

Context Switching Strategies in a Run-Time Reconfigurable system

Kiran Puttegowda

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Dr. Peter M. Athanas, Chair

Dr. Mark T. Jones

Dr. Joseph G. Tront

April 16, 2002

Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: configurable computing, FPGA, run-time reconfiguration, virtual hardware,
context switching, multi-context

Copyright © 2002, Kiran Puttegowda

Context-Switching Strategies in a Run-Time Reconfigurable system

Kiran Puttegowda

Abstract

A distinctive feature of run-time reconfigurable systems is the ability to change the configuration of programmable resources during execution. This opens a number of possibilities such as virtualisation of computational resources, simplified routing and in certain applications lower power. Seamless run-time reconfiguration requires rapid configuration. Commodity programmable devices have relatively long configuration time, which makes them poor candidates for run-time reconfigurable systems. Reducing this reconfiguration time to the order of nano seconds will enable rapid run-time reconfiguration. Having multiple configuration planes and switching between them while processing data is one approach towards achieving rapid reconfiguration. An experimental context switching programmable device, called the Context Switching Reconfigurable Computer (CSRC), has been created by BAE Systems, which provided opportunities to explore context-switching strategies for run-time reconfigurable systems. The work presented here studies this approach for run-time reconfiguration, by applying the concepts to develop applications on a context switching reconfigurable system. The work also discusses the advantages and disadvantages of such an approach and ways of leveraging the concept for efficient computing.

To my Parents and Sisters

without whom nothing would have been possible.

Acknowledgements

While working on my research, a lot of people have supported me with tremendous amount of motivation and co-operation. In the process of writing this chapter I expect to fall short of extending deserving thanks to all of them. The outcome is way below the amount of gratitude intended to be expressed due to my inability to find enough words to express my thanks completely. I begin this section by thanking all persons from past who helped me reach this position due to which all these efforts were made possible.

I first spoke to Dr. Peter M. Athanas before coming to Virginia Tech and expressed my intention to work on configurable computing. He went out of his way to let me work from the day I started my masters. He recognized my interests and let me work in projects pertaining to my interests. I'm highly indebted for his insightful advice and encouraging words which helped me meander through my graduate research. I will cherish my association with him.

I would like to thank Dr. Mark T. Jones and Dr. Joseph G. Tront for their support and motivation. I sincerely appreciate their willingness to serve on my committee.

Special thanks goes to David I. Lehn for all his help during the research. Without his help during our long nights trying to get the system working, the research would not have been possible. I wish him success for the future. I also thank Dr Jae H. Park for his motivation and help.

The people at the configurable computing lab, who form one of the smartest technical communities at Virginia Tech, had always been willing to help me out with ideas and advice. I express my thankfulness to each of them for providing a cordial and a highly motivating atmosphere at the Lab.

I convey my gratitude to my friends in Blacksburg who have been supportive all through my stay here. My life at Blacksburg would not have been the same without them.

Finally, I would like to thank my family for their unconditional love and faith on me. Their persistent support and high expectations from me has driven me throughout my life. Without them being the excellent people that they are, I would never have become half the person that I am today. With their continued wishes and backing, I hope to live up to their expectations.

Kiran Puttegowda

April 2002

Contents

Table of Contents	v
List of Figures	viii
List of Tables	x
1 Introduction	1
1.1 Motivation	2
1.2 Contributions	2
1.3 Thesis Organization	3
2 Background	5
2.1 Configurable computing	5
2.2 Run-Time Reconfiguration (RTR)	8
2.3 Virtual Hardware	12
2.4 Multi-context	13
3 Hardware Platform	17
3.1 Reconfigurable Computing Module (RCM)	17
3.2 CSRC Architecture	19
3.3 Support FPGA design	22

4	Run-Time Environment	24
4.1	Host API	25
4.2	RCMOS	27
4.3	Configuration Caching	27
5	Application Control	29
5.1	Programmable Hardware Control	29
5.2	Context Switching Control	30
5.2.1	Host-driven RTR	31
5.2.2	FSM-driven RTR	31
5.2.3	Data-driven RTR	32
6	FSM-Driven RTR	33
6.1	Motion detection Algorithm	33
6.2	Algorithm	34
6.3	Implementation	35
6.4	Application Support	37
7	Data-Driven RTR	40
7.1	Enigma Encryption	40
7.2	Algorithm	41
7.3	Implementation and Network Processing Model	42
8	Analysis and Results	45
8.1	Area	45

8.2	Average reconfiguration Time	47
8.3	Bitstream size	50
8.4	Conclusion	51
	Bibliography	53
	A Video Applications Demonstration	57
	Vita	60

List of Figures

2.1	SPLASH-2: System architecture	6
2.2	GARP: Block diagram	9
2.3	Stallion: System architecture	10
2.4	An illustration of the stream format for Stallion	11
2.5	WASMII: Chip architecture	14
3.1	The RCM Board Block diagram	18
3.2	CSRC Architecture	20
3.3	Context Switching Logic Cell	21
4.1	Run-time environment stack	25
4.2	Configuration caching hierarchy	28
5.1	Context switching control routes	30
6.1	Flow diagram of the motion detection algorithm	34
6.2	Motion detection algorithm mapped on the RCM platform	36
6.3	State diagram programmed to control the motion detection algorithm	38
7.1	Simplified description of enigma machine	41

7.2	The enigma application mapped to the RCM platform	43
7.3	Format of the data packet used for enigma encryption	44
8.1	Plot of average reconfiguration time	48
A.1	Screen capture of video processing with pass through filter	58
A.2	Screen capture of video processing with the delay filter with fading	58
A.3	Screen capture of video processing with the difference filter	59
A.4	Binary image generated by the motion detection algorithm demonstration	59

List of Tables

8.1	Area requirement for each application. Second column shows the area requirement for each context of application in a multi-context device	46
8.2	Area requirement for motion detection application for different number of application contexts	51

Chapter 1

Introduction

Performance expectations and programmability requirements of computing resources are increasing tremendously. These requirements cannot be met by conventional computing resources like microprocessors or Application Specific Integrated Chips (ASIC). Conventional processors have lower performance due to forced serialization of intrinsically parallel operations and excessive instruction bandwidth for regular data-flow dominated computations. ASICs offer better performance, by eliminating all these deficiencies, but increases the design time for the system because of its non-programmability. Configurable Computing Machines (CCM) provide high performance coupled with smaller design time by being programmable. Traditional Field Programmable Gate Array (FPGA) based computing offers several advantages with speeds approaching ASICs and design time comparable with microprocessor and Digital Signal Processor (DSP) based systems. However, most commercial FPGAs suffer from limitations such as long configuration time and limited reconfiguration bandwidth. To overcome the shortcomings of conventional processors and FPGAs, efforts have been ongoing to create innovative CCMs architectures as an alternative to traditional FPGAs by using hardware that can be reconfigured on the fly. One such approach is called Run-Time

Reconfiguration (RTR).

1.1 Motivation

Traditional FPGAs have relatively long configuration times [1], and applications that use run-time reconfiguration on FPGAs often suffer from this effect. Any configuration delay increases the overall computation time. In an application that requires only a part of the computational logic at a time, there is the possibility of sharing the same computational resources. The devices used to accomplish this store multiple configurations called *contexts* in different sets of internal RAM. Each programmable part of the device is controlled by multiple RAM units, where each RAM bit configures the programmable unit. Global context lines act as addresses for these RAM units to activate a context. Any one device context can be active at a time and another context can be activated in as fast as one clock cycle. This internal configuration activation process is known as *context switching*. Such a device is known as *multi-context device*. This fast switching between contexts reduces reconfiguration time tremendously. This technique enables a number of additional possibilities not achievable with traditional programmable logic like virtual hardware, simplified routing, higher data bandwidth and low power dissipation.

1.2 Contributions

This thesis explores the possibilities of using multi-context devices in RTR systems and analyses the benefits and costs of context switching. The research contributes the following:

- It analyses the effects of context switching on system performance by arriving upon

unique expressions for average configuration time of the device in different operating scenarios.

- The research gives an insight into ways of efficient application development on a context switching reconfigurable system.
- This study helps in the selection of applications that would use the advantages of context switching to the maximum to achieve better performance.
- It suggests ways of partitioning applications for a multi-context configurable system based on the device parameters.

Applications were developed as part of the research to validate all these evaluations. These applications will also be discussed in detail. It gives a comparative study of context switching as against other approaches for run-time reconfiguration.

1.3 Thesis Organization

The thesis is organized in the following manner. Chapter 2 gives a background to the work undertaken in the thesis. It explains the concepts upon which this research is based, along with citing other works that led to this research. Chapter 3 describes the experimental system on which context switching was evaluated. It explains the multi-context device architecture that is present on the platform. The system required the development of an environment for implementing run-time reconfigurable applications. Chapter 4 discusses the run-time environment of the system. Chapter 5 describes ways of controlling the applications running on the platform. The next two chapters discuss in detail the applications developed. Chapter 6 discusses the Motion detection algorithm and Chapter 7 discusses

the enigma encryption application. Finally Chapter 8 presents analysis of the concept based upon the applications developed and the results obtained.

Chapter 2

Background

This chapter introduces the concepts used for the work undertaken as part of the thesis. Configurable computing emerged as a solution to achieve higher flexibility (programmability) and performance when compared to conventional computing resources. Run-time reconfiguration tries to increase the effective utilization of silicon during processing. This can be achieved by the concept of virtual hardware or hardware paging. Multi-context configuration is an effective means of implementing virtual hardware. These concepts are discussed in detail in the following sections.

2.1 Configurable computing

A programmable logic device is an integrated circuit that provides a programmable interface through which the user can dynamically instantiate and implement almost any desired set of hardware images on the available circuit. These devices typically have a programmable set of logic blocks and routing structure. Configurable computing uses these devices to achieve hardware programmability mentioned before. Programmability of the logic block function-

ality and routing structure is achieved through programmable points. In today's technology, these programmable points are typically memory cells (most often SRAM based) or anti-fuses. Anti-fuses are one-time programmable, which when blown create a connection between two conducting materials. A *configuration* is the set of program bits needed to specify the behavior of all the programmable points. Alternately, it may also refer to the hardware image realized on the programmable logic device as a result of programming these points. Such a hardware image in the configurable device is also referred to as a *context*. The earliest known computing system based on configurable hardware was proposed and implemented at UCLA [2]. It was a hybrid machine consisting of a general purpose processor augmented with high speed logic devices which were interconnected via application specific interconnect. In the mid 1980s Xilinx introduced Field Programmable Gate Arrays (FPGA) [1]. This led to research in a lot of computing systems based on FPGAs. PRISM [3] suggested ways to couple a reconfigurable logic with a general purpose microprocessor.

