Ferguson, Phillip David (2012) *Implementation exploration of imaging algorithms on FPGAs.* EngD thesis.

http://theses.gla.ac.uk/3419/

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten:Theses
http://theses.gla.ac.uk/
theses@gla.ac.uk

Implementation Exploration of Imaging Algorithms on FPGAs

Phillip David Ferguson MEng

A thesis submitted to

The Universities of

Glasgow

Strathclyde

Edinburgh

Heriot Watt

for the Degree of

Doctor of Engineering in System Level Integration

© Phillip David Ferguson, 14th May 2012

This thesis is dedicated to my mother and sister, Josephine and Katherine Ferguson, without your patience and support throughout the years this work would not have been possible. I love you both very much.

Abstract

This portfolio thesis documents the work carried out as part of the Engineering Doctorate (EngD) programme undertaken at the Institute for System Level Integration. This work was sponsored and aided by Thales Optronics Ltd, a company well versed in developing specialised electro-optical devices. Field programmable gate arrays (FPGAs) are the devices of choice for custom image processing algorithms due to their reconfigurable nature. This also makes them more economical for low volume production runs where non-recoverable engineering costs are a large factor. Asynchronous circuits have had a remarkable surge in development over the last 20 years, to such an extent that they are beginning to displace conventional designs for niche applications. Their unique ability to adapt to environmental and data dependent processing needs have lead them to out-perform synchronous designs in ASIC platforms for certain applications.

The main body of research was separated into three areas of work presented as three technical documents. The first area of research addresses an FPGA implementation of contrast limited adaptive histogram equalisation (CLAHE), an algorithm which provides increased visual performance over conventional methods. From this, a novel implementation strategy was provided along with the key design factors for future use in a commercial context. The second area of research investigates the ability to create asynchronous circuits on FPGA devices. The main motivation for this work was to establish if any of the benefits which had been demonstrated for ASIC devices can be applied to FPGA devices. The investigation surmised the most suitable asynchronous design style for FPGA devices, a design flow to allow asynchronous circuits to function correctly on FPGAs and novel design strategies to implement consistent and repeatable asynchronous components. The result of this work established a route to implement circuits asynchronously in an FPGA. The final area of research focused on a unique conversion tool that allows synchronous circuits to run asynchronously on FPGAs whilst maintaining the same data flow patterns. This research produced an automated tool capable of implementing circuits on an FPGA asynchronously from their synchronous descriptions. This approach allowed the primary motivators of this work to be addressed. The results of this work show timing, resource utilisation and noise spectrum benefits by implementing circuits asynchronously on FPGA devices.

Acknowledgements

Over the past 4 years, I have been supported by a number of people during this project and I would like to thank those that have offered me encouragement and help. Firstly I would likely to thank a number of individuals at Thales Optronics who have played a pivotal role in my studies and kept my feet grounded, both electrically and physically. I owe my industrial supervisors, Andrew Parmley and Danny Hume a significant amount of gratitude and thanks. Their insights and encouragement were always uplifting and beneficial. I would like to thanks especially Garry Widley, Adam Rixon and Steve Pattinson for instructing me in FPGA firmware design and patiently enduring my technical talks. I was never far from a sensible ear, a joke and a distracting conversation on Formula 1.

I am indebted to my academic supervisors who have been an outstanding source of insight and inspiration. Without the encouragement and suggested avenues of investigation from Tughrul Arslan and Ahmet Erdogan the initial probing of research directions would have been a much longer and troubling process. My eternal gratitude goes to Aristides Efthymiou, our discussions on asynchronous logic were invaluable. Without his support, guidance and encouragement this thesis might not have seen the light of day in the time that it has. I would also like to acknowledge Mathew Marshall for his encouragement and support in the early stages of the asynchronous logic research. A large amount of gratitude goes to my academic support mechanism, Khodor Fawaz. I will miss our conversations on research topics, life and being the only persons investigating asynchronous logic in Scotland.

I would like to address a few individuals that have kept a smile on face throughout the years and encouraged me to submit this thesis in a timely fashion. I would like acknowledge the coffee crew: The laughter was always a welcome break, never allowing the research to overwhelm me. I would like to acknowledge my good friends James Yorkshakes, Christopher Morrison, Iain Taylor and Stephen Bain. Without your support and snowboarding adventures, the days or research might not have been as much fun.

Finally I would like to especially thank my girlfriend and my extended family, Laura, Martin, Paul, Rachel, Michael, Carole, Maime, Jim and Nibbles the cat. I only hope you can forgive the time spent on this research, now it can be spent with you in front of the fire.

Declaration of Originality

I declare that this portfolio thesis was composed entirely by myself and that the work contained herein is my own except where acknowledged in the text. A list of references has been given in the bibliography of each Technical Report. I also declare that this portfolio thesis has not been submitted for any other degrees or professional qualifications at any university. I am the sole author of this thesis and any errors contained herein are my own.

(Phillip David Ferguson)

Contents

List of Figures

List of Tables

List of Abbreviations

AACIF Automated Asynchronous Circuits in FPGAs

API Application Programming Interface

ASIC Application Specific Integrated Circuit

CLAHE Contrast Limited Adaptive Histogram Equalisation

CLB Configurable Logic Block

CMOS Complementary Metal-Oxide Semiconductor

EDA Electronic Design Automation

EDIF Electronic Design Interchange Format

EM Electro Magnetic

FPGA Field Programmable Gate array

HDL Hardware Description Language

IP Intellectual Property

JTAG Joint Test Action Group

LED Light Emitting Diode

LUT Look-up Table

NAND Inverted logical AND gate

PCB Printed Circuit Board

RTEMS Real-Time Executive for Multiprocessor Systems

RTL Register-Transfer Level

SIMD Single Instruction Multiple Data

STG Signal Transition Graph

UML Unified Modelling Language

VHDL VHSIC Hardware Description Language

VLIW Very Long Instruction Word

VLSI Very Large Scale Integration

Part I

Portfolio Introduction

1.1 Executive Summary

The objective of this engineering doctorate (EngD) research project, proposed in conjunction with the Institute of System Level Integration and Thales Optronics Ltd, was to investigate how asynchronous design techniques could be applied to FPGA designs for processing raw image detector data from electro-optical sensors. This potentially included the development of a methodology and practical implementation of the methodology on working hardware. The primary aim was to improve the implementation of FPGA designs by assessing the applicability of asynchronous design techniques to the processing of detector data.

This initial brief was designed to provide a large scope, allowing flexibility in the specific research direction. Although experts in producing optical sensors and imaging algorithms, Thales Optronics, due to the nature of their business environment, are inexperienced in state of the art design methodologies outwith the standard EDA practises that have a long history of reliable hardware implementations. The research methodology adopted involved looking at an advanced, state of the art imaging algorithm, suitable for use on a number of future product lines. Then an investigation into asynchronous circuit design on a FPGA device would determine the best approach to combine the two domains. The result being a methodology to evaluate the applicability to imaging algorithms. As a result there were three distinct areas of work undertaken.

The first major period of research was the design and implementation of a contrast enhancement algorithm. Contrast limited adaptive histogram equalisation (CLAHE) is a contrast enhancement algorithm that had the potential to be utilised in forthcoming projects within Thales Optronics. This provided an opportunity to become familiar with the FPGA design flow used in the company, investigate FPGA device architectures and assess their suitability for region based contrast enhancement algorithms. The novel implementation of the CLAHE algorithm was analysed and characterised in the context of design accuracy with respect to a golden reference model, FPGA resources utilisation, device timing and power consumption. These areas allow a summary of the key design factors to be formed for future projects and products.

The second period of research initially involved investigations into design space of asynchronous logic. There are a number of different asynchronous design styles which exhibit characteristics dependant on the application. In recent times each branch of asynchronous logic has focused on a niche application, it was therefore important to determine the most suitable style for FPGA devices. Further to this an evaluation of design tools and methodologies was performed to determine existing methods of creating reliable asynchronous circuits. With each asynchronous design style came a series of tools from academic and industrial sources. Establishing their compatibility to current FPGA design flows and EDA standards was a critical in guiding the research direction. The main focus then turned to implementing a class of asynchronous circuits that use FPGA resources. This piece of research looked

into connecting the most primitive components available with the FPGA device to suit fundamental asynchronous behaviours. In doing so a design methodology was formed along with specific fabric level asynchronous structures and verification approaches. All of which integrated seamlessly into the conventional FPGA design flow.

The third major period of research moved into investigating the options on how to automate asynchronous circuits into the current FPGA design flow. A unique approach is presented which converts synchronous circuits to operate asynchronously. This approach maximises the compatibility of these circuits with existing EDA standards. The conversion focuses around the post-synthesis EDIF netlist. This process abstracts an EDIF netlist in the domain of graph theory using an object orientated approach to perform the conversion. There is a significant discussion on the stages of conversion including the constraints vital to the correct asynchronous circuit operation. A novel asynchronous controller specifically designed for FPGA devices is presented and compared. The analysis of this conversion process covers its timing accuracy and improvements, resource utilisation and power spectrum noise.

1.2 Portfolio Organisation

The introductory part of this portfolio thesis provides a context to the research period and the supporting work by the author. Firstly, the commercial context outlines what has motivated Thales Optronics Ltd to support the topics of research undertaken by the research engineer. The company has identified these topics as being beneficial to their future development. The second section details the external events that have been attended to publish research results and gain visibility within the wider research community as well as internal divisions of Thales Optronics Ltd. The third section discusses the taught elements to the EngD and external training course that have contributed to the success of this research. The taught business modules were of significant value within the industrial context, providing insight to the commercial forces around the EngD. The taught technical courses and external training provided a succinct technical context to guide the direction of research at a very early stage.

The core of this portfolio thesis is split into three technical reports that reflect the sub-projects there were undertaken during the period of research. The first report documents the research conducted investigated the use of region-based contrast enhancement within the current platforms used by Thales Optronics. The second report documents the investigations into establishing asynchronous circuits on FPGA devices. The third report presents an automated methodology to implement circuits asynchronously on FPGA devices. Each report contains individual aims and motivations as well as technical context and conclusions from for the research conducted.

The final part of this portfolio thesis summarises and concludes the contributions and novel as-

pects from each technical report and provides a direction for any future derivatives of this research. Appendices and supporting material make up the remainder of this portfolio thesis.

1.3 Commercial Relevance

Although this research provides novel technical contributions it must also be of value and commercial relevance to the sponsoring firm, Thales Optronics Ltd. This section will highlight the industrial value of the research and the commercial benefits it may bring in the future.

The first section of work on contrast enhancement produced an academically credible publication, but this project provided great value to the company in determining the FPGA resources required to implement such an algorithm on an embedded device. For thermal imaging cameras a large amount of effort is expended in determining algorithms that can enhance image quality and clarity. To this extent contrast enhancement methods are particularly important in minimising noise from an infrared photodetector and enhancing lower detail regions. FPGA devices are the choice at the heart of most thermal imaging cameras. Their configuration flexibility, performance and ability to service low volume manufacturing make them ideal to contain the required amount of corrective and enhancing image processing algorithms. Region based contrast enhancement algorithms (like CLAHE) show significant image quality benefits over image wide techniques, and so it was a logical conclusion for the firm to support the direction taken by the EngD research to investigate improving the current contrast enhancement algorithms with region based alternatives. This support could give the firm a key advantage over competitors in bringing products to market quicker. The research conducted in this area gives valid estimates on an industrial application of the same functionality. Non-recurring engineering costs are always targeted to minimise overall project costs, having a valid implementation that constitutes a more predictable element of a project massively reduces uncertainty and inevitably cost.

The second section of work on establishing asynchronous logic on FPGAs again has the potential to bring further savings and advantages to the the firm. Traditionally the large asynchronous design space has allowed for many niche applications to benefit from improved performance in terms of speed and power when utilising a particular asynchronous design style.These benefits have always been applied in the context of high-volume ASIC products. This research has identified the most appropriate method/framework that would allow low-volume FPGA products to assess these advantages within their own context. Applying various design styles to FPGA fabric could bring better performance to niche applications of the sponsoring firm. In a synchronous circuit implemented on an FPGA the clock network can consume a significant proportion of power required to operate the device. The drive current required to constantly switching an extensive clock network will affect the power supply design. The demands of many output pins switching altogether mean that printed circuit board designers need

to account for the high currents and cross coupling effects- all of which add the cost of designing an embedded system. Asynchronous systems average out power spikes as they have no clock and as a consequence this reduces supply variability which can reduce the cost of designing power supplies and PCBs. The speed of a synchronous circuits is governed by the slowest propagation delay between registers. In an asynchronous circuit the speed is governed by the average propagation delay between registers. Meaning that system latency can be significantly smaller than the synchronous equivalent. From a cost perspective, using a high speed grade device to sample input data very quickly could be replaced by a simple asynchronous circuit (with its average case performance) to sample input data just as quickly but with the benefit of using a lower speed grade device, which costs significantly less.

The third section of work on automated asynchronous circuits provides a unique ability to the firmware design capabilities of the firm. Automatically implementing circuits asynchronously on a synchronous FPGA device has a number of benefits the firm can leverage. Firstly designs do not need to learn asynchronous design techniques in order to operate circuits without a clock. This allows asynchronous benefits to be easily assessed against each circuit design. There may be applications that call for low EM noise from the PCB, response times where latency is the most important factor, power efficiency where there are limitations on the available current to operate a circuit or changeable environments that require circuits to be robust against single-event upsets and varying temperatures. All of these potential design scenarios may be instances where design implementation explorations reveals that an asynchronous design style maybe the best performing in that context. Providing the firm with the capability to assess the capability of asynchronous design styles with these design requirements is key competitive advantage. Although this a conversion flow, benefits are extracted further down the FPGA design flow. The novel aspect of this work is how the asynchronous circuits are implemented on the FPGA. Changing the behavioural description language to a dedicated asynchronous language or a modelling language could result would allow optimisation of different source languages to FPGA architectures. This low-level access to the FPGA fabric allows operations to be constructed with finer granularity, in the same manner that assembler code can be inserted into C-programs. This approach provides another potential competitive advantage, allowing circuit descriptions to make into products quicker (reducing time-to-market) or performing better than they normal would.

The commercial relevance of this work is very strong due to the industrial motivators for performing this research. This direction adds significant tangible value to the research described in this thesis as well as its novel contributions to knowledge.

1.4 External Events

1.4.1 Academic and Industrial Events

Throughout the period of research there were a number of events that provided welcome discussion and feedback on the work being undertaken. No attempt to contribute to these events was made, however the benefit came in networking opportunities and the ability to discuss and explain the directions of research. Since the (asynchronous) topic of research is significantly outwith the comfort zone of Thales Optronics Ltd, the opportunities provided by these events were significant.

1.4.1.1 IEEE Symposium on Asynchronous Circuits and Systems

The symposium on asynchronous circuits and systems was attended between the 7th and 10th of April 2008. Although not presenting any papers, this symposium was of most value in being able to access, what is regarded as the latest research in asynchronous circuits. The symposium also acted as a networking opportunity to gain a number of peer contacts to question and query the direction of research. There were a few presentations that had commonalities with this line of research and provided an array of markers from which the research can be clearly defined. The most informative parts of this conference were the range of tools being used to design and verify asynchronous circuits on ASICs as well as the minimisation techniques used to reduce the number of asynchronous components that would explicitly limit the usage of asynchronous structures on FPGA devices.

1.4.1.2 UK Asynchronous Forum

As part of maintaining connections made at the symposium for asynchronous circuits and systems, the 20th UK asynchronous forum was attended on the 1st of September 2008 to firstly discuss research ideas and secondly evaluate new ideas and concepts generated by the UK asynchronous research community. There are very few events that cater for the asynchronous design space and so this was a welcomed opportunity to discuss the research progress with an audience that are very familiar with region of interest this research occupies.

1.4.1.3 Thales Group les Journée du Computing

The PhD computing day is an annual event held by the Computing Network of Excellence as part of the Software and Critical Information systems group in Palaiseu, France. This event was attended on the 16th of November 2011 and provided a concluding presentation of the work presented in this thesis to the sponsoring company. The goal of this event is to gather all the Thales Group PhD students working on computing research to share their research topics and recent progress, as well as their experience. This event was reasonably similar to the Asynchronous Forum, but with a 30

minute presentation and subsequent poster session, the finished work generated a significant amount of discussion and interest from various attendees.

1.4.2 Conference Publications

Two contributions were given at major academic conferences. One involved a poster presentation and the other a full length 20 minute presentation slot, both contributions are accompanied by papers published in the conference proceedings for each event

1.4.2.1 IEEE International System on a Chip Conference

Due to an accepted paper submission, *'Evaluation of contrast limited adaptive histogram equalization (CLAHE) enhancement on a FPGA'* on contrast limited adaptive histogram equalisation, the international system on a chip conference was attended between the 15th and 20th of September 2008. The conference took place in Newport California and contained five days worth of paper and poster presentations, tutorials and discussions on many areas relevant to system on chip design- including image and video processing as well as a small asynchronous design tutorial. The paper was presented as a poster and received significant interest. There were a number of professional and academic organisations represented at this conference, presenting and discussing the latest topics of research and development. The networking and critical discussions at this conference were of great value to the direction of this research at that time.

1.4.2.2 Euromicro Conference on Digital System Design

Another paper, *'Optimising Self-Timed FPGA Circuits'* was accepted to the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools in Lille, France, on the 1st to 3rd of September, 2010. This paper was specifically aimed at allowing the work around the design flow of AACIF circuits to be peer reviewed. The conference includes a number of topics within its scope, including system and circuit synthesis which aligns with the intentions of the paper. It focuses on advanced circuit and system design, design automation concepts, paradigms, methods and tools, as well as modern implementation technologies from full custom in nanometre technology nodes to FPGA and multicore infrastructures. As with the previous conference the event provided a good opportunity and discussion forum to seek feedback, different interpretations and impressions on the direction of research- all positive and beneficial.

1.5 Taught Modules and Training

As part of the EngD requirements, a total of 180 credits at Master's level are required. These are made up of 120 technical credits and 60 business and management credits. The technical modules

were sourced from the MSc in System Level Integration offered by the Institute for System Level Integration (ISLI) and Continuing Education in Electronics Systems Integration (CEESI). The business and management modules were sourced from the MBA programme of the University of Strathclyde Graduate School of Business (USGSB).

1.5.1 Technical Modules

Out of the 120 MSc level technical credits required for the EngD course, 4 of them were taken in the first term of the MSc course from the ISLI, and 3 were taken in the second semester. The remaining came from the 3rd semester and external courses.. Table 1.1 summarises the modules taken and their corresponding credit weightings. The most significant courses were:

Subject	Credits
Introduction to Hardware Design Automation	8
IP Block Authoring	15
Microcontrollers and Microprocessors	15
VLSI Design	15
Embedded Software 1	15
Real Lift System Level Integration	8
IP Block Integration	15
System Partitioning	15

Table 1.1: Technical Modules

IP Block Authoring - provided a solid base in various design methodologies and discussed the attributes of the different classifications of I.P blocks. The main emphasis was on I.P. block re-use and power/speed/area evaluations.

Microcontrollers and Microprocessors - discussed design knowledge gained on instruction set architecture design. This also covered parallel and pipelined processor design with investigation into efficient cache architectures and brief coverage of VLIW and SIMD processor architectures.

VLSI Design - provided additional experience in fundamental CMOS circuit design, following the design flow from data path modelling and implementation down to the transistor layout design including automatic place and route techniques.

Embedded Software 1 - developed skills in embedded software construction, with extensive focus on DSP orientated data manipulation. This was tested with development of a responsive real time operating system embedded application using RTEMS.

IP Block Integration - provided hands on experience with hardware verification languages, Specman. The module covered simulation technologies with rapid prototyping, formal and timing verification tools. There was significant emphasis on design for test ability and test bench configuration.

System Partitioning - Increased awareness of modelling languages to aid system partitioning with a depth SpecC project. The module also covered models for computation and communication. The

concise introduction to UML and the benefits from its features as a system specification language were highlighted from an object-orientated design perspective.

There was only one module that was taken outwith the MSc curriculum provided by the ISLI, *'Self-timed logic'* provided by the University of Manchester on behalf of CEESI. This was a 16 week long distance learning course which will began in October 2008 and was worth 15 credits. This module was determined to be particularly useful in the context of the research brief provided by Thales Optronics Ltd. This module was taken by distance learning and provided a clear, concise insight and introduction into the asynchronous design space. The course covered a crucial number of topics that were of significant use in the latter part of the research, providing a preliminary understanding of asynchronous data and control protocols as well as asynchronous synthesis tools.

1.5.2 Business Modules

The 60 business credits were obtained by completion of the classes listed in Table 1.2. Distance learning and part-time study was the most convenient and flexible method to undertake these classes whilst balancing the requirements of the technical research. The business modules provided an opportunity to look at the EngD from a theoretical commercial perspective. Although the EngD is minor in comparison to the activities within Thales Optronics Ltd as a business, considering the EngD in a business environment rather than a research perspective provided an insight into the support measures required.

The business modules also provided an understanding of the key performance indicators chosen by the business, the operations management that drive the internal procedures and the financial responsibilities that balance every decision throughout the company. As with all textbook interpretations, the real-life application of theoretical methodologies are always subject to commercial compromises. The subjects chosen (shown in Table 1.2) were very beneficial in allowing the technical requirements of the EngD to remain fully supported by the company whilst being unaffected by the commercial environment. Subjects such as Finance & Financial Management and Financial & Management Ac-

Subject	Credits
The Learning Manager	3
Operations Management	12
Marketing Management	12
Financial and Management Accounting	9
Finance and Financial Management	12
Making Decisions	6
Data Management	6

Table 1.2: Business Modules

counting gave an insight into the costing practises, balance sheets, capital expenditure, and other financial topics which affect all engineering projects. These aspects are normally encompassed by

project managers and purchasing departments but are rarely tackled by engineers on a wider scale than development work.

Operations Management showed businesses from a strategic perspective, outlining the operations function and how operations management acts to implement strategy by process design and improvement measures. This mechanism outlines how businesses judge their performance and quality of output. In Thales Optronics, primarily being a technology company, there is a strong operations influence on all aspects of the business. This class was a key insight into how the company transforms in adapting to market changes and optimising internal processes.

Marketing Management showed the importance of a customer-facing company. Technology-based companies tend to focus on producing advanced products because the technology is available and assume that the customer will follow. Understanding exactly what the customer needs and creating products to match those needs (as well as bettering the competition) is fundamental to a successful business. This class was particularly enlightening when considering the strategic marketing required by Thales Optronics Ltd to anticipate customer requirements 5 to 10 years in advance and predict how to meet those needs in that time frame. A substantial report was submitted for this class based upon the marketing activities for particular segment of Thales Optronics. This provided the first hand motivation to justify the direction of the research and development activities.

Making Decisions demonstrated the benefits of Multi-Criteria Decision Analysis (MCDA). A unique, methodical approach to decision making within a complex environment such as Thales Optronics Ltd. Decisions matter when an issue is sufficiently complex and detailed that there is conflict between criteria and the importance of criteria in the decision. In this situation a gut feel decision is not sufficient. The principle aim is to help decision makers identify preferred courses of action. This is achieved through the structuring of values and judgements from stakeholders who have influence/interest in the outcome of the decision. In the context of Thales Optronics, a parallel investigation was conducted, demonstrating MCDA on a decision required for the choice of software packages for engineering issue management. The conclusion of this investigation was another angle on the solution possibilities, a traceability document that justifies course of action from top level requirements and a support to the resultant action plan.

1.5.3 Handshake Solutions TiDE Training

Following advice from an academic supervisor under the assumption that the tool of choice for implementing asynchronous structures in the near future would be supplied by Handshake Solutions, this 3-day course was attended between the 12th and 14th of November 2007. This course provided firstly, a very clear overview of the asynchronous design flow, TiDE (Timeless Design Environment) and an explanation of the requirements for a design language (Haste) to accurately represent asynchronous

circuits. Secondly, this course provided the first glimpse into the commercial design flow for creating asynchronous structures on an FPGA However it is important to note, that although this was the only commercially recognised asynchronous design flow it is not the only one in existence. There are a number of academic based asynchronous design flows in existence that automate different areas of the asynchronous design space.

Part II

Technical Reports

Technical Report 1:

Region-Based Contrast Enhancement on a FPGA Platform

Author:	Phillip David Ferguson
Academic Supervisors:	Prof Tughrul Arslan, Univ. of Edinburgh
	Dr Ahmet Erdogan, Univ. Of Edinburgh
Industrial Supervisor:	Andrew Parmley, Thales Optronics Ltd

2.1 Aims and Introduction

The body of work contained in this Technical Report documents the initial period of research at Thales Optronics Ltd. The primary motivation for this research is based on an industrial need to improve the performance of image algorithms used in their products.

A primary element of these products is the contrast enhancement mechanism used to improve the clarity and definition of images from thermal imaging sensors. Previous contrast enhancement techniques have been based around computations that are applied to an entire image. This body of work evaluates the relative merits of using a region based contrast enhancement algorithm on a platform that is common to most imaging products from Thales Optronics Ltd - the field programmable gate array (FPGA). Region based contrast enhancement algorithms show significant image quality benefits over image wide techniques, and so it was a logical conclusion to assess the feasibility of suitable algorithms.

This work discusses the implementation considerations and performance limitations of CLAHE. In doing so the suitability of an FPGA for this type of contrast enhancement is analysed. At the time of writing there were no known FPGA implementations of region based contrast enhancement that evaluated the performance attributes of this type of contrast enhancement algorithm on an FPGA. This novel approach to implementing CLAHE provides the industrial benefit of trialling CLAHE on an FPGA device allowing the firm to surmise the hardware requirements for region based contrast enhancement algorithms.

2.2 Contrast Enhancement Background

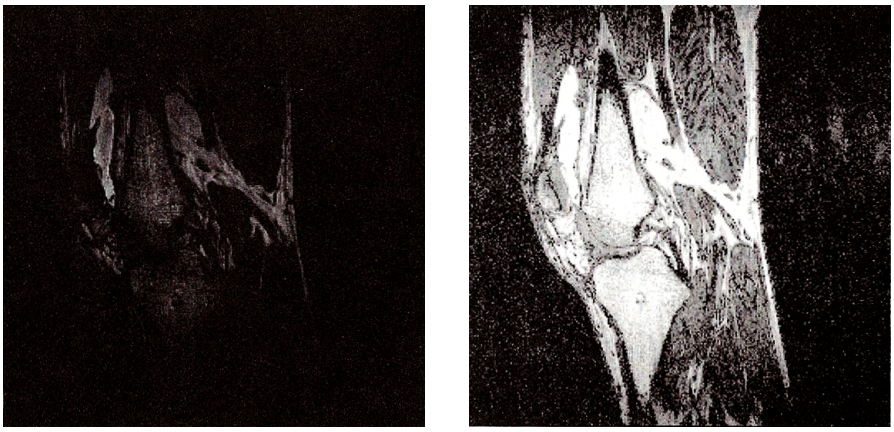
Previous studies [12] have discussed simpler contrast enhancement algorithms within the same industrial context, however due to the superior image enhancement ability of CLAHE, these conclusions need to be reevaluated. The following sections document the considerations and implementations in exploring an FPGA implementation of CLAHE.

2.2.1 Algorithm Developments

Contrast enhancement is familiar to most individuals through picture settings on their television or monitor. Changing this setting allows the contrast of an image to be adjusted to obtain better clarity and definition. This is achieved by scaling or offsetting the intensity values of the pixels so that the full range of the intensity values are used within the image. E.g. if each pixel has an 8-bit range from white to black of 255-0, an image may contain pixels values from 220-10. In this case, visually white may not be pure white and black my not be pure black. Adjusting the contrast setting scales the intensity values so that an image with a white equal to 220 now becomes an image with white equal

to 255. This scaling is a linear ramp with a particular gradient. The contrast setting on a monitor alters the gradient of this ramp that is applied to every pixel in the image.

Histogram equalisation (HE) [1, 4] was initially developed to counteract the imbalance in pixel intensity created from computed tomography and magnetic resonance scanners to cathode ray tube display units. Previously a linear ramp had been used as a re-mapping function in a 1-to-1 pixel transform to improve the contrast of a given image. Histogram equalisation uses a non-linear ramp to provide an intensity level mapping that increases the output intensity level range dependent on the frequency distribution (a histogram) of the intensity levels within an image. This is accomplished by a cumulative distribution function to provide the re-mapping function which transforms the intensity values in an image such that the histogram intensity for a transformed image is constant. However the resultant image in some circumstances is worse than the equivalent image that had been enhanced using the linear ramp windowing method. This is due to large concentrations of background noise, creating peaks in the original histogram that once levelled out (see Figure 2.1) enhanced the visibility of background noise to the same levels as the image detail, thus losing minor local contrast changes.



(a) Human knee MR scan (b) Histogram Equalisation Result

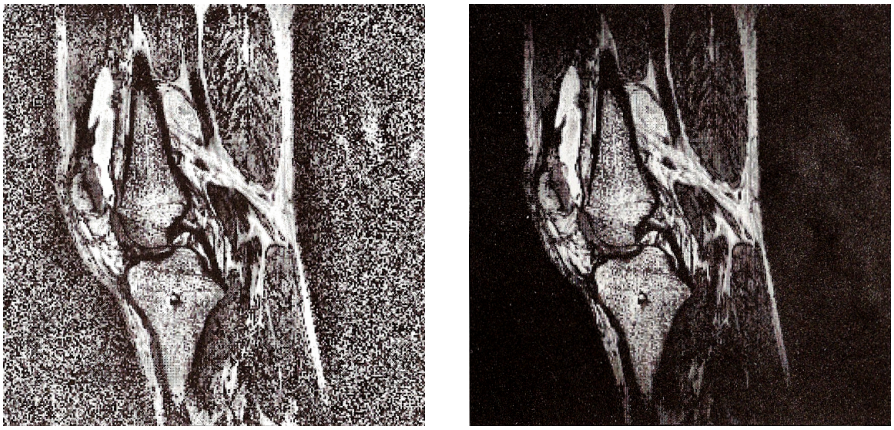
Figure 2.1: Histogram Equalisation Enhancements [15]

As HE uses the entire image to produce a re-mapping function, the next logical step was to minimise the size of the region with which the contrast enhancement was performed. Independent studies [9, 5, 3] of different variants began to explore the subtleties of balancing independent contrast regions. Creating an individual remapping function for a reduced region size creates an image where each region has been adaptively equalised. Adaptive histogram equalisation (AHE) improves overall image contrast for two reasons:

- Large peaks in background noise can be minimised to local regions, thus limiting the image wide noise enhancing effects.
- The human visual system adapts to the local context of images to evaluate the contents, as previously multiple linear ramps were applied to improve regions of interest in an image.

Splitting the image into contextual regions and then performing the remapping will inherently introduce visible contextual boundaries across the image. Thus a bilinear interpolation (discussed further in Section 2.3.3) is performed to remove these boundaries.

However now that important contextual regions can be enhanced with the benefits of histogram equalisation, the background regions of images, which are predominantly dark, suffer (shown in Figure 2.2) a dramatic enhancement in their noise content. To address this issue, spreading the histogram peaks over the entire contextual region removes any majority intensity and so reducing the inter-contextual region contrast. Many methods [2, 13] have been devised to minimise this effect through a series of region weightings or modifications to the cumulative function.



(a) Adaptive HE (b) Contrast limited AHE

Figure 2.2: Contrast Limiting Effects on AHE [15]

Contrast limited adaptive histogram equalisation (CLAHE) [9, 10] solves this inherent problem by limiting the contrast enhancement in homogeneous areas where grey levels are mostly constant. To accomplish this, CLAHE limits the maximum value a bin can hold in the image histogram. This has the cumulative effect of reducing the gradient of the re-mapping function. CLAHE analyses the image histogram and clips the offending bins to a maximum value. The excess of pixels creamed off the top of the histogram must be redistributed to maintain the total number of pixels in the histogram. The resulting transfer function restricts the output range of high concentration pixel intensities in the original histogram to their intended output range in the transformed image.

2.2.2 Platform Developments

Currently implementations of CLAHE have mostly remained in the software domain. Pizer [9] initially performed operations on a VAX 11/780 using C or assembler as the implementation tool. This usually took around 2 minutes to complete with image sizes up to 512 x 512 pixels. The hardware of choice then moved to a dedicated multiprocessor machine, MAHEM [10], composed of 64 pixel processors that was able to reduce the run time of CLAHE to 4 seconds. Gauch [2] in 1992, used floating point

arithmetic to demonstrate various strands of HE including AHE. Using a 7 MIPS workstation, most variants took several minutes to compute for image sizes up to 512x512 pixels. The next development came in a general purpose C based implementation from Zuderveld[1] [14] which due to the growth of general purpose x86 processors meant that CLAHE took less than a second on a HP 9000/720 workstation for 8-bit 512x512 image using 8x8 contextual regions. Another dedicated platform [6] was developed in 1998 with a move to a Xilinx XC4010 FPGA. This silicon level implementation gave a real-time performance jump, however this was only performing HE across the entire image. With an image size of 256x256 pixels and a clock rate of 50MHz, an entire image was transformed in approximately 1 millisecond. In 2000 Stark [13], continued analysis and development on general purpose x86 cpu's which due to their pace of development allowed a great range of image analysis to be performed on a wider variety of AHE variations and there computational implications. In 2002 Matlab added a CLAHE function to its image processing tool box. Based on the Zuiderveld implementation it brought the additional option of redistribution according to a uniform, exponential or Rayleigh distribution. Since then the only mention of CLAHE was a proposal by Reza [11], however this had no results or implementation to compare against.

2.3 Contrast Limited Adaptive Histogram Equalisation

This section will document the various stages and operations required to perform the contrast improvement achieved by CLAHE algorithm in greater detail than touched upon previously.

2.3.1 Histogram Creation

Firstly a decision must be taken to establish the contextual region size relative to the image size. The contextual regions divide the source image space up into equal sized tiles as shown in Figure 2.3. These tiles define the local regions of contrast that will be used to construct the contents of the re-mapping histograms. There is no general purpose ratio, and so a compromise between contrast detail and computational requirements needs to be taken. Larger image sizes will lose contrast detail with large contextual regions at the expense of computational time. Zuiderveld [14] does not limit the image or contextual region size, but does limit the ratio to be 16. Pizer [10] uses an 8x8 kernel for a 512 x 512 image size, at the expense of hardware. Increases in image size disguise larger contextual regions but lose contrast detail, thus if detail is be retained an 8x8 contextual region is likely the lowest practical limit.

[1] Karl Zuiderveld also co-wrote the initial AHE paper by Stephen M. Pizer in 1987

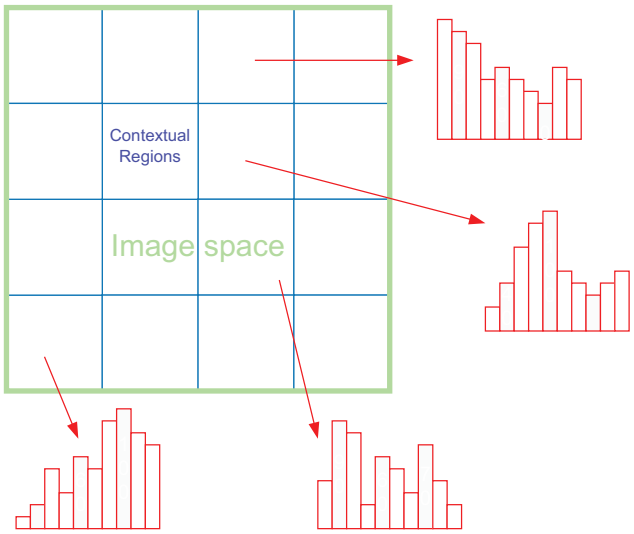


Figure 2.3: Contextual Region Histograms

A histogram of each contextual region is then constructed to identify the contrast deficiencies in that region of the image. The pixel depth must also be taken into account when creating a histogram so that there is sufficient contents to redistribute. The number of bins present in the histogram is a factor of the number of intensity levels (the pixel depth) that can be represented in the image. Reducing the number of bins reduces the dynamic range of the output image, essentially limiting the change in intensity to improve contrast and reduce histogram storage requirements. This also relates to the size of the contextual region, where a lesser number of pixels will not produce a reasonable representative distribution relative to the pixel size and thus clipping the histogram is unlikely to have an effect. For example, if an 8 x 8 contextual region was used then 64 pixels (of 8-bit depth) must be divided up across a histogram that could have between 256 and 2 bins. In the case of 32 bins, the average pixel per bin is only 2, providing very little to redistribute if there isn't a dominant intensity. Thus contextual regions should realistically contain an integer multiple of the pixel depth.

2.3.2 Clipping and Redistribution

The clip limit (or contrast factor) is defined as an integer multiple of the average histogram contents. Based on the peaks in the histogram, the clip limit performs a noise control mechanism between zero contrast enhancement (a low re-mapping slope hence a low factor) and AHE which removes the redistribution operation altogether. If there is a large peak in the histogram, forming the re-mapping function (i.e. accumulating the histogram bins to form a non-linear ramp) will result in a steep gradient change. This steep gradient creates a large change in pixel value, resulting in the noise shown in Figure 2.2a. The clip limit indicates how much excess is trimmed from the histogram, limiting histogram peaks and consequently background noise.

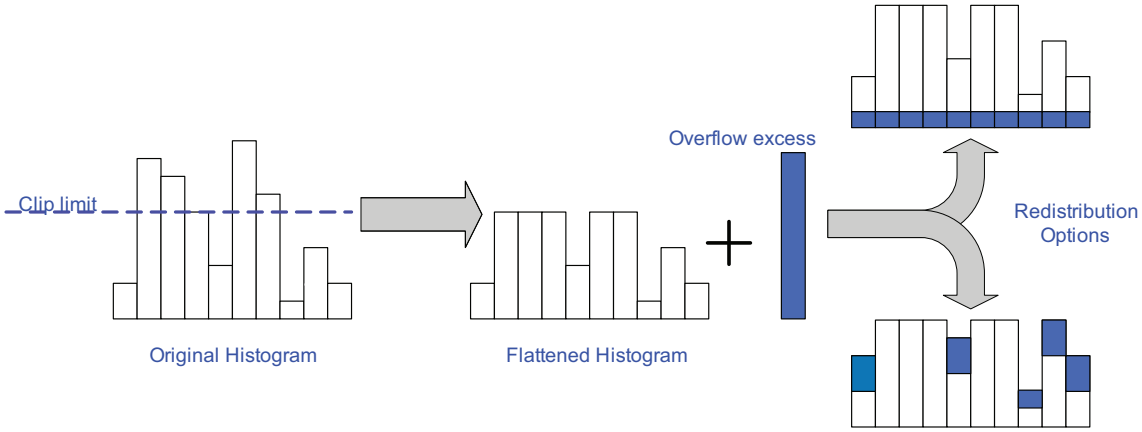


Figure 2.4: Redistribution Options

Clipping a histogram according to the clip limit produces a flattened histogram and an overflow of excess pixels as shown in Figure 2.4. This overflow is redistributed across bins in the flattened histogram that are less than the clip limit. There are various computationally dependent methods (examples shown in Figure 2.4) to redistribute the overflow, from creating an average that is added to every bin to identifying the bins with low contents and adding proportions of the overflow dependent on their contents. The Matlab implementation mentioned previously provides many different methods to redistribute the overflow excess for each histogram including a uniform, Rayleigh, or Exponential distribution. Redistribution is not performed if there is no overflow from clipping the histogram.

2.3.3 Forming the re-mapping function and smoothing artifacts

Once the histogram from each contextual region has been redistributed, it must be transformed with a cumulative distribution function and then scaled to suit the pixel depth of the image. The re-mapping function effectively redistributes or reduces the dynamic range of the source image. In many cases this scales or stretches the image so that instensity differences between pixels are more apparent.

Each contextual region will have its own local contrast specific re-mapping function. If we were to apply these re-mapping functions, the boundaries between contextual regions would become visible introducing artifacts to the resultant image. Thus a bilinear interpolation is used to smooth the boundary artifacts from the local contrast enhancements.

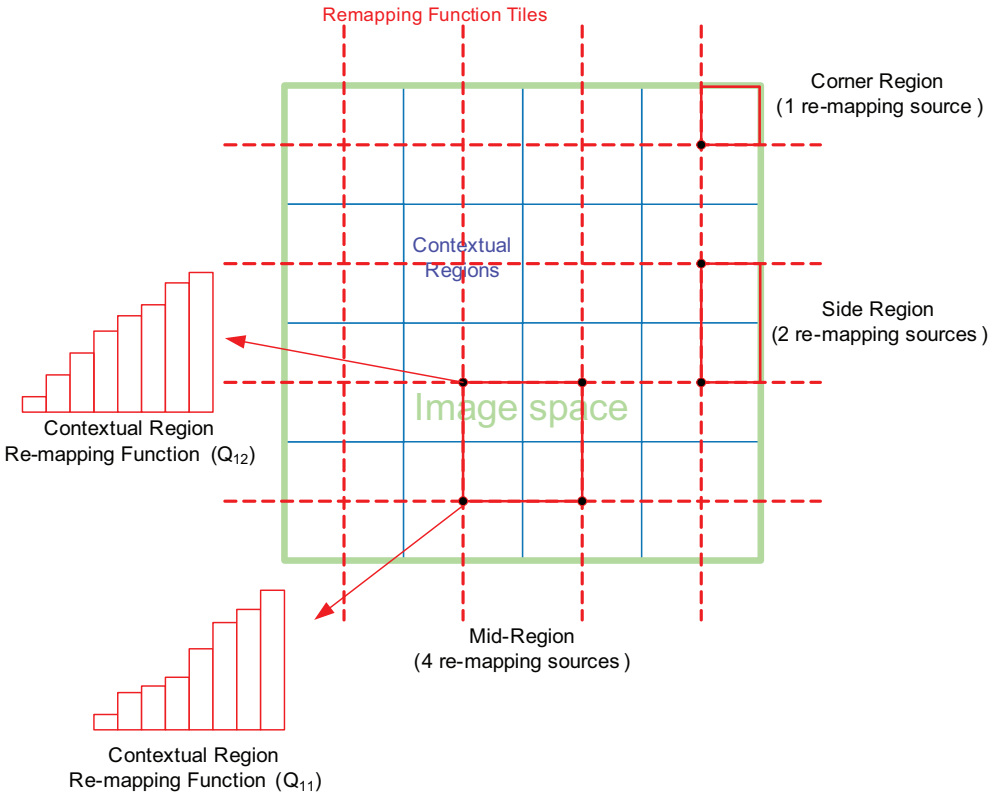


Figure 2.5: Bilinear Regions over Image

For each pixel in the image, the transformed intensity value is a weighted sum of the closest four contextual region re-mapping functions as shown in Figure 2.5. Depending on the position of the current pixel relative to the boundaries of these remaping functions their influence will be increased or decreased. Pizer [9] suggested that the four re-mapping functions could come from interpolation regions that spanned multiple contextual regions offset by half the width of a contextual region. As larger contextual regions will tend to lose image sharpness and detail, the simplest and most accurate solution sets the contextual and interpolations regions to be the same size. Additional considerations need to be made for boundary contextual regions that surround the outer edges of the image. In these cases there may only be one or two influential contextual regions.

For the mid-regions in Figure 2.5 the resultant intensity value requires the re-mapping functions of the four surrounding contextual regions$(Q_{11}, Q_{12}, Q_{21}, Q_{22})$. This is described by

$$\begin{aligned} P_{new} = & \frac{(y-y_1)}{(y_2-y_1)}\left(\frac{(x_2-x)}{(x_2-x_1)}f(Q_{12}(P_{old}) + \frac{x-x_1}{x_2-x_1}f(Q_{22}(P_{old})\right) \\ & + \frac{(y_2-y)}{(y_2-y_1)}\left(\frac{(x_2-x)}{(x_2-x_1)}f(Q_{11}(P_{old}) + \frac{(x-x_1)}{(x_2-x_1)}f(Q_{21}(P_{old}))\right) \end{aligned} \tag{2.1}$$

Here the intensity value of the current pixel P_{old} is passed through the 4 contributing re-mapping functions. Depending on the position of the pixel (shown in Figure 2.6) relative to the contextual

region that each re-mapping function describes, they are weighted according to their contrast changed that would impact the current pixel.

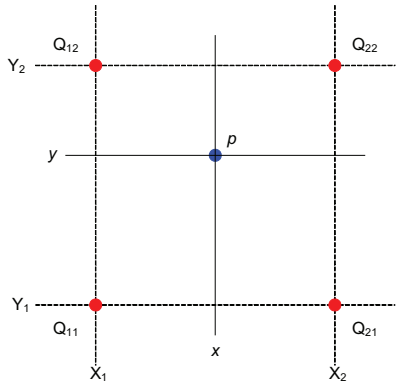


Figure 2.6: Bilinear Interpolation

The side regions in Figure 2.5 only overlap two contextual regions, and so Equation 2.1 reduces to 2.2 where the vertical terms of the equation and the corresponding re-mapping functions,

$$P_{new} = \frac{(y - y_1)}{(y_2 - y_1)} f(Q_{22}(P_{old})) + \frac{(y_2 - y)}{(y_2 - y_1)} f(Q_{21}(P_{old})). \tag{2.2}$$

The corner regions in Figure 2.5 have no external influence from neighbouring re-mapping function, and so equation 2.2 reduces to only one term that performs the re-mapping transform.

$$P_{new} = f(Q_{21}(P_{old})) \tag{2.3}$$

2.4 CLAHE Implementation

The aim of this implementation is to evaluate the implications of CLAHE in an image processing context on an FPGA. This means assessing the necessary hardware functionality, required resources, power and timing implications for such an algorithm within a single device. The platform of choice is the Avnet DS-BD-V4FX12LC development board [19]. During this period of research, Thales Optronics Ltd was using these boards as a basis for evaluating their capability in future products. It therefore made the board the ideal choice to implement CLAHE on. This development board houses a Xilinx Virtex 4 device [22], and consequently the implementation parameters were based on its ability to implement this. A brief discussion of the design tools used in conjunction with this board is presented in Appendix A.1.

Instead of specifying a predefined image size, a contextual region size was used as a basis for CLAHE operations. This allows flexibility in determining common operations and scaling them according to the resources available. A contextual region of 64 pixels, 8-bits wide falls in line with previous implementations where it had been judged that sufficient detail is retained through this contextual region size. This consequently means that image size must be a multiple of 8 pixels in each direction. For testability and proof-of-concept reasons, the resultant histogram was limited to 8 bins. This was the optimal such in accessing local memory stores within the device that would be used to construct histograms. To provide the redistribution operation with sufficient work, a clip limit close to the average bin contents (8) was set. With these two parameters the implementation discussion can move onto the tool flow, subsequent algorithm parameters will be discussed in the context of the following subsections which are most appropriate.

2.4.1 Implementation Tool Flow

The design steps for the implementation of CLAHE are shown in Figure 2.7. There is very little difference from the standard synchronous FPGA design flow that caters for time, power and speed driven performance goals. Assessing the speed and temporal aspects of different parts of the design are mostly encapsulated within each step the design flow from design entry to device programming. Simulation has been used throughout each stage verify the functionality against the HDL design. The verification flow was based around Mathworks in-built MATLAB function *'adapthisteq'* (CLAHE) [7] as the golden reference. This is explored in greater detail in subsequent sections. The use of the on-chip logic analyser was the last stage in verification and debugging, highlighting design (mostly timing) flaws that had escaped detection in earlier simulations. Power analysis, although not captured in Figure 2.7 requires a back annotated timing model and the corresponding standard delay format files (generated from the implementation stage) for the simulator to produce the necessary switching characteristics (value change dump files) used in the power estimator.

