– **balsa-netlist:** It uses a breeze netlist as input to produce a netlist of the required target technology or appropriate CAD framework.

– **balsa-md:** It generates make files in Balsa.

– **balsa-mgr:** It provides a GUI front end to balsa-md with additional project management facilities

– **balsa-make-test:** It generates an automatic testbench for descriptions in Balsa.

– **breeze-sim:** It simulates the breeze descriptions.

– **balsa-verilog-sim:** A package that provides support to make the Verilog simulation of Balsa decriptions easier.

– **GTKWave:** It is a waveform viewer available as a separate package from GTK.

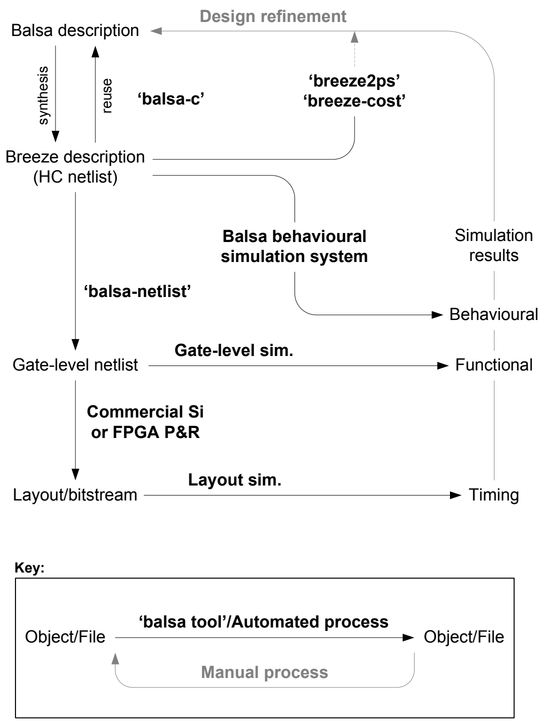


Figure 7.2: *Balsa design flow*

Figure 7.2 on the previous page from [24] gives an outline of the Balsa design flow. To begin with a design specified using the Balsa language is compiled using the Balsa compiler i.e. 'balsa-c' to produce an intermediate breeze netlist of handshaking components. Behavioural simulation can then be performed over the breeze netlist using 'breeze-sim' to visualize channel activity at the handshake circuit level and for source level debugging. Furthermore, waveforms can be produced to monitor the simulation using GTKWave. At this point, Balsa can also produce a graphical representation of the breeze netlist, as shown by the example in Figure 7.3. Then, the breeze netlist is compiled into a gate-level netlist using the command '`balsa-netlist`'. Functional simulation can be performed over the gate level netlist to verify its validity. To get from the netlist to a layout netlist or to a FPGA, a third party backend tool or FPGA place and route tool, must be used. A layout simulation can then be performed on the layout netlist.

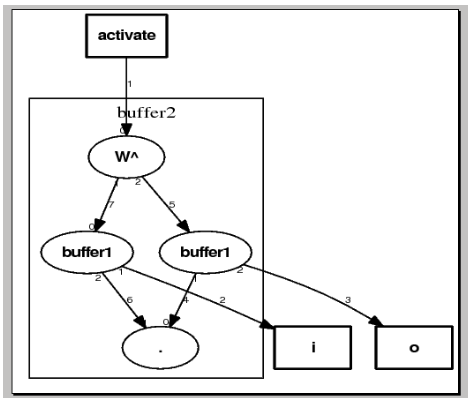


Figure 7.3: *State graph*

"Balsa supports several design styles such as:

– **four_b_rb:** bundled data four-phase, broad, reduced broad protocol
– **dual_b:** dual rail delay insensitive encoding with return to zero signalling
– **one_of_2_4:** one-of-four delay insensitive encoding with return to zero signalling" [24]

Earlier, Balsa provided support for several backend technologies. However, for the Balsa version 4.0 used in this thesis, only two technologies are provided i.e. Balsa-tech-example and balsa-tech-minimum. These technologies are example libraries that can not be used for practical implementations. They are only used as a reference to build a new technology. Besides this, Balsa version 4.0 provides support to build new technologies required for the implementation technology being used. For the purpose of this thesis, a new technology is built, as seen in Chapter 9 on page 95.

Chapter 8

Design implementation, synthesis and verification

This chapter aims at implementing an asynchronous flash readout using Balsa. "As most other design practices, currently, the flash readout at Nordic Semiconductor is done synchronously. Hence, before the flash can actually be read, it waits for power followed by the clock. However, if the memory readout is controlled asynchronously, the actual readout can start right after the power reception without having to wait for the clock. This makes it an independent process which has some time saving potential. Moreover, as mentioned before in Section 2.2 on page 6, asynchronous design leads to efficient power savings. Thus, the design implementation is driven towards achieving an asynchronous flash readout in the system design being used at Nordic Semiconductor." [11]

8.1 Design Implementation

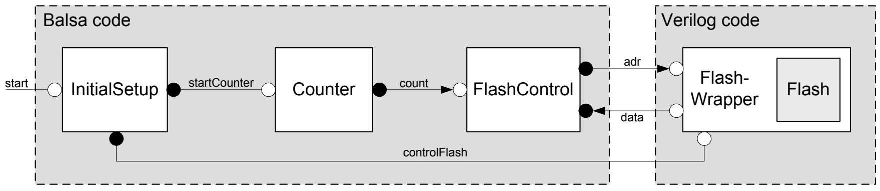


Figure 8.1: *Initial block diagram for the asynchronous flash readout*

Figure 8.1 gives an overview of the initial partitioning for the asynchronous flash readout. It consists of two main blocks: one implemented in Balsa and the other implemented in Verilog. The former controls the asynchronous flash readout and the latter is a wrapper interfacing the handshake signals from balsa to the actual flash. The flash model used at Nordic Semiconductor is implemented in Verilog, thereby posing a need for the wrapper to be written in Verilog, so as to be able to integrate this asynchronous flash readout with Nordic Semiconductors design. The Balsa block is further sub-divided into three blocks: InitialSetup, Counter and FlashControl. Each of these are described in the following sections.

8.1.1 InitialSetup

During the entire readout cycle, the Balsa block controls all the signals to the flash, for example: *XADDR*, *YADDR* and *SE*. During the setup phase, the block InitialSetup sends a request to the FlashWrapper to gain control of the required signals. If this design is integrated into the existing design at Nordic Semiconductor, all the signals to the flash are controlled by the normal synchronous logic when the Balsa block is not reading from the flash.

During the readout cycle some of the signals are static. In this design the FlashWrapper controls these static signals on receiving a request from the InitialSetup block. These signals are set with a predefined value before the first readout from the flash and released after the completion of the last readout. IFREN (Information Block Enable), REDEN (Enable signals for redundancy pages), PV (Program verify level enable), EV (Erase verify level enable) and XE/YE (X and Y address enable) are the static signals that need to be controlled during a readout from the flash, see Figure 8.2 shows the timing diagram for the read cycle for the flash that is being used at Nordic Semiconductor. Most of these signals are allowed to change between every readout from the flash, however they are kept static in this design because those functions are not needed for doing a simple readout.

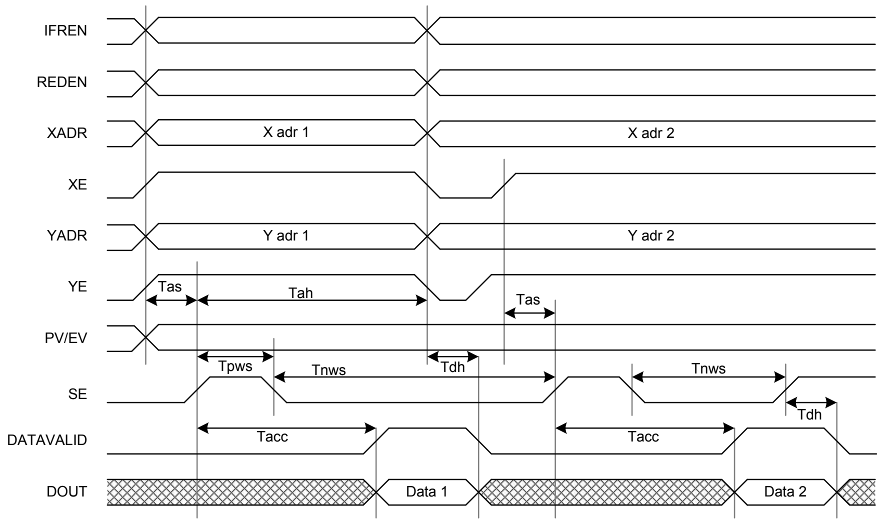


Figure 8.2: *Flash read timing diagram*

InitialSetup has an input channel called *Start*. The active port for this channel is controlled from the testbench outside the Balsa block. Upon the signal *Start.req*, InitialSetup will take control over the various static signals and set them to their predefined value. The signal *Start.ack* is given when all the data is readout from the flash and the control over the static signals is released.

When the static signals are set, InitialSetup communicates with the Counter using a handshake channel without data called *startCounter*. The signal *startCounter.req* from InitialSetup requests the Counter to start, which in turn acknowledges this request using the signal *startCounter.ack* when all the readouts from the flash are done. The handshake circuit graph for the InitialSetup is provided in Figure 8.3. The Balsa code for InitialSetup is given in Appendix A.2 on page 139.

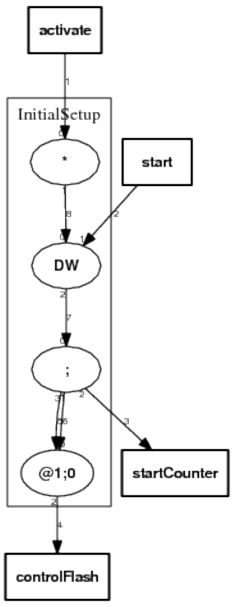


Figure 8.3: *Handshake circuit for the InitialSetup*

8.1.2 Counter

The purpose of the Counter is to control the number of readouts from the flash. Upon a request from the InitialSetup, the counter will start counting. For every value counted, the counter is handshaking the FlashControl with the handshake channel called *count*. This is a data channel containing both the handshake signals and the counter value. For every request from the Counter to the FlashControl i.e *count.req*, the FlashControl executes a single readout from the flash. On completion of the readout the FlashControl acknowledges *count.req* with *count.ack*.

When the counter reaches the predefined value given by the parameter *NUM_COUNT*, an acknowledgement is sent back to the InitialSetup signalling the completion of all the flash readouts.

Given below is an excerpt from Balsa code for the Counter.

```
begin
  select start then
    loop while count_rg < NUM_COUNT then
      tmp := (count_rg + 1 as COUNTER_WIDTH);
      count_rg :=tmp;
      count <- count_rg
    end
  end
end
```

The handshaking with the InitialSetup block is done with the code:

```
  select start then
    ...
  end
```

'`select <channel> then ... begin`' is a handshake enclosure. It is generating a passive input port waiting for a request on the channel *channel*. When the code within the '`then-end`' is complete, an acknowledgement will be given on the *channel*.

'`1 as COUNTER_WIDTH`' is a casting operation to cast the value 1 from a 32 bits integer to the same width as *counter_rg*.

'`count <- count_rg`' is initializing a handshaking on the channel *count*. First the value from the *count_rg* is copied to the data in the channel *count*, followed by a request on *count.req*. Then the code will be waiting until an acknowledgement on *count.ack* is received. All the handshaking is handled by the backend tool for Balsa without the designer having to think about it while writing the code. This exhibits a major strength of Balsa.

The handshake circuit graph for the Counter is provided in Figure 8.4 on the next page. The complete Balsa code for the counter can be found in Appendix A.3 on page 140.

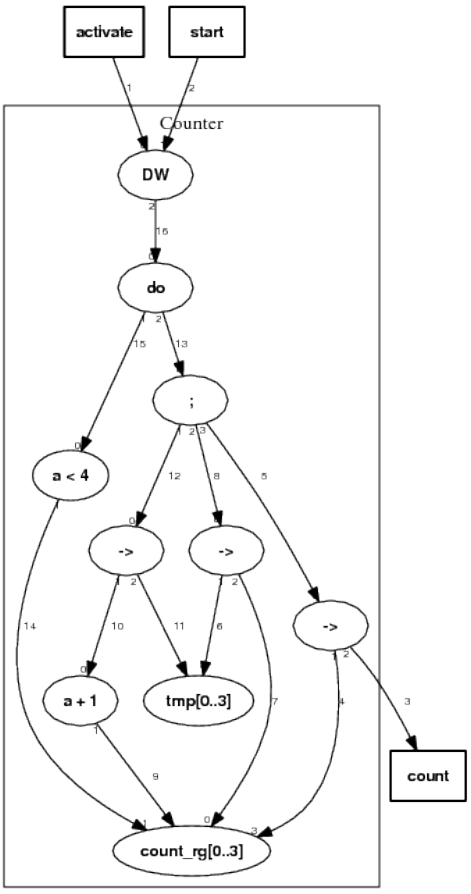


Figure 8.4: *Handshake circuit for the Counter*

8.1.3 FlashControl

The FlashControl is responsible for the actual readout from the flash. The selection of this type of flash is based on what Nordic Semiconductor is using in their design. This would make power comparisons between the asynchronous flash readout and the more traditional synchronous flash readout possible. The flash is from Nordic Semiconductor and the timing waveforms for the read cycle are as previously shown in Figure 8.2.

The flash being adopted for this design is normally used for a clocked operation handling all the timing requirements and the order of operations. On the other hand, Balsa or asynchronous design in general, does not use a clock and must adopt a handshaking methodology to ensure the correctness of timing and operations.

As mentioned in Section 8.1.1 on page 78, some of the signals are static and not a part of the actual readout process. As seen in Figure 8.2, *XE* and *YE* are inactive between consecutive readouts. However, this is not a requirement for the readout from the flash. These signals can be static all along even when the addresses i.e. *XADDR* and *YADDR* are changing. All the static signals are considered to be stable for the entire readout period and will not be discussed any further. Also, *XADDR* and *YADDR* are going to be treated as a single address i.e. *ADDR* and will not be discussed separately.

The flash readout is initialised by setting the address (*ADDR*) which must be followed by a transition from 0 to 1 on *SE* (Sense Amplifier Enable). *ADDR* and *SE* must be separated by a minimum of T_{as} which is 0.1 ns. After the maximum time of T_{acc} i.e. 30 ns, *DATAVALID* will be set indicating that *DOUT* has valid data. *DOUT* will remain valid until the address is changing plus the time T_{dh}. T_{dh} is a minimum of 0.5 ns. To get the *DOUT* associated with the new address, a new transition from 0 to 1 on *SE* must be given.

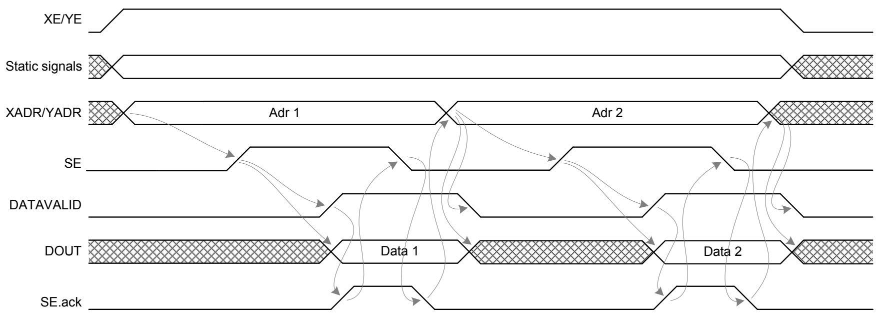


Figure 8.5: *Simplified flash read timing diagram*

Figure 8.5 shows a simplified version of the timing diagram in Figure 8.2, for the asynchronous flash readout. It excludes the static signals mentioned earlier. As seen from the figure, *SE*, *SE.ack* and *DOUT* behave like normal four-phase handshaking channels with data, where *SE* is the request, *SE.ack* is the acknowledgement and *DOUT* represents the data in the channel. Also, as seen from the figure, the address given by *ADDR.data* must stay valid during the entire readout period of the data, i.e. the address must be unchanged during the entire four-phase handshaking for the channel *SE*. Therefore, the lowering of the *SE.ack* will be used as information that the address can change. As will be explained later, this is not done by connecting *SE.ack* to the acknowledgement for *ADDR*, but it is handled internally in the Balsa code block for the *ADDR* and *SE*. Instead, the acknowledgement for the *ADDR*, i.e *ADDR.ack*, is only used as information that the address is received in the flash. However, for the *ADDR* there does not exist an acknowledgement from the flash that indicates that the address is received and set. Instead, the request for the channel *ADDR*, i.e. *ADDR.req*, is simply looped back from the FlashWrapper, through a delay element, as *ADDR.ack*. The delay element is used for fulfilling the timing requirement T_{as} from when the address is given until the signal *SE* can go high.

After the acknowledgement signal *ADDR.ack* is received in the FlashControl, a request is sent on the data channel *SE*. The *SE.req* is connected to the *SE* (Sense Amplifier Enable) input of the flash. This turns on the sense amplifier inside the flash, and after a given period of time, T_{acc}, the data stored at the address *ADDR* is presented on the data output *DOUT*. *DOUT* is connected to the data of the channel *SE*, i.e. *SE.data*. At the same time as *DOUT* becomes valid, *DATAVALID* goes high. *DATAVALID* remains high as long as *DOUT* is valid. When both *SE* and *DATAVALID* are high, *SE.ack* is high. Why *DATAVALID* can not be used as the acknowledgement is explained later in this section. When the data from the flash is received in the FlashControl, it is sent back to the FlashWrapper on one of the four *dataRead* channels and is stored in the FlashWrapper for later use. The storage of the data is the end of a read cycle. After this, the FlashControl signals back to the Counter that the complete data readout has finished.

Designing the FlashControl was a challenging part of the implementation, hence a significant amount of time was spent on writing it. This design has a strict requirement of having the address valid for the entire readout period. The lack of a proper handshaking (acknowledgement) signal to *ADDR* made it difficult to meet this criteria. Also, the minimum timing requirement (of 0.1 ns) between *ADDR* and SE had to be fulfilled. Several solutions were tried, example, insertion of buffers in the Balsa code being one of them. This however did not turn out to be a comprehensive solution, since the delay through a buffer in Balsa code is very hard to predict. A normal Verilog delay element was chosen to be used in the FlashWrapper instead in order to fulfil the timing requirement. This has been discussed in more detail in the remaining part of this section in addition to the main blocks of the code for the FlashControl.

The first main block of the FlashControl, as seen in the code listing 8.1 on the next page, is a mapping from the counter value [1, 2, ...] to the actual addresses in the flash. The flash used is reading out 32 bits at a time, 4 bytes, therefore the address is increasing by 4 every time. Actually, the addresses could be anything, and in the future implementation of the asynchronous flash readout these addresses would be defined by a parameter into the Balsa code, and not hardcoded into the code.

Listing 8.1: *First main block*

```
  loop
   select count then
     case count of
        1 then tmpAdr <- (4 as  ADDR_BITS)
     |  2 then tmpAdr <- (8 as  ADDR_BITS)
     |  3 then tmpAdr <- (12 as ADDR_BITS)
     |  4 then tmpAdr <- (16 as ADDR_BITS)
        else  continue
     end
   end
 end
```

As mentioned earlier, '`select <channel> then ... end`' is a handshake enclosure. It is waiting for an *count.req* before continuing executing the '`then ... end`' code block. In the next line where *count* is used, it is the *data* of the *count* that is used; it is doing a '`case`' based on the value in the channel *count*.

The format of the '`case`' statement in Balse is:

```
case count of
  1 then ...
| 2 then ...
| 3 then ...
  else ...
end
```

On the first match, it will do the command after the '`then`', and then exit the '`case`' statement. If there is no match, it will do the command after the '`else`' and then exit.

The '`tmpAdr <- (4 as ADDR_BITS)`' is using the operator '`<-`' to transfer a result from an expression to a handshake channel. Balsa will internally do the needed handshaking with request and acknowledgement. So in this given command, the value 4, casted to the correct size *ADDR_BITS*, is transferred to the channel named *tmpAdr* before Balsa is doing the handshaking.

Between the two main blocks, there are a '||'. This is a parallel composition operator, meaning that the two blocks will run in parallel. The opposite is the sequence operator '`;`' that means that the clocks will run in sequence.

Listing 8.2: *Second main block*

```
loop
  tmpAdr -> then
    adr <- tmpAdr;
    SE -> dataRead;
    case writeCounter of
       0 then dataRead_0 <-  dataRead
    |  1 then dataRead_1 <-  dataRead
    |  2 then dataRead_2 <-  dataRead
    |  3 then dataRead_3 <-  dataRead
    end;
    writeCounter := (writeCounter + 1 as 2 bits)
  end
end
```

The second main block, listing 8.2, is starting with an handshake enclosure, waiting on a request on *tmpAdr*. Then the data from *tmpAdr* is copied to the channel *adr*, and Balsa is doing the handshaking. After this command there it a '`;`', meaning that the handshaking of the *adr* must be finished before the next command is starting. By delaying the acknowledgement from the FlashWrapper, the delay between setting the address and setting *SE* can be controlled. It is easier to control the delay by inserting a delay element in the Verilog code, than adding buffers in the Balsa code. The actual delay added by the buffers in the Balsa code is hard to predict.

The command '`->`' is handshaking data transfer to a variable from an input port. '`SE -> dataRead`' is setting *SE.req*, asking for data on *SE.data*. When the data is ready, the *SE.ack* will be given, indicating the data is ready. The data from *SE* is then copied into the variable *dataRead* before the *SE.ack* is lowered again. This is followed by a case statement where *dataRead* is written to one of the channels *dataRead_n*, where *n* is ranging from 0 to 3. *dataRead_n* is going to the FlashWrapper, where the data is stored for later use.

Since the '`tmpAdr -> then ... end`' is a handshake enclosure, the data in *tmpAdr* is valid during the entire sequence. This will also make the data in *adr* valid in the same period. Hence, the address is valid during the entire readout from the flash, which is required for a successful readout. At the same time, for the '`SE -> dataRead`' to be complete, the four phase handshaking of SE needs to be completed. From the timing diagram in Figure 8.2 on page 78, *DATAVALID* will stay high until the address is changing. Since, the handshaking of SE must be finished before the address can change, *DATAVALID* cannot directly be used as the acknowledgement for channel *SE*. As long as the *SE.req* is high during the entire readout phase, doing an logical AND of *DATAVALID* with *SE.req*, as seen in the code below, will provide the required acknowledgement for SE, as seen in Figure 8.5 on page 82.

```
assign SE_Ack = SE_Req \& dataValid;}
```

The handshake circuit graph for the FlashControl is provided in Figure 8.6. The complete Balsa code for the FlashControl can be found in Appendix A.4 on page 141.

The data channel *adr* is an output port sending data to the FlashWrapper while the data channel *SE* is an input port receiving the *DOUT* from the flash. Both of them are active ports seen from the FlashControl.

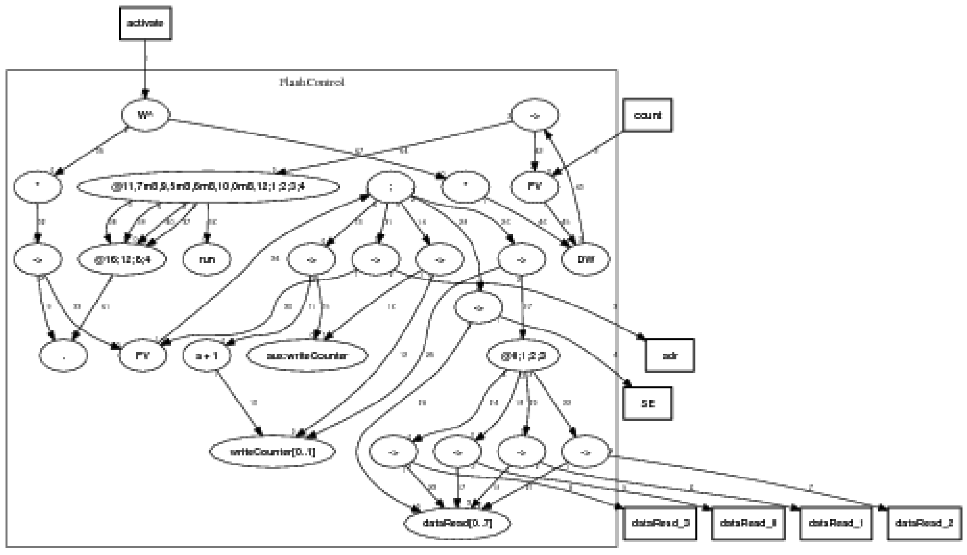


Figure 8.6: *Handshake circuit for the FlashControl*

8.1.4 FlashWrapper

The FlashWrapper serves as an interface between Balsa's asynchronous handshaking and the normal synchronous interface of the flash. It is also where there data will be stored for later use after it is read from the flash. The complete Balsa code for the FlashWrapper can be found in Appendix A.5 on page 143 for the single rail version and in Appendix A.6 on page 146 for the dual rail version.

