Coarse and Fine Grain Programmable Overlay Architectures for FPGAs

by

Alexander Dunlop Brant

B.A.Sc, University of British Columbia, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(Electrical and Computer Engineering)

The University Of British Columbia
(Vancouver)

November 2012

Abstract

Overlay architectures are programmable logic systems that are compiled on top of a traditional FPGA. These architectures give designers flexibility, and have a number of benefits, such as being designed or optimized for specific application domains, making it easier or more efficient to implement solutions, being independent of platform, allowing the ability to do partial reconfiguration regardless of the underlying architecture, and allowing compilation without using vendor tools, in some cases with fully open source tool chains.

This thesis describes the implementation of two FPGA overlay architectures, ZUMA and CARBON. These overlay implementations include optimizations to reduce area and increase speed which may be applicable to many other FPGAs and also ASIC systems. ZUMA is a fine-grain overlay which resembles a modern commercial FPGA, and is compatible with the VTR open source compilation tools. The implementation includes a number of novel features tailored to efficient FPGA implementation, including the utilization of reprogrammable LUTRAMs, a novel two-stage local routing crossbar, and an area efficient configuration controller. CARBON

is a coarse-grain, time-multiplexed architecture, that directly implements the coarse-grain portion of the MALIBU architecture. MALIBU is a hybrid fine-grain and coarse-grain FPGA architecture that can be built using the combination of both CARBON and ZUMA, but this thesis focuses on their individual implementations. Time-multiplexing in CARBON greatly reduces performance, so it is vital to be optimized for delay. To push the speed of CARBON beyond the normal bound predicted by static timing analysis tools, this thesis has applied the Razor dynamic timing error tolerance system inside CARBON. This can dynamically push the clock frequency yet maintain correct operation. This required developing an extension of the Razor system from its original 1D feed-forward pipeline to a 2D bidirectional pipeline. Together the ZUMA and CARBON implementations demonstrate new types of area and delay enhancements being applied to FPGA overlays.

Preface

[1] A. Brant and G. Lemieux. ZUMA: An Open FPGA Overlay Architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2012.

Research and writing was conducted with input and editing from Dr. Lemieux. Initial Verilog implementation of the Malibu architecture and Figures 2.2 and 4.3 were created by Ameer Abdelhadi. The simulation study referenced in chapter 4 was conducted by Douglas Sim, Michael Yui and Tom Tang.

Parts of Chapter 3 have been published in [1].

Table of Contents

List of Tables

List of Figures

Glossary

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
BLE	Basic Logic Element
CG	Coarse Grained
CLB	Configurable Logic Block
CAD	Computer Aided Design
CGRA	Coarse-Grained Reconfigurable
ELUT	Embedded Look-Up Table
EFPGA	Embedded Field Programmable Gate Array
FF	Flip-Flop
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IIB	Input Interconnect Block
LUTRAM	Look-Up Table Random Access Memory
MUX	Multiplexer

STA Static Timing Analyzer

VPR Versatile Place-and-Route

Acknowledgements

I would like to thank my advisor Guy Lemieux for all his support, guidance and patience throughout the program. I am very grateful for having the opportunity to work with him, and explore the cutting edge research he is engaged in.

I would like to thank all the members of the SoC lab who I have had the pleasure of working with. To David Grant for his prior work on the MALIBU system, and help getting started with the system. To Ameer Abdelhadi, for significant work on the MALIBU project. To Aaron Severance for always having insightful ideas to contribute. To Douglas Sim, Michael Yui, Tom Tang for their work on the MALIBU simulation studies.

To my family for their support and encouragement during my education and beyond. To Robin for all her love, help and inspiration.

Financial support from NSERC is gratefully acknowledged.

Chapter 1

Introduction

1.1 Motivation

The logic capacity of modern FPGAs continues to grow with advances in fabrication processes. This growth presents a number of challenges in the design and application of FPGA technologies. Increasing problem sizes lead to higher CAD runtimes which can impede designer productivity and lower the quality of results. Larger capacities also allow many new applications, from replacing highly parallel and word oriented software, to emulating ASIC designs in prototyping stages, which must be considered when designing FPGA architecture and tools. The structure of the word oriented circuits is lost when compiling down to the FPGA resources, increasing FPGA compile times. The flexibility needed to meet all demands of these designs entails compromise.

Due to their complexity and cost, FPGA devices themselves are implemented only by a few large companies. Hence, access to low-level details

about commercial FPGAs is often limited or simply unavailable. There have been calls for a completely open and portable tool flow that allows true portability of designs across all FPGA devices offered by major vendors. This would enable third parties to develop very powerful high-level tools that offer significant gains in productivity and yet can still be efficiently mapped to FPGAs. However, the highly optimized and complex structures in modern FPGAs cannot always be exposed to end users, making creation of such tools difficult.

FPGA overlays are designs loaded into an FPGA that are themselves configurable yet highly optimized for particular problem domains. They can range from simple processors to highly configurable logic arrays. Some advantages of overlays are the ability to use open source tools that enable new features such as lightweight compilation; productivity gains obtained from using pre-optimized structures; and the ability to draw new users to FPGAs who find their current complexity too daunting to use.

In this thesis, we develop two overlay architectures. The first, ZUMA, is a fine-grain architecture based on an island-style FPGA architecture. The second, CARBON, is a coarse-grain architecture implementing the time multiplexed processing elements of the MALIBU architecture [9]. MALIBU itself is a time-multiplexed FPGA which integrates both fine and coarse-grain processing elements into each logic cluster. Together, ZUMA and CARBON implement the fine and coarse-grain logic of MALIBU, respectively.

The development and improvement of the FPGA-like overlay architecture ZUMA is motivated by needs in the research of reconfigurable logic, the desire for flexible reconfiguration in FPGA applications, and the oppor-

tunity for improved designer productivity. ZUMA can help address these goals by providing open access to all of the underlying details of an FPGA architecture without the need for custom silicon fabrication.

ZUMA acts as a compatibility layer, allowing interoperability of designs and bitstreams between different vendors and parts, in an analogous manner to a virtual machine in a computing environment. ZUMA can also be used to embed small amounts of programmable logic into existing FPGA based systems without relying upon the vendor's underlying partial reconfiguration infrastructure, which has traditionally been hard to use, and nonexistent in many cases (e.g. most FPGAs by Altera and Microsemi). The embedded logic will be customizable for specific tasks. More importantly, it can be made portable across different FPGA vendors that have different mechanisms for partial reconfiguration. This also allows for sections of the design, such as glue logic, to be reconfigured without going through an entire CAD iteration with vendor-specific tools. This logic is also sandboxed from the rest of the design, allowing users to safely reconfigure the logic without affecting the rest of the system.

ZUMA is partly intended as a prototyping vehicle for the research and development of future programmable logic architectures. Although incompatible with some architectural features such as bidirectional routing, ZUMA is useful for testing features that go beyond simple density or speed improvements and offer new functionality. In particular, the open nature of the design allows for an open tool flow from HDL-to-bitstream, amenable to current FPGA CAD research. For example, one possible application would be ultra-lightweight place-and-route tools that can run on a soft processor

implemented in the same FPGA as ZUMA, which would allow the FPGA to place and route its own "virtual machine". These tools can be based upon VTR, an open source framework that has been modified to target ZUMA. Alternatively other approaches based on JHDL [2], Lava [21], Torc [22], OpenRec [10], or RapidSmith [14] can also be created.

ZUMA was initially implemented as a generic architecture in Verilog. This was very large in area, so details of the ZUMA overlay were tailored for implementation on modern FPGAs. One key to this is utilizing low-level elements known as LUTRAMs to implement both routing and logic structures in ZUMA. The final area usage of the ZUMA architecture is less than one-third of the generic design, and less than one-half of previous research attempts [17].

The development of the coarse-grain overlay CARBON is motivated by the functional advantages of previous coarse-grain architectures, as well as the opportunity to improve upon them. It is an implementation of the coarse-grain resources found in the time-multiplexed architecture MALIBU. MALIBU was designed to deal with problems of growing FPGA logic capacities. The benefits of MALIBU include much faster compile times, the ability to trade off density and speed by varying the time-multiplexed schedule length, and better performance on word-oriented circuits. However, the use of time-multiplexing can place the coarse-grain elements of MALIBU at a disadvantage in overall clock frequency compared to a normal FPGA. Hence, methods for increasing the clock frequency of MALIBU are highly desired.

Although MALIBU is a hybrid system of both fine-grain and coarse-grain

elements, the MALIBU tools and architecture can both work perfectly using only the coarse-grain elements. Hence, CARBON is an implementation of the coarse-grain only mode, and we can still take advantage of the compile times and density of the architecture, as well as serving as a platform to study further improvements in the MALIBU architecture.

In this thesis, have extended the CARBON architecture by integrating an in-circuit timing error detection system based on Razor [8]. Razor is a system used in pipelined processors to increase performance and reduce power. With Razor, the circuit is operated at a higher frequency than what is deemed safe by static timing analysis tools. This increases the probability of timing errors. Timing errors are detected via shadow registers which capture data a short time after the main clock. If the captured data differs from that stored in the main register, a timing error has occurred. To recover from the error, the circuit typically stalls one cycle and reloads the data from the shadow register.

Although very few general purpose FPGA circuits can support the Razor mechanism directly, we are able to integrate the Razor error correction into the CARBON fabric itself. To do this, this thesis first extends the Razor system to be compatible with a 2D array and implement it in our CARBON FPGA overlay. By implementing this system on an FPGA, we are able to verify the correctness of the architecture and timing error correction system. Using the Razor design, we improve the effective compute speed by 3-20% depending on the benchmark, or an average of 13%. If a hard system deadline is not required, and a consistent execution schedule is not needed, such as when used as a compute accelerator, we can improve the throughput by

5-21% overall, or an average of 14%.

This thesis concretely demonstrates significant area and real delay enhancements to fine-grain and coarse-grain FPGA overlays, respectively. The first overlay developed was a fine-grain architecture based on a traditional FPGA, which was found to suffer from large area overheads. To address this we make contributions towards implementing logic and routing networks in FPGAs using programmable LUTRAMs, an area efficient configuration controller and a novel two-stage local interconnect. The second overlay, a coarse-grain array of processing elements, is inherently slow due to time multiplexing. To improve performance we extend a circuit level technique for pipelined circuits called Razor to detect and correct timing errors cause by overclocking.

1.2 Statement of Contributions

Individual contributions in this thesis are summarized as follows:

- Area efficient implementation of fine-grain routing networks and LUTs on an FPGA via the utilization of LUTRAMs as both multiplexers and logic
- Design of an area efficient 2-stage local routing network and an area efficient configuration controller for implementation on an FPGA
- An extension of Razor circuit level error tolerance from pipelined processors to 2D processing arrays
- Design of on overclockable coarse-grain FPGA overlay with in-circuit detection and correction of timing errors

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 outlines previous work in FPGA overlay systems and related work. Chapter 3 describes the ZUMA fine-grain overlay architecture and improvements incorporated in its implementation. Chapter 4 describes the implementation of CARBON coarse-grain architecture as an overlay, the extension of Razor error detection and correction to a 2D array, and the integration of Razor error tolerance into CARBON. Chapter 5 includes performance and results for both overlays. Finally, Chapter 6 presents conclusions.

Chapter 2

Background

This chapter first presents an overview of FPGAs and FPGA architectures, used as the basis for ZUMA. Previous examples of FPGA overlays are then detailed. The last sections give background information on the MALIBU hybrid FPGA architecture, and the Razor error tolerance system.

2.1 FPGA Overview

Field Programmable Gate Arrays (FPGAs) are configurable integrated circuits that can be programmed to implement nearly any digital circuit. They are attractive to designers due to their fast time-to-market and lower development cost compared to fabricating an application specific integrated circuit (ASIC), as well as the ability to change logic functionality in the field if a bug is discovered or new functionality is desired. Their disadvantages include higher per part cost, lower performance, and higher power consumption.

FPGA are increasingly being used for many functions, including imple-

menting general computation accelerators, and emulating ASIC functionality before they are fabricated.

2.1.1 FPGA Architecture

The general island-style FPGA architecture is shown in Figure 2.1. It is used by many commercial vendors, and is the basis for much research in CAD and architecture [3].

From a high level, the architecture looks like an array of clusters connecting to a global routing network running on each side and between each cluster. The chip is surrounded by IOs which connect to external signals. Signals travel through the global routing network and are processed in the cluster or other heterogeneous blocks. Processed signals will either be routed back to the IO as an output or routed to another destination for further processing.

Each cluster is composed of multiple (N) k-input LUTs. Each LUT is capable of implementing any k-input, 1-output combinational function. A LUT is paired with a flip-flop for sequential designs. The inputs to the cluster, as well as outputs of the LUTs are fed to the LUT inputs through a local interconnect. The global routing tracks are connected to the inputs and outputs of the clusters using sparse crossbars, the connectivities of which are defined by the parameters C_{in} and C_{out}. The number of inputs to the cluster is given by the parameter I.

The global routing architecture of the FPGA is defined by a number of parameters. The number of tracks in each channel is the width (W). The number of clusters spanned by each wire is the length (L). The wires can

be either driven by a single driver, in which case they are unidirectional, or driven by multiple drivers, in which case they may be bidirectional.

Modern FPGAs also feature heterogeneous blocks, such as memories and DSPs. These blocks can be connected much like normal clusters, but may span several blocks to achieve sufficient input and output connectivity.

Parameters such as cluster size and routing width can vary, and different configurations may be better for specific applications, leading to many architectures and chips released by one vendor.

2.1.2 FPGA Implementation in ASICs

Previous research has been conducted on creating FPGAs using ASIC tools. In [11], programmable logic architectures suitable for implementation in ASICs are considered, which include directional architectures used to implement small functions. Implementation issues arising during creation of programmable logic are presented in [25]. In [1], the ASIC CAD flow for implementing programmable logic cores implemented from RTL or HDL descriptions is considered. Issues related to the back-annotation of timing paths for the FPGA CAD flow are presented in [24].