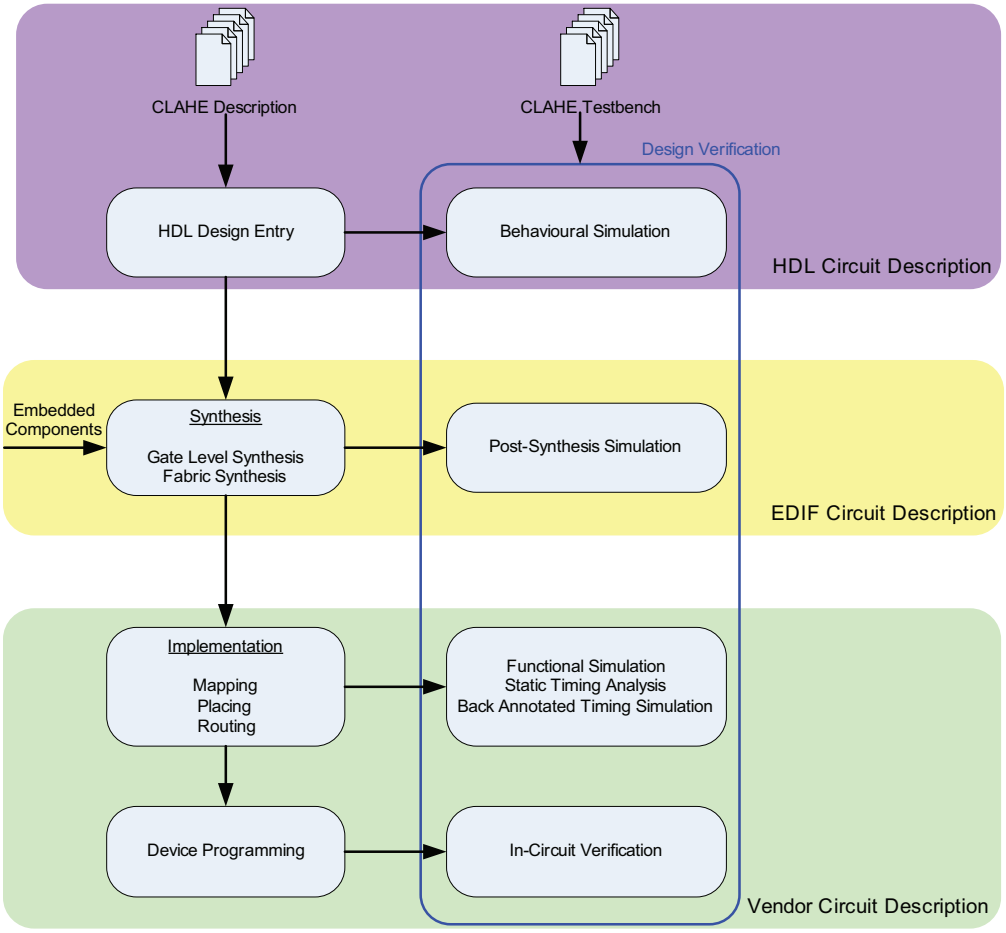


Figure 2.7: Synchronous Design flow

2.4.2 Top Level Overview

A block diagram of the CLAHE processing stages is shown in Figure 2.8. Assuming the source image is stored contiguously in an external frame store, a pixel feeder is required to firstly retrieve the correct pixels for the contextual region required and secondly distribute them to the histogram pipeline requesting that contextual region. Each histogram pipeline will clip and redistribute a histogram and then deposit the accumulated histogram (the re-mapping function) in the histogram RAM (HRAM). Since each histogram pipeline is running in parallel and there is no guarantee how much processing is required on each contextual region, a histogram sequencer is required to sequence write operations into HRAM. Arbitrating this shared resource means that feedback signals are sent up the processing stream to control the flow of contextual regions into each histogram pipeline. The pixel feeder must also arrange the original image to be sent to the weight generator in a rasterised format to coincide with the histogram scheduling. For each pixel in the source image, the weight generator creates the correct pixel weights required by each pixel in the bilinear interpolation process. Along with the original image addresses are generated so that the bilinear sequencer can arrange, the input pixel, the bilinear weights, and the results of the histogram remapping functions from the HRAM in the correct

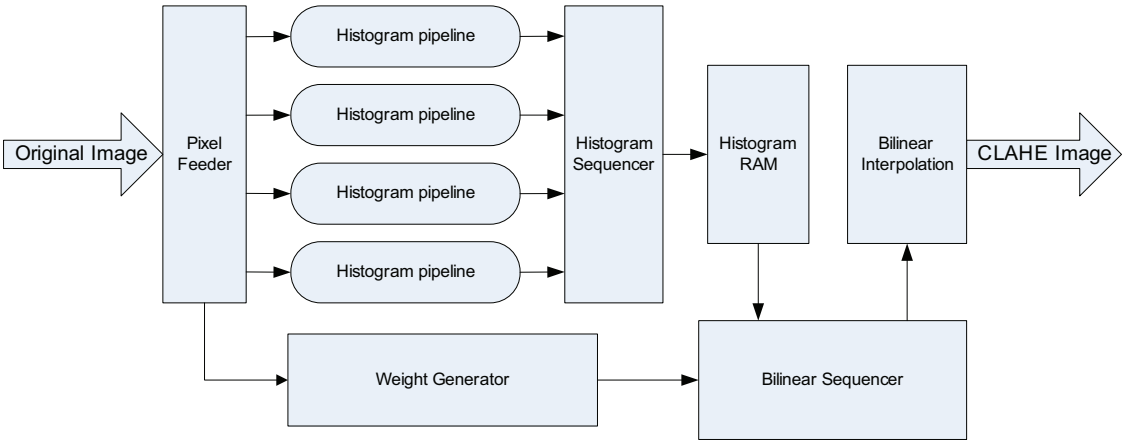


Figure 2.8: CLAHE Processing Stages

order so that the bilinear interpolation is constantly active.

A finite state machine (not shown in Figure 2.8 for simplicity)coordinates the every top level block to retrieve contextual regions from memory, produce re-mapping functions through the histogram pipelines and sequence the bilinear interpolation correctly. To correctly feed the bilinear interpolation, at least two rows of re-mapping functions must be stored to maintain constant operation. For each pixel, at most, four re-mapping functions are required. Pixels from each new interpolation row require re-mapping functions from the previous row. This makes memory requirements crucial in maintaining sufficient precision for bilinear interpolation.

The following sections will decompose each top level block and provide fuirther detailed discussion on their operation and the design choices for implementation.

2.4.3 Memory Management

The key decision decision that governs the parameters of the algorithm implementation is memory. Modern FPGA devices are available with a wide range of on-chip memory, therefore it is a scalable resource that can be chosen to suit the application. The device present on the development platform, a Xilinx Virtex 4 FX12 [22] has 36 block RAM (BRAM) modules totalling 648Kb. Although most modern embedded systems assume an external image store, for proof-of-concept testing the image store was kept purposefully limited to keep simulation and testing times down. Integrating and testing an embedded memory controller to interface to external memory was expected to consume over half of the development time but would have gained little in terms of concluding results.

The on-chip memory resources were consequently split across an image store, histogram RAM, local histogram pipeline memory and various scaling look-up tables which utilised the same primitive BRAM memory resource on the FPGA. Each BRAM can store 18Kbits of data, with sufficient flexibility to change its breadth and depth within that limit.There is multiple configuration options that allow the primitive memory blocks to act as dual port RAMs with separate clocking schemes or pre-loaded

ROM look-up tables.

At most bilinear interpolation requires four re-mapping functions to form one output pixel. To ensure that there are no stalls in creating new pixel values, two rows of remapping functions will need to be stored. The storage required is heavily dependant on the size of the image, the size of the contextual region, and the size of the histograms/re-mapping functions. Starting with an assumption of an 8x8 contextual region, each histogram pipeline will require one BRAM to create a histogram and another BROM for scaling the image into bins. If we use an image size of 256x256, 32 BRAMs would be required, include the memory requirements of the histogram pipeline and we are already over the available BRAMs on the device. This means that an image size of 128x128 was chosen, occupying 8 BRAMs, allowing sufficient margin for additional BRAM utilisation. The total amount occupied would be 18.

2.4.4 Pixel feeder

For simplicity, early testing models of the histogram pipeline assumed that the contextual input pixels would be supplied from a contiguous memory. A translation mechanism is required to supply the histogram pipelines with contiguous data per contextual region and rasterised image data for the interpolation from the same image store. The image space is decomposed into four dual port memory blocks. One port allows image data to be retrieved in a rasterised format for interpolation. In this operation, a combined address bus shares all the low order address for each block RAM and the high order address bits are used as enables to select each memory block. On the other port there are four separate address generators (loadable counters) that sequence themselves to isolate contextual regions in the order shown in Figure 2.9. This order is specific to ensure the correct contextual regions are loaded to the histogram pipelines, allowing the re-mapping functions to align to the correct interpolation operations.

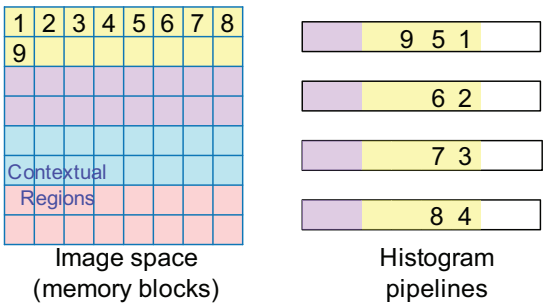


Figure 2.9: Image Memory Configuration

Since there is no definitive time for redistribution, the pixel feeder requires feedback from the histogram sequencer to continue loading contextual regions in the order shown in Figure 2.9.

2.4.5 Weight Generator

The weight generator produces a number of parameters sourced from the pixel feeder. These include the interpolation weights, the interpolation region flags and the image contextual region flags. All output flags and parameters associated with each pixel are derived from one fourteen bit image address counter within the pixel feeder. Firstly the rasterised pixel output is controlled by the sequencing of address counters to each memory module. The contextual regions are identified through the sixth to third bit (for the rows) and thirteenth to tenth bit (for the columns) of the image address counter. The pixel weights that are passed out to the interpolation calculation are sourced from the image address counter, in the case of:

μ-weight (vertical direction) : divide the image address by 128(i.e. counting each row) to produce the quotient, add an offset of 4 to this due to the interpolation regions. Then use the remainder of the division by 8 (the vertical contextual region dimension) to loop the weight between zero and seven. In terms of the equations shown in Section 2.3.3:

$$\mu = \frac{(y - y_1)}{(y_2 - y_1)}$$

η-weight (horizontal direction) : use the remainder of the division of the image address by 128, add an offset of 4 to this due to the interpolation region. Then again use the remainder of the division by 8 (the horizontal contextual region dimension) to loop the weight between zero and seven. In terms of the equations shown in Section 2.3.3:

$$\eta = \frac{(x_2 - x)}{(x_2 - x_1)}$$

The interpolation regions are identified in part again by the image address and also the pixel weights. The vertical interpolation region is identified by dividing the image address by 512 and rounding the output down. In doing this the half size interpolation regions can be identified at the image sides. The horizontal interpolation regions are identified by testing the η-weight for the appropriate interpolation region dimensions and counting them accordingly. With the pixel data and address being delayed by a similar amount the weight generator synchronises each pixel with the correct interpolation weight and region flag.

2.4.6 Bilinear sequencer

The bilinear sequencer acts to align the data from the weight generator with the correct re-mapping function. Once a pixel has its various parameters synchronised the next stage is to probe the histograms held in the histogram RAM ready for the interpolation of that pixel value. This means

generating four RAM addresses from a single pixel value. This would indicate that the generation of the ram address must run four times faster, in this case the pixel feeder runs four times slower than the clock signal. Firstly the pixel value must be passed through the same look-up table that is used to create histograms(see Section 2.4.7.1) in order to identify the correct bin for each address and thus the value to be passed to the interpolation operation. Since there will be 32 histograms stored in waiting for the interpolation, the address calculation must take into account where the four components of the interpolation are logically sourced from in memory. From the interpolation region identifiers(created in the weight generator) the valid addresses for a particular region are identified through a series of range tests shown in Algorithm 2.1 in the form of a psuedo VHDL process. For example: if the pixel passed through is in a corner region there is only one influential weight, all four can be assigned the same address. However if there is a centre, side, top or bottom region the address must be offset to find the correct memory location for that weight. The functional block which identifies the correct memory address operates in a clock wise manner from the lower right weight, in most cases specifying four different locations for each weight that are at most 129 memory locations apart.

Algorithm 2.1 Interpolation Range Testing

```
Interp_Block: process(clk)

    begin
        if (CLK'event and CLK= CLKPOL) then
            if (reset ='0')then
               if (row = 0) then
                 if (col=  0) then
                   -- we are in the top left corner
                 elsif(col = 15) then
                   -- we are in the top right corner
                 else
                   -- then we are in the top row that is not a corner
                 end if;
               end if;
               -- end of top row
               --============================================================
               if (row = 15 ) then
                 if (col =  0) then
                   -- we are in the bottom left corne

                 elsif (col = 15) then
                   -- we are in the bottom right corner

                 else
                   -- then we are in the bottom row that is not a corne
                 end if;
               end if;
               -- end of bottom row
               --============================================================
               if ((row /= 0) and (row /= 15)) then
                 if (col = 0) then
                   -- we are in the left side regio

                 elsif (col = 15) then
                   -- we are in the right side region

                 else
                   -- we are in the middle sections

                 end if;
               end if;
            else
            -- reset condition
            end if;
    end if;

end process Interp_Block;
```

2.4.7 Histogram Pipeline

The histogram pipeline forms the back bone of the CLAHE algorithm. Provided with a singular contextual region, a pipeline will produce a re-mapping function to be used in the bilinear interpolation in order to adjust the contrast in a particular region of the source image. From the top level description, there are four pipelines operating in parallel. Each pipeline retrieves a contextual block from the source image, creates a local histogram, redistributes its contents and then signals to the histogram sequencer that this remapping function is available to be stored in the histogram RAM.

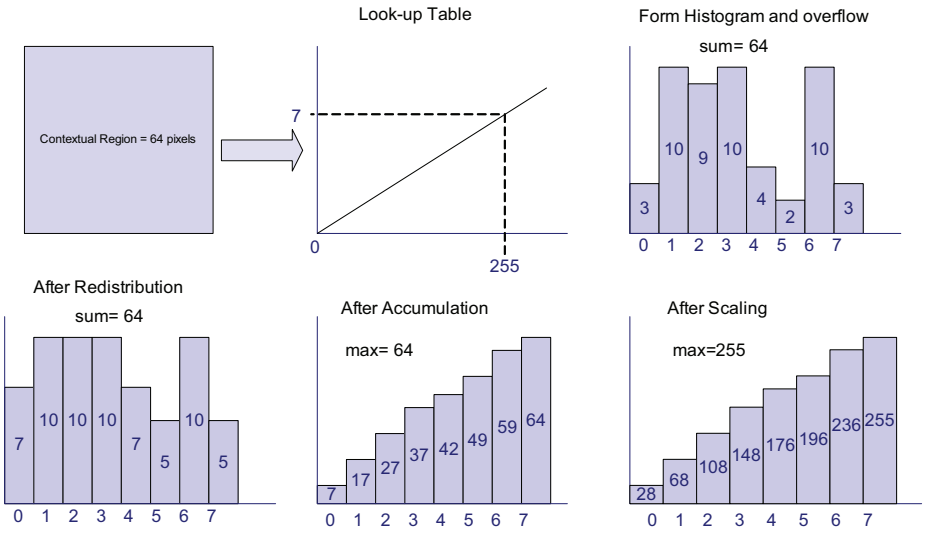


Figure 2.10: Histogram Pipeline Arithmetic

A graphical view of the stages through the pipeline is shown in Figure 2.10. The histogram pipeline contains four distinct parts to accomplish the construction of the re-mapping functions:

1. The look-up table (LUT) translates a pixel value into the histogram bin that it inhabits.
2. When a bin has been identified, the current value is assessed to see if adding to its contents will increase the value past the clip limit.
3. After all the pixels have been assigned a bin, a histogram and overflow will remain.This overflow is then redistributed.
4. The final stage forms a remapping function by accumulating and scaling the contents of the redistributed histogram.

The end result of this pipeline is a re-distributed histogram contained in a block RAM module allowing ease of access for the bilinear interpolation. When requested by the histogram sequencer, the histogram can be accumulated, scaled and transferred out to the appropriate location required by the bilinear interpolation.The following Subsections describe the histogram pipeline in the context of the implementation structure shown in Figure 2.11.

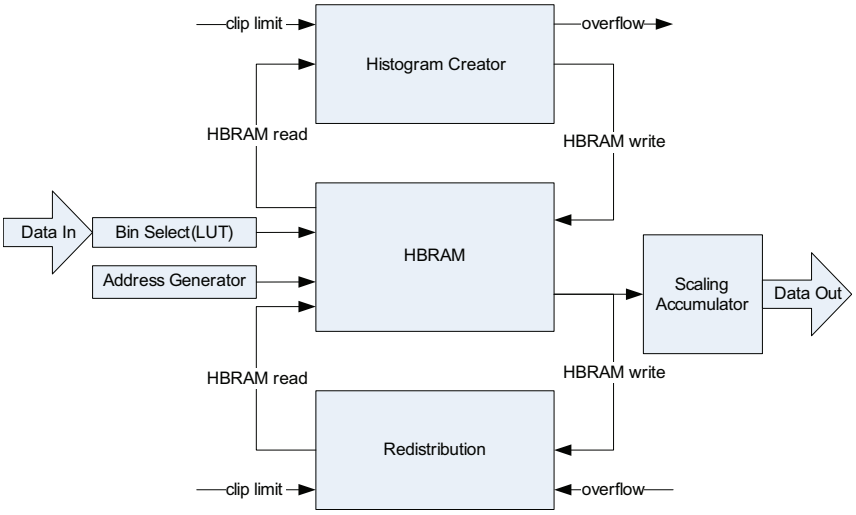


Figure 2.11: Histogram Pipeline Structure

2.4.7.1 Histogram Creation

Histogram creation is the first stage in CLAHE. The aim of this block is to produce a clipped histogram and overflow amount from the clip limit and contextual region pixels. Both the redistribution operation the histogram creation operation are performed around one dual port block RAM. The block RAM module[2] allows one port to write and read the resultant histogram values into the RAM and the other port allows the redistribution operation to have access to the clipped histogram. To store the overflow a register was instantiated which updates with each new histogram created.

From each pixel value supplied to the histogram pipeline, a corresponding address must be generated to indicate which bin the pixel belongs in. This function is easily implemented by a block ROM module (see Figure 2.11) acting as a look-up table. This look-up table divides the pixel values from an eight bit range of 0-255 down to the histogram range of 0-7 for an eight bin histogram. The output from the look-up table provides the address lines for a bin (i.e. a memory location) in the histogram block RAM (HBRAM). The contents of an address are read, one is added to that value, and the updated value written back to the same address. On the condition that the contents read from bin are already equal to the clip limit, one is written to the overflow register and the bin contents remain at the clip limit. Once all the pixel values have been read the HBRAM will contain a clipped histogram and register the overflow contents ready for redistribution. A small FSM is used to sequence these operations and alert the redistribution operation that it is now allowed to operation on the contents held in HBRAM.

[2] A member of the predefined hardware blocks available from Xilinx Core Generator[62]

2.4.7.2 Redistribution

The redistribution process operates on the same block RAM module as the histogram creation operation. It occupies the second port of the HBRAM for its read and write cycles, interfacing solely on the one block RAM. Redistribution is composed of two parts:

1. Histogram bin update calculation
2. Overflow reduction calculation

The redistribution operation begins by storing the overflow count and then reads a value from each memory location. For each bin value that is read a decision must be made upon what action to take on that value. This decision is dependent on the current value of the overflow and how close the bin value is to the clip limit. The overflow must be divided sequentially (not equally) across the histogram and so the overflow is separated into a quotient and a remainder. The quotient is produced by dividing the overflow by the number of bins (shifting to the right 3 binary places for eight bins) and a remainder is produced by subtracting the quotient from the overflow.

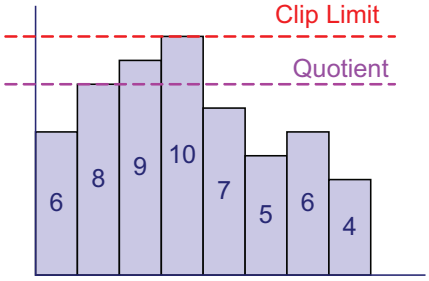


Figure 2.12: Redistribution Options

From Figure 2.12 we can see that there are 3 options for each bin read from HBRAM:

1. If the bin value is equal to the *clip limit* the do nothing and return the same value to the HBRAM
2. If the bin value is equal to the (*clip limit* - *quotient*) then we can only add the quotient
3. If the bin value is in between the *clip limit* and (*clip limit* - *quotient*) then we must add the quotient (creating a value larger that the *clip limit*), subtract the clip limit and add the result to the remainder to be redistributed. The value returned to the HBRAM is the *clip limit*.
4. If the bin value is less than (*clip limit* - *quotient*) then we can add the quotient plus one from the the remainder- the maximum value that can be added to one bin in one cycle. If the remainder value is zero, the quotient must still be added to a bin.

After the quotient has been added the number of bin times (in this case 8), the quotient must be changed to zero. At this point, in most cases the remainder should also be zero and the operation is complete, however due to condition 3 there may still be a remainder left to distribute. All four

Bin Addresses		0	1	2	3	4	5	6	7
Original Bin Values		3	12	9	16	4	2	15	3
Redistribution Parameters		clip = 10, overflow = 13							
Cliped Contents		3	10	9	10	4	2	10	3
Active Condition		4	1	2	1	4	4	1	4
Cycle 1	quotient = 1	1	0	1	0	1	1	0	1
	remainder = 5	1	0	0	0	1	1	0	1
Update		5	10	10	10	6	4	10	5
Active Condition		4	1	1	1	4	4	-	-
Cycle 2	quotient	1	0	0	0	1	1	0	0
	remainder	1	0	0	0	0	0	0	0
Final Bin contents		7	10	10	10	7	5	10	5

Table 2.3: Redistribution Conditions 1, 2 & 4

conditions are demonstrated in Tables 2.3 & 2.4. The first two rows indicate the contents of a histogram bin and its address. The address only indicates by how much the pixel value range has been sectioned. Applying the a clip limit will automatically form an overflow value. The following rows show the overflow being redistributed and the condition that apply from the options discussed above. For Table 2.4, dividing the overflow across eight bins produces a quotient of two and a remainder of four. The quotient must be added to the histogram eight times (to evenly distribute it across eight bins in the histogram) and the remainder must be added where it can fit. This does bias the remainder across the lower bins but should not result in a significant skew. For bin 4 on cycle 2, the value 7 will apply condition 3- this means that only 1 out of 2 can be added to that bin to keep it under the clip limit.In this case, one is added to the bin and the remainder.

Bin Addresses		0	1	2	3	4	5	6	7
Original Bin Values		3	12	9	16	4	2	15	3
Redistribution Parameters		clip = 8, overflow = 20							
Clipped Contents		3	8	8	8	4	2	8	3
Active Condition		4	1	1	1	4	4	1	4
cycle 1	quotient = 2	2	0	0	0	2	2	0	2
	remainder = 4	1	0	0	0	1	1	0	1
Update		6	8	8	8	7	5	8	6
Active Condition		4	1	1	1	3	4	1	4
Cycle 2	quotient	2	0	0	0	1	2	0	2
	remainder	0	0	0	0	0	1	0	0
Final Bin contents		8	8	8	8	8	8	8	8

Table 2.4: Redistribution Condition 3

To accomplish this, on every clock cycle a bin value is evaluated by concurrent conditions that preset multiplexers to indicate the next action to perform on that bin value. There is only one condition where redistribution would be partially successful - if the clip limit was less that the number of histogram bins. However it is predefined that the clip limit must be greater than or equal to the number of histogram bins.

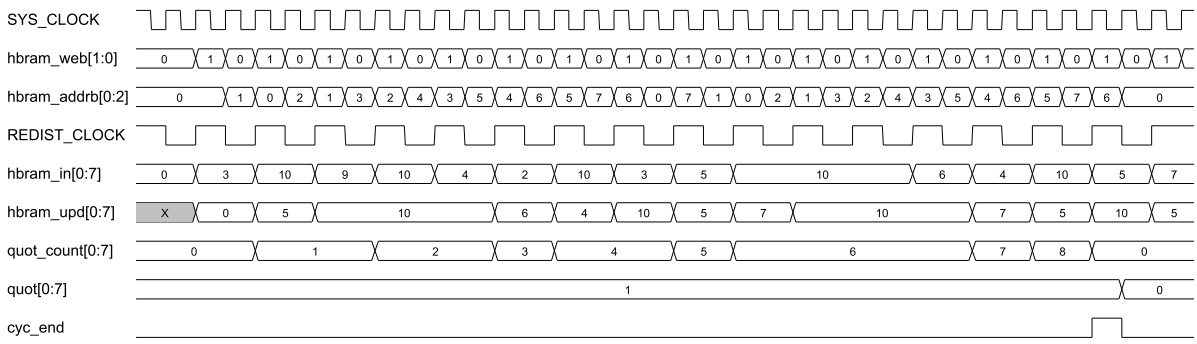


Figure 2.13: Redistribution Waveform

The redistribution operation due to its indeterminacy is the largest bottle neck in the algorithm, it is therefore important that it is very cycle efficient in its implementation approach. The redistribution operation produces a revised bin value within one clock cycle, meaning that the interface to the HBRAM is very active throughout the operation. A typical redistribution operation is shown in Figure 2.13, where the memory interface(*hbram_addr* and *hbram_web*) change every clock to multiplex the read and write update cycles. Between each rising edge of the redistribution clock(*redist_clk*) the memory will have been clock twice. Firstly to read the contents of the current bin and secondly to write out the update to the previous bin. In the context of Figure 2.13 this means that the bin addr will increment on every rising edge of the redistribution clock. However on its falling edge the previous bin address will be updated. Once the quotient has been added the number of bin times(which matches *quot_count*), and the remainder is zero then a flag to indicated the end of the redistribution(*cyc_end*) is raised and the state machine will then schedule subsequent events.

2.4.7.3 Accumulation

The next stage of CLAHE, to create a re-mapping function, is to firstly sum the contents of the histogram, and secondly to blank the HBRAM so that it is empty for the next contextual region. The address and data bus to the HBRAM are shared between the redistribution operation and the accumulation operation. The accumulation operation does not need the complex addressing scheme used by the redistribution operation and so it can be deactivated in favour of a simpler mechanism (a counter) to read the memory contents. On each read, the value is passed to an accumulator and then writes back a zero to the same location in the HBRAM. The output from the accumulator is pick up and passed to the histogram RAM. During the accumulation there are 2 issues to address:

1. Summing the histogram would produce a re-mapping function with an output range of 0-64, a 6-bit number, whereas the input is an 8-bit number, a pixel value range of 0-255 range.

2. Set the re-mapping function to have an offset to match the zero intensity pixels or the maximum intensity pixel.

In the case of point 1 the output pixel values must be in the range 0-255, however they are pre-quantised in their potential values by the number of bins in the histogram (if reduced to less than pixel bit-width). The input pixel values need only pass through the ROM look-up table from the histogram creation (Section 2.4.7.1) to select the correct output value. Scaling the output range is achieved by shifting the accumulated values by 4 (the inverse gradient of the look-up table function), however this would mean that the final value would be 256, wrapping around to 0. To combat this the final value from the memory is altered to ensure that the maximum output value is 255. The scaled re-mapping function is only applied when the histogram is being moved out to the larger interpolation function store.

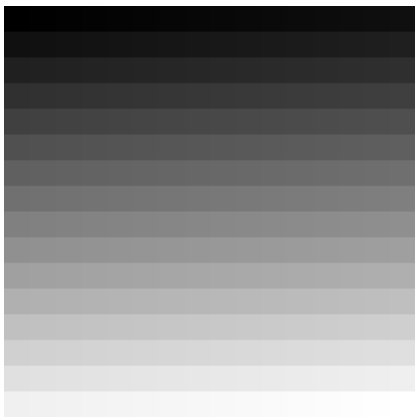


Figure 2.14: Mach Band

Point 2 can be answered with a simple mach band shown in Figure 2.14. Although intended to demonstrate the overshoot and undershoot in brightness of the human visual system, here the mach band is a simple example of how the human visual system can perceive the lighter bands with greater distinction than the darker bands. This justifies the decision to offset the re-mapping function to 255 meaning that there will be greater separation between the lighter band.

2.4.7.4 Histogram Pipeline Finite State Machine

The role of the finite state machine (FSM) is to coordinate the operations within the histogram pipeline(as discussed previously) and conduct the transfer of re-mapping functions to the histogram RAM. As each histogram may require an indeterminate redistribution time, the FSM must handshake to the histogram RAM once redistribution is complete and then sequence the next contextual region from the image store. A hierarchical view of the FSM and histogram pipeline is shown in Figure 2.15.

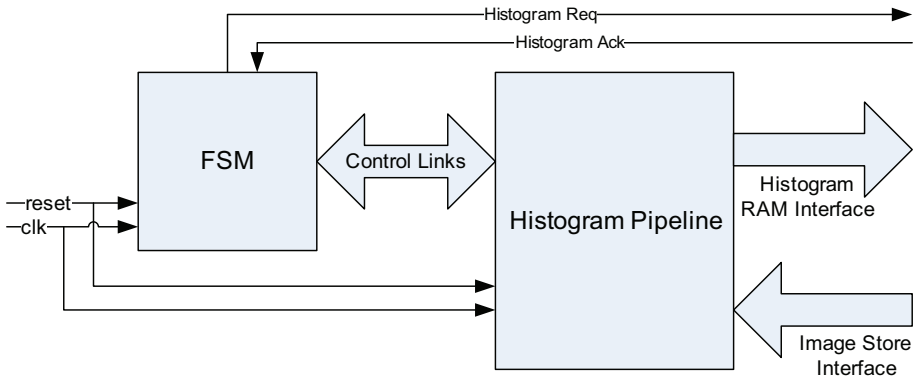


Figure 2.15: Histogram Pipeline with Finite State Machine

The control links are essentially the enable lines of the components in the histogram pipeline. These are primarily memory blocks, address counters and progress indicators such as *'cyc_ end'* shown in Figure 2.13. The image store interface, although a simple memory interface, is governed by the state of the handshake signals that sequence the loading of remapping data into the histogram RAM. The interface to the histogram RAM is simply the output of the scaling accumulator with additional control signals to indicate address locations and write transactions.

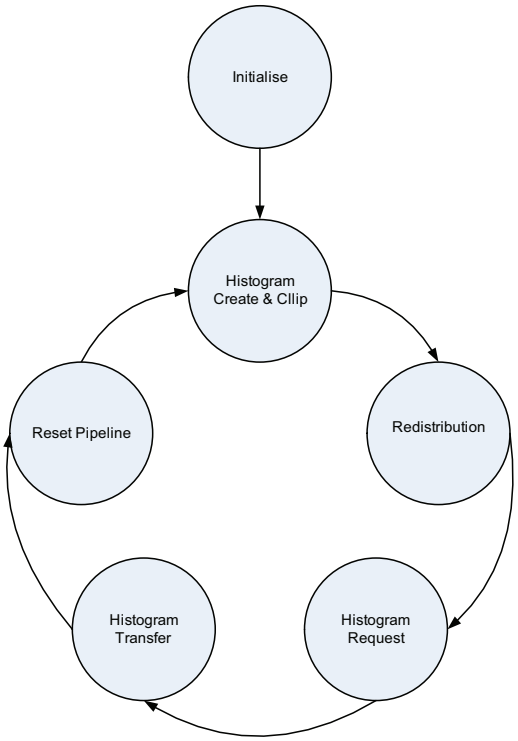


Figure 2.16: Histogram Pipeline States

The state graph in Figure 2.16 shows the top level sequence of histogram pipeline states. The initialise block is active on start up to ensure that no element in the operation is free-running. The next step is to create histograms, clipping their contents as they are assembled in memory. The primary function of this state is to sequence the interface to the image store whilst enabling the

address counter for the HBRAM and the HBRAM itself. The redistribution state is also very similar, this time enabling the redistribution operation and associated memory block. After this process is complete the state machine indicates to the histogram RAM controller that a histogram is ready to be transferred over. In between this handshake the histogram is accumulated and scaled into a re-mapping function. The child states within the Histogram Transfer state sequence the operation to enable the accumulator and allow access to the histogram RAM. The final reset state, controls a sequences of events to clear the HBRAM and resets all address counts to allow another histogram to be processed. The state machine will always stop on the flow request and await confirmation before moving the re-mapping functions out of HBRAM. The create and clip state usually takes approximately 70 clock cycles, the redistribution state although indeterminate typically takes under 3 passes to redistribute the overflow, the transfer of the remapping function out to the histogram RAM is dependent on the size of the histogram in the case it is normally 10 cycles to complete the transfer.

2.4.8 Smoothing Contextual Regions

The inherent use of contextual regions to construct localised histograms means that there will always be contrast differences between contextual regions. Without a smoothing process, a blocking effect will occur allowing pixel intensity differences between contextual regions to be identifiable in the output image. Bilinear interpolation is more commonly associated with resizing images, in the case of CLAHE it is used to eliminate the blocking effect between contextual regions, normalising the contrast change between contextual regions. The parallel nature of FPGA devices mean that they are well suited to implement bilinear interpolation regardless if it is being used to smooth or re-size images. Normally this is an arrangement of registers, dedicated multipliers and adders which perform the majority of the arithmetic, the complication comes in ensuring that the correct parameters and data are aligned in the correct order. When resizing an image interpolation would normally be decomposed into its horizontal and vertical components to produce a low latency, efficient pipelined implementation. In the case of smoothing contextual region boundaries the focus is on reducing the amount of hardware and arithmetic operations need to achieve the desired operation.

The equations from Section 2.3.3 can be reduced and re-factored to:

$$\begin{aligned} P_{new} &= \mu(\eta \cdot f(Q_{12}) + (1-\eta) \cdot f(Q_{22})) \\ &\quad + (1-\mu)(\eta \cdot f(Q_{11}) + (1-\eta) \cdot f(Q_{21})) \end{aligned} \tag{2.4}$$

$$\begin{aligned} &= \mu\eta\, f(Q_{12}) - \mu\eta\, f(Q_{22}) + \mu\, f(Q_{22}) \\ &\quad - \mu\eta\, f(Q_{11}) + \mu\eta\, f(Q_{21}) - \mu\, f(Q_{21}) \\ &\quad + \eta\, f(Q_{11}) - \eta\, f(Q_{21}) + f(Q_{21}) \end{aligned} \tag{2.5}$$

$$
\begin{aligned}
= \quad & \mu\left[\eta\left(f(Q_{12}) - f(Q_{22})\right) + f(Q_{22})\right] \\
& - \mu\left[\eta\left(f(Q_{11}) - f(Q_{21})\right) + f(Q_{21})\right] \\
& + \eta\left(f(Q_{11}) - f(Q_{21})\right) + f(Q_{21})
\end{aligned} \tag{2.6}
$$

If the substitutions are made that:

$$
\begin{aligned}
P_1 &= \eta\left(f(Q_{12}) - f(Q_{22})\right) + f(Q_{22}) \\
P_2 &= \eta\left(f(Q_{11}) - f(Q_{21})\right) + f(Q_{21})
\end{aligned}
$$

then Equation 2.6 reduces to:

$$P_{new} = \mu(P_1 - P_2) + P_2$$

This reordering has reduced the implementation to a simple sequence of multiplication and additions. The resultant hardware is arranged as shown in Figure 2.17. This implementation has the flexibility to reuse and share hardware if there is sufficient memory available. This arrangment is easily arranged into the embedded DSP48 slices [18] provided by the Virtex 4 device

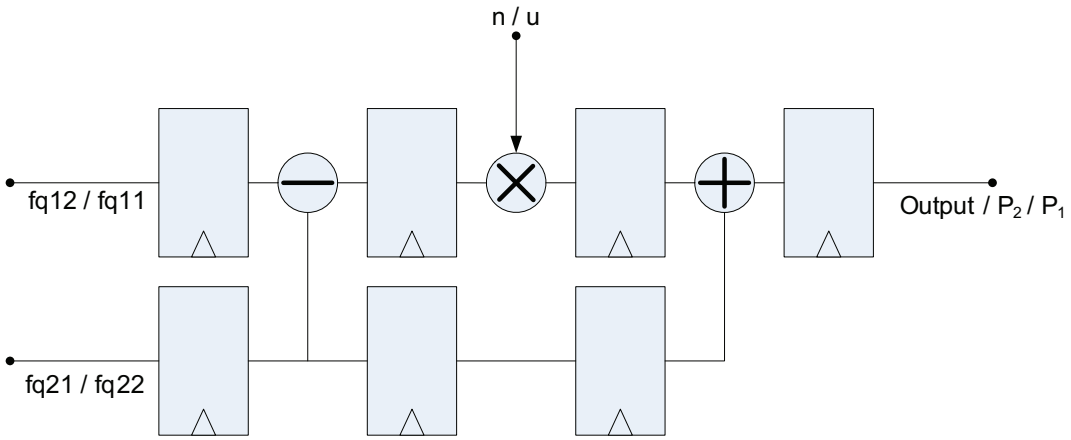


Figure 2.17: Bilinear Interpolation Minimisation

The bilinear sequencer acts to arrange the various parameters for this operation. Depending on the postion of the pixel being processed, the correct remapping function must be aligned and the pixel weights must be introduced at the correct time. As with other memory operations, we assume that it is running at the faster clock speed and the operation, in this case interpolation is clocked to support data being processed in a contiguous manner without any enable lines to validate the data.

2.4.9 Top Level FSM

The top level FSM coordinates the entire algorithm from a block level perspective as discussed in Section 2.4.2. Although some interactions between blocks are autonomous such as loading remapping functions into the histogram RAM the remaining top level modules require an FSM to implicitly sequence their operations correctly. Figure 2.18 shows a graphical view on the sequence of operations.

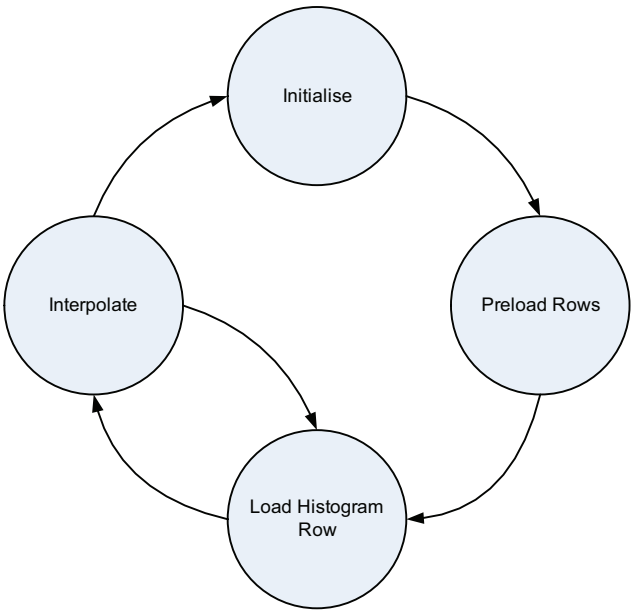


Figure 2.18: Top Level State Machine

A debounced user activated pin activates the algorithm from a reset state. The initialise state then enables the histogram pipelines to process two full rows of contextual regions. After this, contextual regions are loaded one row at a time. Pre-loading 2 rows of contextual regions allows the interpolation to free run until it requires additional histograms.Since there are finite memory resources, the top level will then oscillate between loading and processing contextual regions and interpolating between them. This means that the top level FSM must coordinate with the weight generator, effectively controlling what pixels are read from the image store as well as the histogram sequencer to ensure that the interpolation operation has all its parameters easily accessible. Once the final contextual region, and eventually the final pixel are interpolated the top level FSM will return to the initialise state to begin on another image. As mentioned previous nearly all of the operations are finite in their duration apart from the histogram formation. This is why the interpolation and remapping function generation cannot execute in parallel because there is a danger that the interpolation will be looking for a remapping function that is not available due to delays in redistributing its histogram.

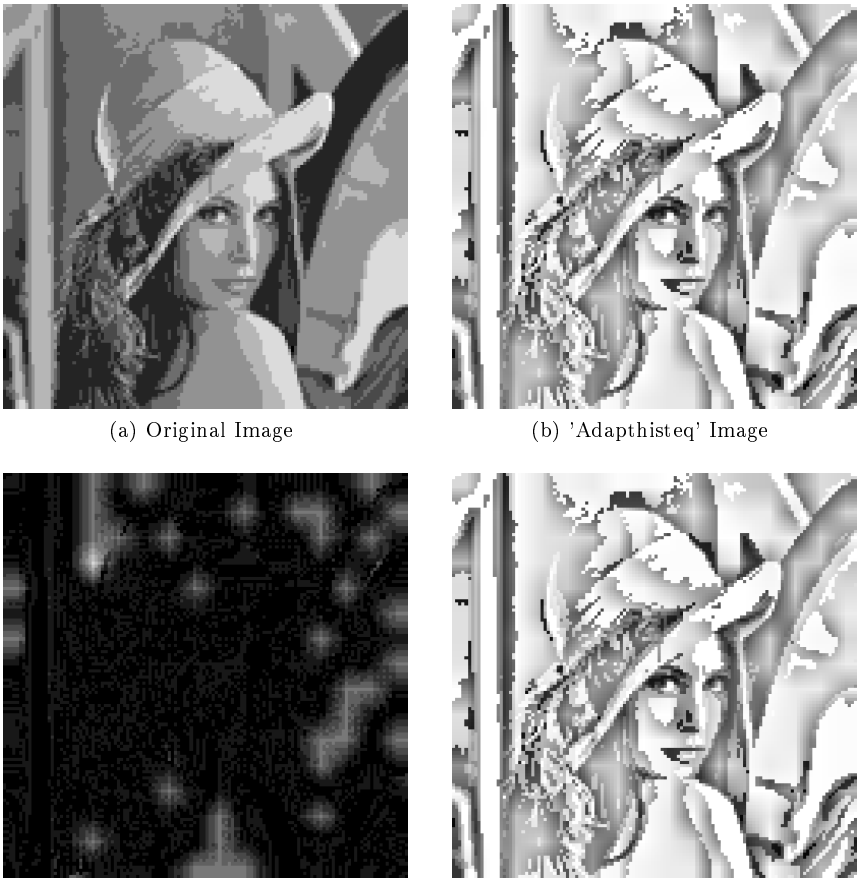
2.5 Analysis and Results

The analysis of the work undertaken in developing the CLAHE implementation has been divided into three sections: the image quality evaluation, the speed of the algorithm and the power consumption of operating the algorithm entirely within an FPGA device.

2.5.1 Image Correctness

In the standard FPGA design flow implementation errors are never entirely discovered during behavioural or even post synthesis simulations. Normally firmware implementation errors continue to exist when the design is fully implemented on the device. To verify that the downloaded bit stream on the FPGA was accurate and functionally correct, the output from the algorithm was captured through a series of trigger points using an on-chip logic analyser, Chipscope Pro [16]. This tool inserts a customisable logic analyser into the synthesised netlist, allowing on-chip events to be monitored and captured. The logic analyser can monitor any internal signal and can be triggered off any number of events. Data can then be captured and communicated back to the host pc via a JTAG boundary scan port. This tool runs synchronous to the source design and thus applies any design constraints to its own logic, reducing any potential impact on the source design. The data captured by the logic analyser is presented as a waveform on the host pc, which is then exported for comparison.

In this context the FPGA implementation is compared against a tailored version of MATLAB's *'adapthisteq'* function with the same parameters. This provides a golden reference model to assess the accuracy of the image from the FPGA implementation.



(a) Original Image (b) 'Adapthisteq' Image

(c) Size and Location of Errors (d) FPGA Image

Figure 2.19: Image Results Comparison

From first observations, the difference in images[3] in Figure 2.19b and Figure 2.19d are minimal. As expected, there is a significant improvement in contrast from the original image in Figure 2.19a. However using a clip limit of 40 indicates that there are few areas where there is any significant contrast enhancement differences between the two implementations. Figure 2.19c displays the results of the *imabsdiff* function in MATLAB, which returns the absolute difference between the two images clearly showing the locations where errors in the FPGA implemenation occur. The black pixels signify no error, the brighter the pixel the larger the size of the error.

[3]Note the images have been rescaled to show the individual pixels in detail

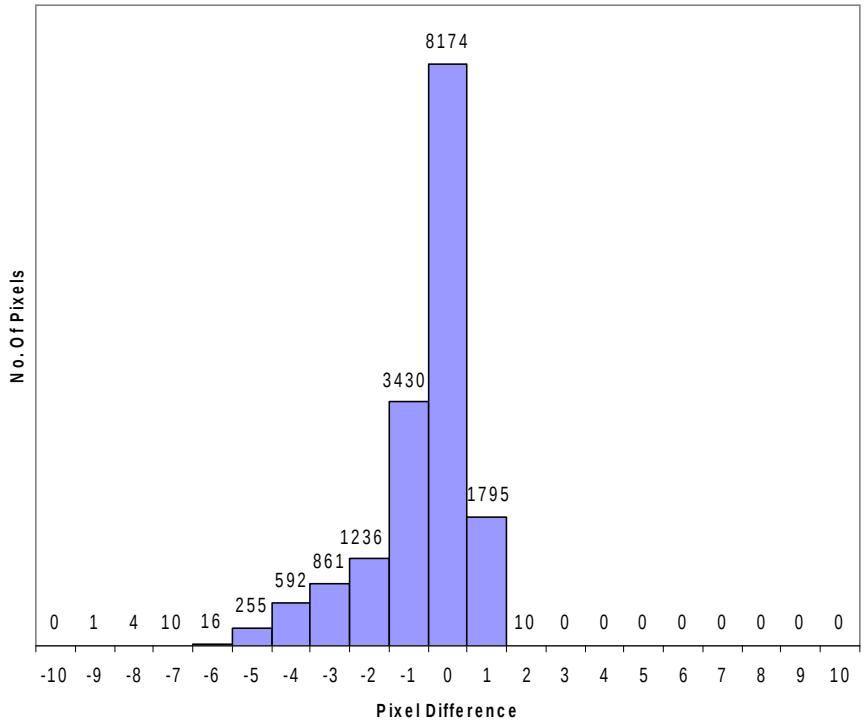
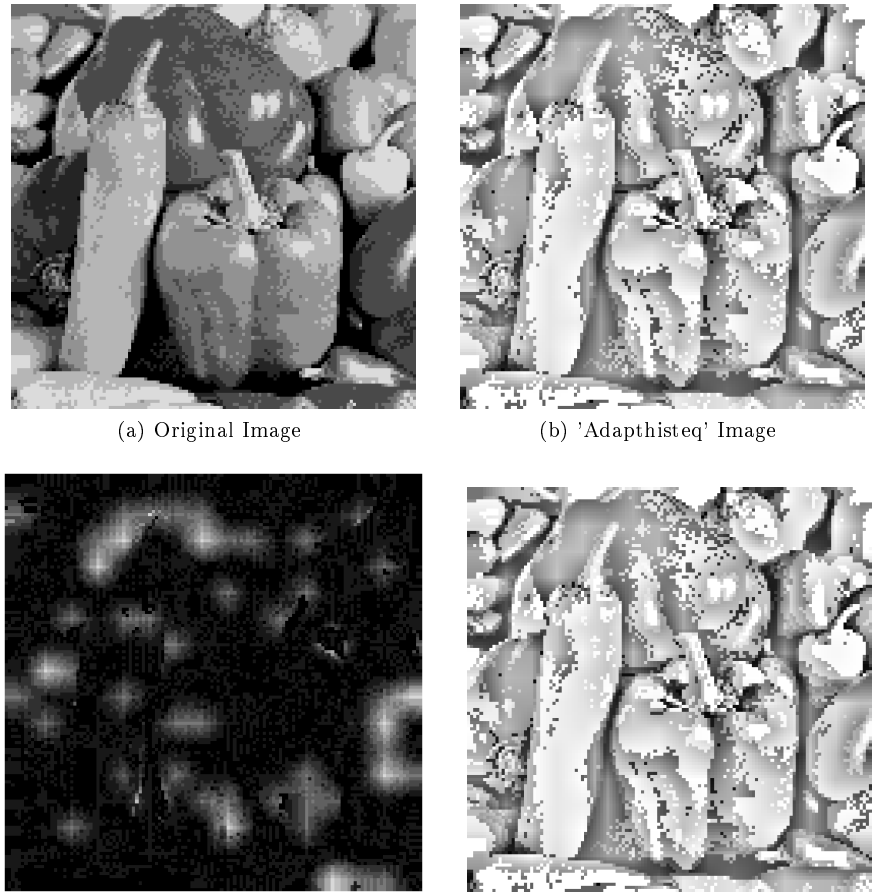


Figure 2.20: Adapthisteq VHDL differences

A histogram of this difference image shows (in Figure 2.20) that the majority of discrepancies are coming from the hardware variant overshooting, i.e. producing an output pixel intensity value that is larger than the same pixel intensity value in the '*adapthisteq*' image. Considering that 82% of the pixels are with ± 1 intensity step value of the MATLAB image this is an acceptable image. Two measures of conformity can be drawn from this function, the sum of absolute differences and the mean squared error. Spread over 16384 pixels the sum of absolute differences equates to 14150, leaving a mean absolute error per pixel of 0.86. The mean squared error for this image is 2.1489.

The same differences can be seen in the peppers image of Figure 2.21 where again there is a minimal difference in intensities that are noticeable to the naked eye.



(a) Original Image (b) 'Adapthisteq' Image

(c) Size and Location of Errors (d) Hardware Image

Figure 2.21: Additional Image Results Comparison

The difference between these test images are comparable to the first test image with a 76% of pixels within ± 1 step value and the sum of absolute differences is 18472 resulting in a mean absolute error per pixel of 1.127 average error. The mean square error are to be expected is greater on this image at 3.5051. As with Figure 2.20 the intensity difference shown in the Figure 2.22 are not identifiable by image locations or artifacts and they must be attributed to specific arithmetic conditions.

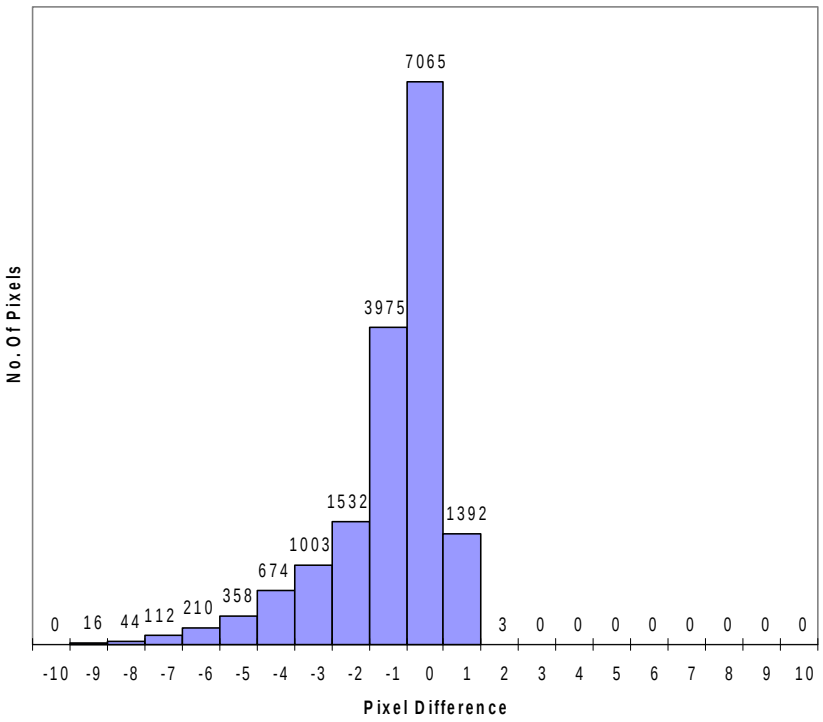


Figure 2.22: Consistent 'Adapthisteq' VHDL Differences

These slight differences can be attributed to two factors:

1. the limited greyscale range of a 3-bit image will force pixel intensity to change greater by a greater amount than an 8-bit image.