As explained earlier in Section 8.1.3 on page 81, the actual flash does not have any signal which can be used as an acknowledgement to *adr.req*. Instead the request is passing through a delay element and is looped back as the acknowledgement. The length of the delay is used to control the time between setting the address and giving a transition from 0 to 1 on SE. The flash comprises of two separate address buses i.e. *XADDR* and *YADDR*. On the other hand, the address in the channel *adr* is represented by a single bus. Therefore, the address in the channel *adr* has to be split between *XADDR* and *YADDR*. The split ratio between *XADDR* and *YADDR* are determined by the parameters *ADDR_X_WIDTH* and *ADDR_Y_WIDTH* respectively.

The request (*SE.req*) and data (*SE.data*) in the channel *SE* are connected directly to the flash. *SE.req* is connected to SE in the flash while *SE.data* is connected to DOUT. *SE.ack* is connected to the logical AND between *SE.req* and DATAVALID.

As mentioned in Section 8.1.1 on page 78, the static signals in to the flash are controlled in the FlashWrapper. If the signal *controlFlash* is low, all the signals into the flash are controlled by the normal synchronous design, e.g. a synchronous logic reading and writing to the flash. When the *controlFlash* is high, all the static signals are locked to a predefined value, and all the other control signals are controlled by the asynchronous Balsa flash readout.

After the FlashControl has read out the data, it is sending it back to the FlashWrapper. The FlashWrapper is storing the data in latches for it to be accessible for later use.

The complete figure for the design implementation can be seen in Figure 8.7 and the handshake circuit graph can be seen in Figure 8.6 on the preceding page. The handshake circuit is only for the asynchronous flash readout, not including the FlashWrapper.

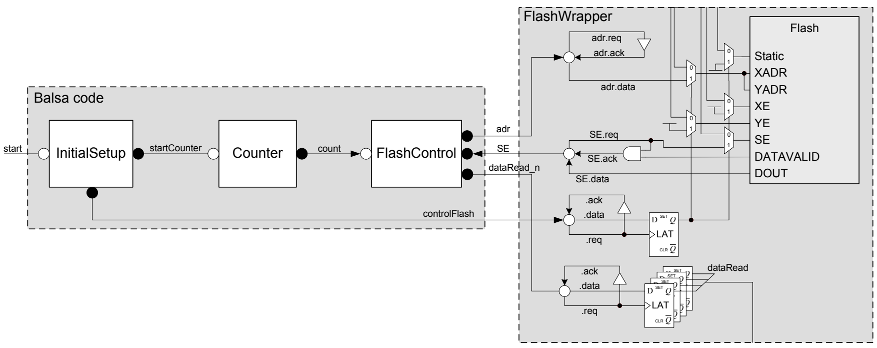


Figure 8.7: *Complete block diagram for the asynchronous flash readout*

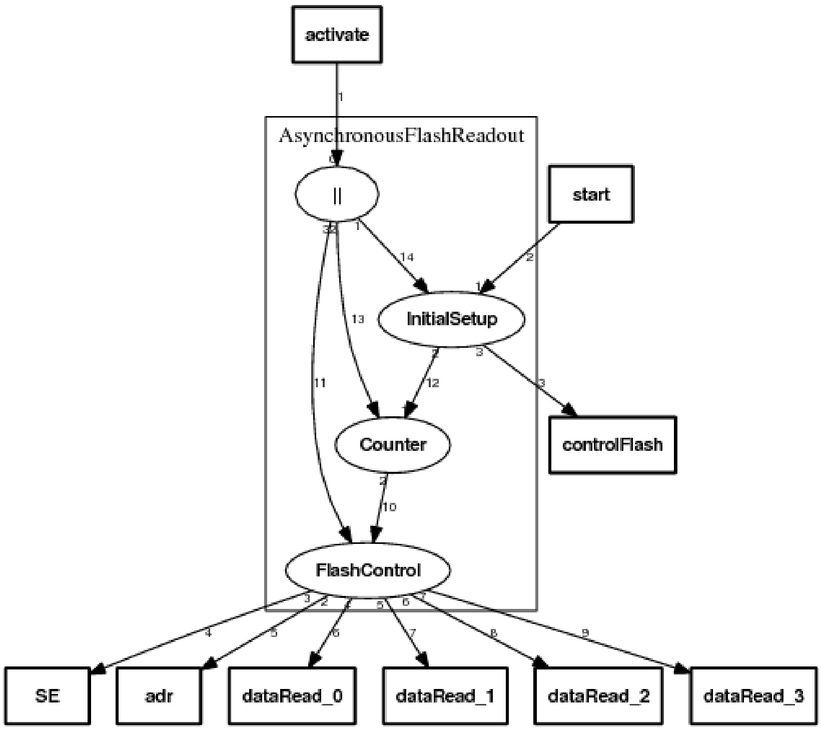


Figure 8.8: *Handshake circuit for the asynchronous flash readout*

8.2 Synthesis and Netlist generation

This section goes on to explain the synthesis and hence the netlist generation by Balsa. As mentioned before in Chapter 7 on page 73, in the Balsa design flow, simulation is not carried out on the code directly, but on the netlist generated by Balsa. Therefore, the netlist is required for the purpose of simulation and also for the back-end work i.e. timing analysis, layout, placement and routing. Balsa is directly generating a Verilog netlist from the Balsa code. Moreover, a Verilog netlist can be generated using different data encoding styles such as: Bundled data, single-rail, 4-phase broad/reduced-broad (i.e. four_b_rb), Dual-rail with broad sync channels (i.e. dual_b), 1-of-4 with dual rail 'odd' bits (i.e. one_of_2_4) and Synchronous style (i.e. sync). Each of these have been explained in Chapter 7.

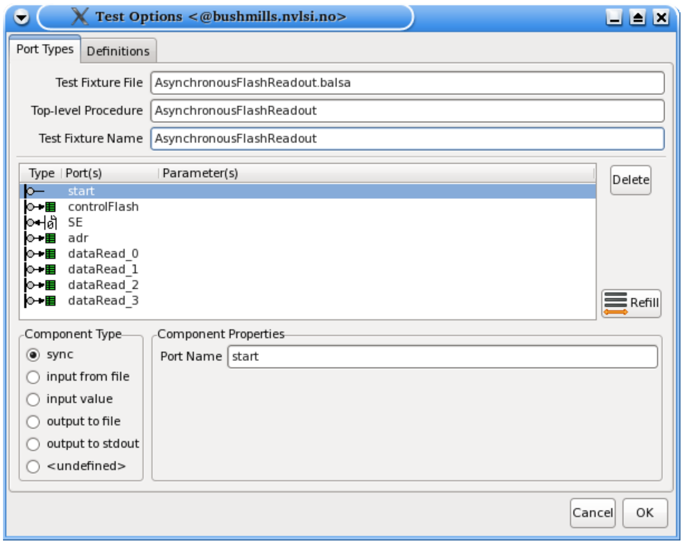
The main goal is to generate a Delay Insensitive (DI) netlist using the dual_b netlist option in Balsa. Since two signals are used to encode every bit in a dual rail netlist, much more effort is required to debug a dual_b netlist. Therefore, the four_b_rb netlist option was used instead while debugging the code to verify its correctness. When the code was verified to be working as desired, the netlist was switched from four_b_rb to dual_b for the final simulation and timing analysis.

Generating the desired netlist in Balsa is a straightforward process using the GUI. The netlists obtained for four_b_rb and dual_b are shown in Appendix A.9 on page 156. The netlists generated are using both standard cells as seen in synchronous logic and non standard cells only used in asynchronous design. Some examples of standard cells are AND, OR, XOR and INV while some examples of non standard cells are the Muller C and its different variations.

8.3 Verification

This section describes the verification of the asynchronous flash readout. First, a testbench automatically generated by Balsa is presented, which only tests the part written in Balsa. This is followed by an explanation for the manually generated Verilog testbench, which is testing the entire design i.e. the Balsa code and the FlashWrapper written in Verilog. As mentioned before, the FlashWrapper consists of a flash model.

8.3.1 Balsa generated automatic testbench


Test Options <@bushmills.nvlsi.no>

Port Types | Definitions

Test Fixture File: AsynchronousFlashReadout.balsa
Top-level Procedure: AsynchronousFlashReadout
Test Fixture Name: AsynchronousFlashReadout

Type	Port(s)	Parameter(s)
	start	
	controlFlash	
	SE	
	adr	
	dataRead_0	
	dataRead_1	
	dataRead_2	
	dataRead_3	

Component Type: [✓ sync] input from file, input value, output to file, output to stdout, <undefined>

Component Properties
Port Name: start

Figure 8.9: *Generating an automatic testbench in Balsa*

Balsa incorporates the possibility of a generating a generic testbench as shown in Figure 8.9. This testbench is able to drive all the inputs and outputs with their corresponding requests, acknowledgements or data. The testbench generated by Balsa is in Verilog. This makes it possible to simulate with this testbench in other simulators supporting Verilog like Mentor Graphics QuestaSim. The complete testbench generated by Balsa is presented in Appendix A.10 on page 156. Furthermore, Balsa has a built in simulator. An example of a simulation session of the Balsa generated testbench in the Balsa simulator can be seen in Figure 8.10. However, at Nordic Semiconductor, QuestaSim is the simulator being employed for the purpose of verification. Moreover, QuestaSim is a much more mature and advanced simulator as compared to the Balsa simulator. Hence, QuestaSim is the preferred choice to carry out simulation. Thus, the remaining part of the thesis will only use QuestaSim for all simulation purposes.

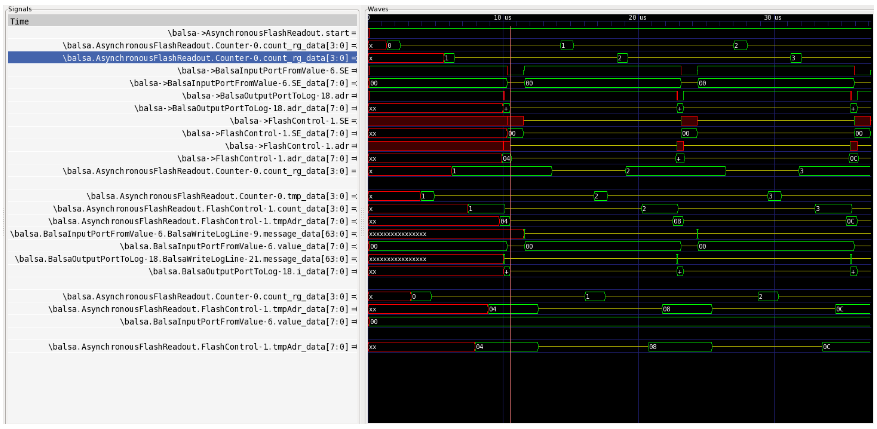

Signals
Time
\balsa->AsynchronousFlashReadout.start =
\balsa.AsynchronousFlashReadout.Counter-0.count_rg_data[3:0] =
\balsa.AsynchronousFlashReadout.Counter-0.count_rg_data[3:0] =
\balsa->BalsaInputPortFromValue-6.SE =
\balsa->BalsaInputPortFromValue-6.SE_data[7:0] =
\balsa->BalsaOutputPortToLog-18.adr =
\balsa->BalsaOutputPortToLog-18.adr_data[7:0] =
\balsa->FlashControl-1.SE =
\balsa->FlashControl-1.SE_data[7:0] =
\balsa->FlashControl-1.adr =
\balsa->FlashControl-1.adr_data[7:0] =
\balsa.AsynchronousFlashReadout.Counter-0.count_rg_data[3:0] =
\balsa.AsynchronousFlashReadout.Counter-0.tmp_data[3:0] =
\balsa.AsynchronousFlashReadout.FlashControl-1.count_data[3:0] =
\balsa.AsynchronousFlashReadout.FlashControl-1.tmpAdr_data[7:0] =
\balsa.BalsaInputPortFromValue-6.BalsaWriteLogLine-9.message_data[63:0] =
\balsa.BalsaInputPortFromValue-6.value_data[7:0] =
\balsa.BalsaOutputPortToLog-18.BalsaWriteLogLine-21.message_data[63:0] =
\balsa.BalsaOutputPortToLog-18.i_data[7:0] =
\balsa.AsynchronousFlashReadout.Counter-0.count_rg_data[3:0] =
\balsa.AsynchronousFlashReadout.FlashControl-1.tmpAdr_data[7:0] =
\balsa.BalsaInputPortFromValue-6.value_data[7:0] =
\balsa.AsynchronousFlashReadout.FlashControl-1.tmpAdr_data[7:0] =
Waves
10 us
20 us
30 us

Figure 8.10: *Simulation in Balsas simulator*

The same testbench was then loaded into QuestaSim without any modification. The result from this simulation is shown in Figure 8.11 on the next page. In this figure, the communication with the flash can be easily seen from the channels *adr* and *SE* where '_0r' represents the request, '_0a' represents the acknowledgement and '_0d' represents the data for the corresponding channels. Seen horizontally, the four pulses (0 to 1 to 0 transitions) on the channels *adr* and *SE* are representing the four readouts from the flash. Furthermore, the figure consists of signals which are not present in the code i.e. *initialise* and *activate*. These signals are automatically generated and added by Balsa in the synthesis. *initialise* is used as a reset and is not a handshake channel. If though the design lacks flip-flops, a reset is required for resetting some other cells like the Muller C elements. *activate* acts like an enable signal for the design, starting the execution of the asynchronous logic. The simulation also comprises of a *start* from the code which is used to start the asynchronous flash readout. When all the four readouts from the flash are complete, an acknowledgement is given on *start*.

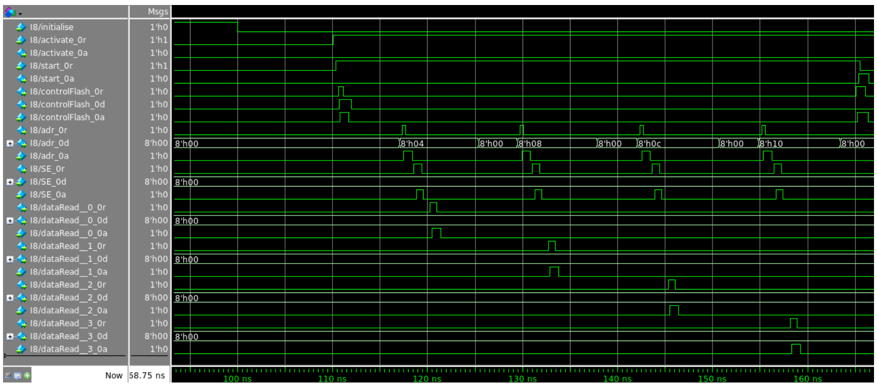


Figure 8.11: *Simulation In QuestaSim with Balsas testbench*

Figure 8.12 shows a zoomed in part of Figure 8.11. As seen, it is easier to see the readout sequence here. First, the address is set before the *adr.req* is asserted. After some time an *adr.ack* is given by the balsa generated testbench. After the four-phase handshaking on channel *adr* is done, *SE.req* is asserted. The balsa generated testbench is not driving *SE.data* to anything else than zero. Therefore, this signal will not change during the entire simulation. On the other hand, the *SE.ack* is given by the testbench as a response to the *SE.req*. At the end, the handshaking of the channel *dataRead_0* is seen. As seen in Figure 8.12, the address given to the flash i.e. *adr_0d*, is kept stable until the handshaking of the *adr* and *SE* channels is complete. This is a requirement given by the flash for a successful readout.

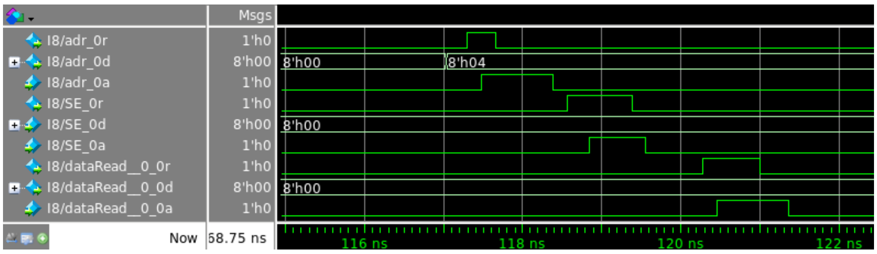


Figure 8.12: *Simulation In QuestaSim with Balsas testbench, zoomed in*

8.3.2 Manually generated Verilog testbench

Figure 8.13 on the next page shows the test environment for the asynchronous flash readout. As mentioned before, the asynchronous flash readout is using a combination of Balsa logic and Verilog logic. A hierarchical level called ’Top’ is used to connect the netlist from the Balsa logic and the FlashWrapper as seen in Figure 8.13. The automatically generated Balsa testbench is not suitable for simulating this new level i.e. ’Top’. This is because Balsa is only capable of generating a testbench for the Balsa part. The FlashWrapper is however not catered for using this testbench. Hence, a manually written testbench is used for simulating ’Top’. The testbench provides stimuli to ’Top’, to verify that it behaves as desired. The testbench code for the asynchronous flash readout has been provided in Appendix A.11 on page 156 for the single rail netlist and Appendix A.12 on page 159 and for the dual rail netlist. An Register-Transfer Level (RTL) simulation is conducted and a wave is produced that verifies the module functionality.

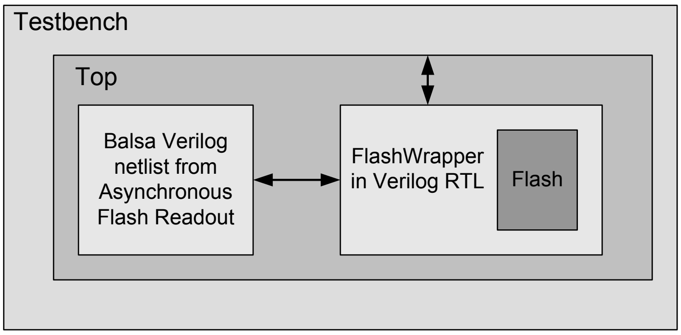

Testbench
Top
Balsa Verilog netlist from Asynchronous Flash Readout
FlashWrapper in Verilog RTL
Flash

Figure 8.13: *Testbench environment for the asynchronous flash readout*

8.3.2.1 Single rail encoding

This section presents the simulation results for the asynchronous flash readout using the four_b_rb i.e. single rail encoding. In this testbench, the only signals that need to be controlled are: *initialise* used as a reset, *activate* starts the asynchronous logic and *start* used to start the flash readout. After giving the initial start, the asynchronous logic is able to do the complete readout without any interaction from the testbench. This is clearly seen in Figure 8.14 on the facing page. As seen from this figure, there are four separate readouts from the flash. As compared to the Balsa generated testbench, there is a real model mimicking the behaviour of the flash in the manually written testbench. This leads to certain notable differences. Firstly, the FlashWrapper is using the correct timing parameters due to which the readout takes much longer in the manually written testbench. Secondly, the data from the flash i.e. *SE.data* is only valid as long as *DATAVALID* from the flash is high. For the rest of the period where *DATAVALID* is zero, *SE.data* is undefined (X). This is because in the real flash, *DOUT* (connected to *SE.data*) will be unknown during this period. This is different from the Balsa generated testbench, where the *SE.data* is zero for the entire simulation.

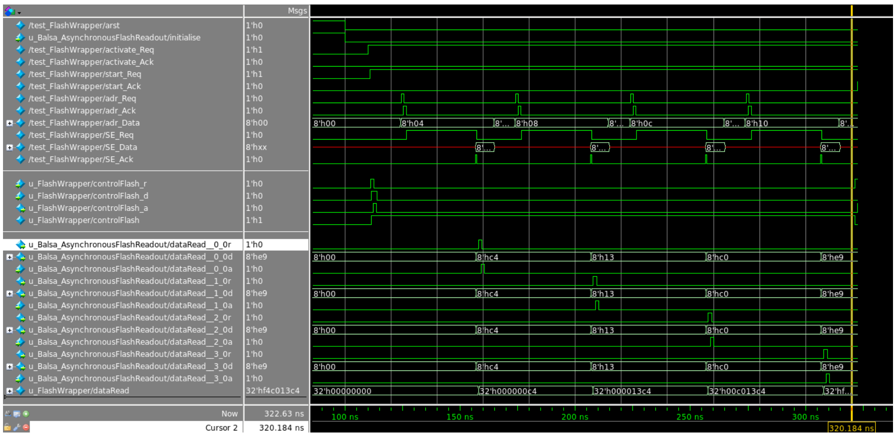

Msgs
/test_FlashWrapper/arst 1'h0
u_Balsa_AsynchronousFlashReadout/initialise 1'h0
/test_FlashWrapper/activate_Req 1'h1
/test_FlashWrapper/activate_Ack 1'h0
/test_FlashWrapper/start_Req 1'h1
/test_FlashWrapper/start_Ack 1'h0
/test_FlashWrapper/adr_Req 1'h0
/test_FlashWrapper/adr_Ack 1'h0
/test_FlashWrapper/adr_Data 8'h00
/test_FlashWrapper/SE_Req 1'h0
/test_FlashWrapper/SE_Data 8'hxx
/test_FlashWrapper/SE_Ack 1'h0
u_FlashWrapper/controlFlash_r 1'h0
u_FlashWrapper/controlFlash_d 1'h0
u_FlashWrapper/controlFlash_a 1'h0
u_FlashWrapper/controlFlash 1'h1
u_Balsa_AsynchronousFlashReadout/dataRead__0_0r 1'h0
u_Balsa_AsynchronousFlashReadout/dataRead__0_0d 8'he9
u_Balsa_AsynchronousFlashReadout/dataRead__0_0a 1'h0
u_Balsa_AsynchronousFlashReadout/dataRead__1_0r 1'h0
u_Balsa_AsynchronousFlashReadout/dataRead__1_0d 8'he9
u_Balsa_AsynchronousFlashReadout/dataRead__1_0a 1'h0
u_Balsa_AsynchronousFlashReadout/dataRead__2_0r 1'h0
u_Balsa_AsynchronousFlashReadout/dataRead__2_0d 8'he9
u_Balsa_AsynchronousFlashReadout/dataRead__2_0a 1'h0
u_Balsa_AsynchronousFlashReadout/dataRead__3_0r 1'h0
u_Balsa_AsynchronousFlashReadout/dataRead__3_0d 8'he9
u_Balsa_AsynchronousFlashReadout/dataRead__3_0a 1'h0
u_FlashWrapper/dataRead 32'hf4c013c4
Now 322.63 ns
Cursor 2 320.184 ns
8'h00 8'h04 8'h08 8'h0c 8'h10
8'h00 8'hc4 8'h13 8'hc0 8'he9
32'h00000000 32'h000000c4 32'h000013c4 32'h00c013c4 32'hf...
100 ns 150 ns 200 ns 250 ns 300 ns
320.184 ns

Figure 8.14: *Simulation In QuestaSim for single rail with testbench written in Verilog*

Figure 8.15 is the zoomed figure to show a single readout. It can be noticed that *SE.data* is giving some real data from the flash during the readout period. After analysing Figure 8.14 and Figure 8.15, it can be concluded that the asynchronous flash readout is working as expected for the four_b_rb (single rail) encoding. The next step involves conducting the same simulation for the dual_b (dual rail) encoding.

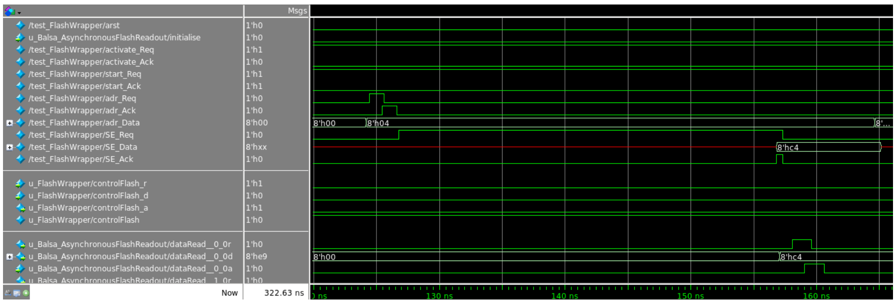

Msgs
/test_FlashWrapper/arst 1'h0
u_Balsa_AsynchronousFlashReadout/initialise 1'h0
/test_FlashWrapper/activate_Req 1'h1
/test_FlashWrapper/activate_Ack 1'h0
/test_FlashWrapper/start_Req 1'h1
/test_FlashWrapper/start_Ack 1'h1
/test_FlashWrapper/adr_Req 1'h0
/test_FlashWrapper/adr_Ack 1'h0
/test_FlashWrapper/adr_Data 8'h00
/test_FlashWrapper/SE_Req 1'h0
/test_FlashWrapper/SE_Data 8'hxx
/test_FlashWrapper/SE_Ack 1'h0
u_FlashWrapper/controlFlash_r 1'h1
u_FlashWrapper/controlFlash_d 1'h0
u_FlashWrapper/controlFlash_a 1'h1
u_FlashWrapper/controlFlash 1'h0
u_Balsa_AsynchronousFlashReadout/dataRead__0_0r 1'h0
u_Balsa_AsynchronousFlashReadout/dataRead__0_0d 8'he9
u_Balsa_AsynchronousFlashReadout/dataRead__0_0a 1'h0
Now 322.63 ns
8'h00 8'h04
8'hc4
8'h00 8'hc4
130 ns 140 ns 150 ns 160 ns

Figure 8.15: *Simulation In QuestaSim for single rail with testbench written in Verilog, zoomed in*

8.3.2.2 Dual rail encoding

This section presents the simulation results for the asynchronous flash readout using the dual_b i.e. dual rail encoding. The FlashWrapper is rewritten to handle dual rail data signals instead of the single rail data signals used so far. Apart from this, everything else is the same as before. The simulation was re-run with the new asynchronous flash dual rail netlist. The simulation for this has been shown in Figure 8.16. The four separate readouts are easily recognized in the figure. Figure 8.17 is the zoomed figure to show a single readout for the dual rail simulation. After analysing Figure 8.16 and Figure 8.17, it can be concluded that the asynchronous flash readout is working as expected for the dual_b (dual rail) encoding.

