Bitstream specialisation for dynamic reconfiguration of real-time applications

Ronnie Rikus le Roux
13077643

Thesis submitted for the degree *Philosophiae Doctor* in Computer and Electronics at the Potchefstroom Campus of the North-West University

Promotor: Prof. G. van Schoor
Co-promotor: Dr. P.A. van Vuuren

December 2014

It all starts here

NORTH-WEST UNIVERSITY
YUNIBESITI YA BOKONE-BOPHIRIMA
NOORDWES-UNIVERSITEIT
POTCHEFSTROOM CAMPUS

To fear the Lord is the beginning of knowledge:
but fools despise wisdom and instruction.

PROVERBS 7:1 (KJV)

ACKNOWLEDGEMENTS

First and foremost, I would like to thank our Heavenly Father for providing me with the strength and patience to pursue a Ph.D. Father, without You the work presented in this thesis would have never seen the light of day. Thank You for the privilege and opportunity to investigate Your creation. While it is man that brought this technology into existence, it was You that provided him with the knowledge.

Father, I would also like to thank You for my loving wife, Nicolene, who stood by me all these years. I know I wasn't always the easiest to live with, but thanks to her undying love, support and understanding, we pushed on and eventually reached the top. Nicolene, you are one in a million and I love you very much.

Father, You also provided me with the best parents, Louis and Lettie, any son could ask for and for that I am eternally grateful. Mom and Dad, thank you for always being there for me. Thank you for always being prepared to listen and for providing a shoulder to cry on. You are truly the best parents anyone could ever ask for and I love you lots. Father, I would also like to thank you for my brother, Christo. In the words of Jolene Perry, "brothers don't let each other walk in the dark alone". Boeta, thank you for being my companion in times when it feels like I am walking in the dark. There's no love like the love for a brother, and there is no love like the love from a brother. Love you man!

Lord, you also blessed me with lots of friends along this journey, who were always there when I needed a well deserved break. There were so many along the way, but I thank You particularly for sending Schalk, Angelique, Mano, Talita, Gert, DB, Jan, Angelique JvR, and Arno my way. Thank You for their friendship over the years. Thank You for the many coffee breaks, motivational speeches, technical support, Tuesday movies, braais, and drinks along the way. It was an absolute pleasure taking this journey with them.

Speaking of technical support, Father I would also like to thank You for the experts you provided just when I thought I've run out of ideas and options. In particular, I am grateful for the help and support from the Hardware and Embedded Systems group from Ghent University. Thank You for the many discussions with Karel Bruneel and Heyse about reconfigurable computing

concepts I didn't quite understand. Thank You that they were always open for discussion, and for Karel Heyse supplying the correct replacement header file for Get/SetClbBits. They saved me lots of frustration, and for that I am thankful. Also, thank You for the kindness of Fabio Cancaré, who was willing to supply the technical report by Davide Castellone. Without him, the breakthrough in analysing the bitstream would've taken significantly longer.

Of course, none of this would be possible without funding. Farther, I praise Your name for the funding I've received during this period. You said You would provide, and You did. Thank You for the privilege to work on a magnitude of THRIP projects, for the NRF grant I received, as well as the OMT grant you provided when I needed it the most.

Last, but surely not the least, I thank you for the two best promotors any post-graduate student could ask for. While I'm pretty sure I bored them to death most of the time, and the other half of the time they had no idea what I was talking about, they provided excellent guidance throughout this study. Thank you Prof and Pieter for your mentorship and support through all my years at McTronX. It is a time I will never forget. I look forward to whatever the future may hold and with God's grace, we will definitely work alongside each other in the future.

Amen

ABSTRACT

The focus of this thesis is on specialising the configuration of a field-programmable gate array (FPGA) to allow dynamic reconfiguration of real-time applications. The dynamic reconfiguration of an application has numerous advantages, but due to the overhead introduced by this process, it is only advantageous if the execution time exceeds reconfiguration time. This implies that dynamic reconfiguration is more suited to quasi-static applications, and real-time applications are therefore typically not reconfigured.

A method proposed in the literature to ameliorate the overhead from the configuration process is to use a block-RAM (BRAM) based, hardware-controlled reconfiguration architecture, eliminating the need for a processor bus by storing the configuration in localised memory. The drawback of this architecture is the limited size of the BRAM, implying only a subset of configurations can be stored.

The work presented in this thesis aim to address this size limitation by proposing a specialiser capable of adapting the configuration stored in the BRAM to represent different sets of hardware. This is done by directly manipulating the bits in the configuration using passive hardware. This not only allows the configuration to be specialised practically immediately, but also allows this specialiser to be device independent. By incorporating this specialiser into the BRAM-based architecture, this study sets out to establish that it is possible to reduce the overhead of the reconfiguration process to such an extent that dynamic reconfiguration can be used for real-time applications.

Since the composition of the configuration is not publicly available, a method had to be found to parse and analyse the configuration in order to map the configuration space of the device. The approach used was to compare numerous different configurations and mapping the differences. By analysing these differences, it was found that there is a logical relationship between the slice coordinates and the configuration space of the device. The encoding of the lookup tables was also determined from their initialisation parameters. This allows the configuration of any lookup table to be changed by simply changing the corresponding bits in the configuration.

Using this proposed reconfiguration architecture, a distributed multiply-accumulate was recon-

figured and its functional density measured. The reason for selecting this specific application is because the multiply-accumulate instruction can be found at the heart of many real-time applications. If the functional density of the reconfigured application is comparable to those of its static equivalent, a strong case can be made for real-time reconfiguration in general. Functional density is an indication of the composite benefits dynamic reconfiguration obtains above its static generic counterpart. Due to the overhead of the reconfiguration process, the functional density of reconfigured applications is traditionally significantly lower than those of static applications. If the functional density of the reconfigured application can rival those of the static equivalent, the overhead from the reconfiguration process becomes negligible.

Using this metric, the functional density of the distributed multiply-accumulate was compared for different reconfiguration implementations. It was found that the reconfiguration architecture proposed in this thesis yields a significant improvement over other reconfiguration methods. In fact, the functional density of this method rivalled that of its static equivalent, implying that it is possible to dynamically reconfigure a real-time application. It was also found that the proposed architecture reduces specialisation and reconfiguration time to such an extent that it is possible complete the reconfiguration process within strict time constraints. Even though the proposed method is only capable of reconfiguring the LUTs of a real-time application, this is the first step towards allowing full reconfiguration of applications with dynamic characteristics.

The first contribution this thesis makes is a novel method to parse and analyse the configuration of a Xilinx® Virtex®-5 FPGA. It also successfully maps the configuration space to the configuration data. Even though this method is applied to a specific device, it is device independent and can easily be applied to any other FPGA. The second contribution comes from using the information obtained from this analysis to design and implement a configuration specialiser, capable of adapting lookup tables in real time. Lastly, the third contribution combines this specialiser with the BRAM-based architecture to allow the reconfiguration of applications typically not reconfigured.

Keywords: reconfigurable computing, dynamic reconfiguration, real-time, bitstream specialisation, direct bitstream manipulation

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application programming interface
ASCII	American standard code for information interchange
ASIC	Application-specific integrated circuit
BEL	Basic element logic
Bil	Bitfile interpretation library
BitMaT	Bitstream manipulation tool
BRAM	Block random access memory
CE	Chip enable
CLB	Configurable logic block
CLK	Clock
CPU	Central processing unit
CRC	Cyclic redundancy check
DALUT	Distributed arithmetic lookup table
DCS	Dynamic circuit specialisation
DCM	Digital clock manager
DES	Data encryption standard
DIP	Dual in-line package
DMA	Direct memory access
DMAC	Distributed multiply-accumulate
DNA	Deoxyribonucleic acid
DPR	Dynamic partial reconfiguration
DRC	Design rule check
DSP	Digital signal processor
EAPR	Early access partial reconfiguration
ECC	Error checking code
EHW	Evolvable hardware
FAR	Frame address register

Continued on next page

FDRO	Frame data register, input register
FF	Flip-flop
FIFO	First in, first out
FIR	Finite impulse response
FPGA	Field-programmable gate array
FSK	Frequency-shift keying
GPP	General purpose processor
HCLK	Horizontal clock
HWICAP	Hardware internal configuration access port
ICAP	Internal configuration access port
IDIW	ICAP data input width
IOB	Input\output block
IP	Intellectual property
IPIF	Intellectual property interface
ISE®	Integrated Synthesis Environment
LOUT	Legacy output register
LSB	Least significant bit
LSN	Least significant nibble
LUT	Lookup table
MAC	Multiply-accumulate
MATLAB	Matrix laboratory
MPMC	Multi-port memory controller
MSB	Most significant bit
MSN	Most significant nibble
MTT	Maximum theoretical throughput
NCD	Native circuit description
NLM	Nibble location method
NOP	No operation
NP-complete	Non-deterministic polynomial time complete
NRE	Non-recurring engineering
OPB	On-chip peripheral bus
PAR	Place and route
PARBIT	Partial bitfile transformer
PBS	Parameterized bitstream specialiser
PID	Proportional-integral-derivative
PLB	Processor local bus
PowerPC	Performance optimization with enhanced RISC-performance computing
PPC	PowerPC
PR	Partial reconfiguration
PWM	Pulse width modulation
RAM	Random access memory
RISC	Reduced instruction set computing

Continued on next page

RTL	Register-transfer level
SAT	Boolean satisfiability
SRAM	Static random access memory
SRL	Shift register lookup table
TCAM	Ternary content-addressable memory
TCON	Tunable connection
TLUT	Tunable lookup table
UART	Universal asynchronous receiver\transmitter
VHDL	VHSIC hardware description language
VHSIC	Very-high-speed integrated circuits
VLSI	Very-large-scale integration
VRC	Virtual reconfigurable circuits
XDL	Xilinx® design language
XPART	Xilinx® partial reconfiguration tookit
XPS	Xilinx® Platform Studio

CHAPTER 1

INTRODUCTION

"Begin at the beginning, and go on till you come to the end: then stop."

— Lewis Carroll, *Alice in Wonderland*

1.1 Understanding the present is knowing the past

The last couple of years have seen a tremendous growth in software size and processing requirements. This is due to hardware following the trend predicted by Gordon Moore (most famously known as Moore's Law) [1], which in turn has a direct effect on the size of software required, as stated by Nathan's Law [2]. Nathan Myhrvold's four laws of software state that:

1. Software can be resembled by a gas in that it always expands to fit the container it is in.
2. Software grows until it is governed by Moore's Law.
3. Software growth makes Moore's law possible, since better hardware is required to run the software.
4. Software is only limited by human ambition and expectation.

It was also stated that the size and complexity of software is constantly rising and that there is no limit in sight. As the continual increase in complexity of the hardware leads to more complex software being developed, the complex software continues to push the boundaries of the hardware and eventually requires more complex hardware.

Nearly 15 years after Nathan's law was introduced, it is evident that software exhibits gas-like behaviour. The multi-core era has dawned and processors gradually require more cores to

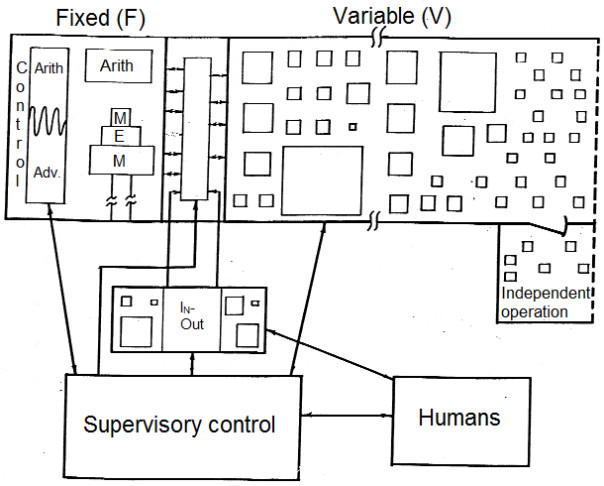


Figure 1.1: Original hand-drawn representation of the F+V structure computer as proposed by Estrin [5])

cope with the scaling of the software. The same applies to embedded systems. Traditionally, processing performance was improved by either using application specific integrated circuits (ASICs) or by increasing the clock frequency and/or the number of cores. The latter is reminiscent of the paradigm dubbed the "Von Neumann syndrome" [3, 4], which refers to the improvement of the Von Neumann system architecture[1] by adding more datapaths in parallel. These types of architectures are limited by the fact that instructions and data are fetched from the same memory, which greatly limits operating bandwidth—also known as the Von Neumann bottleneck. Even though an ASIC delivers the best possible performance, it is only tailored to a specific application and requires a redesign for device functionality changes, which increases the non-return engineering (NRE) costs.

The fixed-plus-variable (F+V) structure computer [6], originally proposed by Gerald Estrin in the 1960's [5] and shown as the original hard-drawn sketch in Figure 1.1, is another paradigm to improve processor performance. The fixed processor provides a programmer with a familiar programming environment (such as C) for implementing general-purpose applications, while the variable processor can be reconfigured for a specific task. This was the first time reconfigurable computing was proposed, but due to the limitations in technology of the time, this concept was not adopted well during that era.

1.2 Marrying high-performance and flexibility

Reconfigurable computing stemmed from the F+V structure computer proposed by Estrin, and is a computer architecture that marries the flexibility of software with the high performance capability of hardware. The primary difference compared to general purpose processors (GPPs), is that reconfigurable computing has the ability to make changes to both the datapath and control flow. Initially, this was done using a modular design with a hardware module that can be substituted with another to perform a specialised function.

[1] An instruction stream-based computing architecture controlled by a program counter.

The invention of field-programmable gate arrays (FPGAs) in 1985 infused new life into the paradigm proposed by Estrin. FPGAs are revolutionary devices that implement circuits in hardware, but can be reprogrammed to suit a specific application. FPGAs are thus a suitable platform for implementing reconfigurable computing. In fact, most of Xilinx®'s FPGAs from the Virtex®-II series incorporate a feature called dynamic reconfiguration, that allows hardware modules to be swapped to and fro while the rest of the device remains operational.

As is typical within a marriage, marrying hardware and software does not come without compromise. While FPGAs provide nearly all of the benefits of both hardware and software, they are only really useful in applications that process large streams of data, such as signal processing and network processing. Compared to ASICs, FPGAs are between 5 and 25 times worse in area, delay and performance [7]. However, the NRE cost of using FPGAs is significantly lower compared to that of an ASIC. FPGAs therefore provide a good compromise between cost, performance and flexibility.

1.3 To reconfigure or not to reconfigure, that is the question

The primary advantage of dynamic reconfiguration is the ability to specialise the circuit architecture during run-time [8]. This specialisation could either improve the execution time of the calculation, or the area utilization since a specialised circuit requires less hardware than its general-purpose equivalent.[2] The cost (C) of a very-large-scale integration (VLSI) circuit is a function of the area (A) and execution time (T), calculated by $C = AT$ [9]. Even though the initial conception of this metric utilised the physical silicon area for A, for FPGAs the number of slices or lookup tables are most commonly used.

Functional density (D) was first proposed by Wirthlin [10] as a measure of the composite benefits dynamic reconfiguration obtains above its static generic counterpart. It measures the computational throughput (in operations per second) per unit hardware resources [10]. For the static case, functional density is defined as the inverse of AT and is given by:

$$D_s = \frac{1}{C_s} = \frac{1}{A_s T_{s,exec}} \tag{1.1}$$

where D_s denotes the static functional density, A_s the static area and $T_{s,exec}$ the execution time of the static implementation. This definition of functional density can be expanded to include the execution time of the reconfigurable implementation, $T_{r,exec}$, and the reconfiguration time, T_{reconf}:

$$D_r = \frac{1}{A_r(T_{r,exec} + T_{reconf})} \tag{1.2}$$

where D_r denotes the reconfiguration functional density. It is thus evident that the moment

[2] Reducing the area of an implementation reduces power consumption, since unnecessary hardware is removed, and allows designs to fit on a smaller device—reducing cost.

an application is reconfigured, the functional density is reduced by the added reconfiguration time. In fact, this is one of the main disadvantages of dynamic reconfiguration; it introduces an additional delay from the moment reconfiguration is required, to the time this new configuration can be used by the application. This delay is not only caused by the reconfiguration process, but also by the time required to generate new hardware. As a result, T_{reconf} is more accurately described as $T_{conf} + T_{gen}$ [11], with T_{conf} the time to configure the device and T_{gen} the time to generate new hardware. Inserting this into (1.2) yields the reconfigurable functional density:

$$D_r = \frac{1}{A_r(T_{r,exec} + T_{conf} + T_{gen})}. \tag{1.3}$$

The effect of these delays is illustrated in the timing diagram shown in Figure 1.2. In the figure, the time required to generate new hardware (T_{gen}) is depicted in orange, the time to configure the device (T_{conf}) in green, the full reconfiguration time (T_{reconf}) in yellow and the execution time of the application ($T_{r,exec}$) in blue. The configuration time is highly dependent on the size of the configuration and the throughput of the process. Typical configuration times are in the order of microseconds, whereas the time to generate new hardware could range from a couple of seconds to hours - depending on the complexity, size, quality and methods used to generate the hardware.

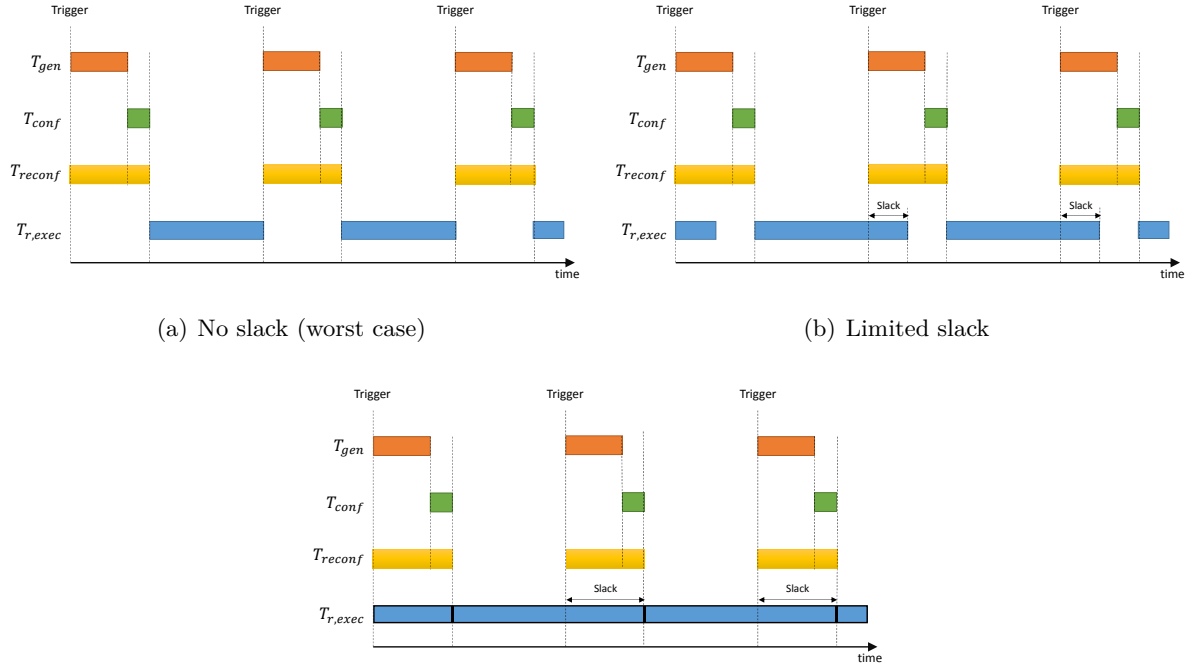
Bruneel [12] defines the time between a new hardware request and the moment processing with the old hardware stops as the slack (T_{slack}). In the case where no slack is available, such as shown in Figure 1.2(a), the execution time has to be interrupted while waiting for new hardware. In an ideal reconfiguration architecture, the process of generating new hardware can run parallel to the application being executed and the reconfiguration event is known in advance. This allows the new hardware to be generated before being required by the reconfiguration process. Not only that, but the dynamic partial reconfigurable nature of some Xilinx® FPGAs allows reconfiguring a section of the device while the application is being executed. This will minimise slack, yielding the situation with limited slack shown in Figure 1.2(b). However, if the available slack is equal to or larger than $T_{conf} + T_{gen}$, the idle time of the FPGA can be reduced to zero—depicted in Figure 1.2(c). For this to be realised, the hardware either has to be unused by the application at the moment of reconfiguration, or has to be implemented in parallel—sacrificing area.

The configuration ratio expresses the relative reconfiguration time to execution time and is given by $f = \frac{T_{reconf}}{T_{r,exec}}$. The functional density can then be expressed as:

$$D_r = \frac{1}{A_r T_{r,exec}(1 + f)}. \tag{1.4}$$

This illustrates an important aspect of reconfigurable computing. In order for the overhead introduced by reconfiguration to become negligible, $f \to 0$ which implies an execution time significantly exceeding reconfiguration time. In systems where this holds true, the maximum functional density, or $D_{max} = lim_{f \to 0} D_r = \frac{1}{AT_e}$ is approached.

If the hardware component can be used multiple times before reconfiguration is required, the



(a) No slack (worst case)

(b) Limited slack

(c) Abundant slack (ideal case)

Figure 1.2: Timing diagrams of a dynamically reconfigurable system

reconfiguration time can be amortized over several executions, n, thus increasing the functional density [10]:

$$D_r = \frac{1}{A_r(T_{r,exec} + \frac{T_{conf}+T_{gen}}{n})}. \tag{1.5}$$

This equation can also be used to find the break-even point (where $D_s = D_r$), which is the minimum number of times hardware should be reused before reconfiguration becomes feasible:

$$n = \frac{A_r(T_{conf} + T_{gen})}{A_s T_{s,exec} - A_r T_{r,exec}}. \tag{1.6}$$

This implies that even if dynamic reconfiguration does not yield a functional density advantage for a specific application, it is possible to amortise the reconfiguration overhead over multiple execution cycles by reusing the hardware. It is worth noting at this point that quasi-static applications have a large n, illustrating why these designs are well suited for dynamic reconfiguration.

1.4 Minimising the cost of reconfiguration

Several researchers aim to minimise the cost of reconfiguration by reducing T_{conf} and T_{gen}. Possibly the biggest contributors of the latter are the placement and routing (PAR) techniques conventional FPGA tools use. Even though it is difficult to confirm due to intellectual property (IP) reasons, it is suspected that most vendors' tools use simulated annealing (or a modified version thereof [13]) to determine the optimal PAR, as this has been proven to provide a good balance between fit and performance. The problem with simulated annealing is that it is an NP-complete (non-deterministic polynomial time complete) problem, and as such, contributes a significant amount of overhead to the process of generating new hardware. The solution is to either sacrifice the quality of the placement and/or routing in return for a reduced T_{gen}, or to reuse the PAR information when generating multiple configuration subsets. Another alternative is to skip the place and route step completely by using partial evaluation, generic netlists, constant multiplication or by manipulating the configuration data at bit-level.

The time required to (re)configure the FPGA, T_{conf}, is directly related to the size of the configuration and the speed at which it can be transferred to the configuration memory through the internal configuration access port (ICAP). As the name suggests, the ICAP is an internal port allowing access to the configuration registers [14]. Compression techniques—both conventional and tailor-made—can be used to reduce the size of the configuration, which not only requires less storage but also improves the reconfiguration time. Combining this with changes to the system and reconfiguration architecture, the throughput of the system can be improved, minimising the reconfiguration time and improving the functional density.

The most promising architecture proposed in the literature to minimise T_{conf} uses the FPGA's block random access memory (BRAM) to store the configuration data. A hardware implemented controller is then used to facilitate the reconfiguration process. By using this specific setup, all unnecessary overhead can be avoided. Not only that, but this configuration requires no additional clock cycles for processing complex algorithms and protocols. A drawback of this approach is that the BRAM is extremely limited and only a subset of configurations can be stored. The different means of reducing configuration overhead is discussed in more detail in the next chapter.

1.5 Research problem

It is evident from the previous sections that the reason reconfigurable computing is only suitable for quasi-static applications, is due to the configuration overhead. Despite the tremendous headway being made in migrating reconfiguration towards more dynamic applications, one thing is certain: reconfiguration is still widely unused in real-time applications. Traditionally, reconfiguring a real-time application would not yield a functional density advantage over a generic implementation, due to its dynamic characteristics.

The research presented in this thesis aims to address just that by proposing a reconfiguration method for real-time applications. Gambier [15] defines a real-time application as "... one in

which the correctness of a result not only depends on the logical correctness of the calculation, but also upon the time at which the result is made available". Due to the tremendous overhead introduced by the PAR process, making it unsuitable to execute in real-time, this research focusses on the cost reduction by improving the transfer of the FPGA configuration to the configuration memory. Another reason for eliminating other cost reduction methods is the additional processing required for their implementation, which would add extra clock cycles to the reconfiguration process. By using the BRAM-based architecture, this research sets out to prove that not only is it possible to reconfigure a real-time application, but it is also possible to improve the functional density of a dynamic application to such an extent that it is comparable to its static equivalent.

Relating the research problem back to (1.3), T_{conf} will be minimised by using the aforementioned BRAM-based architecture. Minimising T_{gen}, while simultaneously addressing the BRAM-limitations of these architectures, will be done by adding a specialisation technique. The hypothesis is that a single configuration stored in the BRAM can be adapted to represent any other set of hardware, without introducing additional overhead. The result will be an optimal architecture capable of reconfiguring a real-time application within a reasonable amount of time—such as a 50 µs control loop—while yielding a significant functional density advantage over other reconfiguration techniques. The methodology for addressing the research problem, as discussed in this section, will now be discussed.

1.6 Research methodology

As was the philosophy of the Roman general Julius Caesar, a "divide et impera"[3] approach will be followed in the research methodology. In broad terms, the research constitutes four areas, with the last three analysed using empirical methods:

- Overview of the most relevant literature relating to reconfiguration and reducing the cost thereof (Chapter 2);
- Investigating hardware controlled reconfiguration methodologies (Chapter 3);
- Finding a means to specialise an FPGA configuration (Chapters 4 and 5);
- Reconfiguring real-time applications using the knowledge obtained (Chapter 6).

In order to address these areas, the methodology portrayed in Figure 1.3 will be employed. The most important areas of the figure, along with the way each chapter contributes to the research methodology mentioned above, is discussed in Sections 1.6.1 to 1.6.4. The contributions this thesis makes, along with the chapters describing these, are also highlighted in the figure. An overview of the contributions are given Section 1.7.

[3] "Divide and rule", or as most commonly used: "divide and conquer"

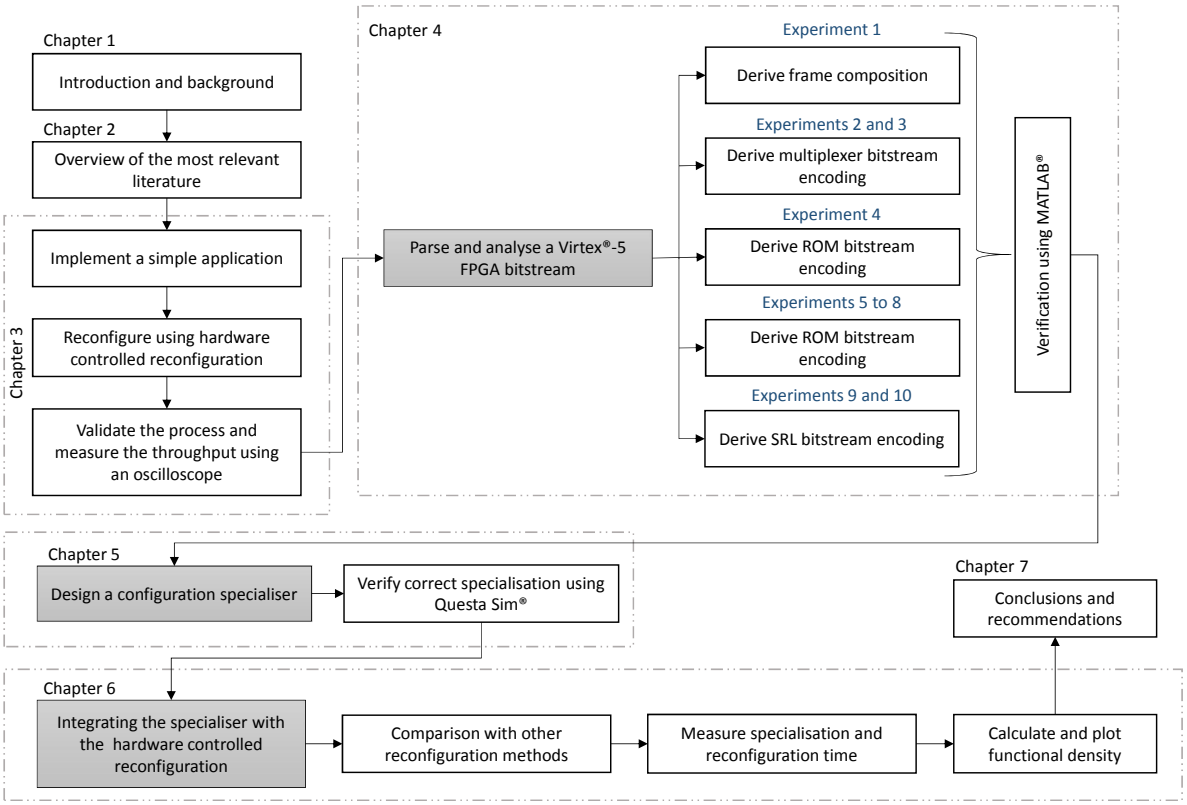


Figure 1.3: Flowchart of the research methodology and chapter breakdown

1.6.1 Overview of the most relevant literature

In order to identify the research contribution, a comprehensive literature survey was undertaken, with specific focus on reducing configuration overhead. The most relevant literature is listed in Chapter 2, which also serves as background and survey for the hardware controlled reconfiguration. The literature relating to the work presented in this thesis but that are not directly relevant, are given in the Appendix A for reference. The literature covered in depth are those touched on in Section 1.4. Adding to this literature are techniques to manipulate FPGA resources at a lower level of abstraction. During the survey it was found that all of these techniques are unsuitable for real-time applications and the hypothesis formulated that FPGA resources can be manipulated by adapting a single configuration stored in the BRAM without additional overhead. This, integrated with the proposed BRAM architecture, will allow an application to be reconfigured in real-time.

1.6.2 Investigating hardware controlled reconfiguration

Chapter 3 investigates hardware controlled reconfiguration by implementing a simple application and reconfiguring it using hardware controlled reconfiguration. In this case the simple application is a blinking LED with the duty cycle being changed by reconfiguration. The development environment used throughout this thesis is Xilinx®'s Integrated Synthesis Environment (ISE®)

Design Suite 14.7. To verify correct operation of both the application and reconfiguration, Questa® Sim 10.0b, an advanced FPGA and system-on-chip (SoC) simulator from Mentor Graphics®, is used. Validation is then done by implementing the design on a Xilinx® ML507 development platform[4].

1.6.3 Specialising an FPGA configuration

Specialising the configuration is done in two parts. The first part, discussed in Chapter 4, is an analysis of the Virtex®-5 configuration with specific focus on the way the lookup tables are encoded. The configurations used for the analysis are generated using the conventional Xilinx® tool workflow, and the analysis thereof is done through ten experiments conducted using MATLAB® R2013b. Each experiment is based on a comparison between a base design's configuration, and modified versions thereof. By comparing the differences between the configurations, and repeating the process for different slices (*SLICEL* or *SLICEM*) and slice configurations (multiplexer, ROM, RAM or SRL), certain characteristics of the configuration are derived.

The second part in specialising the configuration is by using the knowledge obtained by parsing and analysing the configurations to design and implement a configuration specialiser. This is again done using the Xilinx® toolset and verified with Questa® Sim. Validation is done by implementing this specialiser on the development board and measuring the response with an oscilloscope. This is discussed in Chapter 5.

1.6.4 Reconfiguring real-time applications

Lastly, the BRAM-based architecture and bitstream specialiser are combined and verified using Questa® Sim. It is then implemented on the development board and evaluated for real-time reconfiguration by reconfiguring an application—in this case, a distributed multiply-accumulate (DMAC)—using different methods. In each case the reconfiguration and specialisation time (if applicable) are measured and the functional density calculated. The result is then compared by plotting the functional density of each reconfiguration method according to the number of clock cycles the application executes before reconfiguration is required. This determines the break-even point (according to (1.6)) and highlights the advantages and disadvantages of each reconfiguration method. This process and results are given in Chapter 6.

The reason for selecting the DMAC as baseline application, is because it is the foundation of many digital implementations commonly found in real-time applications. Determining whether it is possible to specialise and reconfigure a DMAC within strict time constraints, provides a strong case for reconfiguration in real-time. As a use case[5], the control cycle of a five degree-of-freedom active magnetic bearing system [16] is considered in Chapter 7 to determine whether the specialisation and reconfiguration time of the DMAC fits within one control cycle. In this specific system, proportional-integral-derivative (PID) control is used that relies heavily on

[4] `http://www.xilinx.com/products/boards-and-kits/HW-V5-ML507-UNI-G.htm`

[5] A use case is a methodology used in system analysis to identify, clarify and organise system requirements

multiply-accumulate instructions. The specialisation and reconfiguration of the DMAC is thus a suitable analogue for adapting the control scheme of the system in real-time using dynamic reconfiguration.

1.7 Research contributions

The research presented in this thesis makes three contributions:

Providing new insight into the composition of a Xilinx® FPGA configuration
This is done by proposing a method to parse and analyse the configuration. Even though similar works exist in the literature (as will be discussed in Chapter 2), most rely on a layer of abstraction on top of the configuration which makes them unsuitable for manipulating FPGA resources in real-time. Of particular interest to the work presented in this thesis, is an analysis performed by Castellone [17] on the configuration of a Xilinx® Virtex®-5 VLX110T FPGA. Even though the results he obtained should be usable in real-time, this work is unpublished, unverified and only includes an analysis of lookup tables configured as multiplexers. The work presented in this thesis not only verifies Castellone's work by using a different method, but also expands upon it by considering the encoding of different lookup table constructs. The result is a set of configuration strings capable of being used by a passive specialiser to manipulate FPGA resources at bit-level.

A novel method for specialising an FPGA configuration dynamically
This circumvents the size restriction of the BRAM-based architectures by allowing a configuration stored in the BRAM to be specialised to represent any new set of hardware. The novelty of this specialisation stems from the fact that it is instantaneous, thus allowing usage in real-time applications. It is also device independent and allows the specialisation of any configuration if it is known.

Combining the configuration specialiser with the BRAM-based architecture
This provides a new method of reconfiguring real-time applications. It is shown that despite the additional overhead introduced by the reconfiguration process, it can be reduced to such an extent that reconfiguration is possible for real-time applications. This is quantified using functional density and even though it is automatically higher for a reconfigured application compared to its static counterpart, it is shown that the break-even point can be lowered significantly compared to other reconfiguration techniques.

1.8 Thesis overview

The thesis is divided into seven chapters. This chapter provided some information about the study presented in the thesis and the motivation behind it.

Chapter 2 provides an overview of the supporting literature, with specific focus on methods to reduce (re)configuration overhead. Special attention is given to research aiming at improving

the throughput of the system, as these seem to be the most promising for implementing dynamic reconfiguration in real-time. The focus is then shifted towards different means of manipulating an FPGA configuration.

Chapter 3 investigates hardware controlled reconfiguration, with specific focus on the throughput of these types of systems. In this chapter, the methodology for implementing a hardware-based reconfiguration controller is discussed, and an experimental setup given for verifying the throughput of the system. It is shown that the BRAM-based architecture given in the literature is indeed capable of fast reconfiguration throughput. As a result, this architecture is deemed suitable for implementing real-time reconfiguration. Despite not being novel, the work done in this chapter not only lays the foundation for work in the subsequent chapters, but also aids in understanding why hardware controlled reconfiguration is relevant.

Chapter 4 examines the first contribution made, by discussing the method used to parse and analyse an FPGA configuration. It starts off by providing background on the configuration architecture of a Xilinx® FPGA. Next, the ten experiments performed to parse and analyse the bitstream are discussed and their corresponding results given.

Chapter 5 uses the information obtained in *Chapter 4* to discuss the second contribution. This is done by designing a bitstream specialiser and then integrating it into the reconfiguration architecture. *Chapter 6* uses this new architecture to reconfigure a distributed multiply-accumulate and compares its functional density to other reconfiguration techniques. The thesis concludes with *Chapter 7* by discussing the results of the study and by making recommendations for future work.

1.9 List of publications

1.9.1 Conference contributions

- R. R. le Roux, G. van Schoor, and P. A. van Vuuren, "A survey on reducing reconfiguration cost: reconfigurable PID control as a special case," in *Proceedings of the 19th World Congress of the International Federation of Automatic Control.* IFAC, 2014, pp. 1320–1330
- R. R. le Roux, G. van Schoor, and P. A. van Vuuren, "Block RAM implementation of a reconfigurable real-time PID controller," in *Proceedings of the International Conference on High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th.* IEEE, 2012, pp. 1383–1390

CHAPTER 2

STATE OF THE ART

"No problem can be solved from the same level of consciousness that created it."

— Albert Einstein

The research question proposed by this thesis is whether it is possible to improve the overhead induced by the reconfiguration process to such an extent that it can be used in real-time applications. To answer this question, it is first necessary to investigate the efforts made by other researchers to reduce the reconfiguration cost. The primary contributor to this cost is the placement and routing (PAR) of conventional FPGA design tools. Therefore, methods to reduce this cost are investigated. An alternative approach is to change the way the configuration is generated. This includes using compression techniques, reducing the quality of the design or reusing hardware from a previous configuration. The most common methods are investigated and are listed in this chapter. Of particular interest for real-time reconfiguration is to improve the throughput of the system to rival that of the ICAP. Consequently, it is given special attention. The chapter then concludes by discussing different means of manipulating FPGA resources directly to overcome some of the limitations imposed by the aforementioned architectures to improve throughput.

2.1 Introduction to reconfiguration

Reconfigurable computing is a paradigm in computing architecture that refers to the practice of using interchangeable hardware modules to enhance the performance of conventional Von-Neumann style computing [20]. Initially, this was done by physically swapping a hardware module with another more suitable for the specific application.

As their name suggests, field-programmable gate arrays (FPGAs) allow their hardware to be changed on-the-fly, and as such, is a viable implementation platform for reconfigurable

computing. On start-up, an FPGA is usually *configured* using a serial string of bits, called a bitstream. This only takes place once, and is also referred to as compile time configuration. Some sources (such as [21]) also refer to this as compile time reconfiguration, but for the purpose of this thesis, "*configuration*" is rather used to avoid confusion.

Reconfiguration, on the other hand, refers to the action of modifying the content of the FPGA during run-time. Two types of reconfiguration exist for FPGAs: partial and full. *Full reconfiguration* is identical to the initial *configuration* of the device and replaces the entire configuration of the FPGA. *Partial reconfiguration (PR)*, on the other hand, is "... the ability to reconfigure preselected areas of an FPGA any time after its initial configuration, while the design is operational" [22]. A feature called *dynamic partial reconfiguration* allows FPGAs to change a section of their hardware while the rest of the device remains operational. All Xilinx®'s FPGAs from the Virtex®-II family incorporate this feature, since they include an internal configuration access port (ICAP) that provides access to the configuration registers of the FPGA.

Initially, the Xilinx® toolset supported two flows for implementing dynamic partial reconfiguration: *module-based* and *difference-based* [23]. The former permitted the reconfiguration of distinct modular sections of the design, whereas the latter allowed a designer to make small logic changes on-the-fly. Over the years, the module-based reconfiguration design flow has evolved through different aliases, such as *Early-Access partial reconfiguration (EAPR)*, but eventually Xilinx® settled on a generic "*partial reconfiguration*" for all partial reconfiguration that takes place during run-time. Throughout this thesis, the same naming convention will be used.

Dynamic reconfiguration of a system has numerous advantages, which includes improving the performance of the hardware by tailoring it for a specific application, as well as reducing power consumption and component count [24–26]. Despite these advantages, dynamically reconfiguring an application has one primary disadvantage; it is only advantageous if the execution time exceeds the reconfiguration time [27, 28]. This implies that dynamic reconfiguration is only really suitable for quasi-static applications, such as key specific data encryption standard (DES) [27], sub-graph isomorphism [29], Boolean satisfiability (SAT) [30], adaptive filters [11], reconfigurable artificial neural networks [8], digital signal processing [31], image processing [32], control systems [33, 34] and frequency-shift keying (FSK) modulation [35, 36]. Typical reconfiguration time could range from milliseconds, for dynamic partial reconfiguration, to hours for full reconfiguration.

Figure 2.1 shows a summary of the different types of reconfiguration discussed in the previous sections. In summary, the following nomenclature is used in this thesis when referring to (re)configuration:

Configuration Usually refers to the initial set-up of the device by loading a configuration file, also called bitstream, to the FPGA's configuration memory, but could also indicate any process where a configuration is loaded onto the device, whether dynamically of statically

(Dynamic) Reconfiguration Any adaptation taking place while the device is operational, regardless of the method used. In some cases the "dynamic" is explicitly added to avoid confusion.

Partial reconfiguration Any reconfiguration that modifies a modular section of the device while it is operational.

Difference-based reconfiguration Reconfiguration based on the difference between two designs. Usually only small changes are made.

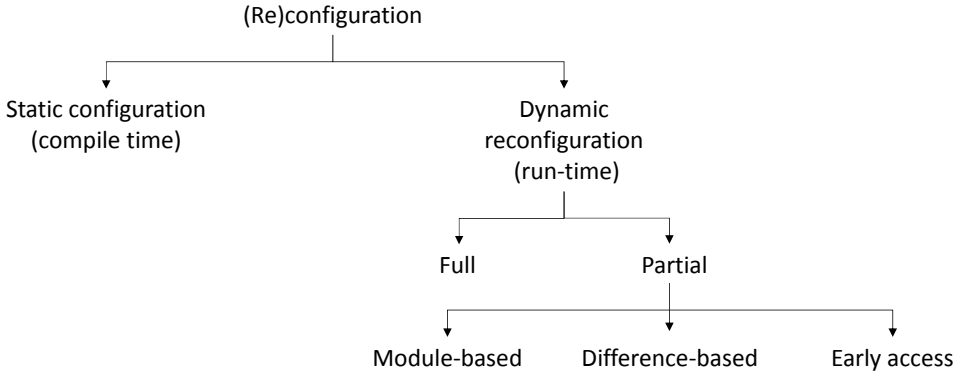


Figure 2.1: Tree diagram showing the different types of (re)configuration

2.2 Reducing reconfiguration cost

As discussed in Chapter 1, the reason why dynamic reconfiguration is unsuitable for real-time applications, is due to the large overhead introduced by the process. This overhead is either from the additional hardware required to facilitate the reconfiguration process, or the time required to generate new hardware. The primary contributor to the latter is the placement and routing (PAR) required to generate instance-specific configurations. Traditionally, negotiation-based algorithms are used to determine the optimal placement and routing, adding significant overhead to the cost of dynamic reconfiguration. Various researchers aim to mitigate this overhead using the methods listed in Table A.2 and Table A.1 found in Appendix A. Even though the research contributions are not always as clear-cut as the tables make out to be, and a lot of the fields could overlap, the aim is to sketch a global picture of the research in the field.

Claus [37] proposes three methods to reduce configuration cost when PAR is completed. The aim of these methods is to improve the throughput of the system to rival that of the ICAP. This allows the ICAP to process new data every clock cycle. In general, these methods can be summarised as:

- reducing the bitstream size,
- optimizing the way the bitstreams are written to the configuration memory,
- optimizing the transfer of the bitstream from the memory to the ICAP.

FPGAs are reconfigured by transferring a serial string of bits, called a bitstream, to the configuration memory. This implies that a smaller bitstream requires less time to be transferred

and by changing the way it is written to configuration memory, allows the reconfiguration to be reduced. As will be shown in the next section, various attempts have been made to adapt the bitstream for faster reconfiguration. The transfer of the bitstream to the configuration memory is governed by the reconfiguration architecture used. As will be shown in section 2.2.2, these architectures can be adapted to reduce the configuration time.

2.2.1 Bitstream generation

The bitstream contains the configuration data of the FPGA and can be generated on-line or off-line. The latter implies that the bitstreams are generated independently from the FPGA, usually with conventional design tools. These bitstreams can contain the information required to configure the FPGA with an initial configuration, or partial configuration data used during dynamic reconfiguration. For a limited set of configurations, these bitstreams can be stored in on-board memory from where the FPGA can be reconfigured. Applications where this technique have been successful include deoxyribonucleic acid (DNA) sequencing [38], neural networks [10] and automatic target recognition [39].

On-line bitstream generation refers to generating a configuration dynamically while the FPGA is running. It is possible to use conventional tools, but this induces a significant amount of configuration overhead due to the time required to complete the process. As a result, various changes are made in the way the bitstreams are generated. The prominent methods are summarised in Appendix A, Table A.3. Note that many of these methods encapsulate principles listed in Table A.2 and Table A.1 for reducing PAR cost.

2.2.2 Reconfiguration throughput

Reconfiguration throughput refers to the maintainable bit transfer rate between the memory housing the configuration data and the configuration memory. Assuming that the ICAP is capable of processing data every clock cycle, the maximum theoretical throughput (MTT) is defined by [37]:

$$MTT = \frac{IDIW}{Clock\ period}, \tag{2.1}$$

with *IDIW* the ICAP data input width, which is 8 and 32-bit for the Virtex®-II and 5 respectively. The maximum recommended clock frequency for the ICAP is 100 MHz. Substituting these values into (2.1) results in an MTT of 800 Mbps for the Virtex®-II and 3.2 Gbps for the Virtex®-5 to 7.

The reconfiguration time of the system is directly related to the size of the bitstream and the throughput of the ICAP. This is illustrated in Figure 2.2, which shows the calculated configuration latency of each device in the Virtex®-5 family of FPGAs when 10%, 25%, 50%, 75% and 100% of the device is respectively reconfigured.