2.2 FPGA Overlays

An FPGA overlay maps an FPGA into another type of programmable architecture. Overlay architectures were first used to describe the design of time-multiplexed and packet-switched routing networks implemented on FPGAs [12]. Previous designs of overlay architectures have mapped the FPGA to a number of different categories. A soft processor can be considered a form

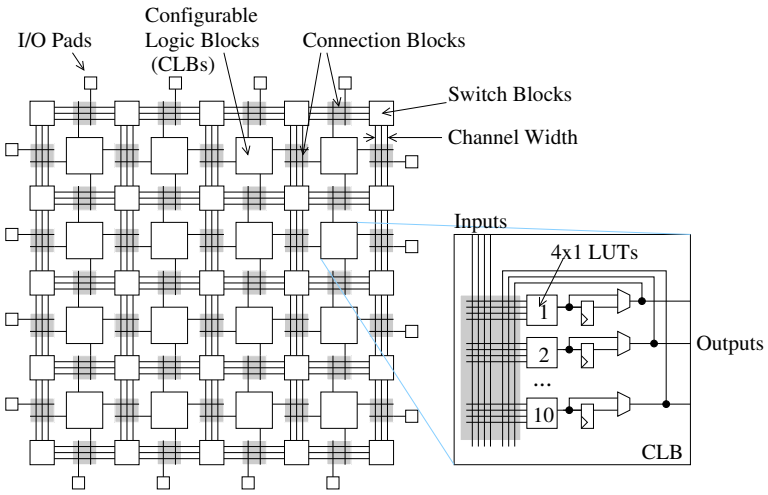

Figure 2.1: Island-style FPGA architecture

of overlay architecture, as it allows a user design in the form of a program to be run on the FPGA. There are many soft processor designs for FPGAs that run the gamut from RISC processors (e.g. Nios II or MicroBlaze), to multiprocessor systems [18], to vector processors [6][26]. There has also been work done on mapping coarse grain reconfigurable arrays (CGRAs) to FPGA systems. CGRAs are similar to FPGAs, but with a larger resource such as an ALU as its basic element, which usually is scheduled between various computations at different time slices. The QUKU CGRA overlay [20] is an example of a system designed to run on top of an FPGA, which runs user applications utilizing many coarse grain logic elements in parallel. There has also been previous work on using FPGAs to implement custom FPGA architectures. The virtual FPGA (vFPGA) [17] is an FPGA overlay architecture that is meant to work with just-in-time (JIT) compilation of FPGA designs for acceleration of sequences extracted from the dataflow graphs of

compiled programs. The vFPGA architecture uses a cluster comprised of four 3-LUTs, and a routing width of 32 with length 1 wires per channel. The design has a 100x area overhead (on 4-LUT Xilinx Spartan-IIe FPGA), meaning each embedded LUT (eLUT) in the vFPGA architecture requires 100 real LUTs from the host Spartan IIe FPGA. Another FPGA-like architecture, called an Intermediate Fabric [23], resembles an island-style FPGA architecture, but replaces the logic cluster with coarse-grain operators such as an adder, and routes data on 8 to 32 bit buses.

2.3 MALIBU Overview

FPGA capacities now can exceed one million LUTs, making them capable of implementing the highly parallel, word-oriented computation demanded by software. As a result, software-to-hardware tools are increasingly used to convert C or other domain-specific languages into HDLs such as Verilog, which can then be compiled by standard FPGA tools. The FPGA tools optimize everything down to the bit level, destroying the word-oriented structure in the original source and needlessly increasing tool run-time. To address this, MALIBU [9], a coarse-grained FPGA that is more amenable to directly implementing parallel software systems, has been developed. In its full form, MALIBU (Figure 2.2) adds a time-multiplexed ALU to every cluster of LUTs, resulting in a system that mixes parallel simulation-like CPU execution with direct LUT implementation. Note that only the coarse-grain ALU (CG), is time-multiplexed, while the fine-grain LUTs are not. On coarse-grained circuits, the tool run-time is up to 300× faster, and a fully custom implementation of MALIBU can perform up to 2× faster compared

to a comparable Altera system [9].

The MALIBU CG includes an ALU capable of performing operations on two 32-bit operands. In addition, the ALU accepts a third input for each operation, called the 'width', which determines how the result of the operation is truncated. This preserves the functionality required for Verilog operations on signals that are less than 32 bits in width.

Four memories (N,S,E,W), store results from neighbouring CGs. The CG also includes a local 'R' memory which is used to store intermediate results, data to be used in later cycles, and constants up to 32 bits. Routing of word-width data is performed by the CG's crossbar. The crossbar takes in values from the four neighbouring CGs, the R memory, and the ALU result, and routes them to a destination CG.

Also shown in the figure is the local instruction memory. The instruction memory stores the schedule of operations to be performed by the CG over all cycles of the schedule. Each instruction is 81 bits in length, and specifies the operation to be performed by the ALU, its source operands and width, as well as routing information for the crossbar for passing operands between the CGs. The CG performs one instruction per system clock cycle. The user clock is the system clock divided by the number of cycles in the schedule.

The fine-grain portion of MALIBU is similar to a traditional FPGA without flip-flops, and with additional inputs and outputs in each cluster to connect to the coarse-grain logic.

The MALIBU architecture has a number of advantages when compared to a standard FPGA. Circuits that contain a high amount of word-oriented computation can run more quickly on MALIBU compared to a standard

FPGA. As the word-oriented operations are extracted from the circuit and implemented with separate hardware that is placed and scheduled using new CAD algorithms, overall compile time can run as much as 300 times faster than on a commercial FPGA [9]. The MALIBU architecture can also achieve a higher density than traditional FPGAs. By sacrificing performance (roughly 2.5× the density for half the performance), it can increase the density by increasing the amount of time-multiplexing of the CGs. Similar density trade-offs are possible with traditional FPGAs by varying the physical architecture and transistor sizes [13]. In comparison, the advantage of the MALIBU architecture is that the performance versus density trade off is possible without changing the underlying architecture or device design.

2.4 Razor Overview

Razor is a system for circuit-level detection and correction of timing errors, aimed at low power operation of pipelined processors. A shadow register, shown in Figure 2.3, is fed by a delayed clock and paired with each register of a pipeline stage. For very low power operation, the supply voltage is reduced below the minimum needed to meet timing requirements. Values are captured by the main register, and passed to the next pipeline stage. Slightly later, the values are captured by the shadow register. It must be ensured that the shadow register will hold the correct value, so the minimum and maximum delay through the circuit must be constrained to meet setup and hold requirements to this shadow register.

When a timing error does occur, it is detected by a mismatch between the main register and the shadow register. One way to correct the error is

to switch the input of the next pipeline stage to the output of the shadow register. The pipeline is then stalled for one cycle, allowing the next pipeline stage time to re-execute with the correct data. Forward progress in the pipeline is ensured, with a worse case performance loss of 50% if an error occurs every other cycle.

Razor was introduced in 2003 [8], and in 2011 an ARM microprocessor was developed with Razor pipelining [5]. To our knowledge, our project is the first to integrate Razor registers in an FPGA architecture, and to produce the first FPGA that can offer timing error correction.

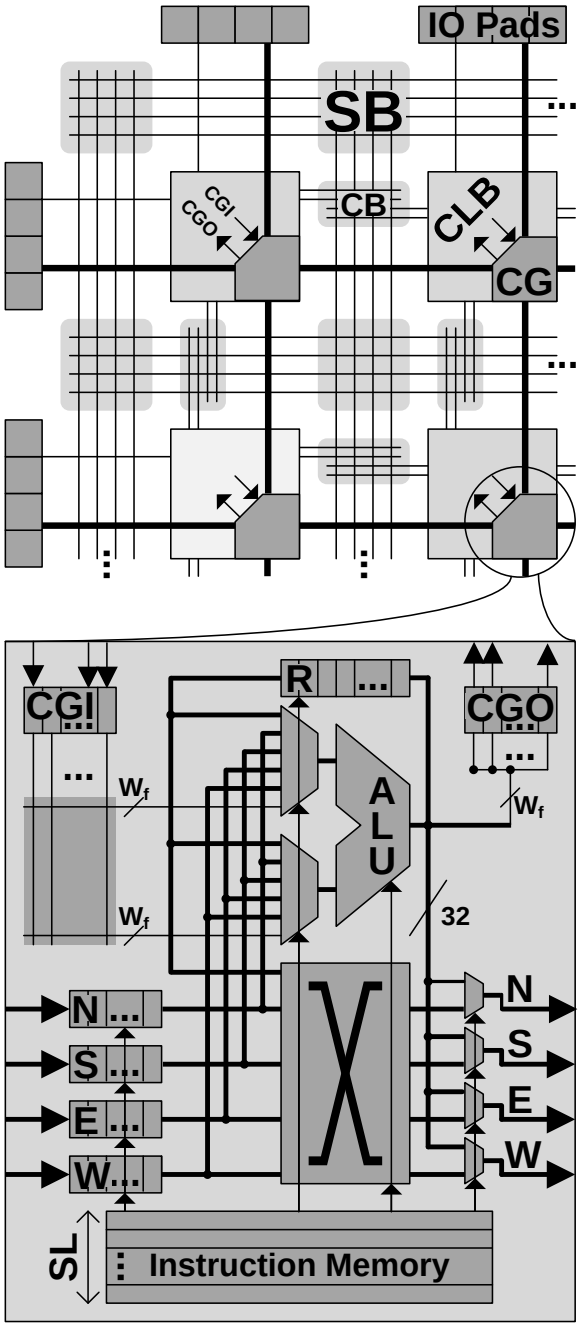

Figure 2.2: The MALIBU architecture CLB - The coarse-grained (CG) part is added to a traditional FPGA CLB

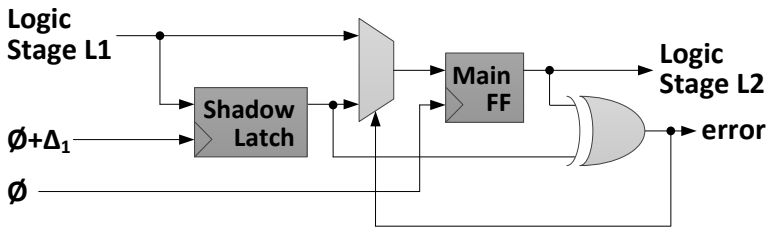

Figure 2.3: Razor shadow register error detection and correction

Chapter 3

ZUMA

3.1 Introduction

This chapter describes the fine-grain ZUMA architecture. Initially a generic architecture was created in pure Verilog, which was used as a baseline for the development as detailed in Section 3.2. Improvements to the ZUMA architecture tailored for area efficient implementation on modern FPGAs are described in Section 3.3. The novel contributions include the utilization of FPGA LUTRAMs as the basis of fine-grain routing networks, a unique two-stage local routing crossbar, and an area-efficient configuration controller. A CAD flow is in place for this architecture utilizing the VTR (VPR 6) project [19], and is detailed in Section 3.4.

Throughout this chapter, the term embedded Look-up table (eLUT) is used to refer to the LUTs in the ZUMA architecture. In contrast the term 'host LUT(s)' is used to refer to the physical resource of a commercial FPGA such as a Virtex 5 device. Efficiency of implementation is measured as host

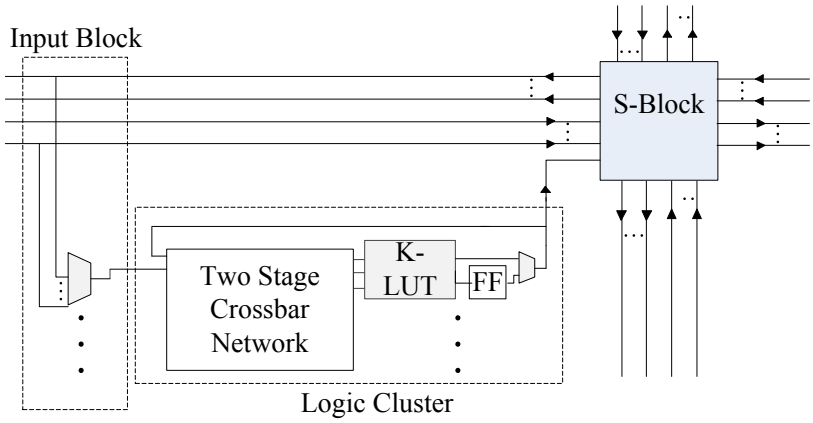

Figure 3.1: ZUMA tile layout and logic cluster design (note: inputs are distributed on all sides in real design)

LUTs per eLUT, with a lower value being better.

3.2 Generic Baseline Architecture

In designing the ZUMA FPGA architecture, we first created a simple LUT-based FPGA architecture supported by VTR. The initial generic architecture design was created as a parameterized Verilog description, similar to the standard architectures used in classical VPR experiments [3]. This section details the design of the logic cluster, global routing and input/output blocks. Initial resource usage was found to be high, and is documented at the end of the section.