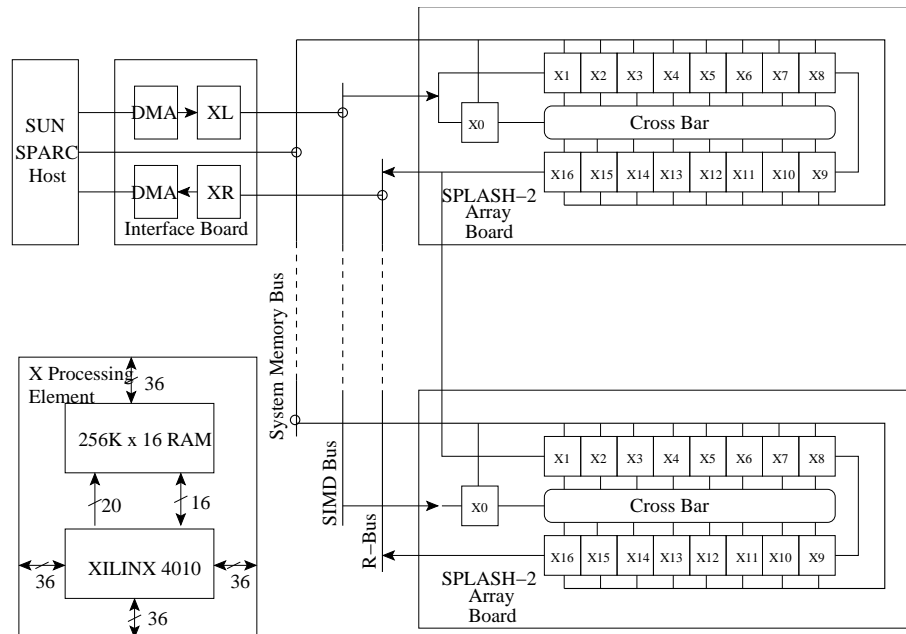


Figure 2.1: SPLASH-2: System architecture

SPLASH-2 [4] was one such reconfigurable board used as a co-processor board to a Sun SPARC workstation. SPLASH-2 was specifically designed to support high-performance linear systolic applications. Figure 2.1 shows the architecture of SPLASH-2. The SPLASH-2 board had sixteen XC4010 FPGAs connected in linear systolic array. There was also a crossbar connecting all these FPGAs. In addition to the systolic data-path a global broadcast data-path and user-defined crossbar data-path were available for custom communication networks. The system could be scaled by adding additional boards which have the feature of continuing the systolic data-path as shown in Figure 2.1. Each of the boards was connected to a local 256K x 16 bit memory that was also mapped to the address space of the host processor. FIFOs on both ends of the systolic array maintained high through-put and buffer data to the host processor. Connections between neighboring FPGAs and the cross bar were 36-bit wide, enough to support 32-bit data and 4-bit tag. Another FPGA on the board did all the control operations.

SPLASH-2 application was developed by first manually partitioning the design into FPGA sized pieces. A single partitioned design was described in VHDL for each of the FPGAs in the system. These designs were simulated in the context of the whole system for initial debugging. This VHDL design was synthesized and downloaded to the actual hardware. Further verification was done by the run-time debugging tools. The manual partitioning of the algorithm was difficult which required the programmer to go through the design cycle in a couple of iterations to fit the partition into a single FPGA. Applications run on SPLASH-2 reported significant gain over conventional computing systems. SPLASH-2 achieved two orders of magnitude speedup on genome sequence matching compared to supercomputers of that time (Cray2). A SPLASH-2 system used for image-processing applications reported significant speed-ups for a wide variety of algorithms [5].

2.2 Run-Time Reconfiguration (RTR)

Reconfiguration can be done in two ways based on the temporal reconfigurability of the devices; *static reconfiguration* and *run-time reconfiguration*. Static reconfigurable systems (also known as rapid prototyping) are those which achieve reconfiguration of the programmable parts before the program is executed. SPLASH-2 is an example of such a system. As explained in Section 2.1 the board is configured before doing any computation. RTR systems reconfigure the programmable device dynamically during the program execution. Run-time reconfiguration enables silicon to be time-shared across multiple tasks and hence improve performance. Reconfiguration in an RTR system can be driven either dynamically by the data generated while processing (*data-driven RTR*) or statically by the host machine controlling the reconfigurable hardware (*host-driven RTR*). Various approaches have been proposed for run-time reconfiguration. Significant among them has been coupling a general purpose microprocessor with a reconfigurable functional unit; thus doing an instruction-set metamorphosis [3]. This would enhance the instruction set to have customized functional units to which the new instruction can send data to be processed. The instruction set will also have a few instructions to achieve configuration control.

An example of a host-driven RTR system is Gate Array Reconfigurable Processor (GARP) [6] developed by the Berkeley Reconfigurable Architectures, Systems and Software (BRASS) group. Garp is a hybrid architecture of a general purpose microprocessor coupled with a reconfigurable fabric on the same die. GARP as shown in Figure 2.2 has a main processor which executes a MIPS-II instruction set extended with instructions for reconfiguring the fabric. These include instructions to move data between the array and the main processor's registers. Garp's reconfigurable array cannot read or write the main processor's registers itself, but the array does contain data registers of its own. Garp makes external storage accessible to the reconfigurable array by giving the array access to the standard memory

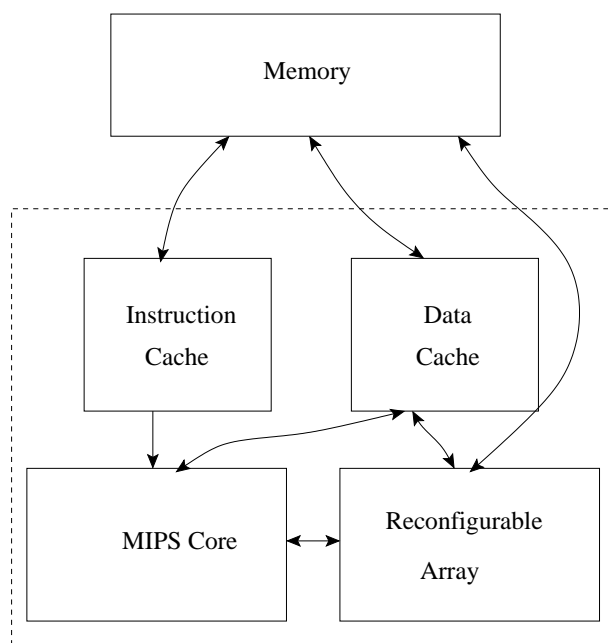


Figure 2.2: GARP: Block diagram

hierarchy of the main processor. This also provides immediate memory consistency between the array and the processor. Garp's reconfigurable hardware is a two-dimensional array of entities called blocks. One block on each row is known as a control block. The rest of the blocks in the array are logic blocks, which correspond roughly to the logic blocks of a commercial FPGA. The Garp architecture fixes the number of block columns at 24. The number of rows is implementation-specific, but can be expected to be at least 32. The architecture is defined so that the number of rows can scale in an upward-compatible fashion. The granularity of the array is two bits. Logic blocks operate on values as 2-bit units, and all wires are arranged in pairs to transmit 2-bit quantities. Operations on 32-bit values thus generally require 16 logic blocks. Multi-bit functions are naturally laid out along array rows. With 23 logic blocks per row, there is space on each row for an operation of 32 bits. Few logic blocks that are free are used for overflow checking, rounding, control functions or wider data sizes. The array logic operate on an array clock, whose frequency is fixed by

the implementation. No relationship between the array clock and the main processor clock is required. Reconfiguration of the array is done only through the processor instructions. Thus it provides a fine grained reconfigurable array fabric which is tightly coupled with a general purpose processor. Simulation results showed that GARP with the array running at 133MHz was found to achieve two to nine times improved performance over a Sun Ultra SPARC with the processor running at 167MHz [6].

GARP is a control-flow architecture which is the conventional control-flow computer with additional features for configurability of customized functional units. Another extreme approach is the wormhole run-time reconfiguration [7]. This is based on the data-flow computer architecture. In wormhole run-time reconfiguration, custom computational pathways are created and modified rapidly using a distributed control scheme (data-driven partial run-time reconfiguration).

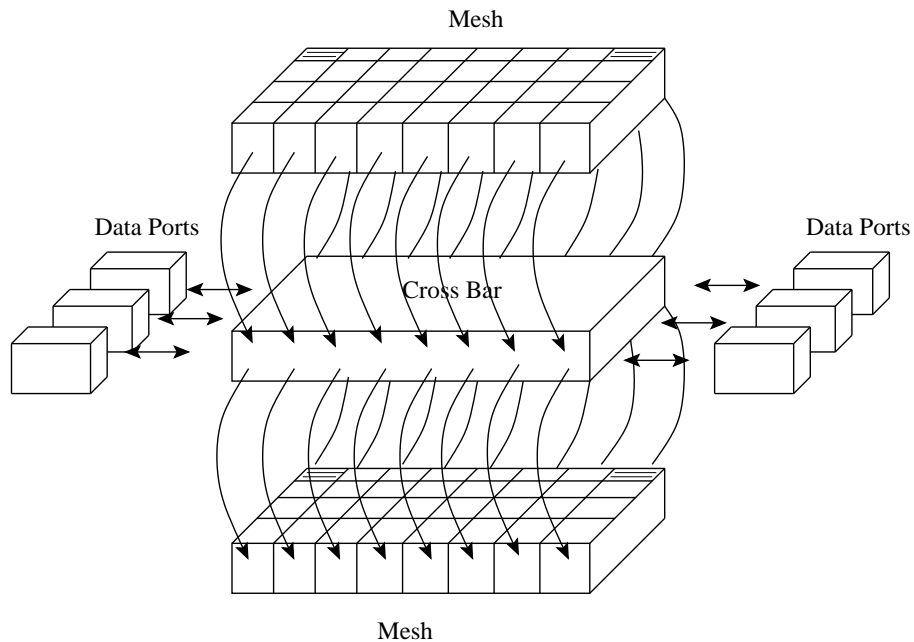


Figure 2.3: Stallion: System architecture

Stallion [8] is a processor based on the wormhole RTR. Its system architecture is shown in Figure 2.3. It consists of data ports through which data is fed into the chip. The cross bar gives full connectivity to the components in the mesh and data ports. Mesh contains an array of Interconnected Functional Units (IFUs) which is the basic programmable processing element of the Stallion architecture. Computational streams are used in the process of reconfiguration of the chip. Wormhole RTR concept is formed from independent streams of configuration data and operand data that move and interact within the architecture to perform the computation.

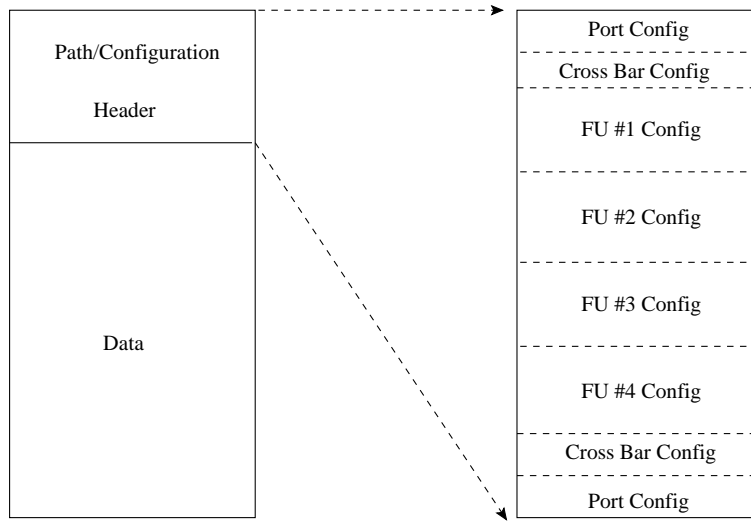


Figure 2.4: An illustration of the stream format for Stallion

Figure 2.4 gives an illustration of a stream for the Stallion. A stream is composed of a header segment and a data segment. The programming information is in the header and the operands follow the header. Programming information in the header configures various programmable resources inside Stallion and creates a computational path that will be followed by operands in the stream. This configured computational path determines what processing will be performed on the operands. As the programming header traverses inside the chip, each unit gets configured. The unit then passes the rest of the stream to other blocks inside

the chip according to its own configuration. Thus, as the data path configuration progresses, the stream gets stripped off its programming header. The header no longer exists after the entire data path configuration is complete. Thus, the order and length of programming information in the header is not fixed. The stream, its programming header and the operand data can be of arbitrary length. The use of such an architecture in the processing layer for implementation of software radios, was found to meet the current 3G data rates [9].