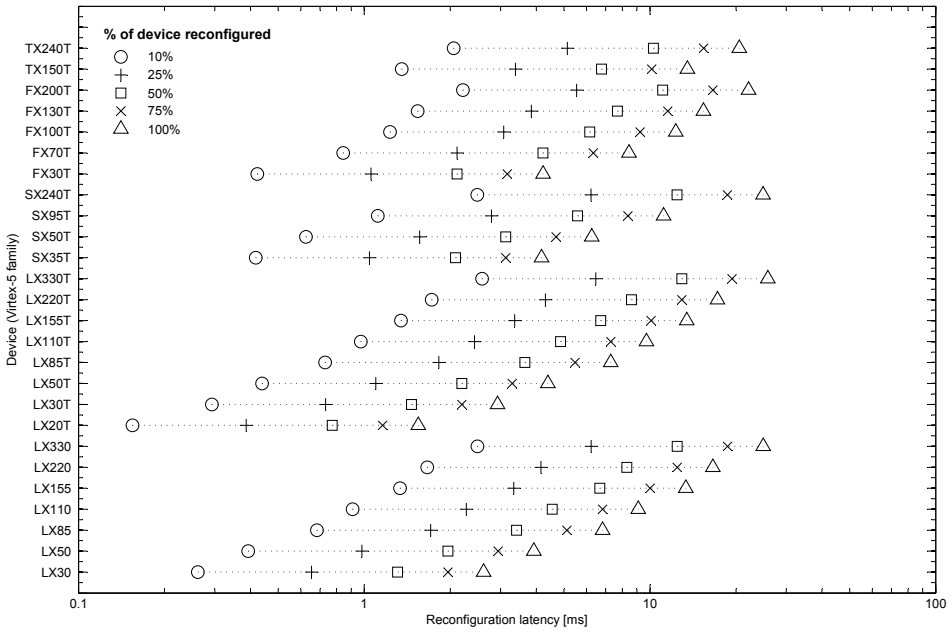


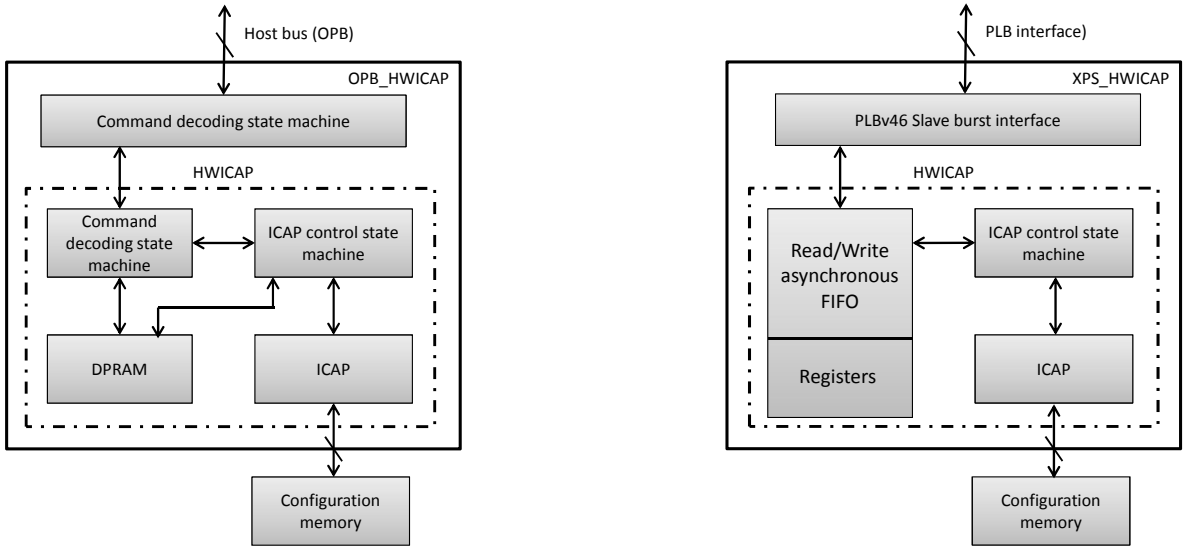
Figure 2.2: The reconfiguration latency of the Xilinx® Virtex®-5 FPGA family

Traditionally, access to the ICAP is made possible by using the hardware internal configuration access port (HWICAP)-peripheral attached to the On-chip Peripheral Bus (OPB) or by using the Xilinx® Intellectual Property Interface (IPIF) attached to the Processor Local Bus (PLB), as illustrated in Figure 2.3. In both these cases, the operations of the ICAP are controlled by software running on the processor core (PowerPC® [40, 41] or MicroBlaze® [42]) of the FPGA. The drawback of these architectures is that the OPB and PLB take relatively large amounts of resources and have high overhead, causing the throughput of the system to be significantly lower than the MTT of the ICAP. In fact, it is estimated that about 40% of the overhead is contributed by the Xilinx® HWICAP driver function [12]. As shown in Table 2.1, the HWICAP is not even capable of 20 MB/s when clocked at the recommended 100 MHz.

The reconfiguration time is bounded by the throughput of the ICAP. One way to lower the reconfiguration time is thus to ensure that the ICAP operates at the maximum available throughput. As shown in Table 2.2, this can be achieved by adding a direct memory access (*DMA*) controller, using custom reconfiguration *Controller*s, *Compress*ion techniques, *Streaming* or overclocking techniques.

Figure 2.4 illustrates a reconfigurable architecture with *DMA*, which allows the *Controller* to access the bitstream located in the *Memory*-location, across a bus, without processor intervention. This improves efficiency since the embedded processor is relieved from the configuration process. The addition of a multi-port memory controller (MPMC) can allow the DMA controller to access the external memory directly without the need of a system bus. Streaming modes are used in conjunction with DMA to improve the throughput by loading the bitstream continuously as needed, compared to the fetch-and-configure model of the traditional reconfiguration process. This ensures that the local buffer, normally a FIFO that feeds the ICAP with configuration data, is always full. Bitstream compression (*Compressed*) reduces the size of the bitstream thereby reducing the time required to transfer it from the memory location. The

primary drawback of this method it that decompression techniques could be detrimental to reconfiguration time due to the amount of overhead it adds.



(a) OPB HWICAP [43]

(b) PLB HWICAP [44]

Figure 2.3: Block diagrams depicting the Xilinx® proprietary ICAP controller

Table 2.1: Reconfiguration throughput of the Xilinx ICAP controllers (obtained from [45])

Method	**Config port**	**Bus type**	**Stream**	**Memory**	**DMA**	**Controller**	**Compressed**	**Throughput [MB/sec]**
OPB_HWICAP (PowerPC® cache disabled)	ICAP32 @100 MHZ	OPB	N	DDR2	N	Vendor	N	0.61
XPS_HWICAP (PowerPC® cache disabled)	ICAP32 @100 MHZ	PLB	N	DDR2	N	Vendor	N	0.82
OPB_HWICAP (PowerPC® cache enabled)	ICAP32 @100 MHZ	OPB	N	DDR2	N	Vendor	N	10.10
XPS_HWICAP (PowerPC® cache enabled)	ICAP32 @100 MHZ	PLB	N	DDR2	N	Vendor	N	19.10
OPB_HWICAP (MicroBlaze cache enabled)	ICAP32 @100 MHZ	OPB	N	DDR2	N	Vendor	N	6.0
XPS_HWICAP (MicroBlaze cache enabled)	ICAP32 @100 MHZ	PLB	N	DDR2	N	Vendor	N	14.6

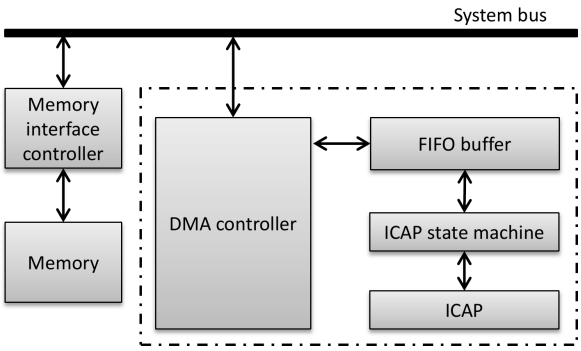


Figure 2.4: Block diagram of a reconfigurable architecture with DMA [45]

The values listed under the table heading *Config port* indicate the width of the ICAP input port as well as the frequency at which it was clocked. Even though Xilinx® recommends a maximum clock frequency of 100 MHz for ICAP stability, various researchers have shown that it is possible to use the ICAP above this frequency, which could result in a higher throughput.

Papadimitriou [46] analysed different reconfiguration architectures and developed a cost model which can be used to calculate the expected reconfiguration time and throughput. This is done by taking into account all the physical components that participate in the reconfiguration process. By using this cost model they tried to predict the reconfiguration time and throughput of a system, with varied success. In some cases they were off by up to 63.1%. However, and most importantly, they also did an analysis of the system factors that contribute to the reconfiguration overhead, which is applicable to all the architectures listed in Tables 2.1 and 2.2. The factors most applicable for this thesis are:

- The throughput cannot be calculated from the bandwidth of the configuration port alone. The overhead of all the components participating in the reconfiguration process also has to be taken into account.
- The characteristics that affect reconfiguration overhead depend on the system set-up, which includes the type of memory used, the memory controller, the reconfiguration controller and interfaces.
- Using a bus-based system to connect the processor, the partially reconfigurable module(s), the static module(s), and the configuration port, can be non-deterministic and unstable due to the contention between data and reconfiguration transfers.
- Using a dedicated controller equipped with DMA capabilities, are capable of nearly the full bandwidth of the configuration port.
- Enabling the cache of the processor can improve the reconfiguration throughput significantly at the expense of additional resource utilisation.
- Using the Xilinx® provided application programming interface (API) for software control of the HWICAP is slow.
- CompactFlash is an extremely slow memory space and using volatile high speed memory can improve performance.

- Implementing large on-chip memory with BRAM attached to the configuration port can allow for fast reconfiguration, but due to the limited size of the BRAM, the utilisation cost has to be considered.

The last factor listed is of particular interest, because even though many of the methods listed in Table 2.2 are capable of reconfiguration throughputs rivalling (in some cases even exceeding) that of the ICAP, most of them suffer from configuration delay. This is either due to the transfer of configuration data from external memory locations, the compression algorithms used, or from the DMA controller. Liu et al. [57] aimed to minimise the configuration overhead by incorporating streaming, compression and DMA into an intelligent ICAP controller. Despite their experimental results showing their implementation nearly saturates the throughput of the ICAP, the DMA and compression adds configuration overhead of 17 and 6 clock cycles respectively.

The BRAM-based architecture, shown in Figure 2.5, is capable of extremely fast reconfiguration times and, because it is tightly coupled to the configuration controller and port, mitigates all bus-induced overhead and has zero delay. Not only that, but Hansen [60] has shown that it is possible to overclock the ICAP to 5.5 times the recommended clock frequency if custom hardware is used. Unfortunately, as mentioned, the BRAM is seen as an expensive resource since it is extremely limited. Bitstreams too large to fit into the BRAM can be loaded into the BRAM using the processor bus, or compressed to fit using compression techniques. However, even though complex compression techniques are capable of reducing the bitstream significantly [62], the decompression algorithm adds to the reconfiguration time. The more complex the algorithms, the bigger the impact on reconfiguration time.

A different approach is to store only a single configuration in the BRAM and then to specialise it to represent different hardware sets. This can either be done by manipulating the bits in the bitstream directly—also called direct bitstream manipulation—or by more elaborate means, such as stack machines. The work in the next section discusses the state of the art relating not only to manipulating the bitstream directly, but also different means of editing the resources of an FPGA on hardware level.

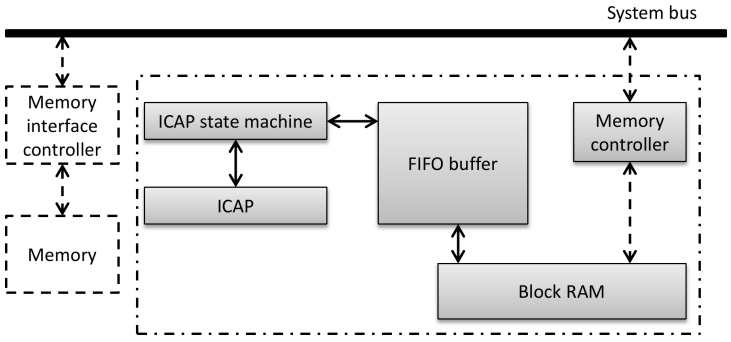


Figure 2.5: Block diagram of a reconfigurable architecture utilizing local BRAM [45]

Table 2.2: Methods to improve reconfiguration throughput

Author	**Config port [width@MHz]**	**Bus type**	**Stream**	**Memory**	**DMA**	**Controller**	**Compress**	**Throughput [MB/sec]**
Papadimitriou [47]	ICAP8@100	OPB	N	CompactFlash	N	Vendor	N	0.15
Papadimitriou [48]	ICAP8@100	OPB	N	BRAM	N	Vendor	N	1.46
Claus [37]	ICAP8@100	OPB	N	DDR	Y	Vendor	Combitgen	4.66
Claus [49]	ICAP8@100	OPB	N	DDR	Y	Vendor	N	4.77
Claus [49]	ICAP32@100	OPB	N	DDR2	Y	Vendor	N	5.07
Lamonnier [50]	ICAP32@33	None	N	Platform Flash	Y	Custom (FSM)	N	33
Delahaye [51]	ICAP8	OPB	N	SRAM	N	Vendor	N	49.2
Liu [45]	ICAP32@100	PLB	N	DDR2	Y	Custom	N	82.1
Claus [49]	ICAP8@100	PLB	N	DDR	Y	Custom	N	89.9
Claus [37]	ICAP8@100	PLB	N	DDR	Y	Custom	Combitgen	91.34
Van der Bok [52]	ICAP8@100	PLB	N	N/A	Y	Custom	N	100 (est.)
Cuoccio [53]	ICAP8	OPB/PLB	N	BRAM	Y	Custom	N	100
Nabina [54]	ICAP32@100	N/A[1]	Y	DRAM	Y	Custom (FlashCAP)	N	170
Hoffman [55]	ICAP32@100	PLB	N	DDR	Y	Custom	N	180
Hoffman [55]	ICAP32@100	PLB	N	DDR2	Y	Custom	N	181
Shelburn [56]	ICAP32@100	NoC	N	BRAM	N	Custom (MetaWire)	N	219.31
Liu [45]	ICAP32@100	PLB	N	DDR2	N	Custom	N	234.5
Claus [49]	ICAP32@100	PLB	N	DDR2	Y	Custom	N	295.4
Liu [45]	ICAP32@100	PLB	N	BRAM	N	Custom	Y	332.1
Hoffman [55]	ICAP32@100	PLB	N	DDR2	Y	Custom	N	340.4
Nabina [54]	ICAP32@100	N/A[1]	Y	DRAM	Y	Custom (FlashCAP)	N/A[2]	385
Liu [57]	ICAP32@100	PLB	Y	SRAM	Y	custom	Y	392.74
Liu [57]	ICAP32@100	PLB	Y	SRAM	Y	Custom	Y	399.79
Hoffman [55]	ICAP32@133	PLB	N	DDR2	Y	Custom	N	424.6
Duhem [58]	ICAP32@200	PLB	N	DDR	Y	Custom (FaRM)	Y[3]	800
Bonamy [59]	ICAP32@255	PLB	Y	N/A	Y	Custom (UPaRC)	Y	1,008
Bonamy [59]	ICAP32@362.5	PLB	Y	N/A	Y	Custom (UPaRC)	N	1,433
Hansen [60]	ICAP32@500	None	N	PC	N	Custom (ICAP_64)	N	2,000
Hansen [60]	ICAP32@533	None	N	PC	N	Custom (ICAP_64)	N	2,132
Hansen [60]	ICAP32@550	None	N	PC	N	Custom (ICAP_64)	N	2,200

[1] Used a LEON3 platform with an AMBA AHB bus

[2] Used a lossless compression algorithm called X-MatchPRO [61]

[3] Run-length encoding

2.3 Manipulating FPGA resources

Evolvable hardware (EHW)[1] is a research field aimed at evolving a circuit according to principles and mechanisms similar to those that characterise natural evolution [64]. The general hypothesis is that circuits can be created in a similar fashion to biological systems in nature, which are seen as superior to the artificial systems created by mankind. It relies on an evolutionary algorithm (EA) to change the bitstream to represent the newly evolved circuit [64–68].

EHW systems are classified as analogue, mixed analogue-digital or digital, with most EHW systems falling into this latter category. As a result, FPGAs are most commonly used as the implementation platform. The first FPGA-based EHW experiment was performed by Thompson [69] in 1996, who used a Xilinx® XC6216 FPGA to evolve a frequency discriminator. The X6200-series FPGA had a known bitstream format, and combined with its multiplexer-based architecture, allowed reconfiguration using any arbitrary bitstream without running the risk of resource contention. This FPGA also allowed dynamic reconfiguration, which permits a section of the device to be reconfigured while remaining operational, making it the perfect platform for implementing adaptive algorithms.

Unfortunately, the FPGAs succeeding the XC6200-series all had proprietary and undocumented bitstream descriptions. This was not only due to the rise of emerging technologies from other FPGA vendors, but also to protect the user from intellectual property (IP) theft. Static RAM (SRAM)-based FPGAs are susceptible to man-in-the-middle attacks, since the configuration file has to be transferred from the non-volatile external memory space when the device is configured. Subsequently, most FPGA vendors provide bitstream encryption as a method of protection. Even though it is possible to break the encryption [70, 71], the interpretation thereof is seen as non-trivial and is referred to as security through obscurity [72].

Due to the obscurity in the bitstream, EHW researchers introduced virtual reconfigurable circuits (VRCs) [73]. A VRC is a virtual reconfiguration layer built on top of the FPGA device fabric using a set of multiplexers and other application specific hardware. Its aim is to reduce the complexity of the reconfiguration while increasing speed and allows safe manipulation of FPGA resources. VRC allowed researchers to investigate EHW on a magnitude of FPGAs, including the Xilinx® Virtex®-5 LX110T FPGA [74] and Zynq®-7000 FPGA [75]. The drawback of VRCs is that it takes a tremendous amount of resources, since the reconfiguration involves writing to a large register, resulting in an extremely large area utilization [76]. VRCs are also unsuitable for real-time applications, as typical evolution times are in the order of microseconds [74, 77] and the evolution has to occur multiple times before an optimal solution is found. As a result, some researchers started investigating the manipulation of the bitstream bits directly.

To allow access into the obscured configuration bits of an FPGA, JBits™ was developed at the University of Massachusetts, Amherst. It provided an application programming interface (API) to the Xilinx® XC4000, Virtex® and Virtex®-II FPGA bitstream [78, 79]. It was later extended to include the Virtex®-E, Virtex®-II Pro and early Spartan® families, but this extended capability was never officially acknowledged or released [80]. Drawbacks of this API was that it did not run on embedded systems and was dependent on a select set of devices. It

[1] A review of the field can be found in [63]

also required the user to have a complete knowledge of the architecture in order to hand craft a solution [81].

Even though JBits was an unofficial, Xilinx®-supported tool, it was largely used for experimental purposes. An attempt to launch an official Xilinx® tool with similar functionality, called XPART (Xilinx Partial Reconfiguration Toolkit) [82], was never released. Instead, access to the resources of newer devices is provided by the Xilinx® design language (XDL), which provides a powerful interface into virtually all the features of their FPGAs [83]. Using this language, it is possible to generate complete descriptions for a specific device, which also contains information regarding FPGA primitives and routing. It is also possible to use the language to constrain systems or to implement modules or macros. For instance, XDL is capable of implementing a complete design as an XDL netlist description, and vice versa. In layman's terms, an XDL netlist description is a human readable version of the Xilinx® Native Circuit Description file (NCD) and describes an implementation after technology mapping of a design to FPGA primitives.

Beckhoff [83] states that there are three types of reconfigurable computing projects that make use of XDL:

1. **Macro generation**: Uses XDL as an interface to create regular structured macros [84–89].
2. **Providing an API to XDL**: Aims to provide a developer with a graphical interface to read, write and manipulate an XDL file [90–92].
3. **Documentation**: The documentation available for XDL is quite limited and the need exists for more recent documentation.

From this list, the most obvious limitation of XDL is the lack of documentation. Even though both XDL descriptions (resource and netlist) are considered self-explanatory, the files generated are device and toolset dependent and up-to-date documentation would be beneficial. It is also worth mentioning that the resource description generated by XDL is massive in file size, and could range from 3.5 GB for a Virtex®-4 SX55 to 73.6 GB for the Virtex®-7 2000T [93]. It is thus difficult to manage and analyse its contents. To make these files more manageable, many of the tools mentioned in the list above use compression to reduce the file size, but this does not aid simplifying the analysis process. Even though XDL is capable of modifying the resources of an FPGA, it is mostly used as a design tool during compile time. While some approaches in the literature also incorporate techniques to generate partial configurations, these are generated off-line since they require manual floorplan design.

The Debit-project [71] was an attempt at providing JBits-esque functionality, by extracting information from the bitstream through correlation. By comparing a design's bitstream with its corresponding XDL netlist, they were able to develop software allowing the composition and decomposition of a bitstream. This is quite significant since their *xdl2bit*-tool for creating a bitstream from an XDL netlist file is significantly faster than Xilinx®'s proprietary BitGen™. In fact, it was shown that a speedup of about 100 times can be achieved, depending on the hardware configuration used to generate the bitstream. They also extracted significant information regarding the bitstream composition of the Virtex®-II, 4 and 5 families of FPGAs.

Unfortunately, the approach followed was very basic, the mapping of the bitstream is largely incomplete and the project has since been discontinued.

The bitfile interpretation library, Bil [70], followed in the footsteps of the Debit-project and aimed at improving the work by considering the XDL resource description file for all subtypes of Virtex®-5 FPGAs in conjunction with the XDL netlist, to obtain additional information. As discussed earlier, the drawback of using the XDL resource description, is the file size. As a result, Bil uses compression techniques to compress the file to 0.001% of its original size. This has to be done for every device family analysed, which can take up to 20 minutes, but a 13 GB file can be compressed to just 15 MB. Unfortunately, neither Bil nor Debit was able to fully reverse the bitstream. They did, however, make significant contributions to the analysis of a bitstream and understanding the relation between XDL and the bitstream.

Several other researchers were also successful in extracting information from the bitstream. PARBIT (PARtial BItfile Transformer) [94] enables a reconfigurable region to be moved to another location by extracting configuration frames from a bitstream and copying it to a partial bitstream. It then generates new values for the address registers, based on the reconfigurable area. In order to achieve this, the creators had to analyse the composition of a Virtex® bitstream in an attempt to locate the configuration data. However, the configuration data itself were not analysed or changed. BitMaT (Bitstream Manipulation Tool) [95] aimed to achieve similar functionality, but only targeted the bitstreams of the Virtex®-II and Virtex®-II Pro FPGAs. Both PARBIT and BitMaT aim to enhance the module-based reconfiguration tool flow by creating partial bitstreams at bit-level, but neither is capable of changing the contents of a module's configuration.

Upegui and Sanchez targeted a Virtex®-II Pro FPGA and modified only the LUTs, while keeping routing intact by using hard macros [68, 96]. Unfortunately, this architecture only has one-dimensional (1D) routing, implying that routing takes place in only one direction [97]. Most newer FPGAs have two-dimensional (2D) routing and Upegui and Sachez's proposed method is unsuitable for newer devices. Similar efforts on more recent FPGA families include those by Cancaré *et al.* [76, 98] who proposed a direct bitstream manipulation technique for Virtex®-4 devices based on the work done by Castagna [99] and Soni *et al.* [80]. Soni proposed an open-source method to produce a bitstream for a Virtex®-5 FPGA, based on the TorC tool flow [91, 100].

Of particular interest to the work presented in this thesis, is an unpublished technical report written by Castellone [17], wherein he proposes a method for mapping the configuration space of a Xilinx® Virtex®-5 VLX110T FPGA to the bitstream. A shortcoming of this method was that only combinational logic was considered when determining the bitstream encoding of the LUTs.

Parameterised configurations is a method proposed by Bruneel [12], wherein the configuration bits of an FPGA are abstracted to a multivalued Boolean function of a set of parameters, called tuning functions. By evaluating these functions, a new configuration is generated. Bruneel chose a stack machine for this purpose, due to the high code density compared to other machines. The main task of the stack machine is to evaluate the parameterised bitstream and to produce a new configuration for the FPGA.

2.4 Concluding remarks

Despite the numerous advantages reconfigurable computing adds to a system, it is mostly limited by the long reconfiguration time. This makes reconfigurable computing only suitable for quasi-static applications. In order to allow reconfiguration of applications with dynamically changing parameters, such as in real-time control, various researchers aim to minimise the cost of reconfiguration. It was shown that a significant portion of this research aims to reduce the cost of PAR, but despite best efforts, this cost could not be reduced significantly enough to allow PAR to be performed in real-time.

Another approach is to change the way the bitstream is generated. The aim is not only to enable bitstreams to be generated on-line, but also to reduce the amount of configuration data inside the bitstream. Less information in the bitstream allows for faster reconfiguration, since less data have to be transferred to the ICAP. Unfortunately, most of these techniques still rely on some form of PAR, which makes it unsuitable for use in real-time applications.

A major contributor to the cost of reconfiguration is the bus-based architectures and HWICAP driver most commonly used. By amending these architectures, researchers have shown that it is possible for the system's throughput to rival that of the ICAP and even exceed it. The most promising architecture for improving the throughput of the system is one that uses a hardware-based reconfiguration controller reading the configuration stored in local BRAM. Despite being capable of saturating the ICAP, BRAM is seen as an expensive resource and due to its limited size, only a subset of configurations can be stored. To address this limitation, techniques were investigated that allow FPGA resources to be adapted at a lower level. This allows a single configuration stored in the BRAM to be specialised according to a set of parameters to represent any other hardware configuration.

In order to achieve this specialisation, it is necessary to know the composition of the bitstream. Unfortunately, this is hindered by obscurity. Researchers aim to address this by either using XDL to extract information from the bitstream, or by novel analysis techniques. However, none of these techniques allow manipulating the FPGA resources at run-time. The only way this can be achieved is by manipulating the bits of the bitstream directly. Even though great headway was made in this regard, most of the work done was either on older devices or the manipulation methods are too complex to apply in real-time. As a result, an alternative method to parse and analyse the bitstream is proposed in this thesis, which will allow the manipulation of FPGA resources in real-time.

Before this method can be discussed, it is first necessary to evaluate the viability of the BRAM-based architecture for real-time applications by implementing a simple application, reconfiguring it and calculating the throughput. This is done in the next chapter.

CHAPTER 3

HARDWARE CONTROLLED RECONFIGURATION

"To achieve great things, two things are needed; a plan, and not quite enough time."

— Leonard Bernstein

The block-RAM (BRAM)-based architecture discussed in the previous chapter was identified as the ideal architecture for implementing real-time reconfiguration. This is due to its minimal overhead (compared to other architectures), which improves the throughput to such an extent that the ICAP is saturated. Unfortunately, the use of this architecture is hindered by the limited size of the BRAM. This led to the hypothesis that it should be possible to specialise the configuration in the BRAM to represent any number of configurations. This specialisation should occur with the least amount of overhead to still allow this architecture to be used in real-time applications. This chapter discusses the proposed architecture and aims to verify the reconfiguration throughput thereof.

3.1 Proposed architecture

The architecture proposed by this thesis is shown in Figure 3.1. It is based on the BRAM-based architecture discussed in Section 2.2.2 and shown in Figure 2.5, with the exception of the configuration specialiser based on the parameterizable configuration proposed by Bruneel [12]. This was included into the architecture, since the idea of specialising the bitstream and the specialiser proposed later in the thesis originated from his work.

Dynamic circuit specialisation (DCS) is a technique proposed in the literature to dynamically specialise an FPGA configuration according to the values of a set of parameters. The idea is that each time a parameter changes, the device is reconfigured with a configuration tailored to the new parameter values. Each DCS implementation had to be hand-crafted at architectural level for a specific application. This led to an increase in development time and cost. Bruneel

proposed a DCS method called parameterizable configuration, wherein the bits representing the FPGA configuration are expressed as a Boolean function of a set of parameters. The result is a parameterised bitstream or PBS. Using a PBS specializer, any specific set of parameters can be evaluated to transform the PBS into a regular configuration. This process is illustrated in Figure 3.2.

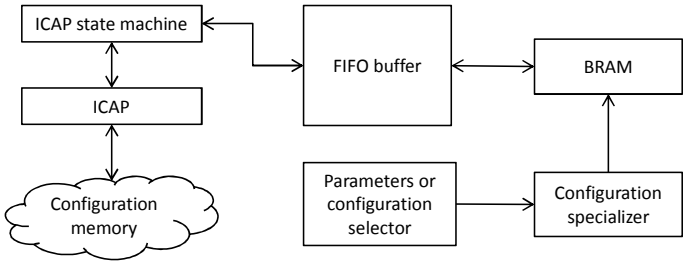


Figure 3.1: Block diagram of the proposed BRAM-based architecture with specialiser

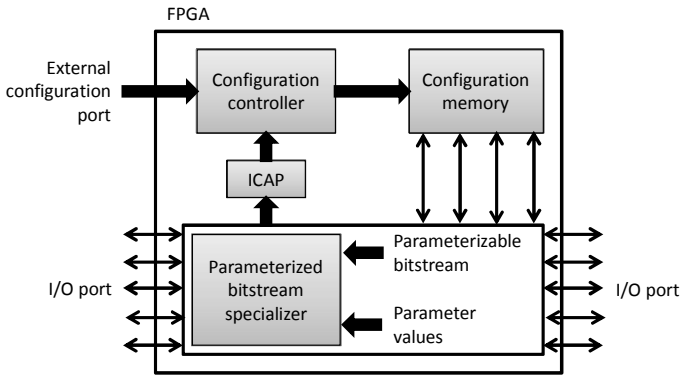


Figure 3.2: A block diagram depicting the architecture of Bruneel's proposed parameterisable configuration

3.2 Parameterizable configuration

The tool set used by Bruneel enables automatic implementation of DCS by means of two methods. The first expresses only the bits of the lookup tables in the configuration file as Boolean functions. These lookup tables are dubbed tunable lookup tables (TLUTs) [101]. All other configuration bits are static. The second method expands on this methodology and adds tunable routing bits. This new method is dubbed the tunable connections, or TCON, method [102]. Even though TCON leads to more compact solutions, TLUT results in faster reconfiguration. Using these methods, Bruneel built several parameterizable systems. These include finite impulse response (FIR) filters, ternary content-addressable memory (TCAM), key-based encryption and DNA alignment systems that run on commercial FPGAs [11, 38, 103]. The specialisation was done using the embedded PowerPC® and ICAP of the Virtex®-II Pro. It was determined that a coefficient change of the FIR filter can be reconfigured in 1.74 ms, whereas the content of the TCAM can be rewritten in 1.72 ms.

Bruneel's proposed reconfiguration method was only used to successfully specialise quasi-static applications. This is due to the significant overhead added by the PBS specialiser that has to execute after each parameter change. Bruneel showed that a 32-tap FIR filter takes around 12.79 ms to be specialised [12]. Parameterising a module might thus not yield the expected benefits of reconfiguration. Nevertheless, this architecture serves as the basis for the specialiser proposed in the introduction of this chapter, which will be discussed in detail in Chapter 5. The rest of this chapter is dedicated to evaluating the BRAM-based architecture as a proof of concept for the research presented in this thesis. The next section is devoted to specific aspects of the design method used to implement this architecture.

3.3 Design flow

The design of the system was done by using Xilinx®'s partial reconfiguration (PR) design flow in the ISE® Design Suite [104]. Even though a newer flow is available for Xilinx®'s latest design suite, Vivado®, it is only available for the newer Virtex®-7, Kintex®-7 and Artix®-7 FPGA families, as well at the Zynq®-7000 All Programmable SoC family. The ISE® PR flow, on the other hand, is not only supported by older FPGA families, but also applicable to the aforementioned families.

The basic premise of the PR flow is shown in Figure 3.3. As shown, the function implemented in *Reconfigurable block 'A'* is modified by downloading several configurations—*A1.bit*, *A2.bit*, *A3.bit* and *A4.bit*—to the device. The logic on the device is either reconfigured or static, the latter implying that it is not affected by the reconfiguration process. Each reconfigurable block and its configuration is created using Xilinx®'s PlanAhead™ tool. Using this tool, the area, placement and requirements of each block can be specified—which also include those for partial reconfiguration. If done correctly, the resulting blocks can be swapped dynamically to and from the device without interrupting operation. A tutorial to using PlanAhead™ for implementing a simple reconfigurable application can be found in [105].

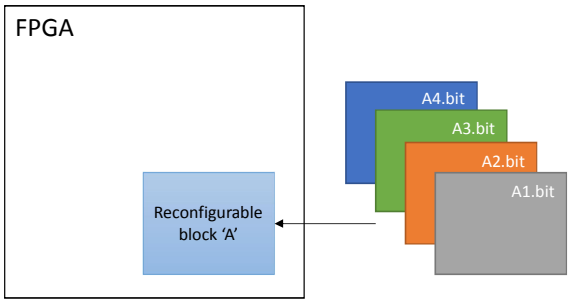


Figure 3.3: Basic premise of partial reconfiguration illustrating configurations being swapped to and from the device [104]

3.4 Hardware controlled reconfiguration

The BRAM-based architecture uses a purely hardware controlled reconfiguration scheme, implying the use of hardware to control the reconfiguration process—compared to conventional methods that require a processor bus, such as the processor local bus (PLB) or Xilinx® Platform Studio (XPS) bus. This involves the use of a state machine to facilitate the reconfiguration process, based on MultiBoot—a Xilinx®-included feature allowing an active application to fall back to a previous good configuration in the event of a failure [106].

3.4.1 Internal configuration access port (ICAP)

The ICAP, as the name suggests, is an internal port closely related to the SelectMAP configuration interface, and provides bidirectional access to the FPGA's configuration logic. The timing for both interfaces is illustrated in Figure 3.4. The only difference is the *IPROG* command used by SelectMAP to prepare the device for dynamic configuration without resetting the configuration logic. If the ICAP is enabled and set for writing, the configuration data are written to the memory one byte at a time. The *DONE* signal is used to indicate successful reconfiguration.

The functionality and description of each ICAP port is summarized in Table 3.1. By connecting these ports to the state machine as shown in Figure 3.5, a method is provided for controlling the reconfiguration process and for supplying the necessary commands to the configuration memory.

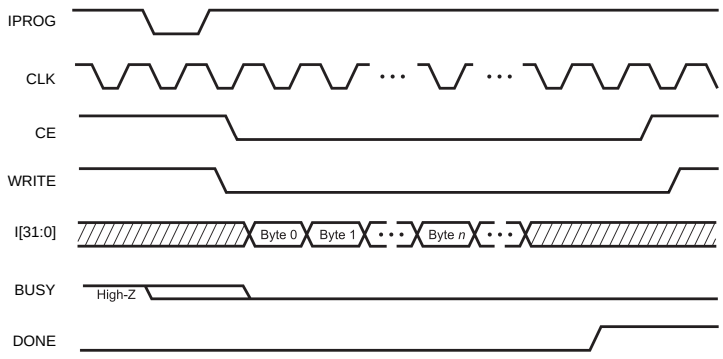


Figure 3.4: Timing diagram for loading configuration data into the ICAP [14]

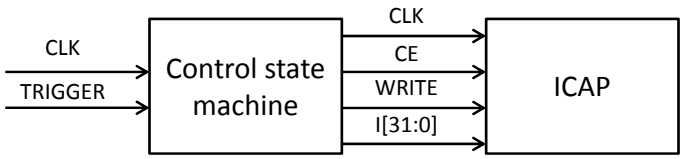


Figure 3.5: Block diagram depicting the interconnectivity of the control state machine and the ICAP

Table 3.1: ICAP pin description

Pin Name	**Type**	**Description**
CLK	Input	ICAP interface clock
CE	Input	Active-low ICAP interface select
WRITE	Input	Selects read or write operation
I[31:0]	Input	ICAP write data bus
O[31:0]	Output	ICAP read data bus
BUSY	Output	Active-high busy status

3.4.2 ICAP state machine

The state machine used is shown in Figure 3.6. The reconfiguration process is initiated from the user logic, which drives *CE* and *WRITE* low to enable write operations to the ICAP. The configuration process then commences by reading 32-bit words from the BRAM via the input port, *I*, and continues to do so until the *DESYNC* command string is detected. This indicates a complete configuration was transferred and releases the configuration logic. Alternatively, the ICAP output, *O*, can also be used to detect the *DESYNC* command string. If a value of *0x9F* is detected at the output, the *DESYNC* command was received and the device is desynchronised. A value of *0xDF*, on the other hand, indicates a synchronised device [50].

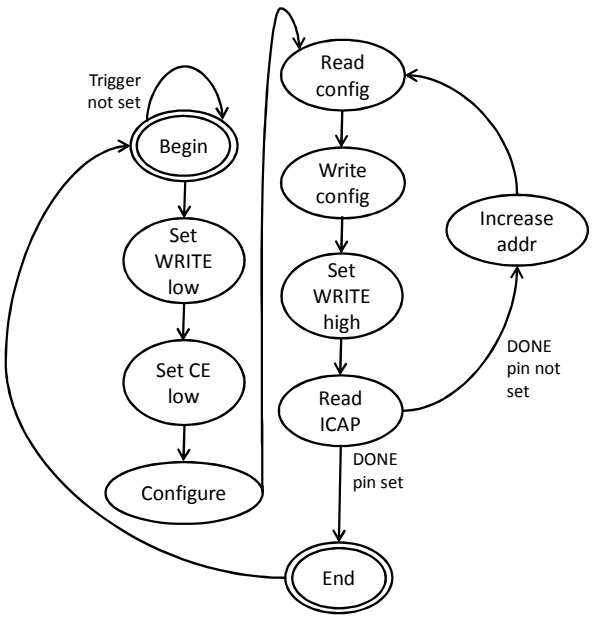


Figure 3.6: Flow diagram of the hardware controlled reconfiguration state machine

3.4.3 BRAM initialization

When initializing the BRAM using Xilinx®'s CORE Generator™, a *.coe*-file has to be used. This is a text-based file containing a header and initialization data for the BRAM. The FPGA configuration data, on the other hand, are formatted as hexadecimal values[1]—complicating the initialization process. A simpler approach is to convert the configuration data into ASCII using BitGen™.[2] The advantage obtained is two-fold. Firstly, the data are neatly organised into 32-bit words, each representing a configuration command—simplifying analysis. Secondly, the resulting file can easily be loaded into the BRAM using the VHDL construct shown in Listing 3.1.

Listing 3.1: VHDL construct to load bitstream in BRAM

```
type <romtype> is array(0 to <rom_width>) of bit_vector(<rom_addr_bits> downto 0);

impure function <rom_function_name> (<rom_file_name> : in string)
    return <romtype> is
    FILE <rom_file>          : text is in <rom_file_name>;
    variable <line_name>     : line;
    variable <rom_name>      : <romtype>;

 begin
    for I in <romtype>'range loop
        readline (rom_file>, <line_name>);
        read (<line_name>, <rom_name>(I));
    end loop;
    return <rom_name>;
 end function;

 signal <rom_name> : <romtype> := <rom_function_name>("<file_name>");
```

3.5 Experimental setup

To verify that the BRAM-based architecture can be used for the dynamic reconfiguration of real-time applications, a simple application was implemented on a Xilinx® ML507 development board wherein the pulse width modulated (PWM) duty cycle of a blinking LED was dynamically reconfigured using the architecture proposed in Section 3.1, the design flow given in Section 3.3, and the hardware controlled reconfiguration discussed in Section 3.4.

The duty cycle of the PWM was defined by three parameters—low, medium and high—stored in memory. A counter was then employed to count from the low value to the high, toggling an LED for each range. Using PlanAhead™, these memory locations were packaged in reconfigurable blocks capable of being swapped dynamically. Due to the limited size of the BRAM, only one configuration was stored and switching between the device's current configuration and the one stored was done using a push button. The reconfiguration process was handled by the state machine discussed in Section 3.4.2, which indicates its current state using different LEDs.

[1] A sectional view of the bitstream can be seen in Figure C.1 of Appendix C

[2] An illustration of the ASCII-formatted bitstream can be found in Figure C.2

The time-lapse between the button push and when the *DONE*-LED illuminates was measured using an oscilloscope to determine the reconfiguration time. To ensure the accuracy of the measurement, it was compared to the expected theoretical reconfiguration time, obtained by dividing the size of the configuration with the throughput of the ICAP. It was found that the oscilloscope provided an accurate measurement of the reconfiguration time. The complete architecture of the experimental setup is shown in Figure 3.7.

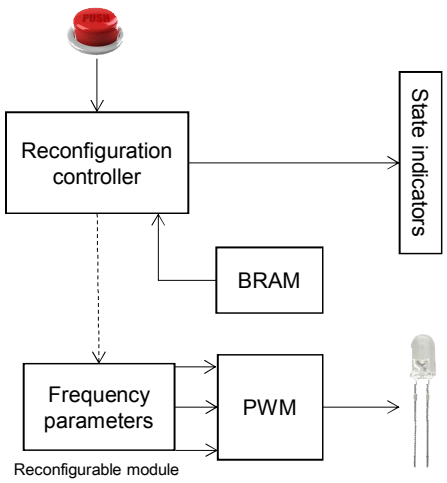


Figure 3.7: Block diagram of the experimental set-up used to evaluate the BRAM-based architecture

3.6 Throughput results

The ASCII-formatted module-based partial reconfiguration data required to reconfigure this application consists of 53,056 bits grouped into 1,658 32-bit words. Considering that the Virtex®-5 ICAP is capable of a maximum theoretical throughput of 400 MBps, it is possible to transfer the entire configuration through the ICAP within 16.58 µs.

In the literature it was shown that it is possible to overclock the ICAP above the Xilinx®-recommended 100 MHz [55, 60]. To investigate the maximum frequency at which the ICAP can be clocked in this application—without specifically designing for optimal clock frequency–the frequency of the ICAP was gradually increased. At double the recommended clock frequency, the reconfiguration time was measured to be 8.29 µs. This result is illustrated in Figure 3.8. Increasing the clock frequency even further to 300 MHz unfortunately caused the reconfiguration process to fail.

Further investigation revealed that even though the maximum clock frequency of BRAM (v6.2) is specified as 450 MHz [107], it is theoretically possible to clock the ICAP at a maximum frequency of 580 MHz if a set-up time of 0.49 ns is assumed for CLB flip-flops. This was confirmed by Hansen [60] who used custom hardware to run the ICAP above its maximum specified clock

frequency. It is thus evident that the 300 MHz limitation can be overcome by careful design, or by using custom hardware.

Using difference-based reconfiguration yields a significantly smaller configuration. For this particular application, the resulting configuration only contains 346 32-bit words. Since the reconfiguration time is directly related to the number of words in the configuration, a reconfiguration time of 1.730 µs can be measured using the experimental setup.

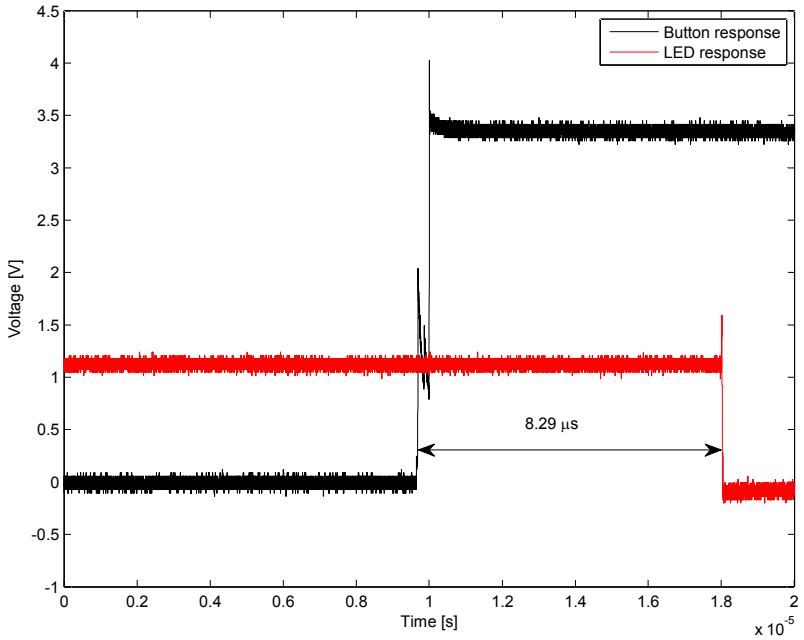


Figure 3.8: Reconfiguration response of the hardware controlled experimental set-up

For the purpose of this discussion, consider an application with a control cycle of 50 µs. If a real-time system can be defined as one which "controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time" [108], this implies that the reconfiguration process needs to fit within the remaining control cycle time after all other processing completes. Considering the worst reconfiguration time of the experimental set-up, 16.58 µs, this leaves 33.42 µs for all other processing. Should this be insufficient, the buffer in the control cycle can be increased by overclocking the ICAP—reducing the reconfiguration time even further. If only small changes need to be made to the design, difference-based reconfiguration can be used to reduce the configuration size and obtain the best possible reconfiguration time.

3.7 Concluding remarks

Hardware controlled reconfiguration is a technique utilising hardware to improve the throughput of the reconfiguration process. It uses the BRAM-based architecture proposed in the literature to saturate the throughput of the ICAP during the reconfiguration process. This architecture

was identified as the optimal architecture for reconfiguring real-time applications. Unfortunately, the BRAM is extremely limited in size, and only a subset of configurations can be stored.

This led to the hypothesis that a specialisation technique, such as PBS proposed by Bruneel [12], can be used to adapt the configuration stored in the BRAM for any set of hardware. Before this can be done, the viability of using the BRAM-based architecture for real-time reconfiguration had to be determined. This was done by implementing a simple application and reconfiguring it using the Xilinx® partial reconfiguration design flow. As was expected, the time this process takes to complete is directly related to the number of words in the configuration. Deviating from the proposed design flow and using the difference-based reconfiguration to generate the partial configurations, reduces the number of words in the configuration—and the reconfiguration time—significantly.