3.2.1 Cluster Design

The generic logic cluster is comprised of a first stage depopulated input block for cluster inputs, followed by a fully populated internal crossbar, which is connected to the N k-LUTs. A total of I inputs are fed into the cluster

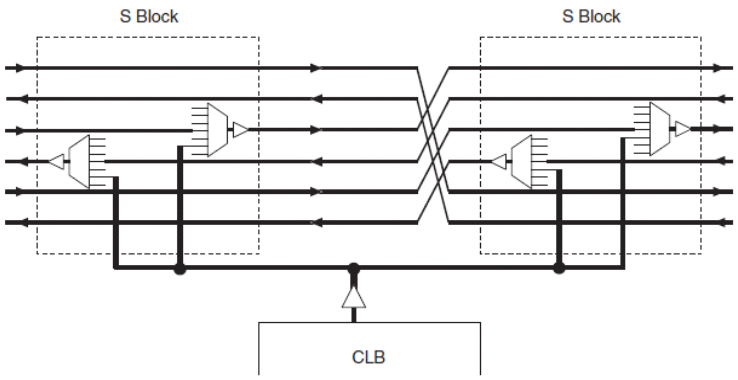

Figure 3.2: Global interconnect with width of 6 and length of 3

by the input block, while all I inputs and N feedback signals are available to any basic logic element (BLE) input pin. The BLEs are single k-LUTs, followed by a single flip-flop which can be bypassed using a MUX. This generic version contains configuration bits stored in shift registers.

The input interconnect block, or IIB, connects the global routing wires and feedback signals to the inputs of the LUTs. The C block forms part of the IIB, while the rest of it is formed by the internal connections from the I+N signals to the k*N BLE inputs. Normally, these internal connections comprise a significant percentage of the area of a modern FPGA [16]. Many FPGA CAD tools, such as VPR, also expect fully populated internal connections. Depopulated clusters can reduce the area significantly, but complicate the placement and routing of designs, and can lead to much higher CAD runtimes [15]. To retain tool compatibility, we chose to allow full routability internal to the cluster.

3.2.2 Global Interconnection Network

The routing architecture of the ZUMA eFPGA is designed with a number of constraints introduced by the FPGA fabric on which it is built. In the ZUMA architecture, a unidirectional interconnection network is constructed using the LUTRAMs as drivers. Bidirectional wires can be implemented from Verilog descriptions on FPGAs, but with significant overhead incurred as they are emulated with unidirectional wires. As such wires can have a vast number of drivers, bidirectional wires are not practical to include in ZUMA.

As each wire can have only one driver, one of the major impacts of unidirectional routing is the combination of the Switch Block and Output Connection Block. Each direction (north, south, east, and west) is driven by a single MUX located in the S-Block, which takes inputs from both other routing wires and neighbouring logic blocks, as in Figure 3.2.

FPGA I/Os are included at the periphery of the array. They follow roughly the same design as the inputs and outputs of the cluster, integrating with the switch block to drive wires, and connecting to a subset of wires to drive outputs.

3.2.3 Initial Resource Usage

The generic architecture was created entirely from standard, portable, fully synthesizable Verilog. All configuration bits are stored in a shift register, which is inferred to use FFs of the host FPGA. The results of synthesizing a single layout tile of the generic architecture on a Virtex 5 part for an architecture with cluster size of N=8, eFPGA LUT size of k=6, wire length

of L=4, and channel width of W=112, are given in Table 3.1.

The biggest use of resources are the embedded BLEs, or eBLEs, which contain the embedded LUTs, or eLUTs. Each requires almost 64 host LUTs and over 64 flip-flops per eLUT. The fully connected internal crossbar is the next-largest use of resources, at about 36 host LUTs per eLUT. Next, the switch and input blocks use many host FFs to hold their configuration data.

3.3 ZUMA Architecture

The implementation of the ZUMA FPGA takes the baseline architecture outlined in the previous section, and utilizes the unique resources available on a modern FPGA to minimize the area overhead. First, programmable LUTRAMS are adopted, both for the user logic, and the routing multiplexers. The LUTRAMS are then used to implement an area efficient two-stage local interconnect in each cluster. An area efficient configuration controller is then designed. Profiling and modeling is done to find the parameters to achieve the highest logic density. Finally, compatibility with the vendor CAD tools for Xilinx and Altera FPGAs is discussed.

Area	Host LUTs	Host FFs
Switch Block	121	156
Input Block	56	288
Internal Crossbar	288	248
eBLEs	528	520
Total	993	1212
per eLUT	124.125	151.5

Table 3.1: Implementation of generic FPGA architecture on Xilinx Virtex 5 (for cluster size N=8)

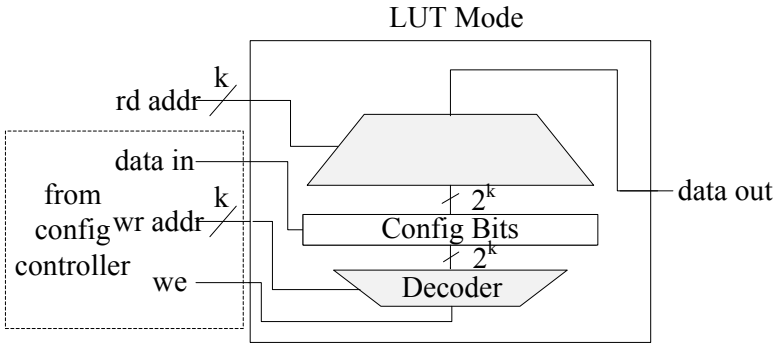

Figure 3.3: LUTRAM used as LUT

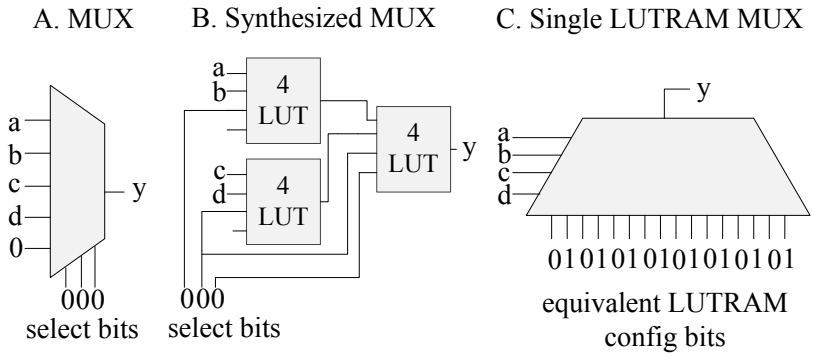

Figure 3.4: A: 4 to 1 MUX (with extra bit tied to ground to turn off track), B: 4-to-1 MUX synthesized from 4-LUTS, C: 4-to-1 MUX created one 4 input LUTRAM

3.3.1 LUTRAM Usage

The ZUMA architecture takes advantage of the reprogrammability of LUTs in the host architectures to create space-efficient configurable logic and routing. In new generations of Altera and Xilinx FPGAs, logic LUTs can be configured as distributed RAM blocks, called LUTRAMs. Both vendors allow fully combinational read paths for these RAMs, permitting them to be used as normal k-input LUTs (Figure 3.3). These are useful for directly implementing the programmable eLUTs of the ZUMA architecture, but they can also be used to improve the implementation efficiency of the routing network. By limiting the LUTRAM configurations to a simple pass-through of one input, a k-input LUTRAM becomes equivalent to a k:1 routing multiplexer. The advantage is the MUX select lines do not need to come from external signals (external configuration flip-flops, but are instead internalized by the LUTs configuration bits. Hence while a 3-LUT would normally only be able to implement a 2:1 MUX if external select lines are needed, it can actually implement a 3:1 MUX. Likewise a 6-LUT can implement a 4:1 MUX with normal approaches, or a 6:1 MUX using the new LUTRAM approach. These MUXs will be the backbone of the global and local interconnection networks of the ZUMA FPGA. These LUTRAM MUXs consume fewer resources than MUXs implemented with generic Verilog constructs, as they require fewer host LUTs and configuration flip-flops due to the lack of configuration bits. As well, to save power by preventing unneeded switching when a routing MUX is inactive, each MUX in the generic version needs to be able to be configured to output ground, which can also increase re-

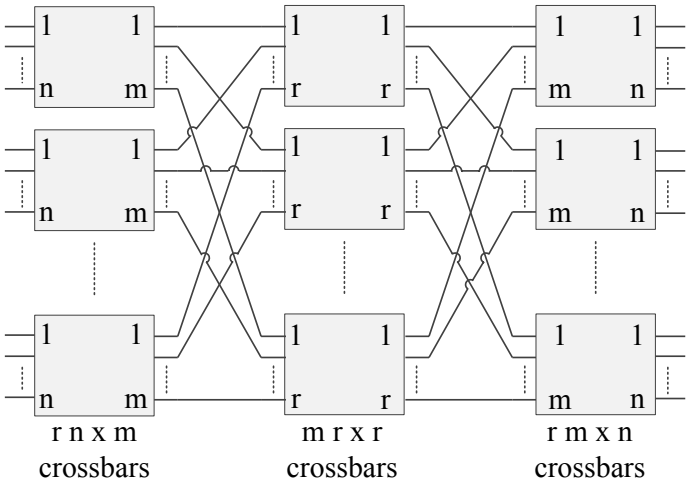

Figure 3.5: Clos network

source usage, while the LUTRAMs can simply be configured by setting all configuration bits to zero. The difference in resource usage for a 4:1 MUX is illustrated in Figure 3.4.

3.3.2 Two-Stage Local Interconnect

The design of the internal connection block of the FPGA cluster is driven by a need for area efficiency, flexibility, and CAD compatibility. To address this, an IIB based on a Clos network was designed. This modified shuffle network is paired with a depopulated first level connection block of MUXs, in order to allow sufficient routing flexibility and compatibility with modern CAD tools.

A Clos network (Figure 3.5), is composed of three stages of connected crossbars. The number of inputs and outputs are identical. It is defined by three parameters: n, m and r, which define the number and dimensions of the

crossbar stages. As long as m is greater than or equal to n, the network can route any permutation of all inputs [7]. Given that we have 6:1 MUXs as our basic switching element for a 6-LUT host FPGA, a first approach is to set the parameters as r=n=m=6. This allows us to build a 36 x 36 Clos network out of 108 LUTs. If we take the outputs of the network to be the inputs of the 6-LUTs, we can see that the last stage can be eliminated, as the order of the input to each LUT is unimportant (see Figure 3.6), and the inputs to each LUT will always be different. We then have a two-stage network with 72 MUXs, feeding into a possible 6 ZUMA eLUTs, giving a total cost of 12 LUTs per eLUT for the IIB, with no reduction in routability. In contrast, a full crossbar implemented in a generic fashion requires 42 LUTs per eLUT.

If the inputs are connected directly to the global routing tracks, only 36 - N tracks will have access to the LUT, since N of the inputs will be reserved for LUT feedback connections. Instead, additional MUXs are added before the first stage to give adequate flexibility, giving an overall design that can be routed by VPR without extensive modification.

When implemented, we found that building a single 6-to-N memory from LUTRAMs is more efficient than N 6-to-1 memories on the Altera platform, as each LUTRAM has overhead for write ports. All LUTRAMs used in a crossbar have the same input signals, and each output, when configured properly will only depend on the value of one of the inputs. As a result, each crossbar can be constructed out of a single multi-bit wide memory to increase density without changing the functionality.

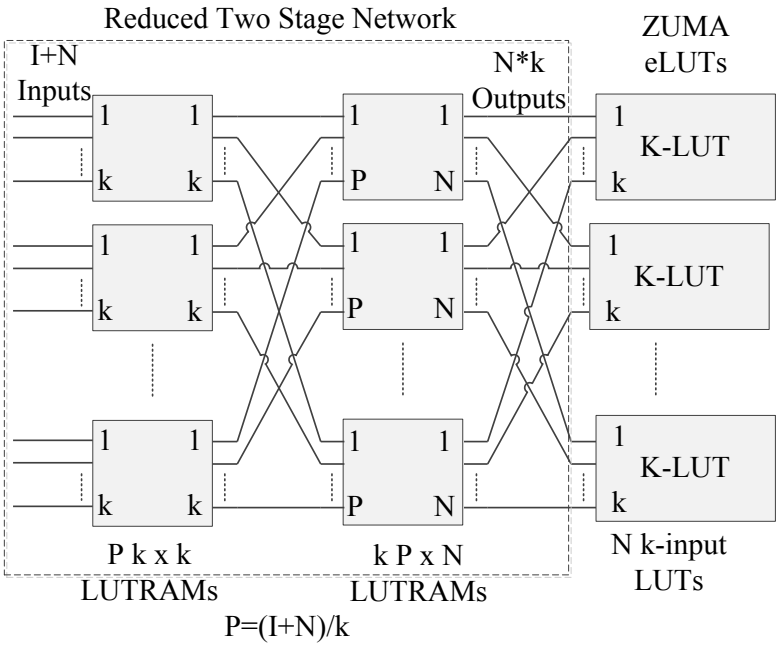

Figure 3.6: Modified two-stage IIB (inputs from global routing input multiplexers)

3.3.3 Configuration Controller

The design of the eFPGA also requires a configuration controller to rewrite the LUTRAMs and flip-flops that control the functionality of ZUMA's logic and routing. A configuration controller was designed to program all of the base elements of the eFPGA. The implementation is parameterized, allowing for tradeoff between resource usage and configuration speed. For a ZUMA overlay comprised of k-input LUTRAMs, each LUTRAM has k address signals, one write-enable, and one write-data signal that must be driven by the configuration controller. The k-bit write address can be shared by all LU-

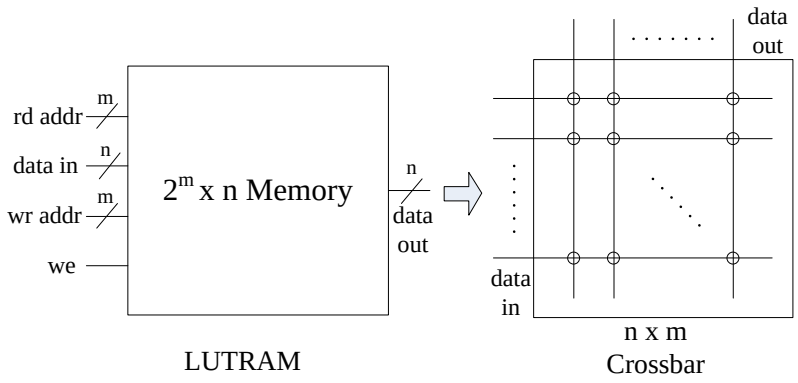

Figure 3.7: n bit wide, 2^m deep LUTRAM used as n x m crossbar

TRAMs in the design. However, to initialize each LUTRAM independently, a set of LUTRAMs can either share a write-enable, or share a write-data signal, but not both. Each unique write-data or write-enable will require at least one additional LUT or flip-flop on the FPGA to drive the signal. ZUMA's LUTRAMs are divided into groups by location, made up of one or more ZUMA tile, and given separate write-data signals, while sharing a write-enable. The groups are written to serially, therefore a shift chain is used to propagate the write-enable between each group. This will utilize only one additional flip-flop per write-enable signal. A 6-bit counter is then used to set the LUTRAM write addresses. For a large design, each group will add only one additional flip-flop to the design. By increasing the number of LUTRAMs programmed simultaneously, faster configuration can be achieved at a higher area cost.

