View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Waterloo's Institutional Repository

Header Parsing Logic in Network Switches Using Fine and Coarse-Grained Dynamic Reconfiguration Strategies

by

Alexander Sonek

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

© Alexander Sonek 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Current ASIC only designs which interface with a general purpose processor are fairly restricted as far as their ability to be upgraded after fabrication. The primary intent of the research documented in this thesis is to determine if the inclusion of FPGAs in existing ASIC designs can be considered as an option for alleviating this constraint by analyzing the performance of such a framework as a replacement for the parsing logic in a typical network switch.

This thesis also covers an ancilliary goal of the research which is to compare the various methods used to reconfigure modern FPGAs, including the use of self initiated dynamic partial reconfiguration, in regards to the degree in which they interrupt the operation of the device in which an FPGA is embedded. This portion of the research is also conducted in the context of a network switch and focuses on the ability of the network switch to reconfigure itself dynamically when presented with a new type of network traffic.

Acknowledgements

I would like to thank, first and foremost Dr. Agnew for providing me with the guidance to see this project through. I would also like to recognize Subbarao Arumilli whose continued assistance in providing both information and hardware resources was also instrumental in its completion. Finally, I do not want to forget Joshua Tan and Peter Gabrovsky who filled in all the remaining gaps in my work with their FPGA expertise.

Dedication

I would like to dedicate this to all my friends and family who never waivered in their patience or support throughout this whole process.

Table of Contents

List of Tables x

List of Figures xii

List of Algorithms xiv

1 Introduction 1

1.1 Problem Statement 2

1.2 Thesis Outline 2

2 Background 4

2.1 FPGA 4

2.1.1 Configuration Memory 5

2.1.2 Logic Elements 6

2.1.3 Example 7

2.2 Network Switch 8

3 Theory of Operation: Fine Grained 10

3.1 BUS and Clock Conversions 10

3.1.1 BUS Transformations 11

3.1.2 Signal Transformations 13

3.2 MainBus Interface 13

3.2.1 MainBus Write to Parser Chain . . . 14
3.2.2 MainBus to DDR2 Ram . . . 14
3.2.3 MainBus System Access . . . 15
3.3 Memory Interface . . . 17
3.3.1 DDR2 Controller . . . 17
3.3.2 On-Chip BRAM . . . 18
3.4 Dynamic Parser Configuration Arbitration . . . 19
3.4.1 ICAP Controller . . . 19
3.4.2 Parser Black-Box Wrappers . . . 21
3.5 Parser Chain . . . 22
3.5.1 Parser Chain Status . . . 24

4 Theory of Operation: Coarse Grained **26**
4.1 Parser Processor . . . 26
4.1.1 Parser Core . . . 27
4.1.2 Small Parser Core Operation . . . 30
4.1.3 Large Parser Core Operation . . . 31
4.2 Parser Interconnect Network . . . 33
4.3 Coarse Parser Programmer . . . 33

5 Framework **35**
5.1 Development Board . . . 35
5.1.1 USB Driver GUI . . . 35
5.2 General Project Structure and Coding Scheme . . . 36
5.3 Bitstream Generation Flow . . . 36
5.4 Optimization . . . 37
5.5 Testing . . . 39
5.5.1 Simulation Setup . . . 40
5.5.2 Hardware Probing . . . 40

6 Specification Analysis Results **42**

6.1 Speed of Configuration . . . 42

6.1.1 Coarse Grained . . . 44

6.1.2 Fine Grained . . . 48

6.2 Size of Implementation . . . 49

6.2.1 Static . . . 50

6.2.2 Coarse Grained . . . 51

6.2.3 Fine Grained . . . 53

6.3 Worst-Case Data Path Latency . . . 54

6.3.1 Static . . . 55

6.3.2 Coarse Grained . . . 56

6.3.3 Fine Grained . . . 58

6.4 Packet Loss Based on Configuration Speed . . . 59

7 Related Work **64**

7.1 Dynamic Reconfiguration of Network Components . . . 64

7.2 Dynamic Reconfiguration Using Custom ICAP Controller . . . 65

7.3 Improvements in the Dynamic Reconfiguration Process . . . 66

7.4 Other Forms of Reconfiguration . . . 67

7.5 Coarse Architectures . . . 68

7.6 Network Processors . . . 69

8 Conclusion **71**

8.1 Discussion . . . 71

8.2 Future Work and Suggested Improvements . . . 72

8.2.1 Coarse Grained . . . 72

8.2.2 Fine Grained . . . 74

APPENDICES **75**

A Additional Results **76**

A.1 Worst Case Delays By Architecture 76

A.1.1 Static 76

A.1.2 Coarse Grained 77

A.1.3 Fine Grained 79

B Additional Framework Details **81**

Glossary **82**

References **87**

List of Tables

3.1 Packet Parser BUS Control Flags 14
3.2 MainBus DDR2 Special Read Addresses 15
3.3 Read Control Functions 16
3.4 MainBus Interfacing Logic Status Codes 17

4.1 Small Parser Core Parameters 31
4.2 Large Parser Core Parameters 32
4.3 Single CLU Operation Truth Table 33

6.2 Coarse Grained Partial Reconfiguration Environment (Simulated) 44
6.3 Coarse Switch Bit Input Requirements 44
6.6 Fine Grained Partial Reconfiguration Environment (Simulated) 48
6.10 Protocol Header Sizes by Level 61

A.1 Static TRILL Parser Delays: Balanced 76
A.2 Static TRILL Parser Delays: Synth Optimized 76
A.3 Static TRILL Parser Delays: Logic Optimized 77
A.4 Static TRILL Parser Delays: Both Optimized 77
A.5 Static Lvl2 Parser Delays: Logic Optimized 77
A.6 Static Lvl2 Parser Delays: Both Optimized 77
A.7 Worst Delays Through Basic Parser: Balanced 77
A.8 Worst Delays Through Basic Parser: Logic Optimized 78

A.9 Worst Delays Through Basic Parser: Synth Optimized 78
A.10 Worst Delays Through Basic Parser: Both Optimized 78
A.11 Worst Delays Through Advanced Parser: Balanced 78
A.12 Worst Delays Through Advanced Parser: Logic Optimized 78
A.13 Worst Delays Through Advanced Parser: Synth Optimized 79
A.14 Worst Delays Through Advanced Parser: Both Optimized 79
A.15 Fine Grained TRILL Parser Delays: Balanced 79
A.16 Fine Grained TRILL Parser Delays: Synth Optimized 79
A.17 Fine Grained TRILL Parser Delays: Logic Optimized 79
A.18 Fine Grained TRILL Parser Delays: Both Optimized 80
A.19 Fine Grained Lvl2 Parser Delays: Logic Optimized 80
A.20 Fine Grained Lvl2 Parser Delays: Both Optimized 80

B.1 Low Delay Synthesis Profile 81

List of Figures

2.1 FPGA Example Circuit 7

3.1 Larger to Smaller BUS Conversion 12
3.2 MainBus Write Control Word 14
3.3 MainBus DDR2 Read/Write Control Word 14
3.4 MainBus Read Control Word 15
3.5 ICAP Write FSM 20
3.6 Basic Parser Chain Layout 23
3.7 MMU State Machine 24

4.1 Parser Comparator Logic Unit 29
4.2 Small Parser Core Memory Map 30
4.3 Large Parser Core Memory Map 32
4.4 Utility Config Packet: A) Interconnect Network Configuration, B) Parser bitstream starting 34

6.1 Simulated Coarse Grained Programming Speeds 47
6.2 Simulated Fine Grained Programming Speeds 49
6.3 Static Trill Parser Resource Usage 51
6.4 Advanced Coarse Parser Resource Usage 51
6.5 Base Coarse Parser Resource Usage 53
6.6 Fine Grained Trill Parser Resource Usage 54

6.7 Static Trill Parser Delay Statistics 56
6.8 Coarse Adv Parser Delay Statistics 56
6.9 Coarse Base Parser Delay Statistics 57
6.10 Fine Grained Trill Parser Delay Statistics 58
6.11 Expected Un-handled Packets During Configuration 62

8.1 Multiple Context Coarse Parser Processor 73

List of Algorithms

4.1.1 Partial Parsing State Pseudo code 28
5.4.1 Logic Optimization Example 39

Chapter 1

Introduction

As is the case with designing any other hardware with an interface to a general purpose processor using a strictly ASIC cell-based approach, when developing the hardware for a typical network switch one must keep in mind that the venues for including future upgradability are somewhat limited. The logic implemented in hardware cannot of course be modified after fabrication and any functionality which is implemented in software then suffers the speed penalty of being restricted to sequential instruction interpretation. Along these lines, if an upgrade or lateral change is required for any component critical to the core functionality of the device, such as a parser in a network switch, then the only recourse is to re-fabricate the whole device. Being that device fabrication requires quite a substantial investment of time and money, the question then becomes if there is a feasible way to allow for the future modification of a design's core components without having to expend such a significant amount of resources or sacrificing its ability to meet any speed specifications. The use of Field Programmable Gate Arrays (FPGAs) which offer the parallelization benefits of ASICs combined with the programmability of software and which can run perform operations at a rate somewhere between the two can be argued as one such solution to this problem.

The research outlined has been performed with the intent of gaining a better understanding of some the considerations that must be made in using FPGAs in this capacity. The actual reconfiguration process of FPGAs was investigated at two different levels, fine and coarse-grained, in regards to how well the resulting designs performed under various hardware engineering constraints. The actual logic tested was the parsing logic from a network switch which provided a good test-case for determining the extent to which the configuration process could be automated.

1.1 Problem Statement

The intent of the research presented in this thesis is to gauge the feasibility of implementing the parsing functionality found in the port logic of network switches with the ability to dynamically adjust to new types of traffic flow. The structure of the switching fabric within these devices can be seen as being relatively sheltered from changes in the higher level protocol changes because it is only responsible for handling packets once they have been classified and partitioned into internal cells. This stability in design allows for a greater focus on ensuring performance without as much concern for flexibility and as such the switching fabric is generally implemented as an ASIC. The control plane portion of the router is responsible for performing functions such as managing and configuring the overall system. Owing to the infrequent and, in many instances, not easily pre-defined nature of these tasks, a large portion of control plane is often implemented in software. While the required operation from this management aspect of the router hardware may be more susceptible to changes in how network traffic should be handled at higher levels of protocol encapsulation (changes in how routing tables are managed for example), a change in its functionality can usually be easily applied by modifying its instruction flow. In terms of weighing these trade-offs in flexibility and performance, the parsing logic, which must operate at speeds best suited for ASIC implementation but with the same instability in specifications reserved for software implementations, poses somewhat of a problem to the designer.

1.2 Thesis Outline

Chapter two attempts to bring the reader up to speed regarding the major concepts surrounding the research performed in this paper. It specifically covers topics on the operation of FPGAs and network switches relevant to the understanding of how the latter could be augmented with the former. The chapter wraps up with an example of how a simple Boolean statement would be implemented in a hardware look-up table which is pertinent to both grasping the underlying idea of FPGA operation and to one of the optimization techniques examined in the analysis section.

Chapters three and four collectively introduce the major components of the two re-configurable parser architectures proposed in this research. Chapter two covers the components specific to the fine-grained re-configurable parser chain along with the supporting logic which is common to both. Chapter three finishes up by only detailing the theory of operation of the components related to the coarse-grained reconfigurable parser logic.

Reading the material provided in these chapters mainly allows for a basic understanding of how it is intended that each of the dynamic parser chains fulfill its roles as a replacement for its ASIC counterpart. In going over this information it should also become clear what type of supporting logic is recommended for both debugging and programming the core logic.

Chapter five steps through the specifics of the development environment used to develop and test the proposed designs in both software simulations and when already transferred to the FPGA fabric. This chapter can be used as a basic set of guidelines for how to set up future experiments in the same field of inquest. It specifically goes through properly structuring the source code for dynamically reconfigurable designs, running the vendor provided synthesis software properly to achieve bitstreams which the FPGA hardware supports as reconfigurable, the types of optimizations which were found to achieve positive results with reconfigurable designs and an assortment of testing strategies which proved useful for these types of designs. The details of the development board used in the project are also covered in this area of the thesis but only in brief where needed to illustrate the methodology.

Chapter six discusses the methods used to analyze both the designs in the context of some common hardware engineering constraints; the results of these analyses are then discussed in terms of which of the design may be more suitable in an actual implementation. The chapter is organized by the specifications examined: speed of configuration, physical resource usage, worst-case path delay. The analysis in each case also goes over the effects of applying a number of basic optimization techniques and how they compare to those from the un-optimized tests. It concludes with a section detailing the impact of the speed of configuration constraint on the operation of an abstract network switch model created specifically for the research.

Chapter seven provides a literature review of work performed by other researches in areas which overlap with this project. Wherever relevant, the methodologies described in these works are compared to the methodologies used to separately create the two types of reconfigurable parser chains. The chapter is organized by the areas of research covered: dynamic reconfiguration of specifically network components, reconfiguration of various other circuits using the ICAP port built into the FPGA fabric, attempts at improving the efficiency of the reconfiguration process using partition based designs, other forms of partial reconfiguration (e.g. differential partial reconfiguration), dynamic reconfiguration of designs built using coarse elements and finally the use of network specific processors.

Chapter 2

Background

2.1 FPGA

While envisioned much earlier the first commercially viable Field Programmable Gate Array (FPGA) was released in 1985 with the founding of Xilinx. Not surprisingly in the relatively long time it has been around, nearly 30 years now, this technology has seen many improvements introduced to its original design in both the form of upgraded capacity and speed as well additional features such as embedded blocks of memory. Throughout its many iterations, however, the design of the FPGA has changed very little in regards to its core theory of operation which is to allow for an overall reconfigurable design using an array of configurable logic blocks interconnected by a network of switchable junctions.

FPGAs can be categorized into two major categories based on whether or not they can be reconfigured after their initial configuration which in turns relies on how their configuration is actually stored. In those that can be reconfigured the configuration state is generally stored in SRAM while in the ones that can only be configured once this data may be stored in read only flash or across a series of anti-fuses [1].

In both cases the purpose of the configuration is to not only modify the function performed at various logic sites within the design but to also to dictate how these logic sites interconnect to create the overall function output of the design. The mechanism by which the programming of the devices within each of these categories with the configuration state induces a change in their logic elements is fundamentally different, however. When it comes to SRAM based devices, the underlying core logic element which is modified during configuration is the memory of a LUT (Look Up Table), whereas with the devices that only support a single configuration this element is the switch inputs of a MUX (Multiplexer).

The intent of the rest of this section is to promote a better understanding of the implications of including a reconfigurable FPGA within an existing ASIC design in the context of the one used within this research. It first covers the basic operation of the major components within such an FPGA and then attempts to demonstrate how they function as a whole to output a logic function in an example where a simple gate described function is translated to its rudimentary FPGA equivalent.

2.1.1 Configuration Memory

In the original series of FPGAs the configuration memory was loaded into the device serially via a slightly modified dynamic phase shift register [2] which had enough stages to span all the configurable points of all the logic elements and interconnections present. The configuration data would be shifted in until the initial bit had reached the final stage of the shift register at which point additional hold logic connected at each stage would be set to feedback the stage's output back into its input. With the hold logic enabled, each stage would thus retain its value and as such the logic elements and interconnection points connected to these stages would be considered programmed.

The concept of streaming the bitstream in sequentially would be expounded on in later models in that the bitstream would be shifted in through series of registers, one for every row of logic elements. Once the bitstream was completely loaded into these memories it would then be shifted in parallel from each storage location horizontally to the configuration memories of the logic elements [3, 4].

To support partial reconfiguration of the devices as well as other features such as CRC checking of the incoming bitstream, however, the idea of strictly shifting in the bitstream was abandoned in favour of one that also includes a routing mechanism. Essentially, in the newest devices such as the one used in this research, bitstreams and other types of data are sent in broken up into chunks and appended to multiple packets containing command and addressing information. The command information contained in these header packets could, for example, let the device know that a stream of data is being sent in to configure a portion of the device and the address data then would be used to send the chunks of the stream to an entry point into a certain portion of the configuration memory. Finally, once the chunks reach this entry point they are then shifted in to make up the actual configuration memory [5].

2.1.2 Logic Elements

If the FPGA as a whole is looked at in its most basic terms as a means to output the results from one or more logical functions, the logic elements can be seen as the individual stages of the functions as chained together by the interconnects. Physically, they are placed on the device in a two dimensional array and are generally partitioned into small groups to form larger logic blocks.

While certain aspects of the circuitry in the logic elements themselves may be tweaked to allow for specializations in performing a certain subset of operations, it will always contain a set amount of LUTs and, in later devices, memory elements per device family. The individuals LUTS can be programmed to perform a number of different logical functions with an input count ranging up to the physical pin count of the component and can be combined together via one or more levels of MUXes to output functions of even greater complexity. The outputs of the function or functions performed can then be multiplexed directly to the output of the logic element to form combinatorial logic, to the memory elements to form sequential logic or to, in certain cases, components dedicated to speeding up arithmetic operations.

2.1.3 Example

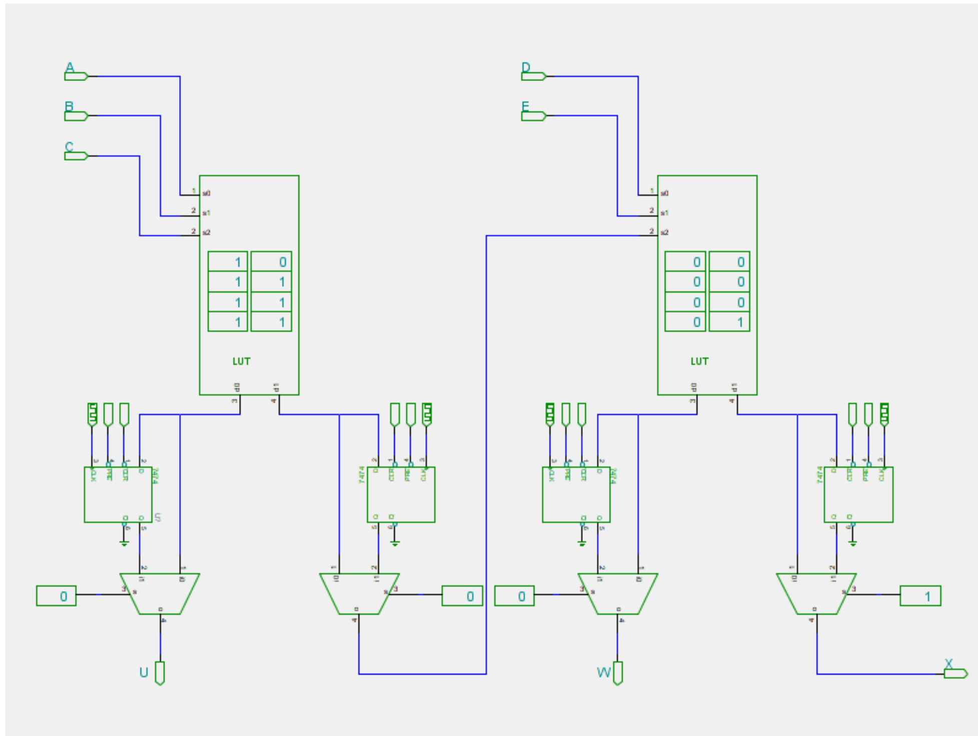


Figure 2.1: FPGA Example Circuit

The intent of this section is to demonstrate how all the pieces of an FPGA work together to produce the output of a logical function. The architecture used in the example as shown in Figure 2.1 is very basic, with LBs (Logic Blocks) composed of a single LUT and two memory elements. The interconnect network is not displayed, however, in the actual implementation it would be responsible for routing in signals to all the inputs and outputs of the two LBs.

The LUT itself is similar in structure to the ones used in the architecture of the FPGA device used in this research. It has three inputs, two outputs and, internally, is comprised of two two input, one output LUTs. The first two inputs to the component are connected in the same order to both sets of inputs of its internal LUTs while the third is connected to a MUX. The MUX allows the component to either output two separate two input functions from each of its outputs or one three input function from just its first output.