From the results presented in this chapter, it is evident that the BRAM-based architecture is suitable for reconfiguring real-time applications and verifies the information presented in the literature. Despite the simplicity of the application used, the methodology presented in this chapter is also used for designing and implementing the reconfiguration architecture presented later in this thesis. This architecture combines the BRAM-based hardware controlled reconfiguration presented in this chapter, with a bitstream specialiser, capable of adapting the configuration stored in the BRAM to represent a different set of hardware. Since this specialiser targets the bits in the bitstream directly, it is first necessary to parse and analyse the bitstream to ensure safe manipulation of FPGA resources at bit-level. This is addressed in the next chapter.

CHAPTER 4

BITSTREAM PARSING AND ANALYSIS

"Things don't have to change the world to be important."
— Steve Jobs

To overcome the size limitation of the BRAM-based architectures, and to allow fast specialisation of the bitstream, it is proposed to only store a single configuration in the BRAM and then to adapt it using direct bitstream manipulation. Unfortunately, the proprietary nature of the bitstream makes it impossible to safely modify the bits at bit-level. Despite various research on the subject, the most promising is an unpublished technical report discussing a method to parse and analyse a Xilinx® Virtex®-5 VLX110T FPGA. A shortcoming of this report is that different LUT constructs were not considered in the technical report. This chapter makes the first contribution by providing new insight into the composition of a Xilinx® FPGA configuration by verifying the methodology proposed in the technical report and expanding upon it by also considering the encoding of different LUT constructs. Also discussed in this chapter is an alternative method, called the Nibble Location Method *to determine the bitstream encoding. Both methods were applied to a Xilinx® Virtex®-5 bitstream, but as will be shown, it is easy to apply these parsing and analysis methods to any FPGA bitstream.*

4.1 Virtex®-5 device architecture

As their name suggests, FPGAs consist of various logic blocks connected in an array using switch matrices. Xilinx® refers to these logic blocks as CLBs, or configurable logic blocks, and the interconnectivity thereof is shown in Figure 4.1. As can be seen in Figure 4.2, each CLB consists of two slices, each each of which consists[1] of:

- four logic generators, i.e. lookup tables,

[1] Refer to Appendix B for more detailed diagrams regarding the slice composition

- four storage elements,
- multiplexers,
- carry logic.

Two types of slices exist in most Xilinx® FPGAs, SLICEL and SLICEM—even though newer devices could also include a SLICEX[2] [109]. SLICEL is tailored to implement logic, whereas SLICEM also contains memory-orientated logic. A SLICEX is similar to a SLICEL, but does not include carry logic. By connecting the elements inside these slices (and consequently inside the CLBs), any combinational and sequential circuit can be implemented. In fact, any computation can be represented as a Boolean expression, which in turn can be represented by a truth table. From this observation, it is thus evident that truth tables are the heart of any FPGA. A hardware element capable of implementing truth tables are LUTs. From an implementation viewpoint, a LUT is simply an N:1 multiplexer. Flip-flops are used in these slices to allow a state to be stored.

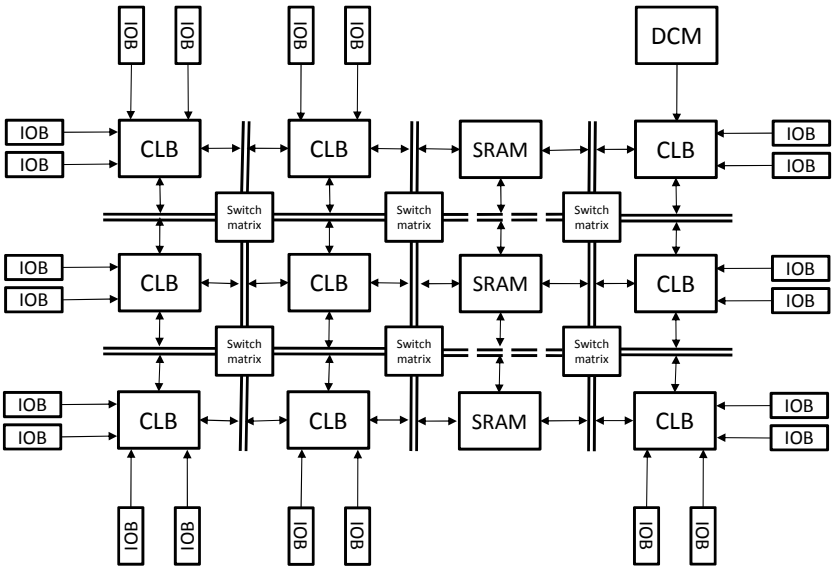


Figure 4.1: Block diagram showing interconnectivity on a Xilinx® FPGA

Each configurable element in an FPGA requires 1-bit of storage to maintain a configuration. This configuration can be seen as a flat binary file, called a bitstream, whose contents maps to the various elements on the FPGA. The FPGA is then configured by loading the bitstream into the configuration memory through a special configuration interface.

The configuration memory of Xilinx® FPGAs is usually arranged in frames that are tiled about the device, as shown in Figure 4.3. A frame is defined as the smallest addressable segment of the configuration memory. Each read and write to the configuration memory must therefore act upon a frame. For the Virtex®-5, the frame spans a vertical stack of 1,312 bits, which correlates to the height of a row. A typical row on a Virtex®-5 includes 20 CLBs, 40 input-

[2]The logic diagrams of these slices can be found in Appendix B

output blocks (IOBs) or 4 BRAMs. The mapping of the configuration words in the bitstream to the configuration bits in a frame is shown in Figure 4.4.

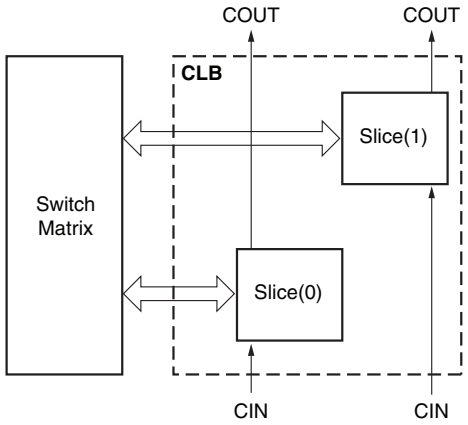


Figure 4.2: Block diagram depicting the composition of a Virtex®-5 CLB

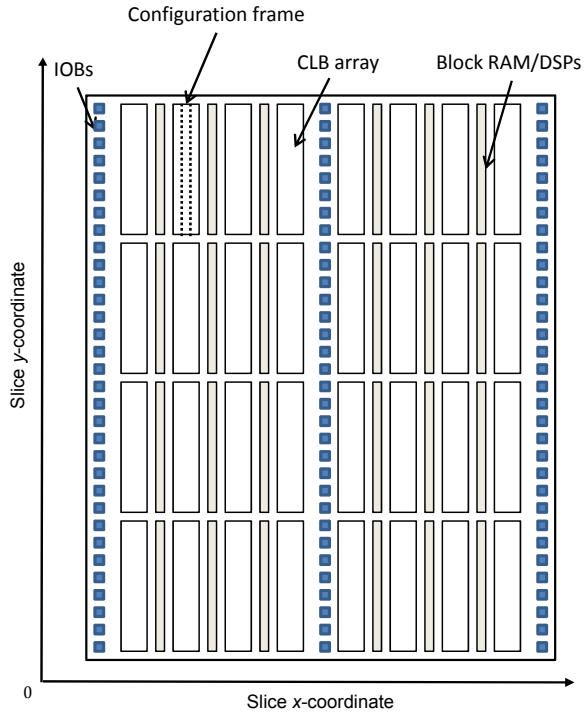


Figure 4.3: An illustration of the Virtex®-5 configuration architecture and slice coordinates

4.2 Virtex®-5 frame addressing

The physical position of each frame on the device is given by a unique 32-bit address. As shown in Figure 4.5, this address is typically divided into five parts. *Block type* refers to the type of block being configured. *Row address* is divided into two parts: a row address and a bit to

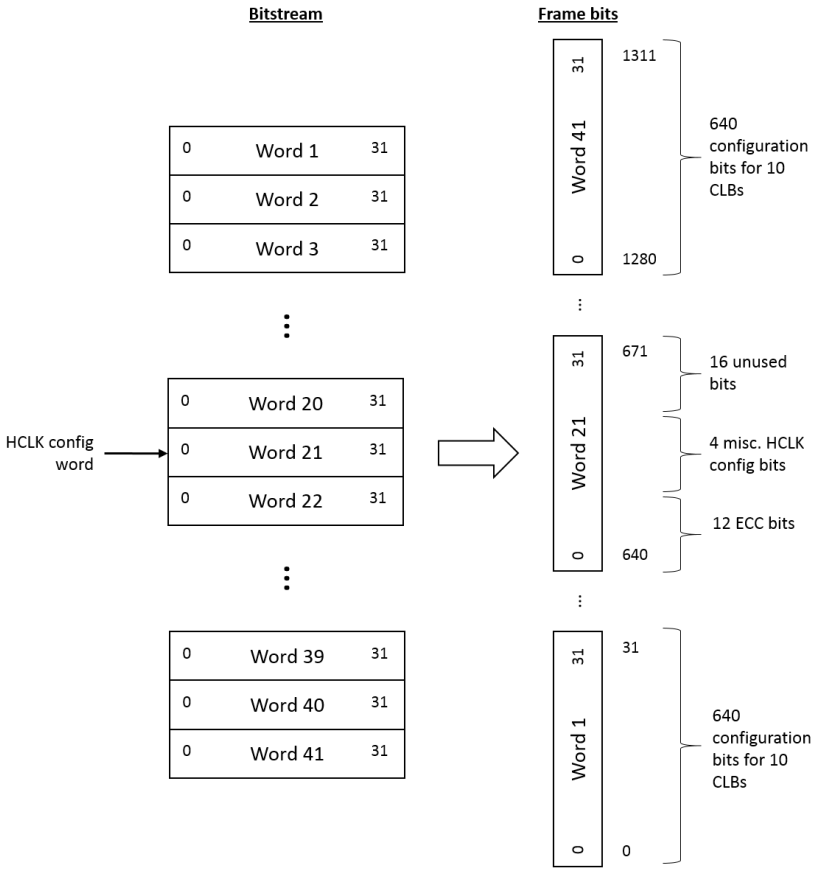


Figure 4.4: Mapping of configuration words in the bitstream to a frame

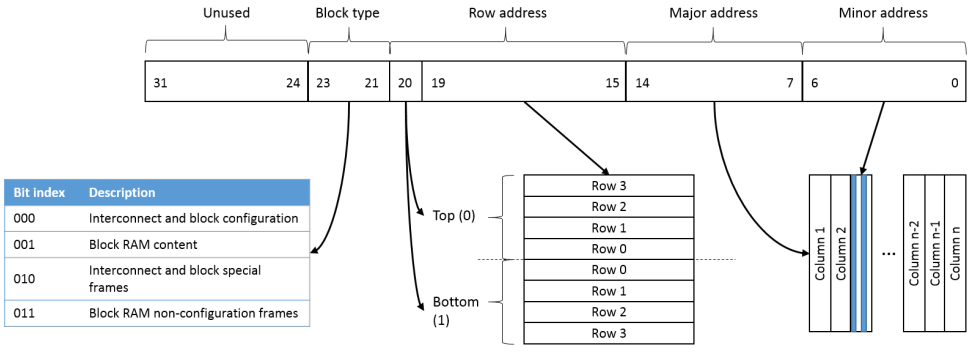


Figure 4.5: The composition of the Frame Address Register (FAR)

indicate whether the row is located in the top or bottom half of the device. Each row is subdivided into columns, indicated by a *major address* (also referred to as the column address), with a column corresponding to a block in the FPGA array. Each column contains a certain number of frames, as indicated by *minor address*. The number of frames contained in each type of column is listed in Table 4.1.

Writing the configuration data to the configuration memory is done using two types of packets: Type 1 and Type 2. A Type 1 packet is used for register reads and writes, whereas a Type 2 packet is used to write long blocks of data. The format of each packet is shown in Tables 4.2 and

Table 4.1: The number of frames per column for a Virtex®-5 FPGA

Block type	Number of frames
CLB	36
DSP	28
Block RAM	30
IOB	54
Clock column	4

Table 4.2: Type 1 packet header format

Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRxxxxx	RR	xxxxxxxxxxx

4.3 respectively. *Header Type* acts as an indication of the packet type. The *Opcode* in a Type 1 packet indicates whether a read or write is performed to the register located at the *Register Address*. Lastly, *Word Count* shows the number of words read or written. Even though a large variety of registers exists, for the purpose of this thesis, only three are of interest for parsing and analysing the contents of the bitstream:

Frame Address Register (FAR) Specifies the address of the frame

Frame Data Register, Input Register (FDRI) Writes to this register configure frame data at the frame address specified in the FAR register.

Legacy Output Register (LOUT) Software uses this register to drive data to the DOUT[3] pin during serial daisy-chain configuration. It is also used during debugging of the configuration.

4.3 Bitstream analysis methodology

To analyse the bitstream, a simple design was created in Xilinx®'s FPGA Editor™. By making small changes to this design and comparing it to the base design, it is possible to determine the location of specific FPGA elements in the bitstream. The base design, shown in Figure 4.6 with the implemented routes in cyan, was in most cases a slice at the origin of the FPGA floorplan, *SLICE_X0Y0*. Even though slight variations of this base design were used, depending on the analysis done, it mostly adheres to the following properties:

Table 4.3: Type 2 packet header

Header Type	Opcode	Word Count
[31:29]	[28:27]	[26:0]
010	RR	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx

[3]Serial data output for downstream daisy-chained devices.

Table 4.4: Type 1 packet opcode format

Opcode	Function
00	NOP
01	Read
10	Write
11	Reserved

1. To determine the composition of a frame, a SLICEL was selected as this is the most basic slice type in most FPGAs. If the device being analysed includes SLICEX slice types, these will rather be selected. However, to analyse more complex constructs, such as RAM and ROM, SLICEM has to be used in the base design, since these are the only slice types capable of implementing these constructs.

2. To keep the design simple, no circuitry was added outside the slice. However, this causes a design rule check (DRC) error and successful bitstream generation is required for comparison. The *-d* switch is thus sent to BitGen™ to ignore any DRC errors when generating the bitstream [110].

3. The slice had to use all the inputs of a LUT to allow for any logic combination.

4. All four LUTs were initialized using the Boolean equation $(A1 + A2 + A3 + A4 + A5 + A6) * 0$, which outputs '0' for all possible input-combinations ($A1$ to $A6$). This provides an effective baseline so that any design with a change in the LUT will immediately be visible in the bitstream when compared to the base design.

5. To keep the number of bit-changes between designs relevant to LUT contents only, the CRC was disabled. This implies the option *-g CRC:disabled* was sent to BitGen™ when generating the bitstream.

Figure 4.7 shows a diagram of the methodology followed. The base design can either just be modified and compared, or moved to a different slice location before modification. Comparing the stationary slice determines the position of the various elements in the particular frame, whereas migrating the design to different slices on the FPGA floorplan highlights how the frames as compounded to form the bitstream.

Migrating and modifying the base design was done using an FPGA Editor™ script. An example of the script is shown in Listing 4.1. In this instance, the script opens the base design, selects the slice entitled *TEST_LUT* and moves it to *SLICE_X1Y0*. After re-routing, the new design is saved with the LUTs reinitialized to produce '0' to avoid any ambiguousness. The LUTs are then initialized to output '1' for all possible inputs, whereafter the design is saved for comparison.

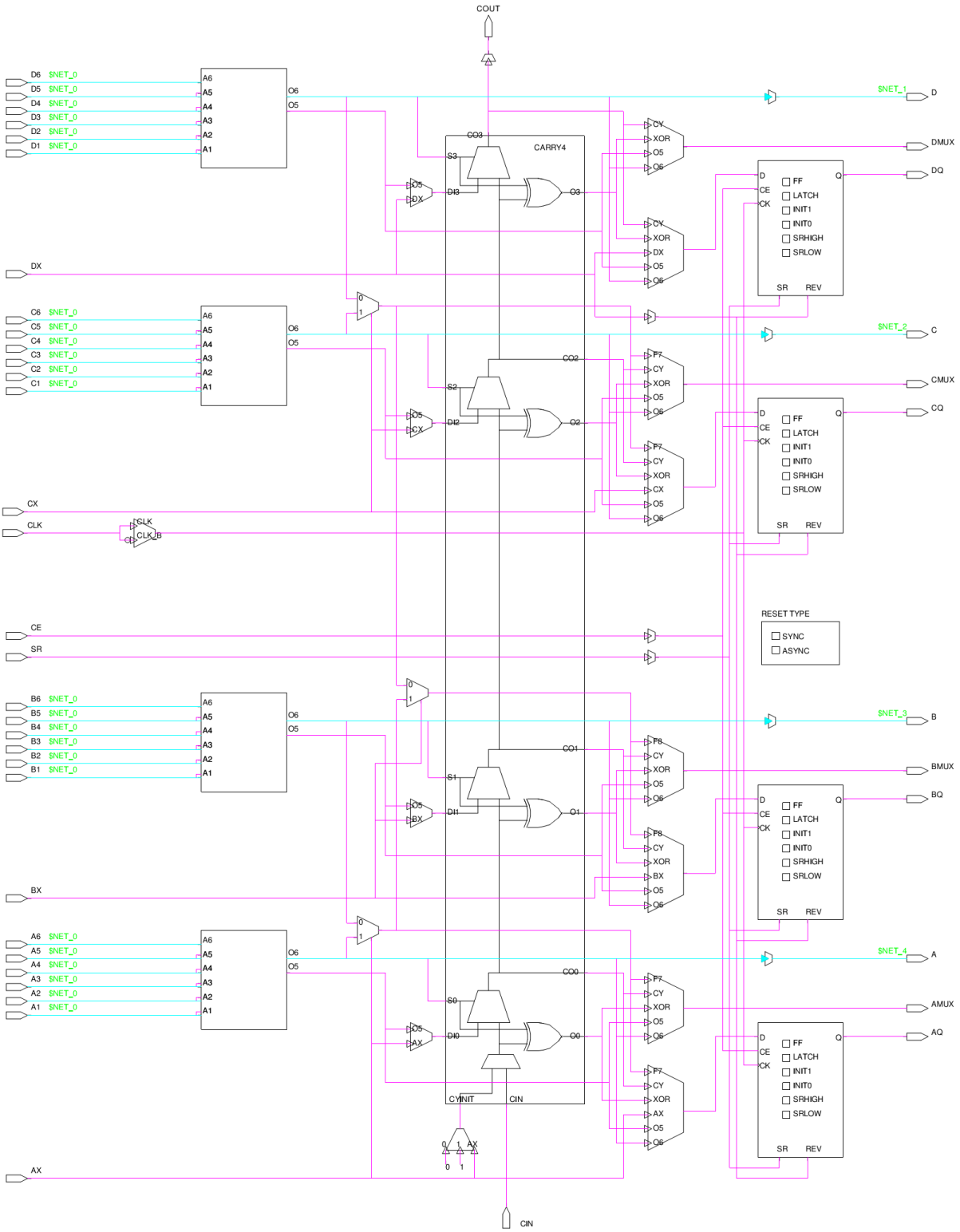


Figure 4.6: Logic diagram of the base design used for analysing FPGA bitstreams

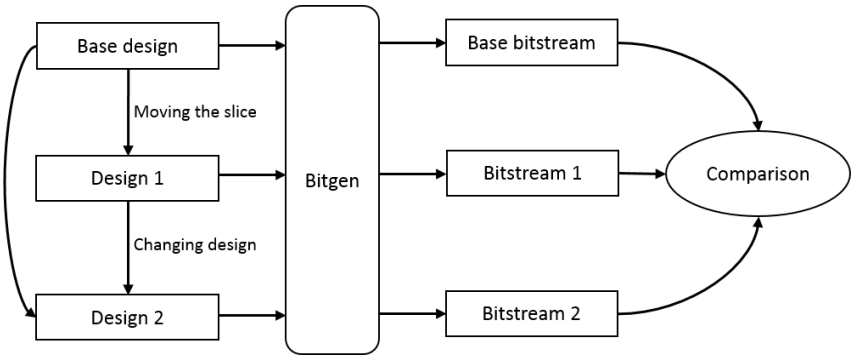


Figure 4.7: Flow diagram of the methodology followed to analyse the bitstream

Listing 4.1: An example of the FPGA Editor™ script used to modify the base design

```
open design ``base.ncd" -nomd
setattr main edit-mode Read-Write
select -k comp 'TEST_LUT'
select -k site 'SLICE_X1Y0'
swap
autoroute -all
setattr comp TEST_LUT A6LUT ``(A2+A3+A4+A5+A6+A1)*0"
block apply
save -w design ``x1y0-LUTA-0.ncd"
setattr comp TEST_LUT A6LUT "(A2+A3+A4+A5+A6+A1)*0+1"
block apply
save -w design ``x1y0-LUTA-1.ncd"
clear
exit
```

To simplify the comparison of the bitstream, it was converted into an ASCII format[4] by supplying BitGen™ with the *-b* switch. This increases the file size, trading processing time for readability. Comparing the bitstreams was done using a MATLAB® program to compare each line of the ASCII-formatted bitstreams, marking the difference between both bitstreams with an 'X'. The remaining matching lines were converted into their equivalent hexadecimal values for simplicity and each Type 1 packet is decoded into its equivalent command. The result is shown in Figure 4.8.

When configuring the FPGA, the FAR is automatically incremented after each frame is written. This makes it difficult to extract specific frames. By giving the option *-g DebugBitstream:Yes* to BitGen™ when creating the bitstream, generates a bit-file that assists in debugging [110]. The key feature of a debug bitstream, relevant to frame extraction, is that each frame is written independently, whereafter the current frame address is written to LOUT. By intercepting the LOUT value after detecting a difference between the two bitstreams, indicates the exact frames where the difference between the designs were detected.

[4] A sectional view of the ASCII converted bitstream, with typical commands used during configuration, can be found in Appendix C

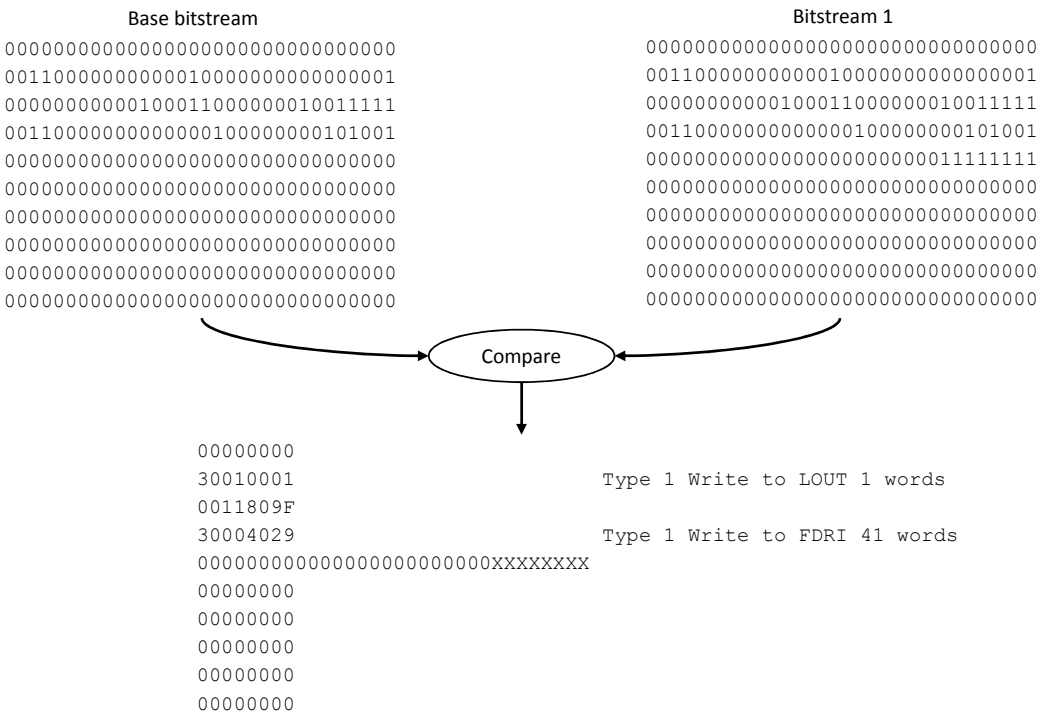


Figure 4.8: Graphical illustration of the comparison between bitstreams of different designs

4.4 Experimental designs

The LUTs in an FPGA are used to implement a variety of elements. Typical elements implemented using LUTs are:

- any 5 or 6-input Boolean expression,
- 6-input multiplexer,
- simple storage elements, such as ROM,
- complex storage elements, such as RAM.

Different LUT constructs are available for implementing the above-mentioned elements. The focus of this thesis is on reconfiguring these constructs by means of direct bitstream manipulation, and ten experiments, listed in Table 4.5, were selected for the experiments performed in this chapter. The logistics of each experiment will now be discussed.

4.4.1 Experiment 1: Frame composition

The first issue to address was the translation from logical coordinates of a slice to frame address. This was done by comparing the base design to a modified one which is identical except for one

Table 4.5: List of experimental designs and relevant slices

Experiment	Configuration investigated	Slice type
1	General frame composition	SLICEL/SLICEM
2	Configuration for a 6-input SLICEL multiplexer	SLICEL
3	Configuration for a 6-input SLICEM multiplexer	SLICEM
4	Storing a hexadecimal value in a ROM	SLICEL
5	Storing 64 1-bit values in RAM64X1	SLICEM
6	Storing 16 8-bit values in RAM16X8	SLICEM
7	Storing 16 4-bit values in RAM16X4	SLICEM
8	Storing 32 8-bit values in RAM32X8S	SLICEM
9	Storing a 16-bit value in a shift register LUT (SRL16)	SLICEM
10	Storing a 32-bit value in a shift register LUT (SRL32)	SLICEM

LUT that has been initialized to have an output of '1' for all input combinations. All four LUTs (*A6LUT* to *D6LUT*) of *SLICE_X0Y0* were considered by individually initialising them with the Boolean expression $(A1 + A2 + A3 + A4 + A5 + A6) * 0 + 1$ and comparing them to the base design.

This design was then migrated horizontally across the FPGA floorplan[5] and the above mentioned process repeated for *A6LUT* for all slices of row $y = 0$. This highlights the differences between the composition of SLICEL and SLICEM frames. To confirm that there are no difference in the frame composition for a column, the design was then migrated across the FPGA floorplan where $x = 0$. Since the slice-columns are homogeneous, this was only done for one row (i.e 40 slices) and every 15^{th} y-coordinate thereafter.

Lastly, a single comparison was done between the base design and a design where all four LUTs were simultaneously initialized to produce an output of '1' for all possible inputs. This forces the maximum number of differences between the two designs—i.e. 256 bits changed.

4.4.2 Experiment 2 and 3: Multiplexer configuration

Experiments 2 and 3 were done to determine the encoding of the LUTs in the bitstream when they are initialized as multiplexers, and compared to the base design. Since two types of slices are available for a Virtex®-5 FPGA, this experiment was first done for a SLICEL, whereafter it was repeated for a SLICEM. In general, a valid Boolean expression for a multiplexer with an output of An, irrespective of input, is $(A1 \vee A2 \vee A3 \vee A4 \vee A5 \vee A6) \wedge 0 \vee An$.

Castellone found that the encoding of some of the LUTs are byte swapped and attributed this to a difference in encoding between the SLICEM and SLICEL. To verify this, and to determine whether this is also the case for the Virtex®-5 VFX70T, the design was again migrated to different horizontal slices, repeating the process.

[5]The floorplan of the FPGA used throughout these experiments are shown in Figure B.4, which acts as a visual representation of the slices and their equivalent coordinates.

4.4.3 Experiment 4: ROM storage

The fourth experiment was done to determine the bitstream encoding of LUTs when they are used for simple storage, such as a ROM configuration. A LUT can store any 64-bit number, taking into consideration that this configuration allows this value to be read a single bit at a time via its one bit output.

Since a LUT is capable of storing 2^{64} different values, it is impossible to test the bitstream encoding for every possible value stored. For the purpose of this experiment, it was found sufficient to only perform two types of experiments. The first determines the effect of changing the least significant nibble (LSN) of the value stored and determining the effect this has on the bitstream generated. The second modifies the stored value in nibble intervals starting from the LSN to the most significant nibble (MSN). Both types of experiments are illustrated in Figures 4.9 and 4.10 respectively. The value represented is a 64-bit value stored in hexadecimal format. Also indicated in the figure are the 16 nibbles used to represent this number. Looking at the LSN, the first value compared to the base design is *0x1* (or *0b0001*). This value is then gradually incremented and compared to the base design until *0xF* (or *0x1111*) is reached. The stored value was then modified starting with the LSN and moving towards the MSN, each time changing the nibble to *0x1* and comparing this to the base design.

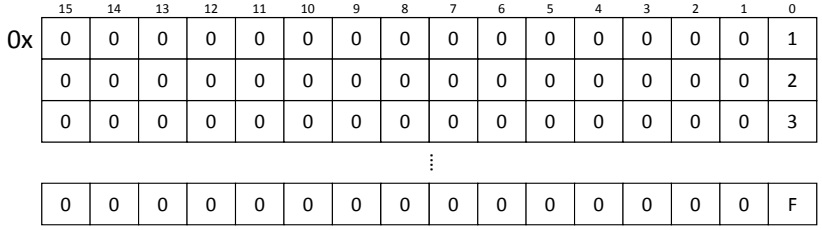


Figure 4.9: Gradual incrementation of the value stored in ROM's LSN

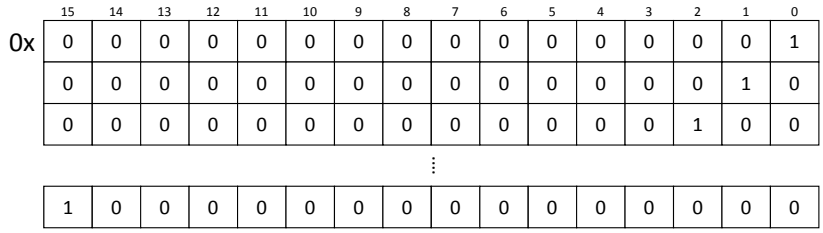


Figure 4.10: Changing the value stored in ROM from LSN to MSN

4.4.4 Experiment 5: RAM64X1 storage elements

The next step was to determine whether a more complicated storage, such as RAM, would yield the same bitstream encoding as the simple ROM. Various different RAM configurations are available, but since the ROM is only capable of producing a single bit of the stored value at a time, *RAM64X1* was selected as the first RAM-construct experiment. This construct uses a

SLICEM to store 64 single bits and is a good comparison to the ROM discussed in the previous section. The only difference is that a RAM construct allows for reading and writing. For the purpose of this thesis, it is assumed that instead of writing a new value to the memory space, it is rather reconfigured. The write operation of the RAM was thus not used, and the construct was initialized with a value to be stored. The analysis of the construct was performed in a manner similar to the ROM discussed in the previous section.

4.4.5 Experiment 6: RAM16X8 storage elements

Both previous experiments analysed configurations where the LUTs are configured to store a value. In these configurations the width of the memory space is only 1-bit. To store larger word widths, multiple LUTs have to cascaded to form a complex construct. Storing an 8-bit value, for example, requires a minimum of four LUTs. This is made possible by the LUT's independent outputs, *O5* and *O6*, that are used when implementing 5-input Boolean expressions. In this configuration, the LUT's 6^{th} input, *A6*, is driven high by Xilinx®'s ISE® Design Suite, or Vivado®. Only SLICEMs can be used, since they are capable of initializing their LUTs into complex memory structures[6] that are joined in such a way to allow different word lengths to be stored. Due to the complexity of these structures, and since it is not ensured that these complex memory structures adhere to the encoding schemes of the previous experiments, it was deemed necessary to analyse them. For the purpose of this experiment, *RAM16X8* was selected, which is capable of storing 16 8-bit values.

Analysing this construct is a bit more challenging than the ROM and simple RAM constructs in the previous two experiments. The reason being that this construct requires eight initialisation parameters, whereas the previous two experiments only had one each. These parameters initialise the LUTs to perform certain functions which is translated into a bitstream configuration. While it is not necessary to know the value stored by the LUT before determining the bitstream encoding, its assists in understanding the effect a high-level change has on the physical hardware and the configuration bits it is mapped to. Conversely, it is necessary to understand the effects bit manipulation has on the hardware, and to translate this to the high-level. For this reason, the mapping of the initialisation parameter was first investigated before focussing on the bitstream encoding. This was done by analysing the initialisation parameters and routing for numerous designs, and comparing the values stored in the LUTs. A trend was observed, which will be discussed later.

Investigating the bitstream encoding was done by first initializing each 6-input LUT in the construct with an initialisation parameter of *0b1111111111111111*. This yields a value of *0x5555555555555555* to be stored in the LUTs. This distinct pattern makes it easy to locate the LUT bits in the bitstream when the design is compared to the base application. This process was then repeated for each 5-input LUT by storing a value of *0x55555555* in each LUT. Next, the same analysis was performed as in the previous experiments where the LSN was gradually increased until *0xF* is reached, and the stored value changed from LSN to MSN for each of the eight initialisation parameters. This was done for both 5- and 6-input LUTs.

[6]Some documentation refer to these as "complexes", but to avoid confusion, this thesis will simply refer to them as constructs.

4.4.6 Experiment 7: RAM16X4 storage elements

The next experiment selects another variation of LUT RAM construct, this time storing 16 4-bit values. Analysing this construct was done in a manner similar to the RAM16X8. It was found that because 4-bit values are stored, each LUT only uses output *O6* and as a result, only the 6-input LUTs are utilized when concatenated to represent a RAM element.

Analysing this memory element was done using three designs. The first starts by configuring the *A6LUT* of a RAM16X4 with a value of *0x0000000000000001*. This value was then gradually incremented until *0x000000000000000F* was reached. The second design repeats this process, this time starting with a value of *0x0000000000000010* and ending with *0x00000000000000F0*. The last design starts with a value of *0x0000000000000001* and shifts the LSN to the left towards the MSN until *0x1000000000000000* is reached. This verifies the previous two results for each nibble position in the initialization parameter.

4.4.7 Experiment 8: RAM32X8 storage elements

The last memory construct analysed is capable of storing 32 8-bit values. As was the case with RAM16X8, both 5 and 6 input LUTs are used for this construct, resulting in both outputs being used. The only difference being that RAM32X8 uses all five remaining inputs (except for *A6* that is driven by ISE® for the 5-input LUTs), compared to the four inputs used by RAM16X8.

Six designs were used to determine the bitstream encoding of the LUT contents. This was done in sets of two, for both the 5 and 6-input LUTs. The first set starts with an initialization of *0x00000001* and increases this value until *0x0000000F* is reached. In a similar fashion, the second design starts with *0x00000010* and ends with *0x000000F0*. The last set shifts the initialization parameter to the left from *0x00000001* to *0x10000000*.

4.4.8 Experiments 9 and 10: Shift register LUT (SRL) configuration

A SLICEM function generator can be configured as a 32-bit shift register without the use of a flip-flop. This allows each LUT to delay serial data with up to 32 clock cycles. By cascading these LUTs, larger function generators can be created. Using all four LUTs in a slice can produce delays of up to 128 clock cycles. In this configuration, the LUTs are known as shift register LUTs, or SRLs.

To analyse the encoding of the SRL the initialization parameter is again used. This value sets the initial value of the output (Q) after configuration. As a result, this value has to be stored in the bitstream. It was found that the high-level initialization parameters directly represent the values to be stored in the LUT. The initialization of a 16-bit shift register LUT (SRL16) is represented by a 16-bit binary value. An initialization value of *0x0001*, for example, will yield a LUT value of *0x0000000000000001*. Analysing this encoding was again done by incrementing this initialization parameter until *0x000000000000000F* is reached. This process was then repeated

for *0000000000000010* to *00000000000000F0*. To verify the encoding for all four nibbles in the initialization parameter, it was shifted to the left from *0x0001* to *0x1000*.

The above mentioned process was then repeated for the 32-bit shift register LUT (SRL32), which has a 32-bit initialization parameter.

4.5 From VHDL to the bitstream

Before discussing the results of the bitstream analysis, it is first necessary to discuss how the values being stored in the LUTs can be derived from the high-level initialisation parameter.

Every LUT in a slice construct requires an initialisation parameter, specified using high-level VHDL, to specify the value to be stored. Even though inference could have been used to implement the constructs, instantiation was rather used to control the exact placement of the constructs. For the experiments with constructs containing a single LUT, only one parameter is required, whereas the more complicated RAM constructs required eight, named *INIT_00* to *INIT_07*. This is due to both LUT outputs being used and an initialisation parameter being required to describe the output response of each. The Xilinx® nomenclature refers to these as $LUT6$ to drive *O6* and $LUT5$ to drive *O5*, and both occupy the same basic element logic (BEL)[7] as their parent LUT. For example, the $ALUT$ is compounded by $A5LUT$ and $A6LUT$, and each of these sub-elements has its own initialisation parameter. The same applies to the other LUTs in the slice.

To determine the value stored in the LUT, it is first necessary to compile the truth table representing the specific LUT. These truth tables map each input combination to a bit in the corresponding initialisation parameter. The mapping is dependent on the hardware, and most specifically, the routing to the construct. If the construct is used to store 64 values, all six address lines are used. For 32 values, only five address lines are used and for 16 values, only four. In the latter two cases, one or more input ports are tied low, since they are not used to decode the address. A tool, such as FPGA Editor™, can be utilised to determine the address lines being used, as well as the order in which they are used. Using this information, the truth table can be compiled by inserting the initialisation parameter into the output of the table at the positions reflected by the hardware.

As an example, the high-level VHDL code used to instantiate a complex LUT construct is shown in Listing 4.2. In this instance, *RAM16X8S*[8] is instantiated, and because only 16 8-bit values are stored, one address line is tied low. Using FPGA Editor™ to open the routed LUT construct[9], this address line was found to be *A1*. This implies that only every second line of the truth table is used where $A1 = 0$ to insert the initialisation parameter.

[7] Used with Xilinx® FPGAs to describe the functional elements that make up a component

[8] The "S" refers to this being a static synchronous RAM

[9] An example is shown in Figure B.5

Listing 4.2: VHDL construct used to instantiate RAM16X8S

```
RAM16X8S_inst : RAM16X8S
    generic map
    (
            INIT_00 => "0000000000001010", -- INIT for bit 0 of RAM
            INIT_01 => "0000000000000000", -- INIT for bit 1 of RAM
            INIT_02 => "0000000000000000", -- INIT for bit 2 of RAM
            INIT_03 => "0000000000000000", -- INIT for bit 3 of RAM
            INIT_04 => "0000000000000000", -- INIT for bit 4 of RAM
            INIT_05 => "0000000000000000", -- INIT for bit 5 of RAM
            INIT_06 => "0000000000000000", -- INIT for bit 6 of RAM
            INIT_07 => "0000000000000000"  -- INIT for bit 7 of RAM
    )
    port map
    (
            O   => Output,                 -- 8-bit RAM data output
            A0  => Address0,               -- RAM address[0] input
            A1  => Address1,               -- RAM address[1] input
            A2  => Address2,               -- RAM address[2] input
            A3  => Address3,               -- RAM address[3] input
            D   => (others => '0'),        -- 8-bit RAM data input
            WCLK => '0',                   -- Write clock input
            WE   => '0'                    -- Write enable input
    );
```

In this instance, initialisation parameter *INIT_00* is used to initialise a LUT with *0b1010*. Since *A1* is tied down, this value is inserted in every second output position, as shown in Figure 4.11, with the entries used highlighted in yellow. If a *LUT5* is being considered, *A6* is driven high and only the bottom half of the truth table needs to be populated. For a *LUT6*, the entire table is populated, but since the configuration bits of *LUT5* and *LUT6* are a subset of each other, the lower 32 bits of the *LUT6*'s initialisation string need to be the same as for the *LUT5* BEL— since they are sharing the hardware involved. The rest of the output is padded with '0's to complete the table. For the initialisation parameter of *0b1010*, this equates to a value of *0x0000004400000044* to be stored in a *LUT6* and *0x00000044* for *LUT5*. Using this method, it is possible to determine the values stored in the LUTs for any construct.

A secondary issue for the complex constructs is determining which of the eight *INIT*-parameters are mapped to which LUT. Even though this mapping is also routing dependent, the relationship shown in Figure 4.12 was most commonly found in the thousands of experiments performed during this thesis. In some odd cases, *LUT5* and *LUT6* would swap, but *INIT_00* and *INIT_01* would always refer to *DLUT*, *INIT_02* and *INIT_03* to *CLUT*, and so forth.

As already mentioned, the entries in the truth table are highly dependent on the routing to the LUT construct. Even though it is relatively consistent between different implementations of the same construct, this can not be guaranteed. It is also not possible to lock the pins in place for RAM-based LUT constructs[10], in order to force specific routing to the LUT. Consequently, the inputs of the LUTs were manually placed using FPGA Editor™ to ensure consistent results during the bitstream parsing and analysis. This will be discussed in the subsequent sections.

[10]Currently, *LOCK_PINS* is only supported on ROM

A6	A5	A4	A3	A2	A1	Output
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	1
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	1
0	0	0	1	1	1	0
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	0	0
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	0
0	1	0	0	0	1	0
0	1	0	0	1	0	0
0	1	0	0	1	1	0
0	1	0	1	0	0	0
0	1	0	1	0	1	0
0	1	0	1	1	0	0
0	1	0	1	1	1	0
0	1	1	0	0	0	0
0	1	1	0	0	1	0
0	1	1	0	1	0	0
0	1	1	0	1	1	0
0	1	1	1	0	0	0
0	1	1	1	0	1	0
0	1	1	1	1	0	0
0	1	1	1	1	1	0
1	0	0	0	0	0	0
1	0	0	0	0	1	0
1	0	0	0	1	0	1
1	0	0	0	1	1	0
1	0	0	1	0	0	0
1	0	0	1	0	1	0
1	0	0	1	1	0	1
1	0	0	1	1	1	0
1	0	1	0	0	0	0
1	0	1	0	0	1	0
1	0	1	0	1	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	0
1	0	1	1	0	1	0
1	0	1	1	1	0	0
1	0	1	1	1	1	0
1	1	0	0	0	0	0
1	1	0	0	0	1	0
1	1	0	0	1	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	0	1	0	1	0
1	1	0	1	1	0	0
1	1	0	1	1	1	0
1	1	1	0	0	0	0
1	1	1	0	0	1	0
1	1	1	0	1	0	0
1	1	1	0	1	1	0
1	1	1	1	0	0	0
1	1	1	1	0	1	0
1	1	1	1	1	0	0
1	1	1	1	1	1	0

Figure 4.11: 64-bit truth table representing the configuration of a 6 input LUT

4.6 Bitstream parsing and analysis results

This section discusses the results of the ten experiments to parse and analyse the bitstream. This is done by first discussing the results of *Experiments 1* to *3*. Despite being performed individually, the results of the three experiments are better explained together.

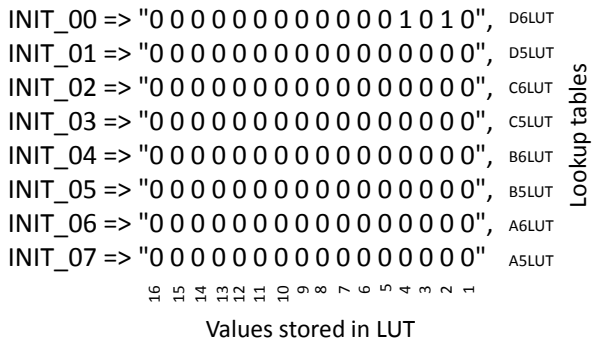


Figure 4.12: Mapping the initialisation parameters to the value stored per LUT

4.6.1 Experiment 1, 2 and 3: Boolean logic and frame composition

Figure 4.13 shows the differences between the base bitstream and the design with the LUTs initialized to produce '1' for all combinations of the inputs. The x and y-axes of the figure respectively refer to the line in the bitstream and position in the 32-bit word where the differences were detected, as was shown in Figure 4.8. Each marker type represents a different LUT. These results are also tabulated in Table 4.6[11]. This table shows the decoded $LOUT$-value and the block-type, row, major and minor addresses are listed (refer back to Figure 4.5 for more information). Also shown, are the specific lines from the bitstreams where the differences were detected. In this instance, bitstream 1 refers to the base design, whereas bitstream 2 is the design with the LUTs initialized to produce '1' for all combinations of inputs.

As can be seen, the configuration of the LUTs is evenly distributed throughout the frames. In fact, each frame contains part of the LUT configuration for all 20 CLBs constituting a row height. This is shown in Figure 4.14 and corresponds to the result obtained by Castellone [17]. However, Castellone stated that the central word (bits 640 to 671) is unused in the Virtex®-5, which is erroneous. According to the Virtex®-5 FPGA configuration user guide [111] the first 12 bits (bits 640 to 651) are used to store the error checking code (ECC) and bits 652 to 655 are miscellaneous horizontal clock (HCLK) configuration bits. The only unused bits are 656 to 671, which are identified as "unused HCLK configuration bits".

From these experiments it is possible to derive a set of mathematical formulae to translate the slice coordinates to the frame address. These are strictly device dependent, but the same process can be followed for any Xilinx® FPGA device. For the Virtex®-5 VFX70T FPGA, $x = \{0, 1, ..., 75\}$, $y = \{0, 1, ..., 159\}$ and the LUTs are given by $L = \{A, B, C, D\}$. Referring back to Figure 4.5, $Type = 0$ for all these formulae since only block configurations were written in these experiments. The top or bottom position, Pos, is given by (4.1), with $\mathbb{1}$ the indicator function defined by (4.2).

$$Top(y) = \mathbb{1}_{\{y \geq 32\}}(y) \tag{4.1}$$

[11] All other results discussed in this section are available in similar tables. However, not all are given. Refer to the data disc or the appendix for more information.