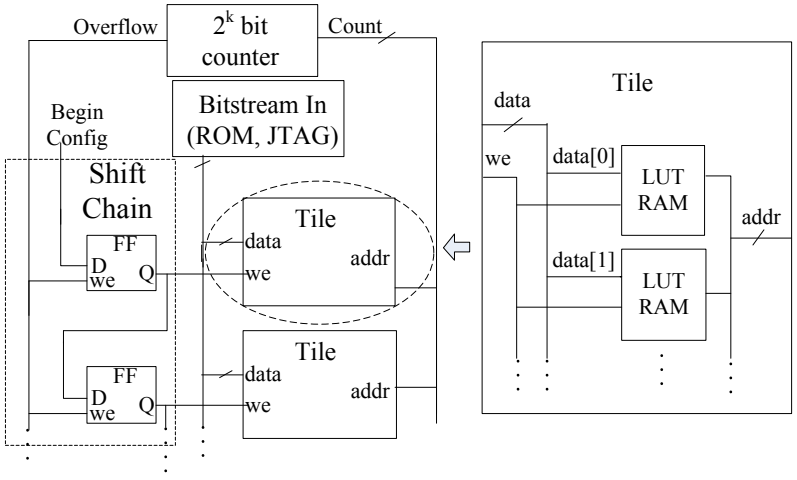

Figure 3.8: Configuration controller

3.3.4 Architectural Model for Area and Routing Efficiency

The correct choice of architectural parameters is very important for the efficiency of the eFPGA, due to the coarseness of the resources of the underlying fabric. Modeling studies were performed to explore optimal settings for the ZUMA FPGA parameters as well as the impact of the host FPGA architecture on the overall mapping efficiency. Given the base resource used in our design, for example a 6 input LUTRAM, parameters that create structures that fit exactly into these LUTs will be most efficient. The internal connection network is most efficient for a 6-input host LUT when the number of inputs is 36 or lower. The formula for an adequate number of inputs to a cluster is given in [3] as (N+1)*k/2. Given that N inputs that must be feedback connections for each LUT, we have the formula (N+1)*k/2+N = 36. Fixing k at 6 and solving for N, we get a cluster size of eight 6-LUTs, which

requires 27 inputs, and a 35 input crossbar. The resource requirements for the ZUMA architecture were modeled by profiling each building blocks resource usage at various configurations, and minimum area overhead was obtained at N=8 eLUTs per cluster. The routing width is varied based on estimates determined from the maximum routing width needed to route all MCNC benchmarks while varying the cluster size. Which was determined by experiments in VPR to find the minimum width allowing the routing of all MCNC benchmarks for this architecture.

3.3.5 Vendor CAD Tool Limitations

In our experimentation, compiling into the host FPGA required some care to ensure the unique requirements of the ZUMA FPGA architecture are mapped efficiently using the vendor CAD tools. The compilation of the ZUMA system to the host FPGA may require additional configuration of the CAD tools used. Naturally occurring features such as combinatorial loops in FPGA structures are not handled well by FPGA CAD tools

In early tests with Altera tools, compilation failed after 2+ days when compiling designs that utilized 50%+ of the host FPGA. Similarly slow compilation time occurred with Xilinx tools. We expect that FPGA CAD tools are tuned to compile generic logic circuits, not the combinational loops and interconnect-intensive properties found in FPGA architectures. As a result, many features of the CAD tools must be bypassed to allow the ZUMA design to be compiled to a host FPGA. Turning off timing analysis and compiler features reduced compile times to roughly one day, but was eventually successful. In Quartus, 'timing driven synthesis' was turned off, and CAD effort

was reduced. In Xilinx ISE, 'timing driven placement' was turned off, the optimization target was set to area, and the 'Optimization effort' was set to 'fast'. Manually partitioning the ZUMA design into smaller sub-arrays can also reduce compile time for larger arrays, but requires an initial synthesis to complete.

3.4 ZUMA Software

This section details the software for mapping of the user design from HDL to the bitstream that is needed by users of the ZUMA architecture. A CAD flow utilizing the VTR framework was created, and is detailed. The current timing models used in the ZUMA architecture are then discussed.

3.4.1 CAD Flow

ZUMA compilation is performed using slightly modified versions of existing tools, and a newly developed tool. The open source VTR CAD Framework is used up to the place and route stage, and the new tool, RTOBITS is used to generate the bitstream. The VTR project contains an entire toolchain that comprises the flow from "Verilog to Routing" [19]. The outputs of the technology mapping, packing, placement and routing stages are all required to generate the bitstream. Each element of the configuration is read in and entered into a data structure corresponding to the architecture of the eFPGA. This description is then serialized into a binary bitstream. The ZUMA tool flow from Verilog to bitstream is summarized in Table 3.2.

A number of files from each stage of the CAD tools are used to generate the bitstream. A shell script was written to generate the correct command

Tool	Purpose	Changes	Input Files	Output Files
ODIN II	Front-end synthesis	None	design.v	odin.blif
ABC	Tech mapping	None	odin.blif	abc.blif
AAPack	Clustering	None	abc.blif	aa.netlist
VPR	Place and route	Modified to output routing resource graph	abc.blif aa.netlist arch.xml	route.r place.p rr.graph
RTOBITS	Building bitstream and ZUMA global routing Verilog	New tool	abc.blif aa.netlist place.p route.r rr.graph	bitstream.hex ZUMAglobal.v

Table 3.2: Summary of tools used in ZUMA CAD flow

line options for each tool, as well as generating the VPR architecture file, given a set of architectural parameters for ZUMA. The configuration of the ZUMA eLUTs is determined using the output of the ABC technology mapping tool, after the elaboration of the input Verilog by the front end synthesis tool ODIN-II. These configurations are written to a blif file, and then packed into clusters by AApack, which outputs a netlist file of logic clusters. The cluster locations are determined from the output of the placement stage, in the VPR place file. The generation of the global routing LUTRAM configurations is determined from the VPR route file. VPR was modified to output the routing resource graph used for the global routing of the FPGA. This graph is read into the RTOBITS tool to generate the connectivity of the global routing switches for bitstream generation, as well as to generate the global routing Verilog design file used to compile the ZUMA architecture.

The RTOBITS tool reads in the technology mapped blif from ABC, the clustered netlist from AAPack, and the place file, route file, and routing

graph from VPR. Initially the routing graph is parsed to generate a model of the FPGA global routing network used to determine the bitstream settings, and if needed, output a Verilog file for compilation of ZUMA to the FPGA. The RTOBITS tool follows each net in the route file, and stores the switch setting for each MUX along the way in the routing data structure. Once all nets have been read, the drivers for all remaining (unused) tracks are configured to logic 0. This way, unused wires don't toggle unnecessarily and waste power. The local interconnect bitstream is then generated. The location of all input pins and LUTs is determined from the placement file, and the Clos network configuration is determined from the connectivity. Routing of the two-stage crossbar was performed by routing each connection individually. If no second stage crossbar is available to route a connection, a wire is randomly selected to be rerouted to allow. Over 1000,000 full capacity crossbar mappings were used to test this algorithm, which was able to route them all successfully. Although this works, a more robust solution is needed. Once all of the configurations have been generated, they must be packed into a serial bitstream. The packing is performed based on the configuration controller data width, and the ordering of the LUTRAMS in the design. The bitstream is output as a HEX file.

The latest version of VPR, 6.0, allows user specification of more complex logic blocks and architectures. However, the ZUMA architecture does not currently utilize any advanced architectural features such as multipliers or block RAMs; these are left for future work.

3.4.2 ZUMA Timing

Since the timing can differ from compilation to compilation, some re-computation of timing may be needed to port a bitstream between different host FPGAs. Though the bitstream is portable, timing will not be guaranteed on any new device. Current timing performance of ZUMA is found by on-board frequency sweeping and verification. The worst-case delay is extracted from the routing fabric and LUTs from the vendor timing tools, and used in the CAD and final timing of the user design. The worst-case delay for the configuration path of each component is also used for the configuration of all elements in the design.

A better approach in the future would be to extract timing information from the host FPGA compilation and use it in the timing models of the ZUMA CAD tools. Back annotation of timing was performed for an FPGA synthesized by ASIC tools in [24]. As the beta of VPR 6.0 is being used for placement and routing, which does not currently allow timing analysis, timing is not considered in this thesis.

3.5 Summary

This chapter presented the ZUMA overlay architecture. ZUMA is an open, cross-compatible embedded FPGA architecture that compiles onto Xilinx and Altera FPGAs. It is designed as an open architecture for open CAD tools. The ZUMA overlay is intended to enable both applications for FPGAs and further research into FPGA CAD and architecture.

Starting from an island-style FPGA architecture, modifications to increase density and resource usage when compiled to a host FPGA were

introduced, while preserving the functionality and CAD tool compatibility with VPR. Through utilization of LUTRAMs as reprogrammable MUXs and LUTs, we are able to reduce the resource overhead of implementing the overlay system by two thirds. Our modified Clos network internal crossbar makes use of these elements, as well as taking advantage of the routing requirements of the cluster architecture, to allow the use of CAD tools such as VPR while reducing resource usage. An area efficient configuration controller was designed to reprogram the architecture. Modeling was performed to find efficient parameters. A CAD flow was developed using the VTR framework.

This new optimized version of ZUMA will be compared to the baseline in Chapter 5. First, however, we present the CARBON overlay in Chapter 4.

Chapter 4

CARBON

4.1 Introduction

This chapter describes the implementation of the coarse-grain CARBON architecture. An initial implementation of CARBON on an FPGA is described in Section 4.2, which included modifying the MALIBU CG to map to FPGA memories, and designing a configuration controller to re-program the overlay without impacting performance. We extend the Razor error tolerance system to work with a 2D array of processing elements in Section 4.3, showing potential performance benefit and improved error recovery. The integration of Razor shadow registers into the MALIBU architecture is described in Section 4.4, and includes modification of the CARBON memories, and error control circuitry to ensure proper data computation and movement.

4.2 CARBON Implementation

The MALIBU CG was implemented in Verilog and compiled to an Altera Stratix III FPGA. The fine grain resources are not included in this implementation, but can be integrated into the design using the ZUMA architecture.

The limitations of the host FPGA necessitated several changes to CARBON from the MALIBU architecture and when implementing memories in the overlay. A configuration controller was developed to re-program the overlay with minimal area overhead and without impacting the speed of the computation path, by utilizing the programmable datapath of the overlay.

4.2.1 FPGA Limitations

The limitations of FPGA block RAMs used as memories in the implementation caused several changes to the functionality of the architecture.

As reads on Altera Block RAMs need to be performed synchronously, the read addresses, which are part of the instruction, must be present at the memory one cycle early. The instruction memory, which contains the addresses must also be read synchronously, so the instruction read must occur two cycles before the instruction is performed. This entails an initial two cycle delay before computation occurs. On subsequent user cycles, the two initial instructions can be loaded at the end of the previous cycle, so no overall impact on the performance occurs.

The synchronous read of the memories also causes problems with user memories located in R memory and accessed with LOAD and STORE operations. As the LOAD operation reads from an address which itself is located

in memory, an extra cycle of latency is needed. Store operations can still be done in one cycle. The datapath controller in CARBON performs a LOAD in two cycles, but further changes to the MALIBU CAD tools are needed to take the extra cycle into consideration. Hence, loads and stores are not fully supported.

Due to the block RAMs inability to emit data written on the previous cycle, changes were needed to the CARBON memories. A 32 bit bypass register was added to each of the CARBON CG memories (N,S,E,W,R). When a write occurs, the written data is stored to the bypass register as well as the memory. In case the address of the current cycle read and previous cycles written register are the same, the data stored in the bypass register is read instead of the memory.

4.2.2 Configuration Controller

The CARBON overlay needs the ability to reconfigure at runtime through a configuration controller. The MALIBU architecture must initialize the instruction memory in each CG, as well as constants in R memory. Additionally, the memory controller that tracks the used and unused addresses in the R memory must be initialized so the constants are not overwritten.

By utilizing the programmable datapath of the CG, the configuration of the constants was achieved with minimal impact to area or speed. As the write port of the R memory is already in use, adding another port to re-program the R memory would entail adding a multiplexer to the write path, thus impacting speed. A similar change to be able to load the memory controller could also impact performance. To work around this, initialization

of the R memory and memory controller is performed by loading a separate bootstrap program into the instruction memory which performs writes to R memory from the immediate value in the instruction, thus initializing both the memory and controller. This also simplifies the design as the R memory controller state does not need to be generated with the bitstream. Constants of over 16 bits can require multiple instructions to be constructed, and large numbers of constants may require multiple blocks of instruction memory to be loaded and executed. This memory initialization program is prepared during compilation, but could be easily constructed on the fly via a state machine or embedded processor to save space from a more compact representation.