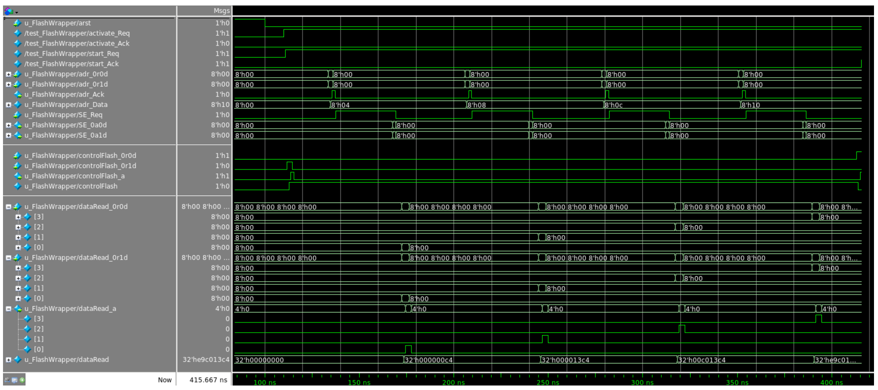

u_FlashWrapper/arst
/test_FlashWrapper/activate_Req
/test_FlashWrapper/activate_Ack
/test_FlashWrapper/start_Req
/test_FlashWrapper/start_Ack
u_FlashWrapper/adr_0r0d
u_FlashWrapper/adr_0r1d
u_FlashWrapper/adr_Ack
u_FlashWrapper/adr_Data
u_FlashWrapper/SE_Req
u_FlashWrapper/SE_0a0d
u_FlashWrapper/SE_0a1d
u_FlashWrapper/controlFlash_0r0d
u_FlashWrapper/controlFlash_0r1d
u_FlashWrapper/controlFlash_a
u_FlashWrapper/controlFlash
u_FlashWrapper/dataRead_0r0d
u_FlashWrapper/dataRead_0r1d
u_FlashWrapper/dataRead_a
u_FlashWrapper/dataRead
Msgs
Now
415.667 ns
100 ns
150 ns
200 ns
250 ns
300 ns
350 ns
400 ns

Figure 8.16: *Simulation In QuestaSim for dual-rail with testbench written in Verilog*

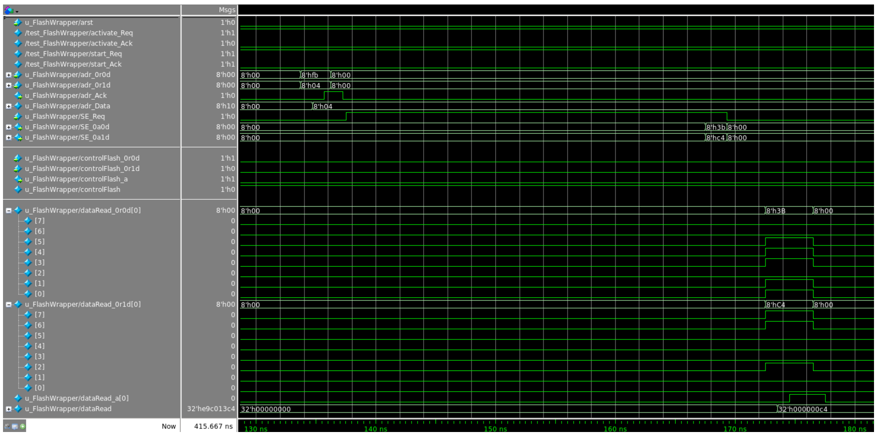

u_FlashWrapper/arst
/test_FlashWrapper/activate_Req
/test_FlashWrapper/activate_Ack
/test_FlashWrapper/start_Req
/test_FlashWrapper/start_Ack
u_FlashWrapper/adr_0r0d
u_FlashWrapper/adr_0r1d
u_FlashWrapper/adr_Ack
u_FlashWrapper/adr_Data
u_FlashWrapper/SE_Req
u_FlashWrapper/SE_0a0d
u_FlashWrapper/SE_0a1d
u_FlashWrapper/controlFlash_0r0d
u_FlashWrapper/controlFlash_0r1d
u_FlashWrapper/controlFlash_a
u_FlashWrapper/controlFlash
u_FlashWrapper/dataRead_0r0d[0]
u_FlashWrapper/dataRead_0r1d[0]
u_FlashWrapper/dataRead_a[0]
u_FlashWrapper/dataRead
Msgs
Now
415.667 ns
130 ns
140 ns
150 ns
160 ns
170 ns
180 ns

Figure 8.17: *Simulation In QuestaSim for dual rail with testbench written in Verilog, zoomed in*

Chapter 9

Balsa technology

The main goal in this chapter is to generate a new technology matching the library used at Nordic Semiconductor. Most of the files in the library called 'BorealisTech' are not provided in the appendix. It contains confidential information from the provider of the standard cell library, and it is not necessary for the purpose of understanding the thesis.

Balsa generates an intermediate technology independent breeze netlist. To generate a Verilog netlist from the breeze netlist, Balsa is able to use different technologies. The different technologies map the breeze netlist to different cell libraries. These libraries could either be for silicon production or FPGA. The technology mapping can be done using: only standard cells, such as AND, OR, XOR and LATCH; custom made cells, like the Muller C; or a combination of standard cells and custom cells. The technology provided by Balsa i.e. balsa-tech-example, is only using standard cells. Even though this technology can be used for synthesis, the netlist generated will be of an inferior quality. This technology only serves as a template for the creation of custom made technologies.

For Balsa to produce a DI netlist, the timing assumptions for all isochronic forks in the breeze components must be met. The example technology provided by Balsa, generates functionally correct cells but does not make sure that the isochronic timing assumptions for the forks are met. Moreover, most of the complex cell structures are constructed by using local feedback loops. Figure 9.1 on the following page shows a Muller C element generated using standard cell (the AO222 cell) in the Balsa example technology. The existence of a feedback loop is clearly visible. Such feedback loops should by far be avoided as they can lead to timing complications in the backend flow. Therefore, the user is expected to generate their own technology, specifically tailored according to their needs and the library used.

Furthermore, all of the technologies presented in the Balsa manual [24], including the example technology, are available in Balsa version 3.5. However, only the example

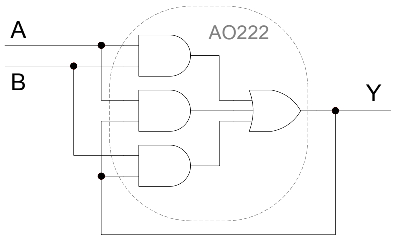


Figure 9.1: *A Muller C element designed using the AO222 standard cell*

technology is working in Balsa version 4.0. It was mentioned on the Balsa webpage [23] that the developer of Balsa could be contacted to obtain technologies to support other cell libraries. Therefore, an email was sent to the developers and they replied by saying that they could not provide other technologies, but the Balsa manual can be referred to generate the required new technology. Hence, a new technology had to be generated for this thesis.

9.1 Generating a new technology

The new technology generated was given the name *BorealisTech*. Given below are the steps that were followed to generate this technology. However, Chapter 9 of [24] can be referred for a more detailed and thorough explanation.

– Initially, a file with the name BorealisTech-cells.net was created. For each of the cells in the target library, an entry in this file had to be created. This entry can either be in the form of a simple list of the inputs and outputs as shown below for the AND gate:

```
(circuit "AN2D0BWP7T"
  (ports
    ("A1" input 1)
    ("A2" input 1)
    ("Z" output 1)
  )
  (nets)
  (instances)
)
```

or in the form of an instruction for Balsa on how to create a more complex structure by connecting several primitive gates together, as shown below in the implementation of a 3 input Muller C using an SR-latch and some other standard primitive gates:

```
; 3 input Muller C
(circuit "MC3X1"
  (ports
    ("Z" output 1)
    ("A" input 1)
    ("B" input 1)
    ("C" input 1)
    ("initialise" input 1)
  )
  (nets
    ("set"    1)
    ("clr"    1)
    ("nReset" 1)
    ("QN"     1)
  )
  (instances
    (instance "RSLATX1" ("Z" ("QN" 0) ("clr" 0) ("set" 0)))
    (instance "AN3D0BWP7T"   ("A" "B" "C" ("set" 0)))
    (instance "AOI31D0BWP7T" ("A" "B" "C" ("nReset" 0) ("clr" 0)))
    (instance "INVD0BWP7T" (("nReset" 0) "initialise"))
  )
  (attributes (global-ports "initialise") (cell-type "helper"))
)
```

To simplify the generation of the file *BorealisTech-cells.net*, the process was automated by making a script which can be found in Appendix C.1 on page 181. In this thesis, three different libraries were used: the standard cell library, the two-input Muller C library and the SR-latch library. Since, the two-input Muller C library and the SR-latch library were not a part of the standard library, they were specially made by the technology group at Nordic Semiconductor for this thesis. All these libraries were combined into a single library file called *BorealisTech-cells.v.* The script automatically extracts the input and output list for the various cells in this library, restructures them into the correct format and places them into the file *BorealisTech-cells.net.* The standard cell library contains many more cell types than that can be used in Balsa. In order to avoid the inclusion of these cells in the file *BorealisTech-cells.net*, another input file called *IncludeOnly* was used to filter the cells that should be included in the file *BorealisTech-cells.net.* The complex structures in the file *BorealisTech-cells.net* were then made manually for the required cells. This was an iterative process i.e. new complex structures were made for the design whenever they were needed.

– The next step was to map all cells that were going to be used in the library by Balsa into something that Balsa understood. This was done in the file called *gate-mapping.* The example given below shows the mapping of the AND gates in the library to those in Balsa.

```
;;; and{n}: out,in1,in2...
```

```
("and2" ("AN2D1BWP7T" 1 2 0)     (0 "AN2D0BWP7T")(1 "AN2D1BWP7T")(2 "AN2D2BWP7T"))
("and3" ("AN3D1BWP7T" 1 2 3 0)   (0 "AN3D0BWP7T")(1 "AN3D1BWP7T")(2 "AN3D2BWP7T"))
("and4" ("AN4D1BWP7T" 1 2 3 4 0) (0 "AN4D0BWP7T")(1 "AN4D1BWP7T")(2 "AN4D2BWP7T"))
```

In the first line of this example, the two-input AND gate *AN2D1BWP7T* of the library is mapped to Balsa's *and2*. '1 2 0' represents the order in which the pins should be connected and '(0 "AN2D0BWP7T")(1 "AN2D1BWP7T")(2 "AN2D2BWP7T")' gives the different drive strengths for the AND gate.

– The next step was to generate the *component.abs* file. This is a file where it is possible to generate different versions of the breeze components in the Balsa language, like the *loop* and *sequencer*. Refer Chapter 13.9 of [24] for a complete list of the breeze components available in Balsa. In this thesis, no special variants of the breeze components were used, therefore only the implementations provided by Balsa were used. Hence, the *components.abs* file is only pointing towards Balsa's standard implementation for breeze components as shown below:

```
;;;
;;; ‘components.abs’
;;; Breeze primitive components for technology verilog
;;;
;;; 10 Aug 2001, Andrew Bardsley
;;;
;;; $Id: components.abs,v 1.6 2002/03/13 15:18:50 bardslea Exp $
;;;

(include tech "common" "components")
```

– Then, the configuration file called *startup.scm* was created and has been provided in Appendix C.2 on page 183 .

– All the generated files such as *BorealisTech-cells.net*, *gate-mapping* and *component.abs* were then placed in the folder *balsa/share/tech/BorealisTech* . The parameter 'BALSATECH' was set to 'BorealisTech'. Then, the following command was run:

```
balsa-make-helpers balsa-cells.net gm
```

Balsa has a long list of cells that it must have in the gate-mappings file. Only some of the required cells in the files *gate-mappings* and *BorealisTech-cells.net* were given manually. Balsa has built in, standard, not optimal implementation for all the cells it needs to use if the user is not defining new implementations. Running the above '`balsa-make-helpers`' command creates two files: *balsa-cells.net* and *gm*. This command generates the remaining contents, i.e. missing cells and mappings, for these files. *balsa-cells.net* is a file with the missing cells for *BorealisTech-cells.net* while *gm* is a file with the missing cells for *gate-mappings*. The contents of the *gm* file must be concatenated to the end of the *gate-mappings* file. By doing so a complete library has now been specified.

– At last, the file *balsa-mgr.cfg* was created. This file was simply copied from the technology example provided by Balsa without any modification. This file is providing balsa-mgr with information on the various options available for the new technology.

Chapter 10
Exploring Balsa's Delay Insensitivity

In order to integrate Balsa with Nordic Semiconductor's design flow, a new Balsa technology is required for the libraries used at Nordic Semiconductor. Chapter 9 describes the generation of this new technology for Balsa.

The objective for this chapter is to verify if and how the Balsa generated netlist is Delay Insensitive (DI). Chapter 8 presents a Balsa design implementation for the asynchronous flash readout. A Balsa netlist for this design is also generated. However, this netlist is quite big, and manually verifying the delay insensitivity for this netlist is too complex and time consuming. Carrying out this experiment on a smaller design was considered to be more feasible. Therefore, the design for the buffer example presented in Section 6.1 on page 59 has instead been used for the delay insensitivity analysis.

10.1 Test Design: Buffer example

Given below is the code from [24] for the buffer used in this test. It has been altered from being an 8 bit wide buffer to a single bit buffer. This is done to simplify the netlist so as to make the manual inspection feasible.

```
import [balsa.types.basic]

procedure buffer1 (input i : 1 bits; output o : 1 bits) is
  variable x : 1 bits
begin
  loop
    i -> x          -- Input  communication
    ;               -- sequence operator
    o <- x          -- Output communication
  end
end
```

This code above will infinitely request for data on the input *i*, transfer it to the internal storage *x*, and request for a handshake to send it out on output *o*. The handshake circuit graph for the given code is shown in Figure 10.1.

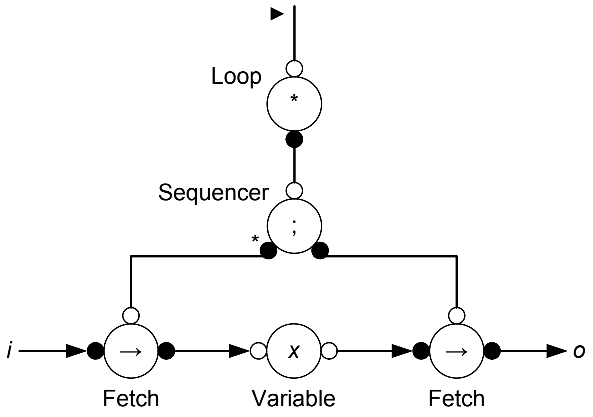


Figure 10.1: *The buffer handshake circuit graph*

In this figure, the big circles are the breeze components. Breeze components can be defined as the handshake components present in the breeze netlist. The lines connecting the different breeze components are handshake channels. The ends of a line are marked by circles indicating a passive or an active port. As mentioned before, a filled circle indicates an active port while an empty circle indicates a passive port. Furthermore, a line with an arrow is a handshake channel with data. The direction of the arrow gives the direction of the data flow. On the hand, a line without an arrow is just a handshake channel without data.

It is easy to recognize the different parts of the code in the handshake circuit graph. Initially, the handshake signal *activate* commences the circuit action. This signal is not included in the code, but is automatically added by Balsa to handshake the different control components. The *activate* signal is connected to the breeze component *loop*. Upon activation, this component will start the handshaking on the active port. When it receives an acknowledgement on the handshake channel, it will automatically start a new handshake session. This will go on forever and the *loop* component by itself will never acknowledge the *activate* signal on its passive input port. The next breeze component is the *sequencer*. Upon receiving a request on its passive input port, the *sequencer* will first do a handshake on the left channel with the star ($\star$), and then do a handshake on the right channel. On successful completion of the handshaking on the right channel, the *sequencer* gives an acknowledgement to the *loop* on its passive port. Upon a request on its passive port, the *fetch* component will request for data from the active data input port on its left, and then send the data out to the active data output port on its right. On successfully sending the data out, an acknowledgement is given on the passive port. The last component is the *variable*. This component has only passive data ports. The port on the left requests to store a value into the *variable* component, while the port on the right requests the value stored in the *variable* component.

To carry this test forward, a normal Balsa synthesis flow, as explained in Section 8.2 on page 88, was followed. Since, Balsa claims the 'dual_b' generated netlist to be delay insensitive, it was selected for the synthesis and a Verilog netlist was produced. This was done to be able to analyze the generated netlist and verify Balsa's claim of DI.

The generated netlist was then manually drawn into the circuit diagram shown in Figure 10.2 on the following page. The netlist code is provided in Appendix B.2 on page 164. The different components in the handshake circuit graph in Figure 10.1 on the facing page are easily noticeable in the circuit diagram. The handshake circuit graph represents the balsa netlist and the circuit diagram represents the Verilog netlist. Balsa makes a one-to-one mapping between the Balsa netlist and the Verilog netlist.

10.2 Circuit operation and delay insensitivity analysis

This section explains the operation of the circuit shown in Figure 10.2 on the next page and analyzes how it is DI. Before getting further deep into the circuit operation or analysis, it is vital to understand the requirements for a circuit to be DI and some general concepts that will serve as basics for further reading. From the discussion in Section 2.9.1 on page 14, it can be concluded that in a purely DI design, none of the delays need to be catered for. However, practically it is almost impossible to achieve a pure DI in a circuit. In practical realizations there exist some paths that must be handled in special way. These paths are usually forks or combinatorial loops. In such cases, all the forks must be made isochronic (refer Section 2.9.2 on page 15) and the combinatorial loops could be controlled by employing a handshaking mechanism. Therefore, if handshaking is employed on a combinatorial loop, no timing assumption needs to be made for that loop. [4] claims that Balsa handles the various combinatorial loops present in the circuit by using handshaking, and if the breeze components are ensured to be QDI by fulfilling the isochronic fork timing assumption, then the design is DI. The breeze components can be made QDI by two methods. The first method involves doing a local timing analysis for isochronic forks, for each breeze component on the netlist. Thus, the isochronic fork assumption must be fulfilled separately in each component by inserting delay elements if necessary to make one path slower than the other. The second method involves making library elements for the different breeze components, where the isochronic fork assumptions

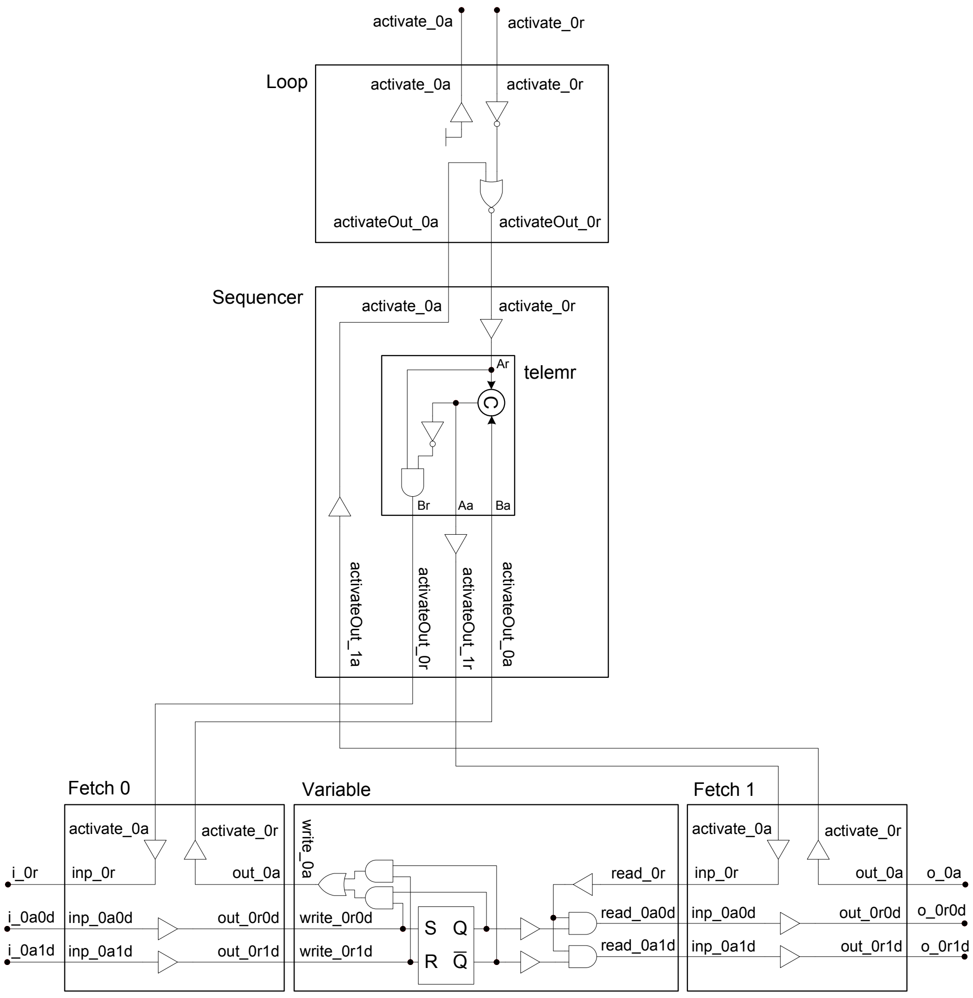


Figure 10.2: *Circuit diagram of the Verilog netlist*

are fulfilled by design. This implies that correct delays must be ensured in the forks while designing the elements.

The analysis of the circuit is done by following the flow of control (handshaking) and data through the circuit, as seen in Figure 10.5 to Figure 10.7. The labels for the various signal names have been removed in these figures, but can be found in Figure 10.2. The green line signifies pure handshake signals. The violet line pair is a dual rail line. It signifies the handshake and data signal coded together in a dual rail line as seen in Figure 10.3 on the next page. At anytime only one line of the dual rail lines can be high. Having both of them high at the same time is an illegal state. For the simplifying the analysis, the breeze component *loop* in Figure 10.2 on the facing page has been changed, as seen in Figure 10.4. Both the left and right circuits are logically equivalent.

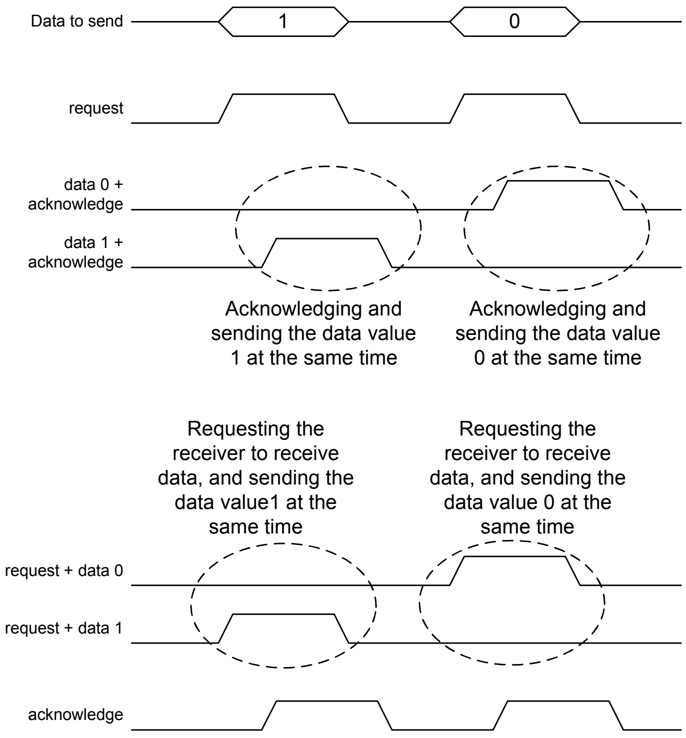

Data to send
1
0
request
data 0 + acknowledge
data 1 + acknowledge
Acknowledging and sending the data value 1 at the same time
Acknowledging and sending the data value 0 at the same time
Requesting the receiver to receive data, and sending the data value1 at the same time
Requesting the receiver to receive data, and sending the data value 0 at the same time
request + data 0
request + data 1
acknowledge

Figure 10.3: *Dual rail encoding*

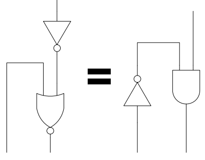


Figure 10.4: *Simplification in the Verilog netlist for the loop component*

10.2.1 Circuit analysis

All the paths in the circuit are initialised to zero.