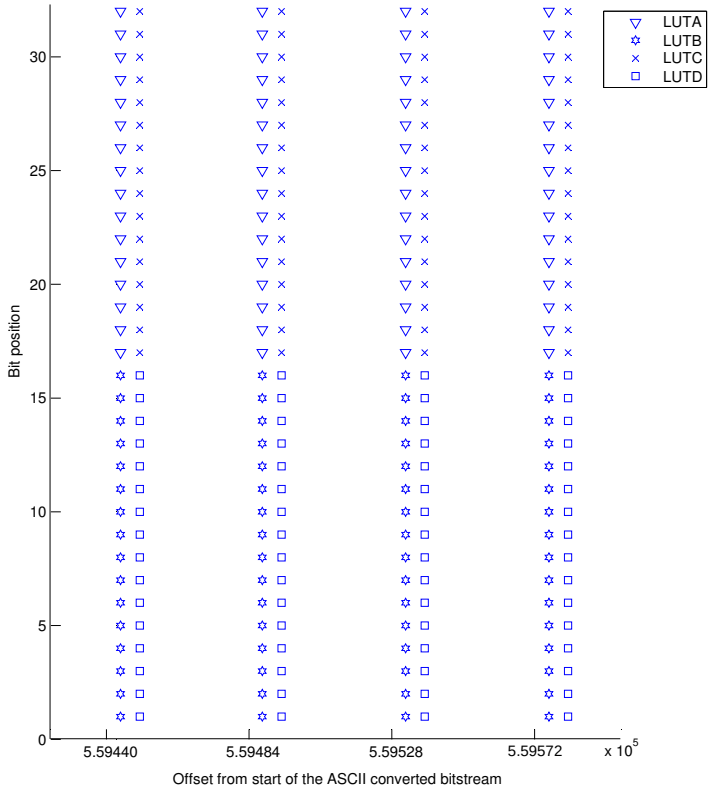


Figure 4.13: Configuration differences between the base design and one with all LUTs initialized to produce '1' for all inputs

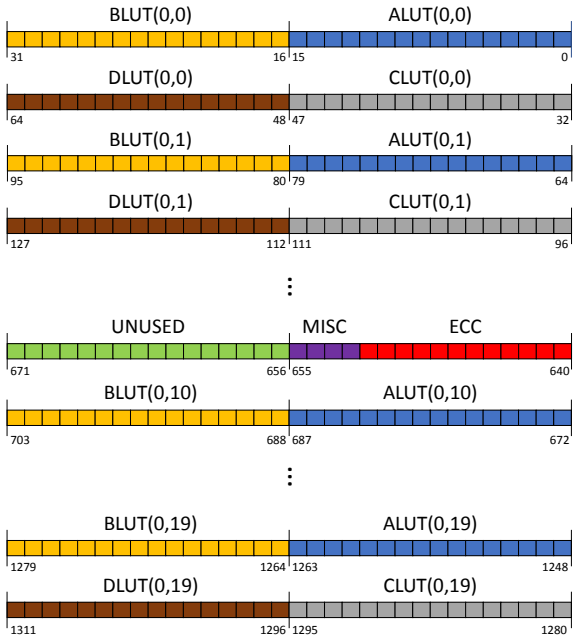


Figure 4.14: Frame composition showing the position of all 80 LUTs in a row

Table 4.6: Differences between the base design and one with all LUTs initialized to produce '1' for all inputs (Experiment 1)

	LOUT (Hex)	**LUT value (binary value of the base bitstream)**	**LUT value (binary value of bitstream 2)**	**Type**	**Pos**	**Row**	**Major addr**	**Minor addr**
Results ALUT	1180A0	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	32
	1180A1	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	33
	1180A2	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	34
	1180A3	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	35
Results BLUT	1180A0	00000000000000000000000000000000	11111111111111110000000000000000	0	1	3	1	32
	1180A1	00000000000000000000000000000000	11111111111111110000000000000000	0	1	3	1	33
	1180A2	00000000000000000000000000000000	11111111111111110000000000000000	0	1	3	1	34
	1180A3	00000000000000000000000000000000	11111111111111110000000000000000	0	1	3	1	35
Results CLUT	1180A0	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	32
	1180A1	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	33
	1180A2	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	34
	1180A3	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	35
Results DLUT	1180A0	00000000000000000000000000000000	11111111111111110000000000000000	0	1	3	1	32
	1180A1	00000000000000000000000000000000	11111111111111110000000000000000	0	1	3	1	33
	1180A2	00000000000000000000000000000000	11111111111111110000000000000000	0	1	3	1	34
	1180A3	00000000000000000000000000000000	11111111111111110000000000000000	0	1	3	1	35

$$\mathbb{1}_A(x) = \begin{cases} 1 & if\, x \in A \\ 0 & if\, x \notin A \end{cases} \tag{4.2}$$

The row is calculated using (4.3). Since the total height of the Virtex®-5 VFX70T FPGA is 159 slices and row numbering is repeated for the top and bottom of the device, y has to be multiplied by 2 and 159 subtracted from it. The '40' below the line refers to number of slices per row this device has.

$$Row(y) = \left\lfloor \left| \frac{2y - 159}{40} \right| \right\rfloor \tag{4.3}$$

The major address calculation (4.4) was derived from the data obtained while moving the slice horizontally across the FPGA floorplan. To keep the differences between the slices at a maximum, the comparison bitstream initialised all the LUTs to be '1'. Table 4.7 shows an excerpt of the data obtained. The increase in offset added to each value is because of various other columns, such as RAM, DSP and I/O, that are situated between the CLB columns.

Castellone found that for every fourth column the configuration is byte-swapped ('1234' → '3421') and attributed this to the encoding used for SLICEM. Experiment 1 to 3 also confirmed this for the Virtex®-5 VFX70T. However, Castellone found that columns 30, 96 and 97 break this convention and also swap. Experiment 1 to 3 showed that this is not the case for the VFX70T, and instead columns 50, 54 and 58 are swapped. Combining this knowledge with the results of the LUT encoding (an excerpt is shown in Table 4.8), the minor address can be decoded using the Boolean expression in (4.6) and the algebraic expression of (4.7).

$$Major(x) = \begin{cases} \lfloor \frac{x}{2} \rfloor + 1, & 0 \le x \le 7 \\ \lfloor \frac{x}{2} \rfloor + 2, & 8 \le x \le 19 \\ \lfloor \frac{x}{2} \rfloor + 3 & 20 \le x \le 31 \\ \lfloor \frac{x}{2} \rfloor + 4, & 32 \le x \le 39 \\ \lfloor \frac{x}{2} \rfloor + 6, & 40 \le x \le 47 \\ \lfloor \frac{x}{2} \rfloor + 7, & 48 \le x \le 51 \\ \lfloor \frac{x}{2} \rfloor + 8, & 52 \le x \le 55 \\ \lfloor \frac{x}{2} \rfloor + 9, & 56 \le x \le 59 \\ \lfloor \frac{x}{2} \rfloor + 10, & 60 \le x \le 67 \\ \lfloor \frac{x}{2} \rfloor + 11, & 68 \le x \le 75 \end{cases} \tag{4.4}$$

$$\begin{aligned} Major(x) &= \left\lfloor \frac{x}{2} \right\rfloor + 1 + \mathbb{1}_{\{x \ge 8\}}(x) + \mathbb{1}_{\{x \ge 20\}}(x) + \mathbb{1}_{\{x \ge 32\}}(x) + \mathbb{1}_{\{x \ge 40\}}(x) \\ &+ \mathbb{1}_{\{x \ge 48\}}(x) + \mathbb{1}_{\{x \ge 52\}}(x) + \mathbb{1}_{\{x \ge 56\}}(x) + \mathbb{1}_{\{x \ge 60\}}(x) + \mathbb{1}_{\{x \ge 68\}}(x) \end{aligned} \tag{4.5}$$

Table 4.7: Excerpt of the experimental results when comparing bitstreams while moving the slice horizontally (Experiment 1)

Slice Pos	LOUT (Hex)	LUT value (binary value of bitstream 1)	LUT value (binary value of bitstream 2)	Type	Pos	Row	Major addr	Minor addr
X0Y0	11809A	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	32
	11809B	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	33
	11809C	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	34
	11809D	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	35
X1Y0	11809A	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	26
	11809B	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	27
	11809C	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	28
	11809D	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	1	29
X2Y0	11809A	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	2	32
	11809B	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	2	33
	11809C	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	2	34
	11809D	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	2	35
X3Y0	11809A	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	2	26
	11809B	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	2	27
	11809C	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	2	28
	11809D	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	2	29
X4Y0	11809A	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	3	32
	11809B	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	3	33
	11809C	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	3	34
	11809D	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	3	35
X5Y0	11809A	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	3	26
	11809B	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	3	27
	11809C	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	3	28
	11809D	00000000000000000000000000000000	00000000000000001111111111111111	0	1	3	3	29

Table 4.8: Determining the LUT configuration strings by moving the multiplexer-configured slice horizontally (Experiment 2 and 3)

Slice Pos	LOUT (Hex)	LUT value (binary value of bitstream 1)	LUT value (binary value of bitstream 2)	Type	Pos	Row	Major addr	Minor addr	LUT config
X0Y0	11809A	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	1	32	AAAA
	11809B	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	1	33	5555
	11809C	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	1	34	AAAA
	11809D	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	1	35	5555
X1Y0	11809A	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	1	26	5555
	11809B	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	1	27	AAAA
	11809C	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	1	28	5555
	11809D	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	1	29	AAAA
X2Y0	11809A	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	2	32	5555
	11809B	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	2	33	AAAA
	11809C	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	2	34	5555
	11809D	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	2	35	AAAA
X3Y0	11809A	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	2	26	5555
	11809B	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	2	27	AAAA
	11809C	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	2	28	5555
	11809D	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	2	29	AAAA
X4Y0	11809A	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	3	32	AAAA
	11809B	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	3	33	5555
	11809C	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	3	34	AAAA
	11809D	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	3	35	5555
X5Y0	11809A	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	3	26	5555
	11809B	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	3	27	AAAA
	11809C	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	3	28	5555
	11809D	00000000000000000000000000000000	00000000000000001010101010101010	0	1	3	3	29	AAAA

Table 4.9: LUT multiplexer configuration strings

Input	Combinational			Complex	
	SLICEL LUT6	**SLICEM LUT6**	**SLICEL LUT5**	**SLICEM LUT6**	**SLICEL LUT5**
A1	5555AAAAAAAA5555	AAAA5555AAAA5555	5500AA00AA005500	00AA005500AA0055	AA005500AA005500
A2	FFFFFFFF00000000	00000000FFFFFFFF	FF00FF0000000000	0000000000FF00FF	00000000FF00FF00
A3	5555555555555555	5555555555555555	5500550055005500	0055005500550055	5500550055005500
A4	3333333333333333	3333333333333333	3300330033003300	0033003300330033	3300330033003300
A5	0F0F0F0F0F0F0F0F	0F0F0F0F0F0F0F0F	0F000F000F000F00	000F000F000F000F	0F000F000F000F00
A6	00FF00FF00FF00FF	00FF00FF00FF00FF	0000000000000000	0000000000000000	0000000000000000

$$Swap(x) \equiv (x \, mod \, 4 = 0 \vee x = 50 \vee x = 54 \vee x = 58) \tag{4.6}$$

$$Minor(x) = \begin{cases} \begin{cases} [32, 33, 34, 35] & (x \, even) \\ [26, 27, 28, 29] & (x \, odd) \end{cases} & if \, \overline{Swap}(x) \\ \begin{cases} [34, 35, 33, 32] & (x \, even) \\ [28, 29, 27, 26] & (x \, odd) \end{cases} & if \, Swap(x) \end{cases} \tag{4.7}$$

When configuring the LUTs as multiplexers, it was found that each input has a unique 64-bit string used to initialize the LUT. Using these strings, any Boolean LUT configuration can be derived by simply replacing each input variable in the Boolean expression with the corresponding string. Say for instance a LUT has to be configured to perform the following simple calculation: $A6LUT = \overline{A1} \vee \overline{A2} \vee \overline{A3} \vee \overline{A4} \vee \overline{A5} \vee \overline{A6}$. By replacing the inputs with the corresponding values listed in Table 4.9, it is possible to show that the configuration is *0xFFFEFFFFFFFFFFFF* and *0xFFFFFFFFFFFFFFFE* for a SLICEL and SLICEM respectively. Similarly, this can also be shown to be true for more complex Boolean arithmetic. Castellone used the example of $A6LUT = ((\overline{A2} \wedge (\overline{A1} \wedge (A3 \oplus (A6 \oplus (A4 \oplus A5))))) \vee (A2 \wedge (A1 \vee (A3 \oplus (A6 \oplus (A4 \oplus A5))))))$ to show that the LUT configuration for a SLICEM is *0x41142882EBBE7DD7*. However, the configuration for a SLICEL was not shown. This was calculated as *0x7DD7EBBE41142882*.

4.6.2 Experiments 4 and 5: Single bit output storage

Each LUT can be modelled as a multiplexer with the inputs serving as the select lines and initialization parameter acting as the data being multiplexed. As an example, a LUT modelled as a 16:1 multiplexer is shown in Figure 4.15. This implies that the multiplexer strings discussed in the previous section can again be used to determine the bitstream encoding for RAM constructed LUTs.

Take a simple initialization parameter such as *0x00000000000000B0*, which initialises the truth table of the LUTs as shown in Figure 4.16. Also shown in the figure is the unsimplified Boolean function represented by the truth table. Using the configuration strings in Table 4.9, it can be shown that the LUT configuration is *0x4000400000004000* for LUTs located in a SLICEM and

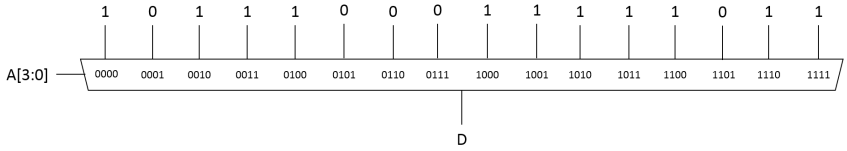


Figure 4.15: Graphical representation of a LUT modelled as a 16:1 multiplexer

A6	A5	A4	A3	A2	A1	Output	
0	0	0	0	0	0	**0**	
0	0	0	0	0	1	**0**	
0	0	0	0	1	0	**0**	
0	0	0	0	1	1	**0**	
0	0	0	1	0	0	**1**	$\overline{A6} \wedge \overline{A5} \wedge \overline{A4} \wedge A3 \wedge \overline{A2} \wedge \overline{A1}$
0	0	0	1	0	1	**1**	$\overline{A6} \wedge \overline{A5} \wedge \overline{A4} \wedge A3 \wedge \overline{A2} \wedge A1$
0	0	0	1	1	0	**0**	
0	0	0	1	1	1	**1**	$\overline{A6} \wedge \overline{A5} \wedge \overline{A4} \wedge A3 \wedge A2 \wedge A1$
0	0	1	0	0	0	**0**	
0	0	1	0	0	1	**0**	

Boolean function: $(\overline{A6} \wedge \overline{A5} \wedge \overline{A4} \wedge A3 \wedge \overline{A2} \wedge \overline{A1}) \vee (\overline{A6} \wedge \overline{A5} \wedge \overline{A4} \wedge A3 \wedge \overline{A2} \wedge A1) \vee (\overline{A6} \wedge \overline{A5} \wedge \overline{A4} \wedge A3 \wedge A2 \wedge A1)$

Figure 4.16: Truth table segment showing the initialization parameter and equivalent Boolean expression

0x4000000040004000 for their SLICEL counterparts. In a similar manner it can also be shown that an initialization parameter of *0xFFF0000123400008* is encoded into a LUT configuration of *0x0207038784070007* for SLICEM and *0x0007840702070387* for SLICEL.

As shown in Figure 4.14, the 64-bit configuration parameter is distributed evenly throughout the bitstream in sets of 16-bits. A configuration of *0x0207038784070007*, for example, is thus distributed in the following frames:

Frame 32: 00000000000000000000001000000111
Frame 33: 00000000000000000000001110000111
Frame 34: 00000000000000001000010000000111
Frame 35: 00000000000000000000000000000111

for ALUT. It was found that SLICEM always uses minor addresses 32 to 35, whereas SLICEL uses 26 to 29.

Another method was found that allows a designer to derive the configuration parameter without using the multiplexer strings in Table 4.9. For the purpose of this thesis, this method is dubbed the *Nibble Location Method*, or *NLM*. This method uses the position and value of each nibble in the high-level initialization parameter to derive the configuration. When considering each nibble, it was found that each position in the 16 nibbles that make up the initialization parameter indicates a unique frame composition as shown in (4.8). As shown in Table 4.10, the value used to encode the configuration when using the *Nibble Location Method*, is simply a binary value, with a one at the nibble position and then flipped around.

Table 4.10: Encoding used for the Nibble Location Method

Nibble Position	**Encoding value**
0x0000000000000000	0b0000000000000000
0x0000000000000001	0b1000000000000000
0x0000000000000010	0b0100000000000000
0x0000000000000100	0b0010000000000000
0x0000000000001000	0b0001000000000000
0x0000000000010000	0b0000100000000000
...etc.	

Consider the initialization parameter *0x00000000000000B0* used in the previous example. As established in Figure 4.10, the nibble in question is at position '1', and the value used for encoding the configuration is *0b0100000000000000*. Since the nibble position is odd, the value of the nibble, *0b1011*, indicates that *0b0100000000000000* is located in minor frames 32, 33 and 35 (for a SLICEM) as illustrated in Figure 4.17. The remaining minor address is populated with *0b0000000000000000*. The configuration value is then obtained by concatenating the encoding values in ascending frame order, to obtain a 64-bit configuration. In this example, the encoding values are concatenated as: $0x0100000000000000 \| 0x0100000000000000 \| 0x0000000000000000 \| 0x0100000000000000$, yielding a result of *0x4000400000004000*. Figure 4.18 shows a similar example with an initialisation parameter of *0x0000000000000010*. Since the nibble is now located at an odd position, the encoding values are now located at minor addresses 35, 34, 33 and 32, as shown in Table 4.11. Also shown in the table are the results obtained when increasing the nibble value to *0x0000000000000060*.

$$Configuration\ composition_{(Single\ output\ bit)} = \begin{cases} \begin{cases} [35,34,33,32] & (x\ even) \\ [26,27,29,28] & (x\ odd) \end{cases} & odd\ nibble\ positions \\ \begin{cases} [34,35,32,33] & (x\ even) \\ [27,26,28,29] & (x\ odd) \end{cases} & even\ nibble\ positions \end{cases} \tag{4.8}$$

Using this method, it is also possible to determine the configuration for storing more complex values such as *0xFFF0000123400008* used in an earlier example. To determine the configuration value for this initialization, one must start with the least significant nibble (LSN), in this case *0x8* or *0b1000*. Since this is the first position, a decode value of *0b1000000000000000* has to be used. The least significant nibble is in position 0 and the frame composition for even nibble positions has to be used. This results in a configuration of:

Frame 32: 00000000000000000000000000000000
Frame 33: 00000000000000000000000000000000
Frame 34: 00000000000000001000000000000000
Frame 35: 00000000000000000000000000000000

or, *0x0000800000000000*. This process should then be repeated for all nibbles until the most

Table 4.11: Excerpt of the experimental results when comparing single bit storage constructs while incrementing the *INIT*-value

INIT Val	Nibble val	LOUT (Hex)	LUT value (binary value of bitstream 1)	LUT value (binary value of bitstream 2)	Pos	Type	Row	Major addr	Minor addr
0x0000000000000010	1	1180A0	00000000000000000000000000000000	00000000000000000100000000000000	1	0	3	1	32
	0	1180A1	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	33
	0	1180A2	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	34
	0	1180A3	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	35
0x0000000000000020	0	1180A0	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	32
	1	1180A1	00000000000000000000000000000000	00000000000000000100000000000000	1	0	3	1	33
	0	1180A2	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	34
	0	1180A3	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	35
0x0000000000000030	1	1180A0	00000000000000000000000000000000	00000000000000000100000000000000	1	0	3	1	32
	1	1180A1	00000000000000000000000000000000	00000000000000000100000000000000	1	0	3	1	33
	0	1180A2	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	34
	0	1180A3	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	35
0x0000000000000040	0	1180A0	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	32
	0	1180A1	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	33
	1	1180A2	00000000000000000000000000000000	00000000000000000100000000000000	1	0	3	1	34
	0	1180A3	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	35
0x0000000000000050	1	1180A0	00000000000000000000000000000000	'00000000000000000100000000000000	1	0	3	1	32
	0	1180A1	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	33
	1	1180A2	00000000000000000000000000000000	00000000000000000100000000000000	1	0	3	1	34
	0	1180A3	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	35
0x0000000000000060	0	1180A0	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	32
	1	1180A1	00000000000000000000000000000000	00000000000000000100000000000000	1	0	3	1	33
	1	1180A2	00000000000000000000000000000000	00000000000000000100000000000000	1	0	3	1	34
	0	1180A3	00000000000000000000000000000000	00000000000000000000000000000000	1	0	3	1	35

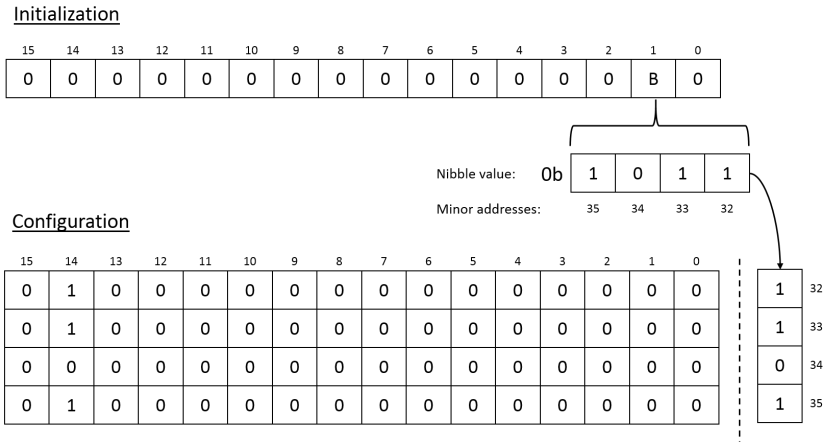


Figure 4.17: Applying *NLM* to a ROM-based LUT construct with an initialisation of $0x00000000000000B0$

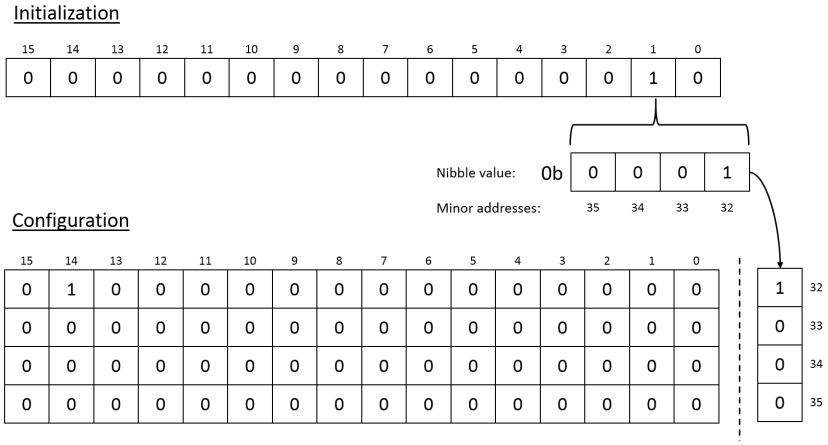


Figure 4.18: Applying *NLM* to a ROM-based LUT construct with an initialisation of $0x0000000000000010$

significant nibble (MSN) is reached. Lastly, the logical disjunction (*OR*) between the 16 values are obtained as the final configuration for the stored value. In this instance, this was again found to be *0x0207038784070007*.

4.6.3 Experiments 6 to 8: Complex storage elements

Despite concatenating multiple LUTs in a slice to create complex LUT constructs, it was found that the methods discussed in the previous section still apply to derive the bitstream encoding scheme. However, since these elements can only be constructed using LUTs from SLICEMs, only the multiplexer strings for SLICEMs are used.

As a first example, consider experiment 6's RAM16X8. This LUT construct uses eight 16-bit initialization parameters, *INIT_00* to *INIT_07*, to initialize the four LUTs. The configuration for each LUT can again be derived by using the configuration strings of Table 4.9. Take an initialization parameter of *0x0800* for example. By inserting this value into the truth table shown

in Figure 4.11, it is possible to derive the Boolean expression as: $(\overline{A1} \wedge A2 \wedge A3 \wedge \overline{A4} \wedge A5 \wedge \overline{A6})$ for the 5-input LUTs and $(\overline{A1} \wedge A2 \wedge A3 \wedge \overline{A4} \wedge A5 \wedge A6)$ for the 6-input LUTs. Note that *A6* is driven high in the case of the 5-input LUTs and that *A1* is tied down. Replacing each variable with its corresponding configuration string, it can be shown that the configuration is *0x0000000004000000* and *0x0000000000040000* for the 5-input and 6-input LUTs respectively. This process can also be repeated for any other initialization parameter.

As a second example, consider the *RAM16X4* used in experiment 7. Four 16-bit initialization parameters are used to initialize the four LUTs. The difference between this design and that of *RAM16X8* is that the former only uses one output per LUT and as a result, the 5-input LUTs are unused. For this specific configuration, both *A1* and *A2* were manually tied down and the corresponding truth table used to derive the Boolean expression from the initialization parameter. Consider an initialization parameter of *0x80A0*. The Boolean expression for this parameter can be derived as $(\overline{A1} \wedge \overline{A2} \wedge A3 \wedge \overline{A4} \wedge A5 \wedge \overline{A6}) \vee (\overline{A1} \wedge \overline{A2} \wedge A3 \wedge A4 \wedge A5 \wedge \overline{A6}) \vee (\overline{A1} \wedge \overline{A2} \wedge A3 \wedge A4 \wedge A5 \wedge A6)$. Using the multiplexer strings, it is shown that the configuration is *0x0501000000000000*.

The last memory construct experimented with was *RAM32X8*. This construct is architecturally similar to *RAM16X8*, but with double the number of bits required per initialization parameter. Since both 5 and 6-input LUTs are required, the result is that the entire 64-bit truth table is used. An initialization parameter such as *0x0000000C* thus results in a value of *0x0000000C0000000C* to be stored in a 6-input LUT and *0x0000000C* in a 5-input LUT. Again, the Boolean expression can be derived from the truth table. In this instance this is found to be $(\overline{A1} \wedge A2 \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5} \wedge \overline{A6}) \vee (A1 \wedge A2 \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5} \wedge \overline{A6})$ for the 6-input LUTs and $(\overline{A1} \wedge A2 \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5} \wedge A6) \vee (A1 \wedge A2 \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5} \wedge A6)$ for the 5-input LUTs. This results in a configuration of *0x0000000000800080* for the former and *0x0000000080008000* for the latter.

The *Nibble Location Method* can also be used to derive the encoding for complex memory constructs. As an example, this method was used to derive the encoding for *RAM16X8*. It was found that this construct is quite similar to that of the storage elements with a single bit output, but with two fundamental differences. The ROM and *RAM64X1* both have an encoding value per each nibble position, while the *RAM16X8*'s encoding value changes every second nibble position. Another difference is the sequence in which the frames at minor addresses 32, 33, 34 and 35 are concatenated. As shown in (4.8), the composition of the simpler constructs differ depending on the nibble position. However, it was found that complex constructs use a fixed composition as shown in (4.9).

$$Configuration\,composition_{(RAM16X8)} = [34, 32, 35, 33] \tag{4.9}$$

As an example, consider the most basic initialization parameter of *0x0001*. As shown in Figure 4.19, the value of the nibble (*0b0001*) still indicates the minor address(es) containing the configuration values. However, this time each 16-bit encoding value is divided into two bytes for the 5 and 6-input LUTs respectively—as also indicated by the yellow blocks. If configuring a 6-input LUT, the encoding value is *0b10000000*, whereas the encoding value for 5-input LUTs is *0b1000000000000000*. Interestingly, it was found that these values are used for both $INIT = 0x0001$ and $INIT = 0x0002$. In fact, it was found that the base

configuration value (*0x00000001* is shifted left or right[12] for every odd bit-position in the binary initialization, i.e. *0b0000000000000100*, *0x0000000000010000*, *0x0000000001000000* etc). Encoding the configuration is thus rather done using the 16-bit binary initialization, also shown in the figure, since each bit set in this value corresponds to a specific configuration value. The nibble value is only used to determine the minor frame(s) containing the encoding value(s).

Figure 4.20 shows the encoding of $INIT = 0x0008$. As can be seen, the value of the nibble is *0b1000*, which indicates that the encoding value is situated in the frame with minor address 34. Since the binary value of the initialization parameter is *0b0000000000001000*, the default encoding value of *0b10000000* has to be shifted right once for the *ALUT* resulting in an encoding value of *0b01000000* for 6-input LUTs and *0b0100000000000000* for 5-input LUTs. Concatenating the values according to ascending minor addresses, results in a configuration of *0x0000000000400000* for the *A6LUT* and *0x0000000040000000* for *A5LUT*.

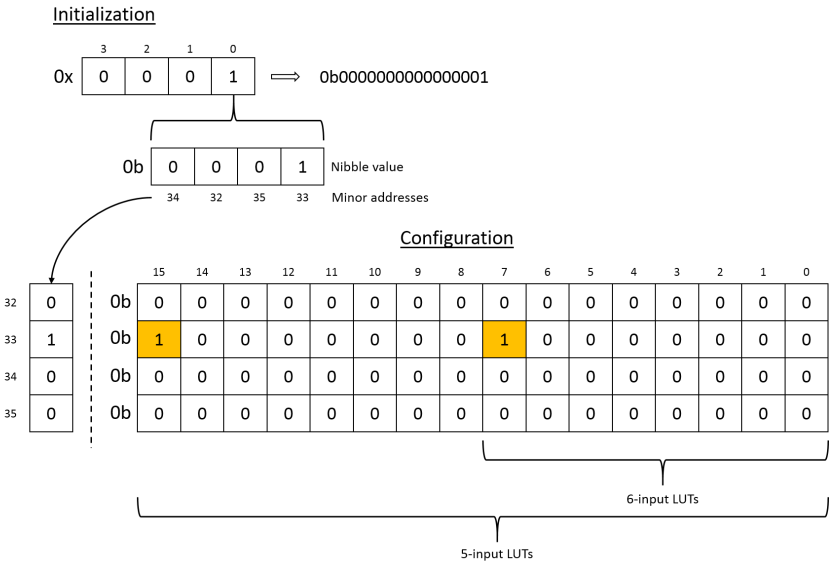


Figure 4.19: Applying *NLM* to a RAM16X8 construct with an initialisation of $0x0001$

Using this encoding scheme, it is possible to derive the configuration for any *RAM16X8* initialization. A value such as $INIT = 0x000A$ is encoded in Figure 4.21. Since the binary representation of this value is *0b1010*, the frames containing the configuration data are 34 and 35. Looking at the binary value of the initialization parameter, it is seen that the bits are set in position 1 and 3. As mentioned previously, the bits set at position 0 and 1 utilise a configuration value of *0b10000000*. The other bit set in the initialization parameter is at position 3. Because the bit-position is odd, the configuration value has to be shifted to the right, resulting in a value of *0b01000000*. The bit set at position 1 indicates that the configuration of *0b10000000* is located in minor frame position 35, whereas the other configuration value, *0b01000000*, is located in minor position 34. Compounding the frames, starting with minor address 32, results in a configuration value of *0x0000000000400080*.

[12]Right for *ALUT* and *CLUT*, left for *BLUT* and *DLUT*.

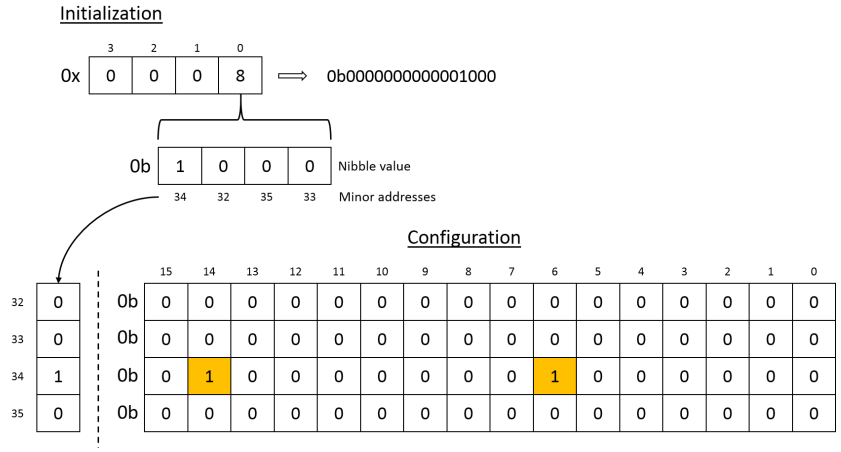


Figure 4.20: Applying *NLM* to a RAM16X8 construct with an initialisation of $0x0008$

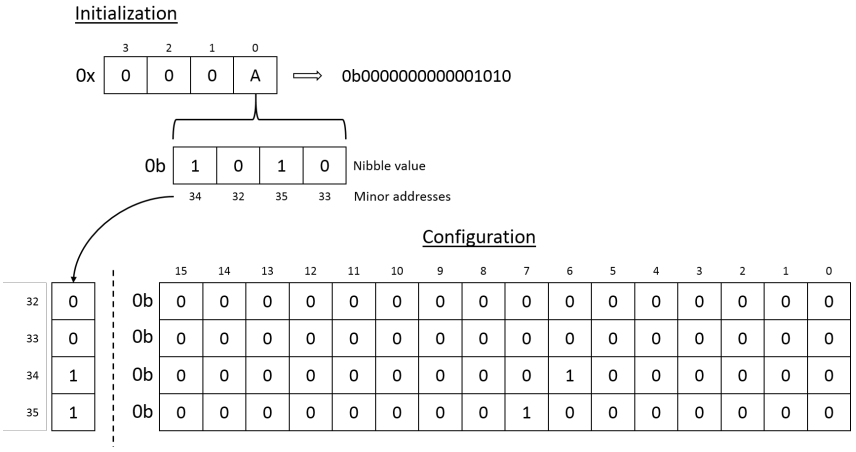


Figure 4.21: Applying *NLM* to a RAM16X8 construct with an initialisation of $0x000A$

4.6.4 Experiments 9 and 10: Shift register LUT (SRL) configuration

Shown in Table 4.12 is an excerpt of the results when comparing a design with an SRL16 construct to the base design. What is interesting to note, is that when these results are compared to those of the single storage constructs (as listed in Table 4.11), an additional duplicate value is found in the bitstream composition. It is thus evident that the SRL cannot simply be modelled as a multiplexer, as was the case for the storage constructs.

According to the Virtex®-5 user guide [111] the SRL is represented by both a shift register and a multiplexer, as shown in Figure 4.22. This is because the SRL has two outputs: a synchronous output allowing access to the most recent bit shifted out (*SHIFTOUT*) and an output where any of the bits in the register can be read by using the address lines. This implies that a shift register has to be added to the LUT model shown in Figure 4.15. Figure 4.22 shows the model of a LUT as an addressable shift register.

As a result, the encoding of the SRL is different from those discussed in the previous sections. The first step is again to compile a truth table for the SRL. Since a dynamic read of the SRL is

Table 4.12: Excerpt of the experimental results when comparing an SRL16 construct while incrementing the LSN

Initialization value	LUTA content	LOUT (Hex)	LUT value (binary value of bitstream 1)	LUT value (binary value of bitstream 2)	Type	Pos	Row	Major addr	Minor addr
0x0001	0x0000000000000001	00D20	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	32
		00D21	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	33
		00D22	00000000000000000000000000000000	00000000000000000000000000000000	0	0	0	26	34
		00D23	00000000000000000000000000000000	00000000000000000000000000000000	0	0	0	26	35
0x0002	0x0000000000000002	00D20	00000000000000000000000000000000	00000000000000000000000000000000	0	0	0	26	32
		00D21	00000000000000000000000000000000	00000000000000000000000000000000	0	0	0	26	33
		00D22	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	34
		00D23	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	35
0x0003	0x0000000000000003	00D20	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	32
		00D21	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	33
		00D22	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	34
		00D23	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	35
0x0004	0x0000000000000004	00D20	00000000000000000000000000000000	00000000000000000000000001000000	0	0	0	26	32
		00D21	00000000000000000000000000000000	00000000000000000000000001000000	0	0	0	26	33
		00D22	00000000000000000000000000000000	00000000000000000000000000000000	0	0	0	26	34
		00D23	00000000000000000000000000000000	00000000000000000000000000000000	0	0	0	26	35
0x0005	0x0000000000000005	00D20	00000000000000000000000000000000	00000000000000000000000011000000	0	0	0	26	32
		00D21	00000000000000000000000000000000	00000000000000000000000011000000	0	0	0	26	33
		00D22	00000000000000000000000000000000	00000000000000000000000000000000	0	0	0	26	34
		00D23	00000000000000000000000000000000	00000000000000000000000000000000	0	0	0	26	35
0x0006	0x0000000000000006	00D20	00000000000000000000000000000000	00000000000000000000000001000000	0	0	0	26	32
		00D21	00000000000000000000000000000000	00000000000000000000000001000000	0	0	0	26	33
		00D22	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	34
		00D23	00000000000000000000000000000000	00000000000000000000000010000000	0	0	0	26	35

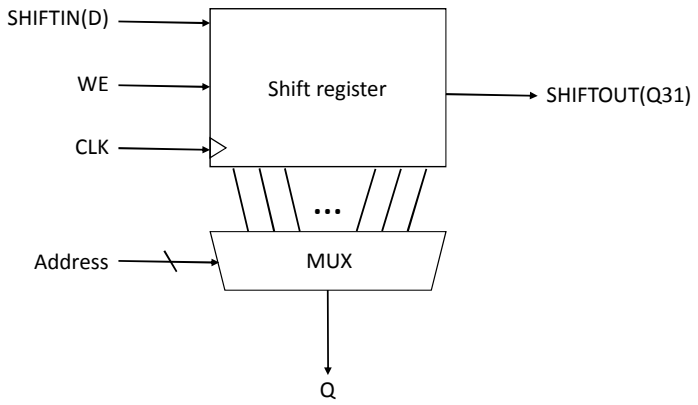


Figure 4.22: Block diagram representation of a shift register LUT

performed using the entire address line, but with the least significant bit (LSB) automatically tied high [111], only every second entry in the truth table is of importance. An example is shown in Figure 4.23. For this setup, *A1* was the LSB tied high. Also shown is an example of an initialization parameter of *0x0000000000000005* and since only the positions in the truth table where $A1 = 1$, two '1's are placed in line 2 and 6. As for the previous LUT constructs, these positions can be used to derive the Boolean functions, which in turn can be used to derive the configuration. In this instance, the Boolean expression is found to be $(A1 \wedge \overline{A2} \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5} \wedge \overline{A6}) \vee (A1 \wedge \overline{A2} \wedge A3 \wedge \overline{A4} \wedge \overline{A5} \wedge \overline{A6})$ which equates to *0x8000400000000000*. However, it can be seen from Table 4.12 that the correct configuration is *0xC000C00000000000*.

As mentioned previously, the configuration of an SRL construct is in reality a combination of two values. The configuration should therefore reflect it, where in the previous example it does not. It was found that the *SHIFTOUT* value can be calculated using the Boolean expression derived from the line preceding the line with a '1' in the truth table. These line are also indicated in the example SRL truth table in Figure 4.22. Using these lines, it was found that the Boolean expression representing the LUT configuration is $(A1 \wedge \overline{A2} \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5} \wedge \overline{A6}) \vee (A1 \wedge \overline{A2} \wedge A3 \wedge \overline{A4} \wedge \overline{A5} \wedge \overline{A6}) \vee (\overline{A1} \wedge \overline{A2} \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5} \wedge \overline{A6}) \vee (\overline{A1} \wedge \overline{A2} \wedge A3 \wedge \overline{A4} \wedge \overline{A5} \wedge \overline{A6})$, which gives the correct configuration of *0xC000C00000000000*.

This method can be used to derive the configuration of both SRL16 and SRL32. The only difference is that since SRL16 has one less address line, it is tied high. The implication is that SR16 only uses the bottom half of the truth table if *A6* is tied high. Consider the example discussed in the preceding paragraph. If this method is to be applied to an SRL16, the Boolean expression derived from the truth table in Figure 4.24 is: $(A1 \wedge \overline{A2} \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5} \wedge A6) \vee (A1 \wedge \overline{A2} \wedge A3 \wedge \overline{A4} \wedge \overline{A5} \wedge A6) \vee (\overline{A1} \wedge \overline{A2} \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5} \wedge A6) \vee (\overline{A1} \wedge \overline{A2} \wedge A3 \wedge \overline{A4} \wedge \overline{A5} \wedge A6)$. This gives a configuration of *0x00C000C000000000*.

In a similar fashion, it can be shown that it is true for more complex initializations. This was verified using *0x000000000000A1B2* and *0x0000000011223344* for SRL16 and SRL32 respectively. For these values, configurations of *0x0028002800B300B3* and *0x5A0A5A0A0AA00AA0* were respectively obtained.

A6	A5	A4	A3	A2	A1	Output	
0	0	0	0	0	0	0	Shift out
0	0	0	0	0	1	1	Dynamic read
0	0	0	0	1	0	0	
0	0	0	0	1	1	0	
0	0	0	1	0	0	0	Shift out
0	0	0	1	0	1	1	Dynamic read
0	0	0	1	1	0	0	
0	0	0	1	1	1	0	

Figure 4.23: Excerpt of a truth table used to calculate the configuration of an SRL32

A6	A5	A4	A3	A2	A1	Output	
1	0	0	0	0	0	0	Shift out
1	0	0	0	0	1	1	Dynamic read
1	0	0	0	1	0	0	
1	0	0	0	1	1	0	
1	0	0	1	0	0	0	Shift out
1	0	0	1	0	1	1	Dynamic read
1	0	0	1	1	0	0	
1	0	0	1	1	1	0	

Figure 4.24: Excerpt of a truth table used to calculate the configuration of an SRL16

4.7 Concluding remarks

This chapter discussed a methodology used to parse and analyse a Xilinx® Virtex®-5 VFX70T bitstream in order to extract specific LUT content. This was done to remove all layers of abstraction from the bitstream to allow changing of FPGA resources at bit-level. This information can be used in a reconfigurable system to adapt the LUTs in a bitstream stored in BRAM. By removing all layers of abstraction, any additional overhead can be removed from the specialisation process, allowing it to take place in real-time.

Despite obscurity in the bitstream, the bitstreams were successfully parsed. It was found that each frame contains a subset (16-bits) of an entire row's LUT configuration and since 64-bits are required to configure each LUT, at least four frames are required. These configurations are equally spaced among four consecutive frames.

By comparing multiple bitstreams to a base design, it was possible to derive a set of mathematical formulae that allows a designer to determine the frame address of a slice through its coordinates. Since Virtex®-5 FPGAs do not have a safe reconfiguration scheme, this protects a designer from accidentally making changes to unknown components, potentially damaging the device.

When knowing the frame address of the slice being reconfigured, the next logical progression is knowing what it is being reconfigured to. Various different LUT constructs were analysed to determine the bitstream encoding of each construct. This makes it possible to know the effect a bit-level change has on a high-level design and vice versa. The multiplexer configuration strings derived by Castellone were verified (and expanded) by modelling each LUT in the two different slices as a multiplexer. It was shown that these strings can be used to determine the

configuration of any LUT construct for any Xilinx® Virtex®-5 FPGA.

Determining the configuration of each LUT construct, is done by starting with the truth table for the specific element. Unfortunately, some manual design is required to determine the corresponding entries in the truth table. A Boolean function can then be derived by looking at the conditions in the truth table where the output is positive. By inserting the multiplexer strings into this Boolean expression, the bitstream configuration of the construct can be derived. The only slight deviation from this methodology is when deriving the SRL bitstream encoding. This is due to the fact that an SRL is modelled as a multiplexer with a shift register as input. To determine the configuration of an SRL construct, both the line where the truth table is positive and the preceding line should be used to derive the Boolean expression.

A second method to derive the configurations, dubbed the *Nibble Location Method*, was also derived from the experiments and discussed in this chapter. It relies on the position of each nibble in the initialisation parameter to derive the configuration of the LUT. The main drawback of this method is that each LUT construct has a unique solution to determine its configuration, complicating automation of the process. Consequently, the method discussed in the previous paragraph is rather used for the bitstream specialiser designed and implemented in the next two chapters.

CHAPTER 5

SPECIALISING THE BITSTREAM

"You're only given one little spark of madness. You mustn't lose it"

— Robin Williams

As shown in the previous chapter, it is possible to derive the configuration of the LUTs by simply knowing the construct, the routing to the address pins and the initialisation parameters. It was also shown that it is possible to derive the frame address from the physical position of the LUT and locate its configuration bits in the bitstream from the slice coordinates. Using this information, it is now possible to derive a bitstream specialiser capable of adapting the configuration bits of a LUT. This chapter discusses the design of this specialiser and makes the second contribution by proposing a novel method for specialising an FPGA configuration. It differs from the specialisation methods found in the literature by using passive, parallel logic to manipulate FPGA resources at bit-level and in real-time.

5.1 Passive bitstream specialisation

The common theme running through the previous chapter is that any LUT can be expressed as a truth table, with each line representing some sort of Boolean function. The output of the LUT is defined by its *INIT*-parameters and the bitstream encoding determined by the LUT construct. It is therefore possible to generalise a LUT by adapting the output to also be a function of the initialisation. As shown in Figure 5.1, this was done by adding an additional initialisation column to the truth table, representing the 64-bits of the initialisation parameter. The general idea is that each bit in the initialisation parameter selects a line in the LUT, thus determining the output.