A 81 bit wide write port is added to each instruction memory, and each instruction is loaded into memory one instruction per cycle. All other logic is the same as for running a program on the array. Like the ZUMA overlay, each CG can be loaded in parallel or serially, with a higher hardware overhead for the parallel implementation. A block RAM is used to store the instructions for each CG. Each instruction RAM can be addressed and written individually from the configuration controller.

4.3 Razor Error Tolerance

To increase the performance of the CARBON architecture, we will introduce Razor error correction to the system to allow frequency boosting. The Razor system, as outline in Chapter 2, works with feed-forward pipelined circuits. We extend the Razor methodology to work with 2D arrays of processing elements. This extension requires the development of a more complex control

scheme, but has the benefit of allowing more errors to be tolerated given the same impact on performance.

In the original Razor system, when an error occurs, a stall occurs in the pipeline, and is passed on to each succeeding stage. In a 2D array, the stall must be passed on to all neighbours. This entails a more complex control mechanism, which we detail in later sections. It also allows more errors to be tolerated when certain conditions are met. In particular, when multiple errors occur in close proximity, their stall signals propagate until they collide and merge. If two error regions collide, the stall does not need to be propagated back, so the two cycles can be tolerated by a single stall cycle. As a result, more errors can be tolerated than the number of spare cycles added to the time-multiplexing schedule.

4.3.1 Simulation Study

A simulation study was performed by another group of students [Douglas Sim, Michael Yui, Tom Tang, private communication, 2012.] to analyze the benefit of this extra error tolerance. This work is presented here as demonstration of the design's potential. Figure 4.1 plots the probability of failure as a function of the number of injected errors. This baseline case is for a 10×10 array with a ten instruction schedule as the number of spare clock cycles is varied.

Of course, the CARBON-Razor system can always tolerate any e errors, where e is the number of spare cycles added to the instruction schedule. However, it can sometimes tolerate more errors if they are merged as a result of occurring within close proximity to each other in both time and

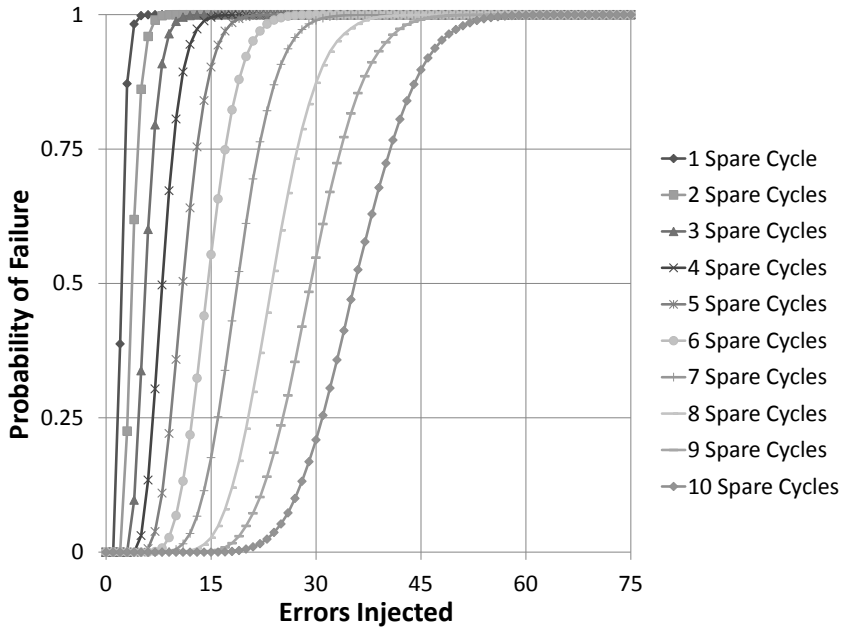

Figure 4.1: Sensitivity of failure to spare cycles, (Array Size 10x10, instruction schedule 10 slots)

space. By close proximity, we mean that the stall signal wavefronts collide and the stall cycles needed to tolerate the errors merge. Although not shown, shorter schedules can tolerate more errors because the errors are more likely to be located closely in time.

Viewed another way, in some applications it may be possible to extend the schedule length dynamically only if needed. Figure 4.2 uses the same parameters as the previous graph to show the average number of spare cycles required to tolerate up to ten errors.

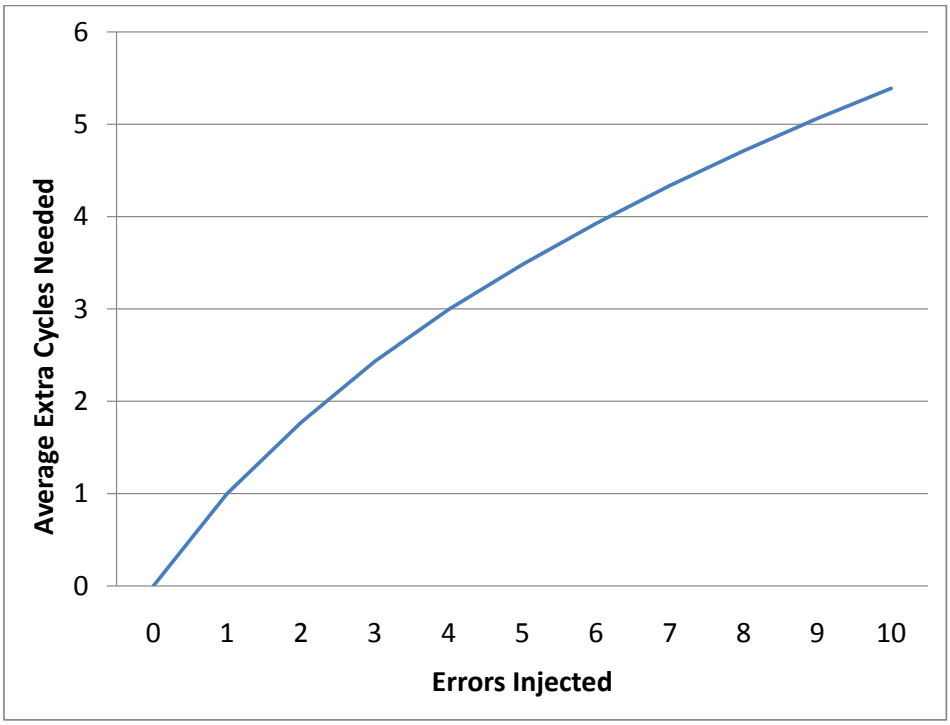

Figure 4.2: Number of errors versus average number of spare cycles

4.4 CARBON-Razor Design

This section details the design of the CARBON-Razor architecture. First, the modification of the CARBON memories is detailed. We demonstrate how the Altera block memory is utilized in a special mode to allow read-back of the word just written. Behaviour of the stall propagation is then discussed, followed by details of the error propagation logic used. Finally, the timing properties of the error tolerance system are discussed.

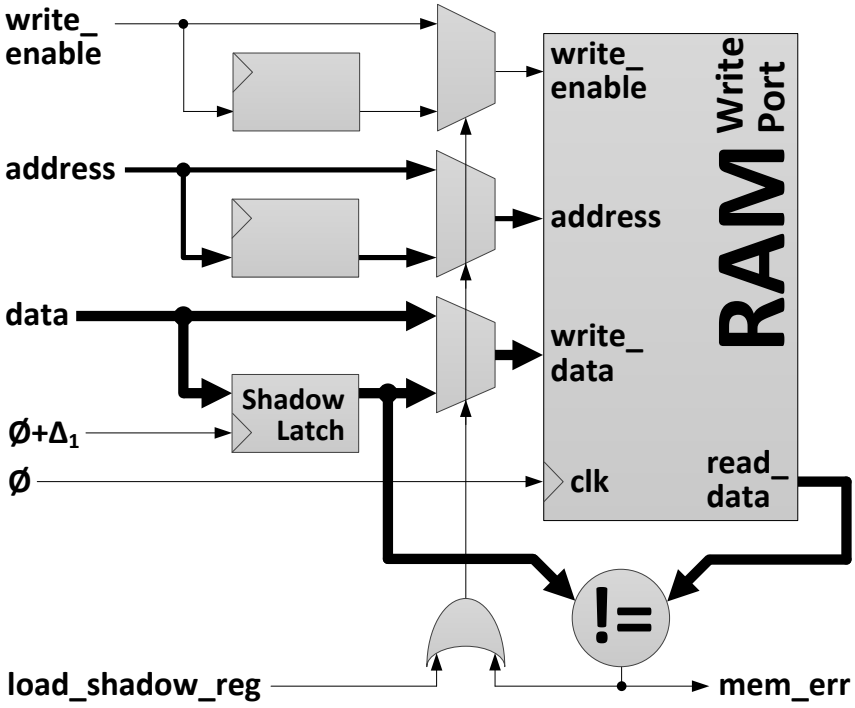

Figure 4.3: Shadow register applied to CARBON memory

4.4.1 Shadow Register For Error Detection

A Razor-style circuit-level error detection and correction system is introduced to the CARBON system to allow for further speedup with error detection and recovery. As the CARBON system uses memories for storing state rather than latches or flip-flops, the Razor design must be modified to shadow and correct values stored in memories rather than registers. The FPGA memories used in CARBON were configured to use an additional read port that emits the written data from the previous cycle. This is compared to a shadow register that captures any word written to the memory a fraction of a cycle late. Another register stores the previous address. If a

mismatch occurs between the Shadow register and read port, an error signal is asserted. This error signal triggers a number of behaviours. First, the data stored in the shadow register is written to the memory at the end of the cycle. The error prevents the current instruction from being executed on the CG, and a stall signal is sent to the four neighbouring CGs on the next cycle. The next cycle, the error will be corrected, and the next instruction is executed, one cycle late. Propagation of the stall signal is discussed in a following section.

4.4.2 Altera RAM Output

In most cases when using Stratix III RAMs, reading and writing from the same address in one cycle is not possible. However, we exploit a property of true dual port RAMs in the Stratix III architecture: they can output the previously written data on the read-output data lines that correspond to the write port that was written in the previous cycle. By utilizing this readout capability, we can ensure the data is stored correctly with low hardware overhead. To enable this immediate read back, we assert the read enable signal along with the write enable when a write occurs. The result in memory is then latched at the read output of the same port.

As CARBON requires three read ports to each of its memories, this is done by replicating the values into three parallel block RAMs. Each of these RAM outputs, as well as the data stored in the bypass register(as described in Section 4.2.1), is compared to the shadow register and 'OR'ed to produce an error signal.

4.4.3 2D Razor Stall Propagation

The CARBON-Razor system extends the stall-propagation of the Razor pipeline to a 2D array of processors. The stall propagation logic must ensure correctness of each memory block being read, as well as deal with multiple errors occurring independently or in groups in the array. A global stall signal would allow for correct functionality and error recovery, but would result in lower performance. The performance reduction is a result of the time needed to propagate the error signal to all CGs, and the inability to recover from multiple independent errors occurring at different timesteps while only extending the schedule by one cycle.

As results from a single ALU operation can be written to memories that communicate with neighbouring CGs, any error recovery that takes place in one CG must be communicated to neighbouring CGs so they cam resynchronize. This is done by propagating a stall signal to the immediate neighbours. This stall must propagate away from the error only, and not come back towards the originator.

Two CGs may stall at nearly the same time and propagate their stall signals. Errors propagate one block at a time to the four immediate neighbours, creating an expanding "diamond" shaped wavefront, as shown in Figure 4.4. In the CARBON system, if two independent errors occur closely in time and with enough distance, it may be sufficient to stall for just one cycle in each CG. This occurs if the second error is detected before the first error propagates all the way to source of the second error. In this case, only one cycle is needed to stall on any one block, and the schedule is only

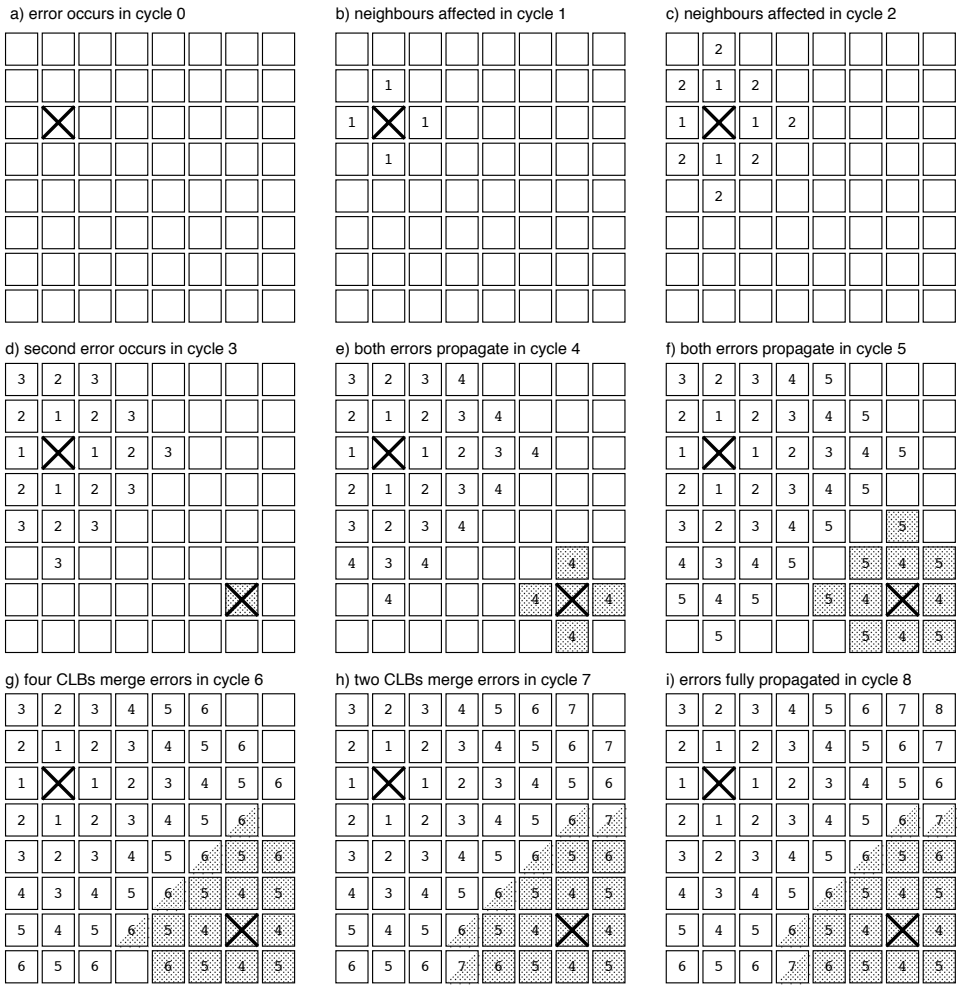

Figure 4.4: Error propagation in the 2D CARBON array. Numbers represent the relative clock period when the CG stalls, centering on the CG where the error occurred

extended by one cycle.