2. there is significant truncation of fractional bits in the interpolation scheme which when compared to the dynamic range of the MATLAB number system would indicate that in some cases rounding has preserved certain contrast changes.

2.5.2 Resource Utilisation Results

A key metric in integrating CLAHE into an image processing system that is based around an FPGA is resource utilisation. Commercial design FPGA designs are chosen so that the implemented logic occupies around 60% of the available resources, allowing for any subsequent specification changes and future upgrades to happen with minimal cost. There is no published FPGA implementation to compare the utilisation results of CLAHE against and so the follow discussion will be in context of its ability to integrated into an FPGA based image processing system.

Device Utilisation Summary			
Logic Utilisation	Used	Available	Utilisation
Number of occupied Slices	1715	5472	31%
Number of Slice Registers	1578	10944	14%
Total Number 4 input LUTs	2671	10944	24%
Number used as logic	2490		22%
Number used as a route-thru	181		
Number of bonded IOBs	70	320	21%
Number of FIFO16/RAMB16s	18	36	50%
Number used as FIFO16s	0		
Number used as RAMB16s	18		
Number of DSP48s	16	32	50%
Number of RPM macros	11		
Total equivalent gate count for design	1,211,190		

Table 2.5: Device Utilisation

As is evident from Table 2.5 there is plenty of space still available on the device. Only 14% of the slice registers and 24% of the slice Look-up tables are used in this implementation leaving ample room for other elements in a video processing pipeline. The Bonded IO Blocks are included as a point to note on external memory interfaces, otherwise the number included is mostly debug pins. The output would only be the width of the data used and a validity indicator, in this case that would be 9 bits. If an external memory device was to be used the pin count would increase to accommodate this interface. As previously discussed in Section 2.4.3 18 BRAMs are used in this implementation, split between image memory and histogram stores. Scaling up the image size and histogram precision would naturally push the image memory onto an external device. The histogram stores would be a design consideration if the contextual regions were to stay the same size. This would be the key balance between other parts of an image processing system, allowing histogram formation to occur on-chip could occupy a significant amount of block RAM. All DSP48 blocks, the embedded arithmetic blocks consisting of a large multiply accumulate unit are utilised within the bilinear interpolation and inferred through the synthesis process for the histogram updates.

This implementation is very well suited to an FPGA implement due to its parallelism in processing contextual regions and remapping functions, however there is concerns over the on-chip memory utilisation given the choice of remapping function size.

2.5.3 Timing Results

Throughout the design flow there are various estimates on the maximum clock that can be used in the circuit. The synthesis tool reports a maximum clock speed of 126.566MHz. Once the netlist has gone through place & route, the static timing analysis tool reports that the maximum clock speed is 107.009 MHz. Both tools report the same critical path which spans the redistribution block. The critical path in terms of primitive components is shown in Appendix A.2. The redistribution operation shown in

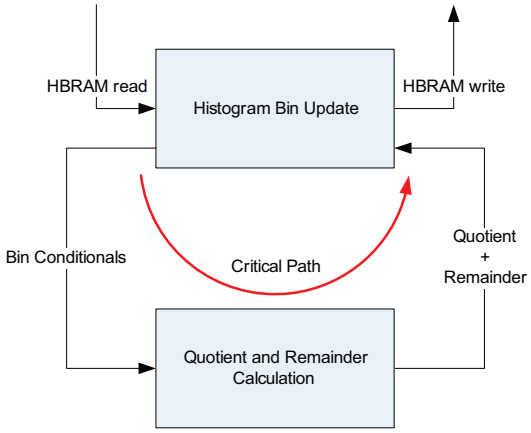


Figure 2.23: Critical Path through Redistribution block

Figure A.1 is made up of two blocks. The histogram bin update accesses the HRAM, reads a value, sets conditional flags (as discussed in Section 2.4.7.2), and waits on the values to update the HRAM location. The new values are calculated in the other block which tracks how and when to distribute the quotient and remainder from the overflow value based on the conditional flags. The quotient and the remainder calculation has a number of feedback lines that are not pipelined due to the timing required for the HRAM interface as detailed in Section 2.4.7.2. The maximum clock reported by this tool means that the logic is more that capable of running at the 100MHz clock supplied by the development board.

For both test images shown in Section 2.5.1, Lena took 796.86μs to complete and the peppers took 794.9μs. This is proximately 80,000 clock cycles. The variability between each image is due to the number of passes of the histogram required to redistribute the contents. In the majority of cases this will only be one or two passes however there may be the odd case that requires more, unfortunately this is image dependant. The only way to guarantee a completion time is to restrict the number of passes, discarding the modulus after at most two passes. At this point the majority of the histogram would be redistributed, but due to mismatch of the number of pixels in a contextual region the re-mapping function would be inaccurate. In a video stream these in-accurate re-mapping functions would require a visual acceptance test.

For a 25 frame per second video stream, a frame must be formed and complete in 40ms. If we were using the implementation discussed previously this would leave 39.2ms to load the image and exporting the results to a frame buffer. However this Figure is only for a small image size, increasing this size to something practical as shown in previous studies would increase the processing time dramatically. Due to the design of the implementation, scaling the amount of processing time required is relatively simple. Assuming that an 8 bit range histogram is used, requiring an average bin contents of 16, the image dimensions would scale to a multiple of 64 and thus an appropriate and historically comparable image size would be 512 x 512. In terms of a single histogram pipeline, there

would be 16 contextual regions processed in total each containing 256 pixels. One contextual region would take 768 clock cycles to load, create the histogram and calculate the excess. If we assume that this is a worst case homogeneous contextual region with only one brightness value, and that the clip limit was set to the minimum (histogram average of 16) then redistribution would take another 256 clock cycles. Transferal to the bilinear store would then take 512 clock cycles. For interpolation, each pixels takes 4 clock cycles to computes and so would take a total of 512 x 512 x 4 clock cycles to complete the image. In total (excluding the control clock cycles) this would take 1146880 clock cycles to complete. Using the development board clock one 512 x 512 image would take 11.47ms to enhance, and assuming that the DDR memory interface is operating at 1600MBs then this is more than capable of sustaining constant video.

2.5.4 Power consumption

Another aspect to characterising CLAHE on an FPGA is the power consumption. Along with resource utilisation, power consumption is a key factor in which FPGA engineers use to characterise their circuit/system. This Section discusses the power consumed during the operation of CLAHE and more specifically the variation when partial image textures are predominant in an image.

The development board used to implement the histogram pipeline had the option to measure the current drawn by the voltage regulator [21] (shown in Figure 2.24) for the FPGA core voltage lines.

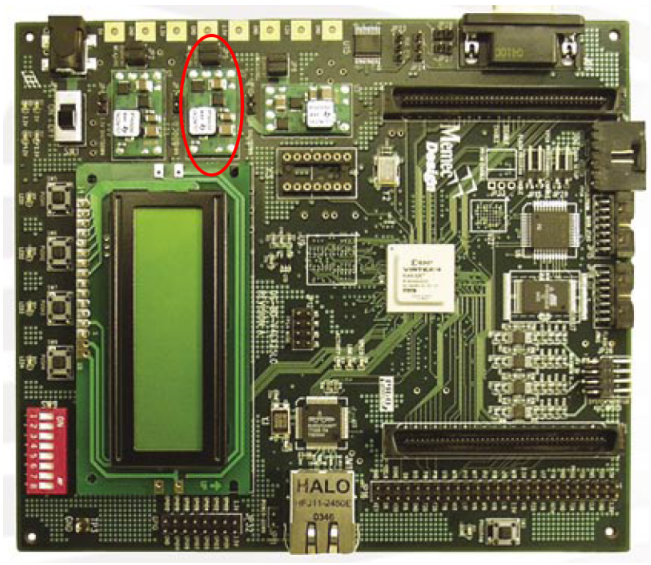


Figure 2.24: Development Board

This allows measurement of the current being drawn from the core FPGA logic and not the current required by the input/output blocks to drive signals on the development board. On activation of power-on reset the Virtex 4 FPGA draws a start-up current of at least 110mA. After this the current will lower and then rise again when the devices is configured with a bit-stream. We assume that any

increase in supply current that is measured when the circuit is not in reset indicates dynamic power consumption.

2.5.4.1 Measurement Setup

The current is measured using a LeCroy 6100A [87] scope with a AP015 current probe. Unlike using a shunt resistor, the current probe does not affect the measurement itself. The only element to skew the measure of current is the voltage regulator. The break-in points, shown in Figure 2.25 are situated on the particular parallel branch of the 5 volt line that supplies the voltage regulator configured for 1.2v. This means that the current drawn from this line has the inefficiencies of the regulator to take into account.

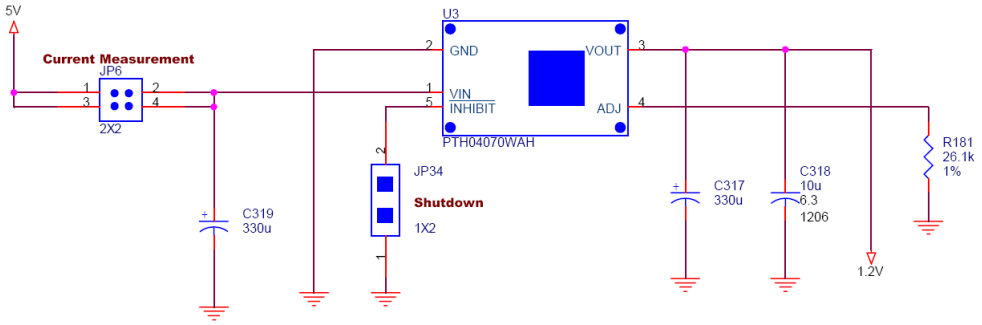
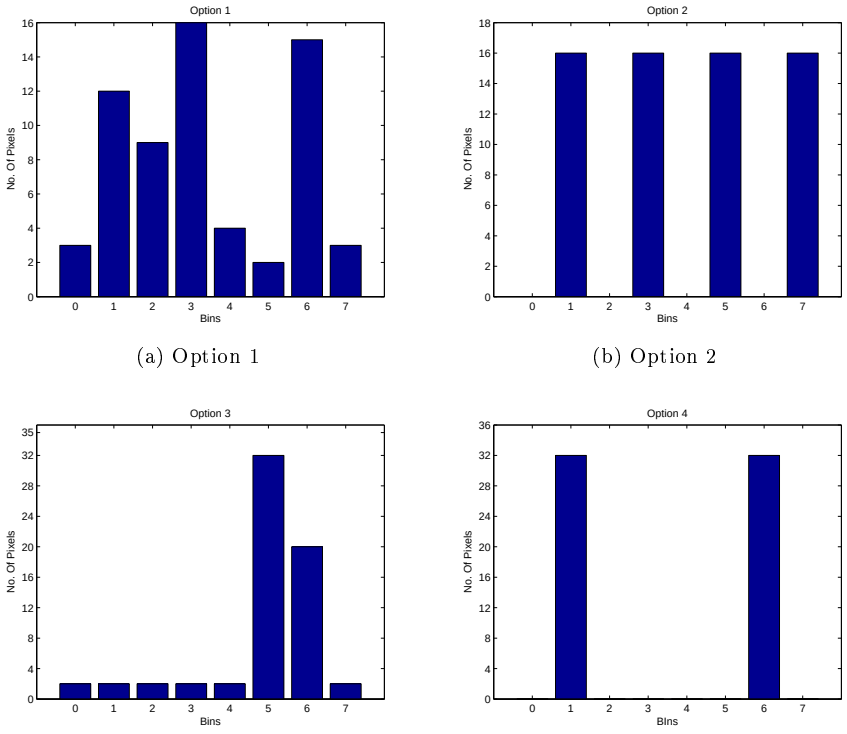


Figure 2.25: Voltage Regulator for FPGA Core

When the regulator is setup to supply 1.2v for the FPGA core the efficiency is stated in the data sheets as 82%. This means that 18% of the measured current is being lost through the voltage regulator. Since one histogram takes approximately 2.7μs to form and create a re-mapping function, attempting to measure the current drain with sufficient temporal resolution over this short period of time is very difficult. To ensure a sufficient run time, the algorithm is triggered and terminated by a push button and the execution status indicated by on-board LED's. This provides a simplistic system to identify the dynamic power consumption when the algorithm is running continuously.

2.5.4.2 Histogram Pipeline Results

Since the majority of the activity in the algorithm is focused around the histogram pipelines, an individual analysis is performed to determine their impact. For an accurate representative power evaluation a histogram pipeline has been supplied with four different contextual regions which generalise the typical histograms (shown in Figure 2.26) that could be formed from 64 pixels. Option 1 was initially created to test the redistribution engine fully, but in this case would approximate a contextual region where there is a consistent texture across the region. Option 2 appears significantly more extreme but could represent a region where there is a soft edge, similar to the mach band transition of Figure 2.14. Option 3 is the case that CLAHE is most effective - here there is single a dominant intensity



(a) Option 1

(b) Option 2

(c) Option 3

(d) Option 4

Figure 2.26: Input Variation Options

that needs to be preserved and the noise minimised at an acceptable ratio. Option 4 is similar option 2 but with is a much harder edge indicated by a sharp transition between two intensities. On each variation redistributing to a clip limit of 8 requires two or three passes of the histogram, employing multiple branches of the redistribution calculation, which in turn should affect the power consumption. From the direct current measurement, the oscilloscope traces are shown in Appendix A.3. Table 2.6 summarises the important parameters to be drawn from the oscilloscope traces. There are a number of factors which appear to be contributing to the current drain on the 1.2v rails:

1. the overflow amount resulting from clipping
2. the quotient and remainder values produced throughout the redistribution
3. the contents of the clipped histogram bins - lower contents involve a simpler decision process during redistribution

All of these issues contribute to the change in current drawn. Options 1 and 2 require two passes of the histogram where as options 3 and 4 due to there larger overflow count require 3 passes. Option 4 is the anomaly that has a lower drain difference due to the lack of decision spread in the redistribution calculation - the majority of the bin contents are zero resulting in no quotient and remainder addition combinations.

Option	1	2	3	4
Max Current Drawn(mA)	87.9	89.1	88.8	87.4
Current Change(mA)	4.5	5.0	5.5	3.6
Overflow	20	32	36	48
Quotient/Remainder	2/4	4/0	4/4	6/0

Table 2.6: Histogram Parameter Summary

As a comparative measure, the XPower power estimation tool [22] was used with option 1 to estimate the power consumption of the FPGA core resources. Other components that drive output pins such as IOB's are explicitly not used in the design so as not to obscure the estimations and measurements. XPower requires a value change dump file to approximate the switching characteristics of the implemented design. This is achieved by generating a post-place & route simulation model and using Modelsim to perform the same test bench simulation as the HDL version. The switching characteristics of the entire design are evaluated by accumulating the power consumed by each element in the design. The recorded transitions allow XPower to produce consumption estimates as listed in Table 2.7. XPower reports a total current (quiescent and dynamic) of 103.06mA which is 17.2% more that the average measured current of 87.9mA. Taking into account the losses from the voltage regulator, the current being drawn by the core is 72.078mA which means that Xpower's over estimate increases to 42.98%. This is consistent with other studies into the accuracy of Xpower [8].

VccInt = 1.2v	Current (mA)	Power (mW)
Dynamic	52.47	62.96
Quiescent	50.59	60.71

Category	Current Drawn (mA)	Power Consumed(mW)	Dominant Source
Clocks	27.11	32.54	Global clock
Logic	21.39	25.67	Block RAMs
Signals	3.97	4.76	Counters

Table 2.7: Pipeline Xpower Consumption Estimate

2.5.4.3 Overall System Results

The overall system power analysis is expected to differ from the histogram pipeline analysis due to the handshaking of data between the interpolation and the histogram pipelines. The oscillation in activity between the two parts is expected to average out the peaks from the histogram pipelines. With the same setup as found in Section 2.5.4.2 the graph shown in Figure 2.27 indicates that even with the addition of additional resources of the interpolation blocks etc, the power drawn form the supply rails is less than what was seen from the histogram pipelines running continuously.

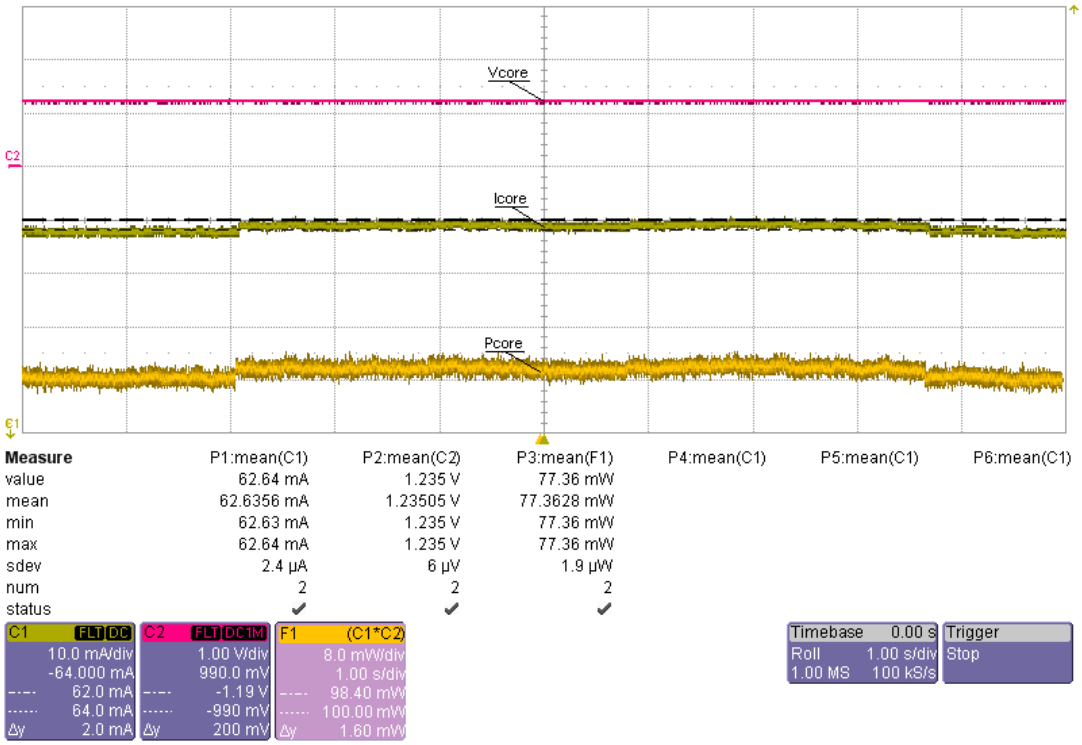


Figure 2.27: Overall Power Drain

The 2mA increase in consumption during run time brings the total power including the inefficiencies of the voltage regulator to 62.98mW. This is quite a reasonable number since the clip limit used in the Lena image was 50 rather than the extreme case of 8 which essential forced further calculations in the redistribution engine. There is quite a difference compared to the values seen in the Xpower estimation as shown in Table 2.8. The summation of the dynamic and quiescent currents totals 99mA, which in terms of an accurate estimate is very poor, however it does provided a wide margin of error in designing a power supply to support the FPGA core.

VccInt = 1.2v	Current (mA)	Power (mW)
Dynamic	48.76	58.49
Quiescent	50.65	60.78

Category	Current Drawn (mA)	Power Consumed(mW)	Dominant Source
Clocks	31.51	37.81	Global clock
Logic	14.16	17.00	Block RAMs
Signals	3.07	3.69	Counters

Table 2.8: Total Xpower Consumption Estimate

The breakdown of components that contribute to the the power drain indicate the effort required to reduce power drain must focus on the clock tree and look to minimise routing across the FPGA.

2.6 Conclusions and Future Work

The work described in this document provides detailed discussion and a valid set of results on a novel FPGA implementation of the CLAHE region based contrast enhancement algorithm. The FPGA implementation provides a valuable commercial feasibility study as well as a novel contribution to knowledge on the suitability of implementing this algorithm on a FPGA. The analysis of this implementation provides results that indicate comparable accuracy to the golden reference MATLAB implementation. The resource utilisation results indicate that the logic for implementing this algorithm is very well suitable to an FPGA architecture. However the key design factors are image size, region size and the amount of hardware memory resources. The timing results suggest that there is no throughput reducing logic that limits the clock speed at which the algorithm can operate. This is very important with most FPGA based imaging systems operating off of a high speed core clock which derives slower interface clocks. The power consumption analysis was designed to assess the impact that the CLAHE algorithm would have on the power consumption. With a set of tests that exercise the main computational element to the algorithm in comparison to a general image the results suggestion that there is minimal impact on the core voltages.

From these results we can summise that the FPGA platform is very suited to implementing regional contrast enhancement. The fast, local, memory stores embedded in its architecture are optimally place to perform contrast operations. The flexibility of the resources mean that it is adaptable to many image sizes and region sizes. The key factor in system design is the memory stores and frame buffers and how they are architected to minimise the amount data transactions to produce an output image.

Although the quality of the enhancement provided by CLAHE has been very well documented and demonstrated, there are a few aspects of this implementation, primarily concerning latency, that would hinder its industrial application:

1. Billinear Interpolation - there is a clear focus to ensure sufficient smoothing in the output image, preventing contextual regions being identified. This approach requires additional computation to create weights and store two contextual rows of re-mapping functions. The implementation effectively requires two passes of data increasing latency

2. Re-mapping Function Formation - the histogram pipelines create re-mapping functions in an indeterminate time. This is due to the redistribution process and the various options there are on how the over flow is distributed. This completion time uncertainty is unwelcome and would likely be restricted in an industrial application, sacrificing accuracy.

Investigating the design and allocation of the memory resources would address these issues. Allowing histograms to be formed in a more dynamic nature as data is sampled would reduce the contention for the histogram RAM and the weighting of the interpolation process. Controlling the redistribution

times would require a changes to the logic that governs access to the small memory blocks that contain each remapping function. These slight alterations would be classified under development and optimisation, the fundamental characteristics of the algorithm lend themselves very well to current FPGA architectures.

2.7 References

[1] K. R. Castleman, *Digital Image Processing*, ser. Englewood Cliffs, NJ. Prentice-Hall Professional Technical Reference, 1979, ISBN: 0132123657.

[2] J. M. Gauch, "Investigations of Image Contrast Space Defined by Variations on Histogram Equalization," *CVGIP: Graphical Models Image Processing*, vol. 54, no. 4, pp. 269–280, 1992.

[3] R. Hummel, "Image Enhancement by Histogram Transformation," *Computer Graphics and Image processing*, vol. 6, pp. 184–195, 1977.

[4] ——, "Histogram Modification Techniques," *Computer Graphics and Image processing*, vol. 4, no. 3, pp. 209–234, September 1975.

[5] D. Ketcham, R. Lowe, and J. Weber, "Real-Time Image Enhancement Techniques," in *Seminar on Image processing, Hughes Aircraft, Pacific Grove, California*, 1976, pp. 1–6.

[6] X. Li, G. Ni, Y. Cui, T. Pu, and Y. Zhong, "Real-time Image Histogram Equalization using FPGA," in *Electronic Imaging and Multimedia Systems II*, vol. 3561. Beijing, China: SPIE, 1998, pp. 293–299.

[7] Mathworks, "Adapthisteq : Contrast-limited adaptive histogram equalization (CLAHE)," Last Accessed: December 2011. [Online]. Available: http://www.mathworks.fr/help/toolbox/images/ref/adapthisteq.html

[8] D. Meintanis and I. Papaefstathiou, "Power Consumption Estimations vs Measurements for FPGA-Based Security Cores," in *International Conference on Reconfigurable Computing and FPGAs 2008*, December 2008, pp. 433 –437.

[9] S. M. Pizer, E. P. Amburn, D. A. John, C. Robert, G. Ari, G. Trey, R. Bart Ter Haar, and B. Z. John, "Adaptive Histogram Equalization and its Variations," *Computer Vision, Graphics, and Image Processing*, vol. 39, no. 3, pp. 355–368, 1987.

[10] S. M. Pizer, R. E. Johnston, J. P. Ericksen, B. C. Yankaskas, and K. E. Muller, "Contrast-limited Adaptive Histogram Equalization: Speed and Effectiveness," in *Proceedings of the First Conference on in Biomedical Computing*, 1990, pp. 337–345.

[11] A. M. Reza, "Realization of the Contrast Limited Adaptive Histogram Equalization (CLAHE) for Real-Time Image Enhancement," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 38, pp. 35–44, 2004.

[12] O. Sims, "Efficient Implementation of Video Processing Algorithms on FPGA," Ph.D. dissertation, Institute of System Level Integration, 2007.

[13] J. A. Stark, "Adaptive Image Contrast Enhancement using Generalizations of Histogram Equalization," *IEEE Transactions on Image Processing*, vol. 9, no. 5, pp. 889–896, 2000, ISBN 1057-7149.

[14] K. Zuiderveld, "Contrast Limited Adaptive Histogram Equalization," *Graphics Gems IV*, pp. 474–485, 1994, Last Accessed: December 2011. [Online]. Available: http://tog.acm.org/GraphicsGems/

[15] P. S. Heckbert, Ed., *Graphics gems IV*. San Diego, CA, USA: Academic Press Professional, Inc., 1994.

[16] *ChipScope Pro 10 User Guide*, Xilinx Inc, Last Accessed: December 2011. [Online]. Available: http://www.xilinx.com/support/documentation/dt_chipscopepro_chipscope10-1_userguides.htm

[17] Core generator. Xilinx Inc. Last Accessed: December 2011. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/PlanAhead_Tutorial_RTL_Design_IP_Generation_w_CORE_Generator.pdf

[18] *Xilinx XtremeDSP DSP48 Slice User Guide*, Xilinx Inc, Last Accessed: December 2011. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug073.pdf

[19] *Avnet V4FX12LC Development Board*, Avnet-Memec, 2007, Last Accessed: September 2008. [Online]. Available: http://www.em.avnet.com

[20] *LeCroy 6100A Series Scope*, Le Croy, 2007, Last Accessed: September 2008. [Online]. Available: http://www.lecroy.com

[21] *PTH04070W Voltage Regulator*, Texas Instruments, 2007, Last Accessed: December 2011. [Online]. Available: http://focus.ti.com/docs/prod/folders/print/pth04070w.html

[22] *Virtex 4 FX12 Data sheets*, Xilinx Inc, 2007, Last Accessed: December 2011. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-4_data_sheets.htm

Technical Report 2:

Establishing Asynchronous Circuits on FPGAs

Author:	Phillip David Ferguson
Academic Supervisors:	Dr Aristides Efthymiou, Univ. of Edinburgh
	Dr Ahmet Erdogan, Univ. Of Edinburgh
Industrial Supervisor:	Andrew Parmley, Thales Optronics Ltd

3.1 Aims and Introduction

The initial aim and motivation for this work was sourced from an industrial perspective. Asynchronous circuits in general have been beginning to gather momentum, demonstrating better performance for niche areas and applications. Some circuit styles are extremely fast and data-driven, demonstrating lower latencies and greater throughput than their synchronous counter parts. Other styles are extremely low power, providing inherent fine grain clock gating. However, these advantages only target high volume ASICs and general purpose processors rather than lower volume products that apply to niche applications such as laser range finding, thermal imaging systems and vehicle based networks as offered by Thales Optronics Ltd.

The question was then asked, are these benefits transferable to FPGA devices which are much more flexible and affordable for niche companies? Can asynchronous design styles exploit greater throughput and reduced latency, or can they offer power benefits for current widespread synchronous FPGA devices? There is an additional motivator in allowing asynchronous circuit designers the same prototyping freedom to test asynchronous circuits on working silicon without the expensive masks etc.

This document describes the challenges and fundamental issues in creating asynchronous circuits on a FPGA device. This discussion is then furthered into a theoretical overview of asynchronous logic and an evaluation of current asynchronous EDA tools. These design styles and tools are then evaluated on their practical implementations and an appropriate style selected. A design flow is then proprosed the methods to create asynchronous components on FPGA fabric tested. A working asynchronous pipeline is then tested and verified, ensuring that there is a firm footing for subsequent work.

3.2 Challenges and Motivation

3.2.1 Previous Contributions

Previously there has been very little crossover in terms of utilising FPGA resources for asynchronous constructs. There are fundamental problems with the ability of the FPGA fabric and the development tools to support the necessary principles of asynchronous design. So much so that research to date has taken the view of re-designing FPGA fabric for asynchronous design rather than attempting to persuade and constrain current devices into housing asynchronous structures.

There have been 3 main approaches to producing reconfigurable asynchronous circuits:

1. Redesigning FPGA architectures specifically for asynchronous logic (and in some cases both synchronous and asynchronous logic [33, 50, 36, 40]), which is still in the academic research environment and designs have been mostly based on derivations of synchronous devices.

2. Redesigning FPGA architectures in an asynchronous style to implement faster synchronous

designs [56, 23]. Mapping synchronous logic to asynchronous structures may not map efficiently and there has been no indication that the core asynchronous fabric on these devices can be utilised for asynchronous design.

3. Mapping asynchronous logic to synchronous FPGA fabric [34, 44, 45, 61]. This has been treated with caution due to the overhead in attempting to make components within a clocked device and their inherent hazards conform to the hazard free design requirements of asynchronous logic design. Thus there has only been a few selected publications which document these attempts.

3.2.2 Fundamental Issues

Historically there have been a few crucial limitations that have prevented FPGAs taking on the prototyping flow and providing a reconfigurable design platform that they have done for synchronous design. These include:

1. Hazard free operation - in a synchronous circuit the clock edge indicates that data or control signals are stable and valid at that moment in time. In between clock edges, data or control signals are assumed invalid and still settling according to their inputs, thus spurious signal transitions are acceptable up until the clock edge. Asynchronous circuits on the other hand define their own validity schemes so spurious signal transitions would upset and corrupt these validity periods. Asynchronous circuits are thus composed of components that do not produce hazards or they are designed in such a manner that hazardous situations are removed from the circuit. The synthesis stage of the FPGA design flow decomposes and translates the asynchronous circuit from a gate level implementation to a look-up table based implementation. Synchronous synthesis tools have no concept of how hazards affect asynchronous circuits, and so the synchronous synthesis procedure will introduce hazards that were not present in the initial design.

2. Signal ordering - since there is no clock present to sequence and synchronise signals, an asynchronous circuit relies upon the ordering of signals to function correctly. Bundled-data circuits require completion detection (from some sort of delay matching approach) to assert the request signal and delay insensitive circuits require isochronic forks that make the assumption that all recipients from a fan-out receive the signal event. Delays and matched routing delays are crucial in maintaining signal ordering regardless of the design style and delay model. The placement and routing stage of the FPGA design flow is usually targeted to find the shortest path with respect to a cost measure. In the case of delay matching it is highly unlikely that the same algorithms can find the shortest path that has a delay greater than a certain value. Often routing delays are greater that the logic delays and so meeting ordering requirements usually requires more resources than routing lines.

3. Arbitration - sharing resources in a synchronous circuit usually involves a time division of the resource to its suitors. However in an asynchronous circuit there is no discrete notion of time and so arbitration must be performed in another way. A mutual exclusion element (MUTEX) is required to process two handshake requests to access a single resource. The MUTEX will block the latter request from proceeding with its handshake until the first request has been de-asserted and the first handshake completes. A problem occurs if both requests appear simultaneously. The MUTEX must make an arbitrary decision on what request proceeds, inevitably risking metastability. This decision is fundamentally based on analogue electronics. In ASICs, cross coupled NAND gates form the simplest solution, however since FPGA devices only house digital logic elements the possibility of providing arbitration that is correct and hazard-free (yet again) is very difficult, resource hungry [45] and normally has fixed priorities [58] to reduce the time for metastability to resolve.

In recent times there have been a few research projects [52, 43, 41] that have demonstrated asynchronous FPGA implementations of small 5 stage pipeline RISC processors. Marshall [41] began from an asynchronous gate level description and converted each gate into the equivalent Xilinx primitive block. This netlist could then be placed and routed on any FPGA. Although this approach was successful, it is impractical to design at the schematic level as it left the designer with significant work in accurately determining matched delay elements and adjusting the sources schematic accordingly. Desynchronization [52] is a conversion flow that takes in synchronous designs and replaces the clock tree network with a series of asynchronous control blocks. Although demonstrated in an FPGA (and an ASIC) it is unclear how many resources were taken up by the asynchronous constructs. The amount of place and route constraint information required to maintain the asynchronous structure indicated that there was a significant amount of development time spent tweaking timing on chip just as with Marshall's effort. Although both projects are modest in their achievements, both used a reasonably similar approach of creating a primitive translation to convert ASIC components into the equivalent gate netlist suitable for FPGA synthesis. From this point various circuit specific timing and routing constraints could be applied to this netlist, guaranteeing operation for the chosen device.

The methods presented in this document are squarely aimed at creating a consistent repeatable implementation that does not need tweaking to operate correctly. This unique approach does not involve a gate level intermediary stage but targets FPGA primitive components directly from a high level description, side-stepping some of the fundamental issues previously discussed. The proposed flow makes use of current synchronous tools to optimise packing combinatorial logic into look-up tables. This allows asynchronous circuits to be implemented consistently on synchronous FPGA devices.

3.3 Technical Background

3.3.1 Synchronous Logic

A simple logical or arithmetic operation involves a set of stable inputs that are passed through a cloud of combinatorial logic. Depending on the information/data presented by these inputs, the corresponding outputs are produced. When simple operations are connected to produce more complicated functions methods to control synchronisation, data flow and determine operation completion are required. In a clocked synchronous system simple operations are partitioned by state holding elements (latches or registers) that allow these control operations to act simultaneously on an event, a global clock. This external timing reliance prevents data being destroyed or corrupted.

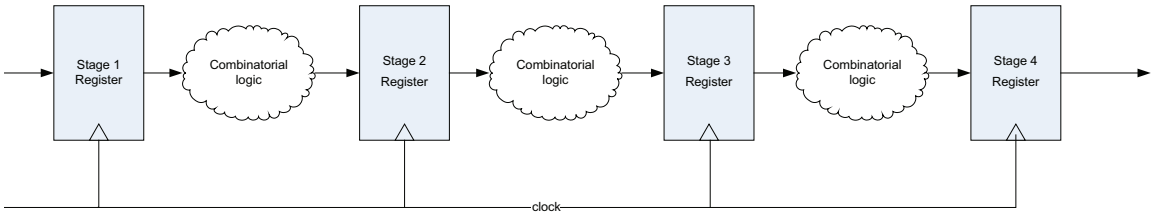


Figure 3.28: 4 Stage Synchronous pipeline

Figure 3.28 shows a pipeline structure where each combinatorial operation is contained between two registers. On each clock tick, data is passed from the outputs of one stage to the inputs of another stage. Each stage can only hold one data entry, thus the function is divided temporally and spatially. The more important consequence of this pipelining is that the throughput of the function has increased at the expense of latency due to the addition of registers. The period of the clock is assumed to be larger that the time taken for the slowest combinatorial operation to complete. Thus the speed of the pipeline is governed by the worst case delay between registers. This is evident in the timing diagram in Figure 3.29, where the different colours indicate the time difference between the next clock edge and the time taken for the operation on that piece of data to complete.

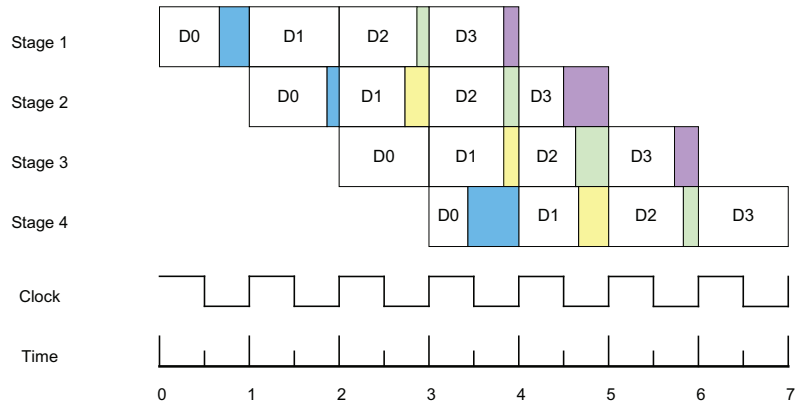
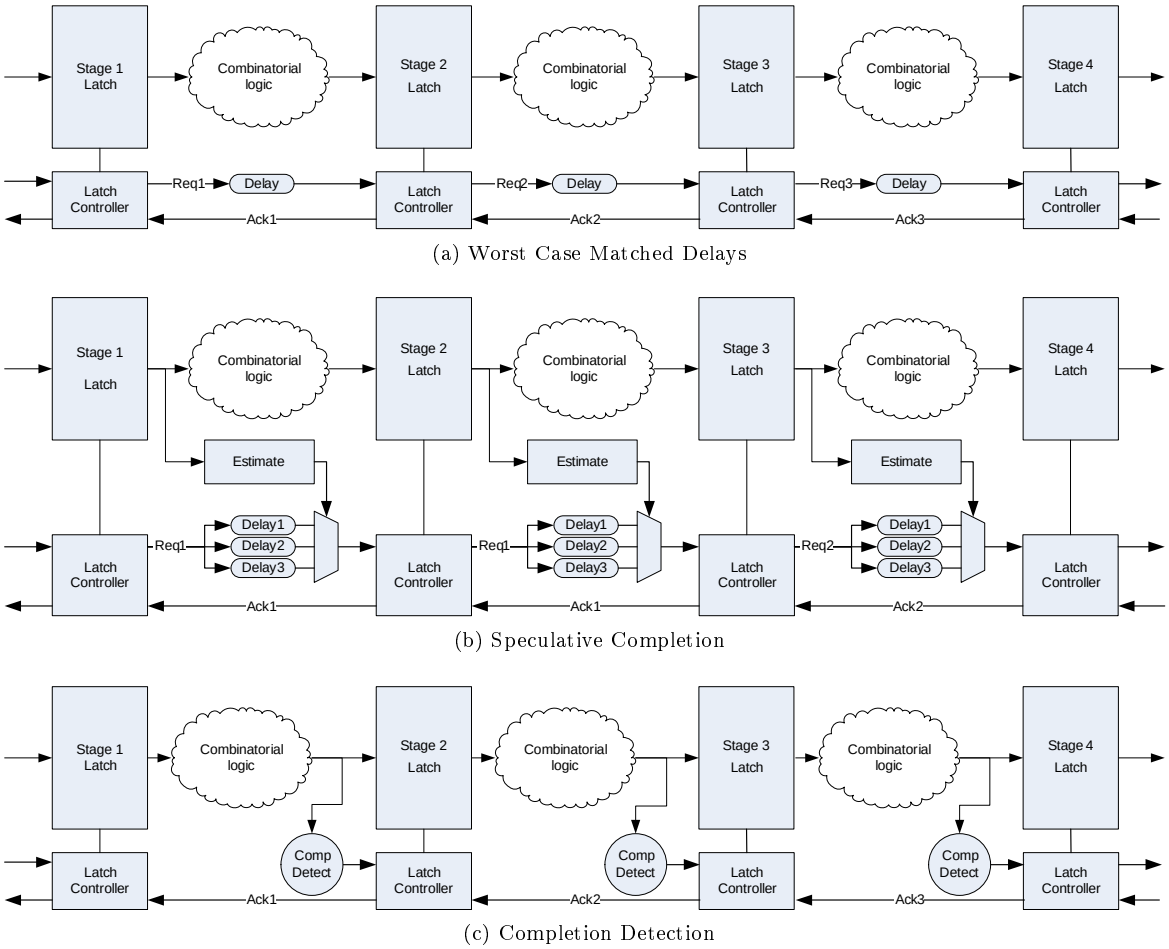


Figure 3.29: Synchronous Pipeline Occupancy



(a) Worst Case Matched Delays

(b) Speculative Completion

(c) Completion Detection

Figure 3.30: 4 Stage Asynchronous Pipeline

3.3.2 Self-timed Logic

Self-timed or asynchronous logic is another approach to provide control mechanisms that coordinate data flow and provide stage completion detection. Without a clock signal to provide timing assumptions, each stage is timed according to its individual propagation delay and so communication mechanisms must take into account that variability. There are a number of asynchronous communication protocols which provide the necessary control information and completion detection between stages [53, 42]. In general each protocol has a mechanism to provide each combinatorial block with a data ready or request signal that indicates that valid data is ready at the inputs. The combinatorial block then processes the data and indicates to the successor stage that processing is completed and valid data can be transferred onwards. An indicator (or acknowledge) to the predecessor stage must also be produced to indicate that the output data has been stored and input data can be changed. This handshake will always enclose the transfer of only one data element.

Firstly stage completion is addressed: there are 3 options to ensure data at the output of a stage is valid.

1. Matched Delay: this assumes that there is an event on the request line simultaneously as new

data is presented at the combinatorial inputs. Shown in Figure 3.30a, the request line is separate from the combinatorial block and passes through a delay equivalent to the worst case delay through that individual combinatorial block. Usually this delay is a stream of gates that can react to process technology and environmental variations in a similar manner as the combinatorial block.

2. Data Selectable Matched Delay [49, 48]: here multiple matched delays (shown in Figure 3.30b)are produced to reflect different operations of the combinatorial block that have different propagation times. The appropriate delay for the request line can be selected depending on a speculative estimate of the data (through additional logic) or an internal event to indicate the required operation.

3. Delay Insensitive Codes[4]: the most accurate indicator of stage completion (shown in Figure 3.30c) is to have the combinatorial block indicate data validity and thus process completion at its outputs (explicitly shown in Figure 3.30). This is accomplished by using data codes to represent validity and neutrality, such that code words in between neutral and valid signify that the process is not complete or has not reset.

The handshaking mechanism can reliably transfer data whilst making no delay assumptions in the transmitter or the receiver. This mechanism provides a number of advantages and disadvantages over the clocked synchronous mechanism. From the timing diagram in Figure 3.31, the first marker to make an impact is the speed. With each stage completing its operation as fast as possible, each stage does not take the worst case delay to complete as assumed in the synchronous implementation.

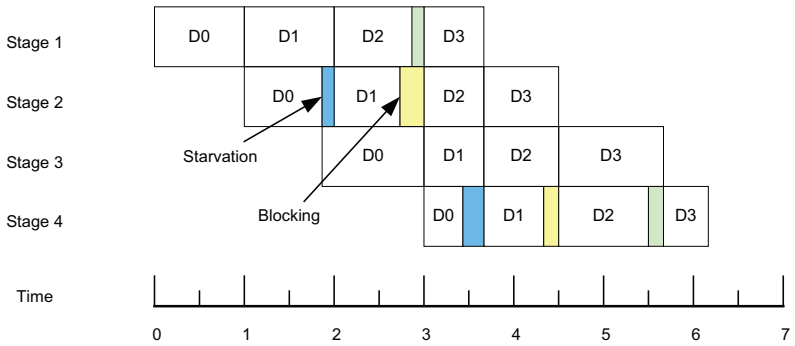


Figure 3.31: Asynchronous Pipeline Occupancy

There are however two different stalls in the system that were dealt with in the synchronous version. The first stall is due to stage starvation. Here stage 1 has not completed its operation before stage 2 is ready to accept data and so stage 2 is forced to wait until valid data is available. The second stall is due to stage blocking where data is trying to move to stage 3 but is unable to because it is still processing the previous data word. In pipelines with too few data elements starvation is common and

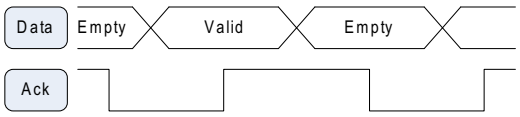
[4]Further details of delays insensitive codes will be covered in the following sections

the throughput is naturally low. The opposite also holds where in the case that there are too many data elements the pipeline with blocking causing a high latency.

3.3.2.1 Data Level Encoding

If Boolean encoding is used to build combinatorial blocks in a similar manner to synchronous design there is no way to accommodate completion detection. This also means that explicit request and acknowledge handshake signals will act in parallel to the combinatorial logic. However in order to guarantee a certain validity region, handshake signal and data signal order must be preserved. In order to do this delay matching is required. The first event that could represent data validity (or completion detection) is the request line, and so the worst case delay through the combinatorial block must be duplicated through the request line. This is the key element in bundled-data (or single-rail) protocols.

If data signals through combinatorial blocks use two wires per bit of information communicated, a code word could be used to represent data validity, negating the need for an explicit request line. In Figure 3.32, two bits with 4 possible values are used to represent an empty value, a valid '0' and a valid '1'. Any codeword in between these values is deemed to be invalid. The main benefit of this encoding scheme is that it has inherent completion detection, and so a matched delay line is not required. This encoding scheme is named dual-rail and forms the simplest of a set of delay insensitive codes [57].



value	Empty	"0"	"1"	Not Used
*wire*0	0	1	0	1
wire1	0	0	1	1

Figure 3.32: Simple Dual-Rail Encoding

To expand further, 1-of-N codes (also referred to as one-hot) uses one wire for each value of data. Where this becomes beneficial is in the number of wires per number of values of data and the number of transitions (hence speed, power and energy) of changing between valid data values.

In Table 3.9 we show a 2 bit data word that has been encoded using dual-rail and 1-of-N. For an N-bit data word, dual-rail uses 2*N wires, in the case of 2 bits, this means 4 wires are required. If the bits of the original word are paired and each pair is 1-of-4 encoded,this coding also requires 2*N wires but only requires N transitions versus 2*N transitions in the assignment of a valid value. To take delay insensitive codes even further 1-of-N codes are a subset of k-out-of-N codes, which instead of using one true bit out of N code bits, k true bits are used to represent a valid code value. The maximum number of valid values for a given N is obtained by choosing k as N/2.

value	Empty	“0”	“1”	“2”	“3”
wire0.0	0	1	0	1	0
wire0.1	0	0	1	0	1
wire1.0	0	1	1	0	0
wire1.1	0	0	0	1	1

(a) Dual Rail Encoding

value	Empty	“0”	“1”	“2”	“3”
wire0.0	0	1	0	0	0
wire0.1	0	0	1	0	0
wire1.0	0	0	0	1	0
wire1.1	0	0	0	0	1

(b) 1-of-N Encoding

Table 3.9: 2-bit Data Channel Encoding Examples

Delay insensitive codes are unfortunately limited by practical requirements that reduce their usage to dual-rail or 1-of-4 encoding. Firstly, validity tests need to be simple and resource efficient, secondly the encoding and decoding of data words must also be simple and efficient, thirdly the overhead in terms of the number of bits used for a code word versus the number of bits used for a data word should be minimal.

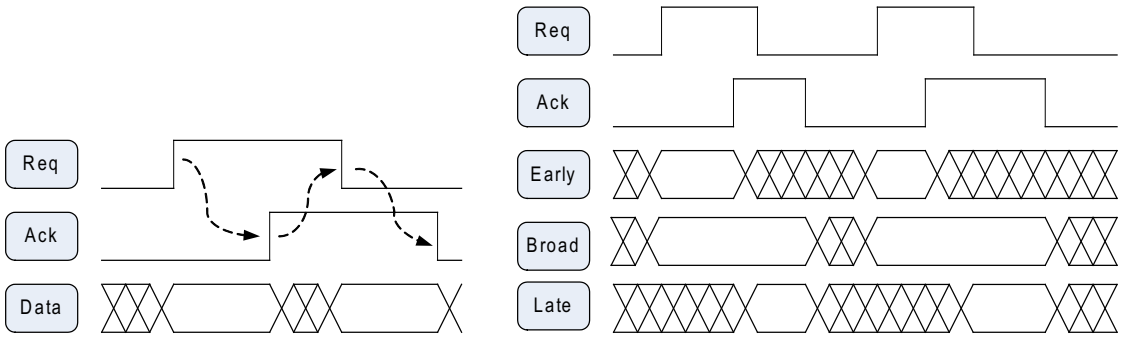
3.3.2.2 Signalling and Validity Regions

The crucial construct to handshake protocols is the signalling. There are 2 common types that have been around since the earliest days (1950's) of asynchronous design. For simplicity examples use bundled data encoding where there are explicit request and acknowledge signals running between the source and destination latch controllers. A communication channel includes the data word and the request and acknowledge signals. If a handshake is initiated by the data source, it is defined as a push channel, where as if the handshake is initiated by the destination of the data, it is defined a pull channel.

2-phase signalling encodes information on the request and acknowledge signals as transitions, so regardless of the voltage level change, $1 \rightarrow 0$ or $0 \rightarrow 1$ both represent a signal event. Figure 3.33(a) shows the behaviour of this protocol on a push channel. Firstly with valid data now available there is a transition on the request line, the receiver then responds with another transition to acknowledge the valid data. Since now the request and acknowledge lines will be in the same state another transaction may begin. Despite being fast in terms of protocol events, implementing circuits (registers, latches and their controllers) that respond to signal events is complex and so very few have ventured down this road[53].

4-phase signalling (shown in Figure 3.33(b)) on the other hand is much easier to implement as transactions are level based. It allows construction of latches and their controllers to be much simpler at the expense of a slightly slower signalling protocol. Here a push transaction is initialised by the request line going high, the acknowledge responds accordingly. At this point the sender pulls request low to indicate that it has received the acknowledge and accordingly the receiver resets the acknowledge line.