Figure 5.2 illustrates how Boolean logic can be used to implement this truth table. Even though excessively large gates are used in the figure, and it could be reduced by using Boolean algebra,

it only aids in illustrating the concept behind the specialiser. The only simplification done was factoring *A6*, which selects the top or bottom of the truth table.

Using this schematic as a guideline, a bitstream specialiser can be implemented in VHDL that takes the LUT initialisation and construct type as inputs, and produces a configuration string based on these inputs.

	A6	A5	A4	A3	A2	A1	Initialisation	Output
1	0	0	0	0	0	0	initialisation(0)	X
2	0	0	0	0	0	1	initialisation(1)	X
3	0	0	0	0	1	0	initialisation(2)	X
4	0	0	0	0	1	1	initialisation(3)	X
5	0	0	0	1	0	0	initialisation(4)	X
6	0	0	0	1	0	1	initialisation(5)	X
7	0	0	0	1	1	0	initialisation(6)	X
8	0	0	0	1	1	1	initialisation(7)	X
9	0	0	1	0	0	0	initialisation(8)	X
10	0	0	1	0	0	1	initialisation(9)	X
11	0	0	1	0	1	0	initialisation(10)	X
12	0	0	1	0	1	1	initialisation(11)	X
13	0	0	1	1	0	0	initialisation(12)	X
14	0	0	1	1	0	1	initialisation(13)	X
15	0	0	1	1	1	0	initialisation(14)	X
16	0	0	1	1	1	1	initialisation(15)	X
17	0	1	0	0	0	0	initialisation(16)	X
18	0	1	0	0	0	1	initialisation(17)	X
19	0	1	0	0	1	0	initialisation(18)	X
20	0	1	0	0	1	1	initialisation(19)	X
21	0	1	0	1	0	0	initialisation(20)	X
22	0	1	0	1	0	1	initialisation(21)	X
23	0	1	0	1	1	0	initialisation(22)	X
24	0	1	0	1	1	1	initialisation(23)	X
25	0	1	1	0	0	0	initialisation(24)	X
26	0	1	1	0	0	1	initialisation(25)	X
27	0	1	1	0	1	0	initialisation(26)	X
28	0	1	1	0	1	1	initialisation(27)	X
29	0	1	1	1	0	0	initialisation(28)	X
30	0	1	1	1	0	1	initialisation(29)	X
31	0	1	1	1	1	0	initialisation(30)	X
32	0	1	1	1	1	1	initialisation(31)	X
33	1	0	0	0	0	0	initialisation(32)	X
34	1	0	0	0	0	1	initialisation(33)	X
35	1	0	0	0	1	0	initialisation(34)	X
36	1	0	0	0	1	1	initialisation(35)	X
37	1	0	0	1	0	0	initialisation(36)	X
38	1	0	0	1	0	1	initialisation(37)	X
39	1	0	0	1	1	0	initialisation(38)	X
40	1	0	0	1	1	1	initialisation(39)	X
41	1	0	1	0	0	0	initialisation(40)	X
42	1	0	1	0	0	1	initialisation(41)	X
43	1	0	1	0	1	0	initialisation(42)	X
44	1	0	1	0	1	1	initialisation(43)	X
45	1	0	1	1	0	0	initialisation(44)	X
46	1	0	1	1	0	1	initialisation(45)	X
47	1	0	1	1	1	0	initialisation(46)	X
48	1	0	1	1	1	1	initialisation(47)	X
49	1	1	0	0	0	0	initialisation(48)	X
50	1	1	0	0	0	1	initialisation(49)	X
51	1	1	0	0	1	0	initialisation(50)	X
52	1	1	0	0	1	1	initialisation(51)	X
53	1	1	0	1	0	0	initialisation(52)	X
54	1	1	0	1	0	1	initialisation(53)	X
55	1	1	0	1	1	0	initialisation(54)	X
56	1	1	0	1	1	1	initialisation(55)	X
57	1	1	1	0	0	0	initialisation(56)	X
58	1	1	1	0	0	1	initialisation(57)	X
59	1	1	1	0	1	0	initialisation(58)	X
60	1	1	1	0	1	1	initialisation(59)	X
61	1	1	1	1	0	0	initialisation(60)	X
62	1	1	1	1	0	1	initialisation(61)	X
63	1	1	1	1	1	0	initialisation(62)	X
64	1	1	1	1	1	1	initialisation(63)	X

Figure 5.1: Generalised truth table representing LUT-output as a function of the initialisation

5.2 Implementation of the bitstream specialiser

The top level instantiation of the bitstream specialiser is shown in Figure 5.3. A rising edge on *ENABLE* triggers the specialisation process and a configuration string is generated according to the initialisation parameter (*INIT_param*) and LUT construct type (*SLICE_type*), which selects the corresponding LUT configuration strings listed in Table 4.9. The resulting configuration string is made available on *ConfigString*.

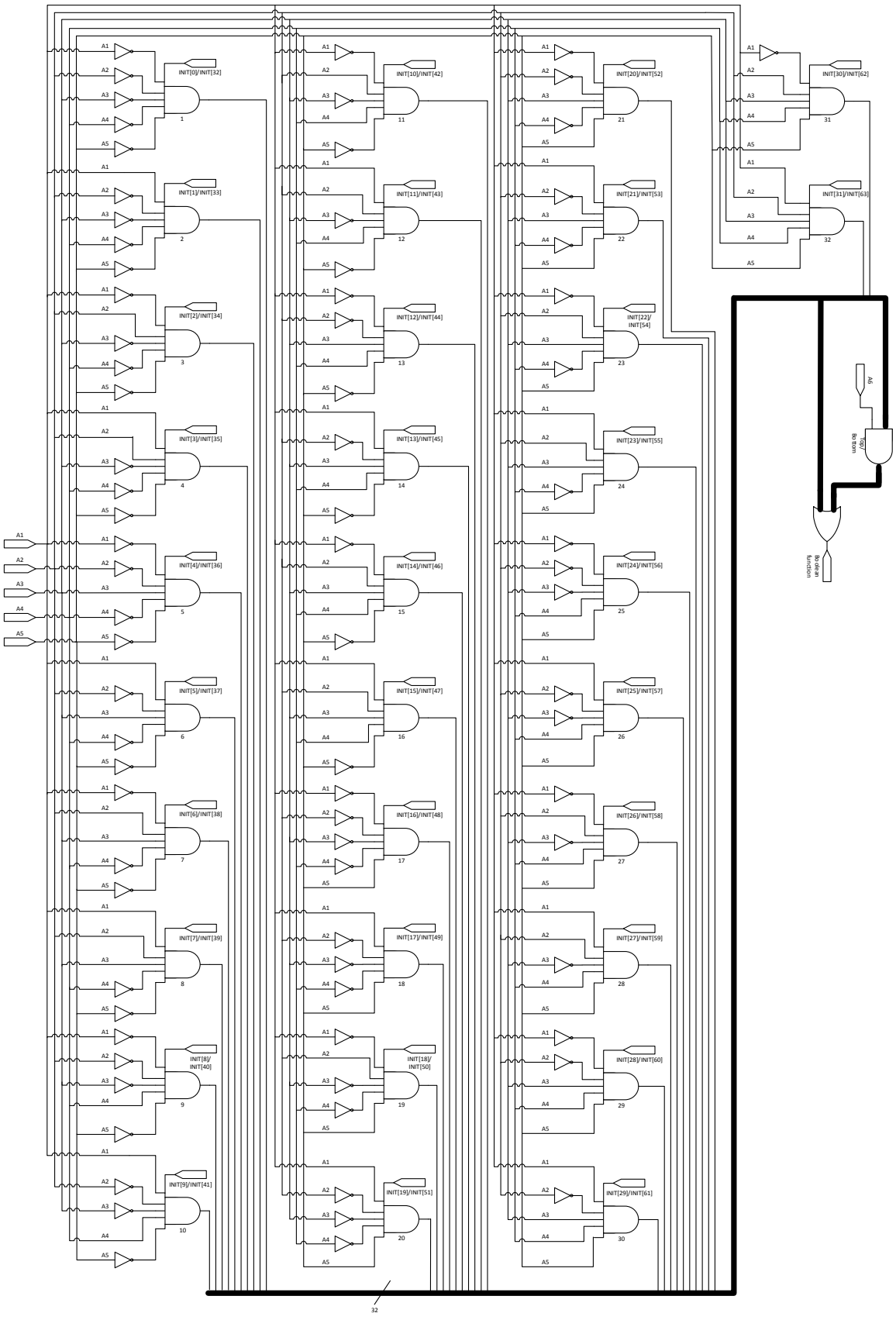


Figure 5.2: Logic diagram depicting the Boolean implementation of the generalised truth table depicted in Figure 5.1

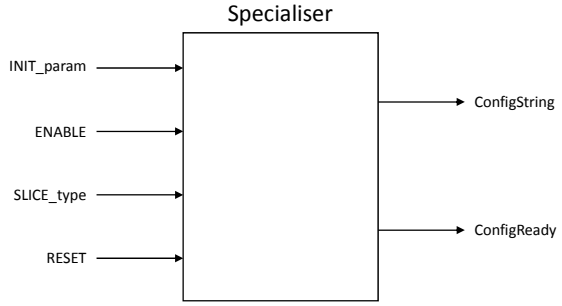


Figure 5.3: Block diagram of the top-level bitstream specialiser initialisation

In an attempt to allow the specialiser to execute at the fastest possible speed, asynchronous design was used and the *ConfigReady* pin used to establish handshaking with the rest of the hardware. Despite the advantages of asynchronous design [112], it is not supported by Xilinx® and it is highly recommended to rather synchronise the specialiser to the rest of the design. This is because asynchronous logic requires glitch-less combinatorial gates and Xilinx LUTs are not guaranteed to be glitch-free [113]. However, for the purpose of this thesis, asynchronous design is sufficient for illustrating the fastest possible specialisation time.

The integration of the specialiser with the rest of the reconfiguration process is illustrated in Figure 5.4, showing the connection to the reconfiguration controller. Referring back to Chapter 4, it was shown that each LUT construct is a composition of 5 and 6-input Boolean expressions, *LUT5* and *LUT6*, with the former being a subset of the latter—as shown in Table 4.9. Each LUT is initialized by an *INIT*-parameter, requiring eight in total. This implies the same number of specialisers are required to allow any construct to be adapted.

As seen in the figure, the even-numbered specialisers are used to generate new configuration strings for the *LUT6*-expressions (*ConfigString_A6* to *ConfigString_D6*), whereas the odd-numbered specialisers are used for *LUT5* (*ConfigString_A5* to *ConfigString_D5*). The AND-gates at the bottom are used to synchronise the asynchronous specialisers to the reconfiguration controller. Each *ConfigReady* signal is driven high as soon as the new configuration string is available on *ConfigString*. These values can only be used in the reconfiguration process once all eight strings are available, after which they are supplied to the reconfiguration controller. This controller uses these values during the reconfiguration process to adapt the configuration of the LUT construct starting at the address supplied on *LUT_baseAddress*. Once the reconfiguration process is completed, *SpecializerRST* can be used to generate a soft reset, if required.

The newly generated configuration strings are injected into the bitstream by matching the number of words read to the address on *LUT_baseAddress*. If they match, the first 16-bits of *ALUT* and *BLUT* are sent to the configuration memory in lieu of the word read. This is followed by the first 16-bits of *CLUT* and *DLUT* for the next word read. This process is then continued for the remaining configuration bits, using the distribution that was shown in Figures 4.13 and 4.14.

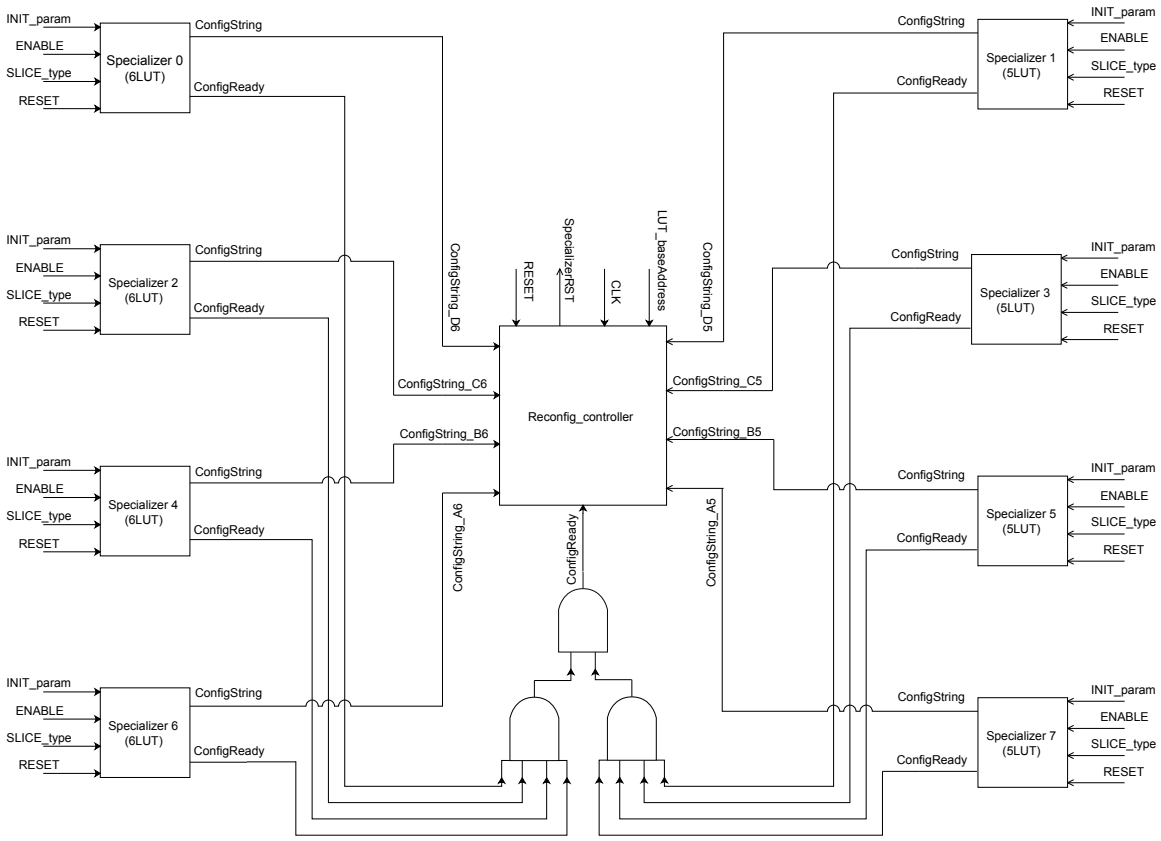


Figure 5.4: Diagram depicting the interconnectivity of the specialiser and the reconfiguration controller

5.3 Verifying the specialisation process

Consider a LUT construct that has to be reconfigured according to the *Initialisation* parameters listed in Table 5.1. The Boolean expression represented by each of the LUTs in the construct can easily be obtained from the truth table information. The configuration strings listed in the table are determined from the values listed in Table 4.9. The results of this translation are also listed in Table 5.1 under the heading *Configuration string*. *LUT name* simply refers to the type of LUT being addressed by the initialisation parameter.

Table 5.1: Initialisation parameters used to verify the specialisation process

Initialisation	**Value**	**LUT name**	**Configuration string**
INIT_00	0x000000000000B5F0	D6LUT	0x0050007000400070
INIT_01	0x00000000000055AE	D5LUT	0x900060009000E000
INIT_02	0x000000000000EA90	C6LUT	0x0060001000300050
INIT_03	0x0000000000001C66	C5LUT	0x900040006000A000
INIT_04	0x0000000000000000	B6LUT	0x0000000000000000
INIT_05	0x0000000000000000	B5LUT	0x0000000000000000
INIT_06	0x0000000000000000	A6LUT	0x0000000000000000
INIT_07	0x0000000000000000	A5LUT	0x0000000000000000

Shown in Figure 5.5 are the simulated timing results of the specialiser. As can be seen, the new *INIT*-parameters are requested and the specialiser enabled via *ENABLE*. The configuration strings corresponding to the LUT-type are then selected (*A1* to *A6*) and new configuration strings generated for each LUT (*ConfigString_D6* to *ConfigString_A5*). The values shown in the figure correspond to those listed in Table 5.1, indicating correct operation of the specialiser.

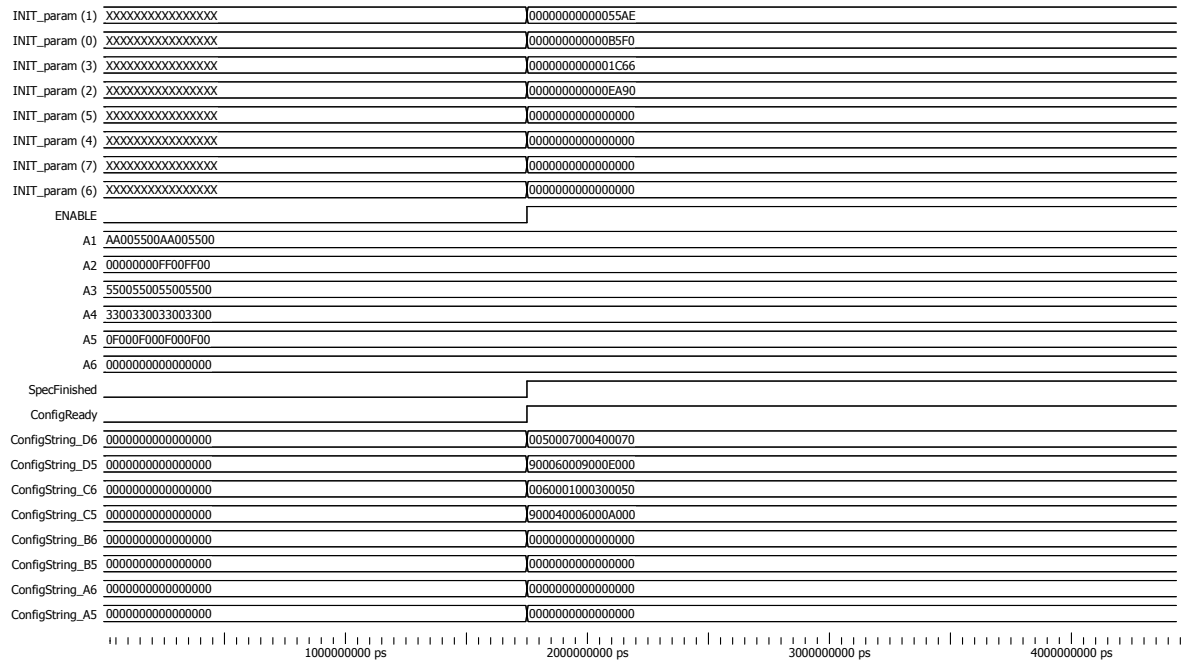


Figure 5.5: Simulated timing results of the configuration specialiser

Figure 5.6 illustrates the timing when the specialiser is used in the reconfiguration process. As already discussed, once the new configuration strings are created, the reconfiguration process is initiated by *ConfigReady*. The configuration data is then read from the memory one word at a time, as shown by *Configuration read* and the word counter, *Word read*, incremented for each word read. The values are then transferred to the ICAP, as indicated by *Configuration data*. If the word counter matches *LUT_baseAddress*, *LUT_baseAddress + 1*, *LUT_baseAddress + 41*, *LUT_baseAddress + 42*, *LUT_baseAddress + 82*, *LUT_baseAddress + 83*, *LUT_baseAddress + 123* or *LUT_baseAddress + 124*, the corresponding configuration strings are injected into the current *Configuration data* while it is transferred to the ICAP. Also note that these values have to be byte swapped before being sent to the ICAP [14], as shown by *ICAP Input*.

In this particular instance, the *LUT_baseAddress* of the LUT to be reconfigured was identified as 28. This implies that the *Configuration data* transferred to the ICAP are simply the *Configuration read*[1]. However, when the word counter reaches 28, a value of 0x00000000 is injected into the *Configuration data*. Similarly, a value of 0x90509060 is injected into the 29^{th} (*LUT_baseAddress + 1*) word read. These values are obtained by concatenating the relevant bits from the configuration strings. For example, 0x00000000 is obtained by concatenating *ConfigString_B5(63:56)*, *ConfigString_B6(55:48)*, *ConfigString_A5(63:56)* and

[1]Signals were used in the VHDL-code, causing the values to be available on the next clock cycle.

ConfigString_A6(55:48)—in that order.[2] To obtain 0x90509060, *ConfigString_D5(63:56)*, *ConfigString_D6(55:48)*, *ConfigString_C5(63:56)* and *ConfigString_C6(55:48)* are concatenated.

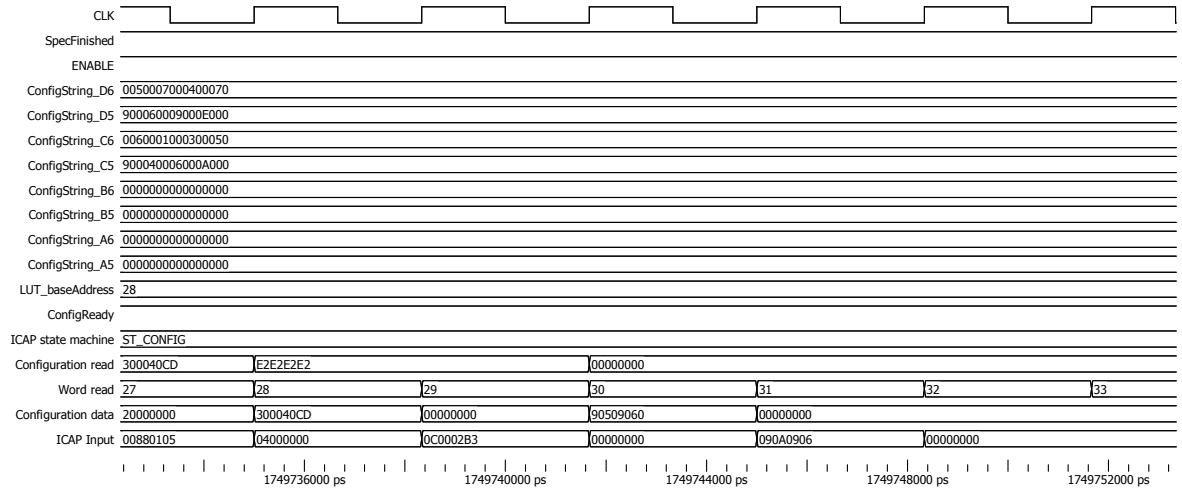


Figure 5.6: Simulated timing results of the reconfiguration process with specialiser

5.4 Area overhead

As can be expected, each of these modules required to specialise a configuration negatively contributes to the area metric of the functional density. However, as will be seen in the next chapter, this area overhead is significantly less than those of conventional reconfiguration techniques. Listed in Table 5.2 is the total hardware utilisation of the specialiser, reconfiguration monitor and controller designed in this chapter. Slight variations can be observed between the specialiser implementations due to the optimisation of the toolset.

In general, it was found that the specialiser requires an average of 3 slices and 7 LUTs per module implementation. The reconfiguration monitor requires 10 slices, 25 slice registers and 31 LUTs to implement. The reconfiguration controller, on the other hand, requires significantly more LUTs to be implemented, due to the complexity of the state machine. In total, the specialisation process contributes 72 slices, 111 slice registers and 208 LUTs to the total resource utilisation.

5.5 Concluding remarks

This chapter discussed the implementation of the bitstream specialiser used to adapt the configuration bits of a LUT. The previous chapter showed that the configuration of each LUT in a construct can be derived from the Boolean expression represented by its truth table. By generalising this truth table so that its output is only dependent on the initialisation parameter

[2]Big endian notation is used for representing the configuration data.

Table 5.2: Hardware requirements for specialising a bitstream

Module name	**Slices**	**Slice Reg**	**LUTs**
Reconfiguration monitor	10	25	31
Reconfiguration controller	46	86	144
Specialiser 0	3	0	7
Specialiser 1	3	0	7
Specialiser 2	3	0	6
Specialiser 3	2	0	6
Specialiser 4	1	0	5
Specialiser 5	4	0	2
Specialiser 6	3	0	7
Specialiser 7	3	0	7
Total	**78**	**111**	**222**

of the LUT, it was shown that passive Boolean logic can be used to derive the configuration for each LUT.

This set of passive Boolean logic was then implemented in VHDL and integrated into the reconfiguration process. This was done by injecting the newly generated configuration strings into the bitstream as it is read from the memory. Using this type of architecture for the specialiser requires the least amount of overhead and results in extremely fast specialisation of the bitstream. As seen in Figure 5.5, the newly generated configuration strings are available immediately when requested.

Using this architecture, it is now possible to generate new configurations for any LUT in real-time. Combining this with a BRAM-based architecture, an application can be specialised and reconfigured with the highest throughput and least amount of overhead. This should reduce the functional density of reconfigured applications to such an extent that real-time applications can be reconfigured. The next chapter aims to prove just that.

CHAPTER 6

LOOKUP TABLE (LUT) RECONFIGURATION

"Do not try and bend the spoon. That's impossible.
Instead only try to realize the truth."
— "Spoon boy", *The Matrix*

Using the bitstream specialiser discussed in Chapter 5, it is now possible to overcome the limitations imposed by using the BRAM-based architecture. This chapter makes the last contribution by illustrating the functionality and advantages of combining this specialiser with the BRAM-based architecture by reconfiguring a distributed multiply-accumulate (MAC). The reason for selecting this particular implementation, is because it represents a good example of a dynamically reconfigurable application. Not only is it the foundation of many digital implementations, including filters and PID control, but it can also be implemented using distributed arithmetic (DA). This is of particular interest, since DA performs multiplication using LUTs and the specialiser developed is capable of adapting the content of these LUTs. This expands the application domain where the specialiser can be used.

6.1 Distributed arithmetic

Distributed arithmetic is so named because the arithmetic operations performing a operation are not lumped together in a familiar fashion [114, 115]. Instead, distributed arithmetic are computational algorithms performing multiplication with lookup table based schemes. Distributed arithmetic specifically targets the sum of products computation, predominately featuring in many important DSP filtering and frequency transformation functions.

The arithmetic sum of products defining the response of a linear, time-invariant network can be represented by:

$$y = A_1x_1 + A_2x_2 + A_3x_3 + \ldots + A_kx_k \tag{6.1}$$

$$\therefore y(n) = \sum_{k=1}^{K} A_k x_k(n), \tag{6.2}$$

with $y(n)$ the response of the network and $x_k(n)$ the k^{th} input variable at time n. A_k is the weighing factor of the k^{th} input and remains constant for all n. Implementing this operation in hardware is typically done using the architecture shown in Figure 6.1.

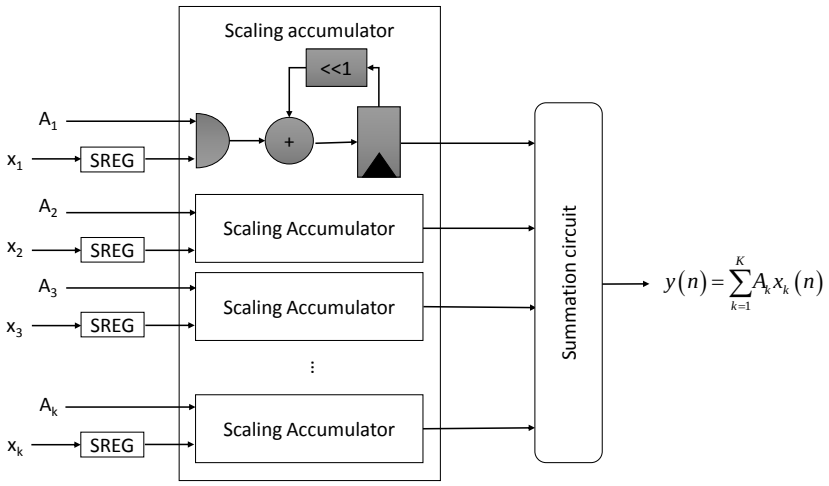


Figure 6.1: Possible hardware implementation for calculating the sum of products

By defining x_k to be a B-bit scaled two's compliment number, in other words,

$$x_k = \left\{x_{k0}, x_{k1}, x_{k2}, \ldots, x_{k(B-1)}\right\}, \tag{6.3}$$

x_k can be expressed as

$$x_k = -x_{k0} + \sum_{b=1}^{B-1} x_{kb}2^{-b}, \tag{6.4}$$

with x_{k0} the sign bit. Substituting (6.4) into (6.2), yields

$$\begin{aligned} y &= \sum_{k=1}^{K} A_k \left[-x_{k0} + \sum_{b=1}^{B-1} x_{kb}2^{-b}\right] \\ &= \sum_{k=1}^{K} -A_k x_{k0} + \sum_{k=1}^{K}\sum_{b=1}^{B-1} A_k x_{kb}2^{-b} \end{aligned} \tag{6.5}$$

Expanding (6.5) even further, it is obtained that

$$\begin{aligned} y &= -\left[x_{10}A_1 + x_{20}A_2 + x_{30}A_3 + \ldots + x_{k0}A_k\right] \\ &+ \left[(x_{11}A_1)\,2^{-1} + (x_{12}A_1)\,2^{-2} + (x_{13}A_1)\,2^{-3} + \ldots + \left(x_{1(B-1)}A_1\right)2^{-(B-1)}\right] \\ &+ \left[(x_{21}A_2)\,2^{-1} + (x_{22}A_2)\,2^{-2} + (x_{23}A_2)\,2^{-3} + \ldots + \left(x_{2(B-1)}A_2\right)2^{-(B-1)}\right] \\ &+ \left[(x_{31}A_3)\,2^{-1} + (x_{32}A_3)\,2^{-2} + (x_{33}A_3)\,2^{-3} + \ldots + \left(x_{3(B-1)}A_3\right)2^{-(B-1)}\right] \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ &+ \left[(x_{k1}A_k)\,2^{-1} + (x_{k2}A_k)\,2^{-2} + (x_{k3}A_k)\,2^{-3} + \ldots + \left(x_{k(B-1)}A_k\right)2^{-(B-1)}\right]. \end{aligned} \tag{6.6}$$

Grouping the relevant terms together and re-arranging the terms, it can be shown that

$$\begin{aligned} y &= -\left[x_{10}A_1 + x_{20}A_2 + x_{30}A_3 + \ldots + x_{k0}A_k\right] \\ &+ \left[x_{11}A_1 + x_{21}A_2 + x_{31}A_3 + \ldots + x_{k1}A_k\right]2^{-1} \\ &+ \left[x_{12}A_1 + x_{22}A_2 + x_{32}A_3 + \ldots + x_{k2}A_k\right]2^{-2} \\ &+ \left[x_{13}A_1 + x_{23}A_2 + x_{33}A_3 + \ldots + x_{k3}A_k\right]2^{-3} \\ &\qquad\qquad\qquad\qquad \vdots \\ &+ \left[x_{1(B-1)}A_1 + x_{2(B-1)}A_2 + x_{3(B-1)}A_3 + \ldots + x_{k(B-1)}A_k\right]2^{-(B-1)} \\ &= -\sum_{k=1}^{K} x_{k0}A_k + \sum_{n=1}^{B-1}\left[\sum_{k=1}^{K} x_{kn}A_k\right]2^{-n}. \end{aligned} \tag{6.7}$$

Each term within the brackets denotes a binary *AND*-operation involving a bit of the input and all the bits of the constant, whereas the plus sign indicates an arithmetic sum operation. The exponential factors indicate that each bracketed pair has to be scaled. Considering that $\left[\sum_{k=1}^{K} x_{kb}A_k\right]$ has only 2^K possible values, it is possible to construct a lookup table addressed by $x_{1b}, x_{2b}, \ldots, x_{3b}$—with b the individual bits in x. This lookup table is referred to as a distributed arithmetic lookup table, or DALUT. The resulting architecture is shown in Figure 6.2. As can be seen, the arithmetic operation has been reduced to addition, subtraction and binary scaling. The result is a design better suited to FPGAs, not only in terms of area utilization, but also in terms of power consumption.

Distributed arithmetic can be used to implement a wide variety of applications, such as filters [116, 117], control systems [118, 119], system-on-chip (SoC) applications [120] and discrete cosine transforms [121–123]. All these implementations utilise lookup tables to store certain aspects of the design. This is of particular interest, since the bitstream specialiser discussed in Chapter 5 is specifically designed to target LUTs. This implies that the configuration of any of these designs can be specialised.

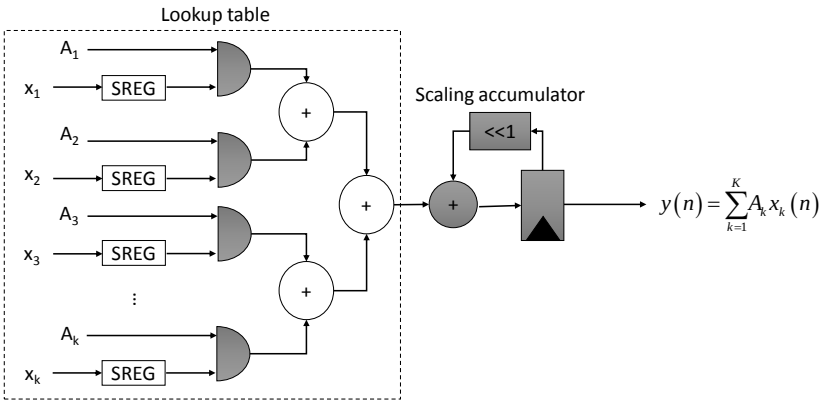


Figure 6.2: Distributed arithmetic realisation of the sum of products

6.2 Distributed multiply-accumulate (MAC)

To illustrate the specialisation and reconfiguration of a distributed application, a multiply-accumulate (MAC) was implemented as shown in Figure 6.3. With the aim to generalise the design, fixed-point arithmetic was selected as shown by the inputs $x1, x2, x3$ and $x4$. Resetting the module can either be done using an external trigger, *RESET*, or locally via *SoftRST*.

Consider a simple MAC multiplying and adding $x = [4, 8, 3, 6]$ and $A = [3, 2, 4, 5]$. Referring back to (6.7), the values to be stored in a LUT are given by $\sum_{k=1}^{4} x_{kb}A_k = x_{1b}A_1 + x_{2b}A_2 + x_{3b}A_3 + x_{4b}A_4$, resulting in the LUT shown in Table 6.1. Storing these values was done by using a RAM32X8S LUT construct. This construct is capable of storing 32 8-bit words and is initialised using the code listed in Listing 6.1—with *INIT_00* to *INIT_07* representing the contents of the LUTs to be instantiated.

The result is the complex LUT construct shown in Figure 6.4, with the timing diagram in Figure 6.5. *x1_SREG* to *x4_SREG* refer to the input shift registers whereas *OutFIFO* refers to the output shift register. As can be seen, the module takes 35 clock cycles to successfully calculate the output. This design will be used in the preceding sections and will be referred to as the "base DA MAC".

Listing 6.1: VHDL construct to instantiate RAM32X8S for the distributed MAC

```
RAM16X8S_inst : RAM32X8S
    generic map
    (
                    INIT_00 => 0x000055AA, -- INIT for bit 0 of RAM
                    INIT_01 => 0x0000A5F0, -- INIT for bit 1 of RAM
                    INIT_02 => 0x00009C66, -- INIT for bit 2 of RAM
                    INIT_03 => 0x0000EA88, -- INIT for bit 3 of RAM
                    INIT_04 => 0x00000000, -- INIT for bit 4 of RAM
                    INIT_05 => 0x00000000, -- INIT for bit 5 of RAM
                    INIT_06 => 0x00000000, -- INIT for bit 6 of RAM
                    INIT_07 => 0x00000000  -- INIT for bit 7 of RAM
    )
```

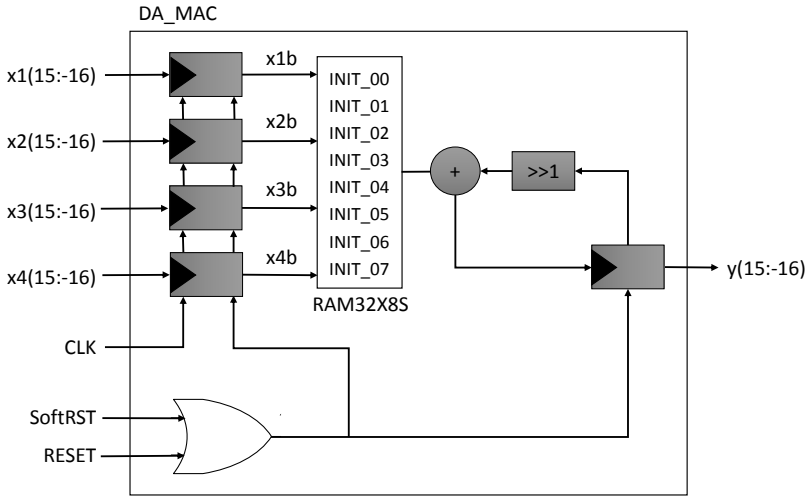


Figure 6.3: Block diagram representation of a multiply-accumulate implemented using distributed arithmetic

Table 6.1: Populated LUT for the distributed MAC

x1b	x2b	x3b	x4b	Content	Value
0	0	0	0	0	0
0	0	0	1	A4	5
0	0	1	0	A3	4
0	0	1	1	A3+A3	9
0	1	0	0	A2	2
0	1	0	1	A2+A4	7
0	1	1	0	A2+A3	6
0	1	1	1	A2+A3+A4	11
1	0	0	0	A1	3
1	0	0	1	A1+A4	8
1	0	1	0	A1+A3	7
1	0	1	1	A1+A3+A4	12
1	1	0	0	A1+A2	5
1	1	0	1	A1+A2+A4	10
1	1	1	0	A1+A2+A3	9
1	1	1	1	A1+A2+A3+A4	14

6.3 Reconfiguration implementations

Traditionally, a static application requiring different datapaths had to implement each path in parallel. This application is also referred to as a generic design, since it caters for the most generic conditions. Depending on the increase in area for each datapath, this method could yield a high functional density since no additional time is added to the execution. Using this static application—and its functional density—as the comparison base, the distributed multiply-accumulate discussed in the previous section was implemented and reconfigured using different

reconfiguration techniques, listed in Table 6.2, and the functional densities compared. This generic, static implementation is referred to as *Implementation 1.*

Currently, the design tools of FPGA manufacturers mostly only support configuration swapping.

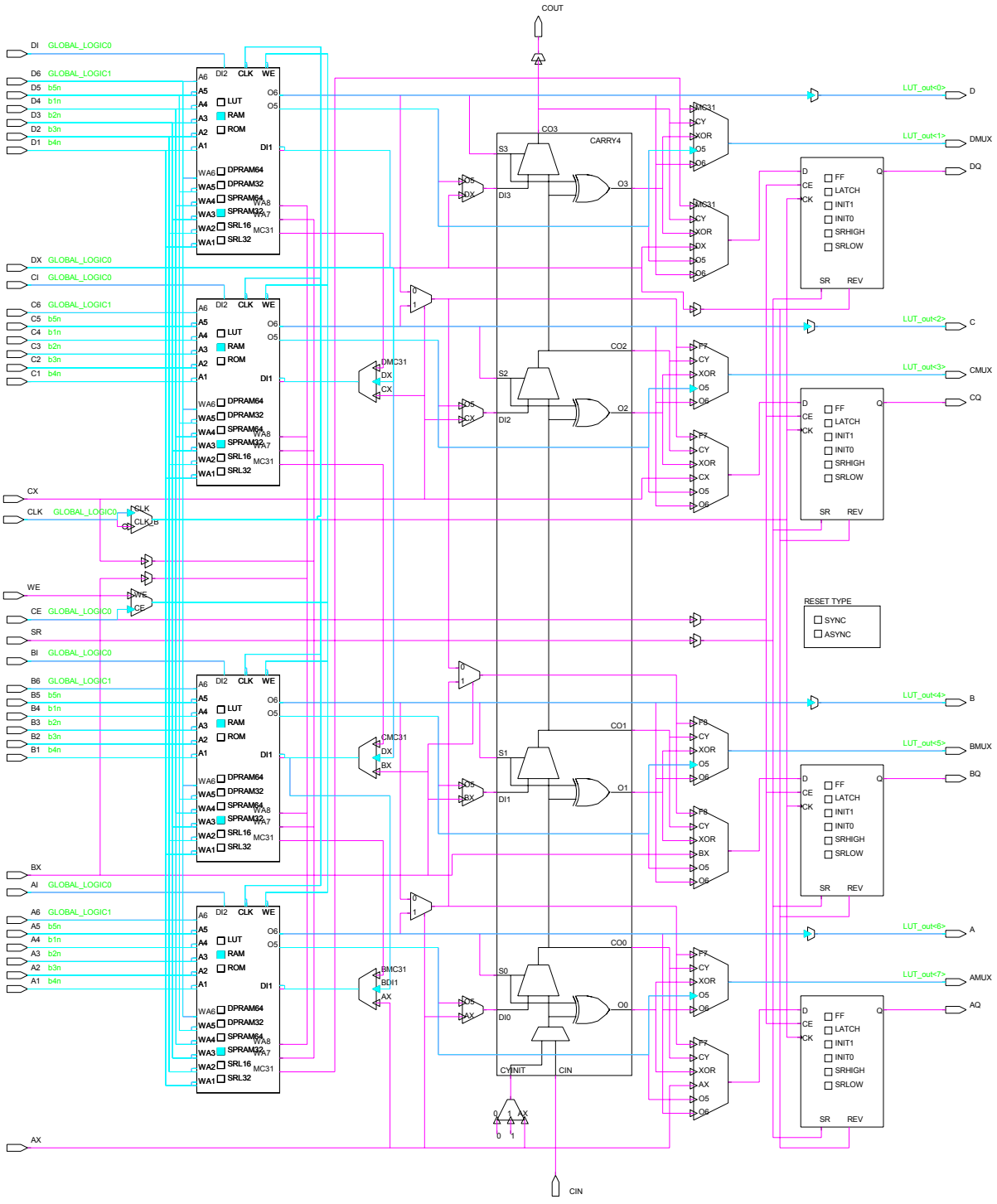


Figure 6.4: Logic diagram of the LUT construct used in the distributed arithmetic multiply-accumulate

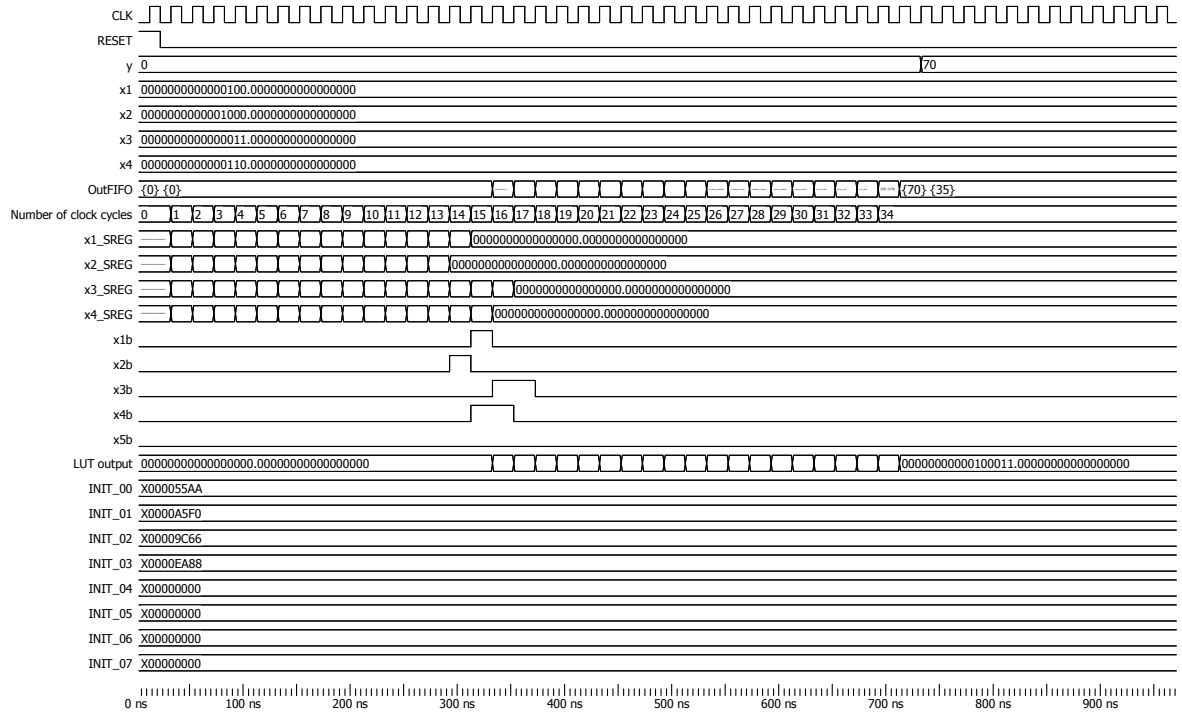


Figure 6.5: Distributed multiply-accumulate simulation results

This is a form of dynamic reconfiguration where a certain number of configurations are generated beforehand and simply swapped to and from the device. Since this is the most commonly used reconfiguration architecture, this was the first design compared to *Implementation 1*. This is referred to as *Implementation 2*, and uses module-based reconfiguration to swap between configurations. In this implementation, the different configurations are stored in CompactFlash and the MAC reconfigured by swapping these configurations with the one currently on the device. This works fine for a small number of configurations, but for implementations with a larger set of configurations, these have to be generated at run-time using the Xilinx® tool flow. This is also discussed for this implementation.

Generating new configurations at run-time adds significant overhead to the reconfiguration process. To mitigate this, the specialiser developed in Chapter 5 was implemented in software and added to *Implementation 2* to allow new configurations to be generated as required. This was the third design compared to *Implementation 1* and is dubbed *Implementation 3*.

Table 6.2: Different methods of reconfiguration used to compare functional densities

Implementation	**Reconfiguration method**
1	Generic (or static implementation, i.e. not reconfigured)
2	Configuration swapping with on-line FPGA toolflow
3	Configuration swapping with software specialiser
4	CLB bit toggle reconfiguration
5	Shift register LUT reconfiguration
6	Hardware-based reconfiguration

Implementation 4 uses an improved dynamic reconfiguration technique provided by Xilinx® that allows direct manipulation of bits inside a CLB using an embedded processor. This aims to reduce configuration overhead by not having to transfer an entire bitstream across a bus.