2.3 Virtual Hardware

Virtual hardware is similar in concept to virtual memory. Virtual hardware is the ability to swap configurations or contexts into and out of a programmable device during run-time to expand the physical hardware size of the reconfigurable functional unit seen by the operating system or the application. The concept of virtual hardware was introduced by Ling et al. [10]. A computational system called WASMII was emulated which demonstrated virtual hardware using a multi-context FPGA. They introduced the term *hardware page* for the stored context and *preloading* for the process of loading a context before it is actually used for computation. Xilinx developed XC6200 [11] a device in which the configuration registers are memory mapped. This provided many features like partial reconfigurability and the ability to configure bus-mapped registers on the array. This device was used to develop a number of RTR systems using the virtual hardware model.

Brebner introduced the Swappable Logic Unit (SLU) [12] in virtual hardware that is analogous to pages or segments in virtual memory systems. Then proposed ways to use it in a virtual hardware environment [13]. SLU is a Xilinx XC6200 FPGA-based logic circuit capable of performing a function in terms of specified inputs and supplying results on specified outputs. For operating systems to handle the SLUs efficiently it has to have a fixed area in its FPGA

implementation and fixed input and output interfaces as part of its design. This is the hardware view of the SLUs. Software on the other hand would view SLUs as functions, sub-routines in a traditional high level design environment like C or Fortran. In object-oriented programming a related set of SLUs can correspond to a set of operations provided by an object class. In parallel programming environments these can be mapped as parallel constructs. Two virtual hardware models were suggested [12]; *sea of accelerators* model, a collection of independent SLUs and *parallel harness* model, a collection of interconnected SLUs. The first model is intended for use with SLUs that have bus-addressable register interfaces and the second model for SLUs that have signal interfaces on their perimeter. The operating system supplies the routing between SLUs. The work suggests the use of a uniform three level interface between the SLUs and the environment. One on the physical level, one on the bit level and last on the functional level. The approach proposes to make the capabilities of SLUs available as library functions that can be called from software programs.

There have been a number of partially programmable devices in the FPGA market like the Xilinx Virtex [1], that has enabled virtual hardware to be implemented more efficiently. JBits [14] is a Java-based interface to FPGA hardware, that uses the partial reconfigurability of devices. This method has reduced the time required to synthesize the design tremendously such that it can be done during run-time. Run-Time Parameterizable cores [15] were introduced which made it possible to parameterize cores during run-time. This gave another dimension to the hardware page to be used with run-time compilation of designs.

2.4 Multi-context

The most effective way of implementing virtual hardware is by having multiple contexts of configurations on the device out of which a particular context is active. The process of

making a particular configuration context active is called *switching*. This has to be done with a small latency in the nanosecond range to effectively implement virtual hardware. In a computational system with a backup storage for other configurations to be swapped with the ones on chip during run-time, context switch can be considered to be a reconfiguration with a different [smaller] latency.

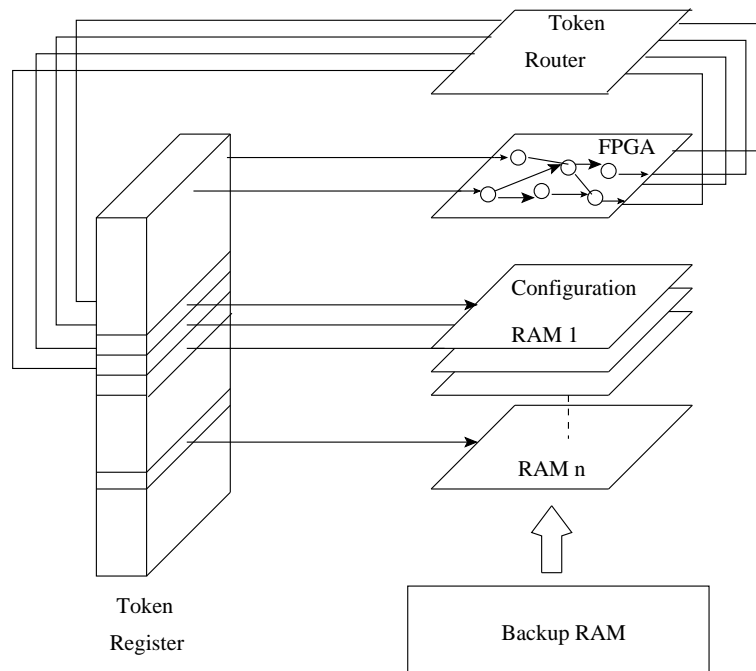


Figure 2.5: WASMII: Chip architecture

WASMII [10] was the first system proposed that was conceptualized on a multi-context device. It is a data driven computational system which uses the concept of Virtual hardware to visualize an infinite hardware for the applications. It uses a multi-context configurable device for effective virtual hardware implementation. WASMII does effective data dependent computation. A data-flow graph is divided such that each part fits into the FPGA. Since the computation of each node in the data-flow graph is driven by tokens, a context or a page can be activated when all the tokens for that page are ready. A page can be replaced when

all the tokens in the page are flushed out. The work proposed a chip structure for data-flow machine based on the virtual hardware. This computation mechanism was called WASMII and the chip structure proposed was called WASMII chip.

The structure of WASMII chip is shown in Figure 2.5. The token router and the input token register are attached to the virtual hardware consisting of multi-context programmable logic. Token router is a packet switching system for transferring tokens between pages. It receives tokens from the activated page and sends them to the input token registers. Input token registers store tokens outside pages. A set of registers is required to every page of the WASMII chip. Outside the WASMII chip a scheduler carries configuration data from the backup RAM to internal pages according to the order decided by a static scheduling algorithm. When all the required input tokens arrive at the input token register, the corresponding page is activated. After all tokens are flushed out of the current activated page, one of ready pages is activated by the order assigned in advance. In each activated page, all nodes and wires are realized by the FPGA resources. Each node starts its computation in a data driven manner. Tokens sent out of the activated page are sent through the token router to the input token register that enable other pages to be ready to be activated. WASMII was later implemented [16] on an experimental multi-context device called Dynamically Reconfigurable Logic Engine (DRLE) developed by NEC.

Parallel to the WASMII work, the concept of a Dynamically Configurable FPGA (DPGA) [17], was proposed by Bolotski, et. al. which also talked about context swapping within an FPGA. The DPGA concept is to program several different sets of configuration RAMs for each logic function. The appropriate set could be selected at run-time using global signal wires. This would mean that the logic values for the function would be taken from a small RAM, and the global context control wires would act as the address used for that RAM. In a later paper, DeHon proposed placing DPGAs on the same die as a normal processor to act as a reconfigurable accelerator [18]. Xilinx filed a patent on the multi-context programmable de-

vice in 1995 [19] [20] and presented the work as a Time-multiplexed FPGA [21]. The device has an architecture similar to the XC4000E [1] and has multiple configuration planes. The reconfigurable communication processor [22] developed by Chameleon systems, Inc. is the first known commercial effort on multi-context programmable device. It has a reconfigurable fabric with two configurable planes; one for executing while the other configures the next part of the algorithm or application. Most of these contributions do not give a study of context switching from a system perspective. This research dealt with the application level and system level issues of a multi-context programmable system. This thesis will present the study of context switching in various modes of operation from a complete control oriented switching to a data oriented switching between the available multiple contexts, and the applications which demonstrate the concepts.

Chapter 3

Hardware Platform

To demonstrate context switching, a system consisting of reconfigurable devices and other support hardware along with an effective environment to control and communicate with these resources, was used. The prototype board was developed by Sanders (now BAE-Systems). It houses devices with multiple contexts, the Context Switching Reconfigurable Computer (CSRC) developed by Sanders (now BAE-Systems). This chapter gives an overview of the hardware platform of the system. It also describes in considerable detail the architecture of the experimental device, CSRC, its features and limitations.

3.1 Reconfigurable Computing Module (RCM)

The RCM is a PCI card that houses the CSRC devices and other necessary hardware used to demonstrate the operation of the context switching reconfigurable computing. The block diagram of the board is shown in Figure 3.1. It consists of two CSRC devices, a Power PC 750 microprocessor, a Xilinx XC4085 acting as the support FPGA and PCI bridge. Synchronous SRAMS provides 1 Megabyte of cache for the Power PC processor. The processor

is primarily through these two sets of FIFOs that are $8K - 64K$ deep. The status flags of the FIFOs are available as inputs to the FPGA that makes it available to the processor and the CSRCs. The Xilinx FPGA is intended to provide a variety of support functions. The PowerPC communicates with the FPGA through its data bus. The design on the FPGA contains as a minimum, the ability to receive interrupt requests from the host processor, manage FIFO control flags, program the CSRC devices, and serve as a DMA controller to move data to and from the CSRC devices.

3.2 CSRC Architecture

The context switching reconfigurable resources present on the RCM board is the Context Switching Reconfigurable Computer [24] designed by Sanders (now BAE-Systems). FPGAs approach higher speedups when implementing algorithms with deep pipelines. However, generating pipeline control signals, implementing state machines and interfacing with external RAM or other integrated circuits require bit-wise programmability. The CSRC device architecture has a 4-bit DSP data-flow engine that is simultaneously capable of efficiently implementing glue logic; thus, it can use both the advantages.

Figure 3.2 taken from [24] shows the architecture of CSRC. CSRC consists of 16-bit wide data pipes. Each pipe consists of context switching logic arrays (CSLAs). A single CSLA consists of context switching logic cells (CSLC) and is capable of processing two 16-bit words and producing a 16-bit result. The result of one CSLA is available as input to two adjacent CSLAs in the pipe. Thus, a pipe can be used as a data path, where data can flow in both directions. This is particularly useful when a part of the algorithm is implemented as a pipe in one context where data flows from left to right. The next context takes the result from the previous context and processes it from right to left to implement the next part of the

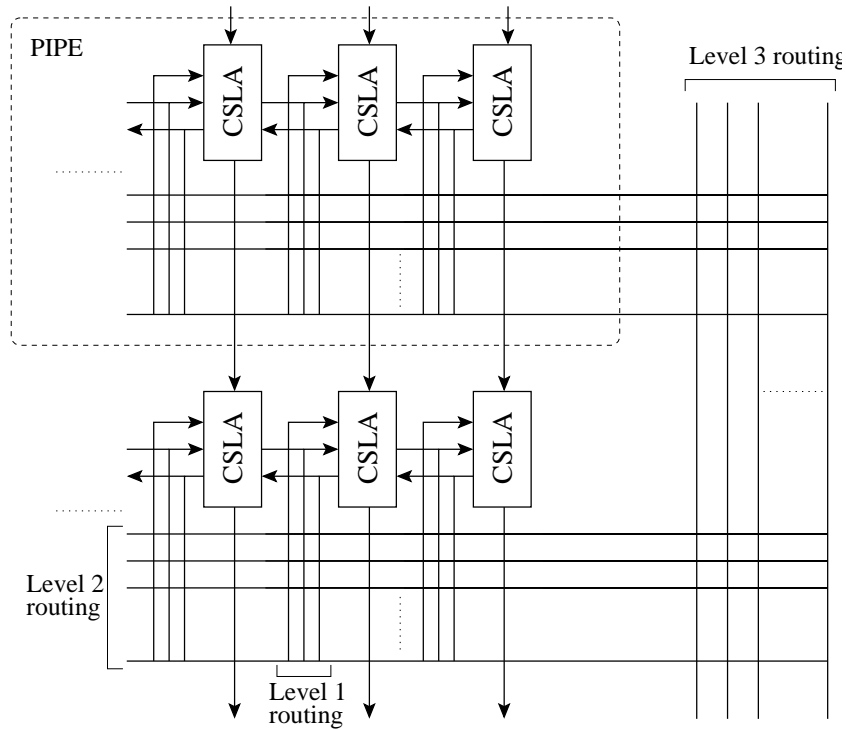


Figure 3.2: CSRC Architecture

algorithm. The advantage of such a data flow mechanism is that it eliminates the need to reroute data from its physical origin in one context to its physical input in the subsequent context.