There is a range of data validity schemes that are possible with 4-phase signalling. The early



(a) 2-Phase Signalling (b) 4-Phase Signalling with validity regions

Figure 3.33: Signalling and Validity Regions

scheme is used when the source has control of a data bus (push channel), placing data on the bus and issuing a request. When the destination has control of the data bus (pull channel) the late scheme is used. Here the request line issues an access request to the data bus. The broad scheme is the generic scheme that will work for both early and late schemes, allowing compatibility between the two. There are other validity schemes that can be created between any event on the request and acknowledge signals, however there are usually targeted for a specific purpose for examples speed or signal transitions.

3.3.2.3 Delay Models

Delay models are normally a trivial consideration for synchronous designers. Outputs from combinatorial logic only need to be valid and stable around clock edges. As a result delay models are relatively simple, e.g. the inertial delay model used in VHDL that defines a delay time and a reject time. In asynchronous design delay models are crucially linked to the resolution of hazards in a circuit.

Hazards are the result of input changes and delays through wires and gates in a circuit. In synchronous design hazards are allowable outside the clock edge regions, in asynchronous design signals must be valid all of the time. Thus, delay models are crucial in filtering out and interpreting hazards, and so make assumptions about the circuit environment and the delays for wires and gates.

Asynchronous circuits that are modelled at the gate level can be classified as being speed independent, delay insensitive or quasi-delay insensitive. These classifications are dependent on the delay assumptions made for the circuit to operate correctly.

Speed -Independence (SI): David Muller's speed independent classification [46] models gates as boolean operators with a non-zero delay. Wires were modelled as ideal. A circuit is then described as a set of concurrent boolean functions. The state of a circuit is the set of all gate outputs. A gate is described as stable if the output is consistent with its inputs. A gate is described as excited if one of its inputs have changes and an output change is required. After an arbitrary delay the gate will fire,

i.e. the output will change to be consistent with its inputs. To describe the dynamic operation of a circuit a state graph of all the possible firing sequences can be constructed.

A circuit is described as speed independent if a gate that has been excited by an input condition still stays in this excited state if that input condition changes before the gate has fired. Since gate delays are unknown we need to assume that a gate will always fire on a correct input condition. An excited gate that does not fire is a potential hazard.

Delay-Insensitive (DI): Delay-insensitive circuits take the speed independent assumptions and expand them to include realistic wire model delays. In assuming wires were ideal speed independent circuits infer that the output of a gate will always appear on the inputs of connected gates. Since DI circuits have arbitrary wire delays the only way to determine that a gate has seen a change on its inputs is by a change on its output, i.e. the property of indication, where every transition must be acknowledged by another event.

Delay-insensitive circuits will operate correctly if gates and wires have unknown positive bounded delays. In this model all transitions have to be acknowledged before transitioning again. This robustness suits communication mechanisms like handshakes very well, however applying this to a gate is impractical as this would mean that each gate would need to have the property of indication. There are only two gates whose inputs can be determined from their output state, the Muller-C-element(see Section 3.4 for details of this component) and the inverter. This means that circuits which compute cannot be realised [53] as the common AND, XOR gates do not indicate their inputs for every output state.

Quasi-Delay-Insensitive(QDI): There is an exception to DI circuits that use wire forks with equal timing delays on each fork. These isochronic forks allow one acknowledgement for all branches of the fork, the assumption being that if one branch has seen a transition and all branches have equal delay then all other branches have seen the transition. Quasi-delay-insensitive circuits require all isochronic forks to have at least one acknowledgement. Most circuits that compute are classified as QDI.

3.4 FPGA Implementation Considerations

As mentioned briefly in the previous section there are a few components that are fundamental to the operation of asynchronous circuits. This section will discuss a few of those and the role they play in maintaining correct operation in an asynchronous circuit.

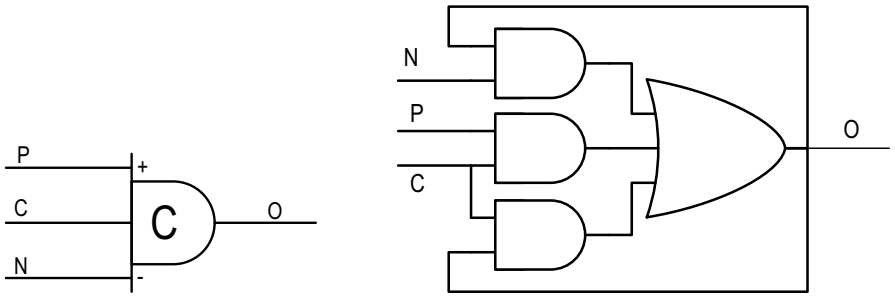


Figure 3.34: Muller C-element

3.4.1 Asynchronous Component Challenges

Muller C-element: A Muller C-element is a fundamental gate that is used in all asynchronous components. Through the indication principle it is able to merge and synchronise signals at its inputs by only changing its output only when both inputs are in the same state. If the inputs are in different states the output will remain the same. Since the indication principle is only partially true for certain inputs of AND and OR gates, the C-element can be built from these basic gates. If the output of an OR gate is '0' then we can determine that for this output state to be correct both of its inputs must be zero. Similarly for an AND gate, if the output is '1' we know that for this output state to be correct both its inputs must be '1'. An asymmetric (or generalised) C-element (shown in Figure 3.34 as a gate and logical decomposition) takes into account an additional two inputs that are biased towards pulling the output high or low. If the positive biased input (P) is high when the common terminal is high the output will go high regardless of the state of the negative (N) input. Conversely the output will go low only when the negative biased input is low and the common terminal(C) is also low. Implementing these devices optimally is crucial in maintaining circuit performance.

A Muller C-element can be implemented in an FPGA in two ways: a set/reset latch structure [34] using two look-up tables and a latch for driving the output net or a singular look-up table with feedback [58] from the output to maintain a consistent value when the outputs do not change in a manner to provoke an output change . The compromising choice is between greater resource usage and greater routing delay in the case of the feedback option. Deciding between implementations will be dependent on routing constraints for delays and synthesis tools to optimise as much logic as possible into the set/reset conditional look-up tables.

Latch Controllers: The structure enclosed below the level sensitive latches in Figure 3.35 is responsible for the transparency of the latch and the coordination of the various handshake signals on each side of the latch. A significant amount of time and research has been spent on the design of latch controllers as their properties and characteristics are crucial in the performance and robustness of an asynchronous system.

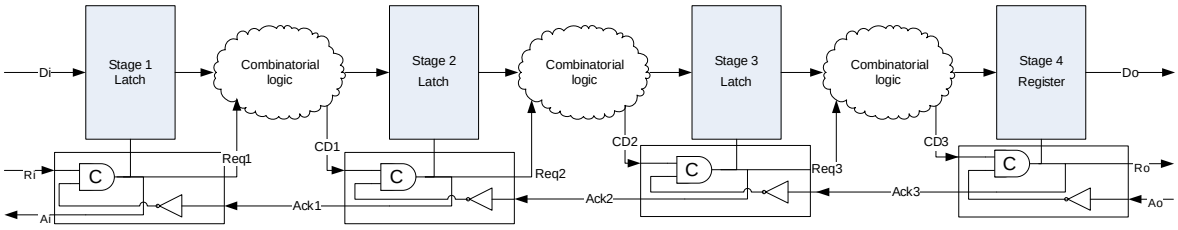


Figure 3.35: Half Latch Controller

The Muller C-element and inverter structure of Figure 3.35 is known as an undecoupled controller [39]. It maintains the sequence of handshakes as well as the transparency of the latch in the simplest manner possible. There is however a drawback to this simplicity. New data can only be latched on Ri↑, when the output handshake link has completed with Ao↓. With this in mind, only 50% of the pipeline stages shown in Figure 3.35 will be holding valid data, the remaining 50% will be transparent and in the return to zero state of a handshake, i.e. 50% of the pipeline capacity is lost.

A semi-decoupled latch controller relaxes the requirement of not initiating an input handshake until the output handshake is complete and allows new inputs after Ro↓. This also means that Ai↑ can be produced independently of the handshaking on the output channel. With the undecoupled latch controller both handshakes link through Ro and Ai, where as with the semi-decoupling, this link is altered to Ao↑ and Ai↑.

A fully decoupled latch controller relaxes the handshake inter-dependency even further by allowing new inputs to be latched after the output link has acknowledged that it has latched the current data, i.e. when Ao↑. This means that after this event the input handshake can be completed without any further interaction from the output link.

All of the above latch controllers have inherently made the assumption that the latch itself is normally transparent. This property means that data will flow through combinatorial blocks regardless. In this case, processing is completed immediately regardless of when it is required. This has power implications as the number of transitions will increase dramatically, however it will allow the pipeline to perform at its quickest. Conversely if the latches are assumed to be normally opaque then superfluous transitions in the combinatorial blocks will not occur and dramatic power savings will be made [38] in a trade off with performance.

Designing controllers means making assumptions about their environment and consequently internal operation. There are two general assumptions that support two design formalisms.

Input-Output mode assumes that a circuit is in a stable state, and only at in this state can the environment change its inputs. When the circuit has produced its corresponding output the environment can change the inputs again. There are no assumptions about the internal signals and so a subsequent input change could occur before the internal signals and other outputs have stabilised from the previous input change. For a circuit to operate in this mode, all of environment changes must

be described by causal relationships between input and output signals transitions. These circuits are therefore speed-independent and can be built using a number of design methods which are documented later.

Fundamental mode assumes knowledge of the stabilisation time (not the state of the internal signals) of a circuit. In this context the environment can only change one input and must wait for the longest delay in the circuit before changing another input. Burst mode is a generalisation of fundamental mode, allowing multiple input changes in bursts, but maintaining fundamental mode between these bursts.There is also a number of synthesis options [60] for this design style that will be consider in later sections.

Implementing controllers in a FPGA is a crucial balance between resource usage, propagation delay, and the ability of the FPGA architecture to lend itself to controllers that minimise both these aspects. This means that the relative position as well as the number of primitive components (look-up tables, latches, registers, multiplexers, XOR gates, etc) used in a controller is critical in evaluating a suitable controller that optimises its overhead and contribution to the critical path of a circuit.

3.4.2 Design Tool Considerations

This section aims to discuss the practicality in using a number of design tools that could aid in implementing asynchronous circuits on FPGA devices. These tools are classified into two domains that are based on their initial behavioural descriptions, syntax directed compilation and transition systems. This has implications for the choices in terms of a behavioural description language, a consistent synthesis system and quality of the circuits implemented on an FPGA. These tools are considered in the context of their ability to: support bundled data circuits, have multiple design templates, ease of use, circuit complexity capability, have a route to integrate into current FPGA design flows. There is an inherent assumption that to program a FPGA and optimise circuits for specific device architectures vendor place and route tools will be required. It is therefore important that any higher level design tools must integrate into these vendor specific tool flows.

3.4.2.1 Syntax Directed Compilation

Timeless Design Environment (TiDE)

Handshake solutions [27] are one of a few companies that have created a sustainable business model[5] from their tool flow, TiDE. Their contribution to asynchronous design is significant as they provide an entire design suite to augment current ASIC design flows. This involves using dedicated tools for asynchronous specific tasks such as compilation and synchronous tools for standard tasks, such as logic optimisation and timing validation. Re-using industry standard synchronous tools requires

[5] Unfortunately this changed in early 2011 when Handshake solutions ceased to exist due to market forces.

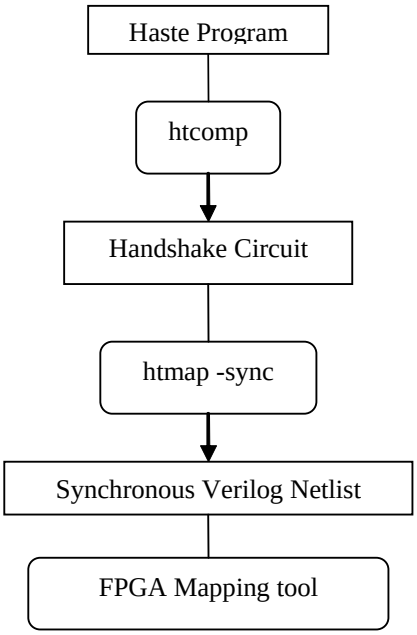


Figure 3.36: Handshake Solutions Prototyping Design Flow

strong constraints but keeps a familiarity to synchronous design for new asynchronous designers.

The design specification begins from a high level language called Haste (originally Tangram) which describes circuits behaviourally. Originally based on CSP and Occam, Haste provides constructs for sequential and parallel execution as well as communication (handshakes) and hardware sharing constructs. The aim of Haste was to describe the behaviour of circuits, hiding the underlying asynchronous design and allowing the compiler to generate handshake protocols via syntax directed translation. This has the potential advantage of avoiding hazards and glitches. The handshake circuit is constructed from a library of approximately 50 components that map into concise, hardware circuit descriptions. The resultant (Verilog) netlist is then optimised and mapped onto an implementation of standard cells.

The resultant circuits are implemented with a low power priority over performance and so crucial restrictions are placed in the finalised design. Handshake solutions use a 4 phase bundled data approach to implement all handshake circuits. This has its benefits in ASIC design but can become particularly troublesome to implement on an FPGA. Four phase protocols are widely accepted as the lowest power protocol where each handshake has a return-to-zero phase. A bundled data approach aligns the request signals with the output from a combinatorial block by using a matched delay. This is not a problem for ASIC implementations however it poses significant challenges to produce a systematic delay line from FPGA fabric that is not automatically altered or optimised out at various stages in the synchronous tool flow.

Handshake solutions provide a specific method for implementing handshake/asynchronous circuits on FPGA devices. However, this is presented as a prototyping flow in Figure 3.36. The problem with this flow is that in order to translate the handshake circuit correctly (so that a mapping tool

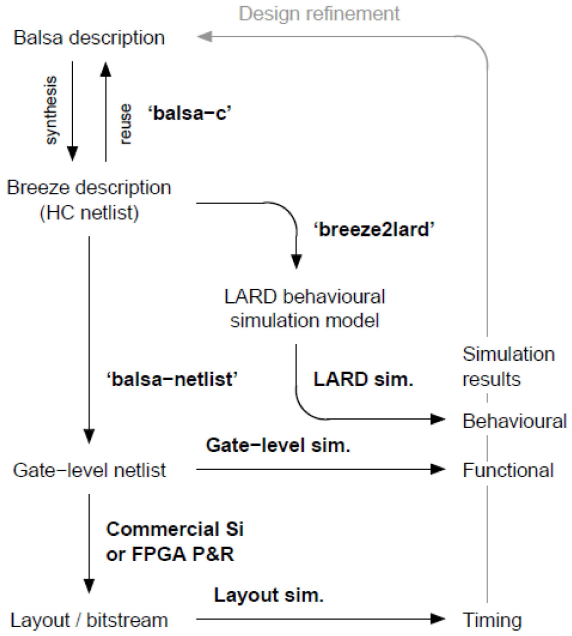


Figure 3.37: Balsa Design Flow [25]

can allocated the appropriate device components) the htmap program decomposes the circuit into primitive behavioural verilog cells then inserts synchronous registers and adds a clock to the entire design. The result is a synchronous verilog netlist that can be mapped by any commercial FPGA tool suite. This guarantees correct functionality but does not represent a true asynchronous circuit any more.

Balsa

Balsa [25] has very close links with the TiDE design flow (see Figure 3.37), however being academically developed it is completely open source and thus offers increased flexibility in terms of implementation options. Balsa also operates on the concept of handshake components and handshake circuits.This allows a number of customisable features that balsa takes advantage of to a much greater extent than TiDE. The balsa description can be targeted to dual-rail and 1-of-4 encoding schemes (as well as bundled data) which being part of the delay-insensitive codes will not need the delay matching required for bundled data. This could potentially offer a mechanism to explore FPGA implementations of asynchronous circuits. However as described previously, only bundled data allows the exploitation of current FPGA synthesis tools and reduces the potential of hazardous asynchronous implementations by attempting to use synchronous synthesis tools on non-binary data encoding.

The FPGA design flow differs slightly from the TiDE options, where the Balsa netlist is converted into a gate level netlist which can then be directed to the technology specific implementation. Again, like TiDE the netlist is synchronised resulting in a synchronous circuit on the FPGA. However, due to the open nature of the tool this provides significant flexibility in what the FPGA interprets as

asynchronous structures. The Balsa cell library could be altered to reflect hard wired components with defined timing assumptions, providing a more accurate representation of asynchronous circuits. The problem with this approach is that a gate level netlist would have no low level asynchronous optimisations (as employed by TiDE) and it would be subject to a significant amount of synchronous synthesis optimisations without respecting the distinct separation between asynchronous datapaths and control networks.

3.4.2.2 Transition Systems

Petrify

Petrify [29] is a tool developed in the late 90's that allowed asynchronous design automation to take a significant step forward. From an initial behaviour description in the form of Petri Nets[47] or Signal transition graphs (STGs) the tool will produce an optimal concurrent description and synthesise the appropriate asynchronous control circuit. To accomplish this, Petrify performs a token flow analysis that produces a safe irrendundant description. States are then assigned by solving the Complete State Coding problem [37], after which state assignment is coupled with logic minimisation and speed independent technology mapping. The resultant netlist is hazard-free under any distribution of gate delays and multiple input changes in accordance with the initial specification.

The final netlist is made up of primitive gates that can be combined into a look-up table. Merging small gates into larger gates can be accomplished without the risk of hazards [28], (only decomposition introduces new hazards) combining gates into look-up tables which have a fixed delay per logic function. This opens the possibility of applying Petrify to create asynchronous circuits for a FPGA. Speed independent circuits assume zero delay in the wires between primitive gates. Since Petrify only produces Speed independent circuits there are potential hazards in the delays introduced by FPGA routing. There are two possibilities to mitigate this risk: constrain the FPGA routing in such a manner to maintain Speed Independence or utilise timing assumptions within Petrify to define aspects of the environment and the circuit to simplify the circuit and maintain hazard free operation.

Petrify is very capable at designing asynchronous controllers that can be utilised within FPGA fabric. This makes it very suitable for design of the control paths within an asynchronous circuit. However it does not cater for the datapath. This means that an additional tool will need to be selected for this aspect of asynchronous circuits.

BESST

Behavioural Synthesis of Self-Timed Systems (BESST) [32] is a design flow that expands upon Petrify and extend the applicability of Petri-Nets for asynchronous system design. The BESST design flow shown in 3.38 describes the behaviour of a system in Verilog/VHDL and splits the control path and data path for separate synthesis flows. Control paths are synthesised with either: a di-

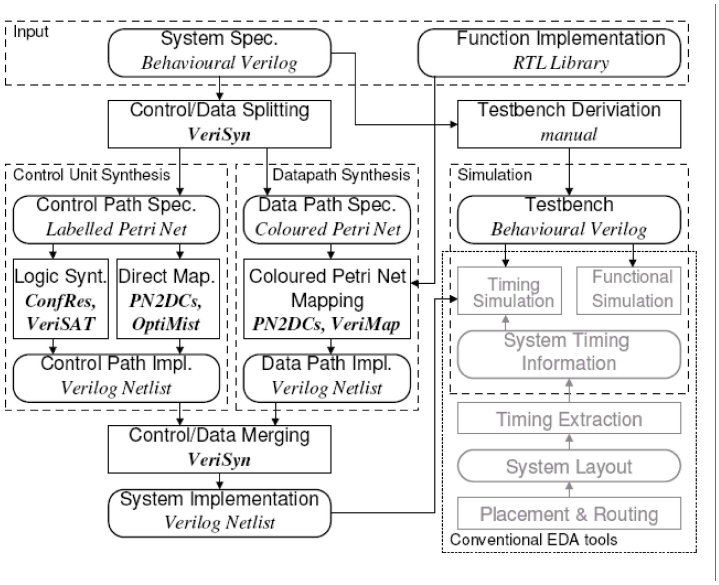


Figure 3.38: BESST Design Flow [32]

rect mapping approach(i.e. syntax directed translation) where Petri-Net structures have direct logic implementations- much like the handshake circuits described in Balsa and TiDE but with a stricter formalism and no control path information, or with an alternative to Petrify (veriSAT [35]) which uses a Boolean Satisfiability approach to avoid the state space explosion seen with creating reachability graphs from STGs. The end result is a synthesis system that is just as effective as Petrify but with significantly less computational costs. Datapaths are synthesised with a colour petri-net mapping approach [51] that partitions the design suitable for a direct mapped translation from Petri-net to logic structures. After synthesis, the two streams are merged into a Verilog netlist that can be passed on to other down stream EDA tools.

From an FPGA context, although very novel in the initial circuit descriptions, much like Balsa, there is no direct mapping (primarily of the data path) to FPGA primitive components. David cells especially, which are used in control path synthesis are particularly difficult to map to a look-up table style fabric. The resourcing overhead required to construct a David Cell from FPGA primitive components is excessive. Considering the potential timing risks from its multiple feedback paths this choice appears unsuitable for current FPGA architectures. Passing the Verilog netlist produced by this flow through a synchronous synthesis tool in order to target FPGA devices would potentially destroy circuit functionality, making it unsuitable for FPGA implementation.

Cascade

The final set of tools which shows promise is the Cascade suite of tools developed by the University of Columbia. These tools have been designed with the objective of design space explorations, and so they offer a few options in specifying asynchronous circuits. Since the focus has been preset on burst mode specifications describing this involves a distinct syntax difference from other HDL's and asynchronous

design languages. Similar to signal transition graphs with restricted transition properties they are exportable to Verilog which can then be directed further down the tool flow. The most promising tool from this toolset is the ATN OPT toolset which optimises dual-rail logic and in turn reduces the hardware cost of implementing completion measures. Supporting VHDL and Verilog, this lends itself promisingly to reducing the amount of completion detection logic required for a FPGA, and so allows the inbuilt synthesis tool to optimise further to suit the architecture. Although this approach shows significant promise for delay insensitive datapath design, the overheads are still too significant and delay prone to suit current FPGA architectures.

3.4.2.3 Summary

It is clear from the time spent on asynchronous tools and the asynchronous circuit theory they implement that the decision on how to integrate asynchronous circuits into a synchronous FPGA is not trivial. There are a series of tradeoffs that must be considered:

i. Using standard HDL languages, from an asynchronous perspective could introduce ambiguity into syntax and the hardware translation.

ii. Using asynchronous tools could possibly introduce debug problems as well as more interventions into the lower level synchronous tool flow.

iii. Determining the asynchronous structures that an FPGA can support may be restricted by the choice of tool/input language.

Implementation considerations must be dependant on the ease of design for asynchronous circuits, the steps necessary to augment the current synchronous design flow and the time taken to learn how to fully utilise the tool. With this in mind the remainder of this text will deal only with 4-phase bundled data circuits. The primary reason for this is to maintain the optimisations performed by the synchronous synthesis tool that are specific to an FPGA device family. The control path can be constructed externally, minimising the disruption to the synchronous design flow. Moreover this approach saves area and power overheads associated with other asynchronous design styles.

3.5 Design Flow Proposal

3.5.1 Standard FPGA Design Flow Modification

The design flow proposal that allows asynchronous circuits to operate on an FPGA is shown in Figure 3.39. The description of the circuit is separated into two synthesis streams, a data path description and a control path description. The datapath is described entirely in VHDL, whilst the control path is also described in VHDL but in a separate file of primitive FPGA components (e.g lookup tables, registers, etc) that require minimal use of a synchronous synthesis tool. The flow then utilises

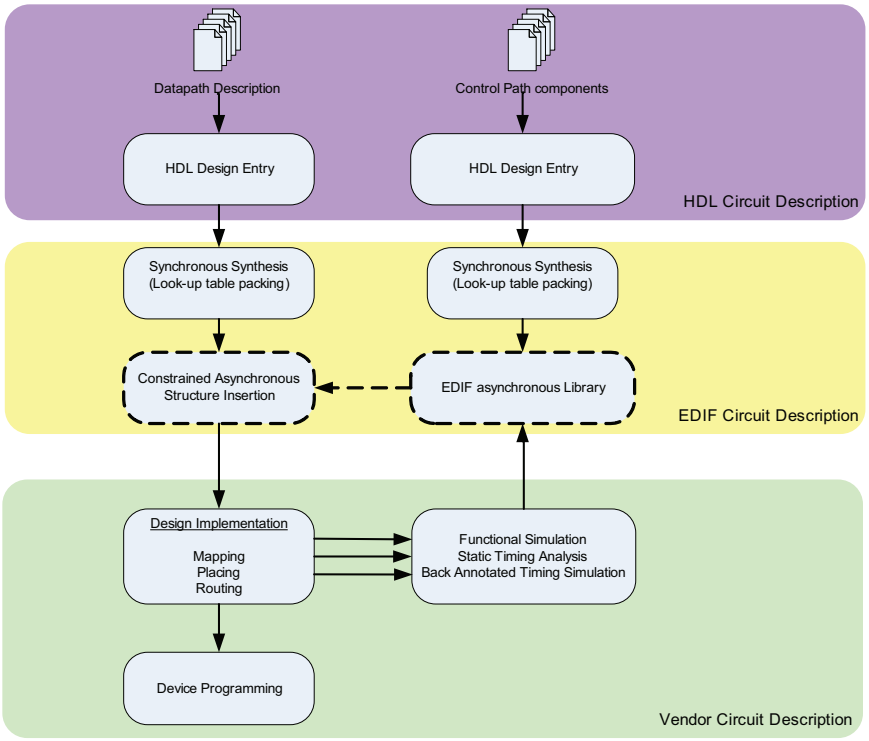


Figure 3.39: Proposed Design Flow

the EDIF [54] netlist as the main focus for modification. In here the asynchronous control path components are combined with data path components. There are a number of reasons why this is the most suitable stage in the standard FPGA design to create asynchronous circuits. Architecturally optimised asynchronous control components can be created independently of the main circuit function, focusing purely on their performance in the device architecture. In the standard FPGA design flow the post-synthesis EDIF netlist will contain optimised boolean encoded logic compacted into look-up tables and other primitive components(adders, multipliers, etc) found on the FPGA fabric. Synthesising the data path in this manner maximises the use of standard design flow tools. This means that hazards are avoided from an asynchronous control path view, and data paths are optimised for the target FPGA fabric. The combined post-synthesis EDIF is then passed to place and route tools with a number of constraints to ensure that optimisations made in the implementation process do not disrupt functional correctness. Assuming that FPGA families share a similar architecture, only vendor or device family specific libraries containing fabric level implementations of control path components are required, i.e. control path components only need to be synthesis once.

As bundled data circuits are the target asynchronous implementation, delay chain matching must occur at a suitable point. On the initial implementation cycle delays are inserted with the same static value to allow placement and routing to be aware of the resources they may allocate in that region. The remainder of circuit will already be compacted into look-up tables, embedded FPGA components[62] and registers. Locking this implementation allows the final place and route cycles to only consider the delay element construction and placement. The feedback flow from place and route

tools alters these delays to reflect the true timing of the combinatorial logic in the datapath, ensuring that the delay chains match the combinatorial delay with sufficient precision.

The following subsections will discuss in greater detail the steps involved in creating the asynchronous control path components and the additional elements required to perform a validation of a simple bundled data circuit on an FPGA, demonstrating the proposed design flow.

3.5.2 Component Construction

It has already been outlined that a library of asynchronous control path components will be used to implement asynchronous circuits on FPGA fabric. In order to achieve this, key components must be tested and verified on a sample FPGA. At this stage, development was restricted to Xilinx devices, specifically the common architectures of the Virtex 2 and Virtex 4 devices [63]. The main reason for this was resource availability and supporting the common usage trend of Xilinx devices within Thales Optronics Ltd. Supporting additional devices would not result in a large deviation of the intended design flow but an increase in the number of libraries to support primitive components specific to other device families. Thus it was not conducive to spend time supporting additional devices.

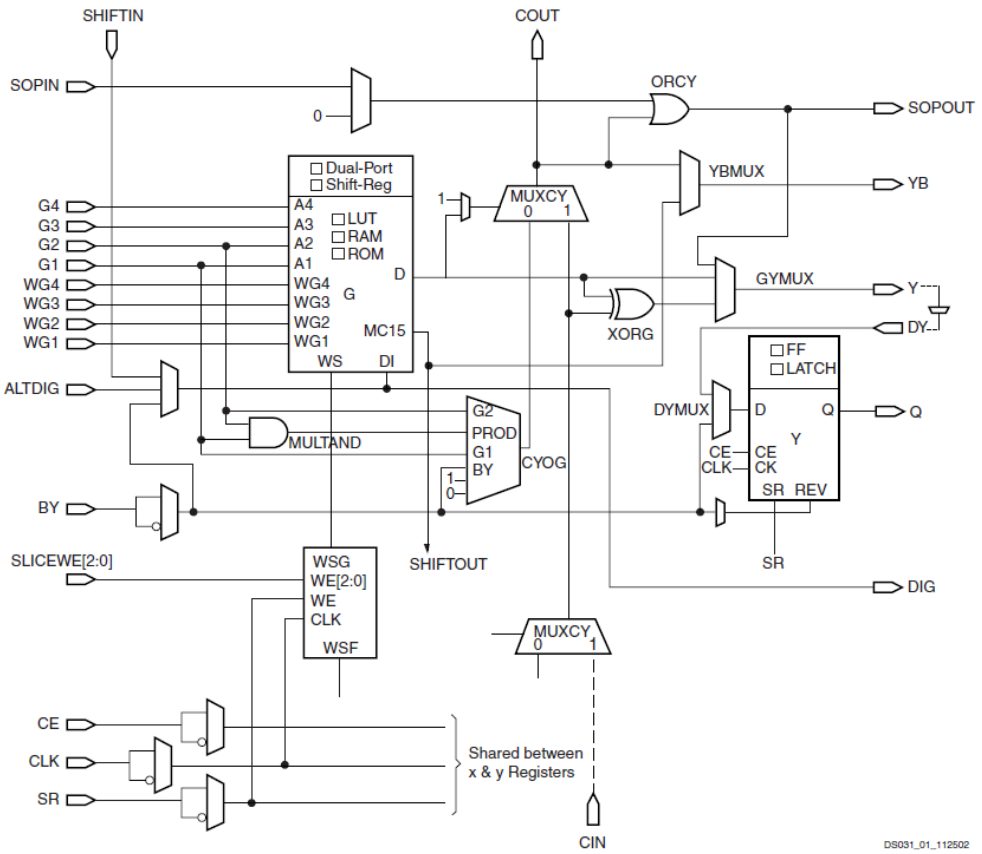


Figure 3.40: Xilinx Slice Architecture [63]

Figure 3.40 shows half of a Xilinx slice. These slices are arranged in blocks of four that form configurable logic blocks (CLB). Constructing circuits from this perspective permits asynchronous

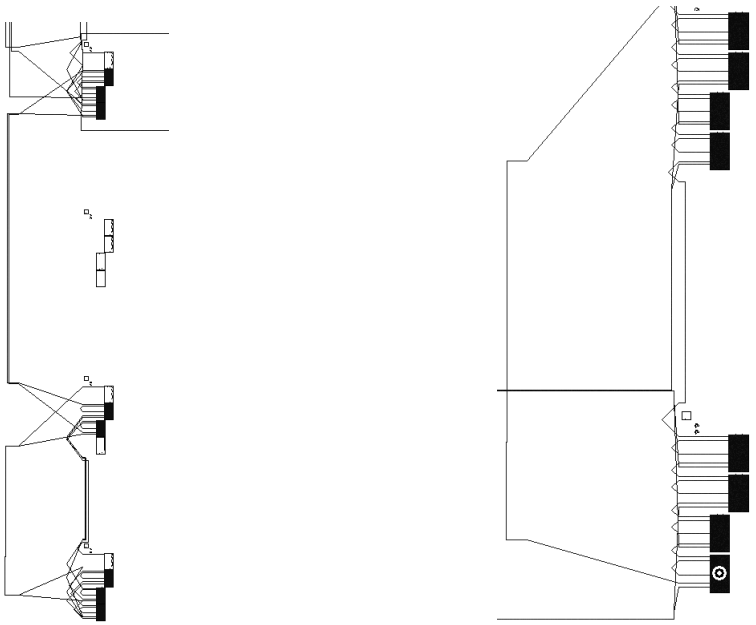
circuits to be constructed using look-up tables, registers, routing matrices and additional slice logic in the same manner as synchronous circuits. A synchronous EDIF netlist will contain a number of cells defined in relation to device primitive components. Cells are instantiated and connected through nets to the ports of other cell instances. These cells make up the hierarchy that has been synthesised from high level descriptions. A bundled-data approach requires several components that must be formed from FPGA fabric. The following sections will describe how these components are constructed.

3.5.2.1 Matched Delays

The simplicity of bundled-data circuits is founded on delay assumptions. This simplicity comes at the expense of data-driven delays, meaning that delay chains are a key performance factor in bundled data circuits. The accuracy of delay chains in representing the worst-case combinatorial delay in a pipeline stage must be managed carefully for both ASIC and, in this context, FPGA circuits as well. Although FPGA devices have a fixed layout there is a large computational requirement in balancing combinatorial delays for synchronous circuits [64]. This requirement is increased again for bundled data circuits as each delay chain may be different from its neighbour depending on what resources are used to construct its matching combinatorial block. Having highly accurate delay chains allows bundled-data circuits to exhibit lower end-to-end latency than their synchronous equivalents.

For each combinatorial block it is imperative that the delays between registers can be firstly determined with a degree of accuracy, and secondly matched with an equal degree of accuracy. Since the FPGA fabric is already laid out, the timing information is predetermined as opposed to ASIC chips that requires process models to simulate this timing information.

Normally ASICs can rely on creating delays by daisy chaining a series of gates, however performing the same technique on an FPGA would occupy more reconfigurable resources and potentially degrade the performance of the combinatorial blocks. As with matched delays in ASICs it is preferable to keep the matched delay close to the combinatorial block it is paired with. Sortiriou [52] uses *'keep'* constraints to prevent synthesis tools removing inverter chains made up of look-up tables. This technique, although valid has been derived from ASIC intentions and not optimised for an FPGA architecture. Minas and Marshall [43] proposed the use of unoccupied carry chain logic from the configurable logic blocks (the XORG gate in Figure 3.40) to build delay chains that could not be optimised away. This freed up the more commonly used look-up tables for circuit design and instead used the carry chain XOR gates which are more commonly used for behavioural looping structures e.g. FOR loops within VHDL. However this technique still left room for significant variation in the distribution of the delay chains relative to their own components and the combinatorial blocks they represent. Without guidance, place and route tools will scatter these instantiated delay chains around the fabric. This introduces further delay within the components of the delay chain, and adds more



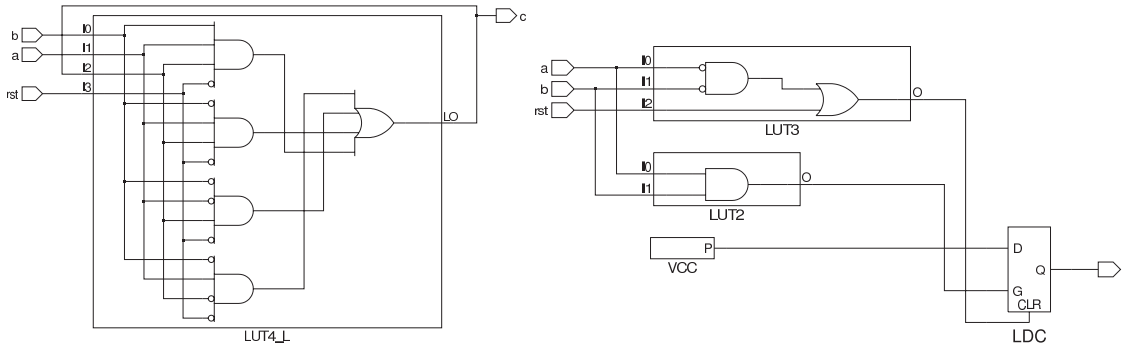
(a) Unconstrained (b) Constrained

Figure 3.41: Delay Chain Variability Reductions

inaccuracy by placing them without any consideration of the combinatorial block they represent.

The novel delay chain structures utilised in this design flow improve upon those previously discussed. Firstly look-up tables are used as well as XOR gates to create delay chains and secondly constraints are applied to limit the divergence of the delay chain components across the device. The impact of this unique approach is shown on an FPGA architecture map in Figure 3.41. The two diagrams demonstrate the impact the appropriate constraints have on the mapping of delay chains and their constituent parts. Figure 3.41a shows the delay chain spanning 16 slices to connect the primitive components, whilst Figure 3.41b shows the impact of constraints to maintain the delay chains in a localised area. The code used to accomplish this is located in Appendices B.1.1 and B.1.2. From VHDL, delay chains of any length can be configured with each component of the delay chain individually assigned constraints in order to be positioned as close as possible to each other. The results of these improvements are shown in Section 3.6.3.

At this stage there is ground for further improvement within the routing of the delay chains. The main justification for creating delay chains based on either XOR gates or Look up tables is delay granularity. These are based on altering the logic used in the delay and constraining the routing. Other studies [59, 55, 26, 31] have shown avenues to control the routing directly with Xilinx Design Language, which allows the lower level routing to be altered before a device is configured with the appropriate bit stream. Although very useful in this context, there is a heavy dependence on lower level tool capability to supply the necessary information to instigate changing the routing configurations at this period in the design flow.



(a) Feedback Implementation (b) Set/Reset Implementation

Figure 3.42: Muller C-element FPGA Implementations

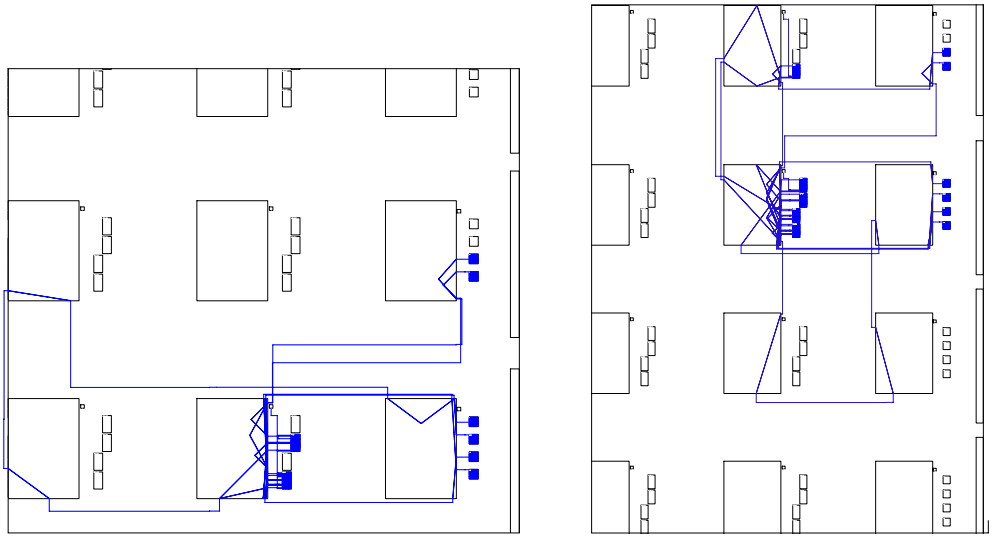
3.5.2.2 Latch Controllers

As with the matched delays, latch controllers must be designed with the target FPGA architecture in mind. Since these controllers will be fully asynchronous, their design must take into account the potential hazards present in FPGA fabric. Synchronous synthesis tools naturally decompose logic for many reasons: re-timing, reducing fan-out, etc. Decomposing logic in asynchronous controllers will produce hazards. It is therefore imperative that logic decomposition is avoided. For this reason, latch controllers have been designed from the primitive components level, i.e. inter-connecting look-up tables and other components shown in Figure 3.40.

In this section latch controllers will be discussed from an implementation perspective. i.e. how gates are allocated into look-up tables and the relevant constraints required to ensure consistent hazard-free behaviour. An additional comparative analysis is performed in the following chapter.

Initially an undecoupled latch controller consisting of a Muller C-element and an inverter was created using two alternate structures [34, 58]. What is important to note about these two implementations is the granularity with which they were designed. Functionality was translated into primitives that could be instantiated in VHDL and arranged in hazard free structures with routing constraints explicitly stated. Figures 3.42a and 3.42b show the gate-level circuit encapsulated by the FPGA resources in constructing the two implementations. The look-up tables have been expanded to show their gate-level contents. From a resource perspective the feedback implementation uses less resources, but at the risk of hazards from the feedback path. We negate this risk by assuming that this controller would be operating in an environment where the feedback path would be faster that the circuit to change the input to the look-up table and potentially create a glitch and hazardous state.

The limitations of an undecoupled latch controller in terms of throughput are well documented [53] so the decision was made to utilise a latch controller with full decoupling at the expense of increased complexity and constraints. The undecoupled latch controller only consumes 1 look-up table. From a resource and timing perspective one CLB was determined to be a reasonable limit for one latch controller regardless of the complexity. The fully decoupled latch controller by Lui [39] could be



(a) Constrained Controller Routing (b) ControllersWithout Constraints

Figure 3.43: Unconstrained Controller Routing

ideally mapped to this architecture. Since the Muller C-element FPGA fabric equivalent had already been explicitly created, adding additional gates to create a fully decoupled variant was a trivial matter in VHDL. However the placement constraints required tuning to ensure that the mapping of look-up tables were properly positioned within the CLB so as to minimise routing delays and consequently the forward and reverse latencies. With a combination of regional location constraints (RLOCs) and basic element of logic location (BEL) constraints, look-up tables can be explicitly positioned to minimise routing delay around routing matrices, much like editing programmable interconnect points but without knowing prior routing information. Figures 3.43a and 3.43b show the impact (much like the delay chains) of how place and route tools will scatter the primitive components without considering the delays associated with their performance and function. In synchronous design, delay constraints organise look-up tables and other primitive components between registers, however in the design of asynchronous controllers, placement constraints are the preferred choice to organise primitive components rather than adding a significant amount of timing constraints to every net in the design.

Describing controllers in such detail in VHDL is not efficient and effectively asking synthesis tools to do additional work in interpreting these structures and potentially changing their functionality based on a synchronous interpretation.The natural description of these controllers is in EDIF where the design is much more connected to the placement and routing directives/constraints. Appendix B.2 provides a small discussion on the differences between the two implementation possibilities.

3.5.2.3 Interfacing to Embedded Components

One of the obvious issues when producing asynchronous circuits on FPGA fabric is the need to interface to dedicated embedded components and external environments. In most cases this may be

embedded multipliers and adders, as well as blocks of RAM and Input/Output blocks. Embedded components can be separated into two groups: arithmetic operations that may not need a clock to perform their intended function, and higher level operations that do require a clock, e,g. RAM blocks. In their synchronous implementations multipliers and adders can be pipelined to improve their performance, however there is no access to these registers and so they must be setup without this fine grain pipelining. RAM blocks on the other hand do require a clock and so must be surrounded by a wrapper to ensure their compatibility with asynchronous circuits.

There are two approaches to the design of such a wrapper:

1. If we assume the wrapper is designed synchronously, the input handshake signals need to be treated as asynchronous control signals and consequently double-registered to minimise the risk of metastability. The state of these handshake signals must be translated into the appropriate control signals for the synchronous block.

2. If we assume the wrapper is designed asynchronously, the clock enable lines must be driven by the handshake lines. There is a risk here that the handshakes will not respond fast enough and data will be lost.

The key decision comes down to whether the wrapper is implemented as a push or pull system. Figure 3.44 shows a simple view of the wrapper to illustrate the input/output connections. If the assumption is made that the asynchronous interface is faster than the clock, a push system is preferred and consequently the synchronous wrapper is the choice due to its simplicity to design. In doing so the only asynchronous input to the wrapper is the acknowledge from the first stage of the asynchronous logic. This must be double registered to remove the possibility of metastability; otherwise the design of the wrapper is reasonably straight forward. However in doing so we reduce the latency of the wrapper significantly. An engineering design was made based on a Xilinx application note [24] (discussing the probability of metastability) to remove the double registers. On initialisation the wrapper will retrieve a value by enabling an address counter and the ROM for one clock cycle, on the next clock this will be detected and under the assumption that valid data is on the output of the pins, the request line to the asynchronous block is raised and the data transferred. Only when the handshake is complete will the RAM and address counter be clocked again. The corresponding VHDL implementation is discussed in Appendix B.3.

Although not explicitly stated the wrappers and techniques discussed here have been used in the following section to ensure that bundled data circuits are implemented and supplied correctly with data.

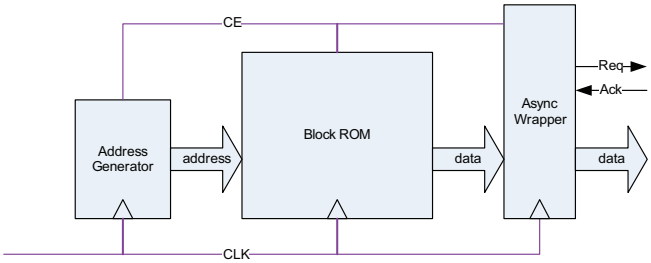


Figure 3.44: ROM Wrapper Example

3.6 Verification and Results

To the validate this design flow a set of tests were constructed on a simple computational pipeline. These tests confirm that the constructed circuits operate correctly on the appropriate device.

A simple pipeline (shown in Figure 3.45) consisting of a subtractor (-3) and multiplier ($\times 3$) was created in VHDL, the code was then synthesised to the equivalent netlist description of the device primitives. Next the clock tree was removed and each pipeline register altered to a latch configuration with a local controller now driving its transparency. An arbitrary length delay element is inserted between latch controllers to allocate routing space for further fine tuning. At this point in the design flow, the single rail combinatorial blocks can be placed and routed as normal and the pre-routed control network components were inserted (via the EDIF netlist) pre-routed.

The key assessment of this pipeline is to confirm that the constraints used to ensure efficient functionality of the asynchronous controllers were effective, the delay chains are implemented with sufficient accuracy,the data is correct and that the handshake protocols operate as expected. In order to test the pipeline, firstly a back annotated netilst was extracted from the post-place and route stage of the design flow. Running the same behavioural simulations on a circuit that has all the place and route timing information confirms that the pipeline is functionally correct. Secondly, an in-circuit simulation (i.e. a on-chip logic analyser) confirms that there is no anomalies from removing the circuit from a behavioural simulation and that it can interact with other embedded components successfully. Finally, an evaluation of the delay chains confirm that the crucial control network can match the combinatorial delays from the data path with sufficient granularity and repeatability across the fabric.