The actual function output in this demonstration is shown in Equation 2.1.1 where A, B, C, D and E are direct inputs to the two LBs of Figure 2.1 and F (Equation 2.1.2) is the logical output of the LB on the left as switched into the input of the LB on the right. The bars within each LUT represent the internal memories of the two constituent LUTs and how they are filled up with the truth tables required to perform each portion of the example function.

$$X = D \cdot E \cdot F \tag{2.1.1}$$

$$F = A + B + C \tag{2.1.2}$$

2.2 Network Switch

A network switch can be loosely described as a device which is used to transparently join together multiple network segments or groups of networked systems [6].

Traffic passing into the most basic example of a network switch is routed to its proper destination using the protocol described in the data link layer of the OSI communication model, meaning that a path is created between the sender and receiver based on their hardware addresses as sent with the data itself. In this sense the switch can associate one or more end-point hardware addresses with each of its ports and then use this mapping to determine which of these ports an incoming frame of data should be sent to. Generally, address learning, an operation defined as part of the transparent bridging method, is used to create this association whereby the port an address should be bound to is learned by extracting the source address of an inbound frame of data and then stored in a one to one mapping.

More complex switches support routing based on protocols described in higher layers of the OSI model and as such have to extract additional information from the data frame other than from the outermost header. In switch architectures like the one from which this research is derived from, the parsing of multiple levels of encapsulated data is supported by a chain of pipe-lined packet parsers each responsible for pulling out routing information from a single level. As a copy of the frame passes through the chain, each packet parser determines if it is able to interpret the data passing through it by checking for certain values within a set offset from where it started receiving the stream. If these values are found then additional fields within this offset are stored and then remaining data outside

this portion is passed to the next parser to allow it to work on the next level in the same way. Essentially this parsing mechanism is the same as the one exploited in pipe-lined network processors [7] except hard coded in ASIC hardware.

Switches can also be broadly classified as belonging to one of two categories based on the point at which they attempt to start forwarding a packet after receiving it. Switches in the first group, or store-and-forward switches, generally will first buffer in an incoming packet in its entirety and run it through a CRC check before starting the forwarding process. Cut-through switches, on the other hand, will hold off on the forwarding only long enough to receive the pertinent portions of the encapsulated headers before they start doing the same. Switches in the latter group may still perform an error check on the packet once all of its contents arrive, but strictly for record keeping purposes as by this time it would be too late to drop it if a malformation were to be found. This additional level of distinction in switch functionality is integral in modelling the expected delay through the device [8], an abstraction of which is used to quantify the impact of applying the ideas proposed in this research.

Chapter 3

Theory of Operation: Fine Grained

In the most general terms the designs created for the fine grained test-case can be described as consisting of two consecutive levels of parsing logic extracted from a network switch and inserted into a framework which provides the functionality to simulate sending a packet through the parser chain, initiate a reconfiguration of the second parser stage upon the arrival of a specific packet type and finally monitor the parser stages by taking snapshots of their status outputs. This section provides an overview of the key operational units of the fine grained design.

3.1 BUS and Clock Conversions

The design as it stands spans across four different clock domains as required by the logic for the MainBus, the parser chain, the interface to the DDR2 RAM, the PLL synchronization clock for the DDR2 controller and the internal FPGA reconfiguration port. The clock frequencies required by these logic areas are the 48 MHz,200 MHz,100MHz,200MHz and 100 MHz respectively. The logic in the design has to at one point interface with one or more of the following buses: the 32-bit MainBus, the 64-bit parser chain, the 256-bit DDR2 RAM controller input and the 32-bit internal FPGA reconfiguration port input. As such various strategies were employed within the design to ensure that when data is transferred between these various domains that it is kept at the correct level for the appropriate interval and/or aggregated or segmented to achieve the correct width.

3.1.1 BUS Transformations

Dual-clock FIFOs are used as the basis for all the multi-bit or BUS transformation schemes used in the design.

In situations where a transfer of data is required between two buses which are of the same width but originate from different clock domains, such as when packet data is transferred from the slower MainBus to the faster packet parser chain bus, the mediating logic is based solely on a subset of status outputs from the interceding FIFO. As long as the FULL output pin of the FIFO stays low, the sending component in one clock domain is presented with an endpoint which is ready to receive and, as long as the EMPTY output is low, the receiving component in another is likewise presented with an endpoint which is ready to send. Since the FULL and EMPTY outputs are synchronized to the clock domains containing the sending and receiving components respectively but also dependent on conditions from the opposite sides, the FIFO then effectively becomes not only a means to transfer data between these two domains but also a point of unified protocol translation between the contained devices. In other words, even though the sending component may be sending in control bits in parallel with the data, the only way it knows if the receiving component is no longer ready to receive is if the FIFO shows its full.

In situations where a smaller bus has to be expanded to a larger bus the speed conversion scheme described above is used but in an expanded form. One or more modulus operations are also included in the conversion process based on how many times the bus needs to be up-converted. For example, in converting a 32 bit bus to a 128 bit bus, the arriving 32 bit data is first stored in either the top or bottom word slot of a 64 bit register based on a modulus two operation. After two writes in this manner, the register is full and so its contents are transferred to a 64-bit FIFO (the largest single size included in the vendor provided libraries), causing the FIFO's EMPTY output to go low at the next clock cycle of the receiving component's clock domain. At the receiving side, as soon as EMPTY is seen as low, the same modulus two operation is used to then write two sequential 64-bit values to the upper and lower double-word slots of a 128-bit register and as soon as this register is full its contents are then transferred to the receiving component.

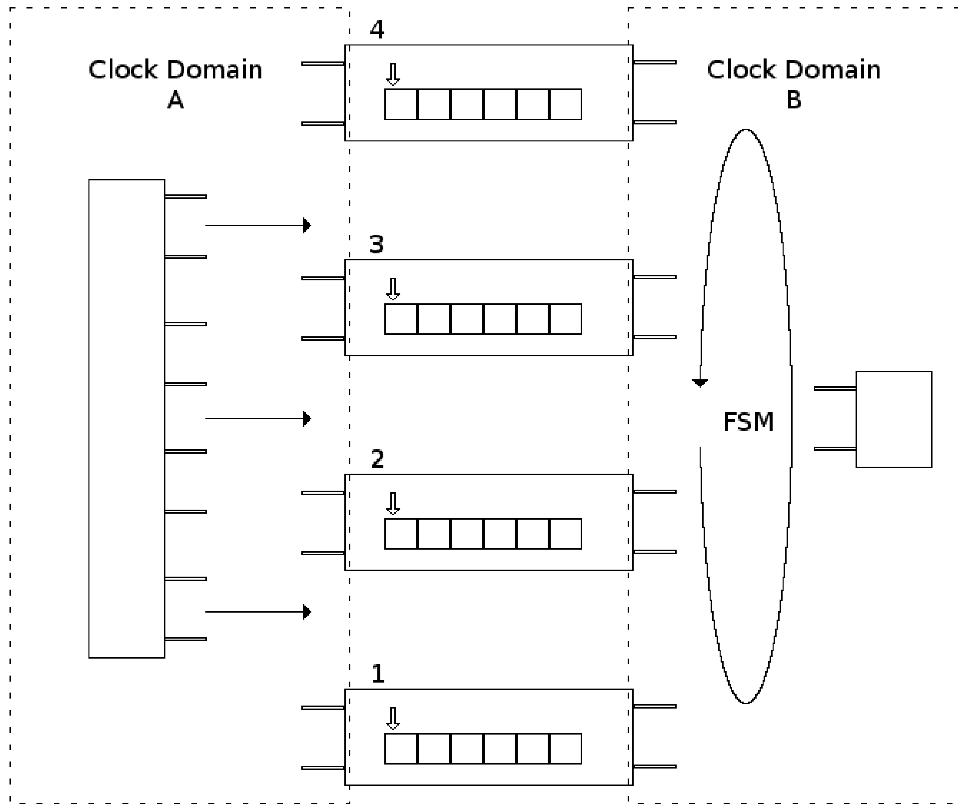


Figure 3.1: Larger to Smaller BUS Conversion

If a larger to smaller bus conversion is required then, again, the same speed conversion is used but in a slightly modified form. In this case, multiple FIFOs of the width of the smaller receiving bus are placed so that their combined input width is equal to the size of the larger bus. At every write cycle from the larger bus, all the FIFOs are written to in parallel with set segments of this value which causes their EMPTY outputs to all go low at the same time. As soon as logic in the receiving side detects that the FIFOs are no longer empty it then starts reading from each FIFO output individually in a cyclic fashion until they are seen as empty again. At each read then the data is transferred to the component on the receiving end and in this regard the data is being transferred from the larger bus to the smaller bus in a round-robin manner.

3.1.2 Signal Transformations

For the most part the components in each clock domain speak to those in other domains exclusively through intermediary FIFOs (discussed in the BUS Transformation section), however, when conditional logic absolutely has to be translated on its own, individual memory elements are employed. If a signal from a slower clock region needs to be sent to a logic operating in a faster one then the signal is passed through a series of memory elements and its value is read by the faster logic only as long as it has only propagated through the first memory element only. If on the other hand a signal from a faster operating region has to be sent to a slower one then it is run through two series of memory elements, one on the fast side and one on the slow side. The sequence on the fast side is long enough to where at least one will show high at the rising edge of the slower clock no matter where the original signal fell within the slower period and the sequence in the slower side is used as in the slower to faster conversion to ensure that only one of the faster memory element outputs is detected by the slower logic per pulse.

3.2 MainBus Interface

The development board was provided with, among other example sources, two Verilog modules which together can be used without modification to allow a design implemented in any of the FPGAs to communicate over the MainBus. The transaction demonstrated by these sources involves having a master endpoint, in the case of this project a PC connected to the board via USB, first write an address to the bus while raising a read or write flag and an FPGA endpoint then either writing data back to the bus or writing data provided by the master endpoint to an internal register. For purposes of this design, this interfacing logic is used in a significantly expanded form to accommodate a protocol which allows for read/write access to status registers from various sections of the design logic, the parser status BRAMs and the DDR2 RAM. Furthermore, while the specification for this bus allows for more than one of the FPGAs to be active and responding to requests from the master endpoint, this feature is unused in this project as only one FPGA chip is currently required in the overall design.

The intent of this section is to provide an overview of the MainBus protocol developed for this project. For its duration any references made to interactions with the MainBus should be taken as meaning that the master endpoint involved is a PC communicating with the board over its USB interface and that accordingly any data written to the bus by the FPGA is intended for this destination.

3.2.1 MainBus Write to Parser Chain

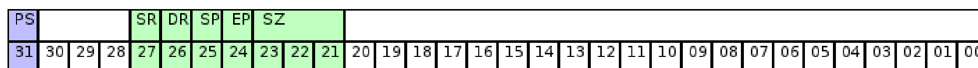


Figure 3.2: MainBus Write Control Word

The current design treats all write requests from the MainBus in which the PS flag in the address word is set low as requests to write packet frame data to the parser chain. The fields in the green coloured area of Figure 3.2 correlate to the signals of the parser bus protocol discussed in the Parser Chain section of the current chapter. Each 32 bit segment of the packet frame is transmitted to the packet parser bus in parallel with a copy of all these signals using the same clock domain conversion scheme used throughout the design. A controller module at the point of entry into the parser chain translates the control flag frame into the individual signals recognized on the parser chain bus. The controller then transmits the packet and the control signals as if it was a parser itself as long as the bus conversion FIFOs containing the packet segments and signal frames are not empty.

Name	Operation
SR	Sender is ready to transmit data
DR	Receiver is ready to receive data
SP	Start of the packet is being sent
EP	End of the Packet is being sent
SZ	Size in bytes of last transmission

Table 3.1: Packet Parser BUS Control Flags

3.2.2 MainBus to DDR2 Ram

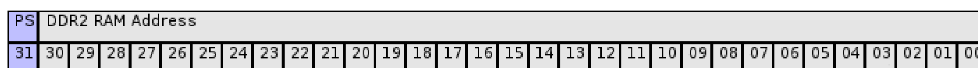


Figure 3.3: MainBus DDR2 Read/Write Control Word

Figure 3.3 shows the format of the address word which is expected by the design during a transaction between the MainBus and the DDR2 RAM memory. This event is triggered

by setting the PS (Path Select) flag within the address word high during either a read or write operation through the bus. As also shown in the illustration, the remainder of the address word is then checked for the physical address in the DDR2 RAM upon which the transfer should be based.

If a write operation is initiated by the MainBus, such as when the partial bitstreams for the parsers are being uploaded to the development board, then the data provided is written to the DDR2 RAM using the process described in the DDR2 Controller portion of the Memory Interface section in this chapter. Four such write operations have to be made to sequential addresses in the DDR2 memory before any data is actually written to the RAM module and then an additional four have to be made before the amount of data written to the memory is the same as one burst write.

If a DDR2 RAM read operation is performed then the data pulled from the egress FIFOs of the DDR2 controller is automatically stored in a series of four FIFOs for later consumption by the smaller bus using the round robin bus transformation scheme described in the BUS and Clock Conversion section also found in this chapter. The data now residing within these FIFOs can then be read onto the actual MainBus by reading from specific addresses which are reserved specifically for this purpose and which cannot be read from directly. To read back all the data returned by one read request these addresses have to be read in the correct order twice for a total of eight reads or 256 bits of data. These special addresses as well as the returned portion of data they represent are listed in table 3.2.

Address (0x)	Data Portion (bit offset)
EFFFFFFFF	127:96
EFFFFFFFE	95:64
EFFFFFFFD	63:32
EFFFFFFFC	31:0

Table 3.2: MainBus DDR2 Special Read Addresses

3.2.3 MainBus System Access

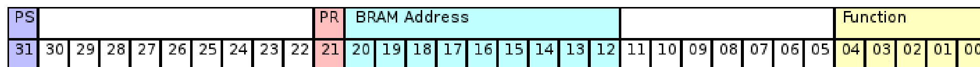


Figure 3.4: MainBus Read Control Word

When the system sees the PS flag low during a read then the address word format shown in Figure 3.4 can be used to dump information regarding the current state of operation of the design to the MainBus. Table 3.3 can be referred to for an overview of what type of information would be returned based on the code provided in the Function field.

The PR (Parser Request) flag and BRAM Address fields are used in conjunction with the NSPI operations to access the info dumps made by the parsers as described in the Parser Chain section of this chapter. The NSPI_REQRES command can be issued with the PR flag set to high to trigger a read request from the BRAM associated with the second parser stage or to low to trigger the same request from the BRAM storing info from the first parser stage. In either case then the BRAM Address field is used with this same command provide the direct offset within the BRAM selected from where to retrieve the info data from. After the NSPI_REQRES command is issued the info data is not yet presented to the MainBus as the BRAM interface is 64 bits wide. To actually place the data on the bus the NSPI_READRES commands must be used to access both the upper and lower words of the info data from temporary 64-bit register.

Name	Code (0x)	Description
FIFO_STATUS	00	Get State of operation of the MainBus interfacing logic
FIFO_LASTDATA	01	Get Last packet written to the parser chain
FIFO_VERSION	02	Get Current version of the design
NSPI_REQRES	03	Start a parser info dump read request
NSPI_READRES_HI	04	Get upper word of parser info dump
NSPI_READRES_LO	05	Get lower word of parser info dump
ICAP_LAST	06	Get last word written to the ICAP

Table 3.3: Read Control Functions

The FIFO_STATUS function is used to query the state of the protocol logic itself and return its value to the MainBus in the form of a 32 bit hex value. Table 3.4 lists all of FIFO_STATUS state codes and their meanings.

State Code (0x)	Description
DA7A2EC1	Last packet portion successfully passed to the parser chain
DA7AFA2F	Last packet portion dropped owing to block in parser chain
DA7AB10C	Last packet portion dropped for any other reason
9E7AD2E5	Last parser info dump request successfully made
ABADC0DE	Unrecognized operation requested from logic

Table 3.4: MainBus Interfacing Logic Status Codes

3.3 Memory Interface

While the large sizes of the generated partial bitstreams already necessitated the use of external memory, it was decided that the on-device BRAM memory would still be used for capturing the info output of the parsers to allow for the overall simplification of the design DDR2 memory was chosen as the external memory type as a module was already available on hand of sufficient size.

This section describes the interfaces created to access both these memory resources.

3.3.1 DDR2 Controller

In its most basic mode of operation, the DDR2 controller core requires that the data being written to it is provided in 256 bit chunks per write cycle as it itself writes 128 bits of data to the actual physical RAM at both the rising and falling edges of the clock signal. Masking, however, is allowed which means that one or more 8 bit wide subsections of the 256 bit wide write data can be set to be ignored per write. Any sequential logic directly interacting with the controller core must operate at a clock frequency as generated by a PLL (Phase Lock Loop) within the core itself, meaning that it resides within its own clock domain.

The protocol to write or read to the controller core is fairly straightforward even though quite a few inputs are required. A write is initiated by first setting both the data write enable and address write enable inputs high and then keeping the data write enable high for another clock cycle while bringing the address write enable low. During the first clock cycle mentioned the first 128 bits of the input data should be provided along with the associated masking bits and starting address while on the second clock cycle the second 128 bits should be given along with any needed masking bits but without any address. A

read request is signalled by first setting the address enable and control type (to signify this is a read transaction) high and then on the same clock cycle providing an address.

The logic handling the MainBus to DDR2 memory data transaction consists of essentially the bus up-conversion scheme covered in the BUS Transformation section, the protocol for the DDR2 controller just discussed and an additional FIFO for passing addresses and control information. The address and control bits FIFO is both written to and then read from at the same time as the 64 bit FIFO storing the data which essentially allows the data, address and control bits to be en-queued while still enforcing their association. As in any other logic handling a bus transformation in this design, the logic between the core controller and the FIFOs starts reading from these components as soon as their EMPTY outputs both show low, at which point it checks for the type of transaction required by what it pulls off of the address and control FIFO.

If the command pulled off of the address and control FIFO is a write request then the logic, as expected, up-converts the incoming data to 128 bits but then must process it through two additional steps before it is ready to be sent to the core controller. As the controller core is expecting 256 bits of data and only half of that is available at this point, the same 128 bit data is provided twice to the controller in one write and another modulus two operation is then used to mask either first 128 bits or the second. This allows the logic to spread out a write request across four clock cycles instead of the regular two when it is coupled with a modulus 8 operation to restrict the address where the data is being written to.

If, on the other hand, the command pulled off the address and control FIFO is a read request then addresses are pulled from the the same FIFO and used in the core controller read protocol until the EMPTY flag is raised. The width of the data per address returned is also 256 bits and it is stored layered in two 64 bit egress FIFOs as soon as the core controller raises its data ready flag. Any component requesting data from the RAM then is notified that the data is ready by when these output FIFOs no longer show that they are empty.

3.3.2 On-Chip BRAM

Interfacing with the BRAM on the FPGA chip can be handled at varying levels of abstraction depending on the amount of control needed over the functionality and resource usage of the resulting component on fabric. For the purpose of this design, the highest level was used as the on-chip RAM is only used to store debugging information and does not figure into the resource requirements of the parser chain. As such, the MMU logic which

is responsible for passing the output from the parsers to this memory only has to raise the write-ready signal on the rising clock edge to pass the parser info output to these RAMS.

3.4 Dynamic Parser Configuration Arbitration

To explore the use of self-initiated partial reconfiguration as a configuration option in the design a logic framework was created which both autonomously handles the reconfiguration transaction and reroutes traffic based on the configuration status of each of the parsing stages. This area of the design also contains constructs required for the partially reconfigurable design flow as well as those which allow the static portions to continue to function as intended no matter the configuration status of the parsers.

3.4.1 ICAP Controller

Initiating a transaction with the ICAP (Internal Configuration Access Port) requires following a fairly basic protocol which is simplified even further if only write functionality or partial reconfiguration is required as is the case with this design. Effectively, the process involves tying down a control input to indicate that the functionality required from the port will always be a write and then driving the clock enable input to active low whenever data is to be written to the FPGA, driving it high if there is an interruption in the data streaming in and then finally driving the clock enable high whenever the input data has finished streaming in. The data itself may be either input at a width of 8,16 or 32 bits depending on a parameter passed to ICAP module and finally care must be take to make sure it is byte-swapped [9].