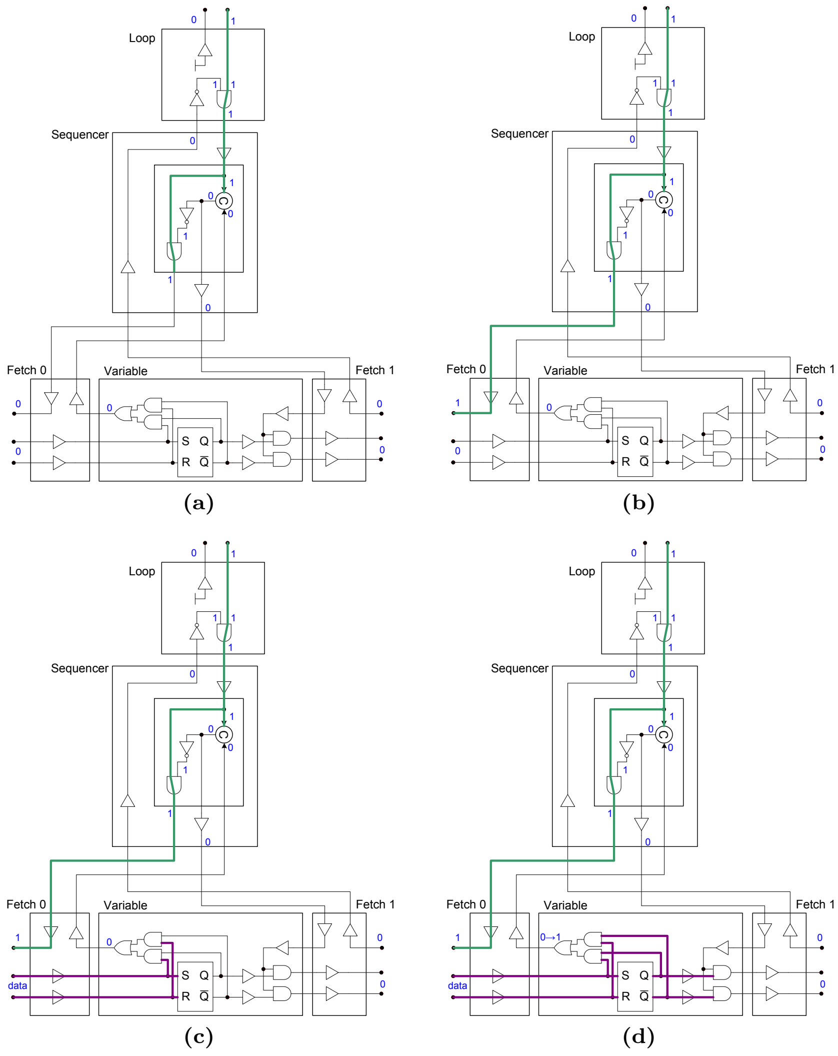


Figure 10.5: *Circuit analysis*

– After the initialization stage, the activate request *activate_0r* into the *loop* component goes high. As seen in Figure 10.5a, the acknowledgement *activate_0a* is never given. This is because the *loop* component runs forever. The *activate_0r* goes through the *loop* component into the *sequencer* and the *t-element* (telemr). Since, the input *Ba* into the *Muller C* element is 0, the *Aa* output from the *t-element* is also 0, while the *Br* output is 1. The *Br* output from the *t-element* becomes the request on the first handshake channel out from the *sequencer*.

– This handshake channel is connected to the activate connection (the *activate_0r* and *activate_0a* pair) of the input *fetch 0* component. The handshake request from the *sequencer* is coming into the *fetch 0* component and goes directly out on the *i_0r*, as seen in Figure 10.5b on the preceding page. This request goes out to the environment and is asking for the data on the dual rail channel pair i.e. either *i_0a0d* or *i_0a1d*. As mentioned before, this dual rail channel pair will contain both the data and the acknowledgement.

– Then, the environment acknowledges the request for the data and will drive either of the lines *i_0a0d* or *i_0a1d* high, as sen in Figure 10.5c on the facing page. The data goes directly out from the *fetch 0* component, and as a request into the *variable* component for sending data.

– Either the *set* or *reset* will go high and cause either Q or $\overline{Q}$ respectively to go high. See Figure 10.5d on the preceding page for reference.

– In the component *variable*, when either both *set* and Q are high or both *reset* and $\overline{Q}$ are high, the *write_0a* goes high. This symbolizes that the data is stored in the latch. The *write_0a* is the acknowledgement from the *variable* component to the *fetch 0* component for receiving the data. This acknowledgement is then further passed on from *fetch 0* to the *sequencer*. This completes the handshake for requesting data. All this has been shown in Figure 10.6a on the following page.

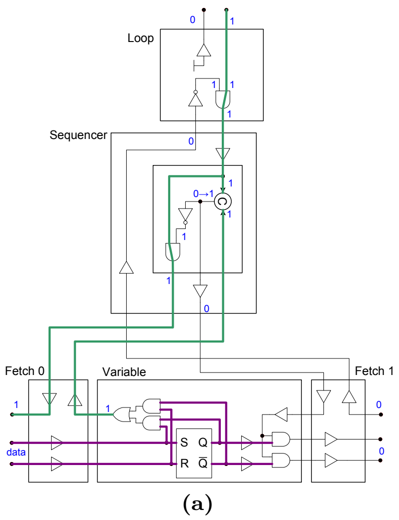
– Both the inputs to the *Muller C* i.e. *Ar* and *Ba* are now 1. Therefore, the output from the *Muller C* becomes 1 as well. This will now turn off the request on the first handshake channel in the *sequencer* and will turn on the request on the second handshake channel. This request goes into the *fetch 1* component. Then, *fetch 1* passes this request on to the *variable* component requesting for data. Refer Figure 10.6b on the next page.

– In Figure 10.6c on the following page, the dual rail channel pair input data is no longer driven, due to the turning off of the request in the first handshake channel. Also, the acknowledgement *write_0a* goes down, followed by the lowering of *activateOut_0a* in the *sequencer*. This completes the four phase handshaking of the first handshake channel.

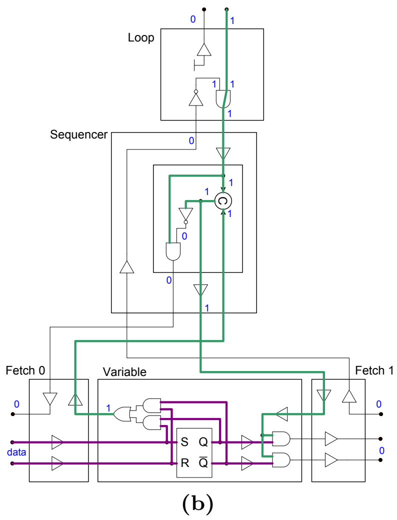
  The request from the *fetch 1* to the *variable* causes the data from *variable* to be sent to *fetch 1* over the dual rail channel pair into the environment.

– In Figure 10.6d on the next page, the acknowledgement from the environment is given to the *fetch 1* component from where it is further passed on the *sequencer*. The sequencer in turn passes the acknowledgement to the *loop*.

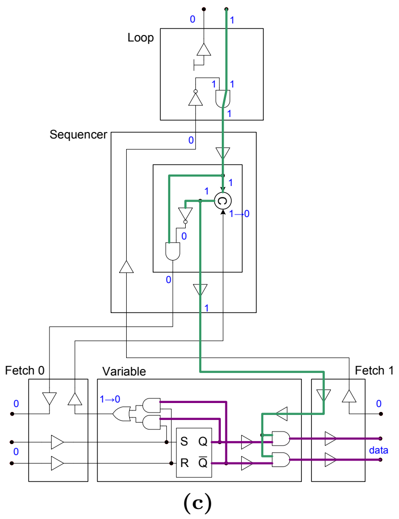
– As seen in Figure 10.7a on page 109, the *activateOut_0a* into the loop element is causing the activate request *activateOut_0r* to go low.



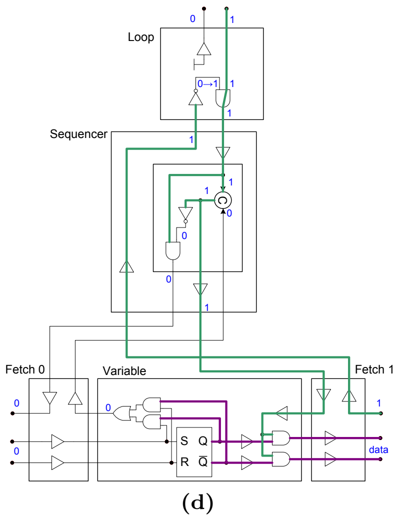
(a)



(b)



(c)



(d)

Figure 10.6: *Circuit analysis*

– Both the inputs to the *Muller C* i.e. *Ar* and *Ba* are now 0, thus making the output from the *Muller C* 0 as well. This turns off the request in the second handshake channel. The dual rail channel pair output data is no longer driven from the *fetch 1* element, due to the turning off of the request in the second handshake channel. Refer Figure 10.7b on the next page.

– As a consequence of this, the acknowledgement *o_0a* is lowered by the envi-

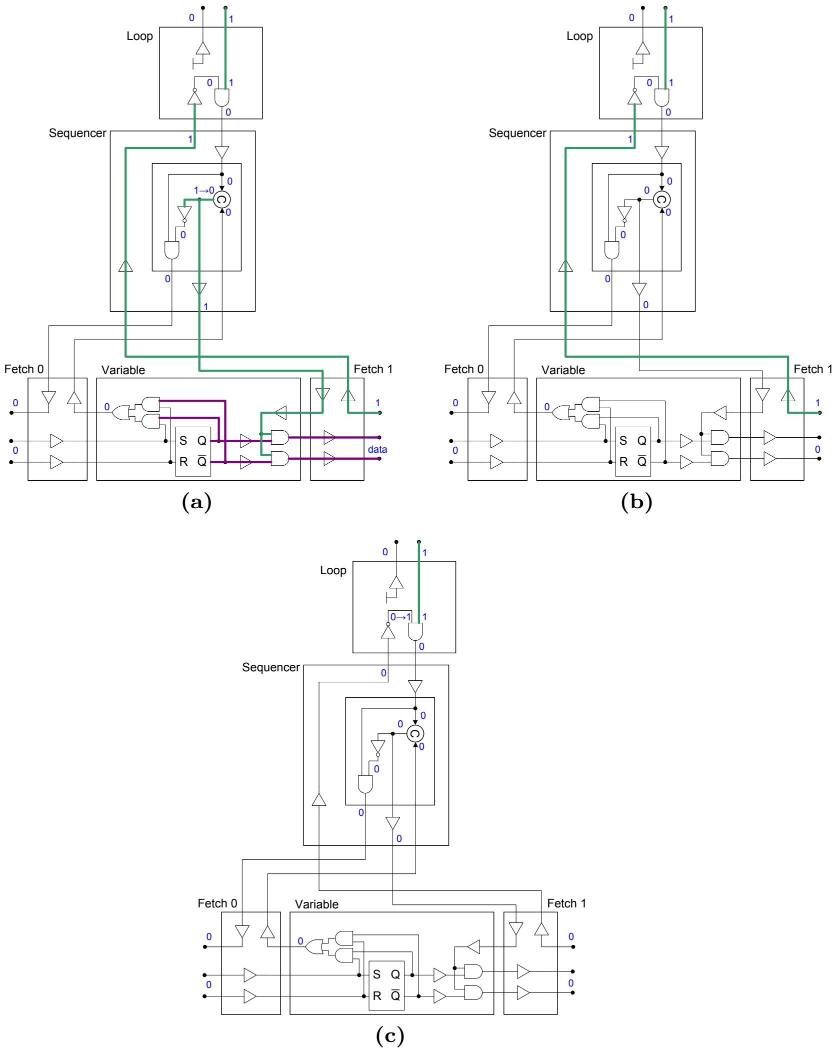


Figure 10.7: *Circuit analysis*

ronment. Refer Figure 10.7c. This acknowledgement is further passed on to the *sequencer* and the *loop*, thus completing the four phase handshaking of the second handshake channel and the four phase handshaking between the *sequencer* and the *loop* respectively. The lowering of *activateOut_0a* in the *loop* turns on *activateOut_0r*. This starts the sequence all over again.

10.2.2 Delay insensitivity analysis

As mentioned in Section 10.2, one of the requirements for a Balsa design to be DI is that all the combinatorial loops in the circuit must be controlled by handshaking signals. This buffer circuit on its own lacks the presence of any combinatorial loops. Therefore, to look into this concept further, it is assumed that this buffer circuit is connected to some other Balsa component attached to the input channel *inp* of the *fetch 0* component or the output channel *out* of the *fetch 1* component. The insertion of this additional circuitry marks the presence of a combinatorial loop as shown in Figure 10.8 with the red color. However, all the combinatorial loops in this figure are formed by the handshaking signals themselves. Since, this is as per the requirement, these combinatorial loops are safe by design and are not breaking the delay insensitivity of this circuit.

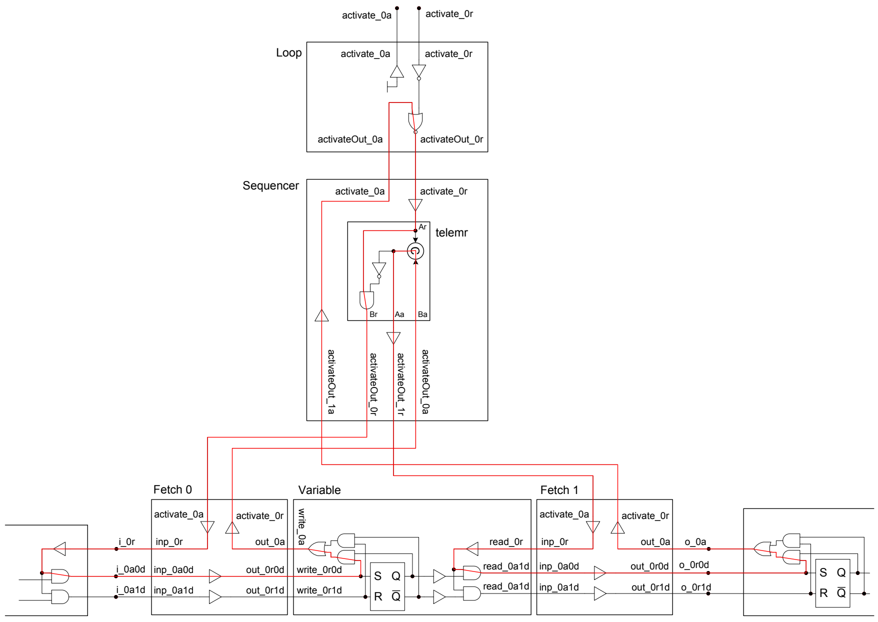


Figure 10.8: *Combinatorial loop in the buffer design*

The second requirement for Balsa's DI operation is that all the components must be ensured to be QDI. This is accomplished by ensuring that the isochronic fork assumption in the components where forks are present must be met. In the buffer circuit, only the components *t-element* (telemr) and *variable* comprise of a fork. Thus, only these elements have to be analysed further.

There exist two forks in the *t-element* (telemr) element: one at the *Ar* input and the other at the output from the Muller C element. The way this element has been used, it is made sure by design that only one line in each of the forks can change at a time. Therefore, this element is safe by design. Also, both the forks are a part of the handshaking and are ensuring the correct order of operation. The working of the telemr element is shown in Figure 10.9. In these figures, the red lines indicate lines which are high (i.e. 1) and the blue lines indicate lines which are low (i.e. 0).

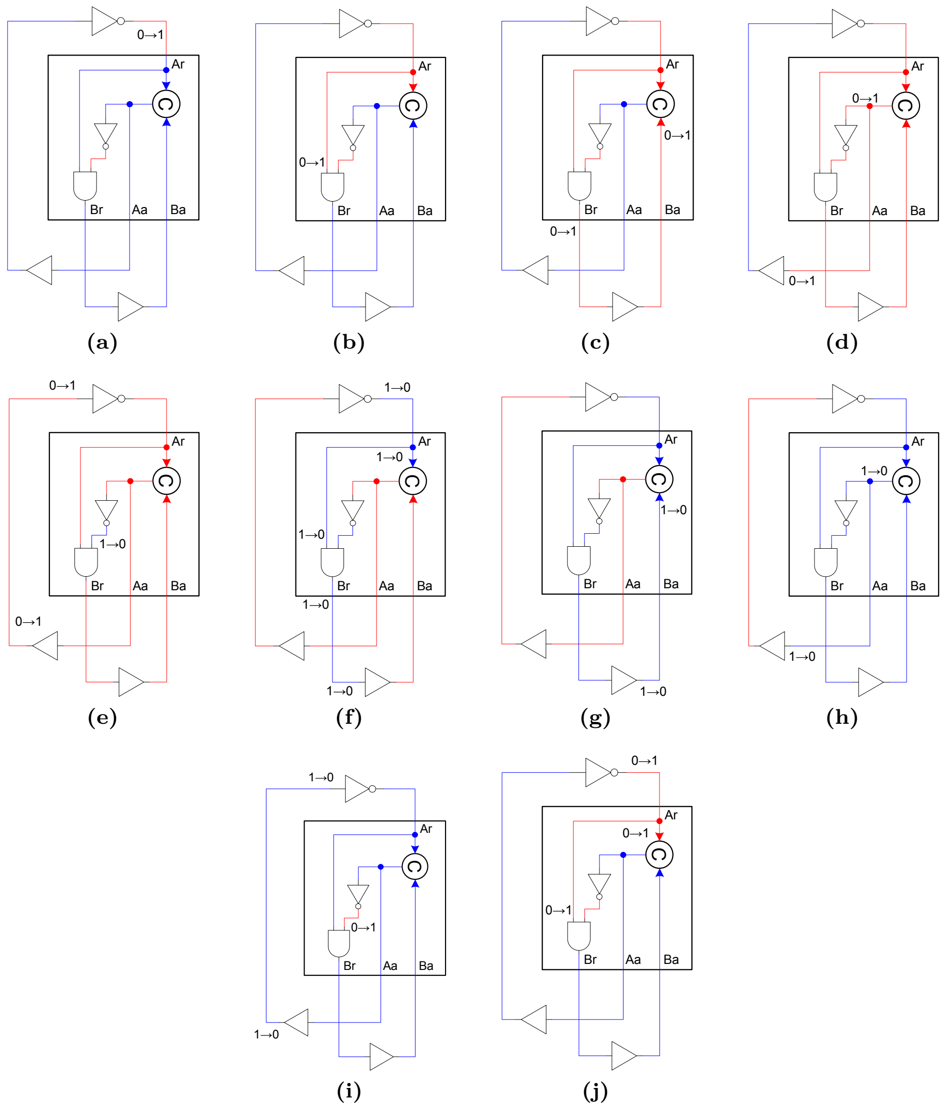


Figure 10.9: *Working of the telemr element*

There also exist two forks in the *variable* component: one at the Q output of the *SR-latch* and the other at the $\overline{Q}$ output of the *SR-latch.* Figure 10.10 highlights the former fork. The latter fork is basically the same but just on the other output of the *SR-latch.* The two paths in the former fork are given by the colours red and blue. There exists a possibility for a hazard in the *variable* component. However, the probability for this hazard to be triggered is extremely less. For this circuit to work, it must be ensured that the red path is slower than the blue path. This is because if the red path is faster than the blue path, the previous old value present on the blue input line to the AND gate will be propagated when the AND gate is opened. This will result in a glitch on the output of the AND gate when the new value finally reaches on the blue input line. This can be accomplished by ensuring that the path from the Q output of the *SR-latch* to output *write_0a* of the *variable* is slower than the blue path. If the same is ensured for the $\overline{Q}$ output of the *SR-latch*, then the *variable* component will be QDI and the circuit will be hazard free.

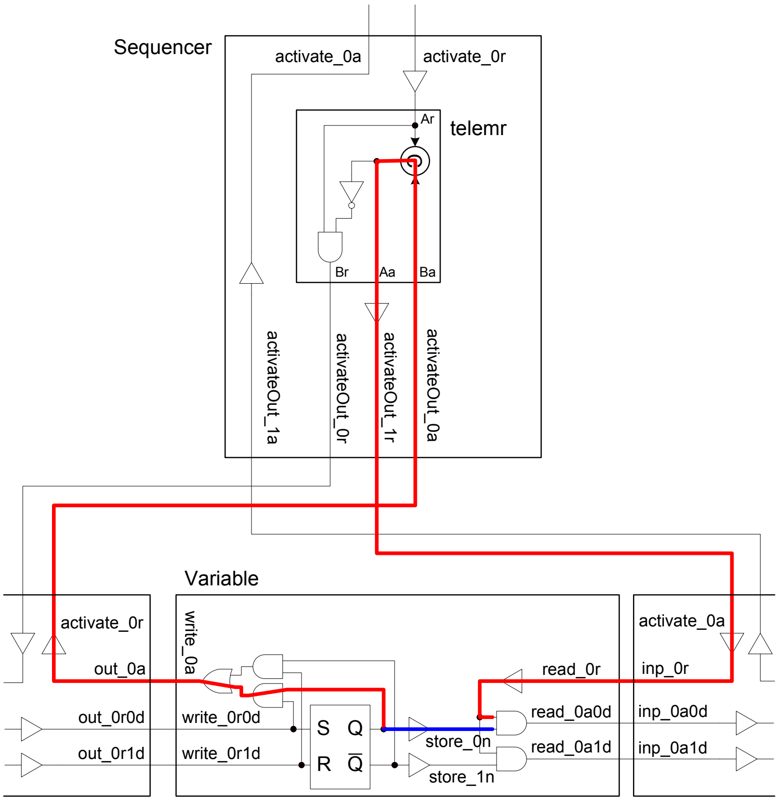


Figure 10.10: *The hazard on the zero line*

10.3 Discussion

In order to generate a DI timing model for this circuit, all components must be ensured to be QDI i.e. the isochronic fork timing assumption for all forks must be fulfilled, and all the combinatorial loops must be controlled by handshaking signals. Since, the *fetch 0*, *fetch 1* and *loop* components lack forks, they are QDI by design. If the other two components that comprise of forks i.e. *t-element* (telemr) and *variable*, are treated in the correct way as mentioned above, they will be QDI as well. Moreover, as explained above, all the combinatorial loops in this circuit are formed by handshaking signals. Since, the two requirements for delay insensitivity are met, it can be said that this circuit is DI.

It is noteworthy that a considerable time was spent analysing the timing model (i.e DI) generated by Balsa and how it works. Balsa does not provide any information on the two requirements (mentioned above) to ensure the delay insensitivity of the Balsa generated circuit. Thus, this had to be investigated with some guidance from [4].

Chapter 11
Integrating Balsa design into Nordic Semiconductor's tool chain

The main objective for this chapter is to try to integrate the Balsa output i.e. the Balsa Verilog netlist with Nordic Semiconductor's tool chain at the earliest possible stage of the design flow. The earliest stage would be to integrate the Balsa Verilog netlist with the normal synchronous Verilog design and run it through Synopsys Design Compiler (DC). This test will be carried out by combining the previously generated Balsa Verilog netlist in Chapter 10 with a normal synchronous Verilog RTL. This will then be tested through a self checking testbench verifying the correct functionality of the Balsa code. The same testbench will also be used on the combined netlist generated by DC. The criteria for successful integration would be that the testbench is still passing for the combined netlist, and a manual inspection of this netlist reveals that there have been no functional changes to the Balsa part of the netlist.

11.1 Integrating Balsa Verilog netlist with synchronous Verilog RTL

The setup for integrating the Balsa Verilog netlist with the synchronous Verilog RTL is shown in Figure 11.1 on the following page. It comprises of two main blocks: the Balsa block and the Verilog block. The Balsa block is the Verilog netlist generated in Chapter 10 for the buffer example. The Verilog block is a synchronous Verilog RTL called *ToyVerilog*, which interfaces to all the inputs and outputs of the Balsa block and controls and communicates with the Balsa asynchronous logic. These two blocks are instantiated inside the Device Under Test (DUT) hierarchical level called *DUT*. The *DUT* is going to be used as the top level of the design and is also written in Verilog RTL. It is instantiated inside the testbench called *test_toyVerilog*, which will provide the necessary stimuli and verify the correct behaviour of the circuit.

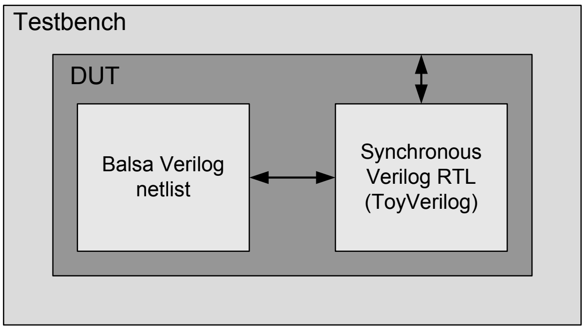


Figure 11.1: *Block diagram for the test setup*

11.1.1 Synchronous Verilog RTL block: ToyVerilog

A more detailed figure of the working design with the various signals and interface connections to and from the *ToyVerilog* module is shown in Figure 11.2 on the next page. The complete code for the module *ToyVerilog* has been provided in Appendix B.3 on page 168.