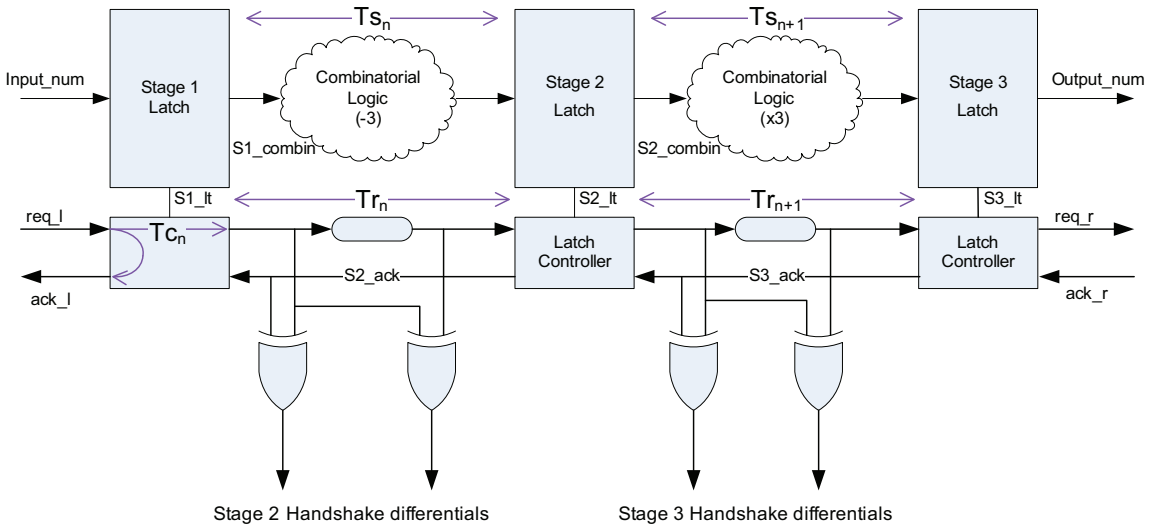


Figure 3.45: Handshake Test Points

3.6.1 Back Annotation

Asynchronous structures are inserted at a very fine granularity and the standard synchronous flow provides various avenues at each stage of the design flow to compare implementation and behavioural functionality. The most accurate simulation used to test correctness is the back annotated timing simulation. This stage has the most accurate timing information for the design and so can be used to fine tune matched delays up to the point of downloading the configuration onto the device. The timing information is derived from the architecture of a Xilinx Virtex 2 Pro, where the pins for the design were locked down and no test bench was included in creating the timing model. Figure 3.46

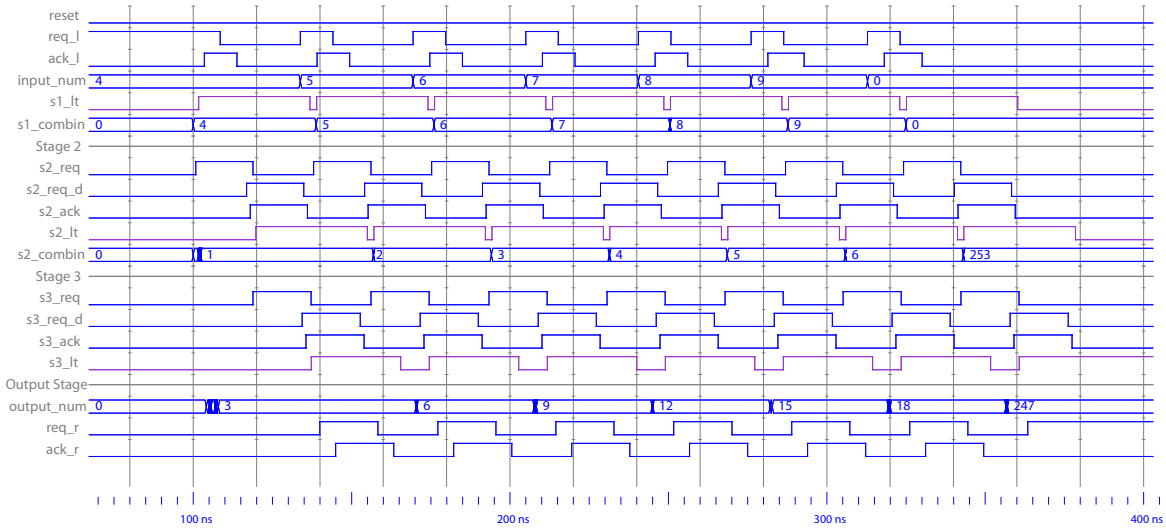


Figure 3.46: Back Annotated Waveform

shows the waveforms produced from the timing simulation, where the pipeline has a subtraction of three and then a multiplication of 3 to act as combinatorial clouds. The waveform includes the request lines from each stage, the delayed request through each delay chain, and the acknowledge for that data

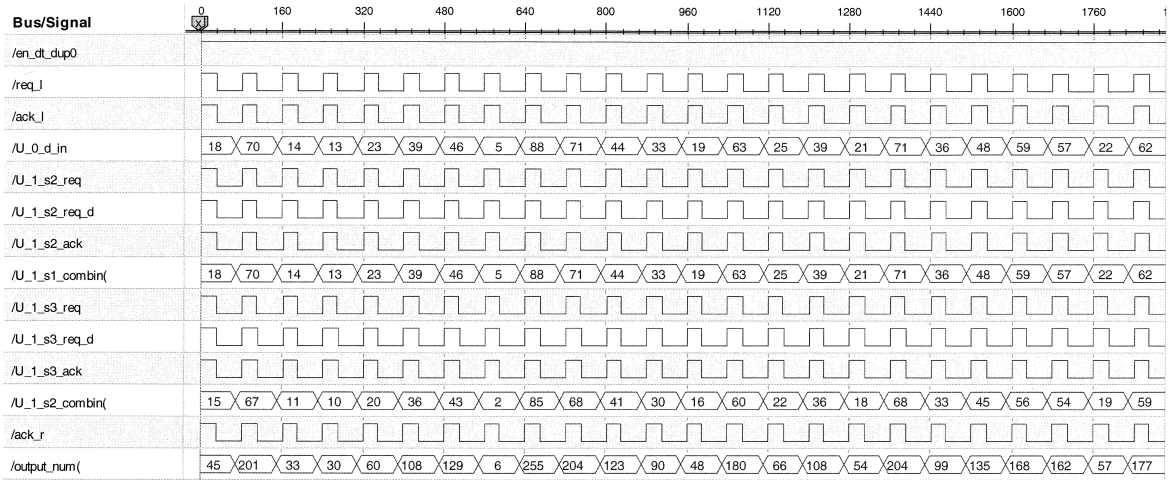


Figure 3.47: Chipscope Waveform

transaction. The waveform also shows the transitioning of the datapath combinatorial logic before the correct value is passed through the latch. Although the data lines take time to settle, their values are always stable and valid over the handshake transaction duration. Please refer to Figure 3.45 for associated circuit signal names.

From this simulation we are able to confirm correct operation of pipeline from a timing perspective and assume that the placement constraints and synthesis partitioning would function correctly for on-chip implementation.

3.6.2 In-circuit Verification

The only method of on-chip verification is by inserting a synchronous logic analyser to the netlist before the design is placed and routed. This tool allows signal values to be recorded on a triggering event and sent back to the host computer for analysis. The problem with this tool is that in being synchronous it must be clocked fast enough to achieve sufficient temporal resolution to sample the asynchronous transitions relative to each other. To verify the pipeline on-chip, we also add a ROM and accompanying wrapper as mentioned in Section 3.5.2.3 to generate source data for the pipeline. From the resultant waveform in Figure 3.47 it is easy to see that although data appears to be correct the transitions are inaccurately sampled. There is no validity information because of insufficient sampling resolution to resolve the validity regions from the data lines. We would have expected to see very similar waveforms to that of the back annotated timing simulation (shown in Figure 3.46) where the stages of the handshake that signify when the data is ready to be allowed through a latch can be seen. From the point of verifying combinatorial logic, the logic analyser is sufficient however it will only backup or disprove the results from the timing simulation.

To verify the handshake protocols additional logic was inserted to isolated the timing differences between each stage of the handshake. In order to do this two XOR gates are attached to the control wires in the arrangement shown in Figure 3.45. The outputs of the XOR gates were then fed to pins

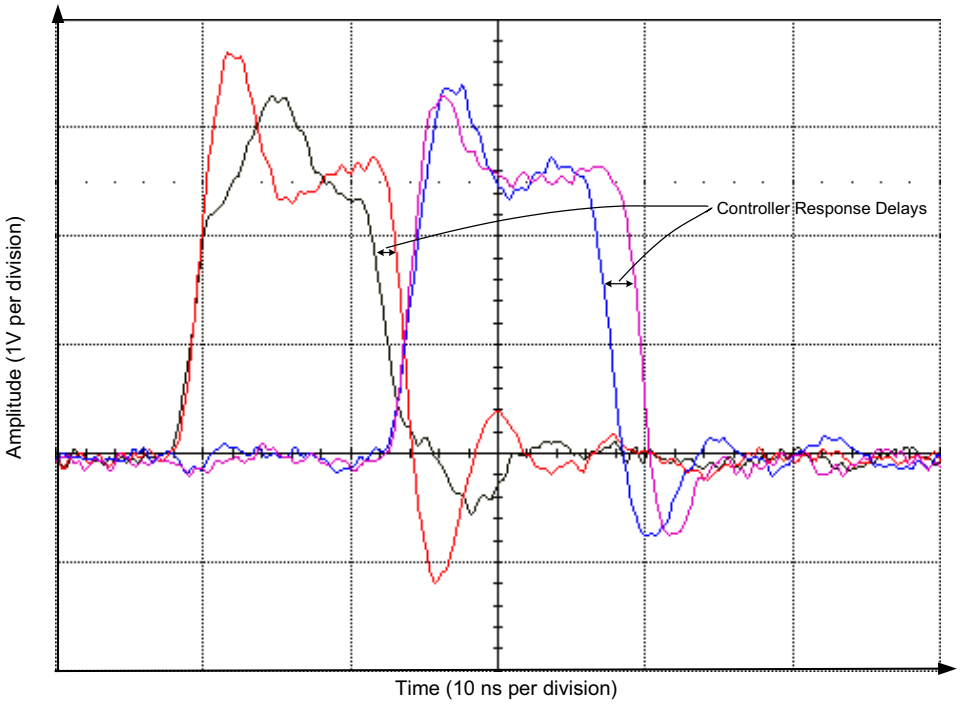


Figure 3.48: Test Pin Outputs of Handshake Differentials

where an oscilloscope could determine their changes in state. Starting from a 4-phase handshake, when the request line goes high, both XOR outputs will rise according to the difference in their inputs between the request, the delayed request and acknowledge lines. The output of the XOR surrounding the delay element will fall first due to the delay element equalising its inputs and the other XOR output will fall a short period later when the connected latch controller decides to raise the acknowledge line. If this sequence of events can be identified on the scope then we can confirm correct operation of the latch controllers. Figure 3.48 shows the output of the oscilloscope from a timebase of 10ns per division. The first two peaks (of the Stage 2 differentials) rise at the same time and fall at different times, indicating the delay between the input Rin and the Ain output of the latch controller. The second two peaks (from Stage 3 differentials) follow the same pattern shortly after. This method offers a unique way to measure forward and reverse latencies for asynchronous controllers on a FPGA. The key factor is making sure the XOR gates function as expected by minimising the delay between the input wires and also the output wires, and positioning them in a manner as so not to influence the operation of the circuit. From this oscilloscope plot we can say with certainty that the handshake sequences function correctly providing valid data transfer across the FPGA.

3.6.3 Delay Chain Matching

A significant amount of effort is expended creating delay chains to match combinatorial delays. This is a key focus for the design of the bundled data circuits used by Handshake Solutions. These are vitally important as the handshake guarantees when the data is valid at the input of the receiving register.

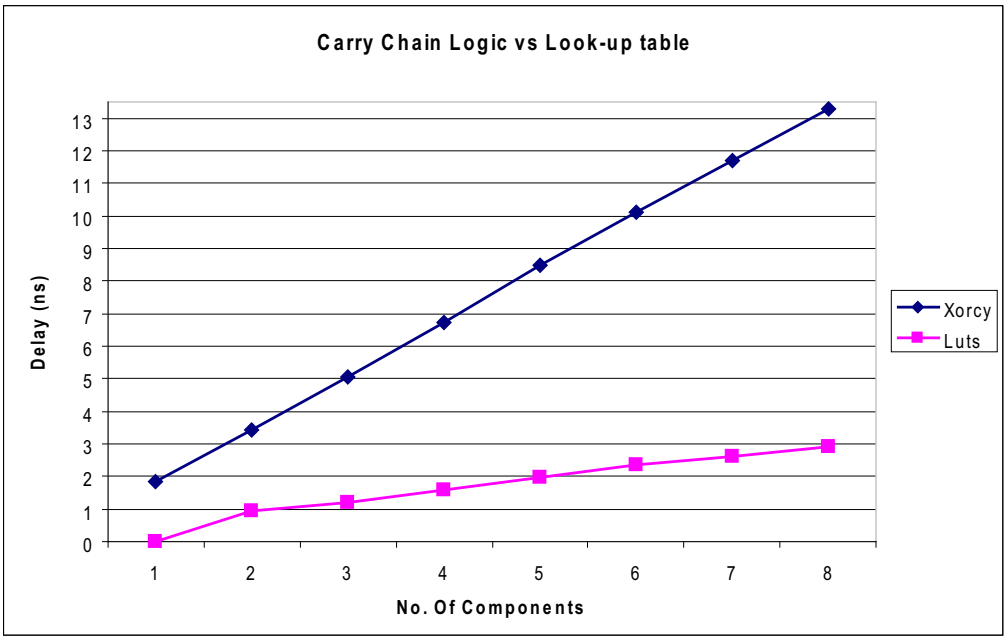


Figure 3.49: Delay Chain Comparison

Normally these control delays are accurate to within 10-20% of the combinatorial delay inclusive of engineering margins. In FPGAs matching delays requires a balance between logic delay and routing delay. In Figure 3.45 this means matching the logic and routing delay of the combinatorial block, Ts_n to the logic and routing delay of the request line, Tr_n. The logic and the routing can be indifferent as long as the total each is the same. This variation means that the only way to guarantee a reasonable match is to iteratively alter the delay chains in coordination with incremental compilation.This ensures that only the delay chains are changing on each iteration. The impact the delay chains have on circuit performance is significant, they mimic the local timing of the combinatorial operations on the asynchronous control network and so an over-estimation in their values degrades overall circuit latency and throughput.

There are two methods of creating delay chains on modern FPGA devices. Look-up tables are commonly used to implement logic functions in a FPGA, they accomplish this with constant delay regardless of the logic function. We can connect look-up tables in series to create a range of delay chains that have consistent delays. The other mechanism involves XOR gates that are only utilised when a reconfigurable logic block is configured for an addition. We can take advantage of the unused carry chains to connect them in series in a similar manner to the look-up tables. Both implementations need significant constraints (as shown in Figure 3.41b) to ensure delay consistency across the device. Figure 3.49 shows the difference in delay granularity from using the carry chain logic and look-up tables. These values in the graph are an average of delay chain lengths (the number of components connected in series) over the entire fabric of a Xilinx Virtex 2 FPGA, so they provide an initial estimate of delay chain lengths. Whereas look-up tables work well for fine grain delays they have two

disadvantages:

1. The amount of logic resources they utilise with large delays is excessive and detrimental to routing delays.

2. The choice of 4 inputs per look-up table means that routing delays have more variation between design iterations.

XOR carry chain logic is more efficient in terms of the maximum delay per the amount of logic utilised and it also has a more consistent routing delay compared to the look-up table delays. However the penalty is the granularity of delay chains constructed, which, for minor boolean logic functions, is a large overhead. Constraints are used to position the components used to create delay chains, the routing however is left to the vendor tools. This flexibility allows the vendors to optimise the routing within the routing matrices, but this comes at the expense slight variation in the input port of the look up table used and thus delay chain variation. The two delay chain components have not been mixed to increase the delay granularity above 3ns because bulk of combinatorial delays are below 3ns. The larger combinatorial delays are easily covered by the XOR based delay chains. The additional complexity of mixing the two delay component types also introduces further routing delay, increasing the variability of the delay chain.

3.7 Conclusions and Future Work

The piece of work described in this document lays the foundations for asynchronous circuits on synchronous FPGA fabric. The key challenge was finding common ground between the theoretical requirements of asynchronous design (that can easily be satisfied in an ASIC) and the limited resources and architecture provided by an FPGA. To this extent the bulk of the work has tackled the problem from both ends, looking at the most practical implementations and configuring FPGA fabric resources in a manner that allows them to operate asynchronously with consistent performance.

To implement circuits asynchronously on an FPGA this work has concluded that 4-phase bundled data circuits are the preferred option against data driven approaches. Although able to absorb delay variations associated with FPGA routing, the overhead in implementing any class of delay insensitive logic is too much for synchronous FPGA devices. Targetting asynchronous circuits for the majority of synchronous FPGA devices will inherently come into conflict with the native tools and language descriptions throughout the synchronous design flow. This is the main reason why synchronous EDA tools have been used to synthesise data path logic. These tools allow the maximum utilisation of the limited resources on an FPGA, allowing the focus of this work to be concentrated on the asynchronous components.

A new design flow has been tested and verified to implement bundled data circuits correctly. This flow contributes a novel implementation of asynchronous controllers and delay chains through the EDIF netlist where a particular set of constraints allow these components to be implemented consistently and reliably across the device. This unique approach of embedding constraints within the primitive descriptions provides a degree of accuracy that allows the technology independent assumptions of asynchronous design to be implemented with the correct constraints that allow asynchronous circuits to operate correctly in a specified FPGA technology. This approach has been verified by the construction of a simple arithmetic pipeline, simulating and validating its results from behavioural to implementation stages.

The debug and verification posed the most problems with the work completed. Since circuits are modified below the RTL description, verification and debugging is through the back annotation cycle. As with synchronous circuits, the addition of accurate timing information causes behavioural simulation times to grow substantially with increased design complexity. Altering the front end tool flow to utilise an asynchronous simulation environment would reduce this dependency. The ability to determine problems and bugs is also compromised with the requirement of using in-circuit verification to determine the correct functionality of the handshake control circuits. However the novel in-circuit verification methods demonstrated prove that functionality is not altered and this approach of designing asynchronous circuit on a FPGA is valid.

The natural evolution of this work is to automate the synthesis of asynchronous circuits for FPGAs. This means that a strict conversion must be created to allow synchronously described circuits to operate asynchronously. An automated process would also allow flexibility in the types of controllers used and an easier delay matching process. The focus for future work must investigate a methodology to identify the correct circuit structures and approaches which allow the functional capabilities of synchronous circuits to be maintained whilst changing the operating nature of the circuits.

This technical report has successfully provided ground work in developing a novel design flow that will allow circuits to operate asynchronously on synchronous FPGA fabric. The results and methods presented have added to the capabilities of Thales Optronics Ltd in implementing asynchronous circuits and understanding the optimisations that can be achieved a the post- synthesis level.

3.8 References

[23] Achronix Semiconductor Corp. (2006) Picopipe White Paper. Last Accessed: December 2011. [Online]. Available: http://www.achronix.com/achronix-picopipe-white-paper.html

[24] P. Alfke, "Metastable Recovery in Virtex-II Pro FPGAs (XAPP094)," Xilinx Ltd, Tech. Rep., February 2005, Last Accessed: December 2011. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp094.pdf

[25] A. Bardsley and D. A. Edwards, "The Balsa Asynchronous Circuit Synthesis System," in *Forum on Design Languages*, September 2000.

[26] E. Bergeron, M. Feeley, M.-A. Daigneault, and J. David, "Using dynamic reconfiguration to implement high-resolution programmable delays on an FPGA," in *Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*, June 2008, pp. 265 –268.

[27] A. Bink. (2008) A Glance at Handshake Solutions. Handshake Solutions. Last Accessed: March 2011. [Online]. Available: http://async.org.uk/async2008/async-nocs-slides/Tuesday/Keynote/GlanceatHandshakeSolutions_Async_2008.pdf

[28] J. Cortadella, M. Kishinevsky, A. Kondratyev, and L. Lavagno, "Introduction to Asynchronous Circuit Design: Specification and Synthesis," in *6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 2000.

[29] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Methodology and Tools for State Encoding in Asynchronous Circuit Synthesis," in *Proceedings for the 33rd Design Automation Conference*, June 1996, pp. 63 –66.

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

[31] L. Dai, Z. bin Liu, S. chi Liang, M. Yang, and L. li Wang, "FPGA Interconnect Testing Algorithm Based on Routing-Resource Graph," in *9th International Conference on Solid-State and Integrated-Circuit Technology*, October 2008, pp. 2087 –2090.

[32] M. S. D. R. Group, "Asynchronous System Design Flow Based on Petri Nets." University of Newcastle upon Tyne, March 2005, last Accessed: March 2011. [Online]. Available: http://www.staff.ncl.ac.uk/alex.yakovlev/home.formal/talks/besst-flow-slides.pdf

[33] S. Hauck, S. Burns, G. Borriello, and C. Ebeling, "An FPGA for Implementing Asynchronous Circuits," *IEEE Design and Test of Computers*, vol. 11, no. 3, 1994.

[34] Q. T. Ho, J.-B. Rigaud, L. Fesquet, M. Renaudin, and R. Rolland, "Implementing Asynchronous Circuits on LUT Based FPGAs," in *12th International Conference on Field-Programmable Logic and Applications*. Springer-Verlag, 2002, pp. 36–46.

[35] V. Khomenko, M. Koutny, and A. Yakovlev, "Logic Synthesis for Asynchronous Circuits based on Petri net Unfoldings and Incremental SAT," in *Fourth International Conference on Application of Concurrency to System Design*, June 2004, pp. 16 – 25.

[36] R. Konishi, H. Ito, H. Nakada, A. Nagoya, N. Imlig, T. Shiozawa, M. Inamori, K. Nagami, and K. Oguri, "PCA-1: A Fully Asynchronous, Self-Reconfigurable LSI," *International Symposium on Asynchronous Circuits and Systems*, 2001.

[37] L. Lavagno and A. L. Sangiovanni-Vincentelli, *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[38] M. Lewis, J. Garside, and L. Brackenbury, "Reconfigurable Latch Controllers for Low Power Asynchronous Circuits," in *ASYNC '99: Proceedings of the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1999.

[39] J. Lui, "Arithmetic and Control Components for an Asynchronous System," Ph.D. dissertation, University Of Manchester, 1998, last Accessed: March 2011. [Online]. Available: ftp://ftp.cs.man.ac.uk/pub/amulet/theses/JianweiPhD.pdf

[40] R. Manohar, "Reconfigurable Asynchronous Logic," *IEEE Conference on Custom Integrated Circuits*, pp. 13–20, September 2006.

[41] M. Marshall and G. Russell, "A Low Power Information Redundant Concurrent Error Detecting Asynchronous Processor," *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pp. 649–656, August 2007.

[42] A. Martin and M. Nystrom, "Asynchronous Techniques for System-on-Chip Design," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1089–1120, June 2006.

[43] N. Minas, M. Marshall, G. Russell, and A. Yakovlev, "FPGA Implementation of an Asynchronous Processor with Both Online and Offline Testing Capabilities," *14th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 128–137, April 2008.

[44] R. U. R. Mocho, G. H. Sartori, R. P. Ribas, and A. I. Reis, "Asynchronous Circuit Design on Reconfigurable Devices," in *Proceedings of the 19th Annual Symposium on Integrated circuits and Systems Design*. ACM, 2006, pp. 20–25.

[45] S. Moore and P. Robinson, "Rapid Prototyping of Self-timed Circuits," *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pp. 360–365, October 1998.

[46] D. Muller and W. Bartky, "A Theory of Asynchronous Circuits," in *Proceedings of the International Sympymposion on Theory of Switching.* Harvard University Press, 1959, pp. 204–243.

[47] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541 –580, April 1989.

[48] S. Nowick, "Design of a low-latency asynchronous adder using speculative completion," *IEE Proceedings -Computers and Digital Techniques*, vol. 143, no. 5, pp. 301 –307, September 1996.

[49] S. Nowick, K. Yun, P. Beerel, and A. Dooply, "Speculative Completion for the Design of High-Performance Asynchronous Dynamic Adders," in *Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1997, pp. 210 –223.

[50] R. Payne, "Asynchronous FPGA Architectures," *IEE Proceedings: Computers and Digital Techniques*, vol. 143, pp. 282–286, September 1996.

[51] D. Shang, F. Burns, A. Koelmans, A. Yakovlev, and F. Xia, "Asynchronous System Synthesis based on Direct Mapping using VHDL and Petri Nets," *IEE Proceedings: Computers and Digital Techniques*, vol. 151, pp. 209 – 220, May 2004.

[52] C. Sotiriou, "Implementing Asynchronous Circuits using a Conventional EDA Tool-Flow," *Proceedings. of the 39th Design Automation Conference*, pp. 415–418, 2002.

[53] J. Sparso and S. Furber, *Principles of Asynchronous Circuit Design - A Systems Perspective.* Kluwer Academic Publishers, December 2001.

[54] P. Standford, P. Mancuso, *Electronic Design Interchange Format Version 2 0 0*, Electronic Industries Association Std., Rev. 2nd edition, March 1988.

[55] M. Tahoori and S. Mitra, "Automatic Configuration Generation for FPGA Interconnect Testing," in *Proceedings of the 21st VLSI Test Symposium*, April 2003, pp. 134 – 139.

[56] J. Teifel and R. Manohar, "Highly pipelined Asynchronous FPGAs," in *Proceedings of the 12th international Symposium on Field Programmable Gate Arrays.* ACM, 2004, pp. 133–142.

[57] T. Verhoeff, "Delay-Insensitive Codes - An Overview," *Journal of Distributed Computing*, vol. 3, pp. 1–8, 1988.

[58] X. Wang, T. Ahonen, and J. Nurmi, "Prototyping a Globally Asynchronous Locally Synchronous Network-On-Chip on a Conventional FPGA Device Using Synchronous Design Tools," *International Conference on Field Programmable Logic and Applications*, pp. 1–6, August 2006.

[59] J. Yao, B. Dixon, C. Stroud, and V. Nelson, "System-level Built-In Self-Test of Global Routing Resources in Virtex-4 FPGAs," in *41st Southeastern Symposium on System Theory*, March 2009, pp. 29 –32.

[60] K. Yun and D. Dill, "Automatic Synthesis of Extended Burst-Mode Circuits. I. (Specification and hazard-free implementations)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 2, pp. 101 –117, February 1999.

[61] Y. Zafar and M. Ahmed, "Globally asynchronous locally synchronous micropipelined processor implementation in FPGA," *Proceedings of the IEEE Symposium on Emerging Technologies*, pp. 277–282, September 2005.

[62] Core Generator. Xilinx Inc. Last Accessed: March 2011. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/PlanAhead_Tutorial_RTL_Design_IP_Generation_w_CORE_Generator.pdf

[63] Virtex-II Datasheet. Xilinx Inc. Last Accessed: March 2011. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-ii_data_sheets.htm

[64] XST User Guide Chapter 3- Flip-FLop Retiming. Xilinx Inc. Last Accessed: March 2011. [Online]. Available: http://www.xilinx.com/itp/xilinx10/books/docs/xst/xst.pdf

Technical Report 3:

Automated Asynchronous Circuits Implemented in FPGAs(AACIF)

Author:	Phillip David Ferguson
Academic Supervisors:	Dr Aristides Efthymiou, Univ. of Edinburgh
	Dr Ahmet Erdogan, Univ. Of Edinburgh
Industrial Supervisor:	Danny Hume, Thales Optronics Ltd

4.1 Aims and Introduction

Following the previous work on establishing asynchronous circuits on FPGAs, this body of research addresses the need for a repeatable automated solution for asynchronous circuits on FPGAs. With embedded system complexities growing year on year, the need to capture the functionality of a digital circuit has also expanded. Abstract behavioural descriptions are now common place with lower level implementations becoming building blocks in larger designs. Digital circuit designers are no longer designing circuits from the ground up, as a result there has been increased focus and scrutiny on the performance and abilities of automated tools to synthesise functional descriptions into the correct implementation. Synchronous design languages have been able to serve these roles for decades, however asynchronous description languages have always remained at a lower abstraction level. Circuits implemented in FPGAs can now be behaviourally described by multiple languages that synthesise to primitive components on the device. Each option is a different design flow which attempts to design circuits in a more effective and efficient manner. Implementing circuits in an FPGA asynchronously with an automated approach provides an additional design flow option, the benefits of this new approach will be presented in this work.

This document introduces an automated design flow to allow asynchronous circuits to be implemented consistently on FPGA devices. This automated design flow is the primary focus of this report and is discussed in significant detail. Firstly EDIF circuit representations are introduced and then the process by which asynchronous circuits are eventually implemented on devices is explained in detail. An asynchronous controller comparison is then performed, concluding with a new asynchronous controller that fits directly into the automated design proposal. To validate this automated design flow, analysis of the timing performance is documented along with comparison on the resource utilisation of the resultant circuits and their effects on the power supply network.

4.2 FPGA Design Flow Proposal

The standard FPGA design flow (as discussed in previous reports) provides a controlled environment for designers to implement circuits described in Hardware Description Languages (HDLs) on their chosen device. Device vendors have a significant amount of control over circuit performance in the form of proprietary place and route tools (and sometimes synthesis tools) which implement a synthesised netlist of primitive device components in the most optimal arrangement. Xilinx place and route tools only accept two formats of synthesised circuits, their proprietary format, NGC files or EDIF files that include proprietary files describing specific embedded components. To increase design portability and reuse capabilities in accordance with the Reuse Methodology Manual [76] most designers will create circuit designs in a high level language that use embedded components as black boxes. This

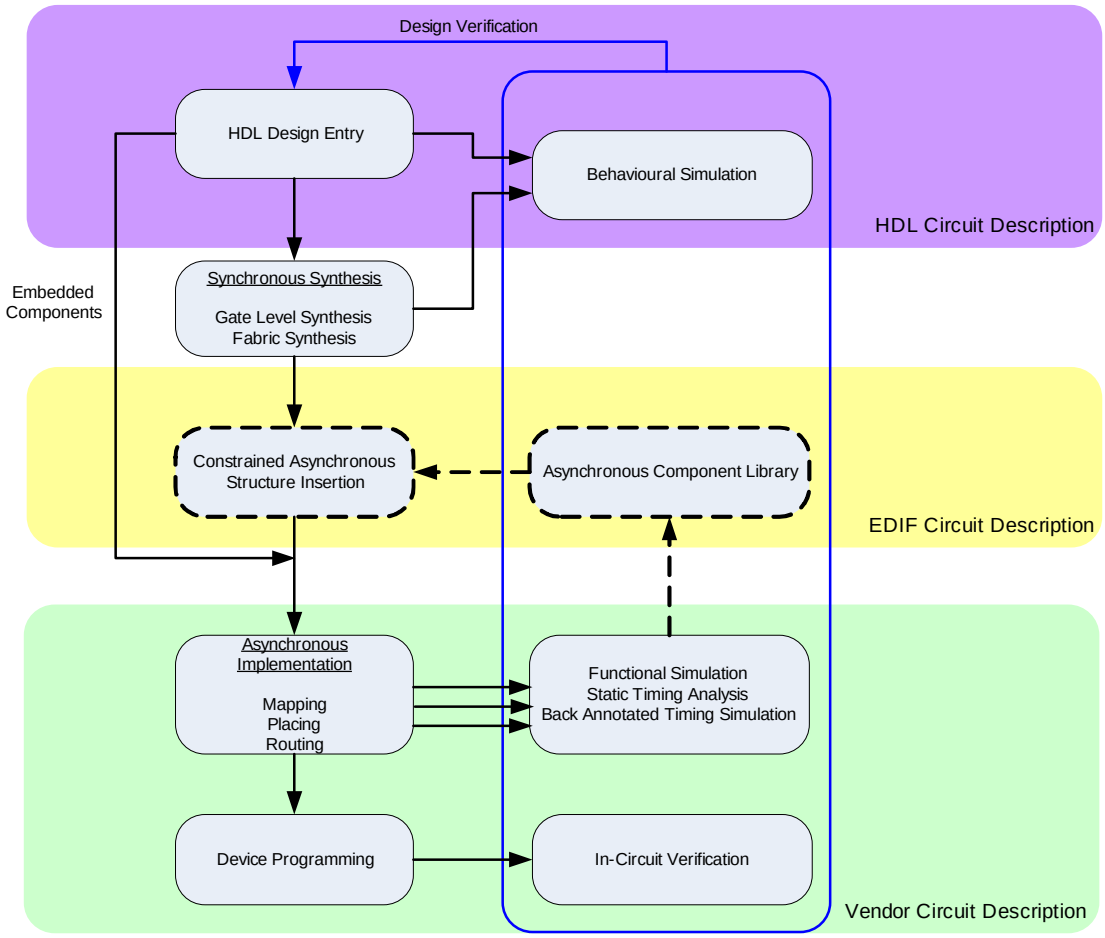


Figure 4.50: FPGA Design Flow

ensures the most optimal components are used during synthesis. These components, such as block RAMs or multiply-accumulate units require additional configuration information post synthesis to be configured correctly for optimal performance. All of this external information resolves into a single EDIF file and a number of Xilinx specific files that represent the configurations for the embedded components used in the design. The EDIF file therefore operates as the primary exchange format, or glue between synthesis and place and route tools. Although the same can be achieved with NGC files, being proprietary, these files are encrypted and there is no direct access to their contents, limiting their reuse capability.

In an asynchronous context, without creating a new design tool for asynchronous circuits (e.g. Balsa or Haste), EDIF files are the most appropriate place to create asynchronous circuits from lower level FPGA primitive components. This decision infers a conversion flow from synthesised synchronous circuits to asynchronous circuits capable of being placed and routed by commercial tools. We make the assumption that additional devices can be serviced with an additional library of asynchronous components customised to that device architecture. The design flow shown in Figure 4.50 provides a simplistic view of the FPGA design flow and the modifications required to implement asynchronous circuits on FPGA fabric. If the additional stages (indicated with a broken outlines) were removed,

a synchronous FPGA design flow would be evident. Where as in previous work asynchronous components were created independently, in this design flow an EDIF asynchronous library would plug into the Constrained Asynchronous Structure Insertion process, removing the need to create a control path separately. The novel method of creating the control path automatically from the datapath structure presented in this report allows a significant reduction in design time and portability of the asynchronous components between different device architectures.

4.3 EDIF Circuit Representations

Initially designed in 1986 the electronic design interchange format (EDIF) is a neutral data format that had the primary goal of capturing all aspects of VLSI design a single respresentation. As a result this format is capable of capturing and representing a multitude of information including: high level design entry, electronic schematics including symbol libraries, physical design libraries (for PCB's, standard cells, FPGAs etc), physical layout information and interconnect routing, simulator stimulus and response data and simulator logic and timing models. With syntax similar to the LISP programming language [81] or postscript printer language, EDIF is particularly difficult to write by hand and thus its main usage in modern EDA flows is an exchange format between tools. The main benefit of this syntax is that it is easy to parse and can be extended with minimal disruption to the language. In its simplest form the EDIF syntax (shown briefly in Figure 4.51 and completely in Appendix C.1) is a series of statements. *Keywords* and associated constructs represent electronic design data, i.e. objects, their characteristics, relationships between objects and placeholders grouping such information.

This format has gone through four version revisions, each gradually enhancing the capability of the format to accurately represent electronic design data. EDIF version 2 0 0 [69] has cemented itself in the FPGA design flow as the default exchange format between synthesis and place and route tools. In this context it can completely capture the post synthesis netlists generated for different devices and vendor place and route tools. The revisions made to EDIF in its 3rd and 4th iteration have not been supported as they target stages of electronic system design outside of the FPGA context.

Design information is arranged in a hierarchical structure. A single EDIF file is the highest level of hierarchy which must contain status information, and may contain several designs and libraries. Each library contains a technology definition and a set of cell definitions, each cell can be represented by one or more views that show the cell in the form of a schematic, layout, behavioural specification, document, etc. A library allows cells to be grouped according to common characteristics such as device primitives or process technology. Each view is defined as a particular view type and contains an interface and a contents section. Different views are linked to other views via a view map. The interface section defines how a cell can be connected to other cells. The contents section defines the

components and interconnections that make up the functionality or operation of that cell. Libraries also contain technology information so that defaults can be specified for given behaviour, graphics and other attributes. The relevant view for FPGA synthesis is the NETLIST view. This view is appropriate as it only provides a description of how internal primitive FPGA components connect through nets.

Design hierarchy is represented by including other instances of cells in the cell description. A design will identify a particular cell within a library as the top cell, providing a starting point for the design within the file. This is graphically shown in Figure 4.51. There are 3 levels of complexity that are contained within an EDIF file. Level 0 means that the file only contains constants and no parameters, level 1 allows cells to be defined with functions, operators, variables and parameters, level 2 allows control constructs such as if statements to be used.

Figure 4.51: EDIF Structure

4.3.1 Mapping EDIF Files to Object Orientated Structures

The EDIF format is an interchange format that is intended to be parsed by EDA tools. Manual modification is both tedious and prone to error. Creating an automated conversion requires parsing of an EDIF file and analysis of its structure. To this end the EDIF Tools API by the EDIF Team at Brigham Young University [85] provides the environment to create a conversion process in JAVA [70].This section will document the abstractions of the EDIF structure into graph theory and consequently an object orientated environment supported by JAVA.

A graph is a set of nodes and the connections between these nodes. In reality these nodes could represent anything from cities to computer terminals. Formally nodes are described as vertices and the connections between them as edges:

> A simple graph is defined, G = (V, E) where V is a finite set called the vertices of G, and E is a finite set called the Edges of G.

In the context of EDIF circuits constructed from cells and nets that interconnect them, we associate primitive FPGA components such as registers, look-up tables, multiplexers to vertices with a range of properties, and nets to be the edges that connect these vertices. This association allows a circuit to be completely described and subsequently queried according to graphing algorithms. Vertices are then distinguishable by the FPGA primitive component they represent.

What we gain by transposing netlists to graphs is a significant amount of information on how circuits interconnect. This is initial phase of conversion where the synchronous netlist must be altered into a format where the structure of registers can be determined.

The BYU API replicates the EDIF file internal structure in an object orientated environment. From the UML diagram in Figure 4.52 we can relate directly to what was described previously. For each object within an EDIF file we have associated classes. For example, a cell has an EdifCell class which has relevant attributes such has the library the cell is contained in, the ports that cells use to connect to other cells and a list of all of the instances of that cell. The EdifCell class has methods to return those attributes and add new aspects to the cell such as adding ports or nets. The UML diagram uses composition to represent the existence dependency of all the class objects, and provides the background information and functions to begin treating the EDIF netlist as a graph . The key benefit of this API is that it allows the netlist to be queried according to graph theory. In essence it produces a richer description of an EDIF file allowing access to the naming conventions and EDIF cell interlinking.

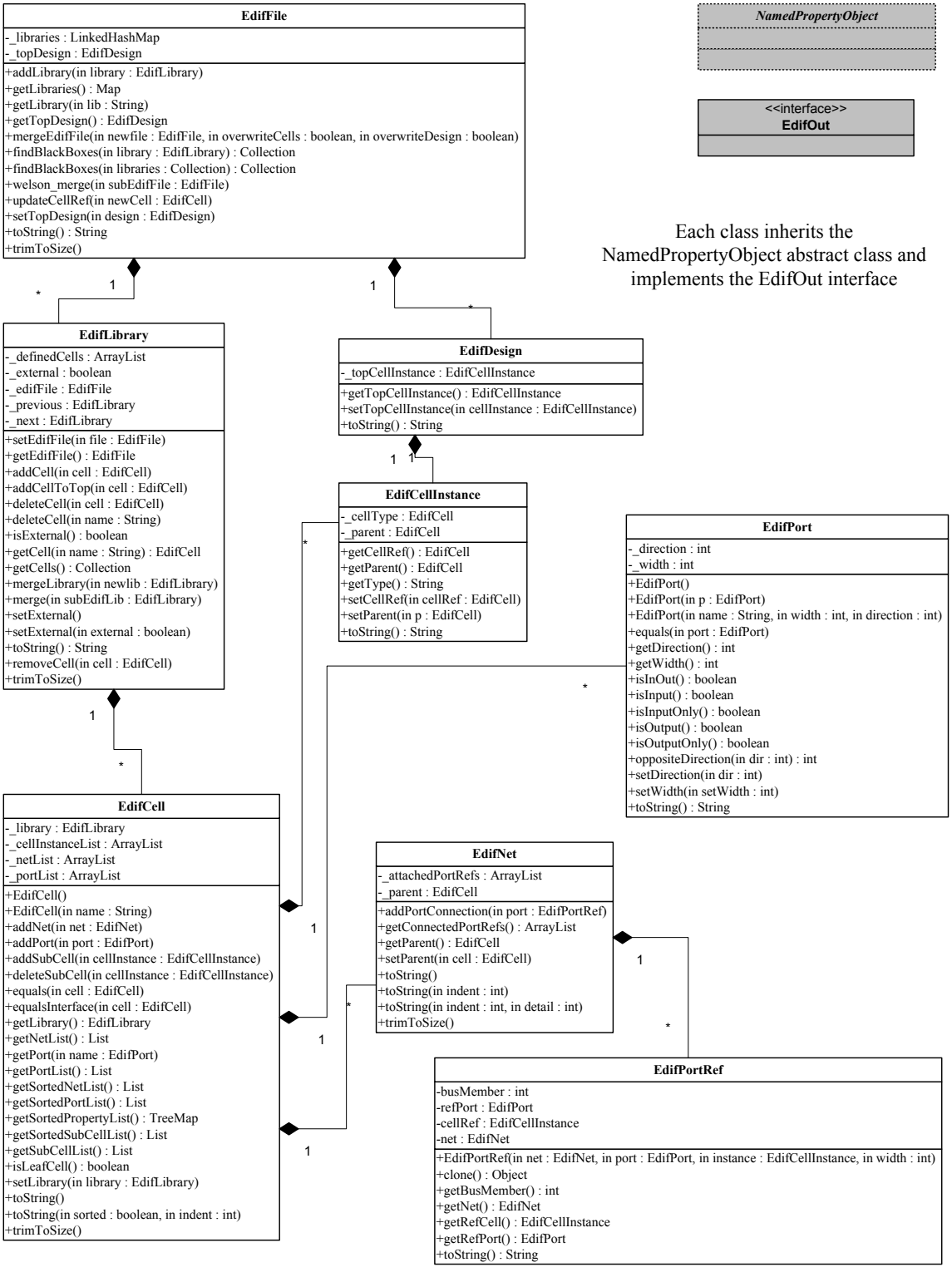


Figure 4.52: UML Diagram of the EDIF Structure

4.4 Conversion Algorithm/Process

The conversion of a synchronous EDIF netlist [75] is split into 5 stages. The resultant circuits are compatible with synchronous data patterns and in doing so they require special considerations for clock enable pins and feedback structures commonly used in FPGA designs.

1. The first stage parses the EDIF netlist in terms of Xilinx primitive components and maps this to a graph structure where nodes can be assigned additional information about surrounding connections. Next data path registers which have been spilt into 1-bit flip-flops during synthesis are identified and grouped according to their EDIF cell name. This assigns a single controller to drive the transparency of the data path registers.

2. The second stage disconnects the global clock from each register group then duplicates the register to connect them in series with the original group. An asynchronous controller is then connected to each group through local clock lines.

3. The third stage analyses each data path stage and creates linking tables indicating the interconnection between other data path register groups.

4. Based on the information gathered in the previous stage, the fourth stage inserts the asynchronous control network (the local request and acknowledge signals) to mimic the data path connections, adding delay chains on the request lines to match the combinatorial delays on the data path, and additional Muller C-elements to accommodate non-linear (fork and join) structures.

5. The fifth and final stage inserts mapping and timing constraints to ensure controllers are kept local to register groups and combinatorial delays, as well as delay chains are minimised.

Each stage will now be discussed in greater detail including the challenges that directed specific design decisions.

4.4.1 Parsing Input Files and Grouping Data Path registers

We assume two input files require parsing into object orientated respresentations as described by the API. One is the EDIF file produced from synchronous synthesis tools targetting a Xilinx device, the other is an EDIF file containing a library of asynchronous components that are specific to that device architecture. These asynchronous EDIF cells are created with Xilinx primitive components that map directly onto the FPGA architecture of that device.

Since subsequent place and route tools only accept one EDIF file, the two files must be merged, i.e. an asynchronous cell library must be created within the synchronous EDIF netlist and the asynchronous cells that will be used in that netlist copied over. However, in order to ensure this is successful,

the primitive FPGA components used in the asynchronous cells must also be present in the source synchronous file, if they are not then the asynchronous cells will not have the correct primitive components to describe their functionality. A small function compares the primitive components used in the synchronous netlist with those in the asynchronous cell library and only copies over the primitives components that are not present. Finally the entire asynchronous cell library can be copied over.

Grouping registers identifies the crucial locations where asynchronous controllers will eventually be inserted. This is achieved by establishing an EDIF bus net naming policy that allows the names of single bit registers to be grouped according to similar names. Every cell that is identified as a register populates a hashmap that contains a list of buses in the design and the registers that made up those buses. From this hashmap a graph is created to establish another layer of abstraction away from the original netlist. It is this graph that is used in subsequent search and insertion processes.

4.4.2 Graphing Structures

Visualising and representing the netlist as a formal graph allows datapaths to be extracted from bit level netlists by grouping nodes. To facilitate identification of register groups, the graphs were split into 4 stages. As a simple example, a linear pipeline with a fixed operand multiplier and an adder between three registers was constructed. Figure 4.53 shows the flattened synchronous netlist. This is graphically equivalent information to the netlist place and route tools will see. Synthesis tools will naturally flatten any hierarchical blocks to identify areas of optimisation. This is what makes tracing signals form RTL level to FPGA fabric level very difficult. The red squares highlighted are input/output buffers and the green squares indicate bit-level registers. At this stage it is very difficult to link this EDIF structure to the source RTL description. The second iteration of this graph identifies register groupings by searching for common naming patterns and begins to rebuild the structure of the circuit into a recognisable form. The common naming patterns are based upon regular expressions that can be formed using the regular expression toolbox supplied with a standard installation of JAVA. Synchronous synthesis tools have their own particular naming conventions when they are creating EDIF netlists. This conversion tool has been tested with a single synthesis tool, Mentor Precision. As a result, only one naming policy has been used:

UNDERSCORE_BITPOSITION_UNDERSCORE_REGEX = "^(.+)_(\\d+)_$";

This regular expression will match against a net name that ends in "_<#>_" where <#> is a decimal number e.g. "test_5_ ". There are no restriction on creating additional naming policies to suit other synthesis tools such as:

PARENTHESIS_BITPOSITION_PARENTHESIS_REGEX = "^(.+)\\((\\d+)\\)$";

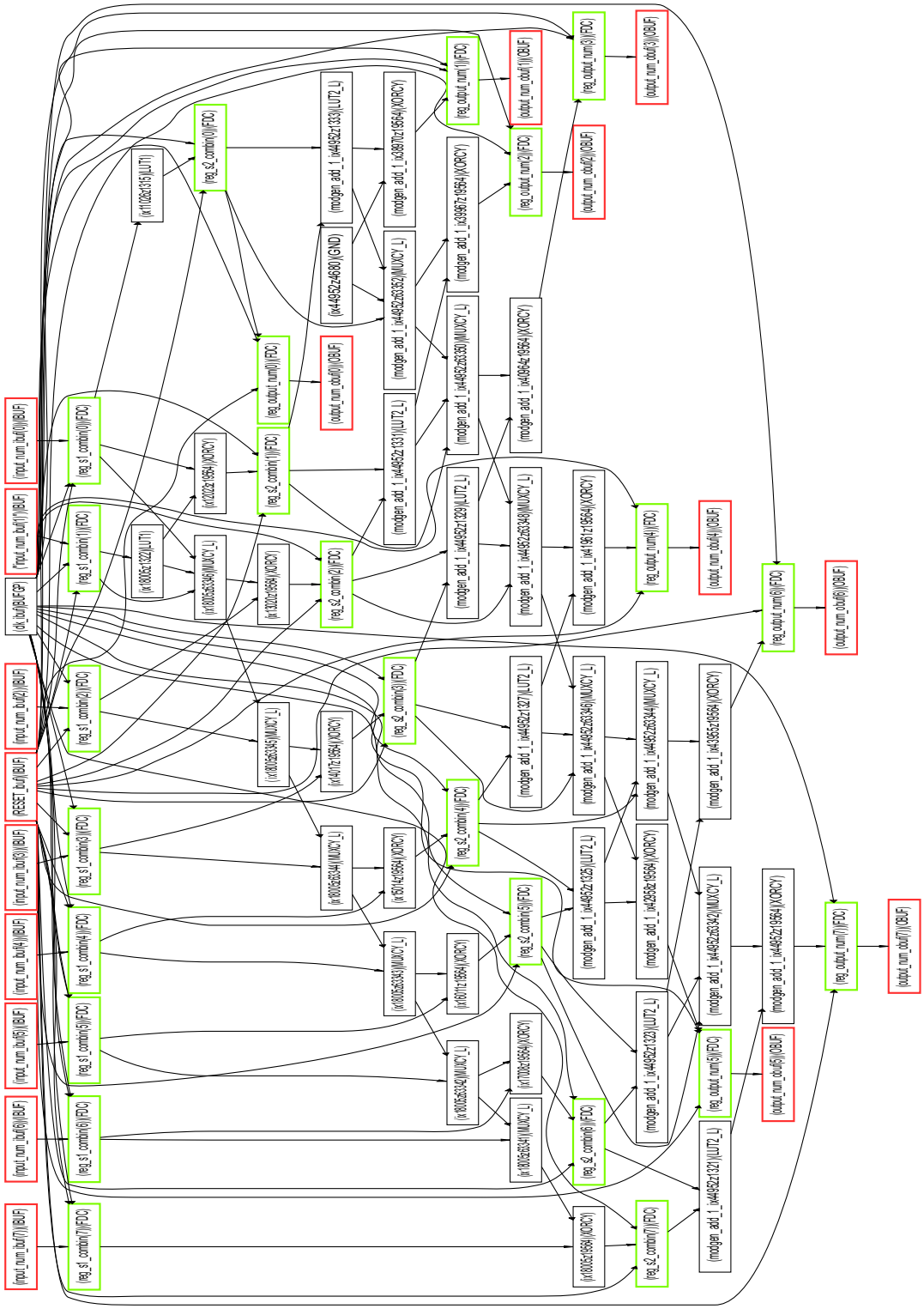


Figure 4.53: Flattened Linear Pipeline

This regular expression will match against a net name that ends in "(<#>)" e.g. "test(5)"[6]. From these regular expressions, we can extract a basename and a bit position indicating the name of the data bus and the position of that bit in the bus. This allows the individual nets to be grouped back into abstracted data paths as described at the RTL level. Figure 4.54 shows the groupings in blue and again the input/output buffers in red. Here we can clearly identify the combinatorial primitives the FPGA uses to construct the multiplier and adder. This information will be used later to construct timing information for representative delay chains. At this stage of graphing we can clearly identify the clock network and what register groupings every register group connects. The final stage, as shown in Figure 4.54 by the purple blocks, replaces the clock tree with asynchronous latch controllers and delay chains.

4.4.3 Register Duplication and Controller Insertion

With a list of register groups now identified, every group requires two modifications before operation specific tailoring begins. The first modification is to insert a secondary register (as shown in Figure 4.55) behind each register in the circuit. There are three main reasons for doing so.

In a synchronous circuit every combinatorial block performs its operation simultaneously. If the equivalent asynchronous circuit uses a 4 phase handshake protocol this level of decoupling has to be designed in. Adding a register means that the circuit is fully decoupled without increasing the combinatorial logic used in the controller.

A synchronous circuit assumes that every register will receive data correctly after an explicit time, thus it does not require an acknowledge line. As an asynchronous system needs this acknowledge. If only one register/latch is used a feedback circuit will cause a circular dependency on the control lines, i.e. the output needs to confirm the reception of data at the input before it initiates another data cycle, a secondary register removes this dependency. The additional register allows new data on the input to be captured whilst the output data is still processing in the subsequent stage, removing this dependency. The acknowledge will still act as expected if there is more input data still waiting on the output data to be transferred. The additional register only adds one extra bubble to each register group.

The third and final reason is flow-equivalence. This justification is explored in the context of controller discussion in Section 4.5.2.

The second modification is to insert controllers that trigger the capturing of data on the input of each register. These controllers are augmented with Muller C-elements (shown in Figure 4.55) on the input request (Ri) and output acknowledge (Ao) pins to deal with non-linear fork and join constructs that combine and synchronise multiple input/output connections for each register group.