The whole transaction for configuring the FPGA through the ICAP should technically be treated as a one sided affair, however, during configuration the output port normally used for reads will also automatically output the configuration status of the FPGA. This feature is used for debugging purposes in the project to monitor when the FPGA has reported that the configuration process has started and if it has detected that the configuration process has ended successfully or otherwise.

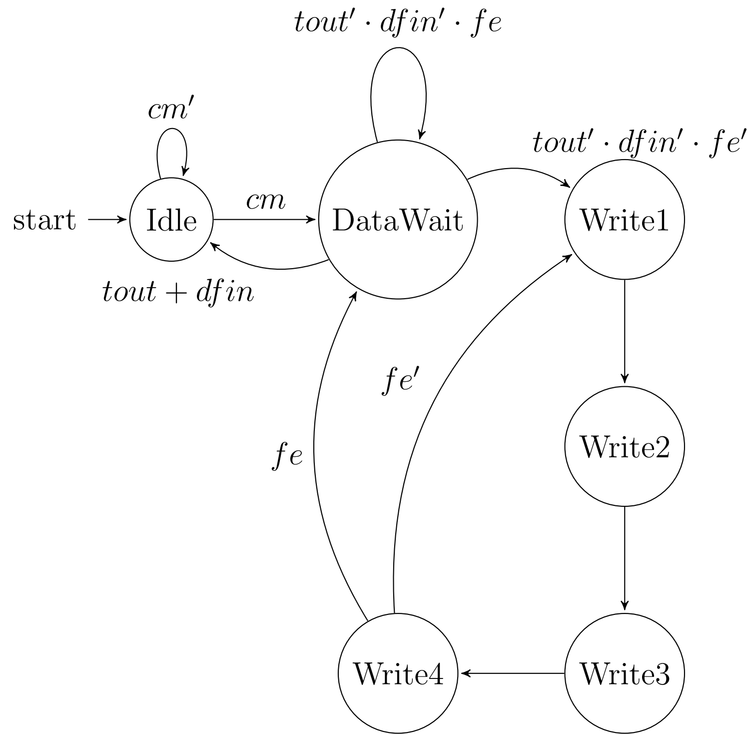


Figure 3.5: ICAP Write FSM

The first major complication which arises in regards to the configuration process with this design is ensuring that it is only triggered at the right time. The decision as to whether to program a parser stage occurs during the processing of a packet in the stage directly preceding it. This earlier stage will notify the controller of the types of packets passing through it by providing it with their EtherType fields. The controller will then in turn first check whether the next stage has already been configured to handle this type and if not will then run the field through an internal CAM memory to see if the new parser type is supported. If the CAM does not return a match then the packet is not supported and the configuration of the next stage will not take place; if there is a match then the configuration process will start and the address returned by the CAM will be translated to an offset within the RAM from where to start pulling the configuration data from.

The second significant complication which presents itself within this process is making sure that the incoming data is converted to a format acceptable by the ICAP port. As is the case with any other read request it is presented with, the DDR2 controller returns the configuration data in chunks of 256 bits at a clock rate dictated by its internal PLL.

The ICAP, however, requires that this data is handed off to it at its maximum supported clock rate and in segments of 32 bits. To accommodate this discrepancy in both width and speed, when passing the configuration data between these two components the Round-Robin FIFO approach is used as discussed in the Bus Transformation section with a depth of four.

3.4.2 Parser Black-Box Wrappers

One of the key requirements of successfully implementing a partially reconfigurable design is ensuring that all modules which are to reside within the same reconfigurable region expose a uniform and buffered IO to the rest of the logic in the design. This is accomplished by first routing the static portion of the design with black-box modules instantiated in the place of any reconfigurable logic and then separately routing the modules which are to reside in these areas, making sure that every one has the exact same buffered input and output ports. The black-box modules are simply modules which have been emptied of all logic except for the input and output ports and in this sense can be seen as place holders for any of the reconfigurable logic which may be placed inside of them.

For this design an additional wrapper level module was created to surround the required black-box modules and act as a multiplexer for the data passing through, an on/off indicator to let the rest of the logic know whether the region is configured or not and finally as a trigger to configure the next parser stage. Essentially if the inner black-box region, a black-box placeholder for a parser, is not configured or in the process of being configured then any packets arriving at the boundaries of the outer wrapper are rerouted to bypass the parser completely and the rest of the static logic is told to ignore any of its info status outputs. The trigger functionality within the outer wrapper can be seen as a packet pre-parser which notifies the ICAP controller of the packet types that pass through it so the controller can make the decision as to whether the following parser stage should be configured.

The trigger functionality of the outer wrapper is essentially a packet parser stripped of everything but its EtherType detection functionality and augmented with the ability to both suggest a configuration for the next parser stage and to store the configuration type for its parser. At the correct offsets within the packet the pre-parser will take a snapshot of the packet's EtherType within a register and then drive a wire shared with the ICAP Controller high to indicate that this configuration type is ready to be considered as a candidate for configuring the next stage. As covered in the ICAP Controller section, the ICAP Controller will then check with the EtherType stored by the trigger of the next

stage to see whether the configuration process should be started. The configuration type of a stage is stored in the form of a registered EtherType and is empty (all signals in the bus are low) if the parser contained within the wrapper is un-configured.

3.5 Parser Chain

To allow for the possibility of eventually testing the design within the actual data plane architecture of the network switch it was pulled from, the outer wrapper for both the parsers was designed to present the same bus and protocol as was expected by the original interfaced components. As such the outer wrapper includes a variable length info status bus with a wire to indicate that the info is ready, two 64 bit buses for transmitting packets to and from the parser, a send/receive ready input/output for the previous stage and a send/receive ready output/input for the next stage. The protocol used between the component which are connected with this setup is fairly basic and involves really only checking whether both the sender is ready to send and the receiver is ready to receive before sending the data at each clock cycle. Additionally, a start and end of packet signal are also interchanged to simplify the calculations the receiving parser has to do to calculate the offset of the data within the packet which it is currently receiving. The protocol does support a supplementary extension in the form of an error in transmission signal but, as it is unused in the switch architecture from which this project is derived, the associated pins on wrapper are currently always left open.

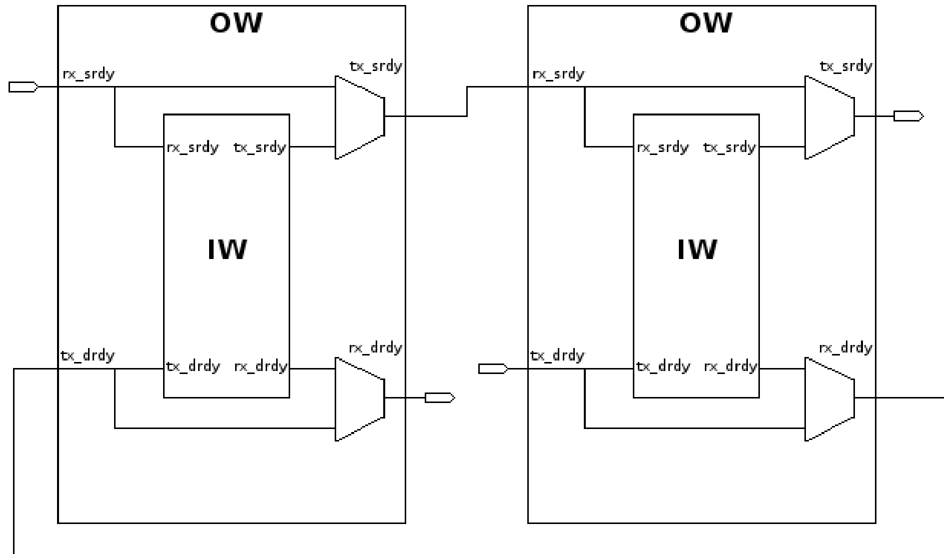


Figure 3.6: Basic Parser Chain Layout

The intent of the multiple parser stage setup used both in the data plane of the source switch as well as in this design is to provide a compartmentalized parsing engine where in each stage handles just one level of packet encapsulation before stripping it off and passing the remainder of the packet to the next stage. Whether a parser stage supports extracting meaningful data from the level of encapsulation it receives is based on whether key fields within the header such as the EtherType match the type it is configured for. These field also determine whether the layer of encapsulation should be stripped off before passing the packet to the next stage and in this sense the packets can be just forwarded along unaltered if they are not supported.

In the original switch architecture, the level 2 parser is used in conjunction with several others to handle the extraction of all the pertinent data link layer fields from the Ethernet frame. It is mainly responsible for extracting the source and destination MAC addresses for forwarding purposes. Once it extracts these values, it passes the remainder of the Ethernet frame header along with the payload to the other data link layer parsers down the chain which in turn determine if the header is embedded with any additional fields related to VLAN and SNAP functionality. In the research presented, none of these extra features were tested and as such only this initial level 2 parser was required to represent the processing the packet would go through at its level.

3.5.1 Parser Chain Status

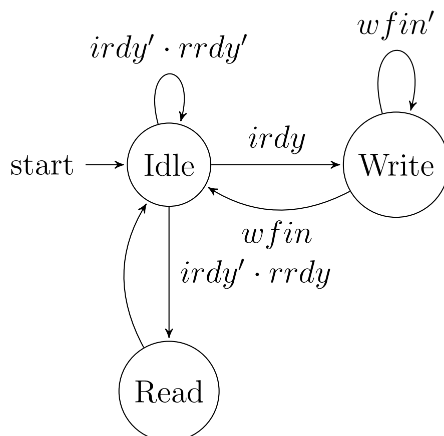


Figure 3.7: MMU State Machine

Each parser effectively outputs its status through a series of info buses shortly after it receives a packet and has had time to process it. To take advantage of this feature, a basic MMU (Memory Management Unit) was created which sequentially dumps the output of the info bus for each packet received to the next free location on one of two dedicated BRAM (Block RAM) memories on the FPGA. After a series of unique packets have been sent through the parser chain, the same MMU can be used to access each of the BRAMs at varying offsets. In this sense, the point at which the switch was able to start processing a new packet type in a packet stream is determined by comparing the data sent in to the data registered at each offset by the parser being reconfigured as well as any other in the chain.

While the width of the info bus on the outer parser wrapper must always be the same as necessitated by it being a reconfigurable module, the actual widths of the info buses of the parsers contained within the wrapper may vary and so the MMU was designed to be flexible in the actual amount of data it writes to allow for more efficient use of the on chip memory and to speed up the process when possible. Referring to the FSM in Figure 3.7, when a parser indicates to the MMU that info is ready with the irdy signal it also passes a width value during the transition between the Idle and Write states. During this same transition the MMU divides the value by the bus width of the BRAM instances used in the design, 64, and then compares the new value against an up counter in the Write state to determine whether to continue writing or not as represented in the FSM by the wfin signal.

During the write state the info value is right shifted by 64 and its lowest double word is stored at the memory address shown in the up counter. Finally, during the transition from the the Write state back to the Idle state the last value of the up-counter is stored as the new address offset from which to start writing the next packet dump. The MMU can run two such processes concurrently for each of the parsers in the chain.

The read back of these values which is initiated by the NSPI_REQRES MainBus initiated operation discussed in Section 3.2.3 can only occur if the MMU is currently not in a Write state and has lower priority if the MMU gets both requests at the same time. Only one clock cycle is required to return the value requested after which the MMU flags the data is ready and the MainBus interfacing logic stores it for later access by additional commands.

Chapter 4

Theory of Operation: Coarse Grained

The coarse grained test-case is built using the same framework as used in the fine grained one and differs only in the implementation of the parsers and the interceding buses. This chapter covers only the abstracted functionality of the operational units which have been changed between the designs.

4.1 Parser Processor

While both the fine grained and coarse grained architectures are built on top of FPGA technology and are therefore configured at the LUT level, in regards to the reconfiguration process of the parser chain itself, the parser processors are considered the base unit of configuration in the latter implementation. Throughout most of the parsing process the actual operation being performed on the packet data can be generalized as being either a conditional or unconditional extraction of data from varying offsets within the stream. As such the coarse grained processor has been designed as a general purpose compare and extract engine which can be programmed to perform this operation against multiple values both stored and found.

Beyond the parsing state, a parsing element only has two other modes of operation, idle and passing the extracted payload to the next stage, both of which entail standard processes largely independent from the parsing one. This fact was exploited in the processor to decrease the size of the programming memory required by hard coding these states and extracting the one responsible for parsing as a module which is referred to as the core of the parser from this point on. In this sense the processor logic consists of a dynamic

state machine, the parsing state of which can be extended to include additional sub-states depending on the amount of parsing cores inserted.

4.1.1 Parser Core

Currently, two different sized parsing core elements have been designed for this research, a smaller one which can be programmed to have the equivalent parsing capability of both a level 2 and TRILL parser in the original architecture of the switch and a larger one which can be configured to function as half of a level 3 parser as well as the two parsing levels already mentioned. As soon as it receives notification to enter the parsing state, each parser core can be seen as running through a process which always starts with a check for the type of packet passing in, then continues through a variable amount of comparisons and finally finishes after passing on info extracted at certain offsets. The actual determination of details such as at what overall offset each of these steps should take place and if the packet parsing should continue if a comparison fails a constraint is handled by a set of flags stored in 128 bits of internal configuration memory. The only major differences between the two core sizes, in terms of this flow, lie in the amount and flexibility of the comparisons which can be performed (the initial data type check is also considered a comparison in this sense) and the formatting of the flags within the memory.

The pseudo code in 4.1.1 outlines the general flow of a core's parsing process and how it is shaped by the flags stored in the parsing element's configuration memory. The IF statement starting on line 4.1.1.10 specifically shows an abstraction of one possible setup of the comparisons step discussed earlier. When the counter initialized on line 4.1.1.3 reaches the first compare count stored in the configuration memory, the configured comparison type is triggered between two fields. If the comparison fails then the error flag is set, the parsing procedure is stopped and the core notifies the rest of the parser logic to pass the remainder of the un-extracted packet to the next parser in the chain. The results of the comparison can be masked to show as always passing if the extraction of the info from the packet is desired unconditionally.

Algorithm 4.1.1 Partial Parsing State Pseudo code

```
1: err ← false                                  ▷ error encountered
2: fr ← false                                   ▷ finished returning info
3: pctr ← 0                                     ▷ parser counter
4: c1r ← 1    ▷ flags that the parser should care about the outcome of this comparison
5: while ¬packet end ∧ ¬fr ∧ ¬err do
6:     if ¬sender ready then
7:         continue
8:     end if
9:     pctr ← pctr + 1
10:    if compare 1 count ∧ compare 1 failed ∧ c1r then
11:        err ← true
12:    end if
13:    if end of info count then
        PassInfo(Info)
14:        fr ← true
15:    end if
16: end while
```

In the case of the smaller parser core the comparison type can only be an equality during the initial type check and the first and second fields can only be of type extracted and stored respectively. With the larger parser the comparison type can be set to an equality or either of the strict inequalities (less than or greater than) and can involve multiple fields both stored in the configuration memory and those extracted. Furthermore, whereas the smaller parser can only perform one comparison at a time the larger parser can perform a range of between one to three comparisons simultaneously via an embedded chain of programmable (configuration memory space shared with the core) hardware comparators illustrated in 4.1.

Beyond the on state in which the parsing states are embedded the parser core has two other states of operation, off and being programmed. Both of these states can be triggered at any time by supervisor input pins and both are, by design, preemptive of any parsing functionality. When the core is in the off state, it cannot be triggered into starting the parsing process by the ready input flag from another core sending data to it and as such will prevent the core from consuming any of the packet until it has been fully configured. When the core is in the programming state it treats the data coming in as configuration packets (see 4.3), meaning that it will store all data coming in at this time in its configuration memory. After all the configuration data has been received, the core will then go through a basic initialization sequence, entailing of mainly the expansion of certain

received variables, and then transition to the off or on state depending on the status of the supervisor pins..

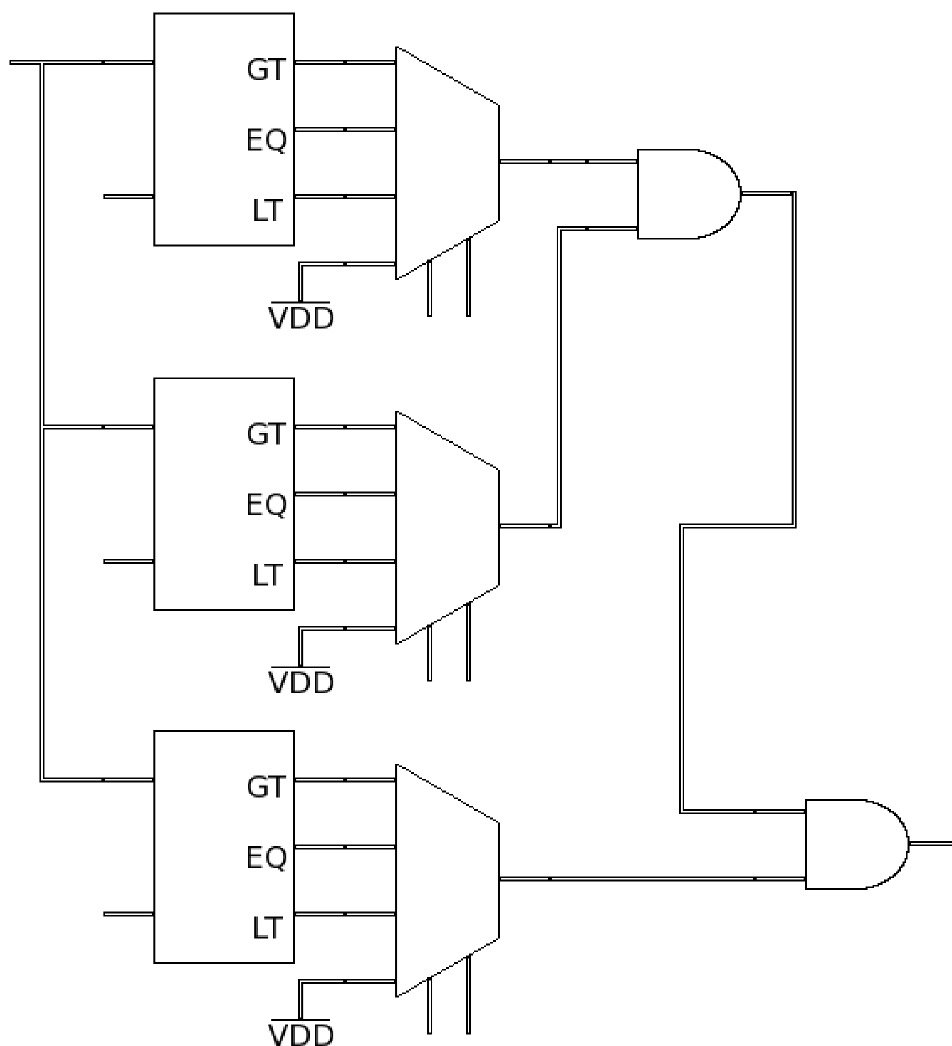


Figure 4.1: Parser Comparator Logic Unit

The on state of the core is also transitioned to by way of the input supervisor pins and consists of both an active and idle parsing sub-state. The active sub-state consists of the process described in pseudo code covered earlier and is, as mentioned earlier, triggered by an external core. Stored state variables in the configuration memory determine what

mode of operation the parser logic external to the core should go to in both the case where parsing process exits with packet data remaining and the case when it exits because the packet ended early. Currently in the former type of exit the external logic is always set to move to passing the data on to the next parser and in the latter it is set to go idle, but this extensibility is included in the design to allow for future research into mapping efficiency versus granularity.