Figure 4.5 conceptually demonstrates the error propagation wavefront when two errors occur in close proximity, followed by a third that is not close enough. Each error occurs at one location and spreads to surrounding

CGs one cycle at a time. When two errors occur at or around the same time, then the two error wavefronts will collide and merge. They will become a unified region of CLBs executing one cycle behind the schedule. If an error occurs at a CG *after* the first error has reached it, it creates a new region that executes two cycles behind schedule. That new region will expand until it reaches every CG in the array. The stall propagation logic is explained in the next section.

Care must be taken to ensure that all memories in a CG store the correct operands for the next scheduled operation before executing. When an error is propagating, data entering into the wavefront must be delayed, also using the shadow register, to prevent data from being overwritten before a CG has the chance to use it.

Consider the case when tile A detects an error at cycle X and reloads its memories. All CGs are initially performing instruction Y of the schedule. When tile B receives a stall signal at cycle X+1, its unstalled neighbour C will have written data from instruction Y at cycle X, but its stalling neighbour A was unable to. During cycle X+1, B will not execute anything, while the data from A for instruction Y is written to the memory, and it will also prevent its neighbour C from writing to B's memory. Instead the data that C would have written at cycle X+1 is captured in the shadow register, to be written at cycle X+2.

At X+2, B has the correct data from both A and C from instruction Y, A is executing the instruction Y+1 and the data C write from instruction Y+1 is being written from the shadow register. B can then execute instruction Y+1 normally and writes the correct data to C. As C does not have the data

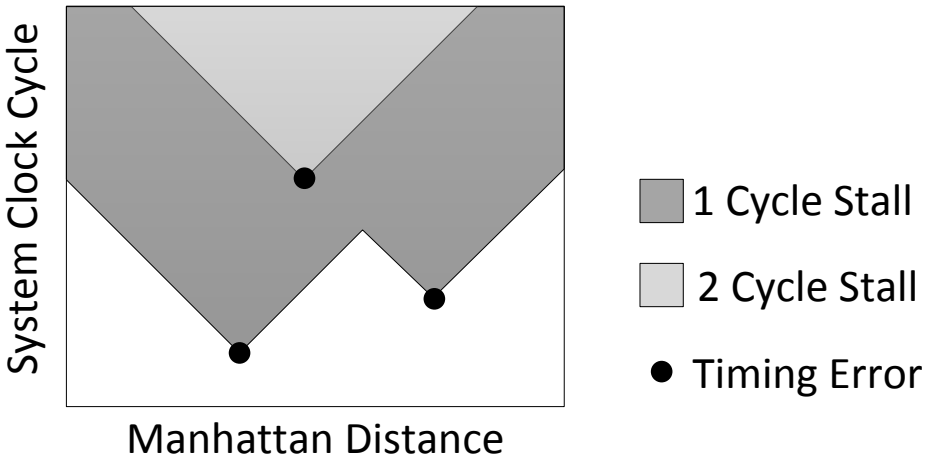

Figure 4.5: Conceptual graph demonstrating how 2 separate errors join to form one stall region, and a later error creates a second stall

from B yet it must wait, and thus will not write more data to B, and the data in the shadow register can be loaded in the memory without blocking a write. The error propagation will continue on until it reaches the edges of the array each CG having stalled one cycle, or until the end of the schedule is reached By the end of the schedule, errors have propagated as far as they need to go. Hence, when the schedule is restarted, all error history is also reset.

4.4.4 Stall Propagation Logic

Stall propagation logic is added to each CG, and controls the recovery actions when an error occurs. The stall propagation logic for one direction is shown in Figure 4.6. When a memory that belongs to a CG encounters an error, the CG initiates a stall that will spread through the array. The stall will

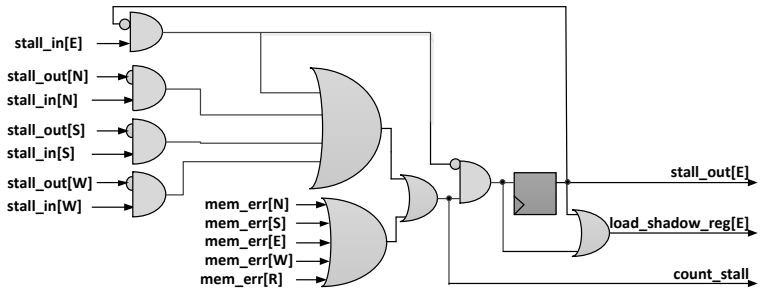

Figure 4.6: Control logic used to control the stall signals (note: only east direction stall_out and load_shadow_reg shown, logic structure is the same for all sides)

affect the operation of a CG for two cycles: one while it is stalling, and one while its neighbours stall. We first detail the signals used in the stall logic. This is followed by an explanation of how the stalling occurs on CG that detects an error, and on CGs that stall due to an error from another tile. Finally, the merging of separately occurring errors is discussed.

The circuit monitors 'mem_error' signals which are asserted when an error occurs at a local memory, and 'stall_in' signals from neighbouring CGs. The circuit outputs 'stall_out' signals triggering neighbours to stall, 'load_shadow_reg' triggering memories to load their shadow register values, and a 'count_stall' signal that holds the instruction counter and prevents writes being made to neighbours memories from the CG.

A stall begins with an error being detected. If on cycle X a value is written incorrectly to memory, part of the way through cycle X+1 the correct value is latched into the shadow register and compared to the memory contents. If the values differ, an error has occurred, and the 'mem_error' signal is asserted. All 5 'mem_error' signals(from the N,S,E,W and R mem-

ories) for a CG are ORed (along with the stall_in signals) to trigger the 'count_stall' signal for the CG.

In the cycle in which the error is detected (x+1), the logic triggers a number of events in the CG. The 'count_stall' signal prevents instruction counter from being incremented, and the current instruction from being updated, and deasserts any write enables to any memory from that CG. The 'load_shadow_reg' signal is sent to all memories, which loads the last cycles data into the memory, correcting any errors, and preventing any writes from the neighbours. These writes must be prevented as they may overwrite data needed in the next instruction executed. The data from the neighbours is not lost though, as it will still be stored in the memory's shadow register.

In the cycle after the stall is detected (x+2), the CG executes the instruction it tried to execute previously. The execution happens normally, but the 'load_shadow_reg' is still asserted to the NSEW memories. This writes any data that neighbouring CGs tried to write previously. No write will be performed on these memories from the neighbours this cycle, as they will be stalling.

CGs that do not encounter an error also need to stall to synchronize with the ones who encounter the error. On the cycle after an error is detected, the CG with the error asserts a 'stall_out' signal to its immediate neighbours on the next cycle. When a CG receives a valid stall signal, it behaves similarity to the CG when a memory error is detected. The instruction counter is not incremented, and no values are written to memory from that CG. Any writes to the CGs memories from neighbours are also delayed, being stored in the shadow register and performed the following cycle, except from neighbours

that requested the stall. These memories are not loaded from the shadow register, as the neighbour who triggered a stall may need to write a result from the instruction it is catching up on. The 'load_shadow_reg', is also registered and used to trigger the 'stall_out', as the signal is not being sent to the originating CG.

When multiple errors occur in the array independently and must be merged, the logic ensures operation occurs correctly. Depending on the distance and timing of the errors, one of two situations will occur: the stalls reach the same tile at once, or two neighbouring tiles stall simultaneously. If the stalls reach the same tiles at once, the tile will stall, but will only propagate the error back to tiles that did not send it a stall, so the regions will merge and be synchronized. If the neighbours stall simultaneously, on the cycle following the stall both will receive the 'stall_in' signal from each-other, but it can be ignored as they are already synchronized.

4.4.5 Performance Tradeoffs

The Razor error detection system performance is determined by both the increase in the clock frequency and the increase in schedule length due to the Razor error correction. The schedule length must be increased sufficiently to allow for the circuit to complete operations, which will depend on the error rate. The error rate will depend on the timing variations in the circuit, the clock frequency, the circuit being executed, and data-dependent results.

The relationship between the minimum safe clock period T_{min} and the Razor clock period T_{rzr} is shown in Figure 4.7. The clock shift (Δ_{rzr}) allows the same result that arrives by the period T_{min} to be latched by the shadow

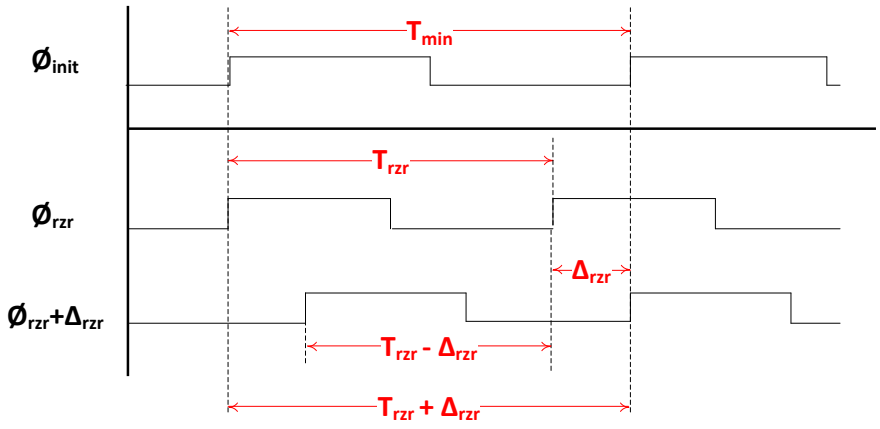

Figure 4.7: Timing comparison for baseline design without error correction vs. Razor enabled design

register. Increases in clock frequency must be accompanied by an increase in delay to the shadow register (Δ_{rzr}). This delay must not exceed the short path through the circuit, from any register fed by the main clock to the shadow registers. This delay is ensured by adding constraints to the Quartus fitter. In practice, as this minimum delay is increased, the nominal F_{max} of the circuit decreases, so the value must be carefully selected to achieve the optimal performance. If the nominal F_{max} of the circuit is given by F_{init} and the minimum path delay to the shadow register is t_{min}, the theoretical safe limit of the system F_{safe} is reached when the sum of the new period and t_{min} equals the period of F_{init}. The safe maximum frequency can be calculated by:

$$F_{safe} = \frac{F_{init}}{1 - F_{init} \times t_{min}}$$

In practice the circuit frequency might be increased beyond this point, but testing is needed to ensure correct operation.

In addition to the minimum path delay to the shadow registers, we must also consider the maximum delay from the shadow register to the rest of the circuit. Once the data is latched it is compared to the data in memory, and if different, the error signal is asserted. The error signal travels to the Razor control unit, and prevents the write enable signal to the CGs memories, as well as preventing the instruction from incrementing. The maximum delay from the output of the shadow register must be less than the clock period minus the shadow register delay in order for the error to be corrected properly (T_{rzr} - Δ_{rzr}) .

Both these path delays can be constrained using the Quartus Timing Analyzer to maximize performance. Many configurations of timing constraints were swept to find the best performing values in practice.

4.5 Summary

This section outlined the design of the coarse-grain FPGA overlay CARBON, and the application of in-circuit error correction to the overlay. Initial implementation of the CARBON overlay was based on the MALIBU CG, and included modifications needed for the Stratix memories, and a high performance configuration controller. The Razor error tolerance system was extended from 1D pipelines to 2D arrays. The Razor registers were applied to the CARBON overlay, which included utilization of Stratix memory properties, and design of an error propagation network. We have developed a unique time-multiplexed FPGA architecture that can recover from timing

errors, as well as the novel application of Razor error detection to a design on a commercial FPGA.

Chapter 5

Results

5.1 Introduction

This chapter describes the area and speed improvements achieved by the two overlay architectures described in previous chapters. First, the results and area reduction achieved by the ZUMA architecture is presented. The final resource usage of the ZUMA architecture is less than one-third of the generic design, and less than one-half of previous research attempts [17]. As few as 40 host LUTs per ZUMA embedded LUT are required.

In addition, the speedup achieved by the CARBON-Razor architecture is presented and analyzed. Results are compared to the unmodified CARBON architecture. When the schedule is extended by a fixed number of cycles, which the number of errors will not extend, performance was increased from 3-20%, depending on the benchmark. When no fixed error cycles are used, and the schedule is extended as needed, the performance is increased from 5-21%.

5.2 ZUMA Results

This section begins with an outline of the parameters chosen for the most efficient architecture with justifications. We then present the area results for the Xilinx Virtex 5, with the efficiency gained by each of our implementation optimizations. This is followed by the resource usage for the configuration controller. We then present area results for the Altera Stratix IV version, which suffers due to CAD limitations.