The state machine seen in Figure 11.2 is used to drive the signals *initialise* and *activate_0r*. The *initialise* signal acts as a reset for the entire Balsa block, and the *activate_0r* signal starts the execution of the Balsa block. A timing diagram showing the operation of these signals has been presented in Figure 11.3 on the next page.

The input channel *i* is connected to the *ToyVerilog* module. The signal *i_0r* is an input to *ToyVerilog*, going directly through this module and into the testbench. On the other hand, the signal *i_0a0d* and *i_0a1d* are outputs from flip-flops inside the *ToyVerilog* module. Both the clock (*ckDataIn*) and data (*a* and *b*) to these flip-flops are controlled directly from the testbench. The reason for passing signal *i_0a0d* and *i_0a1d* through flip-flips instead of controlling them directly from the testbench is to have some logic present inside the *ToyVerilog* module for the *i* and *o* data channels. This was needed to actually test the integration of Balsa with some Verilog code, which would not otherwise be present on the data channel. It is important to see if Balsa can take input from the synchronous logic, therefore, these signals are coming from flip-flops. Even though the *ToyVerilog* module has a main input clock (*ck*) controlling the state machine, a separate clock (*ckDataIn*) is used for these flip-flops. This is done so as to have exclusive control of the delay from receiving the request (*i_0r*) for data, until the data/acknowledgement (*i_0a0d* and *i_0a1d*) is sent back to the Balsa block. Controlling the delay is important to incorporate a random delay on this loop, in order to test that the design does not break if there are large delay variations in the handshake loop.

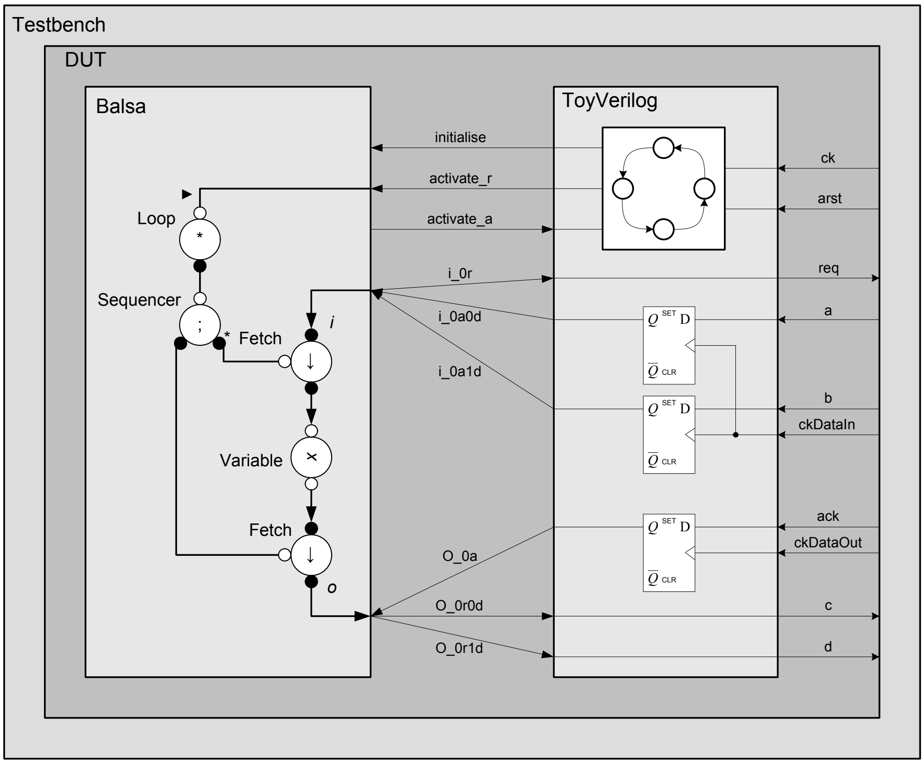


Figure 11.2: *Detailed block diagram for the toy design*

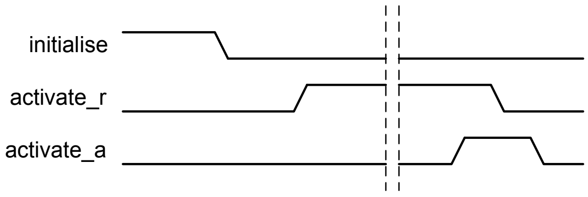


Figure 11.3: *Timing diagram for initialise and activate*

The output channel *o* is also connected to the *ToyVerilog* module. The signals *o_0r0d* and *o_0r1d* are inputs to *ToyVerilog*, going directly through this module and into the testbench. On the other hand the signal *o_0a* is an output from a flip-flop inside the *ToyVerilog* module. This flip-flop also has a separate clock (*ckDataOut*) used to control the timing. The reason for this configuration is the same as for the input channel (*i*).

11.1.2 Testbench description and verification cases

The testbench has two main functions. First, providing stimuli to the DUT and second, verifying the correct behaviour of the Balsa buffer design. The testbench is self checking and gives a pass or fail along with the error count at the end. The complete testbench code has been provided in Appendix B.5 on page 174.

Besides providing the clock and reset, the testbench is responsible for providing stimuli in the form of inputs to the input (*i*) and output (*o*) channel of the buffer. On detecting a posedge on the signal *i_0r* (*req*), the testbench drives the data channel i.e. 1 on either *a* or *b* randomly, but never on both simultaneously. After a random delay, the data is clocked via *ckDataIn* through the flip-flops present inside the *ToyVerilog* module, and into the dual rail channel pair *i_0a0d* and *i_0a1d* in the Balsa buffer. On detecting a negedge on the signal *i_0r* (*req*), the testbench drives both *a* and *b* to 0. After a random delay, the data is once again clocked via *ckDataIn* into the Balsa buffer.

The testbench detects a posedge on either *c* (*o_0r0d*) or *d* (*o_0r1d*), thus indicating receiving of data. The testbench will then set the *ack* high, and after a random delay the *ack* is clocked via *ckDataOut* through the flip-flop present inside the *ToyVerilog* module into the *o_0a* in the Balsa buffer. When the testbench detects a 0 on both *c* and *d*, the *ack* is set to 0, and after a random delay it is once again clocked via *ckDataOut* into the Balsa buffer.

The timing diagram exhibiting the above mentioned operation of the various signals is shown in Figure 11.4.

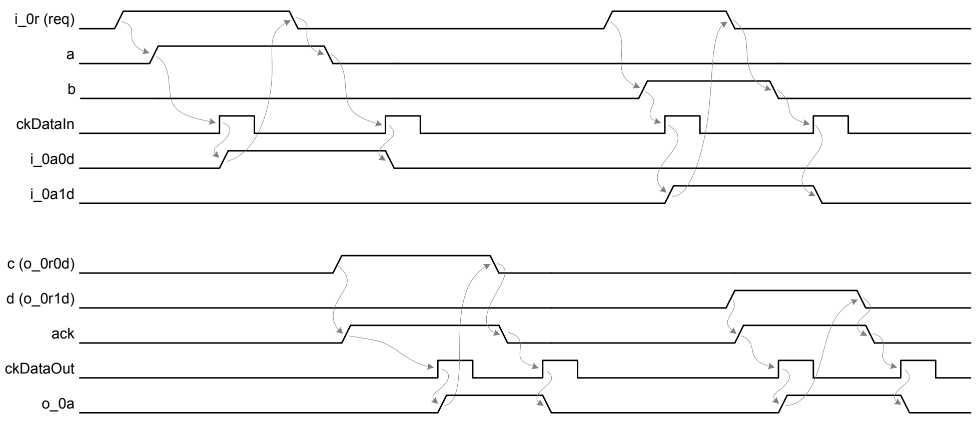


Figure 11.4: *Timing diagram for the toy design*

The testbench is also ensuring that the data received on the output channel *o* is the same as the data sent on the input channel *i*, and that there are no hazards i.e. glitches present on the output channel *o*. If any of the above stated tests fail, the testbench fails with an error message.

The various verification test cases run by the testbench are shown in Table 11.1.

Table 11.1: *Verification cases*

Verification item	Behaviour to be verified
1	Random delay from request of data (i_0r) until data is given (i_0a0d/i_0a1d)
2	Random delay from receiving data (o_0r0d/o_0r1d) until acknowledgement (o_0a) is given
3	Random data stimuli
4	Sending arbitrary number of data bits (upto 10000) through the data bit channel
5	Verifying that received data is the same as sent data
6	Glitch filter verifying that no glitch is present on output

11.1.3 RTL simulation

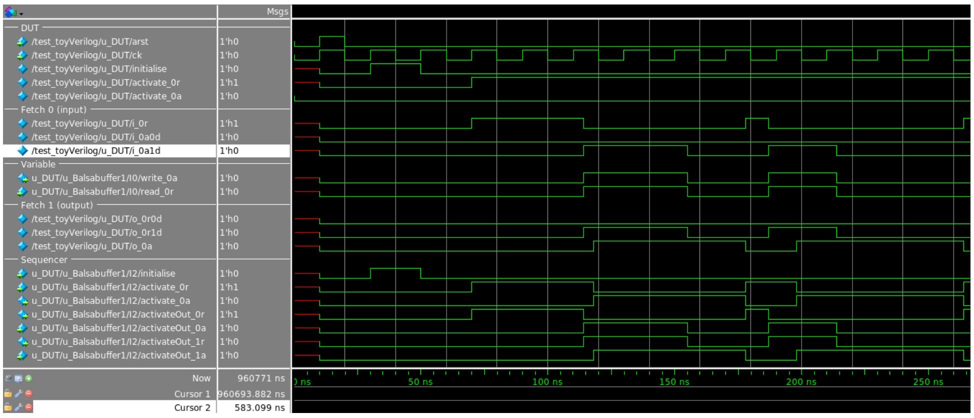
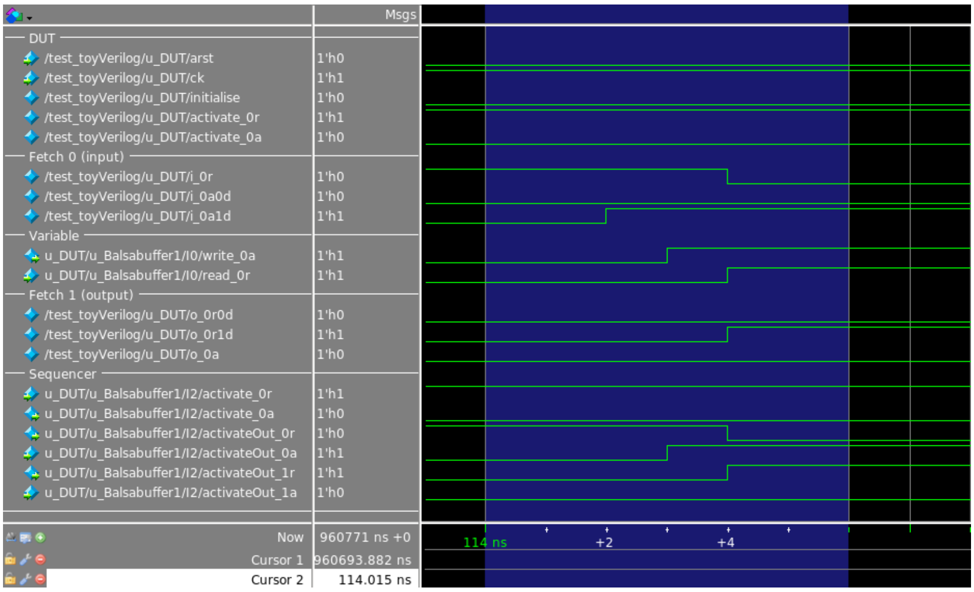

```
                                              Msgs
DUT
  /test_toyVerilog/u_DUT/arst                 1'h0
  /test_toyVerilog/u_DUT/ck                   1'h0
  /test_toyVerilog/u_DUT/initialise           1'h0
  /test_toyVerilog/u_DUT/activate_0r          1'h1
  /test_toyVerilog/u_DUT/activate_0a          1'h0
Fetch 0 (input)
  /test_toyVerilog/u_DUT/i_0r                 1'h1
  /test_toyVerilog/u_DUT/i_0a0d               1'h0
  /test_toyVerilog/u_DUT/i_0a1d               1'h0
Variable
  u_DUT/u_Balsabuffer1/I0/write_0a            1'h0
  u_DUT/u_Balsabuffer1/I0/read_0r             1'h0
Fetch 1 (output)
  /test_toyVerilog/u_DUT/o_0r0d               1'h0
  /test_toyVerilog/u_DUT/o_0r1d               1'h0
  /test_toyVerilog/u_DUT/o_0a                 1'h0
Sequencer
  u_DUT/u_Balsabuffer1/I2/initialise          1'h0
  u_DUT/u_Balsabuffer1/I2/activate_0r         1'h1
  u_DUT/u_Balsabuffer1/I2/activate_0a         1'h0
  u_DUT/u_Balsabuffer1/I2/activateOut_0r      1'h1
  u_DUT/u_Balsabuffer1/I2/activateOut_0a      1'h0
  u_DUT/u_Balsabuffer1/I2/activateOut_1r      1'h0
  u_DUT/u_Balsabuffer1/I2/activateOut_1a      1'h0
                                    Now       960771 ns
                               Cursor 1   960693.882 ns
                               Cursor 2      583.099 ns
```

Figure 11.5: *RTL simulatin of the Toy design*

The RTL simulation of the combined design (Balsa buffer with the *ToyVerilog* module), called *Toy* design from now on, was performed by using Mentor Graphics QuestaSim. A short time slot from the simulation has been shown in Figure 11.5. In



	Msgs
DUT	
/test_toyVerilog/u_DUT/arst	1'h0
/test_toyVerilog/u_DUT/ck	1'h1
/test_toyVerilog/u_DUT/initialise	1'h0
/test_toyVerilog/u_DUT/activate_0r	1'h1
/test_toyVerilog/u_DUT/activate_0a	1'h0
Fetch 0 (input)	
/test_toyVerilog/u_DUT/i_0r	1'h0
/test_toyVerilog/u_DUT/i_0a0d	1'h0
/test_toyVerilog/u_DUT/i_0a1d	1'h1
Variable	
u_DUT/u_Balsabuffer1/I0/write_0a	1'h1
u_DUT/u_Balsabuffer1/I0/read_0r	1'h1
Fetch 1 (output)	
/test_toyVerilog/u_DUT/o_0r0d	1'h0
/test_toyVerilog/u_DUT/o_0r1d	1'h1
/test_toyVerilog/u_DUT/o_0a	1'h0
Sequencer	
u_DUT/u_Balsabuffer1/I2/activate_0r	1'h1
u_DUT/u_Balsabuffer1/I2/activate_0a	1'h0
u_DUT/u_Balsabuffer1/I2/activateOut_0r	1'h0
u_DUT/u_Balsabuffer1/I2/activateOut_0a	1'h1
u_DUT/u_Balsabuffer1/I2/activateOut_1r	1'h1
u_DUT/u_Balsabuffer1/I2/activateOut_1a	1'h0
Now	960771 ns +0
Cursor 1	960693.882 ns
Cursor 2	114.015 ns

Figure 11.6: *Zoomed in at one of the events in the RTL simulation of the Toy design*

the RTL simulation there are no delays through the logic or the wires. As a result, it is impossible to see the propagation of the handshake signals since many signal transitions are taking place simultaneously. Figure 11.6 shows a zoomed in delta cycle mode for the same simulation on one of the edges. A delta cycle mode represents a minuscule level examination of the simulation wave. It shows the order in which the simulator evaluates the different signals. A manual inspection was done for some of the points in the wave using the delta cycle mode to verify the correct behaviour. Besides this, the testbench is self checking and monitors the correct behaviour of the design. Since, the manual inspection and the self checking testbench did not reveal any errors, it can be concluded that the combined design of the Balsa buffer with the *ToyVerilog* module is working as intended.

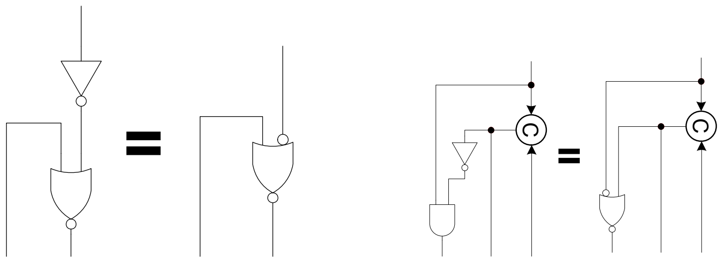
11.2 Synthesis of the *Toy* design

Synthesizing a design using Design Compiler (DC) is the first step in Nordic Semiconductor's tool chain towards fabrication. This is the first place where the Balsa Verilog netlist could be combined with the synchronous design and integrated into Nordic Semiconductors synchronous tool chain. This is what is trying to be achieved in this section.

Initially, the scripts for running DC were configured, and an elaboration and compilation of the design was done. For a more detailed explanation of these initial steps, [11] can be referred. The elaboration of the design gave no errors but two warnings. However, these warning were not critical and were due to some setup issues in the scripts. Hence, they could be ignored. The compilation of the design also gave no errors but three warnings. These warnings were due to a mismatch between the operating conditions of the libraries. However, these warnings could be ignored since the design was not intended for fabrication and the existing libraries were good enough for the purpose of this test. In case a design needs to be taped out, this must be dealt with. After a successful compilation, an optimized netlist for the *Toy* design was obtained. The logs for the elaboration and compilation can be found in Appendix B.6 on page 180 and the netlist can be found in Appendix B.7 on page 180.

The next step was to perform a simulation on the generated netlist. The same testbench as before was used for this purpose. Initially, the netlist simulation failed. This was mainly because of an error while setting up the simulation. A wrong library for the SR-latch that required the power pins (VDD and VSS) to be connected was selected. At this stage the power nets should not be included. Therefore, a different library for the SR-latch that did not require the power nets to be connected was used instead. Once this was fixed, the netlist simulation passed successfully without any errors. It is noteworthy that this netlist lacked timing information. This implies that it does not include any logic or wire delays yet. Hence, the next step was to generate a SDF file giving the timing information for the various propagation delays through the circuit. Synopsis Prime Time (PT) was used to generate the SDF. A new netlist simulation with back annotated timing was then performed. This simulation failed with several errors. Some errors were in the form of glitches detected by the glitch filter in the testbench while the others accrued to receiving incorrect data on the output. After some thorough inspection, an error was found in the testbench. The testbench did not always handle the delay in the design correctly and fell out of synchronization with the design. The testbench was therefore updated to handle any delay given by the design. Then, the netlist was resimulated and it passed without any errors.

The next step was to ensure that the DC did not optimize the asynchronous Balsa buffer part, so that it was still functionally working as intended. This is because while an optimization done by DC for a synchronous design is correct, it could break the working of an asynchronous design. Therefore, the optimized netlist was manually compared component by component to the original Balsa netlist in order to verify the same behaviour. This inspection revealed that most of the Balsa design was left untouched, however there were some minor differences. Firstly, there was a small difference in the *loop* component as seen in Figure 11.7a and in the *t-element* (telemr) as seen in Figure 11.7b. The deviations in both the components from their original ones do not change their functionality at all, thus making them logically equivalent. Secondly, it was observed that some logical elements were replaced by more optimal elements, for example NAND gates were used instead of AND gates and buffers were replaced by inverters, but DC had ensured that the overall functionality remained the same. This is normal and expected behaviour of DC.



(a) ***loop element*** **(b)** ***telemr element***

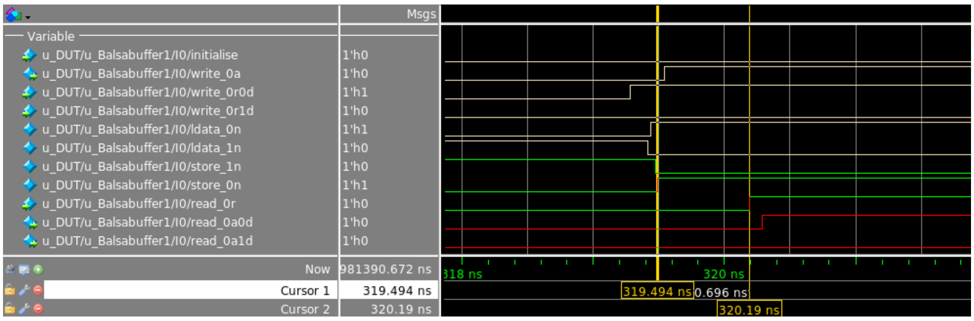
Figure 11.7: *Changes in logic after synthesis*

11.2.1 Triggering the error on the isochronic fork

As discussed in Section 10.2.2 on page 110 there exist two forks in the *variable* component which are not safe by design. This is because the isochronic fork assumption is not fulfilled due to which the *variable* component is not QDI. As mentioned earlier, this is a possible hazard in the design, as seen earlier in Figure 10.10 on page 112. Therefore, at first it was manually verified that this hazard was not present in the *Toy* design. This was accomplished by checking that the read signal (*read_0r*) comes after the dual rail data signal (*store_0n* and *store_1n*). Refer Figure 10.10 on page 112 for the signal names. As seen in Figure 11.8 on the next page, the signal *read_0r* is coming 0.696 ns after the dual rail data signal. This is thereby making this fork safe, as also seen in the same figure where there are no glitches present on the signals *read_0a0d* and *read_0a1d* (denoted by the red lines).

In order to see the consequences of the signal *read_0r* coming before the dual rail data signal, the timing for the signals involved had to be manipulated. The easiest solution would be to increase the delay through the buffers between the *SR-latch* and the two output AND gates. This would simply involve changing the timing for these buffers in the SDF file. However, these buffers were removed by the synthesis of the design. Therefore, in order to keep these buffers, an attribute *dont_touch* was applied to these buffers as given below:

```
set_attribute u_Balsabuffer1/I0/I3  dont_touch true
set_attribute u_Balsabuffer1/I0/I4  dont_touch true
```



```
Variable                                   Msgs
u_DUT/u_Balsabuffer1/I0/initialise         1'h0
u_DUT/u_Balsabuffer1/I0/write_0a           1'h0
u_DUT/u_Balsabuffer1/I0/write_0r0d         1'h1
u_DUT/u_Balsabuffer1/I0/write_0r1d         1'h0
u_DUT/u_Balsabuffer1/I0/ldata_0n           1'h1
u_DUT/u_Balsabuffer1/I0/ldata_1n           1'h0
u_DUT/u_Balsabuffer1/I0/store_1n           1'h0
u_DUT/u_Balsabuffer1/I0/store_0n           1'h1
u_DUT/u_Balsabuffer1/I0/read_0r            1'h0
u_DUT/u_Balsabuffer1/I0/read_0a0d          1'h0
u_DUT/u_Balsabuffer1/I0/read_0a1d          1'h0
Now       981390.672 ns
Cursor 1  319.494 ns
Cursor 2  320.19 ns
```

Figure 11.8: *The hazard when a glitch is not triggered*

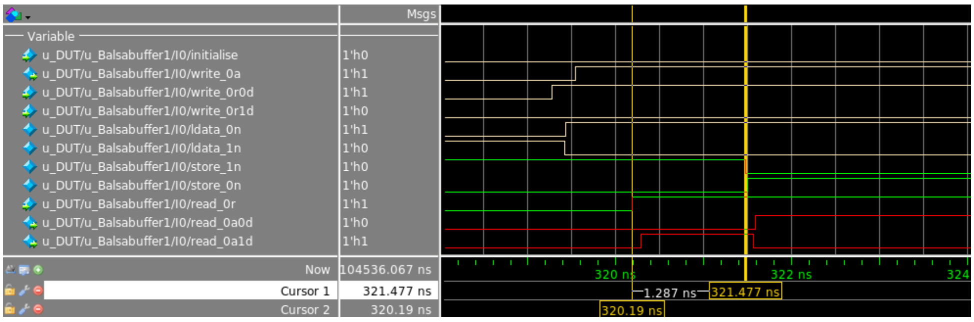
Then, a new compilation was done and an inspection of the new netlist generated revealed that the buffers were not removed. A new SDF was generated and a new simulation was run. However, adding the buffers still did not give a big enough delay to trigger the hazard. Hence, the delay through the buffer had to be increased in the SDF file. Given below are the lines from the SDF file that were changed from

```
(INSTANCE u_Balsabuffer1/I0/I3)
(DELAY (ABSOLUTE (IOPATH I Z (0.057::0.059) (0.060::0.062))))

(INSTANCE u_Balsabuffer1/I0/I4)
(DELAY (ABSOLUTE (IOPATH I Z (0.057::0.059) (0.060::0.062))))
```

to

```
(INSTANCE u_Balsabuffer1/I0/I3)
(DELAY (ABSOLUTE (IOPATH I Z (2.057::2.059) (2.060::2.062))))

(INSTANCE u_Balsabuffer1/I0/I4)
(DELAY (ABSOLUTE (IOPATH I Z (2.057::2.059) (2.060::2.062))))
```

Doing so increases the delay through the buffer by 2 ns. Then, a new simulation was run and the hazard was triggered at several places giving many errors in the testbench. Figure 11.9 on the following page is showing the simulation at the same time as before, but this time with the hazard being present. From this figure, it can be clearly seen that the signal *read_0r* is coming much before the dual rail data signals. Hence, for this period of time, the old value on the dual rail data signals is passed through the AND gates, thus producing a glitch on the output, as seen from the red signals in the figure. This proves that there exists the possibility of hazards in this design, if the delay in these forks is not monitored to be isochronic.