4.1.2 Small Parser Core Operation

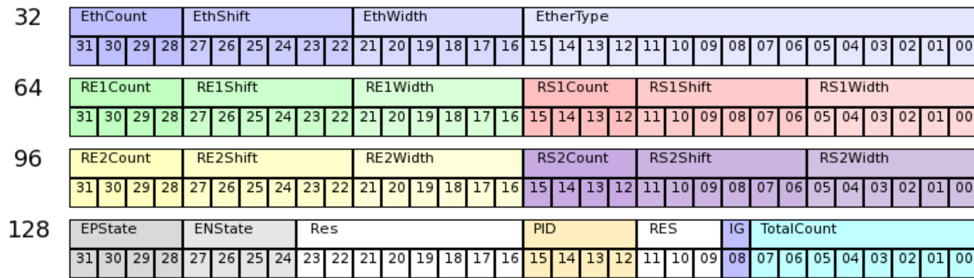


Figure 4.2: Small Parser Core Memory Map

The primary function of the small parser core is to simply pull out one or two consecutive groups of fields from a packet, apply basic formatting to the extracted data and then send it up for consideration by higher functionality in the network switch. The core can either be configured to pull out this data unconditionally or based on whether it detects that a field extracted at a certain offset matches a stored value. This check is mainly intended to serve as the EtherType check during the level two or TRILL (can be considered level 2.5) parsing of data sent over Ethernet. If the core is set to extract info unconditionally then it will always return the checked value on a separate bus from its info bus at the same time it returns the info, but if the core is set to extract conditionally then it will only return the checked value and the info on their respective buses if the checked value matches the stored value. The level two parser uses the unconditional setup while the TRILL parser uses the conditional setup of this parser class.

Figure 4.2 illustrates how the small parser core interprets its configuration memory. The fields prepended with **Eth** are responsible for determining where to look for the the data type check and the **EtherType** field itself is the value against which this check should be performed. As the data actually returned may be greater than a 64 bits, start and end

locations have to be provided to take in account for these larger return types for both the first and second info groups designated as **R1** and **R2**. The significance of the rest of the parameters responsible for driving this core are explained in table 4.1.

Name	Use
EtherType	Initial data field value to run check against
Width	Amount of bits to extract for a given field in a specified data chunk
Shift	Offset of data to be extracted from the end of a specified data chunk
Count	The data chunk to check for a given field
TotalCount	Total amount of chunks to parse before passing the packet
IG	Whether to conditionally or unconditionally extract info
PID	Parser ID to match against during programming
ENState	State enumeration to visit if packet is not finished after parsing
EPState	State enumeration to visit if packet is finished before parsing is complete

Table 4.1: Small Parser Core Parameters

4.1.3 Large Parser Core Operation

In the larger core the ability to pull out more than one group of fields has been sacrificed, along with the ability to dynamically set the end states and finally the resolution of some of the other fields, for the ability to perform additional checks against the incoming packet. This core can programmed to pull out two additional fields beyond the initial value check field and to store two more fields in memory all for the purpose of performing one additional more complex check. The second check is driven by the comparator logic unit shown in figure 4.1 which can be programmed to perform one to three checks against the first of these values pulled with the other three. This additional check is included for the ability to ensure that the Header Length field in IP packets is between two values and that it is less than the Packet Length field. As in the original parser architecture if any of these checks fails then the parsing core releases the packet to the remainder of the logic for forwarding to the next parser. The first check is used in context of IP packets to check for the correct version so for example to ensure that an IPv4 level 3 parser can handle a packet, the EtherType field is set to 0x4. After these two check the parser can only perform an extraction of info and as such cannot act as a full level three parser as additional fields related to packets fragmentation still need to be handled.

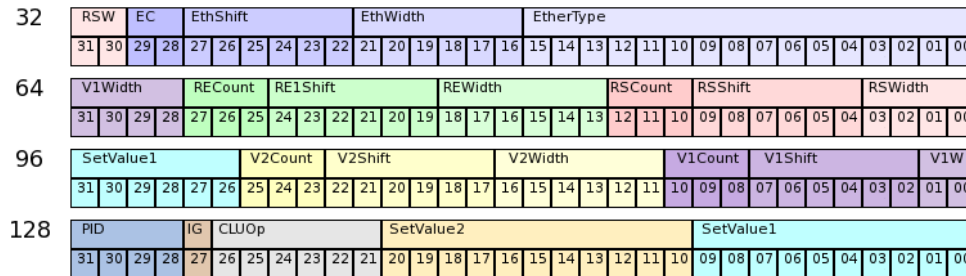


Figure 4.3: Large Parser Core Memory Map

Figure 4.3 illustrates how the large parser core interprets its configuration memory. The fields prefixed with Eth and R are responsible for finding the first check and the return values respectively and serve the same function as in the small parser core. The fields starting with **V1** are required to find the first value in the packet against which all the remaining values will be checked, both found (V2) and stored (SetValue1,SetValue2). The purpose of the rest of the parameters responsible for driving this core are outlined in table 4.2.

Name	Use
EtherType	Initial data field value to run check against
Width	Amount of bits to extract for a given field in a specified data chunk
Shift	Offset of data to be extracted from the end of a specified data chunk
Count	The data chunk to check for a given field
IG	First check mask
PID	Parser ID to match against during programming
CLUOp	Configuration Memory of CLU

Table 4.2: Large Parser Core Parameters

Each two bits within the CLUOp are wired to the inputs of one of the MUXes in the Comparator Logic Unit (CLU) of the large parser. As can be seen in table 4.3 which displays the truth table used in all of the comparison levels within the CLU, these bits can be used to either select one of the outputs of a typical hardware comparator or logic high. The purpose of the logic high selection is to allow that particular level of the CLU to be masked off and effectively then ignored.

S1	S2	Result
0	0	1
0	1	lt
1	0	eq
1	1	gt

Table 4.3: Single CLU Operation Truth Table

4.2 Parser Interconnect Network

In the same vein as the actual underlying hardware of the FPGA, the ports of the base logic elements in the coarse grained parser implementation can be routed to each other in a flexible manner using a programmable interconnect network. Owing to the relatively large granularity of the design, the interconnect network needs only a small amount of configuration memory, 12 bits, to manage the routing between the elements. Each four bits of the configuration memory is responsible for setting the input mapping of either of the two parsers or of the whole network itself.

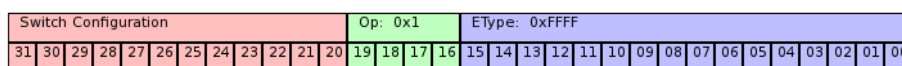
4.3 Coarse Parser Programmer

The coarse parser programmer is responsible for managing the the transaction of data from external memory to the configuration memory of the base logic units of the coarse architecture and as such can be seen as an analogue of the ICAP port in the fine grained design. As with the ICAP port it listens for a data ready signal from the external configuration memory source and then interprets the following stream of data from this source as a series of configuration packets. Unlike the ICAP port, though, it can carry out the programming operation at the system clock rate and therefore interfaces directly with the DDR2 RAM controller via one additional controller with no additional FIFOs. Moreover it can operate on data chunks of up to 64 bits per clock cycle, double the maximum supported by the native hardware configuration port.

The configuration of a parser in the coarse grained architecture is triggered in exactly the same way as in the fine grained one, via matches of EtherTypes extracted from the packet stream against stored addresses in a CAM memory (3.4.1). The configuration packets sent to the parser programmer must have a specific formatting and be sent in the correct order for the configuration of a parser to take place, otherwise the process

is cancelled and the remainder of the configuration data is ignored. Two basic types of configuration flows are currently supported by the programmer, one that involves the set up of one or more parsers and the interconnection network and another which involves just setting up one or more parsers. In either case, the packets sent are expected to be ones designated as utility packets which always start off with the hex sequence 0xFFF and contain instructions for the programmer or ones carrying the operational image of a parser but only if prepended by a properly formatted utility packet.

A)



B)

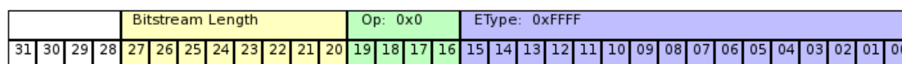


Figure 4.4: Utility Config Packet: A) Interconnect Network Configuration, B) Parser bitstream starting

Chapter 5

Framework

5.1 Development Board

The development board used is a DiniGroup DN9000K10PCIE4GL which, at its core, houses six interconnected Virtex-5 xc5vlx330 FPGAs. While the board provides numerous methods of interfacing with the FPGAs both for programming and for general use IO [10] , the resources utilized in this project included only one FPGA designated as FPGA A, the USB programming/IO interface, the exclusive 32-bit interconnecting bus designated as the MainBus, a SODIMM DDR2 module exclusive to FPGA A, the on-board Compact Flash reader and the JTAG port.

5.1.1 USB Driver GUI

The development board is provided with cross-platform open source drivers for its general purpose USB interface that, while mainly intended as a convenient channel for programming any of the on-board FPGAs, were also coded to facilitate a number of other communication options between the board and a development PC. The USB programming interface was only used at the onset of the project to test the functionality of the static portion of the logic as it does not support sending partial bitstreams, however the driver feature which allowed for transmission of data to and from the FPGAs was used extensively to not only simulate the sending of packets through the parser chain but to also read status information from design and to send partial bitstreams to the DDR2 RAM.

5.2 General Project Structure and Coding Scheme

The design is constructed in a hierarchical scheme with the top level consisting of a module which houses the modules for all the major logic areas of the design, their interconnecting buses and any constant values. Each of the major sub-modules themselves in turn contain a mixture of procedural blocks, structural interconnects and modules as needed to define their functionality. The mapping of the outermost ports of the top-level module to their actual physical counterparts on the FPGA as well as any specifications in regards to actual physical characteristics such as signal strength and SLEW rate are defined in a constraints file.

During pre-bitstream-generation testing or simulation the top level module is instantiated in a special Test Bench module which is also populated with simulation specific constructs used as analogues for hardware components the design will interact with once loaded on the board.

The project is completely coded using the Verilog-2001 HDL dialect save for a script generated CAM (Content Addressable Memory) as provided by Xilinx which is written in VHDL.

5.3 Bitstream Generation Flow

After its header has been stripped and the rest of its data byte-swapped, the bitstream file is a direct representation of the bits which will be passed to the configuration port of an FPGA. In general to take a Verilog or VHDL source file and build it into a bitstream the software in the generation flow first creates a netlist in which the HDL constructs have been converted to references to their hardware primitive constituents, from which it creates another one in which the hardware primitive references have been mapped to the type of hardware resources which they will consist of (Flip-Flops, LUTS, etc) and finally one in which the hardware resources are placed and routed so they now represent specific components on the device as interconnected to others [11].

Before a Partial Reconfiguration (PR) flow can be initiated, first multiple standard design flows must be used to generate the netlists for each of the reconfigurable regions separately from the one for the rest of the design in which their logic has been replaced with black-box place holders. The software related to the PR flow then can be used to create a project in which all the different configurations for each reconfigurable region can be bound to the black-box place holders which match their IO ports. The actual slice

usage of each of the partitions has to then manually be determined by either writing out the physical constraints in the UCF constraints file or by using a software GUI to mask out the slices on a rendition of the FPGA chip. Once the partitions have been successfully instantiated using the steps discussed, they can then be associated with any number of pre-generated netlists which are expected to be run these areas. Finally the software can be used to to build multiple profiles based on the permutations of all the various modes of operation for the partitions, run a timing analysis on each of these incarnations and output the associated bitstreams.

The software binaries associated with the bitstream generation flow can be run individually in turn from a command line interface or can be triggered to automatically run from start to finish via a GUI interface. For the sake of convenience the GUI interface was used in building all portions of the design used in this research except for the initial generation of the netlists describing the reconfigurable region which require a stricter level of control better provided by the command line interface.

5.4 Optimization

The following strategies were used in an attempt to optimize the designs in terms of speed and size to a level beyond what would be achievable with those used by the synthesis software by default:

- Primitives instead of higher constructs such as comparators, etc.
- Case statement instead of nested IFs
- Advance synthesis optimization directives, e.g. register balancing
- Timing constraints and analysis
- Location constraints and floor-planning

The use of primitives and case statements are collectively referred to as logic optimizations in the remainder of this thesis and were profiled by writing them into separate copies of the relevant source files and then switching them in as needed to test their effectiveness. The advanced synthesis optimizations were applied by enabling a slightly modified version of a built in speed profile for the synthesis software which tuned the options passed to the algorithms used in each stage of building the bitstream to seek a faster final design (the build options modified are listed in table B.1 in the Appendix). The timing constraints were applied to both test then the effectiveness of applying one or both of the optimization

strategies just discussed and to constrain certain delay sensitive paths in the actual fabric. Finally, the location constraints were used primarily to optimize the partially reconfigurable design in that they were used to direct the synthesis software to place the partitions in a fabric location favourable to minimizing the delay between each other and the rest of the logic.

The process that was used to apply the logic optimizations is demonstrated in algorithm 5.4.1 which is a pseudo code translation of one of the optimizations actually tested on the design. In the optimized version of the code, the prepention of the "LUT" marker to an input signal name indicates that the logic the signal was involved in before entering the condition has been replaced with one or more LUT primitive instantiations whose final output now serves as the new input. The process by which the logic is mapped into a LUT is the same, in concept, as the one used by the synthesis software to translate an HDL statement into native resources on the FPGA fabric. In fact, unless the synthesis software optimizes the primitive to reside in a different number of resources then the LUT memory entered along with the primitive into the HDL will be directly written to a native LUT of the same size in the final netlist. The manual conversion of a Boolean statement into LUT memory in this fashion is covered in more detail in section 2.1.3 of the introduction.

Algorithm 5.4.1 Logic Optimization Example

```
1:                                                    ▷ Unoptimized
2: if prdy then
3:     if ¬cexit then
4:         if ∧cetype ∧ cop = OPSWITCH then
5:             ns ← SCSWITCH
6:         else
7:             ns ← SCPROC
8:         end if
9:     else
10:        ns ← SCIDLE
11:    end if
12: else
13:    ns ← SCSTART2
14: end if
15:                                                   ▷ Optimized
16: sigbus[3 : 0] ← prdy, ¬cexit, cetype.LUT, cop.LUT
17: case sigbus of
18:    0b1111 :
19:        ns ← SCSWITCH
20:    0b1100 ∨ 0b1110 :
21:        ns ← SCPROC
22:    0b1000 ∨ 0b1001 ∨ 0b1010 ∨ 0b1011 :
23:        ns ← SCIDLE
24:    default :
25:        ns ← SCSTART2
26:
27: end case
```

The motivations and implementations of the remainder of the optimizations are discussed further in the final analysis chapter as they are used to interpret the results obtained.

5.5 Testing

Testing of the design took place in two stages, the first in software through a simulator included in the bitstream generation flow and the second through hardware via a JTAG

debugging interface, the USB communication framework discussed in section 5.1.1 and the basic LED output facilities of the board.

5.5.1 Simulation Setup

To get the best possible idea of how a design will actually interact with the peripheral hardware connected to the IOs of the FPGA before it is loaded on the board, vendors provide simulation models which make extensive use of signal delay constructs in an attempt to recreate the actual propagation delays expected on the physical device. These can be combined in a module with simulation exclusive time triggered changes in signal values and memory values pre-loaded from external files to create what is commonly referred to as a testbench for the design.

For this particular design a testbench was created with simulation models for both the specific make of DDR2 memory used on the development board as well as for the MainBus interface. At the onset of each simulation run the CAM memory was pre-loaded with the EtherTypes required to trigger the configuration of various parser types. Once the simulation was under way, the contents of another memory file was shifted in through the MainBus interface to recreate the action of sending packets through the parser chain so that the triggering of the reconfiguration process could be observed. The waveforms produced from these simulation runs were used to debug and ensure that the design was logically sound as well as to acquire the initial programming speed estimates discussed in the analysis chapter.

5.5.2 Hardware Probing

The JTAG debugging interface was used exclusively to tap into the data transaction between the ICAP port and the DDR2 memory; as such, the final partially configurable design contains JTAG probes only on the write enable and data input ports of the ICAP primitive as well as on its status output bus. The probe on the write enable port along with the one on the data port are used to show whether the bitstream data arriving to the ICAP primitive is properly synced to the ICAP configuration controller logic or not. The probe on the data port itself is also used to determine whether the data coming in is in the correct format (proper endiness, correct offset, etc.) and that it has not been corrupted in either the read or write process to or from the DDR2 memory. Lastly, the use of the probe on the status output port during programming is fairly self explanatory and involves simply checking that the fabric configuration port has detected the bitstream start, not

exited early because of a perceived error in the bitstream and has properly detected that the bitstream is finished.

The USB communication framework and the shifting in of the binary packet file covered in the simulation setup section both represents the same debugging operation as seen in hardware and in software respectively. Essentially the testing process involved in this interaction with the parser architectures consists of, initially, sending in two packets via write requests to the MainBus, the first to trigger the reconfiguration process and the second to allow the newly configured parser a chance to parse a packet. Once the two packets have been sent, a series of read requests are sent to the MainBus which trigger a read-back of the contents of what each parser level extracted from the second packet from the on-chip BRAM. If the expected data is not returned then several other types of read requests are sent to determine the current operational status of both the MainBus to parser interface and the DDR2 memory controller. In the hardware case all these steps are initiated by inserting the data into the correct fields of the USB Driver GUI and then having it send the information in context of either a read or write via a USB connection to the actual physical MainBus controller on the board.

The LED lights of the board are used in the designs as a last line of defence to debug intrinsic failures which would prevent even the other two mentioned hardware testing methods from displaying useful results. In this sense the lights are tied mainly to the status outputs of the inter-clock domain FIFOs and to key status bits of the major controllers. The FIFO LEDs allow for mainly the checking for stuck or residual data as it is passes between clock domains in that they are tied to the FULL and EMPTY lines of the FIFO primitives. The controller LEDS are primarily used to check whether a particular portion of logic is stuck in a particular state. One of the controller LEDs for the DDR2 Memory controller is, for example, useful for showing whether the PLL in this particular controller was able to sync up with the clock of the on-board DDR2 RAM, without which the whole design is rendered useless as the DDR2 controller is unable to communicate with the external memory.

Chapter 6

Specification Analysis Results

The metric of suitability of any proposed hardware design in a particular application is usually constrained by the design's ability to meet base specifications related to, among others, worst-case latency and size on chip. If, however, the design is expected to be able to have its functionality dynamically changed, then additional considerations have to be taken into account such as the amount of memory needed to store its configurations and the amount of time required to switch between these modes of operation. The intent of this chapter is to compare the performance of the fine grained parser chain versus that of the coarse grained parser in these areas, using a design which does not support partial reconfiguration as a baseline.

The efficacy of applying the optimizations discussed in section 5.4 is also examined in this chapter in terms of its impact on meeting the design goals mentioned. To this end, a duplicate profile was created of the parsers with the optimizations included and was run through the same tests.

6.1 Speed of Configuration

The amount of time required to transition a reconfigurable design between its configurations is mainly determined by the following factors:

- Size of the configuration file
- Throughput of the external memory storing the configuration file

- Granularity of minimum programming area
- Throughput of the internal configuration memory controller

The two design types researched in this project were first run through simulations to determine the ideal speeds with which partial configuration could take place. In both cases, the simulation was run through a testbench which is responsible for programming a vendor provided simulated core of the DDR2 memory with a segment of a bitstream and sending packets through the parser chain to trigger reconfiguration. The delta time measurement utility included in the waveform output aspect of the simulation software was used to measure one transaction, or an initialization and transfer of 256 bits of the bitstream from the DDR2 memory to either the input of the ICAP port or the coarse programmer. The programming speed then was calculated by dividing the bitstream sizes by 256 and then multiplying the clock cycles used for the transfer.