Implementation 5 makes use of the shift register functionality of modern FPGA's LUTs. In this setup, the configuration of a LUT can be changed by simply shifting a new value into the shift register lookup table (SRL).

The last implementation, *Implementation 6*, uses hardware controlled reconfiguration, combined with a hardware implementation of the designed specialiser to reconfigure the MAC. Not only does this technique provide reconfiguration with the least amount of overhead, it also allows the reconfiguration process to be overclocked.

To maintain uniformity between the different designs, the reconfiguration process in each implementation was clocked at the Xilinx® recommended 100 MHz, except for the last design which was also overclocked to 200 and 300 MHz respectively to illustrate the increase in functional density at higher clock speeds. For designs with an embedded PowerPC®, the processor bus was clocked at 200 MHz and cache enabled for each design. Each experiment was also performed with the cache disabled, to determine the difference in reconfiguration speed. However, due to the significant improvement the cache adds, these results were omitted.

Table 6.3: Static multiply-accumulate LUT contents

INIT	Datapath 1	Datapath 2	Datapath 3	Datapath 4	Datapath 5	Datapath 6	Datapath 7	Datapath 8
00	0x0000F0F0	0x000055AE	0x000005AE	0x0000002E	0x00000080	0x00000080	0x00000000	0x000001FE
01	0x000055AA	0x0000B5F0	0x00003550	0x00000150	0x00000150	0x00000150	0x00000150	0x00000954
02	0x00003C96	0x00001C66	0x0000146E	0x00000094	0x0000004E	0x0000004A	0x0000015A	0x0000015A
03	0x00006AE8	0x0000EA90	0x00002A94	0x00000002	0x00000094	0x00000004	0x00000104	0x00000104
04	0x000070F0	0x00000000	0x00000112	0x00000000	0x00000002	0x00000000	0x00000020	0x00000024
05	0x00008000	0x00000000	0x00000000	0x00000000	0x00000200	0x00000000	0x00000000	0x00000000
06	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
07	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
y	**224**	**119**	**263**	**103**	**228**	**32**	**64**	**93**

6.3.1 Implementation 1: Generic design

The generic design is generated by describing nine distributed MACs implemented in parallel and running the RTL description through a conventional FPGA tool flow. The resulting architecture is shown in Figure 6.6. Each MAC has a unique set of constants and selecting the datapath currently relevant to the design was done using an 8-bit DIP switch. This implementation is used as a comparison base for all the other implementations described in this chapter. The input to each MAC was kept constant at $x = [4, 8, 3, 6]$, forcing the specific output of each datapath as listed in Table 6.3. This simplifies the verification of the MAC, as each output can now be verified using a comparator that lights up an LED according to each output. An individual MAC requires a total of 141 LUTs distributed amongst 59 slices to be implemented, but due to

some optimization from the tool flow, the nine parallel MACs only require 813 LUTs–including additional logic for the configuration selector and clock management.

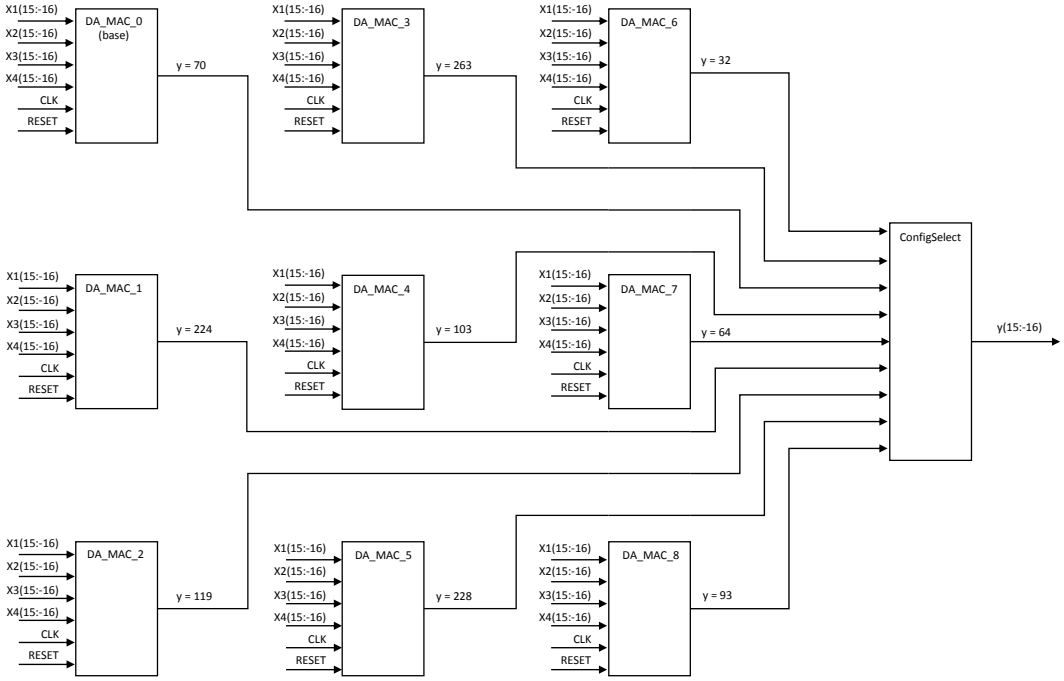


Figure 6.6: Architecture of the parallel static multiply-accumulate

6.3.2 Implementation 2: Configuration swapping with on-line FPGA tool flow

The configurations required for configuration swapping is usually generated off-line and stored in a database. These configurations can either be generated according to the module-based or difference-based reconfiguration design flow.

For the module-based case, an entire module is reconfigured. In order to implement such a system, a rectangular area on the FPGA has to be selected and set up to be reconfigured. The area encapsulated by the rectangle has to be large enough to include the entire MAC, or the sub-components thereof that need to be reconfigured. Module-based reconfiguration is well suited for reconfigurable applications requiring an architectural change. However, since only the constants of the MAC are changed in this implementation, this approach yields unnecessary large configuration files.

For this reason, difference-based reconfiguration [124, 125], was rather used. It is more suited towards reconfiguring applications wherein small changes have to be made on-the-fly. The general idea is to start with a base configuration describing the design to be reconfigured. Next, small changes are made to this configuration and saved as a new design. By comparing both designs, a configuration file is generated based on the differences between the two using the following command: *bitgen -g ActiveReconfig:Yes -g Persist:yes -r* ⟨*original.bit*⟩ ⟨*new.ncd*⟩ ⟨*new.bit*⟩. The *Persist:yes* command is required for partial reconfiguration via the SelectMAP

interface. However, since the ICAP was used for reconfiguration, the SelectMAP port is free for general purpose input/output and this command may be omitted.

Configuration swapping allows the device, or portions of the device, to be reconfigured by effectively swapping between configurations stored in memory. Due to the relatively large size of the configurations (especially when using module-base reconfiguration), external memory is most commonly used. For the research presented in this thesis, nine different difference-based configurations were generated and stored on CompactFlash. The configuration required is then selected using a DIP switch and the reconfiguration process triggered by a push button. Reading these configurations and transferring them to the configuration memory via the ICAP, is facilitated by an embedded PowerPC®. Once the reconfiguration completes, a pin, *ReconfigDone*, is asserted. A UART is also included for debugging purposes and the XPS timer is used to measure execution time on the PowerPC®.

The interconnectivity between these components are shown in Figure 6.7. A total of 2008 LUTs are needed to implement this design, unless a MicroBlaze™ processor is used in lieu of the PowerPC®. Implementing the exact same design on a soft-core processor requires around 600 additional LUTs. However, since the hard-core PowerPC® is available on the Virtex®-5 family of FPGAs, the assumption is made throughout this thesis that this processor is rather used.

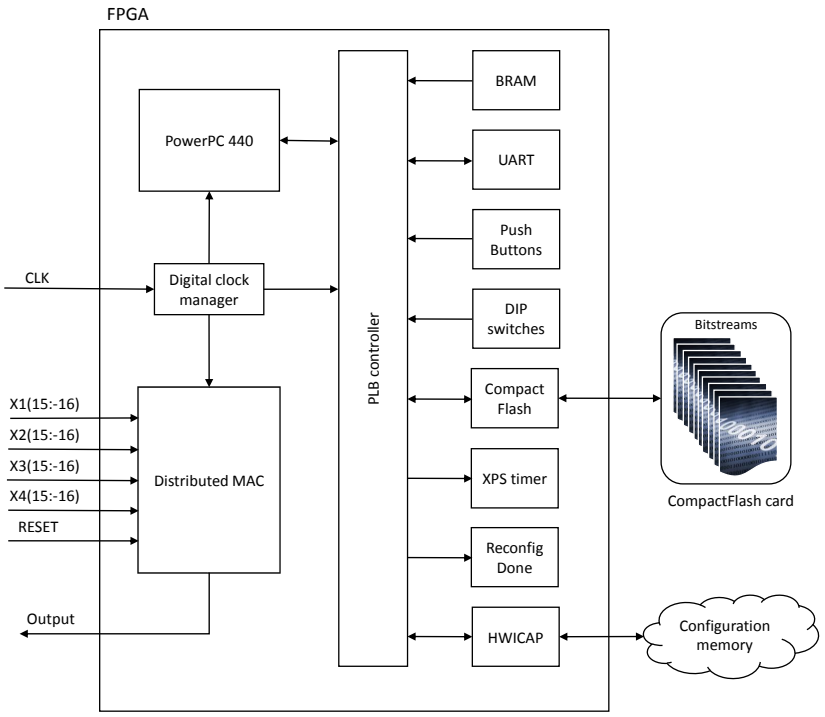


Figure 6.7: Architecture of the distributed MAC reconfigured using configuration swapping

As already mentioned, configuration swapping only works if the number of configurations is relatively small. For applications requiring more configurations, this implies that only a subset can be stored. The solution is then to generate these configurations at run-time. This is mostly done using the conventional FPGA tool flow and leads to a configuration of very high quality, since all the optimization options are available.

However, running the conventional FPGA tool flow is computationally expensive and can only be done for applications with extremely slowly changing parameters. For example, running the Xilinx® tool flow on an Intel® Core™ i7 M620 clocked at 2.67 GHz, with 4.00 GB RAM and hyper-threading enabled in the toolset, requires an average of 521 seconds to generate the full configuration for the architecture shown in Figure 6.7. Fortunately, difference-based reconfiguration only requires the differences between LUTs to generate a new configuration and takes around 81 seconds to complete. To further decrease this time, the specialiser designed in Chapter 5 can be used to generate a new configuration. This is discussed in the next section.

6.3.3 Implementation 3: Configuration swapping with software specialiser

Adding the specialiser to *Implementation 2* discussed in the preceding section, results in the architecture shown in Figure 6.8. While these two architectures are identical in every other way, the specialiser allows a new configuration to be generated on-the-fly. By storing the base configuration in CompactFlash, allows this specialisation to take place with the least amount of overhead. Since the specialiser is purely software-based, the hardware utilization of this design is similar to the one described in the previous section. The reconfiguration time is also similar, but the specialiser drastically reduces the time required to create new hardware by specialising the configuration.

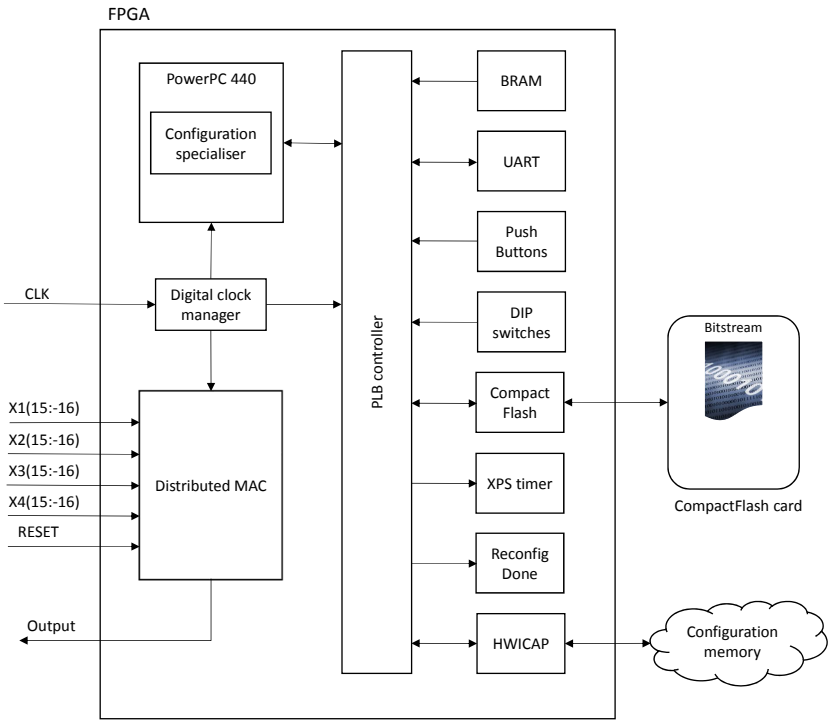


Figure 6.8: Architecture of the distributed MAC reconfigured using configuration swapping and added specialiser

6.3.4 Implementation 4: CLB bit toggle reconfiguration

Toggling the bits in the CLBs were done using Xilinx® predefined functions *XHwIcap_SetClbBits* and *XHwIcap_GetClbBits*, defined by constructs given in Listing 6.2, and the parameters listed in Table 6.4.

Table 6.4: Description of the parameters supplied to functions used to toggle CLB bits

Parameter	Description
Row	The row containing the CLB. The origin, (1,1), is the upper left CLB
Col	Refers to the column containing the CLB.
Resource	The target bits to be toggled
Value	The required value of the targeted bits
NumBits	The number of bits addressed.

Listing 6.2: Declaration of functions used to toggle the bits of a CLB

```
int XHwIcap_GetClbBits
(
        XHwIcap    *InstancePtr,
        long        Row,
        long        Col,
        const u8    Resource[][2],
        u8          Value[],
        long        NumBits
)
```

```
int XHwIcap_SetClbBits
(
        XHwIcap    *InstancePtr,
        long        Row,
        long        Col,
        const u8    Resource[][2],
        u8          Value[],
        long        NumBits
)
```

Unfortunately, there were some issues in using theses functions. The first was the difference in coordinate systems being used by the functions and the way the CLBs are physically numbered on the device. CLB (1,1) is physically located at the bottom left corner of the device, whereas (1,1) points to the top left corner when using *XHwIcap_SetClbBits* and *XHwIcap_GetClbBits*. The second issue seemed to originate from the fact that most of the code was reused from the Virtex®-4 FPGA, resulting in *XHI_CLB_LUT* containing the wrong slice information. The solution was to use a replacement header file[1] containing the correct information. The primary advantage of using these functions is removing the need for external memory and a configuration specialiser. Interesting to note is that this did not affect the area utilisation as much as expected. The resulting architecture requires 1562 LUTs to implement and is shown in Figure 6.9.

6.3.5 Implementation 5: Shift register lookup table (SRL) reconfiguration

An efficient way to change the functionality of the LUTs is to utilise the shift register capabilities of the LUTs [38, 126, 127]. In this architecture, the truth table configuration bits are arranged as a shift register and can be changed by shifting a new truth table into the SRL. This has the advantage that reconfiguration is no longer performed via the ICAP and allows each LUT to be addressed and reconfigured individually, instead of an entire frame. The result is a significant

[1] Supplied by Karel Heyse from Gent University in Belgium.

improvement in reconfiguration time with no unnecessary overhead. An added benefit is that this reconfiguration method also allows multiple LUTs to be reconfigured in parallel.

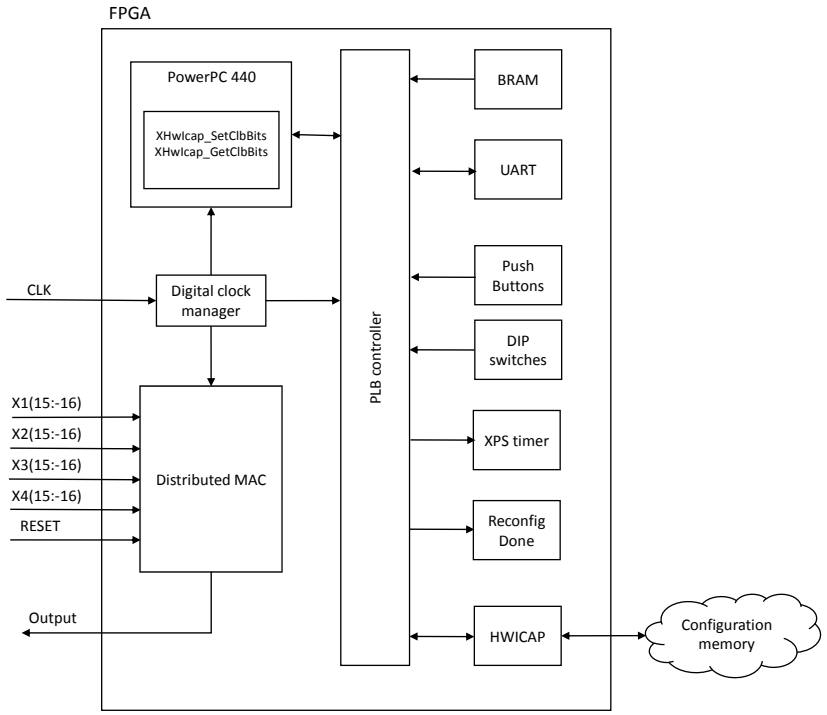


Figure 6.9: Architecture of the distributed MAC reconfigured using Set/GetClbBits

To implement the SRL reconfiguration, the RAM32X8S instantiated inside the distributed MAC module had to be replaced with eight SRL32E constructs—one for each *INIT*-parameter. A configuration bus was then added to the input of the MAC and each line connected to a shift register. When a new configuration is required, the specific set of parameters is selected by *ConfigSelect* and each parameter shifted into a register connected to the *D* port, as shown in Figure 6.10.

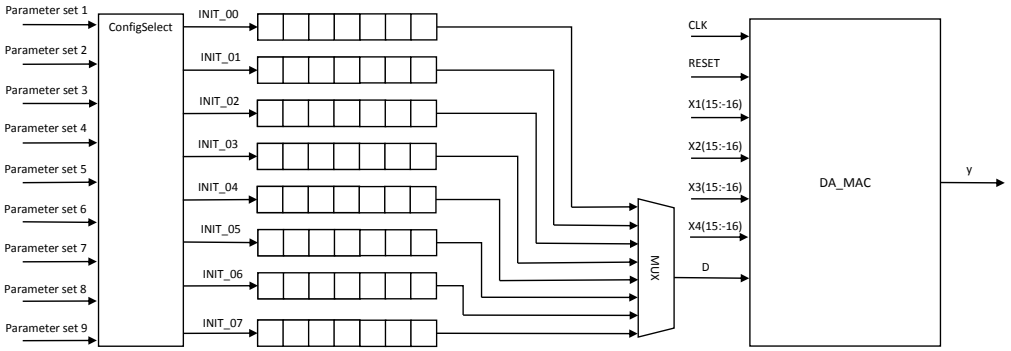


Figure 6.10: Architecture of the distributed MAC reconfigured using SRLs

The other reconfiguration implementations discussed in this chapter all require a new configuration to be generated when changing the hardware. However, this is not the case for SRL

reconfiguration. In this implementation, the new contents of the LUTs are usually stored in local memory from where it is loaded into the shift registers. To allow this implementation to be comparable to the other implementations, it is assumed that the Xilinx® tool-chain is used to generate the new content for the LUTs. Even though this negatively affects functional density, this implementation was only included to acknowledge SRLs as a legitimate means to adapt the contents of LUT, and to illustrate the maximum functional density advantage of this reconfiguration.

6.3.6 Implementation 6: Hardware-based reconfiguration

Combining the hardware controlled reconfiguration controller (discussed in Chapter 3), bitstream specialiser (designed in Chapter 5) and distributed MAC, yields the high-level architecture shown in Figure 6.11. When new MAC constants are required, the specialisation process is triggered and new LUT values calculated. These new values are then sent to the *Reconfiguration controller*, which is responsible for injecting these values into the bitstream read from the block RAM and transferring it to the configuration memory via the ICAP. The primary advantage of this architecture is that not only is it possible to reconfigure with the least amount of overhead, but it is also possible to generate configurations for any number of constants.

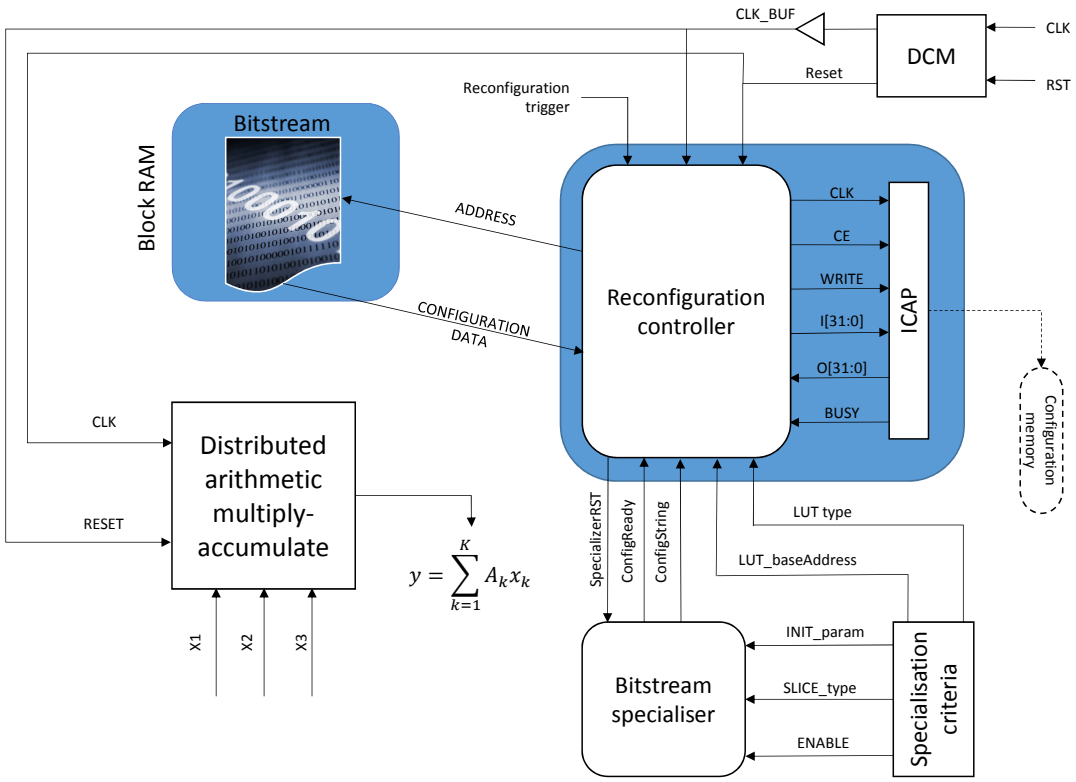


Figure 6.11: Architecture of the MAC reconfigured with hardware-based reconfiguration

6.4 Verification and validation

This section verifies and validates the six reconfigurable implementations by implementing each design on the ML507 development board and measuring the specialisation and reconfiguration time.

6.4.1 Implementation 1: Generic design

The execution time of the static design can be easily calculated from the number of clock cycles it takes to complete a single MAC-instruction and the clock period. From Figure 6.5 it was seen that 35 clock cycles are required. Clocking the application at 100 MHz, yields an execution time, T_e, of 350 ns which is fixed throughout all the reconfigurable implementations (*Implementation 1* to *6*) due to the architecture of the MAC not being reconfigured.

As previously mentioned, the stationary design requires a total of 813 LUTs to be implemented. This yields a functional density of:

$$D_{static} = \frac{1}{AT_e} = \frac{1}{813(350 \times 10^{-9})} = 3514 \text{ operations/s} \tag{6.8}$$

which is the base functional density all other implementations will be compared to. In each of these designs, the generation and/or configuration times were measured using an oscilloscope. This was done by measuring the time between initiating the reconfiguration process (using an external push button) and when it finishes. The duration is indicated by an LED that lights up when the reconfiguration starts and turns off when it finishes. The correctness of the reconfiguration is verified by also lighting specific LEDs based on the output of the MAC. For designs with an embedded PowerPC®, XPS Timer was used to verify the measured values and the average value used in the functional density calculation. The results of these measurements are listed in Figure 6.18 and the functional densities compared in Section 6.4.7.

6.4.2 Implementation 2: Configuration swapping and on-line FPGA tool flow

An excerpt from the module level utilisation required to implement the configuration swapping is listed in Table 6.5. The A/B notation used in each column refers to the number of elements that belong to that specific hierarchical module (A) and the total number of elements from that hierarchical module and any lower level hierarchical modules below (B). Also note that slices can be packed with elements from multiple hierarchies. As a result, only the maximum number of slices (B) from the top level initialisation is considered when calculating the functional density. The rest of the modules and their utilisation are only given as reference.

Difference-based reconfiguration was used to generate the multiple configurations, with each requiring around 81 seconds to generate. Since the conventional tool flow was used, this measurement was done using a command line batch tool, called *timecmd*. The design rule checker (DRC) was also disabled to obtain the best possible time for generating new hardware.

Table 6.5: Hardware requirements to implement configuration swapping

Module name	**Slices**	**Slice Reg**	**LUTs**
Top_main	12/1786	0/2359	19/2008
Inst_system	0/1738	0/2248	0/1914
DIP_Switches_8Bit	0/75	0/122	0/63
FLASH	0/344	0/448	0/323
Push_Buttons_5Bit	0/73	0/101	0/54
RS232_Uart_1	0/114	0/143	0/129
ReconfigDone	0/66	0/80	0/45
CompactFlash	0/105	0/207	0/97
clock_generator_0	0/0	0/0	0/0
jtagppc_cntlr_inst	0/0	0/0	0/0
plb_v46_0	0/198	0/110	0/235
ppc440_0	0/0	0/0	0/0
proc_sys_reset_0	0/65	0/69	0/52
xps_bram_if_cntlr_1	0/175	0/253	0/200
xps_hwicap_0	0/523	0/715	0/716
inst_DA_MAC	36/36	111/111	75/75
Total	**1786**	**1911**	**1685**

The reconfiguration time was measured using XPS timer to be 544.970 ns, and verified using the oscilloscope's results shown in Figure 6.12. Using these times, this implementation's functional density was calculated and compared to those of the other implementations. As mentioned above, the results are discussed in Section 6.4.7.

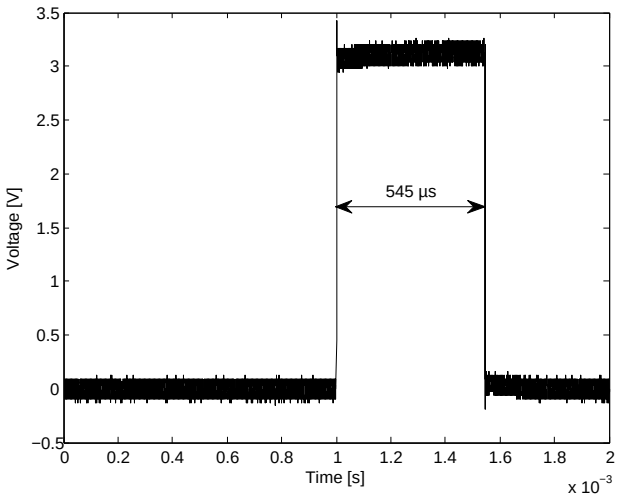


Figure 6.12: Oscilloscope measured reconfiguration response of the configuration swapped design

6.4.3 Implementation 3: Configuration swapping with software specialiser

Adding the specialiser to the configuration swapped design of adds no additional hardware overhead since it was implemented in software. The slight difference in hardware is due to the previous design requiring a CompactFlash interface. The design with the software specialiser requires 1589 LUTs to be implemented, with the specialisation and reconfiguration time measured to be 398 µs and 1.079 ms respectively—as shown in Figure 6.13.

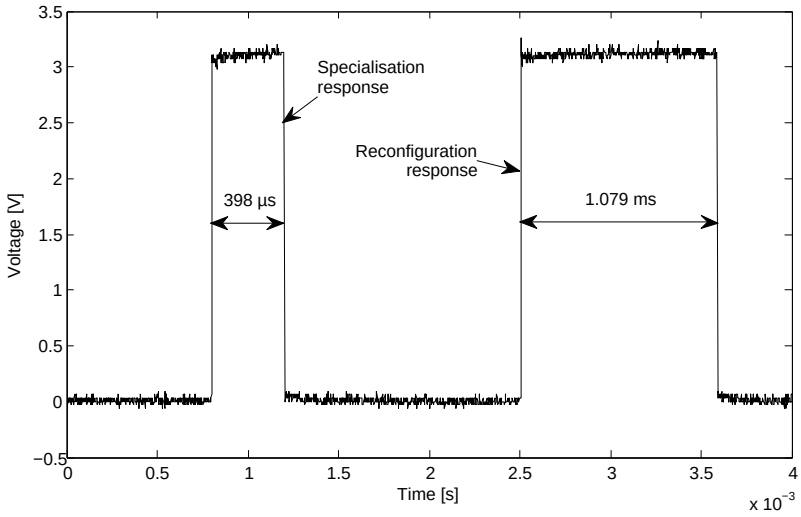


Figure 6.13: Oscilloscope measured reconfiguration response of the configuration swapped design with specialiser

6.4.4 Implementation 4: CLB bit toggle reconfiguration

The list of hardware required to implement the CLB bit toggle reconfiguration is shown in Table 6.6. Interesting to note is that roughly the same number of resources are required to implement this architecture as in the configuration swapping designs. The reason for this being that the hardware underlying the three designs are roughly the same. The major changes occur in software only, which does not affect the underlying hardware.

Since the FPGA is reconfigured by simply toggling the bits in the CLB, no configuration specialisation is required. Instead, specialisation is done by generating a new .elf-file containing the software for the PowerPC® and incorporating it into the configuration by using BitGen™ (or Data2mem). Timing the command was again done by using *timecmd*. It was found that roughly 78 seconds are required to finish this process. The reconfiguration takes around 7.26 µs to complete, as shown in Figure 6.14. Using these times, the functional density of this implementation was calculated and compared to those of the other implementations.

Table 6.6: Hardware requirements to implement the CLB bit toggle reconfiguration

Module name	Slices	Slice Reg	LUTs
Top_main	12/1349	0/1710	19/1562
Inst_system	0/1297	0/1599	0/1468
DIP_Switches_8Bit	0/74	0/122	0/63
Push_Buttons_5Bit	0/74	0/101	0/54
RS232_Uart_1	0/116	0/143	0/129
ReconfigDone	0/67	0/80	0/45
clock_generator_0	0/7	0/8	0/3
jtagppc_cntlr_inst	0/0	0/0	0/0
plb_v46_0	0/190	0/108	0/206
ppc440_0	0/0	0/0	0/0
proc_sys_reset_0	0/66	0/69	0/52
xps_bram_if_cntlr_1	0/161	0/253	0/201
xps_hwicap_0	0/542	0/715	0/715
inst_DA_MAC	40/40	111/111	75/75
Total	**1349**	**1710**	**1562**

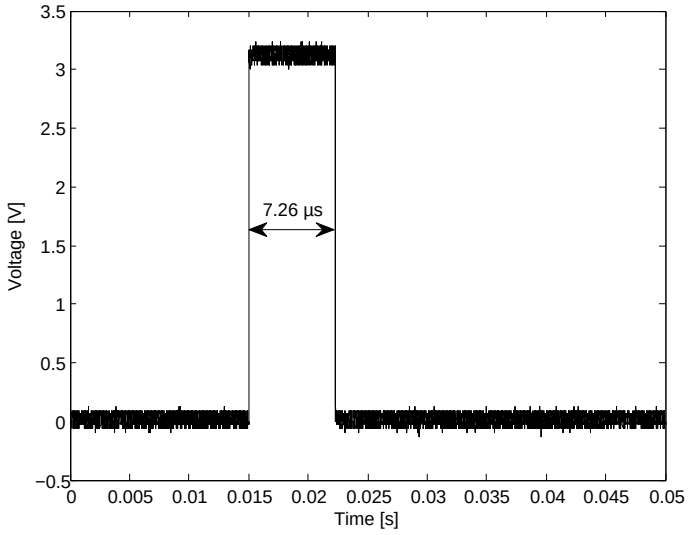


Figure 6.14: Oscilloscope measured reconfiguration response of the CLB bit toggle functions

6.4.5 Implementation 5: Shift register lookup table (SRL) reconfiguration

The SRL reconfiguration design requires 177 slices, 332 slice registers and 279 LUTs to implement. Generating a configuration for this implementation takes around 22 seconds if the difference-based reconfiguration design flow is followed. Instead of using the Xilinx® tool-chain to generate new configurations for this implementation, novel methods (such as Bruneel's TLUT [12]) can be used to mitigate the cost of creating a new configuration by creating LUT configurations without having to generate a bitstream. However, to keep the results comparable to the reconfiguration method proposed in this thesis, it is assumed that a complete bitstream

has to be generated.

Calculating the time required to configure the device is done by taking the length of the SRL registers and multiplying it with the clock period. The SRLs used in this application have 32 clock cycle shift registers. At a clock frequency of 100 MHz, 320 ns is required to shift the new values into the SRLs. This is easily verifiable using an oscilloscope. The measured reconfiguration response is shown in Figure 6.15.

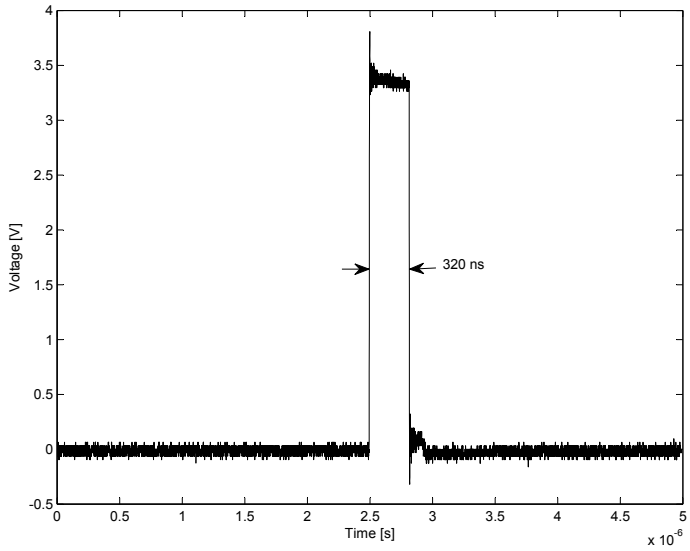


Figure 6.15: Oscilloscope measured reconfiguration response of the SRL reconfiguration method

Despite the tremendous functional density advantage this reconfiguration method can achieve, it has one major drawback: it is extremely device dependent. It is limited to Xilinx® FPGAs and only to families incorporating this feature. Even then, not all the LUTs exhibits this features. Initially, all the LUTs on a Virtex®-II FPGA were capable of being used as SRLs, but this percentage has dropped significantly for latter FPGA families. For the Virtex®-5 FPGA used for analysis in this thesis, only around 25% of LUTs exhibits this functionality. As a result, this implementation cannot be used for reconfiguring all FPGAs. *Implementation 6* aims to provide a similar functional density advantage without being device dependent.

6.4.6 Implementation 6: Hardware-based reconfiguration

This design requires six specialisers to be implemented in parallel—one for each *INIT* parameter. The result is a slightly larger area requirement than the SRL implementation, as shown in Table 6.7, but with the advantage of significantly reducing the time required to generate new hardware. In fact, simulation results show that the specialised values for each *INIT* parameter is available immediately. Unfortunately, the hardware has physical limitations—such as gate delays and set-up and hold times—contributing to the time required to generate new hardware. As a conservative estimate, the actual time was measured at the 50% rise time between the signal requesting a hardware change and the moment the specialisation was completed. This is

Table 6.7: Hardware requirements of the hardware controlled reconfiguration and specialiser

Module name	**Slices**	**Slice Reg**	**LUTs**
Top_main	15/233	0/245	22/362
Gen_control[0].inst_spec	7/7	0/0	7/7
Gen_control[1].inst_spec	7/7	0/0	7/7
Gen_control[2].inst_spec	6/6	0/0	6/6
Gen_control[3].inst_spec	6/6	0/0	6/6
Gen_control[4].inst_spec	1/1	0/0	1/1
Gen_control[5].inst_spec	5/5	0/0	5/5
Top_main	1/1	0/0	1/1
inst_DA_MAC	60/60	111/111	91/91
inst_ReconMon	13/13	25/25	29/29
inst_contrl	104/104	86/86	165/165
inst_debounce	8/8	23/23	22/22
Total	**233**	**245**	**362**

shown in Figure 6.16 to be around 2.1 ns, but this is extremely dependent on the placement and routing of the individual hardware components.

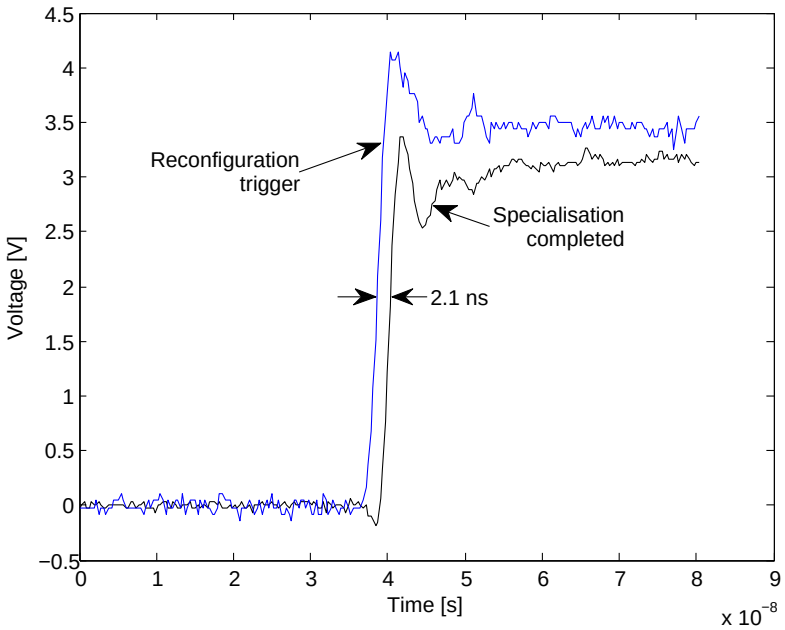
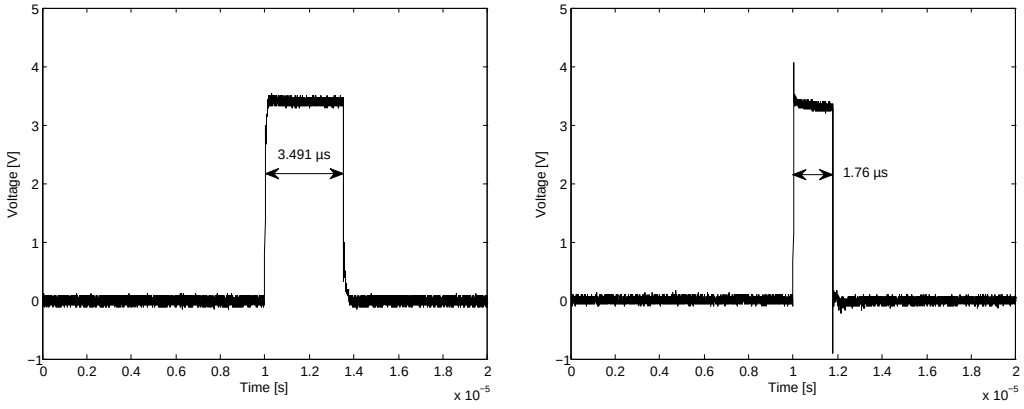


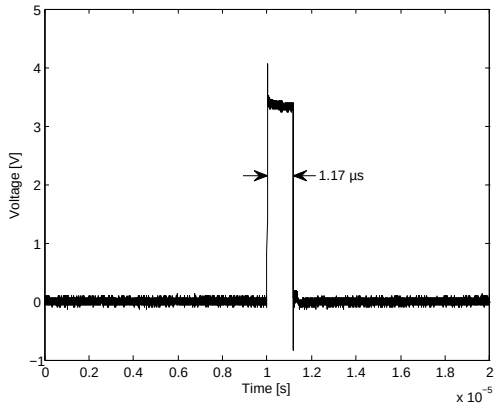
Figure 6.16: Oscilloscope measured specialisation response of the hardware-based reconfiguration

Since this design uses the hardware ICAP—instead of the HWICAP driver—more control can be exercised in terms of timing. As a result, it is possible to increase the clock frequency beyond the Xilinx®-recommended 100 MHz. The Xilinx® tools do not have the necessary timing information to make the optimal choices for instantiating the ICAP. This implies that the maximum clock frequency of the ICAP is a function of the source and sink placement. If they are placed close



(a) 100 MHz

(b) 200 MHz



(c) 300 MHz

Figure 6.17: Oscilloscope measured reconfiguration response of the hardware-based reconfiguration

to each other, the clock frequency may increase to be higher than the recommended frequency of 100 MHz for a Virtex®-5 FPGA.

However, to keep the amount of manual design to a minimum, the ICAP was just constrained using the *MAXDELAY* constraint. The ICAP clock was then gradually increased using a digital clock manager (DCM) until the system stopped functioning. The reconfiguration response for the system is shown in Figure 6.17(a) for 100 MHz, Figure 6.17(b) for 200 MHz and Figure 6.17(c) for 300 MHz.

Verifying these values can easily be done by considering the size of the configuration file and the clock frequency of the ICAP. The configuration file contains 11,072 bits. The word width of the Virtex®'s ICAP is 32-bits, requiring 346 clock cycles to process the configuration file. At a clock frequency of 100 MHz, this relates to 3.46 µs—corresponding to the response displayed in Figure 6.17(a). The same calculations can be performed at 200 and 300 MHz respectively. Using these values, combined with the measured specialisation time, the functional density of this implementation can be calculated and compared to those of the other implementations. This is discussed in the subsequent section.

6.4.7 Functional density comparison

The previous sections discussed several different implementations most commonly used to reconfigure an application. All of these designs were used to implement and reconfigure a distributed MAC. In each case the total number of LUTs used to implement the design were determined and the reconfiguration time measured. This includes the time required to generate new hardware, as well as the time required to configure the device. Using this information, the functional density can be calculated (refer to Section 1.3). At this stage, the worst case is considered and the assumption made that reconfiguration is required after every execution, i.e. $n = 1$ with zero slack available (refer to Figure 1.2). The results are summarized in Table 6.8.

Referring back to Section 1.3, the minimum number of times hardware should be reused before reconfiguration becomes feasible is referred to as the break-even point, defined by:

$$n = \frac{A_r(T_{conf} + T_{gen})}{A_s T_{s,exec} - A_r T_{r,exec}}. \tag{6.9}$$

By plotting the functional density of each design as a function of the average number of executions between hardware changes, the graphs in Figure 6.18 are obtained. It is worth noting that each graph in the figure is relative. By implementing more hardware in parallel for the static design, the functional density will decrease due to an increase in area. The same applies to the reconfigurable designs. In this particular case, only the constants of the MAC are reconfigured. If the reconfiguration processes are adapted to change the architecture of the MAC as well, the area would again influence the functional density of each design.

As can be seen from the figure, dynamic reconfiguration is only beneficial for functional density when the hardware changes occur less frequently. Due to fast execution of the MAC, only the hardware-based and SRL reconfiguration yield an improvement over their stationary counterpart. The overhead from the other techniques are simply too large to improve the functional density. However, the static implementation is bound by the nine datapaths currently implemented. If for example another nine datapaths are required, the area would increase to about 1,600 LUTs—halving the functional density. Only then will reconfiguration using conventional methods yield any functional density benefit.

As mentioned in Section 6.3.5, it is assumed that the Xilinx® tool-chain is used to generate new LUT contents for the SRLs. As can be seen from Table 6.8 and Figure 6.18, this adds significant overhead to the SRL reconfiguration process (21.16 s). Alternative means are proposed in the literature for generating new LUT contents without resorting to bitstream generation. An example is Bruneel's TLUT method [12], which is capable of generating new LUT contents for SRL reconfiguration significantly faster than the Xilinx® toolset. Bruneel showed that a 32-tap finite impulse response (FIR) filter can be reconfigured in 2.6 µs using SRL reconfiguration, which includes the time to generate new LUT contents. This yields a significant improvement in functional density. For this reason, SRL reconfiguration was added to the functional density plots to acknowledge that this is a valid reconfiguration method that would produce a high functional density. As mentioned, a drawback of this reconfiguration method is that it is device dependent and only constrained to some LUTs. In addition, it only allows the contents of LUTs

to be adapted. While this is also currently the case for the reconfiguration method proposed in this thesis (*Implementation 6*), this method will hopefully be expanded in the future to also include the specialisation of routing bits.