Variable	Msgs
u_DUT/u_Balsabuffer1/I0/initialise	1'h0
u_DUT/u_Balsabuffer1/I0/write_0a	1'h1
u_DUT/u_Balsabuffer1/I0/write_0r0d	1'h1
u_DUT/u_Balsabuffer1/I0/write_0r1d	1'h0
u_DUT/u_Balsabuffer1/I0/ldata_0n	1'h1
u_DUT/u_Balsabuffer1/I0/ldata_1n	1'h0
u_DUT/u_Balsabuffer1/I0/store_1n	1'h0
u_DUT/u_Balsabuffer1/I0/store_0n	1'h0
u_DUT/u_Balsabuffer1/I0/read_0r	1'h1
u_DUT/u_Balsabuffer1/I0/read_0a0d	1'h0
u_DUT/u_Balsabuffer1/I0/read_0a1d	1'h1
Now	104536.067 ns
Cursor 1	321.477 ns
Cursor 2	320.19 ns

Figure 11.9: *The hazard when a glitch is triggered*

11.3 Discussion

This chapter tested the integration of the Balsa Verilog netlist with the normal synchronous Verilog design. All the criteria set for successful integration were met: the self checking testbench passed for the combined design (Balsa buffer with the *ToyVerilog* module) both before and after synthesis and the manual inspection for the combined netlist did not show any changes in the functional behaviour for the Balsa part. Hence, it can be concluded that the integration of an asynchronous design written in the Balsa language is possible with Nordic Semiconductor's tool chain for synchronous design.

Chapter 12 Conclusion

In this thesis, several asynchronous design tools were investigated and the most suitable tool for integrating an asynchronous design with Nordic Semiconductor's tool chain was selected. Balsa, the selected tool, was then employed to design an asynchronous flash readout. Furthermore, the netlist from a simple buffer designed in Balsa was investigated to analyze the delay insensitivity nature of Balsa, and was then combined with a synchronous RTL design to be synthesized using Design Compiler.

The task of selecting an asynchronous tool was commenced by studying fifteen asynchronous design tools. Five tools were shortlisted for further investigation based on the following criteria: longevity, tool complexity, cost, estimate of performance of corresponding circuits, commercial or non-commercial implementations, delay model used by the final netlist, support and integration with Nordic Semiconductor's design flow. The five shortlisted tools were Balsa, Petrify, Verisyn, Pipefitter and ACC. The integration aspects, possibilities and issues of these shortlisted tools with Nordic Semiconductor's tool chain were then examined. This was done by installing and running a simple design through each of them. Balsa was the final selected tool. This was primarily because Balsa can generate a Delay Insensitive (DI) Verilog netlist. A DI delay model was preferred because no timing checks are required to assure the correct behaviour of the design.

An asynchronous flash readout was designed using Balsa. The asynchronous flash readout takes control of the flash and performs readouts from the flash. The entire design consists of two main blocks. The first block controls the flash readout and was implemented in Balsa. The second block is a flash wrapper containing the flash, logic interfacing the Balsa control signals to the flash and the storage elements storing the data read from the flash. The second block was implemented in Verilog. The first block containing the Balsa code was synthesized into a Verilog netlist. Balsa can generate a Verilog netlist using several data encoding styles. A Verilog netlist for both single rail and dual rail data encoding styles was generated. Verification of the Balsa code functionality was done by performing a simulation using the single rail netlist. This simulation revealed that the Balsa code was functionally working as intended. However, the single rail data encoding style does not produce a delay insensitive netlist, whereas a dual rail data encoding style does. Therefore, a dual rail netlist was generated as the final result for this implementation. A simulation performed using the dual rail netlist verified that the design was still working correctly.

An analysis of the delay insensitivity nature of Balsa was performed using a simple buffer design. In order to perform this analysis, a new technology matching the library used at Nordic Semiconductor was developed. The netlist for the buffer design was broken down into breeze components and the individual breeze components were then analysed. For a Balsa generated netlist to be DI all the combinatorial loops must be controlled by handshaking signals and all components must be Quassi Delay Insensitive (QDI). For a component to be QDI, all forks must fulfil the isochronic fork timing assumption. For the buffer netlist, all the combinatorial loops were by design controlled by handshaking signals, but not all component implementations used in this thesis met the isochronic fork timing assumption by design. Hence, not all components were QDI by design. There exists a possibility of a hazard in the *variable* component. However, the probability for this hazard to be triggered is extremely less. The timing for the paths present in the hazardous fork were manually checked and were found to fulfil the isochronic fork timing assumption. Therefore, it can be said that for the buffer design the *variable* component as well as all the other components were ensured to be QDI. Since, both the requirements for delay insensitivity were met, it can be concluded that this design is DI.

An attempt to integrate the Balsa Verilog netlist with Nordic Semiconductor's tool chain was made. The Balsa Verilog netlist generated from the buffer design was combined with a normal synchronous Verilog design and run through Synopsys Design Compiler (DC). A self checking testbench was used to verify the correct functionality of the combined design both before and after it was synthesized by DC. Also, a manual inspection of the combined netlist produced by DC was performed. No changes in the functional behaviour for the Balsa part were found. Hence, it can be concluded that the integration of an asynchronous design written in the Balsa language is possible with Nordic Semiconductor's tool chain for synchronous design.

Chapter 13 Future work

This chapter provides the work that can be pursued in the future as a continuation to this thesis.

- A good starting point for future work involves making all the breeze components QDI by design. For example, the *variable* component in the buffer design is not QDI by design. Even though the probability for the hazard to occur is very low, it is not 100 % QDI. Thus, the *variable* component can be made QDI by design.
- This thesis has proven that an asynchronous design written in Balsa can be integrated into Nordic Semiconductor's synchronous tool chain by using a simple buffer design. However, this has to be tested yet for the main design i.e. the asynchronous flash readout. Due to the limited time frame, this could not be done. Therefore, an important work in the future involves integrating the asynchronous flash readout into Nordic Semiconductor's synchronous tool chain.
- Conduct a power analysis and measure the power consumption during the readout for the asynchronous flash readout. Then conduct the same analysis for the existing synchronous flash readout at Nordic Semiconductor, and compare the results derived.
- Some other further potential goals could be to run the asynchronous flash readout design through the complete synchronous tool chain involving layout, placement and route.
- An investigation of asynchronous design for writing to the flash could be made.
- Lastly, the backend tool Balsa CUBE, which is an extension to Balsa, could be used in order to achieve performance improvements in the overall design.

References

[1] S. Hauck, "Asynchronous design methodologies: an overview," *Proceedings of the IEEE*, vol. 83, no. 1, pp. 69–93, Jan 1995.

[2] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A designer's guide to asynchronous VLSI.* Cambridge University Press, New York, 2010.

[3] J. SparsÃ¸ and S. Furber, *An Introduction to Balsa*, J. SparsÃ¸ and S. Furber, Eds. Springer US, 2001. [Online]. Available: http://web.cecs.pdx.edu/~mperkows/CLASS_573/febr-2007/Kluwer=Perspective.pdf

[4] W. B. Toms, "Synthesis of quasi-delay-insensitive datapath circuits," Ph.D. dissertation, Department of Computer Science, University of Manchester, February 2006.

[5] J. T. Udding, "A formal model for defining and classifying delay-insensitive circuits and systems," *Distributed Computing*, vol. 1, pp. 197–204, 1986. [Online]. Available: http://www.eecs.ucf.edu/~mingjie/ECM6308/papers/A%20formal%20model%20for%20defining%20and%20classifying%20delay-insensitive%20circuits%20and%20systems.pdf

[6] M. G. F. H. N. C. Matheus Moreira, Felipe Magalhães, "Power-efficient clockless intrachip communication design with an integrated high to low level flow based on the balsa framework," FACULDADE DE INFORMÁTICA, Pontifícia Universidade Católica do Rio Grande do Sul (GAPH-FACIN-PUCRS), Technical Report series 075, September 2013. [Online]. Available: http://www3.pucrs.br/pucrs/files/uni/poa/facin/pos/relatoriostec/TR075.pdf

[7] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," 1996. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.3124&rep=rep1&type=pdf

[8] Asynchronous high level synthesis tool (verisyn). Async.org.uk. [Online]. Available: http://async.org.uk/besst/verisyn/

[9] I. Blunno and L. Lavagno, "Automated synthesis of micro-pipelines from behavioral verilog hdl," in *Advanced Research in Asynchronous Circuits and Systems,*

2000. (ASYNC 2000) Proceedings. Sixth International Symposium on, 2000, pp. 84–92.

[10] M. Theobald, “Efficient algorithms for the design of asynchronous control circuits,” Ph.D. dissertation, Columbia University, 2002.

[11] S. Khosla, “Asynchronous design through synchronous tool flow,” Department of Telematics, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, Tech. Rep., July 2015.

[12] Y. Li, “Redressing timing issues for speed-independent circuits in deep sub-micron age,” Department of Telematics, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, Technical Report Series NCL-EEE-MSD-TR-2012-180, July 2012. [Online]. Available: http://async.org.uk/tech-reports/NCL-EEE-MSD-TR-2012-180.pdf

[13] A. J. Martin, “The limitations to delay-insensitivity in asynchronous circuits,” Computer Science Department California Institute of Technology, Tech. Rep. Caltech-CS-TR-90-02, November 1989. [Online]. Available: http://authors.library.caltech.edu/26721/2/postscript.pdf

[14] ——, “Compiling communicating processes into delay-insensitive vlsi circuits,” Computer Science Department, California Institute of Science and Technology, Tech. Rep. 5210:TR:86, 2006.

[15] C. J. Myers, *Asynchronous Circuit Design*, C. J. Myers, Ed. Wiley, July 2001.

[16] Wikipedia. Communicating sequential processes. [Online]. Available: https://en.wikipedia.org/wiki/Communicating_sequential_processes

[17] R. David, “Modular design of asynchronous circuits defined by graphs,” *Computers, IEEE Transactions on*, vol. C-26, no. 8, pp. 727–737, Aug 1977.

[18] V. K. Josep Carmona, Jordi Cortadella and A. Yakovlev, “Synthesis of asynchronous hardware from petri nets.” [Online]. Available: http://www.cs.upc.edu/~jordicf/gavina/BIB/files/lcpn04_synth.pdf

[19] D. Shang, F. Xia, and A. Yakovlev, “Asynchronous fpga architecture with distributed control,” in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, May 2010, pp. 1436–1439.

[20] W. B. T. D. A. Edwards, “Design, automation and test for asynchronous circuits and systems,” Information Society Technologies (IST) Programme Concerted Action Thematic Network Contract, 3rd Edition IST-1999-29119, June 2004. [Online]. Available: http://www.bcim.lsbu.ac.uk/ccsv/ACiD-WG/AsyncToolSurvey.pdf

[21] A. Martin, “Synthesis of asynchronous vlsi circuits,” Computer Science Department, California Institute of Technology, August 9 1991. [Online]. Available: http://authors.library.caltech.edu/26746/2/postscript.pdf

[22] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, "An asynchronous low-power 80c51 microcontroller," in *Advanced Research in Asynchronous Circuits and Systems, 1998. Proceedings. 1998 Fourth International Symposium on*, Mar 1998, pp. 96–107.

[23] U. o. M. School of Computer Science Intranet. The balsa asynchronous synthesis system. University of Manchester. [Online]. Available: http://apt.cs.manchester.ac.uk/projects/tools/balsa/

[24] L. J. L. P. . W. T. Doug Edwards, Andrew Bardsley, *Balsa: A Tutorial Guide.*, version v3.5 ed., university of Manchester, May.

[25] S. Furber, D. Edwards, and J. Garside, "Amulet3: a 100 mips asynchronous embedded processor," in *Computer Design, 2000. Proceedings. 2000 International Conference on*, 2000, pp. 329–334.

[26] U. o. M. School of Computer Science Intranet. The teak asynchronous synthesis system. University of Manchester. [Online]. Available: http://apt.cs.manchester.ac.uk/projects/teak/

[27] A. B. D. E. Tiberiu Chelcea, Steven M. Nowick, "Balsa-cube: an optimizing back-end for the balsa synthesis system." [Online]. Available: http://async.org.uk/ukasyncforum14/forum14-papers/forum14-chelcea.pdf

[28] P. Endecott and S. Furber, "Modelling and simulation of asynchronous systems using the lard hardware description language," in *Proceedings of the 12th European Simulation Multiconference on Simulation - Past, Present and Future.* SCS Europe, 1998, pp. 39–43. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.3469&rep=rep1&type=pdf

[29] G. Theodoropoulos, G. Tsakogiannis, and J. Woods, "Occam: an asynchronous hardware description language?" in *EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference*, Sept 1997, pp. 249–256.

[30] (1999) Petrify: a tool for synthesis of petri nets and asynchronous circuits. Universitat Politècnica de Catalunya, Barcelona, Spain. [Online]. Available: http://www.cs.upc.edu/~jordicf/petrify/

[31] T.-A. Chu, "Synthesis of self-timed vlsi circuits from graph-theoretic specifications," Ph.D. dissertation, Massachusetts Institute of Technology (MIT), Dept. of Electrical Engineering and Computer Science., June 1987.

[32] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Synthesizing petri nets from state-based models," in *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, Nov 1995, pp. 164–171.

[33] M. Pena and J. Cortadella, “Combining process algebras and petri nets for the specification and synthesis of asynchronous circuits,” in *Advanced Research in Asynchronous Circuits and Systems, 1996. Proceedings., Second International Symposium on*, Mar 1996, pp. 222–232.

[34] C. Ykman-Couvreur, “Assassin: A synthesis system for asynchronous control circuits,” IMEC, User and Tutorial Manual, September 1994.

[35] S. M. N. Al Davis, “An introduction to asynchronous circuit design,” *Encyclopedia of Computer Science and Technology*, vol. Volume 38, Sup 23, pp. 231–286, 1998. [Online]. Available: https://books.google.no/books?id=PySs1uQ2l3gC&pg=PA231&lpg=PA231&dq=an+introduction+to+asynchronous+circuit+design+allen+kent&source=bl&ots=q6VE_nl6Zx&sig=EOWLBPnWOt9ZFVNhl0lcTbHCJF0&hl=en&sa=X&ved=0CB0Q6AEwAGoVChMI-pWQ34fgxgIVhN0sCh0MdgN0#v=onepage&q=an%20introduction%20to%20asynchronous%20circuit%20design%20allen%20kent&f=false

[36] Workcraft. [Online]. Available: http://www.workcraft.org/

[37] I. Blunno and L. Lavagno, “Designing an asynchronous microcontroller using pipefitter,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 7, pp. 696–699, July 2004.

[38] I. Blunno and V. P. Shah, *PipefitterTutorial*, Microelectronics Group, Politecnico di Torino, November 5 2003.

[39] A. A. F.Burns, D.Shang, “Verisyn: Tool support for verilog asynchronous synthesis - a tutorial guide,” School of Electrical, Electronic & Computer Engineering, University of Newcastle, Tech. Rep. NCL-EECE-MSD-TR-2004-104, September 2004.

[40] J. Cortadella and R. Badia, “An asynchronous architecture model for behavioral synthesis,” in *Design Automation, 1992. Proceedings., [3rd] European Conference on*, Mar 1992, pp. 307–311.

[41] E. Eim, J.-G. Lee, and D.-I. Lee, “Automatic process-oriented control circuit generation for asynchronous high-level synthesis,” in *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, 2000, pp. 104–113.

[42] T. Nguyen, K.-N. Le-Huu, T. Bui, and A.-V. Dinh-Duc, “A new approach and tool in verifying asynchronous circuits,” in *Advanced Technologies for Communications (ATC), 2012 International Conference on*, Oct 2012, pp. 152–157.

[43] S. P. Wilcox, “Synthesis of asynchronous circuits,” Ph.D. dissertation, University of Cambridge, Queens’ College, July 1999. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-468.pdf

[44] Eclipse. Version 3.8.1. [Online]. Available: https://eclipse.org/

[45] QuestaSim. Version 10.2b. Mentor Graphics. [Online]. Available: http://www.mentor.com/products/fv/questa/

[46] Design Compiler. Version j-2014.09. Synopsys. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx

[47] PrimeTime. Version j-2014.06-sp2. Synopsys. [Online]. Available: http://www.synopsys.com/Tools/Implementation/SignOff/Pages/PrimeTime.aspx

[48] Asynchronous circuit design and testing. [Online]. Available: http://polimage.polito.it/~lavagno/async.html

[49] M. T. Moreira, "Design and implementation of a standard cell library for building asynchronous asics," Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul, FACULDADE DE ENGENHARIA, FACULDADE DE INFORMÁTICA ENGENHARIA DE COMPUTAÇÃO, 2010. [Online]. Available: https://www.inf.pucrs.br/~calazans/publications/2010_TCC_MatheusTrevisanMoreira.pdf

[50] A. Bardsley, "Balsa: An asynchronous circuit synthesis system," Master's thesis, Department of Computer Science, University of Manchester, 1998. [Online]. Available: http://apt.cs.manchester.ac.uk/ftp/pub/apt/theses/bardsley_mphil.pdf

[51] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002.

[52] T. Chelcea, A. Bardsley, D. Edwards, and S. Nowick, "A burst-mode oriented back-end for the balsa synthesis system," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002, pp. 330–337.

[53] T. Yoneda, A. Matsumoto, M. Kato, and C. Myers, "High level synthesis of timed asynchronous circuits," in *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, March 2005, pp. 178–189.

[54] D. A. E. A. Bardsley, "The balsa asynchronous circuit synthesis system," *Department of Computer Science,The University of Manchester*. [Online]. Available: http://apt.cs.manchester.ac.uk/ftp/pub/apt/papers/FDL00.pdf

[55] N. C. Matheus Moreira, "Comparing two asynchronous ic design flows," *XXVII SIM - South Symposium on Microelectronics*, 2012. [Online]. Available: http://www.inf.ufrgs.br/sim-emicro/papers2012/sim2012_submission_3.pdf

[56] A. Bardsley, L. Tarazona, and D. Edwards, "Teak: A token-flow implementation for the balsa language," in *Application of Concurrency to System Design, 2009. ACSD '09. Ninth International Conference on*, July 2009, pp. 23–31.

[57] A. J. M. . C. D. Moore, "Chp and chpsim: A language and simulator for fine-grain distributed computation," *Department of Computer Science, California Institute of Technology*, 2012. [Online]. Available: http://www.async.caltech.edu/Pubs/PDF/chpasync2012.pdf

[58] D. D. C. Hyde, "Introduction to the programming language occam," Department of Computer Science, Bucknell University, Lewisburg, PA 17837, 1995. [Online]. Available: http://www.cs.otago.ac.nz/cosc441/occam.pdf

[59] *Behavioural Modelling of Asynchronous Systems for Power and Performance Analysis*, Department of Computer Science, University of Manchester. Department of Computer Science, University of Manchester, October 1998. [Online]. Available: http://www.academia.edu/2610133/Behavioural_modelling_of_asynchronous_systems_for_power_and_performance_analysis

[60] Async.org.uk. Tools. [Online]. Available: http://async.org.uk/tools.html

[61] (2009) Pnml grammar, version 2009. pnml.org. [Online]. Available: http://www.pnml.org/version-2009/version-2009.php

[62] J. Cortadella, *Petrify: a tutorial for the designer of asynchronous ccircuit*, Department of Software, Universitat Politecnica de Catalunya.

[63] A. K. J. Cortadella, M. Kishinevsky and L. Lavagno, "Introduction to asynchronous circuit design: specification and synthesis (tutorial)." 6th. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems, Eilat (Israel), April 2000.

[64] L. L. Alex Yakovlev, Luis Gomes, *Hardware Design and Petri Nets*, L. L. Alex Yakovlev, Luis Gomes, Ed. Springer Science & Business Media, B.V., 2000.

[65] A. K. Frank Burns, Delong Shang and A. Yakovlev, Eds., *An Asynchronous Synthesis Toolset using Verilog.* Newcastle Upon Tyne, NE7 1RU, UK, f.p.burns@ncl.ac.uk: School of Electrical, Electronic & Computer Engineering, University of Newcastle Upon Tyne, 2004. [Online]. Available: http://www.date-conference.com/proceedings1/PAPERS/2004/DATE04/PDFFILES/IP3_03.PDF

[66] U. University of California at Berkeley. Demo session, async 2007, 13th ieee international symposium on asynchronous circuits and systems. [Online]. Available: http://conferences.computer.org/async2007/DMO/dmo.htm

[67] P. Newswire. Silistix announces chainworks(tm) for self-timed interconnect design and synthesis for socs. [Online]. Available: http://www.prnewswire.com/news-releases/silistix-announces-chainworkstm-for-self-timed-interconnect-design-and-synthesis-for-socs-53543012.html

[68] M. Wired. (2006, July 20) Silistix and coware developing esl-based design flow for chips using asynchronous self-timed interconnect. Silistix. [Online]. Available: http://www.marketwired.com/press-release/silistix-coware-developing-esl-based-design-flow-chips-using-asynchronous-self-timed-696745.htm

[69] M. Arora. Ultra low power designs using asynchronous design techniques (welcome to the world without clocks). Design & Reuse. [Online]. Available: http://www.design-reuse.com/articles/17479/asynchronous-design-techniques-ultra-low-power.html

[70] R. Goering. (2006, January 16) Design tools: Tool suite rolls for asynchronous logic. [Online]. Available: http://www.embedded.com/electronics-products/electronic-product-reviews/embedded-tools/4081605/DESIGN-TOOLS-Tool-suite-rolls-for-asynchronous-logic

[71] U. o. M. School of Computer Science. Silistix limited. [Online]. Available: http://www.cs.manchester.ac.uk/industry/spin-offs/silistixlimited/

[72] Tiempo. Acc: Asynchronous circuit compiler. [Online]. Available: http://www.tiempo-ic.com/products/sw-tools/acc.html

[73] M. S. B.ASc, "Desynchronization methods for scheduled circuits," M.Sc. Thesis, Circuits and Systems Group, Department of Microelectronics & Computer Engineering, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2009.

[74] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 1904–1921, Oct 2006.