The initialization delay has to be included in the calculation of the amount of time required for each transfer cycle as the RAM to configuration memory transfer protocol built into both designs involves syncing both the write to the configuration memory and the request for the next chunk of bitstream from the RAM to the data ready signal from the RAM controller. While this protocol precludes the use of the burst read capability of the DDR2 memory used, its use is required so that the controller in either design scenario can reliably end the configuration process when it detects that the bitstream has ended. As discussed in the theory of operation chapters, the controllers must check for values within the data pulled from the DDR2 memory at any given read before it can make the determination that another one should be made.

$$R = \frac{B_W}{D * 8} \tag{6.1.1}$$

Where,

B_W : Total Bits written per transfer between RAM and configuration controller
D : The delay in seconds of the transfer

6.1.1 Coarse Grained

	Clock (MHz)	Bus Width (bits)
Coarse Programmer	210	64
Config Controller	210	127
DDR2 Controller	250	127
Config Transfer Delay	19.568ns	
DDR2 RAM Read Delay	85.274ns	
Programmer Write Delay	23.705ns	
Total Write Delay	128.547ns	
Effective Config Throughput	248.93619 MB/s	

Table 6.2: Coarse Grained Partial Reconfiguration Environment (Simulated)

Figure 6.1 shows the configuration delay in clock cycles for a single coarse parser core to be programmed both by itself and along with the interconnect network when the simulated environment is set up with the parameters listed in table 6.2. The delay for programming just the core is the same as total time required for one read transaction from the DDR2 RAM plus the that required to transfer the bitstream data from the configuration controller to a core. This is the expected result as the memory controller accesses data from the RAM chip in 256 bit quad-words when bursting is not used which is enough to encompass the size of both the configuration memory of the parser core and the initialization header of the bitstream. If the interconnect network is to be programmed as well, then one additional read request is required and the time required jumps to a value a little over double the initial value. The extra cycles present in this next read are representative simply of the configuration controller having to complete the last read before moving on to the new one.

parsers	internal	io
1	1	
2	$2 * (1)$	2
3	$2 * (2)$	2
4	$2 * (4)$	3
5	$5 * (3)$	3
6	$6 * (3)$	3
7	$7 * (3)$	3
8	$8 * (3)$	4

Table 6.3: Coarse Switch Bit Input Requirements

Table 6.3 shows how the required configuration memory size of the current switching network in the coarse design is impacted by adding additional parsers to the chain. The second column represents the total bits required by all the internal switches leading to each of the parsers while the last column reflects the bits required for the final output switch from the parser chain. The actual total required bit size per parser count can be calculated then by adding the internal switch column to the output switch column or by using Equation 6.1.2. Accordingly, it can be observed that the portion of the coarse parser bitstream dedicated to programming the switching logic of the chain would grow to a size of 28 bits if eight parsers were to be added. Even with the addition of this many cores, the configuration controller in the coarse architecture would only still need to make one read request to pull in the data required to program this area of logic as this still leaves its configuration memory load under 128 bits.

$$\begin{aligned} B_{io} &= \lceil \log_2 (P+1) \rceil \\ B_{int} &= \lceil \log_2 (P) \rceil \\ B_{ts} &= B_{io} + P * B_{int} \end{aligned} \quad (6.1.2)$$

Where,

B_{io} : config bits required for the output switch
B_{int} : config bits required for a parser input switch
B_{ts} : total config bits required for all switches
P : parser count

In regards to the reads required for pulling in the actual core configurations, these would grow by one for every two cores added in addition to the first core implemented because the same request needed to read in just one additional core configuration would also pull in the configuration memory for the switching network. The previous example can be extended to illustrate the progression in size of this portion of the bitstream in that the seven additional parsers added would necessitate adding three reads more to program the total collective core configuration memory.

The total amount of reads required per additional parser programmed is found then by aggregating all of the loads on the bitstream discussed so far and can be modelled with equation 6.1.3. The amount of clock cycles needed to read in a coarse parser configuration bitstream based on the number of cores to be programmed could then be modelled by multiplying the result of this equation by the cycles needed for one read.

$$
\begin{aligned}
R_i &= \left\lceil \frac{B_{su} + B_p}{B_r} \right\rceil \\
R_p &= \left\lceil \frac{P}{2} \right\rceil \\
R_{ts} &= \left\lceil \frac{2 * (B_{ts} + B_h)}{B_r} \right\rceil \\
R_t &= R_i + \begin{cases} R_{ts} - 1 & \text{if one core with switches} \\ R_p & \text{if multiple cores without switches} \\ R_p + R_{ts} - 1 & \text{if multiple cores with switches} \end{cases}
\end{aligned} \tag{6.1.3}
$$

Where,

- $R_{i,p,ts}$: Memory read counts related to the initialization, parser configuration and switch configuration packet sizes respectively
- R_t : Total amount of reads required to configure the coarse architecture
- B_r : config bits pulled in by one read (256 with DDR2 memory)
- B_p : config bits needed to configure one parser core (128 with current architecture)
- $B_{su,h}$: config bit sizes of the initialization packet and the header portion of the utility packet respectively (B_{su} is a utility packet itself)

Equation 6.1.3 however only works under the assumption that the total amount of parsers being configured is the total amount of parsers in the whole chain so it can not be used to predict the aggregate cycles required for multiple configurations of individual parsers in a much larger chain. If modelling the read cycles necessitated by the latter scenario, then one of two slightly different approaches must be taken based on what subdivision of the total coarse configuration memory space a particular bitstream targets and the span of time over which the programming is taking place. In the event that the new parser is being inserted into the parser chain as a substitute for another then the previous equation can be used but if it is being inserted to a parser chain as an additional parser then it has to be inserted along with details regarding how it changes the switch memory space in the context of all the other parsers. Equation 6.1.4 has to be used then to model this additional complication which can be seen, in fact, as the coarse configuration equivalent of the minimum configuration frame size in fine-grained partial-reconfiguration.

Subset P_{ss} of parsers being configured out of total parsers P together:

$$R_t = R_i + \sum_{i=1}^{P_{ss}} \big(R_p(i)\big) + R_{ts}(P) \tag{6.1.4}$$

Subset P_{ss} of parsers being configured out of total parsers P separately:

$$R_t = \sum_{i=1}^{P_{ss}} \big(R_i + R_p(1) + R_{ts}(P)\big) \tag{6.1.5}$$

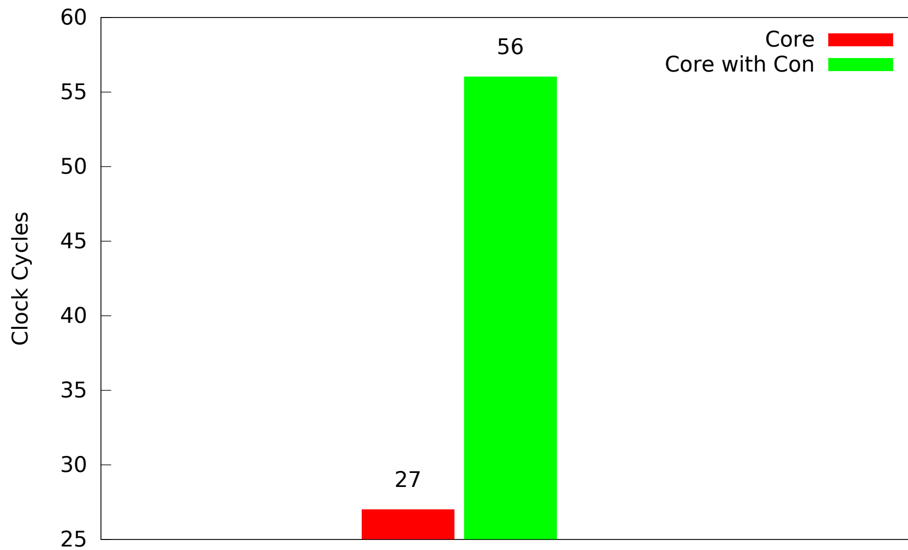


Figure 6.1: Simulated Coarse Grained Programming Speeds

6.1.2 Fine Grained

	Clock (MHz)	Bus Width (bits)
ICAP Port	157	32
Config Controller	315	127
DDR2 Controller	250	127
Config Transfer Delay	64.878ns	
DDR2 RAM Read Delay	92.858ns	
ICAP Write Delay	50.784ns	
Total Write Delay	126.960ns	
Effective Config Throughput	252.04789 MB/s	

Table 6.6: Fine Grained Partial Reconfiguration Environment (Simulated)

Figure 6.2 shows the configuration delay in clock cycles expected when using partial reconfiguration to program the smallest frame sized allowed, the parser partition and the whole device in the simulated environment in the simulation environment shown in table 6.6. Unlike with examination of the configuration speeds of the coarse architecture which were observed directly through the simulation environment, the results here are presented as estimations found with equation 6.1.6. The bitstream sizes for both the parser partition and the device wide logic were found simply by having the development environment OS report the actual bitstream file sizes as generated by the synthesis software for the designs used in this research. As the feasibility of breaking up the design into reconfiguration frame sized (40 SLICEs or SLICEMs) partitions was never investigated directly, the size of the reconfiguration frame was found instead by using the physical placement functionality of the synthesis software to create a frame sized partition in an otherwise empty design.

$$T_t = \frac{B_t * T_r}{B_r} \tag{6.1.6}$$

Where,

B_t : total size of the bitstream
T_r : delay in clock cycles for the transfer of data from external memory to the ICAP primitive
B_r : config bits pulled in by one read

Even just from the sizes of the bitstreams of the fine grained architecture themselves, it already becomes apparent that the delay imposed by having to use this form of reconfiguration would be in the order of magnitudes larger than that of the method used in the coarse architecture.

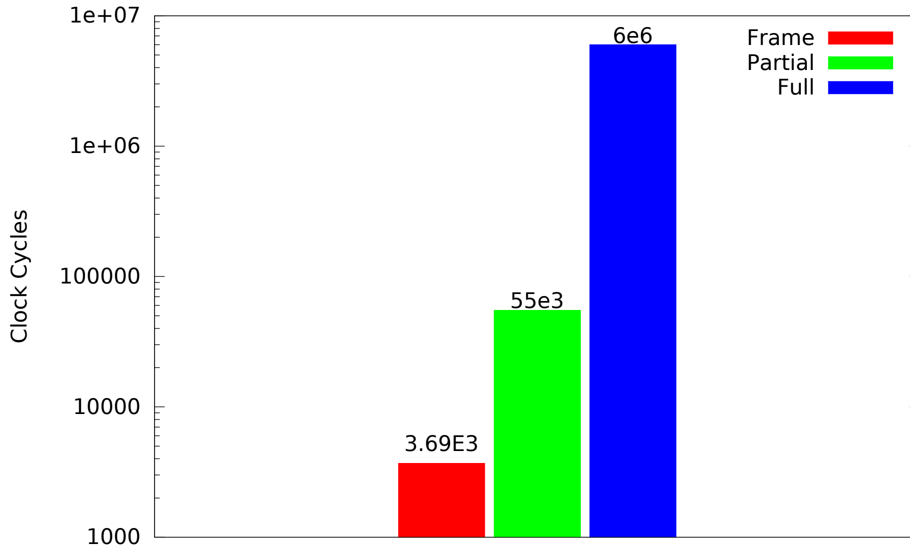


Figure 6.2: Simulated Fine Grained Programming Speeds

6.2 Size of Implementation

As part of the base optimization process, the synthesis software will often absorb or outright delete wires and other elements if it determines there is a more efficient way of routing the logic while retaining the same end functionality. This behaviour poses a problem in terms of calculating the area used of individual logic modules as it essentially may blur the line of where one module stops and the other starts. A synthesis directive may be used to preserve the borders of a module, however, this same directive has to be used to create the partitions in a partially reconfigurable design and as such, if used in all cases for size estimation, would at worst always provide results consistent with a reconfigurable design or at best under-report the size. This limitation was partially overcome to garner estimates of the changes in size by measuring how much resources were used by the logic as a whole with each iteration. These results should provide an accurate enough preliminary idea of

the parser chain size trend between designs because the rest of the logic always remains unchanged in the hardware description.

6.2.1 Static

Figure 6.3 demonstrates the varying effects each optimization method used in this research had on the overall LUT and FD/LD usage of the static design. As the graph shows, applying only the logic based optimizations to the Trill parser results in the lowest overall usage of LUTS among all the other optimization strategies and the base line. Applying this same optimization does not change, however, the memory element count from the base line, which is to be expected owing to the fact that it only targets the combinatorial elements of a design. Running the design through the low delay profile of the synthesis software also decreases its overall LUT usage but not to the same extent as the logic only optimization. The synthesis only strategy does however increase the memory element count moderately. The moderate increase in this sort of resource with this strategy could possibly be as a result of the employment of register duplication along a number of the data paths in an attempt to decrease fan-out. When both strategies are used on the design then the resource usage in both cases mirrors that of the synthesis only optimization but with a very slight shift down the y-axis.

The large decrease in area achieved in the static design by applying the logic only optimization is possibly explained by the nature of how IF-ELSE-IF and CASE statements in Verilog are inferred differently by synthesis software even if coded to be semantically the same. According to older literature released by the vendor of the FPGA chip used in this research [12], when IF-ELSE-IF statements are used the logic which is ultimately placed in hardware is a multi-level priority-encoder whereas when an equivalent CASE statement is used then the logic may be able to be placed in as little as a single, wide multiplexer. While this resource is a bit dated, its recommendation to then use CASE statements in place of IF-ELSE to improve the timing of the design for the afore mentioned reasons is still brought up in newer timing closure guidelines by the same vendor albeit with no explanation as to why [13]. Even though these guides focus on the timing benefits of decreasing the levels of logic with one type of statement versus the other, the obvious implications regarding the area savings of using one over the other is what should be seen as relevant to the results in this section.

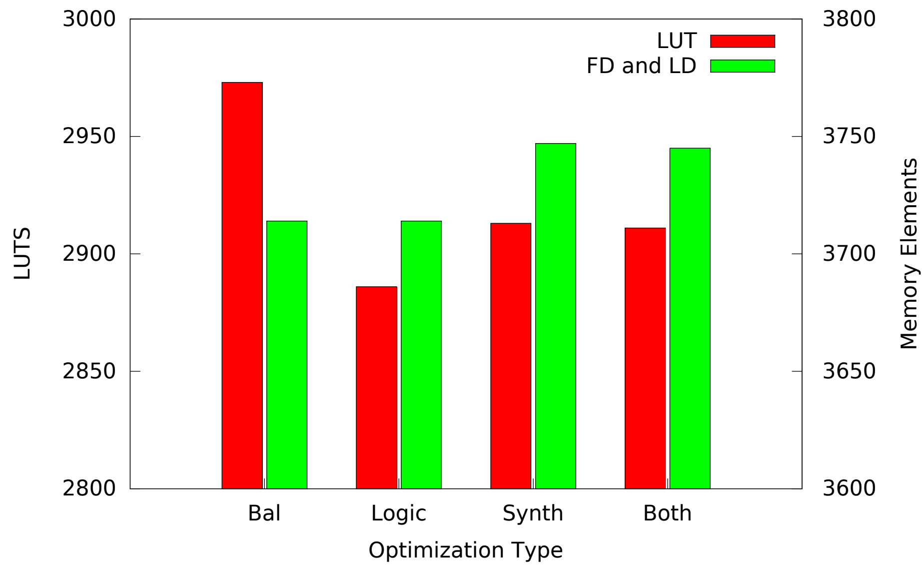


Figure 6.3: Static Trill Parser Resource Usage

6.2.2 Coarse Grained

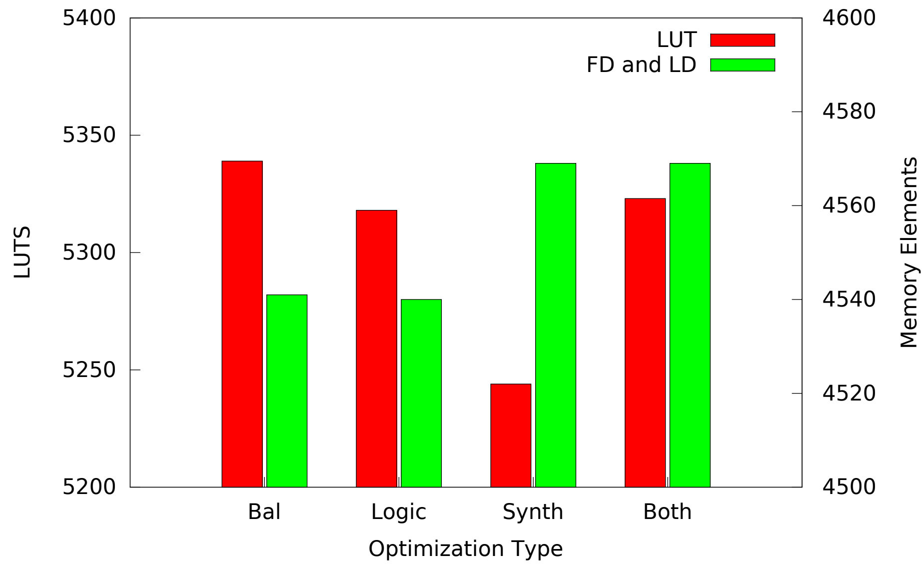


Figure 6.4: Advanced Coarse Parser Resource Usage

Figure 6.4 and 6.5 present the varying effects each optimization method used in this research had on the overall LUT and FD/LD usage in the advanced and basic forms of coarse parser processor based designs. In regards to the effect of applying logic only based optimizations the resource count of both coarse based designs respond in the same way as in the static design but in a much less pronounced manner. In this case it appears that the designs respond much better to the application of the low delay profile in regards to the actual LUT count, lowering this resource usage to the lowest out of all the other strategies. The use of the synthesis only strategy on the coarse designs does nonetheless still increase their memory element count to a moderately higher level than with the base line. Interestingly in also deviating from the overall optimization trend, when both the optimization schemes are applied to the coarse designs then their LUT count rise sharply from their synthesis only to a point approximately mid-way between the base line and the logic optimized counts.

The relatively small benefit gained from performing logic optimizations on the coarse grained architecture may be explained by the fact that this design relies heavily on shift operations to allow the programability of the offsets at which it checks for values within the packet (refer to section 4.1.1 for details on how this is programmed). The analysis of these shift operations by the hardware ultimately dictates then what portion of the packet gets passed along to the next processor which in turn is handled by an additional level of shifts. When synthesized with the balanced or default profile of the synthesis software used, these multiple levels of shift operations on the packet data are translated into an expanding series of LUT columns which can each modify the offset of a portion of a packet by feeding its value to the next column of LUTs at varying offsets based on control values. As explained in the framework chapter, the optimizations only target the switching of bus data and as such do not effect the expression of these LUT based logical shifters.

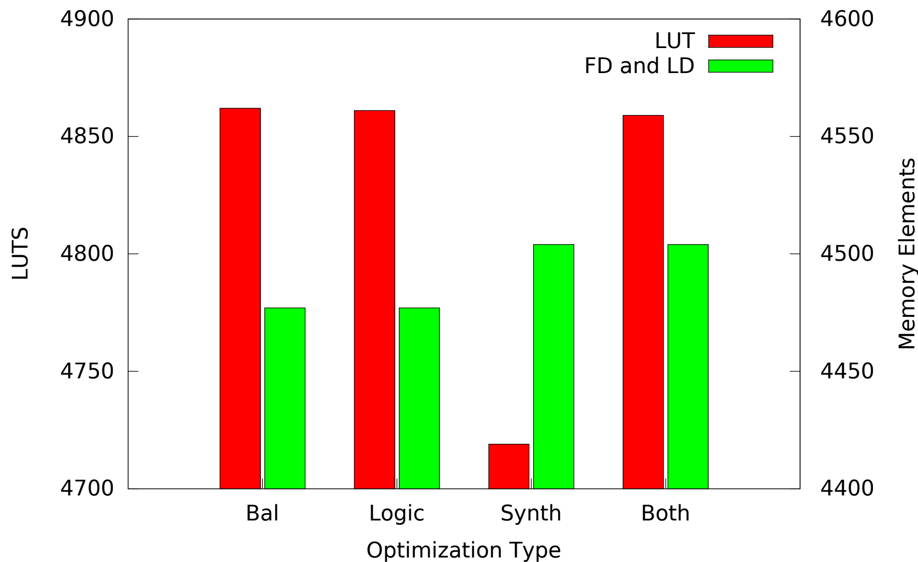


Figure 6.5: Base Coarse Parser Resource Usage

6.2.3 Fine Grained

Figure 6.6 demonstrates the varying effects each optimization method used in this research had on the overall LUT and FD/LD usage of the fine grained design. In this case the results seem to show that partially reconfigurable designs seem to respond poorly in terms of area to any attempts at logic optimization. This issue also appears to be compounded further when both optimization techniques are used on these types of designs as the LUT count rises to its highest level among all the other results. The only optimization technique which appears to have a positive effect on diminishing the area usage of the fine grained design is the one involving only the application of the low delay profile in the synthesis software.