Table 6.8: Functional density for each of the designs

Nr.	Implementation	Area (A) [#LUTs]	Execution time (T_e)	Generation time (T_{gen})	Configuration time (T_{conf})	Functional density (D)[1]
1	Generic (Static) multiply-accumulate	813	350 ns	0	0	3,514
2	Configuration swapping (online)	1,685	350 ns	80.6 s	544.97 µs	7.36×10^{-6}
3	Configuration swapping (with software specialiser)	1,589	350 ns	398.00 µs	1.08 ms	425.86×10^{-3}
4	CLB bit toggle reconfiguration	1,562	350 ns	78.22 s	7.26 ms	8.18×10^{-6}
5	SRL reconfiguration	177	350 ns	21.16 s	320 ns	267×10^{-6}
6	Dynamic reconfiguration (100 MHz)	362	350 ns	2.10 ns	3.49 µs	718.728
6.1	Dynamic reconfiguration (200 MHz)	362	350 ns	2.10 ns	1.76 µs	1,311
6.2	Dynamic reconfiguration (300 MHz)	362	350 ns	2.10 ns	1.17 µs	1,814

[1] In operations per second per unit hardware resources

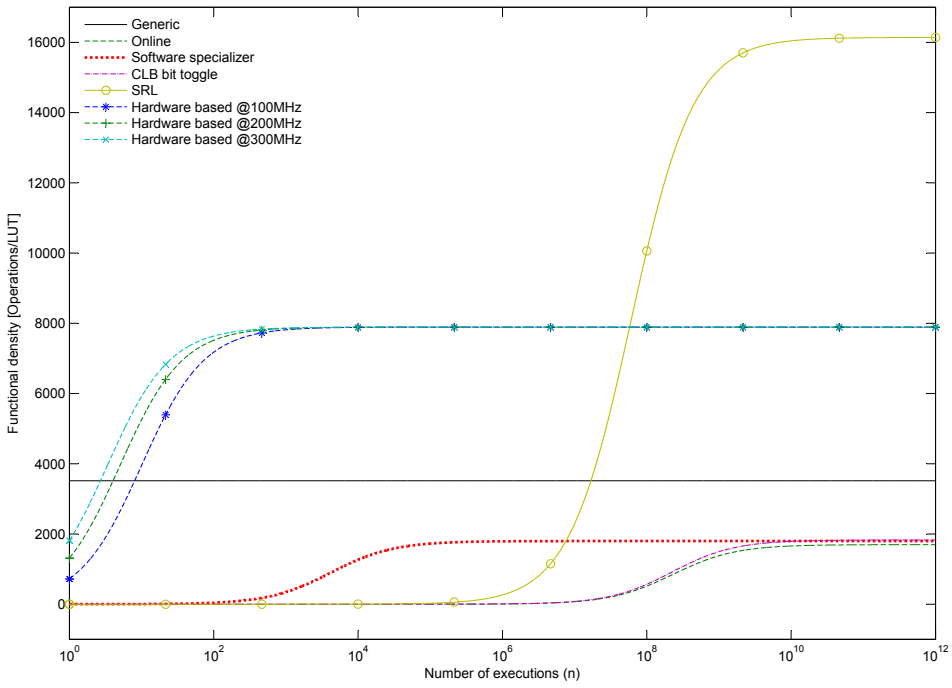


Figure 6.18: Illustration of functional density as a function of the number of executions

The most important thing to note from Figure 6.18 is the significant improvement the hardware-based reconfiguration yields over the other techniques. The only other technique yielding any benefit is the SRL reconfiguration and in its current implementation, only if the MAC is changed once every 16.81 million executions. The hardware-based reconfiguration, on the other hand, only requires the hardware to be changed once every eight execution cycles when clocked at 100

MHz. However, this can be reduced even further by overclocking the design. At 200 MHz only four execution cycles are required and at 300 MHz this is reduced to 2.7.

6.5 Concluding remarks

This chapter showed how the configuration specialiser proposed in Chapter 5 can be combined with the hardware-based reconfiguration discussed in Chapter 3 to reconfigure an application. The application selected was a multiply-accumulate implemented using distributed arithmetic, which uses LUTs to perform multiplication operations. The reason for selecting this particular application is because the sum-of-products prominently features in many functions. More important is the way the MAC was implemented. Distributed arithmetic allows any sum-of-products to be implemented using lookup table based schemes. Since the bitstream specialiser is capable of adapting the contents of any LUT on the FPGA, using distributed arithmetic greatly expands the areas where the specialiser can be used.

To showcase the advantage of using the proposed specialiser and hardware controlled reconfiguration in lieu of other techniques, the MAC was reconfigured using various reconfiguration implementations and compared in terms of their functional density. As can be expected, a generic implementation (*Implementation 1*) yielded the highest functional density, since no additional overhead is required to switch between datapaths. It did, however, require a large area to implement each path in parallel.

The area requirement of the three implementations utilising an embedded processor to facilitate the reconfiguration process (*Implementation 2* to *4*), is significantly larger than the other implementations, which has an adverse effect on functional density. This area is highly dependent on the type of processor used and the peripherals attached to it. Even if this area is neglected when calculating the functional density, either the time required to generate new hardware, or the reconfiguration time is too large to make a significant positive contribution to the functional density.

Even though the SRL implementation (*Implementation 5*) starts off with a much lower functional density than its generic counterpart, by reusing the hardware over multiple executions, the overhead from this process can be amortised to such an extend that its functional density exceeds those of other reconfiguration methods. As with the other conventional implementations, this means that this reconfiguration is only suitable for quasi-static applications. An added disadvantage is that this implementation is strictly device dependent.

The proposed reconfiguration method (*Implementation 6*), on the other hand, yielded a significant improvement compared to the other methods. Even though its functional density is still lower than that of the static application, it is amortised much quicker by re-using the hardware than any of the other implementations, even if the processor area is neglected. In fact, using this reconfiguration method could improve the functional density of applications with dynamic behaviour. Its reduced reconfiguration and specialisation time (when compared to the other implementations) also implies that this process could be applied to real-time applications with strict time constraints. This is addressed in the next chapter.

CHAPTER 7

CONCLUSIONS AND RECOMMENDATIONS

"There's no sense crying over every mistake. You just keep on trying till you run out of cake. And the science gets done."

— GLaDOS, *Portal*

This last chapter ties the different results and conclusions of the thesis together. It discusses each contribution and highlights its relevance in regards to the research as a whole. Recommendations are also made for further investigations on this topic.

7.1 Summary of research

The primary aim of the research presented in this thesis was to migrate dynamic reconfiguration from quasi-static applications to the realm of real-time applications. This was done by not only proposing an optimal architecture for reconfiguring real-time applications, but also by proposing a method to specialise the LUTs in an FPGA bitstream to represent any other configuration.

A summary of the research presented in this thesis is given in Figure 7.1. From the literature it was evident that dynamic reconfiguration is only suitable for quasi-static applications, due to the large overhead involved in the process. One of the primary contributors of this overhead is the bus-based controller most commonly used in the reconfiguration process. Despite various research being capable of maximizing the throughput of these types of systems, most of these methods require additional clock cycles to pre- or post process the configuration data before transferring it to the configuration memory. For these reasons, the most promising architecture identified from the literature for real-time control was the hardware controlled reconfiguration architecture, which stores the configuration data locally in BRAM. Since this BRAM is tightly coupled to the configuration controller, this architecture mitigates the overhead introduced by the bus in conventional reconfiguration architectures. Another advantage of this architecture

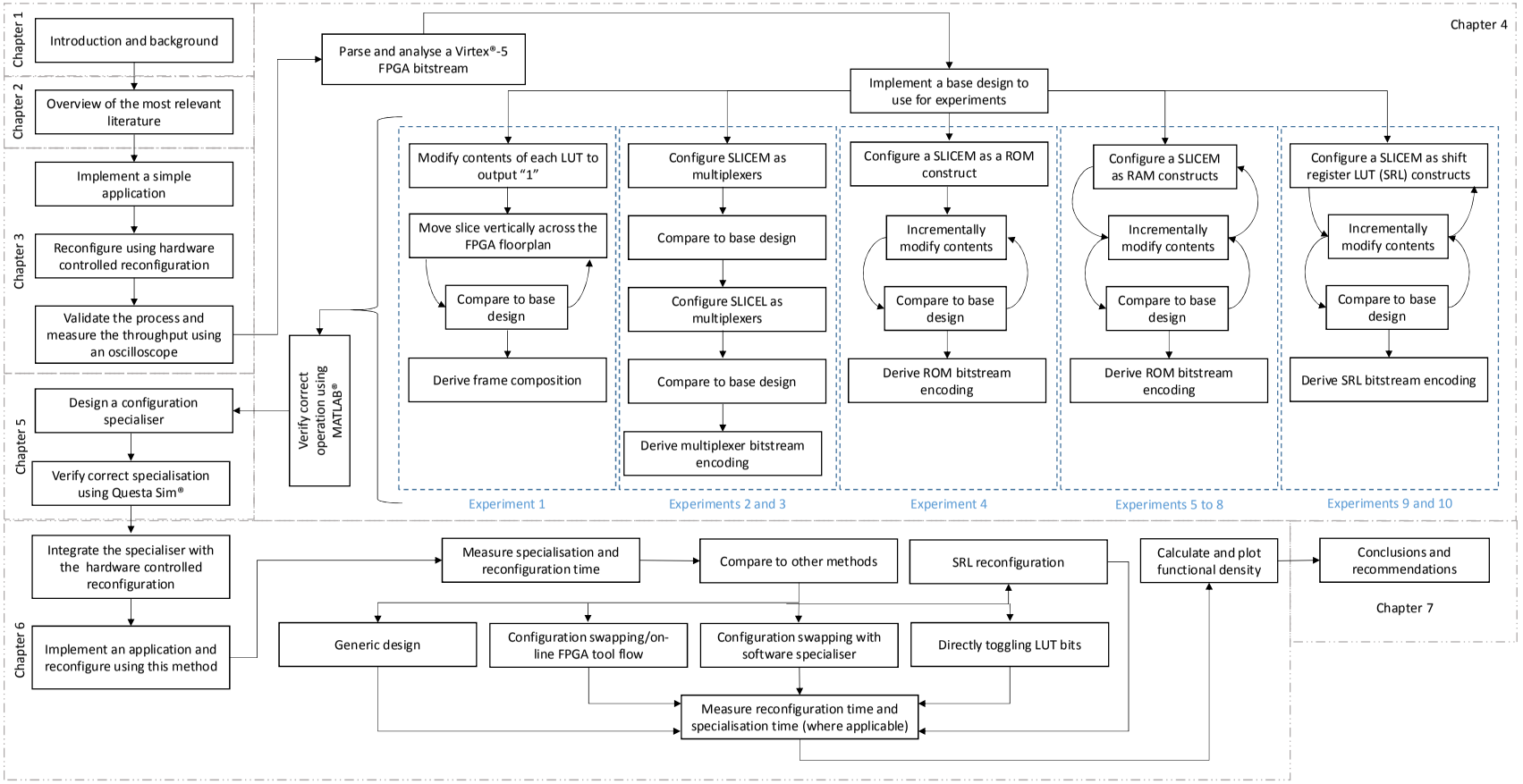


Figure 7.1: Flowchart summarising the research presented in this thesis, along with the chapter breakdown

is that it is easy to increase the operational clock frequency, since all the hardware (with the exception of the ICAP) is hand designed.

The drawback of using this architecture is that the BRAM is considered an expensive resource and because it is extremely limited, only a subset of configurations can be stored. Even though compression techniques can be used, the time added by these algorithms make them unsuitable for real-time control. The solution was then to modify the configurations at bit-level. Despite extensive research in this area, most bitstream manipulation techniques rely on a layer of abstraction to safely alter the FPGA resources. Even though some methods came close to allowing real-time manipulation of the resources, these projects were discontinued before being thoroughly completed. As an alternative, methods were investigated wherein the bits of the bitstream are directly manipulated. Unfortunately, most of the literature only focused on the older generation FPGAs (Virtex®-II and 4).

As a result, a method had to be found to parse and analyse the bitstream of an FPGA. This was done through a series of ten experiments used to determine the frame composition along with the encoding of the configuration space to the bitstream. It was shown that the configuration of any LUT construct can be derived by considering the Boolean equation represented by the LUT and by inserting a unique set of configuration strings into this equation. The Boolean equation represented by each LUT construct can easily be obtained from its corresponding truth table.

Using this knowledge, a bitstream specialiser was designed and implemented using a set of parallel, passive logic, representing the generalised truth table of an LUT construct. The output of this specialiser is a set of configuration strings for each LUT in a construct. This allows the contents of any LUT to be changed according to a set of specialisation criteria by injecting these strings into the necessary locations in the bitstreams during reconfiguration. Even though this specialiser is device independent, it is application dependent and some of the analysis presented in Chapter 4 has to be repeated for other devices. This reconfiguration technique was then compared to alternative techniques, in each case measuring the reconfiguration and specialisation time (if applicable). The limitations of the proposed method, the results obtained and the conclusions drawn are discussed in the subsequent sections.

7.2 Discussion on functional density

Relating this body of work back to functional density discussed in Section 1.3, the aim was to reduce T_{conf} and T_{gen} in (1.5). Comparing these metrics against those of alternative reconfiguration techniques, it was shown that the hardware controlled reconfiguration combined with the proposed bitstream specialiser yields a significant improvement. In fact, T_{gen} of the proposed method is orders of magnitude lower than those of conventional reconfiguration methods.

A reconfigurable implementation will always have an initial functional density lower than that of a static implementation. By reusing the hardware over multiple executions or clock cycles, the overhead of the reconfiguration process is amortised over each cycle. Moreover, a dynamic implementation is capable of improving the execution time of an application and reducing the

utilisation area by removing unneeded hardware. This implies that the dynamic application could have a higher functional density than its static equivalent if the hardware is reused multiple times.

Even though the functional density is highly dependent on the type of application and the specific hardware used, it was shown that the reconfiguration method proposed in this thesis requires significantly less hardware reuse before obtaining a functional density advantage above its static counterpart compared to other techniques. In this particular instance, dynamically reconfiguring a distributed MAC, the hardware controlled reconfiguration with bitstream specialiser only required eight cycles of hardware reuse before exceeding the functional density of the static application. Once this point is reached, the advantage of dynamic reconfiguration is evident in that the functional density keeps increasing until it is twice that of the static application.

Despite dynamic reconfiguration not being used to improve the critical path of the distributed MAC, it is possible to reduce the execution time between each reconfiguration. However, only changing the coefficients of the MAC was sufficient to showcase the operation and functional density advantage obtained by using the hardware controlled reconfiguration with integrated bitstream specialiser.

Also evident from this study is that the distributed MAC used to calculate the functional density is the perfect example of an application not typically reconfigured. As it was implemented for this study, none of the conventional techniques yielded any functional density advantage, regardless of the number of times the hardware was reused. In order to achieve a functional density advantage using conventional reconfiguration techniques, the number of datapaths in the static design had to double, effectively halving the functional density.

7.3 Parsing and analysing the bitstreams of newer devices

Although the work presented in this thesis was based on the parsing and analysis of a Xilinx® Virtex®-5 VFX70T FPGA, it is possible to extrapolate the parsing and analysis methodology to other, newer or older, FPGA families from Xilinx®. It should theoretically be possible to apply this methodology to Altera® FPGAs as well.

The main differences between the different FPGA families' configuration architecture are the frame length, the frame composition and the physical layout of hardware resources. Even though thousands of bitstreams were compared to derive the frame composition, this will not be necessary for newer devices. The experiments performed in Chapter 4 showed that the homogeneous nature of most FPGAs implied that only the differences between the different slice types have to be analysed to obtain the configuration strings. It was also found that despite the complexity of the LUT constructs, the bitstream encoding can always be derived from its truth table. This is a function of the LUTs, and will also be applicable to any other FPGA. This also holds true for FPGAs with 4-input LUTs, but the resulting configuration strings would now only be 16-bits wide. From these assumptions, the following generalised methodology can be derived to parse and analyse the bitstream for any FPGA:

1. Compare the bitstreams of two designs: one with all LUTs initialised to output '0' for all input combinations and one with all LUTs initialised to output '1'. This highlights the distribution of the LUTs in the frame.
2. Repeat the process above, but this time only initialise a single LUT to output '1'. Doing this for all the LUTs in a construct, highlights the bits in the bitstream configuring each of the individual LUTs.
3. Migrate the design to a slice of a different type and repeat the processes mentioned above. Ensure that all types of slices are analysed: SLICEL, SLICEM, and SLICEX. The result should be the same as obtained above, but this ensures that the frame composition is consistent between the different slices.
4. Lastly, the process is repeated for each LUT initialised as a multiplexer, which extracts the configuration strings. Since the configuration bits of the LUT5s[1] are a subset of the LUT6s[2], it is necessary to also repeat this process with the LUTs initialised as 5-input multiplexers. This is done by tying input *A6* low.

A Virtex®-5 configuration frame contains a 16-bit segment for each of the LUTs in a row 20 CLBs high, and four frames are required to fully reconfigure all the LUTs in a row. According to Xilinx®'s partial reconfiguration user guide [104], the 7 series FPGA family has a configuration frame base of 50 CLBs high by 1 CLB wide. It is thus estimated that each frame would again contain a 16-bit segment of the LUT configuration data, but this time for a row of 50 CLBs high. Similarly, for the Virtex®-6 FPGA family it is estimated that a 40 CLBs high configuration frame would contain the 16-bit configuration segments for all the LUTs in a row 40 CLBs high. For an FPGA family with 4-input LUTs, such as the Virtex®-4, it is estimated that each frame contains a 4-bit configuration segment for all the LUTs in a row. In the case of the Virtex®-4, these rows are 16 CLBs high.

Determining the frame address translation for any Xilinx® FPGA should be possible using the generalised formulae below—with the variables as used in Chapter 4:

$$Top(y) = \mathbb{1}_{\{y \geq y_{\max} \div 2\}}(y) \tag{7.1}$$

$$Row(y) = \left\lfloor \left| \frac{2y - y_{\max}}{x_{\max}} \right| \right\rfloor . \tag{7.2}$$

The generalised translation for the major address is given in (7.3). In this equation, each *col*-value refers to the start of each column containing CLBs. These values do not need to be in sequence, since other hardware components could also take up a column address.

$$Major(x) \quad = \quad \left\lfloor \frac{x}{2} \right\rfloor + 1 + \mathbb{1}_{\{x \geq \text{col1}\}}(x) + \mathbb{1}_{\{x \geq \text{col2}\}}(x) + \mathbb{1}_{\{x \geq \text{col3}\}}(x) + \ldots \tag{7.3}$$

[1] LUTs used to implement a 5-input Boolean function
[2] LUTs used to implement a 6-input Boolean function

Throughout the analysis done in Chapter 4, the translation of the minor address could be given by (7.4). Unfortunately, it was also found that the columns where the frames are swapped (i.e. the SLICEMs) are not necessarily uniformly distributed across the FPGA floorplan. For this reason, (7.5) has to be populated with the distribution of the SLICEMs.

$$Minor(x) = \begin{cases} \begin{cases} [32, 33, 34, 35] & (x\,even) \\ [26, 27, 28, 29] & (x\,odd) \end{cases} & if\ \overline{Swap}(x) \\ \begin{cases} [34, 35, 33, 32] & (x\,even) \\ [28, 29, 27, 26] & (x\,odd) \end{cases} & if\ Swap(x) \end{cases} \tag{7.4}$$

$$Swap(x) \equiv \text{(x-locations where the CLB is SLICEM)} \tag{7.5}$$

7.4 Reconfiguration of real-time applications

As was shown in Chapter 6, the proposed reconfiguration method requires eight cycles before obtaining a functional benefit over its static counterpart. Even though it is an adequate measure of determining the advantage of dynamically reconfiguring an application, this advantage is ameliorated over multiple executions of the application. In order for the proposed method to be truly usable in real-time applications, the reconfiguration and specialisation has to be complete by the time the new hardware is required.

As a use case, consider a typical real-time application, such as the high-speed, five degree-of-freedom active magnetic bearing system developed in [16]. The controller receives the position of a rotating rotor and applies proportional-integral-derivative (PID) control to keep it stable at a reference position. The current reference output of the PID controller is filtered, the data logged and transmitted to a power amplifier used to generate current in an electromagnetic coil. In this application, certain rotational speeds of the rotor excites critical frequencies, which could cause the system to become unstable. Consequently, different PID parameters are required from start-up, until the nominal rotational speed is reached.

Say for instance this application is reconfigured with new PID gains instead of reading them from memory. Even though dynamic reconfiguration allows the architecture of the controller to change during run-time, only the gains are adapted to stay with the theme of specialising LUTs. To consider the worst case reconfiguration, assume no slack is available and new parameters are required every control cycle of 50 µs. It was shown in [16] that 39 µs are required to implement the functionality mentioned in the previous paragraph. This only leaves 11 µs for the reconfiguration to complete.

It was found in Chapter 6 that the proposed specialiser and reconfiguration method require a maximum of 2.10 ns and 3.49 µs respectively to complete. It is thus evident that this reconfiguration and specialisation method can be used to reconfigure the application—with around 7.7 µs to spare. By increasing the clock frequency supplied to the ICAP, the time

remaining per control cycle can be increased even further. Figure 7.2 illustrates the timing of the controller if the reconfiguration is also added.

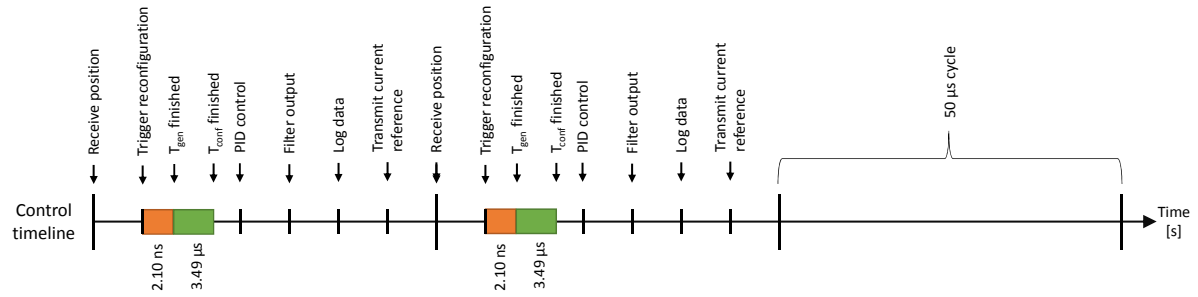


Figure 7.2: A timing diagram of a typical real-time system with reconfiguration overhead included

7.5 Future work

The following areas fir future work have been identified from the findings of this thesis:

Applying the bitstream parsing and analysis method to newer FPGA families
When this study was conducted, the only available platform was a Xilinx® ML507 development board that comes equipped with a Virtex®-5 VFX70T FPGA. Since then, a development board with a Spartan®-6 SLX9 was acquired, which will be used to investigate the proposed bitstream parsing and analysis method for newer FPGA families. The analysis of this specific device family will produce interesting results, since it has a variable configuration frame size [128].

Performing a similar parsing and analysis technique on the routing
Currently the proposed reconfiguration method is only capable of adapting the contents of the LUTs by manipulating the bits in the bitstream. It is theoretically possible to investigate and adapt the routing of a design in a similar manner, but this could potentially damage the device if internal resource contention is created by toggling the bits in the bitstream. Furthermore, the LUTs have the advantage that they can be modelled as multiplexers, which can be represented by a truth table. The routing on the other hand is more stochastic and further investigation is required to locate and identify the routing bits to (hopefully) allow these bits to be safely manipulated.

Re-measure the different functional densities when reconfiguring architectures
Once the routing bits are successfully located and identified, the proposed specialisation and reconfiguration method can be used to not only adapt the LUT contents, but also the architecture of the system. This could improve the critical path of the design, which reduces the execution time of the application. This has a direct effect on the functional density, and the functional density comparison of Chapter 6 has to be repeated.

Applying the proposed specialisation and reconfiguration to a physical system
Up to now, the proposed specialisation and reconfiguration method was only implemented

on the ML507 development board. The next step would be to implement it on a physical, real-time system, such as the one discussed in Section 7.4. An interesting study could also sprout from analysing the effect dynamic reconfiguration has on the control and system response, especially if the reconfiguration switches between different controller types.

Re-implement the specialiser using synchronous design
Even though the asynchronous implementation of the specialiser allows it to execute at the fastest possible rate, this is seen as unsupported flow by Xilinx® and without proper design this would be unsuitable for real-time applications. A better approach would be to sacrifice specialisation time for improved stability by rather using synchronous design. Since the output of the specialiser is available practically immediately, latching its output would only add one clock delay.

Improve the reconfiguration time by increasing the ICAP clock frequency
The literature has shown that it is possible to overclock the ICAP to 5.5 times the recommended clock frequency if custom hardware is added to the ICAP [60]. If this can be achieved, the reconfiguration time obtained in Chapter 6 for the proposed method can be reduced to the nano second domain. This would increase the functional density to just below that of the static application, which implies that the hardware only has to be reused once before obtaining a significant functional density advantage.

Adding CRCs to the proposed reconfiguration method
During the bitstream parsing and analysis the CRCs were disabled to help limit the differences between the compared designs to only the LUTs. CRCs were also ignored during the bitstream injection, to prevent the FPGA from aborting the new configuration. Additionally, calculating a new CRC will again add overhead to the reconfiguration process, possibly causing the reconfiguration to become unusable in real-time applications. For the proposed reconfiguration technique to be truly useful in critical real-time applications, a method has to be included to ensure correct reconfiguration without adding significant overhead. The ECC bits (in conjunction with the CRC) are used to detect single- or double-bit errors in the configuration frame data, and by manipulating the bits in the bitstream can easily be interpreted as such. A method should thus be found to calculate the new CRC for the specialised bitstream loaded onto the device in real-time, without negatively affecting the functional density.

7.6 Unique contributions

Three main contributions were made during the course of this thesis, focussing on the shortcomings of dynamic reconfiguration when applied to real-time applications. These contributions are listed below in the sequence they were made during this thesis. The reason for this being that the contributions build upon one another.

7.6.1 Providing new insight into the composition of a Xilinx® FPGA configuration

Even though numerous works in the literature aim to manipulate the FPGA resources with the least amount of overhead, most rely on a layer of abstraction, provided by the Xilinx® design language (XDL). Due to this additional layer of abstraction, most of these methods are unsuitable for manipulating FPGA resources in real-time. The research that came close to allowing this, is largely unfinished and discontinued. Manipulating FPGA resources at bit-level is risky, since hardware contention can be caused. Consequently, many researchers aimed to map the FPGA configuration space to the bitstream. Unfortunately, most of this research is aimed at older FPGA device families. As a result, a method to parse and analyse an FPGA bitstream was proposed. Several configuration strings were found that can be used to derive the bitstream encoding for any LUT construct, by simply inserting them into the Boolean expression represented by the LUT construct. Methods were also proposed for deriving the frame composition of a specific FPGA device family, as well as formulae for mapping the slice coordinate to the device frame address.

7.6.2 A novel method for specialising an FPGA configuration dynamically

The information obtained from the bitstream parsing and analysis showed that the Boolean expression of a LUT can be used to derive its bitstream configuration. This implies that passive digital logic can be used to generate the bitstream configuration for a LUT. This means that a new configuration can be generated almost immediately, which is orders of magnitude faster than similar specialisation methods proposed in the literature. In fact, this is the only specialisation method allowing the modification of FPGA resources in real-time. An additional advantage is that this specialisation method is device independent, but some modifications have to be made for device families with 4-input LUTs.

7.6.3 Combining the configuration specialiser with the BRAM-based architecture

The last contribution is made by combining the hardware controlled BRAM-based reconfiguration architecture proposed in the literature, with the developed bitstream specialiser. The reason for selecting the BRAM-based architecture is because it allows dynamic reconfiguration with the least amount of system-induced overhead. It also allows the ICAP to be overclocked past the recommended clock frequency of 100 MHz. The drawback of this architecture is the limited amount of BRAM available for storing the configuration data, which implies that only a subset of configurations can be stored. The addition of the bitstream specialiser overcomes this limitation by allowing the configuration stored in the BRAM to be specialised to represent any other hardware set. Even though this methodology currently only allows the content of the LUTs to be adapted, this is the first step towards modifying any FPGA resource by direct bitstream manipulation. This is also the first time a reconfiguration method is proposed for reconfiguring a real-time application. It was showed that the bitstream specialiser combined with hardware

controlled reconfiguration is capable of reconfiguring a real-time system, and that a substantial improvement in functional density can be obtained if the hardware is reused between execution cycles.

7.7 Closure

Due to the overhead introduced by the reconfiguration process, dynamic reconfiguration is only suitable for quasi-static applications. The aim of the research presented in this thesis was to present a method for reducing reconfiguration overhead to such an extent that the functional density of a dynamically reconfigured real-time application would rival that of a static implementation. This was done by selecting an optimal BRAM-based architecture from the literature with the least amount of reconfiguration overhead. This architecture was then amended with a bitstream specialiser that is used to generate a new configuration on-the-fly. The original hypothesis was that this can be achieved without adding overhead to the reconfiguration process, which was confirmed. Using this architecture, it was shown that not only was it possible to dynamically reconfigure a real-time application, but also that the functional density obtained could rival that of its static equivalent. Also shown was that the functional density of this method is a significant improvement upon using conventional reconfiguration techniques. Even though the proposed method is only capable of reconfiguring the LUTs of a real-time application, this is the first step towards allowing full reconfiguration of applications with dynamic characteristics.

APPENDIX A

SUPPLEMENTARY LITERATURE

This appendix provides supplementary information regarding the literature survey presented in Chapter 2, specifically relating to ways to reduce configuration cost as discussed in Section 2.2.

Table A.1: Summary of routing strategies to reduce the cost of routing

Routing technique	**Description**
Rip-up and reroute [129, 130]	Rip-up and reroute was proposed to remedy the unrouted nets of other techniques. The success of the routes are dependent on the order in which they are routed. Additional cost functions can be added to ensure critical paths are routed first.
PathFinder [131]	Pathfinder is a router that aims to find a balance between performance and routability. An iterative algorithm is used to negotiate which signal needs a resource the most. Delay is minimised by allowing critical signals a higher preference in resource-allocation.
Versatile place and route (VPR) [132, 133]	As already discussed, VPR is an optimization and extension to PathFinder and is arguably the most popular placement and routing technique.
Stochastic routing [134, 135]	The idea is to use stochastic methods to locate near-optimal placement solutions without exhaustively enumerating all design points.
Parallel routing [136, 137]	Parallel placement aims to increase performance by implementing standard negotiation-based routing algorithms in parallel without a reduction in the quality of the results.
Hardware assisted [138]	Adding hardware to the routing network, assists the routing network in finding free routes to be used in the routing process.

Table A.2: Summary of research to reduce placement cost

Placement technique	**Description**
Quadratic [139]	Quadratic placement attempts to minimise total squared length by solving linear equations.
Min-cut [140, 141]	The min-cut optimization technique uses recursive partitioning to divide a net-list of circuits into increasingly smaller sub-circuits and maps these smaller circuits onto the FPGA. This leaves the highly connected blocks in one partition thus decreasing placement cost.
Parallel placement [140]	The rapid development of multi-core CPUs make parallelization an appealing solution for providing fast placements. Multiple logic blocks are routed in parallel.
Hybrid algorithms[140, 141]	Hybrid algorithms are usually multi-stage placement algorithms that combine multiple placement techniques, one of which is usually simulated annealing.
Simulated annealing [13]	Simulated annealing is the most widely used algorithm for placement on an FPGA and forms the basis of most placement algorithms. Simulated annealing placement mimics the annealing process used to gradually cool molten metal. The most optimal placement is obtained by initially placing random logic blocks and swapping the blocks to reduce the cost.
Versatile place and route (VPR) [132, 133]	VPR is a time-driven simulated annealing placement and routing technique that is based on PathFinder and includes enhancements that improve run-time and quality. Its annealing schedule is based on calculated parameters, rather than fixed start and end temperatures.
Ultra fast placement (UFP) [142]	UFP aims to improve on VPR by combining VPR with a multi-level clustering strategy. This improves the scalability of the placer at the cost of an increase in wire length.
Analytical placers [143–145]	Analytical placers aim to improve scaling issues without a reduction in quality by creating a smooth placement function that approximates routed wire length. Analytic placers tackle the placement problem from the top-down and considers global connectivity, rather than iteratively evaluating small-scale modifications.
Hardware-assisted simulated annealing [146]	Each space where a LUT could reside is assigned its own processing element. The processing element is responsible for keeping track of which LUT it contains, as well as the connectivity to its neighbours. The processing element is also aware of its position and an estimate is kept of the connected LUTs.

Table A.3: Methods to reduce the cost of generating bitstreams

Bitstream generation	Description
Lean versions [147, 148]	A just-in-time approach is taken to dynamically convert software binary instructions onto FPGAs. These compilers require lean versions of the conventional mapping, placement and routing algorithms to improve mapping, placement and routing times respectively.
Reducing quality [142]	Reducing the quality of the placement and routing allows quicker convergence of the tools that results in quicker bitstream generation. In this context, reduction in quality is defined as an increase in the wiring area of the circuit, a reduction in the operating speed of the circuit, greater wirelength of the mapped circuit, and an unnecessary increase in resource-utilization.
Generic netlists [27, 28, 91, 100]	A generic netlist is a netlist not physically mapped to a device. This research aims to improve mapping of generic FPGA netlists to physical netlists for real architectures.
Reusing place and route [11, 149]	Placement and routing take a significant amount of time. By reusing the place and route netlists saves configuration time.
Constant multiplication [150]	Constant multiplication is a technique used to reduce FPGA resource requirements by exploiting constant-specific optimizations.
Tunable lookup tables (TLUTs) [11, 101, 151–154]	The configuration bits of the lookup tables are expressed as a Boolean function of the parameter inputs. All the other configuration bits are static. These Boolean functions are evaluated at run time to produce a new configuration. This leads to fast reconfiguration, but is not the most compact implementation. A tunable mapper is used to map a gate-level circuit into these tunable lookup tables.
Tunable connections [12, 151, 153]	These aim to expand on TLUTs by also expressing the routing configuration bits of an FPGA as a Boolean function. This allows for faster rerouting.
Combitgen [37, 62]	Combitgen is a technique that combines the advantages of existing Xilinx partial dynamic reconfiguration flows. It also utilizes redundancy to reduce the number of frames in the bitstream without an increase in quantity.
Compression [57, 155, 156]	This aims to improve reconfiguration time by reducing the bitstream size either by eliminating the redundant frames in the bitstream or by traditional compression techniques.
Shift register lookup table (SRL) [12, 127]	The SRL capability of the Xilinx Virtex-series FPGAs are used to reconfigure the functionality of the LUTs. SRLs are LUTs whose elements are organised as a shift register. By shifting the data into the SRL, the functionality is changed.

APPENDIX B

THE XILINX® FPGA ARCHITECTURE

This appendix provides more information regarding the architecture of Xilinx® FPGAs. This is particularly useful for providing a deeper understanding of the bitstream parsing and analysis method described in Chapter 4.

Figures B.1 to B.1 shows the logic diagrams of the three types of slices available on Xilinx® FPGAs, as given in the Xilinx® user guides. The SLICEX is currently exclusively used in the Spartan®-6 family. This is followed by Figure B.4 depicting the floorplan of the FPGA used throughout this thesis, which acts as a visual representation of the slices and their equivalent coordinates. Also shown in the figure is the exploded view fo two CLB's contents.

The last figure supplied, Figure B.5, depicts a logic diagram of a slice when it is configured as a complex RAM construct, with the implemented routes depicted in cyan. In this particular instance, the RAM16X8s construct is shown, and aids in illustrating the constructs used in experiments 6 (Section 4.4.5) to 8 (Section 4.4.7).

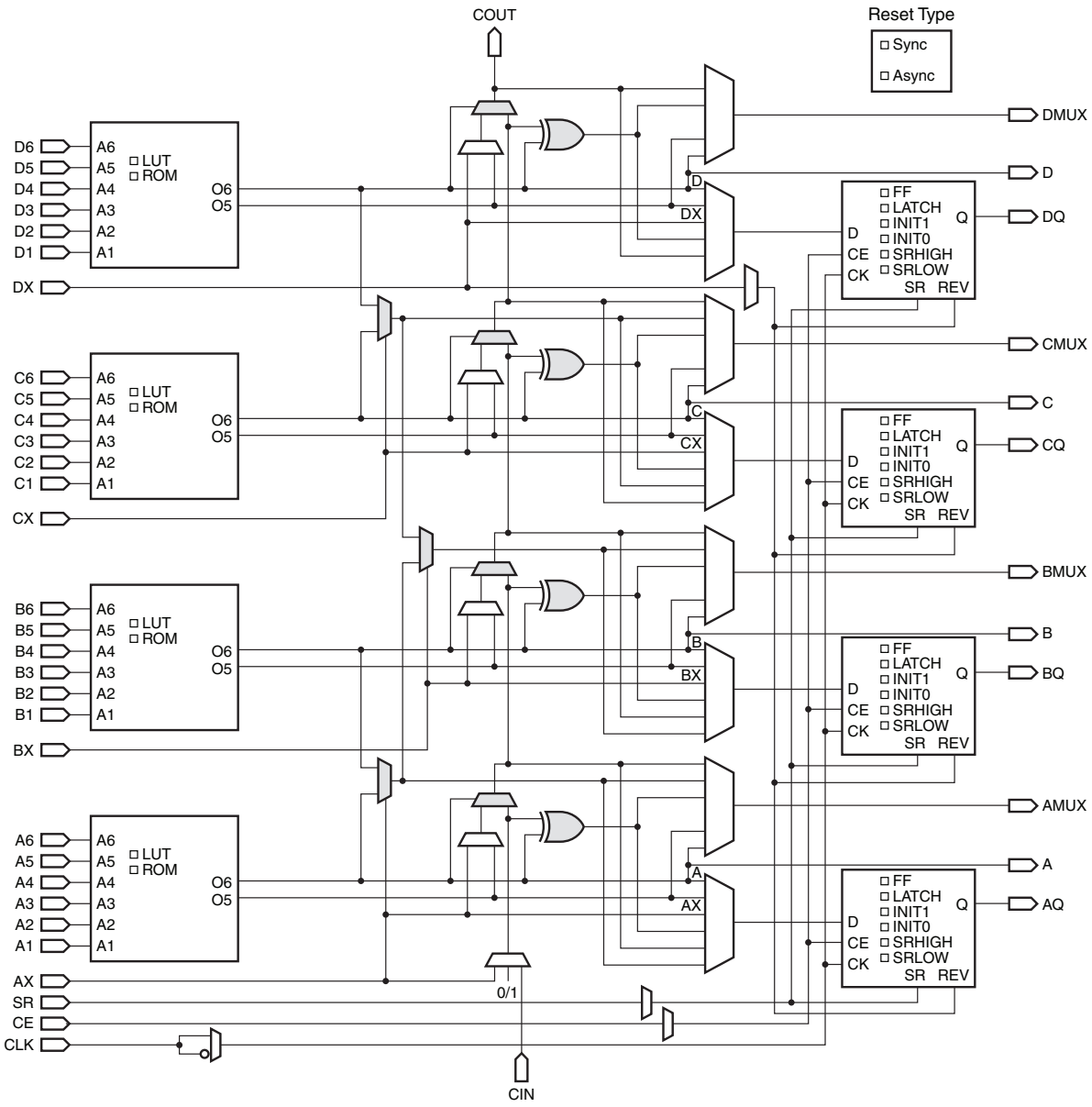


Figure B.1: Logic diagram of a SLICEL

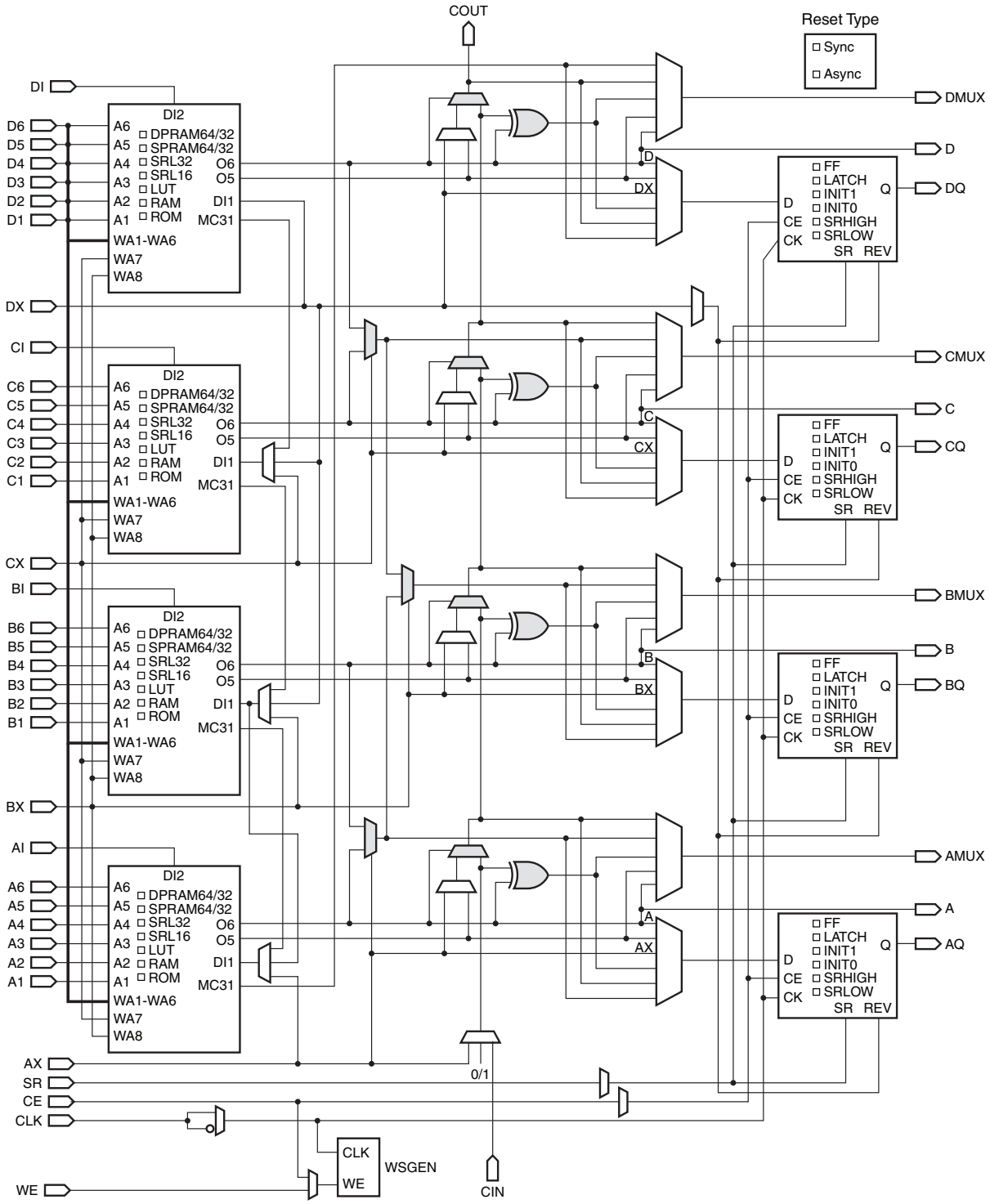


Figure B.2: Logic diagram of a SLICEM

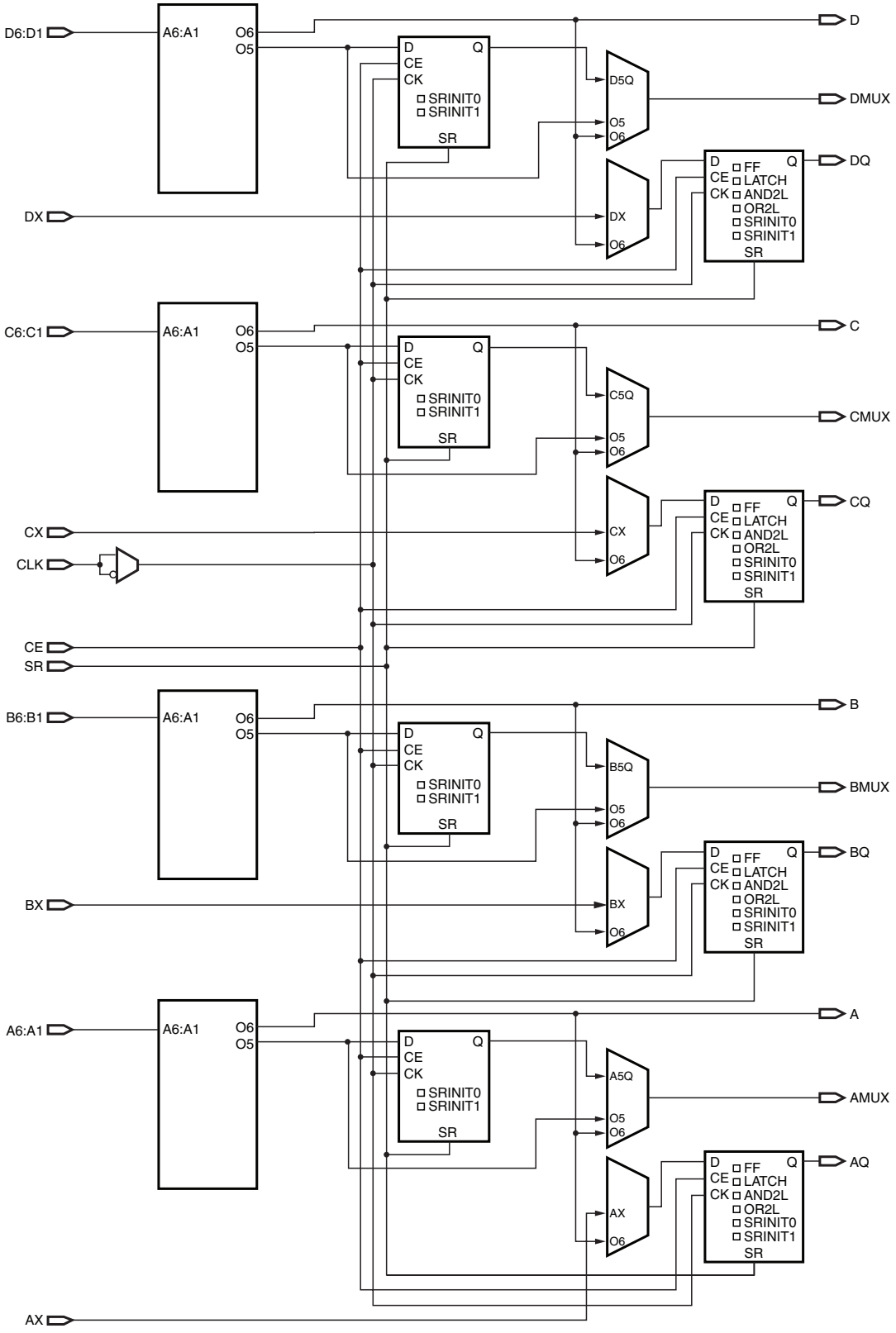


Figure B.3: Logic diagram of a SLICEX

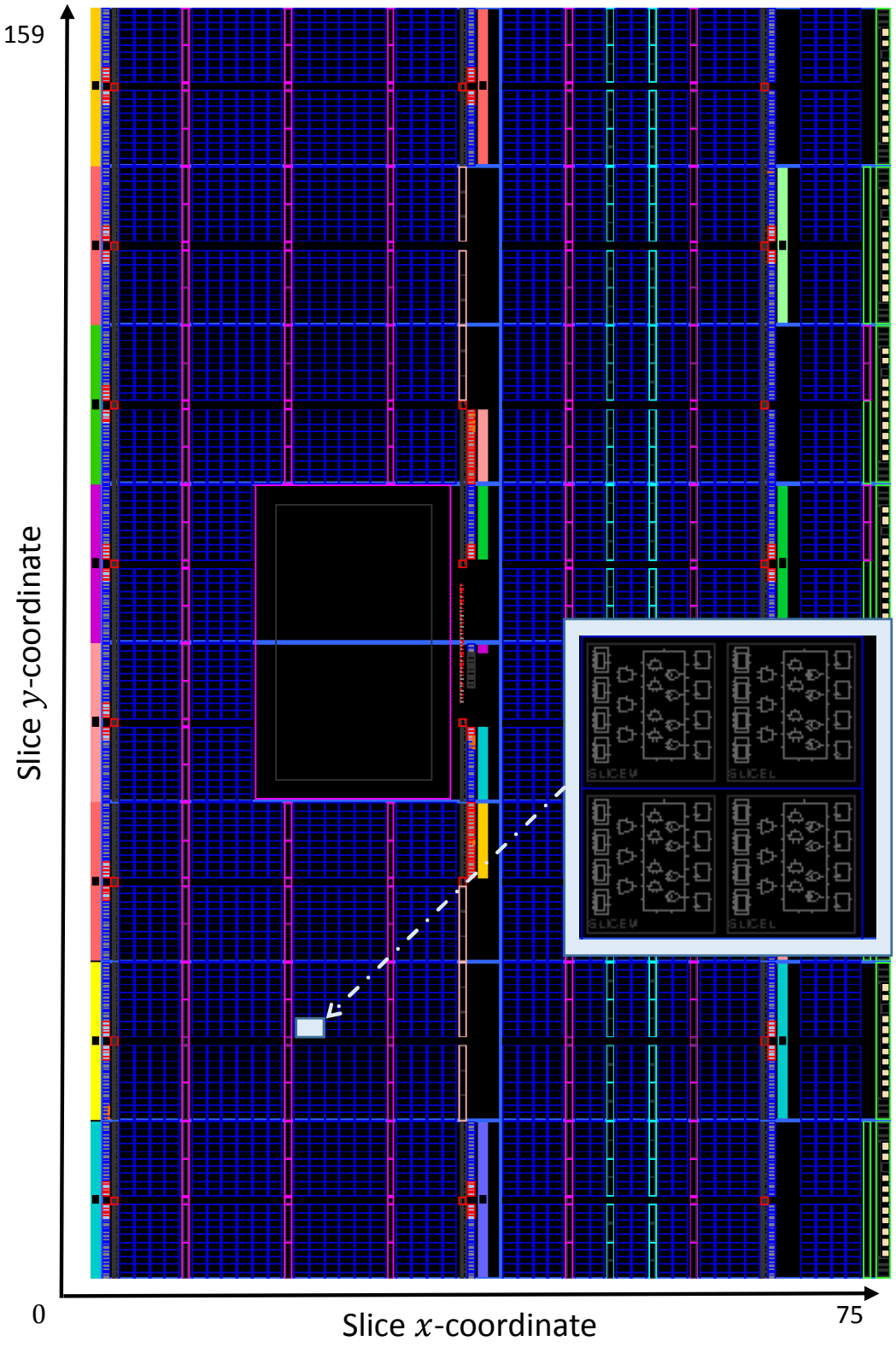


Figure B.4: Virtex®-5 VFX70T floorplan showing slice coordinates and the contents of two CLBs