The CSLC shown in Figure 3.3 is the heart of computation for the CSRC device. It is composed of a four input lookup table (CSLUT), a context switching flip-flop (CSFF) a tri-state buffer and the carry logic. The carry logic is such that the carry chain can be connected, disconnected or fed a logic Zero or One every four bits. This enables a pipeline granularity of 4-bits. Each configurable resource in the CSLC along with each routing resource has four configuration bits among which a single bit is selected as the current configuration. Thus achieving four configuration planes. The CSRam implements the global sharing scheme used for sharing data between contexts. Another way of sharing data between contexts is by the

private/public addressable registers. Each CSLC has a private register for each context and a public register. During a context switch the CSLC value is stored in public register if it is to be shared or kept in a private register if not. When a context is activated it can select between its previous value in the private register and the shared public register value.

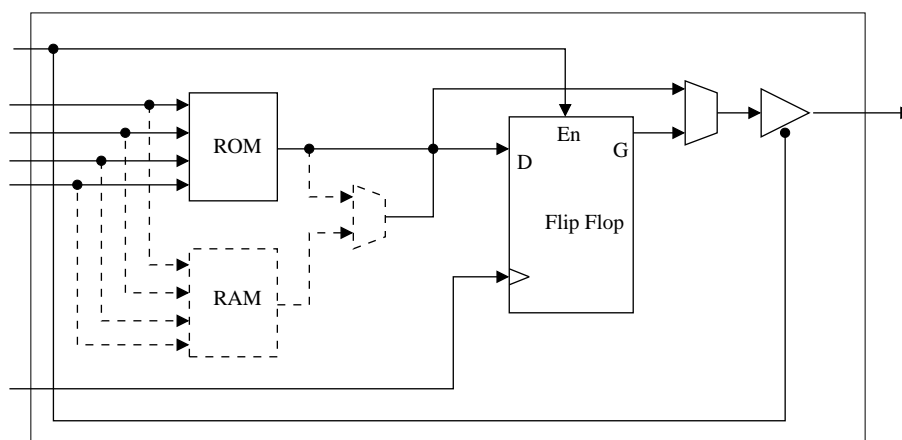


Figure 3.3: Context Switching Logic Cell

The CSRC has three levels of routing. Level 1 routing routes the various CSLCs in a CSLA. Level 2 routing is alongside the pipes and consists of 16-bit buses. They run along the width of the CSRC device and any CSLA in the pipe can tap on to these buses. Each CSLA has two 16-bit inputs, each of which is capable of tapping into any of the Level-2 busses. Level-2 routing can be utilized as a bus architecture, can be broken down and utilized by individual bits, or can be employed as any combination of these. Such pipes are stacked one over the other with Level-3 routing connecting them together. Corresponding CSLAs on adjacent pipes have dedicated wiring that allows them to pass along their carry bit. This feature allows adjacent pipes to be bundled together to form a single 32-bit wide data path. Individual pipes can also be broken down into narrower pipes up to a nibble. The CSRC has two routing modes. Bus routing where the routing is 16-bit wide along the pipe to utilize the pipelined architecture of the CSRC in data flow type of applications. Bit-wise

routing used for routing glue-logic and state machines that are inherently not data flow in nature. The bitstreams for the CSRC FPGAs are downloaded serially. The user is required to specify which context is being loaded and then supply a clock and data. A context can be programmed when another context is active. The present context can also program another context. The prototype CSRC consists of 8 pipes stacked one above the other with 8 CSLAs each. Each CSLA has 16 CSLCs. Thus the prototype consists of 1K CSLCs.

3.3 Support FPGA design

There is a Xilinx XC4085 on the RCM board that is intended to act as the control interface to the CSRC devices. The processor controls all the operations on the CSRC through an hardware interface embedded in the FPGA. Programming the CSRC, context switching on the CSRC, FIFO control are some of the tasks that the controller on the FPGA is designed to do. Typically the FPGA does only the control. There is a 16-bit interface between the Xilinx FPGA and each of the CSRCs. These control lines can be used to implement control logic for the applications. This logic can be hardcoded for a specific task or have some flexibility by being controllable from the host application.

One flexible approach is a host programmable Finite State Machine (FSM). It is a table based design, with a host addressable memory which stores the state table. The outputs of this FSM can be routed to any of the pins interfacing with the CSRCs and the inputs can be drawn from any of the interface pins by control words from the host. The FSM is made programmable from the host by putting appropriate data into the memory holding the FSM table. Apart from being flexible and host programmable this approach also has the advantage of moving logic to high speed hardware. The prototype CSRC parts have limited logic and routing resources which can require some applications to depend on this external

control logic. A description of how the FSM is programmed is given in Section 5.1

All these are the main components which form the hardware platform on which the context switching applications were developed. These resources were integrated on a PC. An Application Programming Interface (API) (described in Chapter 4) provides the interface for these resources from the host PC.

Chapter 4

Run-Time Environment

To access the hardware resources on the RCM platform, the system requires extensive run-time support. The system has a run-time environment that is specific to its hardware [25]. The PowerPC on the board allows flexible programs to be run close to the context switching hardware. This chapter describes in considerable detail the system environment for run-time support. The run-time environment consists of a layered stack of software to access the hardware resources in an abstract manner. This allows the user to develop applications using the context switching features of the system. This run-time stack is shown in Figure 4.1.

The two available ways of communicating with the board are a low bandwidth serial line, and memory mapped reads and writes over the PCI interface. The serial line is used mainly for debugging. Serial line transfers the CSRC and FPGA interface lines, the FIFO status flags and a few other application control signals to the host PC. This is displayed on the host with each clock. This display is used as the debug environment for applications being developed on the RCM platform. The PCI interface is used for high speed communication between the RCM platform and the host. Application interfaces were built on top of the memory-based communication.

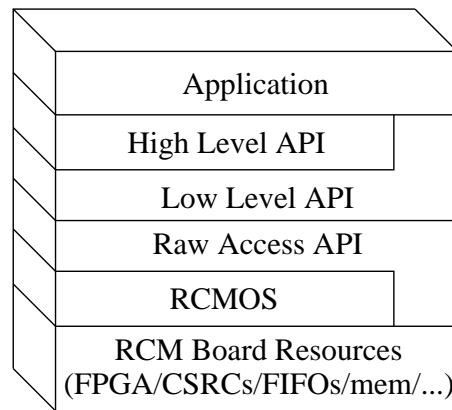


Figure 4.1: Run-time environment stack

4.1 Host API

Applications access the hardware resources through either a high level Application Programming Interface (API), low level API, or a combination of both. The low level API is implemented using the ‘Raw Access API’ which provides only basic services like:

- open/close board, and
- read/write memory mapped areas

The raw API makes no assumptions of the hardware configuration and programmable resources on the board. It does not access special features of the software running on PowerPC or any features of configurations loaded in the CSRCs or the FPGA. The initial setup of software on the PowerPC is done with this API (accessed through a higher level API) and board firmware. The memory controller on the board is configured to map memory ranges to various devices. This is used by the API to provide some direct access to the FIFOs and FPGA. Complex communication and control is implemented with a memory-based handshaking protocol with software on the Power PC. The ‘Low level API’ uses this protocol to

provide transparent access to the hardware from the host application. The features of this API are listed below:

- PowerPC configuration,
- FPGA configuration,
- CSRC configuration (basic and caching),
- CSRC context switching, and
- data streaming.

This low-level API is implemented in two forms. One is the basic C API. The other is through the ACS API [26]. The C API is suitable for high speed direct access. The ACS API is an interface for distributed adaptive computing systems. It allows the board to be accessed in a distributed dynamic network of heterogeneous reconfigurable hardware. In Figure 4.1 both these APIs are represented by the *Low Level API* layer. The Xilinx FPGA is used to control the signals on the board. The CSRC clock, reset, context switching control, programming control and FIFO status and control are the controls that the FPGA manages. Many CSRC connections also go through this FPGA. The low level API does not have information about how these control is implemented. It simply maps the FPGA with address, data and control lines. Specific control is achieved either by the application programmer or another layer of API.

The high level API provides an object oriented view of the board representative of its physical parts. This layer is based on specific features of the FPGA configuration as well as the low level API features implemented in the RCMOS. This API includes the functionality needed to take full advantage of the hardware. This provides a high level view of the system such that applications using it are isolated from many of the details of register access and

bit manipulation. Thus complex functionality involving many low level API calls can be wrapped up into an easier to use interface.

4.2 RCMOS

The PowerPC on the RCM board is used to implement a lot of functionality in the system. All this is achieved through a mini-operating system called the RCM Operating System (RCMOS). It is used to implement many of the low level API functions in an efficient manner. Programming the configurable hardware resources would involve bitwise data writes which can be slow over the PCI bus. An efficient way of implementing this would be to load the configurations into the board memory and letting software in the PowerPC do the work. Many of the CSRC control operations control-driven context switching and clocking are also more efficiently implemented closer to the hardware than across the PCI bus.

4.3 Configuration Caching

Effective implementation of run-time reconfiguration requires fast switching between configurations. In traditional configurable systems this configuration switching latency is large due to long configuration time and communication latency over the system bus. The CSRC device with four on chip contexts provide fast configuration switching. Configuration storage on the PowerPC memory eliminates the latency for the transfer over PCI and enhances the on chip configurations. A virtual hardware hierarchy similar to the concept of memory hierarchy was implemented to exploit temporal locality in the configuration sequence.

This configuration caching hierarchy is shown in Figure 4.2. The on chip device configurations is considered as the high speed and low capacity level that is on top of the hierarchy. At

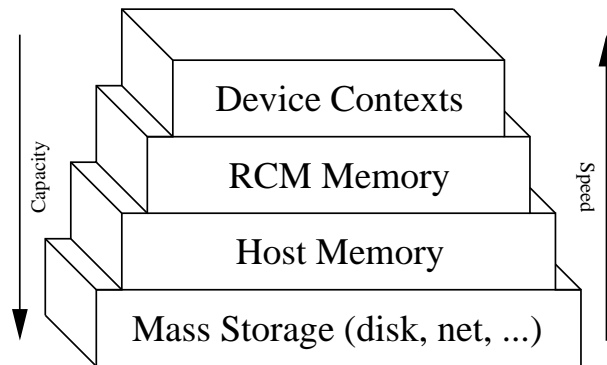


Figure 4.2: Configuration caching hierarchy

the lowest level is the configurations stored in mass storage. This storage can be persistent physical media such as a host hard disk or remote storage accessed over a network. Each level has configuration time and storage capacity lower than the layer immediately below it. Access speed increases and capacity decreases as the configurations move from mass storage towards the device contexts. For an application if a particular level cannot hold the required number of contexts they are stored in the next higher capacity level and loaded on demand. The host uses an API that stores configurations on board via the RCMOS for board level configuration caching. The RCMOS keeps track of configurations that are loaded into individual contexts on the device. The application gives a high level request for a particular configuration to be loaded. If the requested configuration is currently loaded in a context, then switching is a high speed hardware operation. If the requested configuration is not currently on a device context, then RCMOS uses standard replacement algorithms to determine the context to replace with the requested configuration.