5.2.1 Architecture Parameters

Architectural parameters are derived from modeling presented in Chapter 3. The plot in Figure 5.1 shows the relationship between the cluster size (N) and the area usage derived from that modeling. A cluster size of 8 is found to have the best area usage. The routing width of the architecture is 112, with a wire length of 4. The channel width is determined by the highest width needed by the MCNC benchmarks when routed in VTR, as presented in Table 5.1. The final architecture is given as eight 6-LUTs per cluster (k=6,N=8), a routing width of 112 and routing wire length of 4, 28 inputs, an absolute cluster input flexibility of 6, a fractional cluster output flexibility of 3/8 and a switch block flexibility of 3. The input and output flexibilities are greater than what is used in the generic design, as the extra flexibility is supported by the fixed sizes of the LUTRAMs without an increase in logic usage. The IIB then distributes 36 (28 + 8 feedback) inputs to 48 outputs, which we can build using LUTRAM crossbars as shown in Chapter 3.

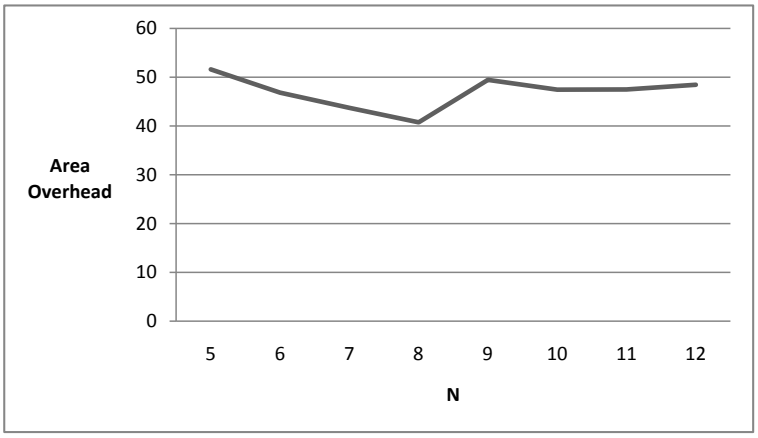

Figure 5.1: Modeled logic usage vs cluster size

5.2.2 Xilinx Results

The Table 5.2 presents Xilinx Virtex 5 results for the baseline, and the improvements from both LUTRAM utilization and the new IIB separately. The LUTRAMs resulted in an over 50% reduction in LUT resources, coming largely from the eLUT. The new IIB reduced the area another 20%, all from the IIB. Together, the results are a two thirds reduction in LUT usage, and an almost complete elimination of all flip-flop usage, in comparison to the generic implementation.

In comparison, a ZUMA architecture corresponding to the vFPGA parameters was also implemented, and is presented in Table 5.3, using a Spartan 3 architecture. The original vFPGA was implemented on a Spartan IIe FPGA, which also uses 4-input LUTs. The Spartan IIe could not be used for ZUMA, as it does not support LUTRAMs. The vFPGA architecture uses a cluster comprised of four 3-LUTs, and a routing width of 32. The

Benchmark	6-LUTs	Array Size	Minimum Routing Track Width
alu4	1173	13x13	60
apex2	1478	14x14	102
bigkey	907	16x16	46
des	554	18x18	50
diffeq	1245	11x11	80
dsip	904	16x16	50
elliptic	3255	17x17	100
ex1010	3093	20x20	110
ex5p	740	10x10	98
frisc	3814	20x20	112
misex3	1158	13x13	78
pdc	3629	22x22	104
s298	1309	13x13	74
s38417	4501	21x21	78
Seq	1325	13x13	104
Spla	3005	20x20	106
Tseng	1182	10x10	76
Maximum	-	-	112

Table 5.1: Track utilization for ZUMA architecture using VPR 6.0

ZUMA design makes use of 4-input LUTRAMs for all logic and routing. As the global routing tracks are connected directly to the LUT inputs, the two-stage crossbar is not used in this design. In the vFPGA results, all resources used for the switch box and input box is presented as one number, as they are presented in the table. As shown in the table, ZUMA shows a large reduction in both routing resources, as well as the resources used to implement the eLUTs. The resource usage is less than one-half of the previous results, a significant improvement over the 90x overhead from the vFPGA paper.

	Generic Architecture		**With LUTRAMs**		**With Two-Stage IIB**		**Final Architecture**	
	Host LUTs	Host FFs	Host LUTs	Host FFs	Host LUTs	Host FFs	Host LUTs	Host FFs
Switch Block	121	156	104	0	121	156	104	0
Input Block	56	288	56	0	56	288	56	0
Crossbar	288	288	248	0	168	152	144	0
eBLEs	528	520	16	8	528	520	16	8
Total	993	1212	424	8	993	1116	320	8
Per eLUT	124.1	151.5	53	1	109	139	40	1

Table 5.2: Resource breakdown per cluster for Xilinx version for generic architecture, LUTRAMs only, two-stage crossbar only, and with both optimizations

5.2.3 Configuration Controller

The area usage for the configuration controller is presented in Table 5.4, versus a baseline design, both compiled to a Xilinx Virtex 5 FPGA. The baseline design is composed of a simple counter and address decoder for each memory. Our design increases the usage of flip-flops, but reduces the usage of logic significantly, which is the limiting resource in ZUMA. Although the controller uses a larger number of flip-flops, these can be packed into LUTs with unused flip-flops in most new architectures. Hence the flip-flops will be free resources. Hence, the configuration controller is small compared to the entire ZUMA design.

5.2.4 Altera Results

The Altera version is currently less efficient than Xilinx, as a LAB can only be used to implement one LUTRAM, regardless of the width of the LUTRAM. As well it uses additional registers for the write port. This greatly restricts the size of the array that can be implemented on a single FPGA, though we are hopeful that a more efficient LUTRAM mapping may

	ZUMA Architecture		**vFPGA Architecture**	
Area	Host LUTs	Host FFs	Host LUTs	Host FFs
Switch Block	64	0	332	158
Input Block	66	0	x	x
eBLEs	4	4	22	42
Total	134	4	354	200
per eLUT	33.5	1	88.5	50

Table 5.3: Implementation of ZUMA FPGA architecture on Xilinx Spartan 3 (both hosts use 4-LUTS, both architectures configured with identical parameters k=3,N=4,w=32,L=1)

		Controller		ZUMA	
Controller Type	Array Size	Host FFs	Host LUTs	Host FFs	Host LUTs
Baseline	2x2	13	160	32	1280
ZUMA	2x2	128	13	32	1280
Baseline	8x8	32	2960	512	20480
ZUMA	8x8	2048	32	512	20480

Table 5.4: Resource usage of ZUMA configuration controller compared to the rest of ZUMA overlay on Xilinx Virtex 5

be available in future software releases. Altera has been contacted about this limitation, but did not have time to add a workaround. Given this drawback, the switch block and input block are implemented in the Altera ZUMA version using generic Verilog multiplexers. Given the same architectural details used to produce the Xilinx results, the resources consumed per tile are presented in Table 5.5, as compiled to an Altera Stratix IV FPGA. Host LUT usage is reduced similarly to Xilinx when compared to the generic version, while flip-flop usage is only halved.

5.2.5 Verification and Timing Performance

The ZUMA overlay was verified in simulation and compiled on board Xilinx and Altera FPGAs. Several Verilog circuits were compiled to a bitstream and configured in the overlay from an external memory. The outputs were compared against the benchmark Verilog, which was also compiled or simulated. The outputs were found to be identical.

On-board testing of timing was performed on an Altera Stratix III EP3SL150F1152C3ES FPGA, as a Stratix IV board was not available at the time. Two benchmarks, a 32 bit OR and a 32 bit AND circuit were compiled to the overlay

	Xilinx Virtex 5 Implementation		Altera Stratix IV Implementation	
	Host LUTs	Host FFs	Host LUTs	Host FFs
Switch Block	104	0	121	156
Input Block	56	0	56	84
Crossbar	144	0	152	137
eBLEs	16	8	16	40
Total	320	8	345	417
Per eLUT	40	1	43.1	52.1

Table 5.5: Resource breakdown per cluster for Xilinx and Altera implementations

and configured into a n=8,k=6, 3x3 ZUMA overlay. The inputs and outputs of the circuit were registered, and the inputs were fed with a set of randomly generated test vectors set by a NIOS processor. One clock cycle after the inputs are set, the outputs of the circuit are captured, and read by the processor, which determined if they were correct. The clock frequency was swept to determine when incorrect results are captured. The circuits worked correctly until the frequency reached 70-100 MHz, varying from compilation to compilation. In comparison, the same benchmarks achieve 500-700 MHz when compiled directly to the host FPGA.

5.3 CARBON-Razor Results

The results of the CARBON Razor architecture are presented in this section. The area and performance of implementing the CARBON architecture with Razor timing error detection and correction on an FPGA is presented first. Then, observations about the impact of the instruction schedule properties are observed.

5.3.1 Area

Both the unmodified CARBON and CARBON-Razor architectures were compiled onto an Altera DE3 evaluation board with a Stratix III EP3SL150F1152C3ES FPGA. Logic and register usage are reported for both architectures in Table 5.6. The ALM and register usage goes up slightly to implement Razor, but RAM and DSP usage is identical for both architectures. With this area footprint, a 10×10 array of CGs can fit into the largest Stratix III, which has 337,500 ALMs. The current implementation supports a schedule length

Area	ALMs	Registers	BRAM Bits	DSP 18-bit Elements
CARBON CG	2,958	304	15,688	4
CARBON-Razor CG	3,082	517	15,688	4

Table 5.6: Resource usage of CARBON implementation per tile

(SL) up to 256 instructions.

5.3.2 Performance

Tests were performed by overclocking both the base CARBON design, and CARBON with Razor error tolerance. The designs were compiled, and the maximum frequency from the static timing analyzer (STA) was recorded. The designs were then loaded into the FPGA, and a set of benchmarks were run. The frequency, and the shadow register delay for the Razor version, were modified to find the maximum performance, which was compared.

The CARBON system was paired with a Nios II processor to perform a large number of tests using random data sets as depicted in Figure 5.2. The data sets are first run with the CARBON-Razor system fed by a slower clock. In each user cycle the inputs are changed and the outputs are read. The frequency and Razor clock delay of the clock is then increased until the circuit fails to produce the correct result, determined by comparison to results obtained at a lower speed. The Quartus STA initially gave an F_{max} of 90 MHz for the baseline CARBON, and 88 MHz to the CARBON-Razor, which loses speed due to the extra multiplexer in the write path of the memories. The STA results are also presented in Table 5.7.

Thousands of iterations of each benchmark were run at various frequencies, and the maximum performance (clock speed divided by the maximum

	Array Size	F_{max}
Baseline	2x2	90
CARBON-Razor	2x2	88

Table 5.7: Maximum operating frequency determined using Quartus STA

schedule length) was chosen. The baseline designs frequency was swept from the STA result to the point of failure. The CARBON-Razor designs frequency was swept until it failed, but when a failure occurred the delay to the shadow register was increased. Increases in the frequency and shadow register delay were performed until the design would not pass at a frequency regardless of the shadow register delay. The maximum overall performance, which is determined by the frequency divided by the minimum schedule length when used with a hard system deadline, and the frequency divided by the average schedule length when used as a compute accelerator for each benchmark was recorded.

Four benchmarks were used to gauge the performance of the circuit. The first benchmark is composed of a chain of random operations on input data. The dataflow and DSP computational circuits PR and WANG were also used. The final benchmark is a calculation of the mean of 256 circuit inputs. Each benchmark was compiled from Verilog into a schedule, which is used to generate the instruction memories. The compilation takes less than five seconds in total.

The results in Table 5.8 show performance of each of the benchmarks. The circuit was run for thousands of cycles with random inputs. The CARBON-Razor frequency and phase lag (Δ_1) controlling the shadow reg-

isters was modified. The amount of improvement possible is shown to be highly benchmark-dependent and data-dependent. For example, benchmarks where the multiplier is never used, or only narrow data is multiplied should get much higher speedups. The harmonic mean of each F_{max} is calculated in the final column. On average, the user F_{max} increases by 13%.

If a hard system deadline is not needed for the outputs, for example if the CARBON overlay is used as a compute accelerator to a processor, we no longer need to budget for recovery cycles, and can stall only when needed. In this case, the average F_{max} of the circuit improves more with the Razor system, especially for the benchmarks with lower speedups. As in the previous case, thousands of iterations of each benchmark were run at various frequencies, and the maximum performance (clock speed divided by the average schedule length) was chosen. Table 5.9 demonstrates the improvement of the average F_{max} compared to the unmodified circuit. Overall, benchmarks that achieved lower results show better speedups compared to the hard system deadline version. In this case, the average F_{max} improves from 6-21% for each benchmark. The harmonic mean of each F_{max} is calculated in the final column. On average, the user F_{max} increases by 14%.

5.3.3 Error Probability Observations

From the experiments done in the previous section, a number of observations can be made about the distribution of Razor errors. The number of errors is influenced by the operations performed by the circuit, as well as the data operated on. The four benchmarks used are individually discussed.