[6]Note that this is not a valid EDIF name but has historically been used by some place and route tools

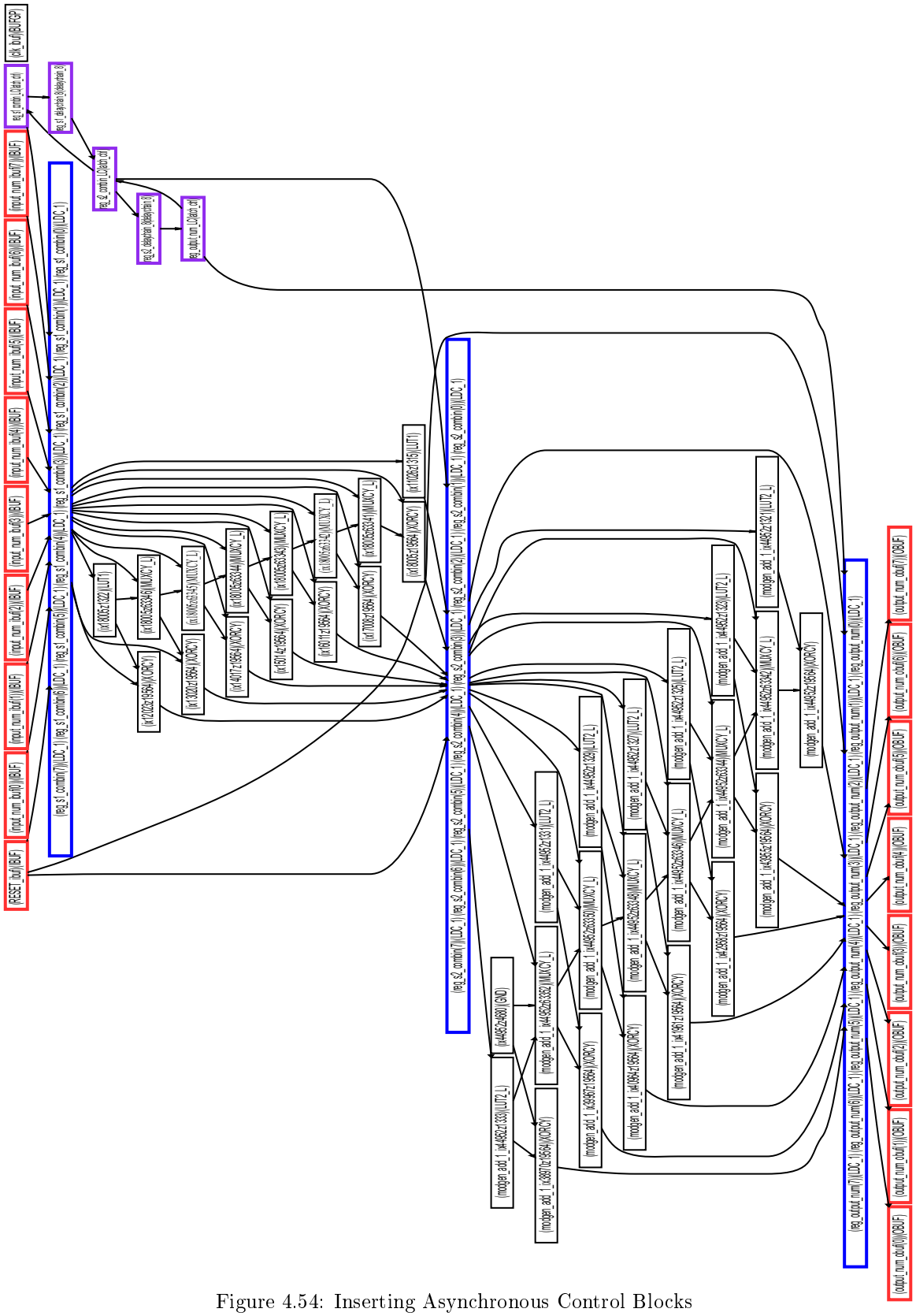
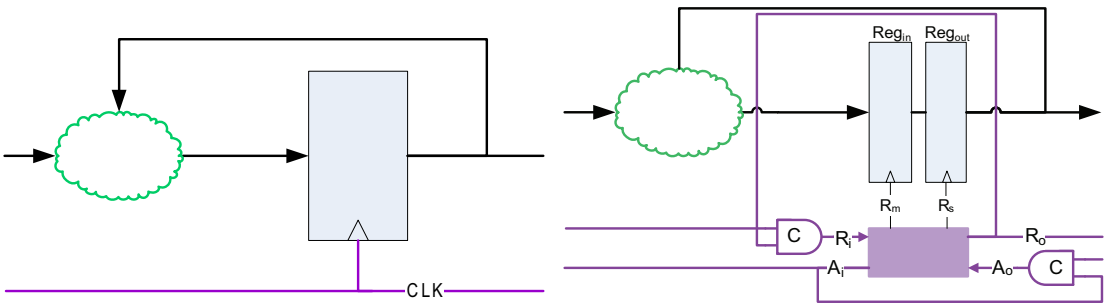


Figure 4.54: Inserting Asynchronous Control Blocks



(a) Synchronous Circuit (b) Asynchronous Equivalent Circuit

Figure 4.55: Single Register Conversion with Feedback

Some controllers may not require additional Muller C-elements and are removed later in the process when the control network is created. Figure 4.70 shows the behaviour of each controller in the form of a signal transition graph (STG) [83]. This graph describes the sequence of events each controller must adhere to, including clocking both register groups. The initial reset conditions are shown with black dots on the STG. To deal with clock enable pins, specialised controllers have been designed to synchronise an additional input request that represents the clock enable data validity before the first register is opened (Rm+). Here we treat the clock enable line as another data input, but a data input that connects to another port on the register (CE). Assuming we have two input requests to indicate that the clock enable is valid and the data is valid, we allow the transfer of data to continue. The value of the clock enable will indicate if the transaction allows new input data through or data that is currently on the output of the first register (Reg_{in}).

4.4.4 Register Mapping and Tracing Interconnections

Mapping register groups with data paths is the key component of conversion. Synchronous synthesis optimisations are already complete, however in doing so key information on data flow through the circuit has been lost. As well as register groups the asynchronous control network also needs to know how register groups are connected in order to sequence the data transfers correctly.

For each register group there are seven possible connections, shown in Figure 4.56 and described below:

Forward Connections

1. The Q output of a register is connected to the D input of another register.
   - this infers that the asynchronous control network needs a request line including a delay element in the control network for each member of this list and a Muller C-element may be required to collect the acknowledge signals if there are multiple outputs
2. The Q output of a register is connected to an output buffer, which then drives an output pad.

- this infers that additional output and input buffers will be required to deal with the outgoing and incoming asynchronous control connections.

3. The Q output of a register connect to the clock enable (CE) input of another register.

   - this infers that request lines from the source register group will not connect to the standard inputs of the controller.

Reverse Connections

4. The D input of a register is connected to the Q output of another register.

   - if there are multiple entries in this list this infers that a combining Muller C-element is required for the incoming request signals of this register group

5. The D input of a register is connected to an input buffer, which is sourced from an input pad.

   - this also infers that additional output and input buffers will be required to deal with the outgoing and incoming asynchronous control connections.

6. The clock enable (CE) input of a register is connected to the Q output of another register.

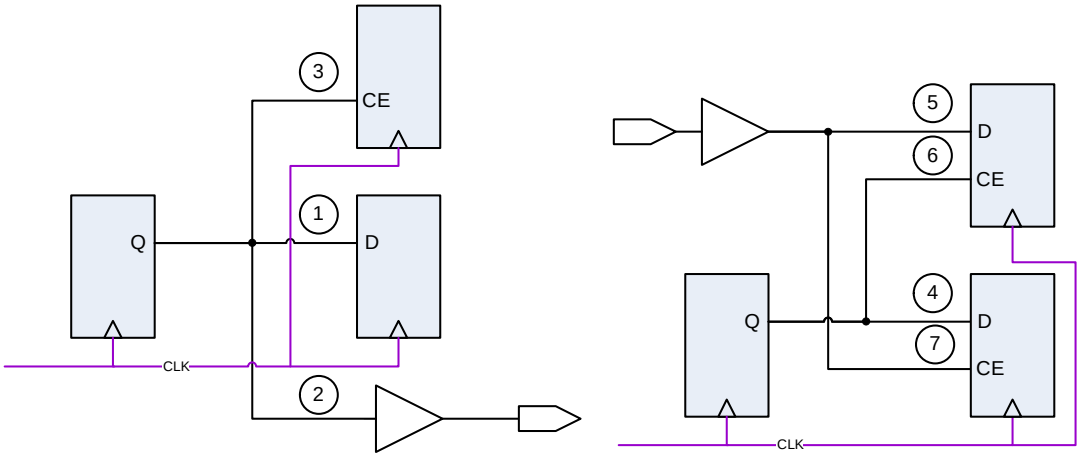
   - this infers that a specific controller is required by this register group to maintain clock enable functionality

7. The clock enable (CE) input of a register connects to an input buffer

   - this infers that additional output and input buffers will be required to deal with incoming clock enable connections.

For each register group(i.e. one node in the graph), all of these attributes of registers are stored in a hierarchical set of hashmaps that characterise register group interconnectivity through specific input/output ports. To determine the interconnections that fill up these data sets we probe abstracted graphs for connections shown in Figure 4.56. In order to accomplish this a Depth First Search (DFS) algorithm[78] is required to iterate through the netlist, probing for data path connections to and from registers. The connections that are represented in graphs by edges are categorised by their direction:

- A back edge connects to a vertex that has already been visited in the graph. This could be a feedback edge within a combinatorial cloud.
- A cross edge connect vertices that are not ancestors of each other. This could be where two look-up tables share inputs and the output from one look-up table feeds into the inputs of the other look up table.



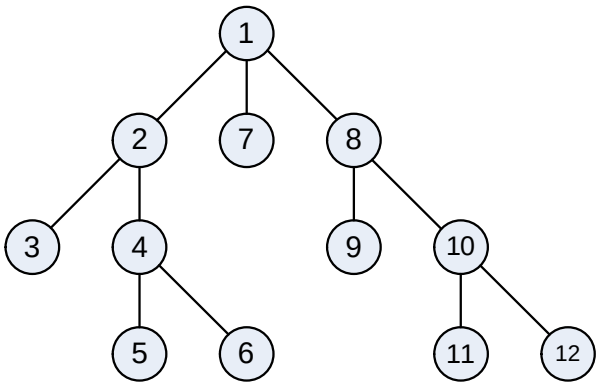
(a) Forward Connections (b) Backward Connections

Figure 4.56: Register DFS connections

- A forward edge connects to a vertex that is a descendant of another branch on the graph. This could be the combination of two edges, e.g. a multiplexer.

Edges in this context represent nets throughout the entire design that enter and exit register groups, only by testing the connection attributes of each edge can we determine if it is connected to a register group and so classify it according to the connection types listed above.This algorithm filters the edges and returns sets of registers that indicate the source registers that drive the input combinatorial block to the current register group and the sink registers at the end of the output combinatorial block from which the current register group drive.

Creating a map (technically a graph) of the registers within the circuit via DFS is the main workhorse of the entire conversion. Processing time here is linked to how large a netlist this algorithm has to search through. The order that a depth first algorithm traverses a tree (or netlist in this case) is graphically demonstrated in Figure 4.57a. Pseudo code of this algorithm is presented in Figure 4.57b and is based around the functionality of stack that records the nodes and edges the algorithm passes through whilst attempting to trace links between register groups. Initially a source node is passed to the algorithm, this is a register group from the abstracted graph, there is no particular register group order required. From this node, edges to other nodes are added to the top of the stack for each register in the source node group. The algorithm then follows the edge on the top of the stack to its connecting node, if this node is a register then it records this in the appropriate list and then pops the current edge off of the stack and follows the next edge on top of the stack. If this node is not a register (i.e. it is a LUT) it then traces more edges from that node(depending on the search direction from the source node) and adds them to the stack to be followed.Once all the edges from the source node have been followed and the set of hashmaps filled with the interconnection information another source node group is selected from the abstracted graph and the process begins again.



(a) Depth-First Search Order

```
while (stack is not empty)
{ if (edge)
  { pop edge
    determine targetNode
    if (targetNode has been visited)
    { determine edge type[forward, back cross,feedback]
      add to list of edges traced
    }
    else(targetNode not visited)
    { push Node onto stack
      add to  edges traced list
    }
    end if
  }
else(node)
  { if (already visited)
    { pop from stack
    }
    else (not visited)
    { mark as visited
      if (register or IO Buffer and not the topNode)
      { add register or IOBUF to the connections list
        // do not extend the tree
      }
      else(not a register or IOBUFF and topNode)
      { // extend the tree as this is a combinatorial block
        add edges to targetLinks list
      }
      end if
      if (topNode)
        { // partially extend the tree
          add direction edges to the targetLinks list
        }
        else (not the topNode)
        { // extend the tree
          add all edges of that node to the targetLinks list
        }
      end if
    }
  end if
    add targetLinks/edges to the stack
  }
end while
```

(b) Register Depth First Search Pseudo Code

Figure 4.57: Register Depth First Search Algorithm

4.4.5 Connecting Controllers and Inserting Delay chains

Based on the information collected for each register group we link the control network. For each register group, we cycle through the possible connection types discussed in Section 4.4.4 that form the contents of the hashmap. For each connection type, D-inputs, Q-outputs, CE-inputs and CE-outputs we trace different lines in the control network. The input connection types establish the acknowledge network by querying another layer of hashmaps. Logical tests confirm if the inputs to that register group are driven by external inputs or registers. Depending in the number of those sources and if those source also drive different register groups, we can establish if the acknowledge (Ai) for that controller will be spilt, connected to another controller, connected to a c-element in front of another controller or connected to an output buffer. The output connection types establish the request network including delay chains in a very similar manner. For each register group we count the number of other register groups the datapath connects to via hashmaps. After this the hashmap of each connecting register group is tested to assess if it has a single or multiple datapath inputs, then we know if the request line will connect directly to a C-element or controller. Only then can we move onto another register group and trace its request network. The information based on the combinatorial logic the delay chains are trying to replicate is placed in the UCF file for later optimisation. The delay chains are initially configured with a generic value of 8 look-up tables. We assume that this choice will have the maximum amount of routing delay and variabiliy. In latter stages this will either be reduce to a lesser amount of look-up tables or increase to an XOR gate, both requiring less routing an making the delay chain more predicable. After all the register groups have been processed, the external connection to Input/Output buffers are created for the control network. These input and output ports were never in the original design and so care must be taken in combining all the relevant request and acknowledge control lines for each data input or output.

4.4.6 Constraint Insertion

Synchronous designs need constraints to maintain register-to-register, IO-block-to register and register-to-IO-block timing consistent with internal or external clocks. The natural assumption would be that these constraints would be of minimal use to asynchronous circuits on FPGAs. However, placement and routing tools have not been designed with asynchronous circuits in mind and so without appropriate constraints, like the synchronous circuits, these tools will not consider the delays in the circuit relevant to component placement or circuit performance. Whilst asynchronous circuits would be more robust to this variation in delays they would suffer the same if not greater slow down in throughput.

User contraint files (UCF) are normally used by place and route tools to guide the placement of components in order to achieve timing closure. Whilst there are explicit constraints to direct tools where exactly to place primitive components, there are no such explicit constraints on what precise

routing resources should be used to interconnect these components. In this context routing constraints with the UCF only limit what routing resources can be utilised. Routing resources can only be altered post place and route via editing of the programmable interconnect points (PIPs).

To extract the benefit of minimum input to output latency in a circuit, constraints must still be applied to the asynchronous circuit. Constraints are placed in two locations, within the EDIF file itself and within the User Constraints File. Normally constraints within the EDIF file are targeted towards the mapping and placement processes within the place and route tools. The relative location

```
(instance (rename reg_c_2_prim "reg_c(2)")
(viewRef PRIM (cellRef FDCE (libraryRef xcv2p)))
(property U_SET (string "reg_c_2_"))
(property RLOC (string "X0Y0")))

(instance reg_c_2_sec
(viewRef PRIM (cellRef FDC (libraryRef xcv2p)))
(property U_SET (string "reg_c_2_"))
(property RLOC (string "X0Y1")))
```

Figure 4.58: EDIF Mapping Constraint

constraint (RLOC) shown in Figure 4.58 allows components to be confined to certain areas of the device, this means that timing between primitive components can be guaranteed. In this example these mapping constraints are applied on the components that have the same U_SET property and restrict the placement of the secondary registers relative to the primary registers. They are also primarily used within the controllers to minimise the physical distrances between Muller C-elements. Timing constraints, which mostly affect placement and routing tools, are contained within the UCF file. They are used to limit the worst case propagation delay through combinatorial logic, this in turn

```
INST "OUT_VIDEO_CB<0>" TNM = PADS outPort_OUT_VIDEO_CB;
INST "OUT_VIDEO_CB<1>" TNM = PADS outPort_OUT_VIDEO_CB;
INST "OUT_VIDEO_CB<2>" TNM = PADS outPort_OUT_VIDEO_CB;
INST "OUT_VIDEO_CB<3>" TNM = PADS outPort_OUT_VIDEO_CB;
INST "OUT_VIDEO_CB<4>" TNM = PADS outPort_OUT_VIDEO_CB;
TIMESPEC "TS_del_reg_OUT_VIDEO_CB_OUT_VIDEO_CB"
 = FROM "reg_OUT_VIDEO_CB_sec" TO "outPort_OUT_VIDEO_CB" 3 ns;
```

Figure 4.59: UCF Constraint Example

makes delay chain matching much more predictable. The example in Figure 4.59 shows output pads being assigned the same timing group (TNM). Timing specifications (TIMESPEC) are then assigned between the groups. In this case 3ns is the delay between the last register in the pipeline and the output buffers.

4.5 Creating a Device Dependent Asynchronous library

To introduce asynchronous operation to a circuit/netlist, an additional set of cells are required to replace the synchronous clock net. Ideally an additional EDIF file containing a library of asynchronous components described in terms of architecture primitive components would provide a simple mechanism to modify the source synchronous EDIF file. This option is facilitated by the use of the *external* keyword in the EDIF specification to allow linking between multiple EDIF files and a sharing of cells. However it was discovered that Xilinx place and route tools will only accept one EDIF file in its input list of design files. This adds an additional complication in that only asynchronous components used in the conversion will be added to the source EDIF file: in essence merging the two EDIF files into one.

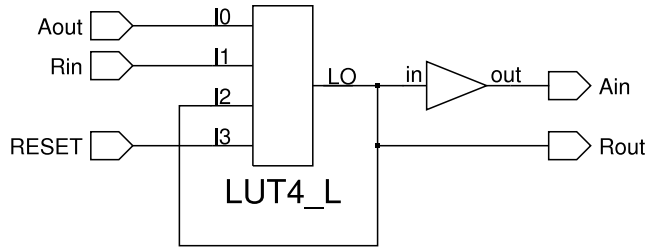
The following subsections will document the asynchronous components required and their resultant structure within the EDIF file.

4.5.1 Delay Chains

For completeness the asynchronous component library contain delay chains as well as controllers. The discussion on how delay chains are created and their performance is documented in the previous report. In the context of the design flow proposed in this document, the delay chains are described in terms of primitive EDIF cell components within the external library. These EDIF components are the result of the code discussed in the previous report. There are sixteen delay chains within the asynchronous library, eight that composed of XOR gates in series and eight that are composed of look-up tables in series. Each has specific constraints based on their component placement and are named according to their delay properties. Both variants are asymmetric delays that allow a larger delay for rising edge events and a smaller delay for falling edge events. The performance of these delays in context of this design flow will be explored further in Section 4.6.

4.5.2 Asynchronous Controllers

The undecoupled controllers demonstrated in previous work, although small, do not have sufficient performance for the resources used. Using an undecoupled controller would mean that only half of the latches/registers in a design would hold valid data at any one instant. At this stage a number of other controllers were investigated for various attributes. Their operations were also considered in terms of the synchronous circuit structures they would augment. The investigation and evaluation is based around a linear pipeline. This provides a testbench platform to determine the throughput potential of the controllers in an ideal environment with accurate implementation delays. This pipeline also provides sufficient flexibility to explore the constraints required to find the optimal implementation that minimises routing delay.



(a) Synthesised EDIF

```
attribute BEL : string;
attribute BEL of c_mull: label is "F";
– limits LUT postition in slice

c_mull : LUT4_L
generic map ( INIT => x"00d4")
– implements c= (rst')(ab'+ c(a+b'))
port map (LO => c_int,
– Altered for better feedback timing
I0 => b_int,
I1 => a_int,
I2 => c_int,
I3 => RESET);
```

(b) VHDL Description

Figure 4.60: Undecoupled Latch Controller

This section documents the controllers considered and their FPGA implementations. A comparison of their linear pipeline performance is discussed along with the evolution of the primary controller used in the conversion flow.

4.5.2.1 Undecoupled Latch Controller

As mentioned in the previous report, the undecoupled latch controller was primarily used as a trial controller to allow simple debugging opportunities and assessing the ease of which Muller C-elements can be transferred and optimised on an FPGA. In the context of creating a library of asynchronous components, this controller was the first entry in this library. It provided useful guidance in programing look-up tables to optimise gates and and inverters. The Figure 4.60 shows a Muller C-element described in VHDL vs its equivalent synthesised EDIF netlist description. This EDIF representation was the key testing platform for the combination of mapping constraints requirement to ensure consistent performance. This means altering the input combinations for particular signals, consequently altering the look-up table contents and ensuring that the look-up table is positioned correctly within a slice to minimise the feedback delay. The downside to this controller is performance. A pipeline using undecoupled latch controllers will only contain half as many valid tokens as there are latches in the pipeline. Figure 4.61 shows the timing waveform of a 4 stage pipeline circuit and the amount of tokens successfully passed after 500ns. In this simulation we assume that the output response is instantaneous and the input response is also instantaneous between the Request and Acknowledge events. The intermediary stages show request signals after the delay chains as well as the acknowledge signals. For these comparisons the delay chains were all the same value to minimise any impact they might have on evaluating the controller performance. These simulations form a simple comparison of the subsequent controllers and their relative merits in relation to the amount of resources they use.

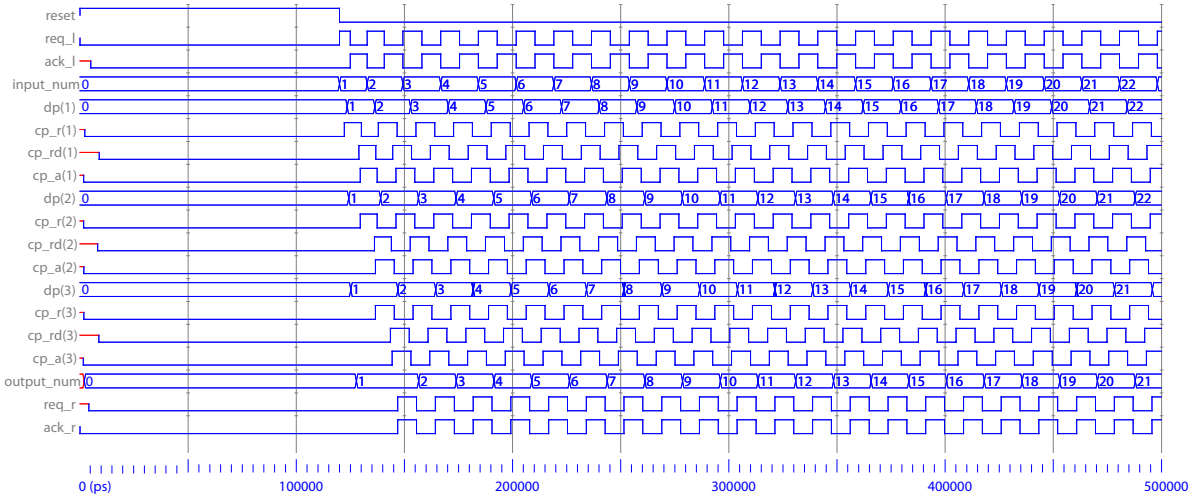


Figure 4.61: Undecoupled Latch Controller Waveform

The result of the undecoupled controller was surprising. Within the 500ns time limit 19 data transactions were concluded. In FPGAs the dominant delays are routing delays and since the undecoupled controller has minimal logic, it benefits from reduced routing congestion and good performance compared with the subsequent controllers.

4.5.2.2 Semi-decoupled Latch Controller

The next addition to the asynchronous library was the semi-decoupled latch controller [71] presented Paul Day and Steve Furber that increased performance to allow every latch in the pipeline to hold valid data. In terms of FPGA resources utilised, Figure 4.62 shows the increase in complexity where, look-up tables require constraints to position themselves relative to their neighbours. Look-up tables now need specific constraints on the suitable input pin for the function (i.e. the correct position for a feedback, request or acknowledge pins), the correct position within the slice to minimise the delay between internal controller signals and the correct position of the controller relative to the datapath latches/registers it is associated with. This means using a number of placement constraints, regional location constraints (RLOC) to maintain relative look-up table positions, BEL constraints to restrict output pins within a slice and specifying look-up input pins explicitly. Evaluating all configuration possibilities required an extensive testing period to find the optimal combination.

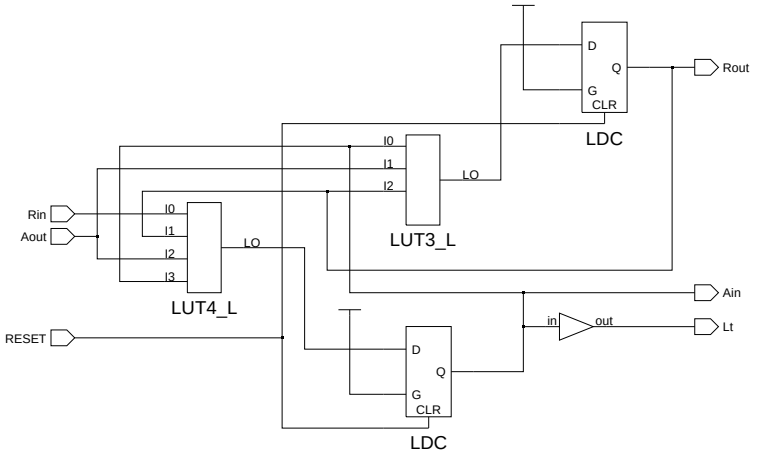


Figure 4.62: Semi-Decoupled Latch Controller

Note that latches are now used within controllers. This is for two reasons: the first concerns how Muller C-elements are reset and the second is concerns the utilisation of resources within an FPGA slice. A 4 input look-up table can singularly represent a three input Muller C-element, with the fourth input used for feedback as shown in Figure 4.60. If a reset pin is required another input is consumed, meaning a single look-up table could only implement a two input Muller C-element. The Xilinx slice architecture has a register/latch primitive located after each look-up table (for synchronous design efficiency) and so can act as a reset point for the output of Muller C-elements. This novel technique allows four input look-up tables to implement a three input Muller C-element.

The timing waveform from the simulation shows the performance differences slight modifications to handshake dependencies can make. The throughput unfortunately has not increased, managing 18 transactions within 500ns. As mentioned previously, the increased complexity means increased routing congestion and thus the benefits of further decoupling are not realised.

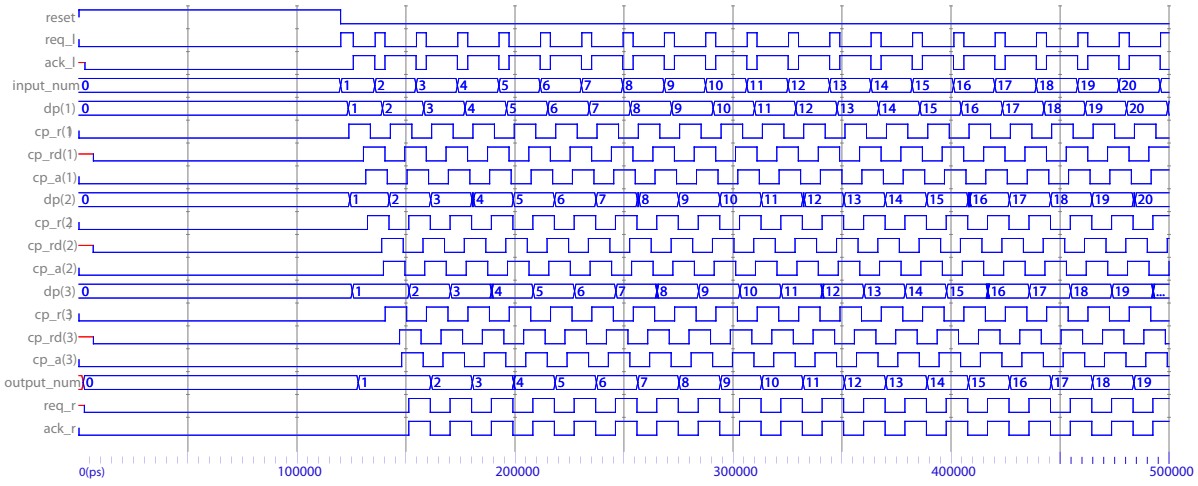


Figure 4.63: Semi Decoupled Latch Controller Waveform

4.5.2.3 Broad Request-Activated Fully Decoupled (BRF) Latch Controller

Originally proposed by Lui [79] this controller proved to have very high performance within the class of 4-phase controllers evaluated. The unique aspect of this controller was achieving fully decoupled performance from only three gates, as opposed to the four gate solution suggested by Paul Day and Steve Furber [71]. From an FPGA context this controller was another step up in complexity, requiring additional routing constraints to maintain optimal routing delay between look-up tables. Increasing the decoupling within an asynchronous controller means increasing the number of internal variables to indicate when the parts of the input and output handshakes are complete and the latch can be enabled. In terms of the amount of resources, shown in Figure 4.64, this fully decoupled controller needs an additional look-up table and latch over and above the semi-decoupled version as discussed previously. Again the locations of these primitive components need to be found empirically to establish the optimal time of the key timing arcs in the STG.

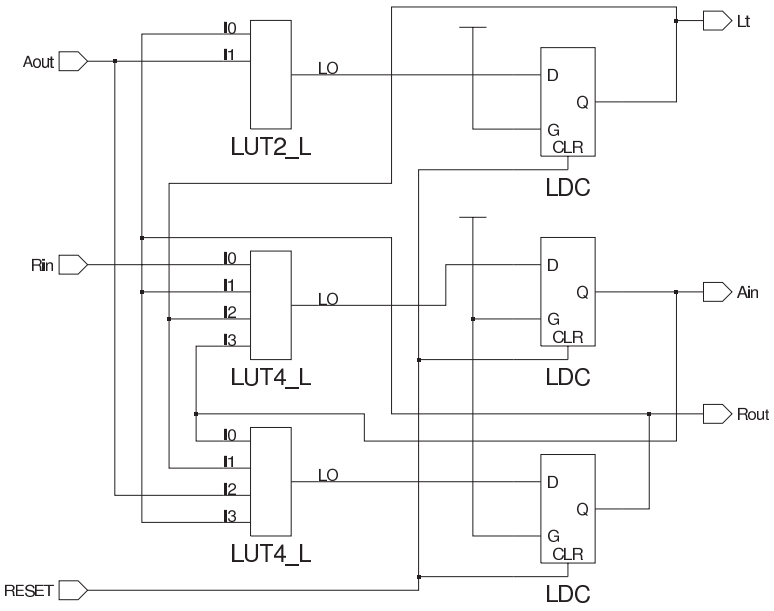


Figure 4.64: Broad Request Activated Fully Decoupled Latch Controller

Figure 4.65 shows minimal increase in speed over the semi-decoupled controller, with only an additional transaction completed in the same time period. There is also a crucial difference in this simulation.

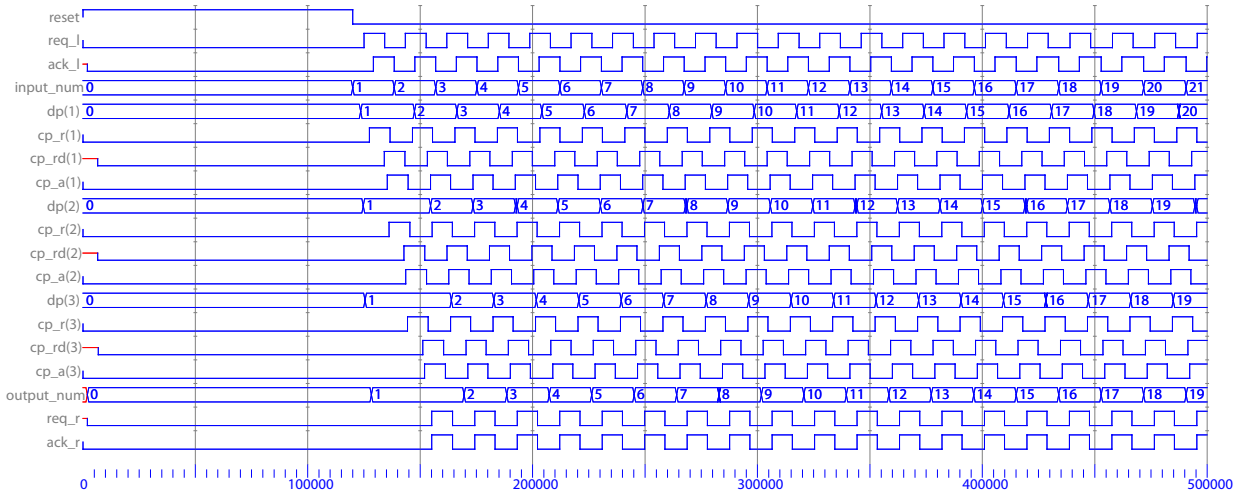


Figure 4.65: BRF Latch Controller Waveform

Previously the input and output response delays to the pipeline were zero, using this controller the request response (Rin−) was delayed by 5ns after every acknowledge event otherwise data would be lost. The primary reason for this was the critical arcs in the STG shown in Figure 4.66. If the internal transitions take longer than either the input or output arcs then data will be lost. Since there is a degree of variability in FPGA routing even with constraints, this assumption could be violated thus making this controller unsuitable for use in FPGAs.

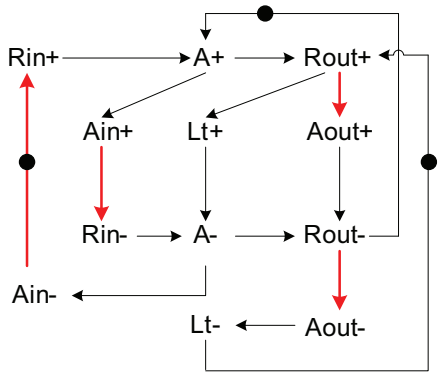


Figure 4.66: Critical Arcs of the BRF Latch Controller

4.5.2.4 Mousetrap Latch Controller

To increase performance and resource utilisation again, the mousetrap latch controller [82] was also considered. This design uses 2-phase signalling for the transfer of data between latches and level-based signalling for the transparency of latches, removing the complex capture/pass register structure of traditional 2-phase designs [84]. It is a very elegant and quick solution, that allows flexibility for non-linear datapaths. Figure 4.67 shows with only one XOR gate in its gate-level description it can be synthesised without using look-up tables at all, instead using the carry chain logic within the slice architecture.

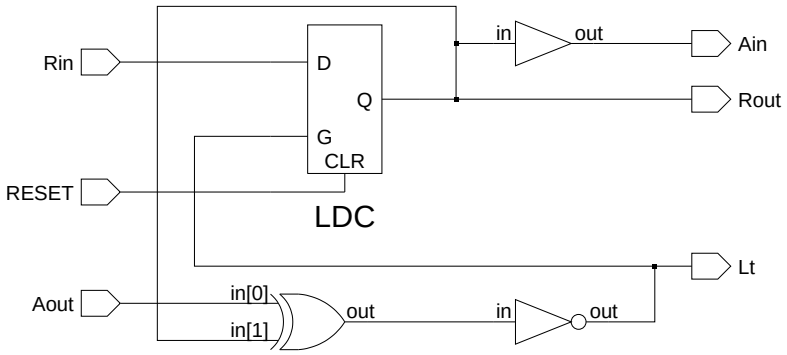


Figure 4.67: Mousetrap Latch Controller

The timing waveform shows a large increase in throughput from using 2-phase signalling nearly doubling the number of transactions to 41. This performance comes from three controller functions acting in parallel: the request to the successor stage is generated, the acknowledge is sent to the predecessor stage and the latch becomes opaque- protecting the data held in the current stage. This speed and simplicity has only one timing constraint that cannot be modelled in an STG: there is a race condition between disabling the latch of the current stage and receiving new data from the previous stage. If the latches are not opaque quick enough data corruption will occur. In a traditionally designed asynchronous circuit these controllers are a very persuasive design decision. In the context of synchronous conversion, there are other issues that must be addressed. As discussed in Section 4.4.4 feedback loops are a major justification for using multiple registers. Using multiple mousetrap controllers, with their lower resource usage would be a simple solution however adapting the mousetrap controller to accommodate the re-ordering of signal events to initiate data transfer would mean re-designing the mousetrap controller to be flow equivalent.

> "Two behaviours are flow equivalent if and only if they have the same domain and their signals hold the same values in the same order." [73]

In other words, if a synchronous and an asynchronous circuit which perform the same operation are

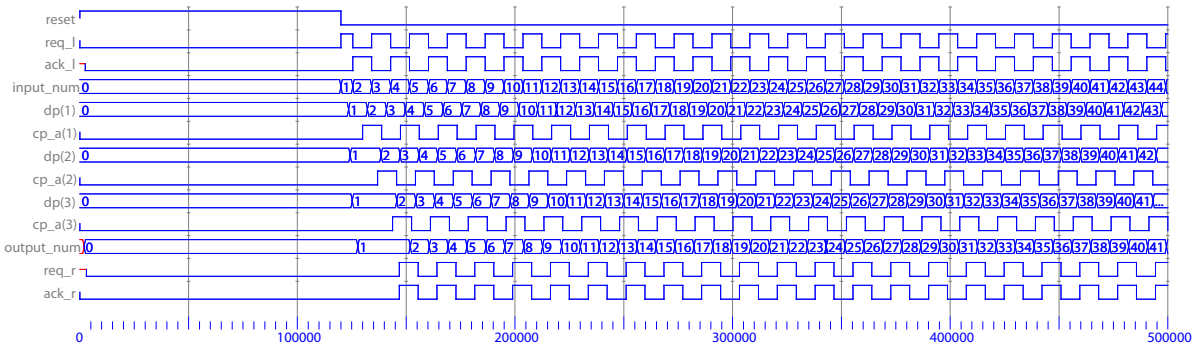


Figure 4.68: Mousetrap Latch Controller Waveform

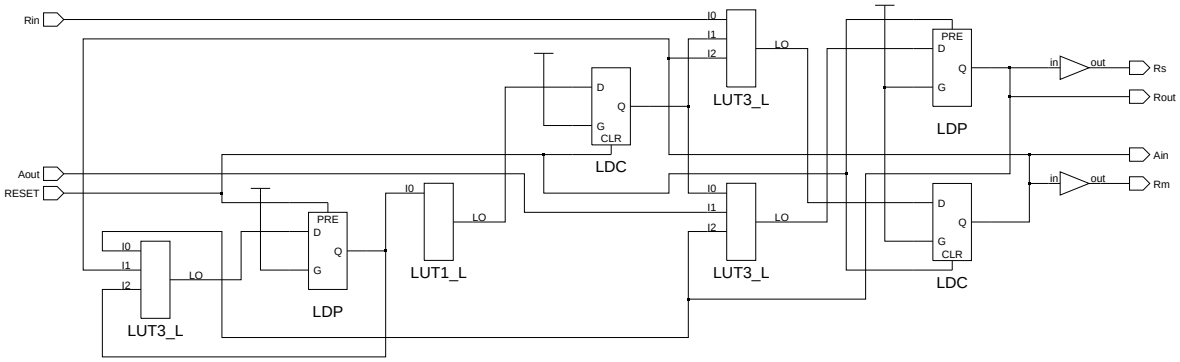


Figure 4.69: AACIF Register Controller

supplied with the same input data, the output data would be exactly the same. Where this becomes a problem is in redundancy used in synchronous design to represent data validity. The classic example is a clock enable line: if we assume that a register and a FIFO are connected in series using the same clock signal. If a logical test is performed on the input data of the register to control the validity[7] of that data, that control signal would drive the clock enable pin of that register. The FIFO would sample the output data from the register on every rising clock edge, including the valid and invalid data. There would be over head in having to sample the same clock enable control signal and bundled it with the data for every subsequent clock enable pin. The equivalent asynchronous design would not output invalid data at all. A true asynchronous design would perform the same logical test to assess data validity, the logical test would then interact the control handshake to inhibit the request signal ever reaching the controller of the corresponding register. In this case the FIFO would not store the same data values as the synchronous circuit. In other words the behaviours are not flow equivalent, although both circuits have the same domain (the same input values), the signals would not hold the same values nor the same order.

The complexity of introducing a further control step to deal with feedback structures and changing the order of transitions proved an investigation point that drifted out of context and thus this design exploration was not continued. The focus then changed to designing a controller that could implement decoupled operation as well as maintain flow-equivalence.

.

4.5.2.5 AACIF Controller

This is a new 4-phase controller (shown in Figure 4.69) which was designed to approximate the performance of the mousetrap controllers whilst accommodating flow equivalence. The controller also maintains the implementation restrictions of one slice per controller, using at most only four look-up tables and 4 registers. The first major difference to the other controllers, as mentioned previously in Section 4.4.3, is the use of double data path registers. There are two motivators for this decision.

[7] Validity does not indicate the validity regions in the context of handshake protocols

Firstly feedback structures require an additional storage element to complete the dependencies required by their combinatorial operation. More detail on this scenario is discussed in Section 4.4.3. The second motivator was decoupling performance. The previous fully decoupled controller discussed in Section 4.5.2.3, in light of the performance difference from the mousetrap controller, is hindered by the fundamental coupling of a pipeline:

> "One pipeline stage may store a new data token from its predecessor stage if its successor stage has input and stored the data token that the pipeline stage was previously holding" [83]

i.e. the performance of the fully decoupled controller is still limited by the dependencies of operation, blocking or starvation of a pipeline stage will still occur. Adding an additional register relieves these dependencies further. Combinatorial operations can execute in parallel, allowing a bundled data implementation to approximate true data dependent execution.

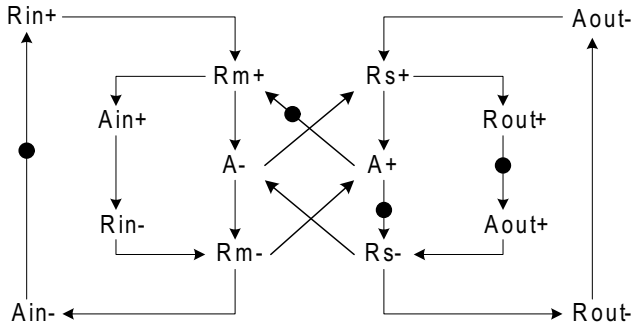


Figure 4.70: AACIF Controller STG

The motivation for using registers as opposed to latches has more implementation origins. Firstly the storage elements in an FPGA are reconfigurable to either registers or latches. There is no area penalty (the common motivator in ASIC implementations) in using a register rather than a latch. Most clocked designs in FPGA devices will naturally use registers with low skew clock inputs and use latches to implement incomplete combinatorial assignments. Edge-triggered components, such as registers, are effectively composed of level-triggered components (i.e. latches) with timing constraints on their setup times and transition rates. These constraints allow the design of the controller to be simpler. The only consideration is the position of the rising edge for the clock input in relation to the handshake signals. A controller for a latch may need more logic to ensure the status of the latch (open or opaque) is managed succinctly. The final motivator for using a register is power savings. Normally open latches have been shown [68] to have greater power consumption (in return for decreased latency) whilst normally opaque latches have lower power consumption because they limit the number of transitions in nearby combinatorial logic. Registers are normally opaque meaning there is very few advantages in using latches in this asynchronous context.

Flow equivalence has been addressed in this controller by altering the sequence of handshake events coupled with the naturally opaque nature of registers. For the STG shown in Figure 4.70 the initial state of the controller indicated by the dots is $Rout+$, $A+$ and $Ain-$. This means that each controller we always push date to its successors. This allows the output environment to control the data rates through the circuit. In the context of the flow equivalence discussion in Section 4.5.2.4 where the clock enable line would invalidate the output data transaction, this controller would still intiate a transaction, maintaining the value and order of data signals. Designing the controller to operate in this manner with registers restricts the data validity regions to early or broad schemes.

With the output handshake starting first, data on the output of the secondary register is passed to the successor stage first. Only then will an input handshake from the predecessor stage be allowed and data can be captured by the primary register ($Rm+$). This new input data will only be captured by the secondary register ($Rs+$) once the output handshake is complete ($Aout-$). This controller was used in all subsequent simulations and performance measurement metrics. Further detail on its response times etc is contained in the Analysis and Results Section. In comparison to the previous controllers the timing waveforms from this controller (in Figure 4.71) are unsurprisingly substantially denser than the fully decoupled controllers with around 50 transactions happening at the output of the pipeline within 500ns. An uncompressed waveform showing the full 50 transactions can be found in Appendix C.2.2. This implementation has demonstrated substantially better performance that other controllers in the same testbench.

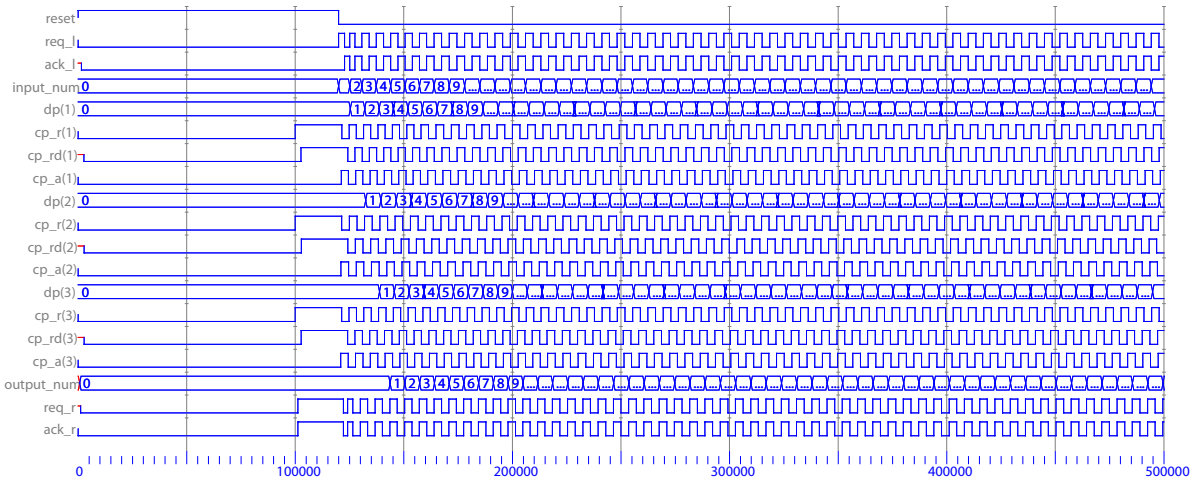


Figure 4.71: AACIF Controller Waveform

4.6 Analysis and Results

Two test circuits have been used to evaluate various aspects of the design. As part of an industrial FPGA based video system used to condition and process multiple streams of 24bit video data, a 422 to 444 format converter and a 444 to RGB format converter [80] are required. These provide a simple

mechanism to evaluate how circuit size and structure affect the performance of the control network. The 422 to 444 circuit only performs data duplication and realignment utilising approximately 10 registers and 30 look-up tables with clock enable logic. The 444 to RGB is a significantly larger circuit utilising over 120 registers and look-up tables as well as 5 embedded multipliers and feedback structures that challenge the conversion tool in a far more expansive manner. In using these two different circuits we can examine the effect of more resource competition and constraint interactions. The test circuits were implemented solely on the device without any testbench circuitry that could skew the results. Although this may affect the performance of the overall circuit due to I/O delays, there is no impact on the local performance of the asynchronous structures evaluated in the following sections.

4.6.1 Timing Results

4.6.1.1 Delay Chain Accuracy

The first area for evaluation is the delay chains, and how close they match the worst case delay through their corresponding combinatorial logic block. Delay chain accuracy was measured using timing groups placed in the UCF files that accompany the altered EDIF files. These timing groups can subsequently be targeted by static timing analysis tools (included the Xilinx tool suite) after the circuit has been placed and routed. The timing information of combinatorial clouds and delay chains can then be directly compared. This information is the key parameter that is passed back up the design flow to the asynchronous structure insertion within the design flow shown in Section 4.2.

Figure 4.72 and Figure 4.73 show the percentage difference between the worst case delay through the data path and the delay through the corresponding delay chain. The delay chain length (i.e. the number of components connected in series) has a 'x' prefix if XOR carry chain logic is used instead of look-up tables. The 422 to 444 converter circuit shows a relatively accurate set of delay chains compared with wide variety of differences for the 444 to RGB circuit. There are occasions in each circuit where delay chains are dominated by routing delay rather than logic delay. This means that the total delay of some delay chains does not correspond to the expected delay from a particular length of delay chain, making the tuning and constraint process even more important. Ideally we would prefer the difference to be zero, however there are a few occasions where the delay chains significantly overshoot and other occasions where they undershoot. A circuit will not function correctly if a delay chain undershoots, thus additional delay elements must be added to restore functionality.