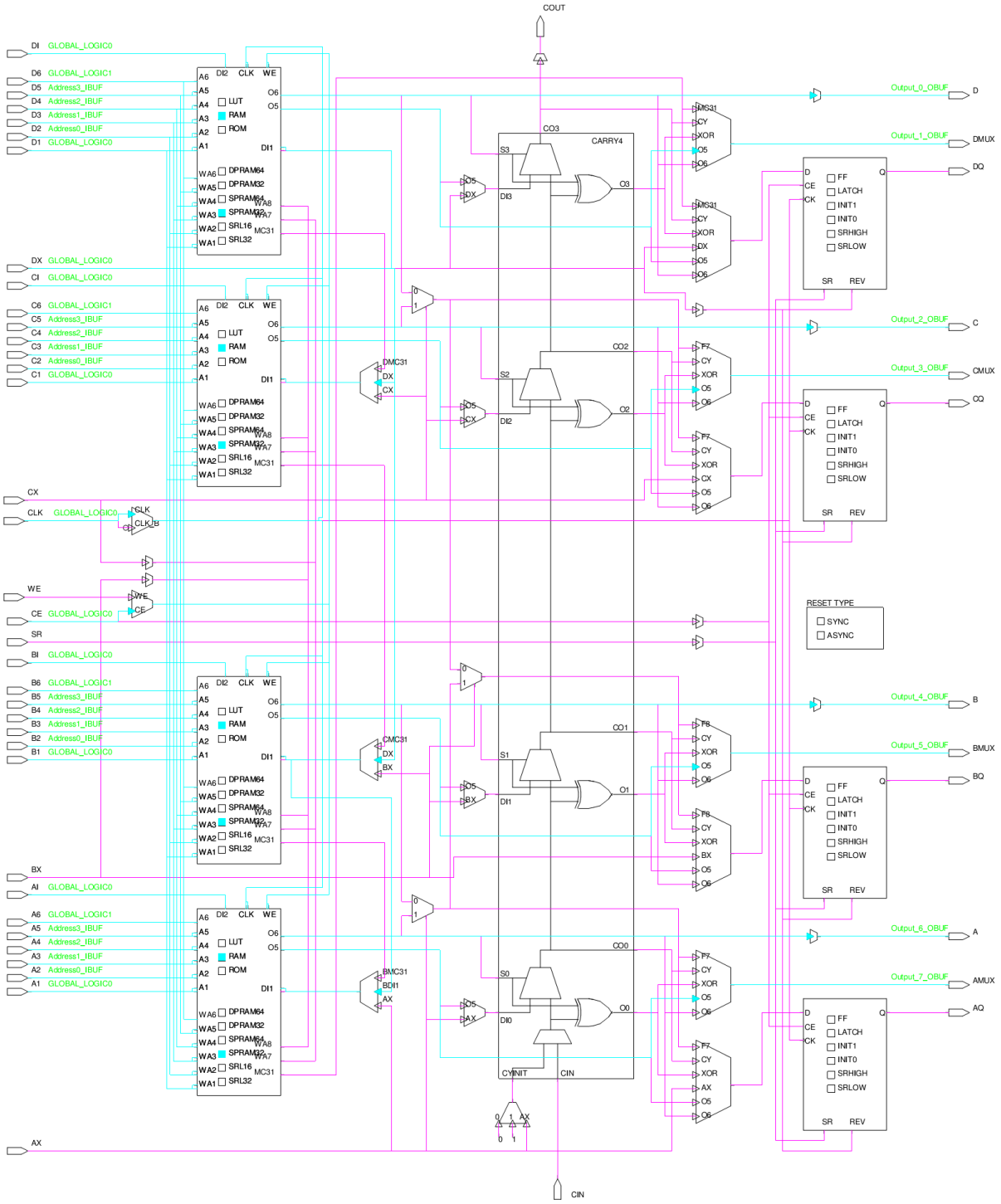


Figure B.5: Logic diagram showing the slice configuration for a RAM16X8s construct

APPENDIX C

ADDITIONAL BITSTREAM INFORMATION

This appendix provides two additional figures depicting the contents of the bitstream formatted in two different ways. The first, Figure C.1 depicts the raw, unformatted hexadecimal contents of the bitstream. The second, Figure C.2 shows the ASCII formatted bitstream contents. This is obtained by supplying the *-b* switch to BitGen™. These formatted bitstreams were used for the bitstream analysis discussed in Chapter 4, Section 4.3. Also shown in the figure are some of the decoded bitstream configuration commands, which were useful in understanding and designing the ICAP state machine, discussed in Section 3.4.2.

0	00	09	0F	F0	0F	F0	0F	F0	0F	F0	00	00	01	61	00	26	63	6F	6E
13	66	69	67	5F	32	5F	72	6F	75	74	65	64	2E	6E	63	64	3B	55	73
26	65	72	49	44	3D	30	78	46	46	46	46	46	46	46	46	00	62	00	0E
39	35	76	66	78	37	30	74	66	66	31	31	33	36	00	63	00	0B	32	30
4C	31	31	2F	30	37	2F	32	38	00	64	00	09	31	38	3A	32	37	3A	35
5F	31	00	65	00	33	8C	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
72	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
85	FF	00	00	00	BB	11	22	00	44	FF	FF	FF	FF	FF	FF	FF	FF	AA	99
98	55	66	20	00	00	00	30	02	00	01	00	00	00	00	30	00	80	01	00
AB	00	00	00	20	00	00	00	30	00	80	01	00	00	00	07	20	00	00	00
BE	20	00	00	00	30	02	20	01	00	00	00	00	30	02	60	01	00	00	00
D1	00	30	01	20	01	00	00	3F	E5	30	01	C0	01	00	00	00	00	30	01
E4	80	01	03	2C	60	93	30	00	80	01	00	00	00	09	20	00	00	00	30
F7	00	C0	01	00	40	00	00	30	00	A0	01	00	40	00	00	30	00	C0	01
10A	00	00	00	00	30	03	00	01	00	00	00	00	20	00	00	00	20	00	00

Figure C.1: Sectional view of the bitstream contents

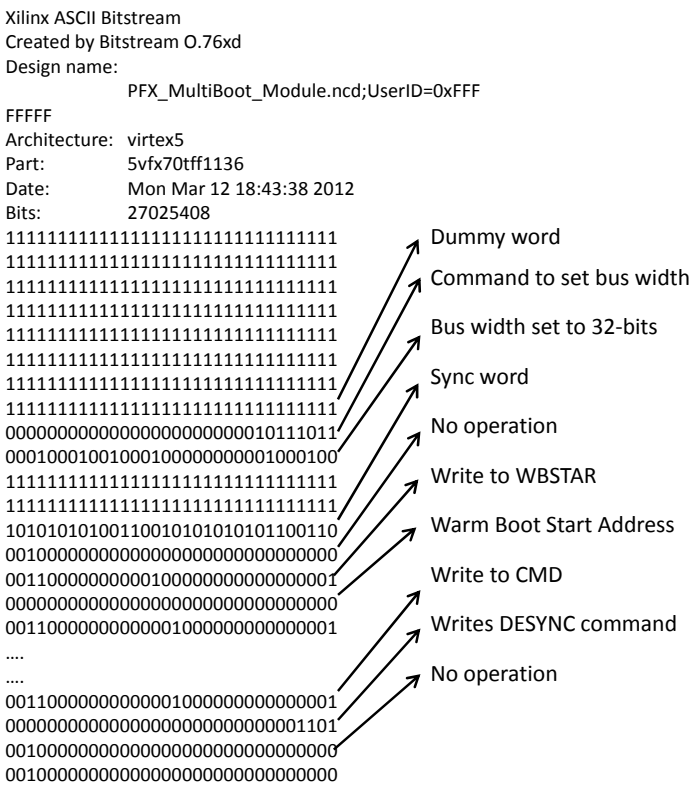


Figure C.2: Sectional view of the ASCII converted bitstream contents and showing typical reconfiguration commands

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 1–4, April 1965.

[2] N. Myhrvold, "The next fifty years of software." Presented at ACM97, 1997.

[3] R. Hartenstein, "The von Neumann syndrome," in *Proceedings of the Stamatis Vassiliadis Memorial Symposium*, 2007.

[4] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the conference on Design, Automation and Test in Europe (DATE 2001)*. IEEE Press, 2001, pp. 642–649.

[5] G. Estrin, "Organization of computer systems - the fixed plus variable structure computer," in *Papers presented at the May 3-5, 1960, Western joint IRE-AIEE-ACM computer conference*, 1960, pp. 33–40.

[6] G. Estrin, "Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer," *IEEE Annals of the History of Computing*, vol. 24, no. 4, pp. 3–9, 2002.

[7] S. Hauck and A. DeHon, *Reconfigurable computing: The theory and practice of FPGA-based computing.* Morgan Kaufmann, 2008.

[8] J. G. Eldredge and B. L. Hutchings, "Run-time reconfiguration: A method for enhancing the functional density of SRAM-based FPGAs," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 12, no. 1, pp. 67–86, 1996.

[9] C. L. Seitz, "Concurrent VLSI architectures," *Computers, IEEE Transactions on*, vol. 100, no. 12, pp. 1247–1265, 1984.

[10] M. J. Wirthlin and B. L. Hutchings, "Improving functional density using run-time circuit reconfiguration," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 6, no. 2, pp. 247–256, 1998.

[11] K. Bruneel, P. Bertels, and D. Stroobandt, "A method for fast hardware specialization at run-time," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2007)*. IEEE, 2007, pp. 35–40.

[12] K. Bruneel, "Efficient circuit specialization for dynamic reconfiguration of FPGAs," Ph.D. dissertation, PhD thesis, Ghent University, 2011.

[13] S. Kirkpatrick, C. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[14] Xilinx Inc., "Virtex-5 FPGA configuration user guide," Xilinx Inc., User guide 191, Aug 2010, UG191.

[15] A. Gambier, "Real-time control systems: a tutorial," in *Proceedings of the 5th Asian Control Conference*, vol. 2. IEEE, 2004, pp. 1024–1031.

[16] R. R. le Roux, "An embedded controller for an active magnetic bearing and drive electronic system," Master's thesis, The school of electrical, electronic and computer engineering, North-West University, 2009.

[17] D. Castellone, "Analysis of the configuration bitstream format of a Xilinx Virtex-5 XCLX110T," Politecnico di Milano, Tech. Rep., Jun. 2011.

[18] R. R. le Roux, G. van Schoor, and P. A. van Vuuren, "A survey on reducing reconfiguration cost: reconfigurable PID control as a special case," in *Proceedings of the 19th World Congress of the International Federation of Automatic Control*. IFAC, 2014, pp. 1320–1330.

[19] R. R. le Roux, G. van Schoor, and P. A. van Vuuren, "Block RAM implementation of a reconfigurable real-time PID controller," in *Proceedings of the International Conference on High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th*. IEEE, 2012, pp. 1383–1390.

[20] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel processing in a restructurable computer system," *IEEE Transactions on Electronic Computers*, vol. EC-12, no. 6, pp. 747–755, Dec 1963.

[21] T. N. Sasamal and R. Prasad, "Module based and difference based implementation of partial reconfiguration on FPGA: A review," *International Journal of Engineering Research and Applications (IJERA)*, vol. 1, no. 4, pp. 1898–1903, 2011.

[22] Z. E. A. A. Ismaili and A. Moussa, "Self-partial and dynamic reconfiguration implementation for AES using FPGA," *International Journal of Computer Science Issues (IJCSI)*, vol. 1, pp. 33–40, Aug 2009.

[23] D. Lim and M. Peattie, "Two flows for partial reconfiguration: Module based or small bit manipulations," Xilinx Inc., Application note 290 (v1.0), May 2002, XAPP290.

[24] C. Kao, "Benefits of partial reconfiguration," *XCell journal*, vol. 55, pp. 65–67, 2005.

[25] T. J. Todman, G. A. Constantinides, S. J. Wilton, O. Mencer, W. Luk, and P. Y. Cheung, "Reconfigurable computing: architectures and design methods," *IEEE Proceedings on Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[26] E. Kusse and J. Rabaey, "Low-energy embedded FPGA structures," in *Proceedings of the International Symposium on Low Power Electronics and Design.* IEEE, 1998, pp. 155–160.

[27] J. Leonard and W. H. Mangione-Smith, "A case study of partially evaluated hardware circuits: Key-specific DES," in *Proceedings of the Field-Programmable Logic and Applications.* Springer, 1997, pp. 151–160.

[28] S. Singh, J. Hogg, and D. McAuley, "Expressing dynamic reconfiguration by partial evaluation," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE, 1996, pp. 188–194.

[29] S. Ichikawa and S. Yamamoto, "Data dependent circuit for subgraph isomorphism problem," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream.* Springer, 2002, pp. 1068–1071.

[30] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating boolean satisfiability with configurable hardware," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE, 1998, pp. 186–195.

[31] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *Journal of VLSI signal processing*, vol. 28, no. 1-2, pp. 7–27, 2001.

[32] C. Claus, F. Altenried, and W. Stechele, "Dynamic partial reconfiguration of Xilinx FPGAs lets systems adapt on the fly: a video-based driver assistance application demonstrates effective use of situation-adaptive hardware," *XCell journal*, vol. 70, pp. 18–23, 2010.

[33] D. Kim, "An implementation of fuzzy logic controller on the reconfigurable FPGA system," *IEEE Transactions on Industrial Electronics*, vol. 47, no. 3, pp. 703–715, 2000.

[34] G. Economakos and C. Economakos, "A run-time reconfigurable fuzzy PID controller based on modern FPGA devices," in *Proceedings of the Mediterranean Conference on Control and Automation (MED 2007).* IEEE, 2007, pp. 1–6.

[35] S. M. Dilek, A. Ayranci, A. Seker, O. Ceylan, and H. B. Yagci, "AX.25 protocol compatible reconfigurable 2/4 FSK modulator design for nano/micro-satellites," in *Proceedings of the 20^{th} Telecommunications Forum (TELFOR 2012).* IEEE, 2012, pp. 416–419.

[36] F. Giovagnini and A. D. Marzo, "How partial dynamic reconfiguration helped make an FSK modulator," *XCell journal*, vol. 80, pp. 38–43, 2012.

[37] C. Claus, F. H. Muller, J. Zeppenfeld, and W. Stechele, "A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007).* IEEE, 2007, pp. 1–7.

[38] T. Davidson, F. Abouelella, K. Bruneel, and D. Stroobandt, "Dynamic circuit specialisation for key-based encryption algorithms and DNA alignment," *International Journal of Reconfigurable Computing*, vol. 2012, pp. 1–13, 2012.

[39] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, "Configurable computing solutions for automatic target recognition," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE, 1996, pp. 70–79.

[40] Xilinx Inc., "PowerPC 405 processor block reference guide," Xilinx Inc., User Guide 018 (v2.4), Jan 2010, UG018.

[41] Xilinx Inc., "Embedded processor block in Virtex-5 FPGAs reference guide," Xilinx Inc., User Guide 200 (v1.8), Feb 2010, UG200.

[42] Xilinx Inc., "MicroBlaze processor reference guide," Xilinx Inc., User Guide 081 (v9.0), Jan 2008, UG081.

[43] Xilinx Inc., "OPB HWICAP (v1.00.b) product specification," Xilinx Inc., Data Sheet 280, Jul 2006, DS280.

[44] Xilinx Inc., "LogiCORE IP XPS HWICAP (v5.00.a) product specification," Xilinx Inc., Data Sheet 586, Jul 2010, DS586.

[45] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time partial reconfiguration speed investigation and architectural design space exploration," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2009)*. IEEE, 2009, pp. 498–502.

[46] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in FPGA systems: A survey and a cost model," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, p. 36, 2011.

[47] K. Papadimitriou, A. Anyfantis, and A. Dollas, "An effective framework to evaluate dynamic partial reconfiguration in FPGA systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 59, no. 6, pp. 1642–1651, 2010.

[48] K. Papadimitriou, A. Anyfantis, and A. Dollas, "Methodology and experimental setup for the determination of system-level dynamic reconfiguration overhead," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. IEEE, 2007, pp. 335–336.

[49] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker, "A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2008)*. IEEE, 2008, pp. 535–538.

[50] S. Lamonnier, M. Thoris, and M. Ambielle, "Accelerate partial reconfiguration with a 100% hardware solution," *XCell journal*, vol. 79, pp. 44–49, 2012.

[51] J. Delahaye, J. Palicot, C. Moy, and P. Leray, "Partial reconfiguration of FPGAs for dynamical reconfiguration of a software radio platform," in *Proceedings of the 16th IST Mobile and Wireless Communications Summit*. IEEE, 2007, pp. 1–5.

[52] K. van der Bok, R. Chaves, G. Kuzmanov, L. Sousa, and A. van Genderen, "Dynamic FPGA reconfigurations with run-time region delimitation," in *Proceedings of the 18th annual workshop on circuits, systems and signal processing (ProRISC)*, 2007, pp. 201–207.

[53] A. Cuoccio, P. R. Grassi, V. Rana, M. D. Santambrogio, and D. Sciuto, "A generation flow for self-reconfiguration controllers customization," in *Proceedings of the 4th IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2008)*. IEEE, 2008, pp. 279–284.

[54] A. Nabina and J. L. Nunez-Yanez, "Dynamic reconfiguration optimisation with streaming data decompression," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2010)*. IEEE, 2010, pp. 602–607.

[55] J. C. Hoffman and M. S. Pattichis, "A high-speed dynamic partial reconfiguration controller using direct memory access through a multiport memory controller and overclocking with active feedback," *International Journal of Reconfigurable Computing*, vol. 2011, pp. 1–10, 2011.

[56] M. Shelburne, C. Patterson, P. Athanas, M. Jones, B. Martin, and R. Fong, "MetaWire: using FPGA configuration circuitry to emulate a network-on-chip," *Computers and Digital Techniques, IET*, vol. 4, no. 3, pp. 159–169, 2010.

[57] S. Liu, R. N. Pittman, and A. Forin, "Minimizing partial reconfiguration overhead with fully streaming DMA engines and intelligent ICAP controller," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays (FPGA 2009)*, 2009, pp. 292–292.

[58] F. Duhem, F. Muller, and P. Lorenzini, "Reconfiguration time overhead on field programmable gate arrays: reduction and cost model," *Computers and Digital Techniques, IET*, vol. 6, no. 2, pp. 105–113, 2012.

[59] R. Bonamy, H. Pham, S. Pillement, and D. Chillet, "UPaRC—ultra-fast power-aware reconfiguration controller," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 2012)*. IEEE, 2012, pp. 1373–1378.

[60] S. G. Hansen, D. Koch, and J. Torresen, "High speed partial run-time reconfiguration using enhanced ICAP hard macro," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Ph.D Forum (IPDPSW 2011)*. IEEE, 2011, pp. 174–180.

[61] J. L. Núñez and S. Jones, "Lossless data compression programmable hardware for high-speed data networks," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT 2002)*. IEEE, 2002, pp. 290–293.

[62] C. Claus, F. H. Müller, and W. Stechele, "Combitgen: A new approach for creating partial bitstreams in Virtex-II Pro devices," in *Proceedings of the Workshop on reconfigurable computing Proceedings (ARCS 2006)*, 2006, pp. 122–131.

[63] F. Cancare, D. B. Bartolini, M. Carminati, D. Sciuto, and M. D. Santambrogio, "On the evolution of hardware circuits via reconfigurable architectures," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 5, no. 4, pp. 22:1–22:22, Dec 2012.

[64] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, N. Salami, N. Kajihara *et al.*, "Real-world applications of analog and digital evolvable hardware," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 3, pp. 220–235, 1999.

[65] H. de Garis, "Evolvable hardware: Genetic programming of a Darwin machine," in *Artificial Neural Nets and Genetic Algorithms*, R. Albrecht, C. Reeves, and N. Steele, Eds. Springer Vienna, 1993, pp. 441–449.

[66] X. Yao and T. Higuchi, "Promises and challenges of evolvable hardware," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 29, no. 1, pp. 87–97, 1999.

[67] F. Cancare, S. Bhandari, D. B. Bartolini, M. Carminati, and M. D. Santambrogio, "A bird's eye view of FPGA-based evolvable hardware," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2011)*. IEEE, 2011, pp. 169–175.

[68] A. Upegui and E. Sanchez, "Evolving hardware by dynamically reconfiguring Xilinx FPGAs," in *Evolvable Systems: From Biology to Hardware*. Springer, 2005, pp. 56–65.

[69] A. Thompson, "An evolved circuit, intrinsic in silicon, entwined with physics," in *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware*, ser. ICES '96. London, UK, UK: Springer-Verlag, 1996, pp. 390–405.

[70] F. Benz, A. Seffrin, and S. A. Huss, "Bil: A tool-chain for bitstream reverse-engineering," in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL 2012)*. IEEE, Aug 2012, pp. 735–738.

[71] É. Rannaud, "From the bitstream to the netlist," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. Citeseer, 2008, pp. 264–264.

[72] J. Beale, "Security through obscurity ain't what they think it is," Electronic, 2000, Bastille Linux Project. [Online]. Available: http://www.portknocking.org/docs/security-through-obscurity-beale.pdf

[73] L. Sekanina, "Modeling of evolvable hardware," Master's thesis, Brno University of Technology, 1999.

[74] R. Salvador, A. Otero, J. Mora, E. de la Torre, T. Riesgo, and L. Sekanina, "Self-reconfigurable evolvable hardware system for adaptive image processing," *Computers, IEEE Transactions on*, vol. 62, no. 8, pp. 1481–1493, Aug 2013.

[75] R. Dobai and L. Sekanina, "Towards evolvable systems based on the Xilinx Zynq platform," in *Proceedings of the IEEE International Conference on Evolvable Systems.* IEEE Computational Intelligence Society, 2013, pp. 89–95.

[76] F. Cancare, M. D. Santambrogio, and D. Sciuto, "A direct bitstream manipulation approach for Virtex-4-based evolvable systems," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS 2010).* IEEE, 2010, pp. 853–856.

[77] R. Salvador, A. Otero, J. Mora, E. de la Torre, T. Riesgo, and L. Sekanina, "Implementation techniques for evolvable HW systems: virtual vs. dynamic reconfiguration," in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL 2012).* IEEE, 2012, pp. 547–550.

[78] S. Guccione, D. Levi, and P. Sundararajan, "Jbits: Java based interface for reconfigurable computing," Xilinx Inc., Tech. Rep., 1999.

[79] S. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based interface for reconfigurable computing," in *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, vol. 261, 1999.

[80] R. K. Soni, N. Steiner, and M. French, "Open-source bitstream generation," in *Proceedings of the IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2013).* IEEE, 2013, pp. 105–112.

[81] J. Zhu, Y. Li, G. He, and X. Xia, "An intrinsic evolvable hardware based on multiplexer module array," in *Proceedings of the 7th International Conference on Evolvable Systems: From Biology to Hardware*, ser. ICES'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 35–44.

[82] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A self-reconfiguring platform," in *Proceedings of Field Programmable Logic and Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2778, pp. 565–574.

[83] C. Beckhoff, D. Koch, and J. Torresen, "The Xilinx design language (XDL): tutorial and use cases," in *Proceedings of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2011).* IEEE, Jun 2011, pp. 1–8.

[84] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-builder: a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2008).* IEEE, 2008, pp. 119–124.

[85] C. Beckhoff, D. Koch, and J. Torresen, "Migrating static systems to partially reconfigurable systems on Spartan-6 FPGAs," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Ph.D Forum (IPDPSW 2011).* IEEE, May 2011, pp. 212–219.

[86] C. Claus, B. Zhang, M. Hübner, C. Schmutzler, J. Becker, and W. Stechele, "An XDL-based busmacro generator for customizable communication interfaces for dynamically

and partially reconfigurable systems," in *Proceedings of the Workshop on Reconfigurable Computing Education at ISVLSI*, 2007, pp. 5–12.

[87] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping," in *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2011)*. IEEE, 2011, pp. 117–124.

[88] S. Korf, D. Cozzi, M. Koester, J. Hagemeyer, M. Porrmann, U. Ruckert, and M. D. Santambrogio, "Automatic HDL-based generation of homogeneous hard macros for FPGAs," in *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2011)*. IEEE, 2011, pp. 125–132.

[89] C. Beckhoff, D. Koch, and J. Torresen, "Design tools for implementing self-aware and fault-tolerant systems on FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, p. 14, Jul 2014.

[90] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapid prototyping tools for FPGA designs: RapidSmith," in *Proceedings of the International Conference on Field-Programmable Technology (FPT 2010)*. IEEE, 2010, pp. 353–356.

[91] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder—a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2008)*, Sept 2008, pp. 119–124.

[92] K. Kepa, F. Morgan, K. Kosciuszkiewicz, L. Braun, and M. Hübner, "FPGA analysis tool: High-level flows for low-level design analysis in reconfigurable computing," in *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ser. ARC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 62–73.

[93] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith: Do-it-yourself CAD tools for Xilinx FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2011)*. IEEE, 2011, pp. 349–355.

[94] E. Horta and J. W. Lockwood, "PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs)," Dept. Comput. Sci., Washington Univ., Saint Louis, MO, Technical report WUCS-01-13, July 2001.

[95] C. J. Morford, "BitMat - bitstream manipulation tool for Xilinx FPGAs," Master's thesis, Virginia Polytechnic Institute and State University, 2005.

[96] A. Upegui and E. Sanchez, "Evolving hardware with self-reconfigurable connectivity in Xilinx FPGAs," in *Proceedings of the 1st NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2006)*. IEEE, 2006, pp. 153–162.

[97] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, Jun 2002.

[98] F. Cancare, M. Castagna, M. Renesto, and D. Sciuto, "A highly parallel FPGA-based evolvable hardware architecture," in *Proceedings of the International Conference on Parallel Computing*, vol. 19, 2009, pp. 608–615.

[99] M. Castagna, "Progettazione di un sistema evolutivo hardware basato su Virtex-4," Master's thesis, Politecnico di Milano, 2008.

[100] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 41–44.

[101] K. Bruneel, F. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2009)*. European Design and Automation Association, 2009, pp. 964–969.

[102] E. Vansteenkiste, K. Bruneel, and D. Stroobandt, "A connection router for the dynamic reconfiguration of FPGAs," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2012, pp. 357–364.

[103] T. Davidson, K. Bruneel, and D. Stroobandt, "Identifying opportunities for dynamic circuit specialization," in *Proceedings of the Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS-2012)*, 2012, pp. 18–21.

[104] Xilinx Inc., "Partial reconfiguration user guide," Xilinx Inc., User Guide 702, Apr 2013, UG702.

[105] Xilinx, Inc., "Partial reconfiguration tutorial: PlanAhead design tool," Xilinx, Inc., Tech. Rep. UG743 (v14.1), May 2012.

[106] J. Hussein and R. Patel, "Multiboot with Virtex-5 FPGAs and Platform Flash XL," Xilinx, Inc., Tech. Rep. XAPP1100, Nov. 2008.

[107] Xilinx Inc., "LogiCORE IP block memory generator v6.2," Xilinx Inc., Data Sheet 512, Jun 2011, DS512.

[108] J. Martin, *Programming real-time computer systems*. Prentice-Hall, 1965.

[109] Xilinx Inc., "Spartan-6 FPGA configurable logic block," Xilinx Inc., User guide 348, Feb 2010, UG384.

[110] Xilinx Inc., "Command line tools use guide," Xilinx Inc., User guide 628, March 2011, UG628.

[111] Xilinx Inc., "Virtex-5 FPGA user guide," Xilinx Inc., User guide 190, March 2012, UG190.

[112] C. Van Berkel, M. B. Josephs, and S. M. Nowick, "Applications of asynchronous circuits," *Proceedings of the IEEE*, vol. 87, no. 2, pp. 223–233, 1999.

[113] A. Moradi and O. Mischke, "Glitch-free implementation of masking in modern FPGAs," in *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST 2012)*. IEEE, 2012, pp. 89–95.

[114] A. Peled and B. Liu, "A new hardware realization of digital filters," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 22, no. 6, pp. 456–462, 1974.

[115] S. A. White, "Applications of distributed arithmetic to digital signal processing: A tutorial review," *ASSP Magazine, IEEE*, vol. 6, no. 3, pp. 4–19, 1989.

[116] W. Sen, T. Bin, and Z. Jim, "Distributed arithmetic for FIR filter design on FPGA," in *Proceedings of the International Conference on Communications, Circuits and Systems (ICCCAS 2007)*. IEEE, 2007, pp. 620–623.

[117] Y. Zhou and P. Shi, "Distributed arithmetic for FIR filter implementation on FPGA," in *Proceedings of the International Conference on Multimedia Technology (ICMT 2011)*. IEEE, 2011, pp. 294–297.

[118] Y. Chan, M. Moallem, and W. Wang, "Efficient implementation of PID control algorithm using FPGA technology," in *Proceedings of the 43rd IEEE Conference on Decision and Control (CDC 2004)*, vol. 5. IEEE, 2004, pp. 4885–4890.

[119] Y. F. Chan, M. Moallem, and W. Wang, "Design and implementation of modular FPGA-based PID controllers," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 4, pp. 1898–1906, Aug 2007.

[120] S. Khawam, T. Arslan, and F. Westall, "Synthesizable reconfigurable array targeting distributed arithmetic for system-on-chip applications," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE, 2004, p. 150.

[121] W. Pan, A. Shams, and M. A. Bayoumi, "NEDA: a new distributed arithmetic architecture and its application to one dimensional discrete cosine transform," in *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS 1999)*. IEEE, 1999, pp. 159–168.

[122] S. Yu and E. Swartziander Jr, "DCT implementation with distributed arithmetic," *Computers, IEEE Transactions on*, vol. 50, no. 9, pp. 985–991, 2001.

[123] Y. Chan and W. Siu, "On the realization of discrete cosine transform using the distributed arithmetic," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 39, no. 9, pp. 705–712, Sept 1992.

[124] R. Kshirsagar and S. Sharma, "Difference based reconfiguration," *International Journal of Adhesion and Advances in Engineering and Technology (IJAET)*, vol. 1, no. 2, pp. 194–197, May 2011.

[125] E. Eto, "Difference-based partial reconfiguration," Xilinx Inc., Application note 290 (v2.0), Dec 2007, XAPP290.

[126] K. Glette, J. Torresen, and M. Hovin, "Intermediate level FPGA reconfiguration for an online EHW pattern recognition system," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2009)*. IEEE, 2009, pp. 19–26.

[127] K. Heyse, B. Al Farisi, K. Bruneel, and D. Stroobandt, "Automating reconfiguration chain generation for SRL-based run-time reconfiguration," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2012, pp. 1–12.

[128] Xilinx Inc., "Spartan-6 FPGA configuration user guide," Xilinx Inc., User guide 380, Jun 2014, UG380.

[129] W. A. Dees Jr and R. J. Smith II, "Performance of interconnection rip-up and reroute strategies," in *Proceedings of the 18th Design Automation Conference*. IEEE Press, 1981, pp. 382–390.

[130] S. Brown, J. Rose, and Z. G. Vranesic, "A detailed router for field-programmable gate arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 5, pp. 620–628, May 1992.

[131] L. McMurchie and C. Ebeling, "PathFinder: a negotiation-based performance-driven router for FPGAs," in *Proceedings of the ACM 3rd international symposium on Field-programmable gate arrays*. ACM, 1995, pp. 111–117.

[132] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *International Workshop on Field Programmable Logic and Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, vol. 1304, pp. 213–222.

[133] J. Lam and J.-M. Delosme, "Performance of a new annealing schedule," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, 1988, pp. 306–311.

[134] M. Lin, J. Wawrzynek, and A. El Gamal, "Exploring FPGA routing architecture stochastically," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 10, pp. 1509–1522, 2010.

[135] M. Lin and A. E. Gamal, "TORCH: A design tool for routing channel segmentation in FPGAs," in *Proceedings of the ACM/SIGDA international symposium on field-programmable gate arrays*, 2008, pp. 131–138.

[136] P. Chan, M. Schlag, C. Ebeling, and L. McMurchie, "Distributed-memory parallel routing for field-programmable gate arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 8, pp. 850–862, Aug 2000.

[137] K. Fatima and R. Rao, "FPGA implementation of a new parallel routing algorithm," in *Proceedings of the IEEE Region 10 Conference (TENCON 2008)*. IEEE, 2008, pp. 1–6.

[138] A. DeHon, R. Huang, and J. Wawrzynek, "Hardware-assisted fast routing," in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2002, pp. 205–215.

[139] Y. Xu and M. A. Khalid, "QPF: Efficient quadratic placement for FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, 2005, pp. 555–558.

[140] X. Shi, "FPGA placement methodologies: A survey," 2009, dept. of Computing Science, University of Alberta.

[141] S. Lee and K. Raahemifar, "FPGA placement optimization methodology survey," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE 2008)*. IEEE, 2008, pp. 001 981–001 986.

[142] Y. Sankar and J. Rose, "Trading quality for compile time: Ultra-fast placement for FPGAs," in *Proceedings of the ACM/SIGDA 7^{th} international symposium on Field programmable gate arrays*. ACM, 1999, pp. 157–166.

[143] P. K. Chan and M. D. Schlag, "Parallel placement for field-programmable gate arrays," in *Proceedings of the ACM/SIGDA eleventh international symposium on field programmable gate arrays*, 2003, pp. 43–50.

[144] M. Xu, "Analytic methods for the FPGA placement problem," Ph.D. dissertation, Canada: University of Guelph, May 2009.

[145] M. Xu, G. Grewal, and S. Areibi, "StarPlace: A new analytic method for FPGA placement," *Integration, the VLSI Journal*, vol. 44, no. 3, pp. 192–204, 2011.

[146] M. G. Wrighton and A. M. DeHon, "Hardware-assisted simulated annealing with application for fast FPGA placement," in *Proceedings of the 2003 ACM/SIGDA 11^{th} international symposium on Field programmable gate arrays*. ACM, 2003, pp. 33–42.

[147] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, no. 3, pp. 659–681, 2006.

[148] R. Lysecky, F. Vahid, and S. X.-D. Tan, "Dynamic FPGA routing for just-in-time FPGA compilation," in *Proceedings of the 41^{st} annual Design Automation Conference*. ACM, 2004, pp. 954–959.

[149] N. McKay and S. Singh, "Dynamic specialisation of XC6200 FPGAs by partial evaluation," in *Field-Programmable Logic and Applications From FPGAs to Computing Paradigm*. Springer, 1998, pp. 298–307.

[150] M. J. Wirthlin, "Constant coefficient multiplication using look-up tables," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 36, no. 1, pp. 7–15, 2004.

[151] K. Bruneel and D. Stroobandt, "Reconfigurability-aware structural mapping for LUT-based FPGAs," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig 2008)*. IEEE, 2008, pp. 223–228.

[152] K. Bruneel and D. Stroobandt, "Automatic generation of run-time parameterizable configurations," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2008)*. IEEE, 2008, pp. 361–366.

[153] K. Bruneel and D. Stroobandt, "Troute: A reconfigurability-aware fpga router," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, vol. 5992, pp. 207–218.

[154] K. Bruneel, F. Abouelella, and D. Stroobandt, "TMAP: A reconfigurability-aware technology mapper," *Design, automation and test Europe: university booth (DATE)*, 2009.

[155] S. Bayar and A. Yurdakul, "Self-reconfiguration on Spartan-III FPGAs with compressed partial bitstreams via a parallel configuration access port (cPCAP) core," in *Proceedings of the Research in Microelectronics and Electronics, 2008. PRIME 2008. Ph. D.* IEEE, June 2008, pp. 137–140.

[156] P. Hemnath and V. Prabhu, "Compression of FPGA bitstreams using improved RLE algorithm," in *Proceedings of the International Conference on Information Communication and Embedded Systems (ICICES 2013).* IEEE, 2013, pp. 834–839.

GLOSSARY

Term	Definition
Configuration	Refers to the initial set-up of the device by loading a configuration file. Could also refer to any process where a configuration is loaded onto the device, whether dynamically of statically.
ALUT	Nomenclature used to label the LUTs contained within a Xilinx® FPGA slice.
BEL	Used with Xilinx® FPGAs to describe the functional elements that make up a component.
Bitstream	A serial string of bits used to configure an FPGA.
Bitstream encoding	The encoding used by vendors to represent FPGA resources in the bitstream.
Bitstream obscurity	A practice wherein the content of the bitstream is not made public. Also refer to "Security through obscurity".
Block-RAM	A dedicated two port memory containing several kilobits of RAM.
BLUT	Nomenclature used to label the LUTs contained within a Xilinx® FPGA slice.
CLUT	Nomenclature used to label the LUTs contained within a Xilinx® FPGA slice.
Compile time configuration	Initial set-up of an FPGA, usually taking place during start-up.
Configuration memory	A memory space defining the connectivity and initialisation of FPGA resources.
Configuration ratio	Expresses the ratio between reconfiguration time and the execution time of the application.
Configuration space	A memory space defining the connectivity and initialisation of FPGA resources.

Continued on next page

Term	Definition
Constant multiplication	Is the process of substituting the values of known constants in expressions at compile time.
Difference-based reconfiguration	Reconfiguration design flow utilising configuration data based on the differences between two designs.
Direct bitstream manipulation	The manipulation of FPGA resources by directly changing the bits in a bitstream.
Distributed arithmetic lookup table	A lookup table describing the contents of a distributed arithmetic implementation
DLUT	Nomenclature used to label the LUTs contained within a Xilinx® FPGA slice.
Dynamic circuit specialisation	A technique proposed to dynamically specialise an FPGA configuration according to the values of a set of parameters.
Dynamic partial reconfiguration	A feature of all Xilinx® FPGAs from the Virtex®-II family, allowing hardware modules to be swapped to and fro while the rest of the device remain operational.
Dynamic reconfiguration	Any reconfiguration taking place while the device is operational. Also referred to as run-time reconfiguration.
Execution time	The execution time of an application. On an FPGA, this can be calculated by considering the number of clock cycles an operation takes to complete, along with the clock period.
Frame	The smallest addressable segment of a Xilinx® FPGA's configuration memory.
Frame address	The address specified by the frame address register, which points to a specific location on the FPGA floorplan.
Full reconfiguration	The entire FPGA configuration is replaced at run-time.
Functional density	A measure of the composite benefits dynamic reconfiguration obtains over a static counterpart.
Functional density break-even point	The minimum number of times hardware should be reused before reconfiguration becomes feasible.
Generic netlists	Netlists that are not mapped to physical primitives in a target device.
Hardware-controlled reconfiguration	An architecture wherein the reconfiguration process is initialised and controlled with a hardware implemented state machine.
ICAP	An internal access port providing reads and writes to the FPGA's configuration memory.
LUT5	Component of a LUT used to describe a 5-input Boolean function.
LUT6	Component of a LUT used to describe a 5-input Boolean function.
Maximum functional density	Is achieved when execution time greatly exceeds reconfiguration time, to such an extend that configuration ratio become negligible.

Continued on next page

Term	Definition
Module-based reconfiguration	Deprecated Xilinx® reconfiguration flow wherein distinct modular sections of the device is reconfigured during run-time.
Moore's law	The observation that, over the history of computing hardware, the number of transistors in a dense integrated circuit doubles approximately every two years.
Nathan's law	The continual increase in software complexity requires more complex hardware. The continual increase in complexity of the hardware leads to more complex software being developed, the complex software continues to push the boundaries of the hardware and eventually requires more complex hardware.
Nondeterministic polynomial (NP) time	Is the set of all decision problems for which the instances where the answer is "yes" have efficiently verifiable proofs of the fact that the answer is indeed "yes".
NP-complete problem	A decision problem is NP-complete when it is both in NP and NP-hard.
NP-hard	Is a class of problems that are, informally, "at least as hard as the hardest problems in NP".
Off-line bitstream generation	Bitstreams are generated independently from the FPGA using conventional design tools.
On-line bitstream generation	FPGA configurations are generated while the device is running.
Parameterised configuration/bitstream	The configuration bits of an FPGA are abstracted to a multivalued Boolean function of a set of parameters.
Partial evaluation	Is an optimisation technique commonly found in functional programming languages. Known arguments to function calls are propagated throughout the definition of a function, yielding a new specialised function.
Partial reconfiguration	The ability to reconfigure preselected areas of an FPGA any time after its initial configuration, while the design is operational.
Quasi-static application	An application wherein the parameters change so slowly, that it resembles an application wherein the parameters are static.
Real-time application/system	An application or system which controls an environment by receiving data, processing it, and returning the results sufficiently quickly to affect the environment at that time.
Reconfigurable computing	A paradigm in computer architectures wherein the flexibility of software is combined with the high performance of hardware.
Reconfiguration	The action of modifying the content of the FPGA during run-time.

Continued on next page

Term	Definition
Reconfiguration overhead	The overhead introduced by dynamically reconfiguring an FPGA. This is either in terms of area, time added to generate new hardware, or time to transfer the new configuration to configuration memory.
Reconfiguration throughput	The maintainable bit transfer rate between the memory housing the partial bitstream and the configuration memory.
Run-time reconfiguration	Reconfiguration taking place after while the device is operational.
Security through obscurity	The use of secrecy of design or implementation to provide security.
Slack	The time between a new hardware request and the moment processing with the old hardware stops.
SLICEL	A slice type found on Xilinx® FPGAs tailored to implement logic.
SLICEM	Similar to a SLICEL, but also contains memory-orientated logic.
SLICEX	Similar to SLICEL, but does not include any carry logic.
Specialisation	The process wherein FPGA resources are adapted according to a set of parameters.
Von Neumann architecture	An instruction stream-based computing architecture controlled by a program counter.
Von Neumann bottleneck	A limitation of Von Neumann type architecture wherein the operating bandwidth is severely limited by the fact that instructions and data are fetched from the same memory.
Von Neumann syndrome	Improving the Von Neumann architecture by adding more datapaths, or increasing the clock frequency.

Stay Hungry.
Stay Foolish.