Chapter 5

Application Control

The applications developed require system level and application level control. This chapter describes the approaches for achieving this control in the context switching system. Section 5.1 describes implementation of application control in the system while Section 5.2 describes system level control for enabling context switching.

5.1 Programmable Hardware Control

The prototype CSRC parts have limited logic and routing resources which require applications to depend on external control logic. This is achieved on the RCM board by logic programmed on the Xilinx XC4085. This logic can be embedded in the FPGA bitstream for a specific task. This approach leads to a specific programmable bitstream for each application. The control logic can also be implemented in a more generic and flexible way by making it controllable from the host application. One such approach is a host programmable FSM (introduced in Section 3.3). This is implemented as a state table based design. There is a table entry for each distinct combination of the present state variable and the input values. The

table entry is the next state variable and the output values for that particular combination of state and input. The state table is implemented as a memory with the present state and input combination providing the address for its table entry. The data from the table entry selected by this address provides the output and next state. This state and the new inputs is the address for the next table entry to be selected at the next clock. The output bits can be mapped to any line interfacing with the CSRC. An abstract description of the FSM is converted into this memory based table format by the high level API. This is then loaded into the FPGA with a protocol of low level register writes. Thus, host programmable FSM allows flexible application control with high speed hardware closer to the CSRC devices.

5.2 Context Switching Control

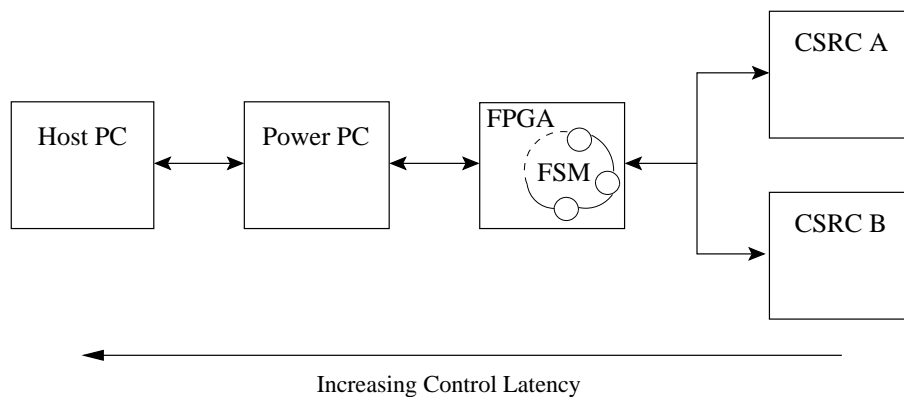


Figure 5.1: Context switching control routes

Context switching is controlled from a number of points in the system. Figure 5.1 shows the path on the RCM board through which the switching control signals pass. The control signals can be initiated at any of these points. The actual route is application specific. Applications requiring user input need the full control path from host to CSRC. Some application can improve performance by moving this control closer to the CSRCs. The latency involved in a

context switch is directly related to the number of layers that signals have to pass through. Latency for control from the host is large. For debugging purposes this communication is used to stall the CSRC logic. Using such a control mechanism the following context switching mechanisms for run-time reconfiguration were demonstrated.

5.2.1 Host-driven RTR

Some applications require user intervention for their context control. In such applications context switching is controlled by the host machine. It will have a high latency between the context switch request and the switch on the device. This is because the context control has to pass all the way from the host to the devices as shown in Figure 5.1. To demonstrate such a context control a continuous video stream was processed by filters stored in the various contexts on the device. A user request from the host causes the context to switch. This enables single-cycle algorithm changes. If more filters are needed than the number of available contexts on the device, the configuration cache is used to store them before configuring. Configuration of the inactive contexts takes place while another filter is running in the active context. Simple filters implemented include basic pass-through, the motion detection difference filter and a delay filter. The screen capture images of the video obtained after processing through the filters in shown in Appendix A.

5.2.2 FSM-driven RTR

In applications that require some external logic to achieve context switching, the host programmable FSM is programmed to control the contexts on the CSRCs. Motion detection algorithm was implemented to demonstrate this FSM-driven control. The application is discussed in detail in Chapter 6. Applications in which the sequence of contexts to be made

active requires such a control which is costly to put on each of the contexts. Such applications require external logic to control it. The FSM on the xilinx FPGA will provide this control closer to the hardware. This has lower latency for switching than controlling it from the host. Being host programmable this provides easier usability to the user.

5.2.3 Data-driven RTR

Context control achieved by the data being processed without any control from support hardware or software is called data driven context switching. This is the lowest latency switching available in the system. But all applications are not inherently data driven. Sometime it also depends on the way the application is partitioned to exploit data driven context switching. The CSRC devices have the capability to achieve data driven context switching. But the design tools are not mature enough to exploit this feature of the device. Hence for demonstration purposes the Xilinx support FPGA was used to implement this switching. It reads in context switch request from the CSRC devices and switches the particular context of the part. A network encryption application where the data packet to be processed contains the context number to be made active is implemented to demonstrate data-driven control. Chapter 7 discusses this application in detail.

Chapter 6

FSM-Driven RTR

6.1 Motion detection Algorithm

The motion detection algorithm [27] is used to detect movements through video and then selects only parts of the image frame where there is motion. Micro-sensor platforms capable of supporting reconnaissance, surveillance and target acquisition operations typically consist of one or more sensors, signal conditioning and processing subsystems, a radio link and a power source. In such remote sensing systems significant energy is spent for transmission. In such applications, transmitting only the selected part of the frame where there is movements instead of the full frame in the video stream conserves energy. Any energy saved directly increases the life of the sensors. This is significant as these micro-sensors are typically one-time deployable devices. Typically this data is transmitted and is analyzed in the base station. Transmitting the whole frame every time by all the micro sensors would flood the base station with unnecessary data to be analyzed. Transmitting only the clipped frame with movements will solve the problem. This chapter discusses the implementation of the motion detection algorithm on the RCM platform with the CSRC devices. The motion

detection algorithm implementation demonstrates the control of CSRC contexts through the host programmable finite state machine.

6.2 Algorithm

The algorithm consists of capturing an image, processing it and sending out a cropped part of the image where there is motion. In a typical implementation of this algorithm on a micro-sensor platform [28], the processing part is done on an FPGA coupled to the Digital Signal Processor (DSP) where all the other tasks are accomplished. Similar approach was used with the CSRC devices replacing the FPGA to do the image processing. The flow diagram of the algorithm is shown in Figure 6.1. The image is processed in the following manner.

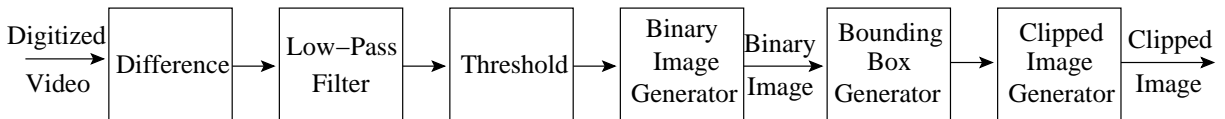


Figure 6.1: Flow diagram of the motion detection algorithm

The absolute difference of the collection of two images is computed. The output of this *Difference* block is another image that has non-zero pixel values where changes have occurred from one image to the next. Considering that the algorithm requires that the camera be stationary while operating, these non-zero pixels corresponds to moving objects in the camera's field of view. However, these non-zero pixels could also be caused by erroneous pixel values or sudden changes in light intensity from one image to the next. Later blocks in the algorithm attempt to compensate for the non-zero values caused by image's variations that are not due to movement. The *low-pass filter* minimizes abrupt discontinuities in the absolute difference image. These discontinuities can be caused by either erroneous pixels due to spot noise or by

small objects moving in the image such as leaves blowing in the wind. Therefore, it is hoped that the smoothing of the filter will cause these undesirable abrupt discontinuities to reside below the *thresholds* and thus not be included in the regions of interest (ROI) inspected by the latter portion of the algorithm. If the thresholds are too high, desirable ROIs for further analysis will not be identified. On the other hand, thresholds that are too low will produce false detects for ROIs. For example, a threshold that is too low will cause an entire frame to be incorrectly identified as a ROI when an abrupt brightness change occurs from one image to the next. This ROI is represented as a binary image by the *binary image generator* block.

Once the binary image has been produced, contiguous regions contained within it can be determined and a bounding box large enough to incorporate the minimum and maximum rows and columns of the regions is determined. The image frame is clipped along this bounding box. This clipped frame contains the motion in the field of view of the camera.

6.3 Implementation

The algorithm mapped onto the system is shown in Figure 6.2. The image is captured by the Host machine and passed to the RCM board through the FIFOs. The four parts of the algorithm are implemented as four different contexts inside the CSRC. The FSM controls the switching of active contexts. The binary image generated by the CSRC can be used to generate the cropped image by the host machine or the Power PC processor on the board.

Video Capturing

Video is captured by a CCD camera that gives an analog video signal. A Bt848 video capturing card is used for generating digital video stream from the analog video signal. The

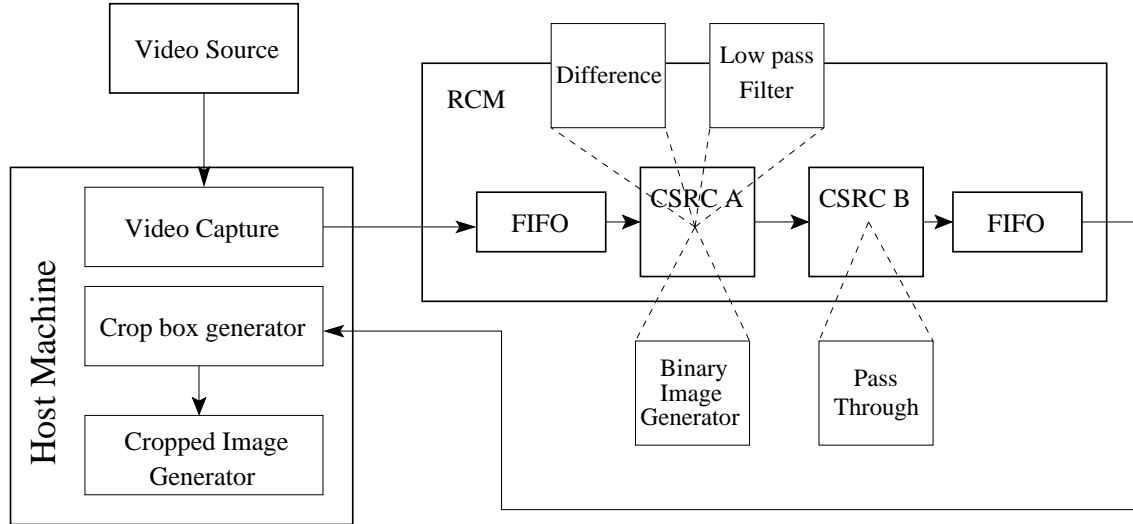


Figure 6.2: Motion detection algorithm mapped on the RCM platform

video stream consists of 160×120 8-bit gray scale image sequences. The image data is stored in the sharable memory between the host PC and the PowerPC of the RCM board. The image is transferred into the CSRC through the FIFO with two pixels per word.