In first analyzing the area usage results from the application of various optimization schemes to the synthesis of the fine grained design, the expectation was that the area usage would follow the same pattern as in the static design results but modestly shifted on the y axis to slightly larger values. Upon closer examination of the log file from the synthesis run, it was determined that at least in part this extra resource usage in the logic optimized design is as a result of the software keeping the manually placed primitives even when their outputs lead to open ports. In the base design without the logic optimizations, these primitives do not exist and the synthesis software is free to remove these ports and

their associated logic. While the same problem exists in the static design, it appears that without the partition constraint placed on these areas of logic as found in the fine grained design, the synthesis software is able to compensate for this deficiency.

A more accurate comparison of the area usage between the static and reconfigurable designs could be derived in future work by either forcing the synthesis software to keep these signals in all these ports on both runs or by removing the manually placed primitives in the logic optimized designs associated with the ports automatically removed in the base line ones.

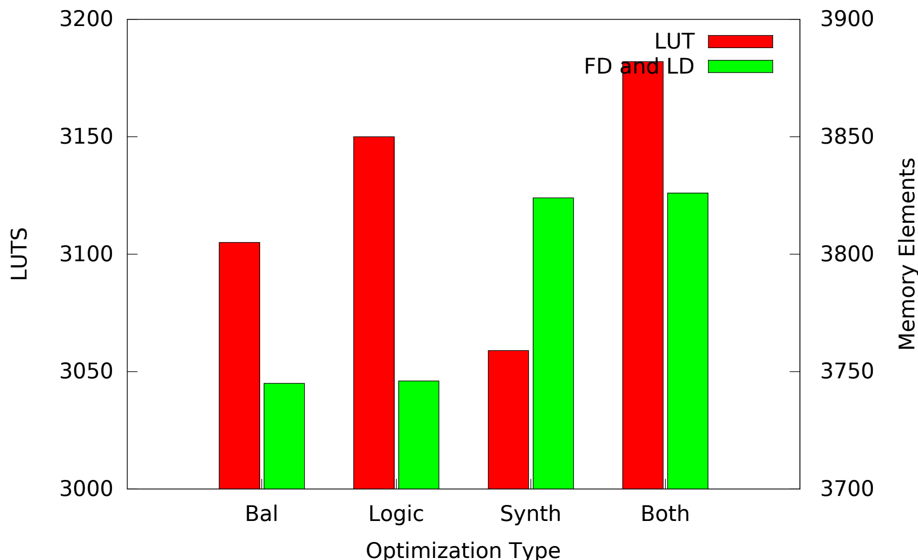


Figure 6.6: Fine Grained Trill Parser Resource Usage

6.3 Worst-Case Data Path Latency

The worst-case latency through the parsers or conversely the fastest frequency that the parser logic could run on its own was measured by placing a timing path constraint on the path taken by the data bus as it travels from the input of the first parser to the output of the second parser. This constraint is normally used to modify the mapping and placing flow of the synthesis software to place emphasis on attempting to make the delay between the sequential end points (such as D-Flops) of a path smaller than a specified value. Regardless of whether this constraint was able to be met during synthesis, the positive and/or negative

skew from the delay required is reported at the end of the process as it relates to the setup time requirements of the end point sequential element. This feature was exploited to simply report the delay of the parser data path with as little impact on the optimization process as possible by using it to constrain the signal with the same value as required to meet the period constraint of the overall system clock.

6.3.1 Static

Figure 6.7 demonstrates the varying effects each optimization method used in this research had on the expected delay of the data path through the TRILL parser when embedded in the static embodiment of the design researched. As can be seen, only applying the low delay profile to the synthesis software flow decreased the levels of combinatorial logic the data had to pass through but increased the overall delay as compared to the base line of no optimizations. Applying only the logic based optimizations to the circuit of the parser appears to have little effect on the maximum logic level count of the parser but does, on the other hand, decrease the actual propagation delay significantly. Finally, the application of both optimization schemes to the static configured parser seems to produce a parser with a propagation delay which is close to the average between the un-optimized design and the logic optimized design but with a drastically reduced logic level count.

The ability of the logic only optimizations to decrease the propagation delays in the static design may be most likely explained by a decrease in logic levels at switching points and the reasoning is explained further in section 6.2.1. The maximum logic level did not decrease along with the delay as may be expected from the trends in the graphs alone but why this occurs can be easily understood by referencing the additional results in the appendix. After the optimizations have been applied the logic paths that they effect are no longer reported as the highest delays and so the paths then with the next highest delays which they did not or only marginally effected are shown. These new highest delay paths then may or may not have the same logic depth as those that benefited more from this optimization and are able to do so while having a lower delay than the latter by virtue of having a lower fanout delay, a lower intrinsic logical effort delay or both per stage.

The synthesis software can be forced to show the details of the lower ranking delay-wise paths which allows the examination of how exactly any given path has changed before and after optimization. This extra data was not extracted as it would mainly be relevant to showing how exactly the routing algorithm responds to these optimizations, the analysis of which is beyond the scope of this research.

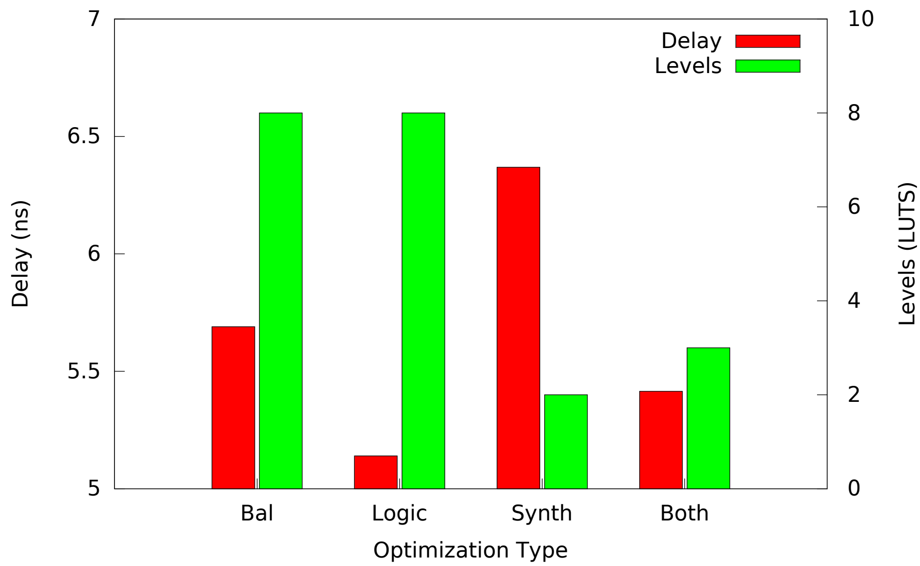


Figure 6.7: Static Trill Parser Delay Statistics

6.3.2 Coarse Grained

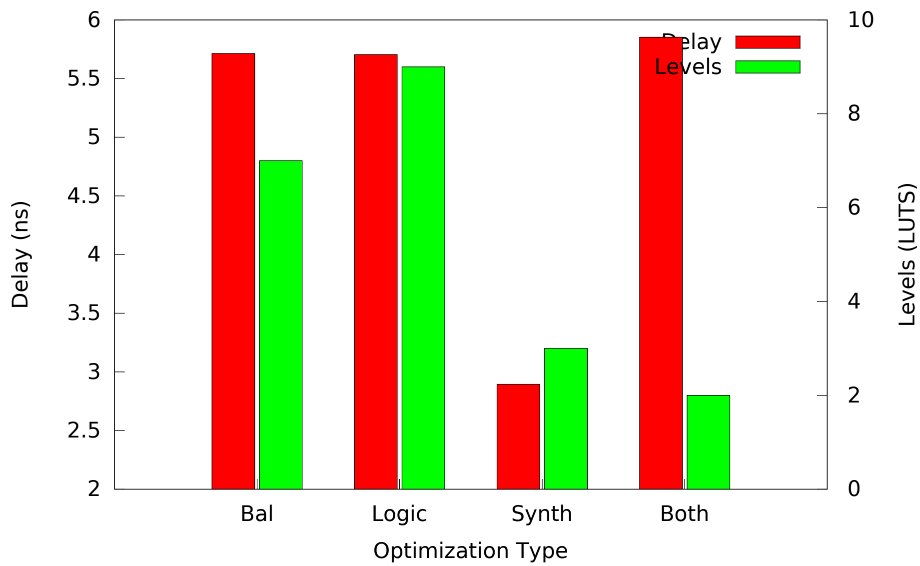


Figure 6.8: Coarse Adv Parser Delay Statistics

The propagation delays through both the advanced and basic coarse reconfigurable parsers are discussed in this subsection; their delay trends across optimization schemes are shown in figures 6.8 and 6.9 respectively. Unlike in the other two designs, the parsers designed for this reconfiguration method do not respond well to only logic optimizations, showing only a very slight decrease in propagation delay and logic level count that is at best the same as in the un-optimized design and at worst increased. Even more notable is the fact that both these parsers responded to synthesis optimization to a much higher degree than in either of the other designs and to the point where the propagation delay fell to almost half of the lowest in the static design. Beyond this, however, the coarse parsers responded to the application of both optimization schemes in the same way as in the other two designs. Furthermore, as would be expected, the advanced parser displays a higher overall propagation delay and logic level count than with the base parser.

As discussed in the results section for area usage owing to the focus of the logic optimizations on conditionally routing signals and buses, very little could be expected in regards to delay reduction in applying them to the coarse grained design. Additional delay optimization venues are discussed in the future work chapter of this thesis.

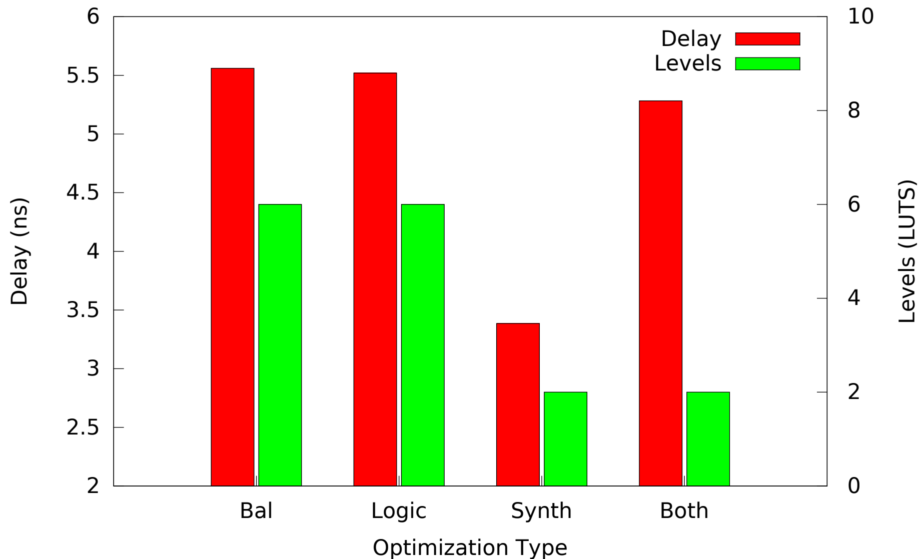


Figure 6.9: Coarse Base Parser Delay Statistics

6.3.3 Fine Grained

Figure 6.10 shows the trend of the propagation delays through the TRILL parser embedded in the fine grained reconfigurable design in response to the application of the different optimization schemes examined in this research. Both the delay and logic level count through the base-line un-optimized design is slightly higher than that in the static design but like in the latter the parser responds well to logic optimization, showing a significant decrease in both logic levels and total combination logic delay. Also as in the statically configurable design, the application of both optimization schemes decreased the logic levels even lower while slightly increasing the delay from the logic optimization scheme to a value close to the average between the logic scheme and the base-line. While applying only the low latency profile to the synthesis software does produce a lower logic level count in the parser, the cross design trend in delay progression is adhered to in that the delay actually rises to a point where it does not even meet the clock period requirement for the overall design.

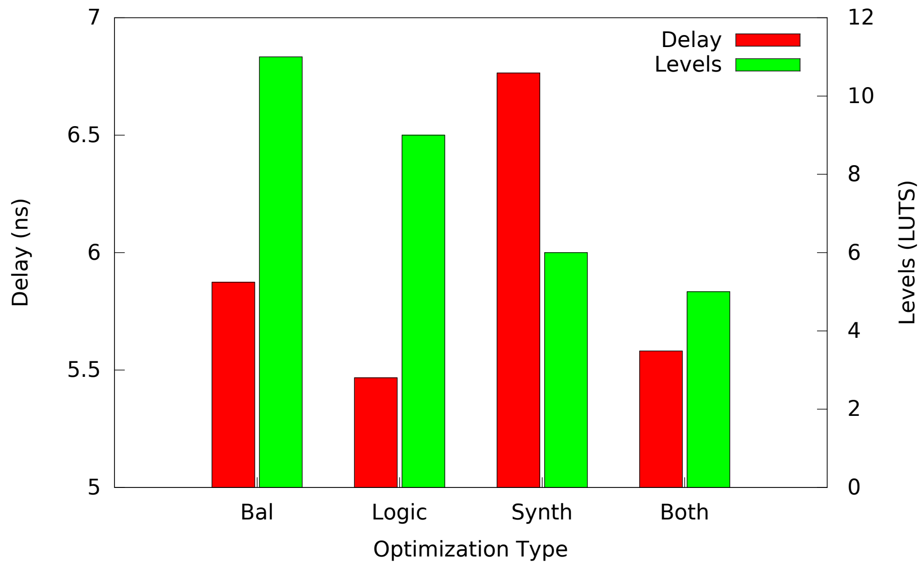


Figure 6.10: Fine Grained Trill Parser Delay Statistics

6.4 Packet Loss Based on Configuration Speed

Approximations of how quickly both a cut-through and a store-and-forward switch would be able to respond to a currently unrecognized packet type using the various forms of reconfiguration examined were obtained by comparing the time required to configure one parsing element to a simplified model of each type of switch's total forwarding delay. The models are both based off of a 1Gb switch architecture which processes packets on a 64 bit wide data bus and then forwards them at full-duplex over a shared-memory switching fabric. Moreover, It is assumed that the switching fabric has a total of eight buffered ports and supports routing up to the network layer (IP). Finally, this general model is also assumed to support both cut-through and store-and-forward forwarding schemes based on explicit user interaction.

To allow the switch model to operate at line rate, its core clock rate is set to 200MHz so that it can generate the 125MHz clock necessary to interface its MAC to the PHY layer [14] with some overhead to possibly run some basic control plane supervisory software routines. The 200MHz core clock also allows for ample speed to sample the incoming data at the rate required to serialize/de-serialize enough bits to fill the 64 bit bus on each ingress or egress port, $\frac{1\text{Gbs}}{64\text{b}} = 15.625\text{MHz}$.

The generally accepted switch latency formula (e.g. [15]) shown in equation 6.4.1 is used to characterize the delay for both of the switch model's modes of operation. The parsing delay parameter represents the delay required by each mode of operation before the forwarding process can be started. When a switch is in cut-through mode, this delay becomes equivalent to the amount of time required to serialize just the portion of the packet relevant to forwarding whereas when it is in store-and-forward mode this delay consists of the period of time required to serialize the whole packet. The look-up delay parameter represents the delay involved in performing the destination port look-up and is the same for both modes of operations. As a shared memory switch fabric architecture is used in the switch model, the fabric delay parameter represents the amount of time required to transfer the packet across the shared memory or, in other words, the time needed to make one write and one read request to this memory back to back [16].

$$D_T = D_P + D_L + D_F \tag{6.4.1}$$

Where,

D_P : Parsing delay
D_L : Look-up delay
D_F : Switch fabric delay

Equation 6.4.2 is used to find the parsing delay of a switch operating in both forwarding modes. The B_L parameter required for the calculation of this delay in cut-through mode is determined by how high up the OSI model the switch supports routing at and the size of the header at each level. In the case of the switch model used in this research, the supported levels are four so the value for B_L is 38B as found from adding up the values in table 6.10. The B_P parameter or the size of the incoming packet can of course vary based on the type of traffic flow; however, in the analysis presented here the size of the packet is always the minimum supported size in a 1Gb network environment, 84B. Using only the smallest supported packet size in this manner constrains the results presented to only those related to the fastest expected rate at which a store-and-forward switch would be expected to start the forwarding process.

$$D_P = \frac{1}{F_T} * \begin{cases} B_L & \text{if in cut-through mode} \\ B_P & \text{if in store-and-forward mode} \end{cases} \tag{6.4.2}$$

Where,

B_L : Portion of packet required for forwarding
F_T : Line rate
B_P : Size of packet

Using the process just discussed the parsing delay for the cut-through forwarding mode of the switch model is found to be $\frac{38*8}{1E9} = 304$ns and $\frac{84*8}{1E9} = 672$ns for the store-and-forward mode.

Level	Size (Bytes)
2	14
3	20
4	4
Total	38

Table 6.10: Protocol Header Sizes by Level

Equation 6.4.3 is used to find the fabric delay of the switch operating in both forwarding modes. The denominator consists of the memory bandwidth required to pass a packet across the switching fabric of a shared memory based switch at line rate and without blocking others. The parameter B_M in the numerator represents the width of the shared memory, the value of which is reliant on the underlying design and count of the memory modules used. In the context of the switch model used, the memory width is set to 64B (the memory can write and access one minimum sized packet at a time), the port count is eight and the line rate is 1Gb so the fabric delay comes out to 125ps in both modes.

$$D_F = \frac{1}{2 * F_T * P} * \begin{cases} 1 & \text{if in cut-through mode} \\ \frac{B_P}{B_M} & \text{if in store-and-forward mode} \end{cases} \tag{6.4.3}$$

Where,

B_M : Shared memory width
P : Total number of ingress and egress ports

The calculation of the minimum total delay in both modes is shown in equation 6.4.4 where the forwarding delay was found as a maximum delay constraint on this stage by dividing the minimum packet size by the line rate. As can be expected the forwarding delay at line rate in store-and-forward mode is larger than the one in cut-through mode even when handling the smallest supported packet sizes. This model could be used to demonstrate the the scaling up of the store-and-forward delay against the static delay of the cut-through architecture with increasing packet sizes, however only packet flows exhibiting the lowest forwarding delay as modelled by the results found are of interest to this research because they demonstrate the worst case scenario of loss of data during reconfiguration.

cut-through mode:

$$D_T = 304\text{E}{-}9 + 672\text{E}{-}9 + 125\text{E}{-}12 = 976.125\text{E}{-}9\,\text{s} \tag{6.4.4}$$

store-and-forward mode:

$$D_T = 672\text{E}{-}9 + 672\text{E}{-}9 + 125\text{E}{-}12 = 1.344\,125\text{E}{-}6\,\text{s} \tag{6.4.5}$$

Finally the total expected packet loss based on forwarding mode of operation and reconfiguration method used was estimated by creating a scenario in which a homogeneous stream of packets of a type currently unsupported by the switch model triggers a reconfiguration event and continues until the switch is able to parse it. Using this setup, the amount of packets un-handled in each case was then determined by dividing the configuration delay values found by the ideal switch latencies from equation 6.4.4 and finally rounding the answer up to the nearest forwarding period.