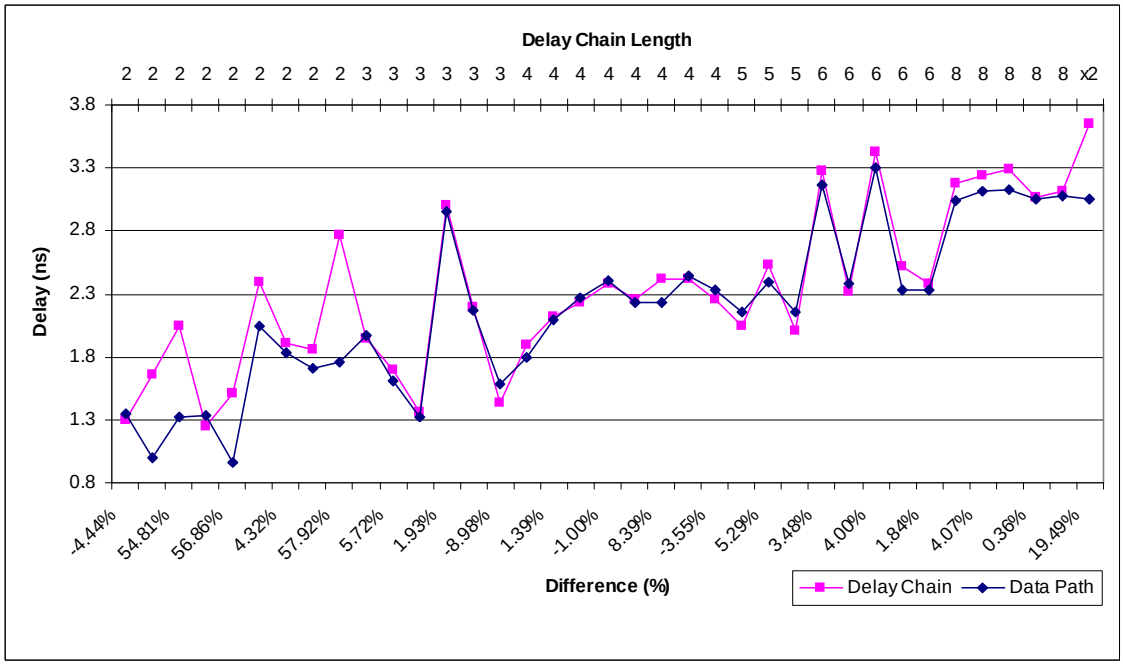


Figure 4.72: Delay Chain Accuracy of 422 to 444 Circuit

There are only 4 delay chains in Figure 4.72 that are significantly larger than the worst case delay through their corresponding logic block. These are due to the minimum delay we can create using 2 look-up tables. The large negative differences seen in Figure 4.73 rely on the fact that the request signal still has to go through another look-up table within the controller and then upwards to the clock pin of the register. We have allowed a maximum difference of -15% to be absorbed in the controller. This is due to controller delay from a transition on the input request port to a transition on the clock input of the first register. This delay is sufficient to move the data out of the setup and hold region for that register, providing the delay chain with additional margin.

The embedded 36 bit multipliers are the only component that force an over estimation. To optimise synchronous designs each multiplier has a register built into the output pins. This means that the static timing analysis tool is only able to measure the delay up until the inputs pins of the multiplier. There are 5 multipliers in the 444 to RGB circuit and Figure 4.73 shows 5 occurrences that have delay chain lengths over-estimated to 3 XOR gates to accommodate for this additional delay. Overall the 444 to RGB circuit has larger delays requiring XOR based delay chains. This comes at the cost of delay overhead which explains the greater variations in accuracy compared to the smaller 422 to 444 circuit.

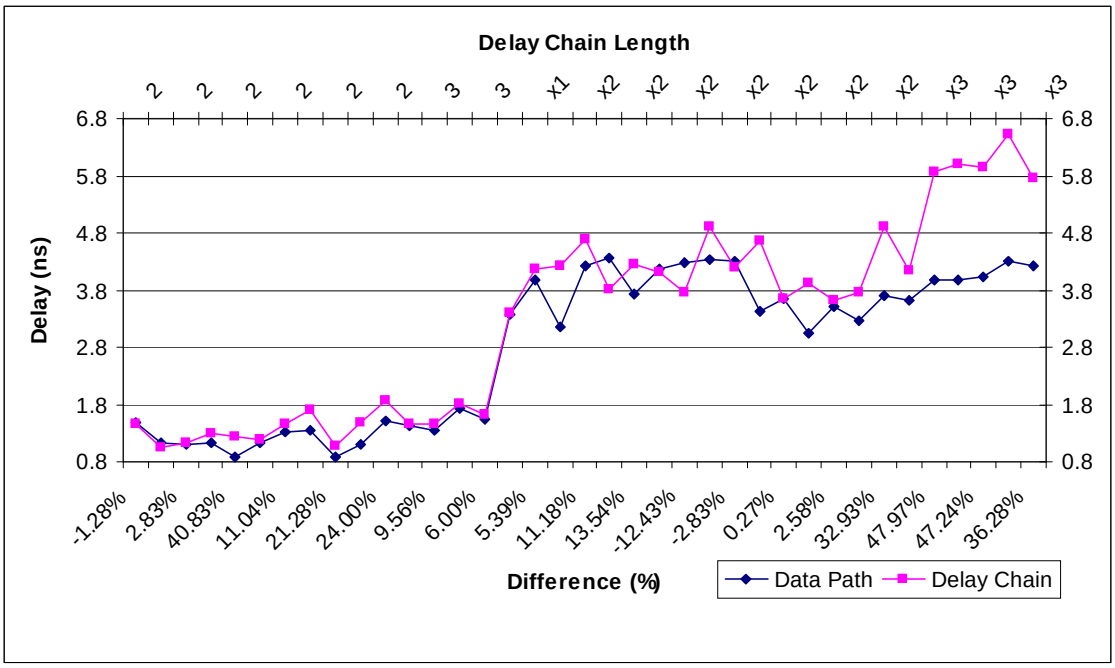


Figure 4.73: Delay Chain Accuracy of 444 to RGB Circuit

4.6.1.2 Clock Skew

Perhaps the most difficult timing issue to solve is the variation in delay(T_{sk} in Figure 4.74) across the nets that drive the clock pins of each register group. Synchronous FPGAs have a few low skew dedicated clock trees to ensure that the rising edge of the clock propagates to all registers in the circuit within a certain time. Since the granularity of the conversion is at the datapath level, the number of registers clocked by different nets means that using the dedicated clock tree is impractical. There are two ways in which we minimise clock skew between the registers:

1. Secondary registers are locked very close to the primary registers, this ensures that the divergence of the clock net is limited between single bit register pairs.

2. Secondly, any remaining skew is absorbed within the controller itself where an internal delay (LUT1_L component shown in Figure 4.69) can ensure that the secondary register never receives a rising edge until all of the primary registers have received theirs.

Figure 4.74 shows the main contributors that affect the position of data relative to rising edge on the clock pin. In order to guarantee successful operation the Rm + T_{dp} must be equal or less than Rs + T_{cd} assuming the switching delay of the register [86] is negligible. Synchronous FPGA designs that approach the timing limitations of the device often suffer from setup and hold violations, i.e. situations where stable data and a clock edge do not align resulting in a register capturing fluctuating data and entering a metastable state.

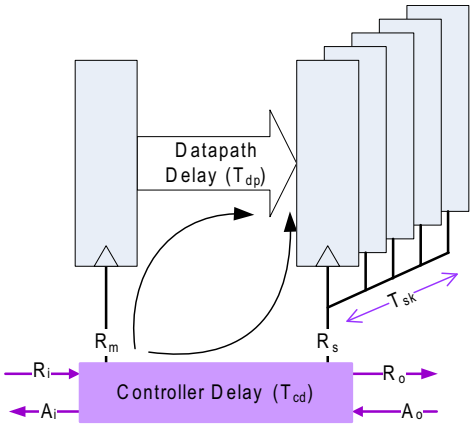


Figure 4.74: Balancing Data Path and Controller Delay

In order to evaluate the potential risks of timing violations, static timing analysis tools were also used to determine the propagation delays through the critical paths. In a similar manner to delay chain measurement, nets were grouped into timing groups which could then be probed to extract the critical timing information to assess timing violations. Figures 4.75 and 4.76 show the difference in delay between Rm and Rs signals for each controller in each circuit. Hold violations occur when the delay through the controller from Rm+ to Rs+ is smaller than the time it takes for the data to reach the input of the secondary register. Ideally we would prefer a negative difference between the times.

There are two observations from the data presented in each Figure. In the smaller 422 to 444 circuit where there was a lesser contention for resources, we are more likely to have a delay imbalance in favour of the secondary registers mitigating setup and hold issues. Where there are less routing resources available, there is a higher chance of setup and hold problems as shown by the predominantly positive differences in Figure 4.76. The highest difference between Rm and Rs is 0.662 ns. The internal delay within the controller consists of 3 look-up tables and 2 latches, which including routing is more than sufficient to re-balance Rs $+T_{cd}$ to match Rm $+ T_{dp}$ (from Figure 4.74) assuming the data path delay is minimal due to the mapping constraints.

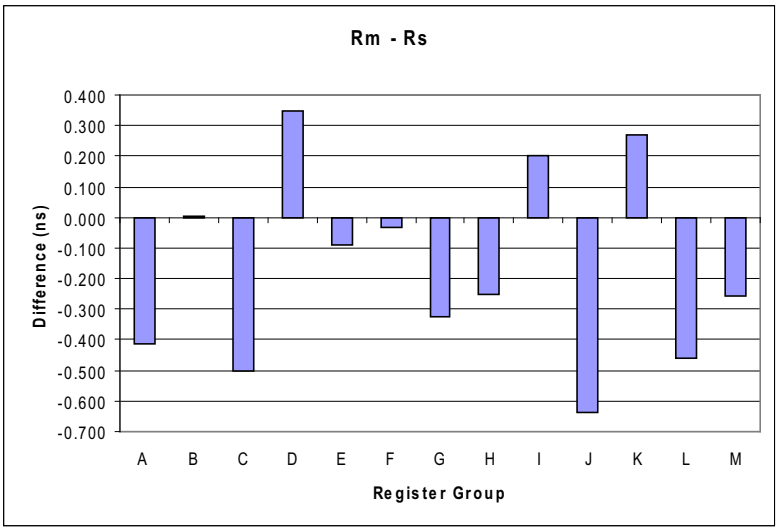


Figure 4.75: Matching Clock Nets of 422 to 444 Circuit

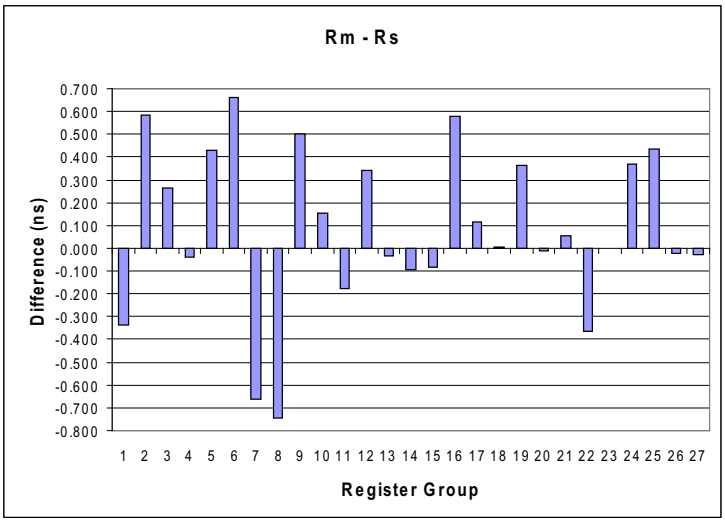


Figure 4.76: Matching Clock Nets of 444 to RGB Circuit

4.6.1.3 Controller Propagation Delays

The next big performance inhibitor on the control network is the delay through the register controllers. Place and route tools have no guidance on how to arrange configurable logic block components optimally for multiple paths and so the consequence of this is that the delays through the critical paths in the controller can be excessive. With the STG shown in Figure 4.70 in context, there are two key parameters to monitor that:

1. Response Time (*Rt*): The time from the input request line going high Rin+ till the output request goes high Rout+

2. Acknowledge Time (*At*): The time from the input request line going high Rin+ till the input acknowledge goes high Ain+

The propagation delay from the primary register capturing data (Rm+) to the secondary register capturing data (Rs+) is vitally important. If the routing delay between the two registers is excessive then data could be lost or setup and hold issues could occur as the delay from Rm+ to Rs+ could be less than datapath delay for that stage. To resolve these potential issues the controllers have an explicit implementation so that look-up tables are confined to a specified area. The look-up tables and latches of each controller have mapping constraints embedded within their EDIF declarations. This means the contributions that the response and acknowledge time make to T_{cd} are prioritised and minimised.

The back annotation process was used to verify the propagation delay through each controller. This allows an accurate timing measure of the performance on the intended device. In Tables 4.10 and 4.11 we compare the constrained controllers[8] in their first iteration of Rt and At (subscript i) against their final iteration (subscript f). These measurements were recorded from back annotated simulations. Here we can detect the edge transitions and record accurate times between them.

Ctrl	At_i(ns)	At_f(ns)	$\triangle At$	Rt_i(ns)	Rt_f(ns)	$\triangle R_T$
A	0.555	0.516	-7.%	2.562	2.886	12.7%
B	0.966	0.516	-46.6%	5.085	2.242	-55.9%
C	1.004	0.516	-48.6%	22.872	20.179	-11.8%
D	1.306	0.977	-25.2%	3.691	2.832	-23.3%
E	1.459	1.073	-26.5%	3.457	3.107	-10.1%
F	1.391	0.99	-28.8%	3.459	3.318	-4.%
G	0.823	0.712	-13.5%	3.411	2.866	-16%
H	0.756	0.536	-29.1%	2.701	2.112	-21.8%
I	1.561	0.734	-53%	19.37	19.638	1.4%
J	0.7	0.653	-6.7%	2.512	2.357	-6.2%

Table 4.10: Controller Delays of the 422 to 444 circuit

Incremental compilation allows the optimal controller placement in relation to the registers and delay chains because it retains all the place and routing information for parts of the circuit that are unchanged. During the last iterative cycle the constraints are relaxed to allow the routing process to have more freedom in optimising net delays. In both tables, columns $\triangle At$ and $\triangle Rt$ indicate the change in Acknowledge time and Response time respectively. For the majority of controllers, the negative percentages indicate that the time has been improved, in some cases as much as 55.9% however there are controllers where timing has increased by as much as 41.7%.

[8]Note: there are three less controllers listed than Section 4.6.1.2 because controllers that duplicate clock enable functionality have multiple input requests making timing measurement dependant on multiple arbitrary sources

Ctrl	At_i(ns)	At_f(ns)	$\triangle At$	Rt_i(ns)	Rt_f(ns)	$\triangle R_T$
1	1.394	1.09	-21.8%	3.684	4.831	31.13%
2	0.798	0.749	-6.2%	2.975	3.136	5.4%
3	0.802	0.684	-14.7%	3.212	3.41	6.2%
4	0.763	0.798	4.6%	3.031	3.022	-0.3%
5	0.7	0.798	14%	3.096	3.693	19.3%
6	1.101	0.615	-44.1%	3.32	2.836	-14.6%
7	1.519	0.712	-53.1%	3.896	2.968	-23.8%
8	0.776	1.012	30.4%	3.471	3.375	-2.8%
9	1.188	1.06	-10.8%	3.906	3.888	-0.5%
10	0.719	0.719	0.00%	3.126	3.126	0.00%
11	0.702	0.555	-20.9%	3.461	2.943	-15%
12	0.779	0.817	4.9%	3.388	3.648	7.7%
13	0.823	0.711	-13.6%	3.13	3.283	4.9%
14	0.634	0.705	11.2%	3.018	3.109	3%
15	0.761	0.884	16.2%	3.345	3.244	-3%
16	0.802	0.802	0.00%	3.063	3.182	3.9%
17	1.055	1.119	6.1%	3.546	3.454	-2.6%
18	0.659	0.616	-6.5%	3.478	2.864	-17.7%
19	1.151	1.027	-10.8%	4.823	4.48	-7.1%
20	1.485	1.015	-31.7%	4.358	3.519	-19.3%
21	1.383	1.183	-14.5%	3.644	3.365	-7.7%
22	0.659	0.78	18.4%	2.87	4.067	41.7%
23	0.821	0.821	0.00%	2.998	3.214	7.2%
24	1.001	0.781	-22%	3.251	2.988	-8.1%
25	0.567	0.567	0.00%	2.917	2.781	-4.7%
26	0.749	0.543	-27.5%	3.136	2.822	-10%
27	0.719	0.821	14.2%	3.109	3.239	4.2%

Table 4.11: Controller Delays of the 444 to RGB circuit

4.6.2 Utilisation Results

FPGA resource utilisation is a key factor in the ability of a device to meet the performance requirements of a particular circuit. A FPGA which does not have sufficient capacity will suffer from resource contention. Since the slowest part of a synchronous circuit governs the throughput of a circuit, resource contention can drammatically affect the performance of circuit purely by poor allocation of resources. Resource utilisation is worthwhile avenue which demonstrates the resource efficiency of this conversion tool, and an indirect performance measurement.

Historical asynchronous ASIC designs have always suffered from increased silicon usage than their synchronous counterparts. In recent times Handshake solutions [89] and Desynchronisation [66] have shown comparable silicon use to synchronous circuits, however in the FPGA environment this difference is expected to widen further. The optimisation of FPGAs (with the exception of Achronix [88]) for synchronous circuits means that asynchronous implementations will always be inferior to synchronous circuit in terms of resource utilisation.

Resource Summary for 422 To 444			
Logic Utilisation	HASTE	AACIF	Sync
Total Number of Slice Registers	83	145	9
Number used as Slice flip flops	83	86	9
Number used as Slice latches	0	59	0
Total number of 4 input LUTS	216	255	26
Number used as logic	216	251	26
Number used as route-thru	0	4	0

(a) 422 to 444 Resource Comparison

Resource Summary for 444 To RGB			
Logic Utilisation	HASTE	AACIF	Sync
Total Number of Slice Registers	261	453	104
Number used as Slice flip flops	261	338	104
Number used as Slice latches	0	115	0
Total number of 4 input LUTS	491	369	120
Number used as logic	485	277	120
Number used as route-thru	6	92	0

(b) 444 to RGB Resource Comparison

Table 4.12: Resource Utilisation Comparison

A useful comparison however is in the resources consumed by different asynchronous circuit styles on FPGAs. In this situation Haste from Handshake solutions has been used to provide a valid benchmark on the resource efficiency of the conversion process. Table 4.12 shows the resource results of the same functionality synthesised from synchronous VHDL, the Haste FPGA prototype flow and the proposed conversion process. The synchronous utilisation has only been added as a reference for the asynchronous results, as mentioned earlier, there is significantly more overhead in the asynchronous implementations. The results have been extracted from the reports produced by the mapping stage that forms a significant part of the place and routing of a circuit design.

In the context of the asynchronous resource utilisation there is an interesting trade-off between the number of look-up tables used and the number of registers used between the Haste implementation and the AACIF implementation. In the case of smaller circuits as demonstrated with the 422 to 444 circuit the Haste implementation has more efficient resource use with approximately 3% less register use and 13% less look-up tables used as logic. This is due to the increased complexity of the asynchronous controllers used in the asynchronous conversion. With larger circuits as shown in Table 4.12b the larger computational requirements override the increased register use. Where the AACIF version uses 42% less look up tables as logic, it uses 29% more registers that the Haste implementation. Here we begin to see the benefits of maintaining the logic optimisation performed during synthesis.

We conclude that there is more computational space available in the AACIF process, the Haste process uses more look-up tables in its resource allocation, limiting the combinatorial depth available. The AACIF implementation uses more registers but provides greater combinatorial flexibility.

4.6.3 Power Spectrum Analysis and Core Voltage Stability

Most large, dense IC's are designed to implement specific tasks. As a result their power supply requirements are within a certain range. Since FPGAs can reconfigure to any given application, their transient switch currents can vary dramatically. This flexibility means that distributing power to an FPGA must be based on a worst cast scenario [65]. The power supply decoupling network must be tuned to the specific transient current needs; otherwise ground bounce and power supply noise will exceed the limits that allow the devices to operate. In most case this is ±5% of the nominal Vcc. Excess on the power rails leads to jitter on all signals in and around the device, leading to inconsistent operation. The transient switching currents in synchronous designs are mostly focused around components at the printed circuit board (PCB) level and the clock tree internal to the FPGA. Being able to remove the switching currents or reduce the switching currents can have an effect on PCB design and component usage.

This section discusses the impact of implementing AACIF circuits on FPGAs with respect to the transient currents that are required from the core voltage rails of the FPGA.[74]

4.6.3.1 Power Supply Network Background

Low frequency variations in power consumption are normally the result of large portions of a device being enabled or disabled. These variations are normally in the millisecond range. High frequency variations are the the result of switching events internal to the FPGA and usually happen on the scale of the clock frequency and the first few harmonics [67]. This has been observed with great effect on microprocessors where asynchronous devices show a lower average noise and less spikes than their synchronous counter parts performing the same operations [72] [77].

Since Vcc is fixed, changing power demands results in a changing current demand. When the current draw changes, the power distribution system cannot respond instantaneously due to its inherent inductance. Although resistance is a factor, inductance is dominant impedance. For a short time before the power supply can adapt, the voltage at the device changes. This is where power supply noise appears. Voltage regulators can only maintain a constant voltage for events at frequencies from DC to a few kilohertz. For all transient events that occur above this range, decoupling capacitors are required to provide the transient current at particular frequencies. This has the effect of reducing the size of the transient currents in the power supply network. Ideally one capacitor should be sufficient however real capacitors have lead inductance and equivalent series resistance that limit their useful frequency range. If the impedance of the power supply network is to remain constant multiple decoupling capacitors are required to reduce the impedance at the desired switching frequencies.

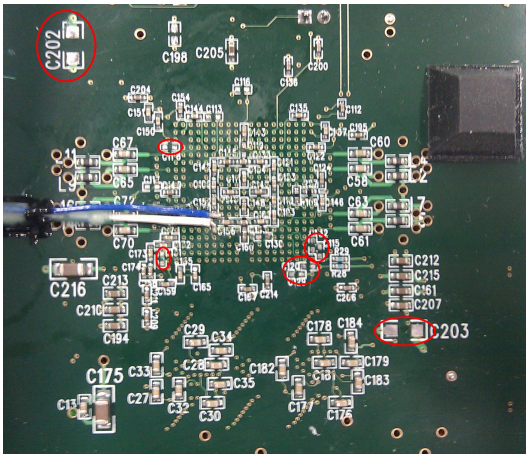


Figure 4.77: Scope Probe with Decoupling Capacitors Removed

4.6.3.2 Measurement and Setup

The aim of this series of measurements is to determine how much noise is introduced to the power supply network by the transient switching currents from the core voltage lines of the FPGA. In this context decoupling capacitors hinder the accuracy of our measurements. To alleviate this and gain a clear reading from the core voltage lines, all of the decoupling capacitors were removed and the oscilloscope probe attached to the pads closest the FPGA to minimise the inductance loop. The removed capacitors are shown in Figure 4.77 along with the soldered probe. A high bandwidth 1 GHz Le Croy oscilloscope [87] was used along with a passive probe (shown in Figure 4.77) which it was assumed had sufficient bandwidth for the measurement with a 50Ω cable and a minimal ground loop. A picture of the developement board from the other side with the connection to the oscilloscope is shown in Appendix C.3. The oscilloscope is capable of performing Fourier analysis in real-time and so it was preferential to use this method to capture the noise spectrum rather than using a spectrum analyser.

Within the FPGA two test benches were constructed to feed the circuits with sufficient data to exercise their full functionality. For simplicity and minimal footprint, the synchronous circuits were driven by a ROM containing video patterns. The asynchronous circuits, although utilising the same ROM required an additional wrapper to push data through the circuit. Before measurements were obtained, a testbench spectrum was captured and subsequently subtracted from the main measurements. An additional background measurement was taken to eliminate any induced voltages. The background noise spectrum is shown in Figure 4.78 where we can see a number of noise spikes or induced voltages in the core voltage lines without any power to the board. The highest 83.335 MHz spike can be attributed to the oscilloscope used. As this scope has a PC based architecture the front side bus is likely to be running at this frequency. The additional spikes are of unknown origin.

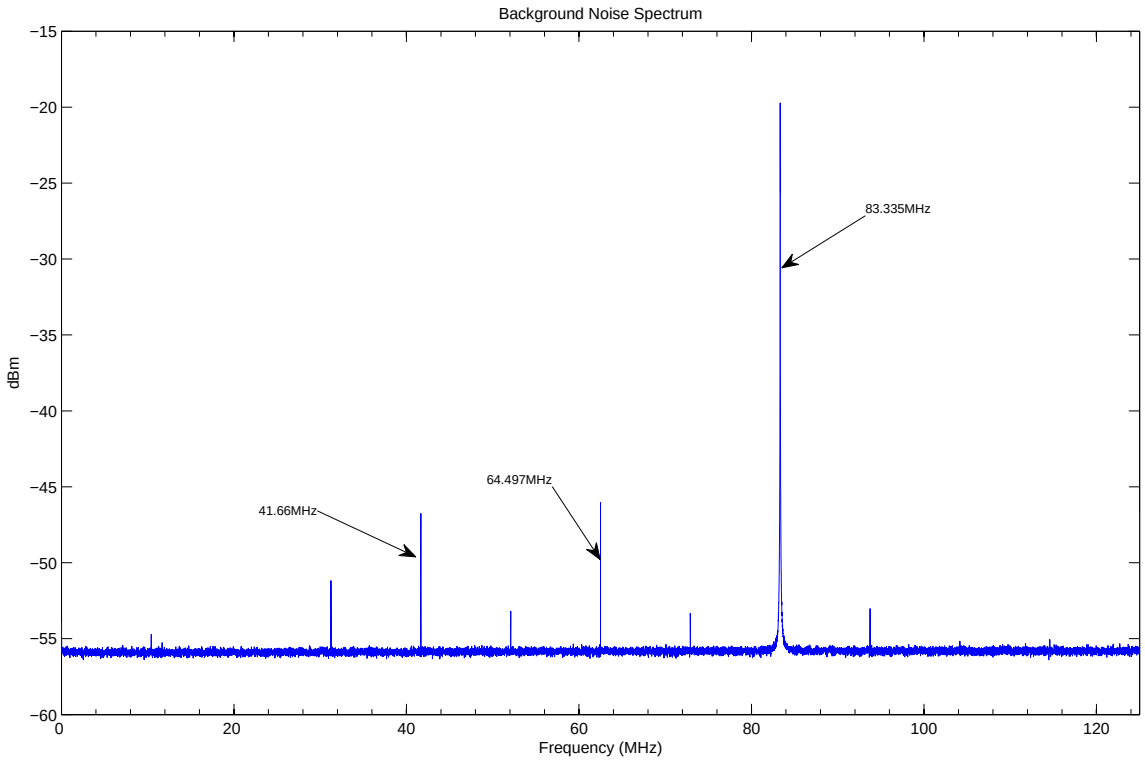


Figure 4.78: Noise Spectrum of the Background

4.6.3.3 Results

The first spectra shown in Figure 4.79 is the synchronous implementation of the 444 to RGB circuit. A similar spectra for the the 422 to 444 circuit was also produced and can be found in Appendix C.4. The background noise has been removed leaving only the perturbations caused by transient currents in the core voltage lines. In both circuits we can identify the 610kHz spike as switching currents from the board regulator on the PCB. The next substantial spike is the clock used in the circuit at 12.5MHz, along with harmonics at 25 MHz and beyond, The largest negative spike is the result of increased background noise at 83.335 MHz as discussed previously. The remainder of the spectrum is the result of individual circuit events around the clock requiring current to transition. These characterise the operation of the circuit from a power consumption perspective.

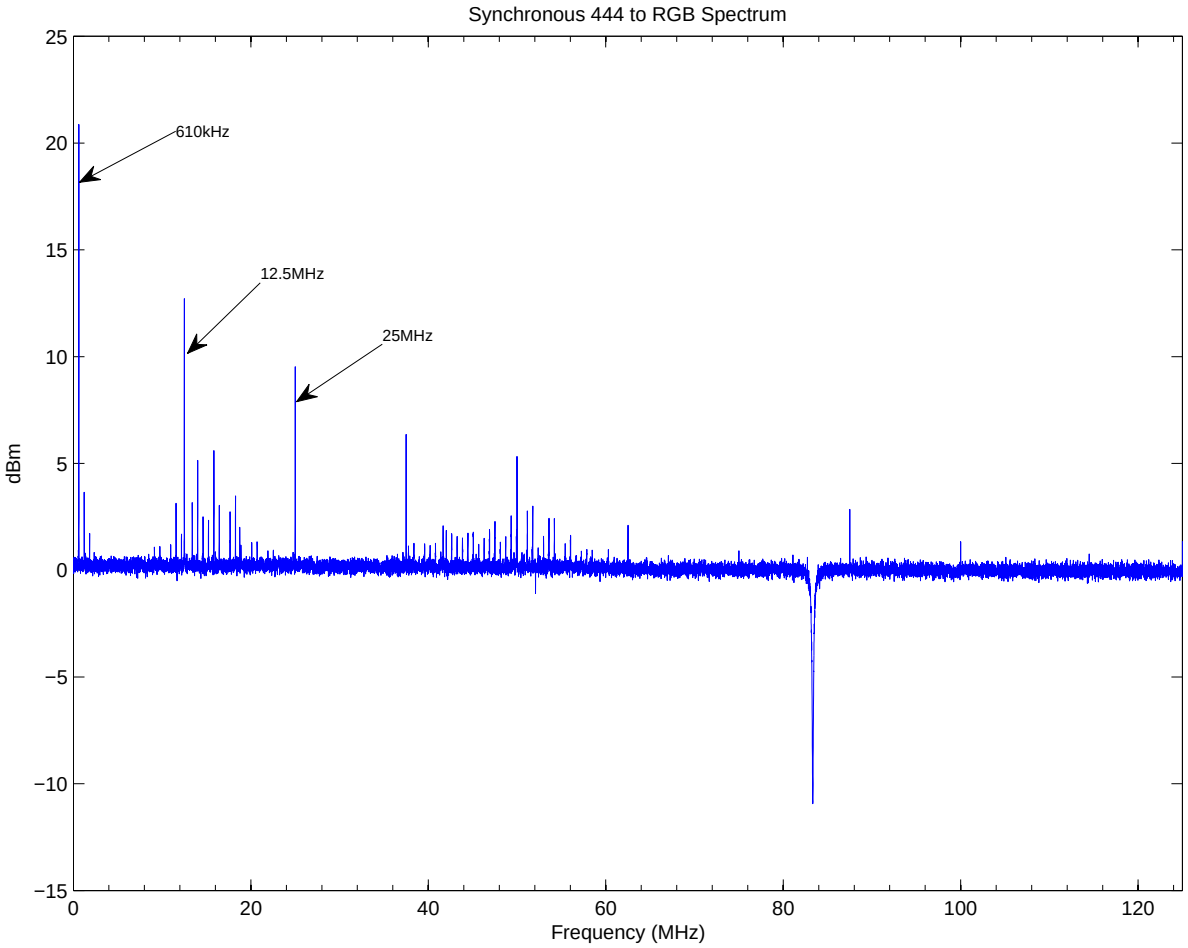


Figure 4.79: 444 to RGB Circuit Noise

On their own the spectra do not reveal too much, however, overlaying the synchronous spectra on top of the same measurements from the asynchronous implementations reveal the benefits from AACIF circuits. The asynchronous measurements have also had the background perturbations removed. Figure 4.80 has the AACIF implementation spectra in pink. Almost every spike in both figures from the AACIF implementation is less than the synchronous implementation, especially the clock harmonics. We still see the board level spikes from the switch regulator, however the maximum spike after the regulator is around 6dBm for the AACIF implementation and 13dBM for the synchronous implementations.

What we see is the extent to which the AACIF circuits affect the power supply network. The clock spikes and their harmonics are not present, meaning there is no need for the board level decoupling capacitors to provide additional charge[9]. This has implications for the number of components used in the board assembly and the overall cost of manufacture, a significant difference from the synchronous implementations.

[9] there is on-chip decoupling that could also have an impact here

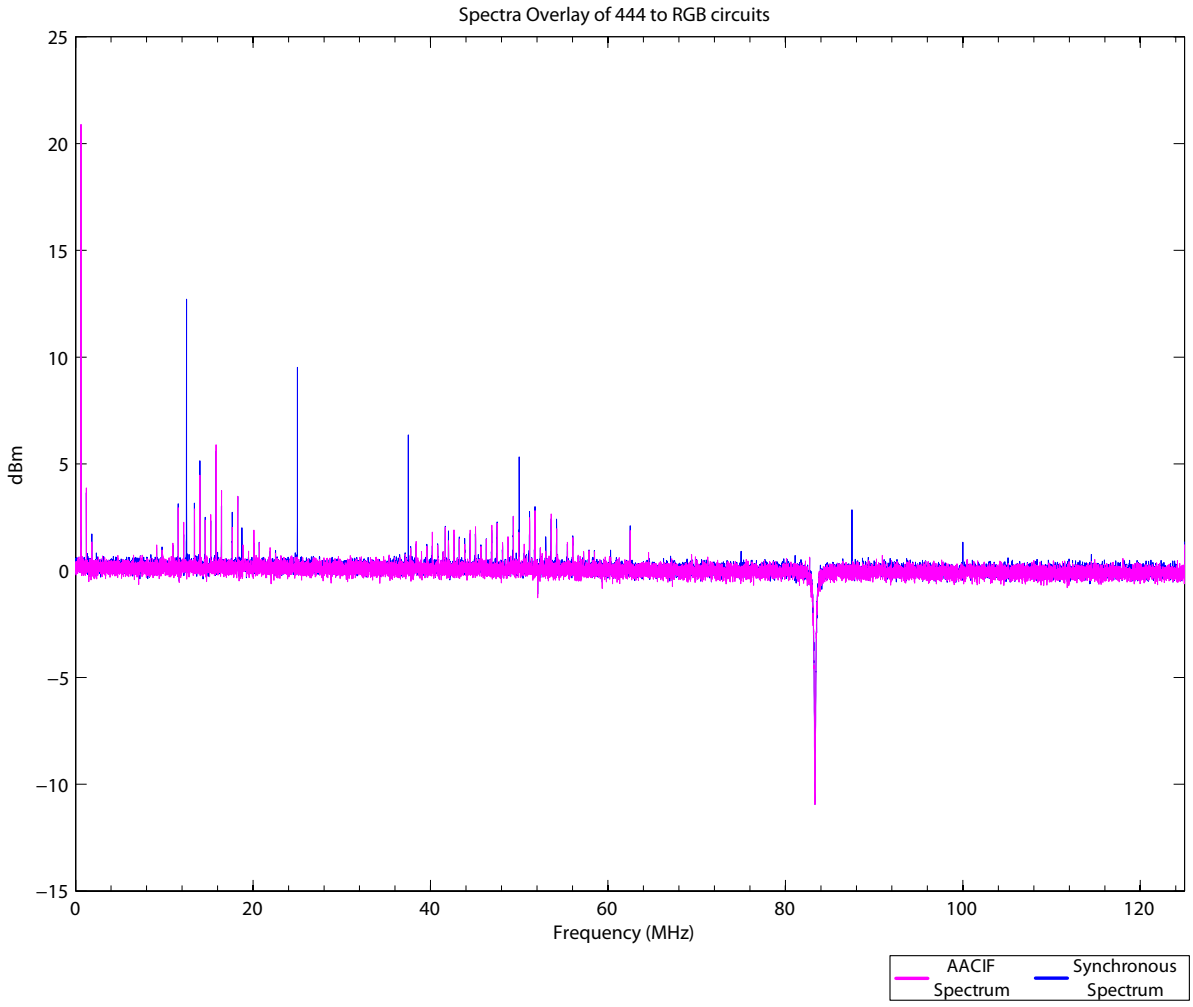


Figure 4.80: 444 to RGB Circuits Noise Spectrum Overlay

4.7 Conclusions and Future work

The work presented in this document provides a complete picture on how to automate the implementation of asynchronous circuits on FPGAs. The revised design flow builds upon efficient, industry proven synthesis tools to optimise datapaths. An iterative approach allows the asynchronous control network to be tuned and optimised to complement the already optimised datapath. The design flow provides an additional implementation option for synchronous designs.

The synchronous conversion algorithm provides a consistent repeatable process to implement asynchronous circuits on FPGAs. The conversion algorithm starts from a synchronous EDIF circuit description and rebuilds a datapath abstraction from its low level primitive description. The clock tree is then removed and an asynchronous control network is inserted. Due to its abstraction approach in mapping circuits, the algorithm has sufficient flexibility to function with multiple synthesis tools that produce EDIF netlists. Basing the conversion on an open industry standard has been key to its success. Although targeted to a specific FPGA device vendor, the use of custom libraries mean that the conversion algorithm can be targeted to many other vendors. The key to the performance of this conversation is the integration of constraints into the asynchronous EDIF netlists.

Using a library of pre-routed components to implement circuits asynchronously is the cornerstone of this work. It means that downstream place and route tools treat the asynchronous control network in the same way as embedded components such as ethernet controllers or embedded memory blocks. The library contains numerous delay chain and controller implementations, designed with different priorities and properties. Along with the novel constraint managment, a new asynchronous controller was added to the library. This controller demonstrated significant performance whilst using minimal resources and maintaining data equivalence to the synchronous design. The library allows designer flexibility in tailoring the control network to resource utilisation or timing performance. Using a common library and merging the used components into the source EDIF netlist provides compatibility with existing place and route tools.

The results of these efforts summarise to timing, utilisation and noise spectrum benefits. The analysis of proven industrial video circuits indicate that the conversion has the ability to find the optimal timing performance for the crucial asynchronous control network. The relevant sections have demonstrated the timing benefits of embedded constraints through delay chain accuracy and controller propagation delay improvements. Device utilisation, although greater than the synchronous equivalent has shown greater flexibility and effectiveness in comparison to the only other asynchronous tool capable of implementing asynchronous circuits on FPGAs. The most significant result was the reduction in the noise spectrum from the FPGA. Removing the clock from the system has reduced the noise on the core voltage lines, reducing the number of compensatory components in the power distribution system of the PCB.

4.7.1 Further Considerations

There have been points where other avenues of research have been assigned a lower priority in the context of achieving the overall goal. This section discusses the viable continuations and improvements on the work presented here. Firstly, the accuracy of delays could be improved further with investigations into utilisation of programmable interconnect points. These provide the flexibility to control the paths that the nets take through the routing matrices of FPGAs. The algorithms used by synchronous tools to balance routing delays are not applicable to asynchronous circuits, and investigation into utilising these points could result in further timing benefits for asynchronous control networks. Another avenue of interest was the granularity of pipelining. The clock trees within FPGA devices have very low skew paths throughout the entire device. Large datapath widths would benefit from using these clock trees rather than the local routing resources. Restricting this work to 4-input look-up table based devices was an imposed constant in developing the conversion algorithm. Since that decision was made new devices have emerged with 6-input look-up tables. Developing a library to suit these devices would provide a useful insight into the effect on overheads for the asynchronous control network. Finally, the overheads associated with an asynchronous control network in an FPGA should be investigated further.This work should establish if introducing timing assumptions or designing a generic controller structure based on look-up tables can tackle synthesis overheads.

4.8 References

[65] M. Alexander. (2005, February) Power Distribtuion System(PDS) Design: Using Bypass/Decoupling Capacitors (XAPP623). Xilinx Ltd. Last Accessed: March 2011. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp623.pdf

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

[67] R. M. Easson, "Analytical Edge: Essential High-Speed PCB Design For Signal Integrity," lasts Accessed: November 2011 http://www.analytical-edge.com.

[68] A. Efthymiou and J. Garside, "Adaptive Pipeline Structures for Speculation Control," in *Proceedingsof the 9th International Symposium on Asynchronous Circuits and Systems*, May 2003, pp. 46 – 55.

[69] Elgris Technologies. (2005) Edif overview. Last Accessed: March 2011. [Online]. Available: http://www.elgris.com/content/edif_overview.html

[70] D. Flanagan, *Java in a Nutshell - A Desktop Quick Reference: Covers Java 5.0*, 5th ed. O'Reilly, 2005.

[71] S. Furber and P. Day, "Four-phase Micropipeline Latch Control Circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 2, pp. 247–253, June 1996.

[72] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, "An Asynchronous Low-Power 80C51 Microcontroller," in *Proceedingsof the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1998, pp. 96 –107.

[73] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann, "Polychrony for system design," *Journal for Circuits, Systems and Computers*, vol. 12, pp. 261–304, 2002.

[74] H. Johnson and M. Graham, *High-Speed Digital Design*. Prentice Hall, 1993.

[75] H. J. Kahn and R. F. Goldman, "The Electronic Design interchange Format EDIF: Present and Future," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*, 1992, pp. 666–671.

[76] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*, 3rd ed. Springer Publishing Company, Incorporated, 2007.

[77] J. Kessels and A. Peeters, "The Tangram Framework: Asynchronous Circuits for Low Power," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2001, pp. 255 –260.

[78] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, 2nd ed. Addison Wesley Longman Publishing Co., Inc., 1998, vol. 3.

[79] J. Lui, "Arithmetic and Control Components for an Asynchronous System," Ph.D. dissertation, University Of Manchester, 1998.

[80] C. Poynton, *Digital Video and HDTV Algorithms and Interfaces.* Morgan Kaufmann Publishers Inc., 2003.

[81] P. Seibel, *Practical Common Lisp.* APress, 2004.

[82] M. Singh and S. Nowick, "MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684 –698, June 2007.

[83] J. Sparso and S. Furber, *Principles of Asynchronous Circuit Design - A Systems Perspective.* Kluwer Academic Publishers, 2001.

[84] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, pp. 720–738, June 1989.

[85] M. Wirthlin, B. Pratt, and J. Johnson. BYU EDIF Tools. Brigham Young University. Last Accessed: March 2011. [Online]. Available: http://reliability.ee.byu.edu/edif/

[86] Virtex-II Datasheet. Xilinx Inc. Last Accessed: March 2011. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-ii_data_sheets.htm

[87] *6100A Oscilloscope*, LeCroy, 2007. [Online]. Available: http://www.lecroy.com

[88] (2006) Speedster Asychronous FPGAs. Achronix Semiconductor. [Online]. Available: www.achronix.com

[89] (2001) HT80C51 Microcontroller. Handshake Solutions. [Online]. Available: www.handshakesolutions.com

Part III

Conclusions

5.1 Thesis Summary

In this final section, there is a summary of the core technical reports that make up this thesis and a summary of the main contributions from this work. The outcomes of the work in each technical report are discussed in relation to the industrial motivations and objectives of each body of work.

This portfolio thesis starts with a technical report on region based contrast enhancement on a FPGA platform. This addresses a topic of research which is of interest and debate from academic communities and of considerable interest to industrial bodies. From an EngD programme perspective this is an excellent opportunity to analyse the topic from two viewpoints. Contrast enhancement algorithms are key to many of the products produced by Thales Optronics Ltd. Field programmable gate arrays are also key to many platforms and embedded systems produced by the company. The results of the investigation returned an implementation strategy and the key contributing factors in implementing Contrast Limited Adaptive Equalisation(CLAHE) on FPGA image processing platforms. Previous implementations were either not region based, non-comparable or targeted for alternative platforms that were not optimal for implementing CLAHE. The implementation is verified as being acceptable and accurate in terms of the golden reference model used and showed very little computational delay that would potential impact the latency of an imaging system. The design factors are defined to be the ratio of image to region size and the amount of memory allocated for both. It is acknowledged that there are potential optimisations that could be investigated further, however these optimisations would be generic to development of most region based contrast enhancement implementations on an FPGA platform.

With a change of supervision and direction, the project progressed onto technical report 2. Following up on the other aspect of the initial brief, research focused on establishing asynchronous circuits on FPGA devices. The main motivation for this work is to establish if any of the benefits which have been demonstrated for ASIC devices can be applied/transformed to FPGA devices. The investigation surmised that 4-phase bundled data is the most suitable asynchronous design style for FPGAs. This is primarily due to the ability of this asynchronous design style to utilise the industry standard synchronous synthesis tools which are highly optimised for FPGA architectures. This decision provoked the creation of a new FPGA design flow that allowed asynchronous circuits to be formed from two HDL circuit descriptions, one for the datapath and another for the control path. Asynchronous components can now be created using FPGA primitives and collected to form the contents of an asynchronous library. Novel delay chain and asynchronous controller design strategies provide the consistent and repeatable implementations that make up the contents of the library . A simple pipeline was presented which demonstrated the verification methodology of the handshake protocols. The key challenge was to find the common ground between asynchronous design theory and a practical FPGA implementation. The result of this work is an established route to implement circuits asynchronously in an

FPGA. This provided the impetus to proceed with an automated approach which could be used to investigate the potential asynchronous design advantages on an FPGA device.

The third body of work drew upon the success of establishing the correct operation of asynchronous control network components on FPGA devices. An automated design flow is presented (AACIF) which performs a conversion of synchronous FPGA circuits, allowing them to operate asynchronously. This approach was primarily focused on the lower level conversion, presenting a novel methodology which abstracts an RTL datapath from FPGA primitive netlist descriptions. At this stage there are a number of circuit manipulations, including removal of the clock tree, which allow the original circuit to operate asynchronously. In doing so a new asynchronous controller designed specifically with FPGA primitive components was compared against other controller implementations. The results of this work are covered under timing, resource utilisation and noise spectrum benefits. The conversion flow demonstrated that it is able to find the optimal timing performace of the delay chain and asynchronously controllers contained within the library of components. It also demonstrated a more flexible utilisation of primitive FPGA resources. The most significant change is the reduction in noise and harmonics on the voltage rails of the core FPGA logic. This reduction has a direct cost saving for design effort and manufacturing costs of a printed circuit board. There were a number of topics that can be derived from this work, including reducing resource overheads and improved delay matching capabilities, as well as applications for optimising current design flows for an FPGA platform.

In conclusion, this thesis has made a number of contributions to the implementation of imaging algorithms and asynchronous circuits on FPGA devices. This has advanced the capabilities of the sponsoring company, Thales Optronics Ltd, and provided new insights into the implementation capabilities of modern FPGA devices.

5.2 Thesis Contributions

There are number of contributions that this thesis and the work undertaken during the research period have made. The industrial nature of the EngD programme means that the contributions extend wider than the technical elements. The following points summarise the contributions this work has made across the technical and industrial environments.

- Image algorithms research is a continuous task within Thales Optronics Ltd. Many algorithms described at high levels of abstraction are evaluated to assess their suitability and quality in addressing the needs of the products and ultimately the needs of the customer. It is seldom that there is an opportunity to extend the evaluation of an algorithm down to the implementation stages. The effort and expertise required to do so is often allocated onto other tasks, whereas the implementation knowledge is often crucial in assessing if an imaging algorithm will suit a particular product or function. Implementing region based contrast enhancement outside the

limitations of project timescales allowed valuable implementation knowledge to be passed up to those that were evaluating imaging algorithms, making a direct impact and contribution to the effectiveness of those engineering teams.

- Although not contributing to the technical element of this thesis, the business modules were always assessed and considered within the context of the day-to-day running of Thales Optronics Ltd. One class in particular contributed to the internal decision on a crucial piece of software that was key to the company processes. The assignment from the Making Decisions module, primarily used to demonstrate the knowledge of multi-criteria decision analysis, contributed a methodical and thorough solution to the discussion on this particular problem. The recommendation provided transparency for the trade-offs in this decision and decision visibility to management that was not present before.

- The technical contributions being with the low level implementation of Muller C-elements with primitive FPGA components. Isolating the key parameters and embedded constraints to ensure the implementation was consistent and repeatable lead the foundations for further developments in asynchronous controllers. A internal intellectual disclosure application was submitted within Thales Optronics Ltd to protect this work, recognising the novelty and contribution this work has made to asynchronous circuits on FPGA devices.

- Accurate delay chains are fundamental to the operation and performance of bundled data circuits. This work contributed new methods used to create delay chains specifically for FPGA devices adds a unique degree of flexibility, consistency and interchangeability that was not present in previous published works.

- There has been no published mention of techniques used to verify asynchronous handshake protocols on FPGA devices. The approach taken in this work contributes simple on-chip verification of event dependencies and validity regions which are crucial to the operating assumptions of asynchronous handshake protocols.. This ability is outwith the current capability of on-chip logic analysers which are the established verification methods for synchronous circuits.

- One of the key elements in constructing the conversion tools was the EDIF Tools API by the EDIF Team at Brigham Young University. As this work added and challenged the use of this API, a number of bug fixes and design discussions were fedback to the authors providing a modest contribution to the future development and reliability of the API.

- This work has provided a contribution to the implementation possibilities of asynchronous circuits on FPGA devices. This is the only approach that used the EDIF netlist to construct asynchronous control networks out of FPGA primitive components. This asynchronous library

contains circuits that are now part of the FPGA design libraries within Thales Optronics Ltd. This approach to constructing circuits has optimisation applications for other design flows.

- The primary contribution of this work to both the sponsoring company and the wider community is the only conversion tool which targets the implementation of circuits asynchronously on FPGA devices. There have been a few approaches that have demonstrated asynchronous circuits on FPGAs, however none have proposed or developed tools specifically targeted to implementing asynchronous circuits on FPGA devices. The novelty of this work is emphasised with the lack of comparative tools for this technology platform.