[75] D. S. Jurgen Galler, Denise Ratasich, "Vu advanced digital design , design 3: Asynchronous bundled data pipeline (32-bit adder, fully-decoupled latch controllers)," January 27 2013. [Online]. Available: https://ti.tuwien.ac.at/ecs/teaching/courses/adide_WS_2012/add-design-solutions/gr5.pdf

[76] K. S. Stevens, "Method and system for asynchronous chip design," US Application US20 090 106 719 A1, 2009. [Online]. Available: https://www.google.com/patents/US20090106719

[77] Welcome to gtkwave. [Online]. Available: http://gtkwave.sourceforge.net

[78] Synthesis and verification of c-element. [Online]. Available: http://www.workcraft.org/tutorial/synthesis/celement/start

[79] Icarus verilog. [Online]. Available: http://iverilog.icarus.com/

[80] Tiempo. Tiempo. [Online]. Available: http://www.tiempo-ic.com/home.html

Appendix A AsynchronousFlashReadout

A.1 AsynchronousFlashReadout

```
-- Design for Asynchronous Flash Readout
import [balsa.types.basic]
import [Flashcontrol]
import [Counter]
import [InitialSetup]

procedure AsynchronousFlashReadout (
  --parameter ADDR_BITS   : 8 bits;
  --parameter DATA_BITS   : 8 bits;
  sync   start;
  output controlFlash: 1 bits;
  input SE : 8 bits;
  output adr : 8 bits;
  output dataRead_0  : 8 bits;
  output dataRead_1  : 8 bits;
  output dataRead_2  : 8 bits;
  output dataRead_3  : 8 bits
) is

-- Inital setup of constants --
constant NUM_COUNT = 4
type COUNTER_WIDTH is 4 bits   -- Must be big enough for the NUM_COUNT
    value

type ADDR_BITS is 8 bits
type DATA_BITS is 8 bits


-- Local variables --
sync startCounter
channel count   : COUNTER_WIDTH
--sync SE
variable dataRead : array 0..3 of DATA_BITS
--array 4 of channel dataRead2 : DATA_BITS
```

```
-- Connecting everything --

begin
  InitialSetup (start, startCounter, controlFlash)
  ||
  --Counter (NUM_COUNT, COUNTER_WIDTH, startCounter, count)
  Counter (startCounter, count)
  ||
  --FlashControl (ADDR_BITS, DATA_BITS, COUNTER_WIDTH, count, adr, SE,
      dataRead_0, dataRead_1, dataRead_2, dataRead_3)
  FlashControl (count, adr, SE, dataRead_0, dataRead_1, dataRead_2,
      dataRead_3)
end
```

A.2 InitialSetup

```
-- Design for Initial Setup
import [balsa.types.basic]

procedure InitialSetup (
  sync start;
  sync startCounter;
  output controlFlash: 1 bits
) is

begin
  loop
    select start then
      controlFlash <- 1;
      sync startCounter;
      controlFlash <- 0
    end
  end
end
```

A.3 Counter

```
-- Design for AsynchronousCounter
import [balsa.types.basic]

procedure Counter (
  --parameter NUM_COUNT : cardinal;
  --parameter COUNTER_WIDTH : type;
  sync start;
  output count: 4bits --COUNTER_WIDTH
) is

--Can't send in parameters
constant NUM_COUNT = 4
type COUNTER_WIDTH is 4 bits

variable tmp, count_rg : COUNTER_WIDTH

begin
  select start then
    loop while count_rg < NUM_COUNT then
      tmp := (count_rg + 1 as COUNTER_WIDTH);
      count_rg :=tmp;
      count <- count_rg
      end
   end
end
```

A.4 Flash Control

```
-- Design for Flash Control
import [balsa.types.basic]
import [BufferControl]

procedure FlashControl (
  --parameter ADDR_BITS     : type;
  --parameter DATA_BITS     : type;
  --parameter COUNTER_WIDTH : type;
  input count             : 4 bits; --COUNTER_WIDTH ;
  output adr              : 8 bits; --ADDR_BITS ;
  input SE                : 8 bits; --DATA_BITS;
  output dataRead_0  : 8 bits; --DATA_BITS;
  output dataRead_1  : 8 bits; --DATA_BITS;
  output dataRead_2  : 8 bits; --DATA_BITS;
  output dataRead_3  : 8 bits --DATA_BITS

) is


type COUNTER_WIDTH is 4 bits
type ADDR_BITS is 8 bits
type DATA_BITS is 8 bits



  procedure SEControl (
    sync startSE ;
    sync SE
  ) is

  begin
    loop
      select startSE then
        sync SE
      end
    end
  end

sync startSE
sync startSE_d
channel tmpAdr : ADDR_BITS
--vaiable dataRead : array 0..3 of DATA_BITS
--variable dataRead : array 0..3 of DATA_BITS
variable dataRead: DATA_BITS
variable writeCounter: 2 bits

begin
  --BufferControl (10, startSE,  startSE_d)  ||
  --SEControl (startSE_d, SE)              ||
  --AddressControl ( tmpAdr, adr)          ||
  loop
    select count then
      case count of
         1 then tmpAdr <- (4 as  ADDR_BITS) --// sync startSE
      |  2 then tmpAdr <- (8 as  ADDR_BITS) --// sync startSE
      |  3 then tmpAdr <- (12 as ADDR_BITS) --// sync startSE
      |  4 then tmpAdr <- (16 as ADDR_BITS) --// sync startSE
         else  continue
      end
    end
  end

  ||

  loop
    tmpAdr -> then
      adr <- tmpAdr;
      --sync startSE;
      --sync startSE
      --SE -> dataRead[writeCounter];
      SE -> dataRead;
      case writeCounter of
         0 then dataRead_0 <-  dataRead
      |  1 then dataRead_1 <-  dataRead
      |  2 then dataRead_2 <-  dataRead
      |  3 then dataRead_3 <-  dataRead
         --else continue
      end;
      --dataRead2[writeCounter] <- dataRead;
      writeCounter := (writeCounter + 1 as 2 bits)
    end
  end
end
```

A.5 FlashWrapper, single rail

```
`timescale 1ps / 1ps


module FlashWrapper#(
                      parameter ADDR_X_WIDTH = 4,
                      parameter ADDR_Y_WIDTH = 3,
                      parameter DOUT_WIDTH   = 8
                    )
                    (
                      input  logic                                 arst
                          ,
                      input  logic
                          controlFlash_r,
                      output logic
                          controlFlash_a,
                      input  logic
                          controlFlash_d,
                      input  logic
                          SE_Req,
                      output logic [DOUT_WIDTH-1:0]
                          SE_Data,
                      output logic
                          SE_Ack,
                      input  logic
                          adr_Req,
                      input  logic [ADDR_X_WIDTH+ADDR_Y_WIDTH-1:0]
                          adr_Data,
                      output logic
                          adr_Ack,
                      input  logic [3:0]
                          dataRead_r,
                      output logic [3:0]
                          dataRead_a,
                      input  logic [3:0][DOUT_WIDTH-1:0]
                          dataRead_d
                    );



  logic                       IFREN;
  logic  [1:0]                REDEN;
  logic  [ADDR_X_WIDTH-1:0]   XADR;
  logic                       XE;
  logic  [ADDR_Y_WIDTH-1:0]   YADR;
  logic                       YE;
  logic                       PV;
  logic                       EV;
  logic                       SE;
  logic                       controlFlash;
  logic                       dataValid;
  logic  [DOUT_WIDTH*4-1:0]   dataRead;
```

```
assign #1ns controlFlash_a = controlFlash_r;
always_latch begin
  if (arst)
    controlFlash = 0;
  else if(controlFlash_r)
    controlFlash = controlFlash_d;
end

always_comb begin
  if (controlFlash == 1) begin
    IFREN = 0;
    REDEN = 0;
    XADR  = adr_Data[ADDR_X_WIDTH-1:0];
    XE    = 1'b1;
    YADR  = adr_Data[ADDR_X_WIDTH+ADDR_Y_WIDTH-1:ADDR_X_WIDTH];
    YE    = 1'b1;
    PV    = 0;
    EV    = 0;
    SE    = SE_Req;
  end
  else begin
    IFREN = 0;
    REDEN = 0;
    XADR  = 0;
    XE    = 1'b0;
    YADR  = 0;
    YE    = 1'b0;
    PV    = 0;
    EV    = 0;
    SE    = 0;
  end
end



assign #1ns adr_Ack = adr_Req;
assign SE_Ack = SE_Req & dataValid;

Flash    #(
          .ADDR_X_WIDTH  (ADDR_X_WIDTH),
          .ADDR_Y_WIDTH  (ADDR_Y_WIDTH),
          .DOUT_WIDTH    (DOUT_WIDTH )
           )
  u_Flash (
           .IFREN      (IFREN),
           .REDEN      (REDEN),
           //.XADR       (adr_Data[ADDR_X_WIDTH-1:0]),
           .XADR       (XADR),
           .XE         (XE),
              //.YADR        (adr_Data[ADDR_X_WIDTH+ADDR_Y_WIDTH-1:
                  ADDR_X_WIDTH]),
              .YADR        (YADR),
              .YE          (YE),
              .PV          (PV),
              .EV          (EV),
              .SE          (SE_Req),
              .DATAVALID   (dataValid),
              .DOUT        (SE_Data)
              );



  assign #1ns dataRead_a = dataRead_r;
  always_latch  begin
    if (arst)
      dataRead = 0;
    else begin
      if(dataRead_r[0])
        dataRead[7:0]   = dataRead_d;
      if(dataRead_r[1])
        dataRead[15:8]  = dataRead_d;
      if(dataRead_r[2])
        dataRead[23:16] = dataRead_d;
      if(dataRead_r[3])
        dataRead[31:23] = dataRead_d;
    end
  end

//initial begin
//  IFREN = 0;
//  REDEN = 0;
//  PV = 0;
//  EV =0;
//  XE = 1'b1;
//  YE = 1'b1;
//end


endmodule
```

A.6 FlashWrapper, dual rail

```
‘timescale 1ps / 1ps


module FlashWrapper#(
                      parameter ADDR_X_WIDTH = 4,
                      parameter ADDR_Y_WIDTH = 3,
                      parameter DOUT_WIDTH   = 8
                    )
                    (
                      input  logic                                  arst
                          ,
                      input  logic
                          controlFlash_0r0d,
                      input  logic
                          controlFlash_0r1d,
                      output logic
                          controlFlash_a,
                      input  logic
                          SE_Req,
                      output logic [DOUT_WIDTH-1:0]
                          SE_0a0d,
                      output logic [DOUT_WIDTH-1:0]
                          SE_0a1d,
                      input  logic [ADDR_X_WIDTH+ADDR_Y_WIDTH-1:0]
                          adr_0r0d,
                      input  logic [ADDR_X_WIDTH+ADDR_Y_WIDTH-1:0]
                          adr_0r1d,
                      output logic
                          adr_Ack,
                      input  logic [3:0][DOUT_WIDTH-1:0]
                          dataRead_0r0d,
                      input  logic [3:0][DOUT_WIDTH-1:0]
                          dataRead_0r1d,
                      output logic [3:0]
                          dataRead_a
                    );


  logic [ADDR_X_WIDTH+ADDR_Y_WIDTH-1:0] adr_Data;
  logic                                 IFREN;
  logic  [1:0]                          REDEN;
  logic  [ADDR_X_WIDTH-1:0]             XADR;
  logic                                 XE;
  logic  [ADDR_Y_WIDTH-1:0]             YADR;
  logic                                 YE;
  logic                                 PV;
  logic                                 EV;
  logic                                 SE;
  logic                                 controlFlash;
  logic                                 dataValid;
  logic [ADDR_X_WIDTH+ADDR_Y_WIDTH-1:0] DOUT;
logic  [DOUT_WIDTH*4-1:0]             dataRead;



Flash    #(
          .ADDR_X_WIDTH  (ADDR_X_WIDTH),
          .ADDR_Y_WIDTH  (ADDR_Y_WIDTH),
          .DOUT_WIDTH    (DOUT_WIDTH )
           )
  u_Flash (
           .IFREN      (IFREN),
           .REDEN      (REDEN),
           .XADR       (XADR),
           .XE         (XE),
           .YADR       (YADR),
           .YE         (YE),
           .PV         (PV),
           .EV         (EV),
           .SE         (SE_Req),
           .DATAVALID  (dataValid),
           .DOUT       (DOUT)
           );



DualRailLatch latch_controlFlash (.arst(arst), .S(controlFlash_0r1d),
     .R(controlFlash_0r0d), .Q(controlFlash), .Qn(), .ack(
    controlFlash_a));



always_comb begin
  if (controlFlash == 1) begin
    IFREN = 0;
    REDEN = 0;
    XADR  = adr_Data[ADDR_X_WIDTH-1:0];
    XE    = 1'b1;
    YADR  = adr_Data[ADDR_X_WIDTH+ADDR_Y_WIDTH-1:ADDR_X_WIDTH];
    YE    = 1'b1;
    PV    = 0;
    EV    = 0;
    SE    = SE_Req;
  end
  else begin
    IFREN = 0;
    REDEN = 0;
    XADR  = 0;
    XE    = 1'b0;
    YADR  = 0;
    YE    = 1'b0;
    PV    = 0;
    EV    = 0;
```

```
      SE    = 0;
    end
  end



  DualRailLatch #(.BITS(ADDR_X_WIDTH+ADDR_Y_WIDTH))
    latch_adr(.arst(arst), .S(adr_0r1d), .R(adr_0r0d), .Q(adr_Data), .
        Qn(), .ack(adr_Ack));


  assign SE_Ack = SE_Req & dataValid;

  assign SE_0a1d = {ADDR_X_WIDTH+ADDR_Y_WIDTH{SE_Req & dataValid}} &
      DOUT;
  assign SE_0a0d = {ADDR_X_WIDTH+ADDR_Y_WIDTH{SE_Req & dataValid}} & ~
      DOUT;


  DualRailLatch #(.BITS(ADDR_X_WIDTH+ADDR_Y_WIDTH))
    latch_dataRead_0(.arst(arst), .S(dataRead_0r1d[0]), .R(
        dataRead_0r0d[0]), .Q(dataRead[7:0]),   .Qn(), .ack(dataRead_a
        [0]));

  DualRailLatch #(.BITS(ADDR_X_WIDTH+ADDR_Y_WIDTH))
    latch_dataRead_1(.arst(arst), .S(dataRead_0r1d[1]), .R(
        dataRead_0r0d[1]), .Q(dataRead[15:8]),  .Qn(), .ack(dataRead_a
        [1]));

  DualRailLatch #(.BITS(ADDR_X_WIDTH+ADDR_Y_WIDTH))
    latch_dataRead_2(.arst(arst), .S(dataRead_0r1d[2]), .R(
        dataRead_0r0d[2]), .Q(dataRead[23:16]), .Qn(), .ack(dataRead_a
        [2]));

  DualRailLatch #(.BITS(ADDR_X_WIDTH+ADDR_Y_WIDTH))
    latch_dataRead_3(.arst(arst), .S(dataRead_0r1d[3]), .R(
        dataRead_0r0d[3]), .Q(dataRead[31:24]), .Qn(), .ack(dataRead_a
        [3]));

endmodule


module DualRailLatch #(parameter BITS = 1)
                      (
                      input  logic arst,
                      input  logic [BITS-1:0] S,
                      input  logic [BITS-1:0] R,
                      output logic [BITS-1:0] Q,
                      output logic [BITS-1:0] Qn,
                      output logic            ack
                      );
  logic [BITS-1:0] ack_int;

  genvar i;
  generate
    for (i=0 ; i<BITS ; i++) begin

      logic Qint;
      logic Qint_n;

      always_comb
        if(R[i] | arst) begin
          Qint   = 1'b0;
          Qint_n = 1'b1;
        end
        else if (S[i]) begin
          Qint   = 1'b1;
          Qint_n = 1'b0;
        end

      assign #1ns Q[i]  = Qint;
      assign #1ns Qn[i] = Qint_n;
      assign #1ns ack_int[i] = S[i] & Q[i] | R[i] & Qn[i];
    end
  endgenerate

  assign ack = &ack_int;

endmodule
```

A.7 Flash

```
‘timescale 1ps / 1ps


module Flash#(
              parameter ADDR_X_WIDTH = 4,
              parameter ADDR_Y_WIDTH = 3,
              parameter DOUT_WIDTH   = 8
             )
             (
              input                       IFREN,
              input  [1:0]                REDEN,
              input  [ADDR_X_WIDTH-1:0]   XADR,
              input                       XE,
              input  [ADDR_Y_WIDTH-1:0]   YADR,
              input                       YE,
              input                       PV,
              input                       EV,
              input                       SE,
              output                      DATAVALID,
              output  [DOUT_WIDTH-1:0]    DOUT
             );


  time Tdh  = 0.5ns;
  time Tacc = 30ns;
  time Tpws = 5ns;

  logic [ADDR_Y_WIDTH + ADDR_X_WIDTH -1 :0]
      addr;
  logic [DOUT_WIDTH-1:0]
      dataAtAddr;
  logic [2**ADDR_Y_WIDTH-1:0][2**ADDR_X_WIDTH-1:0][DOUT_WIDTH-1:0]  mem
      ;

  logic
      valid;
  logic
      dataValid;
  logic [DOUT_WIDTH-1:0]
      dOut;

  time
      SEset;

  logic
      inputValid;


  assign addr = {YADR & {ADDR_Y_WIDTH{YE}}, XADR & {ADDR_X_WIDTH{XE}}
      };
  assign DATAVALID = dataValid;
assign DOUT      = dOut;

initial begin
  $timeformat(-6,3," us",0);

  dataValid = 1'b0;
  valid     = 1'b0;
  for (int y = 0 ; y<(2**ADDR_Y_WIDTH) ; y++) begin
    for (int x = 0 ; x<(2**ADDR_X_WIDTH) ; x++) begin
      mem[y][x]       = $urandom;
    end
  end
end

assign inputValid =  IFREN !== 1'bX && IFREN !== 1'bz &&
                     ^REDEN !== 1'bx &&
                     PV !== 1'bX && PV !== 1'bZ &&
                     EV !== 1'bX && EV !== 1'bZ &&
                     XE !== 1'bX && XE !== 1'bZ &&
                     YE !== 1'bX && YE !== 1'bZ &&
                     ^XADR  !== 1'bx &&
                     ^YADR  !== 1'bx &&
                     SE !== 1'bX && SE !== 1'bZ;

always begin
  @(addr, XE, YE)
    if (!inputValid) begin         // Give out X if any of the
        control signals are X
      #Tdh dataAtAddr = {DOUT_WIDTH{1'bX}};
      valid = 1'b0;
    end
    else if (XE === 1'b1 && YE === 1'b1) begin
      #Tdh dataAtAddr = mem[YADR][XADR];
      valid = 1'b1;
    end
    else begin    // else if (XE === 1'b0 || YE === 1'b0)
                               // Give X out if addresses not enabled
      #Tdh dataAtAddr = {DOUT_WIDTH{1'bX}};
      valid = 1'b0;
    end
end


always begin
  fork
    begin
      @(posedge SE)
        #Tacc dOut = dataAtAddr;
        dataValid  = valid;
    end
    begin
      @dataAtAddr
```

```
          dOut = {DOUT_WIDTH{1'bX}};
          dataValid  = 1'b0;
      end
    join_any;
    disable fork;
  end


  always begin
    @(posedge SE)
      SEset = $time;
    @(negedge SE)
      assert ($time - SEset >= Tpws)
        //$display("%t: SE long enough: %t", $time, $time - SEset);
      else begin
        $error("%t: SE not long enough. Must be minimum %t, but is %t",
             $time, Tpws, $time - SEset);
        dOut = {DOUT_WIDTH{1'bX}};
      end
  end

endmodule
```

A.8 Muller C

```
//
    ######################################################################
//##         Copyright (c) 2010 Nordic Semiconductor ASA, Norway
//
    ######################################################################
//## Created        : 17.09.2014
//## Modified       : $Author: Sonal Khosla
//## Description    :
//
    ######################################################################


module C2 (
                    input  logic A,
                    input  logic B,
                    input  logic arst,
                    output logic Q
               );

`ifdef FPGA

   logic ck;
   logic set;
   logic reset;
   logic resetWire;
   logic C;

   HinstAnd2 u_AndGateSet(
    .a(A),
    .b(B),
    .y(set)
   );

   HinstAnd2 u_AndGateReset(
    .a(~B),
    .b(~A),
    .y(reset)
   );

   assign ck    = 0;
   assign resetWire = reset| arst;

  // synopsys async_set_reset "set"
  // synopsys async_set_reset "resetWire"

   always @(set or resetWire) begin
     if(resetWire)begin
       C = 0;
     end
     else if(set)begin
       C = 1;
     end
   end

`else

  logic C;

  MC2CNX2_ES34 u_MullerC(
    .Z (C),
    .A (A),
    .B (B),
    .RN (~arst)
  );
  //assign C = !arst ? 1'b0 : A & B ;


`endif

   `ifdef RTL
      assign #2ns Q = C;
    `else
      assign Q = C;
    `endif

endmodule



module C3 (
                    input  logic A,
                    input  logic B,
                    input  logic C,
                    input  logic arst,
                    output logic Q
               );


//`ifdef FPGA

   logic ck;
   logic set;
   logic reset;
   logic resetWire;
   logic Y;

   HinstAnd3 u_AndGateSet(
    .a(A),
    .b(B),
    .c(C),
    .y(set)
   );

   HinstAnd3 u_AndGateReset(
    .a(~B),
    .b(~A),
    .c(~C),
    .y(reset)
   );

   assign ck     = 0;
   assign resetWire = reset| arst;

  // synopsys async_set_reset "set"
  // synopsys async_set_reset "resetWire"

   always @(set or resetWire) begin
     if(resetWire)begin
       Y = 0;
     end
     else if(set)begin
       Y = 1;
     end
//      else begin
//        Y <= 0;
//      end
   end

//'else
//
//  logic C;
//
//  MC2CNX2_ES34 u_MullerC(
//    .Z (C),
//    .A (A),
//    .B (B),
//    .RN (~arst)
//  );
//  //assign C = !arst ? 1'b0 : A & B ;
//
//
//'endif

   'ifdef RTL
      assign #2ns Q = Y;
    'else
      assign Q = Y;
    'endif

endmodule
```

A.9 Netlist for the asynchronous flash readout

The produced netlists for the asynchronous flash readout are too big to be printed, but they are provided in the attached zip-file. They can be found in the folder

```
ip\AsynchronousFlashReadout\balsa
```

as

```
impl-AsynchronousFlashReadout-FlashReadout.v
impl-AsynchronousFlashReadout-DualRail.v
```

for the single rail netlist and the dual rail netlist respetively.

A.10 Automatically generated testbench for the asynchronous flash readout

The automatically generated testbench produced for the asynchronous flash readout is too big to be printed, but it is provided in the attached zip-file. The files can be found in the folder

```
ip\AsynchronousFlashReadout\balsa
```

as

```
test-AsynchronousFlashReadoutTb-AsyncFlashReadout-top.v
impl-AsynchronousFlashReadoutTb-AsyncFlashReadout.v
```

A.11 Testbench, single rail

```
‘timescale 1ps / 1ps

module test_FlashWrapper();

  localparam ADDR_X_WIDTH = 4;
  localparam ADDR_Y_WIDTH = 4;
  localparam DOUT_WIDTH   = 8;

  logic SE_Req;
  logic [DOUT_WIDTH-1:0] SE_Data;
  logic SE_Ack;
  logic adr_Req;
  logic adr_Ack;
  logic [ADDR_X_WIDTH+ADDR_Y_WIDTH-1:0] adr_Data;
  logic activate_Req;
  logic activate_Ack;
  logic start_Req;
  logic start_Ack;
logic arst;

logic                       controlFlash_r;
logic                       controlFlash_a;
logic                       controlFlash_d;
logic [3:0]                 dataRead_r;
logic [3:0]                 dataRead_a;
logic [3:0][DOUT_WIDTH-1:0] dataRead_d;

logic  count;

    FlashWrapper#(
                .ADDR_X_WIDTH  (ADDR_X_WIDTH),
                .ADDR_Y_WIDTH  (ADDR_Y_WIDTH),
                .DOUT_WIDTH    (DOUT_WIDTH )
              )
    u_FlashWrapper(
                .arst              (arst),
                .controlFlash_r    (controlFlash_r),
                .controlFlash_a    (controlFlash_a),
                .controlFlash_d    (controlFlash_d),
                .SE_Req            (SE_Req),
                .SE_Data           (SE_Data),
                .SE_Ack            (SE_Ack),
                .adr_Req           (adr_Req),
                .adr_Data          (adr_Data),
                .adr_Ack           (adr_Ack),
                .dataRead_r        (dataRead_r),
                .dataRead_a        (dataRead_a),
                .dataRead_d        (dataRead_d)
               );

    Balsa_AsynchronousFlashReadout u_Balsa_AsynchronousFlashReadout(
               .initialise       (arst),
               .activate_0r      (activate_Req),
               .activate_0a      (activate_Ack),
               .start_0r         (start_Req),
               .start_0a         (start_Ack),
               .controlFlash_0r  (controlFlash_r),
               .controlFlash_0a  (controlFlash_a),
               .controlFlash_0d  (controlFlash_d),
               .SE_0r            (SE_Req),
               .SE_0a            (SE_Ack),
               .SE_0d            (SE_Data),
               .adr_0r           (adr_Req),
               .adr_0a           (adr_Ack),
               .adr_0d           (adr_Data),
               .dataRead__0_0r   (dataRead_r[0]),
               .dataRead__1_0r   (dataRead_r[1]),
               .dataRead__2_0r   (dataRead_r[2]),
               .dataRead__3_0r   (dataRead_r[3]),
               .dataRead__0_0a   (dataRead_a[0]),
                    .dataRead__1_0a  (dataRead_a[1]),
                    .dataRead__2_0a  (dataRead_a[2]),
                    .dataRead__3_0a  (dataRead_a[3]),
                    .dataRead__0_0d  (dataRead_d[0]),
                    .dataRead__1_0d  (dataRead_d[1]),
                    .dataRead__2_0d  (dataRead_d[2]),
                    .dataRead__3_0d  (dataRead_d[3])
                   );



      //Balsa_Counter u_Balsa_AsynchronousFlashReadout(
      //                .initialise      (arst),
      //                .activate_0r     (activate_Req),
      //                .activate_0a     (activate_Ack),
      //                .start_0r        (start_Req),
      //                .start_0a        (start_Ack),
      //                .count_0r        (count),
      //                .count_0a        (count),
      //                .count_0d        ()
      //               );

initial begin

  $timeformat(-6,3," us",0);
  arst = 1'b1;
  activate_Req = 0;
  start_Req = 0;
  #100ns;
  arst = 0;
  #10ns;
  activate_Req = 1'b1;
  #1ns;
  start_Req = 1'b1;
  #100ns;


end


endmodule
```