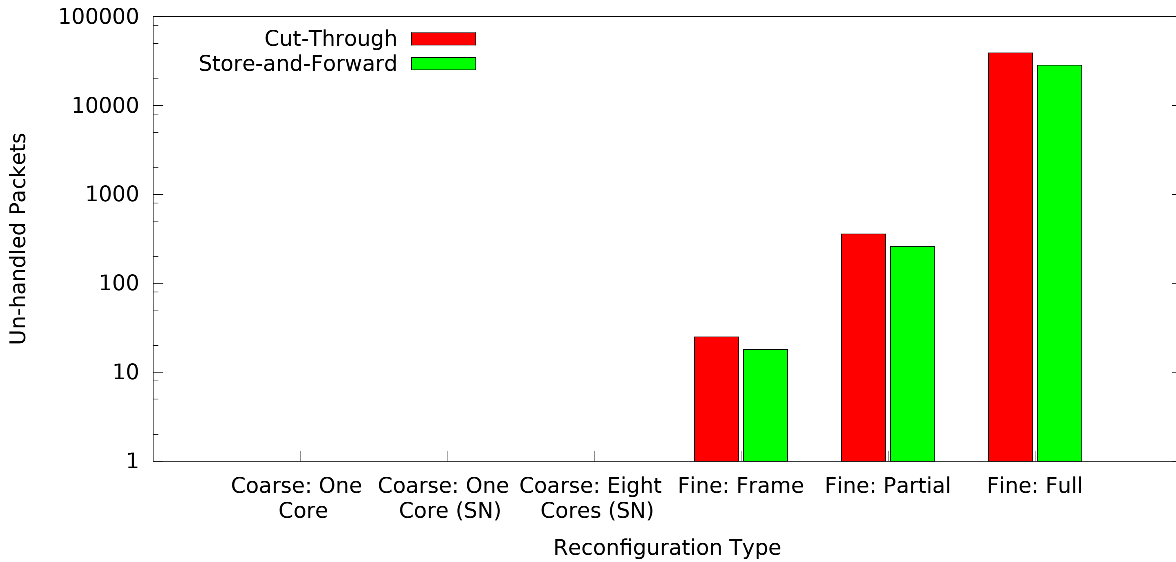


Figure 6.11: Expected Un-handled Packets During Configuration

The results from the un-handled packet calculations, plotted in Figure 6.11, appear to predict that a switch built with a coarse-grained reconfigurable parser chain would be able to dynamically adjust itself to a new type of sustained traffic flow with a much lower penalty in regards to potential data loss than one which is designed with fine-grained reconfigurable parsing logic. In fact at the 1Gb line rate it is assumed that the switch model is operating at, the results indicate that a switch using coarse reconfiguration could adapt after receiving just one minimum sized frame even if it were in cut-through mode and had to configure eight of its parser cores. The best that could be expected from one using fine-grained configuration would be adaption to a new stream after eighteen un-handled frames, but only if it were in store-and-forward mode and its parser logic could be updated with the adjustment of just 40 LUTS.

Chapter 7

Related Work

This section provides a brief survey of additional research performed in the field of dynamic FPGA reconfiguration and how it relates to the work outlined in this thesis. Other than in section 7.4, in all cases where dynamic reconfiguration is discussed it should be taken as referring to modular or partition based dynamic reconfiguration.

7.1 Dynamic Reconfiguration of Network Components

Since modern high end switches already include the infrastructure required to manage software based elements collective referred to as the control plane and complete software based data plane replacement options such as the open source Open vSwitch project [17] are currently available, the groundwork already exists for replacing additional hardware elements with software counterparts to allow for additional flexibility of the device. The major hurdle in taking this approach, however, lies in the ability of a device utilizing this sort of configuration to maintain a packet throughput or line rate comparable to one which retains these elements in hardware. This deficiency has been demonstrated in several studies on the performance of pure virtual networks versus those with aspects of the switch data path implemented using FPGAs [18, 19]. This is essentially why the use of software replacements for the parsing logic in the switch architecture examined was not pursued as part of this research.

The research is motivated in part by the need to characterize what the impact real-time self reconfiguration of a network switch would have on the integrity of the data flowing through it and as such the delivery mechanism for the partial bitstreams to the FPGA was

modelled more on the idea of context based direct memory access of multiple pre-stored values. The context is determined, as already discussed, by the EtherType field of the incoming data packet's header. Goller et. all also propose a reconfigurable network device in a paper in which this field potentially prompts a change in functionality, however, the actual reconfiguration process this are of data potentially triggers in their implementation is significantly different [20]. In the architecture they present, the bitstream needs to be sent in band along with a special EtherType indicating that it is encapsulated within the packet. If a special parser then detects that this field is present it prompts additional logic to strip the segment of bitstream contained out of the packet and to store it in on-board DDR2 memory. Once the complete bitstream is downloaded in this fashion, the actual reconfiguration process begins.

While the reconfiguration process envisioned by Goller and his colleagues does provide a and convenient method by which to disseminate new modes of operation to networking devices, its focus is definitely not on speed. The direct transfer of the bitstream over the network is definitely going to be initially faster than having to obtain and store a set of bitstreams for various parser setups by other means. However, once these multiple configurations are stored then the device will be able to respond to changes in network traffic type much quicker as long as the proper set were obtained before hand.

7.2 Dynamic Reconfiguration Using Custom ICAP Controller

In FPGA designs where partial reconfiguration is not initiated by an external entity, the on-device logic handling the process usually takes on the form of either a FSM machine specifically tailored to respond to events in the rest of the design or software running on a soft processor programmed into the FPGA. The design created for the fine-grained design researched here relies on the former method where the ICAP controller FSM listens for particular fields within an Ethernet based packet stream. The use of this technique can be found in other research projects such as one in which the feasibility of using a custom ICAP controller to trigger the reconfiguration of portions of a DC to DC converter is explored. In this particular case, the goal of adding the reconfigurability functionality was to demonstrate how these particular circuits could be designed to handle dynamic changes in input voltage [21]. Another project in which the use of this method can be found is one in which the controller is integrated into error checking logic that sequential checks all the frames of logic on the device for SEU (Single Event Upsets) [22]. In this second example, if

a frame is found to incorrectly configured then it is reconfigured with a backup bitstream stored on an ECC checked block of ram.

Creating a partial reconfiguration project based around a soft processor involves first generating the cores for the processor and its peripherals, mainly the ICAP controller interface, and then compiling software routines which are responsible for directing the CPU through the memory to ICAP transfer. The rest of the steps from this point on are basically the same as for a custom ICAP controller project (described in the Framework chapter) except that the compiled software has to be integrated into the initial bitstream [23]. Reorda et. all propose a self-repairing design similar in structure to the error checking example already mentioned but which uses this soft processor reconfiguration scheme [24]. The use of the soft processor as the controller for self configuration has also been examined in research pertaining to the application of FPGAs within Avionics systems to allow the bus protocol used between its hardware components to change dynamically [25].

Even though the latter reconfiguration setup would have been easier to implement owing to vendor provided resources just discussed, the former method was chosen and implemented in this research for two reasons. First of all, the hardware only method has been shown to be much faster than the software one [26, 27], which makes it a better candidate for the demanding requirements of switching configurations while ensuring minimal traffic loss in a network switch environment. Secondly, the use of the custom logic of a dedicated controller provides for an easier starting point for future research into the optimization of the configuration process as at its core its just a basic FSM.

7.3 Improvements in the Dynamic Reconfiguration Process

The maximum configuration speeds achievable on FPGA devices are not currently limited by any as of yet unresolved inefficiencies with the transfer of the bitstream to the device but rather by the underlying technology of the configuration port itself. In even the newest devices these ports are designed to operate at a maximum frequency of 100 MHz and with a maximum bus width of 32 bits [28, 29] which caps the configuration performance at 400 KB/ms. While no facilities exist for the end user to increase the width of the data accepted by the port beyond 32 bits, at least in the Xilinx produced chips the user does have the ability to adjust the operational parameters of the PLL (Phase Lock Loops) responsible for these device's configuration clocks [30]. This feature has been taken advantage of in several research projects where they have found that, depending on the family and speed

grade of the Xilinx board used, the configuration port could be overclocked to run at a range of frequencies between 2 to 3 times faster than the original specification [31, 27]. Of these projects, in the one headed up by Claus et. all the correct operation of the port was demonstrated at the highest clock frequency in this range, meaning that the board was being successfully configured at a rate of 1.2 MB/ms.

On the Virtex-5 FPGA used in this research the smallest addressable configuration unit is one LB wide by 20 LB high which translates to a bitstream size of 5904 Bytes as reported by the planAhead tool used in the PR flow. If each parser module were to be segmented into multiple modules of this size then the minimum time the board would take to configure each section using a 3x over-clock of the configuration port would be $5904/1200E6 \approx 4.92s$. As the fastest that this device could possibly reconfigure itself is still within in the micro-second range, the over-clocking strategy alone could not be used in this design to completely address packet loss related to new types of traffic during reconfiguration.

7.4 Other Forms of Reconfiguration

So far the only type of partial reconfiguration discussed has been the one involving reconfiguration within set partitioned areas, the method used in this research. An additional method exists, however, which is referred to as difference or ECO (Engineer Change Order) based partial reconfiguration depending on the manufacturer of the device. The purpose of this design flow is to target very small changes within the logic such as adjusting the equation of one of the LUTs, changing the input level of an individual MUX or modifying a small portion of the data stored within a BRAM, whereas with module based reconfiguration the intent is to make changes to large portions of the logic such as swapping out an adder with a multiplier. The process involves first establishing one implementation of the design as the base and then generating a bitstream consisting only of the areas which have changed for every future rebuild [32].

This reconfiguration scheme has been used in a projects related to both modelling DSMs (Digital Signal Modulator) in hardware as well as increasing the efficiency of the placement stage of the bitstream generation flow. In the former, it is shown that the modulation technique of a DSM implemented on an FPGA can be quickly switched by only making minor changes to a BRAM based LUT [33] . In the latter, an algorithm is proposed which attempts to minimize the difference in layout between multiple iterations of the same design which would in turn then decrease the size of the bitstream generated by the software in the differential reconfiguration flow.

Despite its focus on making only minor changes, the smallest portion of logic which can be addressed by the difference based flow is still the configuration frame. What this means is that, in the context of the hardware used within this research, even if only the equation of one LUT was changed in the overall logic using this method then an area of 20 LB or 40 LUTS would still have to be configured to implement this difference in the hardware.

7.5 Coarse Architectures

As covered in the previous sections, the ability to decrease the reconfiguration time of fine grained FPGA designs is currently limited by the overhead imposed by having to pass the bitstream through the configuration logic at a bit granularity. Coarse grained architectures or systems which support reconfiguration at a higher granularity level have been proposed as alternatives which are not limited by configuration hardware speed, bus width, etc and which can boast faster configuration speeds by the virtue of the greater size of the base configuration unit. Additionally the design process for these devices benefits from the decreased complexity of resources which have to be routed by the synthesis software leading to faster build times. The trade-off for these positive aspects of the coarse grained designs however is increased circuit area used per unit of configuration and a lack of vendor support in regards to synthesis software, necessitating custom solutions [34].

Paul et al. propose a course grained architecture which is able to reconfigure itself at any given execution cycle to act as a sequential step in a software application [35]. The architecture consists of multiple general purpose processors which can only communicate with their direct neighbours via shared registers and which each represent a portion of the function to be executed by the overall system at that particular step of the program counter. While they do also introduce an algorithm which attempts to minimize the amount of configuration time required between execution cycles by locating a route between the processors which is as similar to the previous one as possible, their model still uses the actual configuration memory of the device, meaning that it is still limited by the speed of the device's configuration logic.

A modification of the synthesis process for FPGAs referred to as parameterised configuration [36] has been introduced by Bruneel et al. as a means to mitigate the area impact of using coarse grained architectures. In a parameterised design, the traditional input ports of a component can be made reconfigurable with a meta comment designation in the HDL or left static by not including the comment. Initially, a template netlist is created and a copy is transferred to the configuration memory of the device. During runtime, if a change in configuration is needed then software running on an external processor modifies

the template only at the LUT tables representing the configurable ports using a new set of parameters and then directly transfers the bitstream to device using partial reconfiguration. As with the previous example, however, this design is still limited by the speed of the device's configuration logic.

The coarse architecture presented in this research was devised specifically to demonstrate the trade-offs of a reconfigurable design created with only speed of configuration in mind; as such the configuration memory of the parser elements exists in the logic space of the FPGA and is configured without having to rely on the ICAP port of the underlying hardware. Furthermore, while the interconnection network is also reconfigurable in the coarse design, there is little justification for use of a custom mapping algorithm owing to the limited permutations in how the parser elements can interconnect and the fact that the network also exists in the logic space of the device.

7.6 Network Processors

A network processor can be seen as a general purpose processor (GPP) which has had its scope of functionality constrained to focus on handling a set of tasks related to networking. As with modern GPPs, newer network processors often contain multiple cores which can configured to simultaneously handle multiple segments of a temporally partitioned process. These network processors also, like their general purpose counterparts, contain embedded low latency memory caches as well as interfaces to slower but higher density external memory. Unlike GPPs, however, it is not uncommon to find these processors embedded with a good number of hardware accelerators in the form of either ASIC or FPGA technology. Owing to limitations of the RISC architectures they are based on, the cores in a network processor are unable to interpret some of the more complex instructions required to run an operating system, meaning that software written for these devices has to be done so at a very low level of abstraction.

To decrease the complexity of having to access memory resources within network processors directly in the code, Wu et al. propose an architecture in which this routine is partially implemented in hardware [37] . In a typical design flow for these processors, when particular data structure needs to be accessed in a process, the task of determining such details as which bank within a memory resource and at what offset all have to be handled in the code. In their proposed system, the operating contexts as well as packet data are written to their respective memories at fixed offsets and then accessed during operation with an address shifter. In the actual code, to access these memory location

all that has to be done is to create a static pointer to a fixed memory location and then increment the pointer for each item to be accessed from that memory.

The DynaCORE architecture is proposed by Albrecht et al. as an extension to the functionality of the hardware accelerators built into network processors [38] . The core, which would take on the form of a co-processor external to the network processor, would consist itself of a number of dynamically reconfigurable hardware accelerators, a dispatcher and a reconfiguration manager. The reconfiguration manager would be responsible for determining the optimal configuration of all the hardware accelerators as a set by running the current measured load type and efficiency through an algorithm. The dispatcher would interface with the reconfiguration manager to determine then the most efficient forwarding scheme of the incoming packets to each of the hardware accelerators. The whole system is envisioned as either running under a pipelined or Network on Chip (NoC) interconnect scheme.

Chapter 8

Conclusion

8.1 Discussion

The research presented in this thesis was conducted with the goal of determining the feasibility of designing a network switch which could dynamically and autonomously adjust its packet parsing capabilities based solely on its recognition of new types of traffic. Along these lines, the parsing logic of an existing hardware switch was studied and then re-implemented, in part, to support dynamic reconfiguration within an FPGA fabric using two different approaches, coarse and fine-grained. The efficacy of these two designs in minimizing their impact on typical hardware engineering constraints was compared where applicable along with their impact on the forwarding performance of an abstract switch model created for this purpose. Finally, the performance of the overall configuration framework was discussed in the context of work done by other researchers in similar areas.

While the coarse based configuration scheme designed for this research overtly presents a promising starting point for future commercial applications of flexible parsing logic, it does suffer from many of the same drawbacks found in other coarse grained architectures, mainly resource usage and decreased flexibility. The fine grained scheme also, like other similar implementations researched by others, does not provide an ideal solution owing to limitations imposed by the serial nature of the underlying FPGA fabric. The coarse based configuration scheme, however, was found to be able to support basic reconfiguration at line rate which, added to its vendor agnostic nature, may make it a more suitable choice of the two for future investigation into this area. Nevertheless, it still should be noted that the latter configuration scheme was found to use much less resources and the static reconfiguration method responded much better to optimization attempts.

In summation, neither of the proposed reconfiguration frameworks are presently ready for use in anything beyond theoretical application in an academic setting. With this in mind, the next section provides a number of suggestions regarding what steps could be taken in consolidating the ideas surrounding one or both of these frameworks into something more applicable for practical use.

8.2 Future Work and Suggested Improvements

8.2.1 Coarse Grained

Currently the biggest pitfall of using the coarse grained parser design is the large area requirements and as such future work on this design should be focused on reducing its size footprint. The key aspect of the design which contributes most to this problem is its heavy reliance on internal register memory as both a means to store its configuration memory, to form the shift registers and to store the temporary results of calculations. To address the first issue, the configuration memory could be stored in one or more of the device's BRAM blocks per parser logic unit and the surrounding logic could be modified to pre-read the required configuration bits for each step of the parsing process. To address the second issue, the shift operations could be implemented using the device's DSP48 blocks configured as multipliers. The last of these inefficiencies could be resolved much as the first in that an additional block of BRAMs per device could be used to store the results of intermediate calculations.

Taking the size optimization of the configuration memory one step further, the packing of the bitstream into BRAMs could be also be used as a way to even decrease the number of parser cores needed in the whole parser chain. A single parser core could be made to handle multiple parsing levels by storing multiple configurations in its BRAM memory, adding a scheduler to create time slices to the parser core for both the oldest (highest level) and the youngest (lowest level) portions of the packet and caching the output from each stage with an association to the level context that this fragment of data should be parsed with next. The context associated with each output would have to be stored with the outputs from the previous stage of parsing so that it could be fed into the inputs of the next stage if needed, for example, for the transfer of the EtherType field from the Level 2 parser to the Level 3 parser.

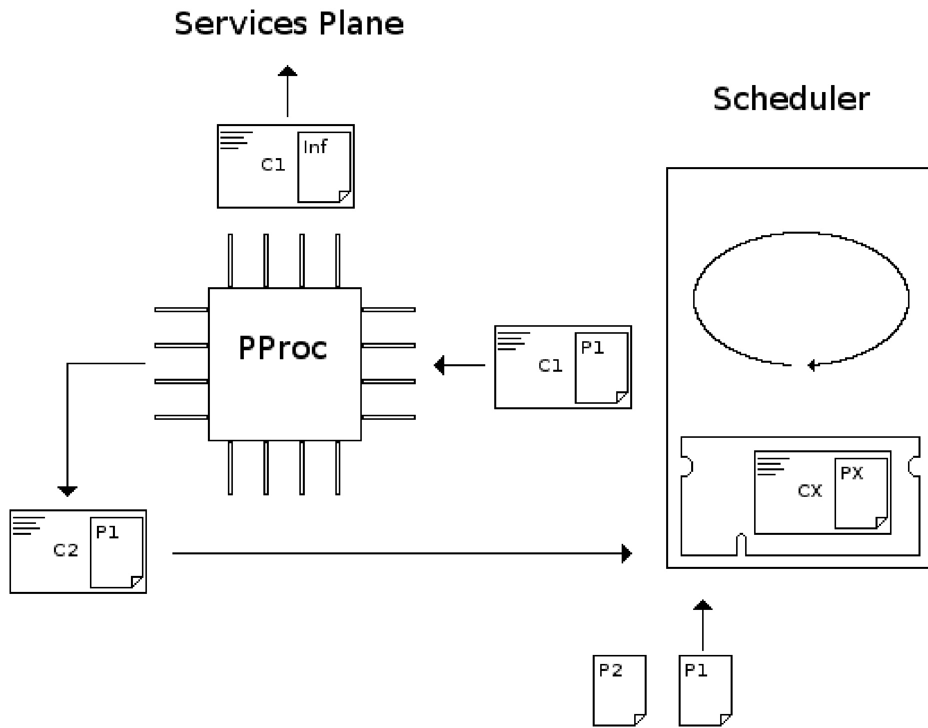


Figure 8.1: Multiple Context Coarse Parser Processor

Both the design were tested with a synthesis mechanism in the vendor provided software which encourages the mapping algorithm to use the DSP48 blocks where possible and these results were compared with the those from the synthesis of the designs without the DSP48 blocks. The routing algorithm, however, is not able to replace the shift registers with an equivalent combination of logic and multipliers, meaning that, if the shifters were to be replaced, the logic would have to be changed manually. The primary complication in making this replacement by hand lies in how the shift amount is currently stored in the configuration memory.

To make a multiplication (or division) operation equivalent to a shift operation, the shift amount would have to be converted to the value that results from raising two to the power of the shift count and then that result would have to be multiplied against the original value. In terms of the current design and hardware implementation in general, the much larger multiplication operand would then either have to be stored in additional memory or calculated with the use of additional DSP48 resources. The storage of the larger operands would involve having to increase the size of the bitstream as well, resulting in both

a slightly longer configuration time and an increase in the complexity of the configuration logic at all levels.