Part IV

Additional Material

Appendix A

CLAHE Implementation and Analysis Supplements

A.1 CLAHE Design Tools

The work on CLAHE followed the standard tool flow set out by Mentor Graphics and Thales Optronics. This is essentially the standard tool flow set out by EDA companies but with an accompanying tool suite called FPGA workbench. This suite adds a number of important functions that are lacking from the standard FPGA design flow. The main aim is to contain the EDA tool version and the design code in one manageable location. This approach builds in version management in the form of CVS or subversion for the behavioural design, synthesis constraints, place and route constraints, and all embedded primitive components. Couple this with the encapsulated tool environment of :

1. Mentor graphics HDL Designer 2005.3
2. Mentor Graphics Precision Synthesis 2005.128c
3. Mentor Graphics Modelsim
4. Xilinx ISE m8.2i

allows the entire designer to be repeatable and easily managed for future projects. HDL design is the primary design entry tool, this interface with all other aspects of the design flow and can be used primarily on its own with the help of TCL scripts to setup the various simulation types, synthesis priorities and place and route settings. This was particularly effective when integrating with Chipscope pro in order to verify the correct operation and using the place and route tools to produce the timing information required for back annotated simulations.

A.2 Critical Path Synthesis View

The critical path through the redistribution block has been deliberately simplified in the main text for the benefits of a succinct discussion. The synthesis and place and route tools have calculated that the critical path in the design is within the redistribution block. Figure A.1 shows the critical path spanning the logical histogram bin update calculation and the various surrounding parameters. The primary reason for this path being long is due to the interface to the block RAM, pipelining this calculation to reduce the critical path length would significantly alter the memory interface.

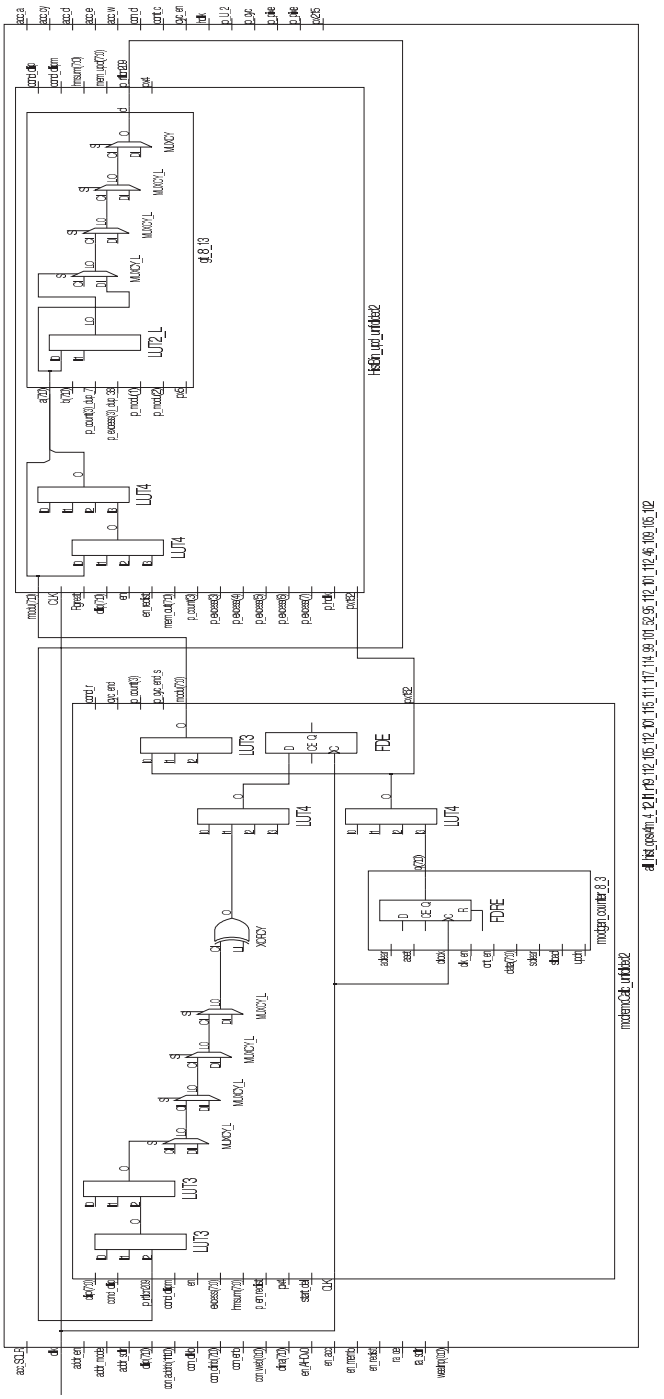
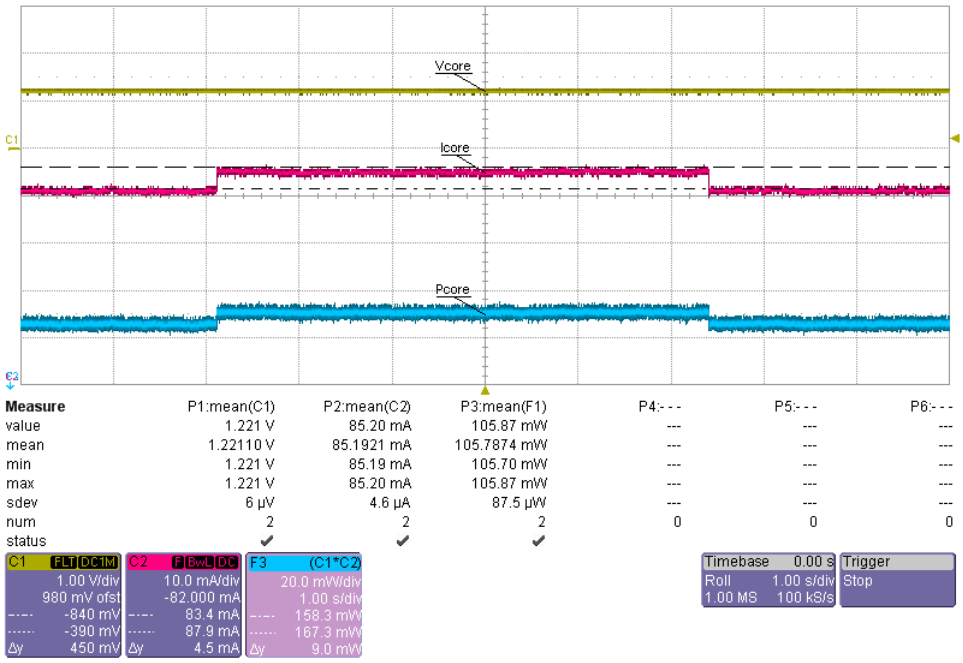


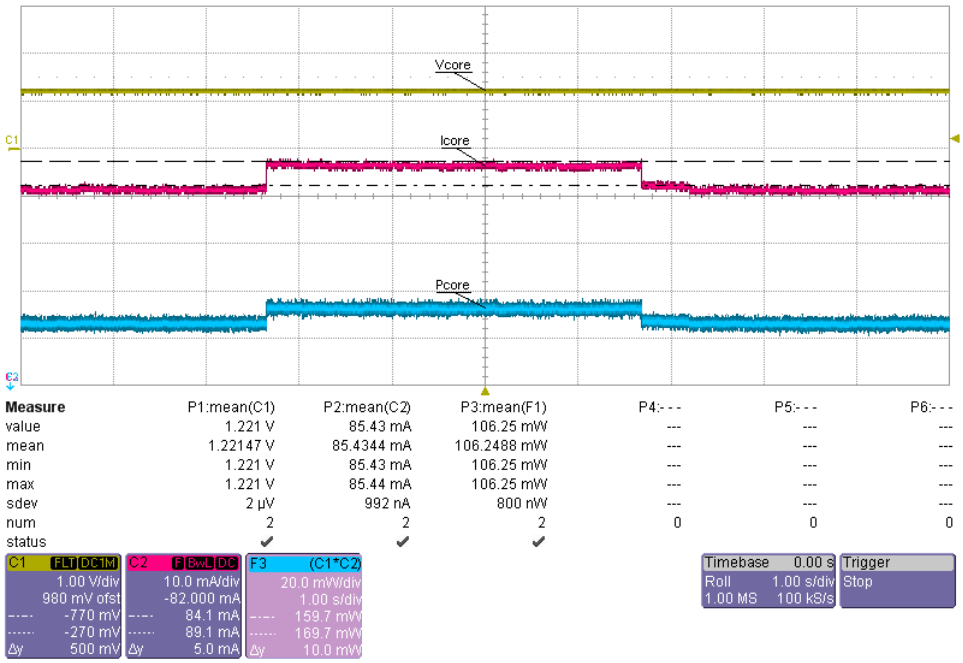
Figure A.1: Critical Path through Redistribution Block

A.3 Histogram Pipeline Power Consumption

This section details the oscilloscope traces from the power consumption investigation on the histogram pipeline. Figures A.2 and A.3 show the impact in the core voltage rails when the algorithm is running on the FPGA and when it is not. The plots show the direct measurements of current and voltage with a selection of statistical measures. The cursors have been placed to isolated the change in current for each histogram option. The power consumption traces have been created by a mathematical function on the oscilloscope.

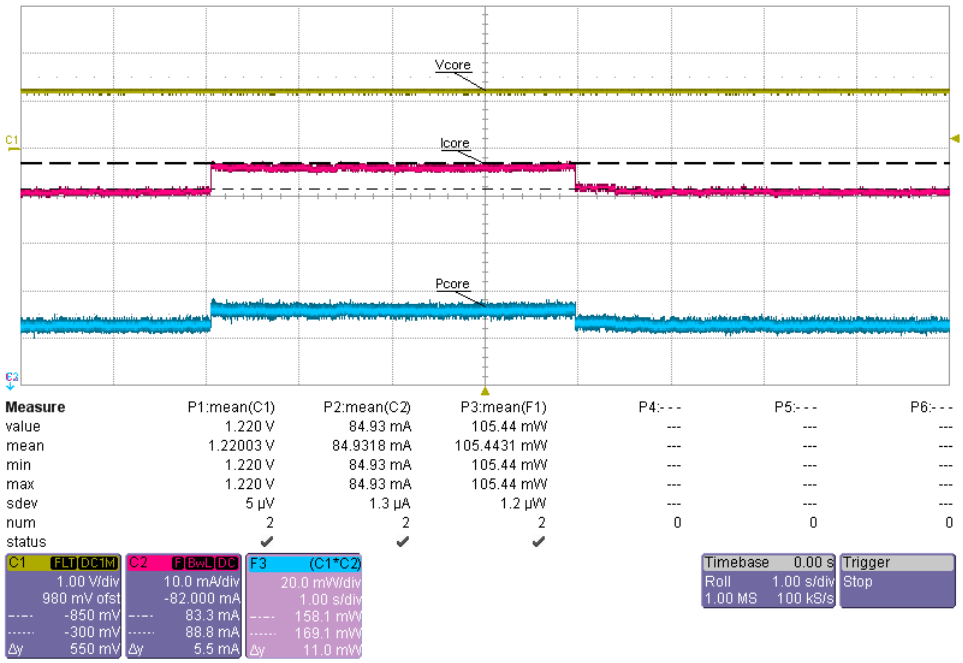


(a) Option 1



(b) Option 2

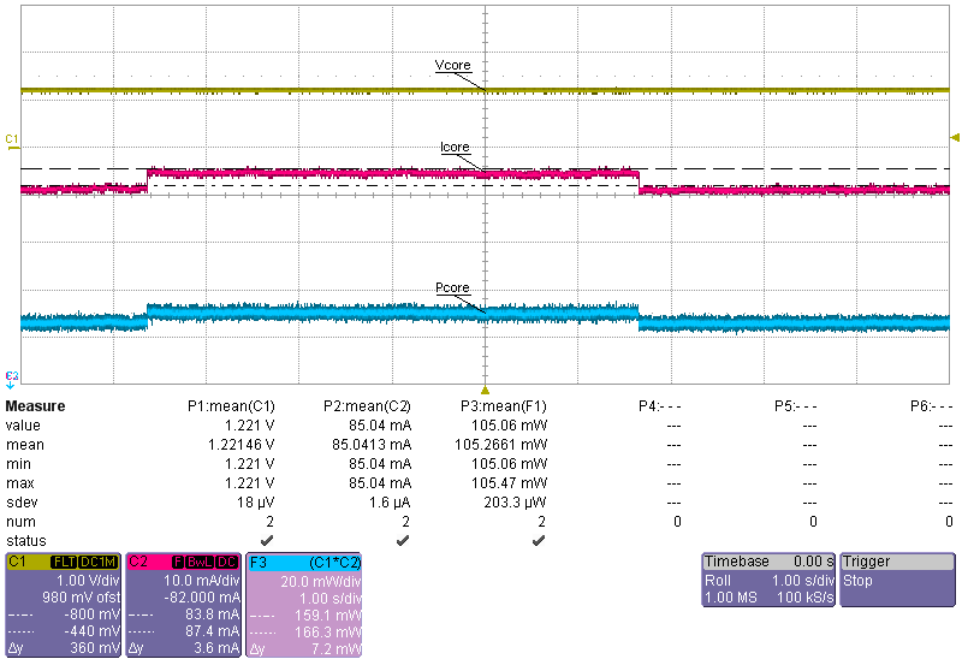
Figure A.2: Current Traces of Options 1 & 2



Measure	P1:mean(C1)	P2:mean(C2)	P3:mean(F1)	P4:- - -	P5:- - -	P6:- - -
value	1.220 V	84.93 mA	105.44 mW	---	---	---
mean	1.22003 V	84.9318 mA	105.4431 mW	---	---	---
min	1.220 V	84.93 mA	105.44 mW	---	---	---
max	1.220 V	84.93 mA	105.44 mW	---	---	---
sdev	5 µV	1.3 µA	1.2 µW	---	---	---
num	2	2	2	0	0	0
status	✔	✔	✔			

C1 FLT DC1M	C2 FBwL DC	F3 (C1*C2)	Timebase 0.00 s	Trigger
1.00 V/div	10.0 mA/div	20.0 mW/div	Roll 1.00 s/div	Stop
980 mV ofst	-82.000 mA	1.00 s/div	1.00 MS 100 kS/s	
-850 mV	83.3 mA	158.1 mW		
-300 mV	88.8 mA	169.1 mW		
Δy 550 mV	Δy 5.5 mA	Δy 11.0 mW		

(a) Option 3



Measure	P1:mean(C1)	P2:mean(C2)	P3:mean(F1)	P4:- - -	P5:- - -	P6:- - -
value	1.221 V	85.04 mA	105.06 mW	---	---	---
mean	1.22146 V	85.0413 mA	105.2661 mW	---	---	---
min	1.221 V	85.04 mA	105.06 mW	---	---	---
max	1.221 V	85.04 mA	105.47 mW	---	---	---
sdev	18 µV	1.6 µA	203.3 µW	---	---	---
num	2	2	2	0	0	0
status	✔	✔	✔			

C1 FLT DC1M	C2 FBwL DC	F3 (C1*C2)	Timebase 0.00 s	Trigger
1.00 V/div	10.0 mA/div	20.0 mW/div	Roll 1.00 s/div	Stop
980 mV ofst	-82.000 mA	1.00 s/div	1.00 MS 100 kS/s	
-800 mV	83.8 mA	159.1 mW		
-440 mV	87.4 mA	166.3 mW		
Δy 360 mV	Δy 3.6 mA	Δy 7.2 mW		

(b) Option 4

Figure A.3: Current Traces of Options 3 & 4

Appendix B

Implementations of Asynchronous Components

B.1 Delay Chains

Fundamental to asynchronous bundled-data circuits are the delay chains that maintain the validity regions between data and handshake signal transitions. The following sections provide a brief summary of the VHDL code used to create the FPGA primitive based implementations. The final section comments on the asynchronous wrappers used to interface to primitive embedded memory blocks.

B.1.1 XOR Carry Chains

The following VHDL code is primarily split into three for-generate statements. These act to daisy chain a number of XOR gates (XORCY) contained within each slice. Each primitive component is assigned a regional location constraint based on the optimal routing within the FPGA. The formulae for these constraints are held in constants and re-calculated on every iteration of the main loop. There are two conditions for the first XOR gate which alter the routing depending on if only one XOR gate delay is required. If there is more than more gate delay required there are a number of conditions for the middle and end gates- if there are more than four gate delays required a delay needs to allocated from gates in neighbouring CLBs , and the correct final gate assigned.

```
entity delaychain is
 generic (
   NUM_gates : positive:=2);
 port(
    input : in std_logic;
    output : out std_logic);
 end delaychain;

ARCHITECTURE chain_arch OF delaychain IS

signal gnd_p: std_logic:='0';
signal vdd_p: std_logic:='1';
signal intercon_g : unsigned(NUM_gates-1 downto 0);
signal intercon_m : unsigned(NUM_gates-1 downto 0);
signal intracon : unsigned(NUM_gates-1 downto 0);

attribute rloc: string;

  begin

    intracon(0)<=input;

    gate_struct: for i in 0 to (NUM_gates-1) generate

      constant xcoord : natural:= ((i mod 4)/2);
      constant ycoord : natural:= (i mod 2)+ ((i/4)*2);
      constant rlocstr : string:= "X" & integer'image(xcoord) & "Y" & integer'image(ycoord);
      -- the numerical sequence for the required x values is simple, 00110011
      -- however the y values need to be 0101232334344545 which is satisfied b
      -- y (i mod 2)+(1/4)*2
      -- need to create 4 generate conditions. One where there is only 1 delay
      -- required, one where there is 2 delays required, one where there is
      -- +3 delays required, and one where there is +5 delays required.

    begin

      firstgate_un: if (i=0 and NUM_gates = 1) generate
        -- covers 1 delay, assigning input and output
        attribute RLOC of instm_f: label is rlocstr;
        attribute RLOC of instg1_f: label is rlocstr;
        attribute RLOC of instg2_f: label is rlocstr;
        begin
          instm_f: MUXCY --inserted with intracon
          port map (DI => gnd_p, CI => gnd_p, S => vdd_p, O => intercon_m(i));
                   instg1_f: XORCY
          port map(LI=>intracon(i),CI=>gnd_p,O=>intercon_g(i));
                  instg2_f: XORCY
          port map(LI=>intercon_g(i),CI=>intercon_m(i),O=>output);
      end generate firstgate_un;

      firstgate_all: if (i=0 and NUM_gates>1) generate
        -- covers initial delay for all other numbers
```

```
    attribute RLOC of instm_f: label is rlocstr;
    attribute RLOC of instg1_f: label is rlocstr;
    attribute RLOC of instg2_f: label is rlocstr;
    begin
      instm_f: MUXCY --inserted with intracon
      port map (DI => gnd_p, CI => gnd_p, S => vdd_p, O => intercon_m(i));
      instg1_f: XORCY
      port map(LI=>intracon(i),CI=>gnd_p,O=>intercon_g(i));
            instg2_f: XORCY
      port map(LI=>intercon_g(i),CI=>intercon_m(i),O=>intracon(i+1));
  end generate firstgate_all;

  midgates_f_all: if (i>0 and i<4 and NUM_gates>4  ) generate
    -- covers delays 5+ where there is an additional delay covered by endgate
    attribute RLOC of instm_mid: label is rlocstr;
    attribute RLOC of instg1_mid: label is rlocstr;
    attribute RLOC of instg2_mid: label is rlocstr;
    begin
      instm_mid: MUXCY
      port map (DI => gnd_p, CI => gnd_p, S => vdd_p, O => intercon_m(i));
      instg1_mid: XORCY
      port map(LI=>intracon(i),CI=>gnd_p,O=>intercon_g(i));
      instg2_mid: XORCY
      port map(LI=>intercon_g(i),CI=>intercon_m(i),O=>intracon(i+1));
  end generate midgates_f_all;

      midgates_ft: if (i>0 and i<(NUM_gates-1) and NUM_gates<5 ) generate
    -- if delays -4 then only need mid gate for RLOCS to stay in ini range
    attribute RLOC of instm_mid: label is rlocstr;
    attribute RLOC of instg1_mid: label is rlocstr;
    attribute RLOC of instg2_mid: label is rlocstr;
    begin
          instm_mid: MUXCY
      port map (DI => gnd_p, CI => gnd_p, S => vdd_p, O => intercon_m(i));
      instg1_mid: XORCY
      port map(LI=>intracon(i),CI=>gnd_p,O=>intercon_g(i));
      instg2_mid: XORCY
      port map(LI=>intercon_g(i),CI=>intercon_m(i),O=>intracon(i+1));
  end generate midgates_ft;

  midgates_e: if (i>3 and i<(NUM_gates-1)and NUM_gates>4 ) generate
    -- only applicable for more than 4 delays
    attribute RLOC of instm_mid: label is rlocstr;
    attribute RLOC of instg1_mid: label is rlocstr;
    attribute RLOC of instg2_mid: label is rlocstr;
    begin
      instm_mid: MUXCY
      port map (DI => gnd_p, CI => gnd_p, S => vdd_p, O => intercon_m(i));
      instg1_mid: XORCY
      port map(LI=>intracon(i),CI=>gnd_p,O=>intercon_g(i));
      instg2_mid: XORCY
      port map(LI=>intercon_g(i),CI=>intercon_m(i),O=>intracon(i+1));
  end generate midgates_e;
```

```
      endgate_all: if (i=(NUM_gates-1) and NUM_gates /=1 and NUM_gates >4 ) generate
        attribute RLOC of instm_e: label is rlocstr;
        attribute RLOC of instg1_e: label is rlocstr;
        attribute RLOC of instg2_e: label is rlocstr;
        begin
          instm_e: MUXCY --inserted with intracon
          port map (DI => gnd_p, CI => gnd_p, S => vdd_p, O => intercon_m(i));
          instg1_e: XORCY
          port map(LI=>intracon(i),CI=>gnd_p,O=>intercon_g(i));
          instg2_e: XORCY
          port map(LI=>intercon_g(i),CI=>intercon_m(i),O=>output);
      end generate endgate_all;

      endgate_lr: if (i=(NUM_gates-1) and NUM_gates /=1 and NUM_gates <5) generate
         -- will have the wrong RLOC if 5+ delays
        attribute RLOC of instm_e: label is rlocstr;
        attribute RLOC of instg1_e: label is rlocstr;
        attribute RLOC of instg2_e: label is rlocstr;
        begin
          instm_e: MUXCY
          port map (DI => gnd_p, CI => gnd_p, S => vdd_p, O => intercon_m(i));
          instg1_e: XORCY
          port map(LI=>intracon(i),CI=>gnd_p,O=>intercon_g(i));
          instg2_e: XORCY
          port map(LI=>intercon_g(i),CI=>intercon_m(i),O=>output);
      end generate endgate_lr;

    end generate gate_struct;
END ARCHITECTURE chain_arch;
```

B.1.2 Look Up Table Carry Chains

The following VHDL code implements a delay chain using look-up table primitives instead of XOR gates. The fundamental structure of the code is very similar to the previous implementation splitting the look-up tables into the first, middle and last sections.. The location constraint constants are slightly different due to the increase number of look-up tables within a slice.

```
ENTITY delaychain_lut IS
    generic (
      NUM_gates : positive:=3);
    port(
       input : in std_logic;
       output : out std_logic);
END ENTITY delaychain_lut;

ARCHITECTURE arch OF delaychain_lut IS

 signal gnd_p: std_logic:='0';
 signal vdd_p: std_logic:='1';
 signal intercon : unsigned(NUM_gates-1 downto 0);

 attribute rloc: string;

BEGIN
  intercon(0)<=input;

  gate_struct: for i in 0 to (NUM_gates-1) generate

    constant xcoord : natural:= ((i mod 8)/4);
    constant ycoord : natural:= ((i mod 4)/2)+ ((i/8)*2);
    constant rlocstr : string:= "X" & integer'image(xcoord) & "Y" & integer'image(ycoord);

    -- the numerical sequence for the required x values is simple, 0000111100001111
    -- however the y values need to be 0011001122332233445 5 which is satisfied by
        -- ((i mod 4)/2)+(i/8)*2

    -- need to create 4 generate conditions. One where there is only 1 delay required, one
    -- where there is 2 delays required, one where there is +3 delays required, and one
        -- where there is +5 delays required.

  begin

    firstgate_un: if (i=0 and NUM_gates = 1) generate
      -- covers 1 delay, assigning input and output
      attribute RLOC of instl_s: label is rlocstr;
      begin
        instl_s : LUT1_L
        generic map (INIT => x"2")
        port map (LO => output, I0 => intercon(i));

    end generate firstgate_un;
```

```
    firstgate_all: if (i=0 and NUM_gates>1) generate
      -- covers initial delay for all other numbers
      attribute RLOC of instl_s: label is rlocstr;
      begin
        instl_s : LUT1_L
        generic map (INIT => x"2")
        port map (LO => intercon(i+1), I0 => intercon(i));

    end generate firstgate_all;

    midgates_all: if (i>0  and NUM_gates>1 and i/=(NUM_gates-1) ) generate

      attribute RLOC of instl_m: label is rlocstr;
      begin
       instl_m : LUT1_L
       generic map (INIT => x"2")
       port map (LO => intercon(i+1), I0 => intercon(i));

    end generate midgates_all;

    endgate_all: if (i=(NUM_gates-1) and NUM_gates /=1 ) generate
    -- this is an AND gate so the delay chain is asymmetric
    attribute RLOC of instl_e: label is rlocstr;
    begin
      instl_e : LUT2_L
      generic map (INIT => x"8")
      port map (LO => output,
                I0 => intercon(0), I1 => intercon(i));

    end generate endgate_all;
  end generate gate_struct;
END ARCHITECTURE arch;
```

B.2 Muller C-element Implementations

During the initial investigations into simple asynchronous components, multiple implementations of a Muller C-element were trialled. The primary aim of these trials was to minimise the resource usage and the timing uncertainty by reducing the routing options. Figure B.1 shows the VHDL architecture of a two input Muller C-element using only one look-up table. The routing of the feedback net is locked via the BEL constraint on the output pin of the slice. The input pin routing is locked by specifying the exact look-up table input to use.

```
ARCHITECTURE arch OF cmuller2 IS

attribute BEL : string;
attribute BEL of c_mull: label is "F";

signal a_int: std_logic;
signal b_int: std_logic;
signal c_int: std_logic;

BEGIN

  -- implements c= (rst')(ab'+ c(a+b')) in one LUT
  c_mull : LUT4_L
  generic map (
  INIT => x"00d4") -- 212 modified from 232 for inverter at input b
  port map (LO => c_int,
  I0 => b_int, -- Altered to suit routing
  I1 => a_int,
  I2 => c_int,
  I3 => RESET);

  Rout<=c_int;
  Ain<=c_int;
  Lt<=c_int;
  a_int<=Rin;
  b_int<=Aout;
   b_int<=Aout;
END ARCHITECTURE arch;
```

Figure B.1: VHDL Implementation of a Muller C-element

If the Muller C-element is being used in a pipeline inverters can be added to each input and output with a simple re-coding of the look-up table vector. The EDIF implementation in figure B.2, although much more difficult to read, contains the same information as described in the VHDL implementation. This EDIF cell is included in the Asynchronous component library, bypassing the synthesis of asynchronous components by synchronous tools.

```
(cell cmuller2 (cellType GENERIC)
   (view arch_unfold_1  (viewType NETLIST)
    (interface
     (port Rin (direction INPUT))
     (port Aout (direction INPUT))
     (port Rout (direction OUTPUT))
     (port Ain (direction OUTPUT))
     (port Lt (direction OUTPUT))
     (port RESET (direction INPUT)))
    (property AREA (string "1.000000"))
    (property KEEP_HIERARCHY (string "TRUE"))
    (contents
     (instance c_mull (viewRef NETLIST(cellRef LUT4_L(libraryRef xcv2p)))
      (property EQN (string "((~I0*I1*~I2*~I3)+(~I0*~I1*I2*~I3)
                             +(~I0*I1*I2*~I3)+(I0*I1*I2*~I3))"))
      (property BEL (string "F"))
      (property NOOPT (string "TRUE"))
      (property INIT (string "00D4")))
     (net Rin
      (joined
       (portRef Rin )
       (portRef I1 (instanceRef c_mull ))))
     (net Aout
      (joined
       (portRef Aout )
       (portRef I0 (instanceRef c_mull ))))
     (net Ain
      (joined
       (portRef Rout )
       (portRef Ain )
       (portRef LO (instanceRef c_mull ))
       (portRef I2 (instanceRef c_mull ))))
     (net RESET
      (joined
       (portRef RESET )
       (portRef I3 (instanceRef c_mull )))))))
```

Figure B.2: EDIF Implementation of a Muller C-element

B.3 Asynchronous Wrapper Implementation

Most of the embedded components on an FPGA; BRAMs, multipliers, accumulators are silicon optimised implementations that have been designed to run synchronously. An wrapper is therefore required to adjust their interface to operate asynchronously. This is an unavoidable overhead. In the case of arithmetic functions, there is sufficient flexibility in their construction that allowing them to operate asynchronously means disabling the pipelining registers within. Block RAMs required a custom wrapper to allow them to operate with an asynchronous circuit. Figure B.3 shows the VHDL description of the wrapper that was used to coordinate data between a synchronous block RAM and an asynchronous circuit.

```
ENTITY async_wrap IS
     generic (
       NUM_BITS : positive:=19);
     port(
       clk          :in std_logic;
       en_CE        :out std_logic;
       Req          :out std_logic;
       Ack          :in std_logic;
       data_in      : in unsigned(NUM_BITS-1 downto 0);
       data_out     : out unsigned(NUM_BITS-1 downto 0);
       RESET        :in std_logic
     );
END ENTITY async_wrap;

ARCHITECTURE arch OF async_wrap IS

signal int_req : std_logic;

  BEGIN

     Req<= int_req;

     req_process : process (clk, Ack, int_req, Reset)
       begin
         if(RESET = '0') then
           if (rising_edge(clk) and Ack= '0' and int_req = '0') then
             data_out<=data_in;
             int_req <='1';
           end if;

           if(Ack= '1') then
             int_req <='0';
           end if;

           en_CE<=((not(Ack)) and (not(int_req)));
         else
           int_req <='0';
           data_out<=data_in;
           en_CE<='0';
           end if;
       end process;

END ARCHITECTURE arch;
```

Figure B.3: VHDL Wrapper Implementation

Once out of reset this piece of code synchronise handshakes with the enabling of the BRAM output via the chip enable (CE) pin. When there is no activity on the control network, the chip enable will go high, on the following rising edge of the clock, data will be passed out of the wrapper along with a handshake request. At this point the BRAM is disable whilst the handshake is concluded. When the transaction has completed the cycle will enable again. This approach allows asynchronous access to the memory but only at the speed of the clock input to the wrapper.

Appendix C

AACIF Supplementary Material

C.1 EDIF Muller C-Element

This section discusses figure C.1, a Muller C-element implemented within an EDIF file from

```
(edif example_file
 (edifVersion 2 0 0)
 (edifLevel 0)
 (keywordMap (keywordLevel 0))
 (status
  (written
   (timestamp 2009 04 21 13 21 53)))
 (library example_library
 (edifLevel 0)
  (technology (numberDefinition ))
  (cell part1 (cellType GENERIC)
   (view arch  (viewType NETLIST)
    (interface
     (port a (direction INPUT))
     (port b (direction INPUT))
     (port rst (direction INPUT))
     (port o (direction OUTPUT)))
    (contents
     (instance comp2(viewRef NETLIST  (cellRef LUT3_L (libraryRef xcv2p )))
      (property EQN (string "((I0*I1*~I2)+(I0*~I1*I2)+(~I0*I1*I2)+(I0*I1*I2))"))
      (property RLOC (string "X0Y0"))
      (property NOOPT (string "TRUE"))
      (property INIT (string "E8")))
     (net a
      (joined
       (portRef a )
       (portRef I1 (instanceRef comp2 ))))
     (net b
      (joined
       (portRef b )
       (portRef I0 (instanceRef comp2 ))))
     (net rst
      (joined
       (portRef rst )
       (portRef CLR (instanceRef comp3 ))))
     (net o
      (joined
       (portRef o )
       (portRef Q (instanceRef comp3 ))
       (portRef I2 (instanceRef comp2 ))))
     (net PWR
      (joined
       (portRef P (instanceRef PWR ))
       (portRef G (instanceRef comp3 )))))))
 )
        (design part1  (cellRef comp1  (libraryRef example_library)))
        )
```

Figure C.1: EDIF Muller C-element Description

FPGA primitive components . Figure C.1 provides a full EDIF description of a Muller C-element and demonstrates the EDIF libraries that contain the EDIF cell components that contain the interconnected primitive components. Here the look-up table initialisation values are explicitly stated along with any constraints.

C.2 Controller Comparisons

This section covers the testbench procedure and methodology used to compare the performance of the latch controllers. All of the controllers were subject to the same 4-phase tester with the exception of the Mousetrap controller because it is a transition sensitive component and required a 2-phase handshake.

C.2.1 Comparison TestBench Files

The testbench instantiates an adaptable Muller pipeline. The length, width, controller type and delay chain length are all customisable via a set of generic values that are passed through to the simulation environment. The tester is adaptable via a VHDL configuration which specifies a difference architecture dependent on the controller being used in the simulation. Both the 4-phase and 2-phase architectures simulate immediate responses from the output environment of the pipeline and simulate delays on the input environment responses. This allows the throughput of the controllers to be evaluated in a controlled environment.

C.2.1.1 Controller TestBench

```
ENTITY controller_bench_tb IS
-- Declarations
END controller_bench_tb ;

ARCHITECTURE struct OF controller_bench_tb IS

   -- Internal signal declarations
   SIGNAL RESET       : std_logic;
   SIGNAL ack_l       : std_logic;
   SIGNAL ack_r       : std_logic;
   SIGNAL input_num   : unsigned(7 DOWNTO 0);
   SIGNAL output_num  : unsigned(7 DOWNTO 0);
   SIGNAL req_l       : std_logic;
   SIGNAL req_r       : std_logic;

 -- Component Declarations
   COMPONENT controller_bench
   GENERIC (
      CONTROLLER : latch_controller;
      NUM_BITS   : positive;
```

```
        LENGTH      : positive;
        STAGE_DEL   : positive
    );
    PORT (
        RESET       : IN      std_logic;
        ack_r       : IN      std_logic;
        input_num   : IN      unsigned ((NUM_BITS-1) DOWNTO 0);
        req_l       : IN      std_logic;
        ack_l       : OUT     std_logic;
        output_num  : OUT     unsigned ((NUM_BITS-1) DOWNTO 0);
        req_r       : OUT     std_logic
    );
    END COMPONENT;
    COMPONENT controller_bench_tester
    PORT (
        ack_l       : IN      std_logic ;
        output_num  : IN      unsigned (7 DOWNTO 0);
        req_r       : IN      std_logic ;
        RESET       : OUT     std_logic ;
        ack_r       : OUT     std_logic ;
        input_num   : OUT     unsigned (7 DOWNTO 0);
        req_l       : OUT     std_logic
    );
    END COMPONENT;

    -- embedded configurations
    FOR ALL : controller_bench USE ENTITY Async_FPGA_lib.controller_bench;
    FOR ALL : controller_bench_tester USE ENTITY Async_FPGA_lib
                                    .controller_bench_tester(archt_4ph);

BEGIN

    -- Instance port mappings.
    U_0 : controller_bench
        GENERIC MAP (
            CONTROLLER => semi_dec,
            NUM_BITS   => 8,
            LENGTH     => 4,
            STAGE_DEL  => 4
        )
        PORT MAP (
            input_num  => input_num,
            req_l      => req_l,
            req_r      => req_r,
            RESET      => RESET,
            ack_l      => ack_l,
            ack_r      => ack_r,
            output_num => output_num
        );
    U_1 : controller_bench_tester
        PORT MAP (
            ack_l      => ack_l,
            output_num => output_num,
```

```
        req_r       => req_r,
        RESET       => RESET,
        ack_r       => ack_r,
        input_num   => input_num,
        req_l       => req_l
    );

END struct;
```

C.2.1.2 Controller Tester

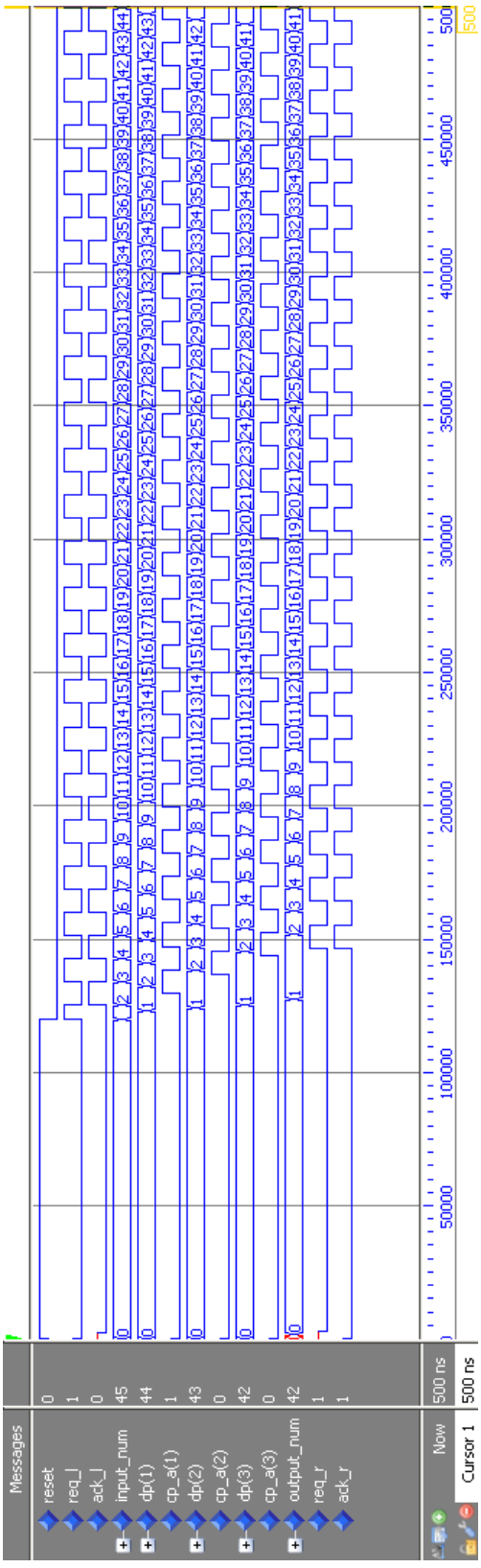
```
ENTITY controller_bench_tester IS
   PORT(
      ack_l       : IN     std_logic;
      output_num  : IN     unsigned (7 DOWNTO 0);
      req_r       : IN     std_logic;
      RESET       : OUT    std_logic;
      ack_r       : OUT    std_logic;
      input_num   : OUT    unsigned (7 DOWNTO 0);
      req_l       : OUT    std_logic
   );
END controller_bench_tester ;

ARCHITECTURE archt_4ph OF controller_bench_tester IS
-- this architecture is for the level sensitive latch controllers
signal rst_int: std_logic;
signal input_num_int : unsigned (7 DOWNTO 0) := (others=>'0');
BEGIN

        RESET<=rst_int;
    input_num<=input_num_int;

        init: process
      begin
            rst_int<='1';
            wait for 20 ns;
            rst_int<='0';
            wait;
      end process;

    leftside: process(ack_l, rst_int)
      begin
        if rst_int = '1' then
          req_l<='0';
        end if;

        if (ack_l='1') then
          req_l<='0' after 5 ns;
        end if;

        if (ack_l='0') and (rst_int='0') then
          input_num_int<= input_num_int + 1;
```

```
          req_l<='1' after 10 ns;
        end if;
      end process leftside;

        rightside: process(req_r, rst_int)
        begin
                if rst_int = '1' then
                ack_r<='0';
            end if;

            if (req_r='1') then
                ack_r<='1';
            end if;

            if (req_r='0') then
                ack_r<='0';
            end if;
         end process rightside;

END ARCHITECTURE archt_4ph;

ARCHITECTURE archt_2ph OF controller_bench_tester IS
-- this architecture is for the transistion sensitive latch controllers
signal rst_int: std_logic;
signal input_num_int : unsigned (7 DOWNTO 0) := (others=>'0');

BEGIN

    RESET<=rst_int;
    input_num<=input_num_int;

    init: process
      begin
            rst_int<='1';
            wait for 20 ns;
            rst_int<='0';
            wait;
      end process;

    leftside: process(ack_l, rst_int)
      begin
        if rst_int = '1' then
          req_l<='0';
        end if;

        if rising_edge(ack_l) then
          input_num_int<= input_num_int+1;
          req_l<='0' after 5 ns;
        end if;

        if falling_edge(ack_l) or falling_edge(rst_int) then
          input_num_int<= input_num_int + 1;
          req_l<='1' after 10 ns;
```

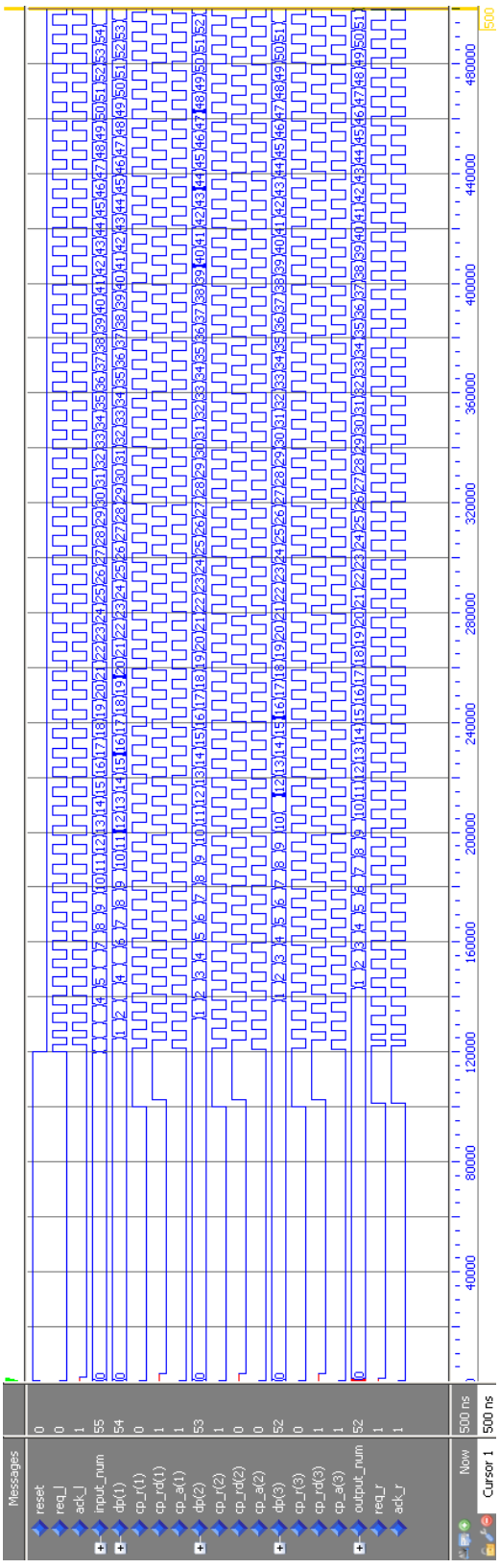
```
        end if;
      end process leftside;

    rightside: process(req_r, rst_int)
      begin
        if rst_int = '1' then
          ack_r<='0';
        end if;

        if  rising_edge(req_r) then
          ack_r<='1';
        end if;

        if falling_edge(req_r) then
          ack_r<='0';
        end if;
      end process rightside;

END ARCHITECTURE archt_2ph;
```

C.2.2 Expanded Waveforms

Figure C.2 shows the expanded waveforms from the Mousetrap and AACIF controllers.



(a) Mousetrap Expanded Waveform



(b) AACIF Expanded Waveform

Figure C.2: Expanded Waveforms

C.3 Power Spectrum Setup

Figure C.3 shows the front of the Virtex 2 pro development board used to measure the power spectrum. The only hardware modifications are to the decoupling capacitors on the other side of the board. The passive probe is connected to the oscilloscope via a SMA connector.

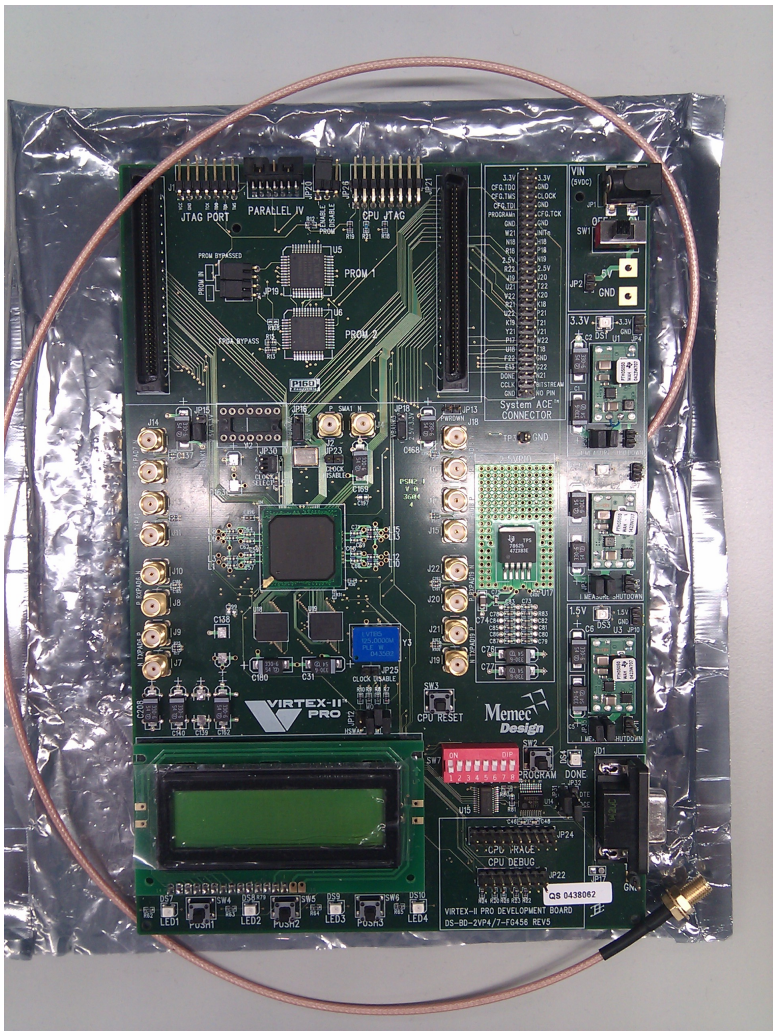


Figure C.3: Board Setup

C.4 Power spectrum for the 422 to 444 circuit

Figure C.4 shows the power spectrum from the 422 to 444 video format conversion circuit. The points on this spectrum are very similar to the 444 to RGB circuit found in the main text. The blue spectrum is from the synchronous implementation and the red spectrum is from the AACIF implemenation. The overlay shows the background spike from the oscilloscope and the voltage regulator spike also present. A similar reduction on clock harmonics can also be seen as well as an average reduction in spikes where individual circuit events are occurring.

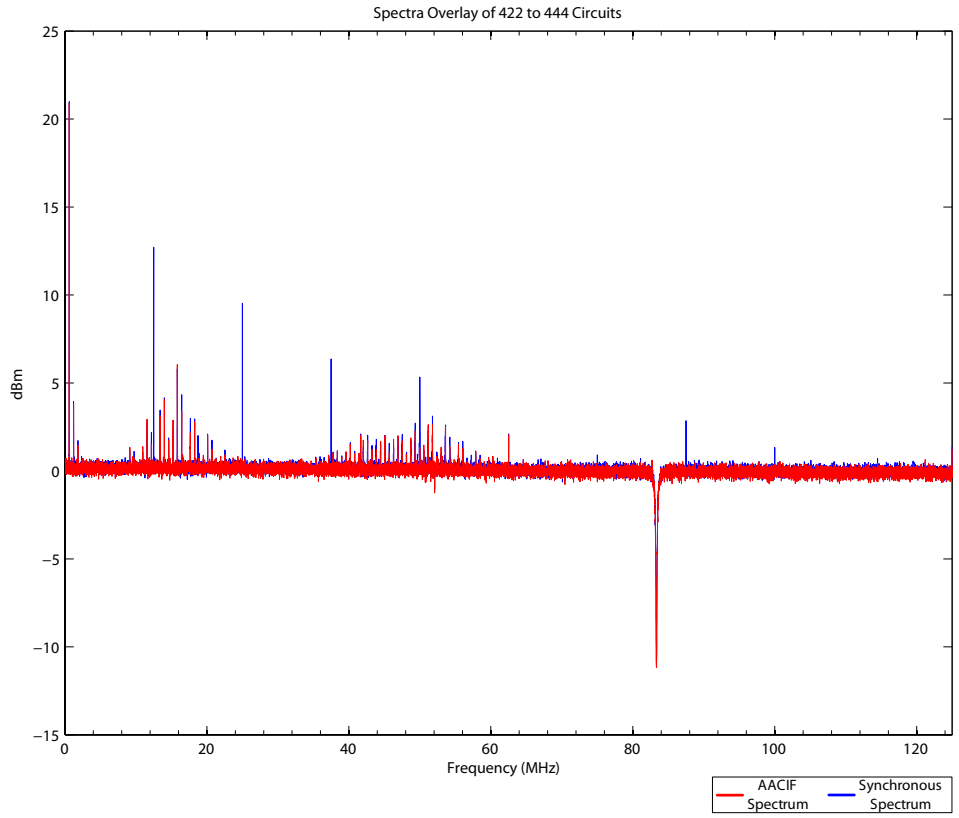


Figure C.4: 422 to 444 Circuits Noise Spectrum Overlay