From the analysis of the Quartus Static Timing Analyzer it is clear the

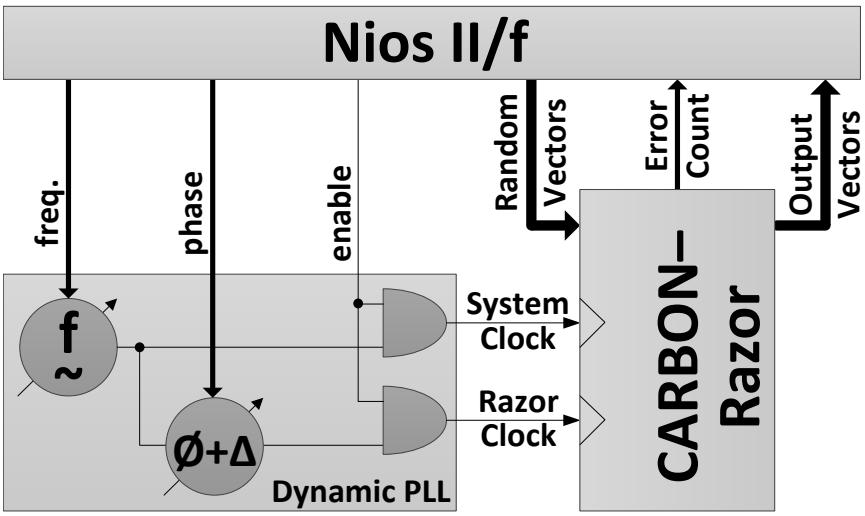

Figure 5.2: CARBON-Razor testing system

longest register-to-register delays run through the multiplier. When paths through the multiplier are removed, other operations also show significantly higher delays than average. The probability of failure thus will depend on the operation performed. The probability of the path failing is also determined by the value of the inputs to the circuit, as higher order bits of data will take longer to propagate to the registers for a multiply operation. This was observed in our tests, as when inputs vectors were all set to high, the worst case error count was achieved in many cases.

The error rate is determined primarily by the program being run, but also by the data being used. If the errors occurred reliably at a certain instruction, the CARBON CAD system and architecture can be arranged to give the offending operation another cycle to complete at compile time. In this case, the run-time error detection would not be needed, and could

be disabled.

The 'Random Operation' benchmark is composed of a series of random operations, using most paths through the ALU. This leads to a wide range of delays to the shadow register, including a number with high delays. Accordingly, this allows the frequency to be increased a significant amount before a large number of errors are detected, leading to many error cycles being needed, and causing slowdown.

Both PR and WANG both use a number of multiply and addition operations. Since most operations occur on the same paths, the frequency can only be increased a small amount before many error cycles are needed, and the circuit begins to slow down.

The 'Mean' benchmark is composed of a large number of addition operations, and a final shift. The shift goes through a multiplier, so it is much slower than the additions. Therefore the shift operation will limit the amount of overclocking for the base design. This error can be tolerated using just one cycle of slack. Since the schedule is also longer than the other benchmarks, it results in the best speedup overall.

5.3.4 Summary

This chapter presented results for both the fine and coarse grain architectures. The final resource usage of the ZUMA architecture is less than one third of the generic design, and less than one-half of previous research attempts [17], at as little as 40 host LUTs per ZUMA embedded LUT. The addition of Razor error detection to the CARBON overlay resulted in a speedup of up to 20% percent versus the baseline, depending on benchmark

used.

Benchmark			Best CARBON (no Razor)		Best CARBON-Razor				
Name	SL	N	System F_{max}	User F_{max}	e	System F_{max}	User F_{max}	Shadow Clock Lag (Δ_1)	Speedup
Random Ops	24 cycles	2	135 MHz	5.63 MHz	2 cycle	163 MHz	6.35 MHz	2ns	11%
Wang	28 cycles	2	131 MHz	4.67 MHz	1 cycle	144 MHz	4.96 MHz	.5 ns	6%
Mean(256)	67 cycles	2	121 MHz	1.80 MHz	2 cycle	147 MHz	2.16 MHz	1.5 ns	20%
PR	29 cycles	2	136 MHz	4.68 MHz	1 cycle	145 MHz	4.83 MHz	.5 ns	3%
Average			130.4 MHz	3.44 MHz		149.4 MHz	3.88 MHz		13%

Table 5.8: Performance of benchmarks given hard system deadline (FPGA mode)

Benchmark			Best CARBON (no Razor)		Best CARBON-Razor				
Name	SL	N	System F_{max}	User F_{max}	Average e	System F_{max}	Average User F_{max}	Shadow Clock Lag (Δ_1)	Speedup
Random Ops	24 cycles	2	135 MHz	5.63 MHz	1.2 cycle	163 MHz	6.46 MHz	2ns	14%
Wang	28 cycles	2	131 MHz	4.67 MHz	.2 cycle	144 MHz	5.11 MHz	.5 ns	9%
Mean(256)	67 cycles	2	121 MHz	1.80 MHz	.9 cycle	147 MHz	2.16 MHz	1.5 ns	21%
PR	29 cycles	2	136 MHz	4.68 MHz	.5 cycle	145 MHz	4.91 MHz	.5 ns	5%
Average			130.4 MHz	3.44 MHz		149.4 MHz	3.93 MHz		14%

Table 5.9: Performance of benchmarks for average number of cycles used (Compute Accelerator mode)

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we developed two overlay architectures. The first, ZUMA, is a fine-grain architecture based on a traditional island-style FPGA architecture. The second, CARBON, is a coarse-grain architecture implementing the time multiplexed processing elements of the MALIBU architecture.

The fine-grain ZUMA architecture enables a number of applications and addresses research needs. ZUMA can help address these goals by providing open access to all of the underlying details of an FPGA architecture, as an instrument for study and for implementation. Initial implementation was found to suffer from large area overheads. To address this we make novel contributions towards implementing logic and routing networks in FPGAs using programmable LUTRAMs. The final resource usage of the ZUMA

architecture is less than one-third of the original generic design, and less than one-half of previous research attempts [17]. It uses as little as 40 host LUTs per ZUMA embedded LUT.

The second overlay, CARBON, is a coarse-grain array of processing elements based on the coarse-grain component of the MALIBU architecture. The MALIBU architecture addresses a number of problems that occur as FPGAs grow and evolve, and features better mapping of word-oriented circuits and faster compile times. By using an FPGA to implement the MALIBU CG, we can take advantage of these properties in a real system, as well as enabling us to explore changes to improve performance. This implementation allowed us to implement circuit level timing improvements and error detection to increase performance. We have extended Razor circuit level timing error detection, which achieves error tolerance through comparing registered data to a shadow register fed by a delayed clock, from 1D pipelines to 2D arrays. The Razor error tolerance is integrated into CARBON, with development of a Razor enabled memory block and a 2D error propagation controller. Performance was increased by 3-20% over our base design depending on the benchmark, or 13% on average. If a hard system deadline is not required, and a consistent execution schedule is not needed, benchmarks that showed less speedup improve significantly, and we can improve the performance by 6-21% overall, or 14% on average.

This thesis concretely demonstrates significant area and delay enhancements to fine-grain and coarse-grain FPGA overlays. The first overlay developed was a fine grain architecture based on a traditional FPGA, which was found to suffer from large area overheads. To address this we made

contributions towards implementing logic and routing networks in FPGAs using programmable LUTRAMs, and a novel two-stage local interconnect. The second overlay, a coarse-grain array of processing elements, which is inherently slow due to time multiplexing. To improve performance we extend a circuit level technique for pipelined circuits called Razor to detect and tolerate timing errors introduced by overclocking.

6.2 Future Work

Future work can be done to integrate the clock skewing approach to hide read and write latency to memory we developed in [4] into the CARBON architecture. This would entail clocking the read and write ports of the CARBON memories with separate clocks that are skewed slightly ahead or behind of the memory. As memory is written to a bypass register, one cycle communication is still possible, while hiding all memory latency. This approach could be implemented alone, or along with Razor for additional performance benefit.

By further analysis of data dependencies, additional logic could be added to CARBON to prevent stalls from being propagated to neighbours when no data dependencies are present. If an error occurs a stall may not be needed if two conditions are met: the memory is not being written to on cycle X+1, and the value is not being read in cycle X+1. In this case, the value can be written from the shadow register to the memory without stalling the processing element. Dependencies however could occur through multiple neighbours, for example if the stall is not passed to the east neighbour, it may travel north, east and south in three cycles and the stall would occur

on that cycle. More logic would then be needed to prevent the errors being passed back to the first CG. The benefits of this logic are highly dependent on the properties of the design being run, and are left for future work.

The timing variation exploited in the CARBON-Razor design could also be utilized with other methods. Architecture and CAD can be modified to utilize the distribution of timing path delays in the CARBON and MALIBU systems, to increase performance without requiring error detection. If the maximum delay of an operation can be calculated based on instruction parameters, such as the operation and result width, a technique such as multicycle paths could be used to boost performance deterministically.

More work can still be done to add features and capabilities both on the CAD and architecture design of ZUMA. Timing can be further considered, both to increase the speed of the underlying architecture, and the ability of the CAD tools to exploit timing variation. The addition of more architectural features, such as multiple clock networks, and asynchronous set and reset of registers, and further work on integrating heterogeneous elements into the CAD flow will further increase the applicability of the ZUMA architecture. More work can be done to integrate the CAD tools and configuration to enable compilation to the fully hybrid MALIBU system.

Bibliography

[1] V. Aken'Ova, G. Lemieux, and R. Saleh. An improved "soft" eFPGA design and implementation strategy. In *IEEE Custom Integrated Circuits Conference*, pages 179 – 182, sept. 2005. doi:10.1109/CICC.2005.1568636.

[2] P. Bellows and B. Hutchings. JHDL – an HDL for reconfigurable systems. In *IEEE WORKSHOP ON FPGAS FOR CUSTOM COMPUTING MACHINES*, pages 175–184, 1998.

[3] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Boston: Kluwer Academic Publishers, 1999.

[4] A. Brant, A. Abdelhadi, A. Severance, and G. Lemieux. Pipeline frequency boosting: Hiding dual-ported block RAM latency using intentional clock skew. In *International Conference on Field Programmable Technologies*, pages 307–314. ACM, 2012.

[5] D. Bull, S. Das, K. Shivashankar, G. S. Dasika, K. Flautner, and D. Blaauw. A power-efficient 32 bit ARM processor using timing-error detection and correction for transient-error tolerance and adaptation to PVT variation. *IEEE Journal of Solid-State Circuits*, 46(1):18–31, January 2011.

[6] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. Lemieux. VEGAS: soft vector processor with scratchpad memory. In *ACM/SIGDA International Symposium on Field programmable Gate Arrays*, pages 15–24, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0554-9. doi:10.1145/1950413.1950420.

[7] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, March 1953.

[8] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *IEEE/ACM International Symposium on Microarchitecture*, pages 7–18, December 2003.

[9] D. Grant, C. Wang, and G. Lemieux. A CAD framework for MALIBU: an FPGA with time-multiplexed coarse-grained elements. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 123–132, New York, NY, USA, 2011. ACM. doi:10.1145/1950413.1950441.

[10] J. Hillman and I. Warren. An open framework for dynamic reconfiguration. In *International Conference on Software Engineering*, pages 594 – 603, May 2004. doi:10.1109/ICSE.2004.1317481.

[11] N. Kafafi, K. Bozman, and S. J. E. Wilton. Architectures and algorithms for synthesizable embedded programmable logic cores. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 3–11, New York, NY, USA, 2003. ACM. ISBN 1-58113-651-X. doi:10.1145/611817.611820.

[12] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. Wilson, M. Wrighton, and A. DeHon. Packet switched vs. time multiplexed FPGA Overlay Networks. In *IEEE Field Programmable Custom Computing Machines, 2006*, pages 205 –216, April 2006. doi:10.1109/FCCM.2006.55.

[13] I. Kuon and J. Rose. Area and delay trade-offs in the circuit and architecture design of FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 149–158, 2008.

[14] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. RapidSmith: Do-it-yourself CAD tools for Xilinx FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 349 –355, Sept. 2011. doi:10.1109/FPL.2011.69.

[15] G. Lemieux and D. Lewis. Using sparse crossbars within LUT clusters. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 59–68, New York, NY, USA, 2001. ACM. ISBN 1-58113-341-3. doi:10.1145/360276.360299.

[16] G. Lemieux and D. Lewis. *Design of Interconnection Networks for Programmable Logic*. Boston, MA: Kluwer, 2003.

[17] R. Lysecky, K. Miller, F. Vahid, and K. Vissers. Firm-core virtual FPGA for just-in-time FPGA compilation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 271–271, New York, NY, USA, 2005. ACM. ISBN 1-59593-029-9. doi:10.1145/1046192.1046247.

[18] A. Patel, C. Madill, M. Saldana, C. Comis, R. Pomes, and P. Chow. A scalable fpga-based multiprocessor. In *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 111 –120, April 2006. doi:10.1109/FCCM.2006.17.

[19] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson. The VTR project: architecture and CAD for FPGAs from Verilog to routing. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 77–86, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1155-7. doi:10.1145/2145694.2145708.

[20] S. Shukla, N. Bergmann, and J. Becker. QUKU: A FPGA based flexible coarse grain architecture design paradigm using process networks. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1 –7, March 2007. doi:10.1109/IPDPS.2007.370382.

[21] S. Singh. Designing reconfigurable systems in Lava. In *International Conference on VLSI Design*, pages 299 – 306, 2004. doi:10.1109/ICVD.2004.1260941.

[22] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French. Torc: towards an open-source tool flow. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 41–44, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0554-9. doi:10.1145/1950413.1950425.

[23] G. Stitt and J. Coole. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters*, 3 (3):81 –84, Sept. 2011. ISSN 1943-0663. doi:10.1109/LES.2011.2167713.

[24] S. Wilton, N. Kafafi, J. Wu, K. Bozman, V. Aken'Ova, and R. Saleh. Design considerations for soft embedded programmable logic cores.

IEEE Journal of Solid-State Circuits, 40(2):485 – 497, Feb. 2005. ISSN 0018-9200. doi:10.1109/JSSC.2004.841038.

[25] S. J. E. Wilton, N. Kafafi, J. C. H. Wu, K. A. Bozman, V. O. Akenova, and R. Saleh. Design considerations for soft embedded programmable logic cores. *IEEE J. Solid-State Circuits*, pages 485–497, 2005.

[26] P. Yiannacouras, J. Steffan, and J. Rose. VESPA: Portable, scalable, and flexible FPGA-based vector processors. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 61–70, 2008.