Difference

The difference block enhances the portions of the frame that have changed due to moving objects. It generates a difference image from two sequential images. Denote each image frame of the video stream as I_i , where i is the sequence index of the video. The previous image, I_{i-1} , is stored in the CSRC A memory. The current image, I_i , is streamed through the input FIFOs. The difference image, $|I_i - I_{i-1}|$, is stored back to the CSRC A memory. The CSRC memory shares the data between the contexts.

Low Pass Filter

The low-pass filter eliminates the spot noise and smoothes out the image. The algorithm suggests a low pass filter with a kernel size of 37×37 [28]. Due to limited computational resources, the low-pass filter was implemented as a 4×4 averaging filter that reads the differenced image from the CSRC memory. Matlab simulations showed that this reduction in complexity did not affect the performance much. The differenced image is read from the CSRC memory and the filtered image is stored in the CSRC memory.

Binary Image Generator

The binary image generator block compares the filtered image with a threshold and generates a binary image with a '1' for pixels above the threshold and a '0' for those below the threshold. Ideally a dynamic threshold value should have been generated. Due to the lack of resources on the prototype CSRC, a static hard-wired threshold is used instead of calculating the threshold. With enough resources, this block can be implemented as a separate context. It uses the filtered image stored in the CSRC memory, generates the binary image and, streams it out through the output FIFO. This binary image has only the pixels showing genuine movements without the spot noise and disturbances due to intensity variations.

6.4 Application Support

This application would require external control for context switching and other application control. This is provided by the host programmable finite state machine. This enables the control of the hardware from resources that are nearer than the host or the processor. This ensures smaller latency for the control signals. Context switching control is achieved by hand

shaking between the FSM and the CSRC contexts. The state diagram of the programmed state machine for this purpose is shown in Figure 6.3.

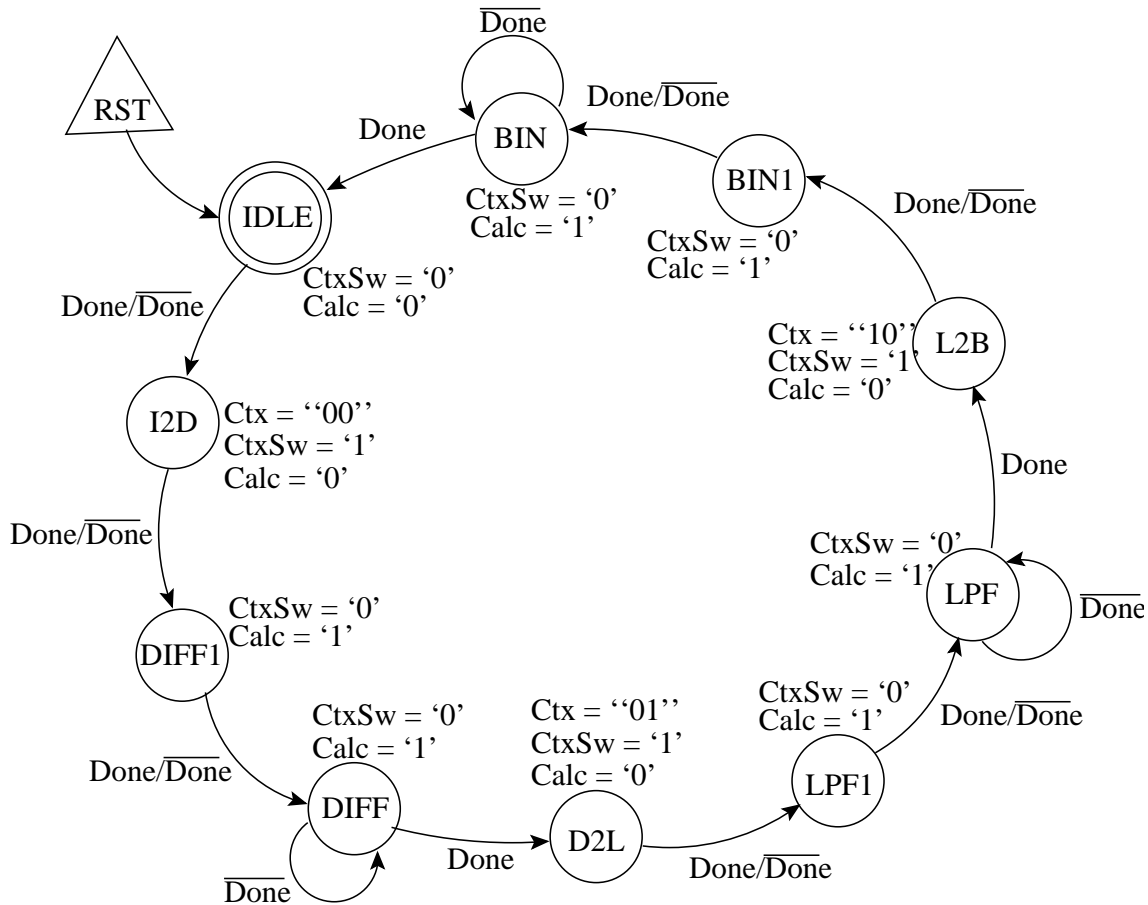


Figure 6.3: State diagram programmed to control the motion detection algorithm

Signal 'Done' is the input to the state machine. State 'IDLE' is the idle state. 'I2D' state is the intermediate state where the CSRC A context is switched to the difference context as indicated in the outputs 'Ctx' and 'CtxSw'. The next state 'DIFF1' starts the calculation on the difference context with the signal 'Calc'. The state machine stays in the 'DIFF' state until the calculation is done which is indicated by the input from the CSRC 'Done'. This sequence of control continues for the other two context with the states 'LPF' for the

Low-pass filter context and the state ‘BIN’ for the Binary image generator context. Hence the programmed FSM acts as both an application controller and a context controller.

Screen capture of the binary image obtained after processing the camera captured video on the CSRC is given in Appendix A.

Chapter 7

Data-Driven RTR

7.1 Enigma Encryption

The implementation of enigma encryption algorithm is discussed in this chapter. This application is a demonstration of the CSRC for data-driven network processing. The application is basically a demonstration of simple encryption and decryption of network traffic for multiple channels. As the resources on the prototype CSRC are limited, a simple Enigma-like encryption scheme was selected.

The Enigma rotor machine is considered to be the best known historical encryption machine. It was used to encrypt most of the radio signals of the German armed forces and intelligence service during the second world war, and to decrypt the messages after receiving them. The principle of the Enigma was based on independent inventions by Edward Hugh Hebern, Arthur Scherbius and Hugo Alexander Koch [29]. The enigma was available commercially in the 1920s. Then the German government acquired the rights and further developed it in secret through the armed forces and diplomatic service.

7.2 Algorithm

The Enigma Machine [29] was based on a system of three rotors that substituted cipher text letters for plain text letters. The rotors spin in conjunction with each other, performing varying substitutions. Figure 7.1 taken from [30] gives a graphical explanation of what happens when a key is pressed on the enigma machine. For the sake of simplicity, it uses only an eight letter alphabet, whereas the real machine uses all 26 letters.

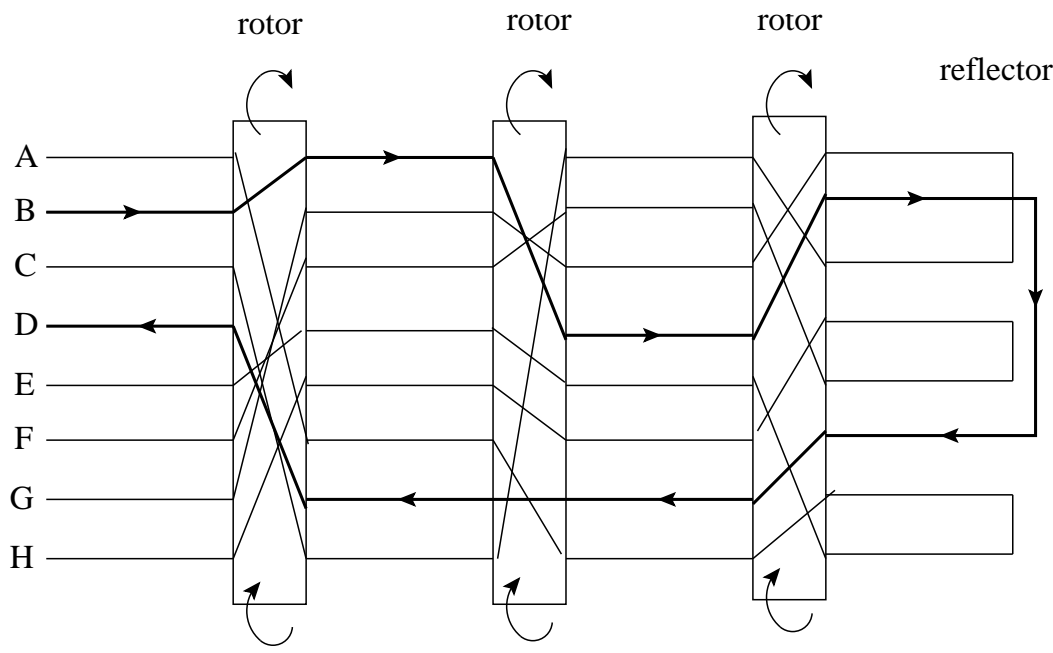


Figure 7.1: Simplified description of enigma machine

A letter typed on the keyboard of the machine is sent through the first rotor, which shifts the letter according to its present setting. The new letter passes through the second and third rotor, where it would be replaced by a substitution according to present settings of the second and third rotor. This new letter is bounced off of a reflector, and back through the three rotors in reverse order. As the plain text letter passes through the first rotor, the

first rotor would rotate one position. The other two rotors would remain stationary until the first rotor rotates 26 times (one full rotation). Then the second rotor would rotate one position. After the second rotor rotates 26 times, the third rotor would rotate one position. This principle of the shifting rotors allowed for $26 \times 26 \times 26 = 17576$ possible positions of the rotors. To decode the message, the rotors are set to the initial settings, and then the cipher text is put through the machine. This gives the plain text back.

This concept was used to build an encryptor for bytes. Each rotor has $2^8 = 256$ slots. The key and the shifting effects of the rotor are realized by adding the key and offset to the byte and obtaining its modulus for 256. The encryptor implementation is pipelined, so returning data through the rotors is implemented by repeating the rotors in the reverse order. The shifting of the repeated rotors is timed accordingly to match the shifting of the original rotor. The obtained byte is then scrambled nibble-wise using two 4×4 tables. The table entries represent the actual rotor settings.