A.12 Testbench, dual rail

```
‘timescale 1ps / 1ps

module test_FlashWrapper();

  localparam ADDR_X_WIDTH = 4;
  localparam ADDR_Y_WIDTH = 4;
  localparam DOUT_WIDTH   = 8;

  logic SE_Req;
  logic [DOUT_WIDTH-1:0] SE_Data;
  logic SE_Ack;
  logic adr_Req;
  logic adr_Ack;
  logic [ADDR_X_WIDTH+ADDR_Y_WIDTH-1:0] adr_Data;
  logic activate_Req;
  logic activate_Ack;
  logic start_Req;
  logic start_Ack;
  logic arst;

  logic                      controlFlash_r;
  logic                      controlFlash_a;
  logic                      controlFlash_d;
  logic [3:0]                dataRead_r;
  logic [3:0]                dataRead_a;
  logic [3:0][DOUT_WIDTH-1:0] dataRead_d;

  logic  count;

      FlashWrapper#(
                   .ADDR_X_WIDTH  (ADDR_X_WIDTH),
                   .ADDR_Y_WIDTH  (ADDR_Y_WIDTH),
                   .DOUT_WIDTH    (DOUT_WIDTH )
                 )
      u_FlashWrapper(
                   .arst             (arst),
                   .controlFlash_r   (controlFlash_r),
                   .controlFlash_a   (controlFlash_a),
                   .controlFlash_d   (controlFlash_d),
                   .SE_Req           (SE_Req),
                   .SE_Data          (SE_Data),
                   .SE_Ack           (SE_Ack),
                   .adr_Req          (adr_Req),
                   .adr_Data         (adr_Data),
                   .adr_Ack          (adr_Ack),
                   .dataRead_r       (dataRead_r),
                   .dataRead_a       (dataRead_a),
                   .dataRead_d       (dataRead_d)
                  );

      Balsa_AsynchronousFlashReadout u_Balsa_AsynchronousFlashReadout(
                    .initialise      (arst),
                    .activate_0r     (activate_Req),
                    .activate_0a     (activate_Ack),
                    .start_0r        (start_Req),
                    .start_0a        (start_Ack),
                    .controlFlash_0r (controlFlash_r),
                    .controlFlash_0a (controlFlash_a),
                    .controlFlash_0d (controlFlash_d),
                    .SE_0r           (SE_Req),
                    .SE_0a           (SE_Ack),
                    .SE_0d           (SE_Data),
                    .adr_0r          (adr_Req),
                    .adr_0a          (adr_Ack),
                    .adr_0d          (adr_Data),
                    .dataRead__0_0r  (dataRead_r[0]),
                    .dataRead__1_0r  (dataRead_r[1]),
                    .dataRead__2_0r  (dataRead_r[2]),
                    .dataRead__3_0r  (dataRead_r[3]),
                    .dataRead__0_0a  (dataRead_a[0]),
                    .dataRead__1_0a  (dataRead_a[1]),
                    .dataRead__2_0a  (dataRead_a[2]),
                    .dataRead__3_0a  (dataRead_a[3]),
                    .dataRead__0_0d  (dataRead_d[0]),
                    .dataRead__1_0d  (dataRead_d[1]),
                    .dataRead__2_0d  (dataRead_d[2]),
                    .dataRead__3_0d  (dataRead_d[3])
                   );


      //Balsa_Counter u_Balsa_AsynchronousFlashReadout(
      //              .initialise      (arst),
      //              .activate_0r     (activate_Req),
      //              .activate_0a     (activate_Ack),
      //              .start_0r        (start_Req),
      //              .start_0a        (start_Ack),
      //              .count_0r        (count),
      //              .count_0a        (count),
      //              .count_0d        ()
      //             );

initial begin

  $timeformat(-6,3," us",0);
  arst = 1'b1;
  activate_Req = 0;
  start_Req = 0;
  #100ns;
  arst = 0;
  #10ns;
  activate_Req = 1'b1;
  #1ns;
  start_Req = 1'b1;
  #100ns;


end


endmodule
```

Appendix B
Toy design

B.1 Buffer

```
-- Design for BufferControl
import [balsa.types.basic]

procedure buffer1 (input i : 1 bits; output o : 1 bits) is
  variable x : 1 bits
begin
  loop
    i -> x          -- Input  communication
    ;               -- sequence operator
    o <- x          -- Output communication
  --i -> o
  end
end
```

B.2 Buffer netlist

```
/*
    ‘impl-buffer-toyNetlist.v’
    Balsa Verilog netlist file
    Created: Tue Jun 16 22:55:03 CEST 2015
    By: sokh@bushmills.nvlsi.no (Linux)
    With net-verilog (balsa-netlist) version: 4.0
    Using technology: BorealisTech/dual_b/sim=modelsim:logic=dims
    Command line : (balsa-netlist --technology BorealisTech/dual_b/sim=
        modelsim:logic=dims -I . -l impl-buffer-toyNetlist.lst -L impl-
        buffer-toyNetlist.log --simulation-initialise --basename impl-
        buffer-toyNetlist [buffer])

    Using ‘simulation-initialise’
    You must set the following preprocessor directives to use this file
        :
        balsa_simulate: set if you wish to initialise signal values
            during sim.
        balsa_init_time: duration of forced initialisation

    Using ‘propagate-globals’
    The design contains the following global nets
                global-signal:  initialise input 1
*/

‘timescale 1ns/1ps

module BrzFetch_1_s5_false (
  activate_0r, activate_0a,
  inp_0r, inp_0a0d, inp_0a1d,
  out_0r0d, out_0r1d, out_0a
);
  input activate_0r;
  output activate_0a;
  output inp_0r;
  input inp_0a0d;
  input inp_0a1d;
  output out_0r0d;
  output out_0r1d;
  input out_0a;
  BUFFD1BWP7T I0 (activate_0r, inp_0r);
  BUFFD1BWP7T I1 (inp_0a0d, out_0r0d);
  BUFFD1BWP7T I2 (inp_0a1d, out_0r1d);
  BUFFD1BWP7T I3 (out_0a, activate_0a);
endmodule

module BrzLoop (
  activate_0r, activate_0a,
  activateOut_0r, activateOut_0a
);
  input activate_0r;
  output activate_0a;
  output activateOut_0r;
  input activateOut_0a;
  wire nReq_0n;
  supply0 gnd;
  INVD1BWP7T I0 (activate_0r, nReq_0n);
  NR2D1BWP7T I1 (nReq_0n, activateOut_0a, activateOut_0r);
  BUFFD1BWP7T I2 (gnd, activate_0a);
endmodule

module telemr (
  Ar,
  Aa,
  Br,
  Ba,
  initialise
);
  input Ar;
  output Aa;
  output Br;
  input Ba;
  input initialise;
  wire s_0n;
  wire nreset_0n;
  MC2CNX2_ES34 I0 (Aa, Ba, Ar, nreset);
  INVD1BWP7T I1 (initialise, nreset);
  INVD1BWP7T I2 (Aa, s_0n);
  AN2D1BWP7T I3 (Ar, s_0n, Br);
endmodule

module BrzSequence_2_s1_T (
  activate_0r, activate_0a,
  activateOut_0r, activateOut_0a,
  activateOut_1r, activateOut_1a,
  initialise
);
  input activate_0r;
  output activate_0a;
  output activateOut_0r;
  input activateOut_0a;
  output activateOut_1r;
  input activateOut_1a;
  input initialise;
  wire [1:0] sreq_0n;
  BUFFD1BWP7T I0 (activateOut_1a, activate_0a);
  BUFFD1BWP7T I1 (sreq_0n[1], activateOut_1r);
  BUFFD1BWP7T I2 (activate_0r, sreq_0n[0]);
  telemr I3 (sreq_0n[0], sreq_0n[1], activateOut_0r, activateOut_0a,
      initialise);
endmodule

module dualrail_latch (
  in_0,
  in_1,
  in_a,
  out_0,
  out_1,
  initialise
);
  input in_0;
  input in_1;
  output in_a;
  output out_0;
  output out_1;
  input initialise;
  wire clr;
  RSLATX1 I0 (out_1, out_0, clr, in_1);
  AO22D0BWP7T I1 (in_0, out_0, in_1, out_1, in_a);
  OR2D1BWP7T I2 (in_0, initialise, clr);
endmodule

module BrzVariable_1_1_s0_ (
  write_0r0d, write_0r1d, write_0a,
  read_0r, read_0a0d, read_0a1d,
  initialise
);
  input write_0r0d;
  input write_0r1d;
  output write_0a;
  input read_0r;
  output read_0a0d;
  output read_0a1d;
  input initialise;
  wire store_0n;
  wire store_1n;
  wire ldata_0n;
  wire ldata_1n;
  wire wack_0n;
  wire readReq_0n;
  AN2D1BWP7T I0 (store_1n, readReq_0n, read_0a1d);
  AN2D1BWP7T I1 (store_0n, readReq_0n, read_0a0d);
  BUFFD1BWP7T I2 (read_0r, readReq_0n);
  BUFFD1BWP7T I3 (ldata_0n, store_0n);
  BUFFD1BWP7T I4 (ldata_1n, store_1n);
  BUFFD1BWP7T I5 (wack_0n, write_0a);
  dualrail_latch I6 (write_0r0d, write_0r1d, wack_0n, ldata_0n,
      ldata_1n, initialise);
endmodule

module Balsa_buffer1 (
  activate_0r, activate_0a,
  i_0r, i_0a0d, i_0a1d,
  o_0r0d, o_0r1d, o_0a,
  initialise
);
```

```
  input activate_0r;
  output activate_0a;
  output i_0r;
  input i_0a0d;
  input i_0a1d;
  output o_0r0d;
  output o_0r1d;
  input o_0a;
  input initialise;
  wire c8_r;
  wire c8_a;
  wire c7_r;
  wire c7_a;
  wire c6_r0d;
  wire c6_r1d;
  wire c6_a;
  wire c5_r;
  wire c5_a;
  wire c4_r;
  wire c4_a0d;
  wire c4_a1d;
  BrzVariable_1_1_s0_ I0 (c6_r0d, c6_r1d, c6_a, c4_r, c4_a0d, c4_a1d,
      initialise);
  BrzLoop I1 (activate_0r, activate_0a, c8_r, c8_a);
  BrzSequence_2_s1_T I2 (c8_r, c8_a, c7_r, c7_a, c5_r, c5_a, initialise
      );
  BrzFetch_1_s5_false I3 (c7_r, c7_a, i_0r, i_0a0d, i_0a1d, c6_r0d,
      c6_r1d, c6_a);
  BrzFetch_1_s5_false I4 (c5_r, c5_a, c4_r, c4_a0d, c4_a1d, o_0r0d,
      o_0r1d, o_0a);
endmodule
```

B.3 Toy Verilog

```
// test module ToyVerilog


module ToyVerilog #()

                          (
                           input  logic   ck,
                           input logic    ckDataIn,
                           input logic    ckDataOut,
                           input  logic   arst,
                           output logic   initialise,
                           output logic   activate_0r,
                           input logic    activate_0a,
                           input logic    i_0r,
                           output logic   i_0a0d,
                           output logic   i_0a1d,
                           input logic    o_0r0d,
                           input logic    o_0r1d,
                           output logic   o_0a,
                           input logic    a,
                           input logic    b,
                           output logic   req,
                           output logic   c,
                           output logic   d,
                           input logic    ack
                   //                        output logic dataRead
                          );



//  logic i_0r_ax;
//  logic i_0r_s;
//  //logic i_0r_edgeDetect;
//  logic [2:0] count;
//  static logic [7:0] data = 8'b10100011;
//  logic d1_ax;




  typedef enum logic [1:0] {InitialiseOn        = 0,
                           InitialiseOff       = 1,
                           ActivateOn          = 2,
                           ActivateOff         = 3
                           } StateType;


//  typedef enum logic         {ReleaseData         = 0,
//                             DriveData           = 1
//                             } StateSendData;
```

```
  StateType state;
//  StateSendData stateSendData;

  always_ff @(posedge ck, posedge arst)
    if (arst) begin
      initialise <= 0;
      activate_Or <= 0;
      state <= InitialiseOn;
    end
    else begin
      case (state)

        InitialiseOn: begin
                        initialise <= 1'b1;
                        state <= InitialiseOff;
                      end

        InitialiseOff: begin
                        initialise <= 1'b0;
                        state <= ActivateOn;
                       end

        ActivateOn:   begin
                        activate_Or <= 1'b1;
                        if (activate_Oa == 1'b1)
                          state <= ActivateOff;
                      end

        ActivateOff:  begin
                        activate_Or <= 1'b0;
                        state <= ActivateOff;
                      end

      endcase

    end


  always_ff @(posedge ckDataIn, posedge arst)
     if(arst) begin
       i_Oa0d <= 0;
       i_Oa1d <= 0;
     end
     else begin
       i_Oa0d <= a;
       i_Oa1d <= b;
     end

 assign req = i_Or;
```

```
  always_ff @(posedge ckDataOut, posedge arst)
     if(arst) begin
       o_0a <= 0;
     end
     else begin
       o_0a <= ack;
     end

assign c = o_0r0d;
assign d = o_0r1d;
//
//   always_ff @(posedge ck, posedge arst)
//     if (arst) begin
//      i_0r_ax <= 0;
//      i_0r_s  <= 0;
//      //i_0r_edgeDetect <= 0;
//     end
//     else begin
//       i_0r_ax <= i_0r;
//       i_0r_s <= i_0r_ax;
//       //i_0r_edgeDetect <= i_0r_s;
//     end
//
//     //assign ne = (~i_0r_s) & i_0r_edgeDetect;
//     //assign pe = (~i_0r_edgeDetect) & i_0r_s;
//
//
//   always_ff @(posedge ck, posedge arst)
//      if(arst) begin
//        i_0a0d <= 0;
//        i_0a1d <= 0;
//        stateSendData <= ReleaseData;
//        count <= 0;
//
//      end
//      else begin
//        case(stateSendData)
//          ReleaseData: begin
//                         i_0a0d <= 0;
//                         i_0a1d <= 0;
//                         if(i_0r_s == 1) begin
//                           stateSendData <= DriveData;
//                         end
//                       end
//
//          DriveData:   begin
//                         i_0a0d <= ~data[count];
//                         i_0a1d <= data[count];
//                         if(i_0r_s == 0) begin
//                           stateSendData <= ReleaseData;
//                           count = count + 1;
//                         end
//                        end
//
//
//         endcase
//       end
//
//
//
//       RSLATX1 u_RSLAT(.Q(d1), .QN(d0), .R(o_0r0d | arst), .S(o_0r1d) )
    ;
//       assign o_0a = (d1 & o_0r1d) | (d0 & o_0r0d);
//       always_ff @(posedge ck, posedge arst)
//         if (arst) begin
//          d1_ax <= 0;
//          dataRead  <= 0;
//         end
//         else begin
//           d1_ax <= d1;
//           dataRead <= d1_ax;
//         end
//
//


endmodule
```

B.4 Device Under Test

```
module DUT (
            input logic ck,
            input logic arst,
            input logic ckDataIn,
            input logic ckDataOut,
            input logic a,
            input logic b,
            output logic req,
            output logic c,
            output logic d,
            input logic ack
  );



            //output logic dataRead);


  logic initialise;
  logic activate_0r;
  logic activate_0a;
  logic i_0r;
  logic i_0a0d;
  logic i_0a1d;
  logic o_0r0d;
  logic o_0r1d;
  logic o_0a;





  ToyVerilog #()

    u_ToyVerilog
        (
        .ck                         (ck),
        .ckDataIn                   (ckDataIn),
        .ckDataOut                  (ckDataOut),
        .arst                       (arst),
        .activate_0a                (activate_0a),
        .initialise                 (initialise),
        .activate_0r                (activate_0r),
        .i_0r                       (i_0r),
        .i_0a0d                     (i_0a0d),
        .i_0a1d                     (i_0a1d),
        .o_0r0d                     (o_0r0d),
        .o_0r1d                     (o_0r1d),
        .o_0a                       (o_0a),
        .a                          (a),
        .b                          (b),
        .c                          (c),
        .d                         (d),
        .req                       (req),
        .ack                       (ack)
       // .dataRead                  (dataRead)
        );

  Balsa_buffer1 #()

    u_Balsabuffer1
        (
        .activate_0r (activate_0r),
        .activate_0a (activate_0a),
        .i_0r        (i_0r),
        .i_0a0d      (i_0a0d),
        .i_0a1d      (i_0a1d),
        .o_0r0d      (o_0r0d),
        .o_0r1d      (o_0r1d),
        .o_0a        (o_0a),
        .initialise  (initialise)
        );



endmodule
```

B.5 Testbench

```
// tb for ToyVerilog

‘timescale 1ns/1ps

module test_toyVerilog();

  localparam NUM_DATA = 10000;

  logic ck;
  logic ckDataIn;
  logic ckDataOut;
  logic arst;
  logic a;
  logic b;
  logic req;
  logic c;
  logic d;
  logic ack;


  // ---------------------------------------------
  // ---------------------------------------------
  // -- Local signals:
  // ---------------------------------------------
  int errorCnt;
  int count = 0;
  int countReceive = 0;
  //logic [9:0] data = 10'b($urandom(1, 0));
  //static logic [NUM_DATA-1:0] data = 10'b1001010100;
  logic data;
  int count_a;
  int count_b;
  int count_c;
  int count_d;

  // ----------------------------------------------------------
  // -- Instantiation of DUT, EventMerger:
  // ----------------------------------------------------------

      DUT #()

          u_DUT
              (
              .ck                         (ck),
              .ckDataIn                   (ckDataIn),
              .ckDataOut                  (ckDataOut),
              .arst                       (arst),
              .a                          (a),
              .b                          (b),
              .c                          (c),
              .d                          (d),
            .req                        (req),
            .ack                        (ack)
            //.dataRead                   (dataRead)
            );



// --------------------------------------------
// -- Clocking :
// --------------------------------------------
initial begin

    //testbench clock
      ck   = 1'b 0;
        while (1) begin
          #(20ns/2) ck = ~ck;
        end
end



//
    ////////////////////////////////////////////////////////////////////////////////

//
    /////////////////////-------------------------------------------------------
    Testbench
    ------------------------------------------------------------------------

//
    ////////////////////////////////////////////////////////////////////////////////


initial begin
  $timeformat (-6, 3, "us", 0);
  errorCnt = 0;
  arst = 0;
  a=0;
  b=0;
  ack=0;
  ckDataIn    = 1'b 0;
  ckDataOut    = 1'b 0;
  #10ns;
  arst = 1;
  #10ns;
  arst = 0;




//ta_sendData(data[count]);
//ta_receiveData(data[countReceive]);
```

```
data = $urandom_range(0,1);
fork
  begin
    fork
      begin
        repeat (NUM_DATA) begin
          //ta_sendData(data[count]);
          ta_sendData(data);
          data = $urandom_range(0,1);
          //count++;
        end
      end

      begin
        repeat (NUM_DATA) begin
          //ta_receiveData(data[countReceive]);
          ta_receiveData(data);
          //countReceive++;
        end
      end
    join;
  end


  begin
    count_a = 0;
    forever begin
      @(a);
      count_a++;
    end
  end

  begin
    count_b = 0;
    forever begin
      @(b);
      count_b++;
    end
  end

  begin
    count_c = 0;
    forever begin
      @(c);
      count_c++;
      assert (count_c == count_a) else begin
        $error("%t : Glitch detected on line c", $time);
        errorCnt++;
      end
    end
  end
  begin
    count_d = 0;
    forever begin
      @(d);
      count_d++;
      assert (count_d == count_b) else begin
        $error("%t : Glitch detected on line d", $time);
        errorCnt++;
      end
    end
  end

join_any;
disable fork;

$display("total errors: %0d", errorCnt);

$display(" $$(                           $$(
                                   $$\    ");
    $display(" $$ |                          (__|
                                      $$ |   ");
    $display(" $$$$$$$(   $$$$$$\  $$$$$$$$(  $$( $$$$$$$(   $$$$$$(
          $$$$$$(   $$ |   ");
    $display(" $$  __$$(  (____$$( (____$$  |$$ |$$  __$$( $$  __$$(
        (____$$(   $$ |   ");
    $display(" $$ |  $$ | $$$$$$$ |  $$$$ _/ $$ |$$ |  $$ |$$ /  $$ |
         $$$$$$$ | (__|   ");
    $display(" $$ |  $$ |$$  __$$ | $$  _/   $$ |$$ |  $$ |$$ |  $$ |
        $$  __$$ |       ");
    $display(" $$$$$$$  |($$$$$$$ |$$$$$$$$( $$ |$$ |  $$ |($$$$$$$
        |($$$$$$$ |  $$(    ");
    $display(" (_______/  (_______|(________|(__|(__|  (__| (____$$ |
         (_______|  (__|   ");
    $display("                                             $$(   $$ |
                         ");
    $display("                                             ($$$$$$  |
                         ");
    $display("                                              (______/
                         ");
 $stop;
end




task ta_sendData(data);
  time randomDelay;

  randomDelay = $urandom_range(1,50);
  //@(posedge req);
  wait(req);
```

```
    a = ~data;
    b = data;

    #randomDelay;
    ckDataIn = 1;
    #10ns;
    ckDataIn = 0;

    randomDelay = $urandom_range(1,50);
    //$display("Delay: %d", randomDelay);
    //@(negedge req);
    wait(!req);
    a = 0;
    b = 0;
    #randomDelay;
    ckDataIn = 1;
    #10ns;
    ckDataIn = 0;

  endtask

  task ta_receiveData(dataReceive);
    time randomDelay;

    randomDelay = $urandom_range(1,50);
    //@(posedge c or posedge d);
    wait(c | d);
    assert (c == ~dataReceive) else begin
      $error ("%t: Received data on 0 not same as data sent. Expecting
          data: %0b, Received data: %0b", $time, ~dataReceive, c);
      errorCnt++;
    end
    assert (d == dataReceive) else begin
      $error ("%t: Received data on 1 not same as data sent. Expecting
          data: %0b, Received data: %0b", $time, dataReceive, d);
      errorCnt++;
    end
    ack = 1;
    #randomDelay;
    ckDataOut = 1;
    #10ns;
    ckDataOut = 0;

    randomDelay = $urandom_range( 1,50);
    //@(negedge c or negedge d);
    wait(!c & !d)
    assert (c == 0) else begin
      $error ("%t: Received data on 0 not same as data sent. Expecting
          data: 0, Received data: %0b", $time, c);
      errorCnt++;
    end
    assert (d == 0) else begin
```

```
      $error ("%t: Received data on 1 not same as data sent. Expecting
          data: 0, Received data: %0b", $time, d);
      errorCnt++;
    end
    ack = 0;
    #randomDelay;
    ckDataOut = 1;
    #10ns;
    ckDataOut = 0;
  endtask

endmodule
```

B.6 Log files for elaboration and compilation

The log files for the elaboration and the compilation of the combined Toy design are too big to be printed, but they are provided in the attached zip-file. The files can be found in the folder

```
ip\Toy\syn\logs
```

as

```
dc_elaborate.log
dc_compile.log
```

B.7 Combined netlist

The netlist for the combined Toy design is too big to be printed, but it is provided in the attached zip-file. The file can be found in the folder

```
ip\Toy\syn\results
```

as

```
DUT.mapped.v
```

Appendix C Library

C.1 make_net

The script used to generate the <library>-cells.net file.

```
#!/bin/env bash

if [ -z "$1" ]; then
  echo "No library given"
fi


inputFile=$1
maskFile=$2

parsingModule=0

mask=($(cat $maskFile | awk '! /^#/ {print $1}'))

#echo ${mask[@]}

while read line
do

  if [ $parsingModule == 0 ]; then
    module=$(echo -e "$line" | awk '/^module/ {print $2}')

    if [ -n "$module" ]; then
      for x in ${mask[@]}; do
        if [ $x = $module ]; then
          parsingModule=1
          echo -e "(circuit \"$module\"\n  (ports"
          break
        fi
      done

    fi
  fi
  if [ $parsingModule == 1 ]; then
    if [[ $line == *"endmodule"* ]]; then
      parsingModule=0
      echo -e "  )\n  (nets)\n  (instances)\n)\n"
    fi

    if [[ $line == *"input"* ]]; then
      inputList=$(echo $line | sed 's/input//' | sed 's/^ *//' | sed 's
          /,//g' | sed 's/;//g' | sed 's/ /\n/g')   # Will not work if
          multiple space between signals
      for ip in $inputList; do
        echo -e "    (\"$ip\" input 1)"
      done
    fi

    if [[ $line == *"output"* ]]; then
      inputList=$(echo $line | sed 's/output//' | sed 's/^ *//' | sed '
          s/,//g' | sed 's/;//g' | sed 's/ /\n/g')   # Will not work if
           multiple space between signals
      for ip in $inputList; do
        echo -e "    (\"$ip\" output 1)"
      done
    fi
  fi

done < $inputFile
```

C.2 startup.scm

The configuration script for the library.

```
;;;
;;; ‘BorealisTech-cells’
;;;

(net-signature-for-netlist-format 'verilog #t)

(set! breeze-gates-net-files '("BorealisTech-cells" "balsa-cells"))
(set! breeze-primitives-file (string-append breeze-tech-dir "components
    .abs"))
(set! breeze-gates-mapping-file (string-append breeze-tech-dir "gate-
    mappings"))

;;; max. no. of inputs for and/or/nand/nor gates and c-elements
(set! tech-gate-max-fan-in 3)
(set! tech-c-element-max-fan-in 2)

;;; use name mapping to keep names less than ~48 chars
(set! tech-map-cell-name (net-simple-cell-name-mapping #f))
(set! tech-map-cell-name-import net-simple-cell-name-import)
(set! tech-map-cell-name-export net-simple-cell-name-export)
(set! tech-cell-name-max-length 48)

;(set! tech-gnd-net-name "!gnd")
;(set! tech-vcc-net-name "!vdd")

(set! tech-netlist-test-includes '("BorealisTech-cells.v"))
```