While the combinatorial logic of the parser process does not factor in to the problem of area usage as much as the sequential logic, it too could be optimized if any of the required resources were left on the board after the impact of the latter were addressed. The Comparator Logic Unit and a portion of the start up logic which occurs directly after configuration and involves performing basic addition and subtraction could both be replaced by a set of DSP48 blocks each. In fact, as these operations occur in mutually exclusive states of operation, the same set of DSP48 blocks could be used in both. Furthermore the operational parameters required to do both of these operations could be stored in BRAM as well and then applied as needed with the onset of each operational state [39].

8.2.2 Fine Grained

As discussed in the literature survey, any future improvements in the efficiency of fine grained reconfiguration process is capped by the limitations of the underlying configuration hardware of the FPGAs so it is hard to envision any significant venues of improvement. There is however a relatively undocumented and, with version of the vendor provided software used, poorly supported feature within the fabric that may offer a means to overcome at least in part the bandwidth restrictions of the ICAP port. The CFGLUT5 primitive which can only be instantiated as such and not inferred offers a means to reserve one of the native MLUTS in a similar fashion as used in the optimization investigations of this research but with the added access to a clock and serial input pin [40]. Furthermore, these primitives have an additional output which can be used to chain it others of the same type, essentially allowing a user to directly configure a portion of the FPGA in the most fundamental sense possible while the design is running. The main drawback of using this approach is that all the logic would have to be manually translated into LUT table memory, meaning that in other words, the bitstream for that portion of the design would have to be created by hand. Also, the type of logic which could be implemented in this type of partition would be limited to one which contains at least in part sequential circuitry. Nevertheless, the potential benefit of using this approach is, as mentioned, that a designer could increase the bandwidth of the configuration process by creating multiple chains of CFGLUT5s to represent the logic in question and then program them in parallel at a clock speed only limited by the system speed supported by the architecture (several times greater than the native ICAP clock). With this in mind, it could be investigated whether the logic within the parsers could be translated into such a partition.

APPENDICES

Appendix A

Additional Results

A.1 Worst Case Delays By Architecture

A.1.1 Static

Input	Output	Logic Levels	Delay (ns)
parser_lvl2/u_ip_enflop_res_rdy/q_55	parser_trill/data_r_32	8	5.690
parser_lvl2/u_ip_enflop_res_rdy/q_20	parser_trill/data_r_32	8	5.688
parser_lvl2/u_ip_enflop_res_rdy/q_52	parser_trill/data_r_32	8	5.641

Table A.1: Static TRILL Parser Delays: Balanced

Input	Output	Logic Levels	Delay (ns)
parser_lvl2/u_ip_enflop_res_rdy/q_27	parser_trill/data_r_20	2	6.369
parser_lvl2/u_ip_enflop_res_rdy/q_27	parser_trill/data_r_20	2	6.265
parser_lvl2/u_ip_enflop_res_rdy/q_23	parser_trill/data_r_20	2	6.173

Table A.2: Static TRILL Parser Delays: Synth Optimized

Input	Output	Logic Levels	Delay (ns)
parser_lvl2/u_ip_enflop_res_rdy/q_54	parser_trill/data_r_43	8	5.140
parser_lvl2/u_ip_enflop_res_rdy/q_54	parser_trill/data_r_42	8	5.128
parser_lvl2/u_ip_enflop_res_rdy/q_56	parser_trill/data_r_43	8	5.097

Table A.3: Static TRILL Parser Delays: Logic Optimized

Input	Output	Logic Levels	Delay (ns)
parser_lvl2/u_ip_enflop_res_rdy/q_57	parser_trill/data_r_54	3	5.415
parser_lvl2/u_ip_enflop_res_rdy/q_57	parser_trill/data_r_26	3	5.408
parser_lvl2/u_ip_enflop_res_rdy/q_57	parser_trill/data_r_53	3	5.402

Table A.4: Static TRILL Parser Delays: Both Optimized

Input	Output	Logic Levels	Delay (ns)
XLXI_32/buf_tx_data0/q_9	parser_lvl2/u_ip_enflop_res_rdy/q_9	5	3.821
XLXI_32/buf_tx_data0/q_1	parser_lvl2/u_ip_enflop_res_rdy/q_17	5	3.648
XLXI_32/buf_tx_data0/q_15	parser_lvl2/u_ip_enflop_res_rdy/q_31	5	3.601

Table A.5: Static Lvl2 Parser Delays: Logic Optimized

Input	Output	Logic Levels	Delay (ns)
XLXI_32/buf_tx_data0/q_3	parser_lvl2/u_ip_enflop_res_rdy/q_19	2	4.832
XLXI_32/buf_tx_data0/q_14	parser_lvl2/u_ip_enflop_res_rdy/q_30	2	4.543
XLXI_32/buf_tx_data0/q_12	parser_lvl2/u_ip_enflop_res_rdy/q_28	2	4.445

Table A.6: Static Lvl2 Parser Delays: Both Optimized

A.1.2 Coarse Grained

Input	Output	Logic Levels	Delay (ns)
parse_con/rx2_in_data_31	parse_con/pld2_out_bus_12	6	5.560
parse_con/rx2_in_data_31	parse_con/pld2_out_bus_30	6	5.497
parse_con/rx2_in_data_31	parse_con/pld2_out_bus_22	6	5.326

Table A.7: Worst Delays Through Basic Parser: Balanced

Input	Output	Logic Levels	Delay (ns)
parse_con/rx2_in_data_41	parse_con/pld2_out_bus_39	6	5.519
parse_con/rx2_in_data_9	parse_con/pld2_out_bus_8	6	5.511
parse_con/rx2_in_data_39	parse_con/pld2_out_bus_39	6	5.394

Table A.8: Worst Delays Through Basic Parser: Logic Optimized

Input	Output	Logic Levels	Delay (ns)
parse_con/rx2_in_data_63	parse_con/pld2_out_bus_0	2	3.385
parse_con/rx2_in_data_28	parse_con/pld2_out_bus_28	3	3.197
parse_con/rx2_in_data_38	parse_con/pld2_out_bus_38	3	3.115

Table A.9: Worst Delays Through Basic Parser: Synth Optimized

Input	Output	Logic Levels	Delay (ns)
parse_con/rx2_in_data_6	parse_con/pld2_out_bus_6	2	5.283
parse_con/rx2_in_data_27	parse_con/pld2_out_bus_17	2	5.107
parse_con/rx2_in_data_30	parse_con/pld2_out_bus_17	2	5.016

Table A.10: Worst Delays Through Basic Parser: Both Optimized

Input	Output	Logic Levels	Delay (ns)
parse_con/rx2_in_data_51	parse_con/pld2_out_bus_12	7	5.714
parse_con/rx2_in_data_34	parse_con/pld2_out_bus_3	7	5.708
parse_con/rx2_in_data_51	parse_con/pld2_out_bus_26	7	5.706

Table A.11: Worst Delays Through Advanced Parser: Balanced

Input	Output	Logic Levels	Delay (ns)
parse_con/rx2_in_data_5	parse_con/pld2_out_bus_3	9	5.705
parse_con/rx2_in_data_38	parse_con/pld2_out_bus_24	6	5.639
parse_con/rx2_in_data_36	parse_con/pld2_out_bus_24	6	5.583

Table A.12: Worst Delays Through Advanced Parser: Logic Optimized

Input	Output	Logic Levels	Delay (ns)
parse_con/rx2_in_data_18	parse_con/pld2_out_bus_18	2	2.894
parse_con/rx2_in_data_42	parse_con/pld2_out_bus_42	3	2.685
parse_con/rx2_in_data_24	parse_con/pld2_out_bus_24	3	2.442

Table A.13: Worst Delays Through Advanced Parser: Synth Optimized

Input	Output	Logic Levels	Delay (ns)
parse_con/rx2_in_data_38	parse_con/pld2_out_bus_30	2	5.852
parse_con/rx2_in_data_47	parse_con/pld2_out_bus_6	5	5.667
parse_con/rx2_in_data_47	parse_con/pld2_out_bus_6	5	5.666

Table A.14: Worst Delays Through Advanced Parser: Both Optimized

A.1.3 Fine Grained

Input	Output	Logic Levels	Delay (ns)
l2a_inner/u_ip_enflop_res_rdy/q_50	trill_inner/data_r_46	11	5.874
l2a_inner/u_ip_enflop_res_rdy/q_50	trill_inner/data_r_22	11	5.858
l2a_inner/u_ip_enflop_res_rdy/q_50	trill_inner/data_r_54	11	5.853

Table A.15: Fine Grained TRILL Parser Delays: Balanced

Input	Output	Logic Levels	Delay (ns)
l2a_inner/u_ip_enflop_res_rdy/q_56	trill_inner/data_r_4	6	6.765
l2a_inner/u_ip_enflop_res_rdy/q_31	trill_inner/data_r_4	6	6.630
l2a_inner/u_ip_enflop_res_rdy/q_21	trill_inner/data_r_4	6	6.625

Table A.16: Fine Grained TRILL Parser Delays: Synth Optimized

Input	Output	Logic Levels	Delay (ns)
l2a_inner/u_ip_enflop_res_rdy/q_31	trill_inner/data_r_47	9	5.467
l2a_inner/u_ip_enflop_res_rdy/q_31	trill_inner/data_r_48	9	5.466
l2a_inner/u_ip_enflop_res_rdy/q_16	trill_inner/data_r_47	9	5.452

Table A.17: Fine Grained TRILL Parser Delays: Logic Optimized

Input	Output	Logic Levels	Delay (ns)
l2a_inner/u_ip_enflop_res_rdy/q_61	trill_inner/data_r_2	5	5.581
l2a_inner/u_ip_enflop_res_rdy/q_61	trill_inner/data_r_30	5	5.563
l2a_inner/u_ip_enflop_res_rdy/q_61	trill_inner/data_r_29	5	5.560

Table A.18: Fine Grained TRILL Parser Delays: Both Optimized

Input	Output	Logic Levels	Delay (ns)
XLXI_32/buf_tx_data0/q_18	l2a_inner/u_ip_enflop_res_rdy/q_18	5	4.772
XLXI_32/buf_tx_data0/q_23	l2a_inner/u_ip_enflop_res_rdy/q_23	5	4.335
XLXI_32/buf_tx_data0/q_15	l2a_inner/u_ip_enflop_res_rdy/q_15	5	4.111

Table A.19: Fine Grained Lvl2 Parser Delays: Logic Optimized

Input	Output	Logic Levels	Delay (ns)
XLXI_32/buf_tx_data0/q_19	l2a_inner/u_ip_enflop_res_rdy/q_19	2	4.131
XLXI_32/buf_tx_data0/q_7	l2a_inner/u_ip_enflop_res_rdy/q_23	2	4.052
XLXI_32/buf_tx_data0/q_20	l2a_inner/u_ip_enflop_res_rdy/q_20	2	3.896

Table A.20: Fine Grained Lvl2 Parser Delays: Both Optimized

Appendix B

Additional Framework Details

Property	Value
Synthesize - XST:Optimization Goal	Speed
Synthesize - XST:Optimization Effort	High
Synthesize - XST:Register Balancing	No
Synthesize - XST:Pack I/O Registers into IOBs	Yes
Map:Placer Effort Level	High
Map:Placer Extra Effort	Normal
Map:Combinatorial Logic Optimization	true
Map:Global Optimization	Speed
Map:Pack I/O Registers/Latches into IOBs	For Inputs and Outputs
Place and Route:Place and Route Mode	Route Only
Place and Route:Place and Route Effort Level (Overall)	High
Place and Route:Extra Effort (Highest PAR level only)	Normal

Table B.1: Low Delay Synthesis Profile

Glossary

FPGA

or Field Programmable Gate Array is a technology consisting of an array of programmable lookup tables which can be configured to perform various logical operations.

LUT

or Look Up Table is a hardware based index into an array of bit vectors usually stored in static memory. The bit vectors can be filled in such a fashion as to emulate the truth table of a logical operation. These devices often act as the fundamental units of operation of an FPGA.

RAM

or Random Access Memory can be defined as hardware memory which can be written to or read from in the same amount of time regardless of the internal location or order of the bits of data accessed. Most types of RAM broadly fall into one of two categories, static or dynamic, depending on how their data is stored but both are considered volatile in that they lose their information once their power supply is removed.

SRAM

or Static RAM is a type of RAM in which a bit is stored using multiple transistors. It can be generally accessed faster than Dynamic RAM but, owing to its internal structure, is less dense and more expensive to produce than its counterpart.

DRAM

or Dynamic RAM is a type of RAM in which a bit is stored using a single capacitor and then accessed with a simple transistor switch. While Dynamic RAM is denser and less expensive to produce, it must be periodically refreshed owing to capacitor leakage.

DDR RAM

or Double Data Rate RAM is a variant of DRAM where values are accessed and stored, in part, on both the rising and falling edge of a controlling clock signal. Newer implementations of this type of memory such as DDR2 and DDR3 essentially just increase the amount of data which can be accessed per request.

FIFO

or First In, First Out in hardware terms refers to usually an SRAM wrapped in logic which as a whole is meant to take on the role of a data buffer where the oldest information stored is then the first accessed on a read request.

References

[1] Marilyn Wolf. *FPGA-Based System Design*. Upper Saddle River, NJ: Prentice Hall PTR, 2004. Chap. FPGA Fabrics.

[2] Ross H. Freeman. “Configurable electrical circuit having configurable logic elements and configurable interconnects”. Pat. 4870302 (US). Feb. 1988.

[3] R.T. Ong and E.M. Young. “Programmable address decoder for programmable logic device”. Pat. 5821772 (US). Oct. 1998.

[4] C.R. Erickson et al. “Configurable parallel and bit serial load apparatus”. Pat. 5995988 (US). Nov. 1999.

[5] D.P. Schultz, L.C. Hung, and F.E. Goetting. “Configuration bus interface circuit for FPGAS”. Pat. 6262596 (US). July 2001.

[6] Dimitrios Serpanos and Tilman Wolf. *Architecture of NETWORK SYSTEMS*. Burlington, MA: Elsevier, 2011. Chap. Bridges and layer 2 switches.

[7] H. Jonathan Chao and Bin Liu. *High performance switches and routers*. Hoboken, N.J.: Wiley-Interscience, 2007. Chap. HIGH-SPEED ROUTER CHIP SET.

[8] Yang Y. *Understanding Switch Latency*. Tech. rep. Cisco, Jan. 2012.

[9] *Virtex-5 FPGA Configuration User Guide*. 3.11. UG191. Xilinx. 2012.

[10] *DN9000K10PCIE4GL User Manual*. 1.0. The Dini Group. 2007.

[11] *Synthesis and Simulation Design Guide*. 13.1. UG626. Xilinx. 2011.

[12] *Synopsys/Xilinx High Density Design Methodology Using FPGA Compiler*. 1.0. XAPP107. Xilinx. 1998.

[13] *Timing Closure User Guide*. 13.4. UG612. Xilinx. 2012.

[14] “Part 3 : Carrier Sense Multiple Access With Collision Detect on (CSMA/CD) Access Method and Physical Layer Specifications”. In: *IEEE Std 802.3* (2000), pp. i–1515.

[15] Paul Congdon, Matthew Farrens, and Prasant Mohapatra. "Packet Prediction for Speculative Cut-through Switching". In: *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. New York, NY, USA: ACM, 2008, pp. 99–108.

[16] S. Iyer, R.R. Kompella, and N. McKeowa. "Analysis of a memory architecture for fast packet buffers". In: *High Performance Switching and Routing, 2001 IEEE Workshop on*. 2001, pp. 368–373.

[17] I. Moldovan and P. Varga. "A flexible switch-router with reconfigurable forwarding and Linux-based Control Element". In: *Electronics and Telecommunications (ISETC), 2012 10th International Symposium on*. 2012, pp. 217–220.

[18] D. Unnikrishnan et al. "Reconfigurable Data Planes for Scalable Network Virtualization". In: *Computers, IEEE Transactions on* 62.12 (2013), pp. 2476–2488.

[19] Muhammad Bilal Anwer and Nick Feamster. "Building a Fast, Virtualized Data Plane with Programmable Hardware". In: *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*. Barcelona, Spain: ACM, 2009, pp. 1–8.

[20] U. Pross et al. "Demonstration of an in-band reconfiguration data distribution and network node reconfiguration". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*. 2010, pp. 614–617.

[21] K.K. Sajeesh and V. Agarwal. "Digital controller implementation for non-inverting buck-boost converter using run-time partial reconfiguration of FPGA". In: *Power Electronics (IICPE), 2012 IEEE 5th India International Conference on*. 2012, pp. 1–6.

[22] U. Legat, A. Biasizzo, and F. Novak. "SEU Recovery Mechanism for SRAM-Based FPGAs". In: *Nuclear Science, IEEE Transactions on* 59.5 (2012), pp. 2562–2571.

[23] *Partial Reconfiguration Tutorial, PlanAhead Design Tool*. 14.1. UG743. Xilinx. 2012.

[24] M.S. Reorda, L. Sterpone, and A. Ullah. "An error-detection and self-repairing method for dynamically and partially reconfigurable systems". In: *Test Symposium (ETS), 2013 18th IEEE European*. 2013, pp. 1–7.

[25] V. Viswanathan et al. "Dynamic reconfiguration of modular I/O IP cores for avionic applications". In: *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*. 2012, pp. 1–6.

[26] Ming Liu et al. “Run-time Partial Reconfiguration speed investigation and architectural design space exploration”. In: *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. 2009, pp. 498–502.

[27] K. Vipin and S.A. Fahmy. “A high speed open source controller for FPGA Partial Reconfiguration”. In: *Field-Programmable Technology (FPT), 2012 International Conference on*. 2012, pp. 61–66.

[28] *LogiCORE IP AXI HWICAP v3.0, Product Guide for Vivado Design Suite*. PG134. Xilinx. 2013.

[29] *Stratix V Device Datasheet*. 2.9. SV53001. Altera. 2013.

[30] *Virtex-5 FPGA User Guide*. 5.4. UG190. Xilinx. 2012.

[31] Christopher Claus et al. “Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems”. In: *Reconfigurable Computing: Architectures, Tools and Applications*. Vol. 5992. Springer Berlin Heidelberg, 2010, pp. 55–67.

[32] *Difference-Based Partial Reconfiguration*. 2.0. XAPP290. Xilinx. 2007.

[33] S.U. Bhandari, S. Subbaraman, and S. Pujari. “Digital Signal Modulator on FPGA Using on the Fly Partial Reconfiguration”. In: *Advances in Computing, Control, Telecommunication Technologies, 2009. ACT ’09. International Conference on*. 2009, pp. 711–713.

[34] R. Ferreira et al. “An FPGA-based heterogeneous coarse-grained dynamically reconfigurable architecture”. In: *Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2011 Proceedings of the 14th International Conference on*. 2011, pp. 195–204.

[35] K. Paul, C. Dash, and M.S. Moghaddam. “reMORPH: A Runtime Reconfigurable Architecture”. In: *Digital System Design (DSD), 2012 15th Euromicro Conference on*. 2012, pp. 26–33.

[36] Karel Bruneel, Wim Heirman, and Dirk Stroobandt. “Dynamic Data Folding with Parameterizable FPGA Configurations”. In: *ACM Trans. Des. Autom. Electron. Syst.* 16.4 (2011), 43:1–43:29.

[37] Qiang Wu, D. Chasaki, and T. Wolf. “Implementation of a simplified network processor”. In: *High Performance Switching and Routing (HPSR), 2010 International Conference on*. 2010, pp. 7–13.

[38] C. Albrecht et al. "DynaCORE: A Dynamically Reconfigurable Coprocessor Architecture for Network Processors". In: *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*. 2006, pp. 101–108.

[39] *Virtex-5 FPGA XtremeDSP Design Considerations*. 3.5. UG193. Xilinx. 2012.

[40] *Virtex-5 Libraries Guide for HDL Designs*. 11.3. UG621. Xilinx. 2009.

[41] Cyriel Minkenberg et al. "Current Issues in Packet Switch Design". In: *SIGCOMM Comput. Commun. Rev.* 33.1 (Jan. 2003), pp. 119–124.