7.3 Implementation and Network Processing Model

The algorithm was used to build an encryptor for bytes. Each rotor has $2^8 = 256$ slots. The key and the shifting effects of the rotor are realized by adding the key and offset to the byte and obtaining its modulus for 256. The encryptor implementation is pipelined, so returning data through the rotors is implemented by repeating the rotors in the reverse order. The shifting of the repeated rotors is timed accordingly to match the shifting of the original rotor. The obtained byte is then scrambled nibble-wise using two 4×4 tables. The table entries represent the actual rotor settings. Using the available RCM prototype board, the following system is implemented. FIFOs are used to stream data into the processing unit. The processing unit consist of the two CSRCs and the Xilinx support FPGA. Output

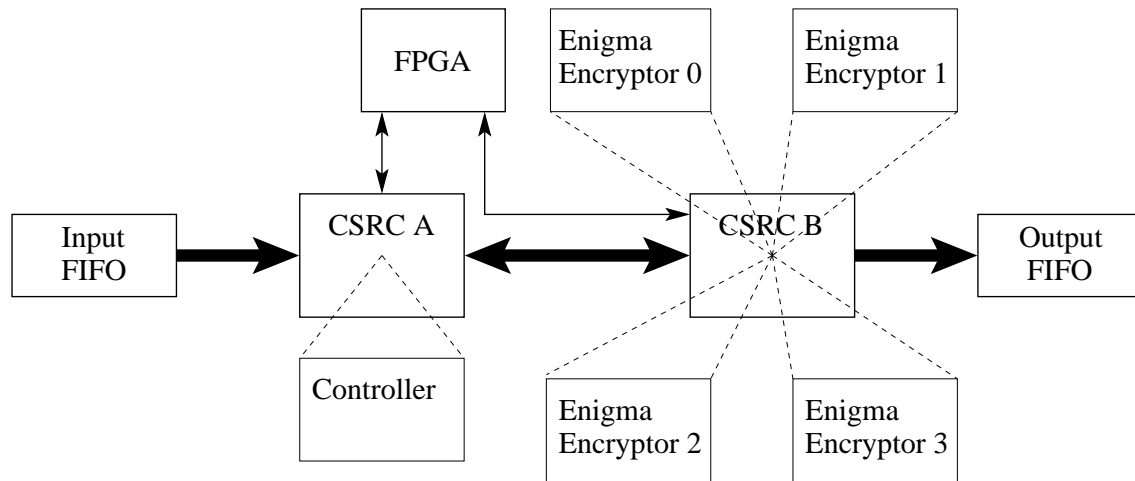


Figure 7.2: The enigma application mapped to the RCM platform

FIFOs are used to stream encrypted data out of the processing elements. The system design partition is shown in Figure 7.2. Context control signals from CSRCs go through the support FPGA back to the CSRCs.

Each of the four contexts on the CSRC B contain an enigma encryptor with a particular rotor configuration and key. The key can be made programmable using the data in the header. Each of these individual rotor configurations is used for each channel. The header as shown in Figure 7.3 has information of the target channel address and packet length. The channel address is used for selecting the particular enigma engine which maps to a particular context to be activated. The CSRC A has a controller context that reads and processes the header of the packet. The data determines the channel for which it is intended, and the number of bytes in the packet. The controller signals the support FPGA to set the particular context on the CSRC B. The packet length is stored in a down-counter register. CSRC A's controller context now acts as a pass-through for the packet data. It simultaneously down-counts the register to keep track of the number of bytes processed. With the counter reaching zero the system would have finished processing the packet. The controller context resets itself and

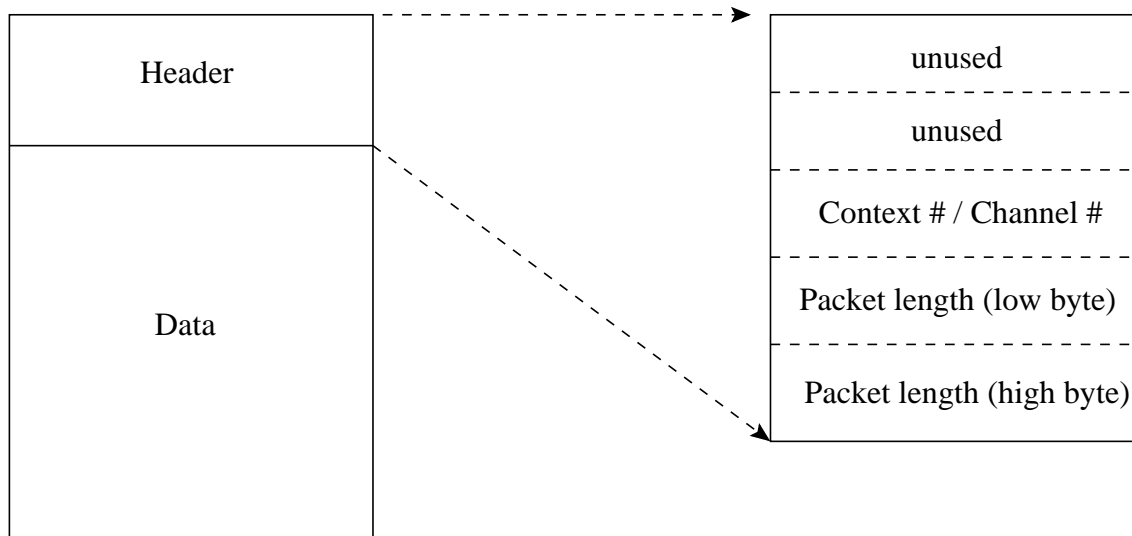


Figure 7.3: Format of the data packet used for enigma encryption

waits for the next packet to arrive.

If each of the contexts process data for a particular channel, then this application demonstrates the data-driven virtual hardware implementation for a network with four channels. If more than four channel are required more configurations can be stored anywhere in the configuration caching hierarchy and loaded on the device when required. With this the algorithm can be expanded to any number of channels. Such an implementation will be very efficient in a network that requires different types of processing for each channel.

Chapter 8

Analysis and Results

The research involved the analysis of the costs and benefits of multi-context and context switching strategies for runtime reconfigurable systems. This chapter discusses the results of the analysis based on the applications mapped to the experimental context switching platform.

8.1 Area

The strongest argument in favor of multi-context devices is the re-usability of area for emulating infinite hardware. To assess this claim, a study of area requirement for the applications was performed.

Table 8.1 shows the gate equivalent areas obtained from Synplify for the applications implemented in a single context and in the four-context device. The table also shows the area counts in each context for the application in the four-context implementation. A device with an equivalent gate count of the highest number will be required for that particular appli-

<i>Application</i>	<i>4-context implementation</i>		<i>1-context implementation</i>
Motion	Difference	783.0	2055.0
Detection	Filter	1188.8	
Algorithm	Bin. image gen.	253.3	
Enigma Encryption	Enigma0	1331.8	5245.0
	Enigma1	1331.8	
	Enigma2	1331.8	
	Enigma3	1331.8	

Table 8.1: Area requirement for each application. Second column shows the area requirement for each context of application in a multi-context device

cation. Motion detection application can be implemented in a three context device with a minimum of 1188.8 equivalent gates without incurring any delay in reconfiguration. But with only a single context, the application requires a device with 2055.0 equivalent gates. For the Enigma encryption we see that in a four-context implementation we require at least 1331.8 equivalent gates for each of the enigma engines. In a single-context implementation of all four enigma engines bound together the application requires at least a 5245.0 equivalent gate device.

It can be inferred that for applications which can be partitioned equally among all the available contexts resource requirement will scale well with the number of contexts on the device. It should be noted that in this study the silicon real estate overhead of adding more contexts is not considered quantitatively.

8.2 Average reconfiguration Time

If applications require more contexts than the available number of contexts in the device, the CSRC configuration cache, discussed in Section 4.3, is used to store additional contexts. This, along with the host memory, will effectively implement a virtual hardware environment as explained in Section 4.3. The context switch time in such a system would have a significant latency if the context is not available on the device. Assuming that each context for the application has an equal probability of being requested when a context switch is required, an expression for the average context switching time was derived:

$$t_{\text{avg}} = \begin{cases} t_s & \text{if } 1 < n \leq k \\ p_s t_s + p_c t_c & \text{if } n > k \end{cases} \quad (8.1)$$

where;

t_{avg} average switching time

t_s context switch time

t_c context configuration time

k number of device contexts

n number of application contexts

p_s probability that context is on the device, $\frac{k-1}{n-1}$

p_c probability of a reconfigure, $1 - \frac{k-1}{n-1}$

The variation of this average reconfiguration time with number of application contexts for different number of device contexts is plotted in Figure 8.1. The plot shows that for an increase in the number of device contexts(k) from one to four and from four to eight there is a tremendous improvement in the average reconfiguration time(t_{avg}). Adding a new context would involve adding new RAM cells for context storage, multiplexers and the necessary

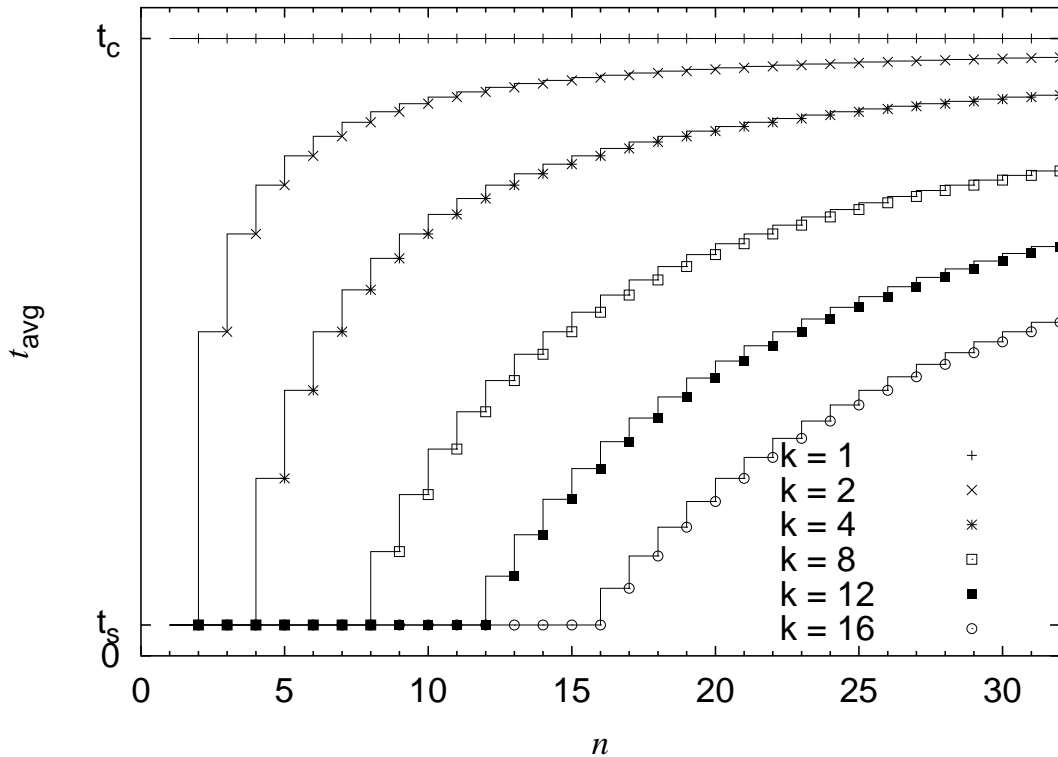


Figure 8.1: Plot of average reconfiguration time

routing for each configurable resource. Out of these RAM cells take up more significant area. VLSI area in adding new contexts increases linearly. For more device contexts we obtain diminishing returns in terms of reduced reconfiguration time. It can also be seen that a match between the number of application contexts(n) and the number of device contexts(k) results in a shorter average switching time(t_{avg}) required. For very high number of application contexts the curves come closer. This implies that when we have a lot of application contexts the advantage of having more device contexts to reduce average switching time is smaller.

It must be noted that this is a worst case situation where there is no locality-of-reference for the contexts. With such a switch request, the probability that the context requested

is on the device will be much higher; thus the average context switch time would be much smaller than that observed in the plot. This locality-of-reference depends on the applications and the way they are programmed. There is also the possibility of configuring a context in the background while another context is processing data. This would reduce the effective context program(configuration) time and hence the average reconfiguration time. The new configuration time is given by Equation 8.2

$$t_{ac} = \begin{cases} t_s & \text{if } t_c \leq t_{exc} \\ t_c - t_{exc} & \text{if } t_c > t_{exc} \end{cases} \quad (8.2)$$

where;

- t_{ac} configuration time adjusted for background configuration
- t_s context switch time
- t_c context configuration time
- t_{exc} context execution time after the next required context is determined

In applications with a known sequence of contexts to be made active t_{exc} will be the actual context execution time. For applications in which the next active context cannot be determined until the present context is executed t_{exc} will be zero and Equation 8.2 reduces to $t_{ac} = t_c$. If we consider a two context device, we can map the motion detection application on it with two of the three contexts on the device at any time. As the sequence of the context to be made active is known, we can configure the next context when the present context is executing. The time required for each of these contexts to execute is more than the configuration time; hence the average reconfiguration time is still t_s . This capability can be utilized in applications that can anticipate the next context required. With these techniques the reduced average reconfiguration time will help the total application execution time to approach the time required for purely processing the data.

8.3 Bitstream size

In this section the effect of context switching strategies on the bitstream sizes for the application is analyzed. This directly affects the storage resource utilization and configuration time.

Table 8.1 shows that the motion detection algorithm implemented in three contexts will have bitstreams for three contexts of a 1188.8 equivalent gate device and the single context will have bitstream for a single 2055.0 equivalent gate context. It shows that the multi-context approach requires more storage resources for its bitstreams than the single context approach. In the enigma encryption application storage requirement for the bitstreams are much more similar and scale well with the number of contexts. With this observation it might seem that context switching on a multi-context device leads to larger bitstreams. But if we consider bigger applications that require run-time reconfiguration, the multi-context approach is scalable better than a single context device of the same capacity. This is observed by mapping the motion detection algorithm on devices with less than three device contexts. In a two context device the inactive context can be loaded with the configuration for the next part of the algorithm to achieve seamless reconfiguration. A similar capacity single context device would hold two contexts of the algorithm in its single available context. These two approaches are indicated in Table 8.2

Table 8.2 shows the area requirement in number of equivalent gates when the motion detection algorithm is implemented in different number of application contexts. Since bitstream size is dependent directly to the area, it suggests the bitstream size required for each context. The first implementation shows a three context implementation with a highest area of 1188.8 equivalent gates. Therefore if the application were implemented in a device with 1188.8 equivalent gates per context, the total bitstream size required will be for around

<i>3-context implementation</i>		<i>2-context implementation</i>	
Difference	783	Difference & Filter	1973.1
Filter	1188.8		
Bin. image gen.	253.3	Bin. image gen.	253.3

Table 8.2: Area requirement for motion detection application for different number of application contexts

$1188.8 \times 3 = 3566.4$ equivalent gates. The second implementation requires a 1973.1 equivalent gate per context device and the total bitstream size required will be for $1973.1 \times 2 = 3946.2$ which is much higher than in the first implementation. Supposing more contexts are added to the image processing algorithm implementation, bitstream length scales at 1188.8 equivalent gate steps in a multi-context device rather than 1973.1 equivalent gate steps. This shows that having more number of smaller context is better in terms of performance if the application can afford the higher effective reconfiguration time.

8.4 Conclusion

Results from Section 8.2 indicate that with the concept of context switching the average reconfiguration time in a run-time reconfiguration system is reduced and thus, the total application execution time is more used to process data than for reconfiguration. Section 8.3 discusses how the approach can reduce the resource utilization in a scalable run-time reconfiguration system.

All the analysis presented here indicate that multi-context approach for run-time reconfiguration helps effective virtual hardware implementation. Although this approach does not

indicate increased savings in terms of silicon real estate when compared to conventional RTR approaches, it increases the application scalability of the RTR system. This can effectively help in development of more generalized RTR systems that can implement varied applications with reduced efforts for application development. This approach is advantageous when the application is divided efficiently for the available resources. Efficient tools to automatically divide the application at hand will reduce the application developers' effort further.

A multi-context device allows application implementation in an effectively smaller area on the FPGA than a device with a single context. This has several positive effects on the overall system design. The routing inside the chip is reduced as the effective area to which a design is mapped is smaller; thus, interconnect delays are smaller. It would be interesting to study the reduction in computational load for a routing tool when the application is divided into smaller contexts. Possibility of having partial reconfiguration on multi-context devices will add another dimension to the scalability of the system. The results suggest that a small context scales application better than a single large context. Section 8.1 suggests an optimum number of contexts in a device. This suggests that increasing the die size by either increasing the number of contexts or the size of each context will provide us diminishing returns in terms of performance and efficiency. A chip with partial reconfiguration with parts of the chip having independent context control will help effective resource utilization and efficient reconfiguration control. It could be an interesting and worthwhile research approach to explore in future.

Bibliography

- [1] Xilinx Inc., *The Programmable Logic Data Book*. San Jose, CA, 1999.
- [2] G. Estrin, “Organization of computer systems - the fixed plus variable structure computer,” in *Proceedings of the Western Joint Computer Conference*, pp. 33–40, 1960.
- [3] P. M. Athanas and H. F. Silverman, “Processor reconfiguration through instruction-set metamorphosis,” *IEEE Computer*, vol. 26, pp. 11–12, March 1993.
- [4] J. M. Arnold, D. A. Buell, and E. G. Davis, “Splash 2,” in *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 316–324, June 1992.
- [5] P. M. Athanas and A. L. Abbott, “Real-time image processing on a custom computing platform,” *IEEE Computer*, vol. 28, pp. 16–24, February 1995.
- [6] J. R. Hauser and J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Coprocessor,” in *Proceedings of IEEE symposium on FPGAs for custom computing machines*, April 1997.
- [7] R. Bittner, *Wormhole Run-Time Reconfiguration: Conceptualization and VLSI Design of a High Performance Computing System*. PhD thesis, Virginia Polytechnic Institute and state University, Jan 1997.
- [8] M. Soni, “VLSI Implementation of a Wormhole Run-time Reconfigurable Processor,” Master’s thesis, Virginia Polytechnic Institute and state University, June 2001.

- [9] S. Srikanteswara, J. Neel, J. H. Reed, and P. M. Athanas, "Soft radio implementations for 3g and future high data rate systems," in *Proceedings of the Global Telecommunications Conference*, pp. 3370–3374, 2001.
- [10] X. ping Ling and H. Amano, "WASMII: a data driven computer on a virtual hardware," in *Proceedings of IEEE workshop on FPGAs for custom computing machines*, April 1993.
- [11] Xilinx, Inc., "XC6200 Advance product information," 1996.
- [12] G. Brebner, "A virtual hardware operating system for the Xilinx XC6200," in *Proceedings of 6th Intl. workshop on Field Programmable Logic and Applications*, Springer, pp. 327–336, 1996.
- [13] G. Brebner, "The swappable logic unit: a paradigm for virtual hardware," in *Proceedings of IEEE symposium on FPGAs for custom computing machines*, April 1997.
- [14] S. Guccione and D. Levi, "XBI: A Java-based interface to FPGA hardware," in *Proc. SPIE Photonics East, J. Schewel (Ed.), SPIE - The International Society for Optical Engineering, Bellingham, WA*, November 1998.
- [15] S. Guccione and D. Levi, "Run-Time Parameterizable Cores," in *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pp. 215–222, 1999.
- [16] Y. Shibata, M. Uno, H. Amano, K. Furuta, T. Fujii, and M. Motomura, "A virtual hardware system on a dynamically reconfigurable logic device," in *Proceedings of IEEE symposium on FPGAs for custom computing machines*, April 2000.
- [17] A. DeHon, "DPGA Utilization and Application," in *MIT Artificial Intelligence Laboratory, Transit Note 129*, September 1995.

- [18] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," in *Proceedings of IEEE Workshop on FPGAs for custom computing machines*, pp. 31–39, 1994.
- [19] Xilinx, "Time multiplexed programmable logic device," July 1997. Patent no. 5646545.
- [20] Xilinx, "Configuration modes for a time multiplexed programmable logic device," February 1997. Patent no. 5600263.
- [21] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A Time-Multiplexed FPGA," in *Proceedings of IEEE symposium on FPGAs for custom computing machines*, April 1997.
- [22] Chameleon Systems, Inc., "CS2000 Advance product information," 2000.
- [23] D. I. Lehn, K. Puttegowda, J. H. Park, P. M. Athanas, and M. T. Jones, "Evaluation of rapid context switching on a csrc device," in *Proceedings of the International conference on Engineering of Reconfigurable Systems and Algorithms*, 2002.
- [24] S. M. Scalera and J. R. Vázquez, "The design and implementation of a context switching FPGA," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
- [25] D. I. Lehn, "Application Framework for a context switching runtime reconfigurable system," Master's thesis, Virginia Polytechnic Institute and state University, April 2002.
- [26] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, "Implementing an API for distributed adaptive computing systems," in *Proceedings of the IEEE International Conference on Communications*, pp. 222–230, April 1999.

- [27] N. Vaswani and R. Chellappa, “Best view selection and compression of moving objects in IR sequences,” in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing Proceedings*, (Salt Lake City, Utah), 2001.
- [28] J. E. Scalera, “Image Chipping with a Common Architecture for Microsensors,” Master’s thesis, Virginia Polytechnic Institute and state University, July 2001.
- [29] Deutsches Museum, “Enigma encryption machine.” http://www.deutsches-museum-bonn.de/ausstellungen/meisterwerke/2_3enigma/enigma_e.html.
- [30] A. Hodges, *Alan Turing: The Enigma*. Simon and Schuster, 1983.

Appendix A

Video Applications Demonstration

This appendix contains a screen captures of the video application demonstrations on the RCM platform.

The first video processing application demonstrated host-driven context switching between the three filters implemented as contexts on the CSRC device. Figure A.1 shows the output of the pass through with a one frame delay. The left image shows the input to the RCM board and right image shows the output image which is delayed by a frame (The person in the background has moved). The second filter is a delay filter with fading. The output of the filter is shown in Figure A.2. Figure A.3 shows the output of the differencing filter

The other video processing application demonstrated was the motion detection algorithm. Figure A.4 shows the final binary image representing movements in the video stream. This is the output of the final context of the application.



Figure A.1: Screen capture of video processing with pass through filter



Figure A.2: Screen capture of video processing with the delay filter with fading



Figure A.3: Screen capture of video processing with the difference filter



Figure A.4: Binary image generated by the motion detection algorithm demonstration

Vita

Kiran Puttegowda was born in Bhadravati a small town in India. He did his early education in and around this place. He joined Karnataka Regional Engineering College (KREC) for his technical education in electronics and communication engineering in 1995. He obtained his Bachelor of Engineering (BE) degree in 1999. He then worked for a year at IBM India's ASIC design group as Design Engineer. He joined Virginia Tech in 2000 for his Masters Degree in Electrical Engineering. Since then he has been involved in reconfigurable computing research. Kiran's hobbies include painting, swimming, music and reading. His technical interests include reconfigurable and high performance computing, computer architecture and VLSI.